Thesis Dissertation

CONVOLUTIONAL NEURAL NETWORKS WITH HESSIAN-FREE OPTIMIZATION FOR ALZHEIMER'S DIAGNOSIS THROUGH MRI IMAGES

Marios Pafitis

University of Cyprus

Computer Science Department

May 2021

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Convolutional Neural Networks with Hessian-Free Optimization for Alzheimer's Diagnosis through MRI Images

Marios Pafitis

Supervisor

Dr. Chris Christodoulou

A thesis submitted in partial fulfillment of the requirements for the award of Bachelor degree in Computer Science at the University of Cyprus

May 2021

Acknowledgments

First of all, I would like to point out that this dissertation represents a fraction of the knowledge I received during my four-year degree in Computer Science. No matter what the outcome is, it taught me the essentials for academy research. Especially, my research, analytical, and programming skills have been improved to deliver a complete and well-written dissertation. Moreover, this thesis taught me how to organize my working process by prioritizing each task, to produce the expected end product in a tight schedule.

Every professor, from preschool to the university, played an essential role in what I am to this day and what I will become in the future, in terms of both personality and education. To reach my current academic level, I had to assimilate all the knowledge these people offered to me throughout the years. Regardless of our age this process never ends, especially for us who want to be devoted to lifelong learning. The highest contribution to this thesis came from the Professor and Computer Science Department's Vice-Chair of the University of Cyprus Dr. Chris Christodoulou. Not only for assigning me this intriguing machine learning problem and being my advisor, but also because he taught me the basics of Machine Learning, and Computational Intelligence.

I would also like to thank Dr. Maria Constantinou for providing me the necessary data and additional guidance for my thesis, based on her experience in Neuroscience and Machine Learning. Additionally, I would like to thank Dr. Constantinos Pattichis, Dr. Stephanos Leandrou, and Dr. Kleo Achilleos for providing me with additional data and valuable suggestions for experiments in this dissertation. Furthermore, I want to show my appreciation to Ms. Maria Tsiolakki, for providing me with access to the Arcadia server and installing all the required software and libraries for executing my experiments.

I would like to thank my family for their continuous support; my parents Irene and Michalis Pafiti; my siblings Elina and Costantinos Pafiti; and my grandparents Eleni and Costakis Christou, and Eleni and Andreas Pafiti. They gave me shelter, food, money, support, and trust, without asking anything in exchange, just to achieve my goals. Also, I would like to thank my best friends for always being by my side, Athina Zissimou, Georgia Demetriadou, Demetris Shimitras, and George Venizelou. Last, even if they cannot read this thesis, I have to thank my pets for keeping me fit and healthy; my dog Max for the walks, and my cats Ginger and Azor for the playtime during breaks.

Abstract

There are many forms of dementia, with Alzheimer's disease (AD) being one of the most severe, which is a chronic condition that degenerates the cells of the brain leading to memory asthenia; while normal cognitive (NC) is a person with no neurological or psychiatric problems whose cognitive abilities decline with normal aging (Arvanitakis et al. 2019). Second-order optimization methods, such as Hessian-Free Optimization (HFO), have proven to be more suitable for optimizing objectives that exhibit pathological curvature (Martens, 2010). Such a case might be the AD/NC problem which uses a deep neural network, a Convolutional neural network (CNN) for the feature extraction of MRI scans. The combination of CNN with HFO for the AD/NC problem has never been attempted before, and this is the main novelty of this thesis.

This dissertation aims to automate the detection of AD from MRI scans of the brain. To achieve this aim, CNN classifiers in combination with second-order optimization were used to classify AD and NC individuals. For the second-order optimization, the implementation of the HFO with the Gauss-Newton matrix for CNN (Wang et al. 2020) was used, based on the HFO algorithm by Martens (2010).

More specifically, 9 datasets were used to compare the performance of the Stochastic Gradient Descent (SGD), Adaptive Moment Estimation (Adam), and HFO optimizers, in Multi-Layer Perceptron (MLP) and CNN. The dataset with the 10 hippocampal features (Achilleos et al. 2020) was tested in MLPs with a different number of hidden layers and neurons per layer. In the meanwhile, for the datasets with 2D slices of T1-weighted MRI scans, five distinct 2D CNN network architectures were utilized with 3, 4, 5, 7, and 19 layers; with multiple configurations such as Dropout, Spatial Dropout, Max-Pooling, L1 & L2 Regularization, Batch Normalization, etc. Also, 3D T1-weighted MRI scans were used in two different 3D network configurations with 4 and 5 layers.

For all the experiments of this thesis, either 5-fold or 10-fold cross-validation was used. The best average validation accuracies of the folds, of all the experiments with the MLPs, were 90% and 87%, for the Adam and HFO optimizers respectively. For the CNN implementations, the highest average validation accuracies were 79% and 81%, for the Adam and HFO optimizers respectively. In general, the HFO algorithm performs better than Adam in 2D CNNs, but Adam is slightly better in MLPs and 3D CNNs. Most of the

experiments suffer from overfitting, even after applying regularization techniques, most probably because the datasets were small due to the absence of data augmentation.

The objective of this thesis is not to compete with state-of-the-art studies. The goal was to test the performance of HFO in comparison with Adam by using “toy” datasets of T1-weighted MRI scans, with multiple network configurations, regularization techniques, and various hyperparameters, to find the optimal ones. Eventually, the outcomes of this thesis can be used in future work, with a larger dataset that will apply data augmentation to compete with state-of-the-art studies.

Contents

Chapter 1

Introduction

1.1 The Importance of Automating Dementia Diagnosis

Dementia is a syndrome with characteristic symptoms of progressive cognitive decline, such as loss of memory, language disorders, and disorientation (Arvanitakis et al. 2019). The person in the early stages may be forgetting events or conversations. As the disease progresses the person might develop severe memory impairment and become unable to carry out everyday tasks. In Europe, approximately 8.8 million people are living with dementia in 2018 and are expected to double by 2050 (Alzheimer Europe, 2020). In Cyprus the statistics are even worse since for the period 2018 and 2050, the number of people suffers from dementia is expected to triple (Alzheimer Europe, 2020).

The most common cause of dementia is Alzheimer's disease (AD). More specifically, AD is a neurodegenerative disorder with characteristic symptoms of progressive loss of memory, language disorders, and disorientation. Identifying early stages of AD patients can influence the effects of the disease by supplying disease-modifying remedies. Therefore, this project focused on developing intelligent methods for automating the detection of AD.

In healthy aging, there is usually a slight decline in a person's cognitive abilities. If some cognitive functions decline to a greater degree than what is expected by healthy aging, then the patient may suffer from mild cognitive impairment (MCI). MCI is characterized by problems with memory, language, thinking, or judgment (Arvanitakis et al. 2019). MCI is diagnosed when the cognitive decline is abnormal for an individual's age and educational level, but it does not meet the criteria to be diagnosed as AD (Petersen et al.

1997). MCI patients have a high risk of progression to AD (Gauthier et al. 2006), who are so-called MCI converters (MCI-C), with the conversion rate estimated to approximately 15% (Allison et al. 2014). Therefore, the diagnosis of MCI is also important for the early diagnosis of AD.

The process for identifying whether a patient suffers from MCI, AD, or another form of dementia includes cognitive tests, interviews with family members of the patient, blood tests, and magnetic resonance imaging (MRI) scan (Arvanitakis et al. 2019). These scans are examined afterward by neurologists to identify the disease that the patient may have. The early diagnosis or the prediction of AD could provide the patient with early treatment and slow down the effects of the disease. The prediction of the disease is something that the neurologist may not always be able to do. The goal is to use advanced machine learning techniques to create models, that are going to assist the neurologist in early predicting AD and other forms of dementia.

1.2 Previous Research on Automating Dementia Diagnosis

Prior work on the problem can be separated into two main categories: attempts of traditional machine learning and deep learning approaches. The combination of traditional machine learning for classification and stacked auto-encoder (SAE) for feature selection (Suk et al. 2015), produced accuracies for the AD/NC classification up to 98.8% and for the AD/MCI classification, up to 83.7%. The deep learning approaches that use neuroimaging data, such as PET scans, with 3D Convolutional Neural Networks (CNN), have yield accuracies up to 96% (Choi and Jin, 2018). Also, the combination of the two methods, the 3D CNNs and 3D Convolutional Auto-Encoders (CAE) with MRI scans, have yielded accuracies for the AD/MCI/NC problem up to 94.8%, and the AD/NC problem up to 99.3% (Hosseini-Asl et al. 2018).

We should keep in mind that the AD/NC problem is much simpler than the AD/MCI/NC, as the MRI scans of the two categories (AD and NC) are dissimilar, therefore, can be easily distinguished by a trained eye. Consequently, the deep learning algorithms for classifying AD/NC, MCI/NC, usually yield great accuracies; but there is still space for improvement for classifying all three of them simultaneously, AD/MCI/NC, which is challenging since usually MRI scans between AD and MCI can be confused.

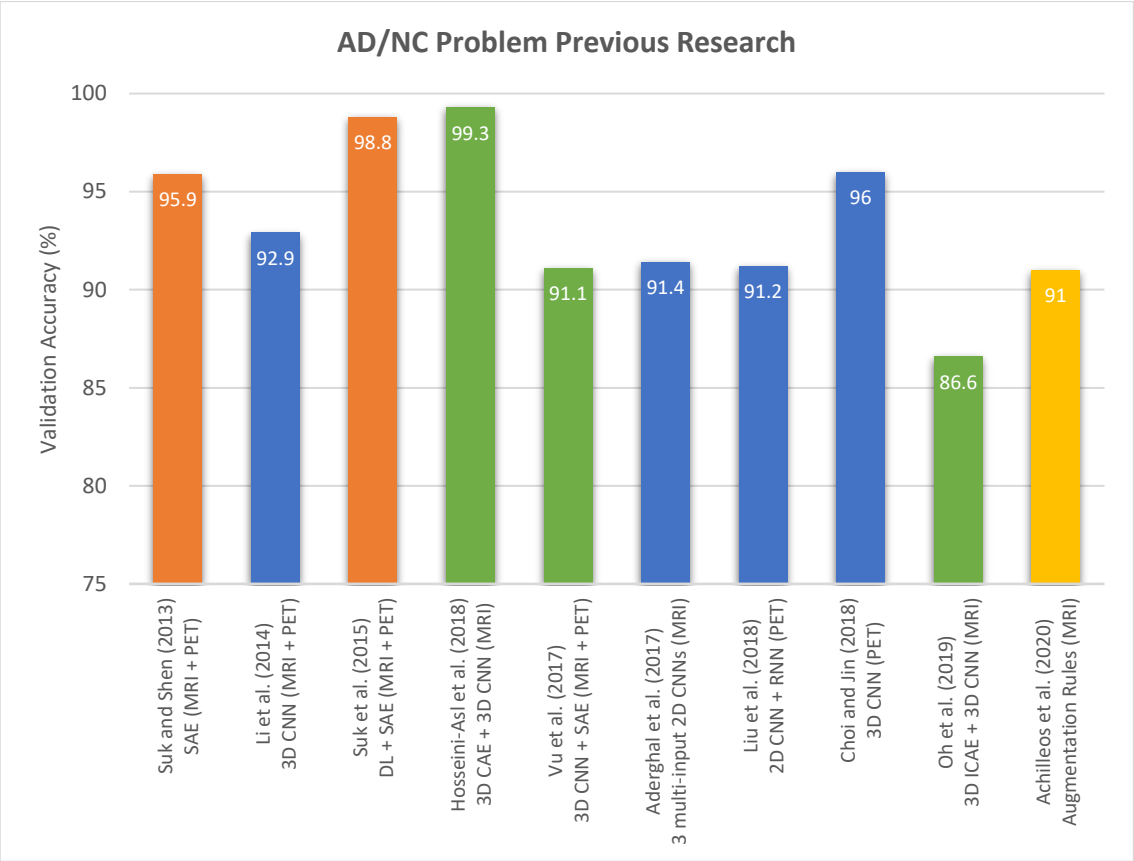


Figure 1.1. AD/NC problem previous research. Comparison based on the validation/testing accuracy. **Blue:** Convolutional Neural Networks (CNN); **Orange:** Auto-Encoders (AE); **Green:** AE + CNN; **Yellow**: Augmentation Rules. (Jo et al. 2019)

Figure 1.1 shows some remarkable previous studies for the AD/NC problem (Jo et al. 2019). With the blue color, we can see studies that used Convolutional Neural Networks (CNN), with orange color studies with Auto-Encoders (AE), with green color, a combination of AE with CNN, and with yellow color studies with Augmentation Rules.

Achilleos et al. (2020), extracted rules from MRI images, by using decision tress (DT) and random forests (RF) algorithms. Then, they integrated those rules in the Gorgias framework, an argumentation-based reasoning framework, and achieved an average accuracy of 91%.

Suk and Shen (2013), yield a 95.9% accuracy by using MRI and PET scans with Sparse Auto-Encoders (SAE) to construct an augmented feature vector by concatenating the original features with outputs of the top hidden layer of the representative SAEs, and a multi-kernel Support Vector Machine (SVM) as a classifier. Later, Suk et al. (2015) extended their work by developing a two-step learning scheme. First, a greedy-layer-wise

pre-training and then a fine-tuning in deep learning, improved their model to achieve accuracy up to 98.8%.

Recently, more deep learning approaches than traditional machine learning methods have been developed, because the CNNs were proven as one of the best techniques for analyzing visual imagery (LeCun et al. 2015), such as image and video recognition, image classification (Krizhevsky et al. 2012), natural language processing and in our case for medical image analysis. Different CNN network configurations have been used in many studies for the AD classification problem. Based on the dataset used, whether it contains 2D slice-level or 3D subject-level MRIs; 2D CNNs (Liu et al. 2018) or 3D CNNs (Hosseini-Asl et al. 2018) are being used respectively.

Li et al. (2014) introduced CNNs for the AD/NC problem with 3D CNN models on subjects with both MRI and PET scans, achieving an accuracy of 92.9%. When using only PET scans, the accuracy was 88.7%. The 3D CNN encoded the non-linear relationship between MRI and PET scans and then used the trained network to estimate the PET patterns for subjects with only MRI data. Hosseini-Asl et al. (2018), used 3D CNNs, pre-trained by 3D Convolutional Auto-Encoders (CAE), that yield an accuracy of 97.6%. The application of SAE and 3D CNN on subjects with MRI and FDG PET scans from Vu et al. (2017), yield an accuracy of 91.1%. In Choi and Jin's (2018) study, the usage of 3D CNN models was reported with multimodal PET scans, which obtained an impressive accuracy of 96%. Oh et al. (2019), used 3D inception model-based CAE (ICAE) with 3D CNNs and achieved an accuracy of 86.6%.

For the 2D CNN approaches, Aderghal et al. (2017), capture 2D slices from the hippocampal region in the axial, Sagittal, and Coronal directions. Then applied those three images as input in a multi-input model with three distinct 2D CNNs, which are combined at the end with a single FFNN. This method, yields an accuracy of 85.9%, not far away from our best accuracy of 81% when using only the Coronal 2D slice of a subject. Liu et al. (2018), used 2D slices from 3D PET scans, in a combination of CNNs and RNNs to learn the intra-slice and inter-slice features for classification. This study yields an accuracy of 91.2% for the AD/NC problem.

Another technique that has been proposed was applying age-correction processing on the MRI images before extracting 2.5D patches which had to feed them in a 2D CNN (Lin et

al. 2018). We got inspired by this technique and we would like to try to provide the age of the patient as an extra feature in the HFO algorithm alongside the features extracted from the CNN in future studies. At the moment, no multi-input model was tested, which could potentially be consist of a Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN), but is highly suggested for future work.

1.3 Automating Dementia Diagnosis with Hessian-Free Optimization (HFO)

This thesis attempts to classify the two different categories: Alzheimer's disease (AD), normal cognitive (NC), by using 2D slice-level and 3D subject-level MRIs as input in Convolutional neural networks (CNNs) combined with the Hessian-Free Optimization (HFO) (Wang et al. 2020). Additional experiments are going to be held with a dataset that contains the left hippocampal structure of patients provided by Achilleos et al. (2020). Also, a direct comparison between us and Achilleos et al. (2020) is going to be performed, by using the same dataset as them, with the 10 features extracted from the left hippocampal structure of AD and NC patients. The "AD/NC problem" in this thesis will be referred to as the classification between AD and NC; while the classification between AD, mild cognitive impairment (MCI), and NC will be called the "AD/MCI/NC problem". The AD/MCI/NC problem is going to be approached by using 2D slice-level MRIs of AD, MCI, and NC patients, as input to CNNs with the HFO (Wang et al. 2020).

The background of the methodology of this thesis is based on Martens (2010) and Martens and Sutskever (2012), about using deep learning with the Hessian-Free Optimization (HFO) algorithm. Martens (2010) used the HFO algorithm to train deep auto-encoders. As he described, on backpropagation-based algorithms, gradient descent (GD) is being used to train the weights of a network with multiple layers of non-linear hidden units. The problem with GD is that it does not seem to generalize well in networks with many hidden layers, such as deep neural networks, as it progresses extremely slow, resulting in poor performance on the training set (under-fitting). Also, GD is unsuitable for optimizing objectives that exhibit pathological curvature. Thus, second-order optimization methods, such as HFO, have proven to be more effective on such objectives as they model the local curvature and correct it.

Our focus will be on using deep learning networks, CNNs with the HFO algorithm to solve the AD/NC problem. The AD problem, which we are trying to solve, is expected to exhibit pathological curvature as it uses a deep neural network, CNN for feature extraction. Thus, given that we are expecting for the GD, a curvature-blind method, it would be hard for it to successfully navigate on the multidimensional error surface fast enough; we are going to use HFO, a second-order method, which models the local curvature. The main novelty of our project lies in the fact that to the best of our knowledge, the CNN with HFO has never been used for AD diagnosis through MRIs.

The general aim of this project is to solve the AD and NC classification problem, by using state-of-the-art machine learning techniques such as CNNs and HFO. Moreover, the project aims to create the foundations, the relatively "best" dataset, and network configuration for future work, which will try to detect multiple forms of dementia such as Parkinson's MCI, MCI-C, AD. In the meantime, we want our dataset to be as less edited as possible. In the best-case scenario, the MRI image of the MRI scanner to be able to be feed immediately on our network without any heavy preprocessing.

Our goal is not to replace the radiologist, the specially trained doctor who reads and analyses MRI scans. We aim to assist radiologists, by providing an additional prediction to their existing methods to avoid false NC diagnosis, in case the disorders AD and MCI are not visible in an MRI scan. More specifically, the radiologists should be able to provide a 2D slice MRI or the unedited 3D T1-weighted MRI brain scan, of a patient who potentially suffers from AD or MCI, and receive a prediction whether the patient has AD, MCI, or is NC.

It is worth mentioning, that most of the previous studies have tried to solve a simpler problem than the one we are trying to solve, as they have tried to classify categories such as AD + MCI/NC, AD/NC, MCI/NC, and AD/MCI which consist only of two target outputs. Only a few studies exist for classifying the combination of all three categories AD/MCI/NC, with the current highest accuracy at 89.1% (Hosseini-Asl et al. 2018). To achieve our objectives, we will examine the performance of different architectures for the deep neural network CNN, (deep, shallow, narrow, and wide CNNs, with Dropout, Batch Normalization, L1 & L2 Regularization) which is going to be responsible for the feature extraction of the MRI scan.

Most of our implementations, are using 2D CNNs with datasets of 2D slices of MRIs. The issue with the 2D slices is that the T1-weighted MRI images are 3D, so, we have to select correctly the 2D Coronal slice (e.g., the 3rd image of Figure 2.4) to be the input pattern. The signs of AD on MRI scans are more evident in the entorhinal cortex and hippocampus (Leandrou et al. 2018; Leandrou et al. 2020). Therefore, the best choice of a 2D MRI slice based on the existing literature would be one that includes the entorhinal cortex and hippocampus.

An advantage of using 2D CNNs over 3D CNNs is that existing CNNs, which had huge success for natural image classification, such as ResNet and VGGNet can be borrowed and used in a transfer learning fashion. Additionally, another benefit of 2D CNN models, is that much less computing power is required than a 3D CNN to be trained.

A disadvantage of using 2D slices of the MRI scans, as input in 2D CNNs, is that usually in studies they suffer from data leakage (information from outside the training dataset is used to create the model) (Wen et al. 2020). This means, that the 2D slices of the same MRI scan should only be used either in the training, validation, or test set. In this thesis, we will address this issue by using StratifiedKFold (Section 3.1.3) and StratifiedGroupKFold (Section 3.1.5) for the splitting of our datasets.

A pitfall that is usually being observed in 2D and 3D CNN implementations for the AD/NC problem is that an imbalanced performance metric can be computed on a severely imbalanced dataset (one class is less than half of the other) (Jo et al. 2019). Usually, for the AD/NC problem, more NCs are available than ADs, which results in an imbalanced test set. This means that the performance metrics are not representative of the model's actual performance. For example, if a test set contains 80% NCs and 20% ADs, a test accuracy for a model that always predicts NC would be 80%, which does not reflect a real-world test case scenario. To encounter this issue, we are going to use balanced training, validation, and tests, all three have an equal number of AD and NC patients; and additionally, other performance metrics such as sensitivity, specificity, PPV, and NPV are going to be tracked.

Experiments with 3D CNNs are going to be performed as well in this thesis, which basically will receive as input the whole 3D T1-weighted MRI scan. We are going to shrink the 3D MRI scan, by taking the mean of each $4 \times 4 \times 4$ block, due to lack of time

and resources. Additionally, 3D CNNs, are going to be used for the dataset with the Left Hippocampus (Achilleos et al. 2020). Finally, the dataset with the brains cropped in the section of the left hippocampus is going to be used for training 3D CNNs as well.

Recently, due to the boost of high-performance computing resources, 3D CNN approaches have been used for the AD problem. Previously, 2D CNN approaches had been used (Wen et al. 2020), in which, the 2D slices have to be selected manually carefully to represent the portion of the brain that will show signs of AD or MCI. On the other hand, with the 3D CNNs, we can provide the whole 3D MRI of a patient, where the spatial information is fully integrated, and so we let the deep neural network freely decide which features are most important. Besides, the 3D CNN approach (Hosseini-Asl et al. 2018), seems very promising, for classifying the three stages of the disease AD/MCI/NC, as in previous studies achieved accuracy 89.1%, by using MRIs as an input to a Pretrained Generic Feature Extraction 3D-ACNN. We should keep in mind though that the risk of overfitting increases with this technique, as the size of our data set is limited by the number of 3D MRI images we have, where on the other hand with the 2D CNN we could provide multiple slices as different patterns of the same subject.

To overcome this issue, we have to increase the size of the dataset, either by finding more patients or performing data augmentation. Due to a lack of time and processing power, data augmentation is not going to be performed in this thesis. Additionally, to avoid overfitting, we are going to use the early stopping method (Caruana et al. 2001), as we are going to compare the training and test error, end terminate the training just before the validation error starts increasing while the training error keeps decreasing. Different techniques to encounter overfitting will be applied in the network's topology as well such as Dropout, Spatial Dropout, Batch Normalization, L1 & L2 Regularization, Weight Decay, and KFold Cross-validation.

Both deep neural network implementations, 2D, and 3D CNNs are going to be tested with two different learning algorithms. The first implementation, our benchmark for both CNN architectures, will be a simple first-order method, the Adam (Kingma and Ba, 2014). The second implementation is going to be the NewtonCG/HFO, a second-order method, which we expect to perform better than the Adam for such a problem, with a multidimensional error surface (Martens, 2010; Martens and Sutskever, 2012). From experiments with the Hessian-Free Optimization of TensorFlow in the MNIST dataset, it

has been proven in practice that Hessian-Free Optimization with the Newton-Gaussian Matrix is much more stable, with fewer fluctuations, and needs less space than the Hessian-Free Optimization with the Hessian Matrix. Thus, with any reference in this thesis to the HFO or NewtonCG algorithm, we mean the Hessian-Free Optimization with the Newton-Gaussian Matrix.

In theory, the combination of CNN and HFO seems easy but in reality, it is not. Stochastic Gradient Descent (SGD) (Krizhevsky et al. 2012) usually was used for deep learning. When Adam was released by Kingma and Ba (2014), who had shown that it performs very well with computer vision, it became the next standard for CNNs. Very complicated operations are required for implementing Newton methods with CNN. This combination of a CNN and HFO had been proved to be feasible since a study after they developed its theoretical implementation, they had also implemented it in just several hundred lines of code in MATLAB and Python (Wang et al. 2020).

A combination of Feed-Forward Neural Networks (FFNNs) with the HFO algorithm, by Charalambous et al. (2020) has shown that this combination achieves accuracies comparable to some of the state-of-the-art methods, for the Protein Secondary Structure Prediction (PSSP) problem. Charalambous et al. (2020) proved the importance of the right choice of algorithms; a powerful second-order algorithm, such as HFO, with one of the simplest neural networks such as FFNN, can be powerful enough to be compared with other methods designed specifically for such a challenging problem as PSSP. Therefore, for the hippocampal features (Achilleos et al. 2020), the three optimizers: SGD, Adam, and HFO are going to be compared, using FFNNs.

The combination of CNN with the HFO algorithm was attempted for the first time too by Leontiou et al. (2021) for a different problem, the PSSP problem. The authors could not use the original HFO algorithm, because of the complex CNN structure, so they used a variation, the Subsamples Hessian Newton (SHN). An advantage of the SHN method (Wang et al. 2020) was that it did not require much tuning of the hyperparameters which made their training process much faster than other state-of-the-art methods of the PSSP problem.

Nine distinct datasets are going to be used for the 80 experiments that are going to be described in this thesis for both the AD/NC and AD/MCI/NC problems. The datasets for

the AD/NC problem are (1) Hippocampus Features – Multiple Scans per Patient (Achilleos et al. 2020), (2) Single 2D Slice per Scan – Single Scan per Patient, (3) Five 2D Slices per Scan – Single Scan per Patient, (4) Single 2D Slice per Scan – Multiple Scans per Patient, (5) Seven 2D Slices per Scan – Multiple Scans per Patient, (6) 3D Left Hippocampus isolated from a Single Scan per Patient (Achilleos et al. 2020), (7) 3D Shrunk Brains – Single Scan per Patient, (8) 3D Brains Cropped containing the Left Hippocampus from a Single Scan per Patient; and for the AD/MCI/NC problem the dataset (9) 2D Slices per Scan – Multiple Scans per Patient. For all five 2D datasets, the slice refers to the Coronal slice of the MRIs.

Chapter 2

Background

2.1 Biological Background

2.1.1 Brain and Hippocampus

The brain and spinal cord make up the central nervous system. The brain consists of the cerebrum, cerebellum, and brainstem (Figure 2.1 Left). The cerebrum consists of two hemispheres with four lobes: frontal, parietal, temporal, and occipital (Figure 2.1 Right).

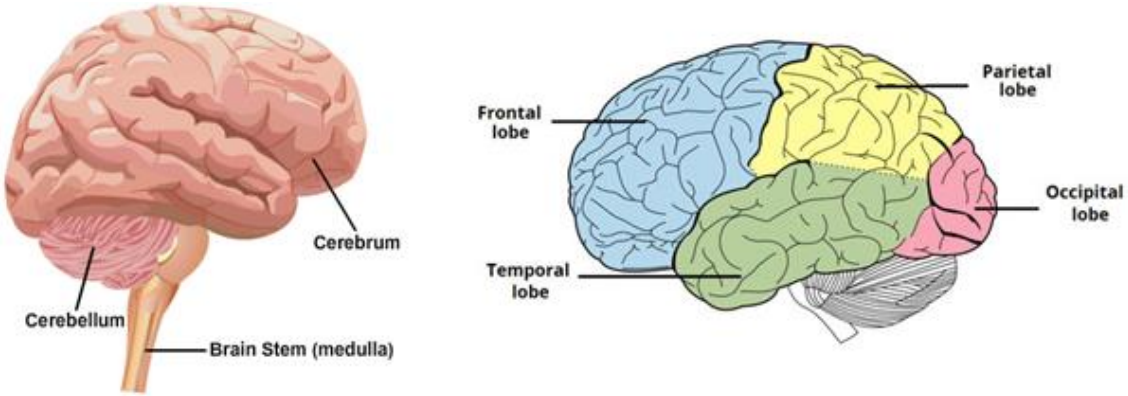


Figure 2.1. The human brain. **Left:** cerebrum, cerebellum, and brain stem are indicated. **Right:** The frontal, temporal, parietal, and occipital lobes of the cerebrum are highlighted.

The frontal lobe is responsible for higher intellect, personality, mood, social conduct, and language in the dominant hemisphere side only (Hoffmann, 2013). The parietal lobe controls the language, calculations on the dominant hemisphere side, and visuospatial function on the non-dominant hemisphere side (Brownsett and Wise, 2010). The occipital

lobe is mostly responsible for vision (Bender et al. 1957). The temporal lobe is responsible for memory, language, and hearing (Kiernan, 2012).

In the temporal lobe, the hippocampus is located, which plays a major role in Alzheimer's disease (Arvanitakis et al. 2019). Humans have two hippocampi, one at each side of the brain, and are part of the limbic system (Figure 2.2). The hippocampus plays a major role in the consolidation of information from short-term memory to long-term memory (Squire and Wixted, 2011). Additionally, the hippocampus is a key component for navigation and spatial memory (O'Keefe and Dostrovsky, 1971; O'Keefe and Recce, 1993; Maguire et al. 2006).

The brain regions which are initially affected by Alzheimer's disease and other forms of dementia are the hippocampus and entorhinal cortex (Braak and Braak 1991; Frisoni et al. 2010). Neurodegeneration in the temporal lobe is visible in structural MRI and is commonly used to confirm the diagnosis of AD (Chandra et al 2019).

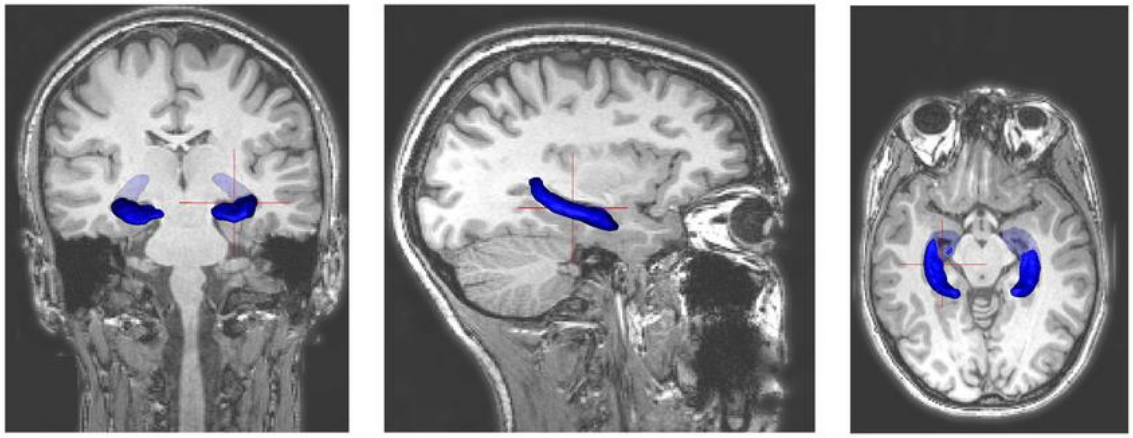


Figure 2.2. Left & Right hippocampus (blue) in coronal, horizontal, and sagittal slices of a brain (Gerardin, 2012).

2.1.2 T1-weighted MRI

Magnetic resonance imaging (MRI) is a medical image that represents the anatomy and the physiological processes of the body. For generating images of the organs in the body, MRI scanners use strong magnetic fields, magnetic field gradients, and radio waves (Figure 2.3). In this study, we are going to use T1-weighted MRI scans, which are 3D MRI images that show the anatomy of the brain. The precision may vary based on the scanner's brand and its capabilities.

Figure 2.3. Brain MRI scanner with a patient and a doctor.

Figure 2.4 shows the three different slices of an MRI scan. The first is the Sagittal slice, the second the Horizontal slice, and the third the Coronal slice of a 3D brain MRI scan of an NC patient. In the Sagittal slice (1st image), we can spot in the image the nose of the patient on the right side. In the Horizontal slice (2nd image), we can spot the two eyeballs of the patient at the bottom of the image. In the Coronal slice (3rd image), the ears of the patient can be spotted on the left and right sides.

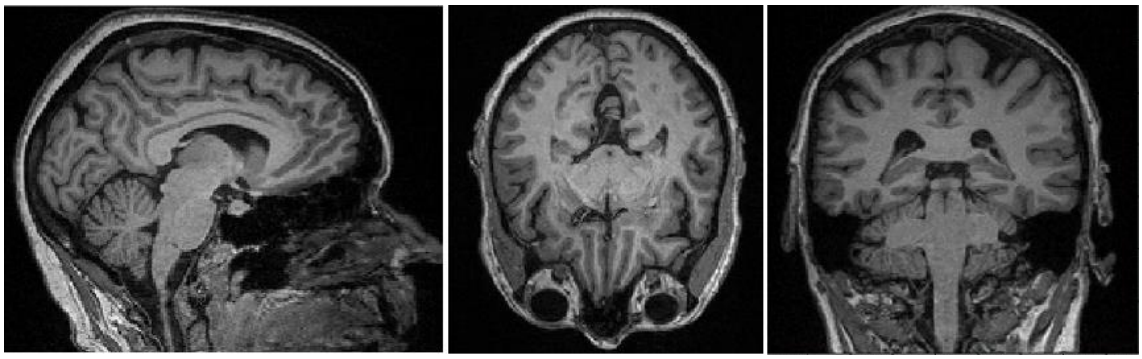


Figure 2.4. Slices of NC patient's MRI scan. **First:** Sagittal Slice. **Second:** Horizontal Slice. **Third:** Coronal Slice

2.1.3 Alzheimer's Disease, Mild Cognitive Impairment, and Normal Cognitive

Alzheimer's disease (AD) is a neurodegenerative disorder with pathological hallmarks of β-amyloid plaques and neurofibrillary tangles (Braak and Braak 1991) and characteristic brain atrophy especially in the temporal lobes (Chandra et al 2019). Some characteristic symptoms are progressive loss of memory, language disorders, and disorientation. AD is the most common cause of dementia. At the early stages, the person may be forgetting recent events or conversations, while as the disease progresses, he might develop severe memory impairment and become unable to carry out everyday tasks (Arvanitakis et al. 2019).

Mild cognitive impairment (MCI) is the decline in some cognitive functions, that is greater than expected by normal aging. MCI is characterized by problems with memory, language, thinking, or judgment, and might progress to become dementia, including AD (Arvanitakis et al. 2019). The cognitive decline in MCI is abnormal for an individual's age and educational level but it does not meet the criteria to be diagnosed as AD (Petersen et al. 1997).

Normal cognitive (NC) is any person with no neurological or psychiatric problem. The cognitive abilities of the person decline with normal aging (Arvanitakis et al. 2019).

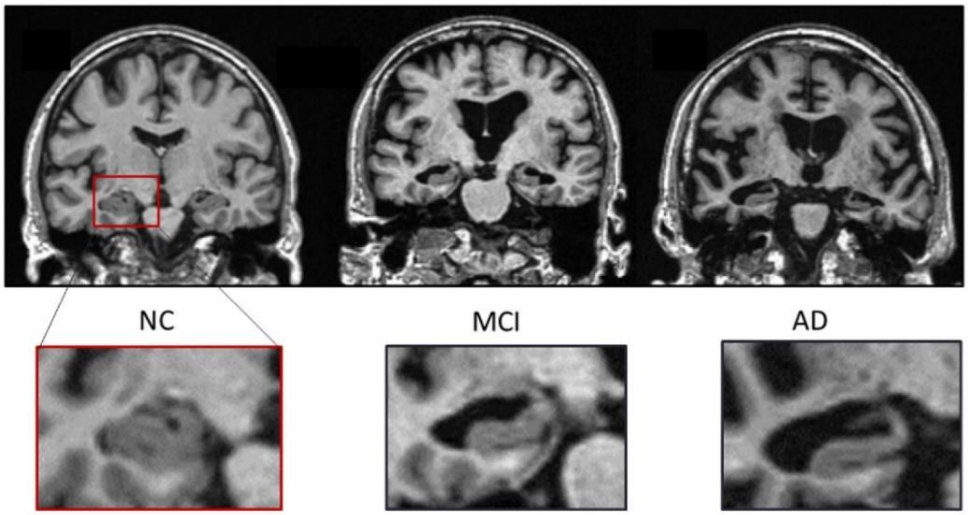


Figure 2.5. Comparison of the Hippocampus region from Coronal slices of the T1-weighted MRI of NC, MCI, and AD patients respectively. (Ahmed et al. 2017)

Figure 2.5 shows Coronal slices of different patients' T1-weighted MRI scans, with each one of them being in a different class; NC, MCI, and AD respectively. We can distinguish the differences between the three conditions by observing for example their hippocampus (zoomed part of the images). The hippocampal area of the NC patient is larger than the MCI, while the area of AD is even smaller than the MCI's.

2.1.4 Biological Neuron and Action Potential

Our brain has 100 billion neurons approximately (Herculano-Houzel, 2012). Neurons or so-called biological neurons can be considered major information processing units because they encode and decode information. For example, when neurons receive a stimulus (input), they generate sharp electrical action potentials, "spikes", across their cell membrane, which are being transmitted along the axon and synapses (output) to reach other neurons (Figure 2.6).

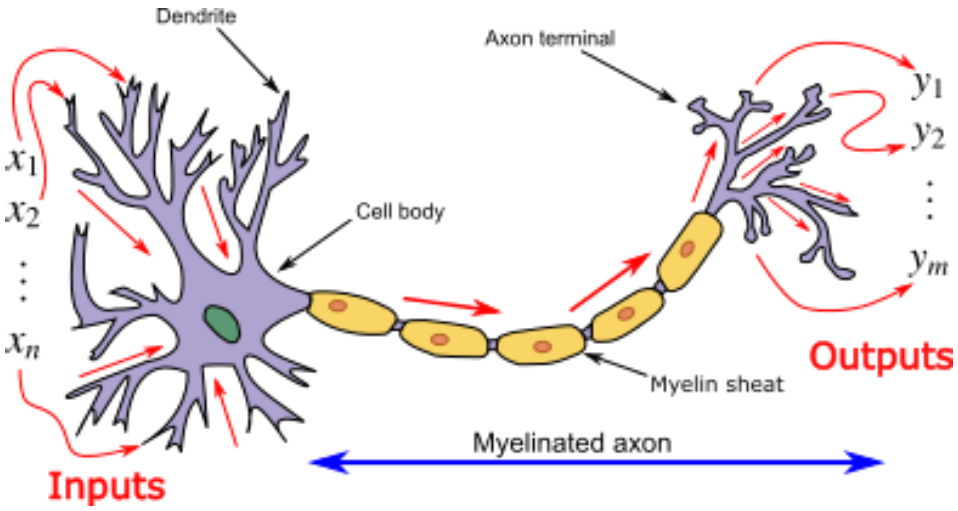


Figure 2.6. Biological neuron, which receives a single short electrical pulse, called "spike", that flows from the dendrites (input) to axon terminals (output). (Prof. Loc Vu-Quoc, University of Florida, 2012)

Action potentials are crucial for the neural code, i.e., 'language' by which neurons communicate in our nerve system (examples include Adrian and Zotterman (1926); Hubel and Wiesel (1959); Henry et al. (1974); Georgopoulos et al. (1982)). More specifically, an action potential occurs when the resting membrane potential of a neuron rapidly rises and falls. It propagates along neurons' axons to reach the synaptic boutons where it can be transmitted to other neurons, motor cells, or glands.

An input signal or multiple input signals can be summed in a neuron and increase its membrane potential. When the membrane potential is greater than a threshold, then the neuron fires and an action potential is being created inevitably (Figure 2.7).

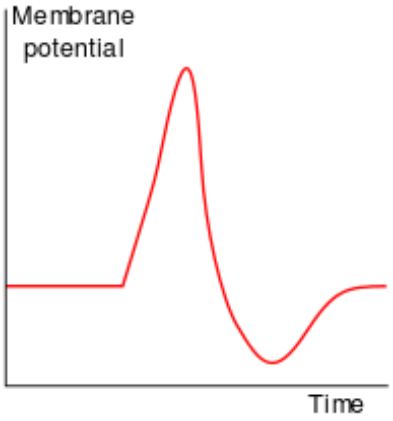


Figure 2.7. The shape of a typical action potential.

Inspired by the biological characteristics of an action potential, artificial neurons (Section 2.3.5) were created, which sum multiple inputs to produce an output based on an activation function, which can play the role of the threshold.

2.2 Mathematical Background

2.2.1 Function

A function f is a process that associates each element of a set X to a single element of a set Y, $f: X \rightarrow Y$. The X and Y are respectively called the domain and subdomain of a function.

2.2.2 Mathematical Optimization (minimization or maximization)

An optimization problem consists of minimizing or maximizing a real function, by selecting the best element, with respect to some criterion, from the available set of values in the function's domain.

For example, given a function, $f: A \rightarrow \mathbb{R}$, from set A to the real numbers, sought an element $x_0 \in A$, such that for minimization $f(x_0) \leq f(x)$, or for maximization $f(x_0) \geq f(x)$ for all $x \in A$.

2.2.3 Quadratic Form

The quadratic form is a polynomial where all terms are of degree two. For example, the polynomial $5x^2 + 4xy + 3y^2$, is in quadratic form.

2.2.4 Derivative / Finite Difference Method (FDM)

The Finite Difference Method (FDM), or the so-called derivative of a function, is one of the methods used to solve differential equations that are very difficult or even impossible to solve analytically. Given a differentiable function f, its derivative at a point x is:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

2.2.5 Chain Rule

The chain rule is a formula that calculates the derivatives of composite functions, which are functions that are composed of functions inside other functions. An example of a composite function is:

$$f(x) = h(g(x))$$

We use the chain rule to find the derivative of $f(x)$ that is equal to:

$$\frac{df}{dx} = \frac{dh}{dg} \cdot \frac{dg}{dx}$$

For example, given a function f that is composed of the nested functions A, B, and C:

$$f(x) = A\Big(B\big(C(x)\big)\Big)$$

By applying the chain rule to the f we get the derivative:

$$\frac{df}{dx} = \frac{dA}{dB} \cdot \frac{dB}{dC} \cdot \frac{dC}{dx}$$

In a different notation, the derivative of f by using the chain rule is:

$$f'(x) = A'(B(C(x))) \cdot B'(C(x)) \cdot C'(x)$$

2.2.6 Gradient

The gradient of a differentiable function f is the vector field ∇f in which each point p is the vector that represents the partial derivative of f at p. Given $f: \mathbb{R}^n \rightarrow \mathbb{R}$, its gradient $\nabla f: \mathbb{R}^n \rightarrow \mathbb{R}^n$ is defined at the point $x = (x_1, \dots, x_n)$ as the vector:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x) \end{bmatrix}$$

2.2.7 Hessian Matrix

The Hessian Matrix, in the case of Neural Networks, is a square matrix with the number of rows and columns equal to the total number of parameters in the Neural Network, which describes the local curvature of a multi-variable function. Each parameter is a second-order partial derivative of a scalar.

$$H(e) = \begin{bmatrix} \frac{\partial^2 e}{\partial w_1^2} & \frac{\partial^2 e}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_1 \partial w_n} \\ \frac{\partial^2 e}{\partial w_2 \partial w_1} & \frac{\partial^2 e}{\partial w_2^2} & & \frac{\partial^2 e}{\partial w_2 \partial w_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 e}{\partial w_n \partial w_1} & \frac{\partial^2 e}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_n^2} \end{bmatrix}$$

2.2.8 Jacobian Matrix

The Jacobian Matrix is the matrix of all first-order partial derivatives of a vector function. If the matrix is squared, its determinant is referred to as the Jacobian determinant. Given $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$, a function such that each first-order partial derivatives exists on $\mathbb{R}^n$. The function takes as input a point $x \in \mathbb{R}^n$, produces as output the vector $f(x) \in \mathbb{R}^m$. So, the Jacobian matrix J of f, is the $m \times n$ matrix in which each entry is $J_{ij} = \frac{\partial f_i}{\partial x_j}$.

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

2.2.9 Convolution

Convolution is the process of adding each element/pixel value of an image to its local neighbors, weighted by a filter. The filter is a matrix of values. To perform the Convolution, we slide the filter, usually from the top left corner, across the image. Afterward, we apply the Hadamard product between the two matrices, the filter, and the underlying image values.

The Hadamard product ($A \circ B$) takes as input two matrices of the same dimensions and produces a third matrix of the same dimensions as well, where each [i, j] value is the product between the [i, j] positions of the two input matrices. The sum of the values of the Hadamard product is the new value of the specific element of the image. For example, the following 3×3 matrix represents the values of the top left corner of an image.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

If we apply Convolution by using the following 3×3 matrix which represents our filter:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The value of the element at coordinates [2, 2], the central element of the top left corner of the input image will be:

$$\sum_{ij}\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \circ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right)_{ij} =$$

$$(a*1)+(b*2)+(c*3)+(d*4)+(e*5)+(f*6)+(g*7)+(h*8)+(i*9)$$

2.2.10 Linear Separability

Linear Separability in Euclidean geometry is a property of two sets of points, where the two sets are linearly separable if at least one line exists in the plane, in which all points of the first set are on the left of the line, and all the points of the second set are on the right of the line. In other words, at least a line should exist that perfectly separates the two sets into two subsets without having elements of one set in the group of the other (Figure 2.8). Linear Separability generalizes to higher-dimensional Euclidean spaces, other than 2-D if the line is replaced with a plane for 3-D and hyperplane for N-D.

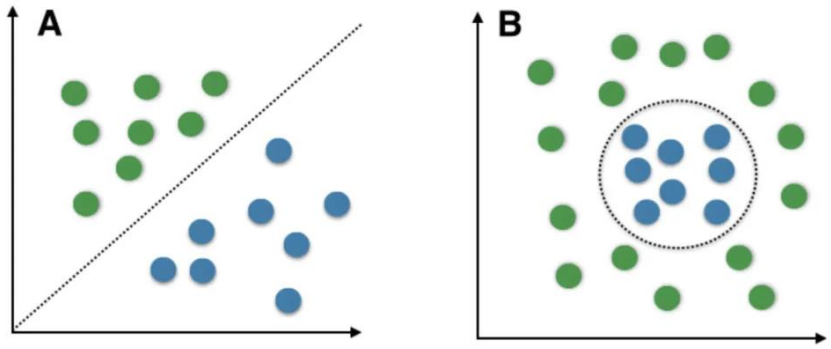


Figure 2.8. Left: Linearly Separable sets. **Right:** Non-linearly Separable sets.

2.2.11 Heaviside Step Function

Heaviside Step Function, H (Figure 2.9) is a step function, the value of which is zero (0) for negative input and one (1) for positive input.

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

The Heaviside Step Function was used in the McCulloch & Pitts Model (Section 2.3.7) as an activation function (McCulloch and Pitts, 1943).

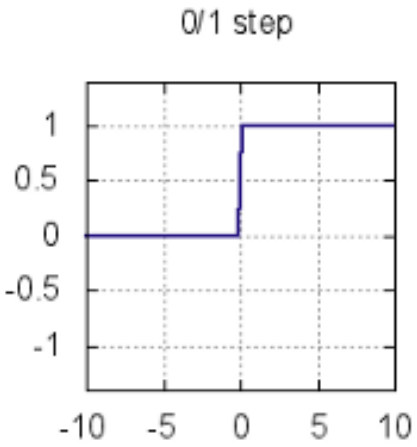


Figure 2.9. The Heaviside Step Function. Outputs 0 for negative inputs and 1 for positive inputs.

2.2.12 Logistic/Sigmoid Function

Logistic function or Logistic curve or Sigmoid function (Figure 2.10) is a common S-shaped curve, with the equation:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

The Sigmoid function is generally been used by neurons as a non-linear activation function in the back-propagation algorithm of Multi-Layer Perceptron (Section 2.3.9). The function is continuous, monotonically increases, and approaches asymptotes at both positive and negative infinity. It is extremely important for an effective learning law in the back-propagation algorithm that the Sigmoid function is differentiable. In contradiction with the Heaviside Step Function's output (0 or 1), the Sigmoid function's output (from 0 to 1) is more informative in terms of how close we are to the threshold, $f(0) = 0.5$.

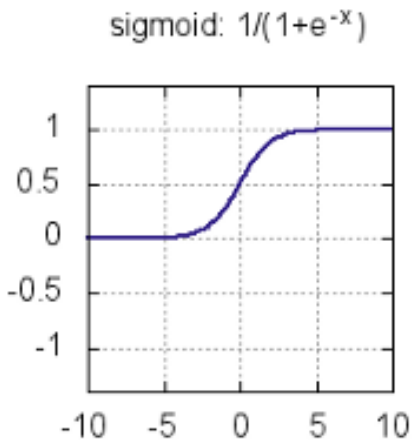


Figure 2.10. Standard Logistic Sigmoid function with $L = 1, k = 1, x_0 = 0$. Outputs a value between 0 and 1.

2.2.13 Rectified Linear Unit (ReLU)

The Rectified Linear Unit (ReLU) (Figure 2.11) is usually being used as an activation function in Artificial Neural Networks (Section 2.3.6). It is defined as the positive value of its input. For each x being the input of a neuron the output results from the following equation:

$$f(x) = \max(0, x)$$

ReLU was first demonstrated to enable better training of deep neural networks in comparison to other widely used activation functions such as the Logistic/Sigmoid or Hyperbolic Tangent (Glorot et al. 2010). Some of its applications are in computer vision, speech recognition using deep neural networks, and computational neuroscience.

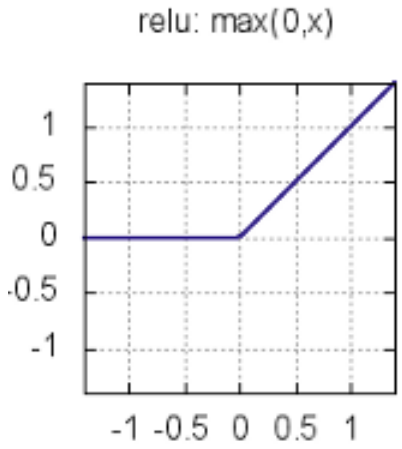


Figure 2.11. ReLU activation function. For negative inputs, it returns 0, otherwise, it returns the input.

2.2.14 SoftMax Function

SoftMax function or so-called SoftArgMax function is a generalization of the Logistic/Sigmoid function to multiple dimensions. It is usually being used as the last activation function of a neural network to normalize the output of a network to the probability distribution of the output classes.

SoftMax takes as input a vector x of n real numbers and normalizes it into a probability distribution to the exponentials of the input numbers. Therefore, the vector can have negative values or greater than one, but after applying softmax they will be in the interval (0, 1) and they will sum to 1 as well. The equation of SoftMax is:

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_i}} \; for\; i = 1, \dots, n \; and \; x = (x_1, \dots, x_n) \in \mathbb{R}^n$$

For example, given an input array:

$$arr = [-2, 1, 5, 0]$$

the output of the SoftMax function will be:

$$softmax(arr) = [0.0009, 0.0179, 0.9747, 0.0066]$$

where the values are normalized between 0 and 1. Also, the sum of SoftMax's outputs is approximately equal to 1:

$$0.0009 + 0.0179 + 0.9747 + 0.0066 = \sim 1$$

2.2.15 ArgMax Function

Given an input vector, the ArgMax function returns the index of the element with the maximum value. It is frequently used in the output layer of the neural networks, to identify which class has the highest probability, therefore, the network predicted matches best the given input to the network. For example, given as input to the ArgMax function the vector:

$$x = ([1, 2, 3, 100, 4, 5])$$

the output will be:

$$argmax(x) = [0, 0, 0, 1, 0, 0] = 4$$

where 4 represents that the 4^{th} element is the largest in x. Another example, if we give to ArgMax the input array:

$$arr = [-2, 1, 5, 0]$$

the same as in the SoftMax (Section 2.2.15), the output of the ArgMax will be:

$$argmax(arr) = [0, 0, 1, 0] = 3$$

2.2.16 Maximum & Minimum

Maximum and Minimum are respectively the largest and the smallest value of a function. For a given range, the largest and smallest value of a function is called local maximum and minimum, otherwise, for function's entire domain is called global maximum and minimum.

2.2.17 Mean

Mean, also known as average is the central value of a finite set. It is the sum of the set's values divided by the set's length.

2.3 Artificial Neural Networks Background

2.3.1 Linear Regression

Linear regression is a machine learning model where the relationship between independent variables x, and dependent variables y, is linear. This means that it predicts a line, an output that is continuous and has a constant slope (Figure 2.12). The line can be modeled by using the linear equation:

$$y = \alpha x + \beta$$

Therefore, in such a case we are searching for the values of α and β. Mostly, is being used for predicting values within a continuous range rather than classification. For example, given the same facial characteristics of a person to predict his age.

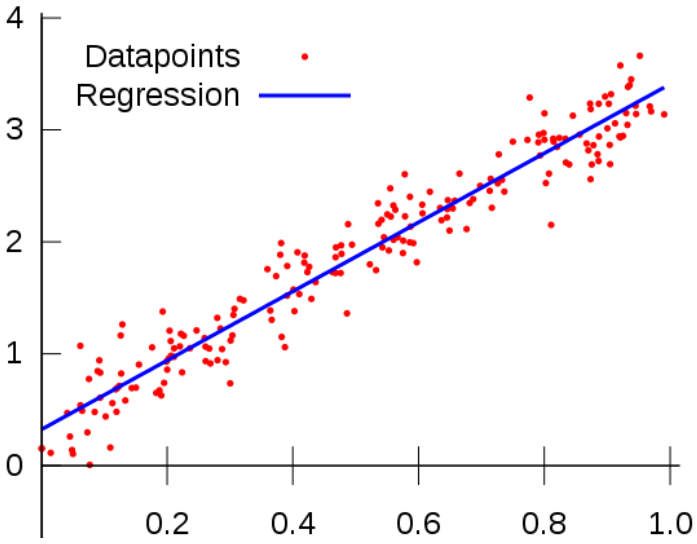


Figure 2.12. Linear Regression model. **Red dot:** Data. **Blue Line:** Best fit a straight line on the given data.

The two main types of regression are simple regression and multivariable regression. In this thesis, we are going to use classification (Section 2.3.2) instead of regression since we want to identify the exact class [AD, MCI, NC] which the patience fits better based on his T1-weighted MRI scan.

2.3.2 Classification

Classification is the process of predicting a categorical output. For example, given an email, predict whether the email is spam or not. This is an example of Binary classification since it predicts one of two possible outcomes. Consequently, if I have to predict one of the multiple possible outcomes, we call it Multi-class classification (Figure 2.13). An example of Multi-class classification is this thesis, which tries to predict the class of an MRI scan from the three possible classes [AD, MCI, NC].

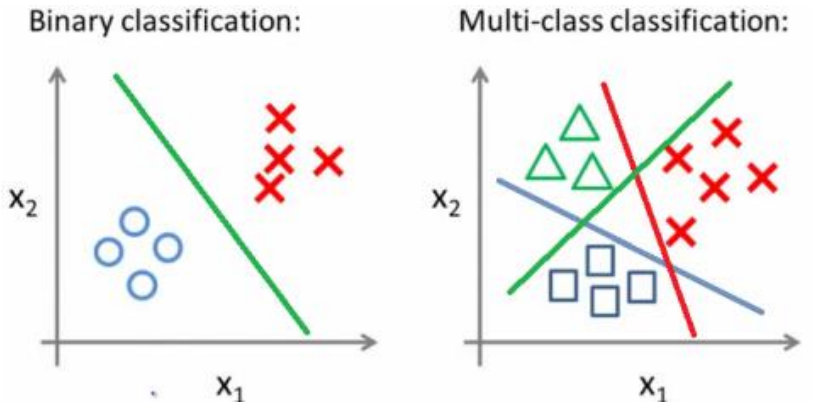


Figure 2.13. Left: Binary classification, classifying the two classes ('o's & 'x's)
Right: Multi-class classification, classifying the three classes ('o's, 'x's, & triangles)

2.3.3 Cost/Loss Function

The cost function or so-called loss function is being regularly used for weight optimization in ANNs (Sections 2.3.8). In general, loss functions measure how far an estimated value is from its true value. We are converting a learning problem into an optimization problem, where the goal is to minimize the loss function by optimizing the algorithm and consequently improve the accuracy of the model.

2.3.4 Mean Squared Error (MSE) Loss

An example of a lost function (Section 2.3.3) that is mostly used in regression problems (Section 2.3.1) is the Mean Squared Error (MSE) which measures the average squared difference between a target and a real output value of the network. The loss is always positive regardless of the sign of the predicted and actual values. A perfect model should have an MSE loss equal to 0. The MSE loss is not sensitive towards the outliers therefore is good to use it when the dataset contains only a small Gaussian noise, however, the model is inaccurate when the data contain outliers (Liano, 1996). This could be an issue

since most of our experiments use MSE loss and no outliers had been removed from our datasets.

2.3.5 Cross-entropy Loss

Another example of a loss function (Section 2.3.3) that is frequently used mostly for classification problems (Section 2.3.2) is the Cross-entropy. More specifically, Cross-entropy measures the performance of a classification model whose output is a value between 0 and 1. The loss of Cross-entropy increases as the predicted probability diverges from the actual label. For example, if the predicting probability is 0.023, and the actual label is 1, then this would result in a high loss value. The perfect model should have a perfect Cross-entropy loss equal to 0.

2.3.6 Supervised Learning

In a supervised learning algorithm, the training set consists of the training examples and the target outputs. The algorithm needs to learn a function that maps the input to the output based on input-output pairs. The goal is to predict the output of inputs that are not part of the training set, thus the algorithm had never seen them before.

The simplest form of such an algorithm is linear regression (Section 2.3.1). In linear regression, the algorithm tries to approximate a line that predicts a y value given as input an x value. Another supervised learning model is classification (Section 2.3.2) where it separates the inputs into two or more distinct classes.

2.3.7 Artificial Neuron

The artificial neurons, inspired by the biological neurons (Section 2.1.4), are a representation of a mathematical function (Section 2.2.1). They are the elementary units of an ANN (Section 2.3.8). Each input of the artificial neuron usually is weighted and the sum passes through an activation function.

2.3.8 Artificial Neural Network (ANN)

Artificial Neural Networks (ANNs) or Neural Networks (NNs) are networks of artificial neurons (Section 2.3.7) that are inspired by biological neural networks of animal and human brains. ANNs have usually the form of a graph, where each vertex is an artificial

neuron, that represents a biological neuron, and the edges of the graph represent the synapses.

Usually, each artificial neuron can receive a signal, process it, and send the signal to adjacent neurons. The edges can be directional or not, and are typically weighted, which value can be increased or decreased. Typically, the neurons are aggregated into layers, and the signal travels from the first layer (input layers) to the last layer (output layer). Also, the layers between the input and output layers are usually called the hidden layers. When an ANN has multiple hidden layers, the is frequently called Deep Neural Network (DNN). Different layers can apply different transformations to their inputs.

2.3.9 McCulloch and Pitts (MCP)

2.3.9.1 Introduction to MCP

The McCulloch & Pitts model is a single artificial neuron model (Section 2.3.8) that is being used for supervised learning (Section 2.3.6) of binary classifiers. In other words, the MCP model is a binary threshold unit that receives multiple weighted inputs and a bias, uses the step function (Section 2.2.11) as its activation function (Section 2.3.11.3), and generates outputs either 0 or 1 (Figure 2.14). The model was inspired by the biological neurons (Section 2.1.4) but cannot be considered biologically realistic. This model is limited to only linearly separable problems (Section 2.2.10). (McCulloch and Pitts, 1943)

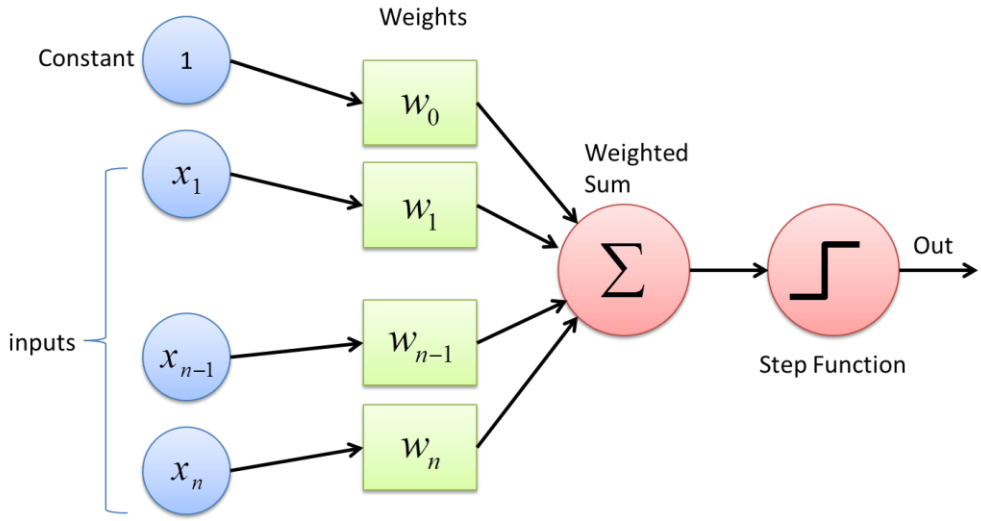


Figure 2.14. A McCulloch & Pitts neuron. Takes as input a vector of size n. The weight w_0 represents the bias. The weighted sum passes through the step function (activation function) and produces the output, which is either 0 or 1. <https://deepai.org/>

2.3.9.2 Mathematics of MCP

Let's assume X is the input vector and W is the vector of the weights, both of size n. The weighted sum is the dot product between the two vectors:

$$X \cdot W = \sum_{i=1}^{n} X_i W_i$$

Afterward, we pass as input the weighted sum into the Heaviside Step Function (Threshold Function), which returns 0 if the weighted sum was less than 0, otherwise returns 1. This is the output of the MCP neuron with W weights, given X as input.

2.3.10 Perceptron

Perceptron is an algorithm for supervised learning (Section 2.3.6) broadly used as a binary classifier, to identify whether an input vector belongs to a specific class. Perceptron is a single-layer feedforward network of MCP neurons (Section 2.3.9). The Perceptron learning algorithm is being used for training an MCP neuron but is limited to only linearly separable problems (Section 2.2.10). Let's assume the Perceptron takes as input a vector of size N, the process is as follows:

1. Randomly initialize N weights to small values close to 0.
2. Multiply the input vector with the weights to produce a weighted sum.
3. Apply the weighted sum to the activation function, to produce Perceptron's output. A simple activation function that is frequently being used for Perceptron is the Heaviside Step Function (Section 2.2.11) which takes as input the weighted sum and returns 0 if the sum is less than zero, otherwise returns 1.
4. Given the target output, since it is supervised learning, and the real output from step 3; adapt the weights based on the following rules:
    a. If the target output is the same as the real output, then the weights remain the same.
    b. If the target output is 1 and the real output is 0, then add to the weights the input vector multiplied by a scalar (learning rate).
    c. If the target output is 0 and the real output is 1, then subtract from the weights the input vector multiplied by a scalar (learning rate).

The weight adaption process transforms the decision line which separates the two classes. A bias can also be used to move the decision line vertically.

2.3.11 Multi-Layer Perceptron (MLP)

2.3.11.1 Introduction to MLP

Multi-Layer Perceptron is a type of feed-forward ANN (Section 2.3.10). Feed-forward means that the flow goes in a single direction from the input layer to the output layer and not the other way (Figure 2.15). MLPs are usually referred to as "vanilla" Neural Networks. MLPs consist of at least three layers of nodes; an input layer, one or two hidden layers, and an output layer. MLP implements backpropagation (Section 2.3.11.5), a supervised learning technique that is being used during training.

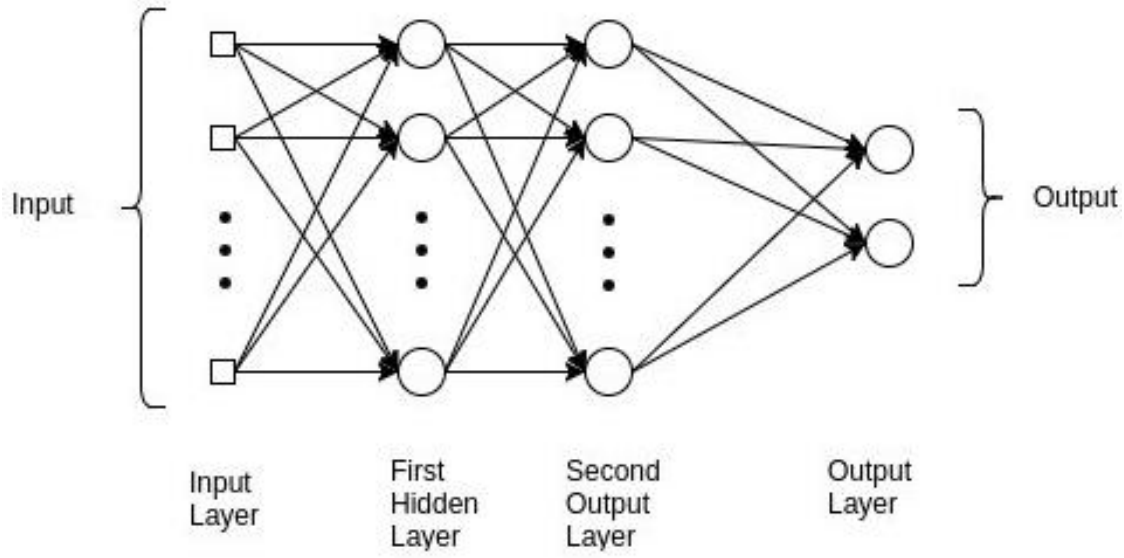


Figure 2.15. Multi-Layer Perceptron (MLP) with 3 layers with McCulloch & Pitts (MCP) neurons (two hidden, and an output layer). The input layer does not contain any neurons.

The issue with the Perceptron unit is that it is not able to work for non-linearly separable problems (Section 2.2.10). MLP on the other hand, due to its multiple layers and non-linear activation function (Section 2.3.11), can distinguish data that are not linearly separable. An example of a non-linearly separable problem is the XOR, which cannot be solved by using Perceptron (Figure 2.16).

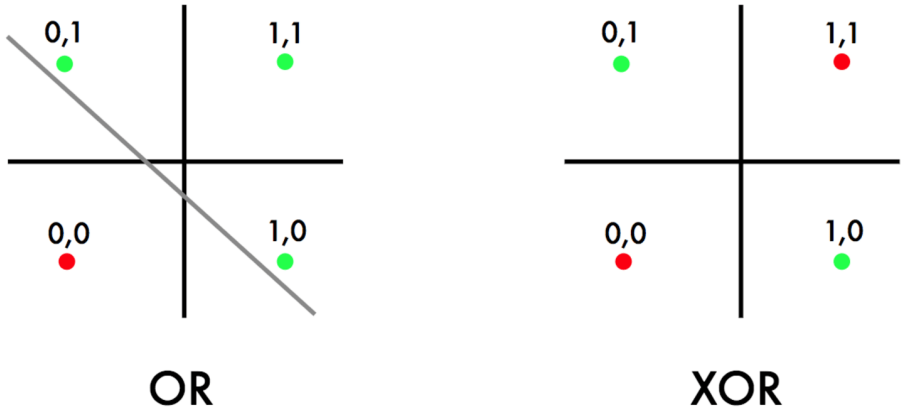


Figure 2.16. Left: 'OR' linearly separable problem. **Right:** 'XOR', non-linearly separable problem

2.3.11.2 Architecture

MLP is a feedforward, usually, fully-connected, network which consists of multiple layers of MCP neurons. The input layer does not consist of MCP neurons, it just provides the input to the hidden layers of the network. The nodes of the hidden and output layers are MCP neurons (Section 2.3.9) that use a nonlinear activation function such as Sigmoid (Section 2.2.12). Three layers of Perceptron units (2 hidden & 1 output layer), can form arbitrary complex shapes, that are capable of separating any classes (Figure 2.17). Thus, based on Kolmogorov Theorem, no more than three layers are needed in an MLP network. (Lippmann, 1987). That is why MLPs are called universal approximations since they can approximate any function that we require (Csáji et al. 2001).

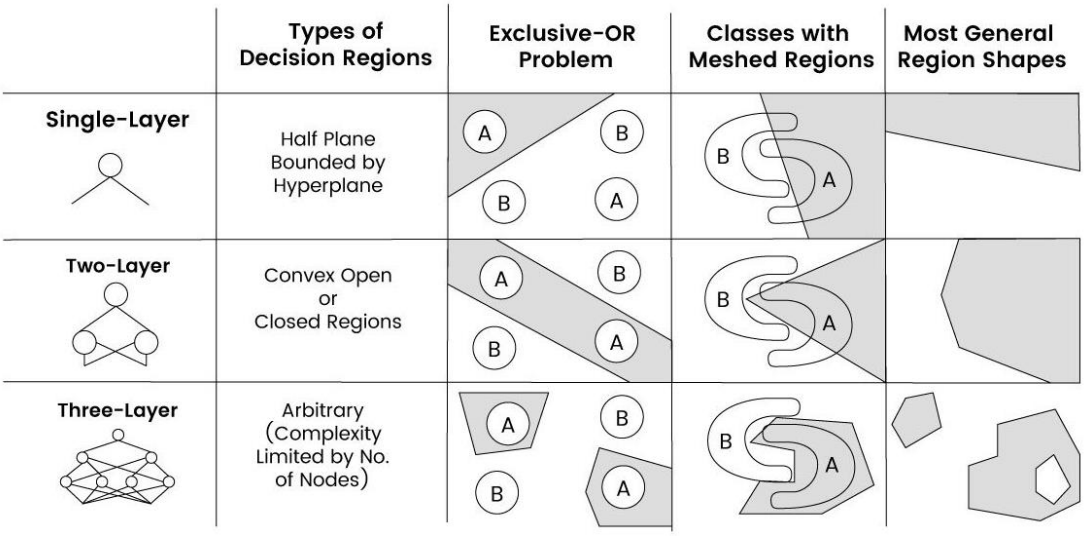


Figure 2.17. Three types of Multi-Layer Perceptron (MLP) and the different types of Decision Regions they create. **Single-Layer or Perceptron:** Solves only Linearly Separable problems. Creates Decision Lines/Planes/Hyperplanes. **Two-Layer:** Solves Non-linearly Separable problems. Constructs Convex Regions**. Three-Layer:** Separates any classes. Any Arbitrary Convex Region. <https://www.verypossible.com/insights/machine-learning-algorithms-what-is-a-neural-network>

2.3.11.3 Activation Function

Activation functions are being used in nodes of ANN (Section 2.3.8) to define the output of each node given an input. One of the simplest forms of binary activation functions, that is being used for Perceptron (Section 2.3.10), is the Heaviside Step Function (Section 2.2.11) which outputs 1 if the input is greater than zero, otherwise outputs 0. Another example of an activation function is the Sigmoid/Logistic (Section 2.2.12), which is frequently being used in MLPs (Section 2.3.11). The most frequently used activation functions can be divided into three categories: ridge functions, radial functions, and fold functions.

Ridge functions act on a linear combination of the input variables since they are multivariate functions. The most frequently used ridge functions are Linear, ReLU, Heaviside, and Sigmoid/Logistic. The activation function is a simplified abstract representation of action potential in biological neurons (Section 2.1.4). The Heaviside Step Function (Section 2.2.11) represents a biological neuron that is firing or not. The Linear function with its positive slope reflects the increase in firing rate when input current increases. The Rectified Linear Unit (ReLU) (Section 2.2.13) simulates the biological neurons that cannot lower their firing rate below zero. Last but not least, Sigmoid (Section 2.2.12) imitates the neurons that cannot fire faster than a certain rate due to their refractory period.

Radial activation functions are also known are radial basis functions (RBFs) and are being used in RBF networks. RBF networks are extremely capable of function approximation. Some of these activation functions are the Gaussian, Multiquadratics, and Polyharmonical Splines.

Folding activation functions are often used in the pooling layers of Convolutional Neural Networks (CNN). These types of functions perform an aggregation over the inputs. Examples of Folding activation functions are the Mean (section 2.2.17), Minimum or Maximum (Section 2.2.16), and SoftMax (Section 2.2.14).

2.3.11.4 Forward Propagation

In forward propagation, the input data given to a network, are propagated forward through the network until the final layer where it outputs the prediction.

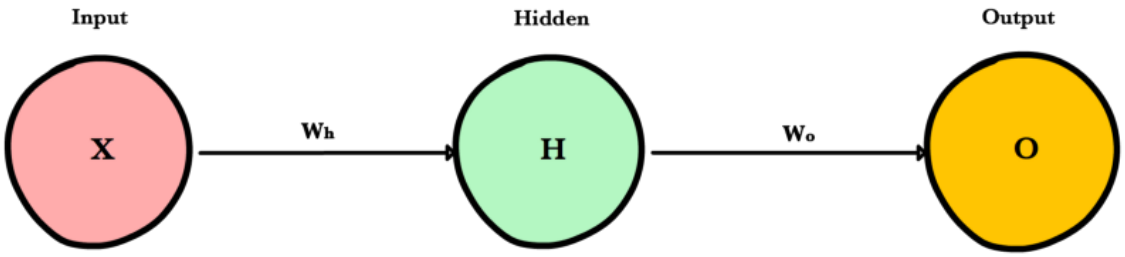


Figure 2.18. Two-layer network with a single neuron per layer.

Given a simple two-layer neural network with a single hidden neuron and a single output neuron (Figure 2.18), the outcome of a forward propagation for an input X, and an activation function A (ReLU, Sigmoid, etc.), would be:

$$Prediction = A(A(XW_h)W_o)$$

Pseudocode for the forward propagation algorithm for the neural network in Figure 2.19 follows in Code Snippet 2.1.

```
# Forward propagation algorithm

def relu(z):
    return max(0,z)

def feed_forward(x, Wh, Wo):
    # Hidden layer
    Zh = x * Wh
    H = relu(Zh)

    # Output layer
    Zo = H * Wo
    output = relu(Zo)
    return output
```

Code Snippet 2.1. Forward propagation of a two-layer network with a single neuron per layer.

2.3.11.5 Backpropagation

In backpropagation, each weight of the networks is being adjusted based on how much it contributes to the overall error. Backpropagation is needed to propagate back the error inside the network to be able to adjust the weights of the hidden layers. This the algorithm that enabled us to solve non-linearly separable problems (Sectopm 2.2.10) by using MLPs (Section 2.3.11), a major limitation that the Perceptron algorithm had (Section 2.3.10).

Since forward propagation is a long series of nested equations, backpropagation is merely an application of chain rule (Section 2.2.5) to find derivatives (Section 2.2.4) of cost with respect to any variable in the nested equation. Given x the network's inputs, and A, B, C

the activation functions in three different layers, the forward propagation function would be:

$$f(x) = A\Big(B\big(C(x)\big)\Big)$$

Using the chain rule, the derivative $f'(x)$ of $f(x)$ for x is:

$$f'(x) = f'(A) \cdot A'(B) \cdot B'(C) \cdot C'(x)$$

For example, to find the derivative of B, we pretend that $B(C(x))$ is a constant, we replace it with a placeholder variable b, and proceed to find the derivative with respect to b:

$$f'(b) = f'(A) \cdot A'(b)$$

We can use the chain rule to calculate the derivative of cost for any weight in the network, where the chain rule will identify how much each weight contributes to the overall error, and in which direction to update each weight to reduce the error. Given the weighted input Z, the ReLU activation R, and the cost function C:

$$Z = XW$$

$$R = \max(0, Z)$$

$$C = \frac{1}{2}(\hat{y} - y)^2$$

Their derivatives are:

$$Z'(X) = W,\ Z'(W) = X$$

$$R'(Z) = \begin{cases} 0 & Z < 0 \\ 1 & Z > 0 \end{cases}$$

$$C'(\hat{y}) = (\hat{y} - y)$$

The $\hat{y}$ represents the predicted output of the network while the y is the target/actual output. The cost of a network with a single neuron can be calculated by using:

$$Cost = C(R(Z(XW)))$$

Figure 2.19 shows the two weighted inputs in the hidden and output layer, given as input to the neural network the value X. The Z_h and Z_o represent the inputs of the hidden and output layer respectively.

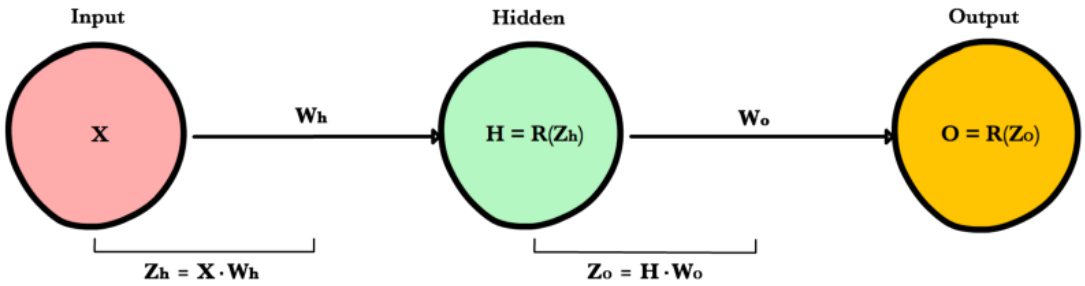


Figure 2.19. Calculating the weighted input of the two-layer network with a single neuron on each layer.

The derivative of the cost with respect to weight W can be calculated by using the chain rule:

$$C'(W) = C'(R) \cdot R'(Z) \cdot Z'(W) = (\hat{y} - y) \cdot R'(Z) \cdot X$$

Given a neural network with a single hidden and a single output layer (Figure 2.19), the derivative of the cost with respect to W_o is:

$$C'(W_o) = C'(\hat{y}) \cdot \hat{y}'(Z_o) \cdot Z'_o(W_o) = (\hat{y} - y) \cdot R'(Z_o) \cdot H$$

To find the derivative of the cost with respect to W_h, we have to apply the chain rule recursively:

$$\begin{aligned} C'(W_h) &= C'(\hat{y}) \cdot O'(Z_o) \cdot Z'_o(H) \cdot H'(Z_h) \cdot Z'_h(W_h) \\ &= (\hat{y} - y) \cdot R'(Z_o) \cdot W_o \cdot R'(Z_h) \cdot X \end{aligned}$$

The layer error is the derivative cost with respect to a layer's input. To calculate the output layer error E_o, we need to find the derivative of cost with respect to the output layer's input, Z_o.

$$E_o = C'(Z_o) = (\hat{y} - y) \cdot R'(Z_o)$$

To calculate hidden's layer error E_h, we need to find the derivative of cost with respect to hidden layer input, Z_h.

$$E_h = C'(Z_h) = (\hat{y} - y) \cdot R'(Z_o) \cdot W_o \cdot R'(Z_h)$$

So by replacing E_o we get:

$$E_h = E_o \cdot W_o \cdot R'(Z_h)$$

This is the core of the backpropagation algorithm. The error of the current layer is being calculated, and the weighted error is being based on the previous layer, continuing the

process until we arrive at the first hidden layer. Along the way, we update the weights using the derivative of cost with respect to each weight.

The formula for the derivative of cost with respect to the output layer weight W_o, by replacing E_o becomes:

$$C'(W_o) = E_o \cdot H$$

Consequently, to find the derivative of cost with respect to any weight in the network, the following equation can be used:

$$C'(w) = CurrentLayerError \cdot CurrentLayerInput$$

Figure 2.20 visualizes step by step the backpropagation algorithm of a network with two layers and a single neuron per layer.

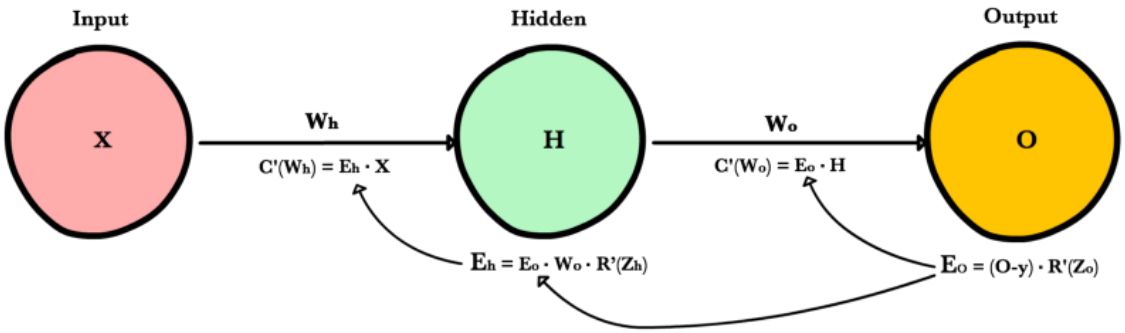


Figure 2.20. Visualization of the backpropagation process.

The Code Snippet 2.2 is the backpropagation algorithm applied in the network of Figure 2.20, which has two layers, a hidden and an output layer, with a single neuron each.

```
# Backpropagation algorithm

def relu_prime(z):
    if z > 0:
        return 1
    return 0

def cost(yHat, y):
    return 0.5 * (yHat - y)**2

def cost_prime(yHat, y):
    return yHat - y

def backprop(x, y, Wh, Wo, lr):
    yHat = feed_forward(x, Wh, Wo)

    # Layer Error
    Eo = (yHat - y) * relu_prime(Zo)
    Eh = Eo * Wo * relu_prime(Zh)
```

```
    # Cost derivative for weights
    dWo = Eo * H
    dWh = Eh * x

    # Update weights
    Wh -= lr * dWh
    Wo -= lr * dWo
```

Code Snippet 2.2. Backpropagation algorithm for a two-layer network with a single neuron per layer

2.3.12 Convolutional Neural Network (CNN)

2.3.12.1 Introduction to CNN

Convolutional Neural Networks (CNNs / ConvNets) are very similar to Neural Networks since they consist of neurons with adjustable weights and biases. They process data that come in the form of arrays. For example, a grayscale image is composed of a 2D array of integers between 0 and 255. More specifically, they are specialized in feature extraction from multidimensional arrays: 1D for sounds, or language processing, 2D for images, and 3D for videos, or volumetric images such as the T1-weighted MRIs (LeCun et. al. 2015).

Some applications of CNNs are image and video recognition, image classification, image segmentation, medical image analysis, natural language processing recommender systems, and time series. CNNs were inspired by biological processes and the connectivity patterns between biological neurons of the cat's virtual cortex. Each cortical neuron responds to a stimulus only in a restricted region of the visual field. For example, some cortical neurons are responsible to recognize moving lines of specific rotation and direction (LeCun et al. 1998).

CNNs are a variation of MLPs (Section 2.3.11), which means are fully connected networks, such as each neuron in one layer is connected to all neurons of the next layer. Overfitting (Section 2.3.13.1) may occur because CNNs are fully connected. As a result, regularization techniques (Section 2.3.13.2) such as data augmentation (Section 2.3.13.5), dropout (Section 2.3.13.9), and early stopping (Section 2.3.13.12) are being used to prevent this. The CNN detects features, such as straight and diagonal lines from the input image, assembles them, to create more complex feature detectors that can recognize more complicated structures such as faces, birds, clothes, cars, figures, etc. (LeCun et. al. 2015). The four main key ideas behind CNNs are the local connections, the shared weights, the pooling, and the use of many layers. Unlike MLPs, more layers for CNN imply better performance (LeCun et. al. 2015).

2.3.12.2 Architecture

The architecture of the CNNs I am going to mostly use in this thesis consists of two main components. The first component performs feature extraction in the input array. The first component is responsible for feature extraction and is composed of two types of layers; the Convolutional Layers (Section 2.3.12.4) and the Pooling/Sub-sampling Layers (Section 2.3.12.5). The second component performs the classification, and it is a fully connected network which usually is an MLP (Section 2.3.11) (Figure 2.21).

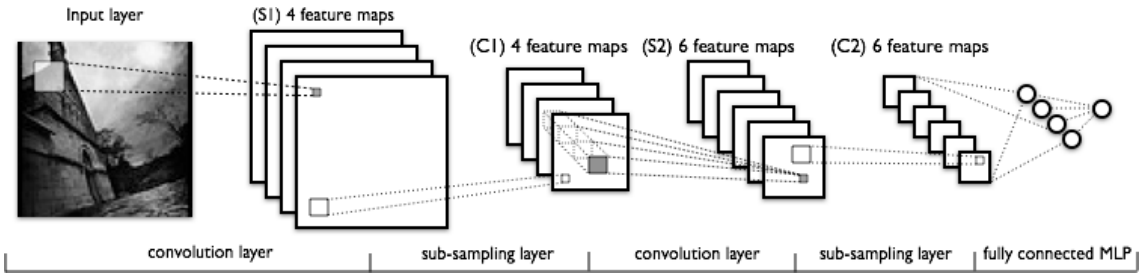


Figure 2.21. Example of Convolutional Neural Network (CNN) architecture with the Convolutional Layers, Pooling/Sub-sampling Layers and A Fully Connected MLP

2.3.12.3 Filter/Kernel

The filter, kernel, Convolution matrix, or mask is a matrix that is being used in image processing for blurring, sharpening, embossing, edge detection, and more. To achieve these results, you must apply Convolution (Section 2.2.9) between a filter and an image.

An example of a filter that detects the edges of an image and is being frequently used for image processing is the Laplacian 3×3 Edge Detection filter. The filter is a 3×3 matrix of integers with the following values:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 2.22 is an example of an application of the Laplacian 3×3 Edge Detection filter. Given the image (left) of a butterfly, it generated a grayscale image (right) which enhances the edges of the image.

Figure 2.22. Left: An image of a butterfly. **Right:** The application of Convolution between the Laplacian 3x3 Edge Detection filter and the image on the left. <https://softwarebydefault.com/tag/laplacian-of-gaussian/>

2.3.12.4 Convolutional Layers

In general, a Convolutional Layer, as its name implies, applies Convolution (Section 2.2.9) to the input and passes its result to the next layer. More specifically, a Convolutional Layer consists of units that are organized in feature maps. A kernel/filter is a set of weights that connects each one of the units with local receptive fields/image patches in the feature maps of the previous layer. The results of the weighted sum, between the image patches and the filters, pass through a non-linearity such as Rectified Linear Unit (ReLU) (Section 2.2.13) (LeCun et. al. 2015).

All units of a feature map share the same filter (weight sharing), and different feature maps in a layer corresponding to different filters. This is required since a feature could be anywhere across the entire image. For example, if a feature detector, searches for a bird's beak, by using weight sharing in that filter, we can discover the beak, even if it is in the right bottom, top left, the center, or anywhere else in that image. This filtering operation that is performed by a feature map mathematically represents a discrete Convolution, hence the name (LeCun et. al. 2015) (Figure 2.23).

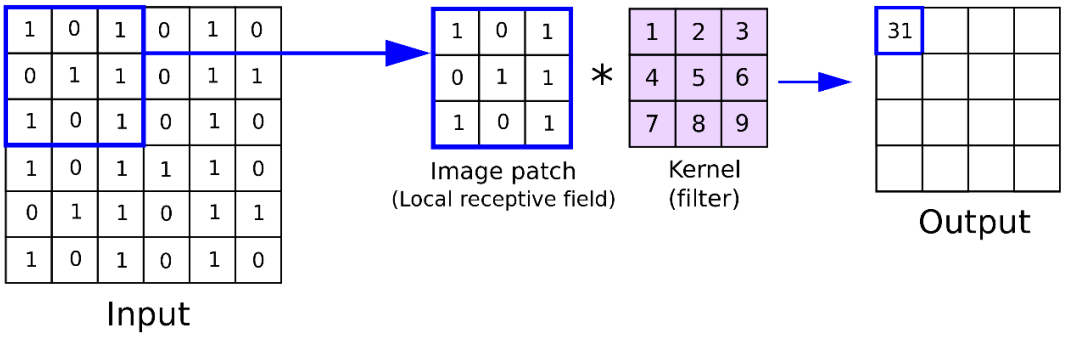


Figure 2.23. Convolutional Layer process in CNN. Convolution, the dot product of two matrices: an image patch and the filter/kernel. (Ph.D. Student Ahh Reynolds, Northwestern University, 2019)

2.3.12.5 Max-Pooling

Max-Pooling is the most common type of pooling applied in the pooling layers of a CNN. It reduces the dimensions of the data, by selecting the maximum value of an $n \times n$ tile of the previous layer to be the input in the next layer in a 2D CNN. Usually, 2×2 tiles are being used in 2D CNNs (Figure 2.24) or $2 \times 2 \times 2$ blocks in 3D CNNs. An alternative type of pooling is the average pooling which takes the average of an $n \times n$ tile.

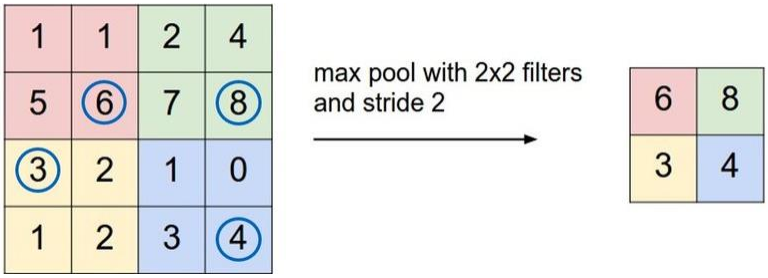


Figure 2.24. Max-Pooling with a tile size of 2x2.
Left: Output of the previous layer. **Right:** Input to the next layer after applying Max-Pooling.

2.3.12.6 Stride

Stride is used during Convolution (Section 2.3.12.4) and Max-Pooling (Section 2.3.12.5). For the Convolution, the Stride controls how the filter convolves around the input volume. For stride equal to 1, the filter shifts 1 pixel horizontally or vertically per filter application in the input volume. Given an input volume of size $n \times n$, a size of kernel/filter $k \times k$, and a stride s, results in an output volume of size:

$$[(n - k + s)/s] \times [(n - k + s)/\mathrm{s}]$$

For the input volume of size 7×7 and a kernel 3×3, the size of the output volume will be 5×5 (Figure 2.25).

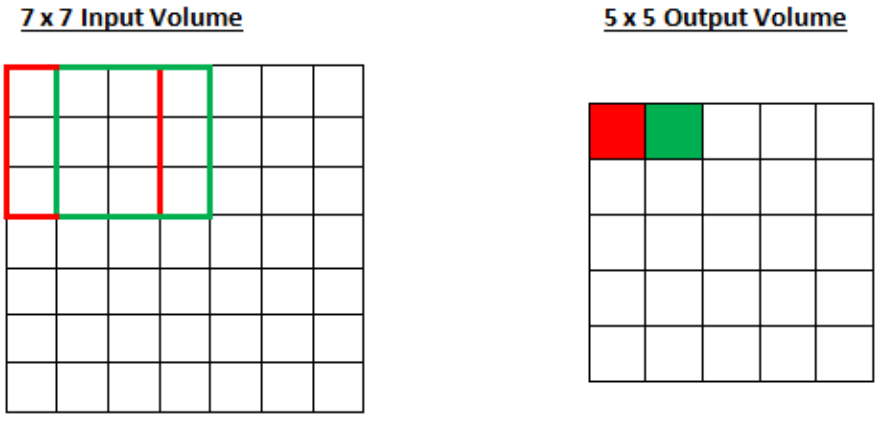


Figure 2.25. Convolution of a 3 x 3 filter with stride = 1.

If the stride was equal to 2, on each filter application the filter would shift by 2 pixels horizontally or vertically. Given an input volume of size 7×7, results in an output volume of size 3×3 (Figure 2.26).

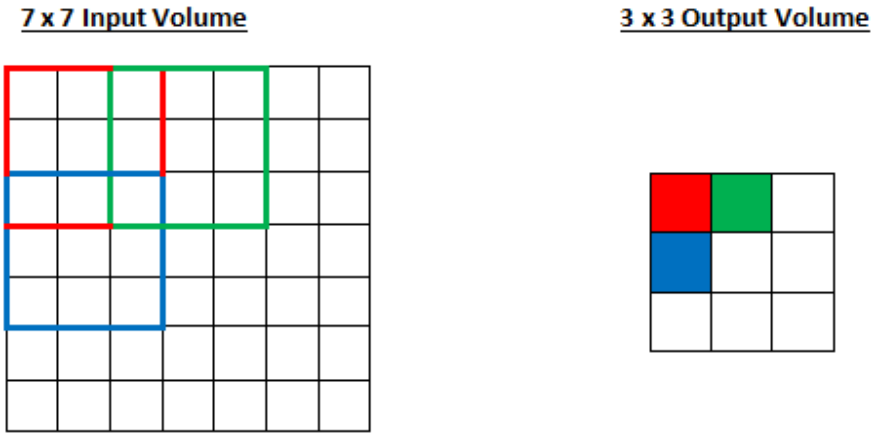


Figure 2.26. Convolution of a 3 x 3 filter with stride = 2.

In terms of the value of the stride during Max-Pooling, usually, it is equal to 2, Therefore the 2×2 filter shifts 2 pixels horizontally or vertically (Figure 2.27). In the experiments of this thesis, the stride for the Convolution that is going to be used will be equal to 1, and for the Max-Pooling equal to 2.

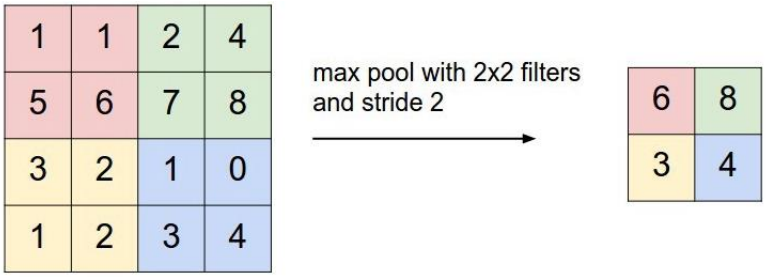


Figure 2.27. Max-Pooling with stride = 2.

2.3.12.7 Padding

During Convolution (Section 2.3.12.4), the application of filters with stride = 1 causes the image to lose its perimeter because the size of the image is reduced from $n \times n$ to $(n-2) \times (n-2)$. Therefore, a few pixels are lost on each Convolutional Layer, but this adds up as we apply many successive Convolutional Layers. A solution to this issue is to apply padding the perimeter with 0s before applying the Convolution (Figure 2.28).

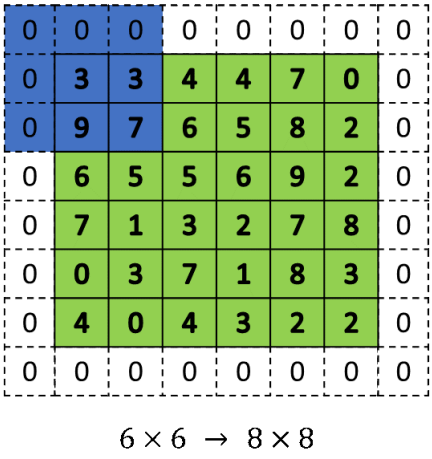


Figure 2.28. Padding of 0s applied before Convolution.

2.3.12.8 Forward Propagation in CNNs

The forward propagation between the fully connected layer, the Convolutional Layers, and the Max-Pooling layers, works slightly differently. For the fully connected layers, the forward propagation remains the same as in a simple MLP (Section 2.3.11.4). Therefore, now we are going to focus on the forward propagation of the Convolutional and Max-Pooling layers.

2.3.12.8.1 Forward Propagation in Convolutional Layers

Suppose we have an $N \times N$ input layer that is followed by a Convolutional Layer. Given an $m \times m$ filter w, the output of the Convolutional Layer before applying any 0s padding would be $(N - m + 1) \times (N - m + 1)$. To compute some unit x_{ij}^{l} in the new layer l, we sum up the contributions weighted by the filter's components of the previous layer $l - 1$:

$$x_{ij}^{l} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} w_{ab} y_{(i+a)(j+b)}^{l-1}$$

Then we apply the nonlinearity/activation function σ in the Convolutional Layer:

$$y_{ij}^{l} = \sigma(x_{ij}^{l})$$

2.3.12.8.2 Forward Propagation in Max-Pooling Layers

The Max-Pooling layers are very simple since they take some region $k \times k$, and the output a single value, the maximum of that region. If the stride is equal to k, given an input layer $N \times N$, the output will be an $\frac{N}{k} \times \frac{N}{k}$ layer.

2.3.12.9 Backpropagation in CNNs

For the backpropagation in CNNs, there are two types of updates performed in the Convolutional Layers, for the weights and the deltas.

2.3.12.9.1 Backpropagation in Convolutional Layers

Given an error function E, and we know the error values at the Convolutional Layer, therefore the partial derivative with respect or each neuron output:

$$\frac{\partial E}{\partial y_{ij}^{l}}$$

To figure out what the gradient component is for each weight, we can apply the chain rule (Section 2.2.5), which will sum the contributions of all expressions in which the variable occurs.

$$\frac{\partial E}{\partial w_{ab}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m}\frac{\partial E}{\partial x_{ij}^{l}}\frac{\partial x_{ij}^{l}}{\partial w_{ab}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m}\frac{\partial E}{\partial x_{ij}^{l}} y_{(i+a)(j+b)}^{l-1}$$

We sum overall x_{ij}^{l} expressions in which w_{ab} occurs. This corresponds to the weight-sharing in the Convolutional neural networks. We knew from forwarding propagation that:

$$\frac{\partial x_{ij}^{l}}{\partial w_{ab}} = y_{(i+a)(j+b)}^{l-1}$$

To compute the gradient, we need to know the deltas:

$$\delta_{i,j}^{l} = \frac{\partial E}{\partial x_{ij}^{l}}$$

which are fairly straightforward to be computed, by using the chain rule again and the derivative of the activation function σ:

$$\frac{\partial E}{\partial x_{ij}^{l}} = \frac{\partial E}{\partial y_{ij}^{l}}\frac{\partial y_{ij}^{l}}{\partial x_{ij}^{l}} = \frac{\partial E}{\partial y_{ij}^{l}}\frac{\partial}{\partial x_{ij}^{l}}\left(\sigma\left(x_{ij}^{l}\right)\right) = \frac{\partial E}{\partial y_{ij}^{l}}\sigma'\left(x_{ij}^{l}\right)$$

We also need to propagate the errors back to the previous layers to compute the weights for this Convolutional Layer, so we apply once more the chain rule.

$$\frac{\partial E}{\partial y_{ij}^{l}} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1} \frac{\partial E}{\partial x_{(i-a)(j-b)}^{l}} \frac{\partial x_{(i-a)(j-b)}^{l}}{\partial y_{ij}^{l}} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1} \frac{\partial E}{\partial x_{(i-a)(j-b)}^{l}} w_{ab}$$

2.3.12.9.2 Backpropagation in Max-Pooling Layers

The Max-Pooling layers do not provide any learning, since they just introduce sparseness. In forward propagation, $k \times k$ blocks are reducing to a single value. From the backpropagation of the next Convolutional Layer, the single value acquires an error, and that error is propagated back to the $k \times k$ block where it came from.

2.3.12.10 Types of Convolution

Based on the input data of a CNN for a certain problem, the type of Convolution applied changes accordingly. Some CNN architectures, frequently used in machine learning are 1D CNNs, 2D CNNs, and 3D CNNs. These architectures apply respectively 1D, 2D, and 3D Convolutions in their Convolutional Layers.

2.3.12.10.1 Convolution Type: 1D Convolutions

1D Convolutions are used on sequence datasets and can be used to extract local 1D subsequences from input sequences and identify local patterns within the window of the Convolution (Figure 2.29). An example of usage is in Natural Language Processing (NLP) where every sentence is represented as a sequence of words. No dataset in this thesis will use 1D Convolutions.

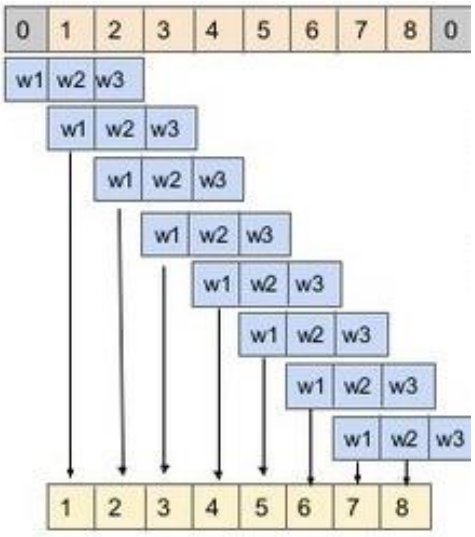


Figure 2.29. Example of 1D Convolution

2.3.12.10.2 Convolution Type: 2D Convolutions

2D Convolutions are mostly used on image datasets. The filter moves in 2-dimensions, horizontally and vertically to calculate the dimensional features from the image data. The output feature map is still a 2D matrix (Figure 2.30). For the 2D slice-level Brain MRI scans in this thesis, 2D CNNs network architectures with 2D Convolutions applied are going to be used.

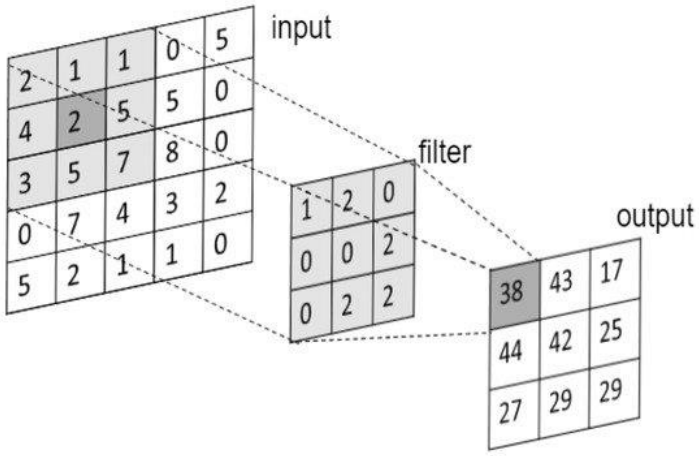


Figure 2.30. Example of 2D Convolution

2.3.12.10.3 Convolution Type: 3D Convolutions

3D Convolutions apply a 3-dimensional filter to the dataset and the filter moves in 3 directions to calculate the feature representations. The output feature map is a 3D matrix as well (Figure 2.31). They are helpful in event detection in videos, 3D medical images, etc. In this thesis, for the 3D subject-level Brain MRI scans, and the 3D Left Hippocampus dataset, network architectures with 3D Convolutions are going to be used.

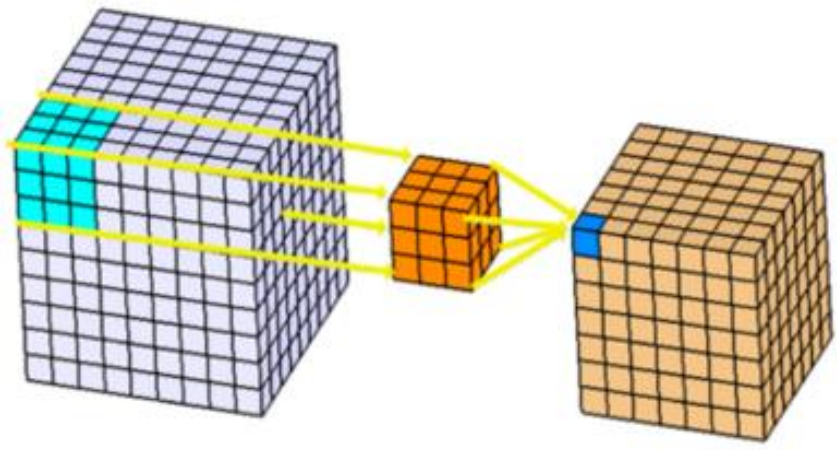


Figure 2.31. Example of 3D Convolution with a 3x3 filter (orange), and stride = 1.

2.3.13 Overfitting / Regularization

2.3.13.1 Overfitting

Overfitting in neural networks is the lack of the network's ability to predict inputs that had never been seen before. That is to say, the network learns extremely well its training set throughout the training process, and as a result, it fails to generalize. This implies much poorer accuracy while predicting future observations, than the accuracy levels performed during fitting.

2.3.13.2 Regularization Techniques

Several techniques are available to find overfitting during the training process of a CNN. Those techniques are so-called regularization techniques and are combating overfitting to improve models' performance on unseen data. As a result, the model generalizes better and achieves better accuracies during testing.

Some of them are Cross-validation, train with more data, remove features, change the dataset, data augmentation, L1 & L2 Regularization, Dropout, Spatial Dropout, early stopping, use a shuffled and balanced dataset, use different activation functions, and batch normalization. In this thesis, most of these techniques will be tested in practice to compare their performance results.

2.3.13.3 Cross-validation (CV)

Cross-validation (CV) is one of the strongest allies against overfitting. It allows us to tune the hyperparameters with the whole initial dataset. That means, that we do not need to separate the set into a test and training set, which will make us lose a percentage of the initial set just for testing. We can use k-fold CV (Section 3.1.2), which splits the dataset into k-folds, and utilizes efficiently all the data in the initial dataset since all of them will be used as part of the training set.

A relatively bad decision we made on this thesis, was to isolate some data for the test set and then use the rest of the data for the CV. There is no real benefit from doing this since we just lost the benefits of CV against overfitting because that data could be used for our training process since the initial dataset was already too small. In future work, I highly suggest using a 5-fold CV and initially do not isolate data for the testing.

2.3.13.4 Number of Samples

Increase the number of samples in the original dataset, and make sure that the data are not noisy. In this thesis, we did not isolate the outliers which is a potentially major drawback for us, since the MSE loss function used is not as sensitive as other loss functions to outliers (Liano 1996).

We tried to increase the number of samples by using multiple scans of different periods of the same patient, or multiple slices of the same 3D MRI scan. An assumption could be made that this decision could encourage the model to overfit instead of preventing it since the model can learn explicitly unique features of a specific patient (e.g., the shape of the brain). For example, if ten scans were used from the same patient, and 7 different slices per scan as input, those sum up in total to 70 samples of 2D MRIs from the same patient. In the case that this patient has a unique feature such as a smaller right hemisphere, this could potentially harm the training process and therefore increase overfitting.

2.3.13.5 Data Augmentation

Having more training data increases the performance of the ML model. To obtain those data is being proven an extremely difficult task, especially for medical images, since the data have to be labeled as well. Data augmentation is a regularization technique that helps to reduce the aforementioned problem by generating new training data from the given original dataset. It increases the amount of training data, cheaply and easily. Examples of data augmentation on images are horizontal/vertical shifting and flipping, random rotation, zoom, changing the brightness, adding noise, etc. (Figure 2.32).

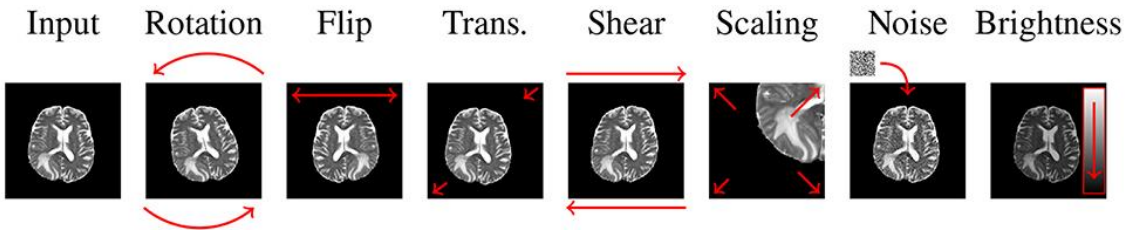


Figure 2.32. Data augmentation techniques of a 2D slice of a brain MRI scan. (Nalepa et al. 2019)

Unfortunately, due to the lack of time in this thesis, data augmentation was not tested in practice. This technique should be really helpful since the size, rotation, and brightness between scans are usually different. Only flipping the image should not be helpful in the case of MRI scans.

2.3.13.6 Different Datasets

Changing completely the dataset at some point may be the only solution against overfitting. If a lot of regularization techniques against overfitting do not show any improvement then maybe the initial data used are not the appropriate ones for the problem. In our case, maybe more detailed scans are needed, or a dataset with the original dimensions of the T1-weighted MRI brain scans, or a different perspective for the 2D slices such as the Horizontal or Sagittal slice instead of the Coronal.

2.3.13.7 L1 & L2 Regularization

Regularization aims to reduce the complexity of a model by adding a penalty term to the loss function. Two common techniques of regularization are being used L1 & L2 which can either be used separately or together (Andrew, 2004). In this thesis, both of these techniques were applied (L2 alone, L1 & L2) with no major improvements.

The L1 (Lasso Regression) penalty aims to minimize the absolute value of the weights (Figure 2.33). The L1 helps to generate a model that is simple and interpretable and is robust to outliers. In general, L1 shrinks the less important feature's coefficient to zero, thus it is good for feature selection.

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Cost function

Figure 2.33. Loss/Cost function with L1 applied (yellow highlighted term). For λ = 0, it is like not using L1.

The L2 (Ridge Regression) penalty aims to minimize the square magnitude of the weights (Figure 2.34). The L2 can learn complex data patterns but is not robust to outliers. Usually, L2 is a better choice than L1 regularization since it is better to learn inherent patterns present in complex data. L1 on the other hand is robust to outliers.

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Cost function

Figure 2.34. Loss/Cost function with L2 applied (yellow highlighted term). For λ = 0, it is like not using L2.

In Code Snippet 2.3, kernel regularizer applies a penalty on the layer's kernel/filter, bias regularizer, applies a penalty on the layer's bias, and activity regularizer applies a penalty on the layer's output.

```
import tensorflow.compat.v1 as tf
. . .
tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
            kernel_regularizer=tf.keras.regularizers.l1_l2(l1=1e-5, l2=1e-4),
            bias_regularizer=tf.keras.regularizers.l2(1e-4),
            activity_regularizer=tf.keras.regularizers.l2(1e-5)
            )
. . .
```

Code Snippet 2.3. L1 and L2 regularization in a Convolutional Layer. λ = 1e-4 for L1, λ = 1e-5 for L2.

2.3.13.8 Weight Decay (C)

After calculating the loss function, to penalize complexity, we could add to all weights the loss function, but this would not work because some weights are negative and others are positive. We could add the square of all weights, but this might result in a huge loss and the best model would be the one with all the weights being equal to 0.

A solution to this problem is the weight decay or so-called in Wang et al. (2020) the regularization parameter C. We are using the term C in this thesis in most of the experiments with CNNs since the implementations are based on Wang et al. (2020). We multiply the sum of the squares with C, the weight decay to avoid the situation where the loss gets too large. The loss function when using MSE loss, as we do in most of our experiments, is equal to:

$$Loss = MSE(\hat{y}, y) + C \cdot \sum_{i} {w_i}^2$$

Generally, values of C such as C = 0.1, or C = 0.01 works pretty well. It is worth mentioning that weight decay is not the same as L2 Regularization. For the two of them to be the same, the SGD optimizer should be used, and the learning rate to be equal to 1. Then the regularization term C for the weight decay is the same as the L2 regularization term, and so their effects are the same. In the case of Adam, AdaGrad, etc. optimizers the learning rate is adaptive, thus, the effect of the two regularization methods is different.

Usually, Adam performs poorly when L2 Regularization is used compare to SGD, while the weight decay performs equally well on both SGD and Adam.

2.3.13.9 Dropout

Dropout is a regularization technique that prevents overfitting as well. Is a frequently used regularization technique, especially in deep neural networks. It reduces overfitting by preventing complex co-adaptions on training data. Unlike L1 and L2 which are modifying the loss function, dropout modifies the network itself. It normally drops neurons from the neural network during training based on a probability (Srivastava et. al. 2014) (Figure 2.35). Code Snippet 2.4 shows an example of the Dropout layer used in this thesis.

```
import tensorflow.compat.v1 as tf
. . .
tf.keras.layers.Dropout(0.3)
. . .
```

Code Snippet 2.4. Dropout layer with drop rate = 0.3.

Therefore, the contribution of those neurons during the forward pass (Section 2.3.12.8) is removed and any weight updates are not being applied to them during the backward pass (Section 2.3.12.9). This means that on each iteration, different neurons are trained which helps to reduce overfitting. Dropout was applied in most of the implementations and it helped significantly the reduction of overfitting.

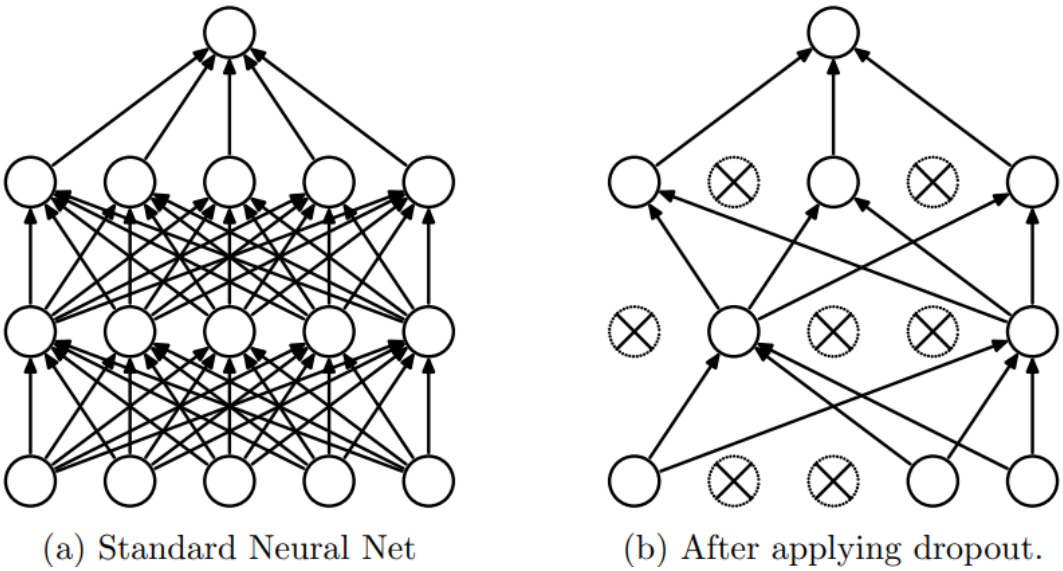


Figure 2.35. Left: A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped. (Srivastava et. al. 2014)

2.3.13.10 Spatial Dropout

Spatial Dropout works similarly to normal Dropout but in CNNs is preferable. The reason is that Dropout drops individual elements while Spatial Dropout drops entire 2D feature maps (Figure 2.36). The feature map is the output of one filter applied to the previous layer. This is preferable because if adjacent pixels within a feature map are strongly correlated, then regular Dropout will not recognize the activation and will otherwise just result in an effective learning rate decrease. In such a case, Spatial Dropout provides independence between feature maps (Tompson et al. 2015).

```
import tensorflow.compat.v1 as tf
. . .
tf.keras.layers.SpatialDropout2D(0.5)
. . .
```

Code Snippet 2.5. Spatial Dropout layer with drop rate = 0.5.

Spatial Dropout was applied in some of the experiments with the 2D Brain scans by using the code in Code Snippet 2.5. Anyhow, no major improvements were reported against the Dropout.

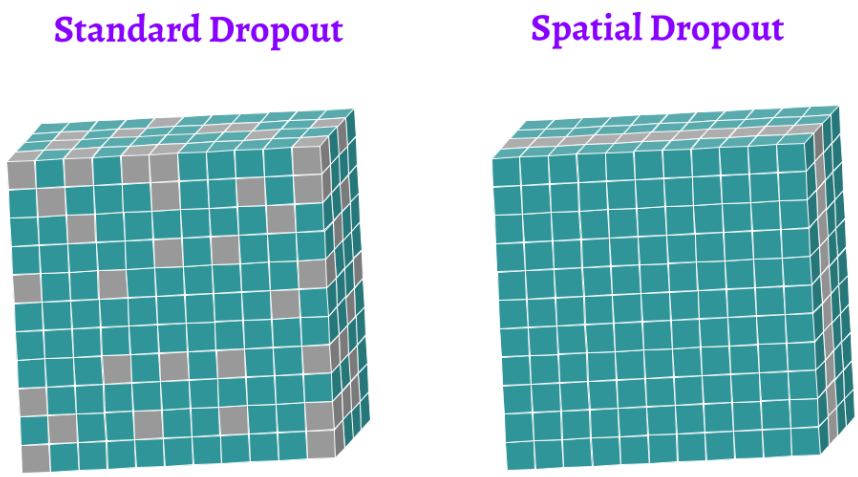


Figure 2.36. Left: Dropout drops randomly individual elements of each feature in the feature map.
Right: Spatial Dropout, drops the entire 2nd feature of the 4 features of the feature map.

2.3.13.11 Batch Normalization

Neural networks with multiple Convolutional Layers are sensitive to the initial random weights. The distribution of the inputs to layers deep in the network may change after each mini-batch when the weights are updated, and this can cause the algorithm to forever chase a moving target. The change of inputs distribution to layers in a network is referred to as “internal covariate shift” (Ioffe and Szegedy, 2015).

Batch normalization is a technique that standardizes the input to a layer for each mini-batch, which stabilizes the learning process and reduces the number of training epochs required, and additionally reduces overfitting. Standardizing means that the data are rescaled to have a mean of zero and a standard deviation of one (whitening).

Batch normalization was tested on some experiments of this thesis but was not being utilized the most. In Code Snippet 2.6 we can see an example of a BatchNormalization layer for a CNN from the scikit-learn's library. In future work, batch normalization should be applied before each Convolutional Layer and not just in the last layer.

```
import tensorflow.compat.v1 as tf
. . .
tf.keras.layers.BatchNormalization()
. . .
```

Code Snippet 2.6. Batch normalization layer.

2.3.13.12 Early Stopping

The training time of the model does not always improve the accuracy of the model. It may seem that the training accuracy increases but in reality, usually the test accuracy at some point starts to decrease. This means that the model overfits the training set.

Early stopping is a technique where you stop training just before the validation loss starts increasing while the training loss keeps decreasing. In such a case, if you do not do so, the training loss will continue to decrease while the generalization error will increase.

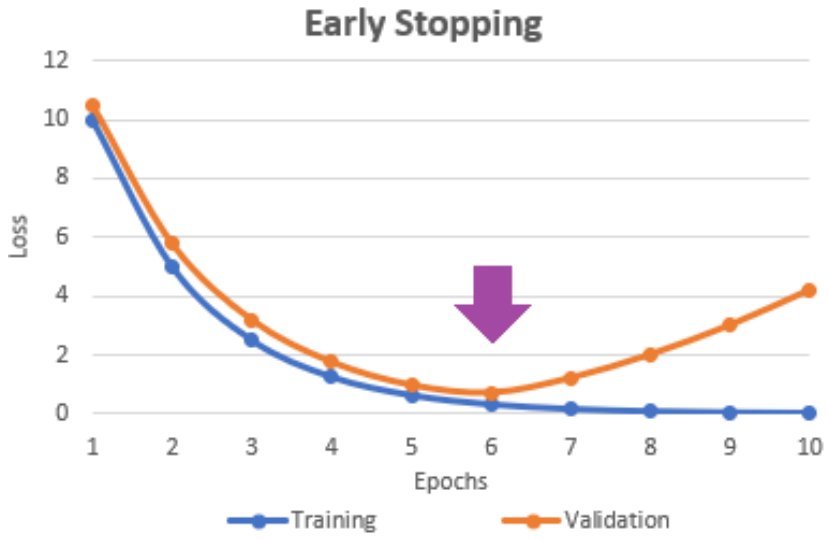


Figure 2.37. Early stopping example. The early stop training process at epoch 6, before validation loss starts increasing while training loss keeps decreasing

As shown in Figure 2.37, we should stop the training process just before the validation accuracy starts increasing. Consequently, early stopping is the method in which the training process is stopped when the accuracy of the model on the validation dataset starts to degrade. In all the experiments of this thesis, early stopping was used. Instead of saving the weights of the model in the last iteration, the model with the best validation accuracy was saved, which was the model just before overfitting.

2.3.13.13 Shuffled & Balanced Dataset

The datasets should be shuffled and balanced to avoid overfitting into one class. For example, if a dataset contains 50 ADs and 50 NCs, it is a mistake for the network to see first all the ADs and then all the NCs. This will destroy the learning of the ADs samples; therefore, the datasets have to be shuffled.

The dataset should be balanced, i.e., the training test and validation sets should have an equal number of samples for each class. Imagine a scenario wherein in a validation set there are 80 NCs and 20 ADs, and a model cannot recognize ADs. Therefore, the validation accuracy would be 80% since it correctly detects all the NCs but no ADs. Does this mean that our model is great? Not at all, in reality, it performs very badly. That is why the datasets need to be balanced and other metrics such as sensitivity and specificity should be taken into account to ensure that something like this will not happen.

2.3.13.14 Network Topology

Another way to reduce overfitting is by modifying the network's topology by decreasing the complexity of the model. We can do this by removing layers or reduce the number of feature maps in the case of CNNs. Usually, deep and narrow CNNs are preferred (Krizhevsky et al. 2012). Several experiments were performed in this thesis with different network topologies to fight against overfitting (Section 4.3).

2.4 Optimizers Background

2.4.1 Optimizers

During the training process, an algorithm tweaks and optimizes the weights of the model, to make the predictions more correct and accurate. The algorithm, which is responsible for when and by how much to update the weights, is called an optimizer. Optimizers

update the weights based on the output of the loss function. In other words, the loss function is the terrain that notifies the optimizer whether it is moving in the right or wrong direction towards the minimum. Examples of optimizers are SGD, AdaGrad, Adam, RMSProp, Newton's Method, etc.

2.4.2 Gradient Descent (GD)

Gradient Descent (GD) is the simplest algorithm for minimizing differentiable functions (Rumelhart et al. 1986). The gradient of a function is defined as the vector of its partial derivatives. As the gradient of a function always moves toward the maximum point, if we take any point and keep moving to the negative of the gradient, we will reach a local or global minimum. As we only use the first derivatives of the function in the gradient descent, this method is being called the first-order method. We can imagine this method not having any problems in a plane, but usually, on our problems the error surface is multidimensional and, in those cases, the curvature of the surface may cause our training to progress very slowly.

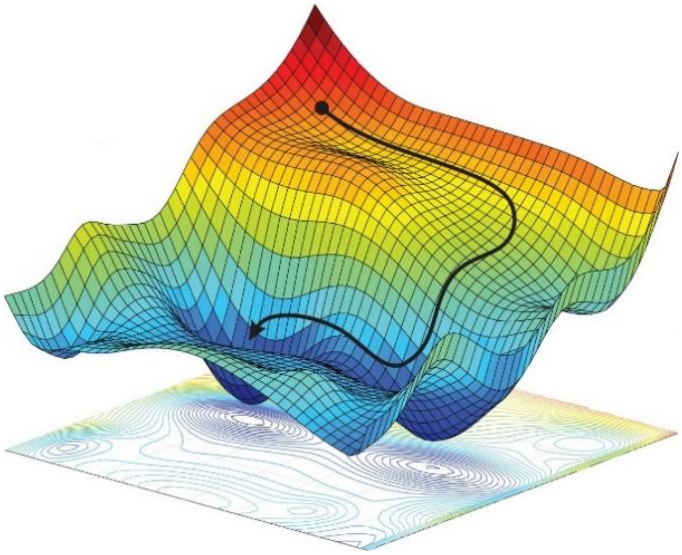


Figure 2.38. Gradient Descent algorithm in an error surface moving towards the global minimum.

In an analogy, in Figure 2.38 the dot represents the optimizer algorithm, and the path taken down the hills represents the sequence of parameter settings that the algorithm explored. The hills represent the error surface, wherewith red being the largest error and blue the smallest. The steepness of the hill represents the slope of the error surface at that point. To measure the steepness we use differentiation, by taking the derivative of the squared error function at that point. The direction the algorithm chooses to travel aligns

with the gradient, which is a vector whose components are the partial derivatives of the error surface at that point. The step size is the amount of time the algorithm travels before taking another measurement.

2.4.2.1 Mathematics behind GD

Consider f a differentiable function which we want to find the minimum. In the case of NN, the differentiable function is the loss function (Section 2.3.3). We differentiate the function with respect to the network weights x, we obtain a vector of partial derivatives or a gradient vector:

$$\nabla f(x) = \langle \frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta x_2}, \dots, \frac{\delta f}{\delta x_n} \rangle.$$

Since we want to minimize the function and the gradient always pointing the direction of the maximum growth, we take the negative of the gradient ($-\nabla f(x)$).

2.4.2.2 GD Algorithm

1. Initialize randomly the weights x_0.
2. Calculate the gradient $\nabla f(x_i)$.
3. Update the parameters in the direction of the negative gradient: $x_{i+1} = x_i - a\nabla f(x_i)$, where a is the learning rate.
4. Repeat from step 2 until the gradient is close to zero.

2.4.3 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD), selects a few samples randomly, or so-called batches, instead of the whole dataset for each iteration as Gradient Descent (Section 2.4.2) does. Batches are useful when the dataset is large, but using the whole dataset is preferable for getting to the minima less noisy or randomly (Robbins and Monro, 1951).

In SGD usually, momentum is being used (Rumelhard et al. 1986). Momentum helps to increase the speed of training since the momentum will produce large changes in the weights if the changes are currently large, and will decrease as the changes become less. The method remembers the update Δw of the weights at each iteration and determines the next update as a linear combination of the gradient and the previous update.

More specifically, the term $\alpha \Delta w$ is added also when updating a weight, where $0 < \alpha < 1$ is the momentum term and $\Delta w = w(t) - w(t-1)$, the difference between the weight's current value and in the previous iteration.

2.4.4 Adaptive Moment Estimation (Adam)

Adaptive Moment Estimation (Adam) is a stochastic gradient-based optimizer that was proposed by Kingma and Ba, (2014). Adam combines ideas from both RMSprop (Hinton et al. 2012) and AdaGrad (Duchi et al. 2011). The algorithm is computationally efficient and has few memory requirements (Kingma and Ba, 2014). The optimizer computes adaptive learning rates for each parameter and uses momentum as well. In contrast, the SGD keeps a maintained learning rate for all the weight updates and does not change during training.

From the AdaGrad, Adam takes the feature of maintaining a per-parameter learning rate, that improves the performance with sparse gradients (e.g., computer vision problems). Similarly, RMSprop maintains per-parameter learning rates, which are also adapted based on the average of recent magnitudes of the gradient for the weight. This means that it controls how quickly the gradient is changing, and therefore the algorithm does well on online and noisy problems.

Adam, instead of adapting the parameter learning rates based on the average first moment (mean) as in RMSprop, makes use of the average of the second moments of the gradients (uncentered variance) as well. More specifically, Adam calculates an exponential moving average of the gradient and squared gradient. The parameters beta1 (β1) and beta2 (β2) control the decay rates of these moving averages. The initial values of the two betas are usually to be close to 1 (beta1 = 0.9, beta2 = 0.999), which results in a bias of moment estimates towards zero.

Momentum was already explained in Section 2.4.3, so now we are going to see what the adaptive learning rate is. The adjustment to the learning rate in the training phase by reducing the learning rate to a pre-defined schedule is called the adaptive learning rate. This means that unlike in SGD where the learning rate is defined initially and remains stable throughout the training process, in Adam the learning rate changes during training. Figure 2.39 is the Adam optimizer algorithm pseudocode as defined in Kingma and Ba's (2014) study.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 2.39. Adam algorithm for stochastic optimization (Kingma and Ba, 2014)

Kingma and Ba (2014) have shown in their paper that Adam performs better than AdaGrad, RMSProp, SGDNesterov, and AdaDelta, other frequently used optimizers, for the MNIST dataset. Also, they have shown that Adam with and without Dropout, outperforms AdaGrad and SGDNesterov in the CIFAR-10 dataset by using CNNs (Figure 2.40). This shows that the Adap optimizer is potentially more efficient than the other aforementioned optimizers in the deep Convolutional neural networks and large datasets. Therefore, in this thesis, the Adam optimizer is going to be used for performance comparison with the HFO optimizer (Section 2.4.7).

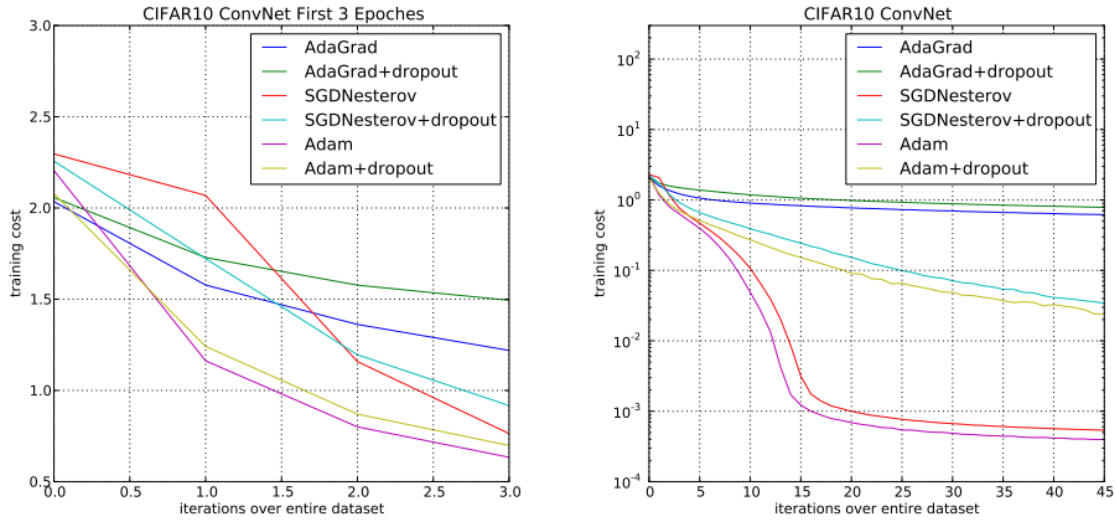


Figure 2.40. Convolutional neural networks training loss with CIFAR-10 dataset.
Left: Training costs for the first three epochs. **Right:** Training costs over 45 epochs. (Kingma and Ba, 2014)

2.4.5 Newton's Method

To encounter the problem of the Gradient Descent which is slow in multi-dimensional error surfaces, we can use the second derivative of a function. The most basic method of a second-order method for minimization is Newton's Method.

2.4.5.1 One Dimension

For some function $f: \mathbb{R} \to \mathbb{R}$, we want to find the zero of its derivatives, as we know that $f'(x) = 0$ at the minimum of $f(x)$. We can approximate then f with a second-order Taylor expansion about some point x_o:

$$f(x_o + x) \approx f(x_o) + f'(x_o)x \ + \ f''(x_o)\frac{x^2}{2}.$$

Then, we would like to choose a x, so that the $f(x_o + x)$ is a minimum. To do that, we take the derivative of this expansion with respect to x and we set it to zero having the following equations:

$$\frac{d}{dx}\left((x_o + x) \approx f(x_o) + f'(x_o)x \ + \ f''(x_o)\frac{x^2}{2}\right) = \ f'(x_o) + \ f''(x_o)x = 0 \Longrightarrow x = \frac{f'(x_o)}{f''(x_o)}$$

Consider f a quadratic function, the aforementioned equation would the absolute minimum. As we would like f to be any non-linear function, to get the minimum we need to repeat this process, starting from a guess x_o, and then get closer and closer with the update rule:

$$x_{n+1} = x_n - \frac{f'(x_o)}{f''(x_o)} \ = \ x_n - (f''(x_n))^{-1}f'(x_n).$$

2.4.5.2 Multiple Dimensions

The algorithm above works only for a single dimension, so, in the case that our function is $f: \mathbb{R}^n \to \mathbb{R}$, we can do the same derivation but replace the derivatives and the second derivatives with Gradients (Section 2.2.6) and Hessians (matrix of second derivatives) (Section 2.2.7) respectively:

$$f'(x) \ \to \ \nabla f(x)$$

$$f''(x) \to H(f)(x).$$

Thus, the new update rule, which commonly referred to as Newton's method is:

$$x_{n+1} = x_n - (H(f)(x_n))^{-1}\nabla f(x_n).$$

2.4.5.3 Problems with Newton's Method

As Newton's Method is a second-order algorithm we may assume that theoretically performs better than the simpler Gradient Descent (Section 2.4.2). If the function is quadratic and its second-order expansion is a good approximation, we can assume that a single step advances towards the global minimum instead of just a minimum. This allows it to make big steps in high-curvature scenarios and small steps in low-curvature scenarios (where $f''(x_n)$ is small). Where in Gradient Descent the direction we move is towards the negative of the gradient ($-\nabla f(x_n)$).

The huge drawback of Newton's Method is that it needs to calculate the Hessian matrix H (Section 2.2.7). In a neural network, backpropagation can be used to compute the Gradient (Section 2.2.6) but not the Hessian. This means that we need a completely different algorithm which makes the whole procedure more complicated. Moreover, for n dimensions, since the Hessian is an $n \times n$, requires $O(n^2)$ storage space and computation to find it.

The Hessian-Free Optimization (HFO) addresses both those issues. We use the insights from Newton's method but come up with a better way to minimize the quadratic function. Given our function f, we approximate it with a second-order Taylor expansion:

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^T \Delta x + \Delta x^T H(f) \Delta x.$$

To find the best Δx, we find the minimum of this approximation by moving to $x + \Delta x$ and we iterate until we have convergence. To find the minimum of this quadratic function, we will use an iterative method called Conjugate Gradient (CG).

2.4.6 Conjugate Gradient (CG)

Gradient Descent (GD), first evaluates the negative gradient of the cost function $d = -\nabla f(x_i)$, then performs a line search in the direction of d with a step size a (learning rate) and updates the parameters using $x_{i+1} = x_i + a\nabla f(x_i)$. The Conjugate Gradient (CG) is one of many speed-up methods to the basic GC algorithm. CG is a method for minimizing a quadratic function, in which any new direction that is being selected for navigating towards the minimum, has to be conjugated to all previous directions, in order

not to cancel out previous work. An advantage of CG is that it guarantees convergence in n steps, where n are the parameters of the quadratic function we are trying to minimize. As illustrated in Figure 2.41, the GD's updates are orthogonal, whereas CG finds a much better direction towards the minimum.

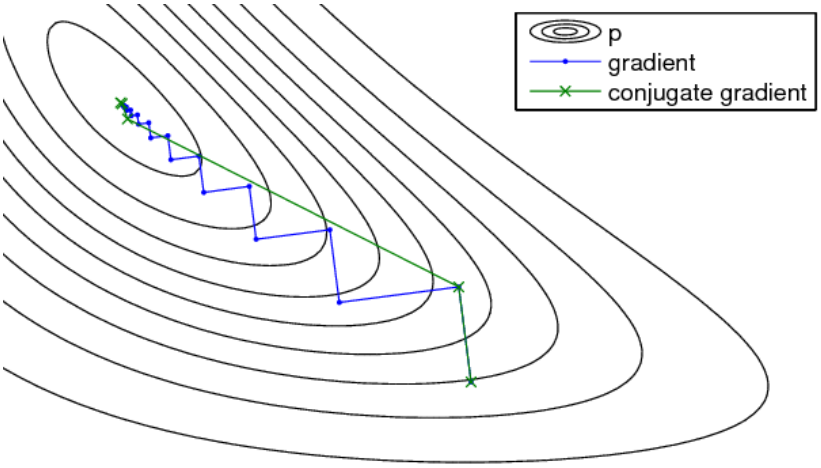


Figure 2.41. Gradient Descent (Blue Line) and Conjugate Gradient (Green Line) converge into a minimum. Note that the first steps are the same, but the following updates of the conjugate gradient are much better. (Kuusela et al. 2009)

2.4.6.1 Mathematics behind CG

Suppose we have $x \in \mathbb{R}^n,\ A \in \mathbb{R}^{n\times n},\ b, c \in \mathbb{R}^n$ and the quadratic function:

$$f(x) = \frac{1}{2}x^T Ax + b^T x + c.$$

We can write any quadratic function in this form, as this generates all the coefficients $x_i x_j$, linear and constant terms. We assume that $A = A^T$, since A is symmetric and from the gradient of f we obtain:

$$\nabla f(x) = Ax + b.$$

Now, in any given location, $-\nabla f$ gives us the direction of the steepest descent. In GD at this point, you use a learning rate α to move in a direction. In CD we do not have a fixed step size α, instead, we perform a line search to find the best α. By computing α is equivalent to minimizing the function:

$$g(a) = f(x_0 + \alpha d_0).$$

Since $g(a)$ is a quadratic function in α, it has a unique global minimum or maximum. We assume that it has a minimum (we can assure that by using the Gauss-Newton Matrix

(Section 2.4.9.2)) since we are not at a saddle point or the minimum of f. For $g'(a) = 0$ we get the minimum, that is when:

$$g'(a) = (d_i^T A d_i)a + d_i^T(Ax_i + b) = 0 \implies a = -\frac{d_i^T(Ax_i + b)}{d_i^T A d_i}$$

These steps give us our second point of the iterative algorithm:

$$x_1 = x_0 - \alpha \nabla f(x_0).$$

At this point, with GD we would iterate this procedure, computing the gradient at each next point moving in that direction. However, there is a problem; by moving a_0 indirection d_0 (to find the minimum in the direction d_0) and then moving a_1 indirection d_1, a chance exists of ruining our work from the previous iteration, and no longer be at a minimum in the direction d_0. Thus, the directions need to be conjugated to one another, to rectify this. To achieve this, we need a scalar value β, that helps find the next direction d_1 which is going to be conjugate to the first direction d_0.

For some semi-definite matrix, A, we define two vectors x and y, if $x^T A y = 0$. Since we have already moved in the $d_0 = -\nabla f(x_0)$ direction, we must find a d_1 conjugate to d_0. To do this, by starting with the gradient x_1, we compute:

$$d_1 = -\nabla f(x_1) + \beta_0 d_0.$$

From the definition of conjugacy, since d_0 and d_1 must be conjugate, we know that $d_1^T A d_0 = 0$, thus, we can derive β_0:

$$\beta_0 = \frac{\nabla f(x_1)^T A d_0}{d_0^T A d_0}.$$

2.4.6.2 CG Algorithm

Let f a quadratic function $f(x) = \frac{1}{2}x^T A x + b^T x + c$ which we wish to minimize.

1. Initialize randomly the weights x_0 and compute $d_0 = -\nabla f(x_0)$.
2. Find the best step size α to minimize the function $f(x_i + \alpha d_i)$ by using the equation $a = -\frac{d_i^T(Ax_i + b)}{d_i^T A d_i}$.
3. Update the current guess: $x_{i+1} = x_i - a d_i$

4. Update the direction: $d_{i+1} = -\nabla f(x_{i+1}) + \beta_i d_i$ where $\beta_i = \frac{\nabla f(x_{i+1})^T A d_i}{d_i^T A d_i}$
5. Repeat from step 2 until we have looked in n directions, where n is the dimension of x.

2.4.7 Hessian-Free Optimization (HFO)

2.4.7.1 Introduction to HFO

The background of the methodology of this thesis is based on Martens (2010) and Martens and Sutskever (2012), about using deep learning with the Hessian-Free Optimization (HFO) algorithm. Martens (2010), used the HFO algorithm to train deep auto-encoders. As he described, on backpropagation-based algorithms, Gradient Descent (GD) (Section 2.4.2) is being used to learn the weights of a network with multiple layers of non-linear hidden units. The problem with GD is that it does not seem to generalize well in networks with many hidden layers, such as deep neural networks, as it progresses extremely slow, resulting in poor performance on the training set (under-fitting). Besides, GD is unsuitable for optimizing objectives that exhibit pathological curvature. Thus, second-order optimization methods, such as HFO, have proven to be more effective on such objectives as they model the local curvature and correct it.

Figure 2.42 demonstrates pathological curvature, where the objective function has the form of a long narrow valley. The red arrow indicates the direction that needs to be followed to have some progress. In Figure 2.42, on the left, the black arrows indicate the steps the Gradient Descent takes, while on the right are the steps of Newton's method.

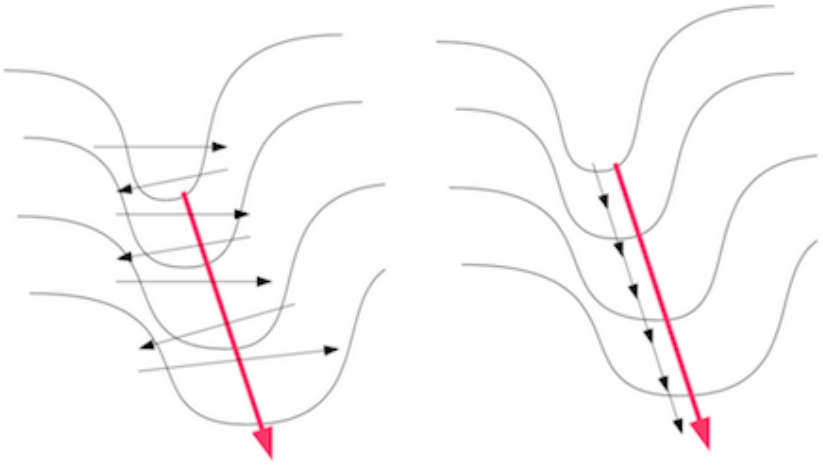


Figure 2.42. Pathological curvature in a long narrow valley. Steps of Gradient Descent (Left Black Arrows), steps of Newton's method (Right Black Arrows), and the direction towards the minimum (Red Arrow). (Martens, 2010)

2.4.7.2 HFO Algorithm (Martens, 2010)

Let f be any function $f: \mathbb{R}^n \to \mathbb{R}$ we are trying to minimize.

1. Initialize randomly the weights x_0 from a normal distribution with zero mean and a relatively small standard deviation.
2. Compute the gradient $\nabla f(x_n)$ and Hessian $H(f)(x_n)$ for the current x_n and consider the following Taylor expansion of f:
   $f(x + \Delta x) \approx f(x) + \nabla f(x)^T \Delta x + \Delta x^T H(f) \Delta x.$
3. Compute x_{n+1} using the Conjugate Gradient (CG) algorithm for the quadratic functions on the Taylor expansion.
4. Repeat from step 2 until x_n converges. (we do not have to necessarily repeat this process for n for the CG to terminate since in practice the CG makes large improvements in the first few iterations)

In the HFO algorithm, it seems that we have to compute the Hessian matrix, which would be a huge disadvantage since it would require $O(n^2)$ storage space, while n is the number of network parameters. In reality, to compute the x_{n+1} the vector we do not need the Hessian H, as we can simply use the CG (Section 2.4.6). Since on our algorithm, we have to compute Hv for some vector $v = x_n$, we can just approximate the $(Hx_n)_i$ using the following equation for some small ε:

$$Hx_n \approx \frac{\nabla \mathrm{f}(\mathrm{e} + \varepsilon x_n) - \nabla f(e)}{\varepsilon}$$

More details on how to do so, we can find in Section 2.4.8.

2.4.8 Hessian-Vector Product (Hv) (Pearlmutter, 1994)

In the Hessian-Free Optimization, as previously mentioned we need to compute the Hessian (matrix of second derivatives) times some vector. Unfortunately, computing the Hessian is incredibly computationally intense, and we cannot use finite differences since it can be numerically unstable. This is our motivation for finding a better way to compute the Hessian-Vector product Hv (Pearlmutter, 1994).

Since a forward and backward propagation algorithm for computing the Gradient is required, we can use those two algorithms to derive a Hessian-Vector product algorithm, which this method is being known as the $R\{\cdot\}$ method.

2.4.8.1 The $R\{\cdot\}$ Method

Let's assume the Hessian of the error function is defined as H. We know that the Hessian-Vector product is a directional derivative:

$$Hv = \lim_{\varepsilon \to 0} \frac{\nabla E(x + \varepsilon v) - \nabla E(x)}{\varepsilon} = \frac{\partial}{\partial \varepsilon} \nabla E(x + \varepsilon v) \Big|_{\varepsilon = 0}$$

We can define now an operator R_v which converts the gradient computation into a Hessian-Vector product:

$$R_v\{f(x)\} = \frac{\partial}{\partial \varepsilon} f(x + \varepsilon v) \Big|_{\varepsilon = 0}$$

Therefore, after a trivial substitution:

$$Hv = R_v\{\nabla E(x)\}$$

We need now to compute the Hessian-Vector product. To do so we will use a forward and backward propagation algorithm with the $R\{\cdot\}$ method.

2.4.8.2 Hessian-Vector product forward propagation algorithm

Forward propagation algorithm for the Hessian-Vector product using the $R_v\{\cdot\}$ operator.

1. Initialize $R_v\{x_i^0\}$. These are constants (input layer); therefore, they will be zero.
2. Compute for the current layer $R_v\{y_i^l\} = \sigma'(x_i^l) R_v\{x_i^l\}$
3. Compute for the next layer $R_x\{x_i^l\} = \sum_j (w_{ji}^{l-1} R_v\{y_j^{l-1}\} + v_{ji}^{l-1} y_j^{l-1})$
4. Repeat steps 2 and 3 until the output layer is reached.

2.4.8.3 Hessian-Vector product backward propagation algorithm

Backward propagation algorithm for the Hessian-Vector product using the $R_v\{\cdot\}$ operator.

1. Initialize at the output layer $R_v\left\{\frac{\partial E}{\partial y_i^L}\right\} = e'_i(y_i^L) R_v\{y_i^L\}$
2. Compute for the current layer $R_v\left\{\frac{\partial E}{\partial x_i^l}\right\} = \sigma'(x_j^l) R_v\{x_j^l\} \frac{\partial E}{\partial y_j^l} + \sigma'(x_j^l) R_v\left\{\frac{\partial E}{\partial y_i^l}\right\}$
3. Compute for the previous layer $R_v\left\{\frac{\partial E}{\partial y_i^l}\right\} = \sum (v_{ij}^l \frac{\partial E}{\partial x_j^{l+1}} + w_{ij}^l R_v\left\{\frac{\partial E}{\partial x_j^{l+1}}\right\})$
4. Repeat steps 2 and 3 until the input layer is reached

5. Compute the components of the Hessian-Vector product using $R_v\left\{\frac{\partial E}{\partial w_{ij}^l}\right\} = R_v\{y_i^l\}\frac{\partial E}{\partial x_j^{l+1}} + y_j^l R_v\left\{\frac{\partial E}{\partial x_j^{l+1}}\right\}$

2.4.8.4 Conclusion

Using Pearlmutter's (1994) method, we were able to develop an adjoint algorithm pair that in a forward and backward pass, computes the Hessian-Vector product Hv for any vector v, without ever computing the Hessian itself.

2.4.9 Hessian-Free Optimization with the Gauss-Newton Matrix (Martens, 2010)

2.4.9.1 Problems with the Hessian

The Hessian-Free Optimization (HFO) uses the quadratic Conjugate Gradient (CG) algorithm at every iteration of the Newton-style method (Newton-CG method). The problem with the Hessian Newton-CG is that we have assumed that the quadratic minimization objective $f(x) = \frac{1}{2}x^T Ax + b^T x + c$ has a minimum. This happens only if $f(x)$ is bounded from below. For this to be true, the matrix A has to not have any negative eigenvalues. Thus, we have to enforce A to be positive semi-definite, by having all its eigenvalues greater or equal to zero.

2.4.9.2 Gauss-Newton Matrix

To solve the problem of the Hessian Newton-CG method, we approximate the Hessian using the Gauss-Newton matrix. The Gauss-Newton matrix is a good approximation of the Hessian since its quadratic optimization objectives have the same minimum, and it is provably positive semi-definite. This means that the algorithm will always decrease the objective, and cannot take steps uphill, away from the minimum, while the Hessian could. Let $E(x)$ be the error function we are trying to minimize using its Gradient $(\nabla \mathrm{E})_i$, with:

$$H_{ij} = \frac{\partial}{\partial x_j}(\nabla \mathrm{E})_i$$

being an element of the Hessian. We can write $E(x)$ as the composition of two functions: $E(x) = \sigma(f(x))$. The function $f(x)$ computes the pre-nonlinearity output of neural network's final layer (e.g., the entire output layer) and the function $\sigma(x)$ computes the

non-linearity itself and the error at the final layer (e.g., SoftMax layer, $S(x)_i = \frac{e^{x_i}}{\sum_{k=0}^{n} e^{x_k}}$).

Thus:

$$H_{ij} = \frac{\partial}{\partial x_j}\frac{\partial \sigma(f(x))}{\partial x_i}$$

and by applying the chain rule, the product rule, and once again the chain rule, we get the final expanded expression for an element of the Hessian which is:

$$H_{ij} = \sum_{k=0}^{n}\sum_{l=0}^{n}\frac{\partial \sigma^2}{\partial f_l(x) f_k(x)}(f(x))\frac{\partial f_l(x)}{\partial x_j}\frac{\partial f_k(x)}{\partial x_i} + \sum_{k=0}^{n}\frac{\partial \sigma}{\partial f_k(x)}(f(x))\frac{\partial f_k(x)^2}{\partial x_j \partial x_i}$$

First derivatives are zero at the minimum of $\sigma(x)$, so, the summations are negligible. Thus, near the minimum the Hessian can be approximated as:

$$H_{ij} \approx \sum_{k=0}^{n}\sum_{l=0}^{n}\frac{\partial \sigma^2}{\partial f_l(x) f_k(x)}\big(f(x)\big)\frac{\partial f_l(x)}{\partial x_j}\frac{\partial f_k(x)}{\partial x_i} = G_{ij}$$

which is being called the Gauss-Newton matrix, denoted as G. By changing the order of the summation in this way:

$$G_{ij} = \sum_{l=0}^{n}\frac{\partial f_l(x)}{\partial x_j}\sum_{k=0}^{n}\frac{\partial f_k(x)}{\partial x_i}\frac{\partial \sigma^2}{\partial f_l(x) f_k(x)}\big(f(x)\big)$$

the equation can be rewritten as a matrix product, with the first-order derivatives coming from the Jacobian J_f and the second derivatives coming from the Hessian, where H_σ is the Hessian of σ:

$$G = J_f^T H_\sigma J_f$$

(In the traditional Gauss-Newton matrix in place of H_σ there is a two (2) because the nonlinearity is simply squaring with the Hessian being twice the identity matrix)

2.4.9.3 Compute the Gauss-Newton Matrix-Vector Product

When we compute G the Gauss-Newton matrix, we usually need to compute the Gauss-Newton matrix multiplied by some vector, $Gv = J^T H_\sigma J v$, so first, we compute the vector Jv and then the vector $J^T v'$, where $v' = H_\sigma J v$.

2.4.9.4 Forward Pass

As mentioned before, to compute the Gauss-Newton Matrix-Vector product, $Gv = J^T H_\sigma J v$ we will compute first the vector Jv. Recall the operator $R_v\{\cdot\}$ as we have defined it previously (Section 2.4.8.1):

$$R_x\{f(x)\} = \frac{\partial}{\partial \varepsilon} f(x + \varepsilon v)\Big|_{\varepsilon = 0} = \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon v) - f(x)}{\varepsilon}$$

We can apply this operator to a vector-valued function $f: \mathbb{R}^n \to \mathbb{R}^m$. The resulting function is the same as if he had applied to each component separately the $R_v\{\cdot\}$. Therefore, the i^{th} component can be written as the directional derivative of f_i in the direction v:

$$R_x\{f(x)\}_i = \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon v)_i - f(x)_i}{\varepsilon}$$

Since it is just a directional derivative, we can rewrite it as:

$$R_v\{f(x)\}_i = \nabla f(x)_i \cdot v = \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j} v_j$$

Mind that this is just the i^{th} element. By writing it in the sum form it looks exactly like a single row of the Jacobian. Considering Jacobian, the matrix where each row consists of the derivatives of one component with respect to all the variables. Therefore, we can merge all the components by simply multiplying the Jacobian by a vector, where J is the Jacobian of $f(x)$ evaluated by x:

$$Jv = R_v\{f(x)\}$$

Thus, we can compute Jv for any vector v by performing a forward pass of the Hessian-Vector product algorithm.

2.4.9.5 Multiplication by Hessian H_σ

Using the forward pass algorithm of the $R_v\{\cdot\}$ operator, we get the Jv, so now the next step should be to compute the $J^T v'$, which is needed for computing the Gauss-Newton matrix. Before doing that though we need to compute $v' = H_\sigma J v$. Fortunately, we have an algorithm already for multiplication by the Hessian, the backward pass algorithms of the $R_v\{\cdot\}$ operator (Section 2.4.8.3).

While we are running the backpropagation algorithm of the $R_v\{\cdot\}$ operator, if we stop it at the right time, we will have $H_\sigma Jv$.

2.4.9.6 Backward Pass

Since we have computed $v' = H_\sigma Jv$, from the previous step, we need to compute $J^T v'$ now. The backpropagation computes the gradient of the error function, where the i^{th} component of the gradient is:

$$(\nabla E)_i = \frac{\partial E}{\partial x_i}$$

By using the chain rule, and all the outputs of the last layer, where f_k is the k^{th} output unit value, the Gradient is:

$$(\nabla E)_i = \sum_{k=0}^{n} \frac{\partial E}{\partial f_k} \frac{\partial f_k}{\partial x_i}$$

If our error function, is a simple Sum of Squared Errors (SSE), then the dependence on every f_k in the output will be linear:

$$E = \sum {f_k}^2 \Longrightarrow (\nabla E)_i = 2 \sum_{k=0}^{n} f_k \frac{\partial f_k}{\partial x_i} = 2f(x) \cdot \frac{\partial f(x)}{\partial x_i}$$

As we can see, we can rewrite the i^{th} component of the gradient as the dot product of the function with its own derivative with respect to x_i. This is the same as the multiplication by the transpose of the Jacobian.

$$(J^T f(x))_i = \left(\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & & \frac{\partial f_n}{\partial x_2} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_m} & \frac{\partial f_2}{\partial x_m} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \right)_i = \sum_{k=0}^{n} f_k \frac{\partial f_k}{\partial x_i}$$

Thus, the backpropagation for the SSE error function computes a multiple by the transpose of the Jacobian:

$$\nabla E = 2J^T f(x)$$

Therefore, if we give backpropagation any vector v at the output units, it simply multiplies by J^T. Consequently, by performing the standard neural network backpropagation, we can compute $J^T v$ for any vector v, while using the components of v as last's layer "errors". To complete the Gauss-Newton matric computation, we need to backpropagate $H_\sigma J v$ through until the input layer, using the normal neural network backpropagation algorithm.

2.4.9.7 Conclusion

Conjugate Gradient goes towards the extremum of the function, which means we have to have a positive semi-definite matrix. In the scenario where the matrix is not positive semi-definite, it means that the Hessian-Free Optimization algorithm will fail to decrease the objective, as well as taking steps uphill away from the minimum.

Since the Hessian is not guaranteed to be positive semidefinite, this assumption turns to be problematic. Therefore, we derived the Gauss-Newton matrix and approximation to the Hessian. The fact that quadratic optimization objectives using the Gauss-Newton matrix instead of the Hessian have the same minimum, and that is provably positive semidefinite, makes the Gauss-Newton matrix a good approximation.

Instead of computing the Gauss-Newton matrix G, we can compute Gv, for any vector v by just running the forward propagation of the $R_v\{\cdot\}$ operator, to get the vector $a = Jv$ and one step of the backward propagation of $R_v\{\cdot\}$ to multiply it by H_σ. To finish computing Gv, we run a normal neural network backpropagation.

As Martens (2010) observed in his paper, this modification is consistently superior to using the Hessian alone, since it fixes the issue that the Hessian is not positive semi-definite, as well as being significantly faster than computing the full Hessian-Vector product.

Chapter 3

Data Handling

3.1 Cross-validation

3.1.1 Introduction to Cross-validation

Using the same data for training and testing in machine learning is a methodological mistake. This would imply a perfect score during training but would fail to predict anything useful on yet-unseen data. To encounter this issue, which is so-called overfitting, we hold out part of the dataset as a test set (values & labels).

We can easily split the data randomly into training and test sets by using Python's library scikit-learn. The Code Snippet 3.1, splits the dataset (X, y) into training (X_train, y_train) and test (X_test, y_test) sets, with 30% (test_size = 0.3) of the data being randomly assigned to the test set.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Code Snippet 3.1. Scikit-learn's train_test_split method for splitting dataset randomly to training and test set.

We could train a model using just the training and test sets that have been produced previously. However, there is a risk of overfitting in the test set while modifying the hyperparameters for the estimators to perform optimally. This way, the knowledge about the test set leaks into the model and the generalization performance decreases. Consequently, the validation set comes to the rescue. The validation set is used during the training process, usually at the end of each epoch to validate the performance of the

network and decide whether it generalizes well or overfits. After a successful training process, the test set is being used for the final evaluation.

Having millions of data, splitting them into three sets (training, test, and validation sets) would not be an issue. Unfortunately, we have limited labeled data for training, especially in our case, where we need medical MRI scans of patients. Therefore, splitting them this way drastically reduces the number of samples for the learning process and may consequently cause overfitting since the generalization error would be high.

To avoid this problem, we can use the Cross-validation (CV) procedure. In CV, we still hold out a test set, but the validation set is no longer needed. This is possible because of a method called k-fold CV, where the training set is being split into k smaller sets, usually having the same size (Figure 3.1). The k-fold CV learning method goes as follows:

1. The model is trained using the k – 1 of the folds of the data.
2. The model is validated on the remaining fold to compute the accuracy, sensitivity, specificity, etc.
3. We repeat the same process k times until all the folds have been used as a validation set, with each time having a new model.
4. The average of the k metrics, for each loop, is the general performance measure.

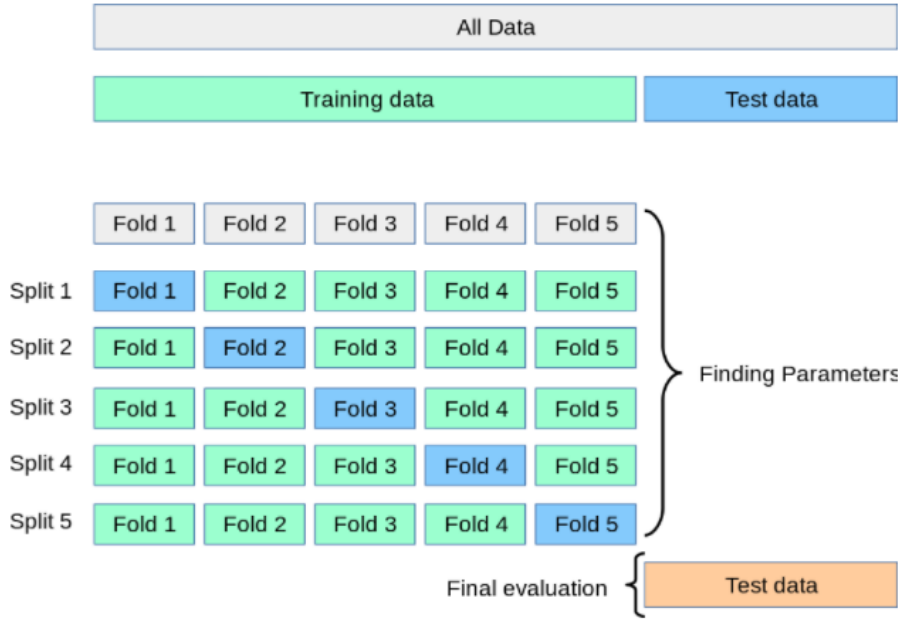


Figure 3.1. Example of 5-fold Cross-Validation split

This approach is computationally expensive since for each set of hyperparameters you have to train the network k times. On the other hand, we do not waste data, which is a major advantage, especially in this thesis where the number of available samples was extremely small and we could not obtain easily more data.

3.1.2 KFold

With the KFold splitting method (Code Snippet 3.2), all the samples as being divided into k groups (folds), of equal sizes if it is possible. In the case of k = n, where n is the size of the dataset, then this is equivalent to the Leave One Out Strategy. The k-1 folds are used for the training, and the fold left out is used for the validation.

```
from sklearn.model_selection import KFold

kf = KFold(n_splits=2)                                    # 2-Fold

for train, valid in kf.split(X):
     print("%s %s" % (train, valid))
```

Code Snippet 3.2. Scikit-learn, KFold Cross-validation method for splitting the dataset into k-folds.

The main issue of the KFold is that it is not affected by classes or groups. These issues can be solved by using StratifiedKFold (Section 3.1.3) and GroupKFold (Section 3.1.4) respectively, or StartifiedGroupKFold for both of them. In Figure 3.2 there is a visualization of a KFold, more specifically a 4-fold based on an unbalanced dataset with 3 different classes and a total size of 100 samples. We can see that each testing set does not contain an equal ratio of samples of each class when splitting with the standard KFold.

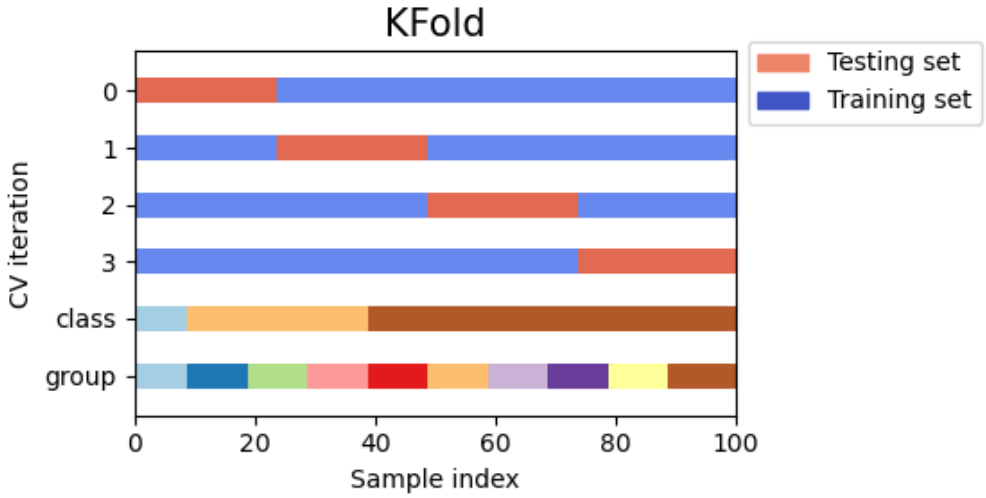


Figure 3.2. Visualization of KFold.100 samples, 3 unbalanced classes, 4-folds, 10 groups

3.1.3 StratifiedKFold

We are going to examine a case that shows why KFold (Section 3.1.2) needs to take into consideration the classes and therefore there is a need for StratifiedKFold. First of all, the classes represent the labels of our data (e.g., AD, NC). So, a 3-fold CV split with the KFold method and initial dataset of 50 MRI scans, with 45 NCs and 5 ADs could be:

1. Train: [AD: 28, NC: 5] ||| Valid: [AD: 17, NC: 0]
2. Train: [AD: 28, NC: 5] ||| Valid: [AD: 17, NC: 0]
3. Train: [AD: 34, NC: 0] ||| Valid: [AD: 11, NC: 4]

This will cause problems such as learning better one of the two classes, and false validation accuracy due to the fact that the validation sets are not balanced. Thankfully, scikit-learn provides the method StratifiedKFold to solve this issue.

The way StratifiedKFold works is by returning stratified folds, where each set contains the approximately same percentage of samples of each target class, given as input a dataset. So, in the 3-fold CV on the previous dataset with 50 samples from two unbalanced classes, we can see that the StratifiedKFold preserves the class ratio to each set to approximately $\frac{1}{10}$:

1. Train: [AD: 30, NC: 3] ||| Valid: [AD: 15, NC: 2]
2. Train: [AD: 30, NC: 3] ||| Valid: [AD: 15, NC: 2]
3. Train: [AD: 30, NC: 4] ||| Valid: [AD: 15, NC: 1]

Code Snippet 3.3 is an example of StratifiedKFold in comparison with KFold where the initial dataset is defined as (X, Y):

```
from sklearn.model_selection import StratifiedKFold, KFold
import numpy as np

################################## StratifiedKFold ##############################

skf = StratifiedKFold(n_splits=3)                                          # 3-Fold

for train, valid in skf.split(X, y):
   print('train {} valid {}'.format(np.bincount(y[train]), np.bincount(y[valid])))

######### Output ##########

# train [30  3] valid [15  2]
# train [30  3] valid [15  2]
# train [30  4] valid [15  1]

##################################### KFold #####################################
```

```
kf = KFold(n_splits=3)                                                    # 3-Fold

for train, valid in kf.split(X, y):
    print('train {} valid {}'.format(np.bincount(y[train]), np.bincount(y[valid])))

######### Output ##########

# train [28  5] valid [17]
# train [28  5] valid [17]
# train [34] valid [11  5]
```

Code Snippet 3.3. Scikit-learn, StratifiedKFold Cross-validation method for splitting the dataset into stratified k-folds. StratifiedKFold preserves the class ratio to each set vs. the KFold which does not.

In Figure 3.3 there is a visualization of a StratifiedKFold, 4-fold follows based on an unbalanced dataset with 3 different classes and a total size of 100 samples. We can see that each fold contains approximately the same ratio of samples of each class. Anyhow, this method stills ignores the groups.

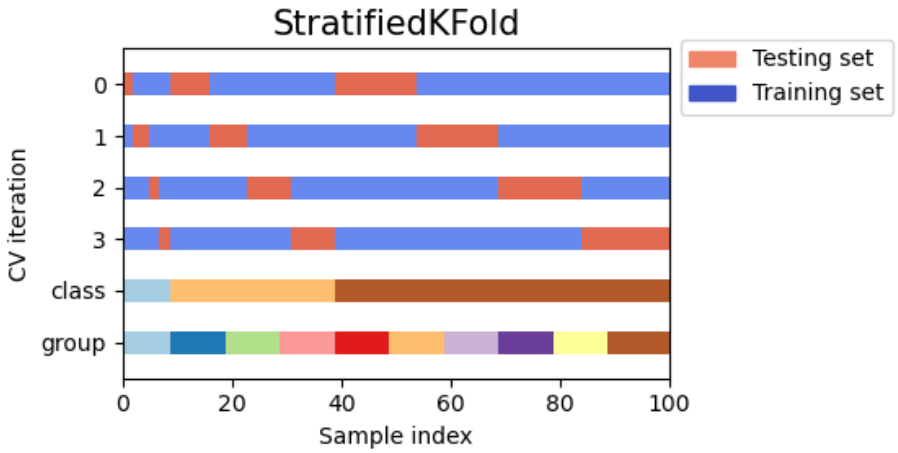


Figure 3.3. Visualization of StratifiedKFold. 100 samples, 3 unbalanced classes, 4-folds, 10 groups.

3.1.4 GroupKFold

StratifiedKFold (Section 3.1.3) solves KFolds' problem (Section 3.1.2) of not taking into consideration the classes, but it still does not take into consideration the groups. That is why we need GroupKFold. The need for groups in this thesis comes in datasets with either "Multiple Scans per Patient" or "Multiple Slices per Scan". All the MRI scans of different periods of the same patient, define a group and the group ID is the patient's ID.

As scikit-learn's documentation mentions, GroupKFold is used in medical applications to avoid data leakage. Data leakage occurs when in a dataset we have multiple MRIs of the same patient from different periods, and those MRIs are not all of them in a single set (training or validation). The patient's ID in such a case will be the group identifier which

is going to ensure that a group never gets split, therefore the same patient does not appear in more than one set. Consequently, the test should contain only unseen groups, to ensure that the model will generalize. The GroupKFold method in Code Snippet 3.4 from the scikit-learn library ensures that samples of the same group are only in a single fold.

```
from sklearn.model_selection import GroupKFold

X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 8.8, 9, 10]
y = ["a", "b", "b", "b", "c", "c", "c", "d", "d", "d"]
groups = [1, 1, 1, 2, 2, 2, 3, 3, 3, 3]

gkf = GroupKFold(n_splits=3)
for train, valid in gkf.split(X, y, groups=groups):
    print("%s %s" % (train, valid))

######### Output ##########

# [0 1 2 3 4 5] [6 7 8 9]
# [0 1 2 6 7 8 9] [3 4 5]
# [3 4 5 6 7 8 9] [0 1 2]
```

Code Snippet 3.4. Scikit-learn, GroupKFold Cross-validation method for splitting the dataset into k-folds, by using group constraints to avoid data leakage.

The issue of GroupKFold is that it takes into consideration the groups, but ignores the classes; the same issue that the KFold had. As we can see from the splitting example in Figure 3.4 by using the GroupKFold, all the samples of a group are part of a single fold, therefore part of either the validation or the training set. On the other hand, the unbalanced classes cause the splitting to have the same issues with the KFold method, where the class ratio diverges between the same sets of different folds. For example, fold-0 (Figure 3.4 first line, red sections), contains samples of both the yellow and brown classes, but no samples from the light blue class.

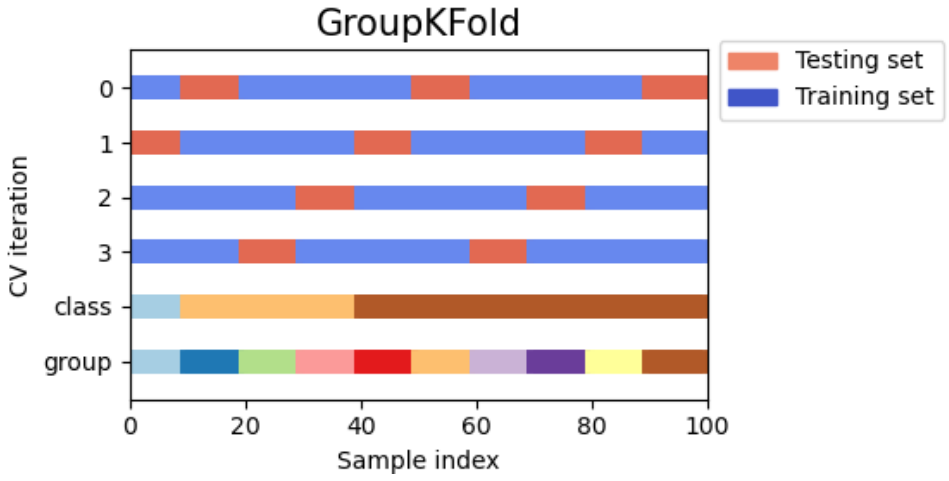


Figure 3.4. Visualization of GroupKFold.100 samples, 3 unbalanced classes, 4-folds, 10 groups

3.1.5 StratifiedGroupKFold

As described in a comment in scikit-learn's GitHub repository (https://github.com/scikit-learn/scikit-learn/issues/13621#issuecomment-557802602) by Michael Rebsamen, the need for StratifiedGroupKFold is a very common use-case in medicine and biology since frequently we have repeated measures of the same patient, which implies to same groups.

Michael gave the same example which our thesis focuses on, where we want to classify a disease such as Alzheimer's disease versus normal cognitive patients through MRI images. In such a case, the datasets could be imbalanced since we have a total of 1000 subjects, with 200 of them being diagnosed with AD.

Similar to the ADNI database, most subjects have more than a single scan available. Having samples of the same patient in different sets could cause data leakage which is a major issue. Therefore, the optimal scenario could be to use 'stratify' to solve the issue of the imbalanced set, and group constraint for a subject not to appear in more than one set at the same time.

By using Michael's implementation of StratifiedGroupKFold which was inspired by Kaggle-kernel (https://www.kaggle.com/jakubwasikowski/stratified-group-k-fold-cross-validation) (Code Snippet D.1), we were able to address both those issues in this thesis. The Code Snippet 3.5 is an example of applying the custom StratifiedGroupKFold which was proposed by Michael Rebsamen.

```
from model_selection.stratified_group_k_fold import StratifiedGroupKFold

sgkf = StratifiedGroupKFold(n_splits=10)                    # 10-folds

group = [. . .]             # A vector of Patient IDs for the corresponding samples

for train_index, valid_index in sgkf.split(X, y, groups=groups):
    print("%s %s" % (train, valid))
```

Code Snippet 3.5. StratifiedGroupKFold code example based on Michael Rebsarmen's implementation

Figure 3.5 visualizes a StratifiedKFold (Section 3.1.3) from the scikit-learn library, on 100 samples for 20 folds and 2 unbalanced target classes. The issue is that the groups are not taken into consideration and data leakage occurs.

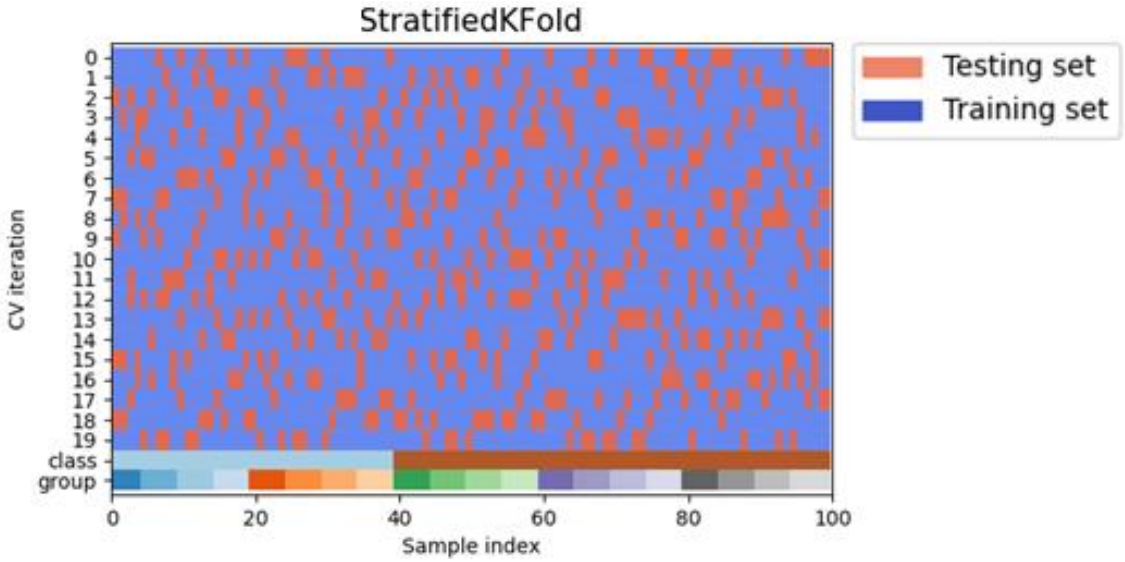


Figure 3.5. Visualization of StratifiedKFold. 100 samples, 2 unbalanced classes, 20-folds.

Figure 3.6 visualizes a StratifiedGroupKFold in the same dataset as in Figure 3.5, where solves the StartifiedKFold's issue. Each group is either on the training or validation set. Additionally, the training and validation sets have similar class ratios.

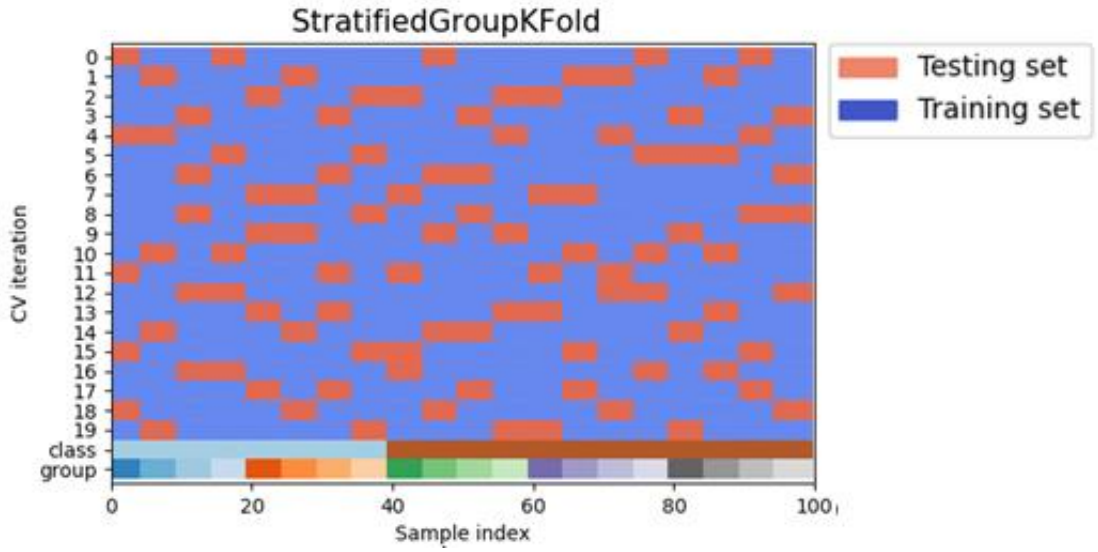


Figure 3.6. Visualization of StratifiedKFold. 100 samples, 2 unbalanced classes, 20-folds.

To conclude, in this thesis, StratifiedGroupKFold is used for splitting into either 5 or 10-folds each dataset which contains multiple scans from different periods of the same patient ("Multiple Scans per Patient" or/and "Multiple Slices per Scan"), and StratifiedKFold for the datasets that contain a single scan from each patient ("Single Scan per Patient" and not "Multiple Slices per Scan").

3.2 Datasets

3.2.1 Main Sources of Data

To develop the model for the classification of AD/NC, structural MRI scans from the Alzheimer's Disease Neuroimaging Initiative (ADNI) database have been used, as well as features extracted from those data. Specifically, the 3 main sources of data to create the datasets for this thesis are (1) T1-weighted MRI scans, (2) Hippocampus Features (Achilleos et al. 2020), and (3) 3D Hippocampus (Achilleos et al. 2020). From these 3 main sets, 9 distinct datasets were created for approximately more than 80 experiments of this thesis.

ADNI's goal is to unite researches with study data for Alzheimer's disease (AD). It includes MRIs, PET scans, genetics, cognitive tests, CSF, and blood biomarkers as a prediction for AD. ADNI includes samples of AD, MCI, and NC patients. Since all 3 main sets are based on ADNI's database, the same patients could appear in all 9 datasets that had been used in our experiments. We did not explicitly split the patient's scans in the same manner for all 9 datasets in this thesis in order for all the folds of different datasets to contain the scans of the same patients. This could potentially eliminate the factor that each dataset may contain different ratios of outliers and consequently could help to compare even more accurately the performance between the different methods and implementations.

For the creation of the main T1-weighted MRI scans dataset, the raw data were obtained from the ADNI database. The images were acquired by using the MPRAGE protocol on either 3-T or 1.5-T scanners from Siemens Medical Solutions, General Electric Healthcare, or Philips Medical Systems at the centers' participation in the ADNI project (Clifford et al. 2008)

After the data acquisition, some preprocessing was applied to create the T1-weighted MRI scans dataset, which is going to be used in sub-datasets of this thesis (Sections 3.2.3, 3.2.4, 3.2.4, and 3.2.7). The downloaded MRI scans were already corrected for gradient inhomogeneity, bias field, and intensity non-uniformity; were scaled to have voxel dimensions of 1x1x1 mm, and had undergone the FreeSurfer cross-sectional processing. These MRI scans contained the skull-stripped brain together with the brainstem. The MRI volumes had a size of $256 \times 256 \times 256$ voxels with many "blank" voxels, i.e., having

values of 0. To reduce the size of the dataset, all blank voxels surrounding the rectangular cuboid bordering the "brain and brainstem" voxels were cropped, using custom-written code in MATLAB. The resulting MRI volumes had sizes in the x, y, and z axes ranging from 125 to 174 voxels, 145 to 200 voxels, and 121 to 174 voxels, respectively.

In Figure 3.7, on the left, we can see an MRI scan of an NC patient without any preprocessing. The skull, skin, and other parts of the patient's head are visible. On the left, after preprocessing, the extra non-meaningful for us parts have been removed, keeping only the brain. This potentially helps the model to learn the important features which are located on the brain of the patient to classify them as AD or NC.

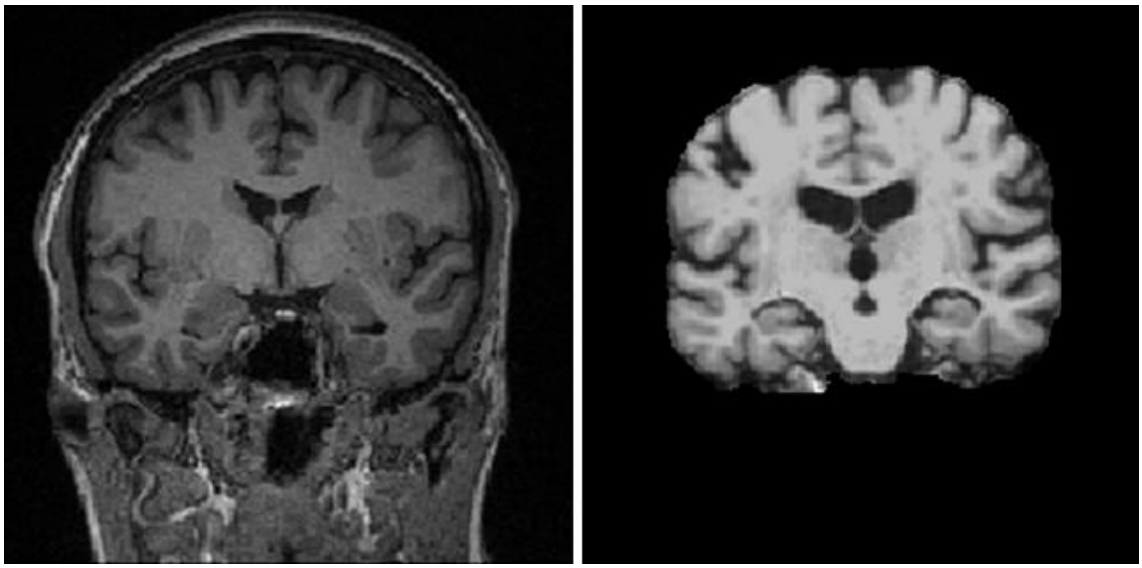


Figure 3.7. 2D Coronal slices of NC patients' MRI. **Left:** Without preprocessing**. Right:** With preprocessing

Achilleos et al. (2020) created the two datasets, Hippocampus Features and the 3D Hippocampus sets, from brain MRI images, part of the ADNI-1 Complete 2- and 3-year datasets. All subjects were on 1.5-T MRI units from Siemens Medical Solutions and General Electric Healthcare. From those datasets, they were able to isolate the 3D structure of the Hippocampus and the Entorhinal Cortex. Additionally, from the same set, 10 features from the Hippocampus had been manually extracted for each patient's scan, which we are going to use in this thesis for direct comparison between them. This dataset during this study is going to be referred to as the "Hippocampus Features".

Due to lack of time and resources, experiments were performed only on the 3D structure of the Left Hippocampus. The Right Hippocampus and the Entorhinal Cortex are not going to be used for now from Achilleos et al. (2020). The Left Hippocampus is estimated

to receive the most damage during Alzheimer's disease (Achilleos et al. 2020), thus, we decided to focus only on those images.

3.2.2 Hippocampus Features [AD, NC]

3.2.2.1 General Information of Hippocampus Features

The Hippocampus Features dataset was provided by Achilleos et al. (2020). For each MRI scan of the Hippocampus Features dataset, the 10 most promising hippocampal features were extracted: (1) Volume, (2) SumAverage, (3) Entropy, (4) ClusterShade, (5) ClusterProminence, (6) SumEntropy, (7) Variance, (8) SumVariance, (9) Contrast and (10) Angular Second Moment. Therefore, each pair in the dataset had 10 floating-point values (features) and the label (AD or NC).

3.2.2.2 Multiple Scans per Patient (HF_M)

In the dataset, Hippocampus Features - Multiple MRI Scans per Patient (HF_M), multiple scans of different periods of the same patient have been used for the feature extraction, from the baseline till 72 months. Each patient was between 55 and 90 years old. In total, 237 samples were included in the study, with two distinct groups NC = 153 (73 males and 80 females) and AD = 84 (40 males and 44 females). Outliers were not removed, however, subjects with missing values were dropped. Consequently, the final dataset consisted of NC = 144 and AD = 69, thus a total of 213 samples. The dataset is unbalanced, therefore, 30% equal distribution test set (NC = 22, AD = 22) was used and an unbalanced training set (NC = 122, AD = 47). Due to the relatively small number of records, no validation set was used, thus, the test set will play the role both of the validation and test set (Table 3.1).

	AD	***NC***	***MCI***	***Total***
All Samples	69	144	-	213
Training Set	47	122	-	169 (80%)
Test Set	22	22	-	44 (20%)
Validation Set	-	-	-	-

Table 3.1. Dataset: Hippocampus Features, Multiple Scans per Patient (HF_M) (Achilleos et al. 2020)

Based on this dataset, 10 random training and test sets were created. The original paper (Achilleos et al. 2020), does not clarify whether 10-fold Cross-validation was used and if

data leakage was taken into consideration due to multiple scans of the same subject. Data leakage occurs when some scans of the same patient are in the training set while others are in the test set. To avoid data leakage, all scans of the same patient should be in a single set; GroupKFold (Section 3.1.4) can be used to resolve this issue.

3.2.3 2D Brain Slices [AD, NC] (slice-level MRI)

3.2.3.1 Preprocessing of 2D Brain Slices

Each T1-weighted MRI from ADNI has different dimensions since it was cropped as close to the brain as possible. Therefore, padding needs to be added to all three dimensions of each scan for all of them to have the same dimensions which are $174 \times 190 \times 174$. For the 2D Brain Slices datasets, we isolate the middle coronal slice of each 3D MRI scan before adding the padding (Code Snippet 3.6).

```
# Isolate the middle slice of the y-axis from a 3D MRI image

. . .
index_middle = round(len(image_3D[0, :, 0]) / 2)
image_2D = image_3D[:, index_middle, :, 0]
. . .
```

Code Snippet 3.6. Isolate the middle slice in the y-axis of a 3D MRI image for the 2D Brain Slices dataset.

By observing coronal slices in different positions of different patients, we concluded that in most of the middle slices the hippocampus is visible. The hippocampal structure plays an important role in memory issues that are strongly associated with AD since it is one of the first structures in the brain that is affected (Achilleos et al. 2020). The anatomical neural changes in terms of hippocampal volume and shape can be measured by using the MRI scan. Therefore, the hippocampus must be visible in the 2D slices.

So, for example, from the figures Figure 3.8 and Figure 3.9, which present different coronal slices of the same patient, the slice in the middle would are the ones we have used in our dataset. More specifically, in Figure 3.8, the slice 86 out of 171 slices in the y-axis of that NC patient is the one that was used as part of the 2D Brain Slices dataset. In Figure 3.9, the slice in the middle for that AD patient is the slice 94 out of 187 slices.

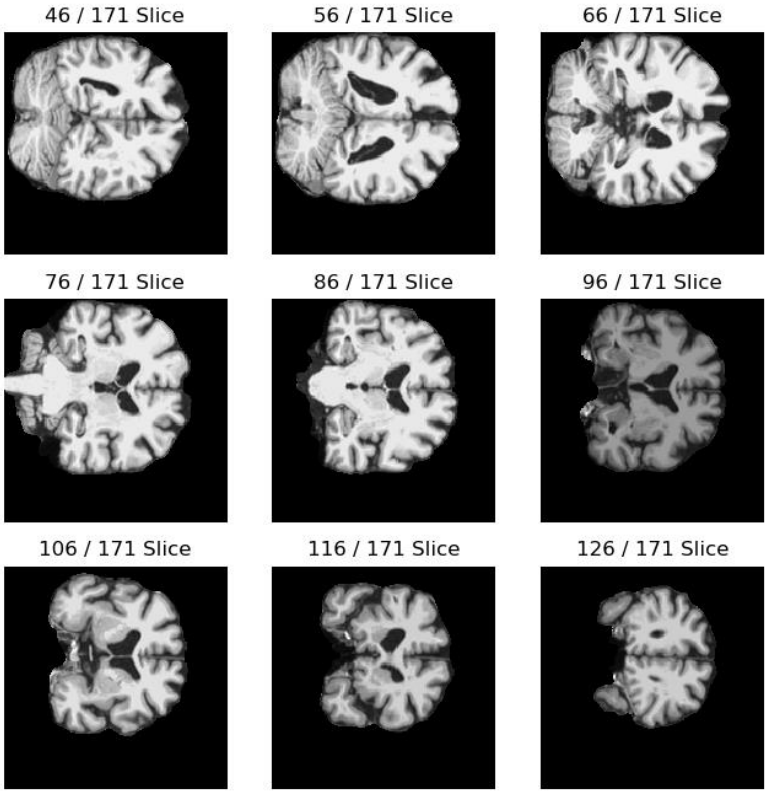


Figure 3.8. Different Coronal slices of the same NC patient's 3D MRI scan. The slice "86/171 Slice" is the one used in the 2D Brain Slices dataset.

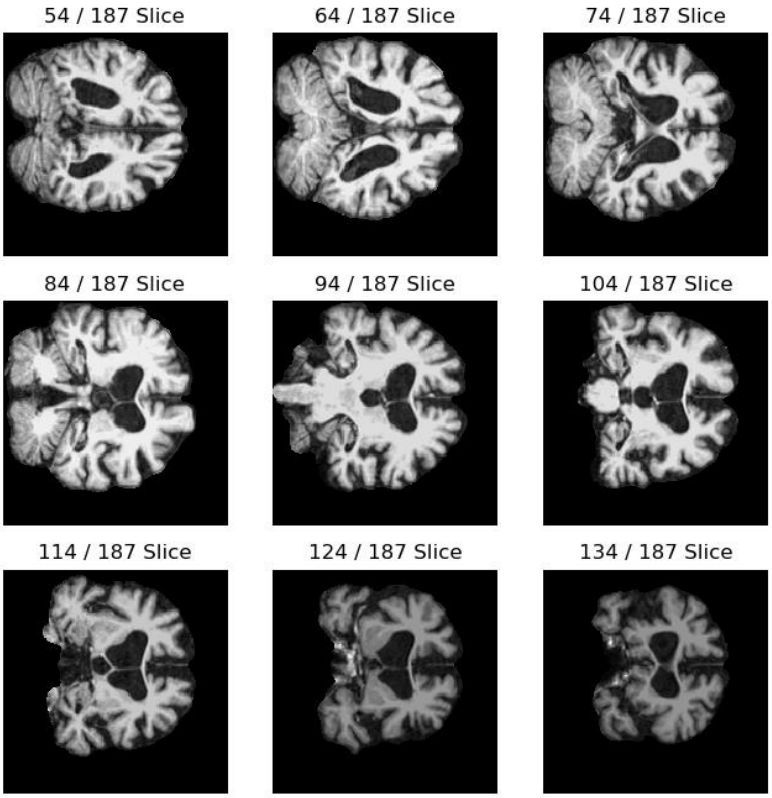


Figure 3.9. Different Coronal slices of the same AD patient's 3D MRI scan. The slice "94 / 187 Slice" is the one used in the 2D Brain Slices dataset.

The brain stem is the white part that is connected with the spinal cord in Figure 3.10.

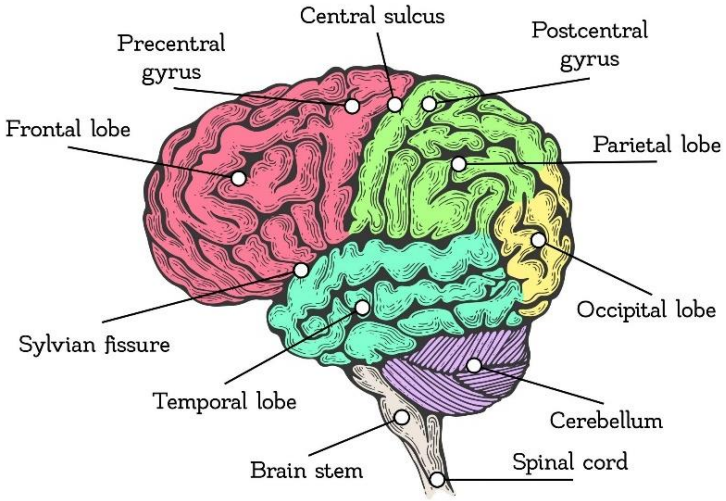


Figure 3.10. Anatomy of the brain.

It is worth mentioning that the slice in the middle of the patient in Figure 3.8 does not contain the brain stem. On the other hand, the brain stem of the slice in the middle slice of the patient in Figure 3.9, is visible. Ideally, the slices of all patients in the dataset, should not contain the brain stem. Since some brain slices have visible the brain stem and the cerebellum while others do not, this could occur some problems during training. In such a case, slice 104 in Figure 3.9 should be the one to be used in our dataset.

Unfortunately, we did not have enough computing power to isolate automatically the slices without the brain stem or/and the cerebellum. The other way could be to select them manually, which still is very time-consuming. Thus, the usage of the middle slice was the easiest solution to get training performance results as fast as possible. In future work, the selection of the slices and the alignment of the brains could be examined to see whether it affects the performance of the training process or not.

After isolating the middle slices, we can apply the padding easily by using the NumPy library of Python. All 2D Coronal slices are 0s padded to have the same dimensions, 174×174 pixels (Code Snippet 3.7).

```
import numpy as np

def add_padding(i):
    max_shape = (174, 174)

    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1]] = i
```

```
    return i_w_pad

# Apply padding of 0s in the image

. . .
image_w_pad = add_padding(image)
. . .
```

Code Snippet 3.7. Add padding of 0s to the 2D image for the dimensions to be 174 x 174.

Figure 3.11 shows a 2D Brain Slice of an NC patient after the padding of 0s being applied. Also, Figure 3.12 shows the 2D Brain Slice of an AD patient in the dataset of 2D Brain Slices with the black padding of 0s already applied.

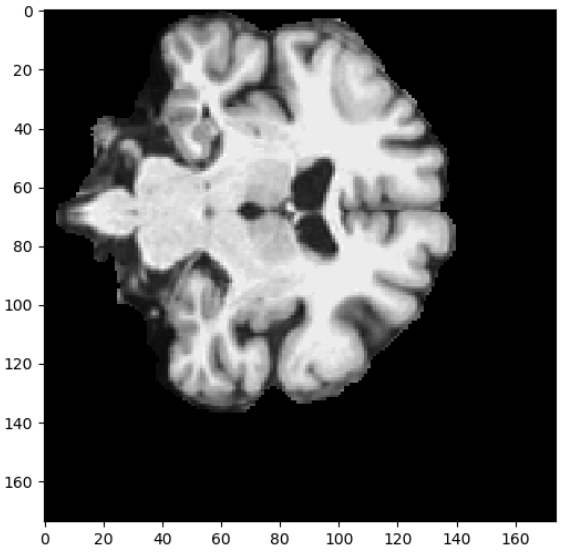


Figure 3.11. 2D Slice of a 3D Brain MRI scan of an NC patient.
image_2D = image_3D[:, round(length(image_3D[0, :, 0,), :]

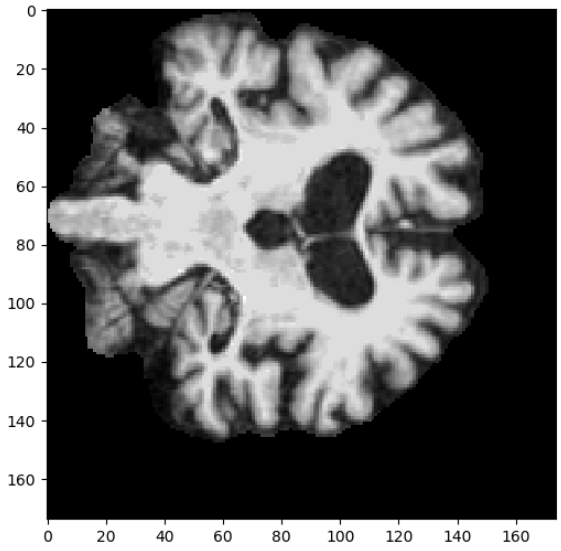


Figure 3.12. 2D Slice of a 3D Brain MRI scan of an AD patient.
image_2D = image_3D[:, round(length(image_3D[0, :, 0])), :]

The values of the pixels of a grayscale image are usually between 0 and 255. The MRI scanner produces images that their pixel values are not in the range of 0 to 255, but are much larger, and this does not benefit the training process. In Wang et al.'s (2020) algorithm, the one we are using for our experiments, automatically normalizes grayscale images from the range 0 to 255, to the range 0 to 1. Therefore, each pixel needs to be normalized between 0 and 255. The Code Snippet 3.8 performs the normalization of each image to the range 0 to 255.

```
# Normalize images between 0 and 255

. . .
image = (image / np.max(image)) * 255
. . .
```

Code Snippet 3.8. Normalize the value of each pixel between 0 and 255

Since each image's dimensions of this dataset are 174×174, this means that each sample is 30,276 floating points. The target class of each sample is either 1 for AD or 2 for NC.

3.2.3.2 Single Scan per Patient – Single Slice per Scan (B_2D_S)

For the 2D Brain Slices, Single Scan per Patient – Single Slice per Scan (B_2D_S) dataset, 10-fold CV was used. More specifically, StratifiedKFold (Section 3.1.3) was used to ensure that each fold would have AD = 19 and NC = 19, therefore, both the training and validation sets will always be balanced. There is no need for StratifiedGroupKFold to be used since the dataset contains only a single scan per patient and from each scan, only a single slice was used; consequently, there is no risk for data leakage.

The B_2D_S dataset contains 199 AD samples and 199 NC samples, with a total of 398 subjects. Before performing the StratifiedKFold, 9 samples were isolated from each target class to be part of the test set. This, later on, was proven a mistake since a really small test set of 18 subjects (4.5%), fails to accurately measure the performance metrics during the testing face (Beleites et. al. 2013). This could also be caused because the selected 18 subjects were not a meaningful representation of the rest training set. Since outliers had not been removed from the initial set, a large proportion of them might end up in the test set as well.

The rest 380 samples (AD = 190 & NC = 190) were used for the 10-fold CV splitting, which resulted in 10 balanced folds with each one of them having 38 samples (AD = 19,

NC = 19). Later on, the 10-fold option was proven not to be the best one based on our results. Mostly, this was an issue because 10 executions per experiment had to be run which was very power demanding and time-consuming. Additionally, since the dataset is relatively small, a 5-fold could be used to increase the size of the validation set as well (Table 3.2).

	AD	***NC***	***MCI***	***Total***
All Samples	199 (50%)	199 (50%)	-	398
Training Set	171	171	-	342 (85%)
Test Set	9	9	-	18 (4.5%)
Validation Set	19	19	-	38 (9.5%)

Table 3.2. Dataset: 2D Brain Slices [AD, NC],
Single Scan per Patient - Single Slice per Scan (B_2D_S) (10-fold CV)

In total, 398 subjects are in the B_2D_S dataset with 48.7% being Females and 51.3% being Males. The percentages between genders are pretty close, so no major performance issue should be observed. If the percentage of one of the genders was significantly larger than the other one, then this could potentially cause an issue since usually, the female brains are significantly smaller than the male ones (Table 3.3).

	Females	***Males***	***Total Patients***
Gender	194 (48.7%)	204 (51.3%)	398

Table 3.3. Gender Percentages: 2D Brain Slices [AD, NC], Single Scan per Patient – Single Slice per Scan (B_2D_S)

Code Snippet C.1 prepares the data needed for the B_2D_S dataset, by storing the AD and NC patients into two '.mat' files. Then, by using the Code Snippet D.7, we perform the 10-fold splitting with the StratifiedKFold.

3.2.3.3 Single Scan per Patient – 5 Slices per Scan (B_2D_5S)

For the 2D Brain Slices, Single Scan per Patient – 5 Slices per Scan (B_2D_5S) dataset, 5-fold CV was used, instead of a 10-fold CV that was used in the B_2D_S. The reason was to increase the size of the validation set, while reducing the number of executions per experiment, from 10 down to 5, which saves computational power and time.

The B_2D_5S dataset contains 1990 samples in total, 5 times more than the B_2D_S. The patients in the B_2D_5S dataset are the same as the ones in B_2D_S. Unlike the B_2D_S dataset, the B_2D_5S contains 5 images of the same scan as 5 distinct samples. Therefore,

StratifiedKFold (Section 3.1.3) could not be used since data leakage could occur when two slices of the same patient end up in the training and validation set respectively. So, StratifiedGroupKFold (Section 3.1.5) was used for creating the 5-fold splitting.

To create the dataset, 5 slices of each 3D Brain MRI scan have to be isolated. To understand which slices were used for the B_2D_5S we need to find the index of the middle slice first. The index of the middle slice can be found using:

$$i_{middle} = round(length(image\_3D[0, :, 0,)$$

Therefore, the middle slice is:

$$image\_2D = image\_3D[:, index_{middle}, :]$$

Hence, the 5 slices were extracted from a single 3D Brain MRI scan of a patient from the following indices:

$$[i_{middle} - 4, i_{middle} - 2, i_{middle}, i_{middle} + 2, i_{middle} + 4].$$

That being so, the dataset is 5 times larger than B_2D_S with AD = 995 and NC = 995, instead of 199 samples each. Consequently, the test set is 5 times larger too with AD = 45 and NC = 45. Nevertheless, the test set is still 4.5% of the whole B_2D_5S set. Considering that the same patients represent the test sets of both B_2D_5S and the B_2D_S sets if outliers had caused any issues in the B_2D_S, are expected to cause the same issued to the B_2D_5S as well. Since 5-fold was used, the validation set now is significantly larger with AD = 190 and NC = 190, which in total represents 19% of the initial dataset instead of 9.5% which was in the B_2D_S dataset. The remaining 4 folds construct the training set (76%) with AD = 760 and NC = 760 (Table 3.4).

	AD	***NC***	***MCI***	***Total***
All Samples	995 (50%)	995 (50%)	-	1990
Training Set	760	760	-	1520 (76%)
Test Set	45	45	-	90 (4.5%)
Validation Set	190	190	-	380 (19%)

Table 3.4. Dataset: 2D Brain Slices [AD, NC], Single Scan per Patient – 5 Slices per Scan (B_2D_5S) (5-fold CV)

In total, 398 subjects/patients are in the B_2D_5S, the same as the B_2D_S set with 48.7% being Females and 51.3% being Males (Table 3.5).

	Females	*Males*	*Total Patients*
Gender	194 (48.7%)	204 (51.3%)	398

Table 3.5. Gender Percentages: 2D Brain Slices [AD, NC], Single Scan per Patient – 5 Slices per Scan (B_2D_5S)

Code Snippet C.2 prepares the data needed for the B_2D_5S dataset, by storing the AD and NC patients into two '.mat' files. Then, by using the Code Snippet D.2, we perform the 5-fold splitting with the StratifiedGroupKFold (Code Snippet D.1).

3.2.3.4 Multiple Scans per Patient – Single Slice per Scan (B_2D_M)

The 2D Brain Slices, Multiple Scans per Patient - Single Slice per Scan (B_2D_M) dataset has in total 1204 samples. For each subject, one or more MRI scans were used from different periods. For example, a patient could have a baseline, 6-, and 12-month scans.

All the patients which were used in the dataset did not change class during their different scans. That is, each patient's scans label was the one that the patient ended up having on its last scan. For example, if a patient was diagnosed as AD after 18 months, but the first 3 scans (baseline, 6-, and 12-months) had been diagnosed as NC, all patient's scans would be labeled as AD.

This was done in purpose to detect whether a patient which seems to be NC today, may convert to AD after 1 or 2 years to provide early treatment early. Additionally, from each sample scan, only the slice in the middle was used, which we had isolated the same way as we did in the B_2D_S dataset (Section 3.2.3.2).

Owing to the fact that more than one samples of the same patient exist in the B_2D_M dataset, the StratifiedGroupKFold (Section 3.1.5) had to be used in order to avoid data leakage. This dataset was one of the first created during this thesis alongside the B_2D_S dataset. Therefore, it follows similar inaccuracies such as the small test set and the 10-fold CV split instead of a 5-fold one.

The B_2D_M dataset consists of 1204 samples in total with each class being perfectly balanced with AD = 602, and NC = 602. For the test set, 44 samples have been isolated where AD = 22 and NC = 22. Afterward, a 10-fold CV was performed, with each fold having 116 samples (AD = 58, NC = 58). The rest 9 folds were used for training with AD = 522 and NC = 522. Therefore, the validation set percentage is 9.6% while the training set is 86.7% (Table 3.6).

	AD	*NC*	*MCI*	*Total*
All Samples	602 (50%)	602 (50%)	-	1204
Training Set	522	522	-	1044 (86.7%)
Test Set	22	22	-	44 (3.6%)
Validation Set	58	58	-	116 (9.6%)

Table 3.6. Dataset: 2D Brain Slices [AD, NC],
Multiple Scans per Patient – Single Slice per Scan (B_2D_M) (10-fold CV)

Since the same patients were used, the gender percentages are the same as the 2D_B_S and 2D_B_5S datasets, with 48.3% being Females and 51.7% being Males (Table 3.7).

	Females	*Males*	*Total Patients*
Gender	582 (48.3%)	622 (51.7%)	1204

Table 3.7. Gender Percentages: 2D Brain Slices [AD, NC],
Multiple Scans per Patient – Single Slice per Scan (B_2D_M)

Code Snippet C.3 prepares the data needed for the B_2D_M dataset, by storing the AD and NC patients into two '.mat' files. Then, by using the Code Snippet D.8, we perform the 10-fold splitting with the StratifiedGroupKFold (Code Snippet D.1).

3.2.3.5 Multiple Scans per Patient – 7 Slices per Scan (B_2D_7M)

The dataset 2D Brain Slices, Multiple Scans per Patient – 7 Slices per Scan (B_2D_7M), uses the same samples as the B_2D_M but differentiates itself by using a 5-fold CV splitting method instead of a 10-fold. Additionally, just like the B_2D_5S dataset which uses 5 adjacent slices of the same 3D MRI scan near the middle, this uses 7 slices per scan. This approach supposedly would improve the performance of the network since the dataset's size increases by a factor of 7, from 1204 samples (B_2D_M) to 8428 samples. This dataset was proven later not to provide any performance benefits, and not only that, it increased significantly the training time as well.

This technique was expected to solve the problem of misaligned brains, where the middle brain slice is not representative of the class. Unfortunately, increasing the size of the dataset by a factor of 7, caused the training process to be as slow as training 3D Shrunk Brains (Section 3.2.5). Anyhow, the benefit of this 2D dataset over the 3D ones is that even if it takes a similar training time, could run without an issue in Arcadia, while 3D datasets could not, due to lack of dedicated GPUs.

For the same reason as B_2D_M (Section 3.2.3.4), StratifiedGroupKFold (Section 3.1.5) should be used for the 5-fold CV splitting. First, 308 samples (AD = 154, NC = 154) were isolated as test set. The validation set (single fold), has 1624 samples (AD = 812, NC = 812) and the training set has 6496 samples (AD = 3248, NC = 3248) (Table 3.8).

	AD	***NC***	***MCI***	***Total***
All Samples	4214 (50%)	4214 (50%)	-	8428
Training Set	3248	3248	-	6496 (77%)
Test Set	154	154	-	308 (3.6%)
Validation Set	812	812	-	1624 (19%)

Table 3.8. Dataset: 2D Brain Slices [AD, NC],
Multiple Scans per Patient – 7 Slices per Scan (B_2D_7M) (5-fold CV)

It is worth mentioning that, the size of the test set may seem large but still is just 3.6% of the initial dataset's size. Therefore, similar poor performance during testing is expected since usually when a slice of a patient is misclassified, the rest 6 slices would be misclassified as well.

The B_2D_7M has the same patients as the B_2D_M set, therefore, the gender percentages are the same, with 48.3% Females, and 51.7% Males (Table 3.9).

	Females	***Males***	***Total Patients***
Gender	582 (48.3%)	622 (51.7%)	1204

Table 3.9. Gender Percentages: 2D Brain Slices [AD, NC],
Multiple Scans per Patient – 7 Slices per Scan (B_2D_7M)

Code Snippet C.4 prepares the data needed for the B_2D_7M dataset, by storing the AD and NC patients into two '.mat' files. Then, by using the Code Snippet D.3, we perform the 5-fold splitting with the StratifiedGroupKFold (Code Snippet D.1).

3.2.4 2D Brain Slices [AD, MCI, NC] (slice-level MRI)

3.2.4.1 Multiple Scans per Patient – Single Slice per Scan (B_2D_M [AD, MCI, NC])

The 2D Brain Slices, Multiple Scans per Patient – Single lice per Scans [AD, MCI, NC] (B_2D_M [AD, MCI, NC]) dataset is the same as the B_2D_M (Section 3.2.3.4) dataset plus 602 MCI samples. The purpose of this dataset was to investigate how much the extra class affects the performance of the classification. The issues that the B_2D_M dataset

had remained the same to this dataset as well since to be able to perform a direct comparison, the 10-fold CV was maintained, as well as the small test set.

For the test set, 66 samples were isolated (AD = 22, MCI = 22, NC = 22). Then, StratifiedGroupKFold (Section 3.1.5) with 10-folds was used with each fold having 174 samples (AD = 58, MCI = 58, NC = 58). Subsequently, 1566 samples remained as the training set (AD = 522, MCI = 522, NC = 522) (Table 3.10).

	AD	***NC***	***MCI***	***Total***
All Samples	602 (33%)	602 (33%)	602 (33%)	1806
Training Set	522	522	522	1566 (86.7%)
Test Set	22	22	22	66 (3.6%)
Validation Set	58	58	58	174 (9.6%)

Table 3.10. Dataset: 2D Brain Slices [AD, NC],
Multiple Scans per Patient – Single Slice per Scan (B_2D_M [AD, MCI, NC]) (10-fold CV)

The gender percentages in the B_2D_M [AC, MCI, NC] dataset differ from the B_2D_M dataset. Instead of being 48.3% Females and 51.7% Males, now 42.2% are Females and 57.8% are Males. That being so, the MCI patients added are more males than females which causes the set to be slightly unbalanced in terms of gender (Table 3.11).

	Females	***Males***	***Total Patients***
Gender	762 (42.2%)	1044 (57.8%)	1806

Table 3.11. Gender Percentages: 2D Brain Slices [AD, NC],
Multiple Scans per Patient – Single Slice per Scan (B_2D_M [AD, MCI, NC])

For this dataset, the labeling has changed though. Instead of the labels being AD = 1 and NC = 2, now the labels are AD = 1, MCI = 2, and NC = 3. This does not affect in any way the training process or the performance of the model. Code Snippet C.8 prepares the data needed for the B_2D_M [AD, MCI, NC] dataset, by storing the AD, MCI, and NC patients into three '.mat' files. Then, by using the Code Snippet D.9, we perform the 10-fold splitting with the StratifiedGroupKFold (Code Snippet D.1).

3.2.5 3D Shrunk Brains [AD, NC] (subject-level MRI)

3.2.5.1 Preprocessing of 3D Shrunk Brains

3D Shrunk Brains dataset are the shrunk versions of the T1-weighted MRI scans of the brains from ADNI. The 3D MRI scans before shrinking are the same ones used for the

2D Brain Slices (Section 3.2.3 & 3.2.4) and 3D Cropped Brains (Section 3.2.7) datasets. To shrink the images, the average (numpy.mean) of each $4 \times 4 \times 4$ block was calculated. This decreased the size of the new image to $\frac{1}{64}$ of its original size. Code Snippet 3.9 shows the shrinking process used in this thesis for a 3D MRI scan.

```
import skimage.measure
import numpy as np

# Shrink MRI image by taking the mean of each 4x4x4 block

. . .
shrunk_image = skimage.measure.block_reduce(image, (4, 4, 4), np.mean)
. . .
```

Code Snippet 3.9. Shrunk each 3D image to 1/16 of its original size.
The mean of a 4 x 4 x 4 block of the original 3D image.

The shrinking was inevitable since the original image size of each sample was:

$$x \times y \times z = 174 \times 190 \times 174 \ = \ 5{,}752{,}440 \ pixels$$

Since each pixel is a floating point (32-bits) then the total size of a single scan would be:

$$Original\ Image\ Size = 5{,}752{,}440 \ \times 32 = 184{,}078{,}080 \ bits = 23 \ MB$$

Therefore, with a total of 398 samples, if only a single scan per patient was included, then the total size of the dataset would be:

$$Original\ Dataset\ Size = 23 \times 398 = {\sim}9.2 \ GB$$

For multiple MRI scans per patient, the total size is 1204 samples, thus the size of the complete dataset would be:

$$Full\ Dataset\ Size = 23 \times 1204 = {\sim}27.7 \ GB$$

Keep in mind that a network for such large 3D images would have millions of parameters. Consequently, the memory available on our machines (RAM) was not enough to store all these network's weights and the samples themselves. Therefore, the CPU would be performing a lot of context switching between the RAM and the Hard Disk which would make the learning process extremely slow.

To overcome those issues, we decided to trade details of the images for a smaller overall size. The new images, after compression, were:

$$x' \times y' \times z' = \ 44 \times 48 \times 44 \ = \ 92{,}928 \ pixles$$

This implies to:

$$Compressed\ Image\ Size = 92{,}928\ \times 32 = 2{,}973{,}696\ bits = 0.37\ MB$$

and the total of 398 samples is:

$$Compressed\ Dataset\ Size = \ 0.37 \times 398 = {\sim}0.15\ GB$$

Significantly smaller than the 9.2 GB without compression. To be more specific, the compressed dataset is $\frac{1}{64}$ times smaller than the uncompressed one:

$$Sizes\ Ratio = \frac{Compressed\ Image\ Size}{Original\ Image\ Size} = \frac{92{,}928}{5{,}752{,}440} = {\sim}\frac{1}{64} = {\sim}1.6\%$$

In Figures 3.13 and 3.14, two slices of a 3D scan are presented. On the left of both figures is a slice from the original 3D image without any compression being applied. On the right of both figures, is the slice resulted after compressing the whole image by replacing each $4\ \times 4\ \times 4$ block with the mean of its pixels values.

More specifically, Figure 3.13 compares the slices of an NC patient. We lose a lot of resolution and the details, and this affects especially tiny areas such as the hippocampal structure, which plays a major role in the learning process. Figure 3.14 compares the slices of an AD patient. We can observe that in this case, the shrunk slice looks brighter than the original one. Because each pixel is the mean of the surrounding 64 pixels, the value of a pixel may vary a lot. This does not benefit our learning procedure since many important details are lost as well.

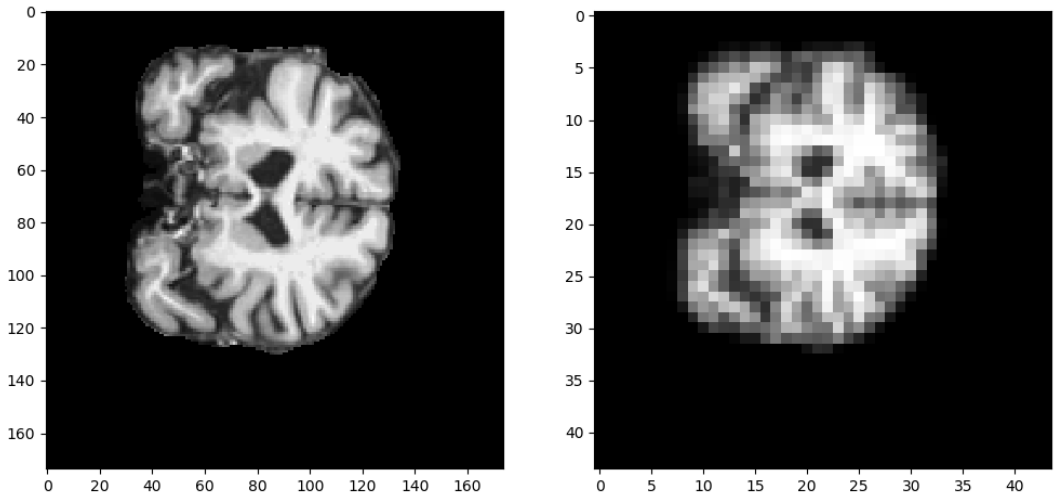


Figure 3.13. 3D Shrunk NC Brain's same slice uncompressed and compressed to 1/64 of the total size. **Left:** Slice [:, 120, :] uncompressed. **Right:** Slice [:, 30, :] compressed using the mean of each 4 x 4 x 4 block.

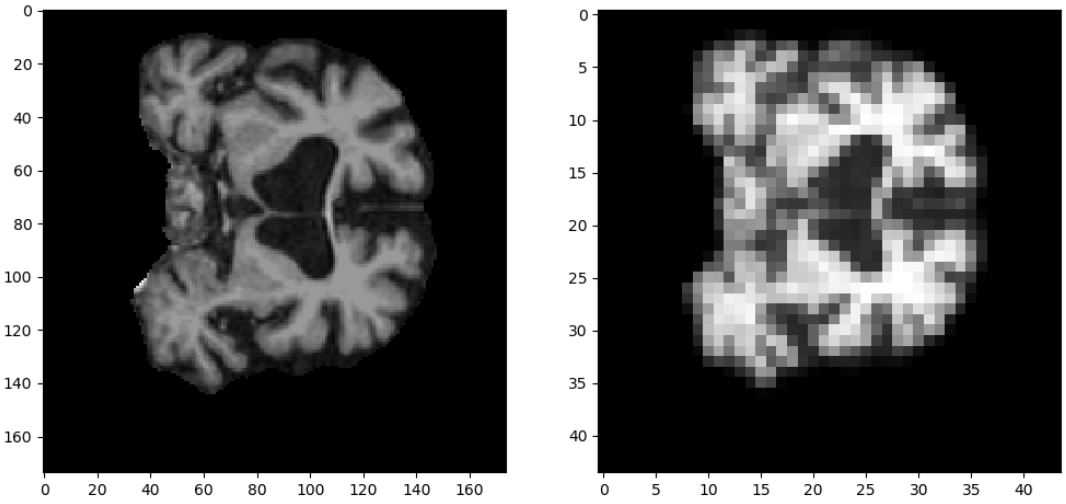


Figure 3.14. 3D Shrunk AD Brain's same slice uncompressed and compressed to 1/64 of the total size. **Left:** Slice [:, 120, :] uncompressed. **Right:** Slice [: , 30, :] compressed using the mean of each 4 x 4 x 4 block.

Later on, during the experiments, this method performs very poorly since the model most of the time fails to learn anything at all. The MRI scanner needs to be as detailed as possible down to the level of the millimeter. By performing such aggressive compression to the images due to lack of resources we have destroyed the data. That being so, in the future, experiments with the original size of 3D scans should be performed.

3.2.5.2 Single Scan per Patient (B_3D_S)

The 3D Shrunk Brains, Single Scan per Patient (B_3D_S) dataset, uses a 5-fold CV for splitting. Since the samples are much larger than the B_2D_S dataset, each execution takes exceptionally much more time to be completed and much more resources. Additionally, due to the small size of the dataset, a 5-fold CV has to be used instead of a 10-fold CV. More specifically, the StratifiedKFold (Section 3.1.3) was used for the 5-fold CV, as there was no need for using StratifiedGroupKFold because only a single scan per patient exists in the dataset, and as a result, there was no risk of data leakage by not taking into consideration the groups.

The B_3D_S dataset, similarly to the B_2D_S, contains the same 199 AD samples and 199 NC samples, with a total of 398 subjects. Before performing the StratifiedKFold, 19 samples were isolated from each target class as the test set (AD = 19, NC = 19). Thus, the test set's percentage is now 10%, significantly larger than the ones in the 2D datasets' test set (3.6% and 4.5%). This choice has been made after observing from the 2D experiments that the test accuracy was notably worse because the test set's size was not sufficient

(Beleites et. al. 2013). The rest 360 samples (AD = 180 & NC = 180) were used for the 5-fold CV splitting, which resulted in 5 balanced folds, with each one having 72 samples. This generates five training sets with 288 samples each (AD = 144, NC = 144), and five validation sets with 72 samples each (AD = 36, NC = 36). The percentage of the validation set based on the total size of the B_3D_S set is 18% which is much larger than the one in the 2D datasets (Table 3.12).

	AD	***NC***	***MCI***	***Total***
All Samples	199 (50%)	199 (50%)	-	398
Training Set	144	144	-	288 (72%)
Test Set	19	19	-	38 (10%)
Validation Set	36	36	-	72 (18%)

Table 3.12. Dataset:3D Shrunk Brains, Single Scan per Patient (B_3D_S) (5-fold CV)

In total, 398 subjects are in the B_3D_S dataset (same as the B_2D_S) with 48.7% being Females and 51.3% being Males (Table 3.13).

	Females	***Males***	***Total Patients***
Gender	194 (48.7%)	204 (51.3%)	398

Table 3.13. Gender Percentages: 3D Shrunk Brains, Single Scan per Patient (B_3D_S)

Code Snippet C.5 prepares the data needed for the B_3D_S dataset, by storing the AD and NC patients into two '.mat' files. Then, by using the Code Snippet D.4, we perform the 5-fold splitting with the StratifiedKFold.

3.2.6 3D Left Hippocampus [AD, NC]

3.2.6.1 Introduction to 3D Left Hippocampus

The 3D Left Hippocampus [AD, NC] dataset was created by Achilleos et al. (2020) in which from the 3D MRI brain images, the hippocampal structure was isolated. Due to a lack of computational resources, from the left and right hippocampal structures, only the left was used for classifying whether the patient was AD or NC.

The hippocampus, especially the left one, takes the most damage during Alzheimer's disease (Achilleos et al. 2020). So, it was expected that with this dataset which is more focused on the source of the problem, the results would be much better than using the

whole brain of the patient. Explicitly, the hippocampus of an AD subject is notably smaller in terms of volume in comparison with an NC patient.

To create the dataset, the 3D images were padded with 0s to change the shape of all images into $37 \times 32 \times 50$. In total 296 unique patients' left hippocampus were used, 148 ADs and 148 NCs.

3.2.6.2 Single Scan per Patient (LH_3D_S)

The 3D Left Hippocampus, Single Scan per Patient (LH_3D_S) consists of 296 samples. The dataset is balanced with 148 ADs and 148 NCs. 5-fold CV was used since the dataset is relatively small. More specifically, StatrifiedKFold (Section 3.1.3) was applied since the dataset contains only a single scan per patient, thus, no danger of data leakage exists.

The validation set is the17% of the original set with 52 samples (AD = 26, NC = 26), while the training set has 208 samples (AD = 148, NC = 148). Finally, the test set, which is much larger than the 2D datasets while being 12% of the original set, has 36 samples (AD = 18, NC = 18) (Table 3.14).

	AD	***NC***	***MCI***	***Total***
All Samples	148 (50%)	148 (50%)	-	296
Training Set	104	104	-	208 (70%)
Test Set	18	18	-	36 (12%)
Validation Set	26	26	-	52 (17%)

Table 3.14. Dataset:3D Left Hippocampus, Single Scan per Patient (LH_3D_S) (5-fold CV)

In this dataset, no gender information was available for the patients, so we were unable to identify whether the dataset is balanced or unbalanced in terms of gender.

Code Snippet C.7 prepares the data needed for the LH_3D_S dataset, by storing the AD and NC patients into two '.mat' files. Then, by using the Code Snippet D.6, we perform the 5-fold splitting with the StratifiedKFold.

3.2.7 3D Cropped Brains [AD, NC]

3.2.7.1 Preprocessing of 3D Cropped Brains

The T1-weighted MRI scans from ADNI had been used to create the 3D Cropped Brains. Without adding any padding on the original images, they were cropped in a specific

matter to include the left hippocampus of the patient. No compression was used; therefore, the results are expected to be better than the 3D Shrunk Brains (Section 3.2.5) dataset. The 3D MRI scans before copping are the same ones used for the 2D Brain Slices (Section 3.2.3 & 3.2.4) and 3D Shrunk Brains (Section 3.2.5) datasets.

To crop the images the pixels 0 to 70 from the x-axis, the pixels 20 to 80 from the y-axis, and the pixels 30 to 90 from the z-axis have been isolated (Figure 3.15 Left). The result is a new 3D image with dimensions $70 \times 60 \times 60$, that mostly includes the left hippocampus of a patient (Figure 3.16 Right).

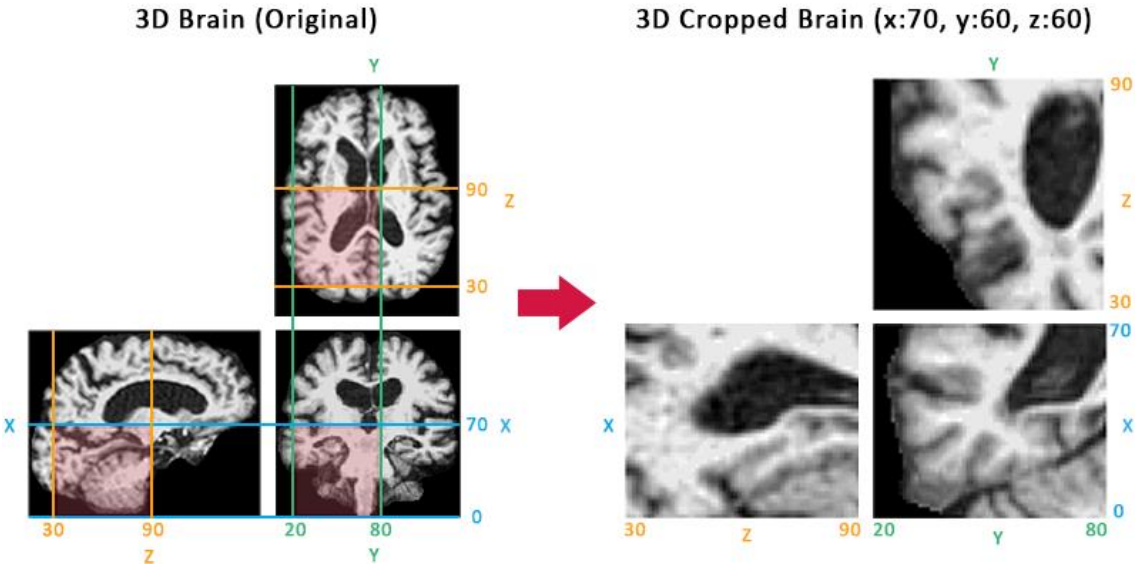


Figure 3.15. Isolating the left hippocampus of a patient. **Left:** 3D Brain (Original). **Right:** 3D Cropped Brain.

3.2.7.2 Single Scan per Patient (CB_3D_S)

The 3D Cropped Brains, Single Scan per Patient (CB_3D_S) dataset uses the same samples as the B_3D_S (Section 3.2.5.2) and B_2D_S (Section 3.2.3.2) datasets. Because only a single scan per patient exists in the dataset for the 5-fold CV, same as with the B_3D_S set, StratifiedKFold (Section 3.1.3) was used.

	AD	***NC***	***MCI***	***Total***
All Samples	199 (50%)	199 (50%)	-	398
Training Set	144	144	-	288 (72%)
Test Set	19	19	-	38 (10%)
Validation Set	36	36	-	72 (18%)

Table 3.15. Dataset:3D Cropped Brains, Single Scan per Patient (CB_3D_S) (5-fold CV)

The number of samples per set is the same as the B_2D_S set. The validation set has 72 samples (AD = 36, NC = 36), the training set 288 samples (AD = 144, NC = 144) and the test set 38 samples (AD = 19, NC = 19). All of them are balanced with an equal number of ADs and NCs (Table 3.15).

The gender percentages are the same on both B_3D_S and B_2D_S sets since both of them use the same patients. The Females are 48.7% and the Males are 51.3%, so, no issue should occur because the dataset is slightly imbalanced in terms of gender (Table 3.16).

	Females	***Males***	***Total Patients***
Gender	194 (48.7%)	204 (51.3%)	398

Table 3.16. Gender Percentages: 3D Cropped Brains, Single Scan per Patient (CB_3D_S)

Code Snippet C.6 prepares the data needed for the CB_3D_S dataset, by storing the AD and NC patients into two '.mat' files. Then, by using the Code Snippet D.5, we perform the 5-fold splitting with the StratifiedKFold.

3.3 Classification Metrics

3.3.1 Confusion Matrix

3.3.1.1 TP, TN, FP, FN

The labeling on the aforementioned datasets (Section 3.2) goes as follows:

- For experiments with AD and NC samples, AD = 1 and NC = 2.
- For experiments with AD, MCI and NC samples, AD = 1, MCI = 2, NC = 3.

Usually, the AD is labeled as 2 and NC as 1, but in my case, I found out that the labeling was important a bit too late, after running half of the experiments. Thus, we can make this convention for this thesis in order not to get confused. For the experiments with AD and NC only, the True Positives (TP) represent how many patients who are NC have been correctly predicted as NC. Consequently, the True Negatives (TN) represent how many patients who are AD have been correctly predicted as AD. False Positives (FP) represent the number of predictions where the patient was AD but was misclassified as NC, and False Negatives (FN) the times where the patient was NC and was misclassified as AD.

The Condition Positive (P) is the sum of TP and FN (P = TP + FN), which are the real positive cases in the data, and the Condition Negative (N) is the sum of TN and FP (N = TN + FP), the real negative cases in the data. In our case, P is the number of NC samples and the N is the number of AD samples.

The AD/NC problem demands that the model achieves small FN and large TN values to be considered a success. Therefore, the Specificity should be high which helps detect when a patient might have AD so we will guide them to further examination. Normally, Sensitivity is the metric that matters the most, but due to the "mistake" on the labeling for AD being 1 instead of 2, the Specificity will be more interesting for us.

3.3.1.2 2 × 2 Confusion Matrix

The confusion matrix or error matrix allows us to visualize the performance of an algorithm. Its dimensions are depended on the number of target classes. For example, for the AD/NC experiments, the dimensions of the confusion matrix will be 2×2, while for the AD/MCI/NC experiments the dimensions will be 3×3. The rows represent the instances of the real/predicted classes, while each column represents an instance of a target/actual class. In general, the confusion matrix tells us whether and how the algorithm confuses the classes between them. In our case, for the AD/NC problem, the confusion matrix will have a similar form to the one in Figure 3.16.

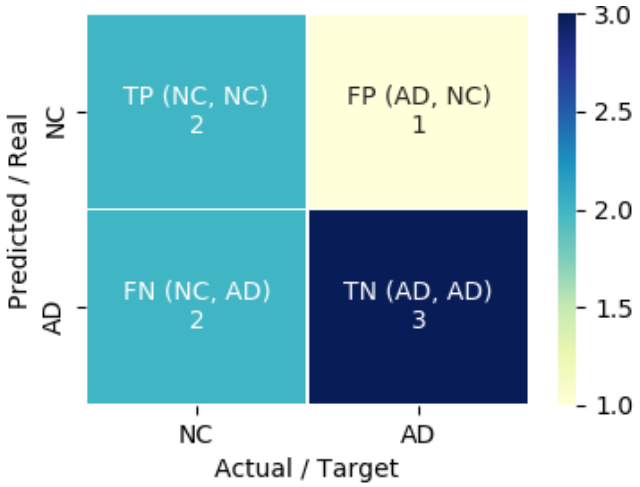


Figure 3.16. Examples of a confusion matrix for the two target classes AD and NC.

Since AD = 1 and NC = 2, the TP would be the cases when the target was NC and the predicted/real value was NC as well. On that account, the TN would be the cases in which the target class and the real value were ADs. Oppositely, FP represents the cases where

the target was AD but the prediction was NC; and FN is the cases where the target was NC and the prediction value was AD. Thus, in the aforementioned example, the TP = 2 the TN = 3, FP = 1 and FN = 2. From those counters, different metrics can be calculated such as Accuracy, Specificity, Sensitivity, Positive Predictive Value, and Negative Predictive Value that are extremely useful for Machine Learning.

The Code Snippet 3.10 shows how we can use the confusion_matrix function of the scikit-learn library to generate the confusion matrix, given as input two arrays with the target and real values.

```
from sklearn.metrics import confusion_matrix

target = [0,0,0,0,1,1,1,1]   # Target classes that the algorithm needs to identify
real = [1,0,0,0,1,0,1,1]     # Predicted classes based on algorithm's decisions

conf_matrix = confusion_matrix(target, real)

tn = conf_matrix[0][0]   # True Negatives
fp = conf_matrix[0][1]   # False Positives
fn = conf_matrix[1][0]   # False Negatives
tp = conf_matrix[1][1]   # True Positives
```

Code Snippet 3.10. Create the Confusion Matrix with scikit-learn

This method produces a confusion matrix which is a 2×2 matrix with 4 values, in the cases that the input arrays have only 2 predictive values. The four values, TP, TN, FP, FN, can be found in the output matrix of that method in the following cells:

1. TN = conf_matrix[0][0]
2. FP = conf_matrix[0][1]
3. FN = conf_matrix[1][0]
4. TP = conf_matrix[1][1]

3.3.1.3 3×3 Confusion Matrix

For the AD/MCI/NC problem the algorithm has to learn three target classes, thus, a confusion matrix of dimensions 3×3 will be created, just like the one in Figure 3.17. In our experiments, we need most of the large numbers (darker blue color) to be in the diagonal of the confusion matrix. This provides us with the information that the algorithm correctly identifies the samples to their target class.

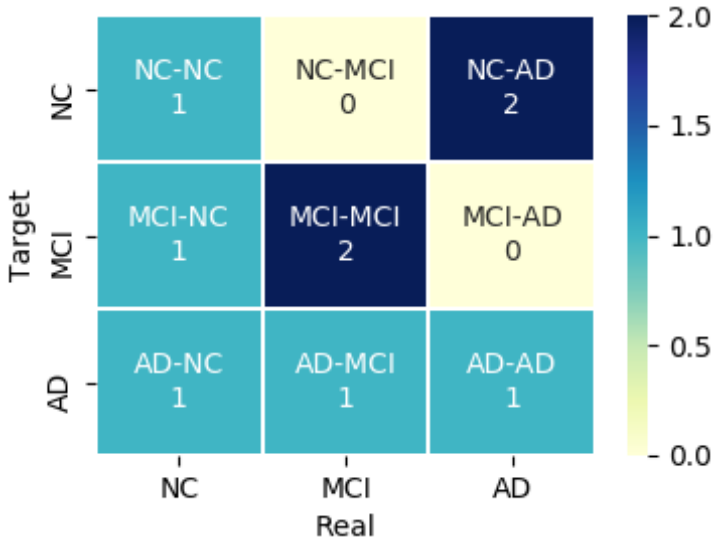


Figure 3.17. Examples of a confusion matrix for the three target classes AD, MCI, and NC.

For the example in Figure 3.17, we can see that the algorithm confuses NCs with ADs since 2 ADs were predicted as NCs (cell in the top right corner). Also, there is no confusion for an MCI to be classified as NC since zero MCI samples were predicted as NC (cell in the middle top). These kinds of conclusions, the confusion matrix helps us to come up with. Since 3 target classes exist, each class will have its own TP, TN, FP, and FN values. These will help us calculate the Sensitivity, Specificity, Accuracy, Positive Predictive Value, and Negative Predictive Value of each class. For the values, TP, TN, FP, and FN to be calculated the method demonstrated in Figure 3.18 is going to be used in an $N \times N$ confusion matrix.

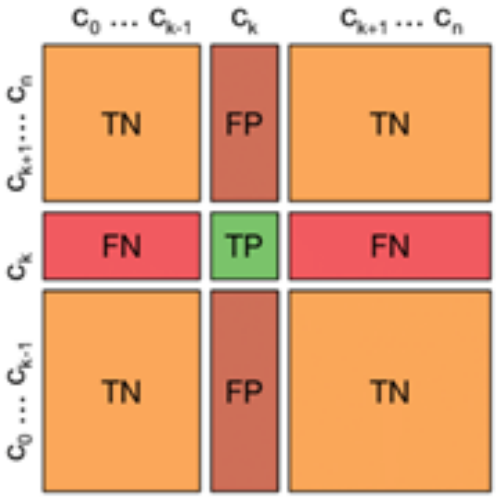


Figure 3.18. Finding TP, TN FP, and FN of the C_k class in an $N \times N$ confusion matrix.

Now that we have calculated the TP, TN, FP, and FN values for each one of the N classes, we can calculate other metrics the same way as in a problem with 2 target classes (Section 3.3.1.2).

3.3.2 Precision / Positive Predictive Value (PPV)

Precision or so-called Positive Predictive Value (PPV) on the AD/NC problem tells us the rate that the algorithm identifies correctly the NC patients, over the total of NC predictions. More specifically, the PPV is the proportion of correctly classified NCs, divided by the total predictions of NCs, and can be calculated from the equation:

$$PPV = \frac{TP}{TP + FP}$$

The ideal value of PPV is 1, and the worst possible value would be 0. If PPV = 1 implies that from all NC predictions, all of them were for real NCs.

3.3.3 Negative Predictive Value (NPV)

Negative Predictive Value (NPV), on the AD/NC problem, represents the rate that the algorithm correctly identifies the AD patients, over the total of AD predictions. Therefore, NPV is the proportion of correctly classified ADs, divided by the total of AD predictions, and can be calculated from the following equation:

$$NPV = \frac{TN}{TN + FN}$$

Similar to PPV, the ideal value of NPV is 1, which implies that from all AD predictions, all of them were for real ADs.

3.3.4 Sensitivity / Recall / True Positive Rate (TPR)

The Sensitivity measure, as well as the Specificity, are statistical measures of the performance of a binary classification problem (e.g., AD/NC problem), that are widely used in medicine.

Frequently, Sensitivity measures how many sick people are correctly identified as sick. However, in this thesis, due to the labeling of the classes where AD = 1 and NC = 2, Sensitivity plays the role of Specificity and Specificity the role of Sensitivity. Therefore, Sensitivity represents how many selected elements are truly negative, which means, how

many healthy people are correctly identified as healthy (NC). So, the Sensitivity is the proportion of correctly predicted NCs, over the total number of NCs.

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN}$$

The ideal value of Sensitivity is 1, and the worst value is 0. For Sensitivity being equal to 1, this implies that all the NC patients are correctly classified as NCs.

3.3.5 Specificity / True Negative Rate (TNR)

Usually, Specificity measures how many healthy people are correctly identified as being healthy. Keep in mind that in our case, due to the labeling AD = 1 and NC = 2, Specificity identifies how many AD patients are correctly identified as ADs, instead of how many NC patients are correctly identified as NCs. Consequently, Specificity in this thesis will refer to how many people who have for real the disease, are correctly identified as having the disease. Therefore, is not the proportion of negatives that are correctly identified, but how many relevant items are selected.

The Specificity or so-called True Negative Rate (TNR) represents how accurately our model can predict the ADs when the subject is an AD and does not misclassify them as NC. In the AD/NC problem, Specificity is the most important parameter because it tells us whether a patient might be an AD, therefore we can perform a further investigation on the patient through other techniques such as interviews, questionnaires, etc. On the other hand, if the Specificity is low, then most ADs would be misclassified as NCs. This is an issue because the doctor may set them free without further examination which could be proven catastrophic for the patient.

$$TNR = \frac{TN}{N} = \frac{TN}{TN + FP}$$

The ideal value of Specificity is 1, and the worst possible value would be 0. If Specificity is equal to 1, it means in this thesis that all AD patients are correctly classified as ADs.

3.3.6 Accuracy (ACC)

Accuracy (ACC) is the performance measure that is frequently used in binary classification and corresponds to the proportion of correctly identified AD and NC samples (correct predictions) over the total size of cases examined.

$$ACC = \frac{TP + TN}{P + N} = \frac{TP + TN}{TP + TN + FP + FN}$$

The best possible value for ACC is 1, which means that all NCs and ACs are classified correctly. In the experiments of this thesis (Chapter 5), three accuracies will be examined; training, validation, and test accuracies.

3.3.7 Loss Functions

3.3.7.1 Mean Squared Error (MSE) Loss

The Mean Squared Error (MSE) (Section 2.3.4) loss is the one that frequently being used for regression problems (Section 2.3.1). MSE is the average of the squared differences between the predicted and actual values. The value of the MSE loss is always positive, regardless of the sign of the predicted and actual values.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(target_i - predicted_i)^2.$$

The squaring means that larger mistakes will punish the model more than smaller mistakes. The ideal value for MSE loss is 0. By default, the NewtonCG implementations (Wang et al. 2020) were using MSE loss as the loss function. Only one experiment was performed with Cross-entropy loss (Section 3.3.7.2) since MSE loss was performing better.

3.3.7.2 Cross-entropy Loss

Cross-entropy loss (Section 2.3.5) measures the performance of a classification problem (Section 2.3.2) in which the output is a probability value between 0 and 1. Cross-entropy loss decreases as the predicted probability converges to the actual label. In a binary classification problem where t is the target value p is the predicted value, the Cross-entropy loss can be calculated as:

$$Cross\ Entropy = -(t\log(p) + (1-t)\log(1-p))$$

Chapter 4

Implementation

4.1 A new approach for Automating Dementia Diagnosis

To approach the automation of Alzheimer's disease diagnosis, we will start from the AD/MCI problem, and then examine whether the techniques generalize well in the AD/MCI/NC problem as well. More specifically, the "AD/MCI/NC problem" is a classification problem since you have to classify whether a patient is a member of one of these three classes (AD, MCI, or NC); while the "AD/NC problem" is the classification between two classes (AD, NC). Since this thesis approaches the problems with Machine Learning models, an ANN (Section 2.3.8) can be used to perform the classification (Section 2.3.2). Based on the format of the dataset, we need to select between an MLP (Section 2.3.11) and CNN (Section 2.3.12).

Supervised learning (Section 2.3.6) is going to be applied to all the experiments of this thesis since the target class is known in our datasets. The Hippocampus Features dataset (Section 3.2.2) will use an MLP to perform the classification since the features (10 floating points) that describe the patient's health condition, are already extracted manually by Achilleos et al. (2020). On the other hand, the rest of the datasets (Section 3.2.3 – 3.2.7) that consist of MRI images, the features have to be extracted, so a CNN needs to be used. This thesis, experiments with both MLPs and CNNs, and compares their performance metrics (accuracy, loss, sensitivity, specificity, etc.).

Mainly, this thesis aims not only to compare the performance of network configurations between MLP and CNN for the AD/NC problem, but also, test the performance of deferent optimizers, such as SGD (Section 2.4.3), Adam (Section 2.4.4), and

HFO/NewtonCG (Section 2.4.7), for each one of these models. Additionally, 9 different datasets are being used in more than 80 experiments with distinct hyperparameters on each one of them. The 9 different datasets were created based on the ADNI dataset (Section 3.2). Table 4.1 defines the naming of those datasets. The naming of the experiments will be based on the dataset's name.

Dataset Name	***Description***
HF_M	Hippocampus Features (Achilleos et al. 2020) from Multiple scans per patient (AD, NC)
B_2D_S	One (1) 2D slice per Brain scan, from a Single scan per patient (AD, NC)
B_2D_5S	Five (5) 2D slices per Brain scan, from a Single scan per patient (AD, NC)
B_2D_M	One (1) 2D slice per Brain scan, from Multiple scans per patient (AD, NC)
B_2D_7M	Seven (7) 2D slices per Brain scan, from Multiple scans per patient (AD, NC)
LH_3D_S	One (1) 3D Left Hippocampus (Achilleos et al. 2020) isolated from a Single scan per patient (AD, NC)
B_3D_S	One (1) 3D shrunk Brain, from a Single scan per patient (AD, NC)
CB_3D_S	One (1) 3D area Cropped from the Brain, from a Single scan per patient (AD, NC)
B_2D_M [AD, MCI, NC]	One (1) 2D slice per Brain scan, from Multiple scans per patient (AD, MCI, NC)

Table 4.1. Names of the datasets used in this thesis.

By saying "Multiple scans per patient", we mean that one or more T1-weighted MRI scans of the same patient exist in the dataset. For example, for the same patient, we might have their baseline, 6-, 12-, and 18-months scans. On the other hand, "Single scan per patient" means that we are using only the first T1-weighted MRI scan of each patient.

"One (1) 2D slice …" (Section 3.2.3.1) refers to the middle slice of a 3D Brain MRI scan of a patient. The index of the middle slice can be found using:

$$i_{middle} = round(length(image\_3D[0, :, 0,)$$

Therefore, the middle slice is:

$$image\_2D = image\_3D[:, index_{middle}, :]$$

Respectively, "Five (5) 2D slices …" (Section 3.2.3.1) refers to the 5 slices near the middle of a 3D Brain MRI scan of a patient with the following indices:

$$[i_{middle} - 4, i_{middle} - 2, i_{middle}, i_{middle} + 2, i_{middle} + 4].$$

Similarly, the "Seven (7) 2D slices …" (Section 3.2.3.1) are the 7 slices with the indices:

$$[i_{middle} - 6, i_{middle} - 4, i_{middle} - 2, i_{middle}, i_{middle} + 2, i_{middle} + 4, i_{middle} + 6].$$

Each 2D slice is (174 × 174) pixels, which is relatively large for the experiments to be executed in my personal computer (2-core CPU, 8GB RAM, no external GPU). So, in the beginning, I was trying to use Google Collab to run my experiments. Unfortunately, Google Collab offers similar hardware capabilities to a personal computer (2-core CPU, 12GB RAM, not always guaranteed access to external GPU), and additionally limits the execution time to 24 hours, and 90 minutes idle time. Consequently, running experiments in Google Collab that take 6 – 10 hours wasn't feasible.

As an alternative solution we had decided to use the Arcadia server of the University of Cyprus, for the experiments, which offers a 32-core CPU, 128GB RAM, but no external GPU). Therefore, the Arcadia server was a great choice for the 2D implementations. Nevertheless, Arcadia could not handle the 3D experimenters, where their images had dimensions, (60 × 60 × 70), (32 × 37 × 50), and (44 × 48 × 44).

In comparison with The Cyprus Institute server which took approximately 4 seconds per epoch, Arcadia needed 4000 seconds or even more for the same task. The Cyprus Institute's servers support dedicated GPUs which are essential for training CNNs, especially for 3D CNNs. I have used my personal computer only for the experiments for the Hippocampus Features (Section 3.2.2) which were used in simple MLP networks with no need for much RAM or any high requirements of CPU and GPU power.

The NewtonCG implementation that was provided by Wang et al. (2020) was in Python 3 and MATLAB, so we had to choose one of them. We decided to use Python 3 for all the implementations in this thesis since it is one of the most frequently used programming languages for Machine Learning. The reason is that Python 3 provides many libraries for data handling, and especially a vast variety of libraries dedicated to Machine Learning, where some of them are TensorFlow, Keras, scikit-learn, Scipy.io, Pandas, and NumPy.

All the source code for the MLP implementations can be found in Appendix A, while all the CNN implementations in Appendix B. Appendix C contains the source code for creating the datasets, while Appendix D the code for the 5- and 10-fold splitting. Finally, Appendixes E.1, E.3 – E.5 provide details on how to run the CNN implementations (Section 4.3) in the Arcadia server, while Appendix E.2 provides information on how to run the MLP implementations (Section 4.2).

4.2 MLP Implementations

For the Hippocampus Features (HF_M) dataset (Section 3.2.2), MLP networks (Section 2.3.11) were used for our experiments. Experiments in MLPs have utilized all three optimizers, SGD (Section 4.2.1.1), Adam (Section 4.2.1.2), and HFO (Section 4.2.1.3). The source code for these optimizers can be found in Appendix A. For the creation of the figures in Section 4.2, with the different network configurations, the tool (http://alexlenail.me/NN-SVG/AlexNet.html) was used.

4.2.1 Optimizers

4.2.1.1 SGD

For the MLP implementations with the SGD (Section 2.4.3) optimizer, Python's library scikit-learn was used. More specifically the sklearn.neural_network.MLPClassifier module (Pedregosa et al. 2011), a Multi-Layer Perceptron classifier. The library provides several activation functions such as the identity, logistic, tanh, and ReLU; and some solvers such as L-BFGS (quasi-Newton methods optimizer), SGD, and Adam. Additionally, the library supports the L2 penalty, a regularization term parameter defined as alpha (α) (Section 2.3.13.7).

The SGD optimizer explicitly supports different learning rate schedules for weight updates. The different options are constant, inverse scaling, and adaptive. By default, our experiments were set to constant, which means that the learning rate will be initially set to a value and remain constant throughout the whole training process. Furthermore, the SGD optimizer supports momentum for the gradient descent update, which is a value between 0 and 1 and helps the algorithm avoid local minimums. The momentum was set to 0.8 for the experiments with the SGD of this thesis. The source code of the MLP with SGD implementation can be found in Code Snippet A.2.

4.2.1.2 Adam

The Adam (Section 2.4.4) implementation with the MLP is the same as the SGD one since it uses the same modules from the scikit-learn library. The difference in the MLPClassifier object is the solver which is set to 'adam' instead of 'sgd'. Also, the momentum is removed, since the Adam optimizer does not support it. The source code of the MLP with Adam's implementation can be found in Code Snippet A.1.

The Adam optimizer supports also a unique hyperparameter; the epsilon (ε), which is responsible for numerical stability and is set by default to the value 1e-8. Mostly ε helps to avoid division by zero while updating the weights when the gradient is almost zero. In the experiments (Section 5.2.4), we are not going to modify the initial value of ε.

4.2.1.3 HFO

For the MLP implementation with the HFO optimizer (Section 2.4.9), the Python library "hessianfree 0.4.0" (https://pypi.org/project/hessianfree/) was used. This library contains all the standard features of Hessian-Free Optimization based on Martens (2010), and Martens and Sutskever (2012). Thus, the implementation includes Gauss-Newton approximation (Section 2.4.9), early termination, CG backtracking, Tikhonov damping, structural damping, etc.

The code works for feedforward networks and provides standard nonlinearities such as logistic, tanh, ReLU, and softmax, while it supports also custom nonlinearities. For the loss functions, it supports MSE (Section 2.3.4), Cross-entropy (Section 2.3.5), sparsity constraints, etc. The library provides the ability to change the number of layers and neurons per layer as well. The source code of the MLP with HFO implementation of MLP with HFO can be found in Code Snippet A.3 & A.4.

A variable which we modify in the MLP with HFO implementations during the experiments is the 'CGiter', which is the Conjugate Gradient iterations until it converges (Section 2.4.6). Usually, this number does not need to be large since, in the first few CG iterations, the model advances a lot.

4.2.2 MLP Network Architectures

The MLP networks (Adam, SGD, HFO) have been used with the Hippocampus Features with Multiple scans per patient (HF_M) dataset (Section 3.2.2.2). Based on 10 floating-point values, the features of the hippocampus, the network needs to identify whether the patient is either AD or NC. In this dataset, the AD target class is 1, while NC is 0.

For the following network architectures (Section 4.2.2.1 – 4.2.2.4), a single output neuron was used. If the network outputs 0 then it predicts NC, otherwise, if it outputs 1 then it predicts AD. All MLP network architectures have 10 input neurons which correspond to the 10 hippocampal features.

4.2.2.1 [10, 8, 8, 1] MLP

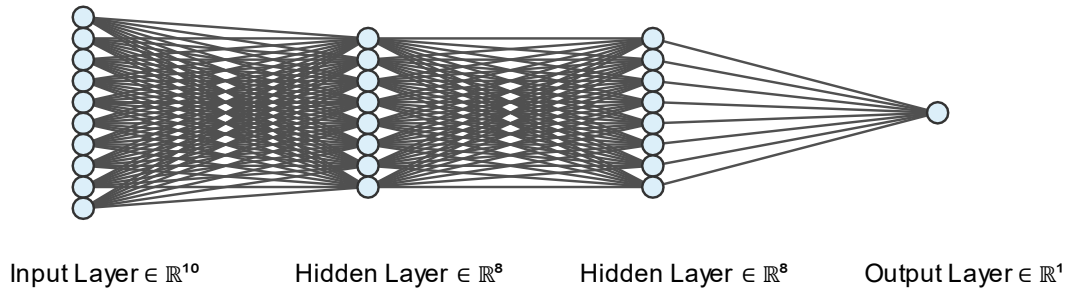


Figure 4.1. [10, 8, 8, 1] MLP Network Architecture

The [10, 8, 8, 1] architecture has 4 layers, the input layer, and 3 layers of neurons. It has 10 inputs, two hidden layers follow up with 8 neurons each, and an output (Figure 4.1). The total number of network weights is:

$$weights = (10 \times 8) + (8 \times 8) + (8 \times 1) = 152$$

If we ignore the input layer, the total of neurons in the rest layers is:

$$preceptrons = 8 + 8 + 1 = 17$$

Since each perceptron has a bias then:

$$biases = perceptrons = 8 + 8 + 1 = 17$$

Therefore, in total the network's learnable parameters are:

$$learnable\ parameters = weights + biases = 152 + 17 = 169$$

4.2.2.2 [10, 20, 20, 1] MLP

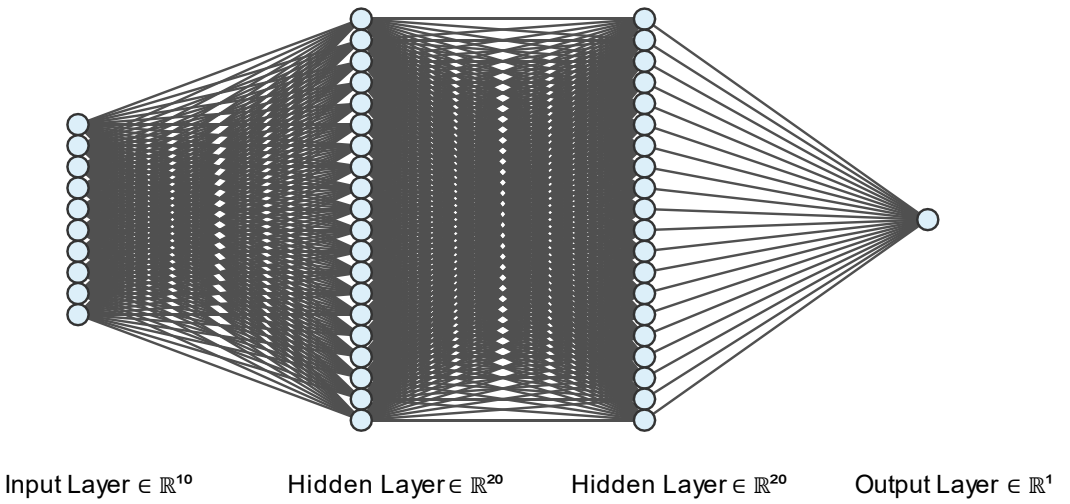


Figure 4.2. [10, 20, 20, 1] MLP Network Architecture

The [10, 20, 20, 1] architecture has 4 layers as well, an input layer and 3 layers with neurons. Similar to other MLP networks of this thesis the input layer has 10 inputs, while two hidden layers follow up with 20 neurons each, and an output (Figure 4.2). The network's weights in total are:

$$weights = (10 \times 20) + (20 \times 20) + (20 \times 1) = 620$$

By ignoring the input layer, the rest of the neurons of the network are in total:

$$preceptrons = 20 + 20 + 1 = 41$$

Since each perceptron has a bias then:

$$biases = perceptrons = 20 + 20 + 1 = 41$$

Therefore, in total the network's learnable parameters are:

$$learnable\ parameters = weights + biases = 620 + 41 = 661$$

This architecture has almost 4 times more learnable parameters than the network architecture [10, 8, 8, 1] (Section 4.2.2.1).

4.2.2.3 [10, 30, 1] MLP

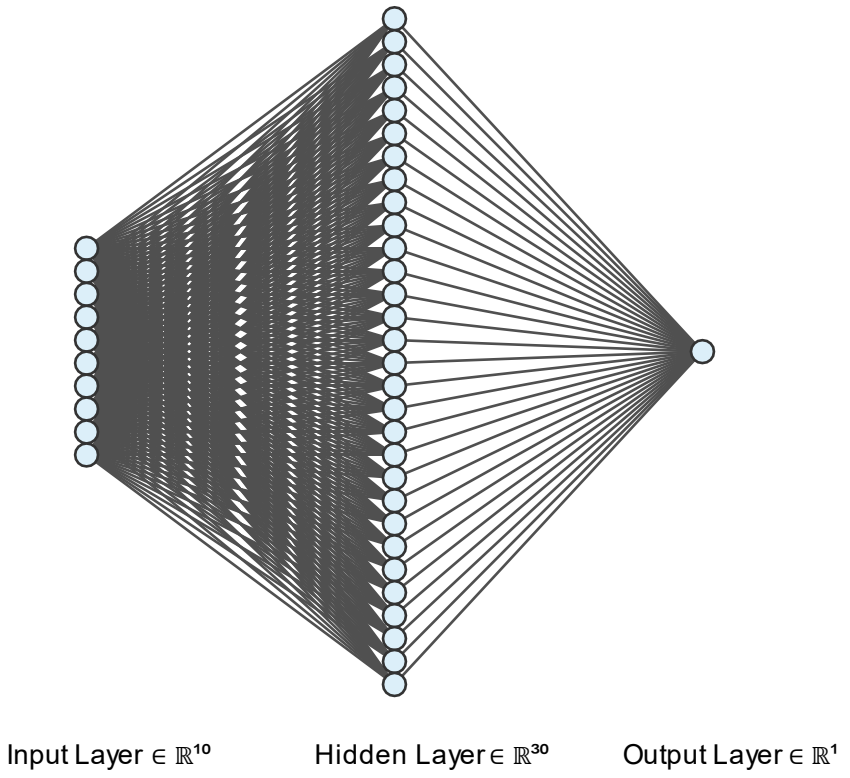


Figure 4.3. [10, 30, 1] MLP Network Architecture

The [10, 30, 1] architecture has 3 layers, an input layer, and 2 layers with neurons. It has 10 inputs, a single hidden layer with 30 neurons, and an output (Figure 4.3). The network's weights in total are:

$$weights = (10 \times 30) + (30 \times 1) = 330$$

By ignoring the input layer, the rest of the neurons of the network are in total:

$$preceptrons = 30 + 1 = 31$$

Since each perceptron has a bias then:

$$biases = perceptrons = 30 + 1 = 31$$

Therefore, in total the network's learnable parameters are:

$$learnable\ parameters = weights + biases = 330 + 31 = 361$$

Consequently, this architecture has approximately 2 times more learnable parameters than the [10. 8. 8. 1] network architecture.

4.2.2.4 [10, 100, 1] MLP

The [10, 100, 1] architecture has 3 layers, an input layer, and 2 layers with neurons. It has 10 inputs, a single hidden layer with 100 neurons, and an output (Figure 4.4). The network's weights in total are:

$$weights = (10 \times 100) + (100 \times 1) = 1100$$

By ignoring the input layer, the rest of the neurons of the network are in total:

$$preceptrons = 100 + 1 = 101$$

Since each perceptron has a bias then:

$$biases = perceptrons = 100 + 1 = 101$$

Therefore, in total the network's learnable parameters are:

$$learnable\ parameters = weights + biases = 1100 + 101 = 1201$$

Consequently, the [10, 100, 1] architecture has approximately 8 times more learnable parameters than the [10, 8, 8, 1] network architecture.

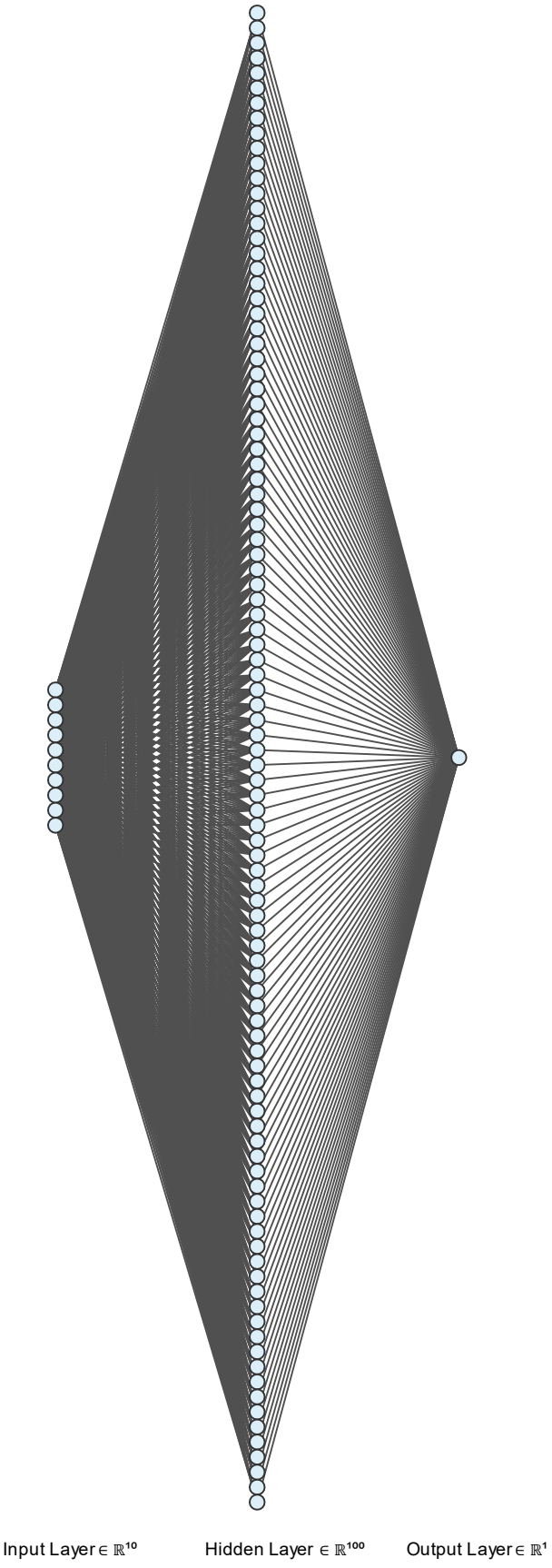


Figure 4.4. [10, 100, 1] MLP Network Architecture

4.3 CNN Implementations

The implementations for all CNN architectures are based on the paper by Wang et al. (2020). A few changes had to be made to support 3D CNNs since initially, the algorithm supported only 2D CNNs. The implementations are based on the TensorFlow 2 library, more specifically, the tensorflow.compat.v1 module was used.

This implementation came by default with 6 different network configurations CNN_4layers, CNN_7layers, VGG11, VGG13, VGG16, and VGG19. In this thesis, only the CNN_4layers, CNN_7layers, and VGG19 from the original configurations were used. Keep in mind that a CNN_4layers network configuration refers to a network with three Convolutional Layers and a single dense layer. The naming of the different network configurations was kept the same as Wang et al. (2020) originally had defined them.

Nevertheless, additionally, CNN_3layers and CNN_5layers were created for experiments and additional features were added in some of the CNN architecture; such as L1 and L2 Regularization (Section 2.3.13.7), Dropout (Section 2.3.13.9), Spatial Dropout (Section 2.3.13.10), a second layer in the FFNN, Batch Normalization (Section 2.3.13.11), etc.; and different activation functions like Sigmoid (Section 2.2.12), ReLU (Section 2.2.13), and SoftMax (Section 2.2.14).

For all the experiments, the stride for Max-Pooling is set to None, which means is the same as the pool_size. For example, for Max-Pooling 2D, with pool_size = (2, 2) the stride would be (2, 2) as well, while for Max-Pooling 3D with pool_size = (2, 2, 2) the stride would be (2, 2, 2).

All the Convolutional Layers apply padding (padding='SAME') (Section 2.3.12.7) after the Convolution which implies that the size of the output features maps is the same as the input feature maps. This means that after applying the filters, a perimeter of 0s is added. On the other hand, if no padding was applied, then the size of the output feature maps of an input image with dimensions 32×32 with filters 3×3, would be 30×30 instead of 32×32.

The CNN implementations of this thesis have two output neurons for the AD/NC problem, and three for the [AD, MCI, NC] problem. Because. the target classes from AD = 0 and NC = 1 have been converted to categorical, a binary class matrix, AD = [1, 0] and

NC = [0, 1], that is why the 2 outputs neurons are needed. Simillarly, for the [AD, MCI, NC] problem the classes from AD = 0, MCI = 1, and NC = 2, have been converted to AD = [1, 0, 0], MCI = [0, 1, 0] and NC = [0, 0, 1].

4.3.1 Wang et al.'s (2020) CNN

4.3.1.1 Implementation

The CNN implementations for this thesis are based on Wang et al.'s (2020) implementation. More specifically, in their study, they provide a MATLAB implementation, but later on, they published a Python version of the same code on their GitHub repository (https://github.com/cjlin1/simpleNN). This thesis modified that Python implementation to make the models suitable for our case. For example, the initial code supported only 2D datasets for 2D CNNs, so, modifications had to be made to support 3D CNNs (Section 4.3.4), and our 3D datasets (Section 3.2.5 – 3.2.7).

The modified code used for the CNNs can be found in Appendix B. To be more precise, train.py (Code Snippet B.1) represents the main; it reads the user's arguments, creates the TensorFlow session, and contains also the SGD/Adam trainer. The newton_cg.py (Code Snippet B.2) is the NewtonCG trainer, the heart of the HFO implementation for the CNN model. The utilities.py (Code Snippet B.3), contains some utility functions that the rest of the files needs, and is responsible for collecting and saving the statistics. The predict.py (Code Snippet B.4) executes a forward pass of the network. The net.py (Code Snippet B.5 – B.18) are different network configurations for 2D and 3D CNNs, that were used during this thesis. More details about those implementations can be found in Sections 4.3.2 & 4.3.4. Also, the vgg.py (Code Snippet B.19) contains the implementation for the VGG networks, inspired by Simonyan and Zisserman (2014).

4.3.1.2 Hyperparameters

In Wang et al.'s (2020) implementation, several command-line arguments could be passed by the user, which modifies the hyperparameters of the network. The way these arguments are passed can be found in Appendix E.6.

4.3.1.2.1 Optimizers

Three optimizers were available in Wang et al.'s (2020) implementation that could be defined in the hyperparameter "optim"; the SGD (Section 2.4.3), Adam (Section 2.4.4),

and NewtonCG (HFO) (Section 2.4.7). The NewtonCG implementation is based on Martens (2010) and Martens and Sutskever (2012). For the experiments on CNN networks, only Adam and NewtonCG were used.

4.3.1.2.2 Epochs – Newton's Iterations

The hyperparameter 'iter_max' is the epochs, which represent the maximal number of Newton iterations for the NewtonCG optimizer. For the Adam and SGD optimizers, the epochs can be set through the 'epoch_max' argument.

4.3.1.2.3 Loss Function

The two different loss functions that the algorithm supports, which can be defined as the "loss" hyperparameter, are the MSE (Section 2.3.4) and Cross-entropy (Section 2.3.5). Most experiments of this thesis use MSE loss and only a single experiment the Cross-entropy loss since its performance was not desirable.

4.3.1.2.4 Batch Size "bsize"

Wang et al.'s (2020) implementation support batch splitting, where it splits the data into batches of size 'bsize' so that each segment can fit into memory. The batch size is a hyperparameter that defines how many training samples are utilized in one iteration. An epoch therefore can have multiple iterations based on the batch size. For batch size equal to one, the learning algorithm is called Stochastic Gradient Descent (SGD).

When the batch size is equal to the size of the training set, the algorithm is called Batch Gradient Descent. On the other hand, when the batch size is greater than one but smaller than the size of the training set is called Mini-Batch Gradient Descent. In the following experiments (Section 5.3 – 5.11), Mini-Batch Gradient Descent was used.

4.3.1.2.5 Weight Decay "C"

For regularization in the loss function, the regularization term C (Section 2.3.13.8), or so-called weight decay will be used. Different values of weight decay will be tested such as C = {0.01, 0.1, 1, 10}.

$$weight\ decay = \frac{learning\ rate}{C \times num\_of\_samples}$$

Weight decay plays a crucial role in the bias-variance tradeoff. More specifically, as we can see in Figure 4.5, having high bias causes underfitting, while having high variance causes overfitting.

To solve the problem of high variance, we can reduce the number of parameters, to prevent our model from getting too complex since it will remove some non-linearities. The problem is that we need those non-linearities to solve more complex problems. That is why we need weight decay to penalize complexity during weight updates where we subtract constant times the weight from the original weight.

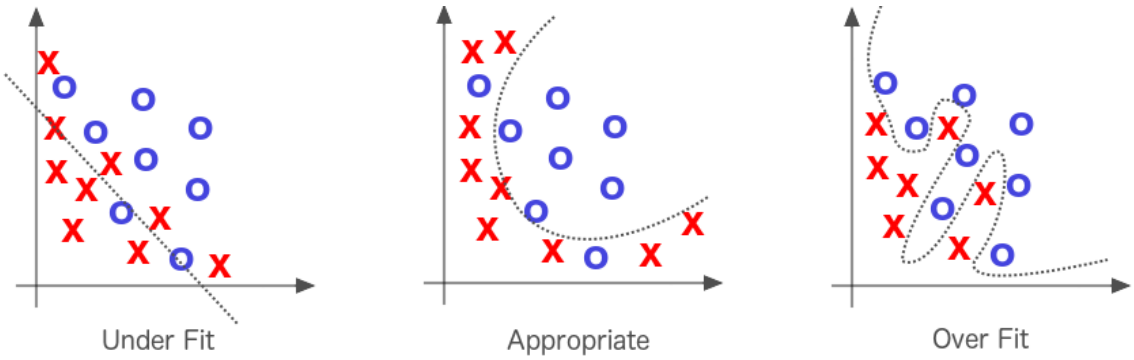


Figure 4.5. Bias-Variance Tradeoff. **1st:** High Bias – Underfitting. **2nd:** Appropriate. **3rd**: Hiag Variance – Overfitting

It has to be mentioned that Wang et al. (2020) in their study have implemented the code for the NewtonCG algorithm in MATLAB. In their source code for Python, which was released afterward in their GitHub repository (https://github.com/cjlin1/simpleNN), say that they are not sure if the implementation of the weight decay (C) was done correctly; since it was done in such a way that the Python and MATLAB codes are the same. I suggest in future experiments the code for MATLAB be used to check whether the implementation in Python affects the performance due to some error.

4.3.1.2.6 Adam's Hyperparameters

Unique to the Adam optimizer; the parameter learning rate can be defined by using the 'lr' argument. By default, the following hyperparameters are initialized as beta1 = 0.9, beta2 = 0.999, and epsilon (ε) = 1e-8. The beta1 and beta2 correspond to the decay rate for the 1st- and 2nd-moment estimates respectively (Kingma and Ba, 2014). The ε is a small constant for numerical stability which is referred to as the 'epsilon hat' in the Kingma and Ba (2014) paper.

4.3.1.2.7 NewtonCG's Hyperparameters

The following hyperparameters are unique to the NewtonCG optimizer. The 'GNsize' is the number of samples for estimating the Gauss-Newton matrix. Based on Wang et al. (2020), the larger the GNsize is, the more time the training process takes but the better the performance of the algorithm was as well on their problem (CIFAR-10 dataset).

The Levenberg-Marquardt (LM) algorithm, adaptively varies the parameter updates between the gradient descent update and the Gauss-Newton update. Based on Gavin (2019), the small parameters of the damping parameter 'lambda' (λ), result in a Gauss-Newton update, and large values of λ result in a gradient descent update. The lambda is initially set to a large value, lambda = 1 so that the first updates are small steps in the steepest-descent direction. More details about the usage of the reasons behind using the LM method can be found in Wang et al. (2020).

Additionally, other than the lambda, our implementation uses two arguments for the LM method, the 'drop', and 'boost'. The drop and boost constants, reduce or increase the lambda variable respectively based on the result of the approximation. If an iteration happens to result in a worse approximation, the λ is increased (boost), otherwise the solution improves and moves towards a local minimum, so the λ is decreased (drop). The default values for the two hyperparameters are boost = $\frac{3}{2}$ and drop = $\frac{2}{3}$.

An additional hyperparameter of the NewtonCG optimizer is 'xi' (ξ) which is the tolerance in the relative stopping condition for CG. Its default value is set to xi = 0.1. Another hyperparameter that can be set for the NewtonCG method is the 'eta' (η) which is the parameter for line search stopping condition. The η is a predefined constant between 0 and 1, which is initially set to the value η = 0.0001.

The last hyperparameter which can be modified through the passing arguments is 'CGmax' which is basically the maximal number of CG iterations and is initially set to the value CGmax = 250. It is the same argument as 'CGiter' in Section 4.2.1.3 of the MLP with HFO implementations.

4.3.2 2D CNN Network Architectures

4.3.2.1 Types of 2D CNNs

Five different architecture were used for the experiments of 2D datasets, of this thesis; the 3-, 4-, 5-, 7-, and 19-layer network configurations. All models of 2D CNN receive as input a single 2D slice image of a preprocessed T1-weighted MRI brain scan (Section 3.2.3.1) with dimensions 174×174.

More experiments in the thesis were performed with 2D CNN network configurations since they require much less processing power and less execution time to complete their

training than the 3D CNNs (Section 4.3.4). The source code for these network architectures can be found in Appendix B. By changing the net.py (Appendix B5) or vgg.py (Appendix B6) file we were able to modify the network architecture that the algorithm would be trained on.

For the creation of the figures with the different network configurations in Section 4.3.2, the tool (http://alexlenail.me/NN-SVG/AlexNet.html) was used in combination with Photoshop CC.

4.3.2.2 3-layer 2D CNN

Two main network configurations were used for 2D CNNs with 3 layers; the 2D_CNN_3L_1 (Shallow & Wide Network) (Section 4.3.2.2.1), and the 2D_CNN_3L_2 (Shallow & Narrow Network) (Section 4.3.2.2.2). I refer to both of them as shallow networks since they only have 3 layers, while in other experiments mostly are 4 layers. Both network configurations have 2 Convolutional Layers and a single dense layer. All the 3-layer CNN implementations use 3×3 filters on each Convolutional Layer. The input of the dense layer, is the output of the last Convolutional Layer, flatten. Both networks apply 2×2 Max-Pooling with stride = 2, and for activation they use ReLU.

Table 4.2 compares two 3-layer networks with the most frequently used network configuration in this thesis, the 2D_CNN_4L_1 (Section 4.3.2.3.1). The f_i is the number of feature maps in the ith Convolutional Layer. The 2D_CNN_3L_1 (Shallow & Wide Network) has f1 = 64, and f2 = 128, hence is a "wide" network, while the 2D_CNN_3L_3 (Shallow & Wide Network), has f1 = 16, and f2 = 32, hence is a "narrow" network. The purpose of these shallow network configurations was to examine whether a shallower network could help to resolve the issue of overfitting where the 4-layer CNN causes.

Network Architecture	**Feature Maps**		
	f1	***f2***	***f3***
2D_CNN_4L_1 *(Baseline)*	32	32	64
2D_CNN_3L_1 *(Shallow & Wide Network)*	64	128	-
2D_CNN_3L_2 *(Shallow & Narrow Network)*	16	32	-

Table 4.2. The number of feature maps fi in the ith Convolutional Layer. 4-layer 2D CNN (Baseline) vs. 3-layer 2D CNNs

4.3.2.2.1 2D_CNN_3L_1

The 2D_CNN_3L_1 or so-called “Shallow & Wide” network configuration (Figure 4.6) in the first Convolutional Layer has f1 = 64 filters, while the second Convolutional Layer has f2 = 128 filters. In total 2,191,368 bytes (~2MB) of variables are needed for this network configuration. The purpose of this network was to test whether a wide network will help the model to detect more useful features from the 2D slice MRI scans and therefore improve its performance during the classification of paints in the AD/NC problem.

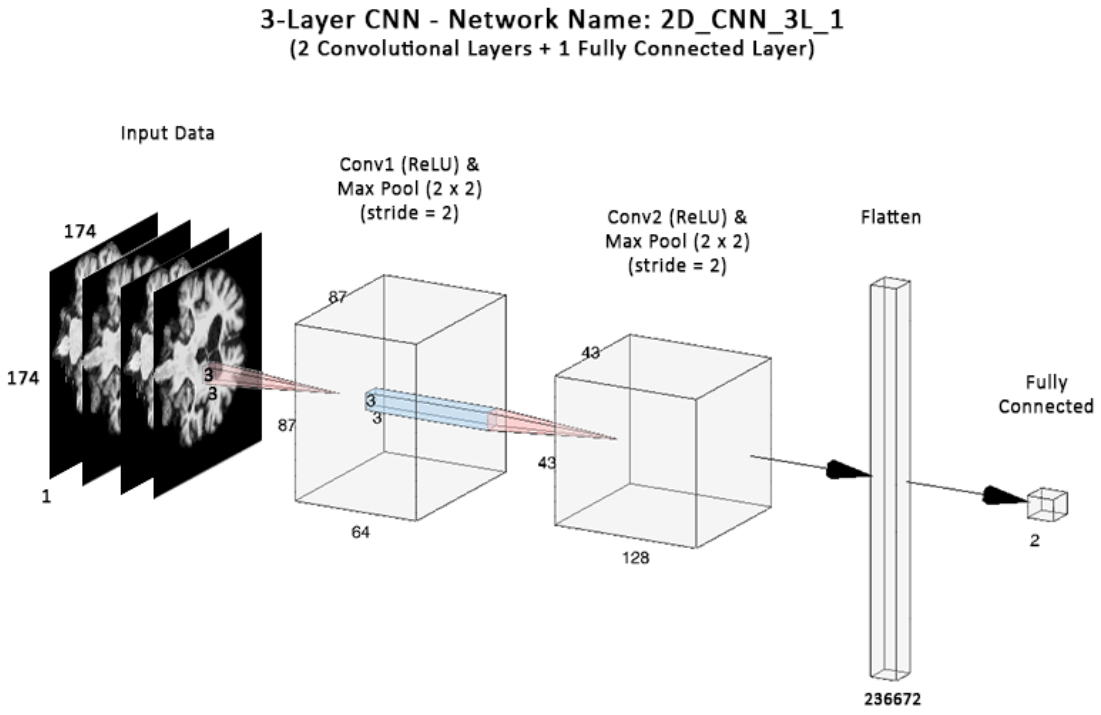


Figure 4.6. Network Configuration: 2D_CNN_3L_1. Shallow & Wide network. 2 Convolutional Layers and a single dense layer. **Feature Maps:** f1 = 64, f2 = 128

4.3.2.2.2 2D_CNN_3L_2

The 2D_CNN_3L_2 or so-called “Shallow & Narrow” network configuration (Figure 4.7) in the first Convolutional Layer has f1 = 16 filters, while the second Convolutional Layer has f2 = 32 filters. In total 492,552 bytes (~0.5MB) of variables are needed for this network configuration. The purpose of this network was to test whether fewer neurons, therefore fewer feature detectors, could help the model overcome its overfitting issues and increase its generalization ability.

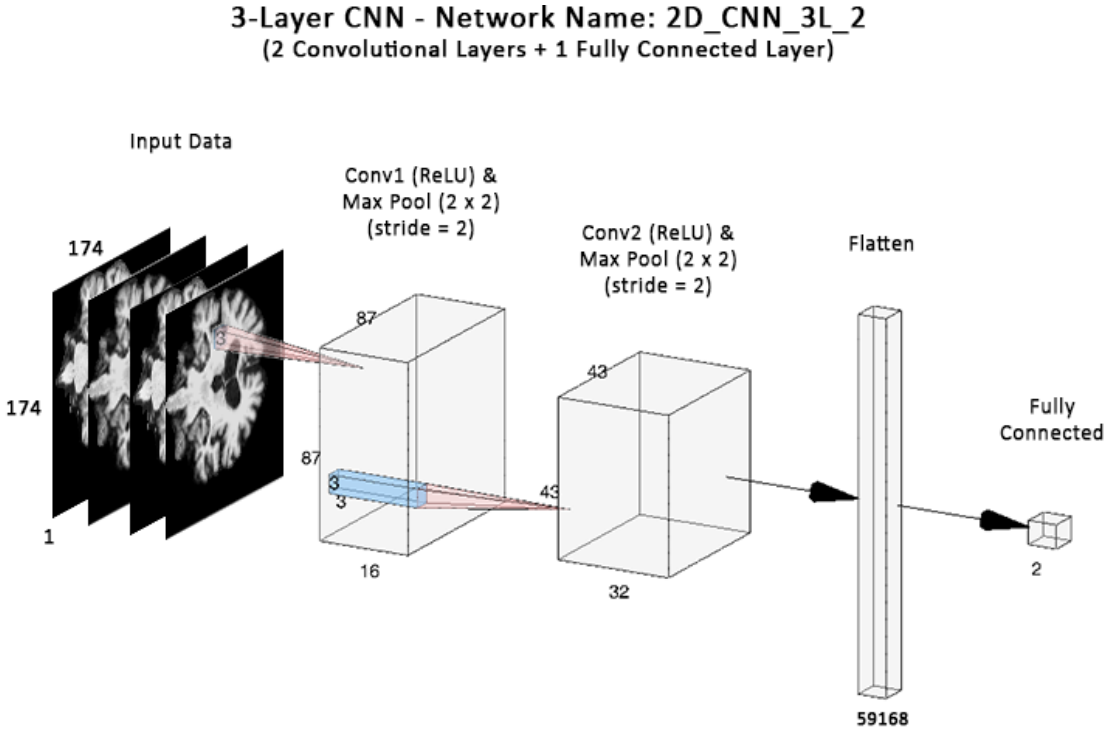


Figure 4.7. Network Configuration: 2D_CNN_3L_2. Shallow & Narrow network. 2 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 16, f2 = 32

4.3.2.3 4-layer 2D CNN

Six main 4-layer 2D CNN network configurations were used in this thesis, with the names 2D_CNN_4L_ {1 – 6}. All six of them have 3 Convolutional Layers and a single dense layer. Most of the 2D experiments in this thesis were conducted with 4-layer CNNs.

4.3.2.3.1 2D_CNN_4L_1 (Baseline)

The 2D_CNN_4L_1 network configuration has f1 = 32 filters in the first Convolutional Layer (Conv1), the second Convolutional Layer (Conv2) has also f2 = 32 filters, while the last Convolutional Layer (Conv3) has f3 = 64 filters. 2D Max-Pooling with pool_size = (2, 2) was applied to each Convolutional Layer as well. In total 338,056 bytes (~0.3MB) of variables are needed for this network configuration. For the specific network filters of size, 3×3 were used.

Most experiments in this thesis use the 2D_CNN_4L_1 (Figure 4.8) network configuration, including the experiment B_2D_M_N10 (Appendix H.2.10) which had one of the best validation accuracies for the AD/NC problem, equal to 80%. In the experiment B_2D_M_N18 (Appendix H.2.17), a similar network configuration to the 2D_CNN_4L_1 was used with an additional component, the batch normalization layer

between the Conv3 layer and the Flatten layer. The batch normalization increases the total size of the network's variables to 338,568 bytes.

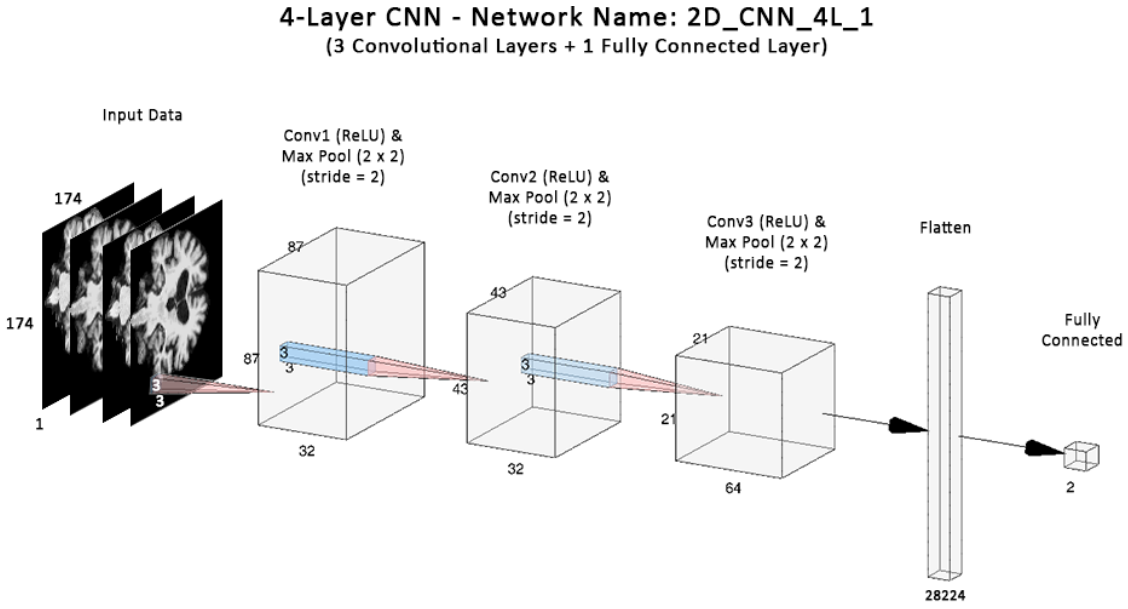


Figure 4.8. Network Configuration: 2D_CNN_4L_1. 3 Convolutional Layers and a single dense layer.
Feature Maps: f1= 32, f2 = 32, f3 = 64, and filter size = [3, 3]

4.3.2.3.2 2D_CNN_4L_2

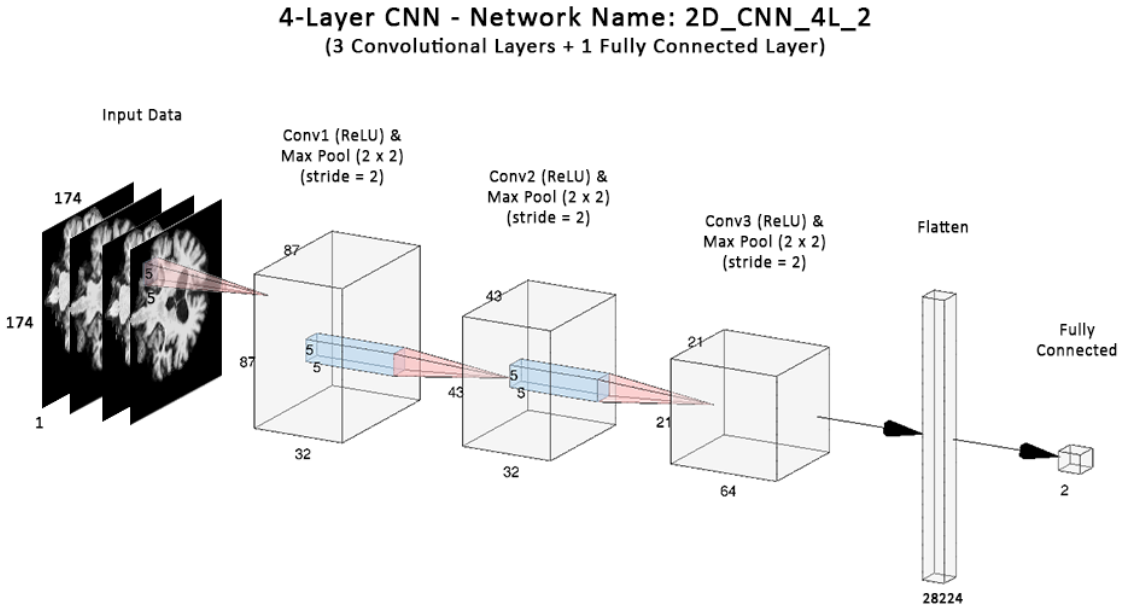


Figure 4.9. Network Configuration: 2D_CNN_4L_2. 3 Convolutional Layers and a single dense layer.
Feature Maps: f1= 32, f2 = 32, f3 = 64, and filter size = [5, 5]

The 2D_CNN_4L_2 (Figure 4.9) network configuration has the same number of layers and number of feature maps as the 2D_CNN_4L_1 (Section 4.3.2.3.1). The only difference is the size of filters for each Convolutional Layer which are 5×5 instead of

3 × 3. The size of the total variables is affected as well which increases now to 536,712 bytes (~0.5MB), while for the 2D_CNN_4L_1 was ~0.3MB. This network is the one used in experiment B_2D_M_N19 (Appendix H.2.18), which yielded the highest validation accuracy equal to 81% for the AD/NC problem between all the CNN models.

4.3.2.3.3 2D_CNN_4L_3

The 2D_CNN_4L_3 (Figure 4.10) network configuration has the same number of layers and number of feature maps as the 2D_CNN_4L_1 (Section 4.3.2.3.1) as well. Again, the only difference is the size of filters for each Convolutional Layer which are 7 × 7 instead of 3 × 3. The size of the total variables is affected as well which increases now to 834,696 bytes (~0.8MB), while for the 2D_CNN_4L_1 was ~0.3MB.

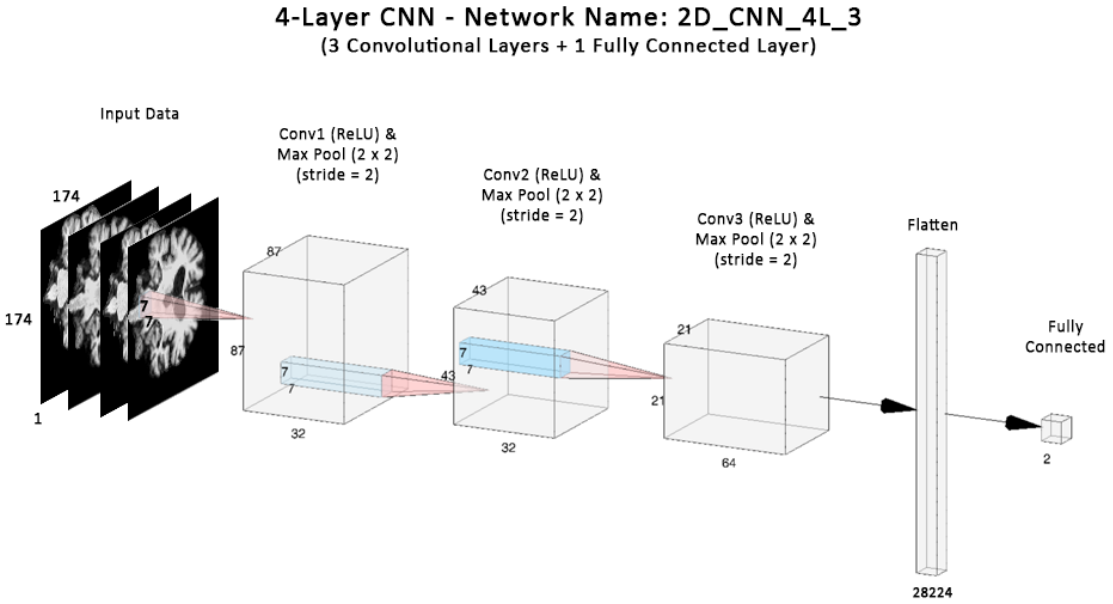


Figure 4.10. Network Configuration: 2D_CNN_4L_3. 3 Convolutional Layers and a single dense layer.
Feature Maps: f1= 32, f2 = 32, f3 = 64, and filter size = [7, 7]

4.3.2.3.4 2D_CNN_4L_4

The difference between the 2D_CNN_4L_4 (Figure 4.11) network configuration and the 2D_CNN_4L_1 (Section 4.3.2.3.1), is the absence of the Max-Pooling. This implies a significant increase in the size of the total variables, from ~0.3MB for the 2D_CNN_4L_1 to 15,613,576 bytes (~15MB), which is approximately 50 times more variables.

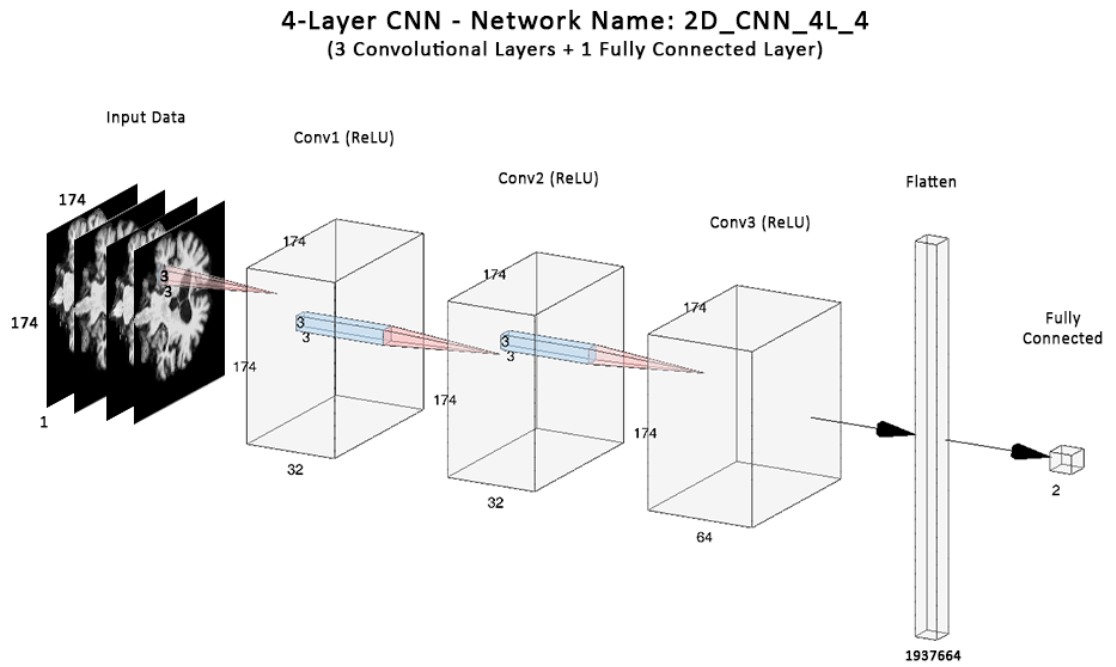


Figure 4.11. Network Configuration: 2D_CNN_4L_4. No Max-Pooling, 3 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 32, f2 = 32, f3 = 64, and filter size = [3, 3]

4.3.2.3.5 2D_CNN_4L_5

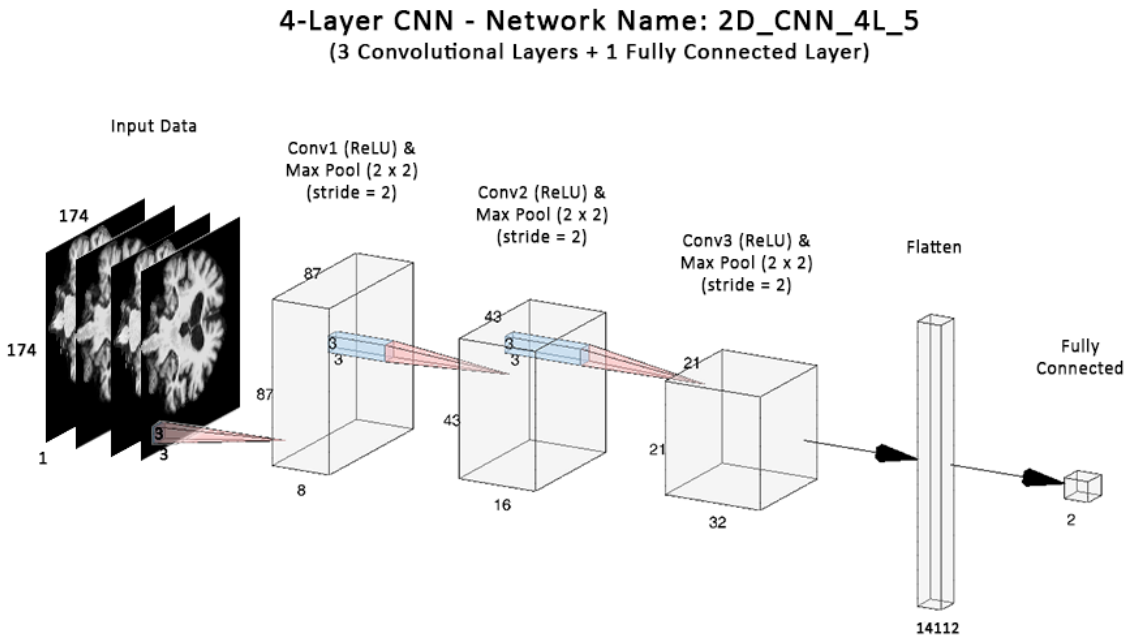


Figure 4.12. Network Configuration: 2D_CNN_4L_5. 3 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 8, f2 = 16, f3 = 32, and filter size = [3, 3]

The 2D_CNN_4L_5 (Figure 4.12) network configuration, is a narrow version of a 4-layer CNN. This model was created to examine whether a narrow network could help to encounter overfitting, which the 2D_CNN_4L_1 (Section 4.3.2.3.2) suffered from; even after applying Dropout (Section 2.3.13.9), L1 & L2 Regularization (Section 2.3.13.7),

and other regularization techniques (Section 2.3.13). The network consists of a single dense layer at the end and three Convolutional Layers with a number of feature maps f_i in the i^{th} layer equal to f1 = 8, f2 = 16, and f3 = 32. This implies significantly less space needed for the variables of this network; from ~0.3MB in the 2D_CNN_4L_1 implementation to 136,456 bytes (~0.1MB).

4.3.2.3.6 2D_CNN_4L_6

The 2D_CNN_4L_6 (Figure 4.13) network configuration is a wider version of a 4-layer CNN. This model was created to examine whether a wider network could help improve accuracy, by potentially encountering underfitting, since more filters would be available, and therefore more feature maps would be created. The filters for the three Convolutional Layers are f1 = 8, f2 = 16, and f3 = 32. In the end, a single dense layer is added. This implies some increase in the total number of variables for the network; from ~0.3MB in the 2D_CNN_4L_1 (Section 4.3.2.3.2) implementation to 822,280 bytes (~0.8MB).

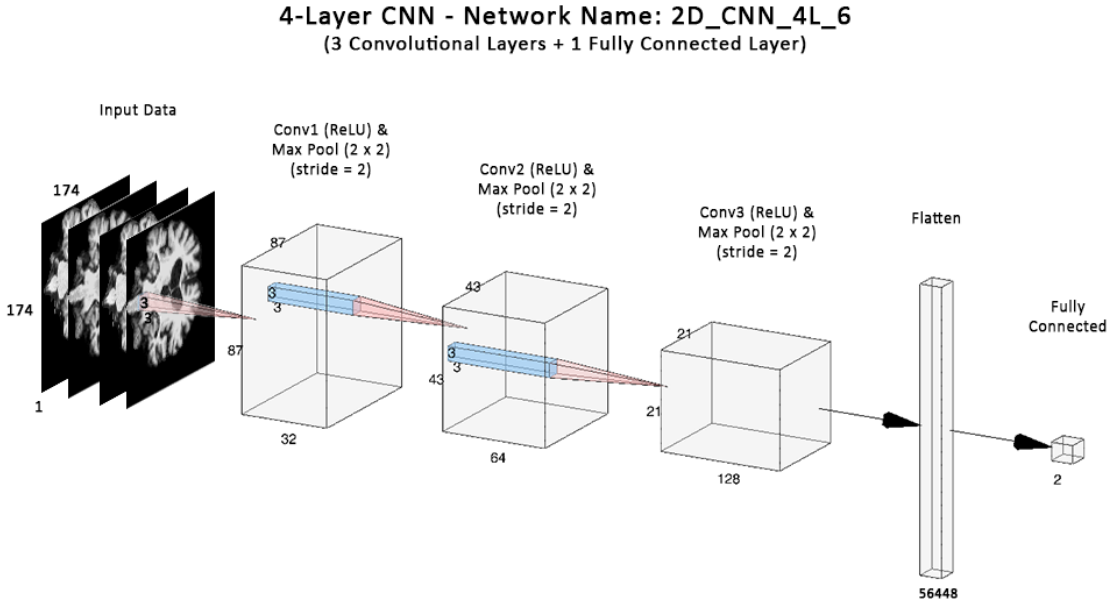


Figure 4.13. Network Configuration: 2D_CNN_4L_6. 3 Convolutional Layers and a single dense layer.
Feature Maps: f1= 32, f2 = 64, f3 = 128, and filter size = [3, 3]

4.3.2.4 5-layer 2D CNN

4.3.2.4.1 2D_CNN_5L_1

The 2D_CNN_5L_1 (Figure 4.14) network configuration is a deeper version of a 4-layer CNN but in terms of the fully connected layers, not the Convolutional ones. More specifically, the 3 Convolutional Layers are the same as the 2D_CNN_4L_1 (Section

4.3.2.3.1) model and the only difference is that instead of one, two fully connected layers exist. The filter size is 3×3, the same as in 2D_CNN_4L_1.

After the flattening, instead of going directly to the two perceptrons of the output layer, a layer in between exists with 256 perceptrons. The idea was to examine whether from 28,224 inputs going directly to 2 output neurons was too much. Now, an in-between layer exists so from the 28,224 inputs we are going to a hidden layer with 256 neurons and finally to the 2 output neurons.

We know that a single layer of perceptron (Section 2.3.11.2) cannot solve non-linearly separable problems (Section 2.2.10). Consequently, I assumed that the output of the Convolutional Layers might not be linearly separable, so a hidden layer may be needed to improve the model's performance. This could mean that trying to perform classification with hyperplanes (1-layer FFNN) for the AD/NC problem maybe was not enough, therefore convex regions (2-layer FFNN) should be used. The increase of the number of hidden layers, results in a drastic increase to the total number of variables as well, from ~0.3MB in the 2D_CNN_4L_1 to 29,016,712 bytes (~29MB).

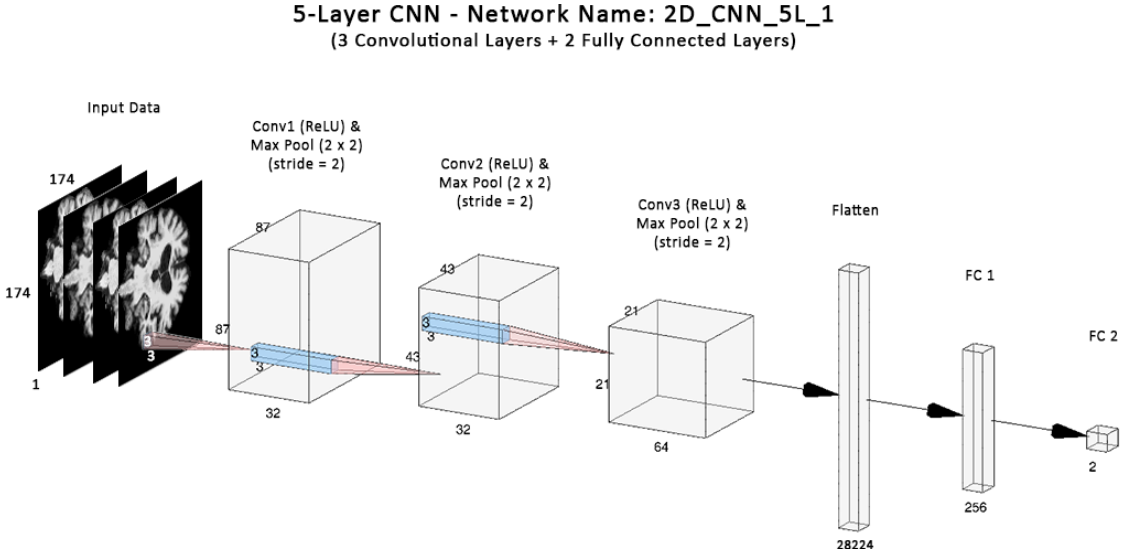


Figure 4.14. Network Configuration: 2D_CNN_5L_1. 3 Convolutional Layers and 2 dense layers.
Number of neurons: FC1 = 256, FC2 = 2. **Feature Maps:** f1= 32, f2 = 32, f3 = 64, and filter size = [3, 3]

4.3.2.5 7-layer 2D CNN

4.3.2.5.1 2D_CNN_7L_1

The 2D_CNN_7L_1 (Figure 4.15) which is a 7-layer network configuration was meant to examine whether a deeper and narrower than the 2D_CNN_4L_1 (Section 4.3.2.3.1), could help to reduce its overfitting issues.

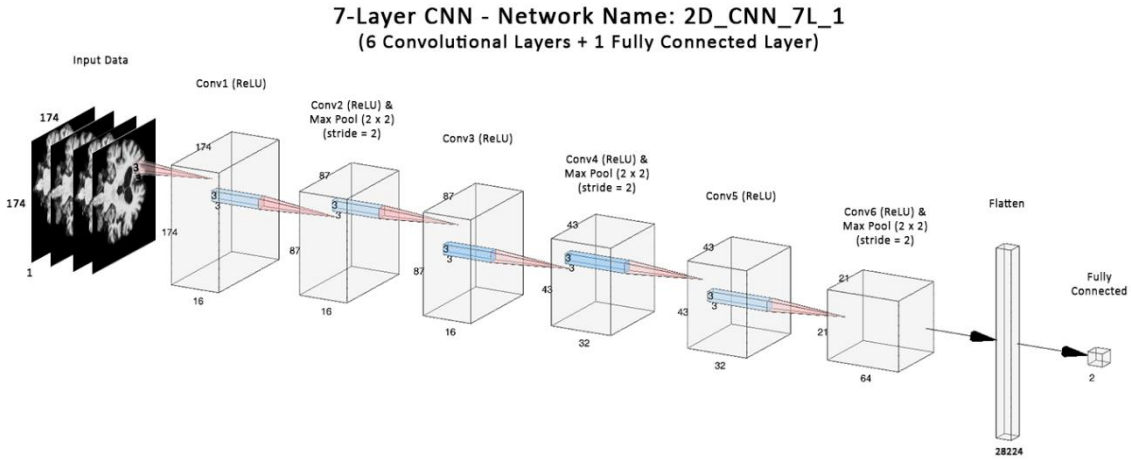


Figure 4.15. Network Configuration: 2D_CNN_7L_1. 6 Convolutional Layers and a single dense layer.
Feature Maps: f1= 16, f2 = 16, f3 = 16, f4 = 32, f5 = 32, f6 = 64, and filter size = [3, 3]

The network has 6 Convolutional Layers and a single fully connected layer. The filter size for all Convolutional Layers is 3×3. Not all Convolutional Layers have Max-Pooling though, only the even index layers (Conv2, Conv4, Conv6) apply Max-Pooling with pool_size = (2, 2) and stride = (2, 2). The filters for each Convolutional Layer are: f1 = 16, f2 = 16, f3 = 16, f4 = 32, f5 = 32, f6 = 64.

The total size of variables is 834,696 bytes (~0.8MB), larger than the 2D_CNN_4L_1 network which is ~0.3MB. This happens because 3 extra Convolutional Layers exist and not all of them apply Max-Pooling.

4.3.2.6 19-layer 2D CNN

4.3.2.6.1 Problems with VGG19

19-layer CNN (VGG19) network configuration is meant to examine whether a much deeper and narrower network topology could affect, and potentially improve the performance of the experiments. Wide networks are very good at memorization but not so good at generalization, therefore overfitting may occur. Subsequently, deeper networks can be used to capture the natural "hierarchy" that is present everywhere in nature. The reason behind the boost in performance from a deeper network is that a more complex, non-linear function can be learned. For example, the first layers could capture simple features of the image such as lines and curves, while higher layer complex features such as the whole hippocampal structure. Supposedly, more Convolutional Layers should help the model's performance to increase Despite that, too deep CNNs are very computationally expensive to be trained (Krizhevsky et al. 2012; Bengio et al. 2013;

Simonyan and Zisserman, 2014; Eldan and Shamir, 2016). Later on, during the experimentation, this assumption will be proven wrong in our experiments since the performance did not improve with a deeper network configuration for the AD/NC problem.

4.3.2.6.2 2D_VGG19_1

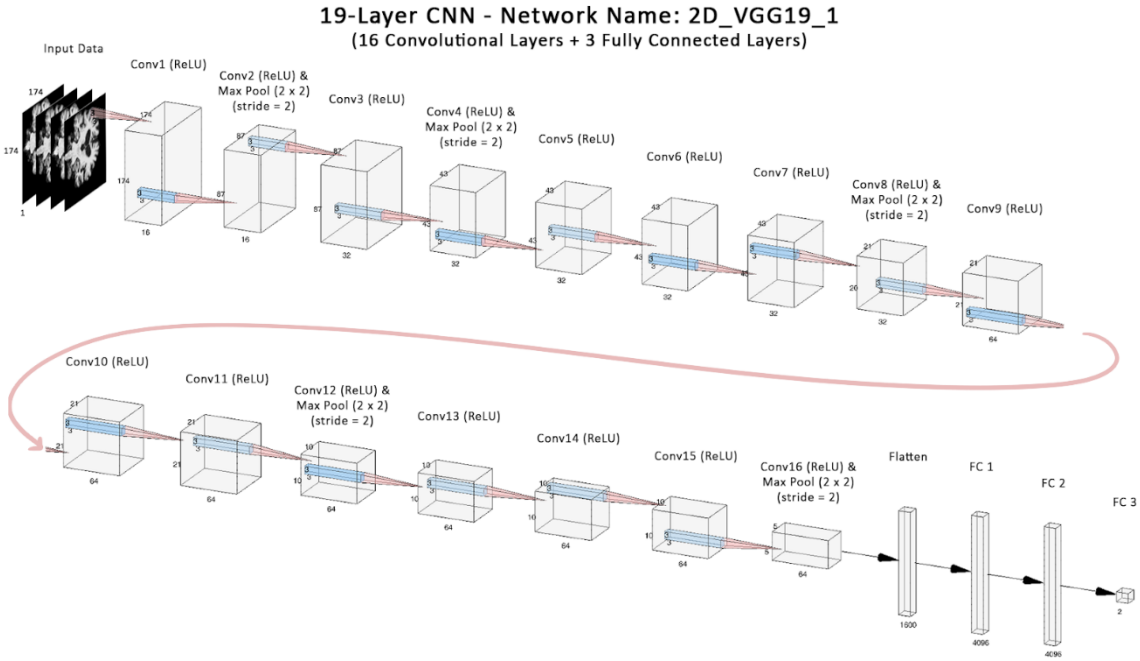


Figure 4.16. Network Configuration: VGG19 2D CNN. 16 Convolutional Layers and 3 dense layers.
{M = Max-Pooling (2, 2), D = Dropout, N is the # of feature maps. FC1 = FC2 = 4096, FC3 = 2; filter size = [3, 3]}
Configuration = [16, 16, M, D, 32, 32, M, D, 32, 32, 32, 32, M, D, 64, 64, 64, 64, M, D, 64, 64, 64, 64, M, D, FC1, FC2, FC3]

The 2D_VGG19_1 (Figure 4.16) network architecture has in total 19 layers, 16 Convolutional Layers, and 3 fully connected layers. This network is a slight modification of Simonyan and Zisserman's (2014) VGG19 network, or so-called in their study the "D ConvNet Configuration". FC1 and FC2 are two fully connected layers with 4096 perceptrons each. FC3 is a fully connected layer with the two output units, same as other network configurations output layer. With the letter 'M' representing a Max-Pooling layer with pool_size = (2, 2) and stride = (2,2); the 'D' representing a dropout layer with rate = 0.3 which is the fraction of input units to be dropped; and N the integer value representing a Convolutional Layer with N filters of filter size 3×3, then the following sequence represents the 2D_VGG19_1 (Figure 4.16) network configuration:

[16, 16, M, D, 32, 32, M, D, 32, 32, 32, 32, M, D, 64, 64, 64, 64, M, D, 64, 64, 64, 64, M, D, FC1, FC2, FC3]

This very deep network, increases drastically the number of total variables for the network configuration, from ~0.3MB for the 2D_CNN_4L_1 (Section 4.3.2.3.1) network to

94,710,216 bytes (~94MB). The 2D_VGG19_1 network configuration has approximately 300 times more variables than the 2D_CNN_4L_1.

4.3.3 2D CNN Networks Correctness

4.3.3.1 MNIST Dataset

I have used the MNIST handwritten digit database from (https://www.kaggle.com/c/digit-recognizer/overview) to test whether the 2D CNN implementation, both for the Adam and NewtonCG optimizers, performs as expected. The MNIST ("Modified National Institute of Standards and Technology"), is the "Hello World" dataset of Computer Vision. It was released in 1999 by LeCun Y. and Corina C., and it has been used as a benchmarking for classification algorithms.

For checking the correctness of my network, I have used 42000 samples, a subset of the dataset. The samples are grayscale images of 28×28 pixels, of the ten different digits, 0 to 9 (examples of samples in Figure 4.17). Therefore, each sample in total consists of 784 pixels, with each pixel being a value between 0 and 255, inclusive.

Figure 4.17. Labeled samples of digits, of the MNIST Dataset. Each sample's dimensions are 28 x 28 pixels, with each pixel having a value between 0 and 255, inclusive.

To split the 42000 samples into 3, relatively balanced sets, the training, validation, and testing sets, the scikit-learn library's method "sklearn.model_selection.train_test_split" in Python 3 have been used (e.g. Code Snippet 3.1). After splitting the MNIST dataset, the proportions of each class for each set can be seen in Table 4.3.

Set	***# of Samples***	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
Total	**42000**	9.8%	11.2%	9.9%	10.4%	9.7%	9.0%	9.8%	10.5%	9.7%	10.0%
Train	**30996 (~74%)**	9.9%	11.0%	10.0%	10.3%	9,6%	9.1%	9.8%	10.4%	9,8%	10.0%
Valid	**6804 (~16%)**	9.6%	11.4%	10.0%	10.3%	10.2%	9.0%	10.0%	10.4%	9.2%	9.9%
Test	**4200 (10%)**	9.5%	11.6%	9.6%	10.6%	9.4%	8.9%	9.8%	11.1%	9.7%	9.8%

Table 4.3. Training, validation, and test sets splitting of the MNIST dataset.

Code Snippet F.1 (digit-recognizer-2D.py), is the source code that was used for the creation of the '.mat' files with the training, validation, and testing sets, to be provided as input in the CNN implementation.

4.3.3.2 Adam – Digits Recognition 2D (MNIST)

The Adam (Section 2.4.4) optimizer with the 2D_CNN_4L_1 (Section 4.3.2.4.1), the 2D 4-layer CNN (3 Convolutional Layers & 1 Fully Connected Layer), performs very well with 0.99 training, validation, and testing accuracies (Table 4.4). The training process was set to 100 epochs, and the testing accuracy was calculated from a single pass of 100 epochs' best model. The validation and training accuracies are the ones in the best model during training as well. The best model is not the one at the last iteration but the model with the highest validation accuracy during training which is being automatically saved.

Optimizer – Dataset	***Model of the Best Validation Accuracy***		
	Training	Validation	Testing
Adam – MNIST (Digits Recognition 2D)	0.99	0.99	0.99

Table 4.4. Training, validation, and testing accuracies of the model with the best validation accuracy with Adam and MNIST dataset.

In Figure 4.18, we can see the training and validation accuracies, while in Figure 4.19 the training and validation losses for 100 epochs, of the model with the MNIST dataset and the Adam optimizer.

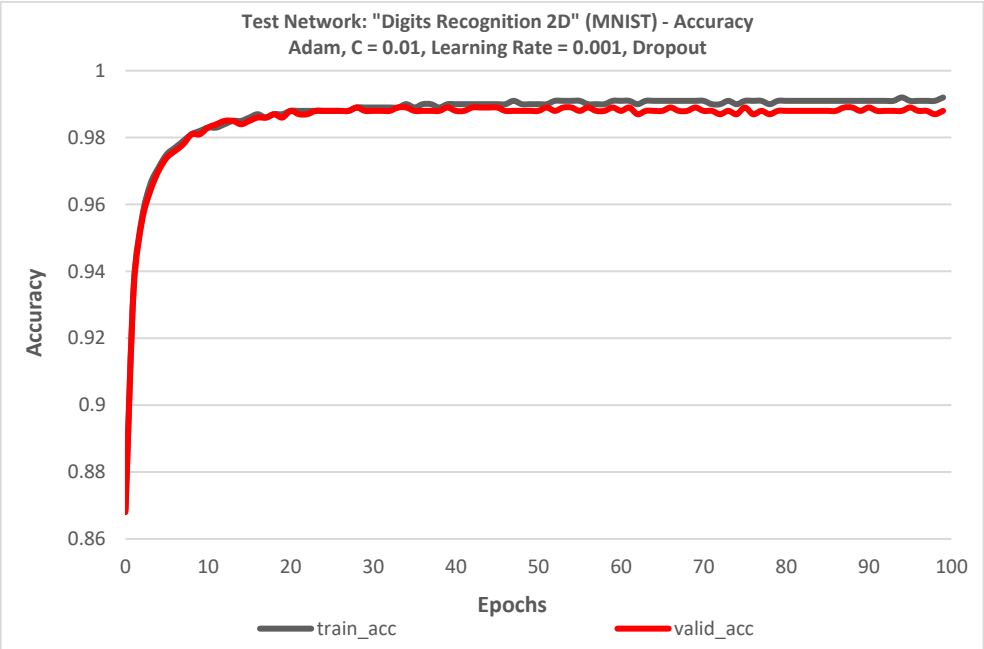


Figure 4.18. Test Network: "Digits Recognition 2D" (MNIST) with Adam - Accuracy

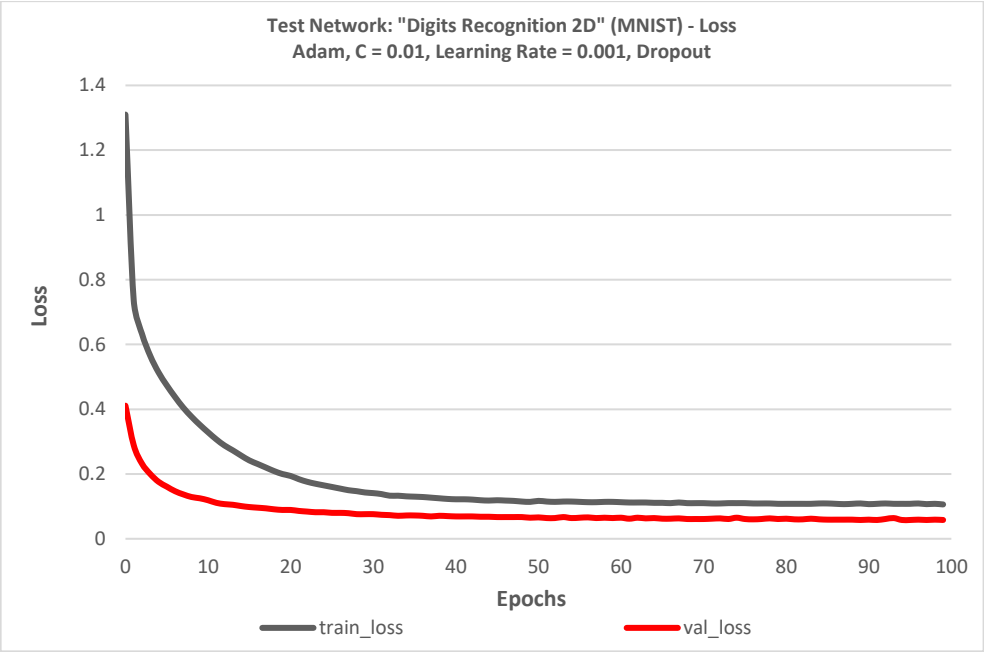


Figure 4.19. Test Network: "Digits Recognition 2D" (MNIST) with Adam - Loss

4.3.3.3 NewtonCG – Digits Recognition 2D (MNIST)

The 2D_CNN_4L_1 (Section 4.3.2.3.1), the same 4-layer CNN network that is being mentioned in Section 4.3.3.2, performs great with the NewtonCG optimizer (Section 2.4.9) as well with training, validation, and testing equal to 0.99 (Table 4.5). Those accuracies are calculated the same way as the ones in the Adam optimizer experiment (Section 4.3.3.2). Based on those two tests, with the Adam and NewtonCG optimizer, it can be safely assumed that the 2D CNN implementation is correct, therefore it can be used for the AD/NC problem as well with the 2D slice-level MRI scans.

Optimizer – Dataset	***Model of the Best Validation Accuracy***		
	Training	Validation	Testing
NewtonCG – MNIST (Digits Recognition 2D)	0.99	0.99	0.99

Table 4.5. Training, validation, and testing accuracies of the model with the best validation accuracy with the NewtonCG optimizer and the MNIST dataset.

In Figure 4.20, we can see the training and validation accuracies, while in Figure 4.21 the training and validation losses for 100 epochs, of the model with the MNIST dataset and the NewtonCG optimizer.

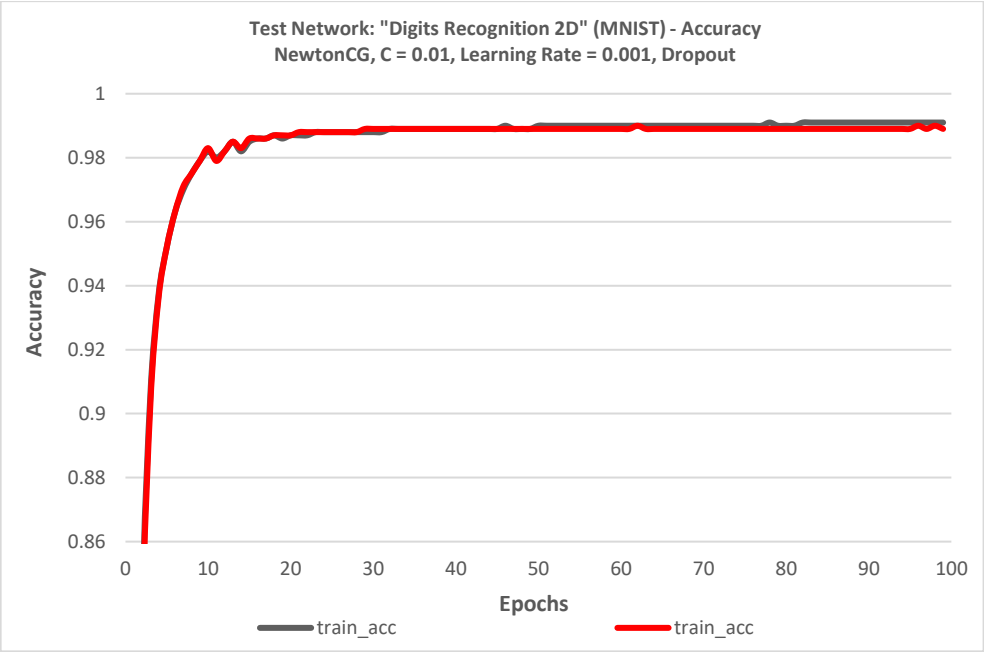


Figure 4.20. Test Network: "Digits Recognition 2D" (MNIST) with NewtonCG - Accuracy

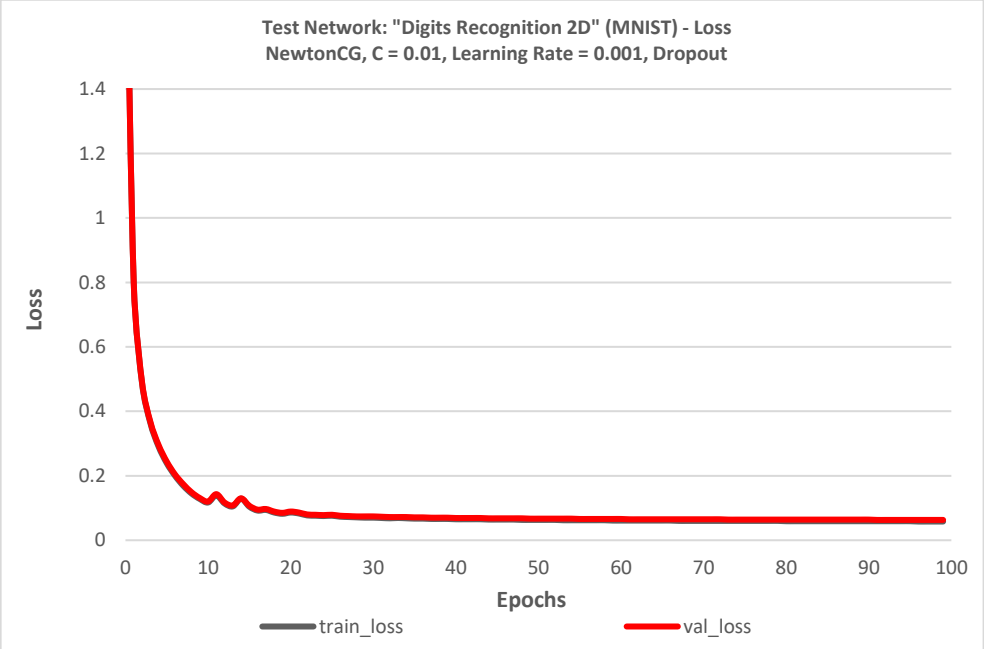


Figure 4.21. Test Network: "Digits Recognition 2D" (MNIST) with NewtonCG - Loss

4.3.4 3D CNN Network Architectures

4.3.4.1 Types of 3D CNNs

The initial set of T1-weighted MRI scans are 3D images (Section 3.2.5 – 3.27), therefore by using those in 3D CNNs we can assume that the results would be better than just providing a 2D slice of them (Section 3.2.3) in a 2D CNN (Section 4.3.2). Thus, a couple

of 3D CNN implementations were created which could take as input the 3D Shrunk Brains (Section 3.2.5), the 3D Left Hippocampus (Section 3.2.6), or the 3D Cropped Brains (Section 3.2.7) datasets.

More specifically, a 4-layer CNN network configuration was used for the 3D Shrunk Brains; two implementations, 4- and 5-layer CNNs for the 3D Cropped Brains; and three distinct 4-layer CNNs for the 3D Left Hippocampus. For all the following network configurations (Section 4.3.4.2 & 4.3.4.3), Dropout or Regularization can be added which does not affect the total number of variables for the network.

An issue in the Wang et al. (2020) implementation of the NewtonCG with Python 3 is that it does not utilize correctly many CPU cores for efficient parallelization in 3D CNNs. This makes the training process unable to run in a reasonable amount of time for large data and complex 3D CNN network architectures. Therefore, only 4-layer architectures and a single 5-layer network configuration were used. Also, all the 3D CNNs run in The Cyprus Institute servers which provide dedicated GPUs which are a must for 3D CNNs. Arcadia server which supported only integrated GPUs was unable to handle 3D CNNs.

For the creation of the figures with the different network configurations in Section 4.3.4, the tool (http://alexlenail.me/NN-SVG/AlexNet.html) was used in combination with Photoshop CC.

4.3.4.2 4-layer 3D CNN

All 4-layer 3D CNNs, similarly to the 2D ones (Section 4.3.2.3), have 3 Convolutional Layers, and a single fully connected layer that has two output neurons and takes as input the flatten output of the Conv3 (last convolutional layer). The output neurons return [1, 0] for AD prediction and [0, 1] for NC.

4.3.4.2.1 Shrunk Brains – 3D_CNN_4L_1

For the 3D Shrunk Brains (Section 3.2.5) dataset the 3D_CNN_4L_1 (Figure 4.22) configuration was used that had f1 = 32, f2 = 32, and f3 = 64, and the filter sizes were $3 \times 3 \times 3$. All Convolutional Layers applied a 3D Max-Pooling with pool_size = (2, 2, 2) and stride = (2, 2, 2). The activation function was set to be ReLU.

The input affects the number of parameters since the filters; size changes. For example, the input dataset (3D Shrunk Brains) consists of 3D images of dimensions $44 \times 48 \times 44$,

therefore the size of the feature maps in the Conv1, Conv2, and Conv3 are $22 \times 24 \times 22$, $11 \times 12 \times 11$, and $5 \times 6 \times 5$ respectively (Figure 4.22). Each axis of the initial input image is divided by 2 in each layer due to the Max-Pooling and pool_size.

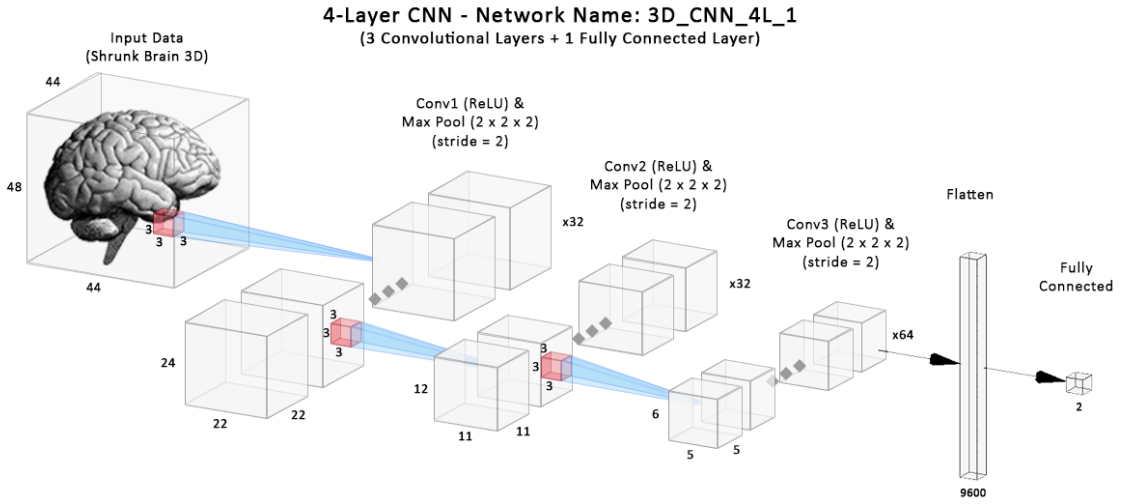


Figure 4.22. Network Configuration: 3D_CNN_4L_1. **Input:** 3D Shrunk Brains. 3 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 32, f2 = 32, f3 = 64; filter size = [3, 3, 3]

Subsequently, the total number of variables for the 3D_CNN_4L_1 network configuration, with input the 3D Shrunk Images is 412,552 bytes (~0.4MB). Consequently, if the original 3D Brains scan was given as input without being shrunk with its initial dimensions $174 \times 190 \times 174$, the size of the feature maps would be much larger and the total number of variables as well.

4.3.4.2.2 Cropped Brains – 3D_CNN_4L_1

The 3D Cropped Brains (Section 3.2.7) dataset used the 3D_CNN_4L_1 network configuration (Figure 4.23) as well. Therefore, the three Convolutional Layers have a number of feature maps f1 = 32, f2 = 32, f3 = 64 with filter size = $3 \times 3 \times 3$. All Convolutional Layers apply 3D Max-Pooling with pool_size = (2, 2, 2) and stride = (2, 2, 2), and they all use ReLU as the activation function.

The input affects the number of parameters since the filter's size changes. The input 3D Shrunk Brains with dimensions $44 \times 48 \times 44$ had a total size of variables equal to ~0.4MB (Section 4.3.4.2.1). Having as input the 3D Cropped Brains, where the dimensions of the images are $60 \times 60 \times 70$, the feature maps in Conv1, Conv2, and Conv3 will become $30 \times 30 \times 35$, $15 \times 15 \times 17$, and $7 \times 7 \times 8$ respectively. Subsequently, the total size of space needed for the variables of the 3D_CNN_4L_1

network configuration with input the 3D Cropped Brains is 536,456 bytes (~0.5MB), relatively more than the 3D Shrunk Brains (~0.4MB) by ~20%.

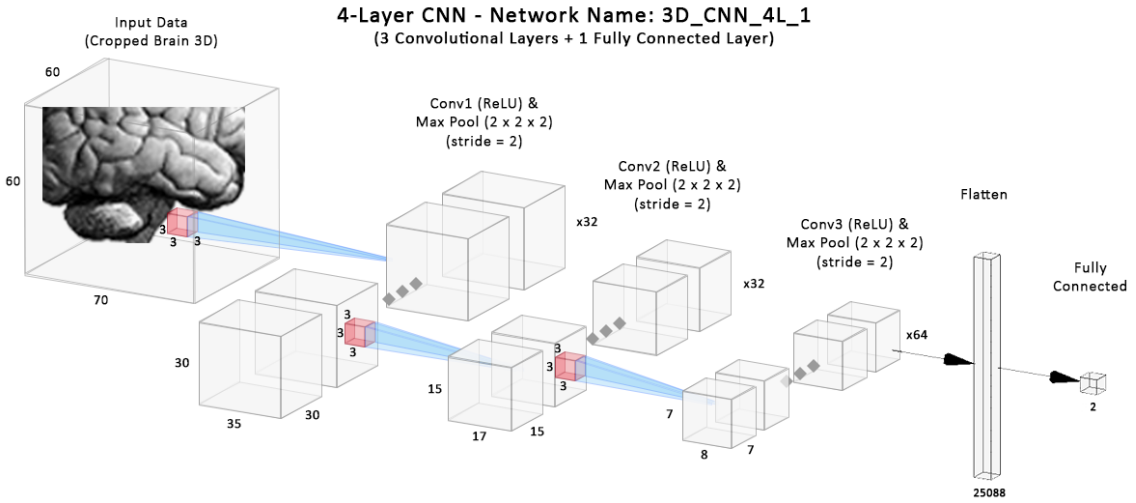


Figure 4.23. Network Configuration: 3D_CNN_4L_1. **Input:** 3D Cropped Brains. 3 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 32, f2 = 32, f3 = 64; filter size = [3, 3, 3]

4.3.4.2.3 Left Hippocampus – 3D_CNN_4L_1

For the 3D Left Hippocampus (Section 3.2.6), the same 3D_CNN_4L_1 network configuration (Figure 4.24) was used, as for the 3D Shrunk Brains (Section 4.3.4.2.1) and the 3D Cropped Brains (Section 4.3.4.2.2). The filters on each Convolutional Layer are f1 = 32, f2 = 32, and f3 = 64, and the filter sizes are $3 \times 3 \times 3$. Still, each Convolutional Layer applied a 3D Max-Pooling with pool_size = (2, 2, 2) and stride = (2, 2, 2).

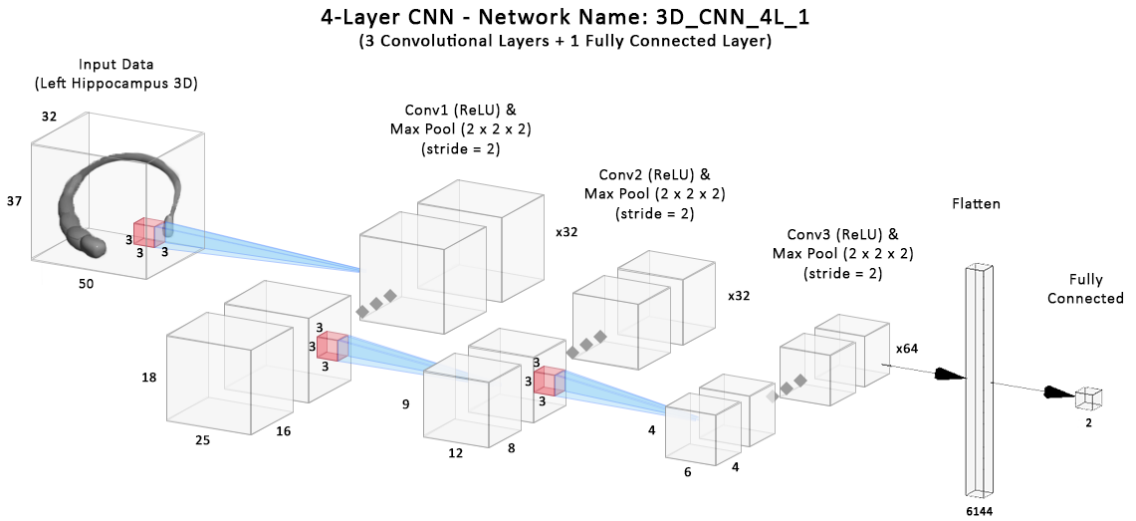


Figure 4.24. Network Configuration: 3D_CNN_4L_1. **Input:** 3D Left Hippocampus. 3 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 32, f2 = 32, f3 = 64; filter size = [3, 3, 3]

The input affects the number of parameters since the filter's size changes. For example, the input for the 3D Shrunk Brains dataset was 3D images of dimensions $44 \times 48 \times 44$, and the total size of variables was ~0.4MB. Having as input the 3D Left Hippocampus, where the dimensions of the images are $32 \times 37 \times 50$, the size of the feature maps in the Conv1, Conv2, and Conv3 will become $18 \times 25 \times 16$, $9 \times 12 \times 8$, and $4 \times 6 \times 4$ respectively (Figure 4.24). Subsequently, the total space needed for variables of the 3D_CNN_4L_1 network configuration with input the 3D Left Hippocampus is 384,904 bytes (~0.3MB), slightly less than the 3D Shrunk Brains.

4.3.4.2.4 Left Hippocampus – 3D_CNN_4L_2

For the 3D Left Hippocampus (Section 3.2.6), another network configuration was used, the 3D_CNN_4L_2 (Figure 4.25). The filters on each Convolutional Layer are the same as in the 3D_CNN_4L_1 (Section 4.3.4.2.3), f1 = 32, f2 = 32, and f3 = 64, and the filter sizes are all $3 \times 3 \times 3$. Contrariwise to the 3D_CNN_4L_1, no 3D Max-Pooling was applied to any of the three Convolutional Layers.

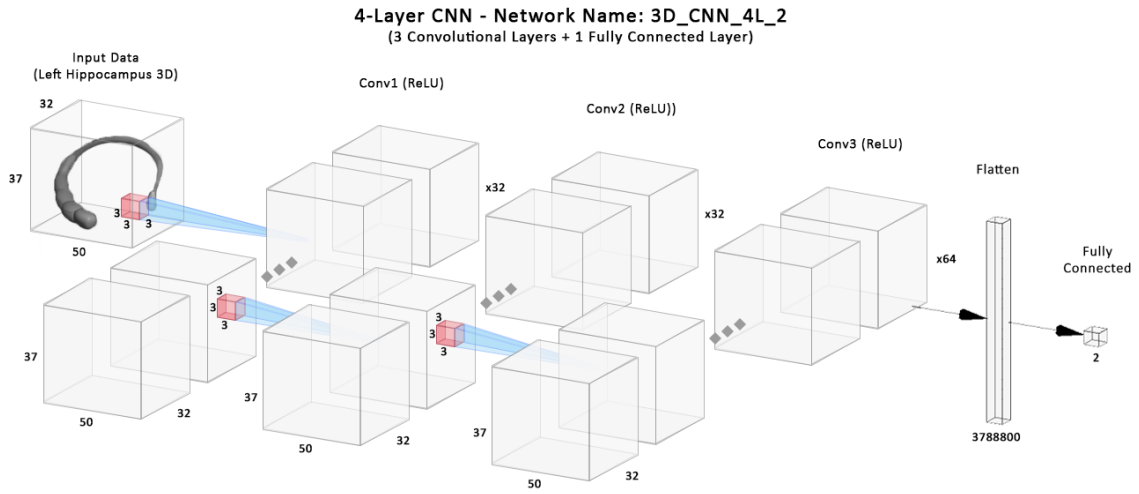


Figure 4.25. Network Configuration: 3D_CNN_4L_1. **Input:** 3D Left Hippocampus. 3 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 32, f2 = 32, f3 = 64; filter size = [3, 3, 3]; No Max-Pooling

This network architecture aims to test whether the Max-Pooling "destroys" the data, and this makes the network unable to extract the correct features from the hippocampal structures. One of the main trade-offs is the massive increase in the network's parameters which implies an increase in training time as well. The absence of Max-Pooling drastically affects the total space need for the variables of the network. While for the 3D_CNN_4L_1 the size was ~0.3MB, now the size is 30,646,152 bytes (~30MB), approximately 100 times more space is needed for the network's parameters.

4.3.4.2.5 Left Hippocampus – 3D_CNN_4L_3

The 3D Left Hippocampus (Section 3.2.6), was used in another network configuration, the 3D_CNN_4L_3 (Figure 4.26), the filters are still the same as in the 3D_CNN_4L_1 (Section 4.3.4.2.3), f1 = 32, f2 = 32, and f3 = 64, and the filter sizes are all $3 \times 3 \times 3$. This network does not apply Max-Pooling to all Convolutional Layers; only the Conv2 has 3D Max-Pooling with pool_size = (2, 2, 2) and stride = (2, 2, 2).

This network architecture is meant to combine the pros of both words, the fewer space requirements of the 3D_CNN_4L_1 implementation (Section 4.3.4.2.3), and the better performance of the 3D_CNN_4L_2 (Section 4.3.4.2.4), which did not destroy the data due to heavy Max-Pooling. Consequently, the total space needed for the variables of this network is 4,022,152 (~4MB), approximately 10 times more than the 3D_CNN_4L_1 (~0.3MB), and roughly 10 times less than the 3D_CNN_4L_2 (~30MB).

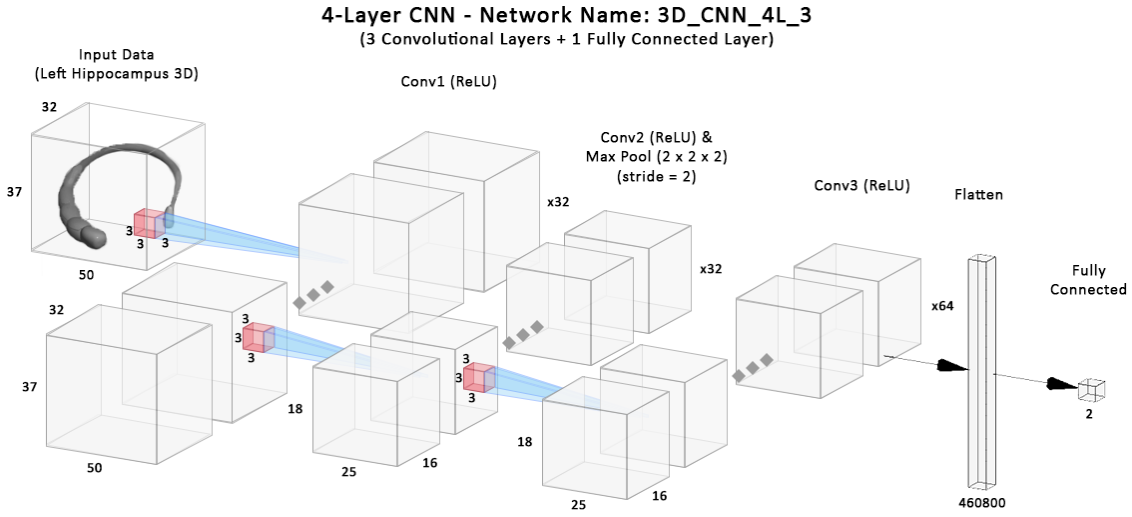


Figure 4.26. Network Configuration: 3D_CNN_4L_1. **Input:** 3D Left Hippocampus. 3 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 32, f2 = 32, f3 = 64; filter size = [3, 3, 3]; 1 Layer Max-Pooling

4.3.4.3 5-layer 3D CNN

4.3.4.3.1 Cropped Brains – 3D_CNN_5L_1

The 3D_CNN_5L_1 network configuration (Figure 4.27) with the 3D Cropped Brains dataset (Section 3.2.7), is the same as the 3D_CNN_4L_1 configuration (Section 4.3.4.2.2) with the only difference being the extra Convolutional Layer. The Conv4 layer that is being added still applies Max-Pooling, uses ReLU as an activation function, and has 64 filters with filter size = $3 \times 3 \times 3$ as well.

The total space needed for the parameters of the 3D_CNN_5L_1 network configuration increase to 796808 bytes (~0.8MB), almost double the ones in the 3D_CNN_4L_1 which needs ~0.4MB.

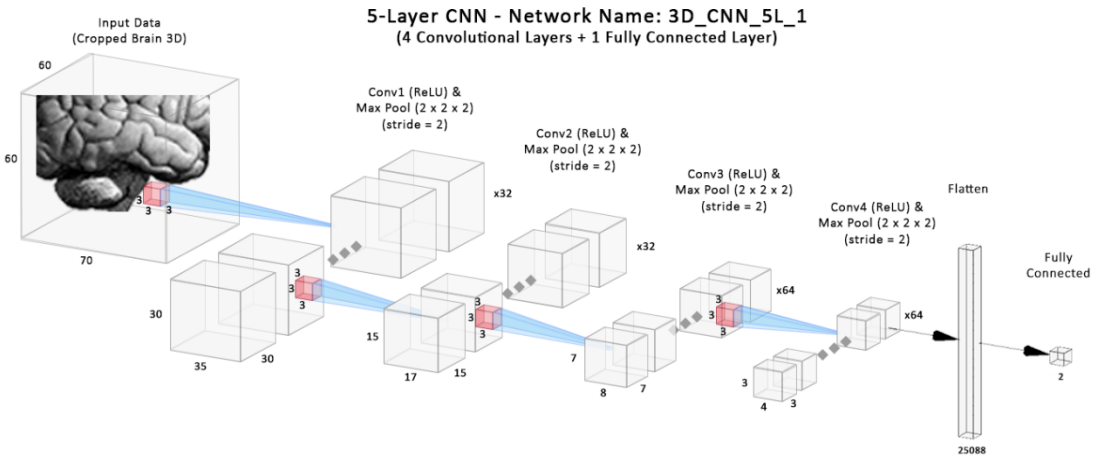


Figure 4.27. Network Configuration: 3D_CNN_5L_1. **Input:** 3D Cropped Brains. 4 Convolutional Layers and a single dense layer. **Feature Maps:** f1= 32, f2 = 32, f3 = 64, f4 = 64; filter size = [3, 3, 3]

4.3.5 3D CNN Networks Correctness

4.3.5.1 3D MNIST Dataset

I have used the 3D MNIST dataset from (https://www.kaggle.com/daavoo/3d-mnist), to test the 3D CNN Implementation (Section 4.3.4). The dataset was already split into two sets, the training and validation set with 10000 and 2000 samples respectively. Since 3D CNN networks recently started to be used, it was harder to find a 3D dataset for benchmarking. This dataset consists of 3D colorful representations of the digits (Figure 4.28 & 4.29) of the 2D grayscale MNIST dataset, the same that was used in Section 4.3.3 to check the correctness of the 2D CNN.

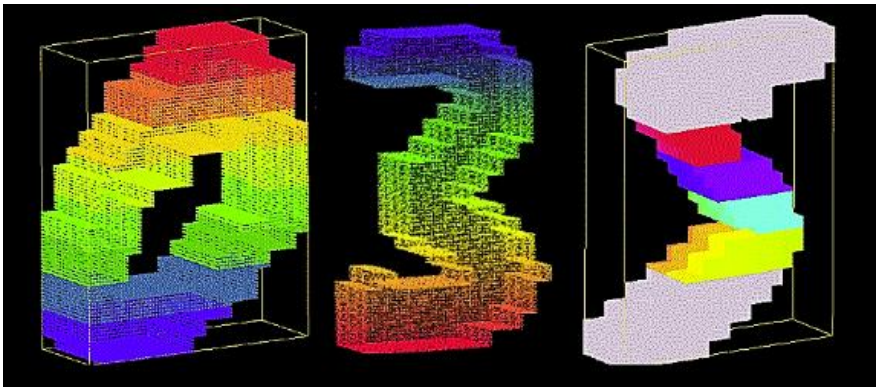


Figure 4.28. Digits '0', '3' and '5' of the 3D MNIST Dataset.

The dataset consists of 12000 3D colored sample images with each image's dimensions being $16 \times 16 \times 16$. Consequently, the total size of each sample is 4096 voxels. In contradiction with the MNIST which consists of grayscale 2D images, the 3D MNIST contains 3D-colored images, so each sample has 3 color channels (RGB).

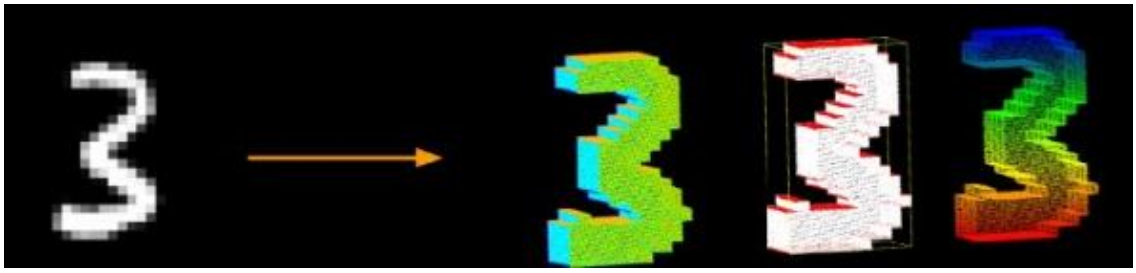


Figure 4.29. Grayscale 2D digit '3' of the MNIST Dataset **(Left)**.
Colored 3D digits of the 3D MNIST Dataset **(Right).**

Figure 4.30 shows a digit '5' of the 3D MNIST dataset. The dimensions of the image are $16 \times 16 \times 16$, and each voxel is represented by 3 values; Red, Green, and Blue (RGB) values since the image is colored. Code Snippet F.2 (digit-recognizer-3D.py), is the source code that was used for the creation of the '.mat' files with the training, validation, and testing sets, to be provided as input in the CNN implementation.

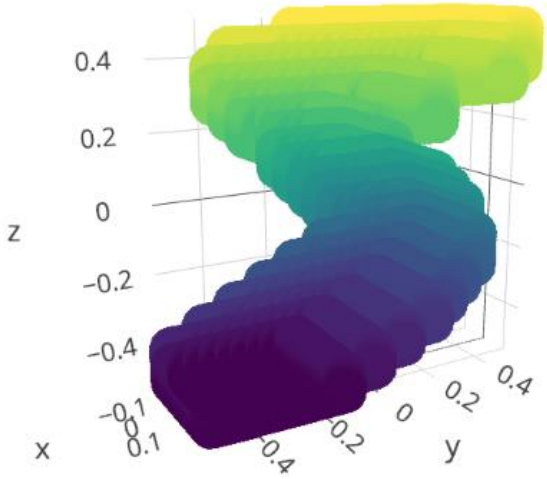


Figure 4.30. Digit 5 in the 3D MNIST Dataset
(https://www.kaggle.com/shivamb/3d-Convolutions-understanding-use-case)

4.3.5.2 Adam – Digits Recognition 3D (3D MNIST)

The CNN has to classify the 3D-colored digits into ten classes. For the 3D CNN implementation, no test set exists, since it was not required to check the model. Since the data were already split into training and validation sets, I have used those for training my

model. The training accuracy and the validation accuracy of the best model were 0.66 and 0.63 respectively.

Optimizer – Dataset	***Model of the Best Validation Accuracy***		
	Training	Validation	Testing
Adam – 3DMNIST (Digits Recognition 3D)	0.66	0.63	-

Table 4.6. Training, validation, and testing accuracies of the model with the best validation accuracy with CNN/Adam and 3D MNIST dataset.

Those accuracies may seem terrible in comparison with the original 2D MNIST Dataset, which they were 0.99 (Table 4.4), but this is not our main objective. Anyhow, those accuracies are not too far from others people's accuracies for the same problem and dataset (e.g., https://medium.com/shashwats-blog/3d-mnist-b922a3d07334) which was approximately 0.75 for the validation set. It is worth mentioning that the aforementioned experiment that yields 75% accuracy, uses a more complex 7-Layer Network (4 Convolutional Layers & 3 Dense Layers) than ours which is a 4-Layer Network (3 Convolutional Layers & 1 Dense Layer). Anyhow, by observing the plot of the training and validation accuracies (Figure 4.31), we can see that the network keeps learning throughout the whole training process.

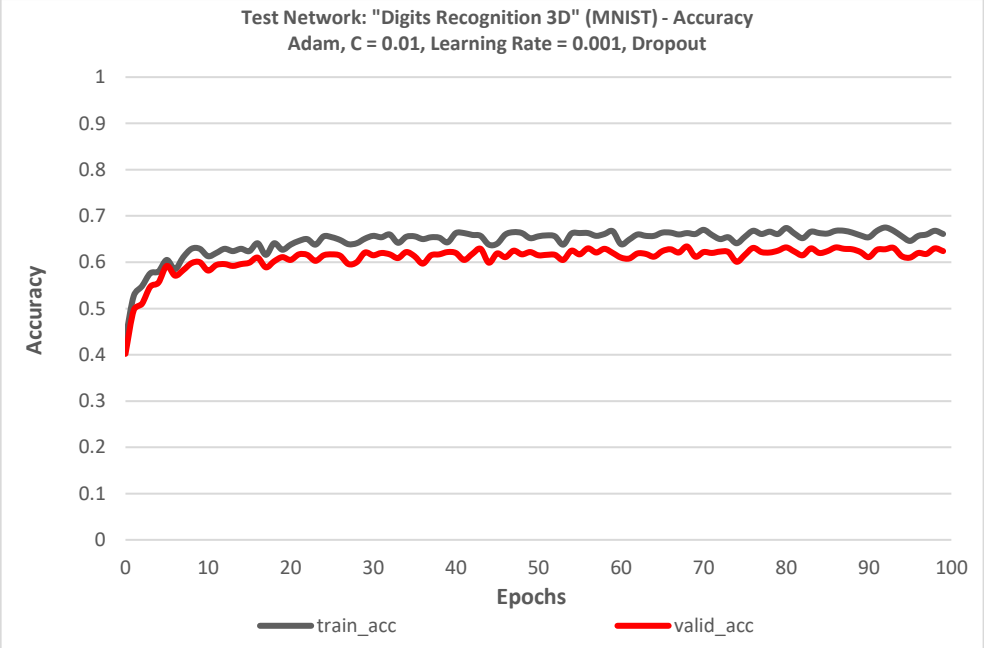


Figure 4.31. Test Network: "Digits Recognition 3D" (MNIST) with Adam – Accuracy

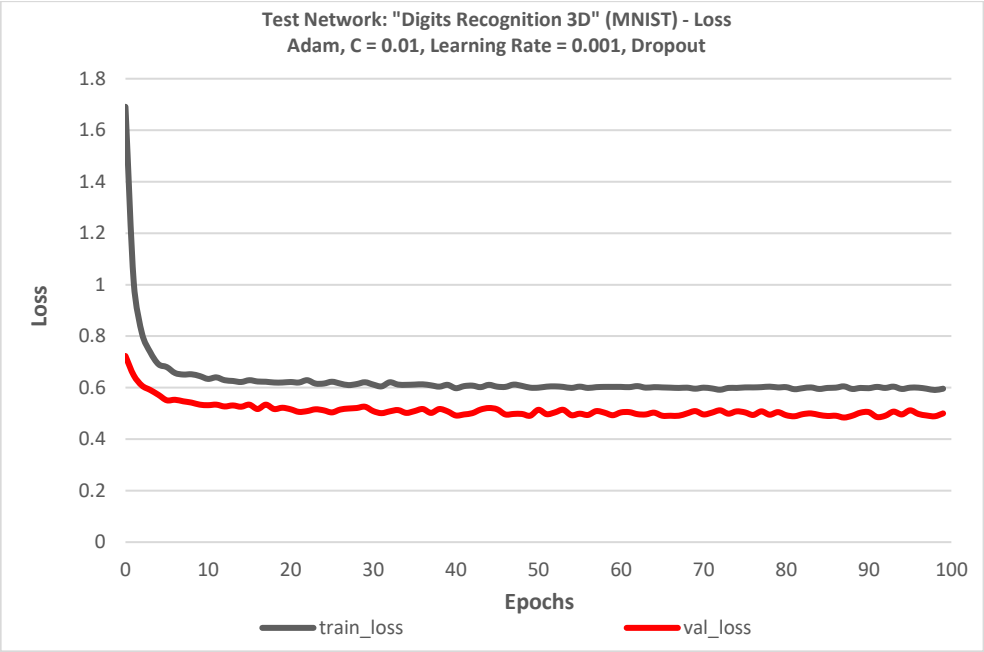


Figure 4.32. Test Network: "Digits Recognition 3D" (MNIST) with Adam – Loss

If we had more time, we could use a different dataset, to make sure that the data causes the error and not something else. Additionally, I could not perform many experiments with 3D CNNs, since I had to execute them in The Cyprus Institute's server, where our resources were limited. It is worth mentioning that based on the Figure 4.32 of the losses the model does not overfit, which was one of my main concerns since many of our experiments suffer from overfitting.

I suggest in future work to test the aforementioned 7-layer CNN network configuration with the Adam optimizer (https://medium.com/shashwats-blog/3d-mnist-b922a3d07334) that achieves an accuracy of 0.75 for the 3D MNIST Dataset. This could be an even better indication that the 3D implementation of the CNN was done correctly. Currently, some doubts exist because for the 3D Left Hippocampus (Section 3.2.6) dataset we were expecting much better results than the ones we got (Section 5.8). The network configuration could be the source of the problem in that case and not the specific dataset.

4.3.5.3 NewtonCG – Digits Recognition 3D (3D MNIST)

NewtonCG has a similar performance with the Adam optimizer for the 3D CNN as well as it did with the 2D CNN. It is worth mentioning that I was more skeptical about the 3D CNN implementations since I have implemented them by using TensorFlow / Keras. For the 2D CNN implementation, I have borrowed it from Wang et al. (2020).

Optimizer - Dataset	*Best Validation Accuracy*		
	Training	Validation	Testing
NewtonCG – 3D MNIST (Digits Recognition 3D)	0.68	0.61	-

Table 4.7. Training, validation, and testing accuracies of the model with the best validation accuracy with CNN/NewtonCG and 3D MNIST dataset.

If we compare the Adam method (Section 4.3.5.2) with the NewtonCG method, Adam's initial validation accuracy at epoch 0 was approximately 0.4, while for the NewtonCG method the initial validation accuracy was 0.1. The reason might be the random initialization of the network's weights. Unfortunately, I run the experiment once and I could not repeat it to understand the reason behind it, since I had to save resources in The Cyprus Institute's servers for running the 3D MRI scans experiments. In the Arcadia server, I could only run 2D CNN experiments since no dedicated GPUs are available that are a must for 3D CNNs.

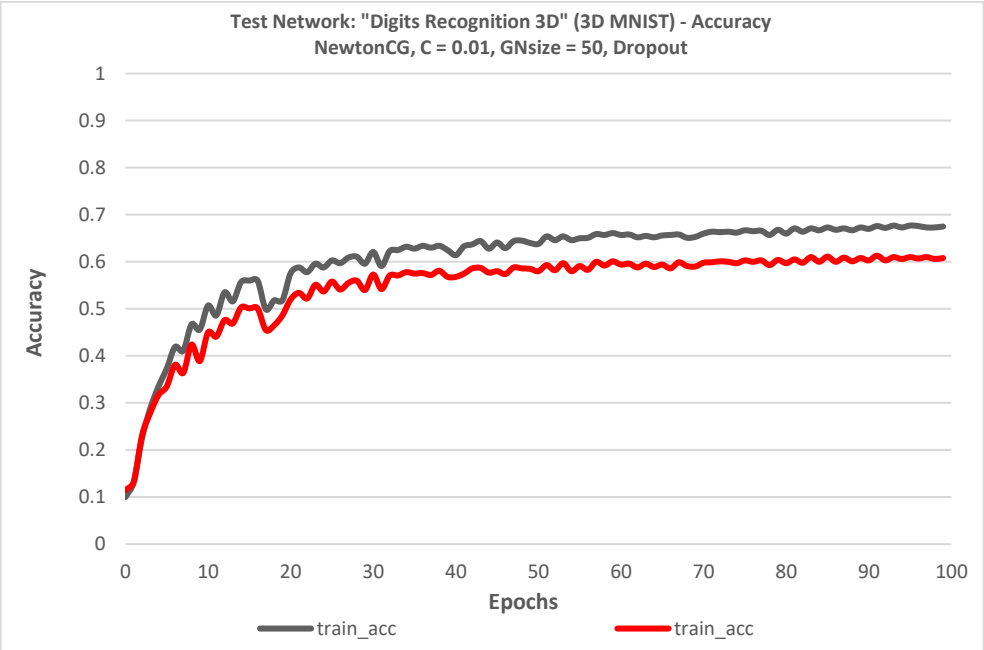


Figure 4.33. Test Network: "Digits Recognition 3D" (MNIST) with NewtonCG – Accuracy

In Figure 4.33, we can see that both the training and validation accuracies increase rapidly without showing any issues of overfitting. The training and validation losses (Figure 4.34) ensure that the model does not overfit since both losses keep reducing throughout the whole training process. We can also see that in the NewtonCG method, even if the initial weights were not ideal in comparison with Adam's case (Section 4.3.5.2), the algorithm still progresses fast and achieves similar accuracies.

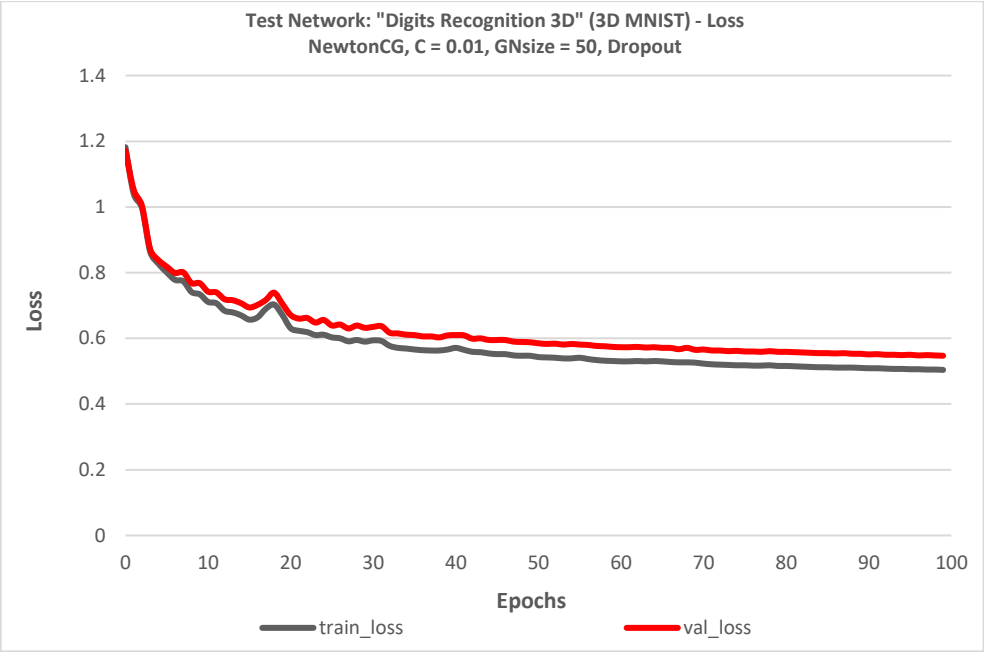


Figure 4.34. Test Network: "Digits Recognition 3D" (MNIST) with NewtonCG - Loss

Chapter 5

Experiments, Results, and Discussion

5.1 Introduction to Experiments, Results, and Discussion

Many experiments have been performed during this thesis, to examine the performance of the different datasets (Section 3.2); but mainly, we are going to focus on the 80 most important ones. The detailed hyperparameters and network configurations for each experiment can be found in Appendix P in Table P.1. Also, in Table P.2, the performance metrics of all the experiments are available. That many experiments were necessary to be performed since many hyperparameters contribute to the end result.

The three distinct main data sources, Hippocampus Features, T1-weighted MRIs, and 3D Left Hippocampus have been used to create the 9 different datasets for our experiments (Section 3.2). For each one of them, the readjusting process of the hyperparameters was necessary. Subsequently, many different network topologies have been used based on the

findings during the training process, and those topologies are mostly already defined in Chapter 4.

5.1.1 Test Set Issues

For the following experiments the test accuracy, test sensitivity, and test specificity will only be mentioned in the introduction of the subsections but not to be considered as valid metrics to judge the networks' performance. The reason is that the performance metrics for the test set do not behave as expected, as they should be close to the metrics of the validation set.

Two main reasons may hold why the test metrics are not as expected. The first one is that the test set is extremely small. The second one, which is less likely to be happening, is that a different preprocessing is applied during the training phase than the testing phase. The reason why this could possibly be happening is that, during training when the best model is saved, the preprocessing pipeline might not be saved as well. This is less likely to be happening because when the networks were checked (Section 4.3.5), the testing accuracy was close to the validation accuracy.

5.1.2 Experiments' Names

The naming of the experiments is based on the dataset and the optimizer used. The first part of the name represents the dataset, the second part the optimizer, and the third part is just a counter between the experiments of the same dataset and the same optimizer. Possible names for the datasets are the ones mentioned in Table 4.1. The second part of the name could be 4 possible values, "A" for Adam, "S" for SGD, "N" for NewtonCG (HFO implementation for CNNs (Section 4.3.1)), and "H" for HFO (HFO implementation for MLPs (Section 4.2.1.3)).

Thus, the name of the first (1) experiment performed with the dataset which contains a single 2D Brain slice of each scan, from Multiple scans of the same patient, from different periods, (B_2D_M); and also uses the Adam (A) optimizer, would be "B_2D_M_A1". If the same dataset is used with the NewtonCG (N) optimizer, and it was the third experiment performed during this thesis for the specific dataset and optimizer, the name of the experiment would be "B_2D_M_N3". The naming will help us later to distinguish the experiments easily.

5.2 Hippocampus Features [AD, NC]

For the Hippocampus Features -Multiple scans per patient (HF_M) dataset (Section 3.2.2), 27 experiments have been performed (Appendix G). Their performance metrics are grouped based on their optimizer (Adam, SGD, HFO) in Table 5.1. The column "Avg. Standard Deviation of Accuracy", represents the average of the standard deviations of the accuracies for the 10-folds. The column "Avg. epoch of best Valid Acc. per fold" is the average epoch that the early stopping was performed, therefore is the average epoch of the best validation accuracies per fold. The column "Best Validation Accuracy" is the best validation accuracy of the 10-folds.

Hippocampus Features – Multiple Scans per Patient [AD, NC]

	Training				Validation					
Experiment Name	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
Adam										
HF_M_A1	0.83	0.04	0.72	0.87	0.85	0.04	0.76	0.94	25	0.91
HF_M_A2	0.82	0.04	0.60	0.91	0.82	0.03	0.67	0.97	22	0.86
HF_M_A3	0.80	0.09	0.69	0.85	0.82	0.05	0.71	0.92	126	0.86
HF_M_A4	0.78	0.06	0.75	0.80	0.82	0.05	0.76	0.88	271	0.91
HF_M_A5	0.82	0.04	0.74	0.85	0.86	0.05	0.78	0.93	12	0.93
HF_M_A6	0.83	0.03	0.76	0.86	0.86	0.05	0.79	0.94	107	0.91
HF_M_A7	0.88	0.05	0.83	0.90	0.89	0.05	0.80	0.97	156	0.93
HF_M_A8	0.89	0.05	0.90	0.88	0.90	0.04	0.85	0.95	191	0.96
HF_M_A9	0.90	0.04	0.93	0.88	0.89	0.05	0.82	0.96	192	0.93
HF_M_A10	0.88	0.07	0.93	0.86	0.89	0.05	0.87	0.91	180	0.93
HF_M_A11	0.88	0.07	0.87	0.89	0.90	0.05	0.83	0.96	155	0.93
HF_M_A12	0.89	0.05	0.85	0.91	0.88	0.05	0.81	0.95	241	0.93
HF_M_A13	0.90	0.06	0.91	0.90	0.89	0.04	0.83	0.96	200	0.96
SGD										
HF_M_S1	0.83	0.05	0.64	0.90	0.82	0.05	0.69	0.95	14	0.89
HF_M_S2	0.83	0.06	0.62	0.92	0.82	0.06	0.65	0.98	146	0.89
HF_M_S3	0.74	0.11	0.66	0.78	0.79	0.04	0.69	0.88	163	0.84
HF_M_S4	0.74	0.09	0.62	0.78	0.74	0.05	0.63	0.85	339	0.82
HF_M_S5	0.94	0.03	0.92	0.95	0.88	0.04	0.82	0.95	272	0.93
HFO										

HF_M_H1	0.91	0.04	0.80	0.95	0.86	0.05	0.75	0.98	34	0.91
HF_M_H2	0.93	0.04	0.82	0.97	0.87	0.05	0.76	0.98	23	0.93
HF_M_H3	0.87	0.07	0.77	0.91	0.86	0.05	0.75	0.97	12	0.91
HF_M_H4	0.92	0.09	0.83	0.95	0.86	0.05	0.75	0.96	28	0.91
HF_M_H5	0.93	0.06	0.83	0.97	0.86	0.05	0.75	0.97	26	0.91
HF_M_H6	0.90	0.10	0.84	0.92	0.86	0.05	0.77	0.94	16	0.91
HF_M_H7	0.93	0.04	0.82	0.97	0.86	0.05	0.75	0.97	24	0.91
HF_M_H8	0.92	0.05	0.80	0.96	0.86	0.04	0.74	0.98	24	0.89
HF_M_H9	0.94	0.04	0.85	0.97	0.86	0.05	0.74	0.98	31	0.91

Table 5.1. Hippocampus Features - Multiples Scans per Patient [AD, NC] (HF_M) dataset (Section 3.2.2). Average training and validation results for the 10-folds.

The hyperparameters and the network configurations of the experiments mentioned in Table 5.1 can be seen in Table 5.2. The Learning Rate is the initial value of the learning rate that was set in both the SGD (Section 2.4.3) and Adam (Section 2.4.4) optimizers. The "CG iterations" is a parameter defined only for the HFO algorithm, which is the Conjugate Gradient (CG) iterations until the CG algorithm converges (Section 2.4.6). The alpha (α) is the regularization term for the L2 penalty (Section 2.3.13.7).

Hippocampus Features – Multiple Scans per Patient [AD, NC] Network Configuration - Hyperparameters						
Experiment Name	Optimizer	Learning Rate	CG iterations	α (alpha)	Max Epochs	Network Architecture
Adam						
HF_M_A1	Adam	0.3	-	7	500	[10, 8, 8, 1]
HF_M_A2	Adam	0.03	-	7	500	[10, 8, 8, 1]
HF_M_A3	Adam	0.003	-	7	500	[10, 8, 8, 1]
HF_M_A4	Adam	0.0003	-	7	500	[10, 8, 8, 1]
HF_M_A5	Adam	0.3	-	5	500	[10, 8, 8, 1]
HF_M_A6	Adam	0.3	-	3	500	[10, 8, 8, 1]
HF_M_A7	Adam	0.3	-	1	500	[10, 8, 8, 1]
HF_M_A8	Adam	0.3	-	0.1	500	[10, 8, 8, 1]
HF_M_A9	Adam	0.3	-	0.01	500	[10, 8, 8, 1]
HF_M_A10	Adam	0.3	-	0.001	500	[10, 8, 8, 1]
HF_M_A11	Adam	0.3	-	0.1	500	[10, 30, 1]
HF_M_A12	Adam	0.3	-	0.1	500	[10, 100, 1]
HF_M_A13	Adam	0.3	-	0.1	500	[10, 20, 20, 1]

SGD						
HF_M_S1	SGD	0.3	-	7	500	[10, 8, 8, 1]
HF_M_S2	SGD	0.03	-	7	500	[10, 8, 8, 1]
HF_M_S3	SGD	0.003	-	7	500	[10, 8, 8, 1]
HF_M_S4	SGD	0.0003	-	7	500	[10, 8, 8, 1]
HF_M_S5	SGD	0.3	-	0.1	500	[10, 8, 8, 1]
HFO						
HF_M_H1	HFO	-	1	-	100	[10, 30, 1]
HF_M_H2	HFO	-	2	-	100	[10, 30, 1]
HF_M_H3	HFO	-	4	-	100	[10, 30, 1]
HF_M_H4	HFO	-	8	-	100	[10, 30, 1]
HF_M_H5	HFO	-	16	-	100	[10, 30, 1]
HF_M_H6	HFO	-	32	-	100	[10, 30, 1]
HF_M_H7	HFO	-	2	-	100	[10, 20, 20, 1]
HF_M_H8	HFO	-	2	-	100	[10, 100, 1]
HF_M_H9	HFO	-	2	-	100	[10, 8, 8, 1]

Table 5.2. Hippocampus Features - Multiples Scans per Patient [AD, NC] (HF_M) dataset (Section 3.2.2). Different Network Configurations and Hyperparameters.

The alpha (α) was the only regularization technique used to combat overfitting by constraining the size of the weights. Larger values for α may fix high variance (overfitting) because it encourages smaller weights and results in a decision boundary plot that appears with lesser curvatures. Decreasing the alpha, on the other hand, may fix high bias (underfitting) by encouraging larger weights, which potentially results in a more complicated decision boundary (Pedregosa et al. 2011).

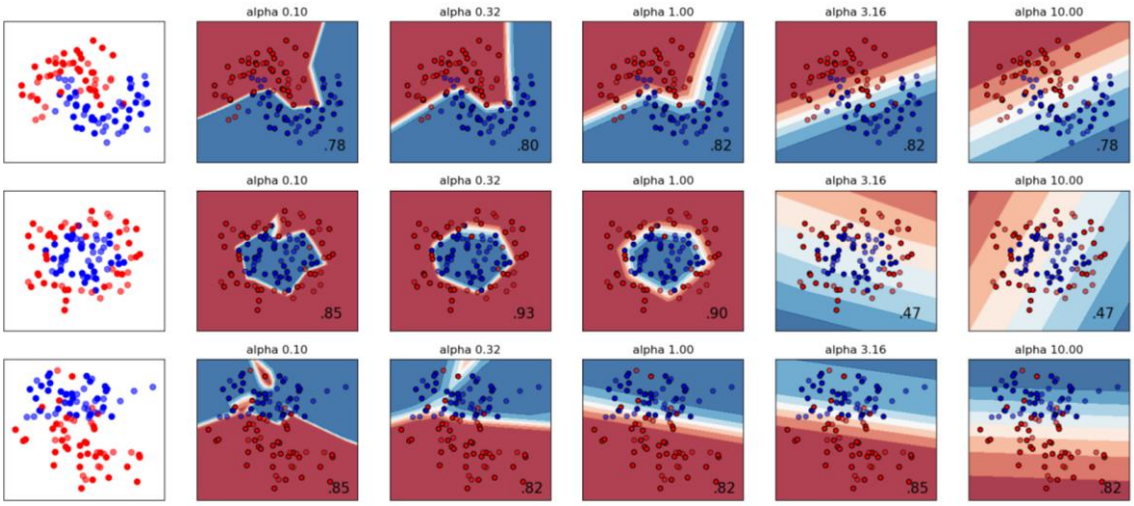


Figure 5.1. Effect of alpha (a) in the MLP classifier (Adam, SGD)

5.2.1 Comparison with Achilleos et al. (2020)

The Hippocampus Features (HF_M) dataset (Section 3.2.2), which was created by Achilleos et al. (2020) is going to be used in this thesis as well for direct comparison with them. Achilleos et al. (2020) used the HF_M dataset as input to their three classifiers, Decision Trees, Random Forests, and Argumentation Rules (more details can be found in their paper). In this thesis, we provided the same training and test sets, with the same 10-fold split as inputs into three distinct implementations; MLPs with Adam, with SGD, and with HFO. Achilleos et al.'s (2020) and our results are presented in Table 5.3.

Hippocampus Features (AD/NC) Comparison with Achilleos et al. (2020)				
	Classifier	***Accuracy***	***Sensitivity***	***Specificity***
		Average on 10 runs		
Achilleos et al. (2020)	**Decision Trees**	77%	66%	88%
	Random Forests	74%	56%	91%
	Argumentation Rules	91%	87%	95%
Ours	**MLP with SGD** *(HF_M_S5)*	88%	82%	95%
	MLP with Adam *(HF_M_A11)*	90%	85%	95%
	MLP with HFO *(HF_M_H2)*	87%	76%	98%

Table 5.3. Classification Results for the AD/NC Problem. Our models vs. Achilleos et al. (2020)

In the HF_M, the NC was labeled as 0 and the ADs as 1, therefore unlike the rest of the datasets (Section 3.2.3 – 3.2.7), the Sensitivity is the metric that matters the most, which is the proportion of correctly predicted ADs, over the total number of ADs. The test set is balanced since it has 22 AD and 22 NC samples. On the other hand, the training set is unbalanced, with 47 ADs and 122 NCs. This is an issue since the Specificity in all six experiments is higher than the sensitivity. This means that the model has a bias towards the NC samples. Consequently, this decreases and the overall validation accuracy as well.

Based on Achilleos et al.'s (2020) results, their classifier with the Argumentation Rules has the highest average validation accuracy and the highest average sensitivity, in comparison with the rest of the experiments, equal to 91% and 87% respectively. The average specificity for 10-folds of the Augmentation Rules is equal to 95% (Figure 5.1).

From our experiments, the MLP with Adam performs better than the SGD and the HFO optimizers, in terms of accuracy and sensitivity which are the two most important metrics for us (Figure 5.1). The HFO was expected to perform better but because the problem was "simple" since it takes as input only 10 features, this could not benefit from its full potential. HFO performs better than Adam in more complex problems (more inputs) where multiple layers of non-linear hidden units are required (Martens (2010); Martens and Sutskever (2012)).

It is worth mentioning that for all three optimizers of the MLP networks that we used, we tried to be as vanilla as possible, for as direct comparison as possible between the different classifiers. This means that not even Dropout was applied which was a key parameter because in all three optimizers overfitting had been observed. More specifically, for the HFO optimizer which is extremely aggressive, on average after 22 epochs we observed overfitting, while for the Adam on average after 155 epochs and for the SGD on average after 272 epochs. The signs of overfitting are even more clear when observing the loss plots of each optimizer, where the training loss decreases the whole training process, while the validation loss starts increasing at some point (SGD: Figures G.103 & G.106; Adam: Figures G.61 & G.64; HFO: Figures G.115 & G.118). The addition of Dropout is expected to improve the results significantly. Another reason, why the addition of Dropout was not tested was that the current implementation of MLP with HFO which was used did not support it and due to lack of time, we did not try a different one.

5.2.2 MLP with Adam

5.2.2.1 Introduction to MLP with Adam

For the Hippocampus Features – Multiple scans per patient [AD, NC] dataset (HF_M) with the Adam optimizer 13 experiments were performed to test different learning rates, values of α (alpha), and different architectures. The implementations of these experiments can be found in Appendix A.1. All experiments were performed for 500 epochs (Appendix G.1). The different values of learning rates used are {0.3, 0.03, 0.003, 0.0003}; the different values of α are {0.001, 0.01, 0.1, 1, 3, 5, 7}; and the different networks are {*[10, 8, 8, 1], [10, 20, 20, 1], [10, 30, 1], [10, 100, 1]*} (Section 4.2.2).

In general, the Adam optimizer is the best choice for this dataset, between SGD and HFO in terms of the performance metrics of our experiments, the validation accuracy,

sensitivity, and specificity. The network architecture does not seem to affect a lot the results. The smaller the α though, the better the accuracy and the sensitivity. This means that the α fixes the high variance in our case since underfitting is observed for large alphas. The learning rate is not very clear how it affects the performance other than decreasing the average epoch where the best model was found to have larger learning rates (0.3, 0.03). We can say that the learning rate of 0.3 provides a nice balance between great accuracy, sensitivity, specificity, and average epochs of the best model (Table 5.4).

Hippocampus Features – Multiple Scans per Patient [AD, NC] ***Optimizer: Adam***													
				Training				Validation					
Experiment Name	Learning Rate	α (alpha)	Network Architecture	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
HF_M_A1	0.3	7	[10, 8, 8, 1]	0.83	0.04	0.72	0.87	0.85	0.04	0.76	0.94	25	0.91
HF_M_A2	0.03	7	[10, 8, 8, 1]	0.82	0.04	0.60	0.91	0.82	0.03	0.67	0.97	22	0.86
HF_M_A3	0.003	7	[10, 8, 8, 1]	0.80	0.09	0.69	0.85	0.82	0.05	0.71	0.92	126	0.86
HF_M_A4	0.0003	7	[10, 8, 8, 1]	0.78	0.06	0.75	0.80	0.82	0.05	0.76	0.88	271	0.91
HF_M_A5	0.3	5	[10, 8, 8, 1]	0.82	0.04	0.74	0.85	0.86	0.05	0.78	0.93	12	0.93
HF_M_A6	0.3	3	[10, 8, 8, 1]	0.83	0.03Net	0.76	0.86	0.86	0.05	0.79	0.94	107	0.91
HF_M_A7	0.3	1	[10, 8, 8, 1]	0.88	0.05	0.83	0.90	0.89	0.05	0.80	0.97	156	0.93
HF_M_A8	0.3	0.1	[10, 8, 8, 1]	0.89	0.05	0.90	0.88	0.90	0.04	0.85	0.95	191	0.96
HF_M_A9	0.3	0.01	[10, 8, 8, 1]	0.90	0.04	0.93	0.88	0.89	0.05	0.82	0.96	192	0.93
HF_M_A10	0.3	0.001	[10, 8, 8, 1]	0.88	0.07	0.93	0.86	0.89	0.05	0.87	0.91	180	0.93
HF_M_A11	0.3	0.1	[10, 30, 1]	0.88	0.07	0.87	0.89	0.90	0.05	0.83	0.96	155	0.93
HF_M_A12	0.3	0.1	[10, 100, 1]	0.89	0.05	0.85	0.91	0.88	0.05	0.81	0.95	241	0.93
HF_M_A13	0.3	0.1	[10, 20, 20, 1]	0.90	0.06	0.91	0.90	0.89	0.04	0.83	0.96	200	0.96

Table 5.4. Hippocampus Features with Multiple Scans per Patient [AD, NC] with the Adam optimizer. Experiments with different network configurations, learning rates, and values of α.

5.2.2.2 Experiments HF_M_A {1 – 4}:

α = 7, lr = {0.3, 0.03, 0.003, 0.0003}, net = [10, 8, 8, 1]

The experiments HF_M_A {1 – 4} with the HF_M dataset (Section 3.2.2) and the MLP implementation with the Adam optimizer, are meant to compare the four different values of learning rates {0.3, 0.03, 0.003, 0.0003}. All four experiments use the [10, 8, 8, 1] network architecture (Section 4.2.2.1) with α = 7. Based on their training/validation

accuracies and losses, in general, the experiments with the learning rates 0.3 (HF_M_A1) and 0.03 (HF_M_A2), behave very similarly, with the 0.3 being a bit noisier.

Detailed results for each experiment can be found in Appendix G. More specifically, for the experiments HF_M_A {1 – 4}, their training/validation accuracies and losses of each fold can be seen in Appendixes G.1.1 – G.1.4. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

The average training accuracies for the 10-folds of the experiments HF_M_A {1 – 4}, can be seen in Figures 5.2. We observe that the training accuracy of the experiment with the 0.0003 learning rate (HF_M_A4), even after 500 epochs, did not reach the accuracy of the rest of the experiments (red, Figure 5.2). The 0.003 (HF_M_A3) reaches the same accuracy as the 0.3 and 0.03 after 100 epochs approximately (green, Figure 5.2).

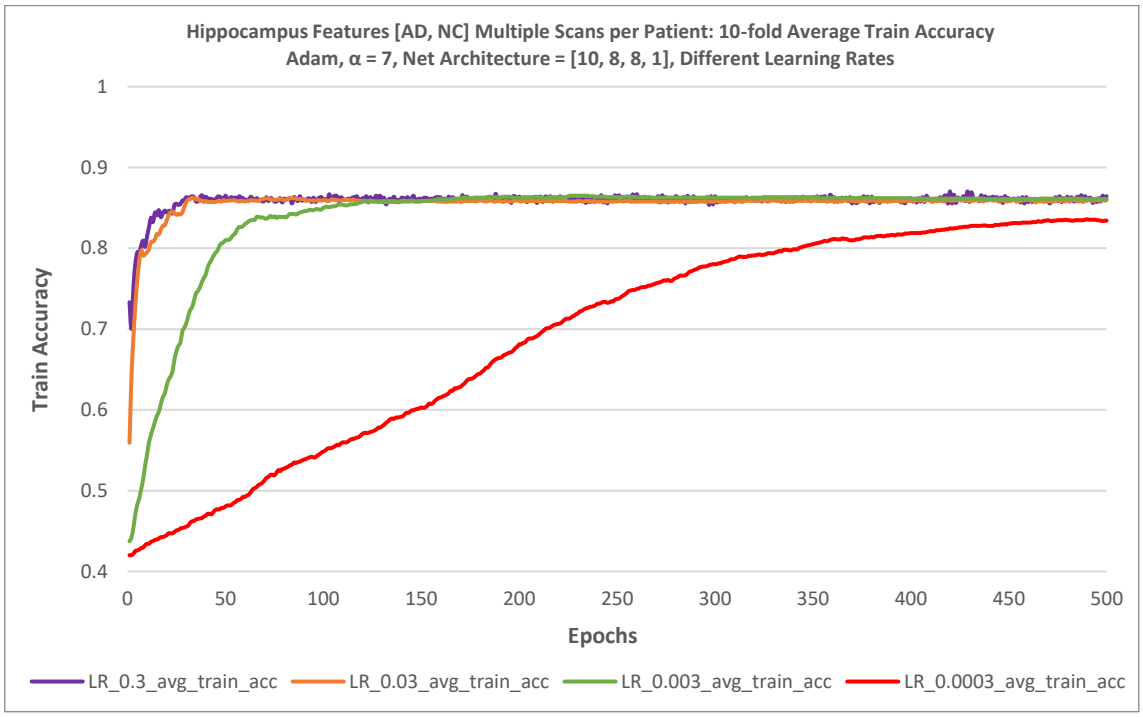


Figure 5.2. Average training accuracies of the 10-folds of the experiments HF_M_A {1 – 4}. Comparing different Learning Rates with the Adam optimizer.

The average training losses for the 10-folds of the experiments HF_M_A {1 – 4}, can be seen in Figures 5.3. The training loss of the experiment with the 0.0003 learning rate (HF_M_A4), was unable to reach the same loss value, as the other three learning rates even after 500 epochs (Figure 5.3), therefore, it progresses very slowly.

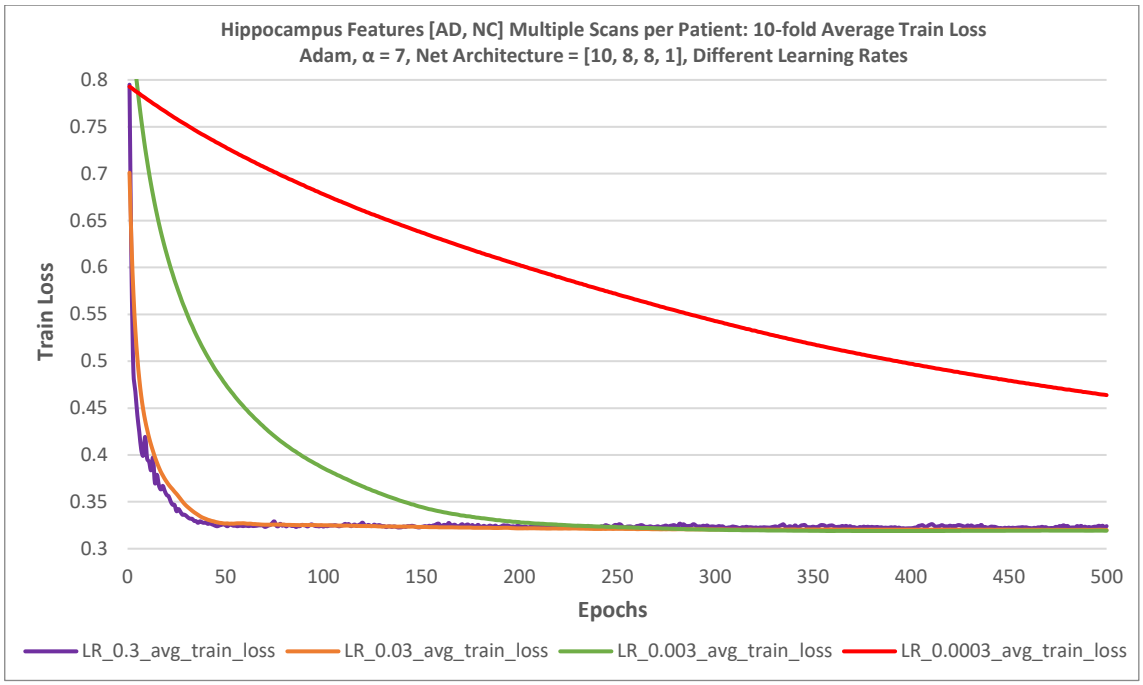


Figure 5.3. Average training losses of the 10-folds of the experiments HF_M_A {1 – 4}.
Comparing different Learning Rates with the Adam optimizer.

The average validation accuracies and losses for the 10-folds of the experiments HF_M_A {1 – 4}, can be seen in Figures 5.4 & 5.5. Similar behaviors to the training accuracies are observed for the validation accuracies as well for the corresponding learning rates.

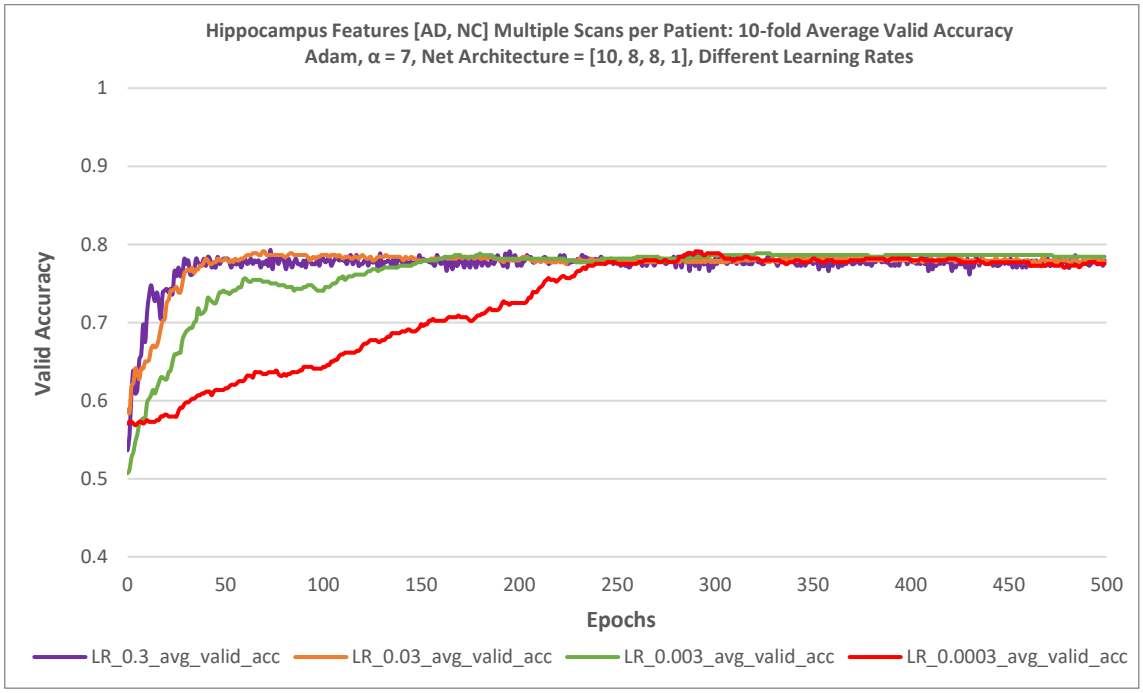


Figure 5.4. Average validation accuracies of the 10-folds of the experiments HF_M_A {1 – 4}.
Comparing different Learning Rates with the Adam optimizer.

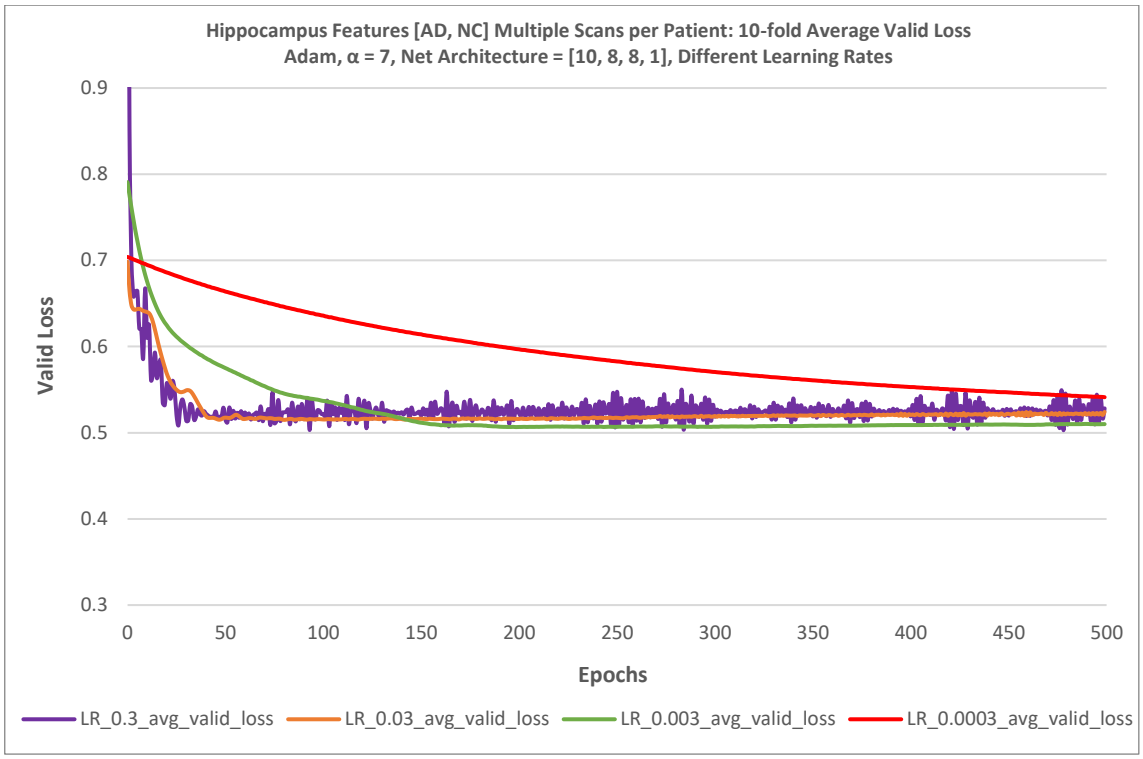


Figure 5.5. Average validation losses of the 10-folds of the experiments HF_M_A {1 – 4}. Comparing different Learning Rates with the Adam optimizer.

The validation loss for the experiment with 0.003 (HF_M_A3) learning rate, manages to catch up with the 0.3 (HF_M_A1) and 0.03 (HF_M_A2) after approximately 150 epochs and the 0.0003 (HF_M_A4) after 250 epochs (Figure 5.5). For all four experiments, from the loss's plots (Figure 5.3 & 5.5), we do not observe any signs of overfitting, since the validation losses keep decreasing while the training losses decrease too.

The conclusion from the experiments HF_M_A {1 - 4} is that the learning rates 0.3 and 0.03 are the best choices for our Hippocampus Features dataset since both the training and validation accuracies increase fast and the loss reduction is very steep. This helps us to converge in fewer epochs which is a major benefit for the training process of an algorithm. Consequently, the 0.3 learning rate is going to be used for the rest of the experiments with the Adam optimizer since based on the average validation accuracy, specificity, and sensitivity had better results than the 0.03 (Table 5.4).

5.2.2.3 Experiments HF_M_A {1, 5 – 10}:

α = {7, 5, 3, 1, 0.1, 0.01, 0.001}, lr = 0.3, net = [10, 8, 8, 1]

The experiments HF_M_A {1, 5 – 10} had been performed to compare the effect of the L2 Regularization term or so-called alpha (α) (Section 2.3.13.7), on our experiments. The

key finding is that α plays a major role in the performance as it controls whether the network will overfit or underfit. The different values of α that are going to be testes are {7, 5, 3, 1, 0.1, 0.01, 0.001}. All 7 experiments HF_M_A {1, 5 – 10}, will have the same learning rate, equal to 0.3, and the same network configuration [10, 8, 8, 1] (Section 4.2.2.1).

Detailed results for each experiment can be found in Appendix G. More specifically, for the experiments HF_M_A {1, 5 – 10}, their training/validation accuracies and losses of each fold can be seen in Appendixes G.1.1, G.1.5 – G.1.10. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

In Figure 5.6, which shows the average training accuracies for the experiments HF_M_A {1, 5 – 10}, we can easily detect two groups; the alphas 0.001, 0.01, 0.1, 1, and the alphas 3, 5, 7. The experiments of the second group, with alphas greater than 1, manage to reach a smaller training accuracy in comparison with the first group. More specifically the training accuracies of the experiments with alphas 3, 5, and 7, remain in a "straight" line after approximately 30 epochs while the first group continues to increase.

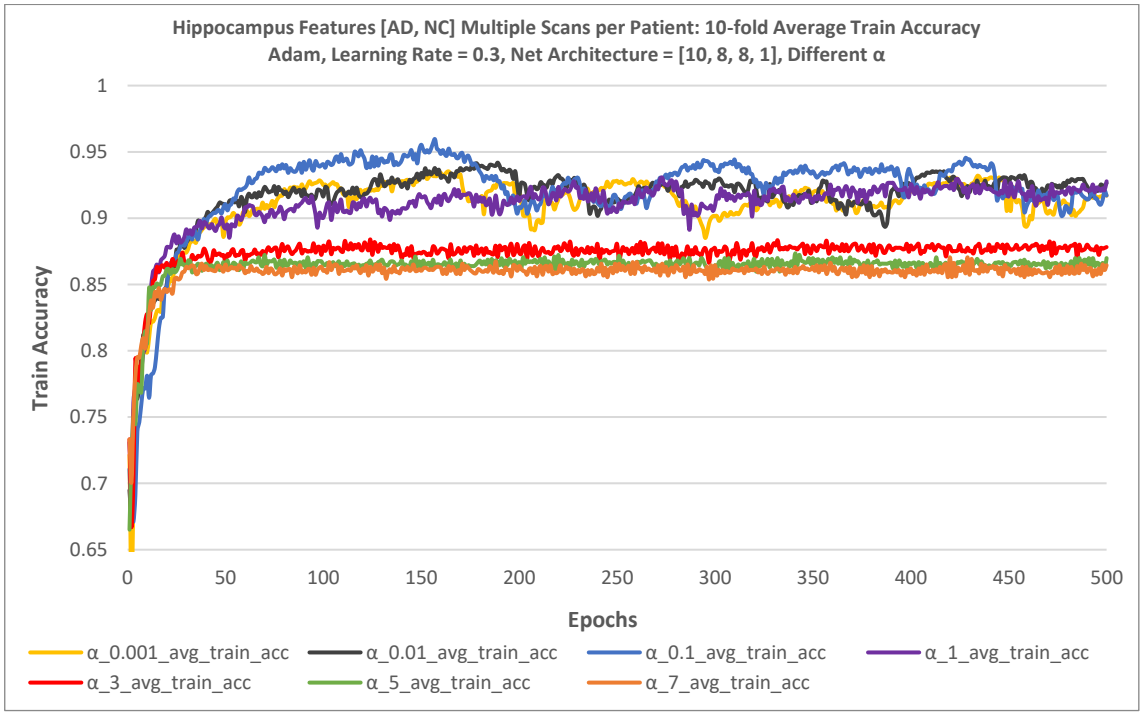


Figure 5.6. Average training accuracies of the 10-folds of the experiments HF_M_A {1, 5 – 10}. Comparing different values of alpha (α) with the Adam optimizer.

Similar behavior as in the training accuracies can be observed in the training losses of those experiments as well (Figure 5.7). More precisely, the training losses of the

experiments with alphas 0.001, 0.01, 0.1, and 1, remain stable after the 30th epoch, while the training losses of the experiments with alphas 3, 5, and 7, continue to decline.

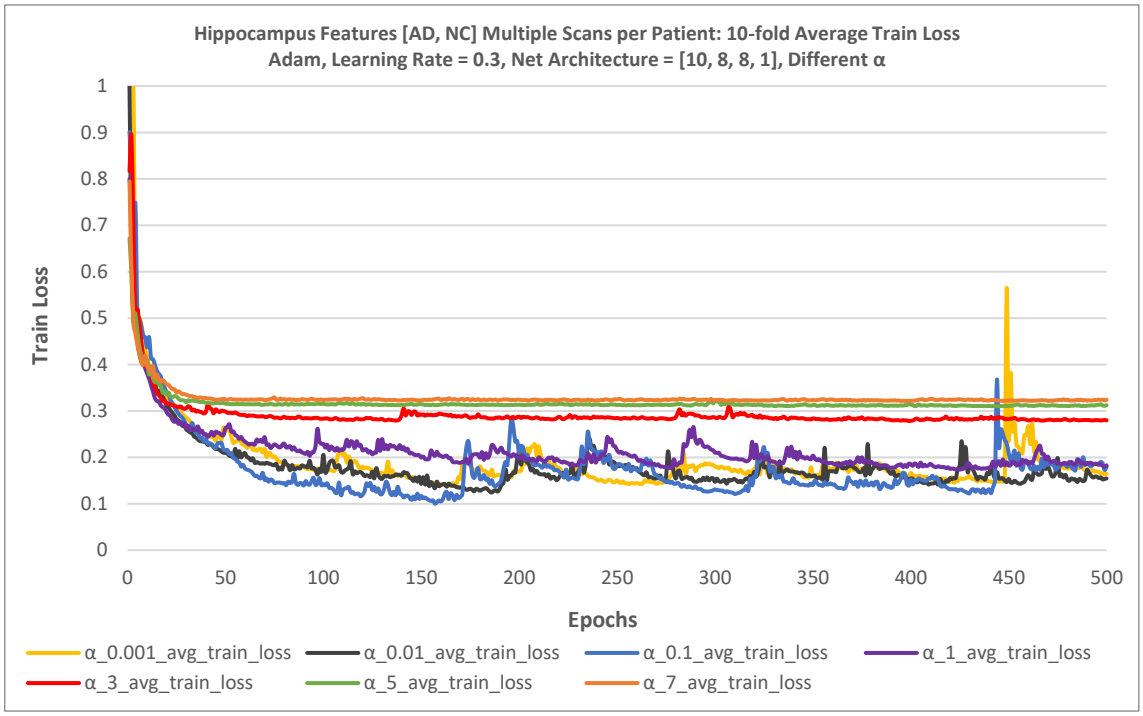


Figure 5.7. Average training losses of the 10-folds of the experiments HF_M_A {1, 5 – 10}. Comparing different values of alpha (α) with the Adam optimizer.

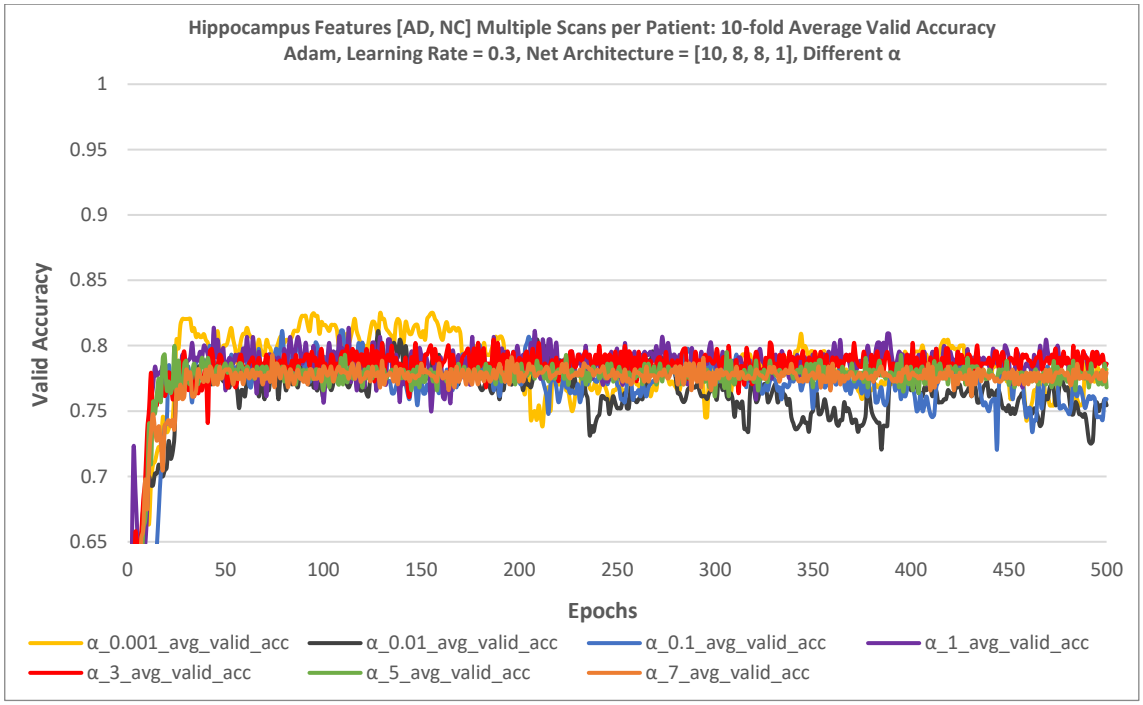


Figure 5.8. Average validation accuracies of the 10-folds of the experiments HF_M_A {1, 5 – 10}. Comparing different values of alpha (α) with the Adam optimizer.

Based on the validation accuracies plot (Figure 5.8), we can see that the experiments with alphas 3, 5, and 7 are 'stable' through the epochs, while the rest of the experiments suffer from overfitting. The reason is that larger values of alpha help to reduce overfitting but may cause underfitting (Pedregosa et al. 2011). The overfitting for the experiments with alphas 0.001, 0.01, 0.1, and 1 can be observed in the validation losses plot more clearly (Figure 5.9). The smaller the alpha is, the more steeply the validation loss increases. Since for all 7 experiments, the training loss decreases, therefore, severe overfitting can be detected in the experiments with alphas 3, 5, and 7 (HF_M_A {6, 5, 1}), and some overfitting in the experiment with alpha equal to 1 (HF_M_A7).

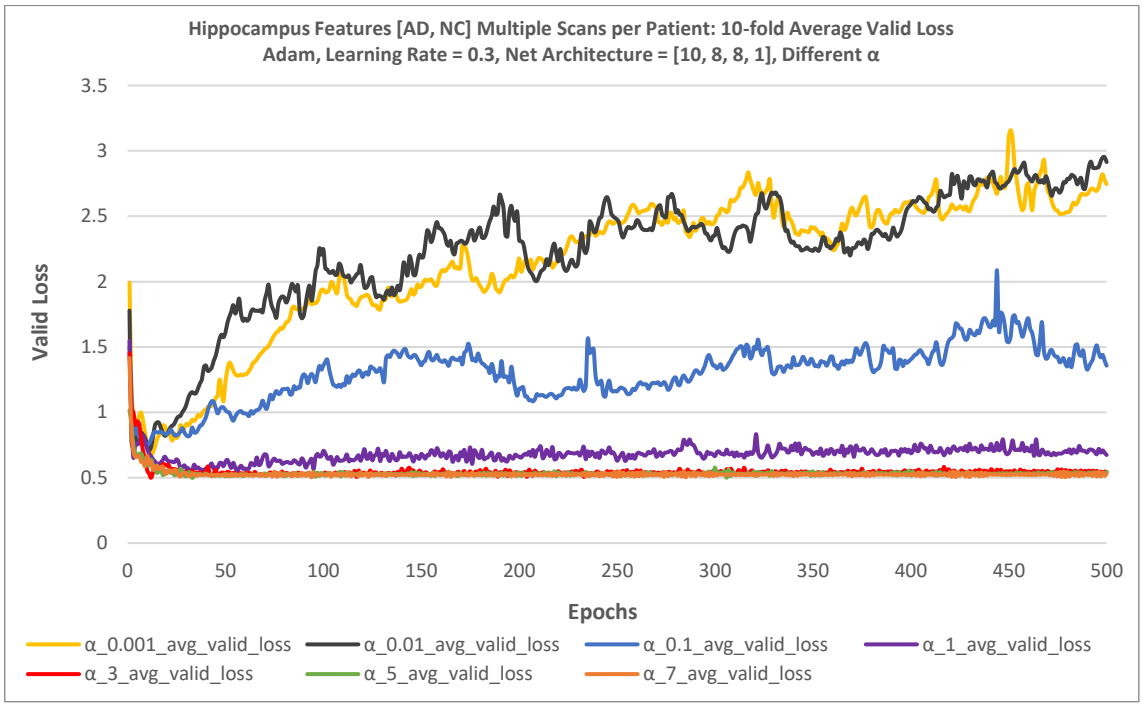


Figure 5.9. Average validation losses of the 10-folds of the experiments HF_M_A {1, 5 – 10}. Comparing different values of alpha (α) with the Adam optimizer.

Consequently, alphas help to reduce underfitting but may also cause overfitting. To improve our performance more, other regularization techniques against overfitting can be used such as Dropout in combination with small alphas.

5.2.2.4 Experiments HF_M_A {8, 11 – 13}:

α = 0.1, lr = 0.3, net = {[10, 8, 8, 1], [10, 30, 1], [10, 100, 1], [10, 20, 20, 1]}

The experiments HF_M_A {8, 11 – 13} are meant to compare how the different network configurations [10, 8, 8, 1] (Section 4.2.2.1), [10, 20, 20, 1] (Section 4.2.2.2), [10, 30, 1] (Section 4.2.2.3), [10, 100, 1] (Section 4.2.2.4), affect the performance of the model with

the HF_M dataset (Section 3.2.2), the MLP implementation (Section 2.3.11), and the Adam optimizer (Section 2.4.4). All four experiments have a learning rate equal to 0.3, and an alpha equal to 0.1. Those two values have been selected based on the performance of the experiments in Sections 5.2.2.2 & 5.2.2.3.

Detailed results for each experiment can be found in Appendix G. More specifically, for the experiments HF_M_A {8, 11 – 13}, their training/validation accuracies and losses of each fold can be seen in Appendixes G.1.8, G.1.11 – G.1.13. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

The training accuracies plot of the experiments HF_M_A {8, 11 – 13} can be seen in Figure 5.10. We can identify two groups in Figure 5.10; the first group consists of the experiments with the single hidden layer topologies [10, 30, 1] (orange) and [10, 100, 1] (green), and the second the experiments with the two hidden layers topologies [10, 8, 8, 1] (purple) and [10, 20, 20, 1] (red). The training accuracies of the experiments of the same group behave very similarly. The training accuracies of the experiments in the first group increase throughout the whole training process. Contrariwise, for the two hidden layers topologies (second group), the training accuracies face a slow start and decrease as the training process progresses.

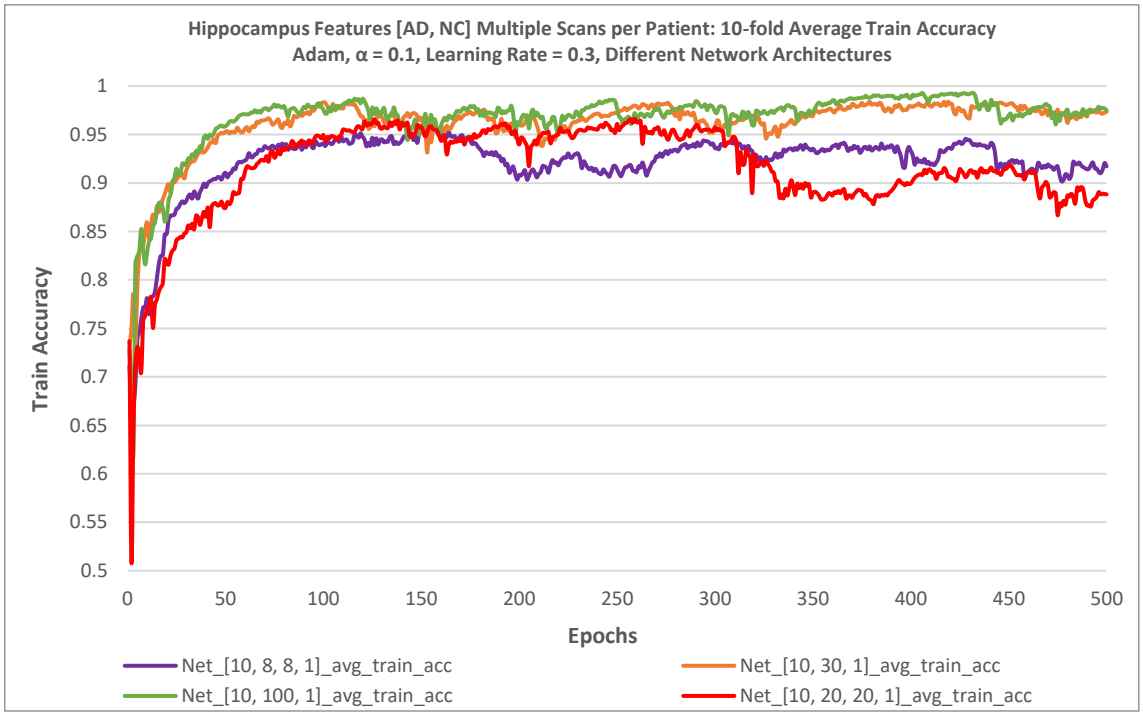


Figure 5.10. Average training accuracies of the 10-folds of the experiments HF_M_A {8, 11 – 13}. Comparing different network configurations with the Adam optimizer.

In Figure 5.11 we can see the training losses of the aforementioned experiments. We observe again the two groups in the losses plot as well. In the experiments with the two hidden layers (HF_M_ A {8, 13}), the training losses increase over time, while for the experiments (HF_M_ A {11, 12}) they remain relatively stable near zero.

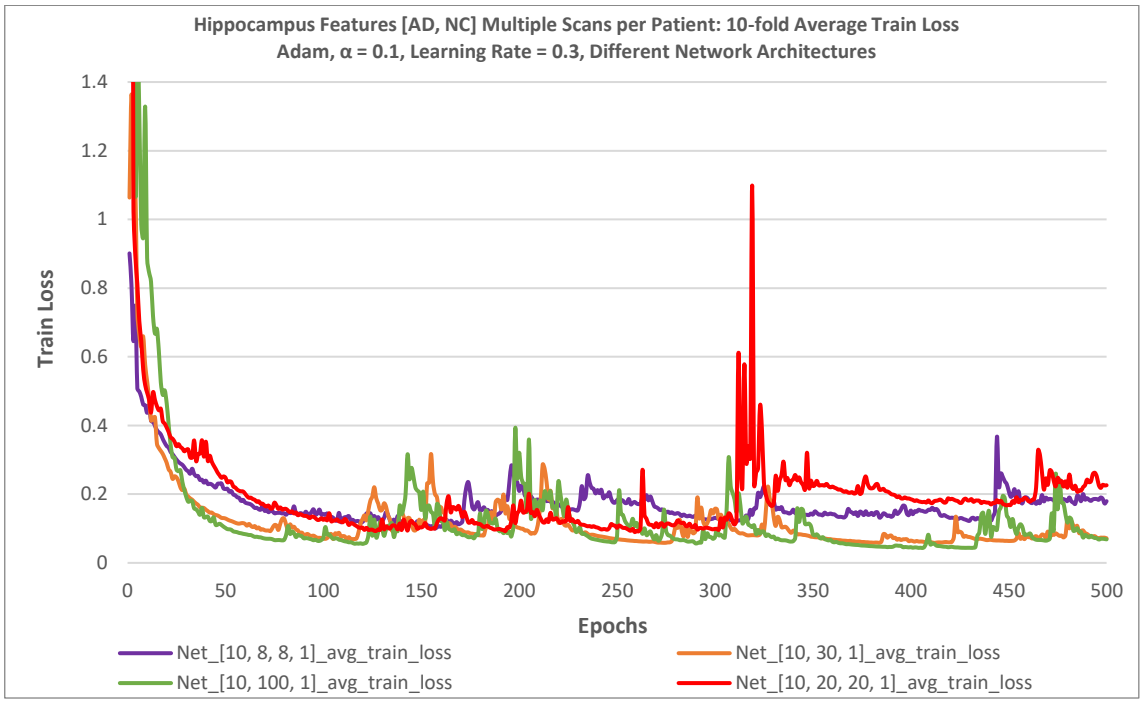


Figure 5.11. Average training losses of the 10-folds of the experiments HF_M_A {8, 11 – 13}. Comparing different network configurations with the Adam optimizer.

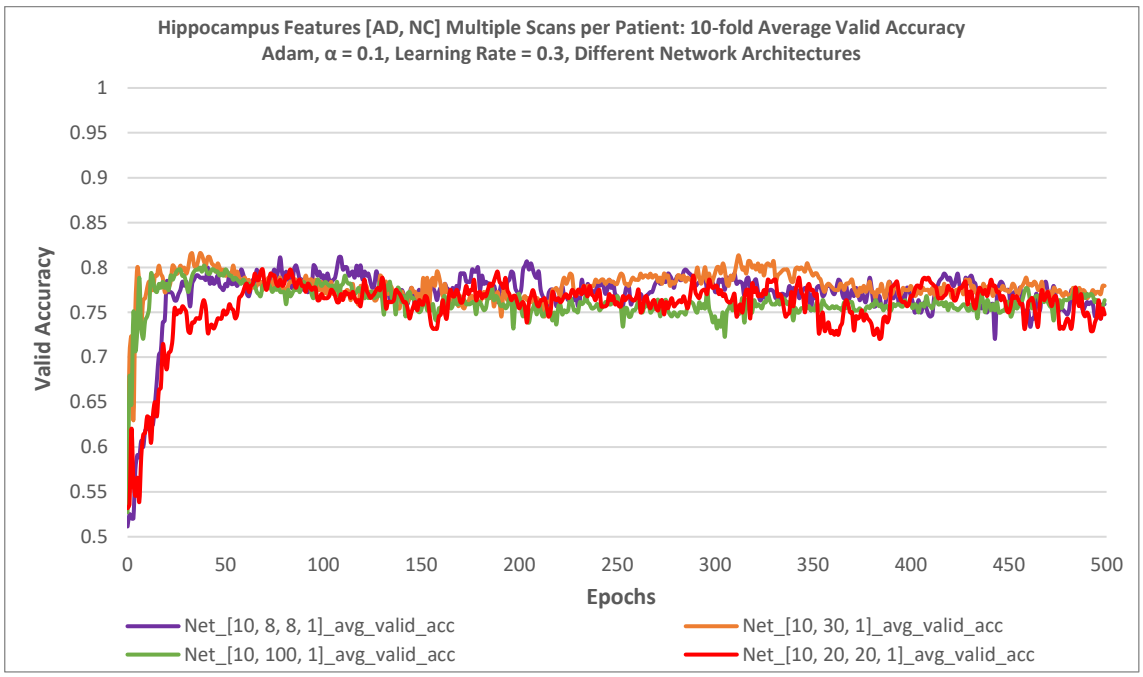


Figure 5.12. Average validation accuracies of the 10-folds of the experiments HF_M_A {8, 11 – 13}. Comparing different network configurations with the Adam optimizer.

The validation accuracies plot of the experiments (Figure 5.12) shows us that very rapidly in all four implementations the highest accuracy is reached in approximately 50 epochs. Especially in the single hidden layer implementations (green & orange), after approximately 40 epochs they reach their highest average validation accuracy. Then, in all four experiments, the validation accuracy fluctuates in a range approximately between 0.75 and 0.8 with a slight decrease over time. This could potentially mean that the model overfits.

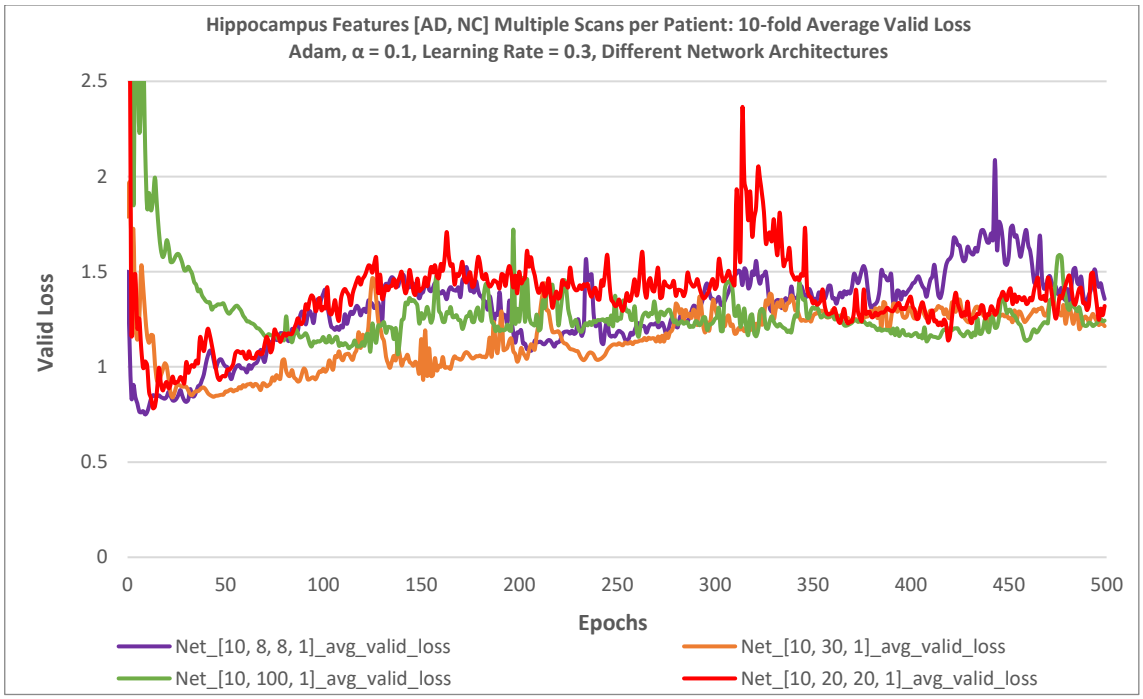


Figure 5.13. Average validation losses of the 10-folds of the experiments HF_M_A {8, 11 – 13}. Comparing different network configurations with the Adam optimizer.

From the validation losses plot (Figure 5.13), we can validate our assumption for overfitting. More specifically, the experiments HF_M_A {8, 11, 13}, with network configurations [10, 8, 8, 1], [10, 20, 20, 1], and [10, 30, 1], reach their minimum validation loss after 20 epochs approximately, and afterward it started increasing. Only the experiments HF_M_A12 with the [10, 100, 1] topology seems not to be overfitting as much.

In conclusion, the topology [10, 30, 1] seems to be the best choice among these four network configurations, since it provides low complexity (Section 4.2.2.3), does not fluctuate a lot (Figures 5.10 – 5.13), and achieves the highest validation accuracy and sensitivity between these experiments (Table 5.4).

5.2.3 MLP with SGD

5.2.3.1 Introduction to MLP with SGD

For the Hippocampus Features – Multiple scans per patient [AD, NC] dataset (HF_M) (Section 3.2.2), the 5 experiments HF_M_S {1 – 5} have been performed with the MLP classifier and the SDG optimizer. The implementations for these experiments can be found in Appendix A.2. The detailed training/validation accuracies and losses for each fold of these experiments can be found in Appendix G.2. The training, validation, and testing confusion matrices are also available in Appendix G.2.

All HF_M_S experiments run for 500 epochs with the network configuration [10, 8, 8, 1] (Section 4.2.2.1). Similar to the Adam optimizer (Section 5.2.2), larger learning rates perform better (Table 5.5). Based on our observations of the Adam optimizer (Section 5.2.2.3), the alphas equal to 0.1 and 7 were chosen, for the SGD optimizer, to be tested. Between the two alphas 0.1 and 7, the 0.1 (HF_M_S5) performs better, as expected based on Adam's results.

Hippocampus Features – Multiple Scans per Patient [AD, NC] ***Optimizer: SGD***													
				Training				Validation					
Experiment Name	Learning Rate	α (alpha)	Network Architecture	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
HF_M_S1	0.3	7	[10, 8, 8, 1]	0.83	0.05	0.64	0.90	0.82	0.05	0.69	0.95	14	0.89
HF_M_S2	0.03	7	[10, 8, 8, 1]	0.83	0.06	0.62	0.92	0.82	0.06	0.65	0.98	146	0.89
HF_M_S3	0.003	7	[10, 8, 8, 1]	0.74	0.11	0.66	0.78	0.79	0.04	0.69	0.88	163	0.84
HF_M_S4	0.0003	7	[10, 8, 8, 1]	0.74	0.09	0.62	0.78	0.74	0.05	0.63	0.85	339	0.82
HF_M_S5	0.3	0.1	[10, 8, 8, 1]	0.94	0.03	0.92	0.95	0.88	0.04	0.82	0.95	272	0.93

Table 5.5. Hippocampus Features with Multiple Scans per Patient [AD, NC] with the SGD optimizer. Experiments with different learning rates, and values of α. Same network configuration [10, 8, 8, 1]

The experiment HF_M_S5 has the best performance among the five experiments (Table 5.5.). More specifically, HF_M_S5's validation accuracy is 0.88, the sensitivity is 0.82, and specificity 0.95. In direct comparison with the Adam optimizer, the SGD optimizer is much slower. The experiment's HF_M_S5 (best model MLP & SGD) average epoch of the best validation accuracies is 272 (Table 5.5), while for the experiment HF_M_A11 (best model MLP & Adam) is 155 (Table 5.4). Overall, the performance of Adam in terms

of average validation accuracy, sensitivity, and specificity for the 10-folds for 500 epochs is better than SGD for the specific dataset (HF_M).

5.2.3.2 Experiments HF_M_S {1 – 5}:

α = {7, 0.1}, lr = {0.3, 0.03, 0.003, 0.0003}, net = [10, 8, 8, 1]

The experiments HF_M_S {1 – 5} are meant to compare the effects of the different learning rates and alphas to the performance of the model with the MLP classifier (Section 2.3.11) and the HF_M dataset (Section 3.2.2). All implementations are going to use the network configuration [10, 8, 8, 1] as described in Section 4.2.2.1. The experiments HF_M_S {1 – 4} apply the learning rates {0.3, 0.003, 0.003, 0.0003} respectively, with alpha = 7. The experiment HF_M_S5's hyperparameters are learning rate = 0.3 and alpha = 0.1.

Detailed results for each experiment can be found in Appendix G. More specifically, for the experiments HF_M_S {1 – 5}, their training/validation accuracies and losses of each fold can be seen in Appendixes G.2.1 – G.2.5. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

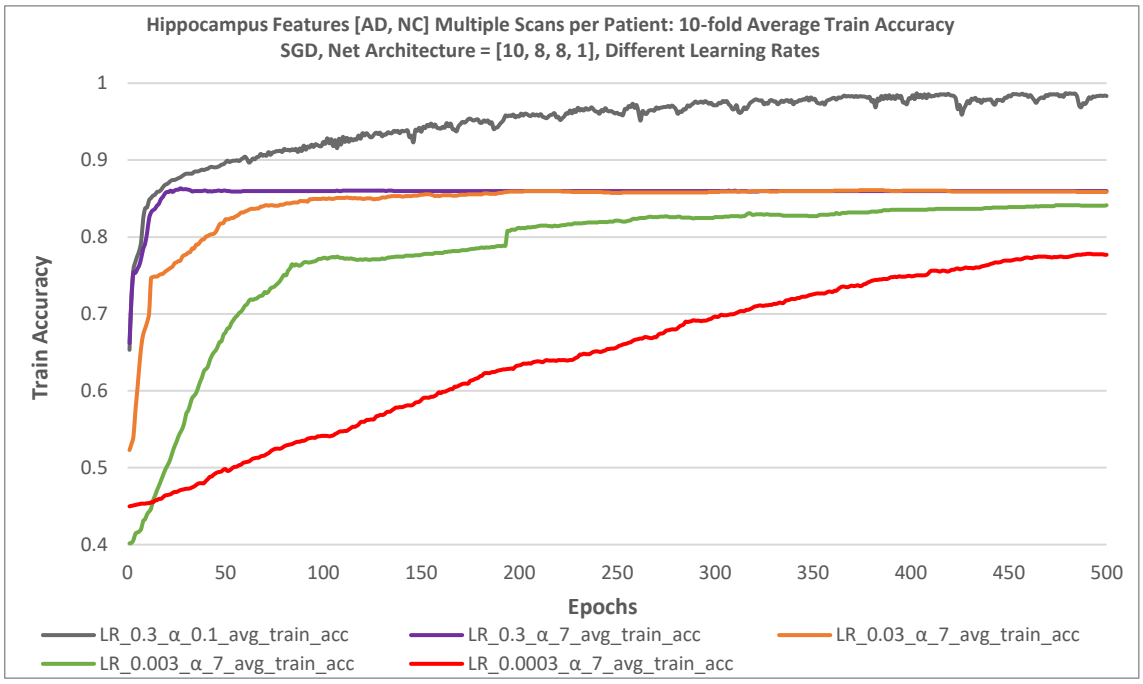


Figure 5.14. Average training accuracies of the 10-folds of the experiments HF_M_S {1 – 5}. Comparing different Learning Rates with the SGD optimizer.

We can see in Figure 5.14 that for α = 7, the larger the learning rate the better the training accuracy (HF_M_S {1 – 4}). The learning rate = 0.3 and the α = 0.1 in the experiment

HF_M_S5 helps to resolve the underfitting issues, since the training accuracy keeps increasing after the 30th epoch, with some minor fluctuations (Figure 5.14). As we can from the training losses plot (Figure 5.15), the experiment HF_M_S5 with a learning rate = 0.3 and α = 0.1, achieves the minimum training loss and a desirable reduction over time. On the other hand, for α = 7, the training loss does not decrease after a certain point.

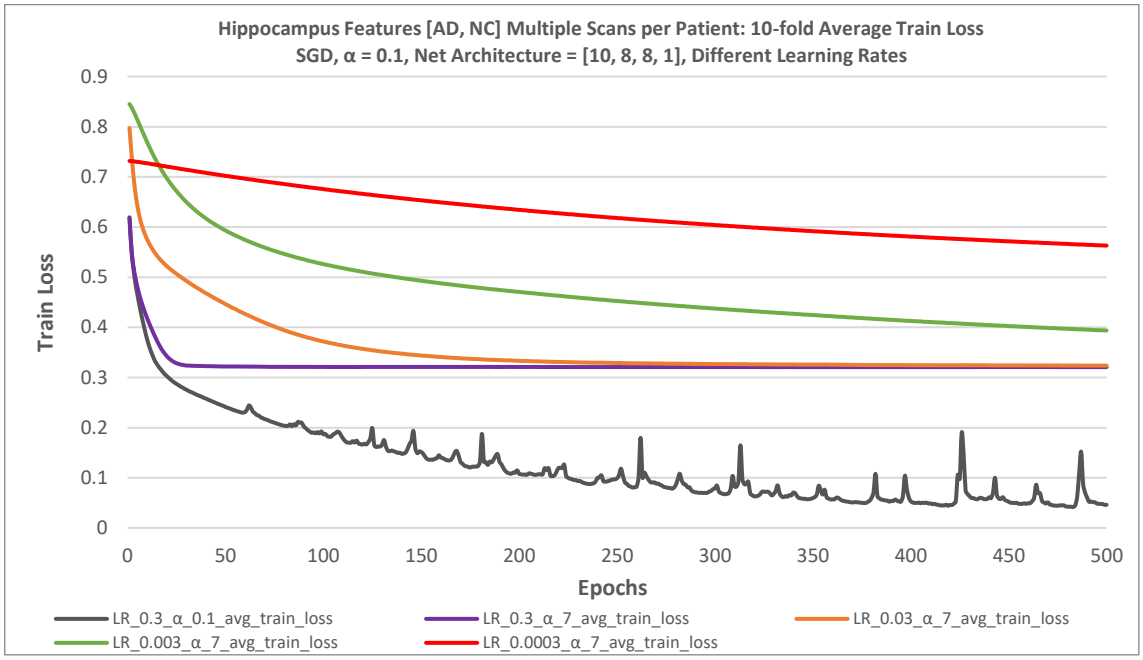


Figure 5.15. Average training losses of the 10-folds of the experiments HF_M_S {1 – 5}. Comparing different Learning Rates with the SGD optimizer.

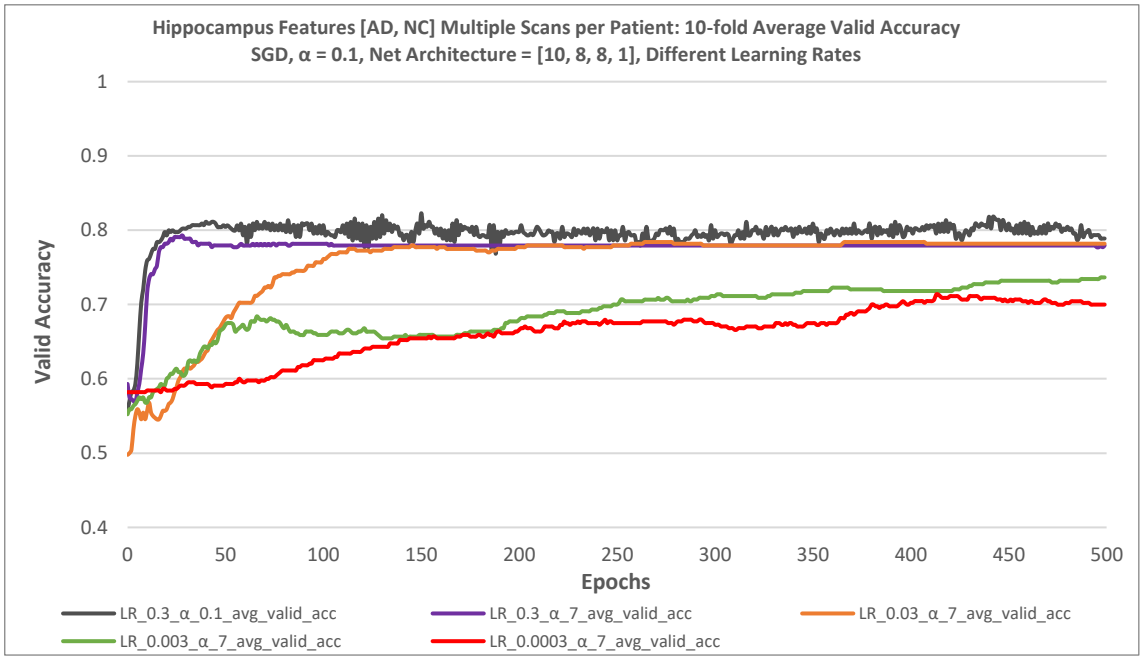


Figure 5.16. Average validation accuracies of the 10-folds of the experiments HF_M_S {1 – 5}. Comparing different Learning Rates with the SGD optimizer.

The average validation accuracies for the 10-folds of each experiment can be seen in Figure 5.16. The experiments HF_M_S {3, 4} with alpha = 7 and learning rates 0.003 and 0.0003 respectively, progress very slowly with the SGD optimizer. In Figure 5.16 we can also see that the experiment HF_M_S2 with the learning rate equal to 0.03, after approximately 100 epochs on average reaches its maximum value and sticks on that for the rest of the epochs. The experiment HF_M_S1 with the learning rate = 0.3 and $\alpha = 7$ (purple, Figure 5.16), reaches its maximum value around the 30th epoch, and then it decreases slightly where it gets stuck for the rest of the epochs. The experiment HF_M_S5 on the other hand, with the learning rate = 0.3 and $\alpha = 0.1$ (black, Figure 5.16) reaches its maximum value at approximately the 50th epoch and then remains relatively stable in a straight line while it fluctuates significantly more than the HF_M_S1.

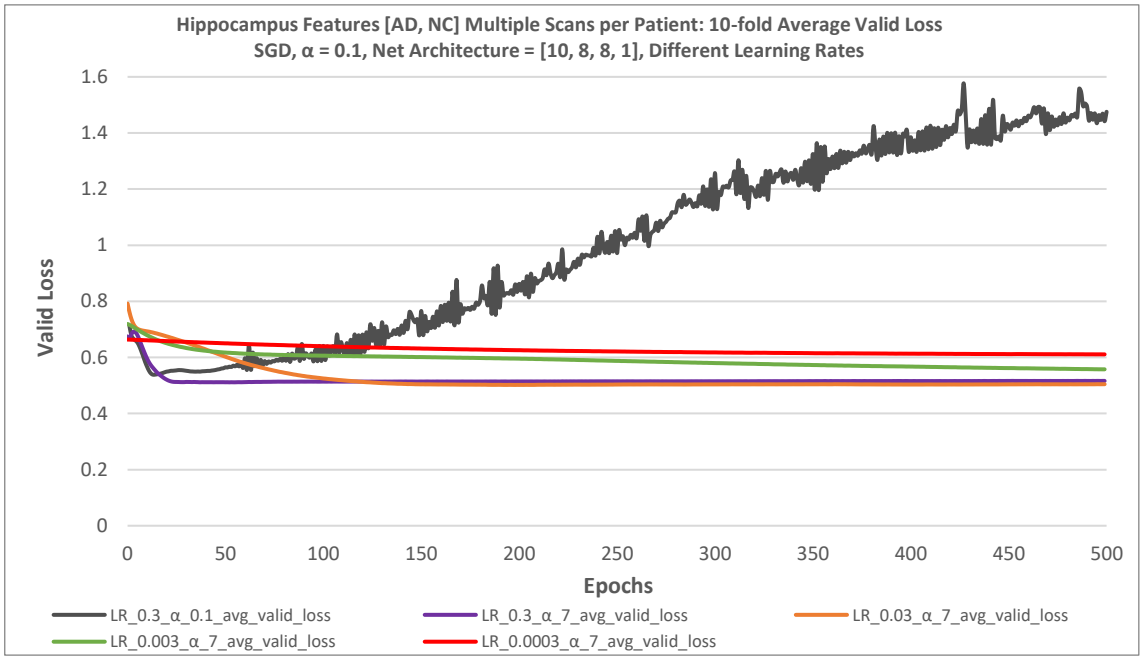


Figure 5.17. Average validation loss of the 10-folds of the experiments HF_M_S {1 – 5}. Comparing different Learning Rates with the SGD optimizer.

As we already know, small alphas help to fight the underfitting but they may cause overfitting (Pedregosa et al. 2011). Indeed, in these experiments, the small alpha of the experiment HF_M_S5 helped to improve the performance of the experiment HF_M_S1 (Figure 5.16). Nevertheless, if we observe the validation losses plot (Figure 5.17), we can see that the learning process of the model HF_M_S5 with the best performance (Table 5.5) cannot be considered "healthy". The model overfits a lot since the validation loss increases (black, Figure 5.17) while the training loss decreases (black, Figure 5.15). As

already mentioned in Adam's experiments (Section 5.2.2), where a similar overfitting problem occurred, adding another regularization technique such as Dropout (Section 2.3.13), is definitely needed.

5.2.4 MLP with HFO

5.2.4.1 Introduction to MLP with HFO

For the Hippocampus Features – Multiple scans per patient [AD, NC] dataset (HF_M) (Section 3.2.2), the 9 experiments HF_M_H {1 – 9} have been performed with the MLP classifier and the HFO optimizer. The implementations for the model used for these experiments can be found in Appendix A.3. The detailed training/validation accuracies and losses for each fold of these experiments can be found in Appendix G.3. The training, validation, and testing confusion matrices are also available in Appendix G.3.

Unlike the SGD (Section 5.2.3) and Adam (Section 5.2.2), the HFO algorithm does not have a learning rate or alpha hyperparameters. The hyperparameters that will be modified in the following experiments are the Conjugate Gradient (CG) iterations of the HFO algorithm (Section 2.4.7) and the network topology. The different network topologies are the same as the ones used for the Adam optimizer (Section 5.2.2.4). All HFO experiments are executed for 100 epochs while the SGD and Adam experiments were performed for 500 epochs.

In Adam's experiments, two groups of experiments, with similar behavior between the members of a group, were observed (Section 5.2.2). Contrariwise, in the HFO optimizer, which is very aggressive, all the following experiments behave very similarly. The MLP with HFO experiments performed worse than the SGD and the Adam for the AD/NC problem with the HF_M dataset since very rapidly they start overfitting. The reason could be the input itself, which is not appropriate for such an algorithm which is good for optimizing objectives that exhibit pathological curvature (Section 2.4.7.1) (Martens 2010).

The following 9 experiments HF_M_H {1 – 9} have very similar results in terms of accuracies, sensitivity, specificity, and the average epoch where the best validation accuracy was found. Most of the validation accuracies are 0.86, the average sensitivities are between 0.74 and 0.77, and the average specificities between 0.94 and 0.98 (Table

5.6). The experiment HF_M_H2 performs slightly better since it achieves a validation accuracy of 87%. From Table 5.6, we can see that the Specificity for the experiments is much larger than their Sensitivity. This means that the models are biased toward the NC patient since they correctly identify the NC patients as NC most of the time. This is an issue for us, since, for the AD/NC problem, the Sensitivity needs to be high, to correctly identify the AD patients as ADs. The reason this happens is the unbalanced training sets (Section 3.2.2) of the HF_M dataset.

Hippocampus Features – Multiple Scans per Patient [AD, NC] ***Optimizer: HFO***												
			Training				Validation					
Experiment Name	CG iterations	Network Architecture	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
HF_M_H1	1	[10, 30, 1]	0.91	0.04	0.80	0.95	0.86	0.05	0.75	0.98	34	0.91
HF_M_H2	2	[10, 30, 1]	0.93	0.04	0.82	0.97	0.87	0.05	0.76	0.98	23	0.93
HF_M_H3	4	[10, 30, 1]	0.87	0.07	0.77	0.91	0.86	0.05	0.75	0.97	12	0.91
HF_M_H4	8	[10, 30, 1]	0.92	0.09	0.83	0.95	0.86	0.05	0.75	0.96	28	0.91
HF_M_H5	16	[10, 30, 1]	0.93	0.06	0.83	0.97	0.86	0.05	0.75	0.97	26	0.91
HF_M_H6	32	[10, 30, 1]	0.90	0.10	0.84	0.92	0.86	0.05	0.77	0.94	16	0.91
HF_M_H7	2	[10, 20, 20, 1]	0.93	0.04	0.82	0.97	0.86	0.05	0.75	0.97	24	0.91
HF_M_H8	2	[10, 100, 1]	0.92	0.05	0.80	0.96	0.86	0.04	0.74	0.98	24	0.89
HF_M_H9	2	[10, 8, 8, 1]	0.94	0.04	0.85	0.97	0.86	0.05	0.74	0.98	31	0.91

Table 5.6. Hippocampus Features with Multiple Scans per Patient [AD, NC] with the HFO optimizer. Experiments with different CG iterations and network configurations.

5.2.4.2 Experiments HF_M_H {1 – 6}: CGiter = {1, 2, 4, 8, 16, 32}, net = [10, 30, 1]

The experiments HF_M_H {1 – 6} were meant to compare the effect of different Conjugate Gradient (CG) iterations in the HFO algorithm (Section 2.4.7.2). More specifically the experiments HF_M_H {1 – 6} with the values of 'CGiter' 1, 2, 4, 8, 16, and 32 respectively will be compared. All six implementations, use the same network configurations [10, 30, 1] (Section 4.2.2.3).

Detailed results for each experiment can be found in Appendix G. More specifically, for the experiments HF_M_H {1 – 6}, their training/validation accuracies and losses of each

fold can be seen in Appendixes G.3.1 – G.3.6. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

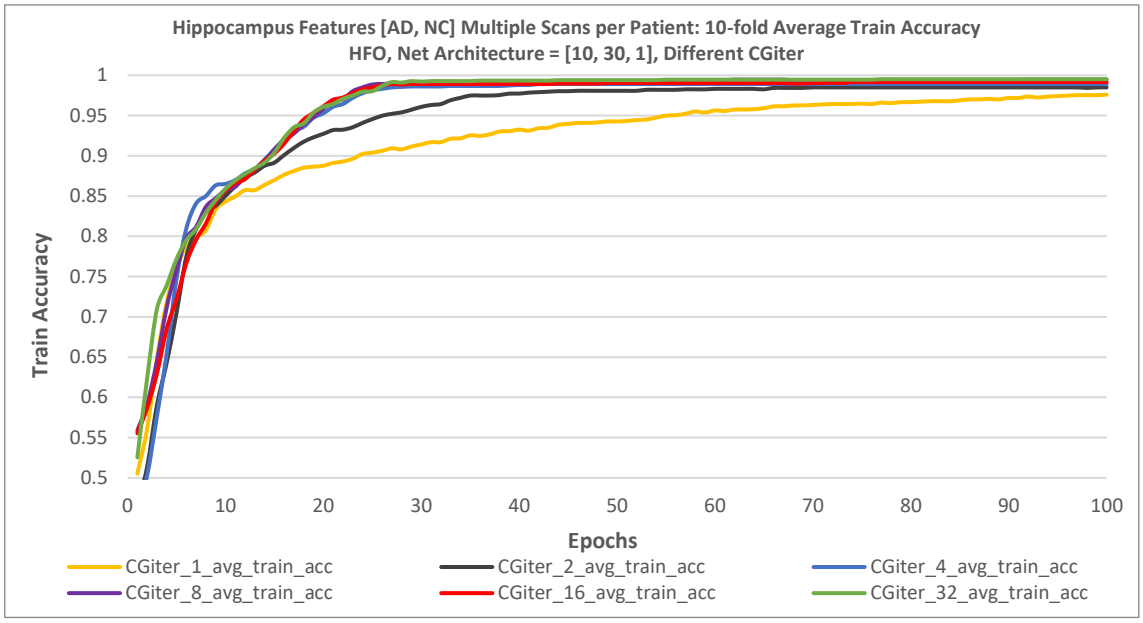


Figure 5.18. Average training accuracies of the 10-folds of the experiments HF_M_H {1 – 6}. Comparing different numbers of CG iterations with the HFO optimizer.

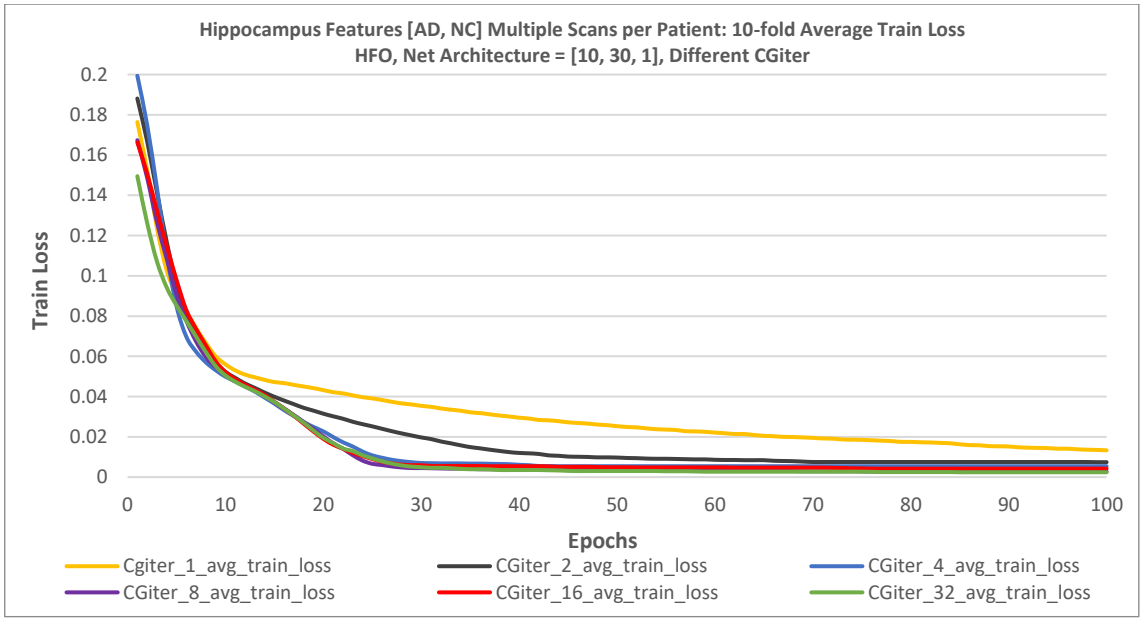


Figure 5.19. Average training losses of the 10-folds of the experiments HF_M_H {1 – 6}. Comparing different numbers of CG iterations with the HFO optimizer.

Figure 5.18, shows the average training accuracies of the 10-folds, of the experiments HF_M_H {1 – 6}. For the experiments HF_M_H1 and HF_M_H2 with CGiters = 1 (yellow) and CGiters = 2 (black) respectively, their training accuracies increase more stable than the rest of the CGiters (4, 8, 16, 32). For the experiments HF_M_H {3 – 6}

we observe that two bumps are visible, at the 10th and 20th epochs approximately. Keep in mind that those are the averages of the 10-folds; the training accuracy of each fold can be seen in the figures of Appendixes G.3.1 – G.3.6. In Figure 5.19, we can see the average training losses of the 10-folds, of the experiments HF_M_H {1 – 6}. We observe that the experiments HF_M_H {1 & 2} decline smoothly, while the experiments HF_M_H {3 – 6} create two valleys while they decline, at the 10th and 20th epochs.

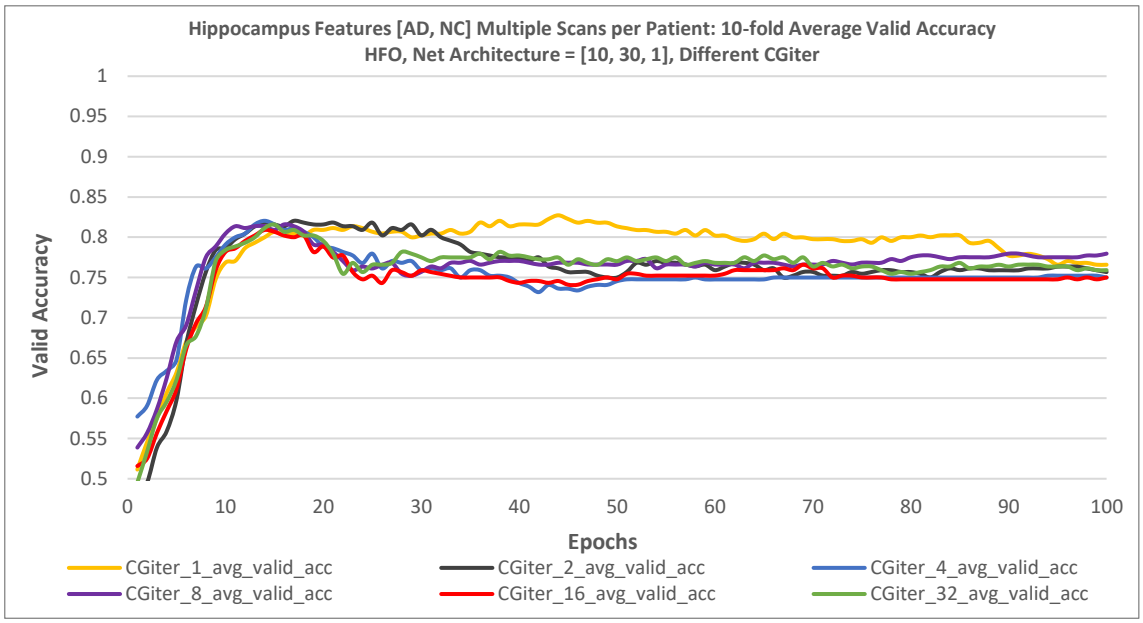


Figure 5.20. Average validation accuracies of the 10-folds of the experiments HF_M_H {1 – 6}. Comparing different numbers of CG iterations with the HFO optimizer.

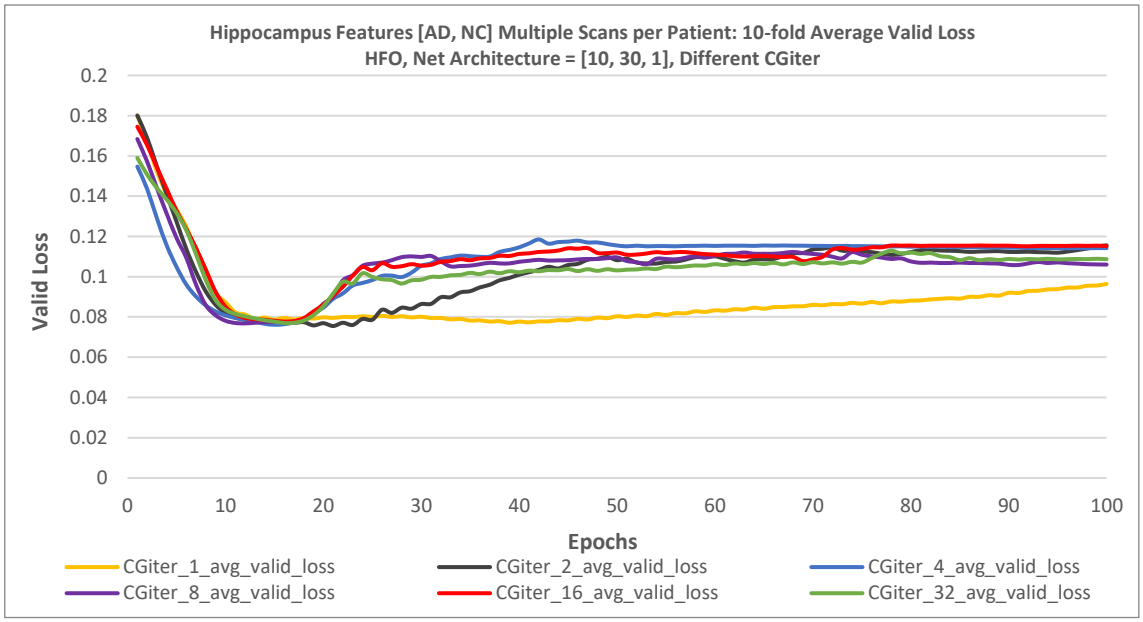


Figure 5.21. Average validation losses of the 10-folds of the experiments HF_M_H {1 – 6}. Comparing different numbers of CG iterations with the HFO optimizer.

Based on the average validation accuracies plot of the 10-folds for the experiments HF_M_H {1 – 6}, we can see that in all cases, a hill can be observed at the 15th epoch approximately (Figure 5.20). Afterward, the experiments with 'CGiter' 4, 8, 16, 32 face a steep decline, while the experiments with 'CGiter' 1 and 2 decline more slowly. This tells us that, whatever the value of the 'CGiter' is, soon or later, our model overfits.

In Figure 5.21, we observe that initially the average validation loss, of all 6 experiments declines, when a valley at around the 15th epoch starts to form. The experiments HF_M_H {3 – 6} soon after they rise again, while for the experiments HF_M_H {1 & 2} it takes more time. Since the training losses of all experiments decrease throughout the whole training process (Figure 5.19), we can safely assume that all the models overfit at some point based on their validation losses in Figure 5.12.

5.2.4.3 Experiments HF_M_H {2, 7 – 9}: CGiter = 2, net = {[10, 30, 1], [10, 20, 20, 1], [10, 100, 1], [10, 8, 8, 1]}

The experiments HF_M_H {2, 7 – 9} aim to compare the performance of different network configurations. More specifically, the network configurations for the experiments HF_M_H {2, 7 – 9} are [10, 30, 1], [10, 20, 20, 1], [10, 100, 1], and [10, 8, 8, 1] respectively (Section 4.2.2). Those are the same network configurations used also in the experiments of the MLP classifier with the Adam optimizer and the HF_M dataset (Section 5.2.2). The Conjugate Gradient (CG) iterations for all four experiments will be set to 2 since is the experiment HF_M_H2 had the highest validation accuracy between the experiments HF_M_H {1 – 6} (Table 5.6).

Detailed results for each experiment can be found in Appendix G. More specifically, for the experiments HF_M_H {2, 7 – 9}, their training/validation accuracies and losses of each fold are in Appendixes G.3.2, G.3.7 – G.3.9 respectively. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

In Figures 5.22 – 5.25 of the training and validation accuracy and loss, the results do not seem to vary a lot between the different network configurations. They look like are the same experiments of the same configuration and hyperparameters with just different initial weights than being completely different network architectures. In Figure 5.23 of the average validation accuracies of the 10-folds, of the HF_M_H {2, 7 – 9}, we can see

that the accuracies of all four experiments increase rapidly, when after the 20th epoch approximately starts to decline. This is an indication of overfitting.

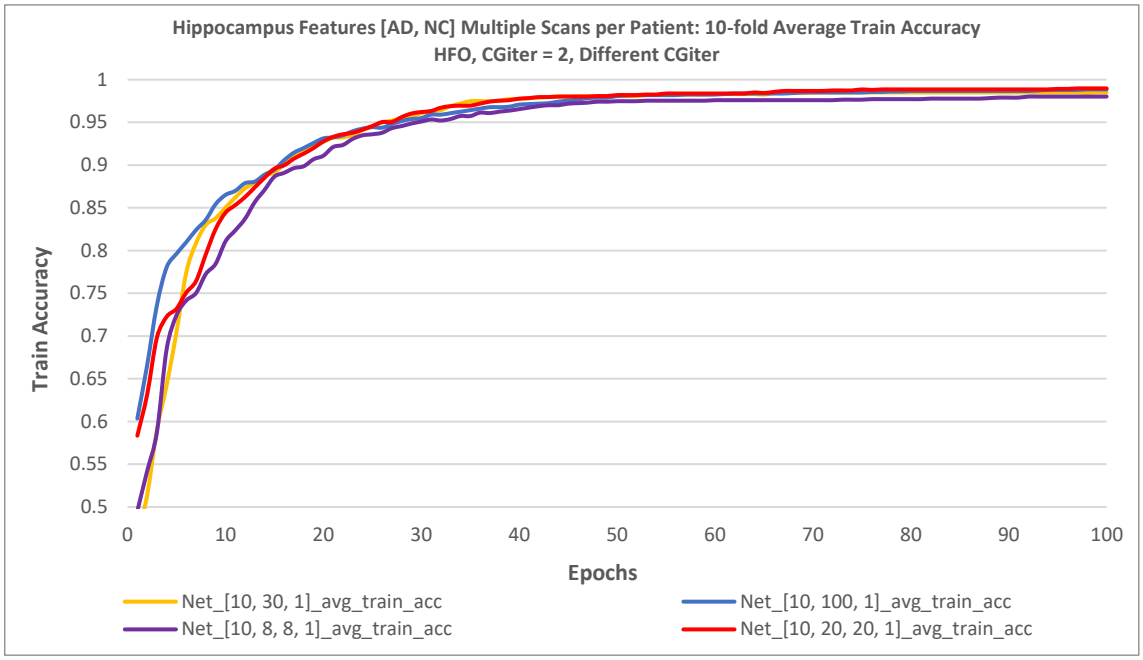


Figure 5.22. Average training accuracies of the 10-folds of the experiments HF_M_H {2, 7 – 9}. Comparing different network configurations with the HFO optimizer.

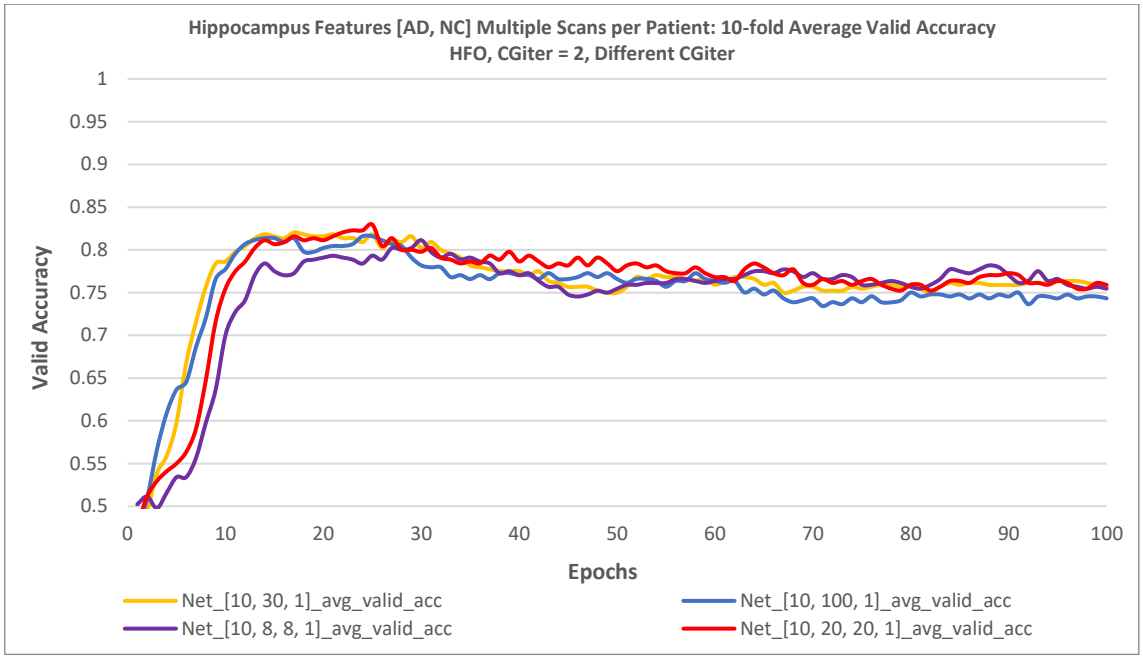


Figure 5.23. Average validation accuracies of the 10-folds of the experiments HF_M_H {2, 7 – 9}. Comparing different network configurations with the HFO optimizer.

By comparing Figure 5.24, with the average training loss of the experiments; with Figure 5.25, with the average validation loss, we can clearly see that our models overfit. This

happens, because the training accuracies decrease throughout the whole training process, while the validation accuracies, after the 20th epoch start increasing.

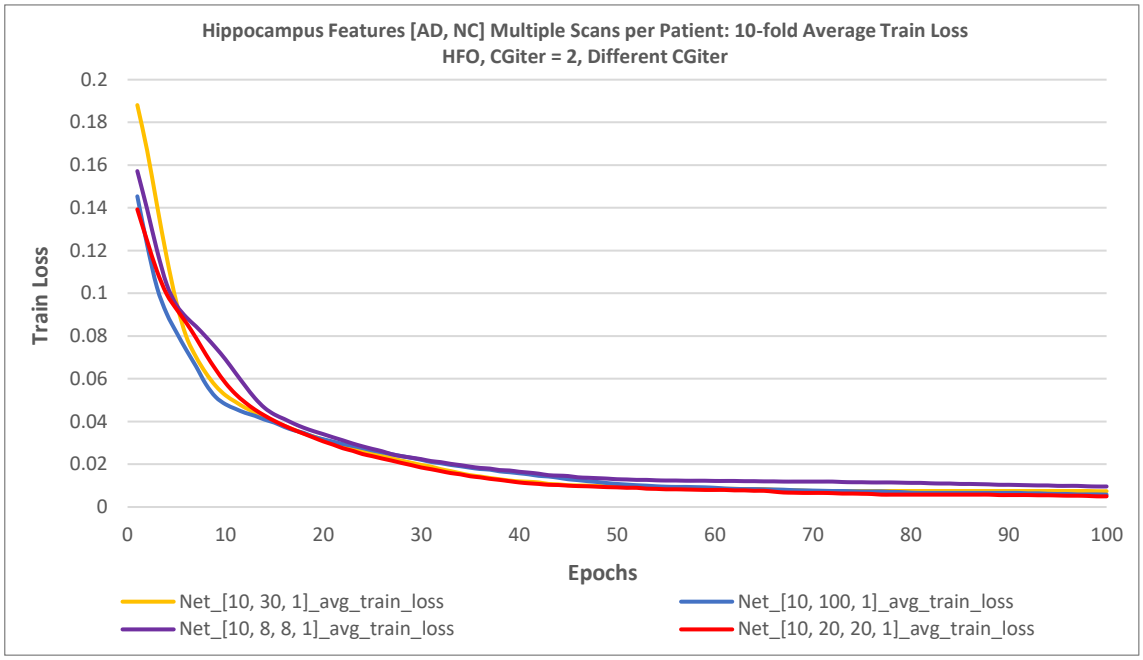


Figure 5.24. Average training losses of the 10-folds of the experiments HF_M_H {2, 7 – 9}. Comparing different network configurations with the HFO optimizer.

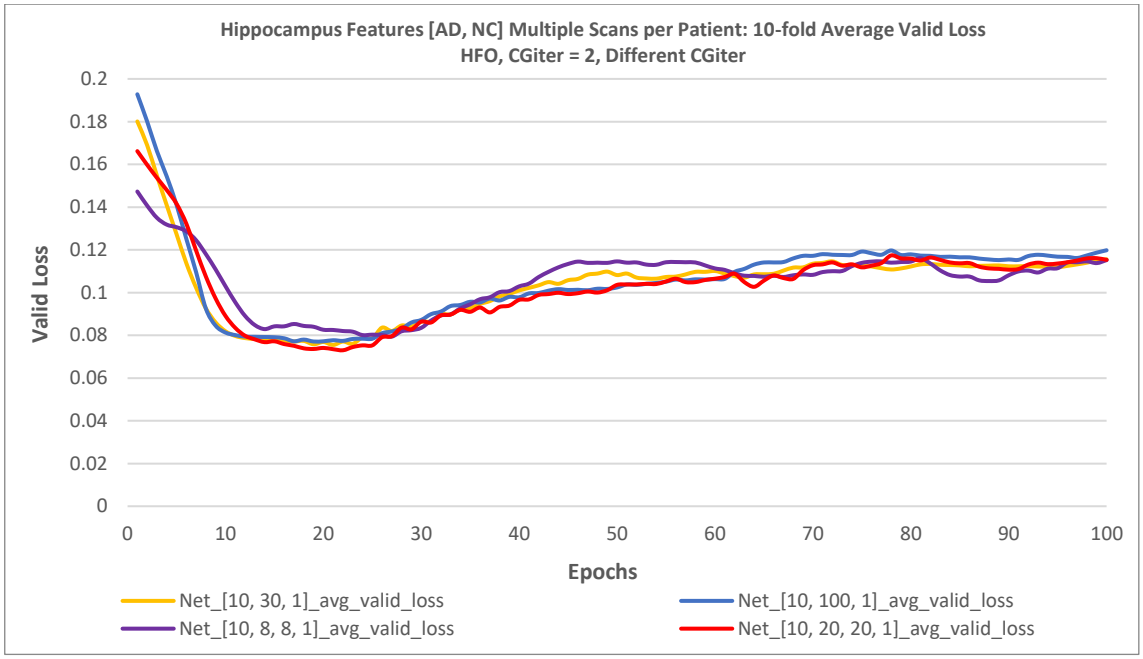


Figure 5.25. Average validation losses of the 10-folds of the experiments HF_M_H {2, 7 – 9}. Comparing different network configurations with the HFO optimizer.

The experiments HF_M_H {2, 7 – 9} have shown us that a more complex network configuration does not necessarily improve the performance or causes more overfitting

since all four network configurations in our case overfit similarly. Maybe if an even simpler network configuration is used, for example [10, 4, 1], could help to fix the issue with overfitting. We should keep in mind that the MLP classifier with HFO, is a vanilla implementation as Martens (2010) and Martens and Sutskever (2012) proposed. Therefore, no regularization techniques, such as Dropout, L1, or L2 Regularization was applied to encounter overfitting (Section 2.3.13). If some kind of regularization is applied, we are expecting a better performance than the ones with SGD or Adam in fewer epochs.

5.3 2D Brain Slices [AD, NC]

In Sections 5.4 – 5.7, 39 experiments will be compared based on the 2D Brain Slices [AD, NC] datasets (Section 3.2.3). More specifically, the four datasets that are going to be tested are the Single Scan per Patient – Single Slice per Scan (B_2D_S) (Section 3.2.3.2), Single Scan per Patient – 5 Slices per Scan (B_2D_5S) (Section 3.2.3.3), Multiple Scans per Patient – Single Slice per Scan (B_2D_M) (Section 3.2.3.4), and Multiple Scans per Patient – 7 Slices per Scan (B_2D_7M) (Section 3.2.3.5). The experiments are going to be performed by using the modified CNN implementations of Wang et al. (2020), as described in Section 4.3.1. The optimizers that are going to be used are Adam and the NewtonCG which is the HFO implementation for the CNN network with the Gauss-Newton matrix from Wang et al. (2020) (Section 2.4.9)

The activation function used for all the experiments in Sections 5.4 – 5.7, is ReLU (Section 2.2.13). The Sigmoid/Logistic function (Section 2.2.12) was also being used but its results were catastrophic since the model underfit as the validation accuracy was approximately 50%. In future work, different activation functions can be used, for potentially better performance, such as Leaky ReLU.

In most of the experiments, Mean Squared Error (MSE) loss function was used (Section 2.3.4). Only experiment B_2D_M_N16 used the Cross-entropy as a loss function but its performance was slightly worse than the MSE (Section 5.4.2.7).

Different size of filters has been used, such as 3×3, 5×5, and 7×7. The filter size 5×5 performed better in the 2D experiments (Section 5.4.2.11), while for the 3D ones, the $3 \times 3 \times 3$ was a better choice. Also, for the experiments with the 2D datasets was better to apply Max-Pooling to all Convolutional Layers (Section 5.4.2.6), while in the

experiments with the 3D datasets is better not to apply Max-Pooling at all (Section 5.8.1.2).

Overfitting was a common issue in most of the experiments, especially the ones with the NewtonCG optimizer. Different hyperparameters are being modified to reduce overfitting which is going to be discussed in each section later on. Some of the techniques against overfitting that have been applied are the L1 & L2 Regularization, Dropout, and Spatial Dropout, but none of these helped to decrease significantly the issue. The reduction of Gauss-Newton matrix size (GNsize) seems to be helping against overfitting, but it reduces the validation accuracy as well.

Since many network configurations such as deep, shallow, narrow, and wide (Section 4.3.2) had been used to reduce overfitting and improve the model's performance, with no much effect; the issue could be the small size of the datasets (Sections 3.2.3). The small datasets, resulting in a small training set which does not help the model to generalize. Therefore, if data augmentation is applied, and the size of the datasets is increased to approximately 10,000 – 20,000 samples, this could significantly help to improve the performance of the models.

It is worth mentioning that from the 2D slice-level MRI datasets, the B_2D_M (Section 3.2.3.4) achieves the best validation accuracies among the B_2D_S (Section 3.2.3.2), B_2D_5S (Section 3.2.3.3), and B_2D_7M (Section 3.2.3.5) datasets. This does not mean that we reject the rest of the datasets. Nevertheless, the dataset B_2D_7M reduces significantly the standard deviation between the accuracies of the different folds. This makes the experiment B_2D_7M_N1 (best of B_2D_7M dataset) more reliable since the minimum best validation accuracy between the 10-folds is 73% while the maximum is 78%. So, the possible validation accuracies are in the 5% range. On the other hand, for the experiment B_2D_M_N19 (best of B_2D_M dataset) the minimum best validation accuracy of the 10-folds is 71%, while the maximum is 90%. Thus, the possible validation accuracies are in a much larger 19% range. To understand how severe this issue is, we can observe the plots of their validation accuracies for each fold. The deviation between the validation accuracies of the experiments B_2D_M_N19 (Figure H.185), is much larger at any given point, in comparison with the experiment's B_2D_7M_N1 (Figure J.15). In future experiments, the use of the B_2D_7M dataset, after applying data augmentation, is highly suggested.

2D Brain Slices – Single / Multiple Scans per Patient – Single / 5 / 7 Slices per Scan [AD, NC]																
	Training				Validation						Testing					
Experiment Name	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
Multiple Scans per Patient – Single Slice per Scan																
Adam																
B_2D_M_A1	0.53	0.07	0.59	0.47	0.56	0.07	0.66	0.65	0.61	0.50	0.51	0.05	0.58	0.44	13	0.74
B_2D_M_A2	0.87	0.06	0.87	0.86	0.79	0.07	0.80	0.79	0.78	0.80	0.60	0.06	0.63	0.58	246	0.90
B_2D_M_A3	0.91	0.07	0.90	0.92	0.79	0.08	0.78	0.81	0.82	0.77	0.64	0.05	0.75	0.53	154	0.91
NewtonCG																
B_2D_M_N1	0.88	0.04	0.89	0.87	0.78	0.08	0.77	0.79	0.80	0.75	0.61	0.03	0.71	0.52	48	0.90
B_2D_M_N2	0.87	0.04	0.87	0.87	0.76	0.08	0.75	0.77	0.77	0.74	0.62	0.06	0.72	0.51	38	0.90
B_2D_M_N3	0.91	0.04	0.90	0.91	0.77	0.07	0.78	0.78	0.78	0.77	0.62	0.03	0.69	0.55	51	0.86
B_2D_M_N4	0.89	0.05	0.91	0.87	0.75	0.10	0.75	0.78	0.79	0.72	0.63	0.04	0.72	0.53	48	0.90
B_2D_M_N5	0.89	0.04	0.90	0.88	0.75	0.08	0.75	0.76	0.77	0.74	0.61	0.04	0.69	0.52	47	0.87
B_2D_M_N6	0.88	0.05	0.88	0.88	0.76	0.09	0.77	0.76	0.75	0.78	0.60	0.04	0.69	0.51	38	0.90
B_2D_M_N7	0.90	0.05	0.91	0.90	0.76	0.08	0.76	0.77	0.77	0.74	0.62	0.05	0.71	0.53	56	0.86
B_2D_M_N8	0.90	0.09	0.90	0.91	0.78	0.09	0.79	0.79	0.79	0.77	0.60	0.04	0.69	0.52	9	0.91
B_2D_M_N9	0.93	0.05	0.94	0.92	0.76	0.08	0.76	0.78	0.78	0.74	0.62	0.05	0.71	0.53	14	0.89
B_2D_M_N10	0.84	0.07	0.84	0.83	0.80	0.05	0.82	0.80	0.79	0.82	0.61	0.06	0.67	0.54	20	0.91
B_2D_M_N11	0.89	0.06	0.87	0.91	0.78	0.07	0.82	0.76	0.74	0.82	0.60	0.06	0.64	0.55	12	0.89
B_2D_M_N12	0.90	0.04	0.89	0.90	0.78	0.07	0.78	0.78	0.77	0.78	0.64	0.05	0.75	0.52	10	0.90
B_2D_M_N13	0.84	0.06	0.86	0.82	0.78	0.08	0.78	0.80	0.79	0.77	0.60	0.04	0.67	0.53	13	0.89
B_2D_M_N14	0.91	0.12	0.89	0.93	0.71	0.10	0.69	0.75	0.76	0.66	0.62	0.06	0.67	0.56	37	0.85
B_2D_M_N15	0.77	0.04	0.78	0.76	0.79	0.08	0.79	0.80	0.79	0.78	0.63	0.07	0.66	0.62	12	0.91
B_2D_M_N16	0.79	0.03	0.80	0.79	0.77	0.08	0.77	0.77	0.78	0.78	0.64	0.05	0.67	0.63	9	0.90
B_2D_M_N17	0.85	0.04	0.85	0.86	0.78	0.08	0.80	0.77	0.78	0.80	0.61	0.03	0.63	0.61	23	0.90
B_2D_M_N18	0.82	0.05	0.84	0.79	0.79	0.08	0.79	0.80	0.80	0.78	-	-	-	-	17	0.91
B_2D_M_N19	0.82	0.04	0.83	0.80	0.81	0.06	0.81	0.81	0.80	0.81	0.63	0.03	0.73	0.52	17	0.90
B_2D_M_N20	0.83	0.10	0.81	0.84	0.79	0.06	0.80	0.78	0.76	0.81	0.62	0.04	0.70	0.54	23	0.86
B_2D_M_N21	0.84	0.04	0.83	0.87	0.79	0.08	0.84	0.73	0.76	0.82	0.60	0.05	0.62	0.59	14	0.91
B_2D_M_N22	0.85	0.03	0.87	0.84	0.78	0.08	0.75	0.81	0.81	0.78	0.63	0.04	0.67	0.61	17	0.89
B_2D_M_N23	0.79	0.09	0.81	0.79	0.78	0.09	0.78	0.78	0.79	0.79	0.60	0.07	0.62	0.58	19	0.91
B_2D_M_N24	0.81	0.04	0.84	0.79	0.76	0.07	0.70	0.82	0.80	0.74	0.62	0.06	0.65	0.61	13	0.86
B_2D_M_N25	0.78	0.04	0.81	0.77	0.75	0.08	0.71	0.79	0.78	0.73	0.63	0.05	0.65	0.62	12	0.88

B_2D_M_N26	0.79	0.06	0.80	0.80	0.77	0.07	0.76	0.77	0.78	0.77	0.61	0.05	0.64	0.60	12	0.85
B_2D_M_N27	0.80	0.07	0.80	0.81	0.75	0.06	0.77	0.73	0.75	0.77	0.59	0.06	0.60	0.59	17	0.83
B_2D_M_N28	0.93	0.06	0.93	0.93	0.77	0.09	0.77	0.77	0.78	0.76	0.60	0.05	0.69	0.50	115	0.90
Multiple Scans per Patient – 7 Slices per Scan																
Adam																
B_2D_7M_A1	0.92	0.06	0.93	0.91	0.75	0.02	0.74	0.75	0.76	0.74	0.64	0.03	0.67	0.60	49	0.78
NewtonCG																
B_2D_7M_N1	0.94	0.04	0.94	0.94	0.76	0.02	0.75	0.77	0.77	0.75	0.64	0.02	0.69	0.58	32	0.78
B_2D_7M_N2	0.94	0.04	0.95	0.93	0.73	0.01	0.73	0.74	0.74	0.72	0.66	0.04	0.68	0.64	34	0.75
B_2D_7M_N3	0.96	0.02	0.97	0.96	0.73	0.02	0.72	0.74	0.75	0.71	0.66	0.02	0.69	0.63	40	0.76
Single Scan per Patient - Single Slice per Scan																
NewtonCG																
B_2D_S_N1	0.85	0.12	0.86	0.84	0.75	0.07	0.75	0.77	0.77	0.74	0.68	0.12	0.68	0.68	22	0.87
B_2D_S_N2	0.90	0.10	0.89	0.90	0.73	0.06	0.75	0.74	0.74	0.73	0.72	0.09	0.72	0.71	41	0.84
B_2D_S_N3	0.85	0.09	0.85	0.86	0.77	0.05	0.79	0.76	0.75	0.78	0.70	0.14	0.62	0.78	23	0.84
Single Scan per Patient - 5 Slices per Scan																
NewtonCG																
B_2D_5S_N1	0.92	0.04	0.90	0.95	0.75	0.03	0.77	0.74	0.73	0.77	0.69	0.05	0.65	0.74	31	0.78

Table 5.7. 2D Brain Slices – Single / Multiple Scans per Patient – Single / 5 / 7 Slices per Scan [AD, NC]. Different experiments training, validation, and testing performance metrics.

5.3.1 Best CNN Model for the AD/NC problem (B_2D_M_N19)

The experiment B_2D_M_N19 had the best performance results (best average validation accuracy) among all the experiments with CNNs for the AD/NC problem. Its average validation accuracy, PPV, NPV, sensitivity, and specificity for 10-folds, are 81%, 81%, 81%, 80%, 81% respectively. The maximum validation accuracy for the 10-folds is 90%, and the standard deviation for the validation accuracy is 6%. The average epoch of the best validation accuracy for the 10-folds is 17. This means that after 17 epochs on average, the best model of each fold occurred with the highest validation accuracy (Table 5.7).

The experiment B_2D_M_N19 uses the NewtonCG optimizer (Section 2.4.9), as implemented by Wang et al. (2020), with the 2D_CNN_4L_2 network topology, which has 3 Convolutional Layers, a single dense layer, filters of size 5×5, and the filters per Convolutional Layer are f1 = 32, f2 = 32, f3 = 64, with fi being the filters in the i^{th} Convolutional Layer (Section 4.3.2.3.2). Dropout is also being applied with probabilities d1 = 0.3, d2 = 0.5 and d3 = 0.5 with di being the dropout probabilities in the i^{th} Convolutional Layer (Section 2.3.13.9). The regularization term or so-called weight

decay (C) was set to 0.01, and the size of the Gauss-Newton matrix (GNsize) equal to 50 (Section 4.3.1). The experiment was executed for 100 epochs with the MSE loss function.

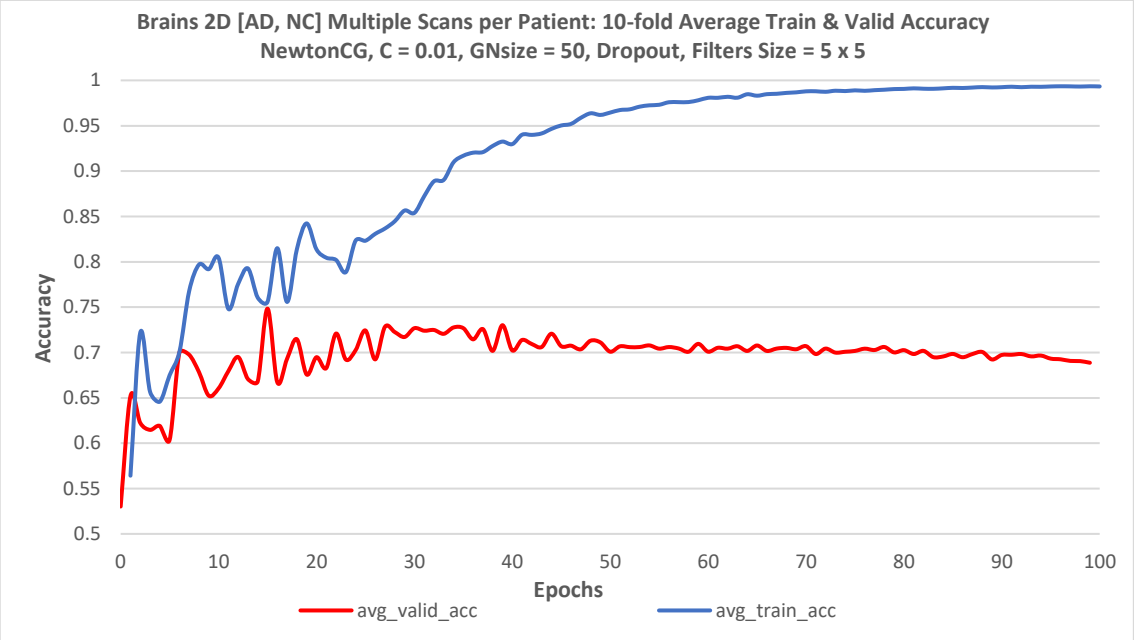


Figure 5.26. Average training and validation accuracies of the 10-folds, of the experiment B_2D_M_N19.

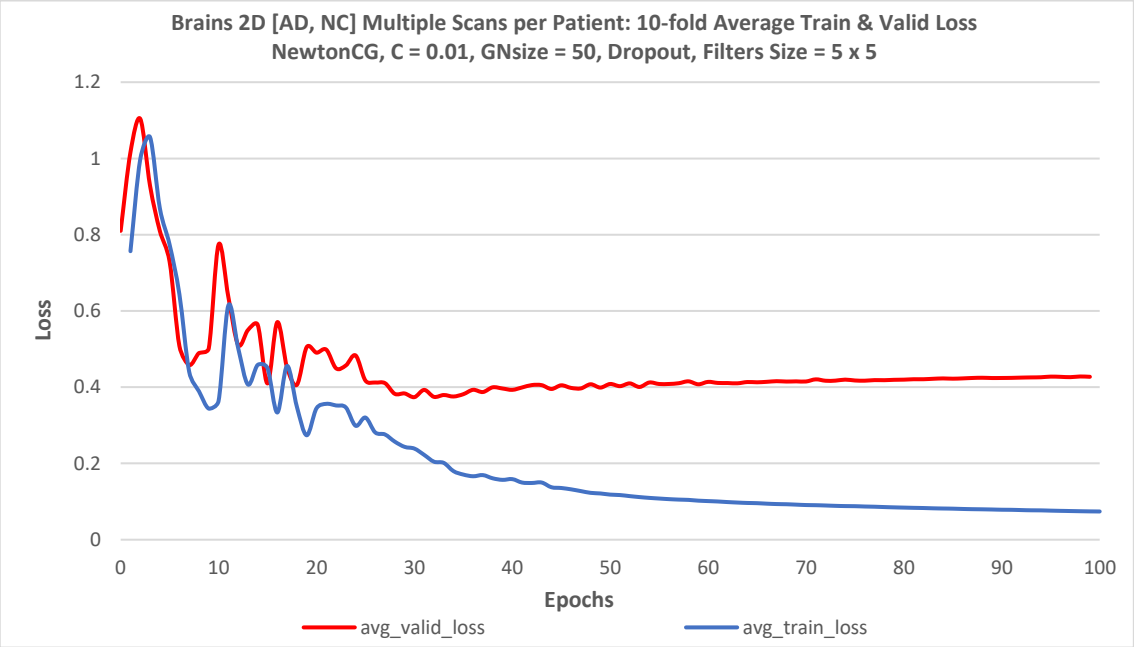


Figure 5.27. Average training and validation losses of the 10-folds, of the experiment B_2D_M_N19.

In the average accuracies plot of the 10-folds, of the B_2D_M_N19 experiment (Figure 5.26), we can see that both the training and validation accuracies fluctuate a lot the first 30 epochs approximately; while later on the training accuracy increases rapidly as the validation accuracy decreases smoothly; which indicates overfitting. From the average

losses plot of the 10-folds, of the B_2D_M_N19 experiment (Figure 5.27), we can see that after the 30th epoch, overfitting can be observed since the validation loss increases slightly, while the training loss continues to decrease.

5.3.1.1 Training

In the accuracy and loss plots for each fold (Figure 5.28 & 5.29), all 10-folds fluctuate a lot at the beginning, but later on, they converged and behave very similarly.

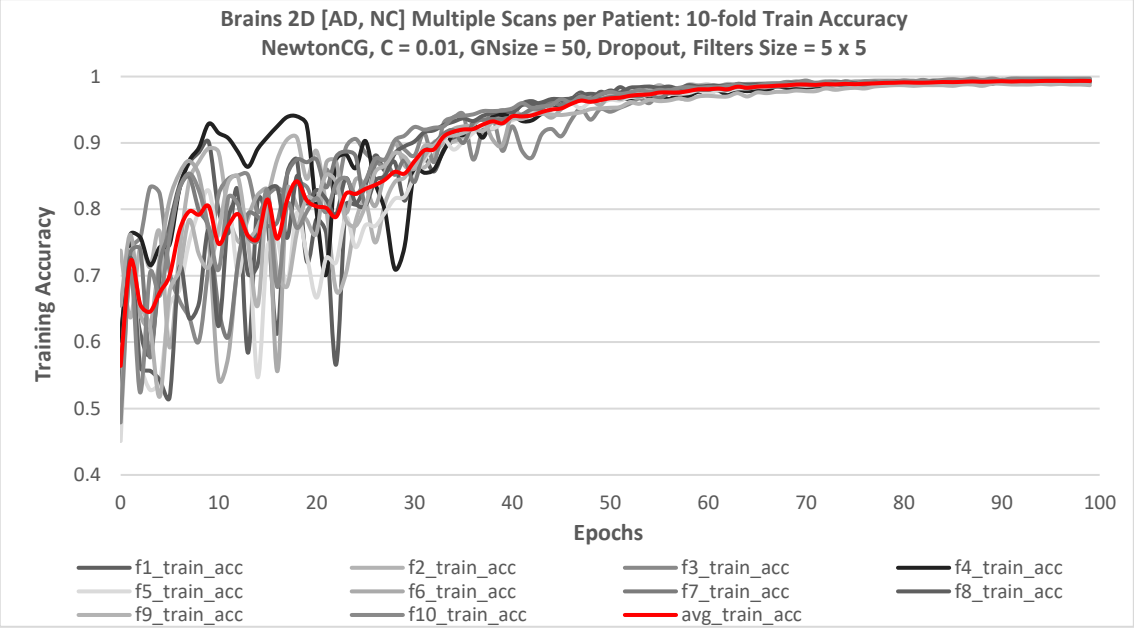


Figure 5.28. Training accuracy of each fold, of the B_2D_M_N19 experiment.

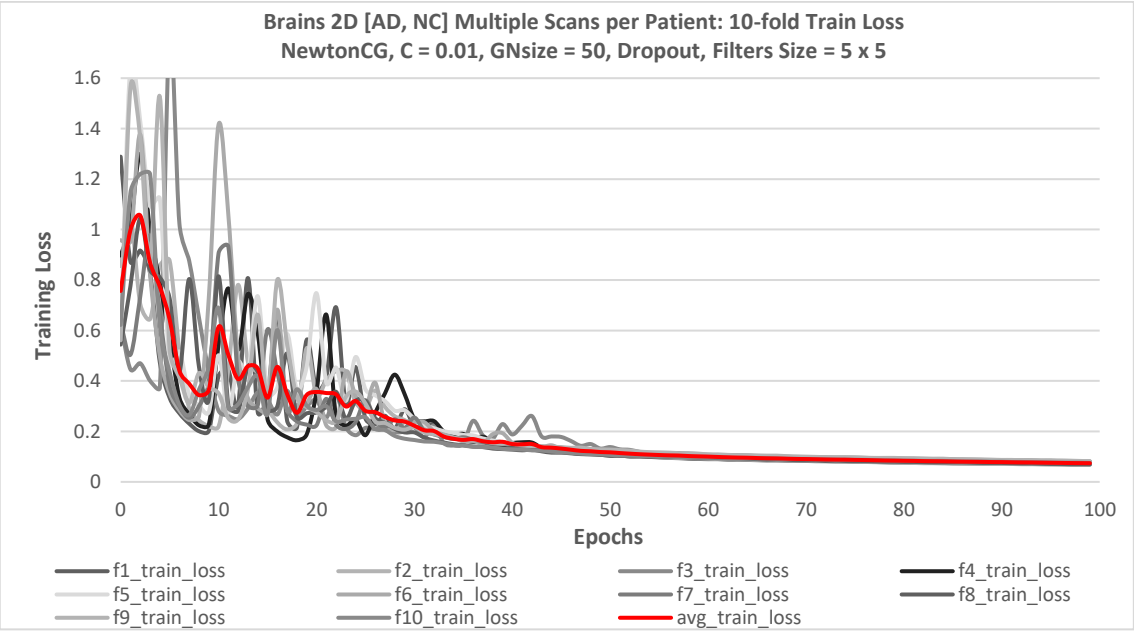


Figure 5.29. Training loss of each fold, of the B_2D_M_N19 experiment.

The training confusion matrix in Figure 5.30, shows the average TP, TN, FP, and FN of the 10-folds for the training set. From the 522 ADs in the training set, 103 samples on average were misclassified as NCs (False Positive) during training.

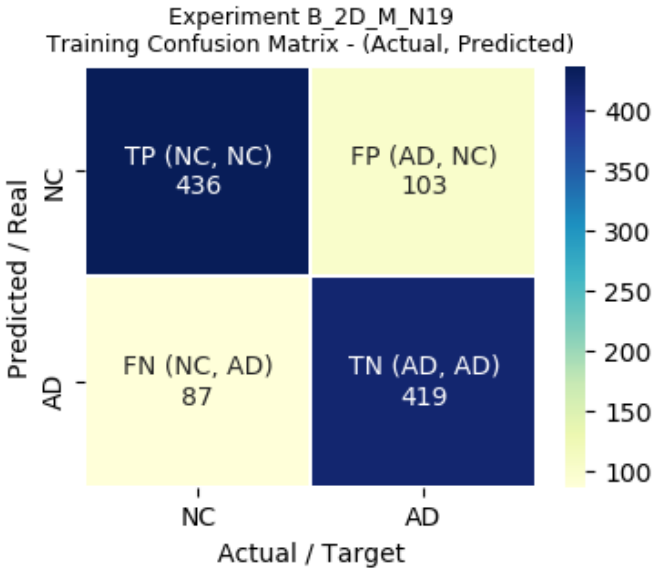


Figure 5.30. Training confusion matrix of the experiment B_2D_M_N19.

5.3.1.2 Validation

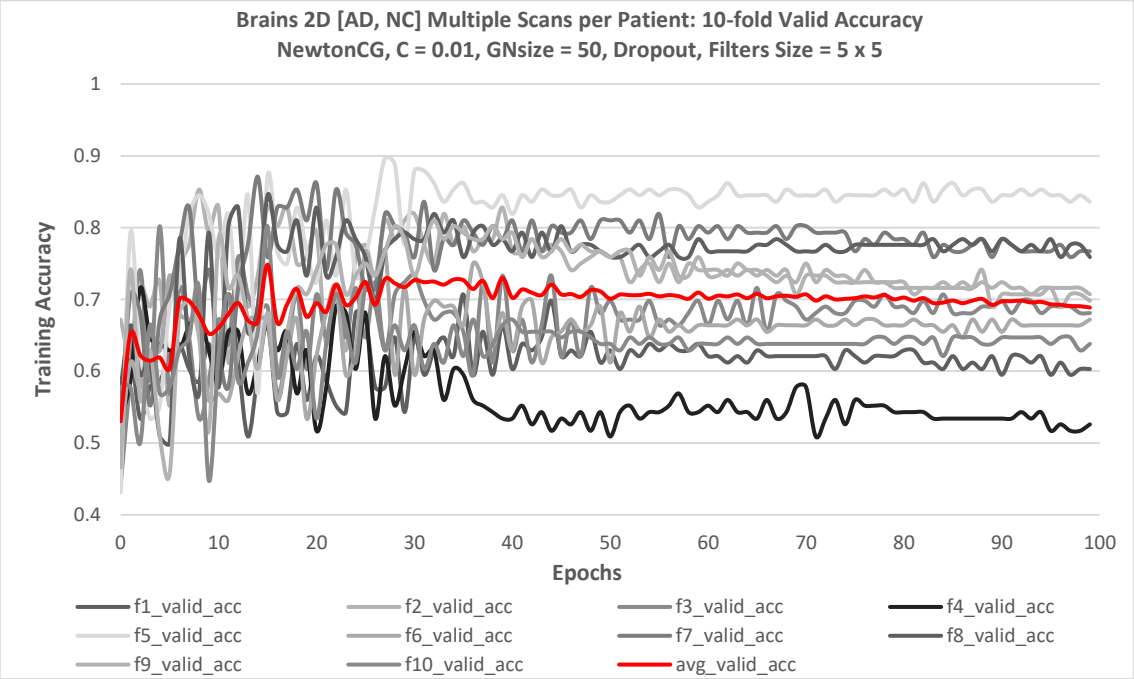


Figure 5.31. Validation accuracy of each fold, of the B_2D_M_N19 experiment.

In Figure 5.31, we can see that after the 30th epoch approximately, the validation accuracy of each fold takes a path and follows it till the last. Some folds are doing great while

others do not; for example, the 5th fold (lightest gray, Figure 5.31) fluctuates between 0.8 and 0.9, while the 4th fold (darkest gray, Figure 5.31) fluctuates between 0.5 and 0.6.

This issue is not unique to this experiment only but is present in all the experiments of the same B_2D_M dataset. Both the B_2D_S (Section 3.2.3.2), and B_2D_M (Section 3.2.3.4) datasets suffer from this high deviation between folds because their datasets are relatively small. The B_2D_5S (Section 3.2.3.3), and B_2D_7M (Section 3.2.3.5) datasets, which are 5 and 7 times larger respectively from the B_2D_S and B_2D_7M datasets, resolve this issue.

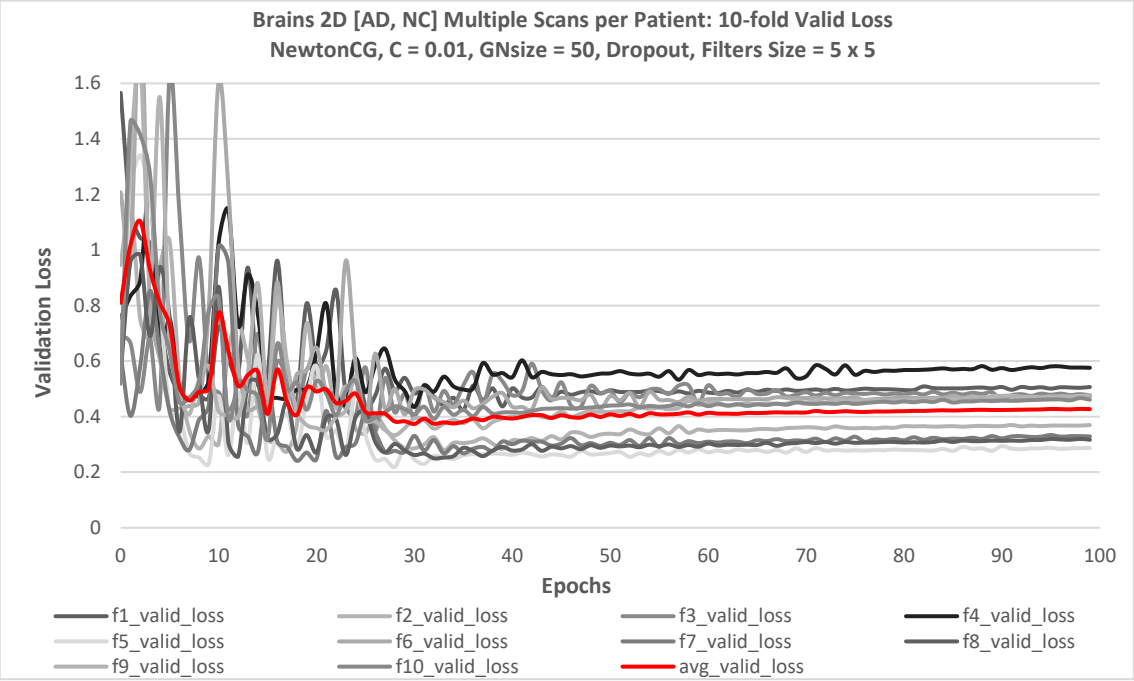


Figure 5.32. Validation loss of each fold, of the B_2D_M_N19 experiment.

The average epoch of the best validation accuracy between folds for the B_2D_M_N19 experiment is 19 (Table 5.7). This indicates that the model does not improves over time since the rest of the accuracies are worse or equal to the best accuracy in around the 19th epoch. This is not a good sign, since our model overfits, as validation loss increased is observed (Figure 5.32). To potentially fix this issue, we need to increase the size of our dataset by applying data augmentation (Section 2.3.13.4 & 2.3.13.5).

From the validation accuracy confusion matrix (Figure 5.33), we can see that on average 11 samples out of 58 ADs are misclassified as NCs (False Positive).

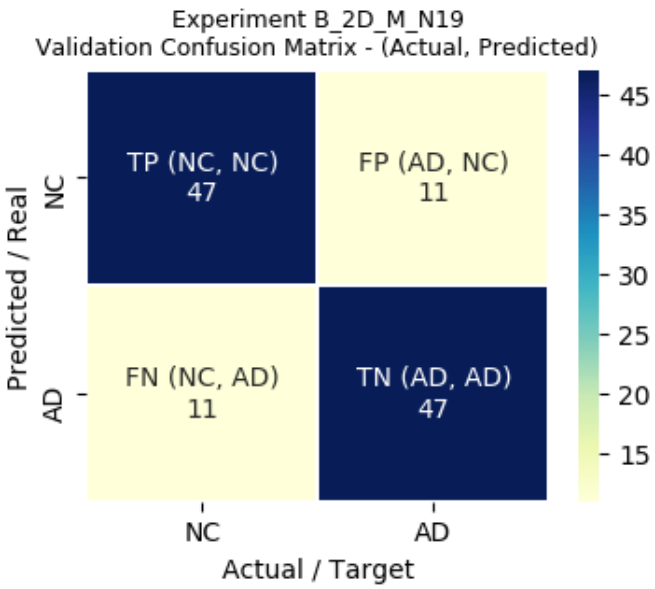


Figure 5.33. Validation confusion matrix of the experiment B_2D_M_N19.

5.3.1.3 Testing

The testing confusion matrix (Figure 5.34) shows that on average 11 out of 22 ADs are misclassified as NCs (False Positive). This means that on average the 50% of the patients were classified as healthy while they were not. This is a very poor performance for a model since the prediction is just random. The reason why the testing performs so poorly is that the test set is very small (Beleites et. al. 2013). In future experiments, it is highly suggested for the tested to be much larger for more accurate performance metrics.

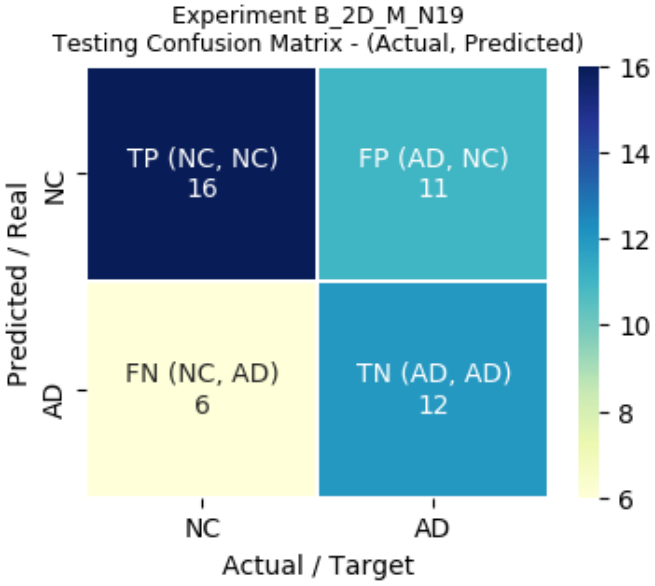


Figure 5.34. Testing confusion matrix of the experiment B_2D_M_N19.

5.3.1.4 Confusion Matrix Concentration

Figures 5.35 & 5.36 show the training and validation confusion matrices of concentration respectively, of the B_2D_M_N19 experiment. The figures show the average percentages of TP, TN, FP, and FN for the 10-folds at any given epoch of the training and validation respectively. We can see from the training concentration (Figure 5.35) that large fluctuations exist in the first 30 epochs; then, the TP and TN increase smoothly while FP and FN decrease, which is very desirable.

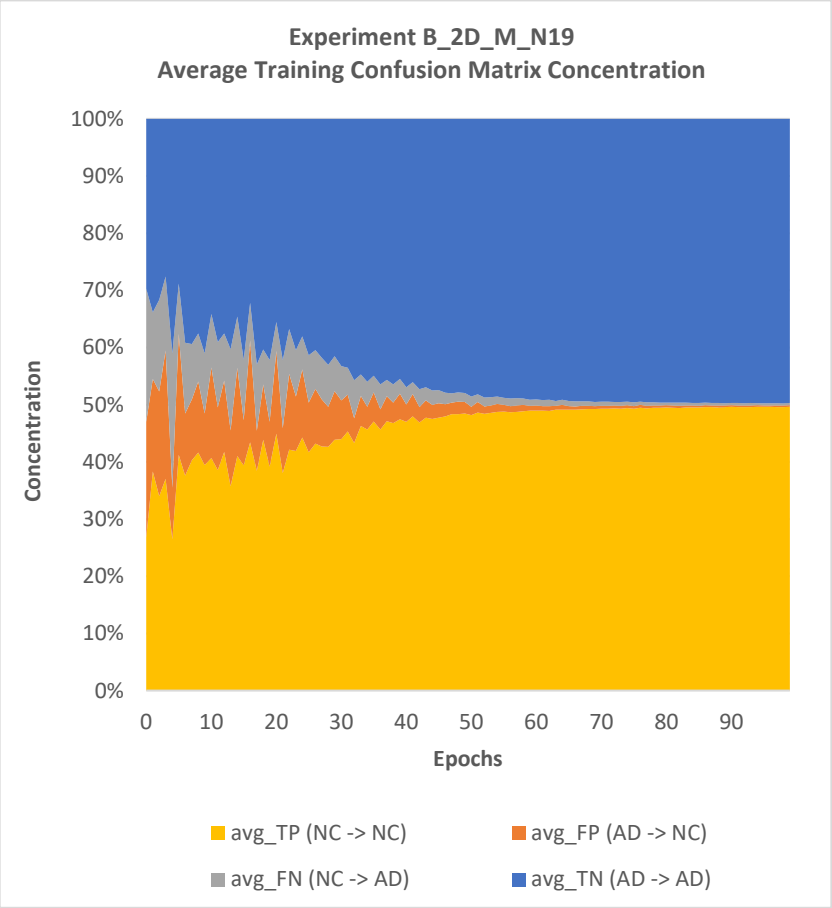


Figure 5.35. Average training confusion matrix of concentration of the experiment B_2D_M_N19.

As we can see from the validation concentration (Figure 5.36), similar to the training concentration (Figure 5.35), a lot of fluctuations can be observed in the first 30 epochs. Then the values are more stable but a slight decrease of TP and TN is observed as the training progresses, which is not desirable.

Our main objective is for the TP and TN of the validation concentration (Figure 5.36) to become as close to 50% as possible. In such a case this would mean that all the NCs are correctly identified as NCs and all the ADs correctly identified as ADs. If this is not

feasible, then the TN should be as large as possible and the FN as small as possible. This would mean that at least the ADs are classified correctly as ADs and not misclassified as NCs.

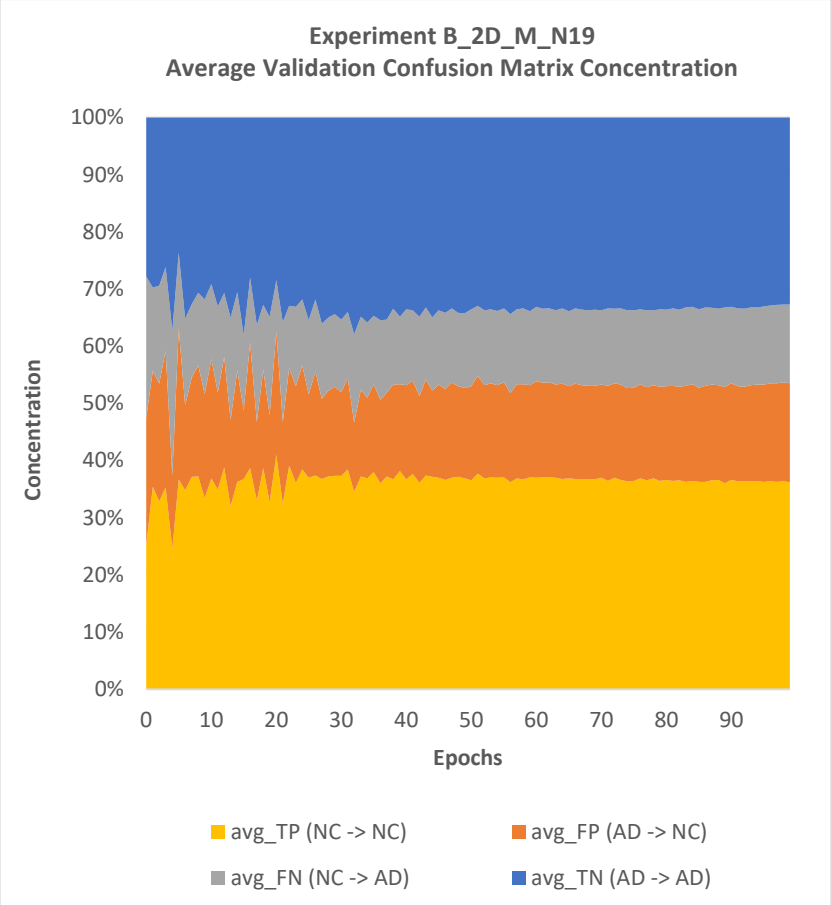


Figure 5.36. Average training confusion matrix of concentration of the experiment B_2D_M_N19.

5.4 2D Brain Slices [AD, NC]: Multiple Scans per Patient (174 × 174)

5.4.1 CNN 4 Layers with Adam

5.4.1.1 Experiments B_2D_M_A {1 – 3}: C = 0.01, lr = {0.1, 0.01, 0.001}, Dropout

The experiments B_2D_M_A {1 – 3} are meant to examine how the learning rate affects the performance of the Adam optimizer with CNNs for the B_2D_M dataset. The different learning rates tested for the experiments B_2D_M_A {1 – 3} respectively are 0.1, 0.01, and 0.001. All experiments were executed in the 2D_CNN_4L_1 network configuration (Section 4.3.2.3.1) for 500 epochs and weight decay, or so-called C (Section 2.3.13.8) equal to 0.01. The standard Dropout (Section 2.3.13.9), with

probabilities d1 = 0.3, d2 = 0.5, and d3 = 0.5 have also been applied to all experiments. No Spatial Dropout (Section 2.3.13.10), L1, or L2 Regularizations (Section 2.3.13.7) have been applied. The source code for the specific network that was used for these experiments, can be found in Appendix B.5.2, in the Code Snippet B.6.

Detailed results for each experiment can be found in Appendix H. More specifically, for the experiments B_2D_M_A {1 – 3}, their training/validation accuracies and losses of each fold are in Appendixes H.1.1 – H.1.3. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

In Table 5.8, we can see that the experiments B_2D_M_A {2, 3} with the learning rates (lr) 0.01 and 0.001 both had an average validation accuracy for the 10-fold CV equal to 79%. In terms of the Specificity metric, i.e., how many ADs were correctly identified as ADs in the CNN experiments (Section 5.3 – 5.11), for the experiment B_2D_M_A2 (lr = 0.01) is 80% while for the experiment B_2D_M_A3 (lr = 0.001) is 77% (Table 5.8). Therefore, the model of the experiment B_2D_M_A2 would be slightly more preferable than the model of B_2D_M_A3.

B_2D_M with Adam – Different Learning Rates															
							Training			Validation					
Experiment Name	Optimizer	C	Max Epochs	Learning Rate	Dropout	Feature Maps	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity
B_2D_M_A1	Adam	0.01	500	0.1	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	0.53	0.59	0.47	0.56	0.66	0.65	0.61	0.50
B_2D_M_A2	Adam	0.01	500	0.01	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	0.87	0.87	0.86	0.79	0.80	0.79	0.78	0.80
B_2D_M_A3	Adam	0.01	500	0.001	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	0.91	0.90	0.92	0.79	0.78	0.81	0.82	0.77

Table 5.8. 2D Brain Slices – Multiple Scans per Patient – Single Slice per Scan [AD, NC] dataset (B_2D_M). Optimizer: Adam. Testing different learning rates.

5.4.1.1.1 Experiment B_2D_M_A1

The B_2D_M_A1 experiment with the largest learning rate = 0.1 between the experiments B_2D_M_A {1 – 3}, does not perform very well. The learning rate for the CNN topology used and the specific dataset (B_2D_M) is too large, therefore the model is underfitting.

The training and validation accuracies for all 500 epochs are close to 50% (Table 5.8) which means that the network does not learn anything, since the AD/NC problem has

only two potential outputs (Figure 5.37). The training and validation losses as we can observe in Figure 5.38 do not decrease. The large learning rate prevents the error functions from minimizing themselves. So, smaller learning rates will be examined in the experiments B_2D_M_A {2 & 3} in Sections 5.4.1.1.2 & 5.4.1.1.3 respectively.

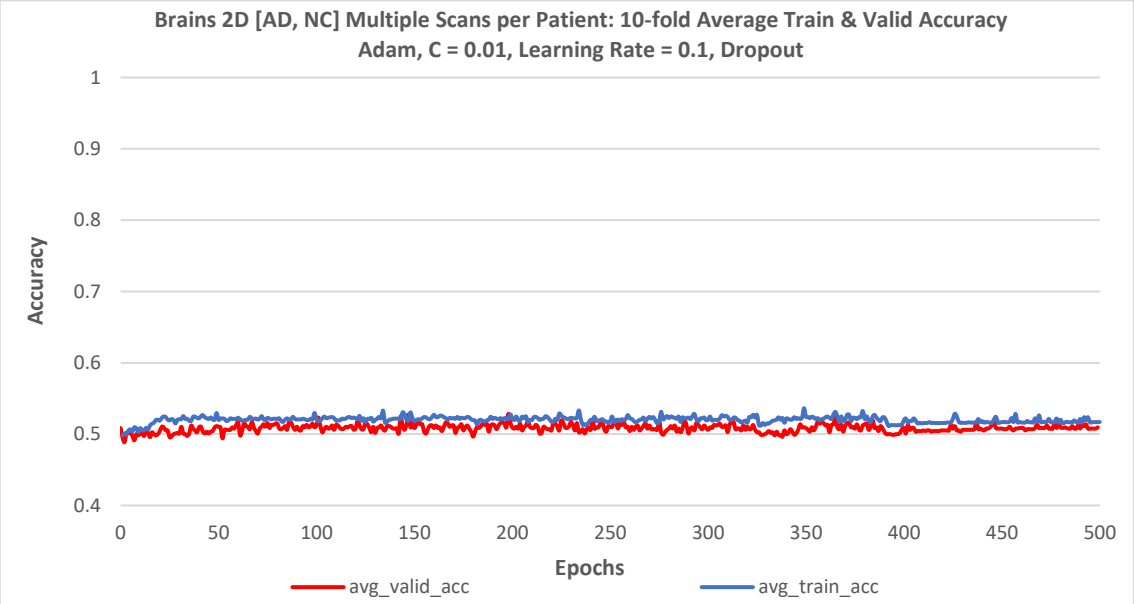


Figure 5.37. Average training and validation accuracies of the 10-folds, of the experiment B_2D_M_A1.

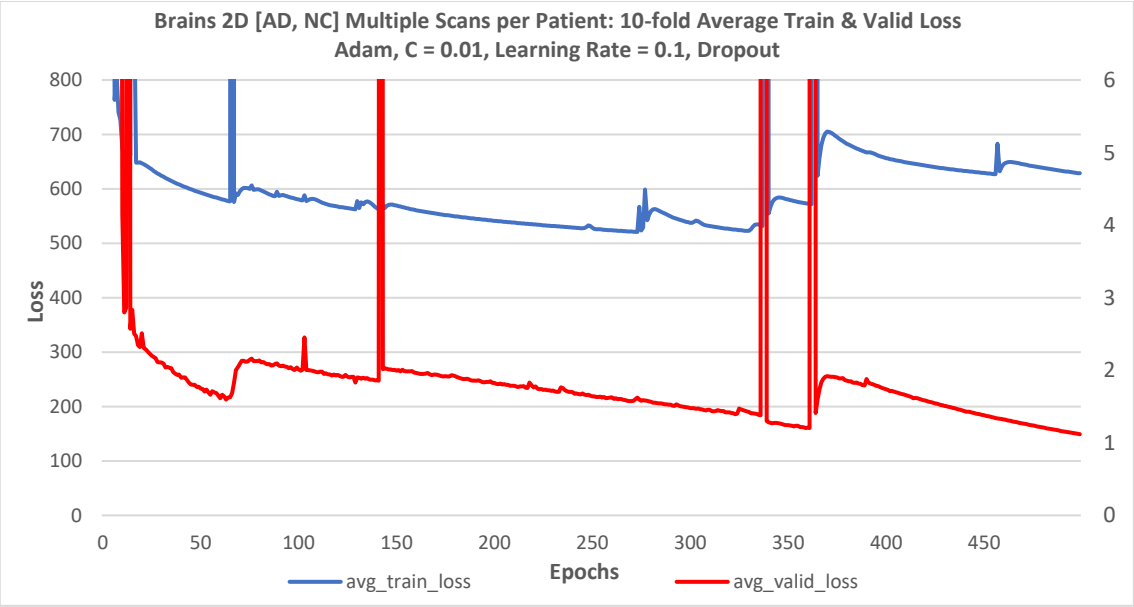


Figure 5.38. Average training and validation losses of the 10-folds, of the experiment B_2D_M_A1.

5.4.1.1.2 Experiment B_2D_M_A2

Since the large learning rate of the experiment B_2D_M_A1 causes the model to underfit (Section 5.4.1.1.1), the experiment B_2D_M_A2 was held with the learning rate being

equal to 0.01. Indeed, that helped a lot in solving the issue of underfitting, and the validation accuracy, sensitivity, and specificity reached an acceptable level, with the accuracy being 79%, sensitivity 78%, and specificity 80% (Table 5.8). Those performance metrics are very close to the best model as well, the B_2D_M_N19 (Section 5.3.1) for the AD/NC problem in CNNs with the NewtonCG optimizer. B_2D_M_N19's validation accuracy is 81%, the sensitivity 80%, and the specificity 81% (Table 5.7).

The main disadvantage of this experiment is that the Adam optimizer is much slower than the NewtonCG. The NewtonCG achieves similar performance in just 17 epochs on average (average epoch of the best validation accuracy per fold), while for the Adam after 246 epochs (Table 5.7).

In Figure 5.39, we can see the average training and validation accuracies of the 10-folds, of the B_2D_M_A2 experiment. They both grow fast at first, but soon they get stuck; the training accuracy near 85%, and the validation accuracy near 70%. If we observe the training accuracies plot for each one of the folds individually (Figure H.12), we can see that the performance between folds varies a lot; some folds perform well at the beginning, and after 100 epochs approximately they crash back to 50%. This shows us that the learning rate is still large, and in some cases the model is underfitting.

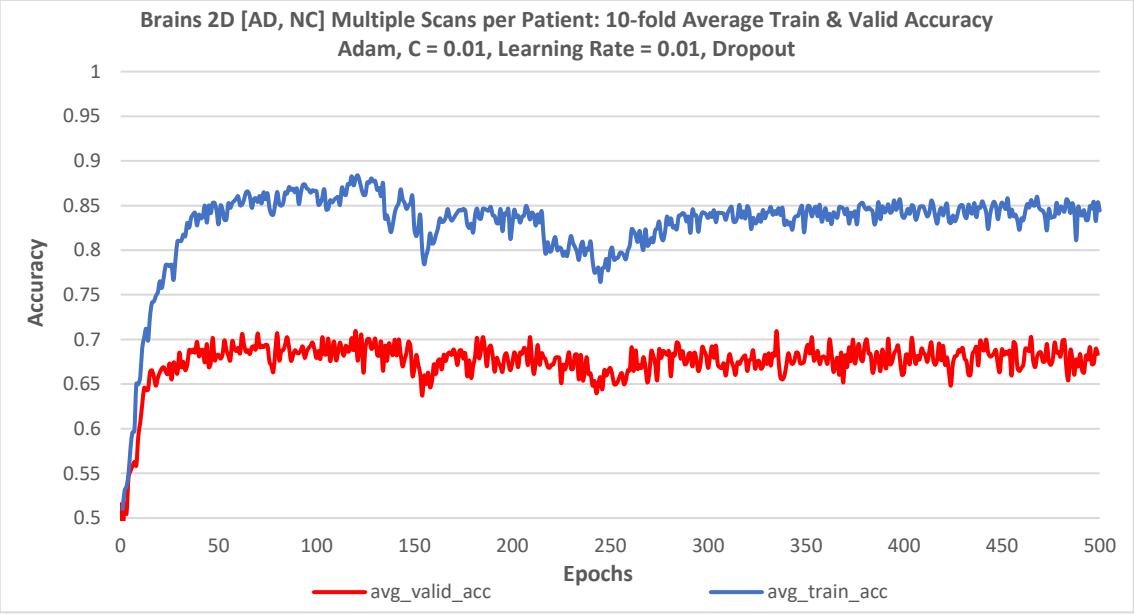


Figure 5.39. Average training and validation accuracies of the 10-folds, of the experiment B_2D_M_A2.

In Figure 5.40, we can see the average training and validation losses of the 10-folds, of the experiments B_2D_M_A2. It seems that around epoch 30, the Adam optimizer's

validation loss gets stuck in a value near 0.4, while the training loss continues to decrease. This indicates that the model is underfitting, since during the learning process most probably gets stuck in a local minimum. To fix this issue, we should decrease the learning rate even more; we are going to examine this in experiment B_2D_M_A3 (Section 5.4.1.1.3).

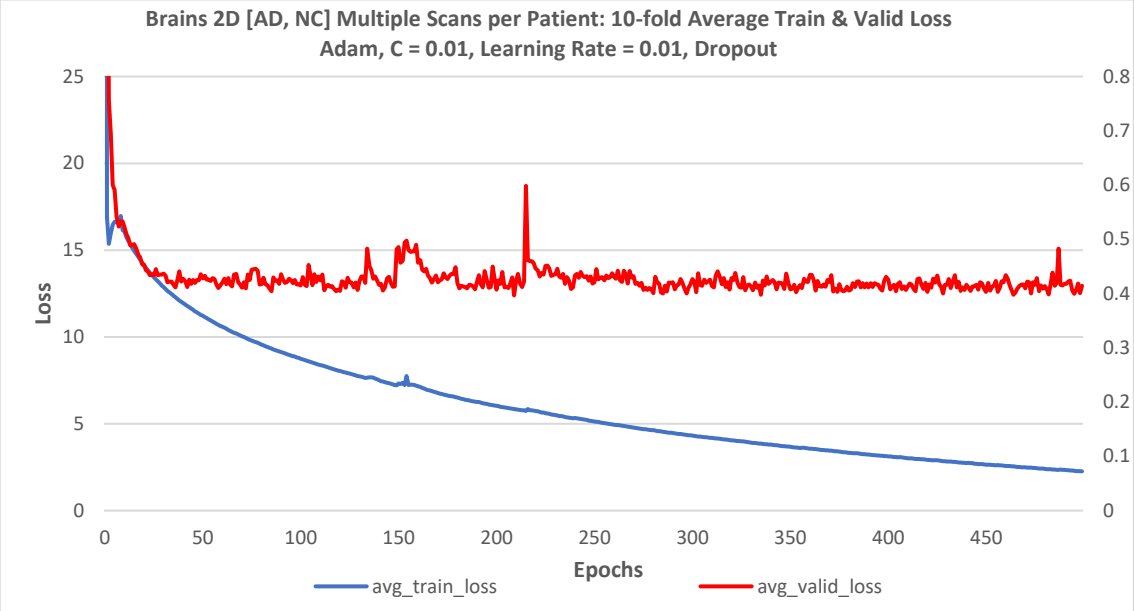


Figure 5.40. Average training and validation losses of the 10-folds, of the experiment B_2D_M_A2.

5.4.1.1.3 Experiment B_2D_M_A3

The experiment B_2D_M_A3 was meant to examine whether an even smaller learning rate than the B_2D_M_A2 (Section 5.4.1.1.2) experiment could help to avoid local minimums and encounter the underfitting that this experiment was phasing. Therefore, a learning rate = 0.001 was used.

In Figure 5.41, with the average training and validation accuracies of the 10-folds, of the B_2D_M_A3 experiment, we observe that the training accuracy grows fast, towards 100%, while the validation accuracy remains stuck at approximately 70%. By observing Figure 5.42, we can assume that the underfitting issues, of the experiments B_2D_M_A {1 & 2}, have been resolved, but now we are facing some overfitting issues. This happens because the average training loss decreases very rapidly, while the average validation loss starts to increase slightly after the 30th epoch approximately. This problem may occur due to the small dataset B_2D_M (Section 3.2.3.4). To validate this, we can see Figure H.24, with the validation accuracies of each fold; where the accuracies between folds differ a

lot, since some of them fluctuate at approximately 80% (5th fold), while others at 55% (4th fold).

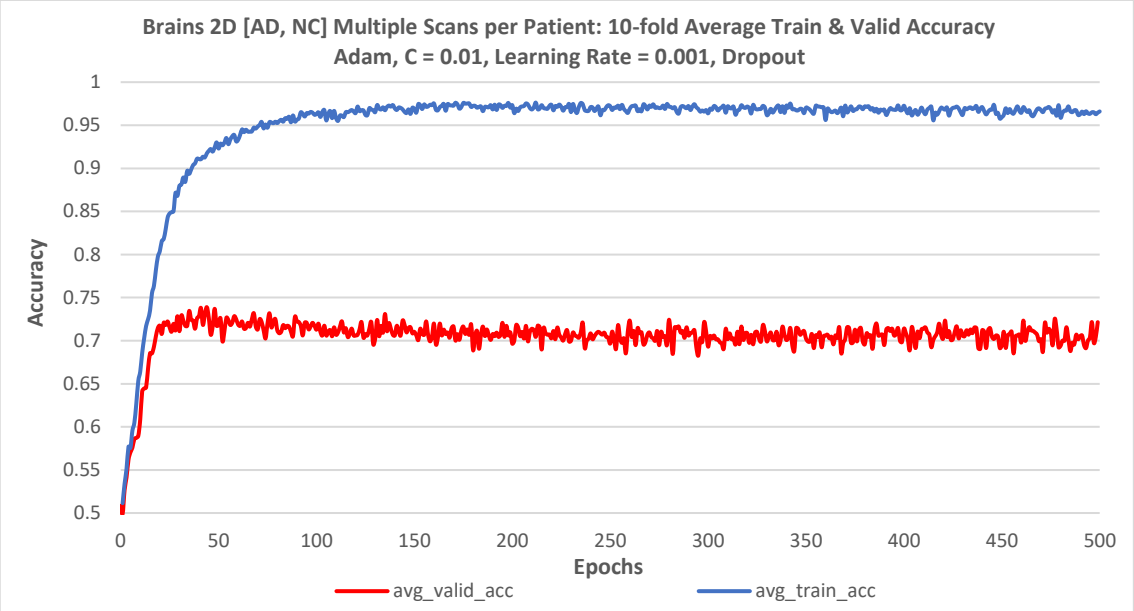


Figure 5.41. Average training and validation accuracies of the 10-folds, of the experiment B_2D_M_A3.

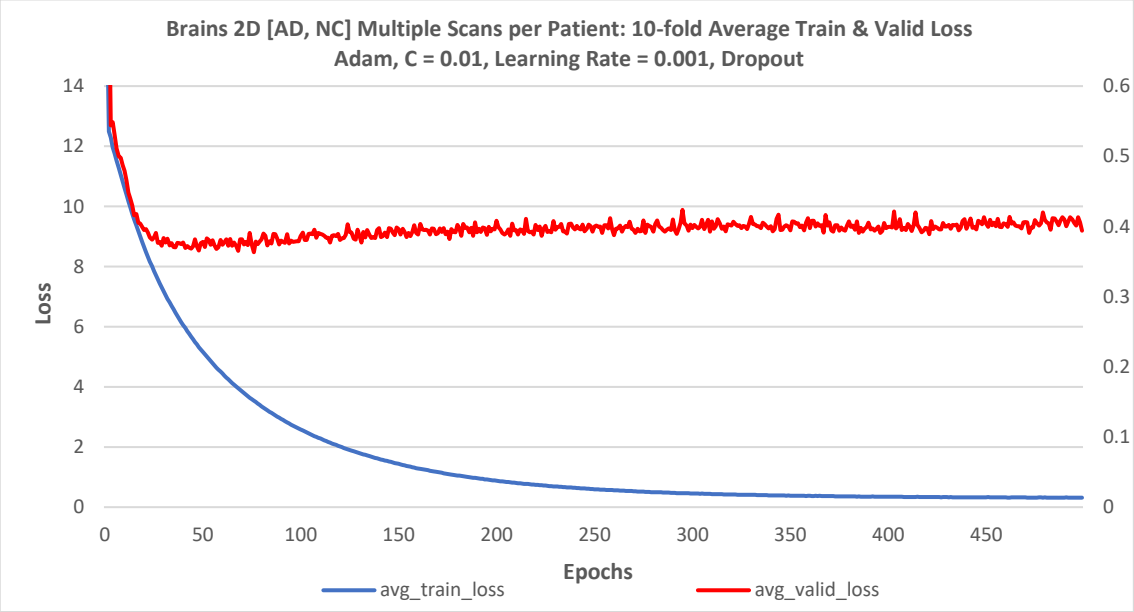


Figure 5.42. Average training and validation losses of the 10-folds, of the experiment B_2D_M_A3.

It is worth mentioning that usually in most of the CNN experiments with the B_2D_M datasets, the 4th fold performs poorly, while the 5th fold performs very well. This could indicate that the 4th fold contains many outliers, while the 5th fold contains many great samples. Due to the small size of each fold (116 samples), this affects a lot the performance metrics of the experiments.

5.4.2 CNN 4 Layers with NewtonCG

5.4.2.1 Experiments B_2D_M_N {1 – 4}: C = {0.01, 0.1, 1, 10}, GNsize = 5

Based on Wang et al. (2020), the regularization parameter C, or so-called weight decay (Section 2.3.13.8), alongside the Gauss-Newton matrix (GN) size (Section 2.4.9.2), seems to affect a lot the performance of the NewtonCG optimizer on their CNN implementations. Initially, the effects of C are going to be examined. Therefore, the four experiments B_2D_M_N {1 – 4} have been performed, with different values of C equal 0.01, 0.1, 1, and 10 respectively. The rest of the hyperparameters remain the same in all four experiments. The 'GNsize' = 5, the network configuration that was used is 2D_CNN_4L_1 (Section 4.3.2.3.1), without Dropout or any other regularization techniques. The source code for the specific network that was used for these experiments, can be found in Appendix B.5.1, in the Code Snippet B.5.

The weight decay (C) helps to control the underfitting and overfitting of the network. A large value of C causes overfitting while a small value could cause underfitting. The regularization term C is not the same as the L2 regularization (Section 2.3.11.7), at least in the case of the Adam and NewtonCG optimizers.

Detailed results for each experiment can be found in Appendix H. More specifically, for the experiments B_2D_M_N {1 – 4}, their training/validation accuracies and losses of each fold can be seen in Appendixes H.2.1 – H.2.4. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

In Figure 5.43, of the average training and validation accuracies of the 10-folds, of the experiments B_2D_M_N {1 – 4}, no much variance can be observed between different values of C. Keep in mind that those are average accuracies of a 10-fold Cross-validation. Later on, we are going to examine the behavior of each fold individually. In general, from Figure 5.43, we can say that the experiment B_2D_M_N3 with C = 1 (green), seems to be a slightly better choice than the rest of them since overall has a better average validation accuracy on each epoch. In Figure 5.44, with the average training and validation losses of the 10-folds, of the experiments B_2D_M_N {1 – 4}, we do not observe any signs of overfitting, hence i.e., the validation loss does not increase as the training loss decreases.

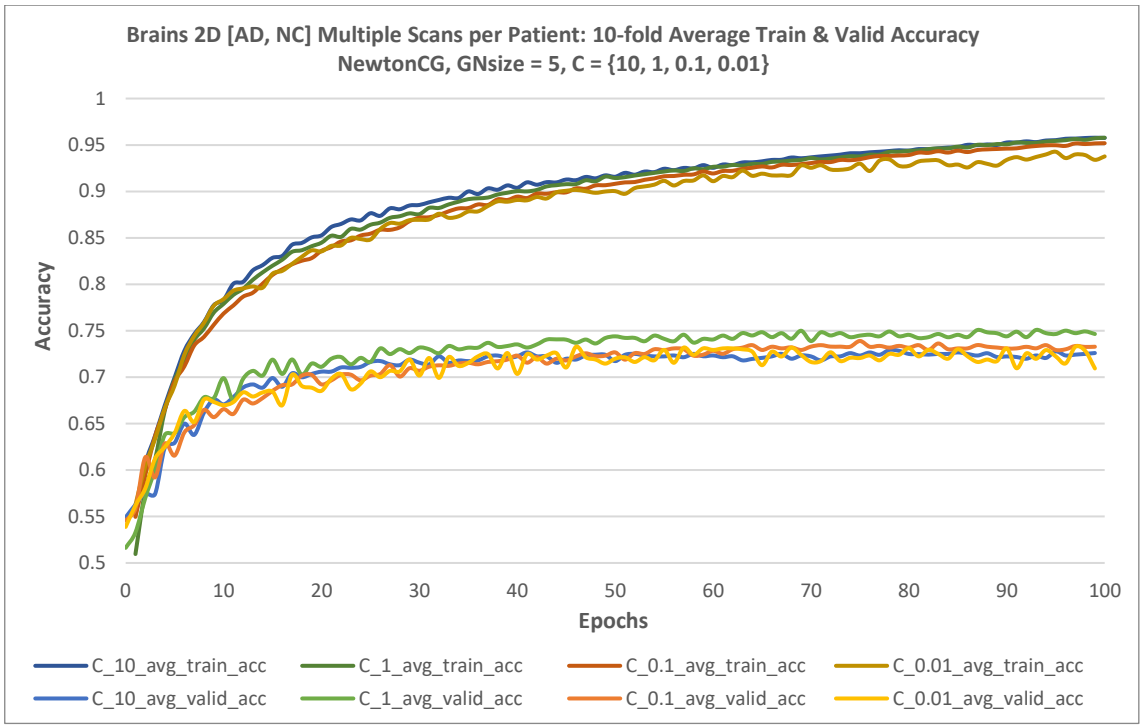


Figure 5.43. B_2D_M_N {1 – 4} experiments' average training and validation accuracies of the 10-folds. Comparing different values of C {0.01, 0.1, 1, 10}.

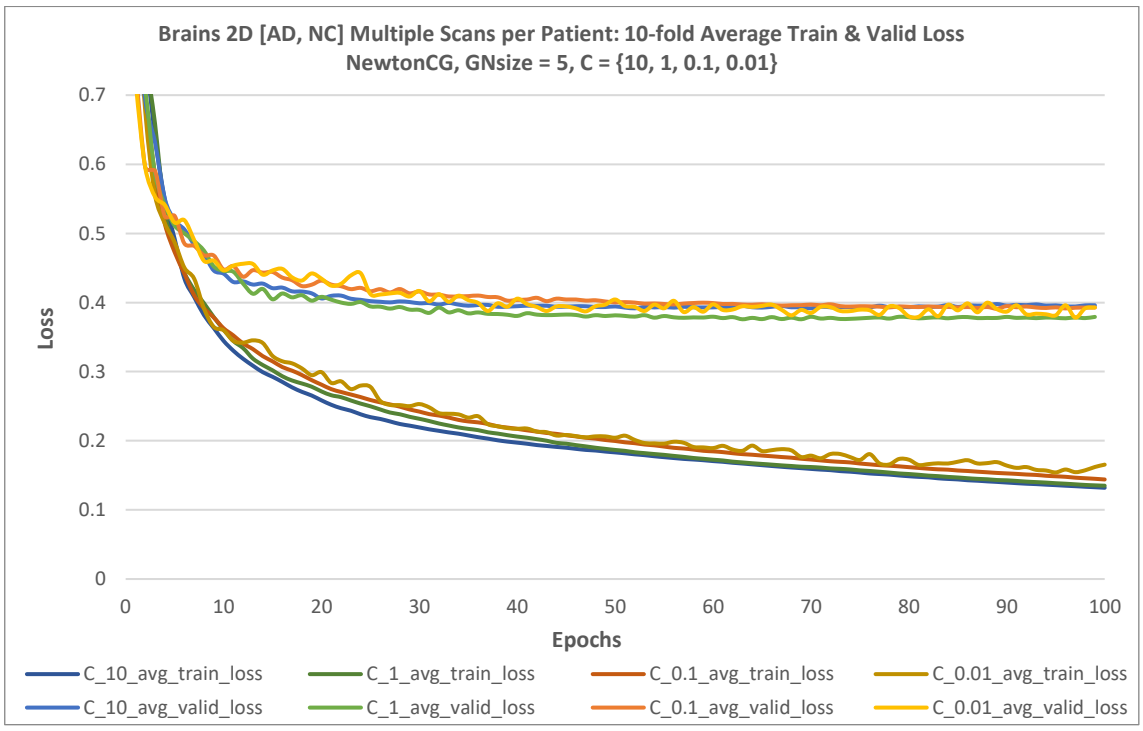


Figure 5.44. B_2D_M_N {1 – 4} experiments' average training and validation losses of the 10-folds. Comparing different values of C {0.01, 0.1, 1, 10}.

Based on the performance metrics of Table 5.9, the experiment B_2D_M_N1 with C = 0.01, is slightly more promising, since the average validation accuracy of the best

validation accuracies per epoch, is 78%, the highest among these four experiments. On the other hand, the experiment B_2D_M_N3 with C = 1, has the highest specificity, i.e. how many ADs are correctly identified as ADs, equal to 77%, which is the most important performance metric for us in the case of the AD/NC problem.

Experiments B_2D_M_N {1 – 4} Performance Metrics																	
		Training				Validation						Testing					
Experiment Name	C	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
B_2D_M_N1	0.01	0.88	0.04	0.89	0.87	0.78	0.08	0.77	0.79	0.80	0.75	0.61	0.03	0.71	0.52	48	0.90
B_2D_M_N2	0.1	0.87	0.04	0.87	0.87	0.76	0.08	0.75	0.77	0.77	0.74	0.62	0.06	0.72	0.51	38	0.90
B_2D_M_N3	1	0.91	0.04	0.90	0.91	0.77	0.07	0.78	0.78	0.78	0.77	0.62	0.03	0.69	0.55	51	0.86
B_2D_M_N4	10	0.89	0.05	0.91	0.87	0.75	0.10	0.75	0.78	0.79	0.72	0.63	0.04	0.72	0.53	48	0.90

Table 5.9. Experiments B_2D_M_N {1 - 4} performance metrics.

However, even though Table 5.9 showed that the experiment B_2D_M_N1 with C = 0.01 had the best average validation accuracy among the four experiments, if we examine the individual accuracies per fold, new conclusions may come up. The shades of gray in Figure 5.34, show the validation accuracies of each fold and with red color the average validation accuracy of all folds (the one in Figure 5.43), for the experiment B_2D_M_N1.

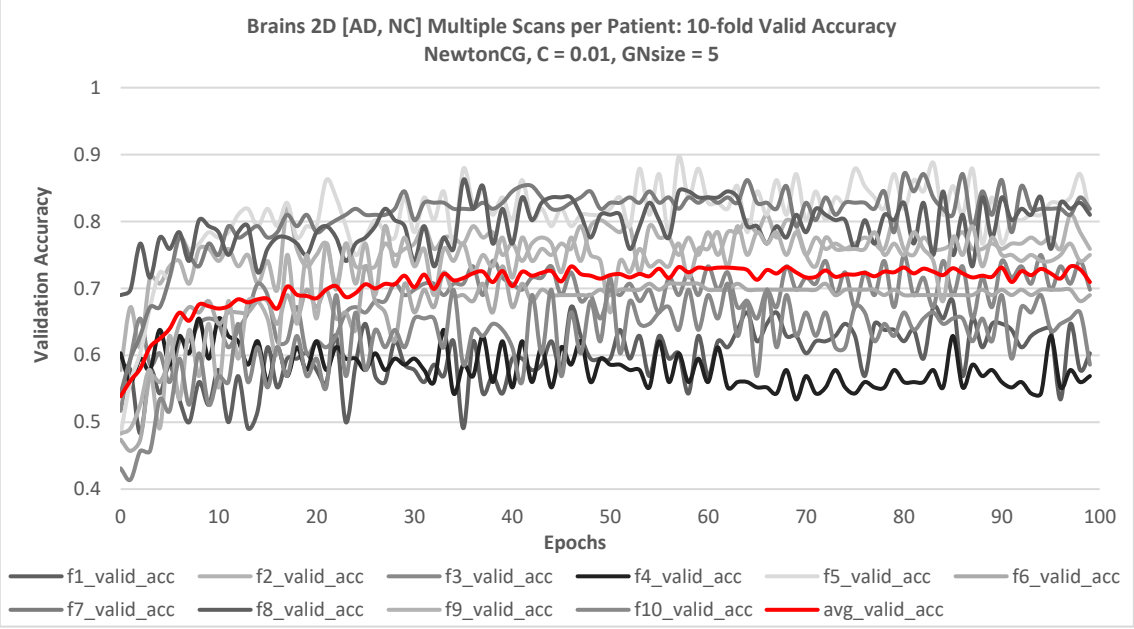


Figure 5.45. Experiment B_2D_M_N1 with C = 0.01. Validation accuracy per fold.

The experiment B_2D_M_N2 with C = 0.1, based on the validation accuracies per fold in Figure 5.46, seems to perform better than the experiment B_2D_M_N1 with C = 0.01 in Figure 5.45. The reason is that the validation accuracies range between 0.6 and 0.9 for the experiment with C = 0.1, while for the experiment with C = 0.01 range between 0.55 and 0.85, in their last epoch. Anyhow, i.e., the large variation between the folds is caused due to the small size of the B_2D_M dataset.

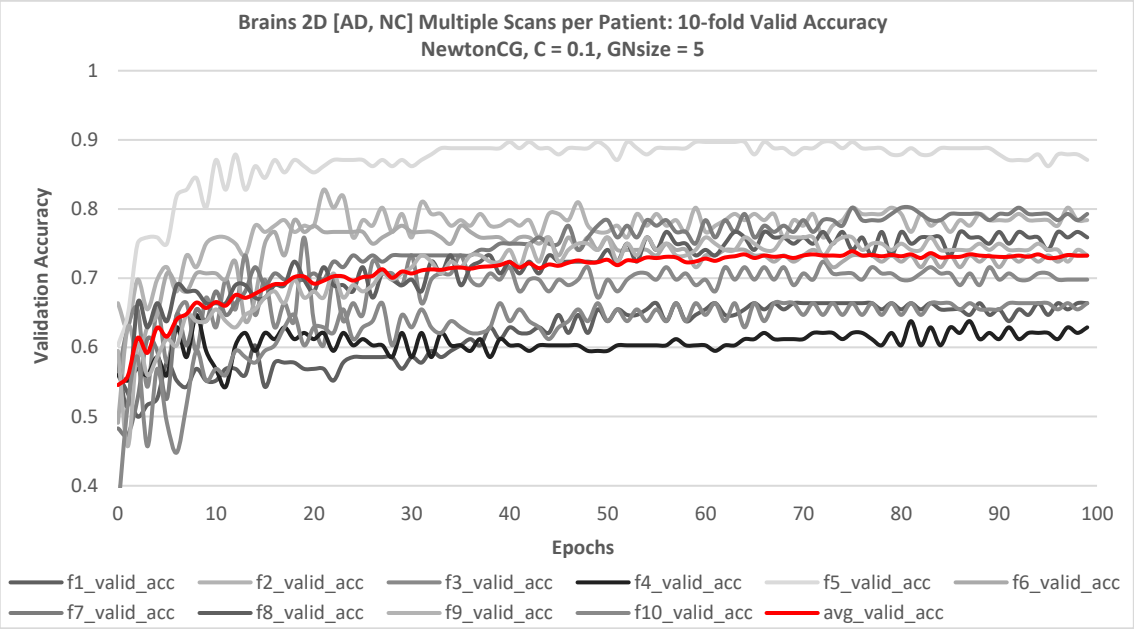


Figure 5.46. Experiment B_2D_M_N2 with C = 0.1. Validation accuracy per fold.

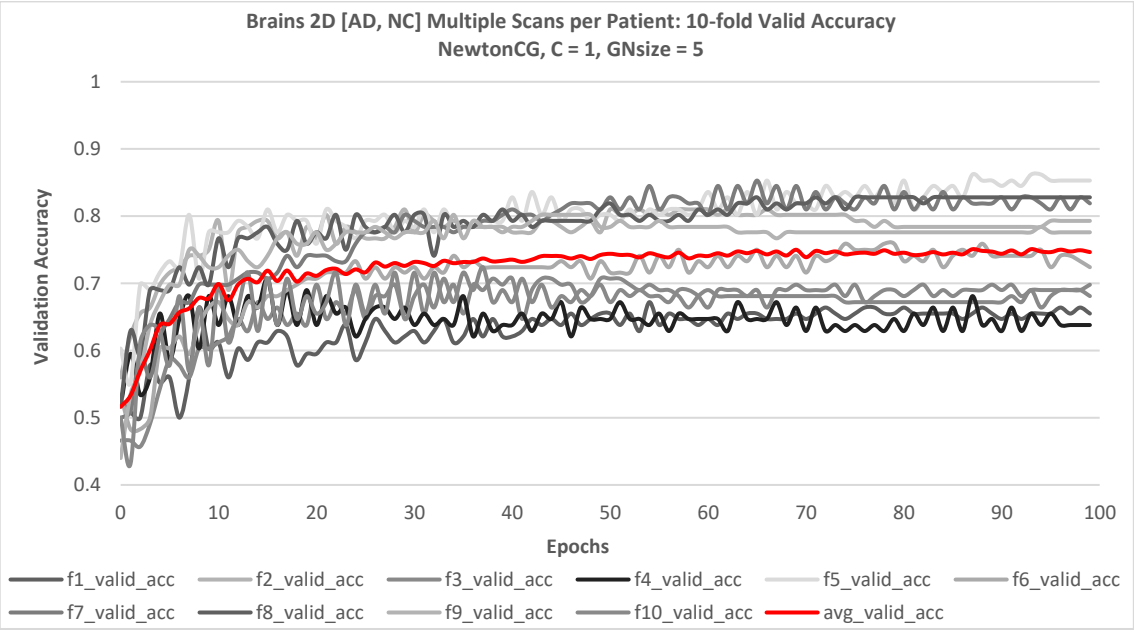


Figure 5.47. Experiment B_2D_M_N3 with C = 1. Validation accuracy per fold.

Figure 5.48 shows the validation accuracies per fold of the B_2D_M_N4 experiment. The variation between folds is similar to the B_2D_M_N1 experiment. The experiment B_2D_M_N3 with C = 1, based on the validation accuracies per fold in Figure 5.45, seems to be the best one among all four experiments. The reason is that the validation accuracies of the 10-folds range between 0.65 and 0.86 for the experiment with C = 1 in the last epoch. while for the experiment with C = 0.01 ranges between 0.55 and 0.85, in their last epoch. This is a 0.2 range between the maximum and the minimum validation accuracy at the 100th epoch, while the B_2D_M_N1 experiment was approximately 0.3.

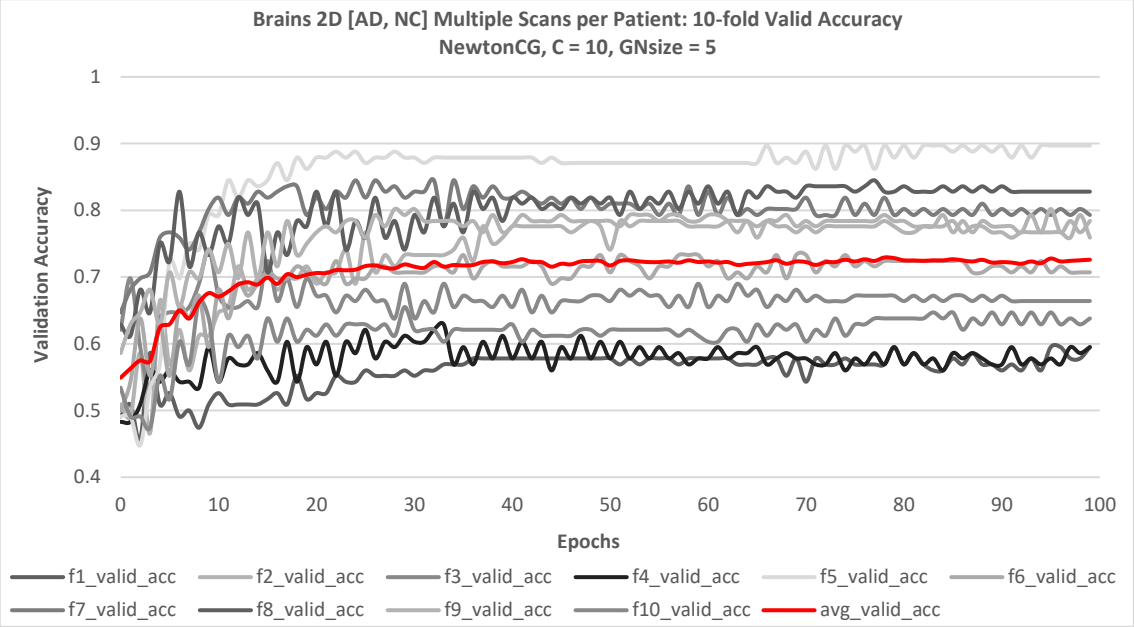


Figure 5.48. Experiment B_2D_M_N4 with C = 10. Validation accuracy per fold.

In general, the C = 1 seems to be the best choice, but if we compare the average accuracy of the 4th fold on each one of the four experiments (Figure 5.45 – 5.48), it always has the worst performance between the experiments; i.e., the 4th fold contains outliers, therefore the cause behind this large variation is the small dataset (B_2D_M).

5.4.2.2 Experiments B_2D_M_N {3, 5 – 9}: C = 1

Based on the observations in Section 5.4.2.1, the choice between C = 0.01, and C = 1, is not clear, since the first one has the highest average validation accuracy, while the second one has less variation between folds. Consequently, some experiments will be held with C = 1 (Section 5.4.2.2 & 5.4.2.3), while others with C = 0.01 (Section 5.4.2.5 – 5.4.2.12). In this section, we are going to compare all the experiments B_2D_M_N {3, 5 – 9} with

C = 1, how the different hyperparameters and network configurations affect their performance (Table 5.10).

Detailed results for each experiment can be found in Appendix H. More specifically, for the experiments B_2D_M_N {3, 5 – 9}, their training/validation accuracies and losses of each fold can be seen in Appendixes H.2.3, H.2.5 – H.2.9. There, are also available the confusion matrices for the training, validation, and test sets of these experiments.

In Table 5.10 the hyperparameters (L2 Regularization, Dropout, GNsize), network architecture, and the different performance metrics of the experiments B_2D_M_N {3, 5 – 9}, are available. All experiments were executed for 100 epochs, with the MSE loss function (Section 2.3.4), using the same dataset (B_2D_M). The two network topologies that are going to be used are the 2D_CNN_4L_1 (Section 4.3.2.3.1) and 2D_CNN_5L_1 (Section 4.3.2.4.1).

The source code for the 2D_CNN_4L_1 network, that was used for the experiments B_2D_M_N {3, 6, 8}, is available in Appendix B.5.1, in the Code Snippet B.5. A sample section of the source code for the 2D_CNN_4L_1 network with Dropout, which was used for the experiments B_2D_M_N {7, 9}, is available in Appendix B.5.2, in the Code Snippet B.6. In Appendix B.5.3, in the Code Snippet B.7, is available the source code of the network 2D_CNN_5L_1, which was used for the experiment B_2D_M_N5.

Experiments B_2D_M_N {3, 5 – 9} – Different Configurations with C = 1

							Training			Validation			Testing				
Experiment Name Prefix in Plot	Experiment Name	C	GNsize	Dropout	L2 Regularization	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
C_1	B_2D_M_N3	1	5	-	-	2D_CNN_4L_1	0.91	0.90	0.91	0.77	0.78	0.77	0.62	0.69	0.55	51	0.86
C_1_FFNN2	B_2D_M_N5	1	5	-	-	2D_CNN_5L_1	0.89	0.90	0.88	0.75	0.77	0.74	0.61	0.69	0.52	47	0.87
C_1_Reg	B_2D_M_N6	1	5	-	0.01	2D_CNN_4L_1	0.88	0.88	0.88	0.76	0.75	0.78	0.60	0.69	0.51	38	0.90
C_1_Drop	B_2D_M_N7	1	5	0.3	-	2D_CNN_4L_1	0.90	0.91	0.90	0.76	0.77	0.74	0.62	0.71	0.53	56	0.86
C_1_GN50	B_2D_M_N8	1	50	-	-	2D_CNN_4L_1	0.90	0.90	0.91	0.78	0.79	0.77	0.60	0.69	0.52	9	0.91
C_1_GN50_Drop	B_2D_M_N9	1	50	d1 = 0.3 d2, d3 = 0.5	-	2D_CNN_4L_1	0.93	0.94	0.92	0.76	0.78	0.74	0.62	0.71	0.53	14	0.89

Table 5.10. Experiments B_2D_M_N {3, 5 – 9} – Different Configurations with C = 1

Based on the results of Table 5.10, it seems that the addition of L2 Regularization (B_2D_M_N6), or Dropout (B_2D_M_N7), or the 2-layer FFNN (B_2D_M_N5) instead of a single layer, does not improve the performance of the B_2D_M_N3 experiment. In general, it seems that the experiments B_2D_M_N {5 – 7} have worse performance than the baseline experiment B_2D_M_N3, without extra modifications.

A difference in terms of average validation accuracies makes the GNsize (Table 5.10). The experiments B_2D_M_N {3, 5 – 7} all have GNsize = 5, and they behave very similarly. On the other hand, the experiments B_2D_M_N {8, 9} have GNsize = 50. As we saw in Wang et al. (2020) the size of the subsampled Gauss-Newton matrix for approximating the Hessian Matrix (GNsize), is one of the key hyperparameters that affects the performance of the model when using the NewtonCG optimizer. In their experiments, the GNsize was tested to be the 1%, 5%, and 10% of the training set, with the GNsize = 10% scoring best.

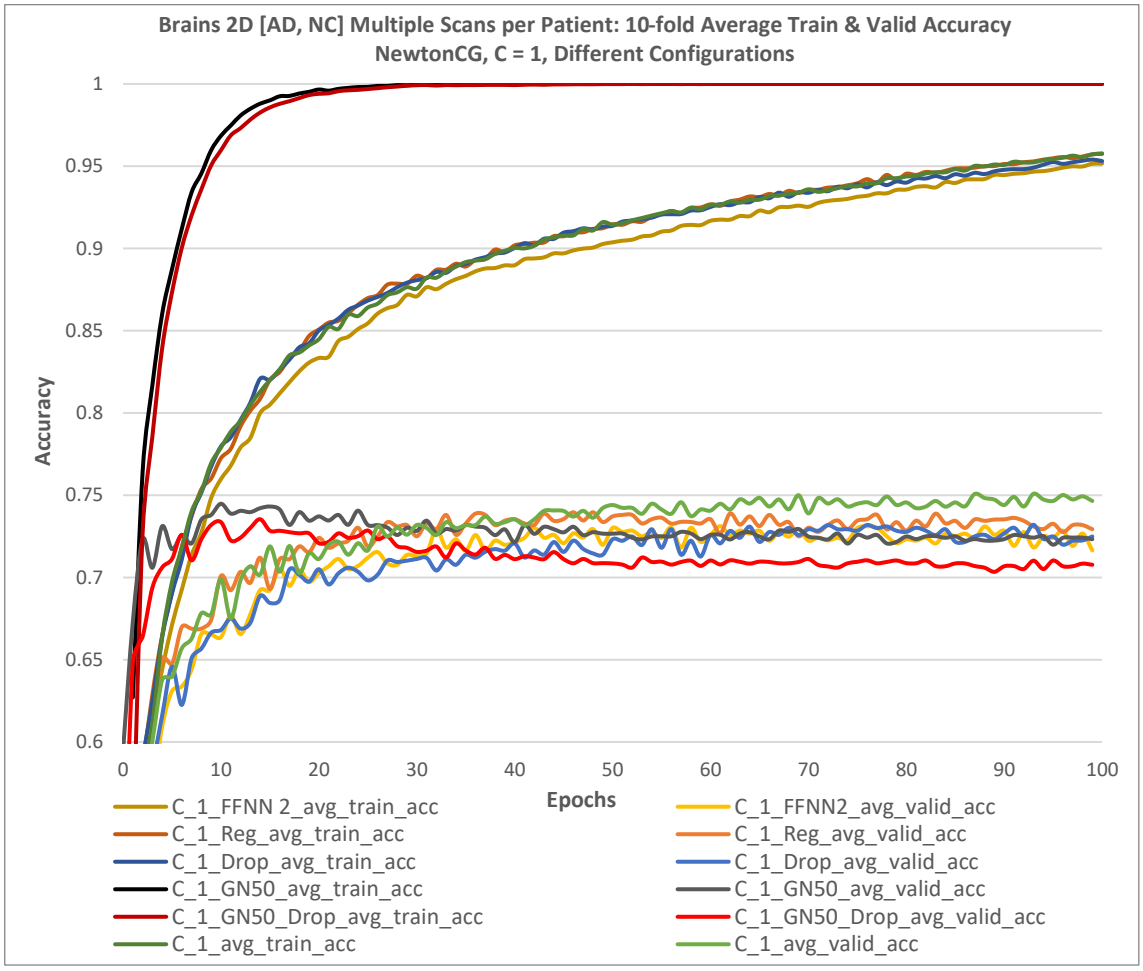


Figure 5.49. Average training and validation accuracies of the experiments B_2D_M_N {3, 5 – 7} with C = 1. Examining different hyperparameters and network configurations.

For the experiments B_2D_M_N {3, 5 – 7}, the GNsizes 5, and 50 have been tested, which represent the ~0.5%, and ~5% respectively of our training set of 1044 samples (B_2D_M). While in the experiments with GNsize = 5, the average validation accuracies increase smoothly throughout the whole training process, for GNsize = 50 we observe a steep increase in the first 10 epochs, and then a slight decrease, where they remain relatively stable for the rest of the epochs (Figure 5.49).

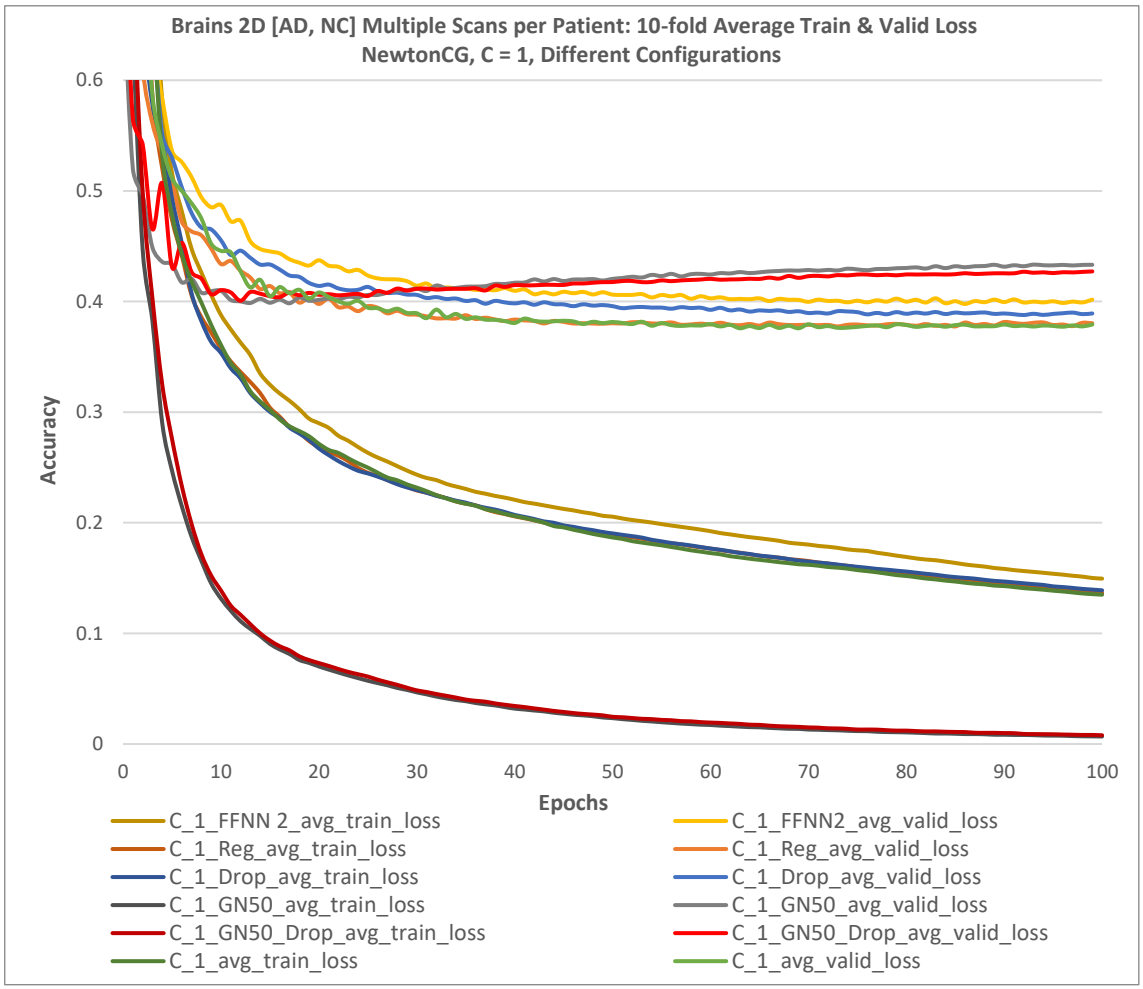


Figure 5.50. Average training and validation losses of the experiments B_2D_M_N {3, 5 – 7} with C = 1. Examining different hyperparameters and network configurations.

This behavior can be observed in Table 5.10 as well as in the "Avg. epoch of best Valid Acc. per fold" metric. For the experiments with GNsize = 50, the average epoch of the best validation accuracy per fold was 9 for the B_2D_M_N8 experiment and 14 for the B_2D_M_N9 experiment. Contrariwise, the average epoch of the best validation accuracy per fold for the rest of the experiments with GNsize = 5, was between 38 and 56. Therefore, based on our experiments and Wang et al.'s (2020), the larger the GNsize is, the fewer epochs the model needs to converge.

The average training accuracies of the GNsize = 50 experiments, manage to hit 100% after 20 epochs while for GNsize = 5, after 100 epochs is still around 95%. Keep in mind that in Table 5.10 the training accuracies that are being shown are the ones where the best model was saved, so when the best validation accuracy was recorded, not the maximum training accuracy. The fact that the training accuracies of the NewtonCG with GNsize = 50 converge that fast, could potentially mean that the models suffer from overfitting. This can be validated by observing Figure 5.50 with the average training and validation losses of the 10-folds, of the experiments B_2D_M_N {3, 5 – 7}.

As we can see in Figure 5.50, the average validation losses of the experiments with GNsize = 50 (light red & gray), after the 10^{th} epoch approximately, start to increase. In the meantime, their training losses decrease rapidly (dark red & black). The rest of the experiments do not seem to have any overfitting issues. Therefore, the GNsize plays a major role in overfitting the NewtonCG algorithm. Additionally, even dropout does not help to fight against overfitting when the GNsize is large (B_2D_M_N9); this indicates how aggressive the NewtonCG optimizer is. This issue can potentially be resolved when a larger dataset is used. Another option could be the development of more advanced regularization techniques to encounter overfitting that would be specialized for second-order optimizers.

5.4.2.3 Experiments B_2D_M_N {3, 28}: C = 1, Epochs = {100, 500}

The experiment B_2D_M_N28 has been performed to compare how the epochs in the NewtonCG algorithm, affect the performance. I observed that the average validation accuracy of the experiment B_2D_M_N3 (Figure 5.47), was increasing steadily over time. Therefore, its hyperparameters and network configuration were used in the B_2D_M_N28 experiment, for 500 epochs instead of 100.

As we can see in Figure 5.51, with the average training and validation accuracies of the experiments B_2D_M_N {3, 28}, the results are a bit strange, since for the first 100 epochs it seems that the validation accuracy of the experiment B_2D_M_N28 does not increases the same way as for the B_2D_M_N3 experiment. After the 100^{th} epoch it starts increasing, but the final average validation accuracy, after 500 epochs is less than the one of the B_2D_M_N3 experiment after 100 epochs.

The average validation accuracy of the best models on each fold is 77% for both experiments (Table 5.7). The main difference between them is that for the experiment B_2D_M_N3 the best accuracy of each fold was on average in the 51st epoch, while for the B_2D_M_N28, was in the 115th epoch. A better validation accuracy overall was expected from the experiment with the 500 epochs. I assume that the initialization of the weights was very bad for the B_2D_M_N28, but I cannot tell for sure since I did not repeat the experiment to identify whether this was the case.

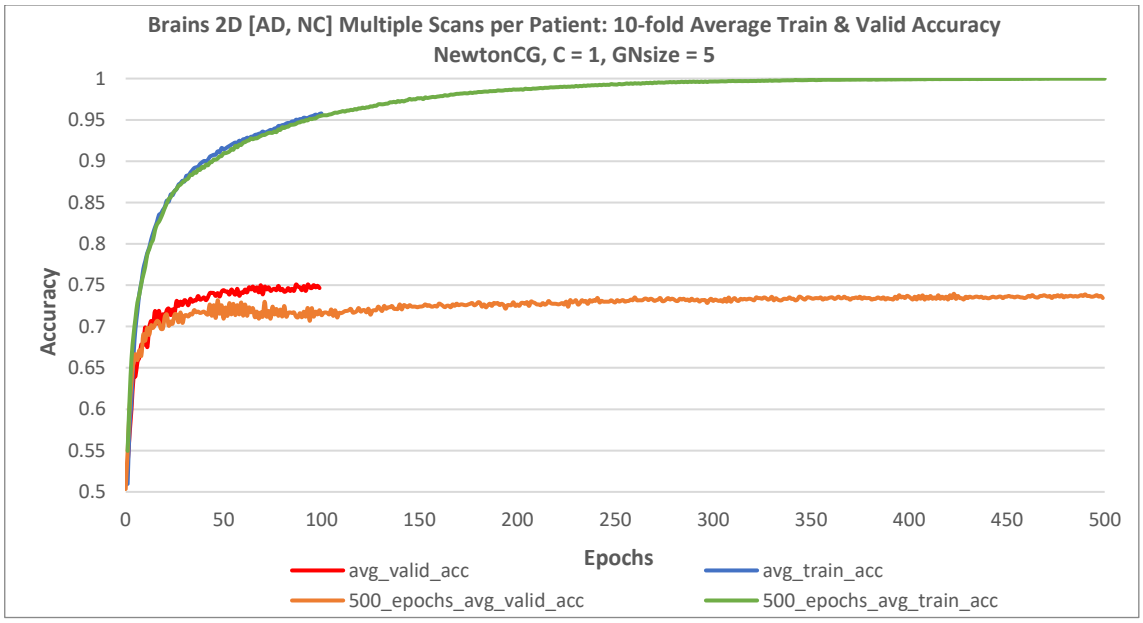


Figure 5.51. Average training and validation accuracies of the experiments B_2D_M_N {3, 28}. Compare epochs.

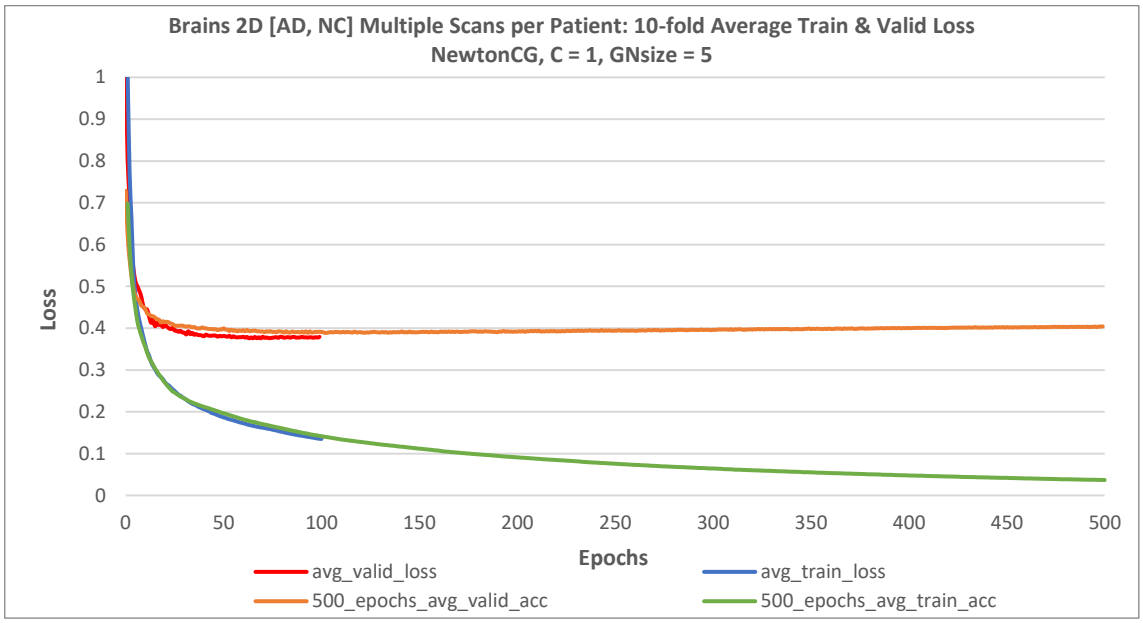


Figure 5.52. Average training and validation accuracies of the experiments B_2D_M_N {3, 28}. Compare epochs.

In Figure 5.52 we can see the average training and validation losses of the two experiments. The experiment B_2D_M_N28 seems to overfit since its validation loss increases slightly as its training loss decreases.

5.4.2.4 Experiments B_2D_M_N {1, 13, 12}: C = 0.01, GNsize = {5, 50, 200}

The experiments B_2D_M_N {1, 8, 12} compare the effect of different sizes of the subsampled Gauss-Newton matrix for approximating the Hessian Matrix (GNsize) (Section 2.4.9). More specifically, in the experiments B_2D_M_N {1, 8, 12} with GNsizes of 5, 50, and 200 respectively, are going to be compared. All three experiments have the same hyperparameters, no Dropout, no Regularization, and the same weight decay equal to C = 0.01. The network architecture that is going to be used in all three experiments is the 2D_CNN_4L_1 (Section 4.3.2.3.1), and the source code for the network of the experiments can be found in Appendix B.5.1, at Code Snippet B.5.

The GNsizes 5, 50, and 200, correspond to the ~0.5%, ~5%, and ~20% respectively of B_2D_M's training set (Section 3.2.3.4). Based on Table 5.11 with the performance metrics of the three experiments, the average validation accuracy for the 10-folds is 78% in all three configurations. The average validation specificity is a bit better in the experiment with GNsize = 200 (78%), while for the experiments with GNsize = 50 and GNsize = 5, are 77% and 75% respectively. Nevertheless, those differences are minor, since the experiments with larger specificity, have smaller sensitivity.

Experiments B_2D_M_N {1, 13, 12} – Different GNsize = {5, 50, 200}																
				Training			Validation					Testing				
Experiment Name	C	GNsize	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
B_2D_M_N1	0.01	5	2D_CNN_4L_1	0.88	0.89	0.87	0.78	0.77	0.79	0.80	0.75	0.61	0.71	0.52	48	0.90
B_2D_M_N12	0.01	200	2D_CNN_4L_1	0.90	0.89	0.90	0.78	0.78	0.78	0.77	0.78	0.64	0.75	0.52	10	0.90
B_2D_M_N13	0.01	50	2D_CNN_4L_1	0.84	0.86	0.82	0.78	0.78	0.80	0.79	0.77	0.60	0.67	0.53	13	0.89

Table 5.11. Performance metrics of the experiments B_2D_M_N {1, 13, 12}. Comparing different GNsize = {5, 50, 200}

The main difference between the three methods is the average epoch of the best validation accuracy per fold, which is smaller as the GNsize gets larger. This potentially means that the larger the GNsize, the earliest the network converges. For GNsize = 5 is 48, for

GNsize = 50 is 13, while for GNsize = 200 is 10 (Table 5.11). It is worth mentioning though that the GNsize affects the execution time since for larger GNsizes it takes more time to complete an epoch (Wang et al. 2020).

In general, the performance metrics in the Table 5.11 look very similar. Contrariwise, the plots of the average accuracies (Figure 5.53) and losses (Figure 5.54) are completely different between the experiments.

In Figure 5.53, the average training and validation accuracies of the 10-folds, of the experiments B_2D_M_N {1, 13, 12} are visible. The average validation accuracy, of the experiment B_2D_M_N1 with GNsize = 5 (purple), increases smoothly over time. On the other hand, the average validation accuracies of the experiment B_2D_M_N {12, 13} with GNsize = 200 and GNsize = 50 respectively, initially fluctuate a lot, and after the 30th epoch approximately remain stable for the rest of the epochs. This could indicate that for large GNsize fewer epochs are needed since the algorithm progresses fast at the beginning.

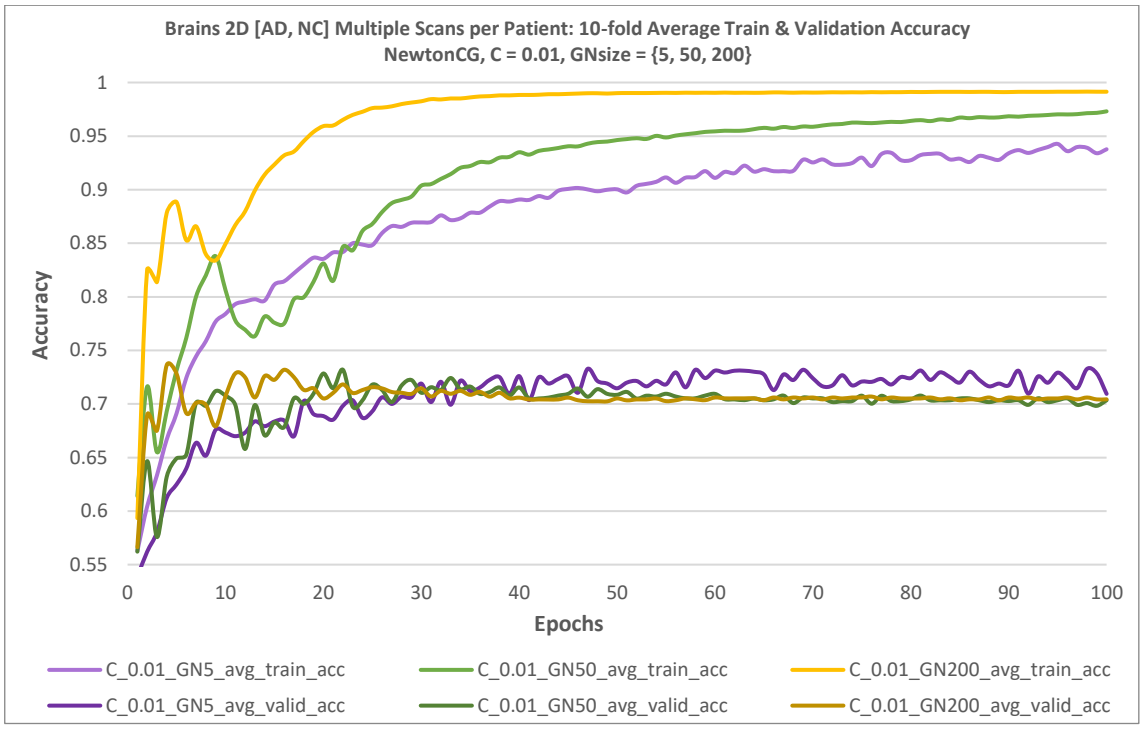


Figure 5.53. Average training and validation accuracies of the experiments B_2D_M_N {1, 13, 12}. Comparing different GNsize = {5, 50, 200}

In both experiments with GNsize = 50 (B_2D_M_N13) and GNsize = 200 (B_2D_M_N12), a spike in the average training accuracy is observed, around their 5th

and 10th epoch respectively (Figure 5.53). Then a steep decline occurs in the next 5 epochs of both experiments, before starting to increase again towards 100%. No such behavior is observed with the experiment B_2D_M_N1 of GNsize = 5.

In Figure 5.54, the average training and validation losses of the 10-folds for the three experiments are available. We can say that the experiments with GNsize = 50 and GNsize = 200 suffer from overfitting. Their training losses continue to decline while the validation losses face a slight increase over time. This is an issue for the experiments with CNN and the NewtonCG optimizer in this thesis since the large GNsizes are required to improve the network's performance (Wang et al. 2020), but at the same time, they cause overfitting as well.

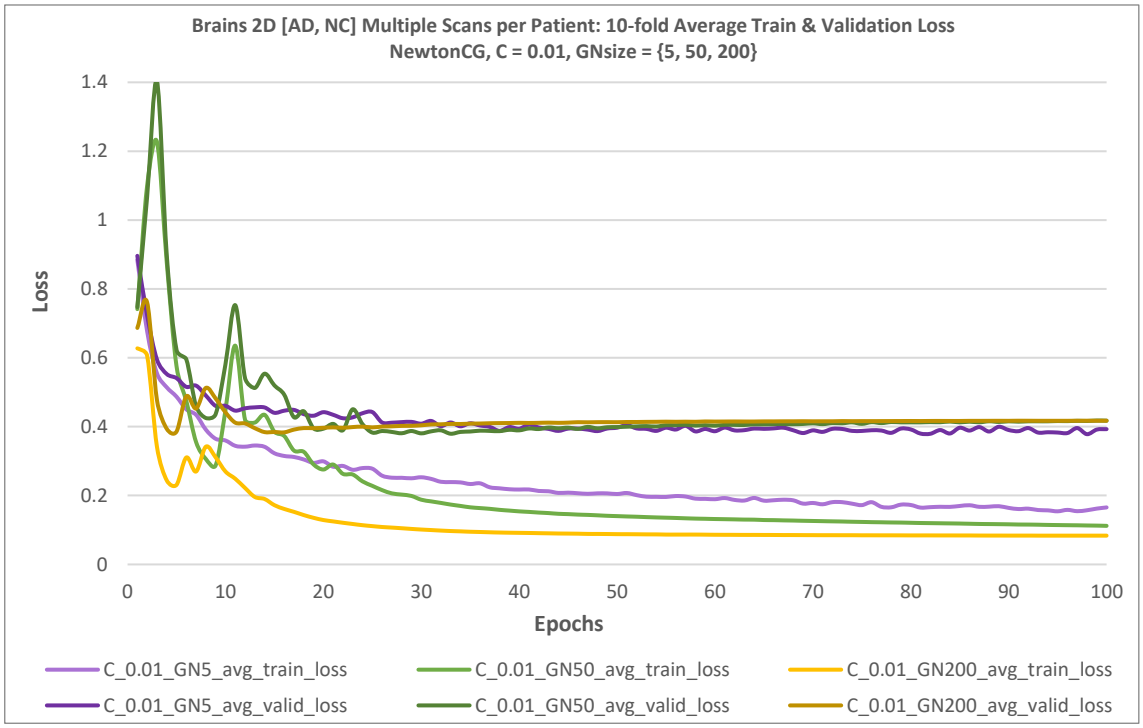


Figure 5.54. Average training and validation accuracies of the experiments B_2D_M_N {1, 13, 12}. Comparing different GNsize = {5, 50, 200}

5.4.2.5 Experiments B_2D_M_N {1, 10 – 20}: C = 0.01

The experiments B_2D_M_N {1, 10 – 20} have in common the value of the weight decay, equal to C = 0.01. They are meant to compare different network configurations and hyperparameters. More specifically, examine the effects of different network architectures, filter sizes, the loss functions, the Max-Pooling, the Dropout, and L1 & L2 Regularizations on the model's performance. These hyperparameters are described in detail in Table 5.12.

The experiments B_2D_M_N {1, 10 – 13, 16 – 18} used the 2D_CNN_4L_1 (Section 4.3.2.3.1); the experiment B_2D_M_N14 the 2D_CNN_4L_4 network configuration (Section 4.3.2.3.4); the experiment B_2D_M_N15 the 2D_CNN_7L_1 network (Section 4.3.2.5.1); and the experiments B_2D_M_N {19, 20} the 2D_CNN_4L_2 (Section 4.3.2.3.2), and 2D_CNN_4L_3 (Section 4.3.2.3.3) networks respectively.

The source code of B_2D_M_N {1, 12 – 13}'s network can be found in Appendix B.5.1, at Code Snippet B.5. For the experiments B_2D_M_N {10 – 11, 14 – 16, 19 – 20} which apply Dropout, a sample source code of the network is in Appendix B.5.2, at Code Snippet B.6. For the experiments B_2D_M_N {19, 20}, the number of the filters in the source code has to be changed from '3', to '5' and '7' respectively. A sample section of the source code for B_2D_M_N18's network, which uses Batch Normalization, is available in Appendix B.5.9, at Code Snippet B.13. Finally, in Appendix B.5.4, at Code Snippet B.8 is the source code for the network of the experiment B_2D_M_N17 which applies L1 & L2 Regularization.

Experiments B_2D_M_N {1, 10 - 20} – Different Configurations

											Training			Validation					Testing				
Experiment Name	GNsize	Dropout	Max-Pooling	Batch Normalization	L1 Regularization	L2 Regularization	Feature Maps	Filters / Kernels Size	Loss Method	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
B_2D_M_N1	5	-	Yes	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1	0.88	0.89	0.87	0.78	0.77	0.79	0.80	0.75	0.61	0.71	0.52	48	0.90
B_2D_M_N10	50	d1 = 0.3 d2, d3 = 0.5	Yes	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1	0.84	0.84	0.83	0.80	0.82	0.80	0.79	0.82	0.61	0.67	0.54	20	0.91
B_2D_M_N11	200	d1 = 0.3 d2, d3 = 0.5	Yes	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1	0.89	0.87	0.91	0.78	0.82	0.76	0.74	0.82	0.60	0.64	0.55	12	0.89
B_2D_M_N12	200	-	Yes	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1	0.90	0.89	0.90	0.78	0.78	0.78	0.77	0.78	0.64	0.75	0.52	10	0.90
B_2D_M_N13	50	-	Yes	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1	0.84	0.86	0.82	0.78	0.78	0.80	0.79	0.77	0.60	0.67	0.53	13	0.89
B_2D_M_N14	50	d1 = 0.3 d2, d3 = 0.5	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_4	0.91	0.89	0.93	0.71	0.69	0.75	0.76	0.66	0.62	0.67	0.56	37	0.85
B_2D_M_N15	50	d1 - 3 = 0.3 d4 - 6 = 0.5	Yes	-	-	-	f1 - 3 = 16 f4, 5 = 32, f6 = 64	3 x 3	MSE	2D_CNN_7L_1	0.77	0.78	0.76	0.79	0.79	0.80	0.79	0.78	0.63	0.66	0.62	12	0.91
B_2D_M_N16	50	d1 = 0.3 d2, d3 = 0.5	Yes	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	Cross Entropy	2D_CNN_4L_1	0.79	0.80	0.79	0.77	0.77	0.77	0.78	0.78	0.64	0.67	0.63	9	0.90
B_2D_M_N17	50	d1 = 0.3 d2, d3 = 0.5	Yes	-	10^{-5}	10^{-4}	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1	0.85	0.85	0.86	0.78	0.80	0.77	0.78	0.80	0.61	0.63	0.61	23	0.90
B_2D_M_N18	50	d1 = 0.3 d2, d3 = 0.5	Yes	Yes	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1	0.82	0.84	0.79	0.79	0.79	0.80	0.80	0.78	-	-	-	17	0.91
B_2D_M_N19	50	d1 = 0.3 d2, d3 = 0.5	Yes	-	-	-	f1, f2 = 32 f3 = 64	5 x 5	MSE	2D_CNN_4L_2	0.82	0.83	0.80	0.81	0.81	0.81	0.80	0.81	0.63	0.73	0.52	17	0.90
B_2D_M_N20	50	d1 = 0.3 d2, d3 = 0.5	Yes	-	-	-	f1, f2 = 32 f3 = 64	7 x 7	MSE	2D_CNN_4L_3	0.83	0.81	0.84	0.79	0.80	0.78	0.76	0.81	0.62	0.70	0.54	23	0.86

Table 5.12. Performance metrics of the experiments B_2D_M_N {1, 10 - 20}.
Comparing different configurations.

Based on Table 5.12 we can see that in most of the experiments, the validation accuracy is between 77% and 81%, except the experiment B_2D_M_N14 where the validation accuracy is 71%. That model most probably overfits since the network configuration is too complex while no Max-Pooling is used and that reduced its generalization ability. In most configurations, overfitting was observed, but it can be mostly be caused because the B_2D_M dataset is relatively small. Data augmentation, could increase the size of the dataset and potentially help reduce overfitting. Among these experiments in Table 5.12, are the two experiments with the best results in this thesis for the AD/NC problem in CNNs; the B_2D_M_N10 and B_2D_M_N19, with average validation accuracies of 80% and 81% respectively.

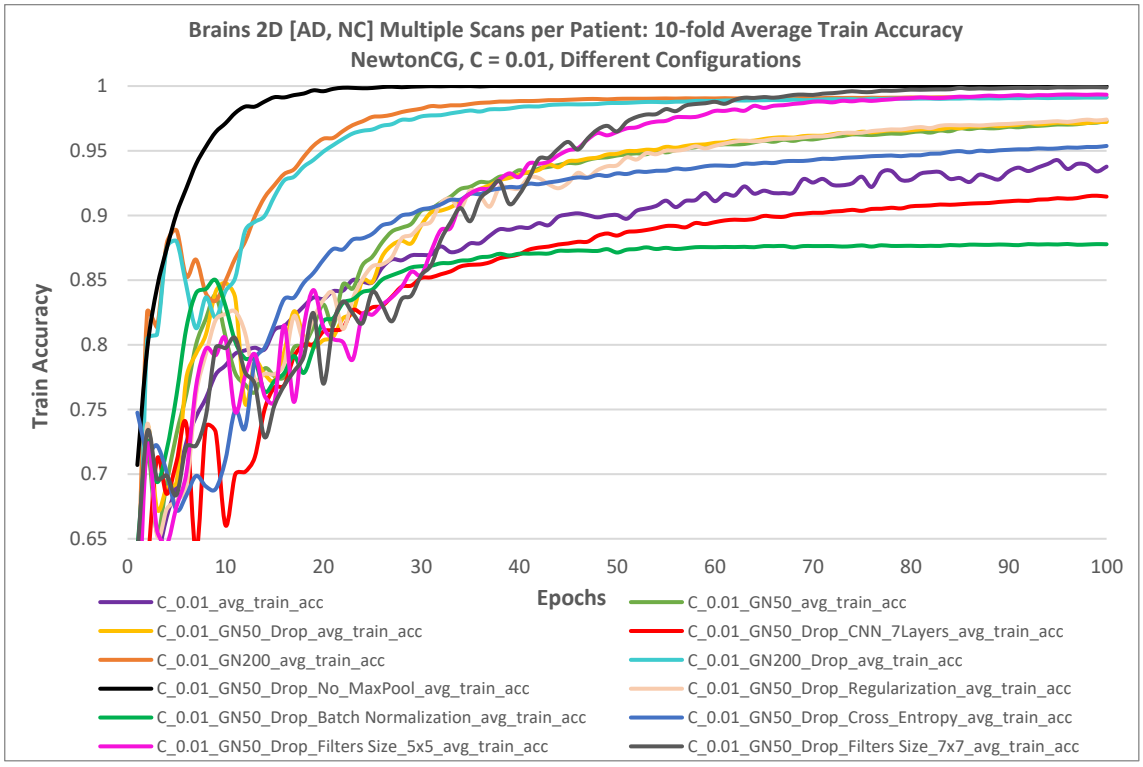


Figure 5.55. Average training accuracies of the experiments B_2D_M_N {1, 10 - 20}. Comparing different configurations.

Figure 5.55 shows the average training accuracies of these experiments. We can distinguish groups of experiments that behave similarly. A pair of experiments that have similarities are the experiments B_2D_M_N19 (pink) and B_2D_M_N20 (gray). Both of them examine how the size of the filters affects the performance, with the experiment B_2D_M_N19 having a filter size = 5×5 and the experiment B_2D_M_N20 a filter size = 7×7. Another group of experiments, that we can spot easily, consists of the experiments B_2D_M_N {11, 12} with GNsize = 200.

The experiment B_2D_M_N14 without Max-Pooling (black) (B_2D_M_N14) has the worst validation accuracy among all the experiments B_2D_M_N {1, 10 – 20}. The reason is that Max-Pooling helps the algorithm to detect features in different locations of the image (Section 2.3.13.5). If no Max-Pooling is applied, then the algorithm will fail to generalize and will overfit. That is why the training accuracy of that experiment goes fast towards 100%. The second worst experiment is the 7-layer one (B_2D_M_N15). In general, the more Convolutional Layers you have, the better the performance should be based on the literature (Krizhevsky et al. 2012). For the AD/NC problem with CNNs and the B_2D_M dataset, it seems that this is not the case.

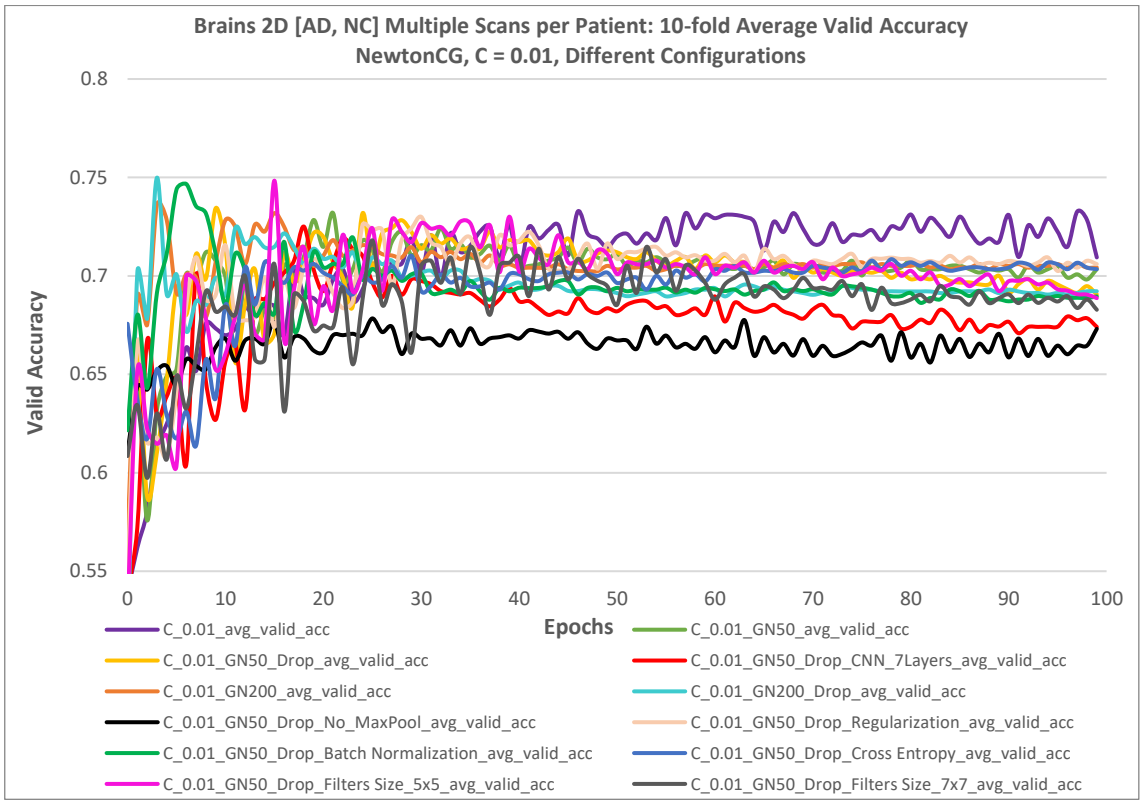


Figure 5.56. Average validation accuracies of the experiments B_2D_M_N {1, 10 - 20}. Comparing different configurations.

In Figure 5.56 of the average validation accuracies, we can see that the average accuracy of the baseline experiment (purple) (B_2D_M_N1) seems to perform better than the rest of the experiment since it increases over time, while the others decline. In Figures 5.57 & 5.58 we can see the average training and validation losses respectively, of the 10-folds, of the experiments B_2D_M_N {1, 10 – 20}. Most of the experiments, seem to have overfitting issues since their validation loss increases slightly over time while their training loss continues to decrease.

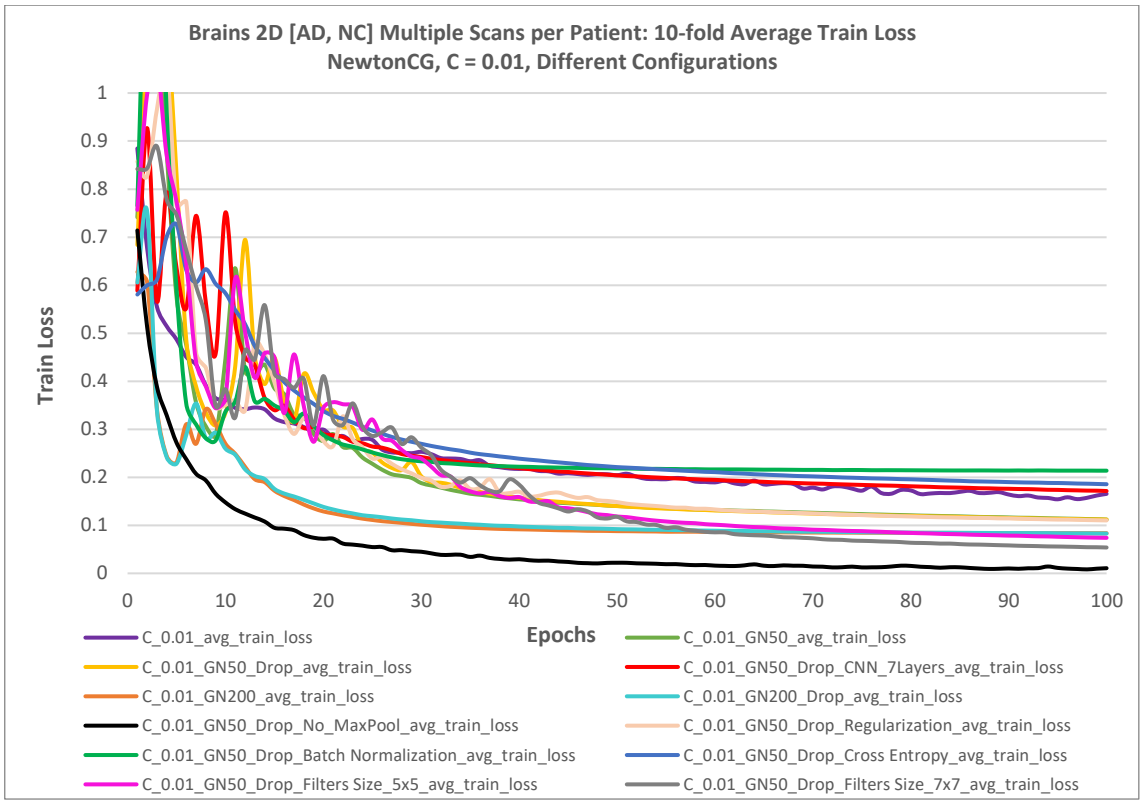


Figure 5.57. Average training losses of the experiments B_2D_M_N {1, 10 - 20}. Comparing different configurations.

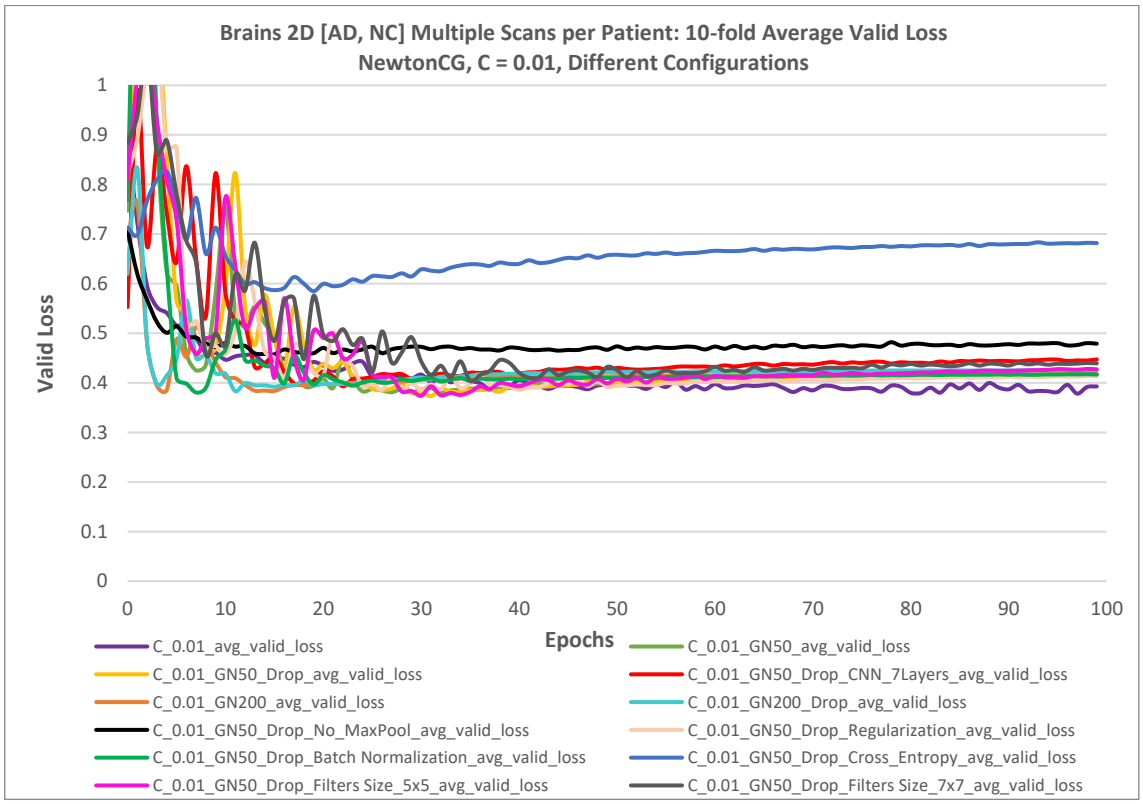


Figure 5.58. Average training losses of the experiments B_2D_M_N {1, 10 - 20}. Comparing different configurations.

5.4.2.6 Experiments B_2D_M_N {10, 14}: C = 0.01, GNsize = 50, With/Without Max-Pooling

The experiment B_2D_M_N14 was meant to compare whether the absence of Max-Pooling (Section 2.3.12.5) affects somehow the performance of the model. In this experiment, Max-Pooling was applied to none of the three Convolutional Layers. We are going to compare this experiment with the B_2D_M_N10 which has the same hyperparameters, and also applies Max-Pooling with pool_size = (2, 2) and stride (2, 2) to all three Convolutional Layers. The experiment B_2D_M_N14 is going to use the 2D_CNN_4L_4 (Section 4.3.2.3.4) network configuration, while the B_2D_M_N10 the 2D_CNN_4L_1 (Section 4.3.2.3.1).

In Figure 5.59, the average training accuracy of the experiment without Max-Pooling in any Convolutional Layer (blue) goes fast towards 100%, while at the same time it has the worse validation accuracy among all the experiments with C = 0.01 (Section 5.4.2.5). This could mean that the model potentially overfits, which does not surprise us since the algorithm does not apply Max-Pooling which helps to recognize similar features in an image that are slightly relocated (Section 2.3.12.5). Additionally, since no Max-Pooling is applied the feature maps have the same size as the original input image of 174×174 pixels since padding (Section 2.3.12.7) is applied. Therefore, the network's complexity increases as well as the execution time.

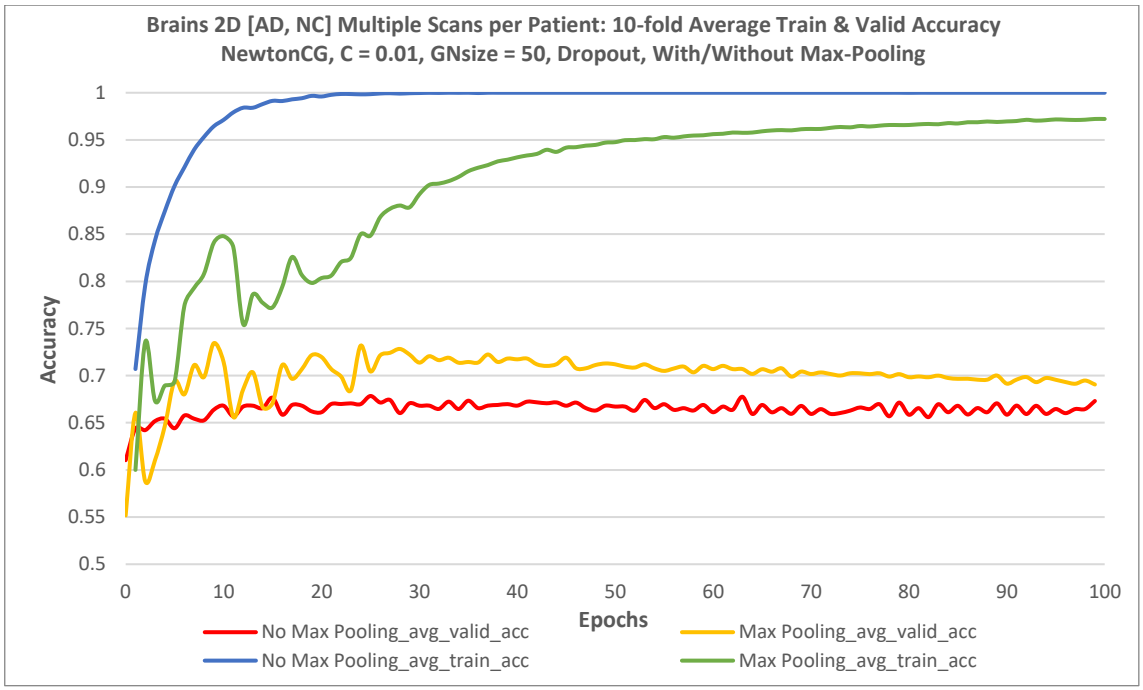


Figure 5.59. Average training and validation accuracies of the experiments B_2D_M_N {10, 14}. With or Without Max-Pooling.

In Figure 5.59, we can see that the average validation accuracy of the experiment B_2D_M_N14 without the Max-Pooling, remains stable after the 20th epoch; while in the experiment B_2D_M_N10 with Max-Pooling, it decreases as the training progresses. If we compare the values of the average validation accuracies from Table 5.12, we can see that the experiment B_2D_M_N14 without Max-Pooling has an average validation accuracy of 71% while for the experiment B_2D_M_N10 with Max-Pooling is 80%.

In the Figure 5.60, of the average training and validation losses of the 10-folds, of the two experiments, we can see that indeed that the validation loss of the experiment without Max-Pooling (red) is larger than the validation loss of the experiment with Max-Pooling (yellow). The training loss of the experiment without Max-Pooling (blue) is very close to zero while the training loss of the experiment with Max-Pooling (green) is close to 0.1 at the last epoch. Both experiments suffer from overfitting since their training loss decreases while their validation loss increases slightly over time.

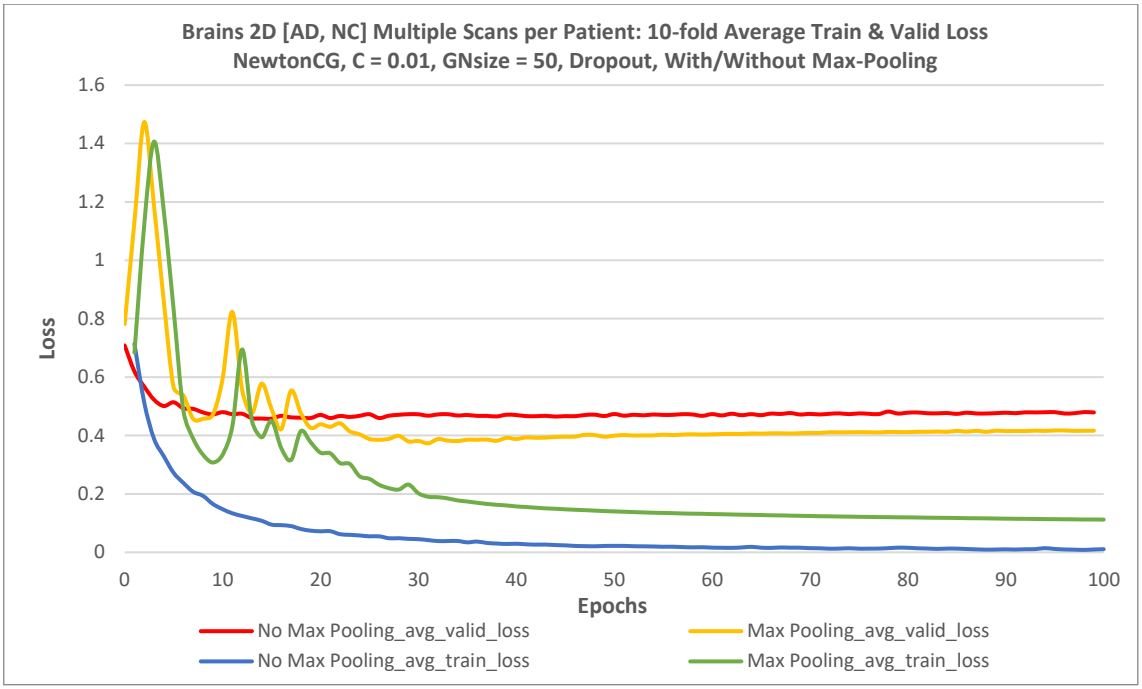


Figure 5.60. Average training and validation losses of the experiments B_2D_M_N {10, 14}. With or Without Max-Pooling.

Detailed results for each fold of the experiment B_2D_M_N14 can be found in Appendix H. More specifically, its training/validation accuracies and losses of each fold are available in Appendix H.2.14. There, are also available the confusion matrices for the training, validation, and test sets of the experiment.

5.4.2.7 Experiments B_2D_M_N {10, 16}:C = 0.01, GNsize = 50, Loss Function = Mean Squared Error (MSE) or Cross-entropy

The experiments B_2D_M_N10 and B_2D_M_N16 use the same network configurations and hyperparameters, except that the first one uses MSE loss (Section 2.3.4) and the other Cross-entropy (Section 2.3.5) as their loss functions. Both experiments use the 2D_CNN_4L_1 network configuration (Section 4.3.2.3.1). The GNsize was set to 50, and the C = 0.01 for the two experiments. Cross Entropy based on the literature, should be used for classification while MSE loss for regression. The reason is that MSE loss does not punish misclassifications enough. In our case, the AD/NC problem is a classification problem but the experiment with Cross-entropy (B_2D_M_N16) performs worse with 77% validation accuracy, while the MSE loss (B_2D_M_N10) has a validation accuracy of 80% (Table 5.7).

In Figure 5.61, the average training and validation accuracies of the 10-folds for the two experiments are visible. We can see with red color the average validation accuracy of the experiment B_2D_M_N16 with the Cross-entropy loss, that it increases over time, while with yellow color for the B_2D_M_N10 with the MSE loss is decreases.

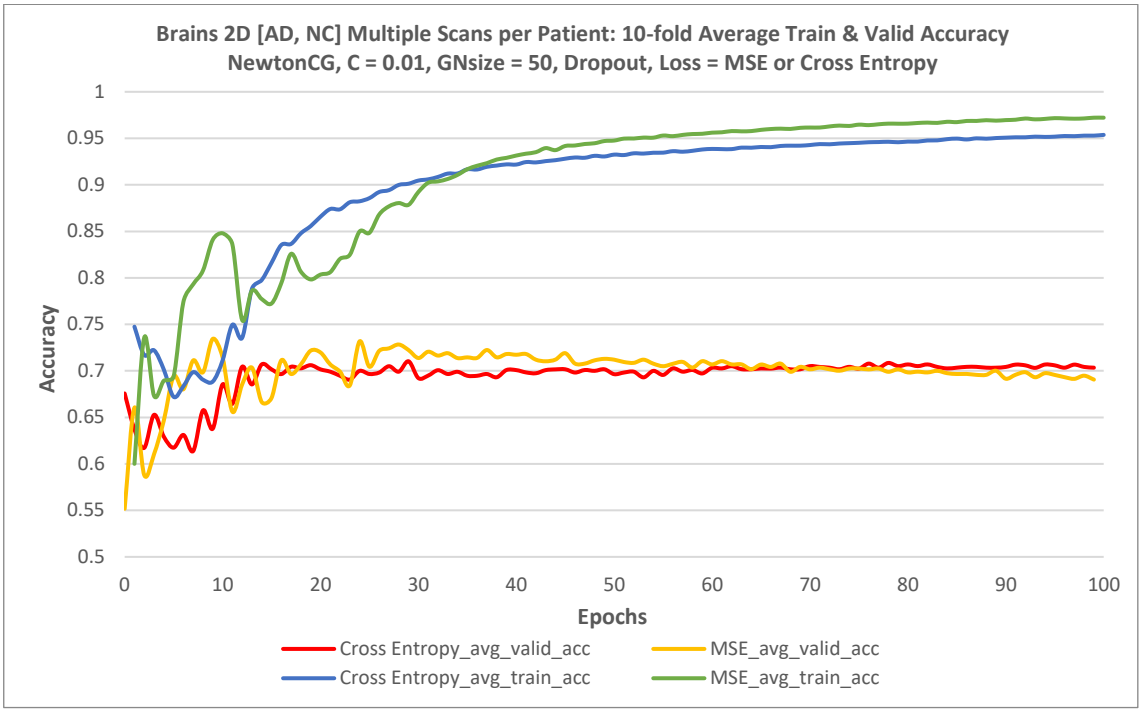


Figure 5.61. Average training and validation accuracies of the experiments B_2D_M_N {10, 16}. MSE or Cross-entropy loss.

In Figure 5.62, we can see the average training and validation losses of the 10-folds, of the two experiments. The validation accuracy of the experiment with the Cross-entropy loss (red) is worse than the experiments with the MSE loss (yellow). Also, it seems that the experiment B_2D_M_N16 overfits more than B_2D_M_N10, since its validation accuracy increases more rapidly over time, while its training accuracy decreases.

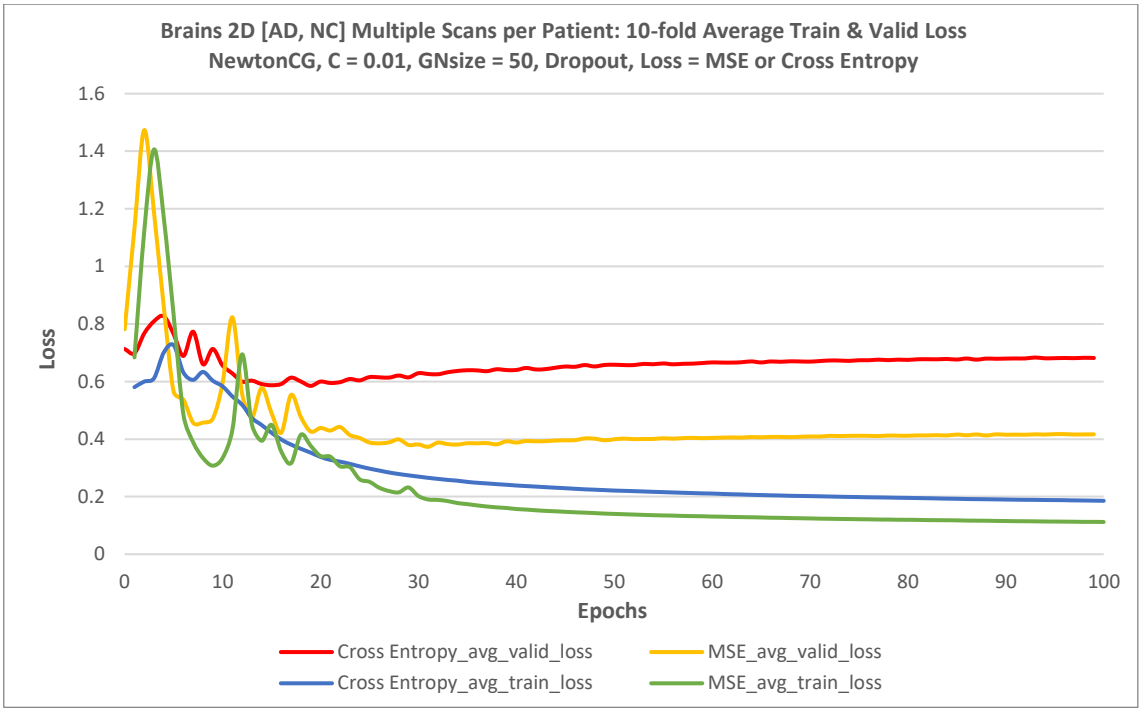


Figure 5.62. Average training and validation losses of the experiments B_2D_M_N {10, 16}. MSE or Cross-entropy loss.

5.4.2.8 Experiments B_2D_M_N {10, 17}: C = 0.01, GNsize = 50, With/Without L1 & L2 Regularization

The experiment B_2D_M_N17 applies L1 & L2 Regularization (Section 2.3.13.7), as an attempt to reduce the overfitting issues of the B_2D_M_N10 experiment. The training/validation accuracies and losses per fold, of the two experiments B_2D_M_N {10, 17}, are available in Appendix H.2.10 and Appendix H.2.16 respectively, with their confusion matrices as well.

The values for the regularization in experiment B_2D_M_N17 that have been applied, are L1 = 0.00001 and L2 = 0.0001, the most common values that are frequently used in CNNs. The L1 & L2 Regularizations were applied to all three Convolutional Layers. The basic structure of the network configuration used in both experiments is 2D_CNN_4L_1 (Section 4.3.2.3.1). Detailed source code for the modifications made in the network configuration of the B_2D_M_N17 experiment, can be found in Appendix B.5.4, at Code Snippet B.8. Both experiments have GNsize = 50, C = 0.01, and apply Dropout (Section 2.3.13.9) to all three Convolutional Layers.

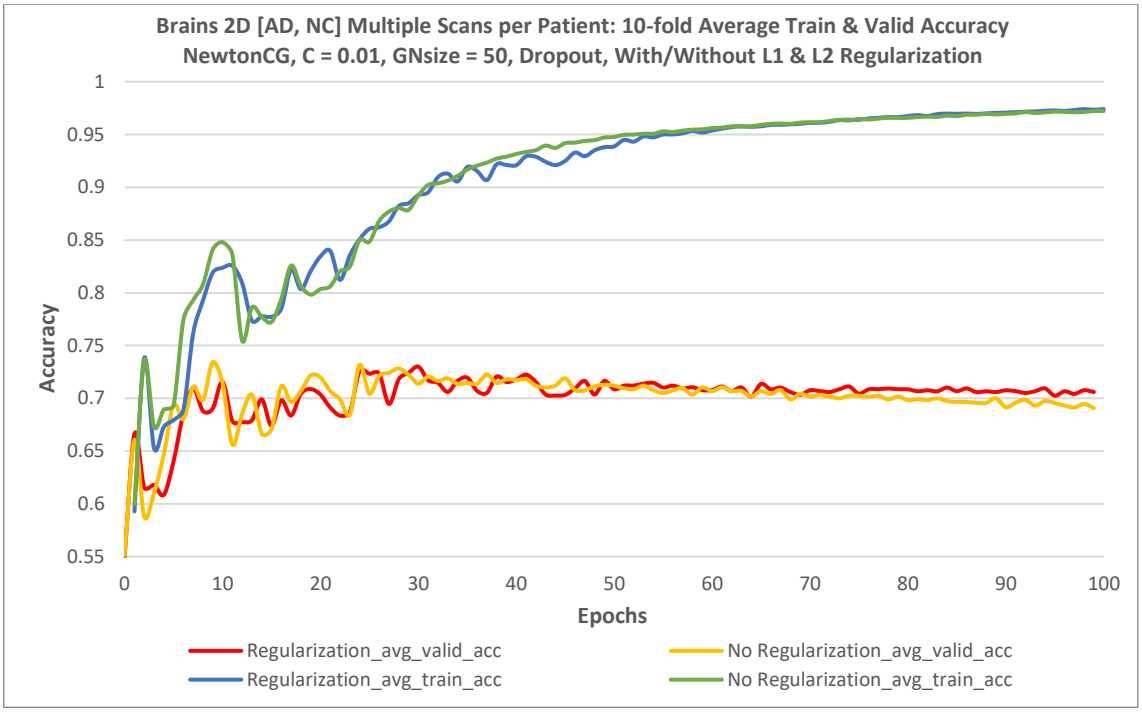


Figure 5.63. Average training and validation accuracies of the experiments B_2D_M_N {10, 17}. With or Without L1 & L2 Regularization.

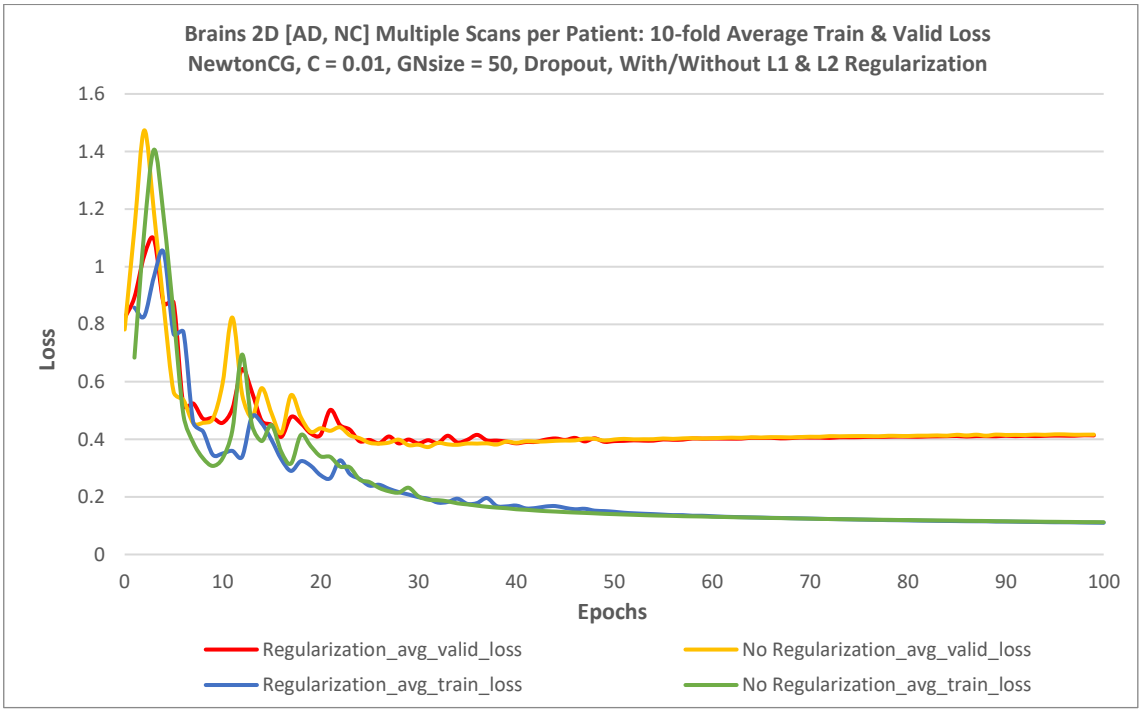


Figure 5.64. Average training and validation losses of the experiments B_2D_M_N {10, 17}. With or Without L1 & L2 Regularization.

Figure 5.63 shows the average training and validation accuracies of the 10-folds, for the two experiments that we are comparing. Overall, their behavior is the same; except that, the validation accuracy of the experiment B_2D_M_N10 without regularization (yellow)

seems to slightly decrease over time after the 30th epoch, while the validation accuracy of the experiment B_2D_M_N17 with regularization is more stable. Additionally, in Figure 5.64, the average training/validation losses of the two experiments are available. The experiment B_2D_M_N17 with regularization is slightly better since its validation loss increases less than the validation loss of the experiment B_2D_M_N10 without.

We can say that the regularization helped somehow to reduce the overfitting problem, but not to solve it, since the average validation accuracy of the experiment B_2D_M_N17 was 78% while for the B_2D_M_N10 was 80% (Table 5.7). Maybe the values for L1 and L2 that were used were very small, that is why no much variation was observed between the two experiments. In future experiments, larger values of L1 and L2 may need to be checked whether can encounter overfitting.

5.4.2.9 Experiments B_2D_M_N {10, 18}: C = 0.01, GNsize = 50, With/Without Batch Normalization

The experiment B_2D_M_N18 was meant to examine whether Batch Normalization affects the performance of the model. Batch normalization though was applied between the last Convolutional Layer and the dense layer which it was turned out, not to be the best practice. Based on the literature (Ioffe and Szegedy, 2015) Batch Normalization should be applied before each Convolutional Layer. Batch normalization has the potential to improve the performance of the model. The network used for the experiment was the 2D_CNN_4L_1 (Section 4.3.2.3.1), and the source code for the network's implementation with Batch Normalization is available in Appendix B.5.6, at Code Snippet B.13. We are going to compare the results of the experiment B_2D_M_N18 with the baseline B_2D_M_N10 since except for the Batch Normalization, the rest of the hyperparameters are the same.

As we can see in Figure 5.65, batch normalization significantly decreased the training accuracy, while the validation accuracy remains relatively the same between the two experiments. This is a good sign because the difference between the validation accuracy and the training accuracy is less. Therefore, if batch normalization is applied optimally in feature work, it could increase substantially the performance of the network.

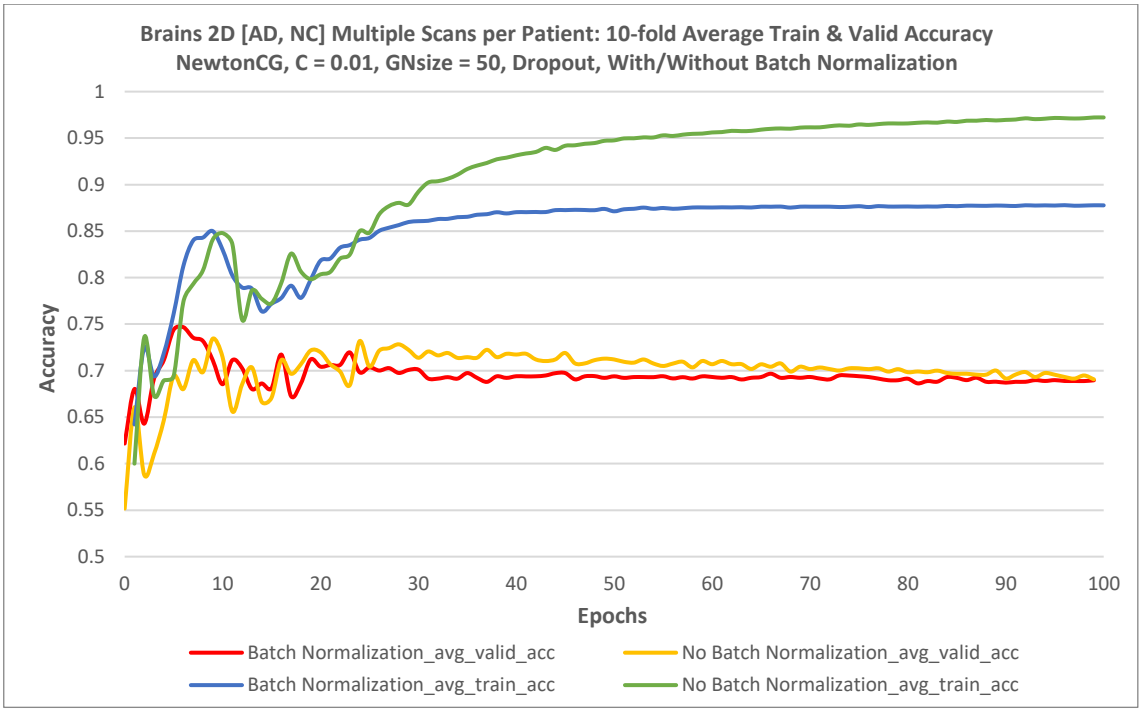


Figure 5.65. Average training and validation accuracies of the experiments B_2D_M_N {10, 18}. With or Without Batch Normalization.

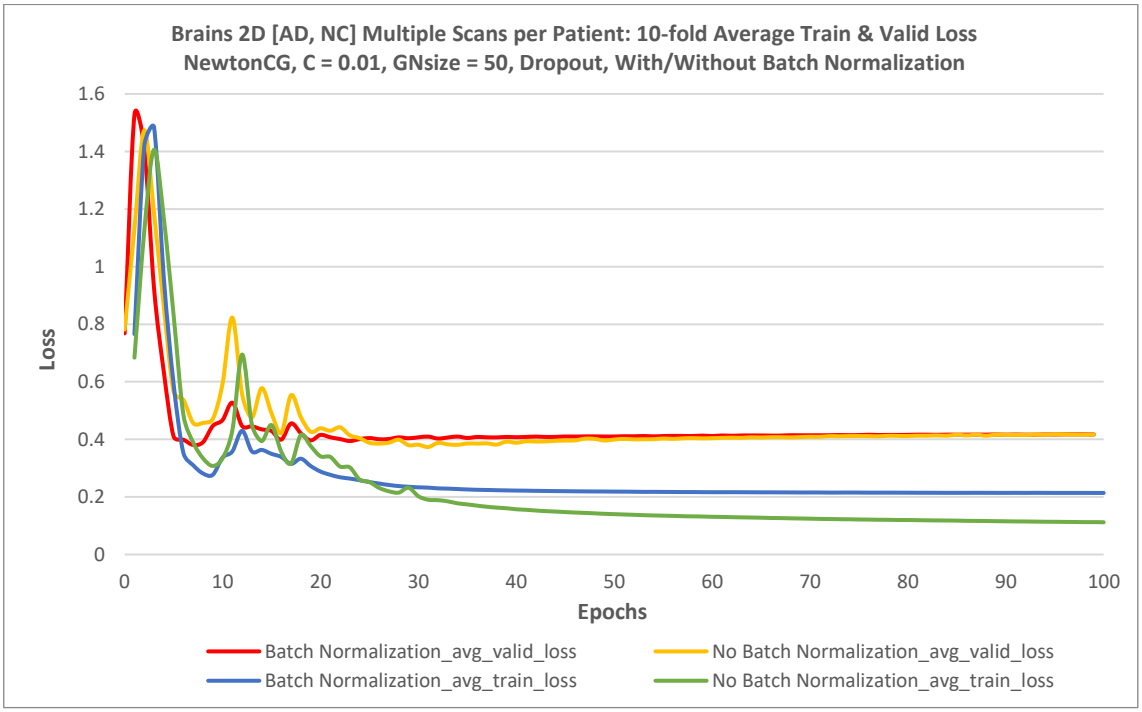


Figure 5.66. Average training and validation losses of the experiments B_2D_M_N {10, 18}. With or Without Batch Normalization.

The average validation accuracy, of the experiment B_2D_M_N18, with the batch normalization, is 79%; while for the experiment B_2D_M_N10, without batch normalization is 80% (Table 5.7). In Figure H.177 of Appendix H.2.17, of the

B_2D_M_N18 experiment's validation accuracies per fold; each fold behaves differently. Therefore, the Batch Normalization did not resolve the issues of variation between folds that the B_2D_M_N10 experiment also had.

Figure 5.66, shows the average training and validation losses for the two experiments. Their validation losses are very similar, but the training loss of the B_2D_M_N18 experiment is significantly larger than the one in the B_2D_M_N10.

5.4.2.10 Experiments B_2D_M_N {10, 19, 20}: C = 0.01, GNsize = 50, Filters/Kernels Size = {3 × 3, 5 × 5, 7 × 7}

The experiments B_2D_M_N {10, 19, 20} compare the effect of different filter/kernel sizes. The filer size for all three Convolutional Layers of the experiments B_2D_M_N {10, 19, 20} are 3×3, 5×5, and 7×7 respectively. All three implementations have the same hyperparameters, C = 0.01, GNsize = 50, and apply Dropout. The three network configurations of the experiments are 2D_CNN_4L_1 (Section 4.3.2.3.1), 2D_CNN_4L_2 (Section 4.3.2.3.2), and 2D_CNN_4L_3 (Section 4.3.2.3.3) for the experiments B_2D_M_N {10, 19, 20} respectively.

In Figure 5.67, the average training and validation accuracy per fold of the three experiments are available. The training accuracy of the experiment with filter size = 3×3 is slightly less than the one in the other two experiments. Additionally, the validation loss of the experiment with filter size = 7×7 is slightly less than the other two experiments.

The major difference between the different configurations is in the average of the best accuracies. The experiment with filer size = 3×3 (B_2D_M_N10) has a validation accuracy equal to 80%; for filter size = 5×5 (B_2D_M_N19) the validation accuracy is equal to 81%, while for filter size = 7×7 (B_2D_M_N20) the validation accuracy is 79% (Table 5.7). This makes the filter size = 5×5 the best choice for the specific network architecture with NewtonCG and the specific dataset (B_2D_M).

In Figure 5.68, with the average training and validation losses of the experiments, we can see that all three of them suffer from overfitting, since their validation loss increases slightly after the 30^{th} epoch approximately, while their training loss continues to decrease.

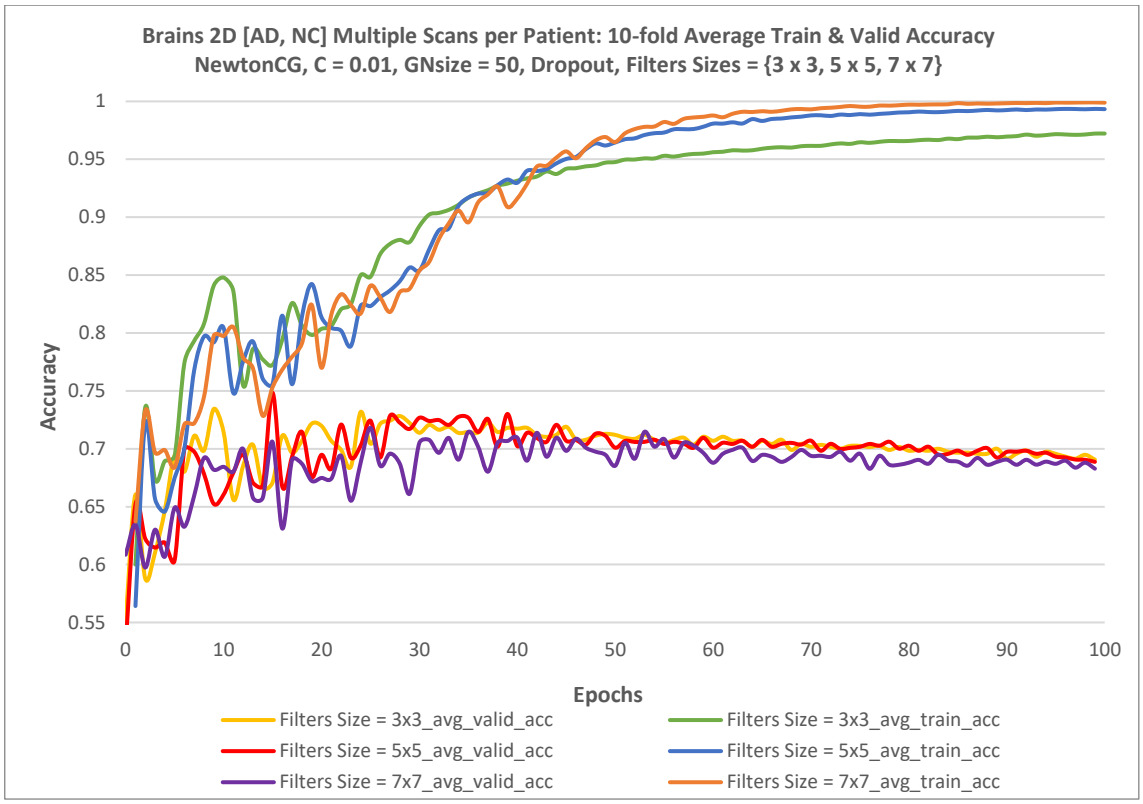


Figure 5.67. Average training and validation accuracies of the experiments B_2D_M_N {10, 19, 20}. Different filter sizes {3 × 3, 5 × 5, 7 × 7}.

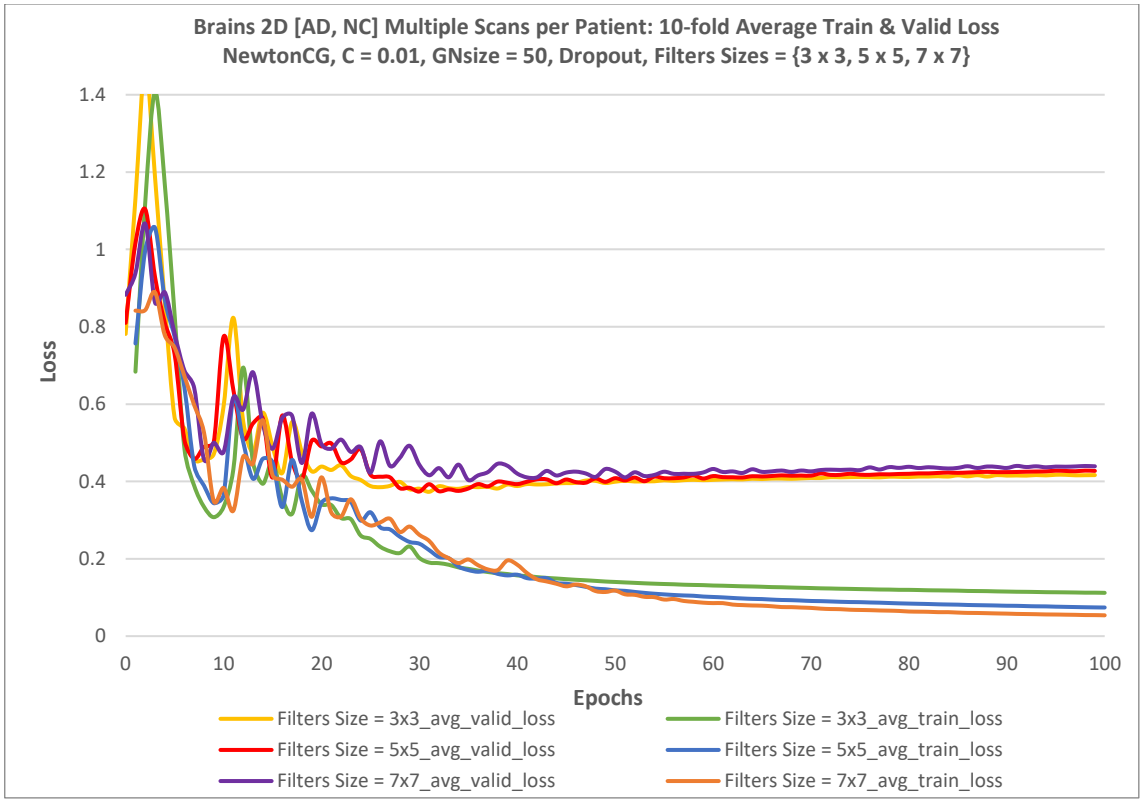


Figure 5.68. Average training and validation losses of the experiments B_2D_M_N {10, 19, 20}. Different filter sizes {3 × 3, 5 × 5, 7 × 7}.

5.4.2.11 Experiments B_2D_M_N {10, 21}: C = 0.01, Dropout vs. Spatial Dropout

For CNN architectures the Spatial Dropout (Section 2.3.13.10) is preferred over the standard Dropout (Section 2.3.13.9). Their difference is that the standard Dropout drops individual neurons/elements while the Spatial Dropout, drops entire feature maps. To compare the differences between Dropout and Spatial Dropout in practice, the experiments B_2D_M_N10 (Dropout) and B_2D_M_N21 (Spatial Dropout) have been performed. The rest of the networks' hyperparameters remain the same, as well as the network configurations. The Spatial Dropout was set to sd1 = sd2 = sd3 = 50%, where sdi is the probability of a feature map to be dropped in the i^{th} Convolutional Layer. The standard Dropout used d1 = 30%, d2 = d3 = 50%, where di is the drop probability of an element in the i^{th} Convolutional Layer.

The network configuration that implements the Spatial Dropout can be found in Appendix B.5.6, at Code Snippet B.10; while for the standard Dropout, in Appendix B.5.2, at Code Snippet B.6. Both experiments use the network 2D_CNN_4L_1 (Section 4.3.2.3.1).

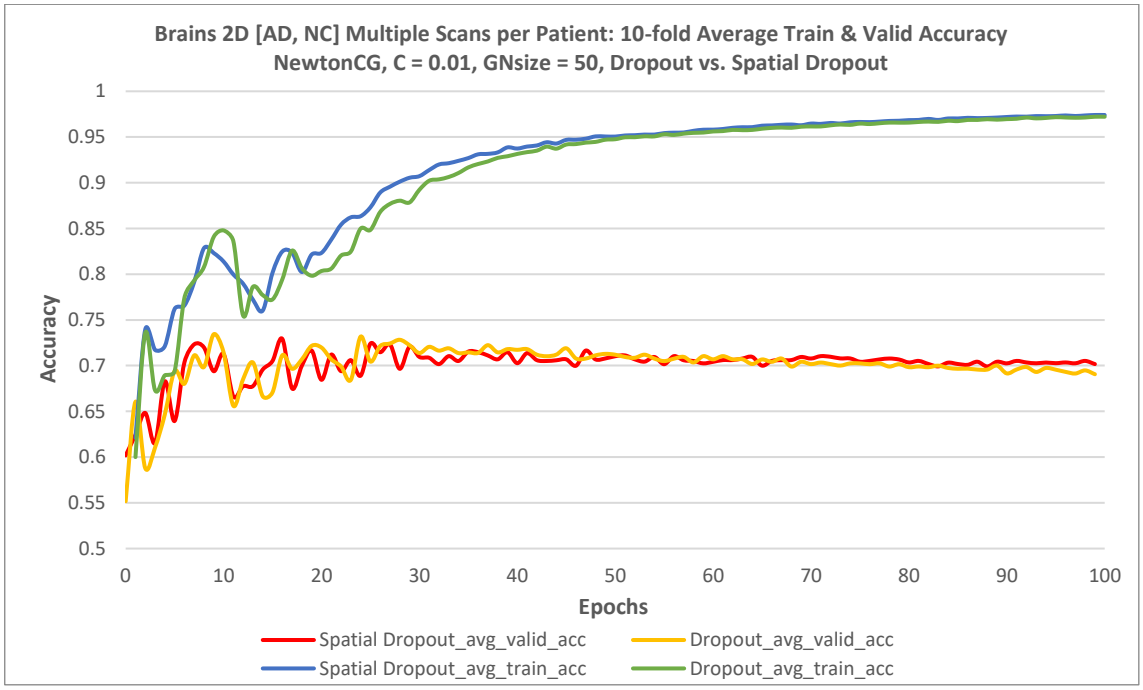


Figure 5.69. Average training and validation accuracies of the experiments B_2D_M_N {10, 21}. Dropout or Spatial Dropout.

In Figures 5.69 & 5.70, we observe that the average training and validation accuracies and losses of the two experiments are very similar. We notice that after the 30^{th} epoch the validation accuracy of the experiment with Dropout (yellow) shows a slight decline (30^{th}

epoch: 72%; 100th epoch: 68%), while the experiment with the Spatial Dropout (red) remains stable in a certain average validation accuracy (30th – 100th epoch: ~71%).

The average validation accuracy of the best accuracies per fold for the Spatial Dropout and Dropout experiments are 79% and 80% respectively. The specificity, i.e., ADs correctly classified as ADs, is the same in both experiments, equal to 82% (Table 5.7).

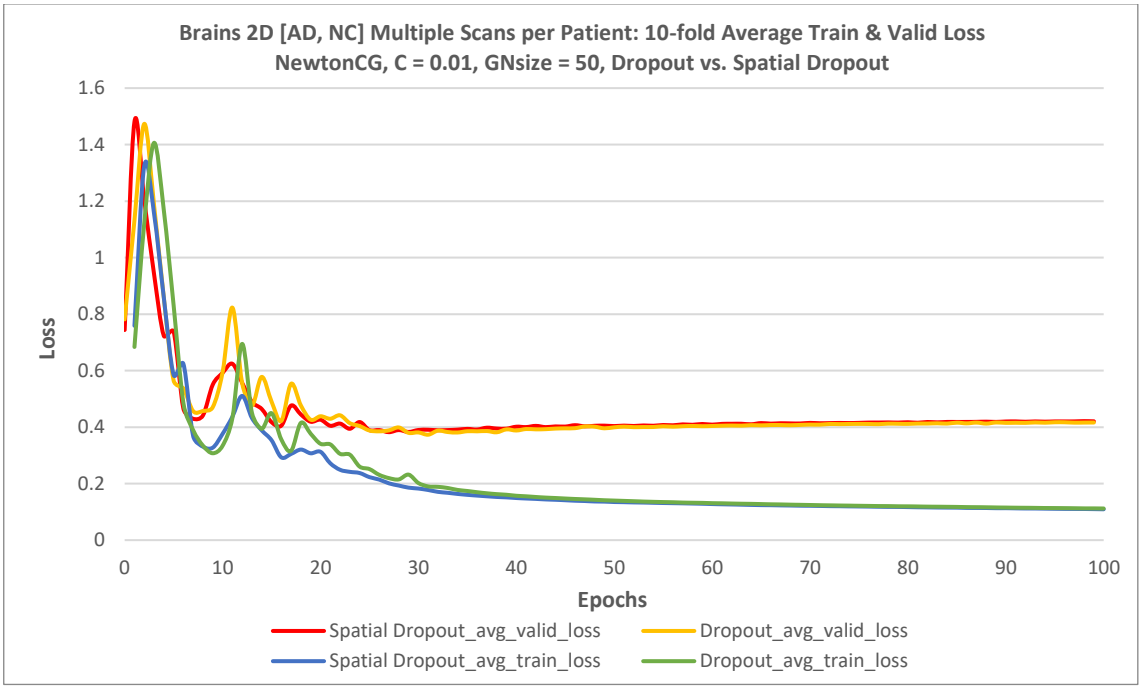


Figure 5.70. Average training and validation losses of the experiments B_2D_M_N {10, 21}. Dropout or Spatial Dropout.

5.4.2.12 Experiments B_2D_M_N {21 – 27}: C = 0.01, Spatial Dropout

All the experiments B_2D_M_N {21 – 27} have the same hyperparameters, C = 0.01, GNsize = 50, epochs = 100, and they apply Spatial Dropout to all three Convolutional Layers, with a probability of drop equal to 50%. The filter sizes used are all 3×3, and the loss function is MSE loss for all the experiments. The experiments B_2D_M_N {22, 25 – 27} that apply standard Dropout as well, their probabilities of dropping an element per Convolutional Layer are d1 = 30%, d2 = d3 = 50%. In Table 5.13, the performance metrics of each experiment, and their hyperparameters with their network configurations are available.

In Appendixes, H.2.20 – H.2.26, the training/validation accuracy and loss per fold, as well as the training, validation, and testing confusion matrices of the experiments B_2D_M_N {21 – 27} respectively, can be found.

Experiments B_2D_M_N {21 – 27}: C = 0.01, GNsize = 50, Spatial Dropout, Different Configurations																						
								Training			Validation					Testing						
Color in Figures 5.71 – 5.74	Experiment Name	Spatial Dropout	SoftMax	Dropout	L1 Regularization	L2 Regularization	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
	B_2D_M_N21	Yes	-	-	-	-	2D_CNN_4L_1	0.84	0.83	0.87	0.79	0.84	0.73	0.76	0.82	0.60	0.62	0.59	14	0.91
	B_2D_M_N22	Yes	-	Yes	-	-	2D_CNN_4L_1	0.85	0.87	0.84	0.78	0.75	0.81	0.81	0.78	0.63	0.67	0.61	17	0.89
	B_2D_M_N23	Yes	-	-	0.00001	0.0001	2D_CNN_4L_1	0.79	0.81	0.79	0.78	0.78	0.78	0.79	0.79	0.60	0.62	0.58	19	0.91
	B_2D_M_N24	Yes	Yes	-	-	-	2D_CNN_4L_1	0.81	0.84	0.79	0.76	0.70	0.82	0.80	0.74	0.62	0.65	0.61	13	0.86
	B_2D_M_N25	Yes	Yes	Yes	-	-	2D_CNN_4L_1	0.78	0.81	0.77	0.75	0.71	0.79	0.78	0.73	0.63	0.65	0.62	12	0.88
	B_2D_M_N26	Yes	Yes	Yes	0.00001	0.0001	2D_CNN_4L_1	0.79	0.80	0.80	0.77	0.76	0.77	0.78	0.77	0.61	0.64	0.60	12	0.85
	B_2D_M_N27	Yes	Yes	Yes	0.00001	0.0001	2D_CNN_5L_1	0.80	0.80	0.81	0.75	0.77	0.73	0.75	0.77	0.59	0.60	0.59	17	0.83

Table 5.13. Performance metrics of the **e**xperiments B_2D_M_N {21 – 27}.
Different configurations with Spatial Dropout.

In Figures 5.71 – 5.74, we can easily distinguish two groups of experiments. The first group (red, blue, green) contains the experiments B_2D_M_N {21 – 23}, and the second group (purple, black, orange, yellow) contains the experiments B_2D_M_N {24 – 27}. The first group does not apply SoftMax (Section 2.2.14) on the output layer while the second group does. The differences between experiments in the same group are minor. The validation accuracies of members of the same group are similar as well since the first group's accuracies are between 78% and 79%, while for the second group are between 75% and 77% (Table 5.13). Since the first group in general performs better than the second one, we can assume that SoftMax does not improve the performance of our model.

Based on Table 5.13, from the first group, the experiment B_2D_M_N21 has the best results, the largest average validation accuracy equal to 79%, and the best specificity equal to 82%. The experiment B_2D_M_N21 does not apply any standard Dropout, L1, or L2 Regularizations neither has a more complex 5-layer architecture. In Figures 5.71 & 5.72, we can see the average training and validation accuracies respectively, for the experiments B_2D_M_N {21 – 27}. The first group has in general greater average training accuracies than the second one (Figure 5.71). Based on the validation accuracies

(Figure 5.72), the experiment B_2D_M_N27 (yellow) with the two dense layers, has the worst performance between the members of the second group.

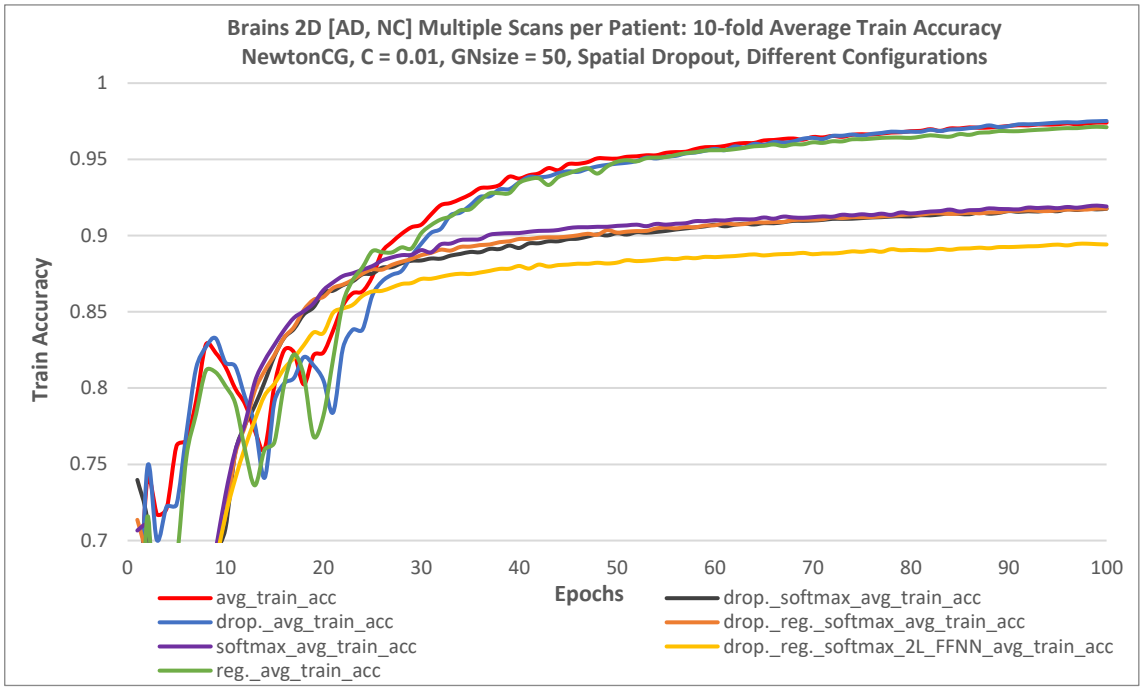


Figure 5.71. Average training accuracies of the experiments B_2D_M_N {21 - 27}.

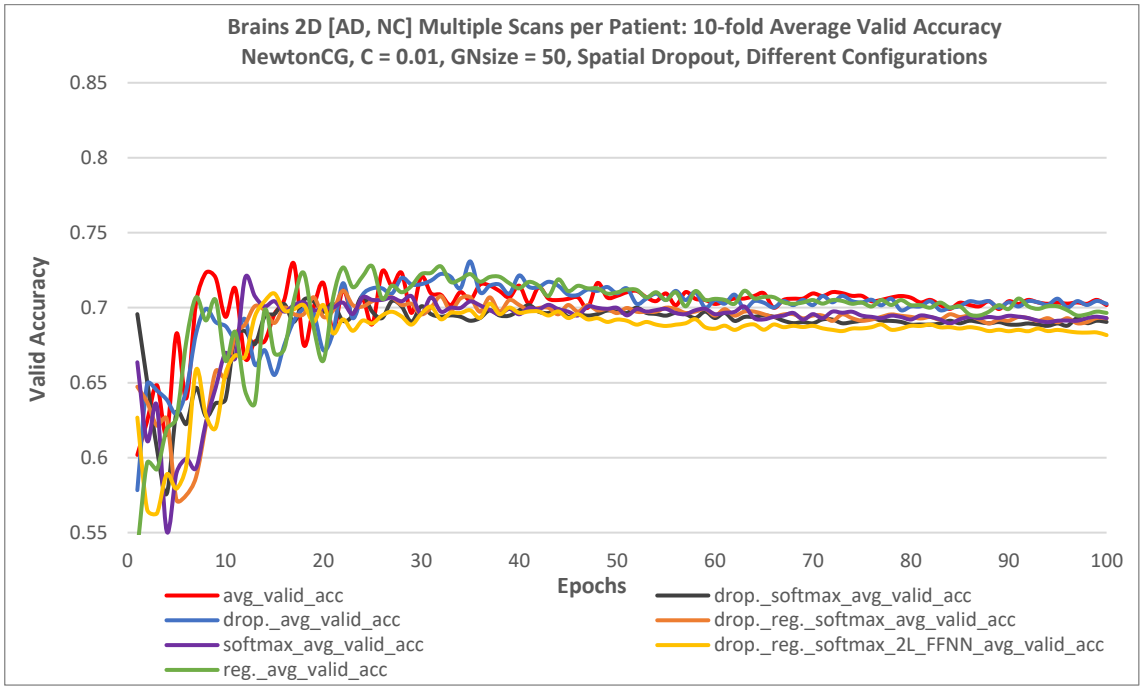


Figure 5.72. Average validation accuracies of the experiments B_2D_M_N {21 - 27}.

A slight increase in the validation losses (Figure 5.74) is observed for all seven experiments, which means that in any case the overfitting issue is not resolved. The

experiments B_2D_M_N {21 – 23} without SoftMax, and the experiment B_2D_M_N27 with the two dense layers, suffer a bit more from overfitting, since their training loss decreases (Figure 5.73), while their validation loss increases slightly more than the rest of the experiments.

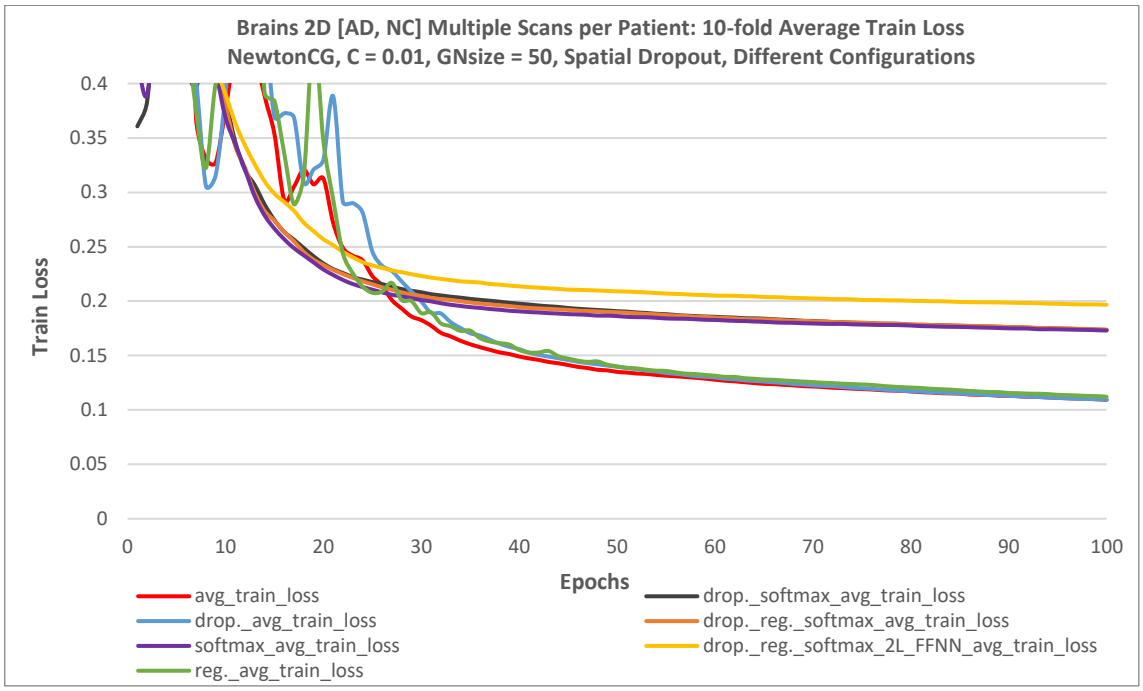


Figure 5.73. Average training losses of the experiments B_2D_M_N {21 - 27}.

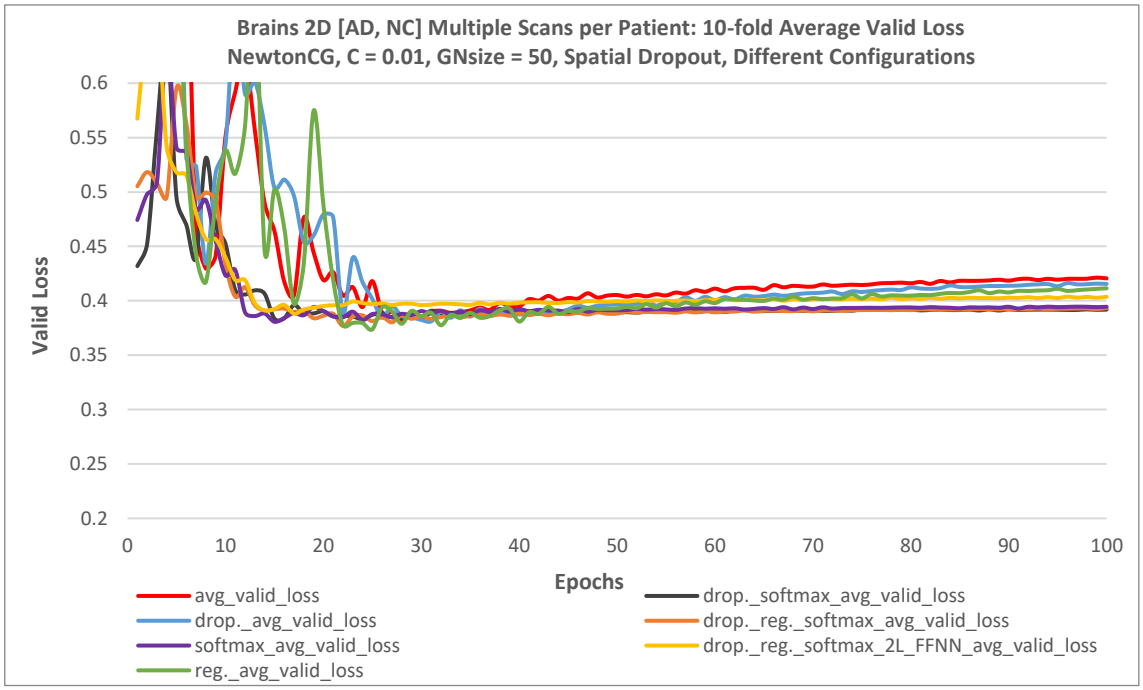


Figure 5.74. Average validation losses of the experiments B_2D_M_N {21 - 27}.

5.4.3 CNN 7 Layers with NewtonCG

5.4.3.1 Experiments B_2D_M_N {10, 15}: C = 0.01, GNsize = 50, Dropout, 4 Layers vs. 7 Layers

The experiment B_2D_M_N15 was held to compare the performance of a deeper 7-layer CNN network the 2D_CNN_7L_1 (Section 4.3.2.5.1), over the 4-layer 2D_CNN_4L_1 (Section 4.3.2.3.1), of the experiment B_2D_M_N10 (Appendix H.2.10). Based on Krizhevsky et al. (2012), the deeper the Convolutional Neural Network, the better its performance would be. Therefore, the network of the B_2D_M_N15 experiment, has 7 layers, 6 Convolutional Layers, and a single dense layer, while the B_2D_M_N10 experiment, has 4 layers, 3 Convolutional Layers, and a single dense layer. The rest of the hyperparameters are completely the same. The source code for the two network implementations is available in Appendix B.5.2, at Code Snippet B.6.

Both networks apply Dropout as well. For di being the probability of dropping a neuron in the i^{th} Convolutional Layer, the values for the experiment B_2D_M_N15 are d1 = d2 = d3 = 30%, and d4 = d5 = d6 = 50%, while in the B_2D_M_N10 experiment are d1 = d2 = 30%, and d3 = 50%. The number of filter maps fi of the i^{th} Convolutional Layer are f1 = f2 = f3 = 16, f4 = f5 = 32, and f6 = 64, for the B_2D_M_N15, and f1 = f2 = 32, f3 = 64, for the B_2D_M_N10.

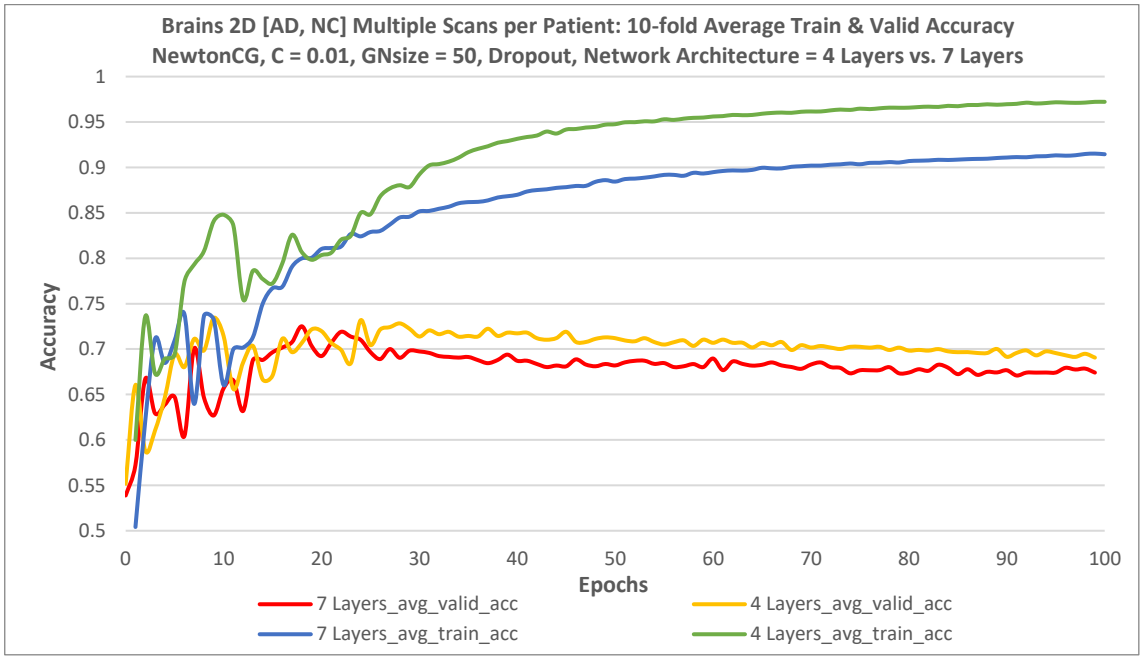


Figure 5.75. Average training and validation accuracies of the experiments B_2D_M_N {10, 15}. 4-layer or 7-layer CNN.

The extra Convolutional Layers were expected to increase the validation accuracy, but in our case, they did not. Thus, the 7-layer CNN performs worse than the 4-layer CNN in the AD/NC problem, with the B_2D_M dataset. The average validation accuracy for the 4-layer experiment is 80% while for the 7-layer is 79% (Table 5.7). In Figure 5.75, we observe that both the average training and validation accuracies of the 7-layer experiment are significantly less than the corresponding ones in the 4-layer experiment. Consequently, the extra complexity increases only the execution time. Based on Figure 5.76, equally the 7-layer experiment (B_2D_M_N15) and the 4-layer experiment (B_2D_M_N10) suffer from overfitting.

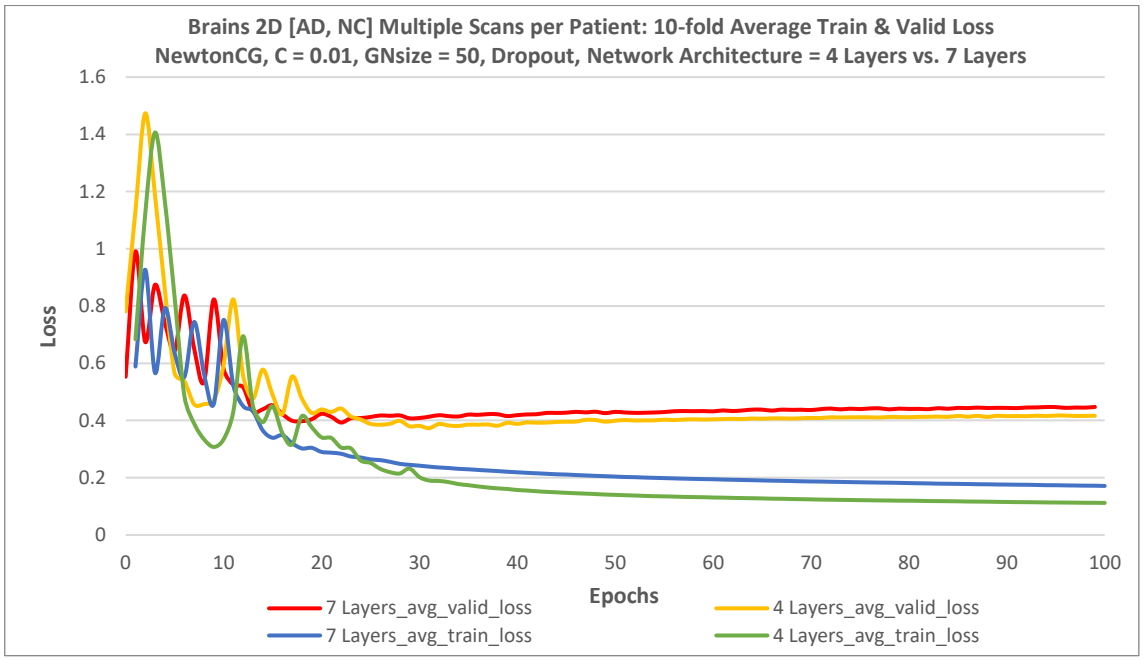


Figure 5.76. Average training and validation losses of the experiments B_2D_M_N {10, 15}. 4-layer or 7-layer CNN.

5.5 2D Brain Slices [AD, NC]: Single Scan per Patient (174 × 174)

After observing overfitting in most of the experiments (Section 5.4) with the B_2D_M dataset (Section 3.2.3.3), I assumed that this could be happening because multiple scans of the same patient are used in the same dataset. Since the network, learns multiple images of the same person, this could imply that it memorizes specific features of that person; e.g., the shape of a patient brain's left hemisphere. So, a dataset that would contain only a single scan per patient was created, the B_2D_S (Section 3.2.3.2). Therefore, the experiments of this section, are going to test the performance of the B_2D_S dataset, in CNNs and NewtonCG optimizer.

Additionally, in these experiments, we are going to investigate how the 2D_CNN_4L_5, a narrow network (Section 4.3.2.3.5) of the 2D_CNN_4L_6, a wide network (Section 4.3.2.3.6), affects the model's performance in comparison with the baseline network, 2D_CNN_4L_1 (Section 4.3.2.3.1). Usually, very wide and shallow networks are not being used, since they tend to be very good at memorization, but not so good at generalization. Usually, deeper CNNs are preferred, since in a large spectrum of problems with image recognition, had been proven to perform better, than shallower network configurations (Kaiming et al. 2015).

5.5.1 CNN 4 Layer with NewtonCG

5.5.1.1 Experiments B_2D_S_N1 & B_2D_M_N7: C = 1, GNsize = 5, Dropout, Multiple Scans per Patient vs. Single Scan per Patient

The experiments B_2D_S_N1 and B_2D_M_N7 have the same network configuration 2D_CNN_4L_1 (Section 4.3.2.3.1) with Dropout (Section 2.3.13.9) applied on each Convolutional Layer with a 30% probability of drop. The weight decay, C was set equal to 1, and the GNsize equal to 5. The loss function for both experiments is MSE loss and the optimizer NewtonCG. The only difference between the two experiments is the dataset that had been used. The B_2D_S_N1 uses the 2D Brain Slices [AD, NC], Single Scan per Patient – Single Slice per Scan (B_2D_S) dataset (Section 3.2.3.2), while the B_2D_M_N7 uses the 2D Brain Slices [AD, NC], Multiple Scans per Patient – Single Slice per Scan (B_2D_M) dataset (Section 3.2.3.2).

Experiments B_2D_S_N1 & B_2D_M_N7 Multiple Scans per Patient vs. Single Scan per Patient													
	Training			Validation					Testing				
Experiment Name	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
B_2D_M_N7	0.90	0.91	0.90	0.76	0.76	0.77	0.77	0.74	0.62	0.71	0.53	56	0.86
B_2D_S_N1	0.85	0.86	0.84	0.75	0.75	0.77	0.77	0.74	0.68	0.68	0.68	22	0.87

Table 5.14. Performance metrics of the experiments B_2D_S_N1 & B_2D_M_N7. Multiple Scans per Patient vs. Single Scan per Patient

Based on Table 5.14, with the performance metrics of the two experiments, the validation performance metrics (accuracy, PPV, NPV, sensitivity, specificity) of the two datasets are very similar. The main difference in terms of performance is in the metrics of the training set. The B_2D_M_N7 experiment has a training accuracy equal to 90% while the B_2D_S_N1 is 85%.

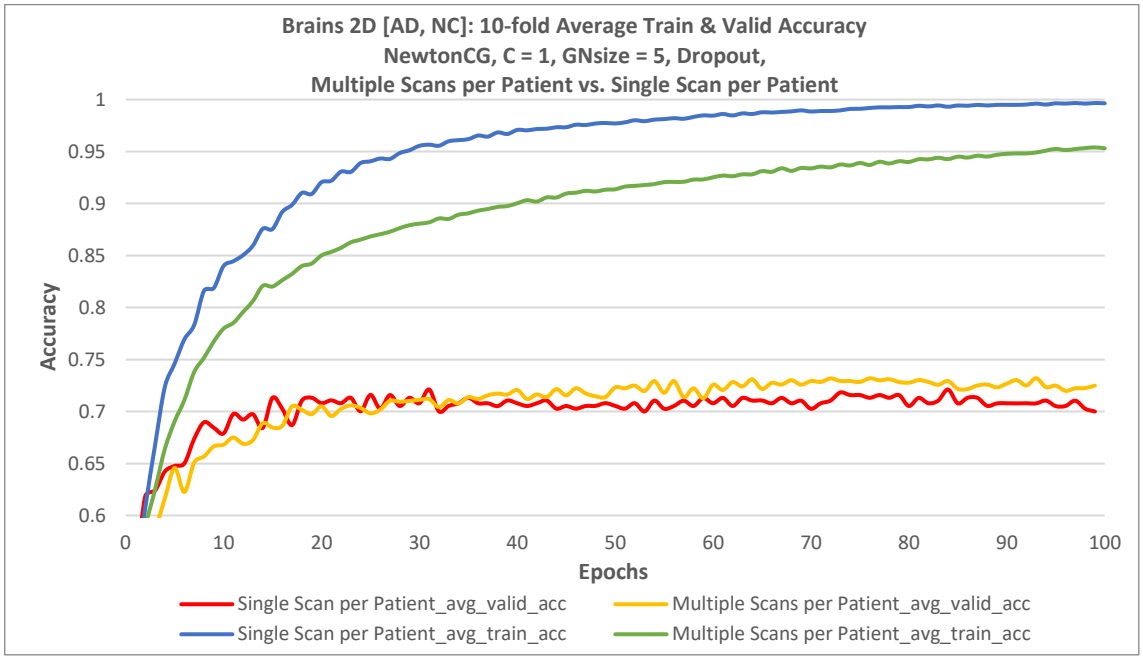


Figure 5.77. Average training and validation accuracies of the experiments B_2D_M_N7 & B_2D_S_N1. Multiple Scans per Patient vs. Single Scan per Patient.

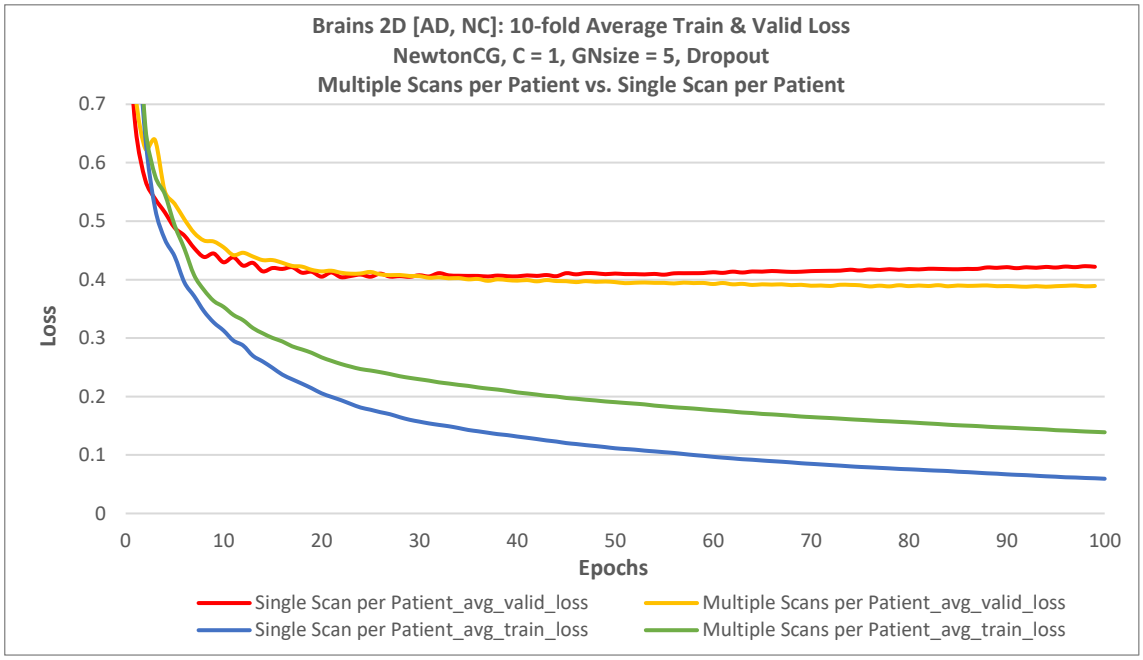


Figure 5.78. Average training and validation losses of the experiments B_2D_M_N7 & B_2D_S_N1. Multiple Scans per Patient vs. Single Scan per Patient.

From the plot of average accuracies plot (Figure 5.77), we observe that the training accuracy of the B_2D_S_N1 (blue) increases more rapidly than the training accuracy of the B_2D_M_N7 (green). The validation accuracy of the B_2D_S_N1 (red) after 20 epochs remains almost stable till the end of the training process. On the other hand, the validation accuracy of the B_2D_M_N7 experiment (yellow) after the 20th epoch continued to increase slightly, from 70% to 74% approximately in the 100th epoch.

In Figure 5.78, with the average training and validation loss of the two experiments, we can see that experiment B_2D_S_N1 shows signs of overfitting since its validation loss increases a bit after the 20th epoch, while the training loss continued to decrease rapidly towards 0. Therefore, our initial assumption in Section 5.5, that the dataset B_2D_S could reduce the overfitting issues, most probably was wrong. Not only that, it increases the overfitting issues since the dataset is even smaller than the B_2D_M dataset.

5.5.1.2 Experiments B_2D_S_N {1 – 3}: C = 1, GNsize = 5, Dropout, Different number of Feature Maps

With the B_2D_S dataset (Section 3.2.3.2), the three experiments B_2D_S_N {1 – 3} have been performed, to examine whether a narrower or wider network configuration could help to reduce overfitting in the network that was usually used for most of the experiments (2D_CNN_4L_1).

Experiments B_2D_S_N {1 – 3}: C = 1, GNsize = 5, Dropout, Different Feature Maps

					Training			Validation					Testing				
Experiment Name	Dropout	L2 Regularization	Feature Maps	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
B_2D_S_N1	0.3	-	f1, f2 = 32 f3 = 64	2D_CNN_4L_1	0.85	0.86	0.84	0.75	0.75	0.77	0.77	0.74	0.68	0.68	0.68	22	0.87
B_2D_S_N2	0.5	0.01	f1 = 8, f2 = 16 f3 = 32	2D_CNN_4L_5	0.90	0.89	0.90	0.73	0.75	0.74	0.74	0.73	0.72	0.72	0.71	41	0.84
B_2D_S_N3	0.5	-	f1 = 32, f2 = 64 f3 = 128	2D_CNN_4L_6	0.85	0.85	0.86	0.77	0.79	0.76	0.75	0.78	0.70	0.62	0.78	23	0.84

Table 5.15. Performance metrics & hyperparameters of the experiments B_2D_S_N {1 – 3}. Different Feature Maps.

The B_2D_S_N2 experiment uses a narrower network, the 2D_CNN_4L_5 (Section 4.3.2.3.5) than the one in the 2D_CNN_4L_1 that was used in the experiment B_2D_S_N1 (Section 4.3.2.3.1). The B_2D_S_N3 on the other hand, uses the 2D_CNN_4L_6, a wider network configuration (Section 4.3.2.3.6) than the

2D_CNN_4L_1. The terms narrow and wide will determine the number of feature maps in each Convolutional Layer. The basic source code, that the three network configurations 2D_CNN_4L_ {1, 5, 6} use is the same, and can be found in Appendix B.5.1, in the Code Snippet B.5. In that source code, the filter sizes have to change accordingly based on the values of Table 5.16.

Experiment	Feature Maps		
	f1	***f2***	***f3***
B_2D_S_N1 *(Baseline)*	32	32	64
B_2D_S_N2 *(Narrow Network)*	8	16	32
B_2D_S_N3 *(Wide Network)*	32	64	128

Table 5.16. Experiments B_2D_S_N {1 - 3} – Feature Maps fi in the ith Convolutional Layer

Figure 5.79, shows the average training and validation accuracies of the 10-folds, of the three experiments. The average validation accuracy of experiment B_2D_S_N2 with the narrow network (purple) is much worse than the rest of the experiments. This most probably indicates that not enough filters exist in the network to understand the data and create the appropriate feature maps needed for the AD/NC classification. The wider network of the experiment B_2D_S_N3 on the other hand does not seem to affect that much the performance of the model, since its validation accuracy (yellow) is very similar to B_2D_S_N1's (red). Consequently, there is no need for the extra complexity, which increases dramatically the training time, since it does not provide any significant improvement. In terms of average validation accuracies, the accuracy of the experiment with the B_2D_S_N3 wider network is 77%, much better than the 75% of the baseline experiment B_2D_S_N1 (Table 5.15).

The average training and validation losses per fold, for the three experiments, are available in Figure 5.80. The average validation loss per epoch of the experiment B_2D_S_N2 with the narrow network (purple) is worse than the other two. All three network configurations, overfit slightly since, their validation loss increases over time, while their training loss decreases rapidly. I was expecting some overfitting from the wider network since it can memorize more easily the training dataset and then find it difficult to generalize. Contrariwise, the wider network has the best average validation accuracy among the three experiments, equal to 77% (Table 5.15). Since all three

experiments overfit, the issue is not the network, but the dataset itself. The B_2D_S dataset is too small since only a single slice of a single scan per patient was used and no data augmentation was applied (Section 2.3.13.5).

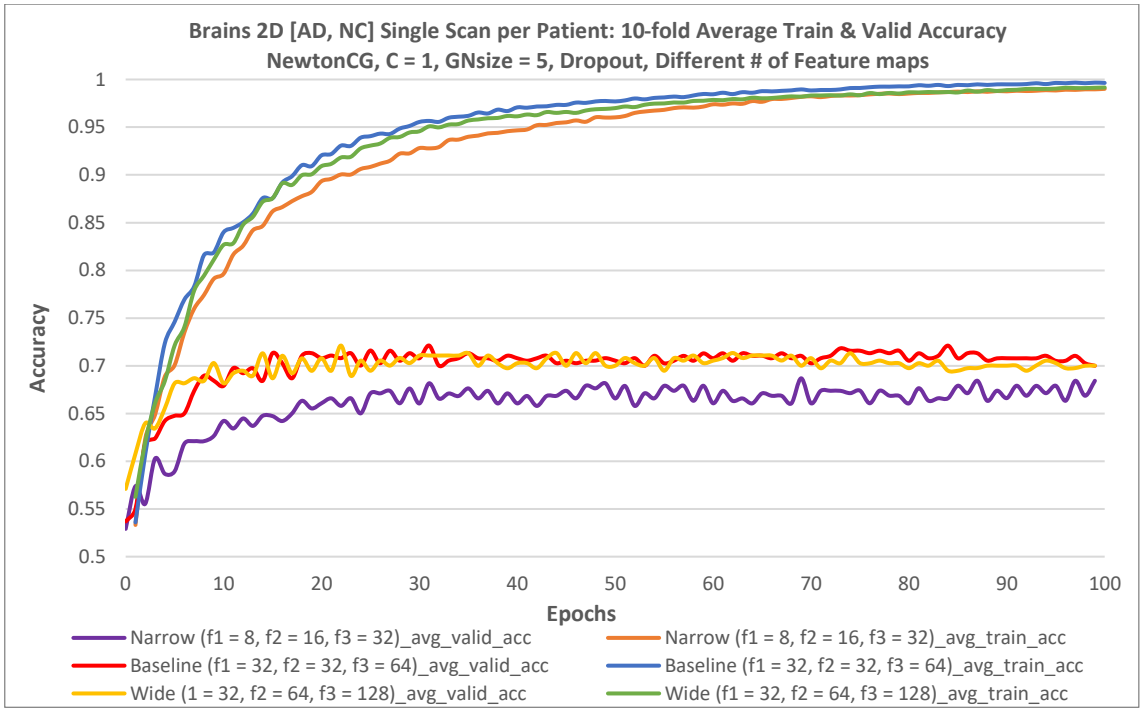


Figure 5.79. Average training and validation accuracies of the experiments B_2D_S_N {1 – 3}. Narrow, Baseline, or Wide network.

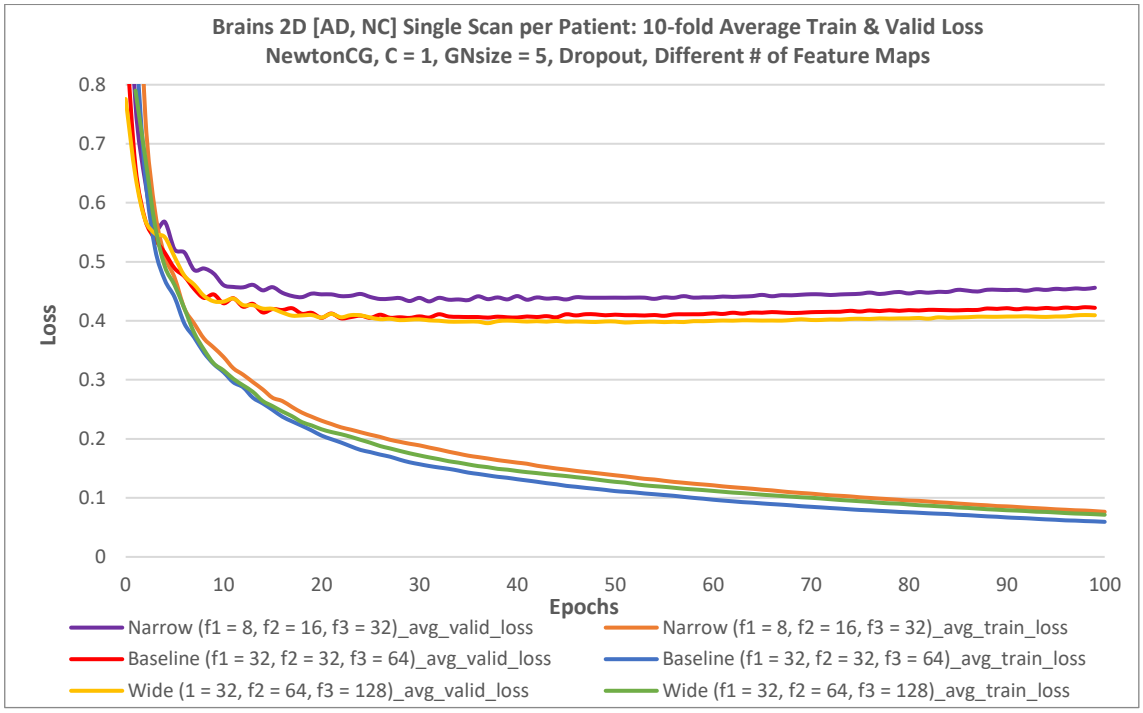


Figure 5.80. Average training and validation losses of the experiments B_2D_S_N {1 – 3} Narrow, Baseline, or Wide network.

5.6 2D Brain Slices [AD, NC]: Single Scan per Patient, 5 Slices per Scan (174 × 174)

In Section 5.6, we are going to examine the performance of a single experiment that uses the 2D Brain Slices [AD, NC], Single Scan per Patient – 5 Slices per Scan (B_2D_5S) dataset (Section 3.2.3.3). The purpose of this experiment was to test whether the dataset B_2D_5S, which contains multiple slices of the same MRI scan, could help the training model to generalize better and encounter overfitting. More specifically, in Section 5.6.2, we are going to compare the performance of different datasets with the same network configuration and hyperparameters, to validate our assumptions.

The 3D T1-weighted MRI scans of the patients were not aligned perfectly, therefore, the i^{th} slice of the j^{th} patient, that we use in the B_2D_S (Section 3.2.3.2) and B_2D_M (Section 3.2.3.4) datasets, might be slightly off than the i^{th} slice of the $(j + 1)^{th}$ patient. Therefore, using 5 different slices of the same patient in the B_2D_5S dataset could potentially help to encounter the problem of translations between slices of different patients. Consequently, it increases 5 times the size of the B_2D_S dataset, which could whether these assumptions were valid.

5.6.1 CNN 4 Layers with NewtonCG

5.6.1.1 Experiments B_2D_5S_N1 & B_2D_M_N10 & B_2D_7M_N1: C = 0.01, GNsize = 50, Dropout

We are going to compare the performance of three different datasets; the B_2D_5S, B_2D_M, and B_2D_7M, in the same network configuration and hyperparameters. The three experiments that are going to be compared are the B_2D_5S_N1 (Appendix K.1.1), B_2D_M_N10 (Appendix H.2.10), and B_2D_7M_N1 (Appendix J.2.1).

All three experiments use the NewtonCG optimizer in CNNs (Section 4.3.1) with the 2D_CNN_4L_1 network configuration (Section 4.3.2.3.1). The weight decay was set to C = 0.01 (Section 2.3.13.8) and the sizes of the subsampled Gauss-Newton matrix for approximating the Hessian Matrix to GNsize = 50 (Section 2.4.9.2). Also, Dropout (Section 2.3.13.9) was applied to all three Convolutional Layers equal to d1 = 30%, d2 = d3 = 50%, with di being the probability of a drop in the i^{th} Convolutional Layer.

Experiments B_2D_5S_N1, B_2D_M_N10, and B_2D_7M_N1 C = 0.01, GNsize = 50, Dropout - Different Datasets													
	Training			Validation					Testing				
Experiment Name	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
B_2D_M_N10	0.84	0.84	0.83	0.80	0.82	0.80	0.79	0.82	0.61	0.67	0.54	20	0.91
B_2D_7M_N1	0.94	0.94	0.94	0.76	0.75	0.77	0.77	0.75	0.64	0.69	0.58	32	0.78
B_2D_5S_N1	0.92	0.90	0.95	0.75	0.77	0.74	0.73	0.77	0.69	0.65	0.74	31	0.78

Table 5.17. Performance metrics of the experiments B_2D_5S_N1, B_2D_M_N10, and B_2D_7M_N1 Different Datasets: B_2D_5S vs. B_2D_M vs. B_2D_7M.

The training set of the B_2D_5S dataset has 1520 samples (Section 3.2.3.3), while the B_2D_M has 1044 (Section 3.2.3.4). A 50% increase in the size of the dataset was expected to improve the validation accuracy as well, but in our case, it did not; since, for the B_2D_5S_N1 experiment, the average validation accuracy is 74%, while for the B_2D_M_N10 is 80% (Table 5.17). The training set of the B_2D_7M dataset has 6496 samples (Section 3.2.3.5), approximately 600% larger than the B_2D_M. The average validation accuracy of the B_2D_7M_N1 experiment is equal to 76%.

The datasets B_2D_5S, and B_2D_7M have larger training accuracies though; 92% for the B_2D_5S_N1 experiment, and 94% for the B_2D_7M_N1, while it is 84% for the B_2D_M_N10. This indicates that most probably the model suffers from overfitting since the training accuracy increases while the validation accuracy was decreased.

In Figure 5.81, the average training and validation accuracies of the three experiments are available. We can see that both the training accuracy (orange) and the validation accuracy (purple) of the experiment B_2D_7M_N1 are smoother than the other two experiments. All three experiments, after the 50th epoch approximately, start to decline.

Figure 5.82 shows the average training and validation losses of the folds, of the three experiments. The experiment B_2D_7M_N1 overfits less than the other two, since the validation losses of the experiments B_2D_M_N10 (yellow), and B_2D_5S_N1 (red), after the 30th epoch approximately start to increase, while their training loss continues to decrease. The average training (green) and validation losses (yellow) of the experiment B_2D_M_N10 fluctuate much more than the rest of the experiments.

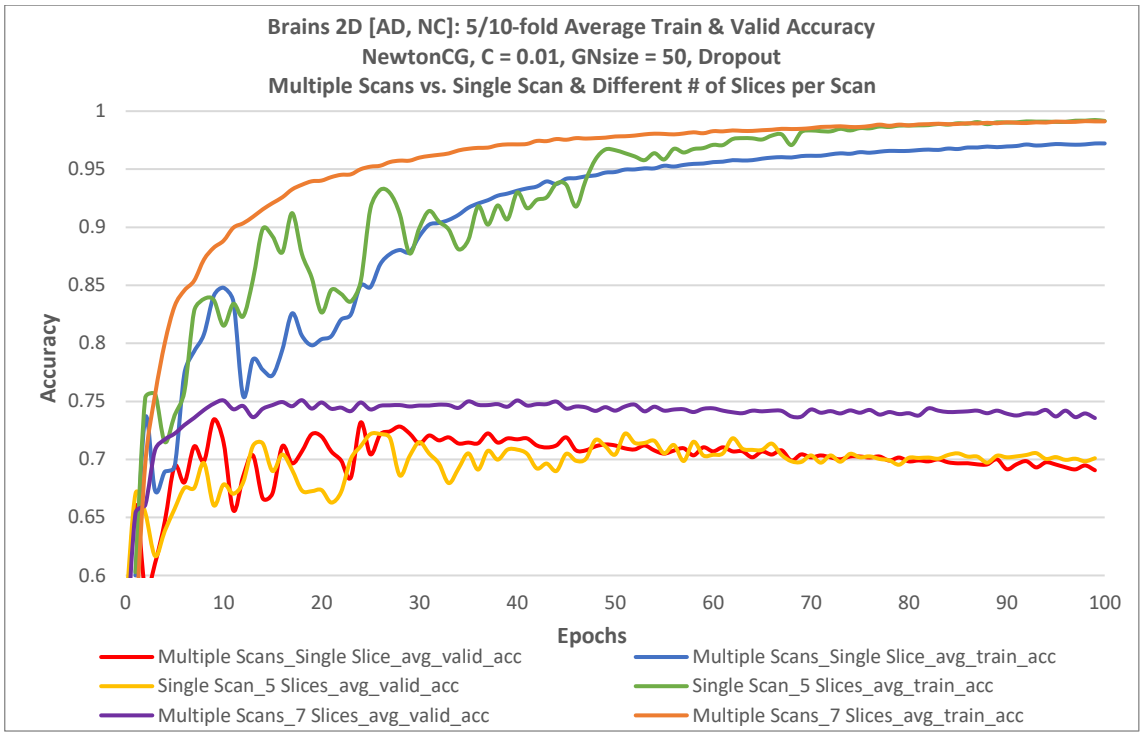


Figure 5.81. Average training and validation accuracies of the experiments B_2D_5S_N1, B_2D_7M_N1, and B_2D_M_N10.

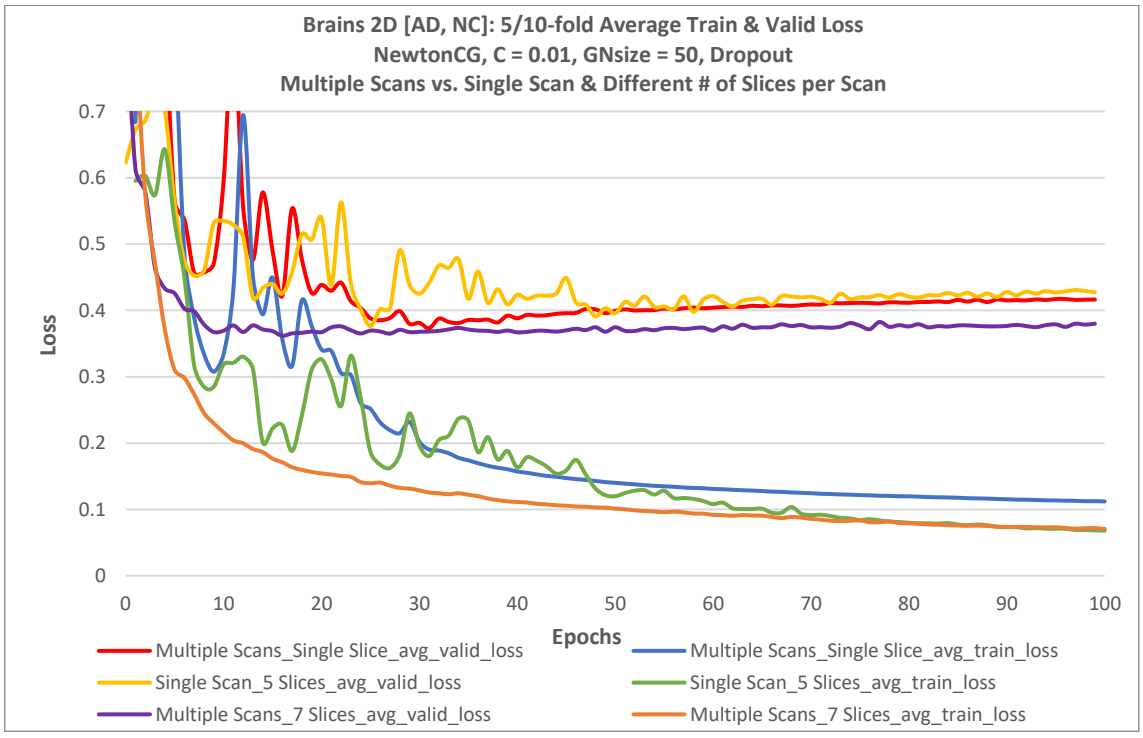


Figure 5.82. Average training and validation losses of the experiments B_2D_5S_N1, B_2D_7M_N1, and B_2D_M_N10.

If we compare the validation accuracies of each fold individually for the three experiments, we can see that they differ a lot (Figure 5.83 – 5.85). In Figure 5.83 of the

experiment B_2D_7M_N1, we can see that the validation accuracies of all 5-folds are between 0.70 and 0.76 in the 100th epoch. The accuracies of the 1st and 3rd fold decrease slightly overtime after the 40th epoch approximately, while the accuracies of the 2nd, 4th, and 5th fold keep increasing. The standard deviation between the best validation accuracies per fold of the B_2D_7M_N1 experiment is 2%; which is one of the smallest standard deviations among all the experiments with 2D Brain Slices (Table 5.7).

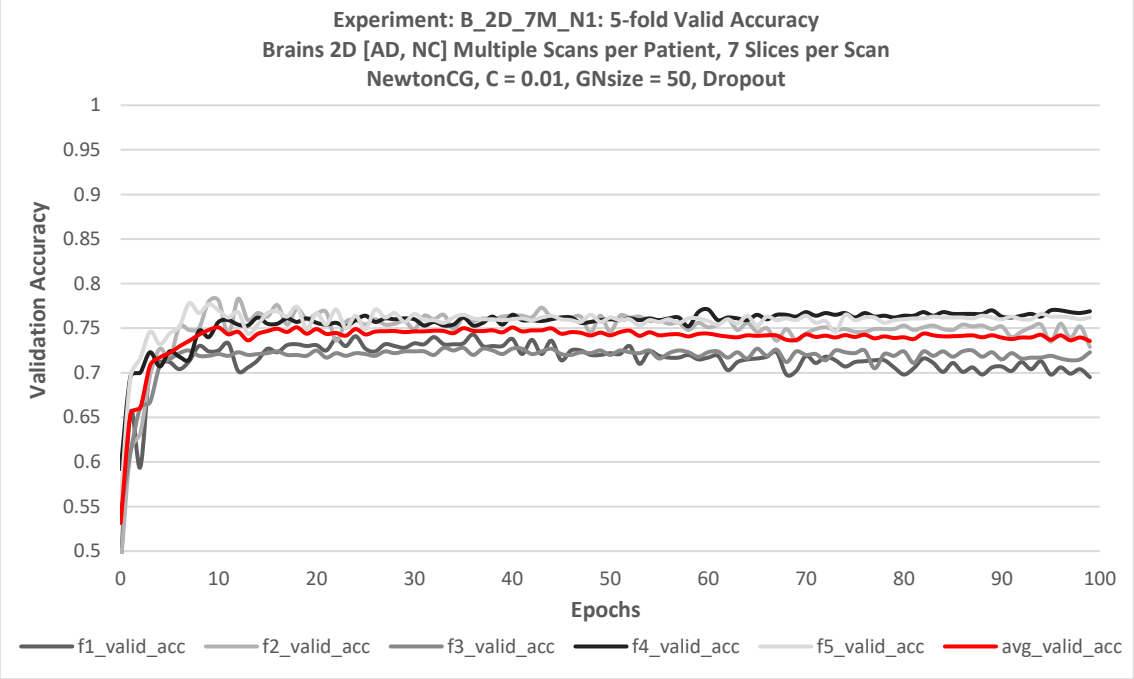


Figure 5.83. Validation accuracy per fold of the experiment B_2D_7M_N1.

The validation accuracies of each one of the 5-folds in the B_2D_5S_N1 experiment (Figure 5.84) are more spread than the ones in the B_2D_7M_N1 experiment (Figure 5.83). We observe larger fluctuations, especially the first 50 epochs of the experiment B_2D_5S_N1. The standard deviation between the best validation accuracies per fold is 3% for the B_2D_5S_N1 experiment (Table 5.7).

The validation accuracies per fold on the B_2D_M_N10 experiment (Figure 5.85), are much worse than the other experiments since they rise and fall very aggressively. The standard deviation of the best validation accuracies per fold is 5%, larger than the B_2D_5S_N1 (3%) and B_2D_7M_N1 (2%) (Table 5.7). In Figure 5.85, we can see that the validation accuracies at the last epoch, are between 0.5 and 0.85; an undesirable range for validation accuracies since it makes our model non-reliable because 0.5 means it does not learn anything, while 0.85 means it performs great.

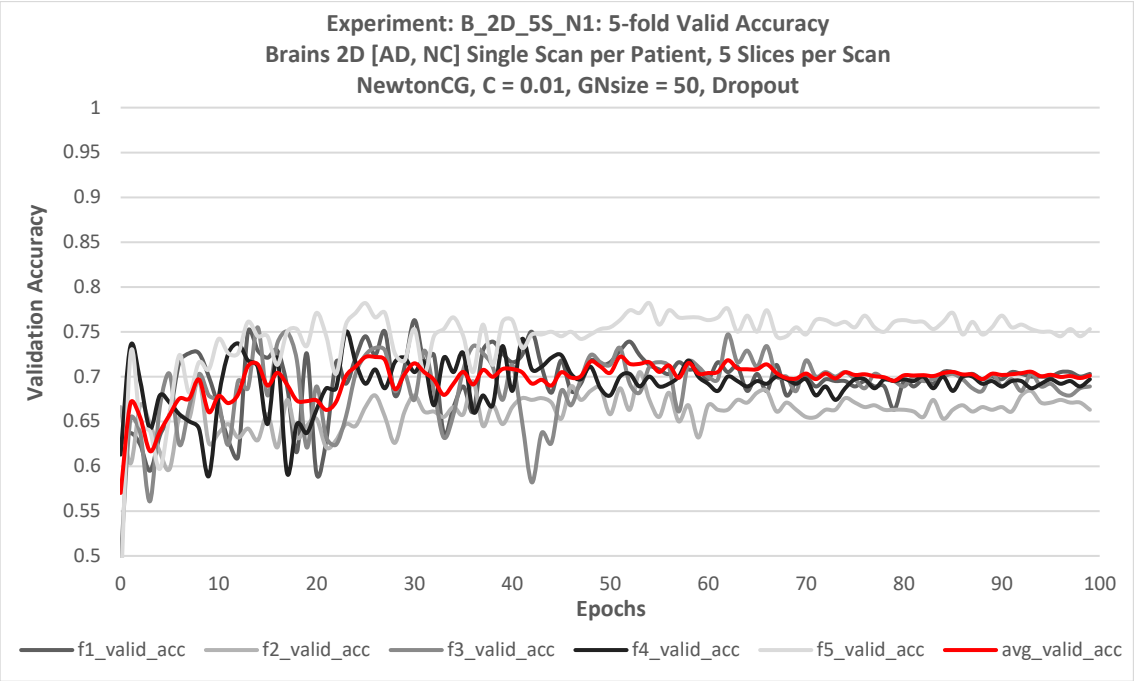


Figure 5.84. Validation accuracy per fold of the experiment B_2D_5S_N1.

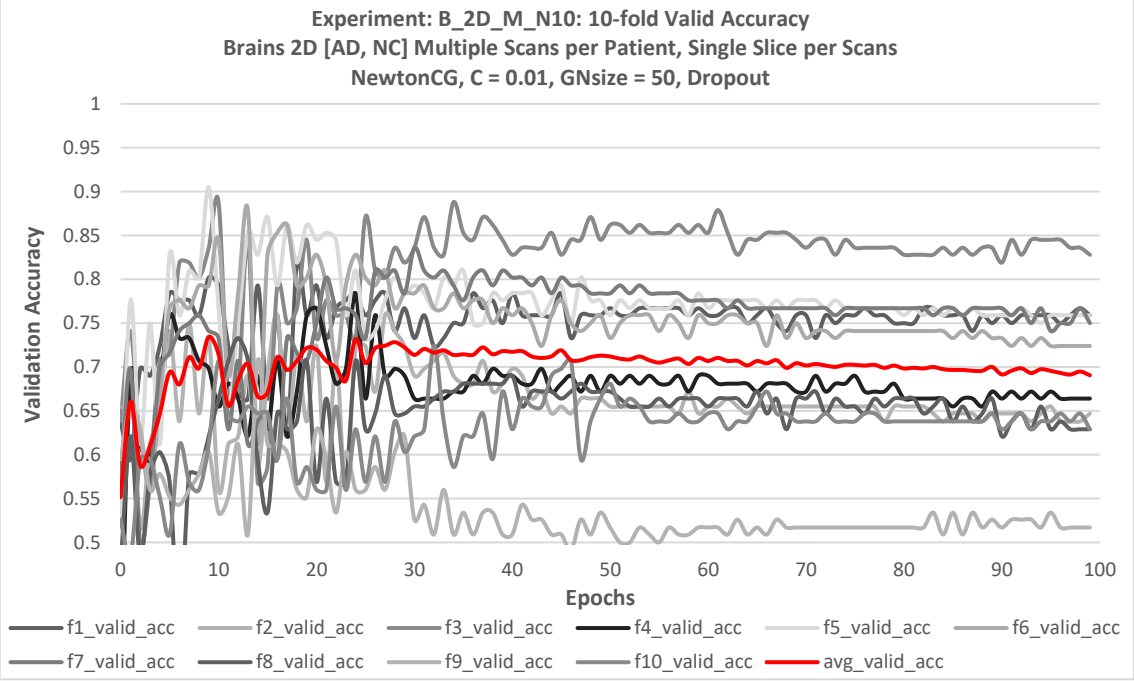


Figure 5.85. Validation accuracy per fold of the experiment B_2D_M_N10.

The network configuration, and the hyperparameters, have a crucial role in the model's performance, but the dataset is even more important. Based on the aforementioned observations, a dataset to be reliable should contain much more training samples than the B_2D_5S and B_2D_M datasets. The dataset B_2D_7M may not have the best performance, but the variation between folds is significantly less. I believe that if data

augmentation (Section 2.3.13.5) is applied in the B_2D_7M dataset, then a dataset much more reliable will be created for the AD/NC problem.

It is worth mentioning that the datasets B_2D_7M and B_2D_5S use a 5-fold CV while the dataset B_2D_M a 10-fold. It is not clear whether this is the issue behind the performance decrease for the two datasets or something else (Table 5.17).

5.6.2 VGG19 with NewtonCG

In theory, a deeper CNN is expected to outperform a shallow one, in terms of validation accuracy (Krizhevsky et al. 2012). For the 2D Brain Slices [AD, NC], Single Scan per Patient – 5 Slices per Scan (B_2D_5S) dataset (Section 3.2.3.3), this is not the case. The 4-layer 2D_CNN_4L_1 (Section 4.3.2.3.1) network (3 Convolutional Layers + 1 fully connected layer) of the experiment the B_2D_5S_N1 (Appendix K.1.1), outperforms the 19-layer 2D_VGG19_1 (Section 4.3.2.6.2) network (16 Convolutional Layers + 3 fully connected layers). The poor performance of the 2D_VGG19_1 network cannot be explained by overfitting, since both the average training and validation accuracies were equally terrible, approximately 50% after the first 2 – 3 epochs.

A possible explanation of this behavior, as Kaiming et al. 2015 said, is that when a neural network is too deep for a given problem, it tends to try to recreate the identity function. The first portion of the network has found a set of weights, capable of optimizing the objective, so, the latter portion of the deep neural network essentially adds noise. Therefore, the latter portion of the network attempts to create an identity function from a nonlinear set of activations, which is terrible in our case. As an analogy, it is like trying to approximate a line with polynomials of a degree greater than 1; as a result, you will get a wavy mess (Kaiming et al. 2015).

5.7 2D Brain Slices [AD, NC]: Multiple Scans per Patient, 7 Slices per Scan (174 × 174)

The 2D Brain Slices [AD, NC], Multiple Scans per Patient – 7 Slices per Scan (B_2D_7M) dataset (Section 3.2.3.5), increases 7 times the size of the B_2D_M dataset (Section 3.2.3.4). The experiments in Section 5.7.1 & 5.7.2, are going to examine how the increase of the dataset's size, affects both the Adam and the NewtonCG optimizers.

Experiments with the B_2D_7M dataset																						
								Training			Validation						Testing				
Experiment Name	Optimizer	C	GNsize	Learning Rate	Dropout	Feature Maps	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
Dataset: B_2D_M																					
B_2D_M_A3	Adam	0.01	-	0.001	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	2D_CNN_4L_1	0.91	0.90	0.92	0.79	0.08	0.78	0.81	0.82	0.77	0.64	0.75	0.53	154	0.91
B_2D_M_N10	NewtonCG	0.01	50	-	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	2D_CNN_4L_1	0.84	0.84	0.83	0.80	0.05	0.82	0.80	0.79	0.82	0.61	0.67	0.54	20	0.91
Dataset: B_2D_7M																					
B_2D_7M_A1	Adam	0.01	-	0.001	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	2D_CNN_4L_1	0.92	0.93	0.91	0.75	0.02	0.74	0.75	0.76	0.74	0.64	0.67	0.60	49	0.78
B_2D_7M_N1	NewtonCG	0.01	50	-	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	2D_CNN_4L_1	0.94	0.94	0.94	0.76	0.02	0.75	0.77	0.77	0.75	0.64	0.69	0.58	32	0.78
B_2D_7M_N2	NewtonCG	0.01	50	-	d1 = 0.3 d2 = 0.5	f1 = 64, f2 = 128	2D_CNN_3L_1	0.94	0.95	0.93	0.73	0.01	0.73	0.74	0.74	0.72	0.66	0.68	0.64	34	0.75
B_2D_7M_N3	NewtonCG	0.01	50	-	d1 = 0.3 d2 = 0.5	f1 = 16 f2 = 32	2D_CNN_3L_2	0.96	0.97	0.96	0.73	0.02	0.72	0.74	0.75	0.71	0.66	0.69	0.63	40	0.76

Table 5.18. Performance metrics and hyperparameters of the experiments with the B_2D_7M dataset.

Table 5.18, shows the performance metrics of the best experiments (largest average validation accuracies) with the B_2D_M dataset, for both the Adam and NewtonCG optimizers; and all the experiments with the B_2D_7M dataset. As we can see, all the experiments with the B_2D_7M dataset, both for the Adam and NewtonCG optimizer, have better training accuracy but worse validation accuracy than the experiments with the B_2D_M dataset (Table 5.18). Additionally, the experiments with the B_2D_7M dataset, have the smallest average standard deviation between the best validation accuracies per fold, among all the different 2D Brain Slices [AD, NC] experiments (Table 5.7).

5.7.1 CNN 4 Layers with Adam

5.7.1.1 Experiments B_2D_7M_A1, B_2D_M_A3: C = 0.01, lr = 0.001, Dropout

The experiments B_2D_7M_A1 (Appendix J.1.1) and B_2D_M_A3 (Appendix H.1.3), are meant to compare how the performance of the B_2D_7M dataset (Section 3.2.3.5) differs from the B_2D_M dataset (Section 3.2.3.4), given the Adam optimizer (Section 2.4.4). The network configuration 2D_CNN_4L_1 (Section 4.3.2.3.1) is the same for both experiments, as well as the C = 0.01 and learning rate = 0.001.

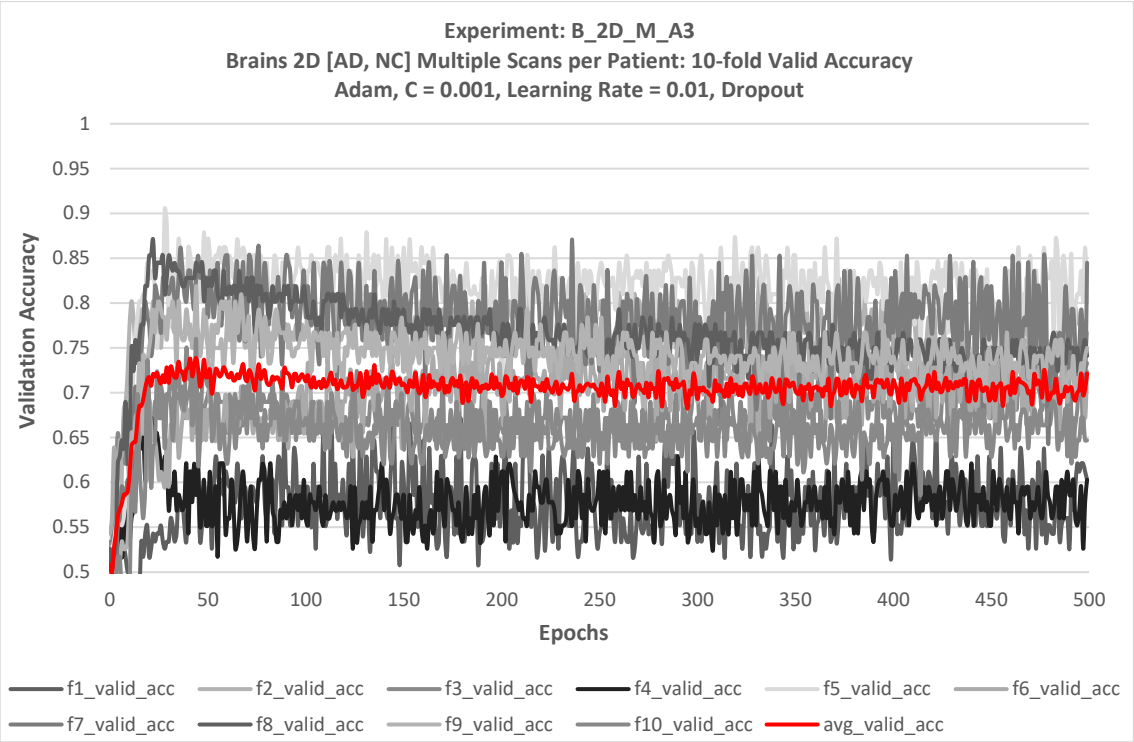


Figure 5.86. Validation accuracy per fold of the experiment B_2D_M_A3.

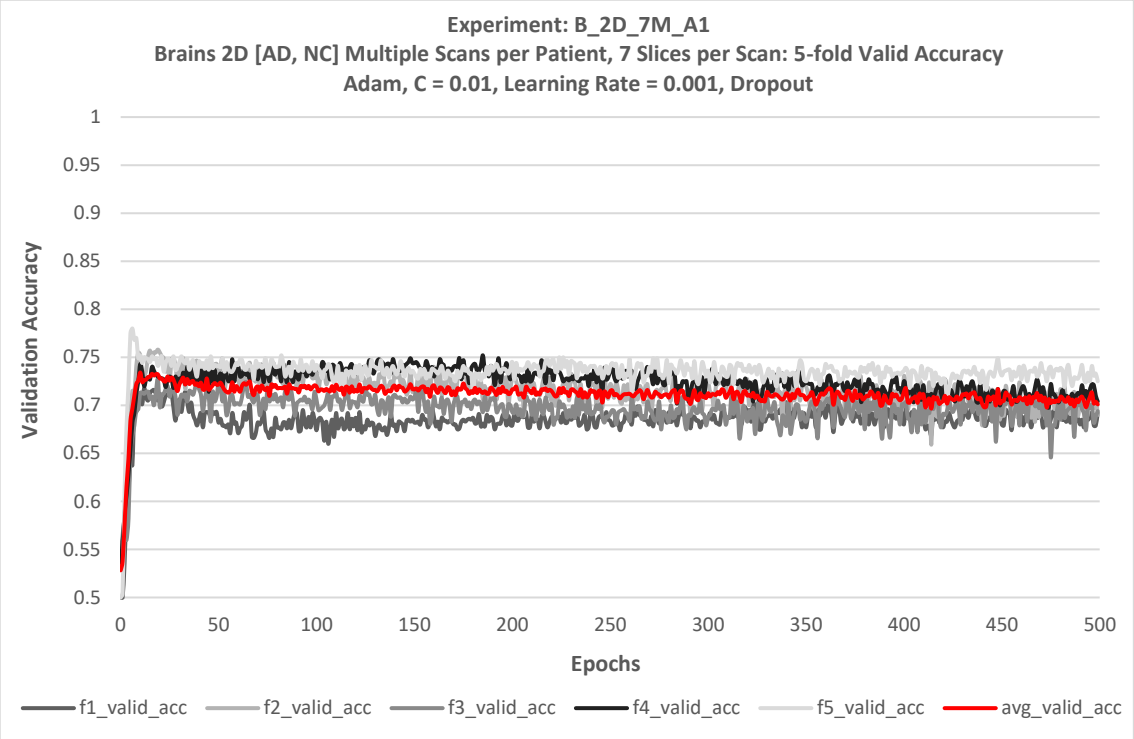


Figure 5.87. Validation accuracy per fold of the experiment B_2D_7M_A1.

In Figure 5.86, the validation accuracies per fold of the experiment B_2D_M_A3 fluctuates a lot. Also, the validation accuracies during training range between 0.5 and 0.9.

On the other hand, in Figure 5.87, the validation accuracies perf fold of the experiment B_2D_7M_A1, range between 0.65 and 0.75 approximately. This certainly is a crucial advantage of the B_2D_7M dataset, over the B_2D_M. For example, in the case of the B_2D_M dataset (Figure 5.86), the average validation accuracy of the 4th fold was 57%; while for the 5th fold was 83%. Consequently, our model with the B_2D_M dataset is unreliable, since sometimes it does not learn anything, while other times is a great learner.

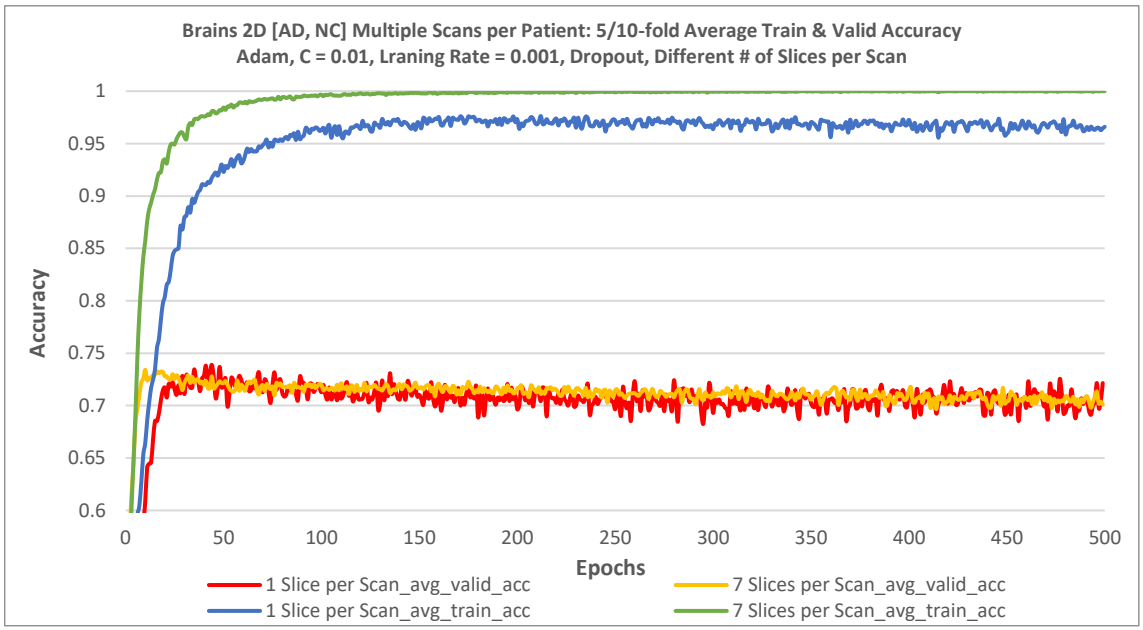


Figure 5.88. Average training and validation accuracies of Adam's experiments B_2D_M_A3 & B_2D_7M_A1. 1 slice vs. 7 slices per scan.

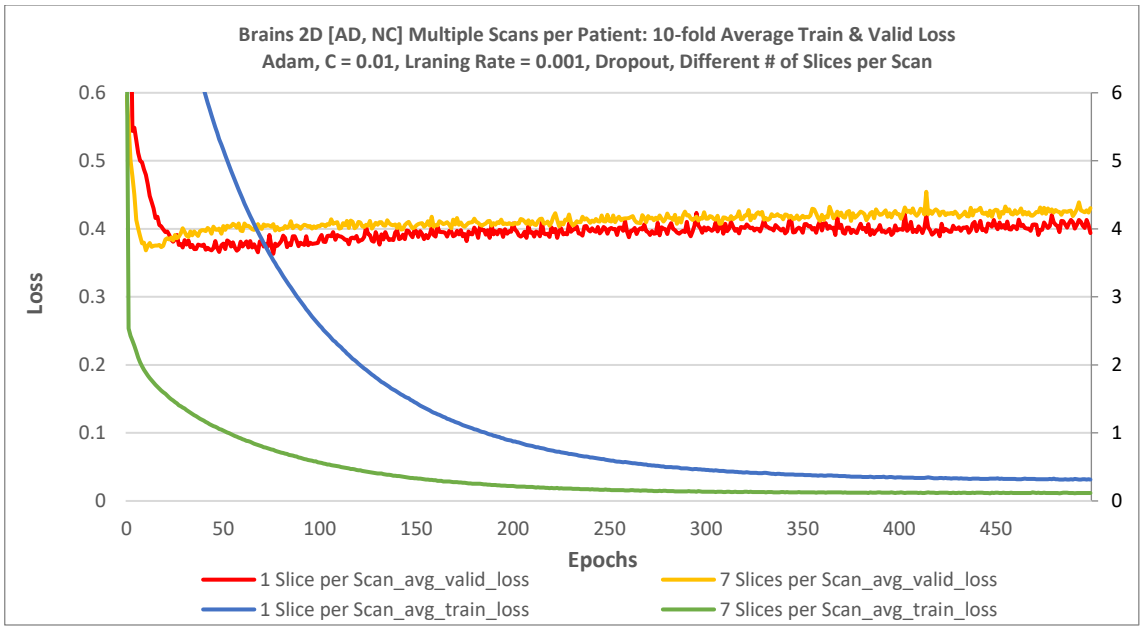


Figure 5.89. Average training and validation losses of Adam's experiments B_2D_M_A3 & B_2D_7M_A1. 1 slice vs. 7 slices per scan.

In Figure 5.88, we can see the average training and validation accuracies per fold for the two experiments. The validation accuracy of the experiment B_2D_7M_A1 (yellow), reaches fast its highest value and then decreases slightly over time. For the B_2D_M_A3 (red), it takes more epochs to reach its maximum value, but then it has similar behavior to the other experiment. The average validation accuracy of the experiment with the B_2D_M dataset, has more fluctuations, while for the B_2D_7M is smoother. This can be validated also by Table 5.18, since the average standard deviation of the best validation accuracies per fold, for the experiments B_2D_M_A3, is 8%, while for the experiment B_2D_7M_A1 is 2%.

In Figure 5.89, with the average training and validation losses per fold of the two experiments, we observe that the B_2D_M_A3 (red) overfits a bit less than the B_2D_7M_A1 (yellow). The average validation loss of the B_2D_M_A3 (red) increases less than the other experiment's (yellow) over time. Overfitting exists in both experiments since their average training loss continues to decrease, while their validation loss starts increasing at some point.

5.7.2 CNN 3 & 4 Layers with NewtonCG

5.7.2.1 Experiments B_2D_7M_N {1 – 3}: C = 0.01, GNsize = 50, Dropout

The experiment B_2D_7M_N1 did not perform great, as we saw in Section 5.7.1.1, with its average validation accuracy equal to 76% (Table 5.18); therefore, the experiments B_2D_7M {2 – 3} were held to examine whether a shallower network configuration could improve the performance. Since some overfitting was observed in the B_2D_7M_N1, because the training accuracy was high while the validation accuracy was low, a simpler network architecture could potentially decrease this issue.

The B_2D_7M_N1 experiment used the 4-layer network architecture 2D_CNN_4L_1 (Section 4.3.2.3.1), with 3 Convolutional Layers and a single dense layer. The number of feature maps fi in the i^{th} Convolutional Layer are f1 = f2 = 32, and f3 = 64. The experiments B_2D_7M_N {2 – 3} on the other hand, use two 3-layer architectures (2 Convolutional Layers + 1 dense layer). The 2D_CNN_3L_1 network (Section 4.3.2.2.1) that was used in the experiment B_2D_7M_N has f1 = 64 and f2 = 128 filters, and its source code is available in Appendix B.5.10 at Code Snippet B.14; while the 2D_CNN_3L_2 network (Section 4.3.2.2.2) for the experiment B_2D_7M_N3, which

has f1 = 16 and f2 = 32 filters, is available in Appendix B.5.11 at Code Snippet B.15 (Table 5.19).

Experiment	Feature Maps		
	f1	***f2***	***f3***
B_2D_7M_N1 *(Baseline)*	32	32	64
B_2D_7M_N2 *(Shallow & Wide Network)*	64	128	-
B_2D_7M_N3 *(Shallow & Narrow Network)*	16	32	-

Table 5.19. Experiments B_2D_7M_N {1 – 3} – Feature Maps fi in the i_{th} Convolutional Layer

Based on Table 5.20, with the performance metrics of the three experiments of the B_2D_7M dataset, the experiment with the 4-layer CNN (B_2D_7M_N1), has a better validation accuracy than the 3-layer CNNs (B_2D_7M_N {2 – 3}). More specifically, the average validation accuracy of the B_2D_7M_N1 experiment is 76% while for the experiments B_2D_7M_N {2 – 3} is on both of them 73%. The training accuracies are very similar for all three experiments between 94% and 96%. This indicates that potentially all three models overfit since their training accuracies are high, but their validation accuracies not. This indicates that maybe even the 4-layer CNN, is too simple for our AD/MC problem with the larger B_2D_7M dataset. Maybe a 5-layer or even deeper network configuration should be used to examine whether the poor performance is caused by the simplicity of the network.

Experiments B_2D_7M_N {1 – 3}: C = 0.01, GNsize = 50																
				Training			Validation					Testing				
Experiment Name	Dropout	Feature Maps	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
B_2D_7M_N1	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	2D_CNN_4L_1	0.94	0.94	0.94	0.76	0.75	0.77	0.77	0.75	0.64	0.69	0.58	32	0.78
B_2D_7M_N2	d1 = 0.3 d2 = 0.5	f1 = 64, f2 = 128	2D_CNN_3L_1	0.94	0.95	0.93	0.73	0.73	0.74	0.74	0.72	0.66	0.68	0.64	34	0.75
B_2D_7M_N3	d1 = 0.3 d2 = 0.5	f1 = 16 f2 = 32	2D_CNN_3L_2	0.96	0.97	0.96	0.73	0.72	0.74	0.75	0.71	0.66	0.69	0.63	40	0.76

Table 5.20. Performance metrics and hyperparameters of the experiments B_2D_7M_N {1 – 3}

In Figure 5.90 with the average training and validation accuracies of the 5-folds, for the three experiments, we can see that the average validation accuracy of the experiment B_2D_7M_N1 (purple), in general, is greater than the rest (yellow, red); while the training accuracies of all three experiments behave very similarly.

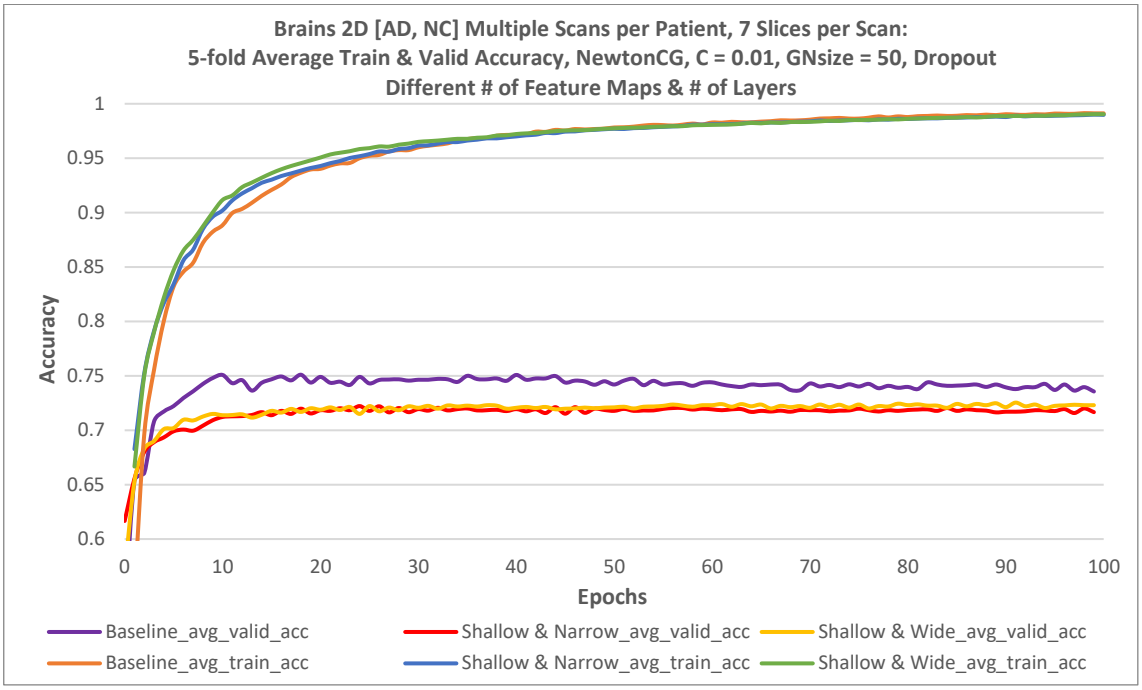


Figure 5.90. Average training and validation accuracies of the experiments B_2D_7M_N {1 – 3}.

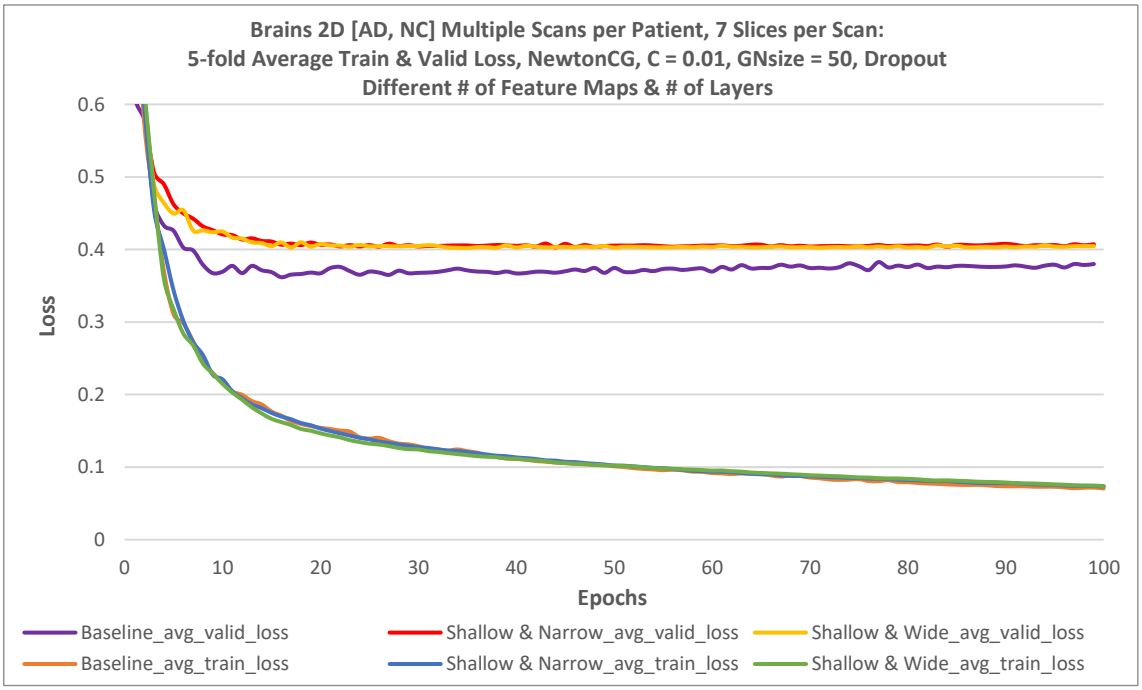


Figure 5.91. Average training and validation losses of the experiments B_2D_7M_N {1 – 3}.

The average validation loss of the experiment B_2D_7M_N1 (purple) is less than the ones in the experiments B_2D_7M_N {2 – 3}, which means it performs better. We can see that the purple line increases over time while the training loss continuously decreases, and that indicates overfitting (Figure 5.91).

5.8 3D Left Hippocampus [AD, NC]: Single Scan per Patient (37 × 32 × 50)

The experiments with the 3D Left Hippocampus (LH_3D) dataset (Section 3.2.6) were expected to perform better than any other experiment with different datasets (Section 3.2.3 – 3.2.5, 3.2.7) in CNNs. The reason is that the hippocampus, especially the left one faces the most damage in an AD patient, therefore the difference between an NC patient is very significant (Leandrou et al. 2018; Leandrou et al. 2020).

The best experiment with the Adam optimizer (LH_3D_S_A4) has an average validation accuracy equal to 77%, while the best experiment with the NewtonCG optimizer (LH_3D_S_N2) is 73% (Table 5.21). In comparison with the 2D Brain Slices datasets (B_2D), where the average validation accuracy of the best model is 81% (Section 5.3.1), the results of the LH_3D are mediocre.

3D Left Hippocampus – Single Scan per Patient [AD, NC]

							Training			Validation					Testing				
Experiment Name	Learning Rate	C	GNsize	Epochs	Dropout	Max-Pooling	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
Adam																			
LH_3D_S_A1	0.001	0.01	-	500	Yes	Yes	0.66	0.78	0.54	0.72	0.69	0.79	0.81	0.62	0.65	0.84	0.46	37	0.75
LH_3D_S_A2	0.0001	0.01	-	500	Yes	Yes	0.73	0.83	0.63	0.76	0.73	0.84	0.84	0.68	0.69	0.87	0.51	117	0.81
LH_3D_S_A3	0.00001	0.01	-	500	Yes	Yes	0.73	0.69	0.76	0.72	0.74	0.70	0.68	0.75	0.62	0.68	0.56	208	0.75
LH_3D_S_A4	0.0001	0.01	-	500	Yes	-	1.00	1.00	1.00	0.77	0.73	0.84	0.86	0.67	0.67	0.73	0.60	209	0.83
NewtonCG																			
LH_3D_S_N1	-	0.01	500	100	-	Yes	0.73	0.79	0.68	0.70	0.70	0.76	0.77	0.64	0.67	0.80	0.53	4	0.73
LH_3D_S_N2	-	0.01	50	100	Yes	-	0.86	0.87	0.84	0.73	0.72	0.76	0.78	0.68	0.69	0.74	0.63	21	0.83

Table 5.21. Performance metrics of the experiments with 3D Left Hippocampus – Single Scan per Patient [AD, NC].

The experiments with the NewtonCG optimizer perform worse than the ones with the Adam optimizer, which was something unexpected (Table 5.21). Therefore, to identify the reason behind this, not only the weight decay (C) and GNsize were modified during the experiments with the LH_3D dataset, but also the eta (η), xi (ξ), CGmax, boost, and drop (Section 4.3.1). The only hyperparameter that was not modified in the NewtonCG method, is the initial value of lambda (λ) since it updates itself based on the results of the approximation in the Gauss-Newton update (Wand et al. 2020).

The modification of these hyperparameters did not help to improve the performance of the model. For example, reducing the CGmax iterations from 250 to 25, helped to run the experiment in Arcadia, since each epoch took less time, but the validation accuracy after 2 – 3 epochs was stuck to 50%. Reducing the value of eta from 0.0001 to 0.00001 showed some improvement but only an incomplete fold was executed to identify its benefits. Additionally, reducing the GNsize to 5 from 50, showed some potential, but again no complete 5-fold experiment was executed. The reason was that every 3D experiment was incapable to be run in the Arcadia server and therefore had to be executed in The Cyprus Institute, where we had the issue of lack of resources.

Since the performance was not changing that much by modifying the network's hyperparameters, the network configuration was the next thing that should change. More specifically, four configurations are being compared to check the effect of Max-Pooling on the 3D Left Hippocampus dataset. The assumption was that the Max-Pooling may be destroying meaningful information in the data. Consequently, the four network configurations are the network 3D_CNN_4L_2 without Max-Pooling (Section 4.3.4.2.4), the network 3D_CNN_4L_3 with Max-Pooling in the 2nd Convolutional Layer only (Section 4.3.4.2.5), and the network 3D_CNN_4L_1 with Max-Pooling to all three Convolutional Layers (Section 4.3.4.2.3). The source code for the three networks 3D_CNN_4L_ {1 – 3} can be found respectively in Appendix B.5.2, B.5.12, and B.5.13; at Code Snippets B.6, B.16, and B.17.

As we can see in Figure 5.92, the network 3D_CNN_4L_1 with Max-Pooling to all three layers (blue) got stuck in a validation accuracy equal to 0.635, with no signs of improvement. The network 3D_CNN_4L_3 with Max-Pooling in the 2nd layer (yellow), had a great start, but after 40 epochs started overfitting as its validation accuracy started decreasing. The network 3D_CNN_4L_2 without Max-Pooling (gray), kept improving

during the whole training process. Therefore, the network 3D_CNN_4L_2 without Max-Pooling at any Convolutional Layer performed better than the rest of them.

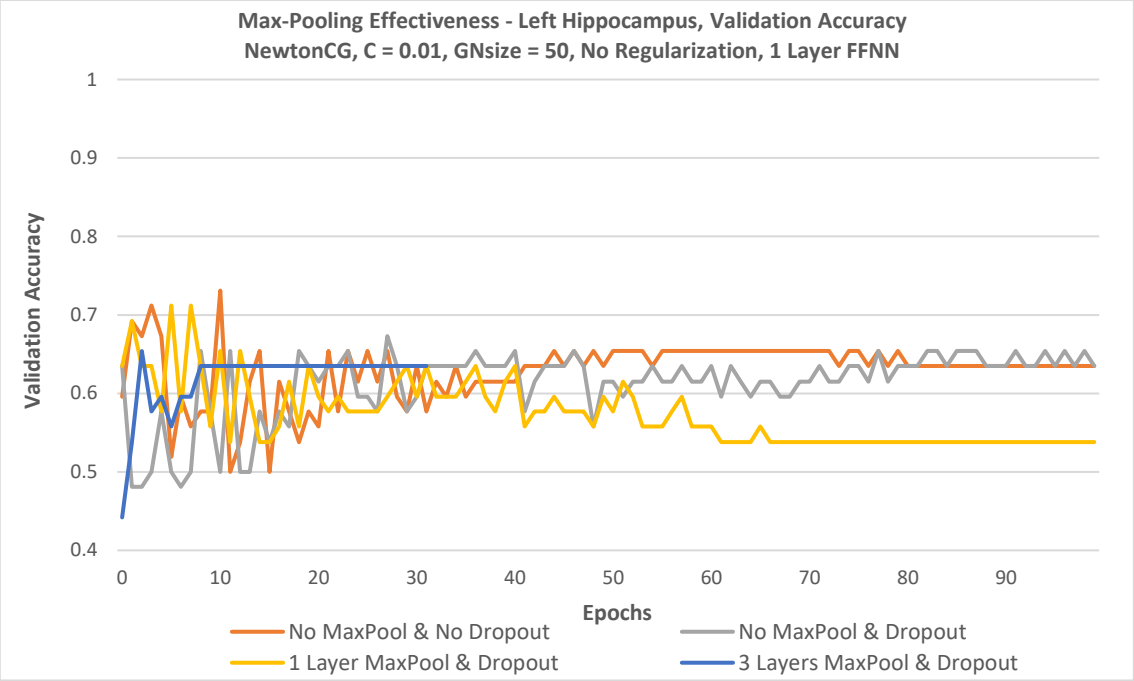


Figure 5.92. Validation accuracies of the 1st fold of the 3D Left Hippocampus dataset. Comparing the effect of Max-Pooling.

The experiment LH_3D_S_A4 with the Adam optimizer, as well as the experiment LH_3D_S_N2 with the NewtonCG, unlike the rest of the experiment, do not apply Max-Pooling to any Convolutional Layer and they are the two experiments with the highest validation accuracies in their group of experiments of the same optimizer (Table 5.21). The absence of Max-Pooling, clearly affects the training accuracy as well, which is much higher than the experiments with Max-Pooling.

5.8.1 CNN 4 Layer with Adam

5.8.1.1 Experiments LH_3D_S_A {1 – 3}: C = 0.01, lr = {0.001, 0.0001, 0.00001}, Dropout

The experiments LH_3D_S_A {1 – 3} compare the effect of the learning rate on the network's performance with the LH_3D dataset (Section 3.2.6). All the experiments were held for 500 epochs, with C = 0.01 (Section 2.3.13.8) and the Adam optimizer (Section 2.4.4). Also, the Dropout d1 = d2 = 32, d3 = 64 was applied, on each Convolutional Layer, with di being the dropout probability in the i^{th} layer. Additionally, the 3D_CNN_4L_1 network configuration was used in all three experiments (Section 4.3.4.2.3). The source code for the network configuration is available in Appendix B, in Code Snippet B.6.

Detailed plots with the training and validation accuracies and losses per fold, for the experiments LH_3D_S_A {1 – 3}, can be found in Appendix L.1.1 – L.1.3, respectively.

Figure 5.22 shows the performance metrics of the experiments LH_3D_S_A {1 – 3}. The experiment LH_3D_S_A2 with a learning rate = 0.0001 has the best average validation accuracy equal to 76% while for the others' is 72%. The sensitivity is much better in the experiment LH_3D_S_A2 with learning rate = 0.0001 (84%) than the experiment's LH_3D_S_A3 with learning rate = 0.00001 (68%). The specificity, on the other hand, is better for the LH_3D_S_A3 experiment since it is equal to 75%, while for the LH_3D_S_A2 experiment is 68%. The specificity is the metric that matters the most for us since is the value that indicates how many ADs were correctly identified as ADs.

3D Left Hippocampus – Adam, Different Learning Rates															
			Training			Validation					Testing				
Experiment Name	Learning Rate	Dropout	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
LH_3D_S_A1	0.001	d1 = 0.3 d2, d3 = 0.5	0.66	0.78	0.54	0.72	0.69	0.79	0.81	0.62	0.65	0.84	0.46	37	0.75
LH_3D_S_A2	0.0001		0.73	0.83	0.63	0.76	0.73	0.84	0.84	0.68	0.69	0.87	0.51	117	0.81
LH_3D_S_A3	0.00001		0.73	0.69	0.76	0.72	0.74	0.70	0.68	0.75	0.62	0.68	0.56	208	0.75

Table 5.22. Performance metrics of the experiments with 3D Left Hippocampus & Adam. Different Learning Rates.

In Figure 5.93, the average training and validation accuracies of the 5-folds per experiment are available. We can see the average training (dark green) and validation (light green) accuracies of the experiment LH_3D_S_A1, where the learning rate is equal to 0.001. The value of the learning rate was too small; therefore, the model is underfitting. That is why both accuracies kept decreasing until both of them reached the value of 50%; consequently, the network was unable to learn.

From the average training (dark red) and validation (light red) accuracies of the LH_3D_S_A2 experiment, with the larger learning rate = 0.0001, we can see that the underfitting problem is fixed. The average training accuracy (dark red) in the first 50 epochs rises and then begins to fall. The average validation accuracy (light red) of the experiment LH_3D_S_A2 on the other hand increases the first 30 epochs approximately and then remains stable with minor fluctuations for the rest of the training process.

Contrariwise, the accuracies of the experiment LH_3D_S_A3 with the learning rate = 0.00001, both kept increasing smoothly during the 500 epochs training process (Figure 5.93).

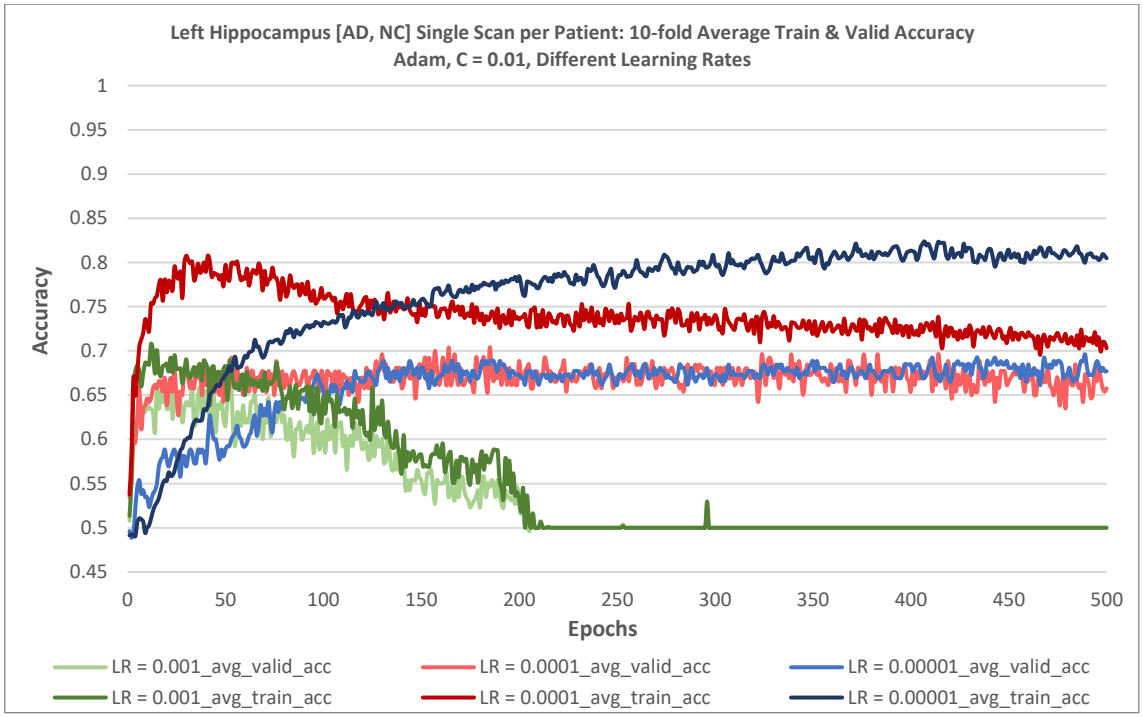


Figure 5.93. Average training and validation accuracies of the experiments LH_3D_S_A {1 – 3}.

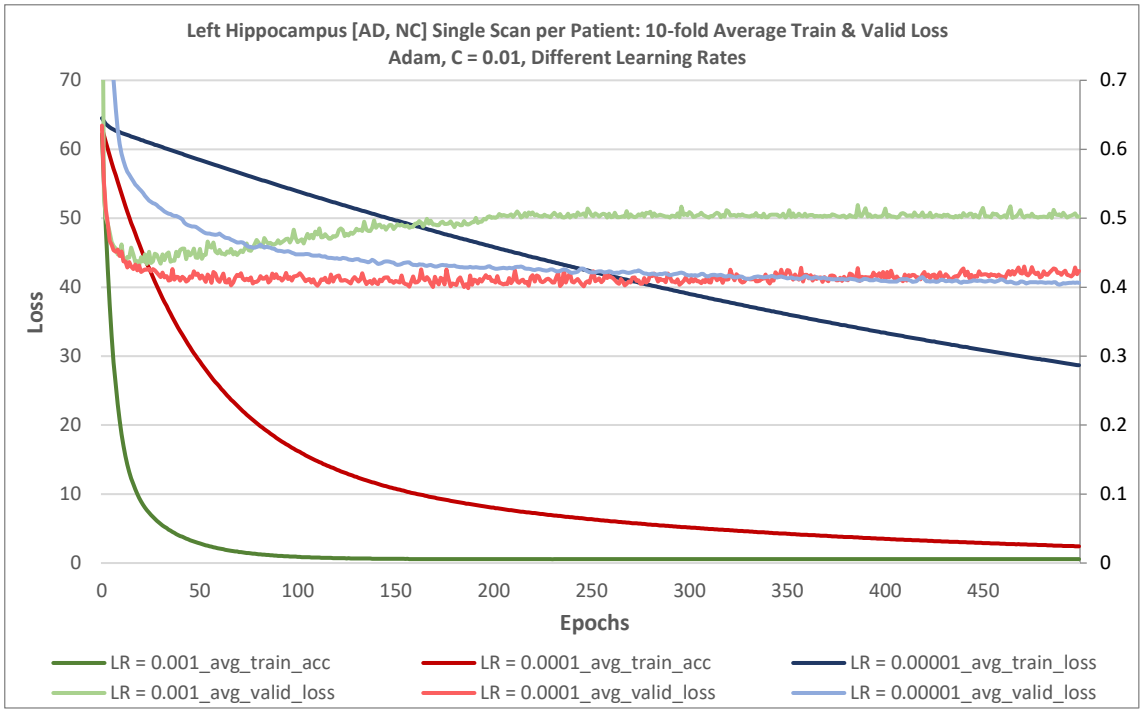


Figure 5.94. Average training and validation losses of the experiments LH_3D_S_A {1 – 3}.

From Figure 5.94, with the average training and validation losses per fold, of the experiments, we can see that the average training loss of the experiment LH_3D_S_A3, decreases very slowly. The experiment LH_3D_S_A3 though does not show any signs of overfitting, since, for the first 500 epochs, it is the only experiment that both its validation loss and training loss decrease the whole training process.

5.8.1.2 Experiments LH_3D_S_A {2, 4}: C = 0.01, lr = 0.0001, Dropout, With/Without Max-Pooling

The experiment LH_3D_S_A4, with the Adam optimizer, compares the effect of no Max-Pooling applied at any Convolutional Layer with the experiment LH_3D_S_A2 that applies Max-Pooling to all three Convolutional Layers. The experiment LH_3D_S_A4 uses the 3D_CNN_4L_2 network (Section 4.3.4.2.4), and the experiment LH_3D_S_A2 the 3D_CNN_4L_1 network (Section 4.3.4.2.3).

Based on Table 5.23, with the performance metrics of the two experiments, the average validation accuracy is a bit better in the LH_3D_S_A4 experiment, where it is equal to 77%, while in the LH_3D_S_A2 experiment is equal to 76%. Another difference between the two experiments is that for the LH_3D_S_A2, the average epoch of the best validation accuracy per fold is 117 while for the LH_3D_S_A4 is 209.

Experiments LH_3D_S_A {2, 4}: Adam - With/Without Max-Pooling																	
					Training			Validation					Testing				
Experiment Name	C	Learning Rate	Dropout	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
LH_3D_S_A2	0.01	0.0001	d1 = 0.3 d2, d3 = 0.5	3D_CNN_4L_1	0.73	0.83	0.63	0.76	0.73	0.84	0.84	0.68	0.69	0.87	0.51	117	0.81
LH_3D_S_A4				3D_CNN_4L_2	1.00	1.00	1.00	0.77	0.73	0.84	0.86	0.67	0.67	0.73	0.60	209	0.83

Table 5.23. Performance metrics and hyperparameters of the experiments LH_3D_S_A {2, 4}. Adam - With/Without Max-Pooling

In Figure 5.95, we can see the average training and validation accuracies per fold, for the two experiments. The validation accuracy of the experiment LH_3D_S_A4 without Max-Pooling (light blue), increases more than the validation accuracy of the experiment LH_3D_S_A with Max-Pooling (light red) the first 150 epochs approximately, but then it faced a slight decrease over time. By observing Figure 5.96, with the average training

and validation losses per fold, for the two experiments, we can see that this happens because the validation loss of the model without Max-Pooling started increasing rapidly after the 100th epoch approximately, while its training loss kept decreasing, which means that the model overfits.

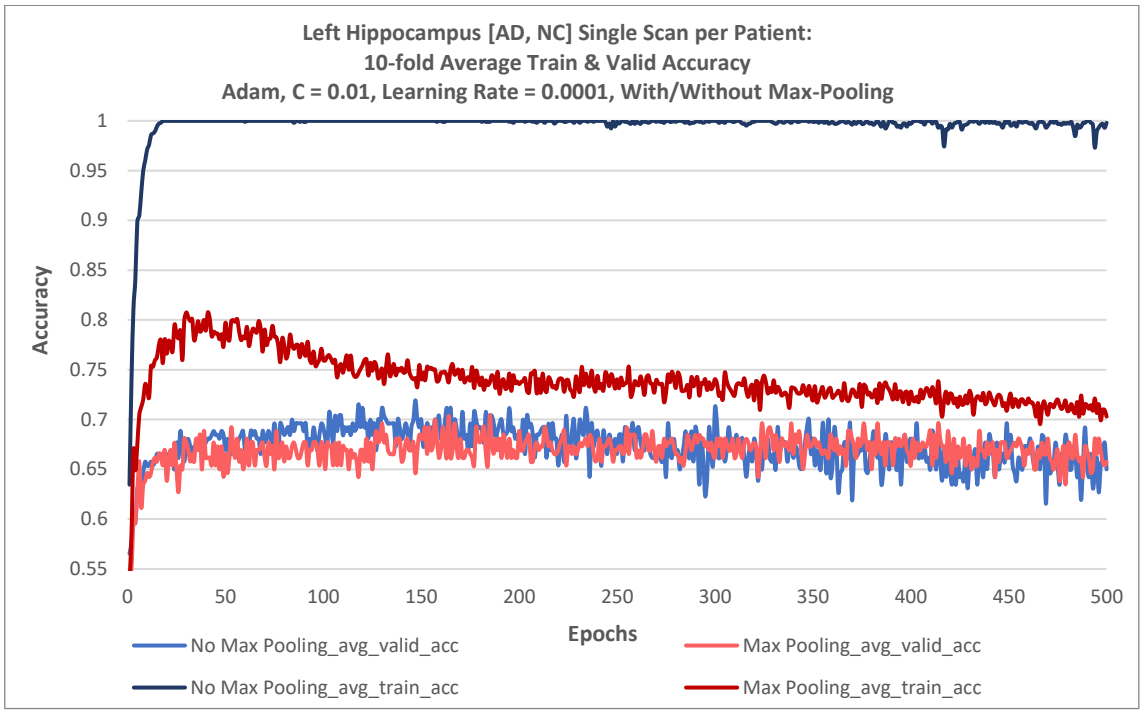


Figure 5.95. Average training and validation accuracies of the experiments LH_3D_S_A {2, 4}.

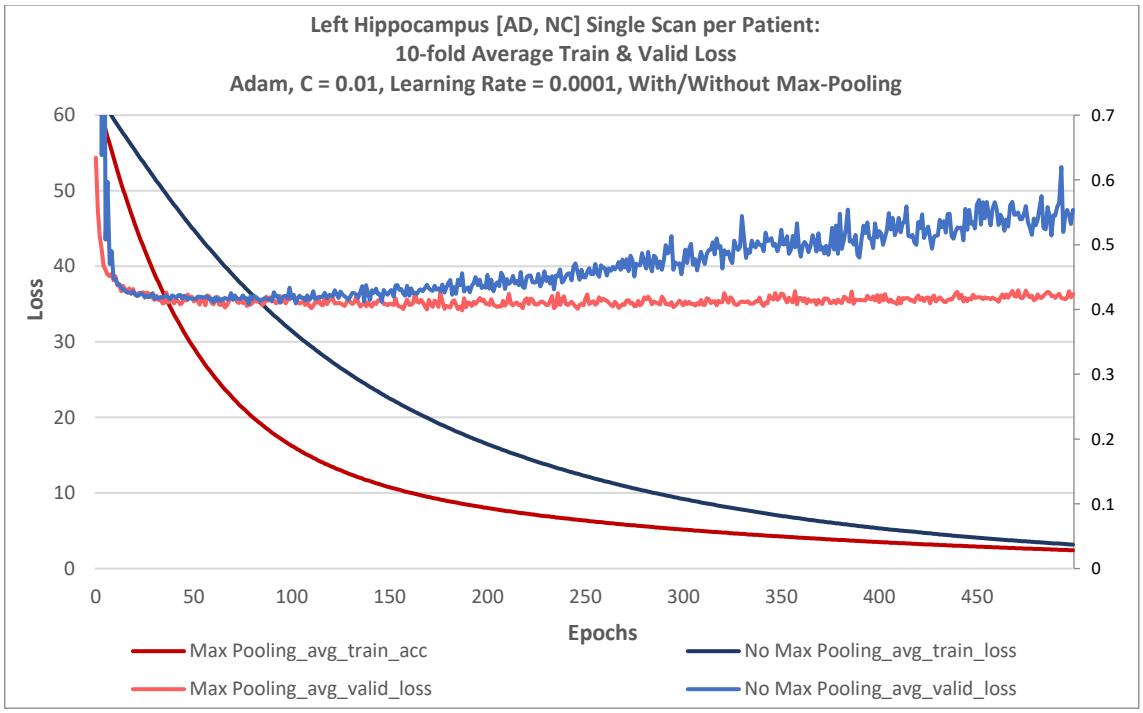


Figure 5.96. Average training and validation losses of the experiments LH_3D_S_A {2, 4}.

Overfitting was expected in the experiment LH_3D_S_A4 since the purpose of Max-Pooling is to help with the translation of the features in a given image (Section 2.3.12.5). The absence of Max-Pooling may cause generalization issues, which is what happened in our case as well. That is why the average training accuracy went fast to 100%, after the first 20 epochs in the LH_3D_S_A4 experiment.

5.8.2 CNN 4 Layer with NewtonCG

5.8.2.1 Experiments LH_3D_S_N {1, 2}: C = 0.01, GNsize = {50, 500}, With/Without Dropout, With/Without Max-Pooling

The experiments LH_3D_S_N {1, 2} compare the effect of Max-Pooling in the NewtonCG algorithm with the LH_3D dataset (Section 3.2.6). A key difference between the experiments with the Adam (Section 5.8.1) and the NewtonCG optimizer, is that the experiments with the NewtonCG were unable to train the model; for example, the experiment LH_3D_S_N1 (dark/light red, Figure 5.97). Different values of eta (η), xi (ξ), boost, drop, and CG max iterations were tried with no significant improvements (Section 4.3.1). The absence of Max-Pooling (LH_3D_S_N2) helped a lot against the underfitting.

Detailed training/validation accuracies/losses per fold for the experiments LH_3D_S_N {1, 2}and their confusion matrices for the training, validation, and testing, can be found in Appendixes L.2.1, L.2.2. The source codes for the network configurations 3D_CNN_4L_ {1, 2} are available respectively in Appendixes B.5.1 and B.5.12, at Code Snippets B.5 and B.16.

The experiment LH_3D_S_N2 uses the 3D_CNN_4L_2 network (Section 4.3.4.2.4), and the experiment LH_3D_S_N1 the 3D_CNN_4L_1 network (Section 4.3.4.2.3). The experiment LH_3D_S_N1 does not apply Dropout, it has a GNsize = 500, and C = 0.01. The experiment LH_3D_S_N2 has a GNsize = 50, C = 0.01, and applies Dropout to all three Convolutional Layers equal to d1 = d2 = 32, d3 = 64, with di being the drop probability in the i^{th} Convolutional Layer.

Table 5.24, shows the performance metrics of the two experiments. Since each fold of the LH_2D_S_N1 experiment fails to learn and terminates itself on average in the 32^{nd} epoch out of the 100 epochs; its average validation accuracy is 70%. The average validation accuracy for the 5-folds of the LH_3D_S_N2 experiment is 73%.

Experiments LH_3D_S_N {1, 2}: NewtonCG - With/Without Max-Pooling																
				Training			Validation					Testing				
Experiment Name	C	GNsize	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
LH_3D_S_N1	0.01	500	3D_CNN_4L_1	0.73	0.79	0.68	0.70	0.70	0.76	0.77	0.64	0.67	0.80	0.53	4	0.73
LH_3D_S_N2	0.01	50	3D_CNN_4L_2	0.86	0.87	0.84	0.73	0.72	0.76	0.78	0.68	0.69	0.74	0.63	21	0.83

Table 5.24. Performance metrics of the experiments LH_3D_S_N {1, 2}. NewtonCG - With/Without Max-Pooling

In Figure 5.97, the average training and validation accuracies of the two experiments are available. After 20 epochs, the average training accuracy (dark blue) of the experiment LH_3D_S_N2 grows fast and reaches 100%, while in the meantime, the average validation accuracy (light blue) starts decreasing. Both the average training (dark red) and validation (light red) accuracies of the experiment LH_3D_S_N1, do not increase, since each one of the 5-folds, terminated itself before the 32nd epoch. In Figure 5.98, the average training and validation losses of the two experiments are available. We can see that the experiment LH_3D_S_N2 without Max-Pooling overfits, as the validation loss increases after the 15th epoch approximately, while its average training loss decreases.

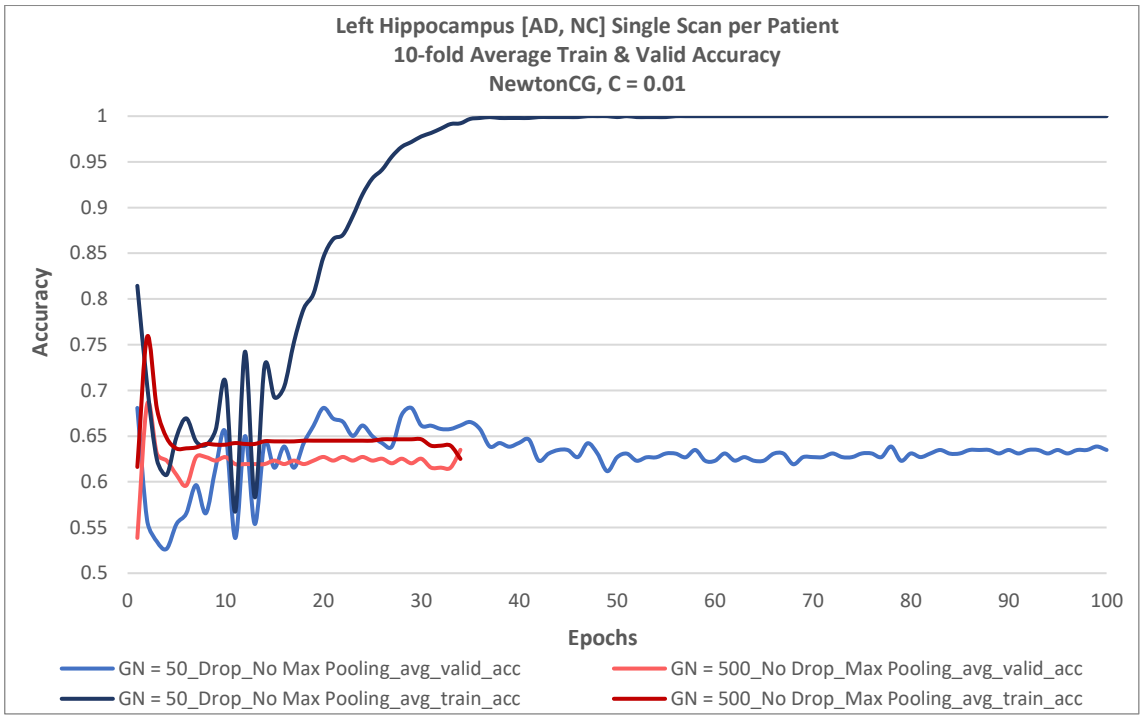


Figure 5.97. Average training and validation accuracies of the experiments LH_3D_S_N {1, 2}.

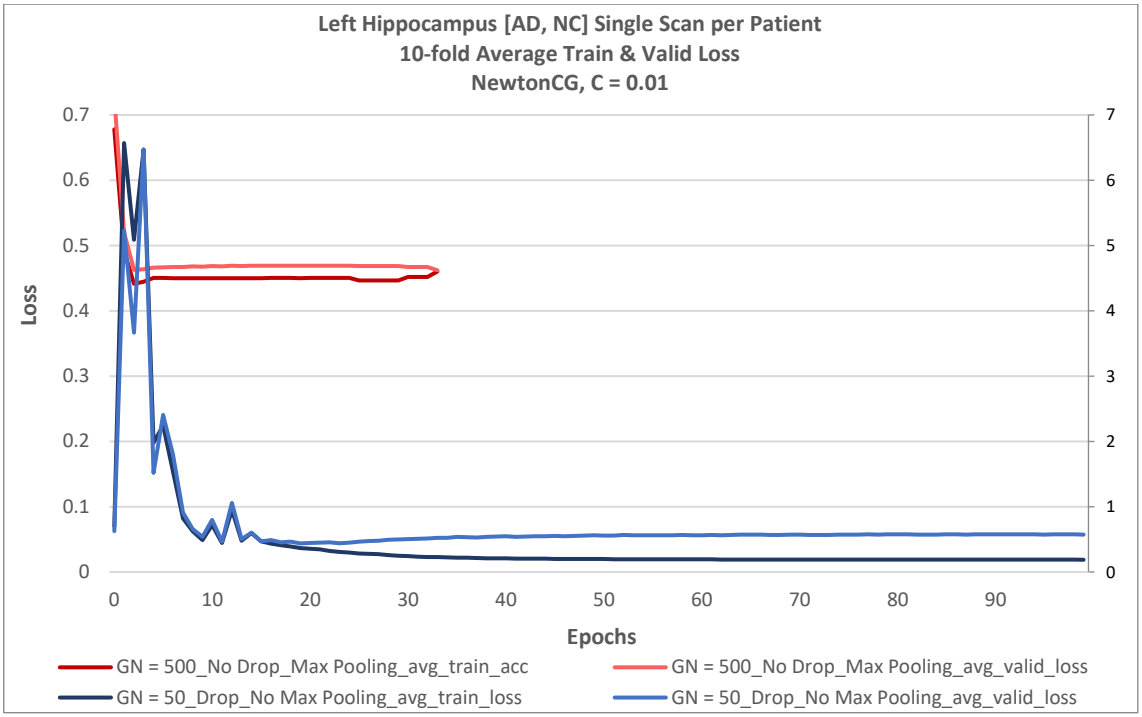


Figure 5.98. Average training and validation losses of the experiments LH_3D_S_N {1, 2}.

5.9 3D Shrunk Brains [AD, NC]: Single Scan per Patient (44 × 48 × 44)

The 3D Shrunk Brains, Single Scan per Patient (B_3D_S) dataset (Section 3.2.5.2) contains scaled-down 3D T1-weighted MRI scans from AD and NC patients, the same that was used in the 2D Brain Slices datasets (Section 3.2.3). The unscaled versions of the 3D images were too large to be used for the training of our models, based on the available computational power.

The main assumption was that the whole 3D image of the brain should perform better than a 2D slice of it. Based on the results of the experiments that have been performed, our assumption was not validated, especially for the case of the NewtonCG optimizer. Since the downscaling destroys much information of the image, we decided to try a different dataset, the 3D Cropped Brains (Section 3.2.7), which did not downscale the 3D image. In future work, the 3D Shrunk Brains dataset can be used in a CNN without Max-Pooling, since, such a configuration showed significant improvement with the 3D Left Hippocampus dataset (Section 5.8.1.2). The main issue is that more computational resources and execution time is going to be needed.

The experiments with the 2D datasets (Section 5.4 – 5.7) were executed with batch size = 256 since the input data were smaller in size and could fit in the RAM. For the 3D experiments (Section 5.8 – 5.10), as well as for the 3D Shrunk Brains datasets the batch size was set to 32 since each 3D image takes much more memory to be stored.

The 3D Shrunk Brains dataset performed better with the Adam optimizer, with a 75% average validation accuracy for the 5-folds, instead of 64% with the NewtonCG optimizer (Table 5.25).

3D Shrunk Brains – Single Scan per Patient [AD, NC]																
	Training				Validation						Testing					
Experiment Name	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
Adam																
B_3D_S_A1	0.83	0.05	0.80	0.86	0.75	0.03	0.81	0.73	0.69	0.82	0.70	0.02	0.67	0.73	131	0.81
NewtonCG																
B_3D_S_N1	0.72	0.12	0.80	0.64	0.64	0.04	0.62	0.73	0.79	0.50	0.64	0.07	0.80	0.47	6	0.68

Table 5.25. Performance metrics of the experiments with 3D Shrunk Brains – Single Scan per Patient [AD, NC]

5.9.1 CNN 4 Layers with Adam

5.9.1.1 Experiment B_3D_S_A1: C = 0.01, lr = 0.0001, Dropout

For the experiment B_3D_S_A1, the 3D Shrunk Brains [AD, NC], with a Single Scan per Patient (B_3D_S) dataset was used with the Adam optimizer. The network configuration that was used was the 3D_CNN_4L_1 (Section 4.3.4.2.1). The training process took 500 epochs, the learning rate was set to 0.0001 (the best of the 3D Left Hippocampus experiments in Section 5.8.1.1), and the C equal to 0.01. Standard Dropout (Section 2.3.13.9) d1 = d2 = 32, and d3 = 64, was applied to all three Convolutional Layers, with di being the drop probability in the i^{th} layer.

Figure 5.99, shows the average training and validation accuracies of the 5-folds, for the B_3D_S_A1 experiment. Both the average training and validation accuracies behave very similarly to the ones in the 3D Left Hippocampus dataset with the Adam dataset and learning rate = 0.0001 (Figure L.10 in Appendix L.1.2). The training accuracy increases fast in the first 50 epochs and then decreases slightly where it remains relatively stable on

an average value of 83% for the rest of the training process. The average validation accuracy increases smoothly but fluctuates a lot between 65% and 70%. At epoch 250 approximately, both the training and the validation losses begin to decline. In Figure 5.100 of the average training and validation losses, we do not observe any significant signs of overfitting.

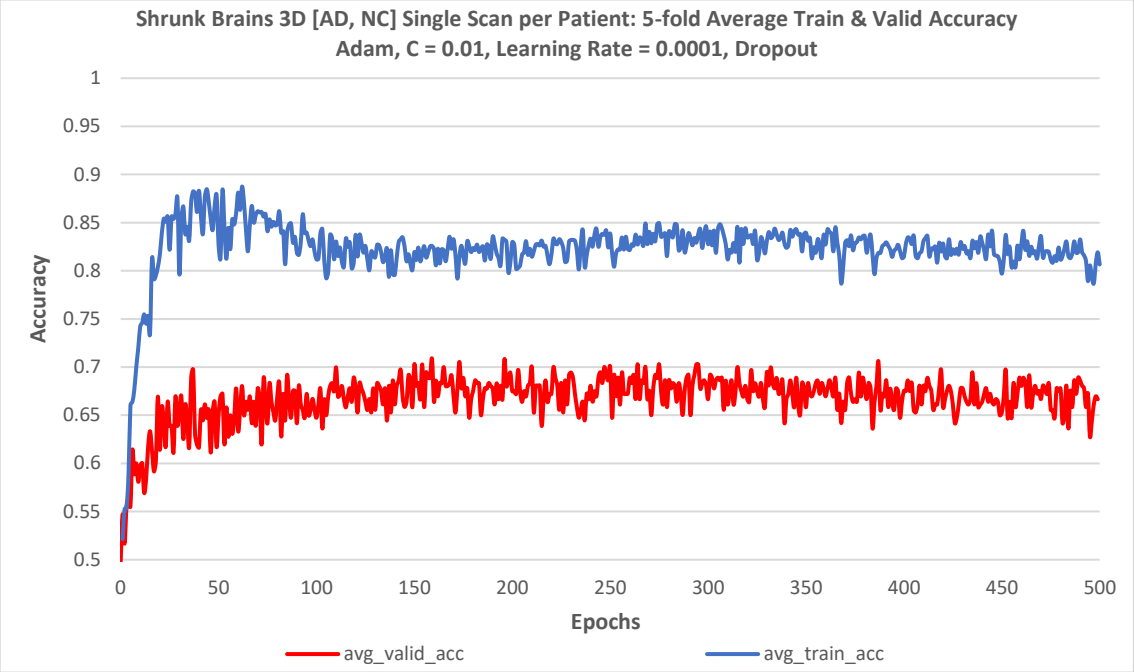


Figure 5.99. Average training and validation accuracy of the experiment B_3D_S_A1.

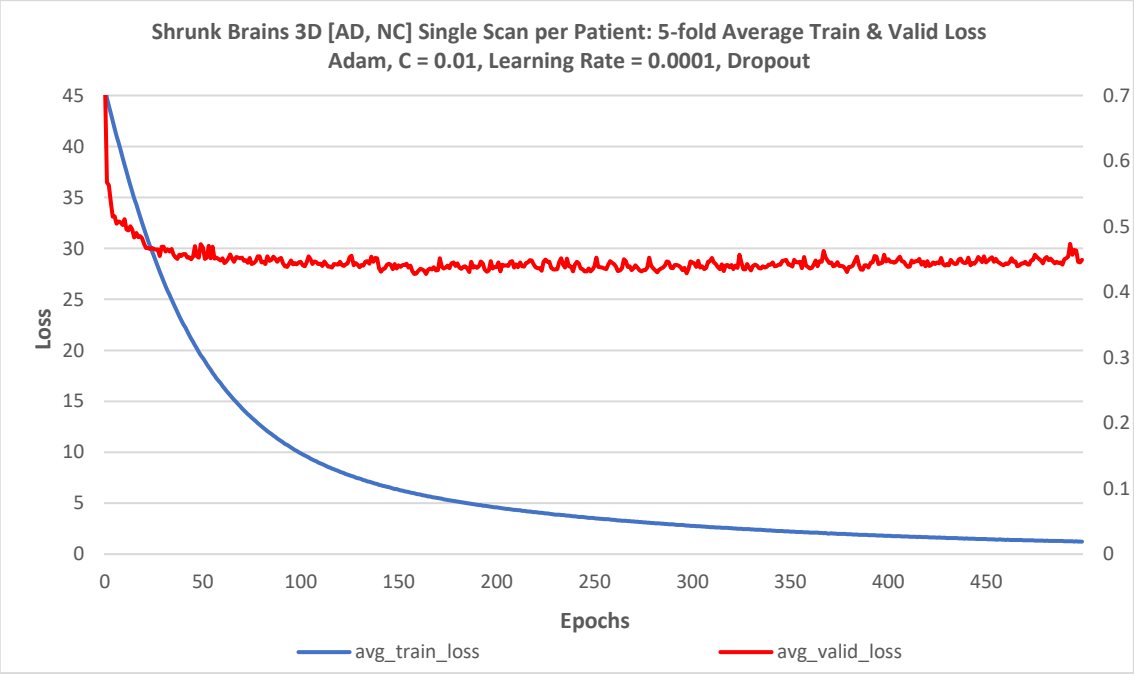


Figure 5.100. Average training and validation accuracy of the experiment B_3D_S_A1.

5.9.2 CNN 4 Layers with NewtonCG

5.9.2.1 Experiment B_3D_S_N1: C = 0.01, GNsize = 200, Dropout

Unlike the experiment B_3D_S_A1 with the Adam optimizer and the 3D Shrunk Brains dataset (Section 5.9.1.1), the experiment B_3D_S_N1 with the NewtonCG optimizer underfits, since it fails to learn and 3 out of 5-folds terminate themselves during the middle of the training process.

5.9.2.1.1 Accuracy

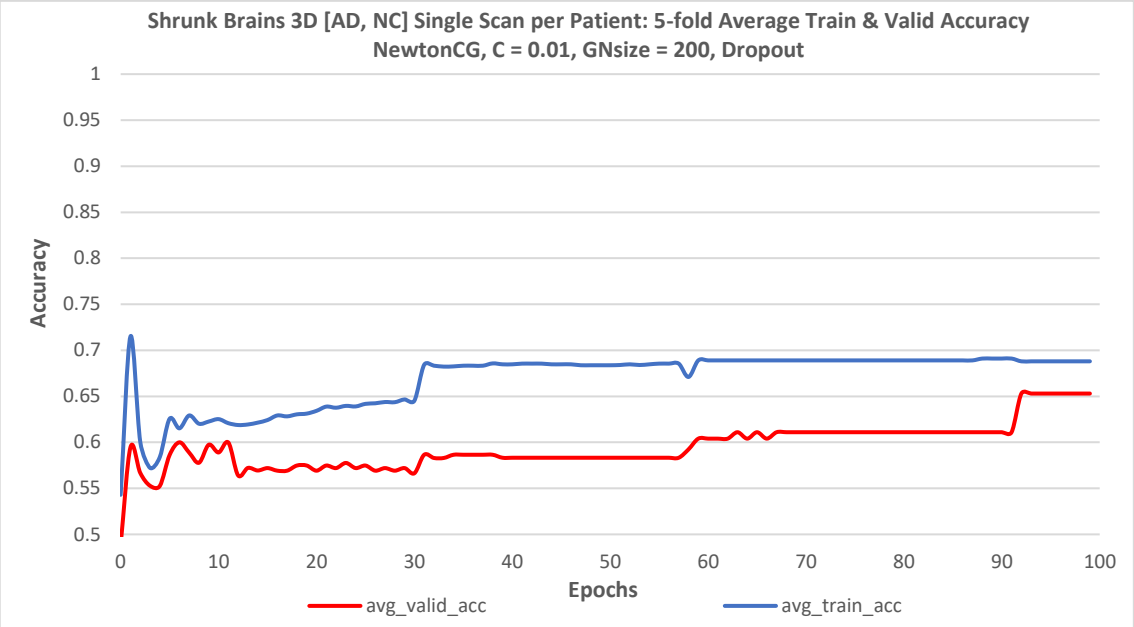


Figure 5.101. Average training and validation accuracy of the experiment B_3D_S_N1.

Figure 5.101, shows the average training and validation accuracies of the 5-folds, for the experiment B_3D_S_N1. Their shape is not representative of the model's performance. The average accuracies show a promising outcome, with a lot of potential for learning but in reality, the average validation accuracy is equal to 64% (Table 5.25). We have to observe the training and validation accuracy of each fold separately to understand that the network suffers from underfitting.

As we can see in Figures 5.102 & 5.103, with the training and validation accuracies respectively of each fold, the 2nd, 4th, and 5th folds terminate themselves before the 60th epoch. The NewtonCG implementation from Wang et al. (2002) does this automatically if for 15 continuous epochs no change is observed in the validation accuracy.

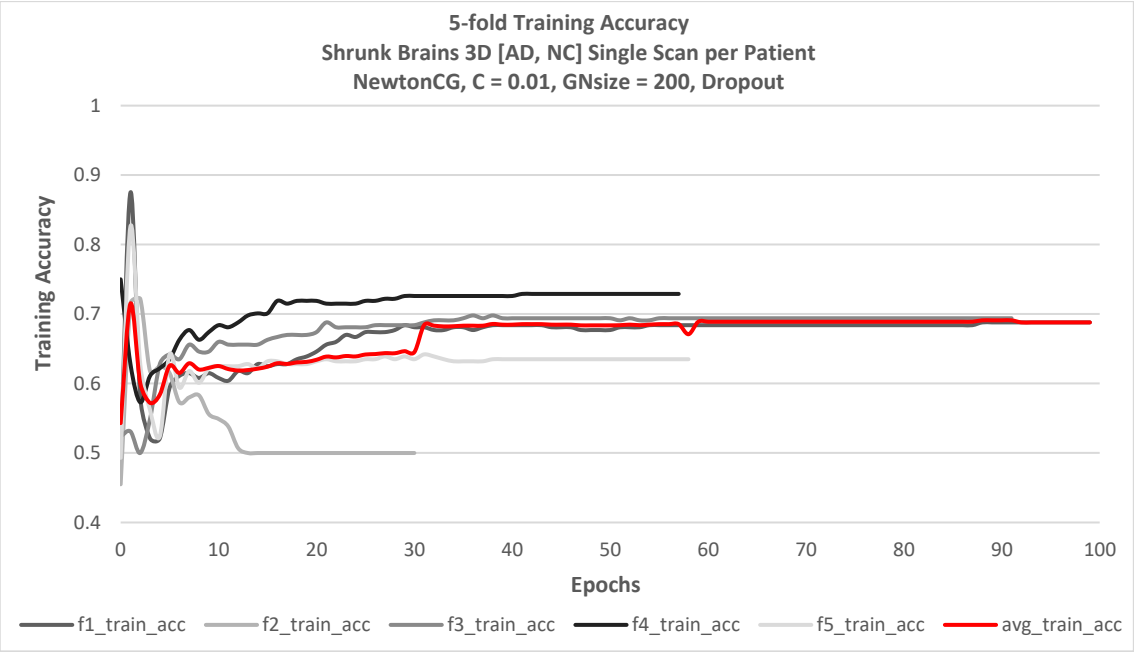


Figure 5.102. Training accuracy per fold of the experiment B_3D_S_N1.

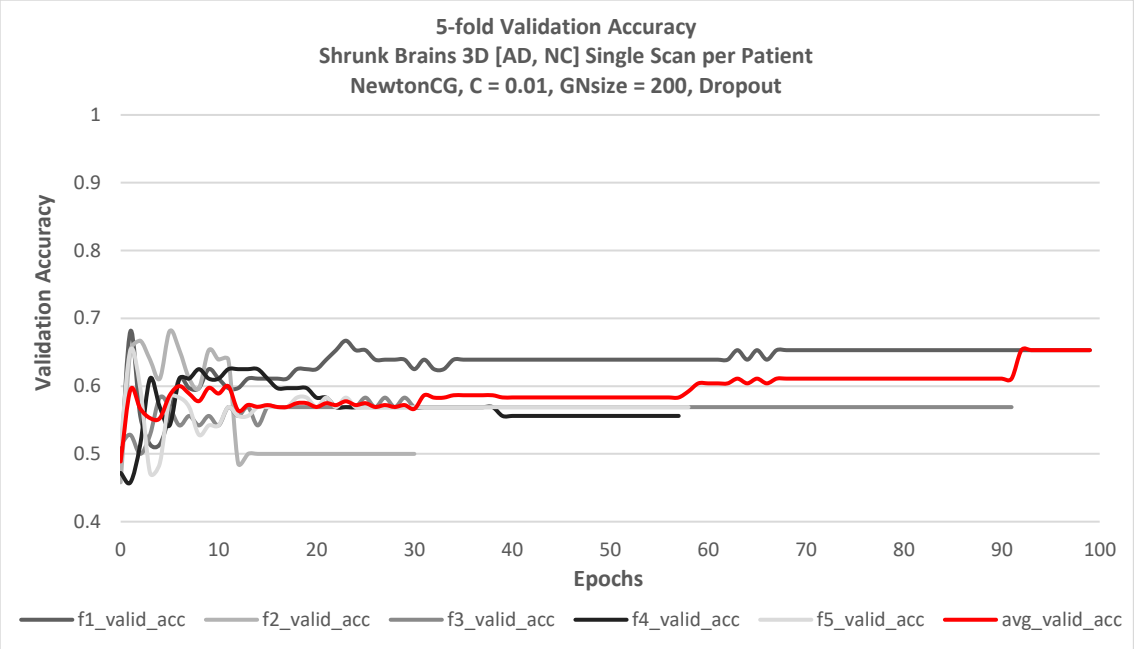


Figure 5.103. Validation accuracy per fold of the experiment B_3D_S_N1.

5.9.2.1.2 Loss

Figure 5.104, with the average training and validation losses of the 5-folds, for the experiment B_3D_S_N1, is not representative of the real performance of the model as well, for the same reason why the average accuracy (Figure 5.101) is not. If we observe the training and validation losses plots for each fold (Figure 5.105 & 5.106), especially the validation loss (Figure 5.106), we can see that it does not reduce at all during the

training process; it remains stable approximately at a value close to 0.5. The conclusion is that our model is underfitting, and most probably the main reason behind this is the dataset, which is incapable of teaching our network to detect AD and NC patients.

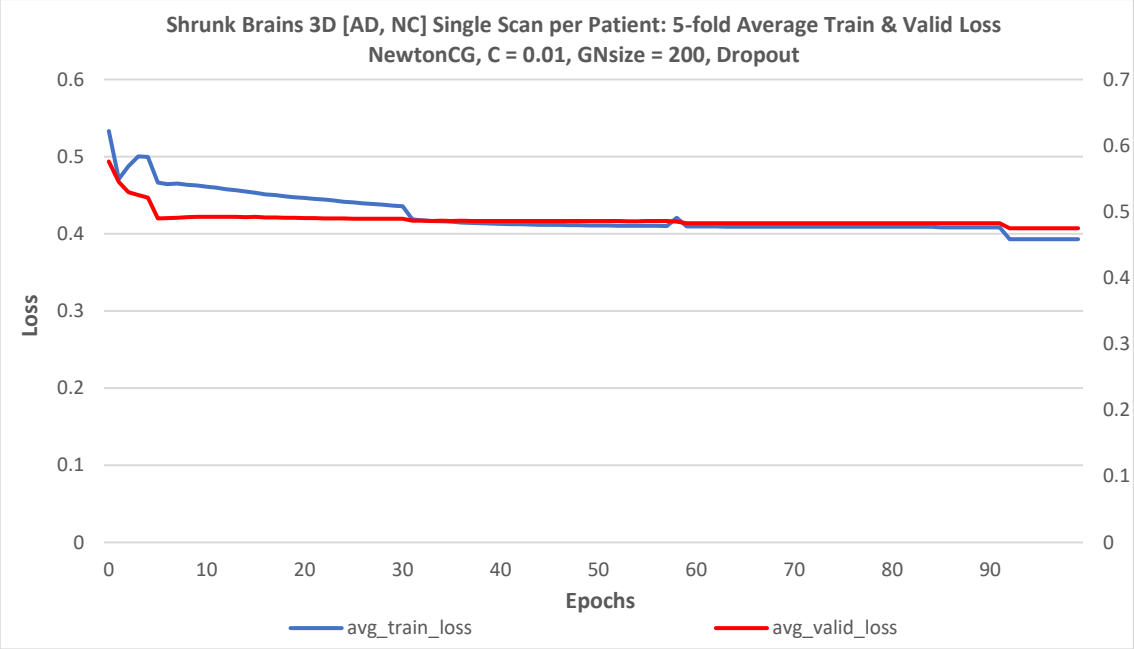


Figure 5.104. Average training and validation loss of the experiment B_3D_S_N1.

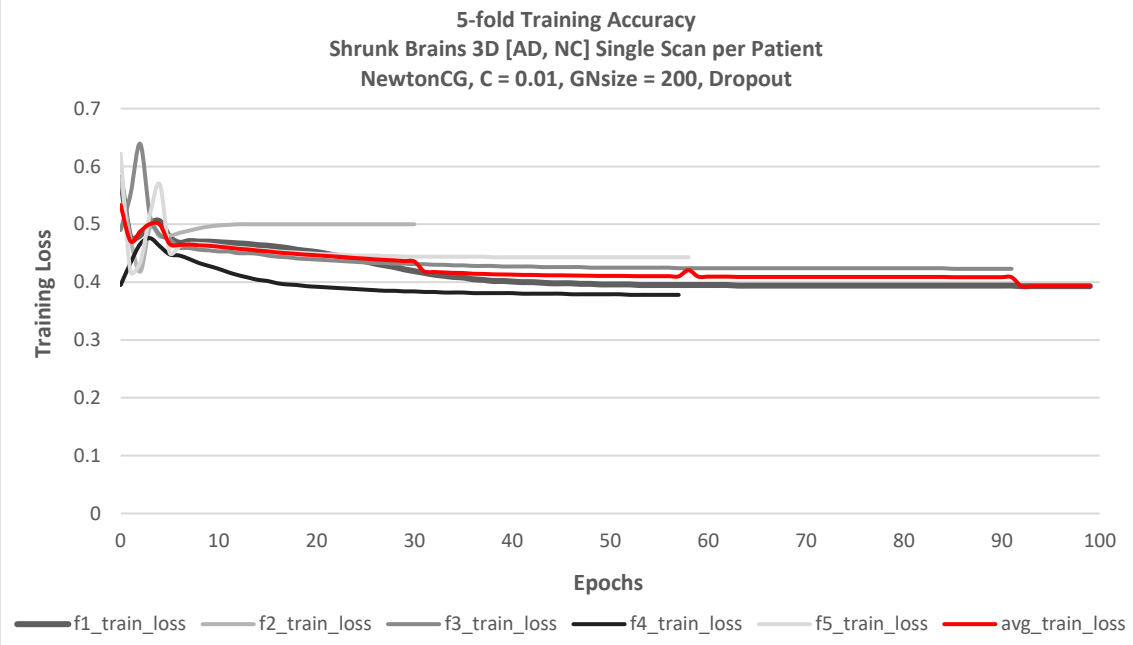


Figure 5.105. Training loss per fold of the experiment B_3D_S_N1.

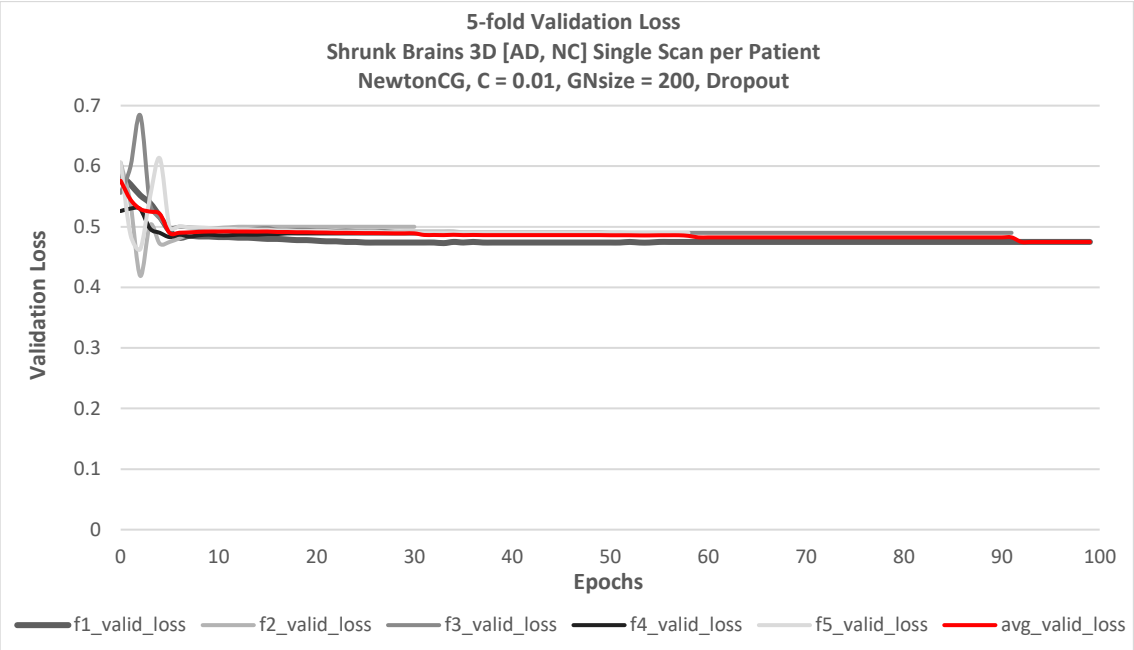


Figure 5.106. Validation loss per fold of the experiment B_3D_S_N1.

5.10 3D Cropped Brains [AD, NC]: Single Scan per Patient (70 × 60 × 60)

The 3D Cropped Brains [AD, NC], Single Scan per Patient (CB_3D_S) dataset (Section 3,2,7,2) was created because the 3D Shrunk Brains dataset was destroying useful information about the patient's target class due to the downscaling of the original 3D T1-weighted MRIs (Section 5.9). Also, using the 3D scan of the image is expected to be better than just a 2D slice, as the 2D datasets do (Section 5.4 – 5.7).

The performance of the CB_3D_S dataset was worse than the best experiment with the 2D dataset B_2D_M_N19 (Table 5.7), but it has some potential for improvement. For example, if no Max-Pooling is applied in future work, this dataset might perform better based on our observation in the 3D Left Hippocampus dataset (Section 5.8.1.2). Unfortunately, this will increase the computational power and time needed for the training process. Especially in the CB_3D_S, this is even more demanding because the size of the feature maps on each Convolutional Layer will be $70 \times 60 \times 60$.

In the CB_3D_S dataset, similarly as in the rest of the 3D datasets, the 3D Left Hippocampus (LH_3D_S) (Section 5.8), and 3D Shrunk Brains (B_3D_S) (Section 5.9), the Adam optimizer performs better than the NewtonCG. More specifically, based on the

best experiments of the CB_3D_S dataset, Adam's average validation accuracy is 73%, while for the NewtonCG is 71%.

3D Cropped Brains – Single Scan per Patient [AD, NC]																
	Training				Validation						Testing					
Experiment Name	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. PPV	Avg. NPV	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
Adam																
CB_3D_S_A1	0.98	0.03	1.00	0.96	0.73	0.02	0.72	0.75	0.77	0.69	0.65	0.06	0.71	0.60	71	0.76
NewtonCG																
CB_3D_S_N1	0.76	0.08	0.77	0.75	0.71	0.04	0.72	0.73	0.72	0.71	0.67	0.04	0.68	0.65	13	0.76
CB_3D_S_N2	0.68	0.02	0.65	0.71	0.69	0.06	0.70	0.68	0.64	0.73	0.69	0.04	0.72	0.67	9	0.74

Table 5.26. Performance metrics of the experiments with 3D Cropped Brains – Single Scan per Patient [AD, NC].

The reason could be that in Adam, we can control the learning rate, for example, reduce it if the model is underfitting. The NewtonCG has multiple hyperparameters, that play an essential role in underfitting and how aggressive is the algorithm. Some of them are the eta (η), xi (ξ), boost, drop, and CG max iterations (Section 4.3.1). To find the ideal combination of them, many experiments are needed to be performed, but since our resources were limited it was impossible to do that.

5.10.1 CNN 4 Layers with Adam

5.10.1.1 Experiment CB_3D_S_A1: C = 0.01, lr = 0.0001, Dropout

The experiment CB_3D_S_A1 (Appendix N.1.1) uses the CB_3D_S dataset (Section 3.2.7.2) with the Adam optimizer. The experiment uses the 3D_CNN_4L_1 network configuration (Section 4.3.4.2.2). Also, the learning rate = 0.0001 and C = 0.01. The experiment applies to all three Convolutional Layers the standard Dropout (Section 2.3.13.9) with d1 = d2 = 32, and d3 = 64, where di, is the drop probability in the i^{th} layer.

Figure 5.107, shows the average training and validation accuracies of the 5-folds, for the CB_3D_S_A1 experiment. In the first 50 epochs, the average training and validation accuracies increase. Afterward, the average training accuracy (blue) goes towards 100%, while the validation accuracy (red) starts declining. This resulted in an average training

accuracy of 98% and an average validation accuracy of 73% for this experiment (Table 5.26).

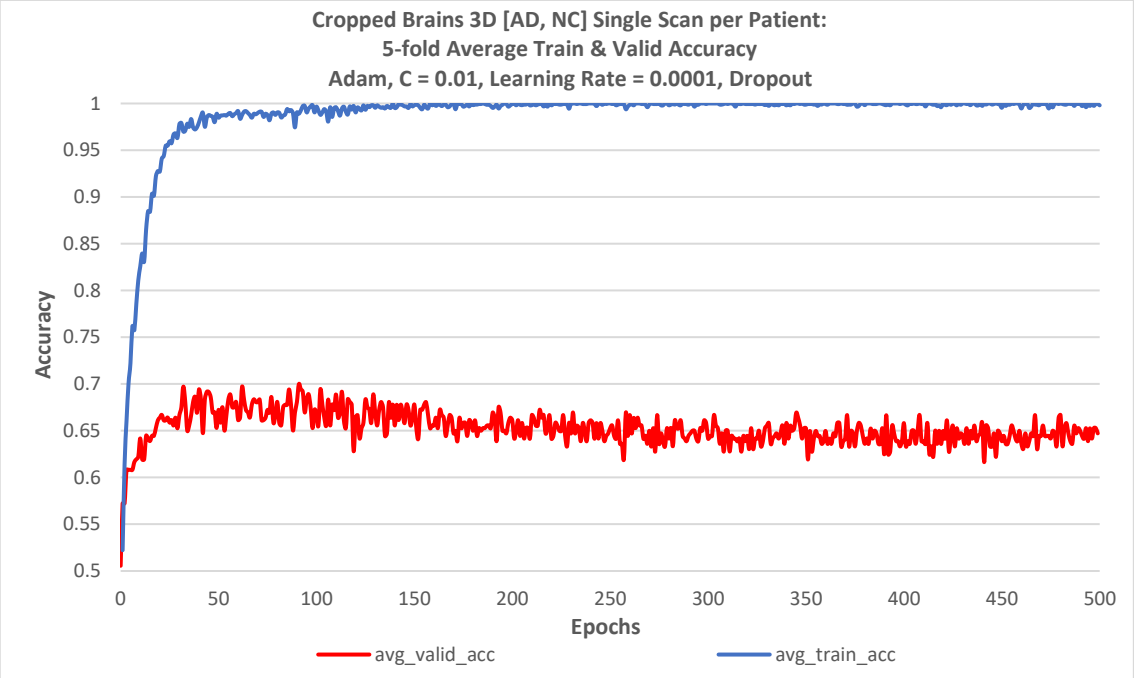


Figure 5.107. Average training and validation accuracy of the experiment CB_3D_S_A1.

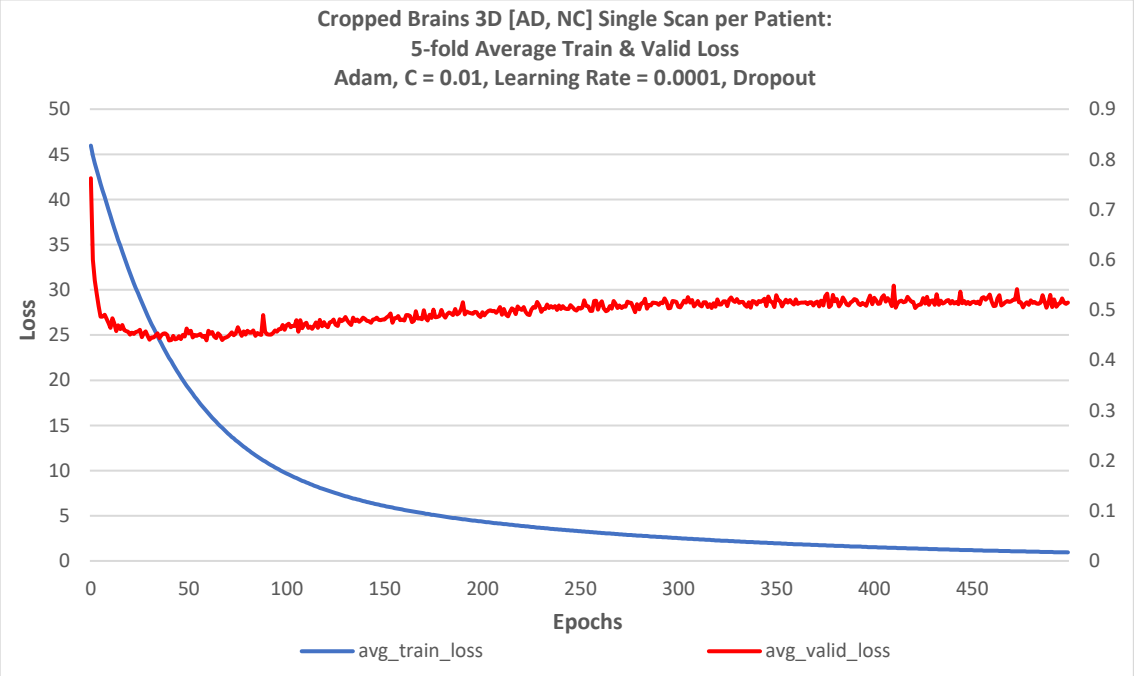


Figure 5.108. Average training and validation loss of the experiment CB_3D_S_A1.

In Figure 5.108, the average training and validation losses for the experiment are available. After the first 50 epochs, the average validation loss starts increasing while the

training loss continues to decrease; therefore, the model overfits and fails to generalize. Additionally, the average epoch of the best validation accuracy per fold is 71 out of the 500 epochs. This verifies us that the model overfits very early during the training process.

5.10.2 CNN 4 Layers with NewtonCG

5.10.2.1 Experiment CB_3D_S_N1: C = 0.01, GNsize = 50, Dropout

The experiment CB_3D_S_N1 (Appendix N.2.1) uses the CB_3D_S dataset (Section 3.2.7.2) with the NewtonCG optimizer. The hyperparameters are the same as in the experiment B_2D_M_N10 (Appendix H.2.10) that used the B_2D_M dataset (Section 3.2.3.4), and achieved average validation accuracy equal to 80% (Table 5.7). Contrariwise, this experiment achieves a validation accuracy equal to 71% (Table 5.26). The CB_3D_S_N1 has GNsize = 50, C = 0.01, and applies Dropout (Section 2.3.13.9) to all three Convolutional Layers with d1 = d2 = 32, and d3 = 64, where di, is the drop probability in the i^{th} layer.

In Figure 5.109, we can see the average training and validation accuracies of the 50folds, for the experiment CB_3D_S_N1. After the first 10 epochs, the validation accuracy declines while the training accuracy goes rapidly towards 100%. In Figure 5.110, the average training and validation losses are available. The figure verifies us that the model overfits very early during the training process, just after the 10 first epochs.

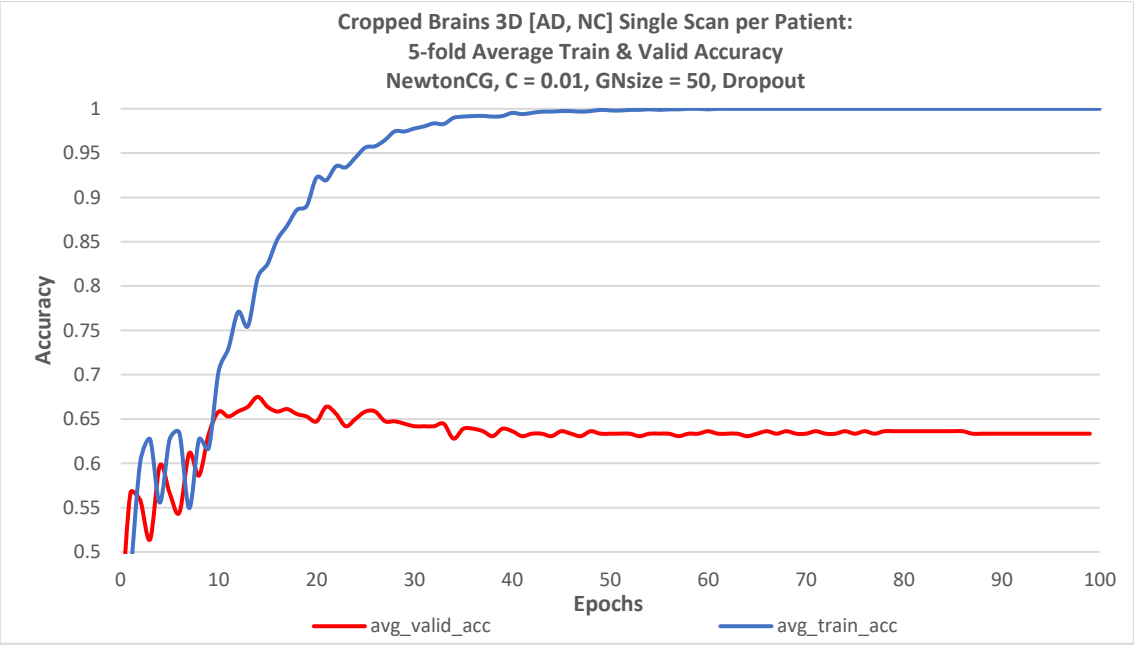


Figure 5.109. Average training and validation accuracy of the experiment CB_3D_S_N1.

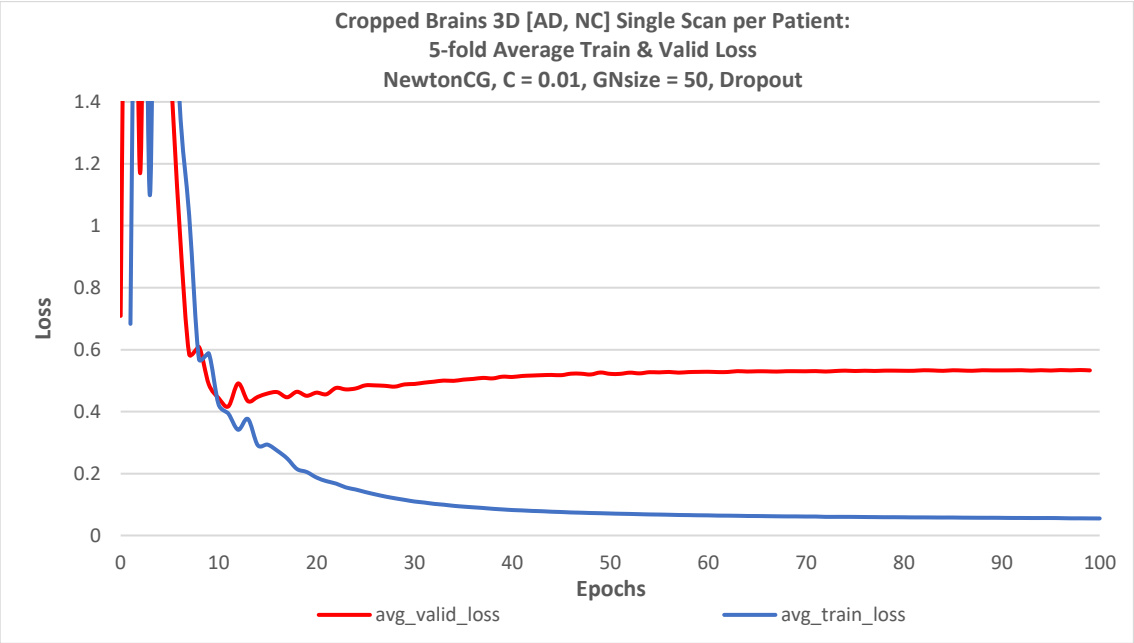


Figure 5.110. Average training and validation loss of the experiment CB_3D_S_N1.

A possible solution to the overfitting problem could be to remove the Max-Pooling from each Convolutional Layer since the 3D Left Hippocampus experiment showed some improvement (Section 5.8.1.2), but this would be much more computationally expensive than it is now.

5.10.3 CNN 5 Layers with NewtonCG

5.10.3.1 Experiment CB_3D_S_N2: C = 0.01, GNsize = 50, Dropout

The experiment CB_3D_S_N2 uses the 5-layer CNN network 3D_CNN_5L_1 (Section 4.3.4.3.1), instead of the 4-layer CNN network 3D_CNN_4L_1 (Section 4.3.4.2.2) which the CB_3D_S_N1 experiment was using. The source code for the 3D_CNN_5L_1 network is available in Appendix 5.8.14, at Code Snippet B.18. The reason was to check whether the CB_3D_S_N1 is underfitting or overfitting. If it was underfitting, a deeper network architecture should potentially show some improvement. Unfortunately, the performance was worse, since the average validation accuracy is 69% while for the CB_3D_S_N1 experiment was 71% (Table 5.26).

Consequently, we can assume that the model 3D_CNN_4L_1 did not underfit but overfit, therefore other regularization techniques for overfitting should be used to solve the problem (Section 2.3.13). Since we are using already Cross-validation and Early Stopping, we could train with more data, use Regularization, remove Max-Pooling, etc.

We can see in Figure 5.111 that the experiment CB_3D_S_N2 performs significantly worse than the CB_3D_S_N1. The reason is that this experiment does not even manage to finish the training for 100 epochs. The performance, and more specifically the average validation accuracy remains completely stable for 15 epochs, which causes the training process to terminate automatically based on Wang et al.'s (2020) algorithm with the NewtonCG and CNNs for Python.

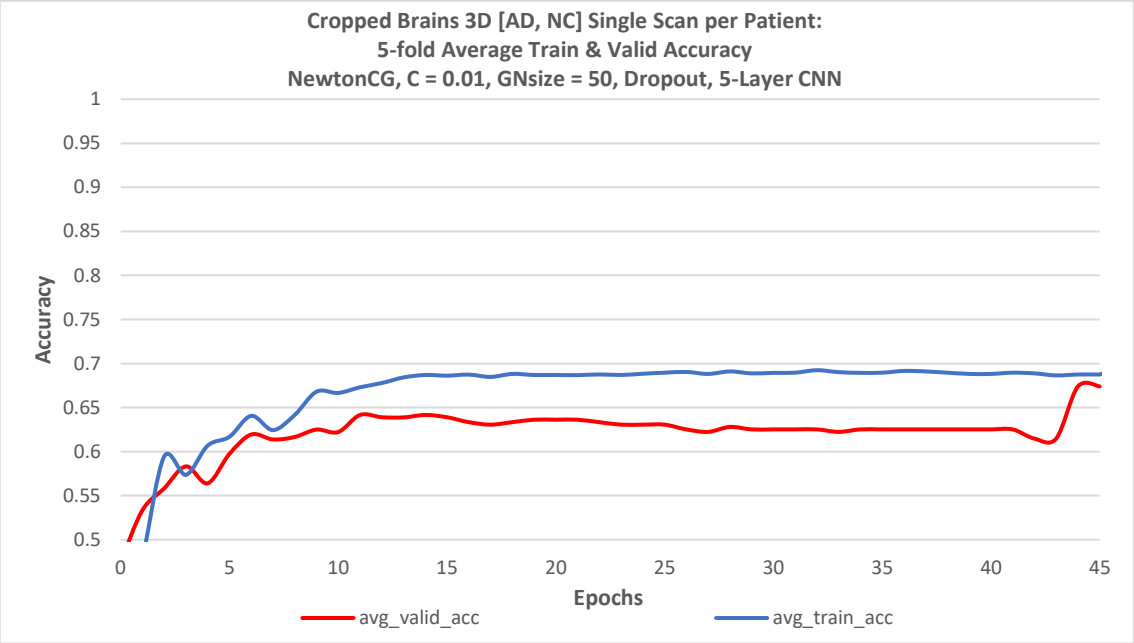


Figure 5.111. Average training and validation accuracy of the experiment CB_3D_S_N2.

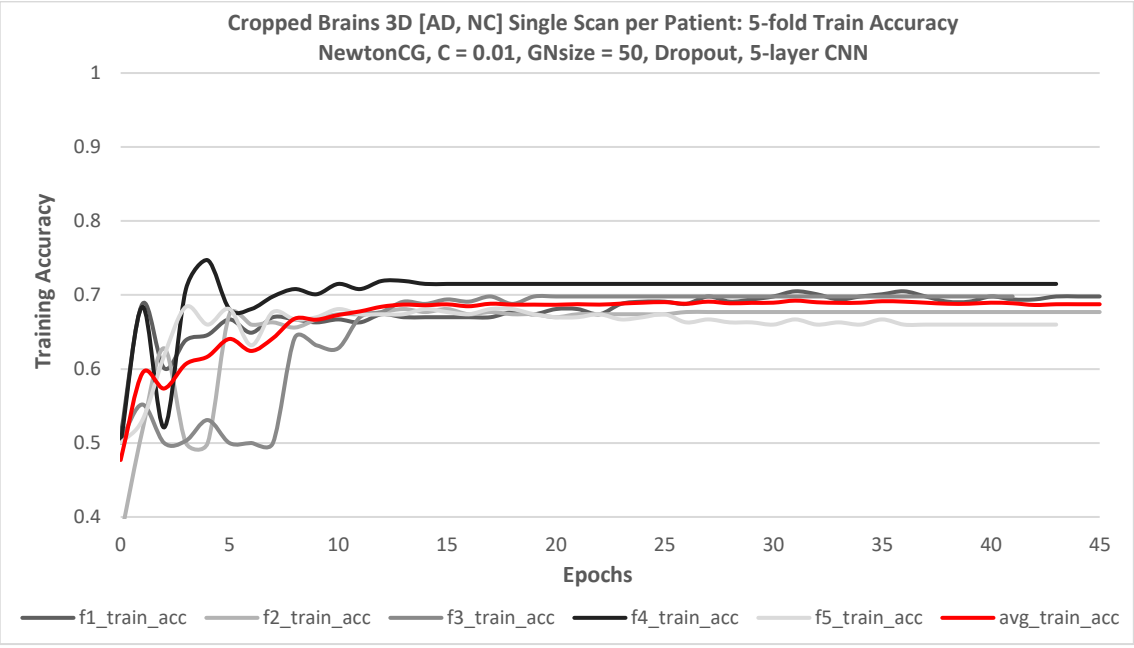


Figure 5.112. Training accuracy per fold of the experiment CB_3D_S_N2.

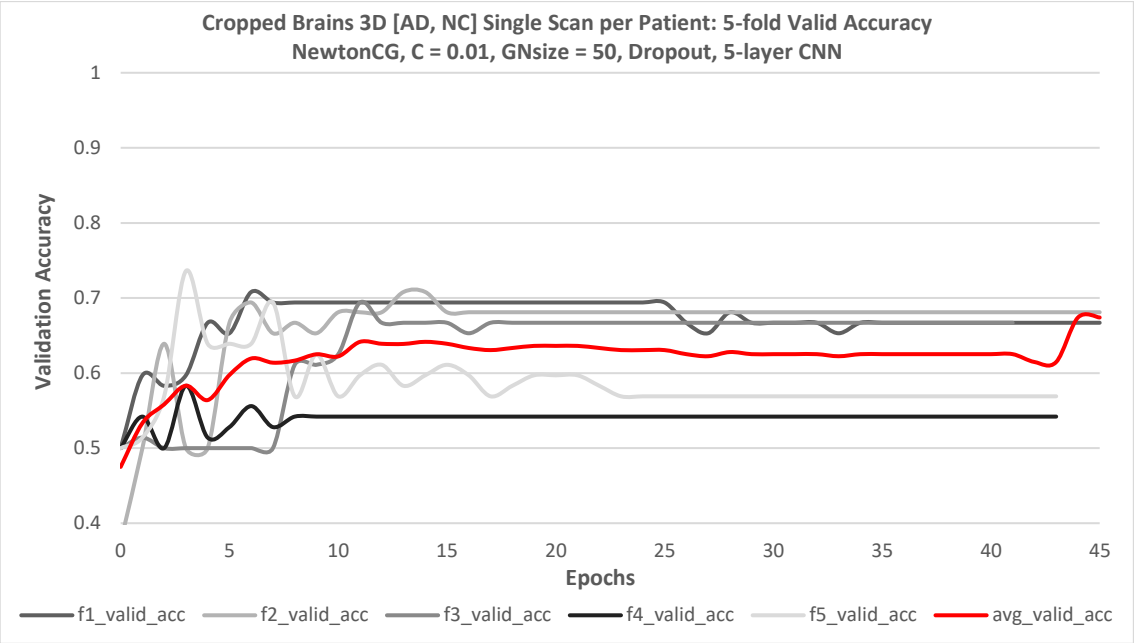


Figure 5.113. Validation accuracy per fold of the experiment CB_3D_S_N2.

In Figures 5.112 & 5.113 of the training and validation accuracies respectively for each fold, of the experiment CB_3D_S_N2, we can see that the model is underfitting. Basically, all folds perform horribly and they fail to learn.

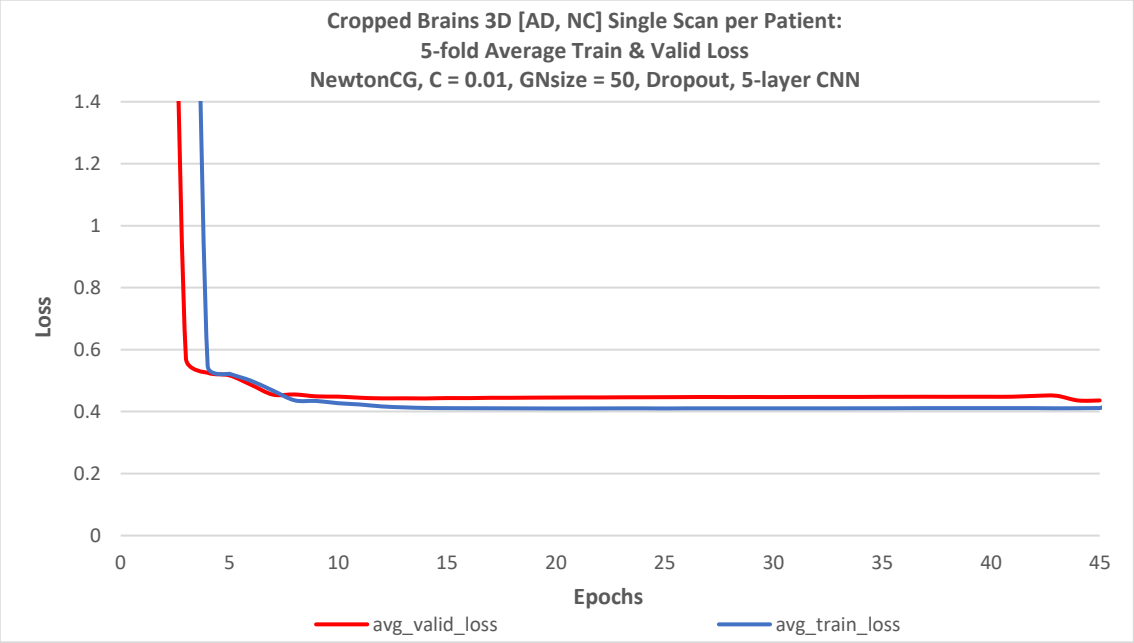


Figure 5.114. Average training and validation loss of the experiment CB_3D_S_N2.

Figure 5.114, shows the average training and validation losses of the 5-folds, of the experiment CB_3D_S_N2. They show us that the model is unable to learn. The losses decrease very rapidly in the first 5 epochs to a value close to 0.4. Then, they remain stuck

in both the training and validation loss for the rest of the training process. Figures 5.115 & 5.116 show the training and validation loss respectively, of each fold, for the experiment CB_3D_S_N2. The behavior of all experiments is very similar as we can see in these figures.

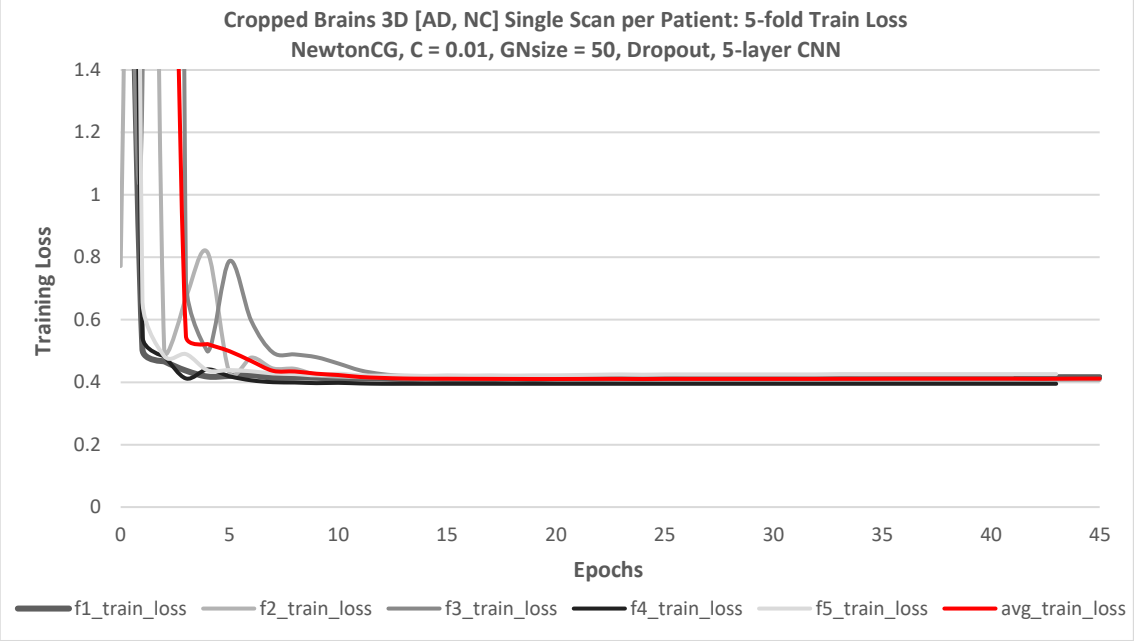


Figure 5.115. Training loss per fold of the experiment CB_3D_S_N2.

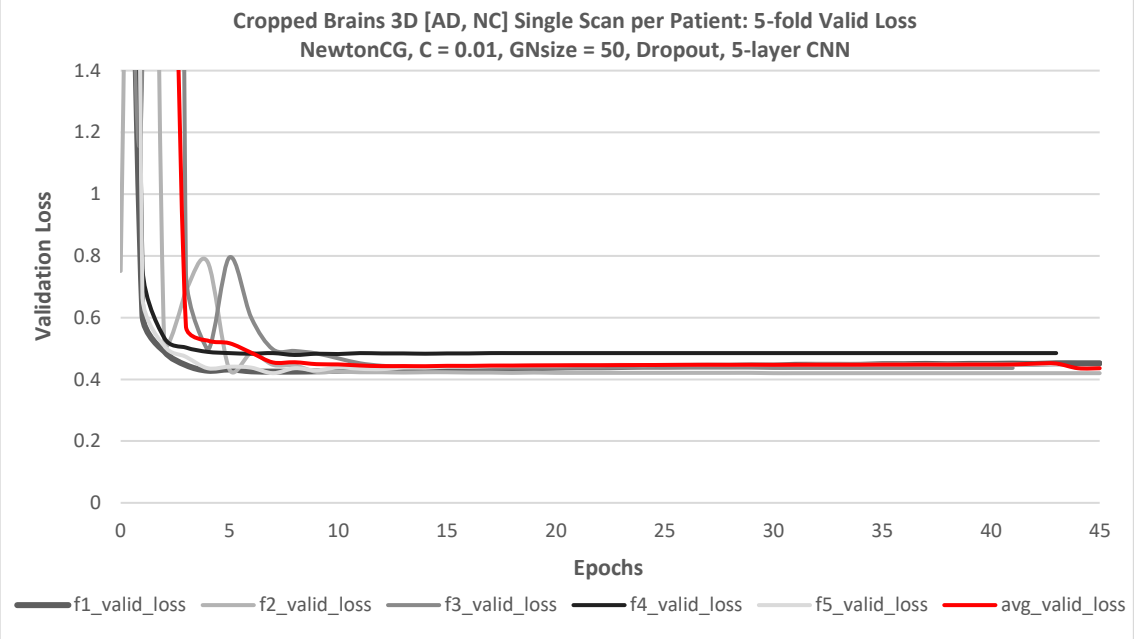


Figure 5.116. Validation loss per fold of the experiment CB_3D_S_N2.

5.11 2D Brain Slices [AD, MCI, NC]: Multiple Scans per Patient (174 × 174)

The 2D Brain Slices [AD, MCI, NC], Multiple Scans per Patient – Single Slice per Scan, or so-called B_2D_M [AD, MCI, NC] dataset (Section 3.2.4.1) is the same as the B_2D_M dataset (Section 3.2.3.4) with MCI patients also added. The same 10-fold Cross-validation splitting was used as in the B_2D_M dataset, for direct comparison with the B_2D_M experiments (Section 5.4). the source code for splitting the dataset is available in Appendix D.3.3, at Code Snippet D.9.

In general, the performance of this model is mediocre, with the Adam optimizer scoring up to 54% average validation accuracy, while the NewtonCG up to 55% (Table 5.27). The main conclusion is that the 2D slices used in the dataset of the T1-weighted MRI scans, do not provide enough information, to distinguish easily an MCI from an NC or an AD patient.

2D Brain Slices – Multiple Scans per Patient [AD, MCI, NC]																				
						Training				Validation					Testing					
Experiment Name	Optimizer	C	GNsize	Max Epochs	Learning Rate	Avg. Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. Accuracy	Best Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. epoch of best Valid Acc. per fold
B_2D_M_A1 [AD, MCI, NC]	Adam	0.01	-	500	0.001	0.83	0.89	0.87	0.89	0.54	0.58	0.71	0.68	0.71	0.35	0.63	0.51	0.56	26
B_2D_M_A2 [AD, MCI, NC]	Adam	0.01	-	500	0.0001	0.87	0.92	0.91	0.91	0.53	0.62	0.72	0.66	0.69	0.32	0.61	0.47	0.55	45
B_2D_M_N1 [AD, MCI, NC]	NewtonCG	0.01	50	100	-	0.78	0.88	0.83	0.85	0.55	0.60	0.71	0.67	0.71	0.33	0.59	0.49	0.58	23

Table 5.27. Performance metrics and hyperparameters of the experiments with the 2D Brain Slices – Multiple Scans per Patient [AD, MCI, NC] dataset.

5.11.1 CNN 4 Layers with Adam

5.11.1.1 Experiments B_2D_M_A {1, 2} [AD, MCI, NC]: C = 0.01, Dropout, lr = {0.001, 0.0001}

The experiments B_2D_M_A {1, 2} [AD, MCI, NC], compare the performance of the learning rates 0.001, and 0.0001, with the Adam optimizer in the AD/MCI/NC problem, by using the B_2D_M [AD, MCI, NC] dataset. The dataset contains an equal number of AD, MCI, and NC patients, and the training, validation, and test sets are all balanced. The network used for the experiments is the 2D_CNN_4L_1 (Section 4.3.2.3.1) with Dropout,

d1 = d2 = 32, and d3 = 64, where di is the drop probability in the i^{th} Convolutional Layer. The source code for the network is available in Appendix B.5.2, at Code Snippet B.6.

The experiment B_2D_M_A1 [AD, MCI, NC] with the learning rate = 0.001 and the average validation accuracy of 54%, performs slightly better than the experiment B_2D_M_A2 [AD, MCI, NC] with the learning rate = 0.0001 and the average validation accuracy of 53% (Table 5.28).

Experiments B_2D_M_A {1, 2} [AD, MCI, NC]																				
						Training				Validation					Testing					
Experiment Name	Optimizer	C	GNsize	Max Epochs	Learning Rate	Avg. Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. Accuracy	Best Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. epoch of best Valid Acc. per fold	
B_2D_M_A1 [AD, MCI, NC]	Adam	0.01	-	500	0.001	0.83	0.89	0.87	0.89	0.54	0.58	0.71	0.68	0.71	0.35	0.63	0.51	0.56	26	
B_2D_M_A2 [AD, MCI, NC]	Adam	0.01	-	500	0.0001	0.87	0.92	0.91	0.91	0.53	0.62	0.72	0.66	0.69	0.32	0.61	0.47	0.55	45	

Table 5.28. Performance metrics and hyperparameters of the experiments B_2D_M_A {1, 2} [AD, MCI, NC].

Since the B_2D_M [AD, MCI, NC] dataset is balanced, if classifying all inputs to a single class, then the validation accuracy of that model with this dataset would be 33%. Contrariwise, for the experiments with the B_2D_M dataset (Section 5.4), the validation accuracy in such a case would be 50%, since the dataset contains only two classes.

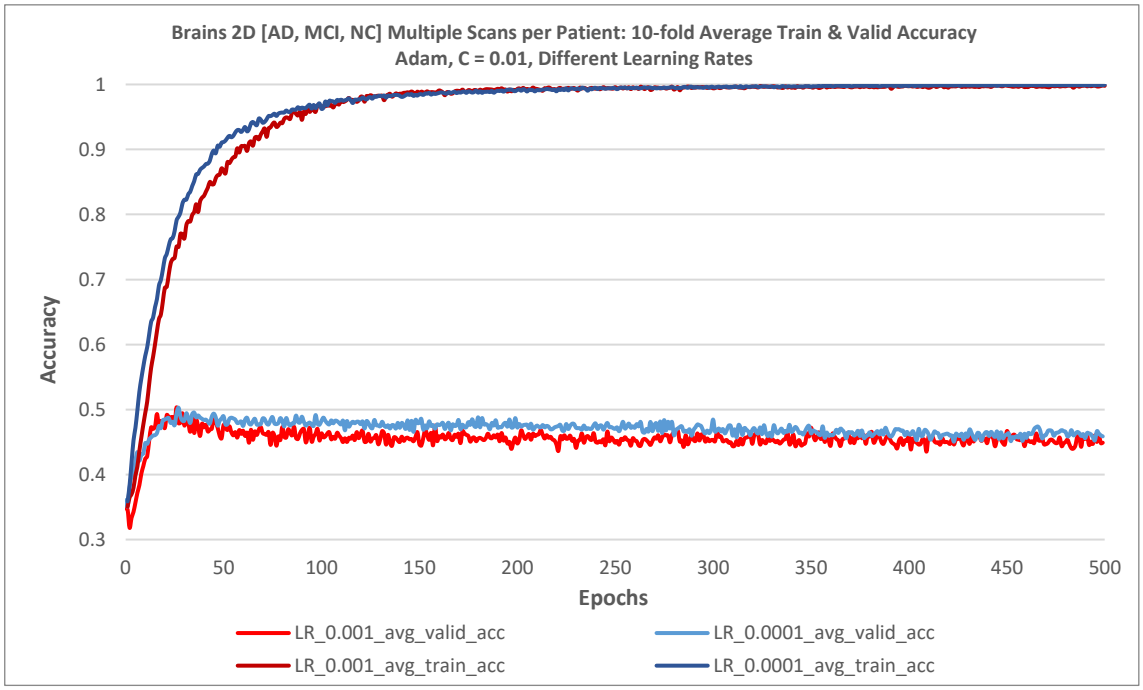


Figure 5.117. Average training and validation accuracy of the experiments B_2D_M_A {1, 2} [AD, MCI, NC].

Figure 5.117, shows the average training and validation accuracies of the 10-folds, of the experiments B_2D_M_A {1, 2} [AD, MCI, NC]. The average validation accuracy of the B_2D_M_A1 [AD, MCI, NC] with a learning rate = 0.001 (light red), overall seems to be slightly worse than the experiment's B_2D_M_A2 [AD, MCI, NC] (light blue).

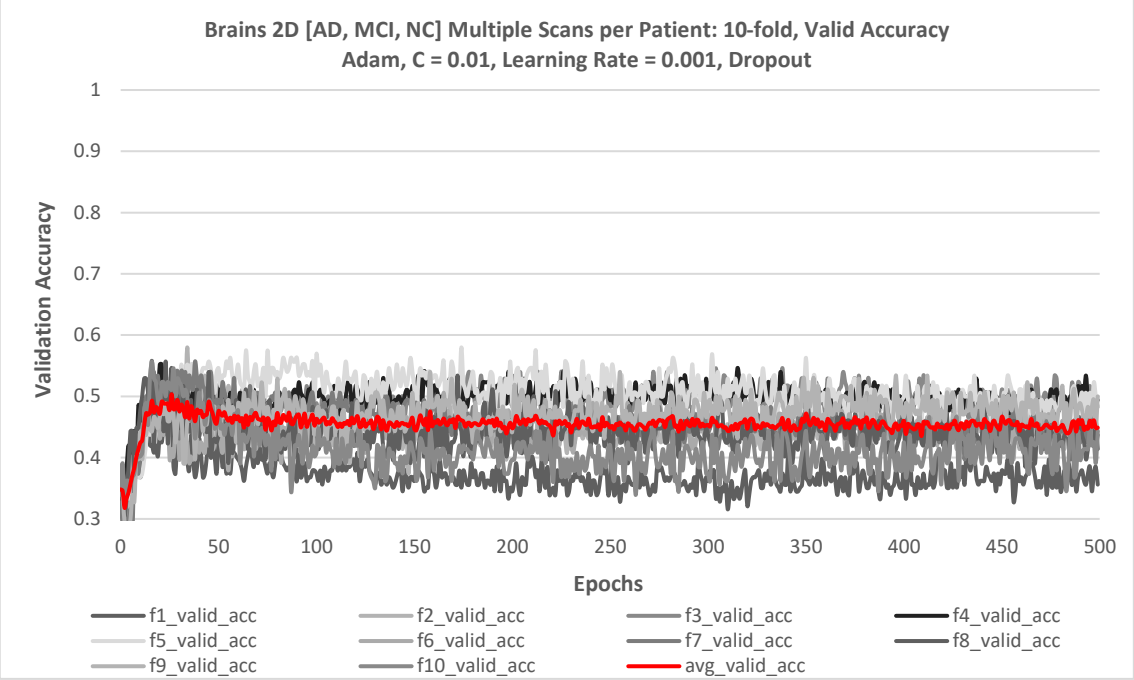


Figure 5.118. Validation accuracy per fold of the experiment B_2D_M_A1 [AD, MCI, NC].

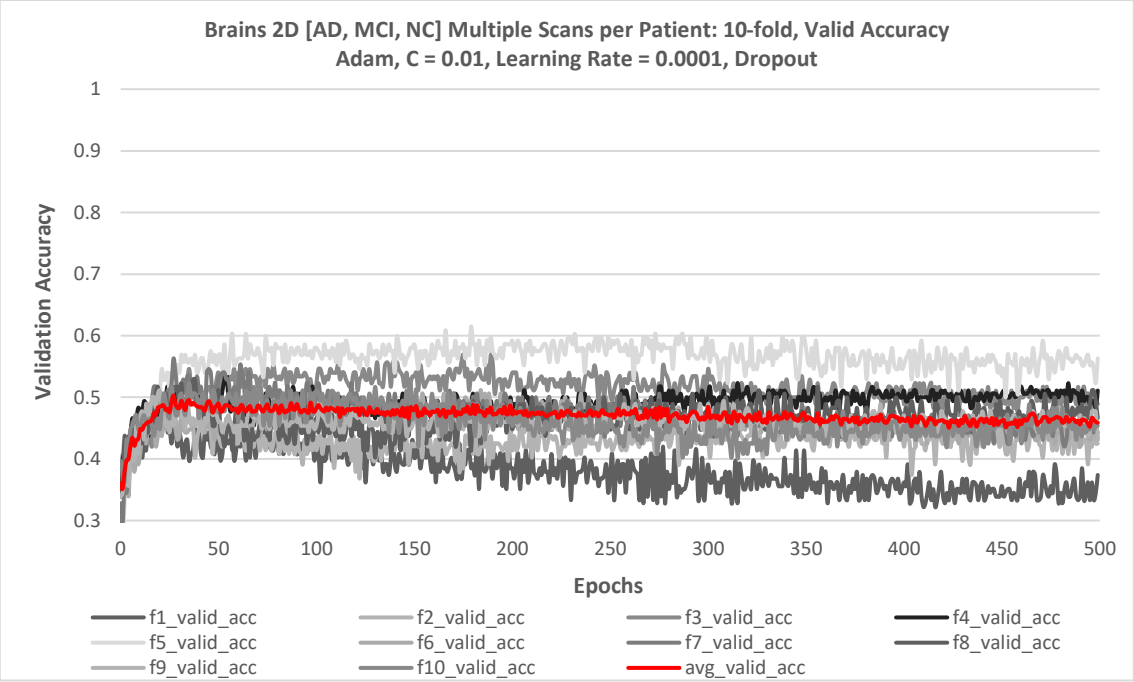


Figure 5.119. Validation accuracy per fold of the experiment B_2D_M_A2 [AD, MCI, NC].

In Figure 5.118 & 5.119, we can see the validation accuracies per fold, of the experiments B_2D_M_A {1, 2} [AD, MCI, NC] respectively. In both experiments, the 1st fold seems to perform worse than the rest of them, since it approaches accuracies near 33%. This potentially indicates that the 1st fold may contain many outliers.

Figure 5.120, shows the average training and validation losses per experiment. Both models overfit, since their validation loss increase while their training loss decrease; with the experiment B_2D_M_A1 [AD, MCI, NC] with the learning rate = 0.001 (red) overfitting more than the experiment B_2D_M_A2 [AD, MCI, NC] with the learning, rate = 0.0001 (blue).

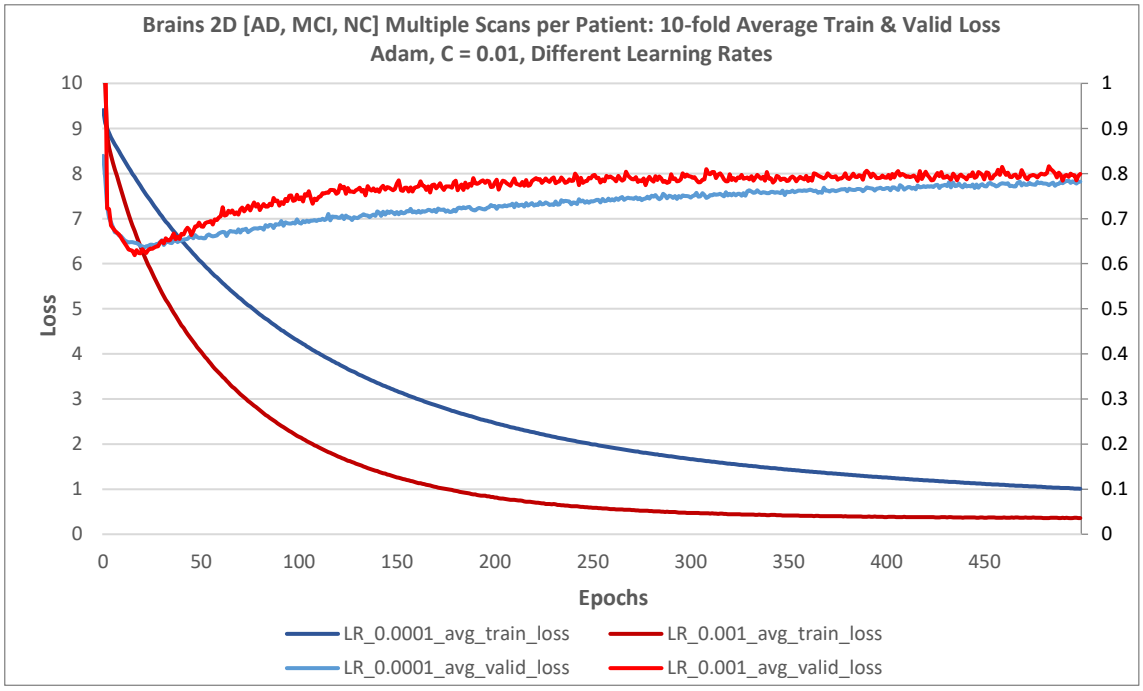


Figure 5.120. Average training and validation loss of the experiments B_2D_M_A {1, 2} [AD, MCI, NC].

Figure 5.121 & 5.122, show the validation loss per fold for the experiments B_2D_M_A {1, 2} [AD, MCI, NC] respectively. We can see that in both cases, all ten folds suffer from overfitting and are having very similar behavior. Therefore, most probably the issue behind overfitting is no the model itself, but the dataset. The size of the dataset is very small, with the training being only 1566 samples (Section 3.2.4.1). Since obtaining easily more data is not feasible, data augmentation (Section 2.3.13.5) should be applied to increase the size of the dataset by at least 50 times. This will potentially resolve the overfitting issues, help the model to improve its performance, and provide more accurate performance metrics.

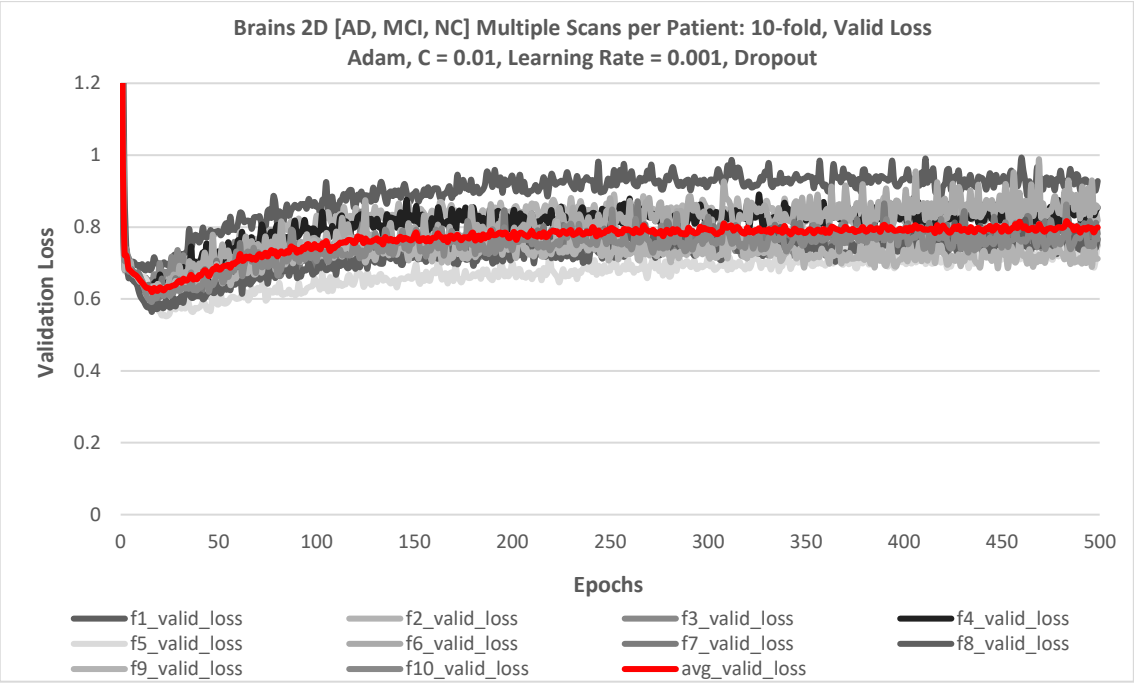


Figure 5.121. Validation loss per fold of the experiment B_2D_M_A1 [AD, MCI, NC].

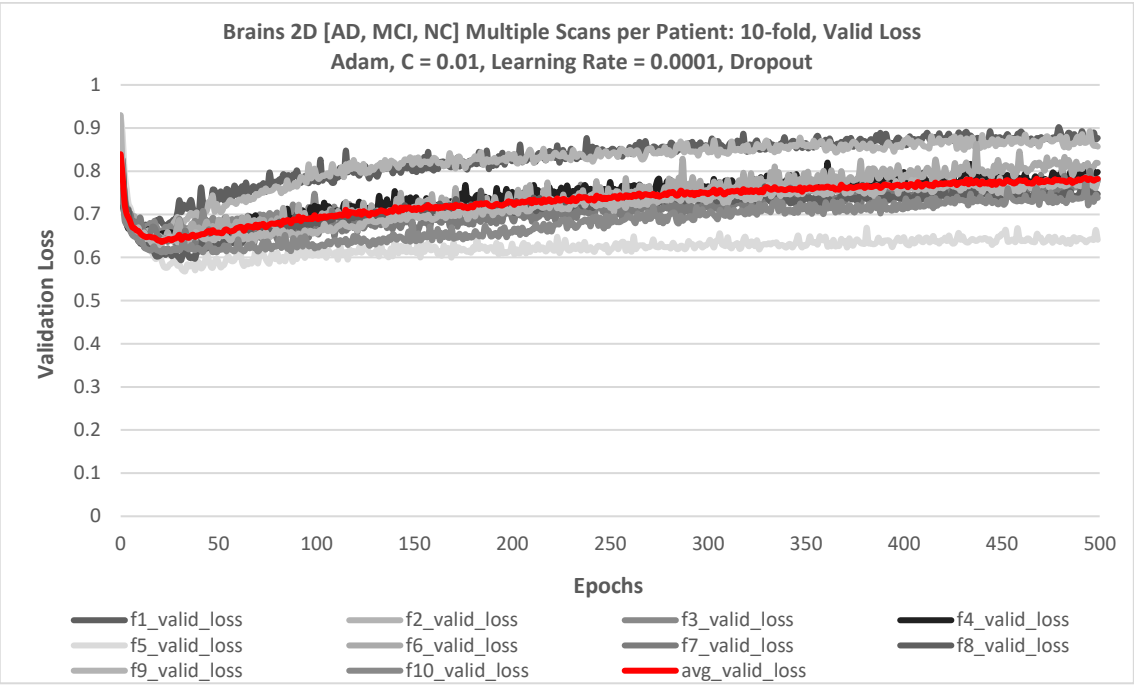


Figure 5.122. Validation loss per fold of the experiment B_2D_M_A2 [AD, MCI, NC].

5.11.2 CNN 4 Layers with NewtonCG

5.11.2.1 Experiment B_2D_M_N1 [AD, MCI, NC]: C = 0.01, GNsize = 50, Dropout

The experiment B_2D_M_N1 [AD, MCI, NC] uses the NewtonCG optimizer with the 2D_CNN_4L_1 (Section 4.3.2.3.1) network configuration. The model applies standard

Dropout (Section 2.3.13.9), d1 = d2 = 32, and d3 = 64, where di is the drop probability in the i^{th} Convolutional Layer. Basically, the experiment uses the same network configuration and the same hyperparameters (C = 0.01; GNsize = 50) of the experiment B_2D_M_N10 (Appendix H.2.10). The source code for the network is available in Appendix B.5.2, at Code Snippet B.6.

In Table 5.29, we can see that the total average validation accuracy of the 10-folds for the experiment B_2D_M_N1 [AD, MCI, NC] with the NewtonCG optimizer is 55%, while in the best Adam experiment, the B_2D_M_A1 [AD, MCI, NC], is 54%. Also, the average validation accuracies for each target class separately, the AD, MCI, and NC are 71%, 67%, and 71% respectively, very similar to the ones of the best Adam experiment (B_2D_M_A1 [AD, MCI, NC]) which are 71%, 68%, and 71% (Table 5.29).

Experiments: B_2D_M_A1 [AD, MCI, NC] vs. B_2D_M_N1 [AD, MCI, NC]																				
						Training				Validation					Testing					
Experiment Name	Optimizer	C	GNsize	Max Epochs	Learning Rate	Avg. Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. Accuracy	Best Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. Accuracy	Avg. Accuracy AD	Avg. Accuracy MCI	Avg. Accuracy NC	Avg. epoch of best Valid Acc. per fold	
B_2D_M_A1 [AD, MCI, NC]	Adam	0.01	-	500	0.001	0.83	0.89	0.87	0.89	0.54	0.58	0.71	0.68	0.71	0.35	0.63	0.51	0.56	26	
B_2D_M_N1 [AD, MCI, NC]	NewtonCG	0.01	50	100	-	0.78	0.88	0.83	0.85	0.55	0.60	0.71	0.67	0.71	0.33	0.59	0.49	0.58	23	

Table 5.29. Performance metrics and hyperparameters of the experiments B_2D_M_A1 [AD, MCI, NC] and B_2D_M_N1 [AD, MCI, NC].

Figure 5.123, shows the average training and validation accuracies of the experiment B_2D_M_A1 [AD, MCI, NC]. Also, it shows the average training and validation accuracies per target class, AD, MCI, and NC. Based on Figure 5.123, we understand that the model struggles to generalize, since at the beginning the average validation accuracy is 40%, and after 100 epochs is close to 46%. On the other hand, the training accuracy keeps rising where it reaches a value near 100% after 100 epochs.

In Figure 5.13, we can see three extra lines for validation and training accuracies which correspond to the accuracies per target class; AD, MCI, and NC. They are larger than the average accuracies of all three classes together because they have been calculated by using the following equation where t corresponds to each target class {AD, MCI, NC}:

$$ACC_t = \frac{TP_t + TN_t}{TP_t + FP_t + FN_t + TN_t}$$

The TP_t represents how many samples of the t class were correctly identified as t, and the TN_t represents how many non t samples were correctly identified to their class. Those values can be found in a 3×3 confusion matrix by using the method in Figure 3.18. In Figure 5.124, with the average training and validation losses for the experiment, we can see that the model is overfitting since after 30 epochs the average validation loss (light red) rises while the average training loss (dark red) declines.

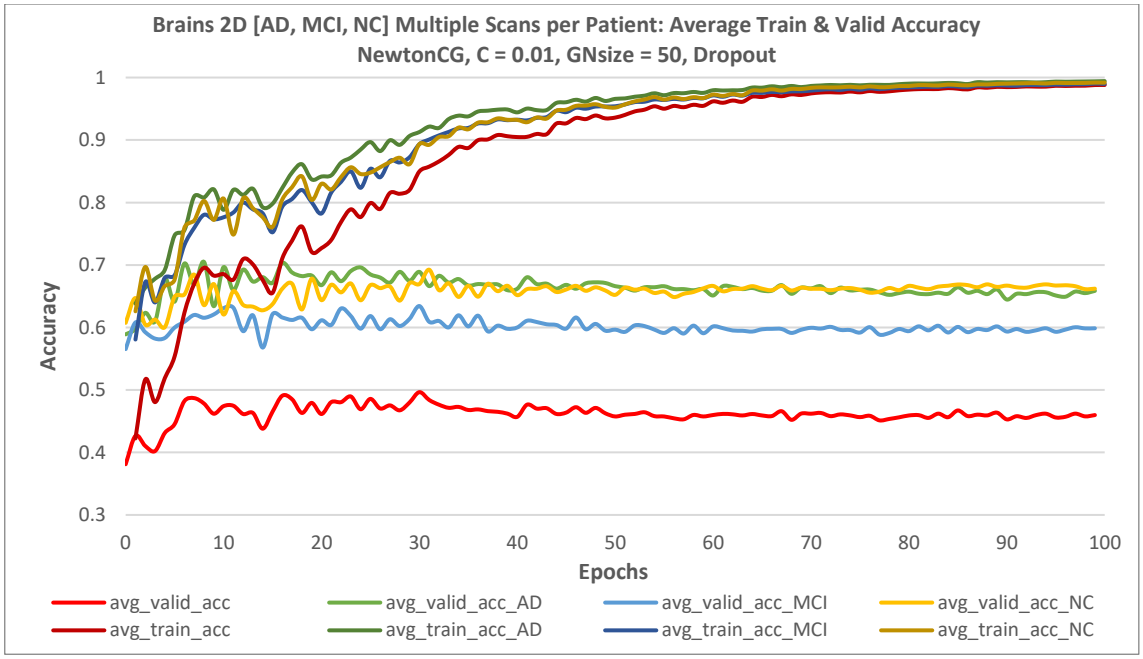


Figure 5.123. Training and validation accuracy per target class of the experiments B_2D_M_A1 [AD, MCI, NC].

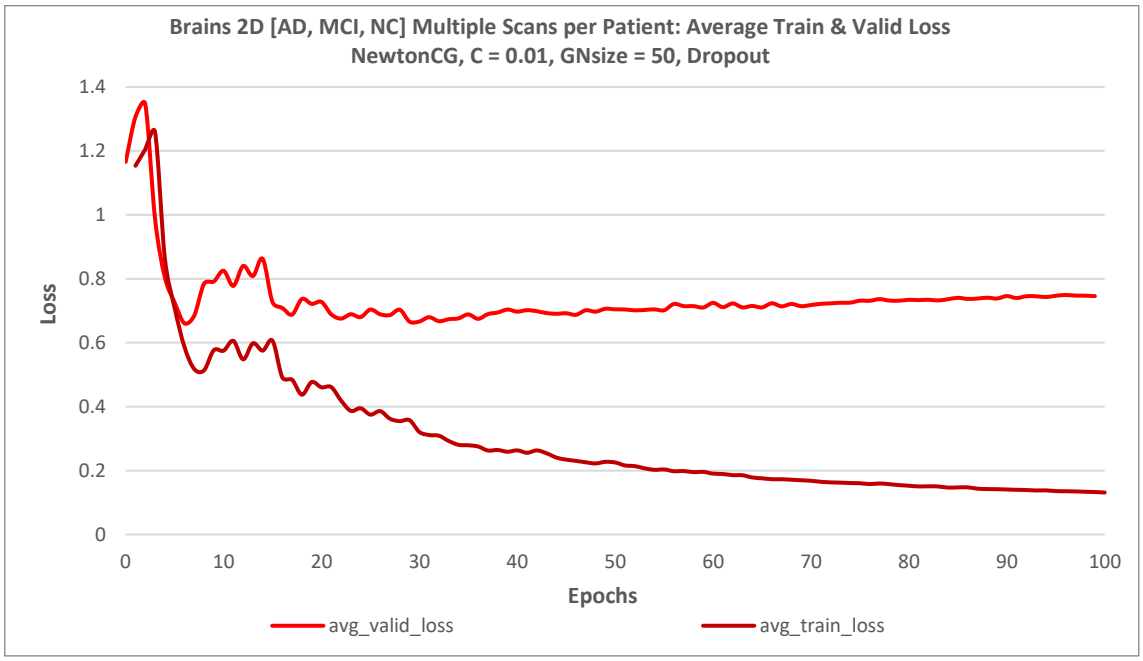


Figure 5.124. Average training and validation loss of the experiments B_2D_M_A1 [AD, MCI, NC].

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In general, 5 main types of data were used in more than 80 experiments for the AD/NC and AD/MCI/NC problems. Those types of data are 2D slices of T1-weighted MRI scans ("B_2D"; Section 3.2.3), 3D shrunk subject level T1-weighted MRI scans ("B_3D_S"; Section 3.2.5), 3D images of the patients left hippocampus isolated ("LH_3D_S"; Section 3.2.6), 3D cropped images of the original T1-weighted MRI scans in the area where the left hippocampus is ("CB_3D_S"; Section 3.2.7), and a dataset with the features of the hippocampus as extracted in the Achilleos et al. (2020) paper ("HF_M"; Section 3.2.2).

From the 2D slice-level experiments, 4 distinct datasets for the AD/NC problem have been created. The datasets differ based on the number of scans used per patient, and the number of slices used per T1-weighted MRI scan. More specifically, the 4 datasets are Single Scan per Patient – Single Slice per Scan ("B_2D_S"; Section 3.2.3.2), Single Scan per Patient – 5 Slices per Scan ("B_2D_5S"; Section 3.2.3.3), Multiple Scans per Patient – Single Slice per Scan ("B_2D_M"; Section 3.2.3.4), and Multiple Scans per Patient – 7 Slices per Scan ("B_2D_7M"; Section 3.2.3.5).

For the AD/MCI/NC problem, the dataset 2D Brain Slices, Multiple Scans per Patient – Single Slice per Scan (B_2D_M) was used that included also an equal number of MCI patients as ADs and NCs (B_2D_M [AD, MCI, NC]). Therefore, in this thesis, 9 unique datasets have been used for the experiments, alongside the 5-fold Cross-validation or 10-fold Cross-validation (Section 3.1) to measure the performance of NewtonCG (HFO) against Adam or SGD.

Best NewtonCG (HFO), Adam, SGD experiments per Dataset																				
													Training			Validation				
Dataset	Experiment Name	Optimizer	C	GNsize	Max Epochs	Learning Rate	α (alpha)	CG iterations	Dropout	Feature Maps	Filters / Kernels Size	Network Architecture	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
Multi-Layer Perceptron (MLP)																				
HF_M	HF_M_S5	SGD	-	-	500	0.3	0.1	-	-	-	-	[10, 8, 8, 1]	0.94	0.92	0.95	0.88	0.82	0.95	272	0.93
	HF_M_A11	Adam	-	-	500	0.3	0.1	-	-	-	-	[10, 30, 1]	0.88	0.87	0.89	0.90	0.83	0.96	155	0.93
	HF_M_H2	HFO	-	-	100	-	-	2	-	-	-	[10, 30, 1]	0.93	0.82	0.97	0.87	0.76	0.98	23	0.93
Convolutional Neural Networks (CNN)																				
B_2D_M	-	SGD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	B_2D_M_A2	Adam	0.01	-	500	0.01		-	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3	2D_CNN_4L_1	0.87	0.87	0.86	0.79	0.78	0.80	246	0.90
	B_2D_M_N19	NewtonCG	0.01	50	100	-	-	250	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	5 x 5	2D_CNN_4L_2	0.82	0.83	0.80	0.81	0.80	0.81	17	0.90
B_2D_7M	-	SGD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	B_2D_7M_A1	Adam	0.01	-	500	0.001	-	-	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3	2D_CNN_4L_1	0.92	0.93	0.91	0.75	0.76	0.74	49	0.78
	B_2D_7M_N1	NewtonCG	0.01	50	100	-	-	250	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3	2D_CNN_4L_1	0.94	0.94	0.94	0.76	0.77	0.75	32	0.78
B_2D_S	-	SGD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	-	Adam	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	B_2D_S_N3	NewtonCG	1	5	100	-	-	250	0.5	f1 = 32, f2 = 64 f3 = 128	3 x 3	2D_CNN_4L_6	0.85	0.85	0.86	0.77	0.75	0.78	23	0.84
M_2D_5S	-	SGD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	-	Adam	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	B_2D_5S_N1	NewtonCG	0.01	50	100	-	-	250	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3	2D_CNN_4L_1	0.92	0.90	0.95	0.75	0.73	0.77	31	0.78
LH_3D_S	-	SGD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	LH_3D_S_A4	Adam	0.01	-	500	0.0001	-	-	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3 x 3	3D_CNN_4L_2	1.00	1.00	1.00	0.77	0.86	0.67	209	0.83
	LH_3D_S_N2	NewtonCG	0.01	50	100	-	-	250	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3 x 3	3D_CNN_4L_2	0.86	0.87	0.84	0.73	0.78	0.68	21	0.83
B_3D_S	-	SGD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	B_3D_S_A1	Adam	0.01	-	500	0.0001	-	-	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3 x 3	3D_CNN_4L_1	0.83	0.80	0.86	0.75	0.69	0.82	131	0.81
	B_3D_S_N1	NewtonCG	0.01	200	100	-	-	250	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3 x 3	3D_CNN_4L_1	0.72	0.80	0.64	0.64	0.79	0.50	6	0.68
CB_3D_S	-	SGD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CB_3D_S_A1	Adam	0.01	-	500	0.0001	-	-	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3 x 3	3D_CNN_4L_1	0.98	1.00	0.96	0.73	0.77	0.69	71	0.76
	CB_3D_S_N1	NewtonCG	0.01	50	100	-	-	250	d1 = 0.3 d2, d3 = 0.5	f1, f2 = 32 f3 = 64	3 x 3 x 3	3D_CNN_4L_1	0.76	0.77	0.75	0.71	0.72	0.71	13	0.76

Table 6.1. Best NewtonCG (HFO), Adam, SGD experiments per Dataset

The best validation accuracies for each optimizer and each one of the 8 datasets for the AD/NC problem can be seen in Table 6.1. Based on the performance metrics, and more specifically, the average validation accuracy, the best combination of dataset and optimizer, is the HF_M dataset with the Adam optimizer; which yields an average

validation accuracy equal to 90% (HF_M_A11). The highest average validation accuracy in the rest of the experiments with CNNs is equal to 81% and it comes from the combination of the dataset B_2D_M with the NewtonCG optimizer (B_2D_M_N19).

The sensitivity of the experiment HF_M_A11, which for the HH_M dataset is the ratio of the correctly identified ADs from the total number of ADs, is 83%. For the rest of the experiments in CNNs, the specificity is the ratio of the correctly identified ADs from the total number of ADs, and the highest comes from the B_2D_M_N19 experiment as well, which is equal to 81%. Therefore, the Hippocampus Features (HF_M) with MLP (Adam), manage to correctly identify 83% of ADs, while the 2D slice-level MRI scans of the brains (B_2D_M) with CNN (NewtonCG), manage to correctly identify the 81% of ADs. Based on Achilleos et al. (2020), their best model which uses Argumentation Rules with the Hippocampus Features dataset (HF_M), the percentage of correctly identified ADs is 87%. Thus, if we take into consideration, that the procedure of obtaining the 2D slices of the MRI scans is much simpler than collecting the hippocampal features, our performance metrics are respectable.

Generally, most experiments suffer from overfitting, especially the experiments with the NewtonCG optimizer. This can be easily observed in the average loss plots of the experiments, where the average validation loss at some point starts increasing while the average training loss keeps decreasing. Data augmentation can be used, increase the size of the dataset, and therefore help the models to generalize better.

The NewtonCG optimizer is very aggressive and in most of the experiments, only the hyperparameters C and GNsize were modified. Other hyperparameters such as the eta (η), xi (ξ), lambda (λ), boost, drop, and CG maximum iterations could help to encounter the problem of overfitting (Section 4.3.1). Unfortunately, due to the tight schedule and lack of resources, it was not feasible to test all these hyperparameters and their effects for all the experiments. It seems that eta (η) plays an important role in how aggressive the NewtonCG optimizer is, based on some observations in experiments with the 3D Left Hippocampus dataset. These experiments were incomplete, and therefore, have not been included in this thesis; so, further experimentation is needed to tell for sure.

Many network configurations were examined such as narrow, wide, shallow, and deep CNNs (Section 4.3), with a different number of Convolutional Layers and dense layers

as well. Additionally, experiments with or without the application of Dropout, Spatial Dropout, L1 & L2 Regularization, Batch Normalization were held (Table P.1 & P.2, Appendix P). The absence of Max-Pooling showed significant improvement against the overfitting issue in the 3D Left Hippocampus dataset (Section 5.8.1.2 & 5.8.2.1) but not in the 2D ones (Section 5.4.2.6). Batch Normalization (Section 5.4.2.9) could potentially help against overfitting is if it is applied after each Convolutional Layer and not just in the last Convolutional Layer.

In terms of filter sizes, 5×5 works the best for 2D slice-level MRI scans (Section 5.4.2.10). For the 3D subject-level datasets, we cannot say for sure whether the filter size $5 \times 5 \times 5$ is a better choice, since all the experiments that have been performed, used filter sizes of $3 \times 3 \times 3$.

In general, based on my experience, no 10-folds should be used because the datasets are relatively small, so 5-fold is a better choice for future experiments. Moreover, the test set should not be excluded before the split of the data into folds. The validation set is enough for representing both the validation and test set. This way, more data could be used for training and since the test sets were too small (~4%), their results were not representative of the models' actual performance (Section 3.2).

The NewtonCG method, as expected, converges faster than Adam. The average epoch of the best validation accuracy per model is approximately 20 for the NewtonCG optimizer while it is 200 for the Adam (Table 5.7). This means that in order to achieve similar average validation accuracies between models with the NewtonCG and the Adam optimizer, Adam needs 10 times more epochs approximately than the NewtonCG.

6.2 Future Work

The main focus in future work should be on how to improve the performance of the 3D experiments. This implies that more computational resources are needed than those we had during this thesis, to be able to execute them since, for the 3D CNNs, dedicated GPUs are needed. Arcadia server which provides only multiple CPU cores is unable to execute them. It is highly suggested to use the MATLAB implementation instead of the one in Python for the CNNs with the NewtonCG optimizer since the authors (Wang et al. 2020)

mention in their Python comments, that are not sure whether the regularization was implemented correctly.

First, the size of the current datasets needs to increase, and 5-fold instead of 10-fold should be used. Data augmentation needs to be applied in the images, such as rotation, flipping, transition, shearing, scaling, change of image's brightness, and addition of noise (Section 2.3.13.5). This way, the size of the dataset will increase to 20,000 – 70,000. Then, I suggest, without extracting a test set as we did in this thesis, a 5-fold Cross-validation with StratifiedGroupKFold should be applied as described in Section 3.1.5, in order to avoid data leakage.

In this thesis, from the experiments in CNNs, the B_2D_M_N19 with the B_2D_M dataset had the highest average validation accuracy (81%), but I believe that the B_2D_7M, with data augmentation, could be a better dataset for the AD/NC problem. The B_2D_7M dataset has the smallest standard deviation (1% – 2%) between the best validation accuracies of each fold (Table 5.7). Additionally, based on the results of Section 5.7.2.1, a 5-layer CNN, or even deeper network configuration should be used with a larger dataset such as the B_2D_7M.

Based on my observations (Section 5.4.2.10), for the 2D slice MRI scans datasets, 5×5, filter size should be used, since it performs better than 3×3, or 7×7. For the 3D subject-level MRI scans datasets, experiments with only $3 \times 3 \times 3$ filter size were held, so further examination is needed for different sizes. In terms of network configurations, for the 3D experiments, is highly recommended to remove the Max-Pooling from all Convolutional Layers (Section 5.8.1.2; Section 5.8.2.1; Table 5.92). The absence of Max-Pooling improved the validation accuracy usually or even fixed underfitting issues with some experiments in 3D CNNs. Keep in mind that the removal of Max-Pooling caused overfitting in the 3D Left Hippocampus experiments (Section 5.8.1.2; Section 5.8.2.1), where the average training accuracy went fast towards 100%, while the average validation accuracy was still 77%.

For the 3D experiments, smaller eta (η) could potentially reduce the aggressiveness of the NewtonCG optimizer. Also, a smaller GNsize seems to reduce overfitting, while Wang et al. (2020) showed that larger GNsize are better.

Different network topologies should be tried, for example, the wider 4-layer CNN network 2D_CNN_4L_6 (Section 4.3.2.3.6) in the B_2D_S_N3 experiment (Section 5.5.1.2) seemed to be performing better than the baseline 4-layer CNN network 2D_CNN_4L_1 (Section 4.3.2.3.1). The number of feature maps fi, of the i^{th} Convolutional Layer of the 2D_CNN_4L_6 network, were f1 = 32, f2 = 64, and f3 = 128, while for the 2D_CNN_4L_1 network, the filters were f1 = f2 = 32, and f3 = 64.

Another experiment that could be performed in the future, is a multi-input Keras model. More specifically, two or even three inputs can be provided to a network with a single output. For example, input A could take numeric values, such as the hippocampal features extracted by Achilleos et al. (2020), in an MLP; the input B could be fMRI images in a CNN, and the input C the T1-weighted MRI scans in another CNN. All the inputs have to be referring to the same patient which is something feasible since the ADNI dataset provides all this information. An example of such implementation with multiple inputs can be found here or in Keras API: (https://www.pyimagesearch.com/2019/02/04/keras-multiple-inputs-and-mixed-data/). A multi-input implementation has been proven by Venugopalan et al. (2021), a better approach than a single input, both for the AD/NC and the AD/MCI/NC problem.

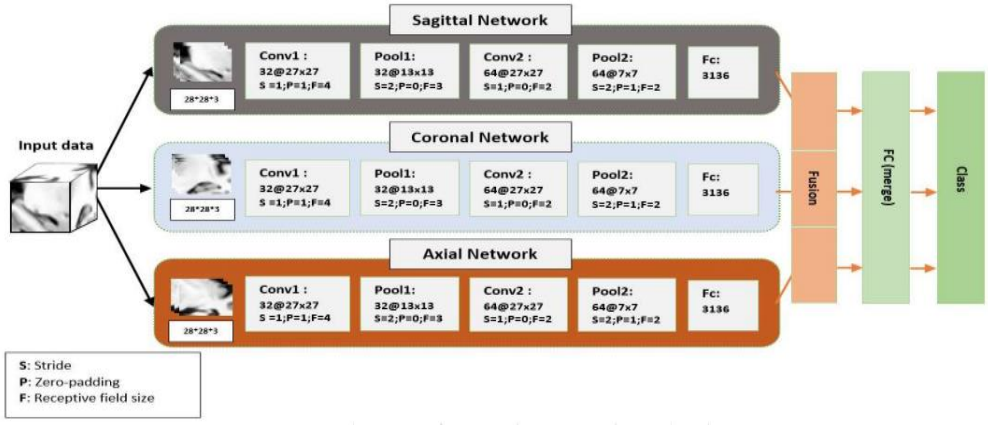


Figure 6.1. Multi-level network. Three 3D CNN for the sagittal, coronal, and horizontal slices. (Aderghal et al. 2017)

An additional experiment of a multi-input Keras model could be to use three CNNs, one for each perspective of the same brain scan, the Sagittal, Coronal, and Horizontal slices (Figure 2.4). This could be a convenient alternative instead of using the whole 3D T1-weighted MRI scan, to save computational power and reduce the execution times while providing different perspectives of the patient's brain. This method was tried by Aderghal et al. (2017), and its results were very promising with validation accuracy equal to 91.4%

for the AD/NC problem (Figure 6.1). Since in this thesis only T1-weighted MRIs were used, in the future T2-weighted MRIs could be used as well.

A piece of advice for future work is that batch normalization (Section 2.3.13.11) should be applied before each Convolutional Layer. Batch normalization has the potential to improve the performance of the model. In this thesis, batch normalization was only applied after the last Convolutional Layer which it was turned out not to be the best practice (Ioffe and Szegedy, 2015).

Different activation functions, other than ReLU could be applied, such as Leaky ReLU. Sigmoid is not suggested as an activation function, since the models with CNNs did not perform as well as with ReLU in the experiments of this thesis.

To fight against overfitting, the plots of the Receiver Operating Characteristic curve (ROC) and the Area Under the ROC Curve (AUC) may help. ROC compares the True Positive Rate (TPR) with the False Positive Rate (FPR). The curve of the model which is the closest to the left top corner has the highest sensitivity, and it is the best model for binary classification. AUC is the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance. Usually, ROC and AUC matrices are being used in medical studies for the evaluation of the different techniques (Choi and Jin, 2018; Achilleos et al. 2020).

In this thesis, most datasets were focused on the AD/NC problem. In a real-world scenario, a patient could have many different types of dementia and not just AD. For example, a patient could have Parkinson's, Vascular Dementia, MCI, and other different syndromes. Therefore, the CNN models should be trained to detect multiple diseases for the model to be applicable for public use.

In addition, transfer learning is suggested for future work since for this medical problem a limited number of labeled data exists, thus a CNN cannot easily be trained. This should not be an issue for the Gradient Descent implementations (e.g., SGD, Adam), since available pre-trained models for both 2D and 3D CNNs exist (Hosseini-Asl et al. 2018). For the CNN with HFO implementations, no pre-trained models were found, but in future work, a pre-trained model could be created. For example, you can initially train the Convolutional Layers with the CIFAR-10 dataset and then train furthermore the model with the MRI scans, to train the FFNN as well.

The next step would be to use MRI scans that are registered to standard brain space (spatial normalization) to get the input to the model. Spatial normalization is a geometrics transformation that aligns a brain scan to a standard brain image. This will solve the issues that we faced in which the different structures of the brain corresponded to different coordinates in each 3D scan and in some 2D scans the spinal cord and the cerebellum were visible, while in others were not. From these spatially normalized scans, different data can be extracted and used as input: 3D MRI of the whole brain, 3D MRI of the isolated hippocampus and/or entorhinal cortex, and 2D MRI slices which contain the hippocampus. It is expected that using the hippocampus and entorhinal cortex as the input to the model will give better results because it has already been found that the features of these structures are affected by MCI and AD (Leandrou et al. 2020; Achilleos et al. 2020).

Also, the datasets with multiple scans per patient in this thesis could contain 1 to 11 scans per patient. It would be more appropriate to create and compare datasets that contain only the baseline scan, or the baseline and 6-month scans, etc. This approach has been used in several studies (Suk and Shen, 2013; Suk et al. 2015; Choi and Jin, 2018; Achilleos et al. 2020).

Finally, we need to consider that no collaboration with a medical expert took place during this thesis. Therefore, some decisions that have been made during the creation of the datasets, that have been proven later not to be optimal, could be avoided. The state-of-the-art algorithms, such as the HFO with the CNNs are available, thus, a collaboration with a medical expert is essential for using state-of-the-art techniques of the medical field as well.

References

Achilleos, K., G., Leandrou, S., Prentzas, N., Kyriacou, P., A., Kakas A., C., & Pattichis, C., S., "Extracting Explainable Assessments of Alzheimer's disease via Machine Learning on brain MRI imaging data". *2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (BIBE)*, pp. 1036–1041, (2020), doi: 10.1109/bibe50027.2020.00175.

Aderghal, K., Benois-Pineau, J., Afdel, K., & Catheline, G., "FuseMe: classification of sMRI images by fusion of deep CNNs in 2D+ϵ projections", *Proceedings of the 15th International Workshop on Content-Based Multimedia Indexing*, no. 34, pp. 1–7, (2017)

Adrian, E.& Zotterman, Y., "The impulses produced by sensory nerve-endings: Part 2: The response of a single end-organ", *The Journal of physiology*, J. Physiol. (Lond.), vol. 61, pp. 151–171. (1926)

Ahmed, O., Benois-Pineau, J., Allard, M., Catheline, G., & Amar, C. "Recognition of Alzheimer's disease and Mild Cognitive Impairment with multimodal image-derived biomarkers and Multiple Kernel Learning". *Neurocomputing*, vol. 220, pp. 98–110, (2017), doi: 10.1016/j.neucom.2016.08.041.

Allison, J., Rivers, R., Christodoulou, J., Vendruscolo, M., & Dobson, C., "A relationship between the transient structure in the monomeric state and the aggregation propensities of α-synuclein and β-synuclein", *Biochemistry*, vol. 53, pp. 7170–7183, (2014)

Alzheimer Europe, "Dementia in Europe Yearbook 2019. Estimating the prevalence of dementia in Europe", *Luxemburgo: Alzheimer Europe*, (2020)

Andrew, N., "Feature selection, L 1 vs. L 2 regularization, and rotational invariance", *Proceedings of the twenty-first international conference on Machine learning*, pp. 78, (2004)

Arvanitakis, Z., Shah, R. & Bennett, D., "Diagnosis and Management of Dementia: Review", *JAMA* vol. 322, pp. 1589–1599, (2019), doi: 10.1001/jama.2019.4782

Beleites, C., Neugebauer, U., Bocklitz, T., Krafft, C., Popp, J., “Sample size planning for classification models”. *Analytica Chimica Acta*, vol. 760, pp. 25–33, ISSN 0003-2670, (2013), doi: 10.1016/j.aca.2012.11.007

Bender, M., Postel, D. & Krieger, H., “Disorders of oculomotor function in lesions of the occipital lobe”, *Journal of Neurology, Neurosurgery & Psychiatry*, vol. 20, pp. 139–143, (1957)

Braak, H. and Braak, E., “Neuropathological staging of alzheimer-related changes”, *Acta Neuropathologica*, vol. 82, pp. 239–259, (1991)

Bengio, Y., Courville, A. & Vincent, P., “Representation Learning: A Review and New Perspectives”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, pp. 1798–1828, (2013), doi: 10.1109/tpami.2013.50

Brownsett, S. & Wise, R., “The contribution of the parietal lobes to speaking and writing”, *Cerebral cortex (New York, N.Y. : 1991)*, vol. 20, pp. 517–523, (2010), doi: 10.1093/cercor/bhp120

Caruana, R., Lawrence, S. & Giles, C., “Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping”, In: Leen, T., Dietterich, T., & Tresp, V. (eds.), MIT Press (Cambridge, MA, United States), *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*, vol. 13, pp. 402–408. (2001)

Chandra, A., Dervenoulas, G. & Politis, M., “Magnetic resonance imaging in Alzheimer’s disease and mild cognitive impairment”. *Journal of neurology*, vol. 266, pp. 1293–1302, (2019)

Charalambous, K., Agathocleous, M., Christodoulou, C., & Promponas, V., “Solving the protein secondary structure prediction problem with the hessian free optimization algorithm”, *IEEE Access*, (2020), Under review.

Choi, H. & Jin, K., “Predicting cognitive decline with deep learning of brain metabolism and amyloid imaging”, *Behavioural brain research*, vol. 344, pp. 103–109, (2018), doi: 10.1016/j.bbr.2018.02.017

Clifford, J., Bernstein, M., Fox, N., Thompson, P., Alexander, G., Harvey, D., Borowski, B., Britson, P., Whitwell, J., Ward, C., Dale, A., Felmlee, J., Gunter, J., Hill, D., Killiany, R., Schuff, N., FoxBosetti, S., Lin, C., Studholme, C., DeCarli, C., Krueger, G., Ward, H., Metzger, G., Scott, K., Mallozzi, R., Blezek, D., Levy, J., Debbins, J., Fleisher, A., Albert, M., Green, R., Bartzokis, G., Glover, G., Mugler, J. & Weiner, M., "The Alzheimer's Disease Neuroimaging Initiative (ADNI): MRI methods", *Journal of magnetic resonance imaging: JMRI*, vol. 27, pp. 685–691, (2008), doi: 10.1002/jmri.21049

Csáji, B. & et al."Approximation with artificial neural networks", *Faculty of Sciences, Etvs Lornd University, Hungary*, vol. 42, pp. 7, (2001)

Duchi, J., Hazan, E., & Singer, Y., "Adaptive subgradient methods for online learning and stochastic optimization", *The Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, (2011).

Eldan, R & Shamir, O., "The power of depth for feedforward neural networks", *In Conference on learning theory*, pp 907–940, (2016)

Frisoni, G., Fox, N., Jack, C., Scheltens, P. & Thompson, P., "The clinical use of structural MRI in Alzheimer disease", *Nature reviews. Neurology*, vol. 6, pp. 67v77, (2010), doi: 10.1038/nrneurol.2009.215

Gavin, H., P., "The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems". *Department of Civil and Environmental Engineering, Duke University*, pp. 1-19 (2019)

Gauthier, S., Reisberg, B., Zaudig, M., Petersen, R., Ritchie, K., Broich, K., Belleville, S., Brodaty, H., Bennett, D., Chertkow, H., Cummings, J., Leon, M., Feldman, H., Ganguli, M., Hampel, H., Scheltens, P., Tierney, M., Whitehouse, P. &Winblad, B., "Mild cognitive impairment". *International Psychogeriatric Association Expert Conference on mild cognitive impairment*, vol. 367, pp. 1262–1270, (2006), doi: 10.1016/s0140-6736(06)68542-5

Georgopoulos, A., Kalaska, J., Caminiti, R. & Massey, J., "On the relations between the direction of two-dimensional arm movements and cell discharge in primate motor cortex", *Journal of neurophysiology*, vol. 2, pp. 1527–1537, (1982)

Gerardin, E., “Morphometry of the human hippocampus from MRI and conventional MRI high field”. *Université Paris Sud - Paris XI, 2012. English*, NNT: 2012PA112375, HAL ID: tel-00856589, (2012)

Glorot, X., Bordes, A., & Bengio, Y., “Deep sparse rectifier neural networks”, *Proceedings of the fourteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings*. vol. 5, pp. 315–323, (2011)

Henry, G., Dreher, B. & Bishop, P., “Orientation specificity of cells in cat striate cortex”, *Journal of neurophysiology*, vol. 37, pp. 1394–1409, (1974)

Herculano-Houzel, S., “The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost”, *Proceedings of the National Academy of Sciences*, vol. 109, pp. 10661–10668, (2012), doi: 10.1073/pnas.1201895109

Hinton, G., E., Srivastava, N., & Swersky, K., “Lecture 6a overview of mini-batch gradient descent.” *Coursera Lecture slides https://class.coursera.org/neuralnets-2012-001/lecture,[Online]*. (2012)

Hoffmann, M., “The human frontal lobes and frontal network systems: an evolutionary, clinical, and treatment perspective”, *ISRN Neurology*, vol. 2013, pp. 2090–5505, (2013), doi: 10.1155/2013/892459

Hosseini-Asl, E., Ghazal M, Mahmoud, A., Aslantas, A., Shalaby, A., Casanova, M., Barnes, G., Gimel'farb, G., Keynton, R. & El-Baz, A., "Alzheimer's disease diagnostics by a 3D deeply supervised adaptable convolutional network", *Frontiers in Bioscience (Landmark Ed)*, pp. 584–596, (2018), doi: 10.2741/4606

Hubel, D. & Wiesel, T., “Receptive fields of single neurons in the cat’s striate cortex”, *The Journal of physiology*, vol. 148, pp. 574–591, (1959)

Ioffe, S., & Szegedy, C., "Batch normalization: Accelerating deep network training by reducing internal covariate shift", JMLR.org, *In International conference on machine learning*, vol. 37, pp. 448–456. (2015)

Jo, T., Nho, K. & Saykin, A., “Deep Learning in Alzheimer's Disease: Diagnostic Classification and Prognostic Prediction Using Neuroimaging Data”. *Frontiers in Aging Neuroscience*. vol. 11, no. 220, (2019), doi: 10.3389/fnagi.2019.00220.

Kaiming, H., Xiangyu, Z., Shaoqing, R. & Jian, S. “Deep Residual Learning for Image Recognition.” *arXiv preprint* arXiv: 1512.03385. (2015)

Kiernan, J., “Anatomy of the temporal lobe”, *Epilepsy research and treatment*, vol. 2012, pp. 176157, (2012), doi: 10.1155/2012/176157

Kingma, D. P., & Ba, J. “Adam: A method for stochastic optimization.” *arXiv preprint* arXiv:1412.6980. (2014)

Kochanek, K., Xu, J., & Arias, E., “Mortality in the United States, 2019”, *NCHS data brief*, vol. 395, pp. 1–8, (2020)

Krizhevsky, A., Sutskever, I. & Hinton, G., E., “ImageNet Classification with Deep Convolutional Neural Networks”. In: Pereira, F., Burges, C., J., C., Bottou, L. & Weinberger, K., Q. (eds.), Association for Computing Machinery (New York, NY, United States), *Advances in Neural Information Processing Systems*, vol. 25, no. 2, pp 1097–1105, (2012), doi: 10.1145/3065386.

Kuusela, M., Raiko, T., Honkela, A., & Karhunen, J., "A gradient-based algorithm competitive with variational Bayesian EM for a mixture of Gaussians," *2009 International Joint Conference on Neural Networks, Atlanta, GA*, pp. 1688–1695, (2009), doi: 10.1109/ijcnn.2009.5178726

Leandrou, S., Lamnisos, D., Mamais, I., Kyriacou, A., P. & Pattichis, C., S., “Assessment of Alzheimer’s Disease Based on Texture Analysis of the Entorhinal Cortex”, *Frontiers in Aging Neuroscience*, vol. 12, no. 176, (2020), doi: 10.3389/fnagi.2020.00176

Leandrou, S., Petroudi, P., Kyriacou, A., P., Reyes-Aldasoro, C., C. & Pattichis, C., S., “Quantitative MRI Brain Studies in Mild Cognitive Impairment and Alzheimer's Disease: A Methodological Review”, *IEEE Reviews in Biomedical Engineering*, vol. 11, pp. 97–111, (2018), doi: 10.1109/RBME.2018.2796598.

LeCun, Y., Bengio, Y. & Hinton, G., E., “Deep learning”, *Nature*, vol. 521, pp. 436–444 (2015), doi: 10.1038/nature14539.

LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P., “Gradient-based learning applied to document recognition”, *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, (1998), doi: 10.1109/5.726791

Leontiou, P., Agathocleous, M. & Christodoulou, C., & Promponas, V., “Protein secondary structure prediction using Convolutional neural networks and hessian free optimization”, (in preparation), (2021).

Li, R., Zhang, W., Suk, H., Wang, L., Li, J., Shen, D. & Ji, S., “Deep learning based imaging data completion for improved brain disease diagnosis”, *International Conference on Medical Image Computing and Computer-Assisted Intervention*, vol. 17, pp. 305–312, (2014)

Liano, K., “Robust error measure for supervised neural network learning with outliers”, *IEEE Transactions on Neural Networks*, vol. 7, pp 246–250, (1996), doi: 10.1109/72.478411

Lin, W., Tong, T., Gao, Q., Guo, D., Du, X., Yang, Y., Guo, G., Xiao, M., Du, M. & Qu, X., for the Alzheimer’s Disease Neuroimaging Initiative, “Convolutional Neural Networks-Based MRI Image Analysis for the Alzheimer’s Disease Prediction from Mild Cognitive Impairment”. *Frontiers in Neuroscience*, vol. 12, no. 777, (2018), doi: 10.3389/fnins.2018.00777

Lippmann, R., “An introduction to computing with neural nets”, *IEEE ASSP Magazine*, vol. 4, pp. 4–22, (1987), doi: 10.1109/massp.1987.1165576

Liu, M., Cheng, D., Yan, W., & Alzheimer's Disease Neuroimaging Initiative, “Classification of Alzheimer's disease by combination of Convolutional and recurrent neural networks using FDG-PET images”, *Frontiers in neuroinformatics*, vol. 12, no. 35, (2018), doi: 10.3389/fninf.2018.00035

Maguire, E., Woollett, K., & Spiers, H., “London taxi drivers and bus drivers: A structural mri and neuropsychological analysis”, *Hippocampus*, vol. 16, pp. 1091–1101, (2006)

Martens, J. “Deep learning via hessian-free optimization”. In: Bottou, L. and Littman, M. (eds.), *Proceedings of the 27th International Conference on Machine Learning* (ICML’10), pp. 735-742, (2010).

Martens, J. & Sutskever, I., “Training Deep and Recurrent Networks with HessianFree Optimization”. In: Montavon G., Orr G.B., Müller KR. (eds), *Neural Networks: Tricks* of the Trade, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, vol. 7700, pp. 479-535 (2012).

McCulloch, W.S. & Pitts, W., “A logical calculus of the ideas immanent in nervous activity”. *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, (1943), doi: 10.1007/BF02478259

Nalepa, J., Marcinkiewicz, M. & Kawulok, M., “Data Augmentation for Brain-Tumor Segmentation: A Review”. *Frontiers in Computational Neuroscience*, vol. 13, no. 83. (2019), doi: 10.3389/fncom.2019.00083

O’Keefe, J. & Dostrovsky, J., “The hippocampus as a spatial map. preliminary evidence from unit activity in the freely-moving rat”, *Brain Research*, vol. 34, pp. 171–175, (1971)

O’Keefe, J. & Recce, M. L., “Phase relationship between hippocampal place units and the EEG theta rhythm”, *Hippocampus*, vol. 3, pp. 317–330, (1993)

Oh, K., Chung, Y., Kim, K., Kim, W. & Oh, I., “Classification and Visualization of Alzheimer’s Disease using Volumetric Convolutional Neural Network and Transfer Learning”, *Scientific Reports*, vol. 9, no. 1, pp. 2045–2322, (2019), doi: 10.1038/s41598-019-54548-6

Pearlmutter, B., “Fast Exact Multiplication by the Hessian”. *Neural Computation*, vol. 6, (1994), doi: 10.1162/neco.1994.6.1.147

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E., “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, (2011)

Petersen, R., Smith, G., Waring, S., Ivnik, R., Kokmen, E. & Tangelos, E., “Aging, Memory, and Mild Cognitive Impairment”. *International Psychogeriatrics*, vol. 9, pp. 65–69, (1997), doi: 10.1017/s1041610297004717

Robbins, H. & Monro, S., “A stochastic approximation method”, Institute of Mathematical Statistics, *The Annals of Mathematical Statistics*, vol. 22, no.3, pp. 400–407, (1951), doi: 10.1214/aoms/1177729586

Rumelhart, D., E., Hinton, G., E., & Williams, R., J., “Learning representations by back-propagating errors”. *Nature*, vol. 323, pp. 533-536 (1986), doi: 10.1038/323533a0

Simonyan, K. & Zisserman, A., “Very deep Convolutional networks for large-scale image recognition”, *arXiv preprint arXiv*, pp. 1409–1556, (2014)

Squire, L. & Wixted, J., “The cognitive neuroscience of human memory since hm”, *Annual Review of Neuroscience*, vol. 34, pp. 259–288, (2011)

Srivastava, N., Hinton, G., E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. “Dropout: a simple way to prevent neural networks from overfitting”. *The journal of machine learning research*, vol. 15, no. 1, pp. 1929-1958, (2014)

Suk, H., Lee, S. & Shen, D.., “Latent feature representation with stacked auto-encoder for AD/MCI diagnosis”. *Brain Struct Funct*, vol. 220, pp. 841–859, (2015), doi: 10.1007/s00429-013-0687-3

Suk, H. & Shen, D., “Deep learning-based feature representation for AD/MCI classification”, *International Conference on Medical Image Computing and Computer-Assisted Intervention*, vol. 16, pp. 583–590, (2013)

Tompson, J., Goroshin, R., Jain, A., LeCun, Y. & Bregler, C., “Efficient object localization using Convolutional networks”, *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 648–656, (2015)

Venugopalan, J., Tong, L., Hassanzadeh, H. & Wang, M., “Multimodal deep learning models for early detection of Alzheimer’s disease stage”, *Scientific Reports*, vol. 11, no. 1, pp. 1–13, (2021), doi: 10.1038/s41598-020-74399-w

Vu, T., Yang, H., Nguyen, V., Oh, A. & Kim, M., “Multimodal learning using Convolution neural network and Sparse Autoencoder”, *2017 IEEE International Conference on Big Data and Smart Computing*, pp. 309–312, (2017)

Wang, C.-C., Tan, K., & Lin, C.-J., “Newton Methods for Convolutional Neural Networks”. *ACM Transaction on Intelligent Systems and Technology*. vol. 11, 2, Article 19, (2020), doi: 10.1145/3368271.

Wen, J., Thibeau-Sutre, E., Diaz-Melo, M., Samper-González, J., Routier, A., Bottani, S., Dormont, D., Durrleman, S., Burgos, N. & Colliot, O., “Convolutional Neural Networks for Classification of Alzheimer's Disease: Overview and Reproducible Evaluation”. *Medical Image Analysis*, vol. 63, pp. 101694, (2020), doi: 10.1016/j.media.2020.101694.

Appendix A

MLP Implementations

A.1 MLP with Adam

The following is the MLP implementation used for the experiments with the Adam optimizer. It executes 10 times each one of the 10-folds and the performance metrics is the average of the best execution of the 10 repetitions of each fold. Modifying the 'reps' variable changes the number of repetitions per fold. The following code tests the performance of different learning rates [0.3, 0.03, 0.003, 0.0003].

```
from sklearn.metrics import accuracy_score, log_loss
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from utilities import stats_output
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import time
import joblib
import os
import sys

# np.set_printoptions(threshold=sys.maxsize)
all_best_train_accuracies = [0]*10
all_best_test_accuracies = [0]*10
best_train_accuracies = []
best_test_accuracies = []
best_experiments = [1]*10

# 10-fold cross-validation (10 training & test sets)
folds = 10

reps = 10


def read_excel(file):
    url = "./data/mri_features/"
    excel_data = pd.read_excel(url + file)
    # Read titles
    titles = excel_data.columns.ravel()

    # Read training set
    X = []
    i = 0
    for i in range(len(excel_data)):  # Each row
        X.append(excel_data.iloc[i].to_numpy()[2:-1])

    # Read test set
    y = np.array(excel_data[titles[-1]].tolist())
    i = 0
    for i in range(0, len(y)):
        if (y[i] == "NC"):
            y[i] = 0
        elif (y[i] == "AD"):
            y[i] = 1

    return X, y
```

```
def normalize(X_train, X_test):
    # Scale values between -1 and 1
    scaler = StandardScaler()
    # Don't cheat - fit only on training data
    scaler.fit(X_train)

    X_train = scaler.transform(X_train)

    # apply same transformation to test data
    X_test = scaler.transform(X_test)
    return X_train, X_test


def run_model(folder, index, train_file, test_file, lr):
    X_train, y_train = read_excel(train_file)
    X_test, y_test = read_excel(test_file)

    X_train, X_test = normalize(X_train, X_test)

    # For overfitting
    alpha = 7

    # Epochs
    N_EPOCHS = 500
    N_BATCH = 512
    N_TRAIN_SAMPLES = X_train.shape[0]
    N_CLASSES = np.unique(y_train)

    # ‘lbfgs’ is an optimizer in the family of quasi-Newton methods.
    # ‘sgd’ refers to stochastic gradient descent.
    # ‘adam’ refers to a stochastic gradient-
based optimizer proposed by Kingma, Diederik, and Jimmy Ba
    solver = 'adam'

    # momentum = 0.9 only used for SGD

    # ‘identity’, no-op activation, useful to implement linear bottleneck, returns f(x) = x
    # ‘logistic’, the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
    # ‘tanh’, the hyperbolic tan function, returns f(x) = tanh(x).
    # ‘relu’, the rectified linear unit function, returns f(x) = max(0, x)
    activation = 'relu'

    # Classification predicts label, regression predicts quantity
    mlp = MLPClassifier(activation=activation,
                        # early_stopping=True,
                        # momentum=0.9,
                        hidden_layer_sizes=[8, 8],
                        solver=solver, alpha=alpha,
                        max_iter=N_EPOCHS,
                        learning_rate_init=lr)

    scores_train = []
    scores_test = []

    loss_train = []
    loss_test = []

    # Print to a format friendly for excel
    metrics_train_file = open(
        folder + "metrics_train_file_" + str(index) + ".txt", 'w')
    metrics_valid_file = open(
        folder + "metrics_valid_file_" + str(index) + ".txt", 'w')

    print("f{0}_train_epoch\tf{0}_train_tn\tf{0}_train_fp\tf{0}_train_fn\tf{0}_train_tp\tf{0}_
train_acc\tf{0}_train_ppv\tf{0}_train_npv\tf{0}_train_sensitivity\tf{0}_train_specificity\tf{0
}_train_b_acc\tf{0}_train_loss\tf{0}_train_epoch_time\tf{0}_train_lr".format(index),
          file=metrics_train_file)
    print("f{0}_valid_epoch\tf{0}_valid_tn\tf{0}_valid_fp\tf{0}_valid_fn\tf{0}_valid_tp\tf{0}_
valid_acc\tf{0}_valid_ppv\tf{0}_valid_npv\tf{0}_valid_sensitivity\tf{0}_valid_specificity\tf{0
}_valid_b_acc\tf{0}_valid_loss\tf{0}_valid_epoch_time\tf{0}_valid_lr".format(index),
```

```
        file=metrics_valid_file)

# Force to write immediately to file
metrics_train_file.flush()
metrics_valid_file.flush()

# Save the accuracies
# with open(folder + 'accuracies_' + str(index) + '.txt', 'w') as f1:
#     print('epoch:\ttrain_acc:\ttest_acc:', file=f1)

#     # Force to write immediately to file
#     f1.flush()

max_train_acc = 0
max_test_acc = 0
epoch = 0
while epoch < N_EPOCHS:

    epoch_start = time.time()

    # SHUFFLING
    random_perm = np.random.permutation(X_train.shape[0])
    mini_batch_index = 0
    while True:
        # MINI-BATCH
        indices = random_perm[mini_batch_index:mini_batch_index + N_BATCH]
        mlp.partial_fit(X_train[indices],
                        y_train[indices], classes=N_CLASSES)
        mini_batch_index += N_BATCH

        if mini_batch_index >= N_TRAIN_SAMPLES:
            break

    # exclude data loading time for fair comparison
    epoch_end = time.time()

    # SCORE TRAIN
    scores_train.append(mlp.score(X_train, y_train))

    # SCORE TEST
    scores_test.append(mlp.score(X_test, y_test))

    # print(epoch, '\t', scores_train[-1],
    #         '\t', scores_test[-1], file=f1)

    # # Force to write immediately to file
    # f1.flush()

    if max_test_acc < scores_test[-1]:
        max_train_acc = scores_train[-1]
        max_test_acc = scores_test[-1]
        # Save the model
        model_filename = folder + \
            'finalized_model' + str(index) + '.sav'
        if os.path.exists(model_filename):
            os.remove(model_filename)
        joblib.dump(mlp, model_filename)

    # Training Statistics
    train_real = mlp.predict(X_train)
    train_predict_proba = mlp.predict_proba(X_train)
    loss_train.append(log_loss(y_train, train_predict_proba))
    # train_acc = accuracy_score(y_train, train_real)
    stats_train_output, stats_train_file_output = stats_output(
        "Training", epoch, y_train, train_real)

    # Print train metrics in a text file friendly for Excel
    stats_train_file_output += "{:.3f}\t{:.3f}\t{:.5f}".format(
        loss_train[-1],
        epoch_end-epoch_start, lr)
    print(stats_train_file_output, file=metrics_train_file)

    # Force to write immediately to file
```

```
        metrics_train_file.flush()

        # Test Statistics
        test_real = mlp.predict(X_test)
        test_predict_proba = mlp.predict_proba(X_test)
        loss_test.append(log_loss(y_test, test_predict_proba))
        # test_acc = accuracy_score(y_test, test_real)
        stats_test_output, stats_test_file_output = stats_output(
            "Validation", epoch, y_test, test_real)

        # Print test metrics in a text file friendly for Excel
        stats_test_file_output += "{:.3f}\t{:.3f}\t{:.5f}".format(
            loss_test[-1],
            epoch_end-epoch_start, lr)
        print(stats_test_file_output,  file=metrics_valid_file)

        # Force to write immediately to file
        metrics_valid_file.flush()

        epoch += 1

    best_train_accuracies.append(max_train_acc)
    best_test_accuracies.append(max_test_acc)

    # Save to file accuracies
    # rep_acc_file = open(folder + 'run_accuracies.txt', 'a')

    # if (index == 1):
    #     print('Fold\tTrain Acc\tValid Acc\t')

    print(str(index), ':\t', max_train_acc, '\t', max_test_acc)
    # print(str(index), ':\t',
    #       max_train_acc, '\t', max_test_acc, file=rep_acc_file)

    # Force to write immediately to file
    # rep_acc_file.flush()

    # # load the model from disk
    # loaded_model = joblib.load('finalized_model.sav')
    # result = loaded_model.score(X_test, Y_test)
    # print(result)

    # plt.clf()
    # plt.plot(scores_train, color='green', alpha=0.8, label='Train')
    # plt.plot(scores_test, color='magenta', alpha=0.8, label='Valid')
    # plt.title("Accuracy over epochs", fontsize=14)
    # plt.xlabel('Epochs')
    # plt.legend(loc='upper left')
    # plt.savefig(folder + 'accuracy_plot_' + str(index) + '.png')

    # plt.clf()
    # plt.plot(loss_train, color='green', alpha=0.8, label='Train')
    # plt.plot(loss_test, color='magenta', alpha=0.8, label='Valid')
    # plt.title("Loss over epochs", fontsize=14)
    # plt.xlabel('Epochs')
    # plt.legend(loc='upper left')
    # plt.savefig(folder + 'loss_plot_' + str(index) + '.png')
    return mlp.get_params()


def read_file(folder):
    modes = ["train", "valid"]
    for m in range(2):
        all_lines = [""] * 501
        for fold in range(1, 11):
            file_path = folder + "Experiment_" + \
                str(fold) + "_" + str(best_experiments[fold - 1]) + "/" + "metrics_" + \
                str(modes[m]) + "_file_" + str(fold) + ".txt"
            print(file_path)
            file = open(file_path, 'r')
            lines = file.readlines()
            count = 0
```

```
            for line in lines:
                line_str = line.strip()
                if (count == 0):
                    first_line = line.strip().split("\t")
                    line_str = "\t".join(first_line)
                all_lines[count] = all_lines[count] + line_str

                if(fold != 10):
                    all_lines[count] = all_lines[count] + "\t\t"
                count += 1

        file_w = open(folder + modes[m] + "_file.txt", 'w')
        for line in all_lines:
            print(line, file=file_w)


def main():
    lrs = [0.3, 0.03, 0.003, 0.0003]
    for lr in lrs:
        print("\n\nLearning Rate: " + str(lr))
        global all_best_train_accuracies
        global all_best_test_accuracies
        global best_train_accuracies
        global best_test_accuracies
        global best_experiments

        all_best_train_accuracies = [0]*10
        all_best_test_accuracies = [0]*10
        best_train_accuracies = []
        best_test_accuracies = []
        best_experiments = [1]*10

        start_time = datetime.now().strftime("%d-%m-%Y_%H-%M-%S")
        main_folder = "./Experiments/MLP_Adam/Experiments_LR" + \
            str(lr).replace(".", "") + "_" + start_time + "/"
        os.makedirs(main_folder)

        for r in range(1, reps + 1):
            print("\n\nRepetition " + str(r) + "\n")

            for f in range(1, folds + 1):
                sub_folder = main_folder + "Experiment_" + \
                    str(f) + "_" + str(r) + "/"
                os.makedirs(sub_folder)

                parameters = run_model(sub_folder, f, 'TrainSet'+str(f) + '.xlsx',
                                       'testset' + str(f) + '.xlsx', lr)

                if r == 1 and f == 1:
                    info_file = open(
                        main_folder + "network_info_parameters.txt", 'w')
                    print(parameters, file=info_file)
                    info_file.flush()

            # Find the best trianing and test accuracy based on the maximum test accuracy
            # Print in concole
            print(best_train_accuracies)
            print(best_test_accuracies)
            print("\n")
            print("Average of Best Train Accuracies: ",
                  sum(best_train_accuracies)/folds)
            print("Average of Best Valid Accuracies: ",
                  sum(best_test_accuracies)/folds)

            for b in range(0, len(best_test_accuracies)):
                if best_test_accuracies[b] > all_best_test_accuracies[b]:
                    all_best_test_accuracies[b] = float(
                        best_test_accuracies[b])
                    all_best_train_accuracies[b] = float(
                        best_train_accuracies[b])
                    best_experiments[b] = r
```

```
            best_train_accuracies.clear()
            best_test_accuracies.clear()

        print("\nBest Experiments per Fold:")
        print(best_experiments)

        print("\nBest Accuracies per Fold:")
        print("Train\tValid")
        for i in range(len(all_best_train_accuracies)):
            print(str(all_best_train_accuracies[i]) +
                  "\t" + str(all_best_test_accuracies[i]))
        print("\n\nCollecting Data:")
        read_file(main_folder)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet A.1. MLP network with Adam optimizer.

A.2 MLP with SGD

The following code which is being used for MLP experiments with the SGD optimizer is very similar to the Adam one. It repeats 10 times each one of the 10-folds and takes the best ones per fold to calculate the average performance metrics.

```
from sklearn.metrics import accuracy_score, log_loss
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from utilities import stats_output
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import time
import joblib
import os
import sys

# np.set_printoptions(threshold=sys.maxsize)
all_best_train_accuracies = [0]*10
all_best_test_accuracies = [0]*10
best_train_accuracies = []
best_test_accuracies = []
best_experiments = [1]*10

# 10-fold cross-validation (10 training & test sets)
folds = 10

reps = 10


def read_excel(file):
    url = "./data/mri_features/"
    excel_data = pd.read_excel(url + file)
    # Read titles
    titles = excel_data.columns.ravel()

    # Read training set
    X = []
    i = 0
    for i in range(len(excel_data)):  # Each row
        X.append(excel_data.iloc[i].to_numpy()[2:-1])

    # Read test set
```

```
    y = np.array(excel_data[titles[-1]].tolist())
    i = 0
    for i in range(0, len(y)):
        if (y[i] == "NC"):
            y[i] = 0
        elif (y[i] == "AD"):
            y[i] = 1

    return X, y


def normalize(X_train, X_test):
    # Scale values between -1 and 1
    scaler = StandardScaler()
    # Don't cheat - fit only on training data
    scaler.fit(X_train)

    X_train = scaler.transform(X_train)

    # # apply same transformation to test data
    X_test = scaler.transform(X_test)
    return X_train, X_test


def run_model(folder, index, train_file, test_file, lr):
    X_train, y_train = read_excel(train_file)
    X_test, y_test = read_excel(test_file)

    X_train, X_test = normalize(X_train, X_test)

    # For overfitting
    alpha = 7

    # Epochs
    N_EPOCHS = 500
    N_BATCH = 512
    N_TRAIN_SAMPLES = X_train.shape[0]
    N_CLASSES = np.unique(y_train)

    # ‘lbfgs’ is an optimizer in the family of quasi-Newton methods.
    # ‘sgd’ refers to stochastic gradient descent.
    # ‘adam’ refers to a stochastic gradient-
based optimizer proposed by Kingma, Diederik, and Jimmy Ba
    solver = 'sgd'

    # momentum = 0.9 only used for SGD

    # ‘identity’, no-op activation, useful to implement linear bottleneck, returns f(x) = x
    # ‘logistic’, the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
    # ‘tanh’, the hyperbolic tan function, returns f(x) = tanh(x).
    # ‘relu’, the rectified linear unit function, returns f(x) = max(0, x)
    activation = 'relu'

    # Classification predicts label, regression predicts quantity
    mlp = MLPClassifier(activation=activation,
                        # early_stopping=True,
                        momentum=0.8,
                        hidden_layer_sizes=[8, 8],
                        solver=solver, alpha=alpha,
                        max_iter=N_EPOCHS,
                        learning_rate_init=lr)

    scores_train = []
    scores_test = []

    loss_train = []
    loss_test = []

    # Print to a format friendly for excel
    metrics_train_file = open(
        folder + "metrics_train_file_" + str(index) + ".txt", 'w')
    metrics_valid_file = open(
```

```
        folder + "metrics_valid_file_" + str(index) + ".txt", 'w')

    print("f{0}_train_epoch\tf{0}_train_tn\tf{0}_train_fp\tf{0}_train_fn\tf{0}_train_tp\tf{0}_
train_acc\tf{0}_train_ppv\tf{0}_train_npv\tf{0}_train_sensitivity\tf{0}_train_specificity\tf{0
}_train_b_acc\tf{0}_train_loss\tf{0}_train_epoch_time\tf{0}_train_lr".format(index),
          file=metrics_train_file)
    print("f{0}_valid_epoch\tf{0}_valid_tn\tf{0}_valid_fp\tf{0}_valid_fn\tf{0}_valid_tp\tf{0}_
valid_acc\tf{0}_valid_ppv\tf{0}_valid_npv\tf{0}_valid_sensitivity\tf{0}_valid_specificity\tf{0
}_valid_b_acc\tf{0}_valid_loss\tf{0}_valid_epoch_time\tf{0}_valid_lr".format(index),
          file=metrics_valid_file)

    # Force to write immediately to file
    metrics_train_file.flush()
    metrics_valid_file.flush()

    # Save the accuracies
    # with open(folder + 'accuracies_' + str(index) + '.txt', 'w') as f1:
    #     print('epoch:\ttrain_acc:\ttest_acc:', file=f1)

    #     # Force to write immediately to file
    #     f1.flush()

    max_train_acc = 0
    max_test_acc = 0
    epoch = 0
    while epoch < N_EPOCHS:

        epoch_start = time.time()

        # SHUFFLING
        random_perm = np.random.permutation(X_train.shape[0])
        mini_batch_index = 0
        while True:
            # MINI-BATCH
            indices = random_perm[mini_batch_index:mini_batch_index + N_BATCH]
            mlp.partial_fit(X_train[indices],
                            y_train[indices], classes=N_CLASSES)
            mini_batch_index += N_BATCH

            if mini_batch_index >= N_TRAIN_SAMPLES:
                break

        # exclude data loading time for fair comparison
        epoch_end = time.time()

        # SCORE TRAIN
        scores_train.append(mlp.score(X_train, y_train))

        # SCORE TEST
        scores_test.append(mlp.score(X_test, y_test))

        # print(epoch, '\t', scores_train[-1],
        #         '\t', scores_test[-1], file=f1)

        # # Force to write immediately to file
        # f1.flush()

        if max_test_acc < scores_test[-1]:
            max_train_acc = scores_train[-1]
            max_test_acc = scores_test[-1]
            # Save the model
            model_filename = folder + \
                'finalized_model' + str(index) + '.sav'
            if os.path.exists(model_filename):
                os.remove(model_filename)
            joblib.dump(mlp, model_filename)

        # Training Statistics
        train_real = mlp.predict(X_train)
        train_predict_proba = mlp.predict_proba(X_train)
        loss_train.append(log_loss(y_train, train_predict_proba))
        # train_acc = accuracy_score(y_train, train_real)
        stats_train_output, stats_train_file_output = stats_output(
```

```
            "Training", epoch, y_train, train_real)

        # Print train metrics in a text file friendly for Excel
        stats_train_file_output += "{:.3f}\t{:.3f}\t{:.5f}".format(
            loss_train[-1],
            epoch_end-epoch_start, lr)
        print(stats_train_file_output, file=metrics_train_file)

        # Force to write immediately to file
        metrics_train_file.flush()

        # Test Statistics
        test_real = mlp.predict(X_test)
        test_predict_proba = mlp.predict_proba(X_test)
        loss_test.append(log_loss(y_test, test_predict_proba))
        # test_acc = accuracy_score(y_test, test_real)
        stats_test_output, stats_test_file_output = stats_output(
            "Validation", epoch, y_test, test_real)

        # Print test metrics in a text file friendly for Excel
        stats_test_file_output += "{:.3f}\t{:.3f}\t{:.5f}".format(
            loss_test[-1],
            epoch_end-epoch_start, lr)
        print(stats_test_file_output,  file=metrics_valid_file)

        # Force to write immediately to file
        metrics_valid_file.flush()

        epoch += 1

    best_train_accuracies.append(max_train_acc)
    best_test_accuracies.append(max_test_acc)

    # Save to file accuracies
    # rep_acc_file = open(folder + 'run_accuracies.txt', 'a')

    # if (index == 1):
    #     print('Fold\tTrain Acc\tValid Acc\t')

    print(str(index), ':\t', max_train_acc, '\t', max_test_acc)
    # print(str(index), ':\t',
    #       max_train_acc, '\t', max_test_acc, file=rep_acc_file)

    # Force to write immediately to file
    # rep_acc_file.flush()

    # # load the model from disk
    # loaded_model = joblib.load('finalized_model.sav')
    # result = loaded_model.score(X_test, Y_test)
    # print(result)

    # plt.clf()
    # plt.plot(scores_train, color='green', alpha=0.8, label='Train')
    # plt.plot(scores_test, color='magenta', alpha=0.8, label='Valid')
    # plt.title("Accuracy over epochs", fontsize=14)
    # plt.xlabel('Epochs')
    # plt.legend(loc='upper left')
    # plt.savefig(folder + 'accuracy_plot_' + str(index) + '.png')

    # plt.clf()
    # plt.plot(loss_train, color='green', alpha=0.8, label='Train')
    # plt.plot(loss_test, color='magenta', alpha=0.8, label='Valid')
    # plt.title("Loss over epochs", fontsize=14)
    # plt.xlabel('Epochs')
    # plt.legend(loc='upper left')
    # plt.savefig(folder + 'loss_plot_' + str(index) + '.png')
    return mlp.get_params()


def read_file(folder):
    modes = ["train", "valid"]
    for m in range(2):
```

```
        all_lines = [""] * 501
        for fold in range(1, 11):
            file_path = folder + "Experiment_" + \
                str(fold) + "_" + str(best_experiments[fold - 1]) + "/" + "metrics_" + \
                str(modes[m]) + "_file_" + str(fold) + ".txt"
            print(file_path)
            file = open(file_path, 'r')
            lines = file.readlines()
            count = 0
            for line in lines:
                line_str = line.strip()
                if (count == 0):
                    first_line = line.strip().split("\t")
                    line_str = "\t".join(first_line)
                all_lines[count] = all_lines[count] + line_str

                if(fold != 10):
                    all_lines[count] = all_lines[count] + "\t\t"
                count += 1

        file_w = open(folder + modes[m] + "_file.txt", 'w')
        for line in all_lines:
            print(line, file=file_w)


def main():
    lrs = [0.3, 0.03, 0.003, 0.0003]
    for lr in lrs:
        print("\n\nLearning Rate: " + str(lr))
        global all_best_train_accuracies
        global all_best_test_accuracies
        global best_train_accuracies
        global best_test_accuracies
        global best_experiments

        all_best_train_accuracies = [0]*10
        all_best_test_accuracies = [0]*10
        best_train_accuracies = []
        best_test_accuracies = []
        best_experiments = [1]*10

        start_time = datetime.now().strftime("%d-%m-%Y_%H-%M-%S")
        main_folder = "./Experiments/MLP_SGD/Experiments_LR" + \
            str(lr).replace(".", "") + "_" + start_time + "/"
        os.makedirs(main_folder)

        for r in range(1, reps + 1):
            print("\n\nRepetition " + str(r) + "\n")

            for f in range(1, folds + 1):
                sub_folder = main_folder + "Experiment_" + \
                    str(f) + "_" + str(r) + "/"
                os.makedirs(sub_folder)

                parameters = run_model(sub_folder, f, 'TrainSet'+str(f) + '.xlsx',
                                       'testset' + str(f) + '.xlsx', lr)

                if r == 1 and f == 1:
                    info_file = open(
                        main_folder + "network_info_parameters.txt", 'w')
                    print(parameters, file=info_file)
                    info_file.flush()

            # Find the best trianing and test accuracy based on the maximum test accuracy
            # Print in concole
            print(best_train_accuracies)
            print(best_test_accuracies)
            print("\n")
            print("Average of Best Train Accuracies: ",
                  sum(best_train_accuracies)/folds)
            print("Average of Best Valid Accuracies: ",
                  sum(best_test_accuracies)/folds)
```

```
            for b in range(0, len(best_test_accuracies)):
                if best_test_accuracies[b] > all_best_test_accuracies[b]:
                    all_best_test_accuracies[b] = float(
                        best_test_accuracies[b])
                    all_best_train_accuracies[b] = float(
                        best_train_accuracies[b])
                    best_experiments[b] = r

            best_train_accuracies.clear()
            best_test_accuracies.clear()

        print("\nBest Experiments per Fold:")
        print(best_experiments)

        print("\nBest Accuracies per Fold:")
        print("Train\tValid")
        for i in range(len(all_best_train_accuracies)):
            print(str(all_best_train_accuracies[i]) +
                  "\t" + str(all_best_test_accuracies[i]))
        print("\n\nCollecting Data:")
        read_file(main_folder)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet A.2. MLP network with SGD optimizer.

A.3 MLP with HFO

The files main.py and FFNet.py are being used for the experiments with MLP and the HFO optimizer. The code repeats 10 times the execution of each one of the 10-folds and takes the best one for each fold to compute the average performance metrics of the model. The code runs experiments for the following distinct values of CG iterations = [1, 2, 4, 8, 16, 32].

A.3.1 main.py

```
from __future__ import print_function
from sklearn.preprocessing import StandardScaler

import ast
from cProfile import Profile
import pickle
import pstats
import os
import sys

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime

import hessianfree as hf
import time
from FFNet.FFNet import FFNet

# np.set_printoptions(threshold=sys.maxsize)
all_best_train_accuracies = [0]*10
all_best_test_accuracies = [0]*10
```

```
best_train_accuracies = []
best_test_accuracies = []
best_experiments = [1]*10

# 10-fold cross-validation (10 training & test sets)
folds = 10

# Repetitions per cross-validation
reps = 10


def read_excel(file):
    url = "./data/mri_features/"
    excel_data = pd.read_excel(url + file)
    # Read titles
    titles = excel_data.columns.ravel()

    # Read training set
    X = []
    i = 0
    for i in range(len(excel_data)):  # Each row
        X.append(excel_data.iloc[i].to_numpy()[2:-1])

    # Read test set
    t = np.array(excel_data[titles[-1]].tolist())
    y = []
    i = 0
    for i in range(0, len(t)):
        if (t[i] == "NC"):
            y.append([0])
        elif (t[i] == "AD"):
            y.append([1])

    return X, y


def normalize(X_train, X_test):
    # Scale values between -1 and 1
    scaler = StandardScaler()
    # Don't cheat - fit only on training data
    scaler.fit(X_train)

    X_train = scaler.transform(X_train)

    # # apply same transformation to test data
    X_test = scaler.transform(X_test)
    return X_train, X_test


def get_accuracy(ff, X, y):
    outputs = ff.forward(X)[-1]
    correct = 0
    for i in range(len(outputs)):
        # print("-" * 2)
        # print("input", X[i])
        # print("target", y[i])
        # print("output", outputs[i])
        if (outputs[i] < 0.5):
            n_output = 0
        else:
            n_output = 1
        # print('n_output', 0)
        if (n_output == y[i]):
            correct += 1
    acc = correct / len(outputs)
    return acc


def run_model(folder, fold, train_file, test_file, N_EPOCHS, CGiter, hidden_layers_net, use_hf
=True):
    """Run a basic xor training test.
```

```
    :param bool use_hf: if True run example using Hessian-Free Optimization,
        otherwise use stochastic gradient descent
    """
    X_train, y_train = read_excel(train_file)
    X_test, y_test = read_excel(test_file)

    inputs_dim = len(X_train[0])

    X_train, X_test = normalize(X_train, X_test)

    X_train = np.asarray(X_train, dtype=np.float32)
    y_train = np.asarray(y_train, dtype=np.float32)

    X_test = np.asarray(X_test, dtype=np.float32)
    y_test = np.asarray(y_test, dtype=np.float32)

    net = [inputs_dim] + hidden_layers_net + [1]
    # print("Inputs Dimensions: " + str(inputs_dim))
    # print("Network architecture: ", str(net))

    if use_hf:

        # Use the hessian free optimizer
        # Create feed forward net with 12 inputs, 5 hidden nodes and 2 output
        ff = FFNet(net, debug=False)
        max_train_acc, max_test_acc = ff.run_epochs(X_train, y_train, folder, fold,
                                                    optimizer=hf.opt.HessianFree(
                                                        CG_iter=CGiter),
                                                    test=(X_test, y_test),
                                                    test_err=hf.loss_funcs.SquaredError(),
                                                    print_period=200,
                                                    max_epochs=N_EPOCHS, plotting=False,
                                                    file_output=folder + "model_" + str(fold))
    else:
        # using gradient descent (for comparison)
        ff = FFNet([inputs_dim, 10, 1], debug=False)
        ff.run_epochs(X_train, y_train, folder, fold, optimizer=hf.opt.SGD(l_rate=1),
                      max_epochs=1000, print_period=100, test=(X_test, y_test), test_err=hf.lo
ss_funcs.SquaredError(),
                      file_output=folder + "model_" + str(fold), plotting=False)

    best_train_accuracies.append(max_train_acc)
    best_test_accuracies.append(max_test_acc)

    # Save to file accuracies
    rep_acc_file = open(folder + 'run_accuracies.txt', 'a')

    if (fold == 1):
        print('Fold\tTrain Acc\tValid Acc\t')

    print(str(fold), ':\t', max_train_acc, '\t', max_test_acc)
    print(str(fold), ':\t',
          max_train_acc, '\t', max_test_acc, file=rep_acc_file)

    # Force to write immediately to file
    rep_acc_file.flush()

    # ff.run_epochs(X_train, y_train,
    #               optimizer=hf.opt.HessianFree(CG_iter=250,
    #                                            init_damping=45),
    #               minibatch_size=7500, max_epochs=125,
    #               plotting=True)
    # print("classification error",
    #       hf.loss_funcs.ClassificationError().batch_loss(y_test, X_test))


def display_plots(filename):
    # View Plots
    hf.dataplotter.run(filename)


def read_file(folder):
```

```
    modes = ["train", "valid"]
    for m in range(2):
        all_lines = [""] * 101
        for fold in range(1, 11):
            file_path = folder + "Experiment_" + \
                str(fold) + "_" + str(best_experiments[fold - 1]) + "/" + "metrics_" + \
                str(modes[m]) + "_file_" + str(fold) + ".txt"
            print(file_path)
            file = open(file_path, 'r')
            lines = file.readlines()
            count = 0
            for line in lines:
                line_str = line.strip()
                if (count == 0):
                    first_line = line.strip().split("\t")
                    line_str = "\t".join(first_line)
                all_lines[count] = all_lines[count] + line_str

                if (count == 0):
                    all_lines[count] = all_lines[count] + "\t\t"
                elif(fold != 10):
                    all_lines[count] = all_lines[count] + "\t\t\t"
                count += 1

        file_w = open(folder + modes[m] + "_file.txt", 'w')
        for line in all_lines:
            print(line, file=file_w)


def main():

    # modes = ['HFO', 'SGD']

    CGiters = [1, 2, 4, 8, 16, 32]
    N_EPOCHS = 100
    mode = 'HFO'
    hidden_layers_net = [30]

    for CGiter in CGiters:
        global all_best_train_accuracies
        global all_best_test_accuracies
        global best_train_accuracies
        global best_test_accuracies
        global best_experiments

        all_best_train_accuracies = [0]*10
        all_best_test_accuracies = [0]*10
        best_train_accuracies = []
        best_test_accuracies = []
        best_experiments = [1]*10

        start_time = datetime.now().strftime("%d-%m-%Y_%H-%M-%S")
        main_folder = "./Experiments/MLP_" + mode + \
            "/Experiments_CGiter" + str(CGiter) + "_" + start_time + "/"
        os.makedirs(main_folder)

        info_file = open(
            main_folder + "network_info_parameters_CG" + str(CGiter) + ".txt", 'w')
        print(str(hidden_layers_net) + ", epochs: " + str(N_EPOCHS) +
              ", CG iter: " + str(CGiter) + ", mode: " + mode, file=info_file)
        # Force to write immediately to file
        info_file.flush()

        if (mode == 'HFO'):
            use_hf = True
        else:
            use_hf = False

        for r in range(1, reps + 1):
            print("\n\nRepetition " + str(r) + "\n")

            for f in range(1, folds + 1):
```

```
                sub_folder = main_folder + "Experiment_" + \
                    str(f) + "_" + str(r) + "/"
                os.makedirs(sub_folder)

                run_model(sub_folder, f, 'TrainSet'+str(f) + '.xlsx',
                          'testset' + str(f) + '.xlsx', N_EPOCHS,  CGiter, hidden_layers_net,
use_hf=use_hf)

            print(best_train_accuracies)
            print(best_test_accuracies)
            print("\n")
            print("Average of Best Train Accuracies: ",
                  sum(best_train_accuracies)/folds)
            print("Average of Best Valid Accuracies: ",
                  sum(best_test_accuracies)/folds)

            for b in range(0, len(best_test_accuracies)):
                if best_test_accuracies[b] > all_best_test_accuracies[b]:
                    all_best_test_accuracies[b] = float(
                        best_test_accuracies[b])
                    all_best_train_accuracies[b] = float(
                        best_train_accuracies[b])
                    best_experiments[b] = r

            best_train_accuracies.clear()
            best_test_accuracies.clear()

        print("\nBest Experiments per Fold:")
        print(best_experiments)

        print("\nBest Accuracies per Fold:")
        print("Train\tValid")
        for i in range(len(all_best_train_accuracies)):
            print(str(all_best_train_accuracies[i]) +
                  "\t" + str(all_best_test_accuracies[i]))
        print("\n\nCollecting Data:")
        read_file(main_folder)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet A.3. The main of the MLP network with HFO optimizer.

A.3.2 FFNet.py

```
"""Implementation of feedforward network, including Gauss-Newton approximation
for use in Hessian-Free Optimization.

.. codeauthor:: Daniel Rasmussen <daniel.rasmussen@appliedbrainresearch.com>

Based on
Martens, J. (2010). Deep learning via Hessian-Free Optimization. In Proceedings
of the 27th International Conference on Machine Learning.
"""

from __future__ import print_function
import matplotlib.pyplot as plt
import time

from collections import defaultdict, OrderedDict
import pickle
import warnings

import numpy as np

import hessianfree as hf

from utilities import stats_output
```

```
from sklearn.metrics import confusion_matrix


class FFNet(object):
    """Implementation of feed-forward network (including gradient/curvature
    computation).

    :param list shape: the number of neurons in each layer
    :param layers: nonlinearity to use in the network (or a list giving a
        nonlinearity for each layer)
    :type layers: :class:`~.nonlinearities.Nonlinearity` or `list`
    :param dict conns: dictionary of the form `{layer_x:[layer_y, layer_z],
        ...}` specifying the connections between layers (default is to
        connect in series)
    :param loss_type: loss function (or list of loss functions) used to
        evaluate network
    :type loss_type: :class:`~.loss_funcs.LossFunction` or `list`
    :param dict W_init_params: parameters passed to :meth:`.init_weights`
        (see parameter descriptions in that function)
    :param bool use_GPU: run curvature computation on GPU (requires
        PyCUDA and scikit-cuda)
    :param load_weights: load initial weights from given array or filename
    :type load_weights: `str` or :class:`~numpy:numpy.ndarray`
    :param bool debug: activates expensive features to help with debugging
    :param rng: used to generate any random numbers for this network (use
        this to control the seed)
    :type rng: :class:`~numpy:numpy.random.RandomState`
    :param dtype: floating point precision used throughout the network
    :type dtype: :class:`~numpy:numpy.dtype`
    """

    def __init__(self, shape, layers=hf.nl.Logistic(), conns=None,
                 loss_type=hf.loss_funcs.SquaredError(), W_init_params=None,
                 use_GPU=False, load_weights=None, debug=False, rng=None,
                 dtype=np.float32):

        self.debug = debug
        self.shape = shape
        self.n_layers = len(shape)
        self.dtype = np.float64 if debug else dtype
        self.mask = None
        self._optimizer = None
        self.rng = np.random.RandomState() if rng is None else rng

        # note: this isn't used internally, it is just here so that an
        # external process with a handle to this object can tell what epoch
        # it is on
        self.epoch = None

        self.inputs = None
        self.targets = None
        self.activations = None
        self.d_activations = None
        self.scores_train = []
        self.scores_test = []
        self.loss_train = []
        self.loss_test = []
        self.test_errs = []
        # initialize layer nonlinearities
        if not isinstance(layers, (list, tuple)):
            if isinstance(layers, hf.nl.Nonlinearity) and layers.stateful:
                warnings.warn("Multiple layers sharing stateful nonlinearity, "
                              "consider creating a separate instance for each "
                              "layer.")
            layers = [layers for _ in range(self.n_layers)]
            layers[0] = hf.nl.Linear()

        if len(layers) != len(shape):
            raise ValueError("Number of nonlinearities (%d) does not match "
                             "number of layers (%d)" %
                             (len(layers), len(shape)))
```

```
        self.layers = []
        for t in layers:
            if isinstance(t, str):
                # look up the nonlinearity with the given name
                t = getattr(hf.nl, t)()
            if not isinstance(t, hf.nl.Nonlinearity):
                raise TypeError("Layer type (%s) must be an instance of "
                                "nonlinearities.Nonlinearity" % t)
            self.layers += [t]

        # initialize loss function
        self.init_loss(loss_type)

        # initialize connections
        if conns is None:
            # set up the feedforward series connections
            conns = {}
            for pre, post in zip(np.arange(self.n_layers - 1),
                                 np.arange(1, self.n_layers)):
                conns[pre] = [post]

        self.conns = OrderedDict(sorted(conns.items(), key=lambda x: x[0]))
        # note: conns is an ordered dict sorted by layer so that we can
        # reliably loop over the items (in compute_offsets and init_weights)

        # maintain a list of backwards connections as well (for efficient
        # lookup in the other direction)
        self.back_conns = defaultdict(list)
        for pre in conns:
            for post in conns[pre]:
                self.back_conns[post] += [pre]

                if pre >= post:
                    raise ValueError("Can only connect from lower to higher "
                                     "layers (%s >= %s)" % (pre, post))

        # add empty connection for first/last layer (just helps smooth the code
        # elsewhere)
        self.conns[self.n_layers - 1] = []
        self.back_conns[0] = []

        # compute indices for the different connection weight matrices in the
        # overall parameter vector
        self.compute_offsets()

        # initialize connection weights
        if load_weights is None:
            if W_init_params is None:
                W_init_params = {}
            self.W = self.init_weights(
                [(self.shape[pre], self.shape[post])
                 for pre in self.conns for post in self.conns[pre]],
                **W_init_params)
        else:
            if isinstance(load_weights, np.ndarray):
                self.W = load_weights
            else:
                # load weights from file
                self.W = np.load(load_weights)

            if len(self.W) != np.max(list(self.offsets.values())):
                raise IndexError(
                    "Length of loaded weights (%s) does not match expected "
                    "length (%s)" % (len(self.W),
                                     np.max(list(self.offsets.values()))))

            if self.W.dtype != self.dtype:
                raise TypeError("Loaded weights dtype (%s) doesn't match "
                                "self.dtype (%s)" % (self.W.dtype, self.dtype))

        # initialize GPU
        if use_GPU:
            try:
```

```
                import pycuda
                import skcuda
            except Exception as e:
                print(e)
                raise ImportError("PyCuda/scikit-cuda not installed. "
                                  "Set use_GPU=False.")

            hf.gpu.init_kernels()

        self.use_GPU = use_GPU

    def get_binary_target(self, arr):
        output = []
        for i in range(len(arr)):
            # print("-" * 2)
            # print("input", X[i])
            # print("target", y[i])
            # print("output", outputs[i])
            if (arr[i] < 0.5):
                output.append(0)
            else:
                output.append(1)
        return output

    def get_acc(self, target, real):

        conf_matrix = confusion_matrix(target, real)

        tn = conf_matrix[0][0]
        fp = conf_matrix[0][1]
        fn = conf_matrix[1][0]
        tp = conf_matrix[1][1]
        total = len(target)

        # print(classification_report(target, real))

        acc = "{:.3f}".format((tp + tn) / total)
        return acc

    def stats(self, epoch, epoch_time, X_train, y_train,  X_test, y_test,   metrics_train_file
, metrics_valid_file):
        # Training Statistics
        train_real = self.get_binary_target(self.forward(X_train)[-1])
        train_loss = self.error(self.W, X_train, y_train)

        self.loss_train.append(train_loss)
        self.scores_train.append(self.get_acc(y_train, train_real))

        stats_train_output, stats_train_file_output = stats_output(
            "Training", epoch, y_train, train_real)

        # Print train metrics in a text file friendly for Excel
        stats_train_file_output += "{:.3f}\t{:.3f}".format(
            train_loss, epoch_time)
        print(stats_train_file_output, file=metrics_train_file)
        metrics_train_file.flush()

        # Test Statistics
        test_real = self.get_binary_target(self.forward(X_test)[-1])
        test_loss = self.error(self.W, X_test, y_test)

        self.loss_test.append(test_loss)
        self.scores_test.append(self.get_acc(y_test, test_real))

        stats_test_output, stats_test_file_output = stats_output(
            "Testing", epoch, y_test, test_real)

        # Print test metrics in a text file friendly for Excel
        stats_test_file_output += "{:.3f}\t{:.3f}".format(
            test_loss, epoch_time)
        print(stats_test_file_output, file=metrics_valid_file)
        metrics_valid_file.flush()
```

```
    def run_epochs(self, inputs, targets, folder, index, optimizer,
                   max_epochs=100, minibatch_size=None, test=None,
                   test_err=None, target_err=1e-6, plotting=False,
                   file_output=None, print_period=10):
        """Apply the given optimizer with a sequence of (mini)batches.

        :param inputs: input vectors (or a :class:`~.nonlinearities.Plant` that
            will generate the input vectors dynamically)
        :type inputs: :class:`~numpy:numpy.ndarray` or
            :class:`~.nonlinearities.Plant`
        :param targets: target vectors corresponding to each input vector (or
            None if a plant is being used)
        :type targets: :class:`~numpy:numpy.ndarray`
        :param optimizer: computes the weight update each epoch (see
            optimizers.py)
        :param int max_epochs: the maximum number of epochs to run
        :param int minibatch_size: the size of the minibatch to use in each epoch
            (or None to use full batches)
        :param tuple test: tuple of (inputs,targets) to use as the test data
            (if None then the same inputs and targets as training will be used)
        :param test_err: a custom error function to be applied to
            the test data (e.g., classification error)
        :type test_err: :class:`~.loss_funcs.LossFunction`
        :param float target_err: run will terminate if this test error is
            reached
        :param str file_output: output files from the run will use this as a
            prefix (if None then don't output files)
        :param bool plotting: if True then data from the run will be output to
            a file, which can be displayed via dataplotter.py
        :param int print_period: print out information about the run every `x`
            epochs
        """

        self.scores_train.clear()
        self.scores_test.clear()
        self.loss_train.clear()
        self.loss_test.clear()
        self.test_errs.clear()

        self.best_W = None
        self.best_error = None
        prefix = "HF" if file_output is None else file_output
        minibatch_size = minibatch_size or inputs.shape[0]
        plots = defaultdict(list)
        self.optimizer = optimizer

        # Print to a format friendly for excel
        metrics_train_file = open(
            folder + "metrics_train_file_" + str(index) + ".txt", 'w')
        metrics_valid_file = open(
            folder + "metrics_valid_file_" + str(index) + ".txt", 'w')

        print("f{0}_train_epoch\tf{0}_train_tn\tf{0}_train_fp\tf{0}_train_fn\tf{0}_train_tp\tf
{0}_train_acc\tf{0}_train_ppv\tf{0}_train_npv\tf{0}_train_sensitivity\tf{0}_train_specificity\
tf{0}_train_b_acc\tf{0}_train_loss\tf{0}_train_epoch_time\tf{0}_train_lr".format(index),
              file=metrics_train_file)
        print("f{0}_valid_epoch\tf{0}_valid_tn\tf{0}_valid_fp\tf{0}_valid_fn\tf{0}_valid_tp\tf
{0}_valid_acc\tf{0}_valid_ppv\tf{0}_valid_npv\tf{0}_valid_sensitivity\tf{0}_valid_specificity\
tf{0}_valid_b_acc\tf{0}_valid_loss\tf{0}_valid_epoch_time\tf{0}_valid_lr".format(index),
              file=metrics_valid_file)

        # Force to write immediately to file
        metrics_train_file.flush()
        metrics_valid_file.flush()

        max_train_acc = 0
        max_test_acc = 0

        for i in range(max_epochs):

            self.epoch = i
            # printing = print_period is not None and (i % print_period == 0 or
            #                                          self.debug)
```

```
            printing = False
            if printing:
                print("=" * 40)
                print("epoch", i)

            epoch_start = time.time()
            # run minibatches
            indices = self.rng.permutation(inputs.shape[0])

            for start in range(0, inputs.shape[0], minibatch_size):
                # generate minibatch and cache activations
                self.cache_minibatch(
                    inputs, targets, indices[start:start + minibatch_size])

                # validity checks
                if self.inputs.shape[-1] != self.shape[0]:
                    raise ValueError(
                        "Input dimension (%d) does not match number of input "
                        "nodes (%d)" % (self.inputs.shape[-1], self.shape[0]))
                if self.targets.shape[-1] != self.shape[-1]:
                    raise ValueError(
                        "Target dimension (%d) does not match number of "
                        "output nodes (%d)" % (self.targets.shape[-1],
                                               self.shape[-1]))

                assert self.activations[-1].dtype == self.dtype

                # compute update
                update = optimizer.compute_update(printing)

                assert update.dtype == self.dtype

                # apply mask
                if self.mask is not None:
                    update[self.mask] = 0

                # update weights
                self.W += update

                # invalidate cached activations (shouldn't be necessary,
                # but doesn't hurt)
                self.activations = None
                self.d_activations = None
                self.GPU_activations = None

            epoch_end = time.time()
            # compute test error
            if test is None:
                test_in, test_t = inputs, targets
            else:
                test_in, test_t = test[0], test[1]

            if test_err is None:
                err = self.error(self.W, test_in, test_t)
            else:
                output = self.forward(test_in, self.W)
                err = test_err.batch_loss(output, test_t)
            self.test_errs += [err]

            epoch_time = epoch_end - epoch_start

            self.stats(i, epoch_time, inputs, targets,
                       test[0], test[1], metrics_train_file, metrics_valid_file)

            if printing:
                print("test error", self.test_errs[-1])

            # save the weights with the best error
            if self.best_W is None or self.test_errs[-1] < self.best_error:
                self.best_W = self.W.copy()
                self.best_error = self.test_errs[-1]

            # dump plot data
```

```
            if plotting:
                plots["update norm"] += [np.linalg.norm(update)]
                plots["W norm"] += [np.linalg.norm(self.W)]
                plots["test error (log)"] += [self.test_errs[-1]]

                if hasattr(optimizer, "plots"):
                    plots.update(optimizer.plots)

                with open("%s_plots.pkl" % prefix, "wb") as f:
                    pickle.dump(plots, f)

            # dump weights
            if file_output is not None:
                np.save("%s_weights.npy" % prefix, self.W)

            # check for termination
            # if self.test_errs[-1] < target_err:
            #     if print_period is not None:
            #         print("target error reached")
            #     break

            # Overfitting
            # if test is not None and i > 10 and self.test_errs[-10] < self.test_errs[-1]:
            #     print("overfitting detected in epoch: " + str(i))

            # if test is not None and i > 10 and self.test_errs[-10] < self.test_errs[-1]:
            #     if print_period is not None:
            #         print("overfitting detected, terminating")
            #     break

            if max_test_acc < float(self.scores_test[-1]):
                max_train_acc = float(self.scores_train[-1])
                max_test_acc = float(self.scores_test[-1])

        return max_train_acc, max_test_acc

        # Create accuracy and loss plots
        # plt.clf()
        # plt.plot(self.scores_train, color='green',
        #          alpha=0.8, label='Train')
        # plt.plot(self.scores_test, color='magenta',
        #          alpha=0.8, label='Test')
        # plt.title("Accuracy over epochs", fontsize=14)
        # plt.xlabel('Epochs')
        # plt.legend(loc='upper left')
        # plt.savefig(folder + 'accuracy_plot_' + str(index) + '.png')

        # plt.clf()
        # plt.plot(self.loss_train, color='green', alpha=0.8, label='Train')
        # plt.plot(self.loss_test, color='magenta', alpha=0.8, label='Test')
        # plt.title("Loss over epochs", fontsize=14)
        # plt.xlabel('Epochs')
        # plt.legend(loc='upper left')
        # plt.savefig(folder + 'loss_plot_' + str(index) + '.png')

    def forward(self, inputs, params=None, deriv=False):
        """Compute layer activations for given input and parameters.

        :param inputs: input vectors (passed to first layer)
        :type inputs: :class:`~numpy:numpy.ndarray`
        :param params: parameter vector (weights) for the network (defaults to
            ``self.W``)
        :type params: :class:`~numpy:numpy.ndarray`
        :param bool deriv: if True then also compute the derivative of the
            activations
        """

        params = self.W if params is None else params

        if isinstance(inputs, hf.nl.Plant):
            inputs.reset()

        activations = [None for _ in range(self.n_layers)]
```

```
        if deriv:
            d_activations = [None for _ in range(self.n_layers)]

        for i in range(self.n_layers):
            if i == 0:
                if isinstance(inputs, hf.nl.Plant):
                    inputs = inputs(None)
                else:
                    inputs = inputs
            else:
                inputs = np.zeros((inputs.shape[0], self.shape[i]),
                                  dtype=self.dtype)
                for pre in self.back_conns[i]:
                    W, b = self.get_weights(params, (pre, i))
                    inputs += np.dot(activations[pre], W)
                    inputs += b
                    # note: we're applying a bias on each connection to a
                    # neuron (rather than one for each neuron). just because
                    # it's easier than tracking how many connections there are
                    # for each layer (but we could do it if it becomes
                    # important).
            activations[i] = self.layers[i].activation(inputs)

            if deriv:
                d_activations[i] = self.layers[i].d_activation(inputs,
                                                               activations[i])

        for i, a in enumerate(activations):
            if not np.all(np.isfinite(a)):
                raise OverflowError("Non-finite nonlinearity activation "
                                    "value (layer %d) \n %s" %
                                    (i, a[not np.isfinite(a)]))
        if deriv:
            return activations, d_activations

        return activations

    def error(self, W=None, inputs=None, targets=None):
        """Compute network error.

        :param W: network parameters (defaults to ``self.W``)
        :type W: :class:`~numpy:numpy.ndarray`
        :param inputs: input vectors (defaults to the cached (mini)batch for
            current epoch)
        :type inputs: :class:`~numpy:numpy.ndarray`
        :param targets: target vectors (defaults to the cached (mini)batch for
            current epoch)
        :type targets: :class:`~numpy:numpy.ndarray`
        """

        W = self.W if W is None else W
        inputs = self.inputs if inputs is None else inputs

        # get outputs
        if (W is self.W and inputs is self.inputs and
                self.activations is not None):
            # use cached activations
            activations = self.activations
        else:
            # compute activations
            activations = self.forward(inputs, W)

        # get targets
        if isinstance(inputs, hf.nl.Plant):
            # get targets from plant
            targets = inputs.get_vecs()[1]
        else:
            targets = self.targets if targets is None else targets

        # note: np.nan can be used in the target to specify places
        # where the target is not defined. those get translated to
        # zero error in the loss function.
```

```
        error = self.loss.batch_loss(activations, targets)

        return error

    def cache_minibatch(self, inputs, targets, minibatch=None):
        """Pick a subset of inputs and targets to use in minibatch, and cache
        the activations for that minibatch."""

        if minibatch is None:
            minibatch = np.arange(inputs.shape[0])

        if not isinstance(inputs, hf.nl.Plant):
            # inputs/targets are vectors
            self.inputs = inputs[minibatch]
            self.targets = targets[minibatch]

            # cache activations
            self.activations, self.d_activations = self.forward(self.inputs,
                                                                self.W,
                                                                deriv=True)
        else:
            # input is a dynamic plant
            if targets is not None:
                raise ValueError("Cannot specify targets when using dynamic "
                                 "plant to generate inputs (plant should "
                                 "generate targets itself)")

            # run plant to generate batch
            inputs.shape[0] = len(minibatch)
            self.activations, self.d_activations = self.forward(inputs, self.W,
                                                                deriv=True)
            self.inputs, self.targets = inputs.get_vecs()

        # cast to self.dtype
        if self.inputs.dtype != self.dtype:
            warnings.warn("Input dtype (%s) not equal to self.dtype (%s)" %
                          (self.inputs.dtype, self.dtype))
        self.inputs = np.asarray(self.inputs, dtype=self.dtype)
        self.targets = np.asarray(self.targets, dtype=self.dtype)
        self.activations = [np.asarray(a, dtype=self.dtype)
                            for a in self.activations]
        self.d_activations = [np.asarray(a, dtype=self.dtype)
                              for a in self.d_activations]
        self.d2_loss = self.loss.d2_loss(self.activations, self.targets)

        # allocate temporary space for intermediate values, to save on
        # memory allocations
        self.tmp_space = [np.zeros(a.shape, self.dtype)
                          for a in self.activations]

        if self.use_GPU:
            # TODO: we could just allocate these on the first timestep and
            # then do a copy rather than an allocation after that, if this
            # ever became a significant part of the computation time
            self.load_GPU_data()

    def load_GPU_data(self):
        """Load data for the current epoch onto GPU."""

        from pycuda import gpuarray

        # clear out old data (this would happen eventually on its own, but by
        # doing it first we make sure there is room on the GPU before
        # creating new arrays)
        if hasattr(self, "GPU_W"):
            del self.GPU_W
            del self.GPU_activations
            del self.GPU_d_activations
            del self.GPU_d2_loss
            del self.GPU_tmp_space

        self.GPU_W = gpuarray.to_gpu(self.W)
        self.GPU_activations = [gpuarray.to_gpu(a)
```

```
                               for a in self.activations]
        self.GPU_d_activations = [gpuarray.to_gpu(a)
                                  for a in self.d_activations]
        self.GPU_d2_loss = [gpuarray.to_gpu(a) if a is not None else None
                            for a in self.d2_loss]
        self.GPU_tmp_space = [gpuarray.empty(a.shape, self.dtype)
                              for a in self.activations]

    @staticmethod
    def J_dot(J, vec, transpose_J=False, out=None):
        """Compute the product of a Jacobian and some vector."""

        # In many cases the Jacobian is a diagonal matrix, so it is more
        # efficient to just represent it with the diagonal vector.  This
        # function just lets those two be used interchangeably.

        if J.ndim == 2:
            # note: the first dimension is the batch, so ndim==2 means
            # this is a vector representation
            if out is None:
                # passing out=None fails for some reason
                return np.multiply(J, vec)
            else:
                return np.multiply(J, vec, out=out)
        else:
            if transpose_J:
                J = np.transpose(J, (0, 2, 1))

            if out is None:
                # passing out=None fails for some reason
                return np.einsum("ijk,ik->ij", J, vec)

            if out is vec:
                tmp_vec = vec.copy()
            else:
                tmp_vec = vec

            return np.einsum("ijk,ik->ij", J, tmp_vec, out=out)

    def calc_grad(self):
        """Compute parameter gradient."""

        for l in self.layers:
            if l.stateful:
                raise TypeError("Cannot use neurons with internal state in "
                                "a one-step feedforward network; use "
                                "RNNet instead.")

        grad = np.zeros_like(self.W)

        # backpropagation
        # note: this uses the cached activations, so the forward
        # pass has already been run elsewhere

        # compute output error for each layer
        error = self.loss.d_loss(self.activations, self.targets)

        error = [np.zeros_like(self.activations[i]) if e is None else e
                 for i, e in enumerate(error)]

        deltas = [np.zeros_like(a) for a in self.activations]

        # backwards pass
        for i in range(self.n_layers - 1, -1, -1):
            for post in self.conns[i]:
                error[i] += np.dot(deltas[post],
                                   self.get_weights(self.W, (i, post))[0].T)

                W_grad, b_grad = self.get_weights(grad, (i, post))
                np.dot(self.activations[i].T, deltas[post], out=W_grad)
                np.sum(deltas[post], axis=0, out=b_grad)

            self.J_dot(self.d_activations[i], error[i], transpose_J=True,
```

```
                    out=deltas[i])

        grad /= self.inputs.shape[0]

        return grad

    def check_grad(self, calc_grad):
        """Check gradient via finite differences (for debugging)."""

        eps = 1e-6
        grad = np.zeros_like(calc_grad)
        inc_W = np.zeros_like(self.W)
        for i in range(len(self.W)):
            inc_W[i] = eps

            error_inc = self.error(self.W + inc_W, self.inputs, self.targets)
            error_dec = self.error(self.W - inc_W, self.inputs, self.targets)
            grad[i] = (error_inc - error_dec) / (2 * eps)

            inc_W[i] = 0
        try:
            assert np.allclose(calc_grad, grad, rtol=1e-3)
        except AssertionError:
            print("calc_grad")
            print(calc_grad)
            print("finite grad")
            print(grad)
            print("calc_grad - finite grad")
            print(calc_grad - grad)
            print("calc_grad / finite grad")
            print(calc_grad / grad)
            input("Paused (press enter to continue)")

    def calc_G(self, v, damping=0, out=None):
        """Compute Gauss-Newton matrix-vector product."""

        if out is None:
            Gv = np.zeros(self.W.size, dtype=self.dtype)
        else:
            Gv = out
            Gv.fill(0)

        # R forward pass
        R_activations = [np.zeros_like(a) for a in self.activations]
        for i in range(1, self.n_layers):
            for pre in self.back_conns[i]:
                vw, vb = self.get_weights(v, (pre, i))
                Ww, _ = self.get_weights(self.W, (pre, i))

                R_activations[i] += np.dot(self.activations[pre], vw,
                                           out=self.tmp_space[i])
                R_activations[i] += vb
                R_activations[i] += np.dot(R_activations[pre], Ww,
                                           out=self.tmp_space[i])

            self.J_dot(self.d_activations[i], R_activations[i],
                       out=R_activations[i])

        # backward pass
        R_error = R_activations

        for i in range(self.n_layers - 1, -1, -1):
            if self.d2_loss[i] is not None:
                # note: R_error[i] is already set to R_activations[i]
                R_error[i] *= self.d2_loss[i]
            else:
                R_error[i].fill(0)

            for post in self.conns[i]:
                W, _ = self.get_weights(self.W, (i, post))

                R_error[i] += np.dot(R_error[post], W.T,
                                     out=self.tmp_space[i])
```

```
            W_g, b_g = self.get_weights(Gv, (i, post))
            np.dot(self.activations[i].T, R_error[post], out=W_g)
            np.sum(R_error[post], axis=0, out=b_g)

        self.J_dot(self.d_activations[i], R_error[i],
                   out=R_error[i], transpose_J=True)

    Gv /= len(self.inputs)

    Gv += damping * v  # Tikhonov damping

    return Gv

def GPU_calc_G(self, v, damping=0, out=None):
    """Compute Gauss-Newton matrix-vector product on GPU."""

    from pycuda import gpuarray

    if out is None or not isinstance(out, gpuarray.GPUArray):
        Gv = gpuarray.zeros(self.W.shape, self.dtype)
    else:
        Gv = out
        Gv.fill(0)

    if not isinstance(v, gpuarray.GPUArray):
        GPU_v = gpuarray.to_gpu(v)
    else:
        GPU_v = v

    # R forward pass
    R_activations = self.GPU_tmp_space

    for i in range(self.n_layers):
        R_activations[i].fill(0)
        for pre in self.back_conns[i]:
            vw, vb = self.get_weights(GPU_v, (pre, i))
            Ww, _ = self.get_weights(self.GPU_W, (pre, i))

            hf.gpu.dot(self.GPU_activations[pre], vw,
                       out=R_activations[i], increment=True)
            hf.gpu.iadd(R_activations[i], vb)
            hf.gpu.dot(R_activations[pre], Ww,
                       out=R_activations[i], increment=True)

        hf.gpu.J_dot(self.GPU_d_activations[i], R_activations[i],
                     out=R_activations[i])

    # backward pass
    R_error = R_activations

    for i in range(self.n_layers - 1, -1, -1):
        if self.GPU_d2_loss[i] is not None:
            # note: R_error[i] is already set to R_activations[i]
            R_error[i] *= self.GPU_d2_loss[i]
        else:
            R_error[i].fill(0)

        for post in self.conns[i]:
            W, _ = self.get_weights(self.GPU_W, (i, post))
            W_g, b_g = self.get_weights(Gv, (i, post))

            hf.gpu.dot(R_error[post], W, transpose_b=True,
                       out=R_error[i], increment=True)

            hf.gpu.dot(self.GPU_activations[i], R_error[post],
                       transpose_a=True, out=W_g)

            hf.gpu.sum_cols(R_error[post], out=b_g)

        hf.gpu.J_dot(self.GPU_d_activations[i], R_error[i], out=R_error[i],
                     transpose_J=True)
```

```
        # Tikhonov damping and batch mean
        Gv._axpbyz(1.0 / len(self.inputs), GPU_v, damping, Gv)

        if isinstance(v, gpuarray.GPUArray):
            return Gv
        else:
            return Gv.get(out, pagelocked=True)

    def check_J(self):
        """Compute the Jacobian of the network via finite differences."""

        eps = 1e-6
        N = self.W.size

        # compute the Jacobian
        J = [None for _ in self.layers]
        inc_i = np.zeros_like(self.W)
        for i in range(N):
            inc_i[i] = eps

            inc = self.forward(self.inputs, self.W + inc_i)
            dec = self.forward(self.inputs, self.W - inc_i)

            for l in range(self.n_layers):
                J_i = (inc[l] - dec[l]) / (2 * eps)
                if J[l] is None:
                    J[l] = J_i[..., None]
                else:
                    J[l] = np.concatenate((J[l], J_i[..., None]), axis=-1)

            inc_i[i] = 0

        return J

    def check_G(self, calc_G, v, damping=0):
        """Check Gv calculation via finite differences (for debugging)."""

        # compute Jacobian
        J = self.check_J()

        # second derivative of loss function
        L = self.loss.d2_loss(self.activations, self.targets)
        # TODO: check loss via finite differences

        G = np.sum([np.einsum("aji,aj,ajk->ik", J[l], L[l], J[l])
                    for l in range(self.n_layers) if L[l] is not None], axis=0)

        # divide by batch size
        G /= self.inputs.shape[0]

        Gv = np.dot(G, v)
        Gv += damping * v

        try:
            assert np.allclose(calc_G, Gv, rtol=1e-3)
        except AssertionError:
            print("calc_G")
            print(calc_G)
            print("finite G")
            print(Gv)
            print("calc_G - finite G")
            print(calc_G - Gv)
            print("calc_G / finite G")
            print(calc_G / Gv)
            input("Paused (press enter to continue)")

    def init_weights(self, shapes, coeff=1.0, biases=0.0, init_type="sparse"):
        """Weight initialization, given shapes of weight matrices.

        Note: coeff, biases, and init_type can be specified by the
        `W_init_params` dict in :class:`.FFNet`.  Each can be
        specified as a single value (for all matrices) or as a list giving a
        value for each matrix.
```

```
        :param list shapes: list of (pre,post) shapes for each weight matrix
        :param float coeff: scales the magnitude of the connection weights
        :param float biases: bias values for the post of each matrix
        :param str init_type: type of initialization to use (currently supports
            'sparse', 'uniform', 'gaussian')
        """

        # if given single parameters, expand for all matrices
        if isinstance(coeff, (int, float)):
            coeff = [coeff] * len(shapes)
        if isinstance(biases, (int, float)):
            biases = [biases] * len(shapes)
        if isinstance(init_type, str):
            init_type = [init_type] * len(shapes)

        W = [np.zeros((pre + 1, post), dtype=self.dtype)
             for pre, post in shapes]

        for i, s in enumerate(shapes):
            if init_type[i] == "sparse":
                # sparse initialization (from martens)
                num_conn = 15

                for j in range(s[1]):
                    # pick num_conn random pre neurons
                    indices = self.rng.choice(np.arange(s[0]),
                                              size=min(num_conn, s[0]),
                                              replace=False)

                    # connect to post
                    W[i][indices, j] = self.rng.randn(indices.size) * coeff[i]
            elif init_type[i] == "uniform":
                W[i][:-1] = self.rng.uniform(-coeff[i] / np.sqrt(s[0]),
                                             coeff[i] / np.sqrt(s[0]),
                                             (s[0], s[1]))
            elif init_type[i] == "gaussian":
                W[i][:-1] = self.rng.randn(s[0], s[1]) * coeff[i]
            else:
                raise ValueError("Unknown weight initialization (%s)"
                                 % init_type)

            # set biases
            W[i][-1, :] = biases[i]

        W = np.concatenate([w.flatten() for w in W])

        return W

    def compute_offsets(self):
        """Precompute offsets for layers in the overall parameter vector."""

        self.offsets = {}
        offset = 0
        for pre in self.conns:
            for post in self.conns[pre]:
                n_params = (self.shape[pre] + 1) * self.shape[post]
                self.offsets[(pre, post)] = (
                    offset,
                    offset + n_params - self.shape[post],
                    offset + n_params)
                offset += n_params

        return offset

    def get_weights(self, params, conn):
        """Get weight matrix for a connection from overall parameter vector."""

        if conn not in self.offsets:
            return None

        offset, W_end, b_end = self.offsets[conn]
        W = params[offset:W_end]
```

```
        b = params[W_end:b_end]
        return W.reshape((self.shape[conn[0]], self.shape[conn[1]])), b

    def init_loss(self, loss_type):
        """Set the loss type for this network to the given
        :class:`~.loss_funcs.LossFunction` (or a list of functions can be
        passed to create a :class:`~.loss_funcs.LossSet`)."""

        if isinstance(loss_type, (list, tuple)):
            tmp = loss_type
        else:
            tmp = [loss_type]

        for t in tmp:
            if not isinstance(t, hf.loss_funcs.LossFunction):
                raise TypeError("loss_type (%s) must be an instance of "
                                "LossFunction" % t)

            # sanity checks
            if (isinstance(t, hf.loss_funcs.CrossEntropy) and
                    np.any(self.layers[-1].activation(
                        np.linspace(-80, 80, 100)[None, :]) <= 0)):
                # this won't catch everything, but hopefully a useful warning
                raise ValueError("Must use positive activation function "
                                 "with cross-entropy error")
            if (isinstance(t, hf.loss_funcs.CrossEntropy) and
                    not isinstance(self.layers[-1], hf.nl.Softmax)):
                warnings.warn("Softmax should probably be used with "
                              "cross-entropy error")

        if isinstance(loss_type, (list, tuple)):
            self.loss = hf.loss_funcs.LossSet(loss_type)
        else:
            self.loss = loss_type

    def _run_epoch(self, inputs, targets, minibatch_size=None):
        """A stripped down version of run_epochs that just does the update
        without any overhead.

        Can be used for optimizers where the cost to compute an update is
        very cheap, in which case the overhead (e.g., computing test error,
        saving weights, outputting data for plotting, etc.) becomes
        non-negligible.
        """

        minibatch_size = minibatch_size or inputs.shape[0]
        indices = self.rng.permutation(inputs.shape[0])
        for start in range(0, inputs.shape[0], minibatch_size):
            # generate minibatch and cache activations
            self.cache_minibatch(
                inputs, targets, indices[start:start + minibatch_size])

            # compute update
            self.W += self.optimizer.compute_update(False)

    @property
    def optimizer(self):
        return self._optimizer

    @optimizer.setter
    def optimizer(self, o):
        self._optimizer = o
        o.net = self
```

Code Snippet A.4. The FFNet (HFO algorithm) of the MLP network with HFO optimizer.

Appendix B

CNN Implementations

The following files are required to execute the CNN implementations for the Adam, NewtonCG, and SGD optimizers.

B.1 train.py

This is the file is the main and is responsible for reading the user's arguments to create the Session with the network's parameters. Also, the implementation for the Adam and SGD optimizers is located here.

```
from datetime import datetime
from utilities import read_data, predict, ConfigClass, normalize_and_reshape, stats_output, st
ats_output3
from newton_cg import newton_cg
from net.net import CNN
import argparse
import math
import time
import pdb
import numpy as np
import tensorflow as tf
import os
from sklearn.metrics import confusion_matrix, classification_report
import tf_slim as slim
tf.compat.v1.disable_eager_execution()

def parse_args():
    parser = argparse.ArgumentParser(description='Newton method on DNN')
    parser.add_argument('--C', dest='C',
                        help='regularization term, or so-called weight decay where' +
                        'weight_decay = lr/(C*num_of_samples) in this implementation',
                        default=0.01, type=float)

    # Newton method arguments
    parser.add_argument('--GNsize', dest='GNsize',
                        help='number of samples for estimating Gauss-Newton matrix',
                        default=4096, type=int)
    parser.add_argument('--iter_max', dest='iter_max',
                        help='the maximal number of Newton iterations',
                        default=100, type=int)
    parser.add_argument('--xi', dest='xi',
                        help='the tolerance in the relative stopping condition for CG',
                        default=0.1, type=float)
    parser.add_argument('--drop', dest='drop',
                        help='the drop constants for the LM method',
                        default=2/3, type=float)
    parser.add_argument('--boost', dest='boost',
                        help='the boost constants for the LM method',
                        default=3/2, type=float)
    parser.add_argument('--eta', dest='eta',
                                        help='parameter for the line search stopping condition',
                                        default=0.0001, type=float)
    parser.add_argument('--CGmax', dest='CGmax',
                        help='the maximal number of CG iterations',
                        default=250, type=int)
    parser.add_argument('--lambda', dest='_lambda',
                        help='the initial lambda for the LM method',
                        default=1, type=float)
```

```
    # SGD arguments
    parser.add_argument('--epoch_max', dest='epoch',
                        help='number of training epoch',
                        default=500, type=int)
    parser.add_argument('--lr', dest='lr',
                        help='learning rate',
                        default=0.01, type=float)
    parser.add_argument('--decay', dest='lr_decay',
                        help='learning rate decay over each mini-batch update',
                        default=0, type=float)
    parser.add_argument('--momentum', dest='momentum',
                        help='momentum of learning',
                        default=0, type=float)

    # Model training arguments
    parser.add_argument('--bsize', dest='bsize',
                        help='batch size to evaluate stochastic gradient, Gv, etc. Since the \
                      sampled data for computing Gauss-Newton matrix and etc.might not fit \
                      into memeory for one time, we will split the data into several \
                      segements and average over them.',
                        default=1024, type=int)
    parser.add_argument('--net', dest='net',
                                      help='classifier type',
                                      default='CNN_4layers', type=str)
    parser.add_argument('--train_set', dest='train_set',
                        help='provide the directory of .mat file for training',
                        default='data/mnist-demo.mat', type=str)
    parser.add_argument('--val_set', dest='val_set',
                        help='provide the directory of .mat file for validation',
                        default=None, type=str)
    parser.add_argument('--model', dest='model_file',
                        help='model saving address',
                        default='./saved_model/model.ckpt', type=str)
    parser.add_argument('--log', dest='log_file',
                                      help='log saving directory',
                                      default='./running_log/logger.log', type=str)
    parser.add_argument('--screen_log_only', dest='screen_log_only',
                        help='screen printing running log instead of storing it',
                        action='store_true')
    parser.add_argument('--optim', '-optim',
                        help='which optimizer to use: SGD, Adam or NewtonCG',
                        default='NewtonCG', type=str)
    parser.add_argument('--loss', dest='loss',
                        help='which loss function to use: MSELoss or CrossEntropy',
                        default='MSELoss', type=str)
    parser.add_argument('--dim', dest='dim', nargs='+', help='input dimension of data,' +
                        'shape must be:  height width num_channels',
                                      default=[32, 32, 3], type=int)
    parser.add_argument('--seed', dest='seed', help='a nonnegative integer for \
                        reproducibility', type=int)
    args = parser.parse_args()
    return args


args = parse_args()


def init_model(param):
    init_ops = []
    for p in param:
        if 'kernel' in p.name:
            weight = np.random.standard_normal(
                p.shape) * np.sqrt(2.0 / ((np.prod(p.get_shape().as_list()[:-1]))))
            opt = tf.compat.v1.assign(p, weight)
        elif 'bias' in p.name:
            zeros = np.zeros(p.shape)
            opt = tf.compat.v1.assign(p, zeros)
        init_ops.append(opt)
    return tf.group(*init_ops)
```

```
def gradient_trainer(folder_dir, config, sess, network, full_batch, val_batch, saver, num_clas
ses, test_network):
    x, y, loss, outputs,  = network

    global_step = tf.Variable(
        initial_value=0, trainable=False, name='global_step')
    learning_rate = tf.compat.v1.placeholder(
        tf.float32, shape=[], name='learning_rate')

    # Probably not a good way to add regularization.
    # Just to confirm the implementation is the same as MATLAB.
    reg = 0.0
    param = tf.compat.v1.trainable_variables()
    for p in param:
        reg = reg + tf.reduce_sum(input_tensor=tf.pow(p, 2))
    reg_const = 1/(2*config.C)
    batch_size = tf.compat.v1.cast(tf.shape(x)[0], tf.float32)
    loss_with_reg = reg_const*reg + loss/batch_size

    if config.optim == 'SGD':
        optimizer = tf.compat.v1.train.MomentumOptimizer(
            learning_rate=learning_rate,
            momentum=config.momentum).minimize(
            loss_with_reg,
            global_step=global_step)
    elif config.optim == 'Adam':
        optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate=learning_rate,
                                                     beta1=0.9,
                                                     beta2=0.999,
                                                     epsilon=1e-08).minimize(
            loss_with_reg,
            global_step=global_step)

    train_inputs, train_labels = full_batch
    num_data = train_labels.shape[0]
    num_iters = math.ceil(num_data/config.bsize)

    print(config.args)
    if not config.screen_log_only:
        log_file = open(config.log_file, 'w')
        print(config.args, file=log_file)
        # Force to write immediately to file
        log_file.flush()

    sess.run(tf.compat.v1.global_variables_initializer())

    # Print to a format friendly for excel
    metrics_train_file = open(
        folder_dir + "running_log/metrics_train_file.txt", 'w')
    metrics_valid_file = open(
        folder_dir + "running_log/metrics_valid_file.txt", 'w')

    if num_classes == 2:
        print("epoch\ttn\tfp\tfn\ttp\tacc\tppv\tnpv\tsensitivity\tspecificity\tb_acc\tloss\tep
och_time\tlr",
              file=metrics_train_file)
        print("epoch\ttn\tfp\tfn\ttp\tacc\tppv\tnpv\tsensitivity\tspecificity\tb_acc\tloss\tep
och_time\tlr",
              file=metrics_valid_file)
    elif num_classes == 3:
        print("epoch\ttn_0\tfp_0\tfn_0\ttp_0\tacc_0\tppv_0\tnpv_0\tsensitivity_0\tspecificity_
0\tb_acc_0\ttn_1\tfp_1\tfn_1\ttp_1\tacc_1\tppv_1\tnpv_1\tsensitivity_1\tspecificity_1\tb_acc_1
\ttn_2\tfp_2\tfn_2\ttp_2\tacc_2\tppv_2\tnpv_2\tsensitivity_2\tspecificity_2\tb_acc_2\tacc\tlos
s\tepoch_time\tlr",
              file=metrics_train_file)
        print("epoch\ttn_0\tfp_0\tfn_0\ttp_0\tacc_0\tppv_0\tnpv_0\tsensitivity_0\tspecificity_
0\tb_acc_0\ttn_1\tfp_1\tfn_1\ttp_1\tacc_1\tppv_1\tnpv_1\tsensitivity_1\tspecificity_1\tb_acc_1
\ttn_2\tfp_2\tfn_2\ttp_2\tacc_2\tppv_2\tnpv_2\tsensitivity_2\tspecificity_2\tb_acc_2\tacc\tlos
s\tepoch_time\tlr",
              file=metrics_valid_file)
    else:
        print("epoch\tacc\tloss\tepoch_time\tlr",
              file=metrics_train_file)
```

```
        print("epoch\tacc\tloss\tepoch_time\tlr",
              file=metrics_valid_file)

        # Force to write immediately to file
    metrics_train_file.flush()
    metrics_valid_file.flush()

    print('-------------- initializing network by methods in He et al. (2015) --------------')
    param = tf.compat.v1.trainable_variables()
    sess.run(init_model(param))

    total_running_time = 0.0
    best_acc = 0.0
    lr = config.lr

    for epoch in range(0, args.epoch):

        loss_avg = 0.0
        start = time.time()

        for i in range(num_iters):

            load_time = time.time()
            # randomly select the batch
            idx = np.random.choice(np.arange(0, num_data),
                                   size=config.bsize, replace=False)

            batch_input = train_inputs[idx]
            batch_labels = train_labels[idx]
            batch_input = np.ascontiguousarray(batch_input)
            batch_labels = np.ascontiguousarray(batch_labels)
            config.elapsed_time += time.time() - load_time

            step, _, batch_loss = sess.run(
                [global_step, optimizer, loss_with_reg],
                feed_dict={x: batch_input, y: batch_labels, learning_rate: lr}
            )

            # print initial loss
            if epoch == 0 and i == 0:
                output_str = 'initial f (reg + avg. loss of 1st batch): {:.3f}'.format(
                    batch_loss)
                print(output_str)
                if not config.screen_log_only:
                    print(output_str, file=log_file)
                    # Force to write immediately to file
                    log_file.flush()

            loss_avg = loss_avg + batch_loss
            # print log every 10% of the iterations
            if i % math.ceil(num_iters/10) == 0:
                end = time.time()
                output_str = 'Epoch {}: {}/{} | loss {:.4f} | lr {:.6} | elapsed time {:.3f}'\
                    .format(epoch, i, num_iters, batch_loss, lr, end-start)
                print(output_str)
                if not config.screen_log_only:
                    print(output_str, file=log_file)
                    # Force to write immediately to file
                    log_file.flush()

            # adjust learning rate for SGD by inverse time decay
            if args.optim != 'Adam':
                lr = config.lr/(1 + args.lr_decay*step)

        # exclude data loading time for fair comparison
        epoch_end = time.time() - config.elapsed_time
        total_running_time += epoch_end - start
        config.elapsed_time = 0.0

        if val_batch is None:
            output_str = 'In epoch {} train loss: {:.3f} | epoch time {:.3f}'\
                .format(epoch, loss_avg/(i+1), epoch_end-start)
        else:
```

```
            if test_network == None:
                val_loss, val_acc, _ = predict(
                    sess,
                    network=(x, y, loss, outputs),
                    test_batch=val_batch,
                    bsize=config.bsize
                )
            else:
                # A separate test network part have been done...
                val_loss, val_acc, _ = predict(
                    sess,
                    network=test_network,
                    test_batch=val_batch,
                    bsize=config.bsize
                )

            output_str = 'In epoch {} train loss: {:.3f} | val loss: {:.3f} | val accuracy: {:
.3f}% | epoch time {:.3f}'\
                .format(epoch, loss_avg/(i+1), val_loss, val_acc*100, epoch_end-start)

        # Find the best accuracy till now
        if val_acc > best_acc:
            best_acc = val_acc
            checkpoint_path = config.model_file
            save_path = saver.save(sess, checkpoint_path)
            print('Saved best model in {}'.format(save_path))

        # Training Statistics
        avg_train_loss, avg_train_acc, train_real = predict(
            sess, network, full_batch, config.bsize)
        train_target = np.argmax(full_batch[1], axis=1)

        # Print train metrics in a text file friendly for Excel
        if num_classes == 2:
            stats_train_output, stats_train_file_output = stats_output(
                "Training", epoch, train_target, train_real)
            stats_train_file_output += "{:.3f}\t{:.3f}\t{:.5f}".format(
                loss_avg/(i+1), epoch_end-start, lr)
        elif num_classes == 3:
            stats_train_output, stats_train_file_output = stats_output3(
                "Training", epoch, train_target, train_real)
            stats_train_file_output += "{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}".format(
                avg_train_acc, loss_avg / (i + 1), epoch_end - start, lr)
        else:
            stats_train_file_output = "{}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}".format(
                epoch, avg_train_acc, loss_avg / (i + 1), epoch_end - start, lr)
            stats_train_output = stats_train_file_output

        print(stats_train_file_output,  file=metrics_train_file)

        # Force to write immediately to file
        metrics_train_file.flush()

        # Val;idation Statistics
        avg_val_loss, avg_val_acc, val_real = predict(
            sess, network, val_batch, config.bsize)
        val_target = np.argmax(val_batch[1], axis=1)

        # Print validation metrics in a text file friendly for Excel
        if num_classes == 2:
            stats_valid_output, stats_valid_file_output = stats_output(
                "Validation", epoch, val_target, val_real)
            stats_valid_file_output += "{:.3f}\t{:.3f}\t{:.5f}".format(
                val_loss, epoch_end-start, lr)
        elif num_classes == 3:
            stats_valid_output, stats_valid_file_output = stats_output3(
                "Validation", epoch, val_target, val_real)
            stats_valid_file_output += "{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}".format(
                avg_val_acc, val_loss, epoch_end - start, lr)
        else:
            stats_valid_file_output = "{}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}".format(
                epoch, avg_val_acc, val_loss, epoch_end - start, lr)
            stats_valid_output = stats_valid_file_output
```

```
        print(stats_valid_file_output,  file=metrics_valid_file)

        # Force to write immediately to file
        metrics_valid_file.flush()

        # Print all statistics
        print(output_str)
        print(stats_train_output)
        print(stats_valid_output)
        if not config.screen_log_only:
            print(output_str, file=log_file)
            print(stats_train_output, file=log_file)
            print(stats_valid_output, file=log_file)

            # Force to write immediately to file
            log_file.flush()

    if val_batch is None:
        checkpoint_path = config.model_file
        save_path = saver.save(sess, checkpoint_path)
        print('Model at the last iteration saved in {}\r\n'.format(save_path))
        output_str = 'total running time {:.3f}s'.format(total_running_time)
    else:
        output_str = 'Final acc: {:.3f}% | best acc {:.3f}% | total running time {:.3f}s'\
            .format(val_acc*100, best_acc*100, total_running_time)

    print(output_str)
    if not config.screen_log_only:
        print(output_str, file=log_file)
        log_file.flush()
        log_file.close()


def newton_trainer(folder_dir, config, sess, network, full_batch, val_batch, saver, num_classe
s, test_network):

    _, _, loss, outputs = network
    newton_solver = newton_cg(config, sess, outputs, loss)
    sess.run(tf.compat.v1.global_variables_initializer())

    print('-------------- initializing network by methods in He et al. (2015) --------------')
    param = tf.compat.v1.trainable_variables()

    print("\n\n-------------- Model Architecture --------------\n")
    slim.model_analyzer.analyze_vars(param, print_info=True)
    print("\n\n\n")

    sess.run(init_model(param))
    newton_solver.newton(folder_dir, full_batch, val_batch,
                         saver, network, num_classes, test_network)


def main():
    # Number of maximum cores to use in the server
    cores_to_use = 4

    full_batch, num_cls, label_enum = read_data(
        filename=args.train_set, dim=args.dim)

    if args.val_set is None:
        print('No validation set is provided. Will output model at the last iteration.')
        val_batch = None
    else:
        val_batch, _, _ = read_data(
            filename=args.val_set, dim=args.dim, label_enum=label_enum)

    num_data = full_batch[0].shape[0]

    config = ConfigClass(args, num_data, num_cls)

    if isinstance(config.seed, int):
```

```
        tf.compat.v1.random.set_random_seed(config.seed)
        np.random.seed(config.seed)

    if config.net in ('CNN_4layers', 'CNN_7layers', 'VGG11', 'VGG13', 'VGG16', 'VGG19'):
        x, y, outputs = CNN(config.net, num_cls, config.dim)
        test_network = None
    else:
        raise ValueError('Unrecognized training model')

    if config.loss == 'MSELoss':
        loss = tf.reduce_sum(input_tensor=tf.pow(outputs-y, 2))
    else:
        loss = tf.reduce_sum(
            input_tensor=tf.nn.softmax_cross_entropy_with_logits(logits=outputs, labels=y))

    network = (x, y, loss, outputs)

    sess_config = tf.compat.v1.ConfigProto(
        allow_soft_placement=True, intra_op_parallelism_threads=cores_to_use, inter_op_paralle
lism_threads=cores_to_use)
    sess_config.gpu_options.allow_growth = True

    with tf.compat.v1.Session(config=sess_config) as sess:

        full_batch[0], mean_tr = normalize_and_reshape(
            full_batch[0], dim=config.dim, mean_tr=None)
        if val_batch is not None:
            val_batch[0], _ = normalize_and_reshape(
                val_batch[0], dim=config.dim, mean_tr=mean_tr)

        param = tf.compat.v1.trainable_variables()

        mean_param = tf.compat.v1.get_variable(name='mean_tr', initializer=mean_tr, trainable=
False,
                                               validate_shape=True, use_resource=False)
        label_enum_var = tf.compat.v1.get_variable(name='label_enum', initializer=label_enum,
trainable=False,
                                                   validate_shape=True, use_resource=False)
        saver = tf.compat.v1.train.Saver(var_list=param+[mean_param])

        # Create directory for logging
        date = datetime.now().strftime("%d-%m-%Y_%H-%M-%S")
        dim_str = str(len(config.dim) - 1) + "D"
        if config.optim in ('SGD', 'Adam'):
            epochs_str = str(args.epoch) + "epoch_"
        else:
            epochs_str = str(args.iter_max) + "epoch_"
        num_classes = max(np.argmax(full_batch[1], axis=1)) + 1
        if (num_classes == 2):
            cls = "_AD_NC_"
        else:
            cls = "_AD_MCI_NC_"

        folder_dir = './Experiments/CNN_' + config.optim + "/" + dim_str + cls + \
            epochs_str + date + "/"

        # Create model direcotry
        saved_model_dir = folder_dir + 'saved_model/'
        if not os.path.exists(saved_model_dir):
            os.makedirs(saved_model_dir)

        config.model_file = saved_model_dir + 'model.ckpt'
        print(config.model_file)

        # Create log direcotry
        log_dir = folder_dir + 'running_log/'
        if not os.path.exists(log_dir):
            os.makedirs(log_dir)

        config.log_file = log_dir + 'logger.log'
        print(config.log_file)

        # Copy net.py
```

```
        if config.net in ('CNN_4layers', 'CNN_7layers'):
            os.popen('cp ./net/net.py ' + folder_dir + "net.py")
        else:
            os.popen('cp ./net/vgg.py ' + folder_dir + "vgg.py")

        # Call optimizers
        if config.optim in ('SGD', 'Adam'):
            gradient_trainer(folder_dir,
                             config, sess, network, full_batch, val_batch, saver, num_classes,
 test_network)
        elif config.optim == 'NewtonCG':
            newton_trainer(folder_dir,
                           config, sess, network, full_batch, val_batch, saver, num_classes, t
est_network=test_network)


if __name__ == '__main__':
    main()
```

Code Snippet B.1. Main for the CNN implementations. (train.py)

B.2 newton_cg.py

This is the implementation of the NewtonCG algorithm.

```
import pdb
import tensorflow as tf
import time
import numpy as np
import os
import math
from utilities import predict, stats_output, stats_output3


def Rop(f, weights, v):
    """Implementation of R operator
    Args:
            f: any function of weights
            weights: list of tensors.
            v: vector for right multiplication
    Returns:
            Jv: Jaccobian vector product, length same as
                    the number of output of f
    """
    if type(f) == list:
        u = [tf.zeros_like(ff) for ff in f]
    else:
        u = tf.zeros_like(f)  # dummy variable
    g = tf.gradients(ys=f, xs=weights, grad_ys=u)
    return tf.gradients(ys=g, xs=u, grad_ys=v)


def Gauss_Newton_vec(outputs, loss, weights, v):
    """Implements Gauss-Newton vector product.
    Args:
            loss: Loss function.
            outputs: outputs of the last layer (pre-softmax).
            weights: Weights, list of tensors.
            v: vector to be multiplied with Gauss Newton matrix
    Returns:
            J'BJv: Guass-Newton vector product.
    """
    # Validate the input
    if type(weights) == list:
        if len(v) != len(weights):
            raise ValueError("weights and v must have the same length.")

    grads_outputs = tf.gradients(ys=loss, xs=outputs)
```

```
    BJv = Rop(grads_outputs, weights, v)
    JBJv = tf.gradients(ys=outputs, xs=weights, grad_ys=BJv)
    return JBJv


class newton_cg(object):
    def __init__(self, config, sess, outputs, loss):
        """
        initialize operations and vairables that will be used in newton
        args:
                sess: tensorflow session
                outputs: output of the neural network (pre-softmax layer)
                loss: function to calculate loss
        """
        super(newton_cg, self).__init__()
        self.sess = sess
        self.config = config
        self.outputs = outputs
        self.loss = loss
        self.param = tf.compat.v1.trainable_variables()

        self.CGiter = 0
        FLOAT = tf.float32
        model_weight = self.vectorize(self.param)

        # initial variable used in CG
        zeros = tf.zeros(model_weight.get_shape(), dtype=FLOAT)
        self.r = tf.Variable(zeros, dtype=FLOAT, trainable=False)
        self.v = tf.Variable(zeros, dtype=FLOAT, trainable=False)
        self.s = tf.Variable(zeros, dtype=FLOAT, trainable=False)
        self.g = tf.Variable(zeros, dtype=FLOAT, trainable=False)
        # initial Gv, f for method minibatch
        self.Gv = tf.Variable(zeros, dtype=FLOAT, trainable=False)
        self.f = tf.Variable(0., dtype=FLOAT, trainable=False)

        # rTr, cgtol and beta to be used in CG
        self.rTr = tf.Variable(0., dtype=FLOAT, trainable=False)
        self.cgtol = tf.Variable(0., dtype=FLOAT, trainable=False)
        self.beta = tf.Variable(0., dtype=FLOAT, trainable=False)

        # placeholder alpha, old_alpha and lambda
        self.alpha = tf.compat.v1.placeholder(FLOAT, shape=[])
        self.old_alpha = tf.compat.v1.placeholder(FLOAT, shape=[])
        self._lambda = tf.compat.v1.placeholder(FLOAT, shape=[])

        self.num_grad_segment = math.ceil(
            self.config.num_data/self.config.bsize)
        self.num_Gv_segment = math.ceil(self.config.GNsize/self.config.bsize)

        cal_loss, cal_lossgrad, cal_lossGv, \
            add_reg_avg_loss, add_reg_avg_grad, add_reg_avg_Gv, \
            zero_loss, zero_grad, zero_Gv = self._ops_in_minibatch()

        # initial operations that will be used in minibatch and newton
        self.cal_loss = cal_loss
        self.cal_lossgrad = cal_lossgrad
        self.cal_lossGv = cal_lossGv
        self.add_reg_avg_loss = add_reg_avg_loss
        self.add_reg_avg_grad = add_reg_avg_grad
        self.add_reg_avg_Gv = add_reg_avg_Gv
        self.zero_loss = zero_loss
        self.zero_grad = zero_grad
        self.zero_Gv = zero_Gv

        self.CG, self.update_v = self._CG()
        self.init_cg_vars = self._init_cg_vars()
        self.update_gs = tf.tensordot(self.s, self.g, axes=1)
        self.update_sGs = 0.5 * \
            tf.tensordot(self.s, -self.g-self.r-self._lambda*self.s, axes=1)
        self.update_model = self._update_model()
        self.gnorm = self.calc_norm(self.g)
```

```
    def vectorize(self, tensors):
        if isinstance(tensors, list) or isinstance(tensors, tuple):
            vector = [tf.reshape(tensor, [-1]) for tensor in tensors]
            return tf.concat(vector, 0)
        else:
            return tensors

    def inverse_vectorize(self, vector, param):
        if isinstance(vector, list):
            return vector
        else:
            tensors = []
            offset = 0
            num_total_param = np.sum(
                [np.prod(p.shape.as_list()) for p in param])
            for p in param:
                numel = np.prod(p.shape.as_list())
                tensors.append(tf.reshape(
                    vector[offset: offset+numel], p.shape))
                offset += numel

            assert offset == num_total_param
            return tensors

    def calc_norm(self, v):
        # default: frobenius norm
        if isinstance(v, list):
            norm = 0.
            for p in v:
                norm = norm + tf.norm(tensor=p)**2
            return norm**0.5
        else:
            return tf.norm(tensor=v)

    def _ops_in_minibatch(self):
        """
        Define operations that will be used in method minibatch

        Vectorization is already a deep copy operation.
        Before using newton method, loss needs to be summed over training samples
        to make results consistent.
        """

        def cal_loss():
            return tf.compat.v1.assign(self.f, self.f + self.loss)

        def cal_lossgrad():
            update_f = tf.compat.v1.assign(self.f, self.f + self.loss)

            grad = tf.gradients(ys=self.loss, xs=self.param)
            grad = self.vectorize(grad)
            update_grad = tf.compat.v1.assign(self.g, self.g + grad)

            return tf.group(*[update_f, update_grad])

        def cal_lossGv():
            v = self.inverse_vectorize(self.v, self.param)
            Gv = Gauss_Newton_vec(self.outputs, self.loss, self.param, v)
            Gv = self.vectorize(Gv)
            return tf.compat.v1.assign(self.Gv, self.Gv + Gv)

        # add regularization term to loss, gradient and Gv and further average over batches
        def add_reg_avg_loss():
            model_weight = self.vectorize(self.param)
            reg = (self.calc_norm(model_weight))**2
            reg = 1.0/(2*self.config.C) * reg
            return tf.compat.v1.assign(self.f, reg + self.f/self.config.num_data)

        def add_reg_avg_lossgrad():
            model_weight = self.vectorize(self.param)
            reg_grad = model_weight/self.config.C
            return tf.compat.v1.assign(self.g, reg_grad + self.g/self.config.num_data)
```

```
        def add_reg_avg_lossGv():
            return tf.compat.v1.assign(self.Gv, (self._lambda + 1/self.config.C)*self.v
                                       + self.Gv/self.config.GNsize)

        # zero out loss, grad and Gv
        def zero_loss():
            return tf.compat.v1.assign(self.f, tf.zeros_like(self.f))

        def zero_grad():
            return tf.compat.v1.assign(self.g, tf.zeros_like(self.g))

        def zero_Gv():
            return tf.compat.v1.assign(self.Gv, tf.zeros_like(self.Gv))

        return (cal_loss(), cal_lossgrad(), cal_lossGv(),
                add_reg_avg_loss(), add_reg_avg_lossgrad(), add_reg_avg_lossGv(),
                zero_loss(), zero_grad(), zero_Gv())

    def minibatch(self, data_batch, place_holder_x, place_holder_y, mode):
        """
        A function to evaluate either function value, global gradient or sub-sampled Gv
        """
        if mode not in ('funonly', 'fungrad', 'Gv'):
            raise ValueError('Unknown mode other than funonly & fungrad & Gv!')

        inputs, labels = data_batch
        num_data = labels.shape[0]
        num_segment = math.ceil(num_data/self.config.bsize)
        x, y = place_holder_x, place_holder_y

        # before estimation starts, need to zero out f, grad and Gv according to the mode

        if mode == 'funonly':
            assert num_data == self.config.num_data
            assert num_segment == self.num_grad_segment
            self.sess.run(self.zero_loss)
        elif mode == 'fungrad':
            assert num_data == self.config.num_data
            assert num_segment == self.num_grad_segment
            self.sess.run([self.zero_loss, self.zero_grad])
        else:
            assert num_data == self.config.GNsize
            assert num_segment == self.num_Gv_segment
            self.sess.run(self.zero_Gv)

        for i in range(num_segment):

            load_time = time.time()
            idx = np.arange(i * self.config.bsize,
                            min((i+1) * self.config.bsize, num_data))
            batch_input = inputs[idx]
            batch_labels = labels[idx]
            batch_input = np.ascontiguousarray(batch_input)
            batch_labels = np.ascontiguousarray(batch_labels)
            self.config.elapsed_time += time.time() - load_time

            if mode == 'funonly':

                self.sess.run(self.cal_loss, feed_dict={
                    x: batch_input,
                    y: batch_labels, })

            elif mode == 'fungrad':

                self.sess.run(self.cal_lossgrad, feed_dict={
                    x: batch_input,
                    y: batch_labels, })

            else:

                self.sess.run(self.cal_lossGv, feed_dict={
                    x: batch_input,
                    y: batch_labels})
```

```
        # average over batches
        if mode == 'funonly':
            self.sess.run(self.add_reg_avg_loss)
        elif mode == 'fungrad':
            self.sess.run([self.add_reg_avg_loss, self.add_reg_avg_grad])
        else:
            self.sess.run(self.add_reg_avg_Gv,
                          feed_dict={self._lambda: self.config._lambda})

    def _update_model(self):
        update_model_ops = []
        x = self.inverse_vectorize(self.s, self.param)
        for i, p in enumerate(self.param):
            op = tf.compat.v1.assign(p, p + (self.alpha-self.old_alpha) * x[i])
            update_model_ops.append(op)
        return tf.group(*update_model_ops)

    def _init_cg_vars(self):
        init_ops = []

        init_r = tf.compat.v1.assign(self.r, -self.g)
        init_v = tf.compat.v1.assign(self.v, -self.g)
        init_s = tf.compat.v1.assign(self.s, tf.zeros_like(self.g))
        gnorm = self.calc_norm(self.g)
        init_rTr = tf.compat.v1.assign(self.rTr, gnorm**2)
        init_cgtol = tf.compat.v1.assign(self.cgtol, self.config.xi*gnorm)

        init_ops = [init_r, init_v, init_s, init_rTr, init_cgtol]

        return tf.group(*init_ops)

    def _CG(self):
        """
        CG:
                define operations that will be used in method newton

        Same as the previous loss calculation,
        Gv has been summed over batches when samples were fed into Neural Network.
        """

        def CG_ops():

            vGv = tf.tensordot(self.v, self.Gv, axes=1)

            alpha = self.rTr / vGv
            with tf.control_dependencies([alpha]):
                update_s = tf.compat.v1.assign(
                    self.s, self.s + alpha * self.v, name='update_s_ops')
                update_r = tf.compat.v1.assign(
                    self.r, self.r - alpha * self.Gv, name='update_r_ops')

                with tf.control_dependencies([update_s, update_r]):
                    rnewTrnew = self.calc_norm(update_r)**2
                    update_beta = tf.compat.v1.assign(
                        self.beta, rnewTrnew / self.rTr)
                    with tf.control_dependencies([update_beta]):
                        update_rTr = tf.compat.v1.assign(
                            self.rTr, rnewTrnew, name='update_rTr_ops')

            return tf.group(*[update_s, update_beta, update_rTr])

        def update_v():
            return tf.compat.v1.assign(self.v, self.r + self.beta*self.v, name='update_v')

        return (CG_ops(), update_v())

    def newton(self, folder_dir, full_batch, val_batch, saver, network, num_classes, test_netw
ork=None):
        """
        Conduct newton steps for training
        args:
             full_batch & val_batch: provide training set and validation set. The function
```

```
                will save the best model evaluted on validation set for future prediction.
            network: a tuple contains (x, y, loss, outputs).
            test_network: a tuple similar to argument network. If you use layers which behave
            differently in test phase such as batchnorm, a separate test_network is needed.
        return:
                None
        """
        # check whether data is valid
        full_inputs, full_labels = full_batch
        assert full_inputs.shape[0] == full_labels.shape[0]

        if full_inputs.shape[0] != self.config.num_data:
            raise ValueError('The number of full batch inputs does not agree with the config \
               argument. This is important because global loss is averaged over those inputs')

        x, y, _, outputs = network

        # Create summary direcotry
        summary_dir = folder_dir + 'summary/'
        if not os.path.exists(summary_dir):
            os.makedirs(summary_dir)

        summary_dir = summary_dir + 'train'

        # Print to a format friendly for excel
        metrics_train_file = open(
            folder_dir + "running_log/metrics_train_file.txt", 'w')
        metrics_valid_file = open(
            folder_dir + "running_log/metrics_valid_file.txt", 'w')

        if num_classes == 2:
            print("epoch\ttn\tfp\tfn\ttp\tacc\tppv\tnpv\tsensitivity\tspecificity\tb_acc\tloss
\tepoch_time\tf\t|g|\talpha\tratio\tlambda\t#CG\tactred\tprered",
                  file=metrics_train_file)
            print("epoch\ttn\tfp\tfn\ttp\tacc\tppv\tnpv\tsensitivity\tspecificity\tb_acc\tloss
\tepoch_time\tf\t|g|\talpha\tratio\tlambda\t#CG\tactred\tprered",
                  file=metrics_valid_file)
        elif num_classes == 3:
            print("epoch\ttn_0\tfp_0\tfn_0\ttp_0\tacc_0\tppv_0\tnpv_0\tsensitivity_0\tspecific
ity_0\tb_acc_0\ttn_1\tfp_1\tfn_1\ttp_1\tacc_1\tppv_1\tnpv_1\tsensitivity_1\tspecificity_1\tb_a
cc_1\ttn_2\tfp_2\tfn_2\ttp_2\tacc_2\tppv_2\tnpv_2\tsensitivity_2\tspecificity_2\tb_acc_2\tacc\
tloss\tepoch_time\tf\t|g|\talpha\tratio\tlambda\t#CG\tactred\tprered",
                  file=metrics_train_file)
            print("epoch\ttn_0\tfp_0\tfn_0\ttp_0\tacc_0\tppv_0\tnpv_0\tsensitivity_0\tspecific
ity_0\tb_acc_0\ttn_1\tfp_1\tfn_1\ttp_1\tacc_1\tppv_1\tnpv_1\tsensitivity_1\tspecificity_1\tb_a
cc_1\ttn_2\tfp_2\tfn_2\ttp_2\tacc_2\tppv_2\tnpv_2\tsensitivity_2\tspecificity_2\tb_acc_2\tacc\
tloss\tepoch_time\tf\t|g|\talpha\tratio\tlambda\t#CG\tactred\tprered",
                  file=metrics_valid_file)
        else:
            print("epoch\tacc\tloss\tepoch_time\tf\t|g|\talpha\tratio\tlambda\t#CG\tactred\tpr
ered",
                  file=metrics_train_file)
            print("epoch\tacc\tloss\tepoch_time\tf\t|g|\talpha\tratio\tlambda\t#CG\tactred\tpr
ered",
                  file=metrics_valid_file)

        # Force to write immediately to file
        metrics_train_file.flush()
        metrics_valid_file.flush()

        tf.compat.v1.summary.scalar('loss', self.f)
        merged = tf.compat.v1.summary.merge_all()
        train_writer = tf.compat.v1.summary.FileWriter(
            summary_dir, self.sess.graph)

        print(self.config.args)
        if not self.config.screen_log_only:
            log_file = open(self.config.log_file, 'w')
            print(self.config.args, file=log_file)
            # Force to write immediately to file
            log_file.flush()

        self.minibatch(full_batch, x, y, mode='fungrad')
```

```
        f = self.sess.run(self.f)
        output_str = 'initial f: {:.3f}'.format(f)
        print(output_str)
        if not self.config.screen_log_only:
            print(output_str, file=log_file)

            # Force to write immediately to file
            log_file.flush()

        best_acc = 0.0

        total_running_time = 0.0
        self.config.elapsed_time = 0.0
        total_CG = 0

        for k in range(self.config.iter_max):

            # randomly select the batch for Gv estimation
            idx = np.random.choice(np.arange(0, full_labels.shape[0]),
                                   size=self.config.GNsize, replace=False)

            mini_inputs = full_inputs[idx]
            mini_labels = full_labels[idx]

            start = time.time()

            self.sess.run(self.init_cg_vars)
            cgtol = self.sess.run(self.cgtol)

            avg_cg_time = 0.0
            for CGiter in range(1, self.config.CGmax+1):

                cg_time = time.time()
                self.minibatch((mini_inputs, mini_labels), x, y, mode='Gv')
                avg_cg_time += time.time() - cg_time

                self.sess.run(self.CG)

                rnewTrnew = self.sess.run(self.rTr)

                if rnewTrnew**0.5 <= cgtol or CGiter == self.config.CGmax:
                    break

                self.sess.run(self.update_v)

            print('Avg time per Gv iteration: {:.5f} s\r\n'.format(
                avg_cg_time/CGiter))

            gs, sGs = self.sess.run([self.update_gs, self.update_sGs], feed_dict={
                self._lambda: self.config._lambda
            })

            # line_search
            f_old = f
            alpha = 1
            while True:

                old_alpha = 0 if alpha == 1 else alpha/0.5

                self.sess.run(self.update_model, feed_dict={
                    self.alpha: alpha, self.old_alpha: old_alpha
                })

                prered = alpha*gs + (alpha**2)*sGs

                self.minibatch(full_batch, x, y, mode='funonly')
                f = self.sess.run(self.f)

                actred = f - f_old

                if actred <= self.config.eta*alpha*gs:
                    break
```

```
                alpha *= 0.5

            # update lambda
            ratio = actred / prered
            if ratio < 0.25:
                self.config._lambda *= self.config.boost
            elif ratio >= 0.75:
                self.config._lambda *= self.config.drop

            self.minibatch(full_batch, x, y, mode='fungrad')
            f = self.sess.run(self.f)

            gnorm = self.sess.run(self.gnorm)

            summary = self.sess.run(merged)
            train_writer.add_summary(summary, k)

            # exclude data loading time for fair comparison
            end = time.time()

            end = end - self.config.elapsed_time
            total_running_time += end-start

            self.config.elapsed_time = 0.0

            total_CG += CGiter

            output_str = '{}-
iter f: {:.3f} |g|: {:.5f} alpha: {:.3e} ratio: {:.3f} lambda: {:.5f} #CG: {} actred: {:.5f} p
rered: {:.5f} time: {:.3f}'.\
                format(k, f, gnorm, alpha, actred/prered,
                       self.config._lambda, CGiter, actred, prered, end - start)

            # Training Statistics
            avg_train_loss, avg_train_acc, train_real = predict(
                self.sess, network, full_batch, self.config.bsize)
            train_target = np.argmax(full_batch[1], axis=1)

            # Print train metrics in a text file friendly for Excel
            if num_classes == 2:
                stats_train_output, stats_train_file_output = stats_output(
                    "Training", k, train_target, train_real)
                stats_train_file_output += "{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}\t{:.3e}\t{:.3f}\t{:
.5f}\t{}\t{:.5f}\t{:.5f}".format(
                    avg_train_loss, end - start, f, gnorm, alpha, actred/prered,
                    self.config._lambda, CGiter, actred, prered)
            elif num_classes == 3:
                stats_train_output, stats_train_file_output = stats_output3(
                    "Training", k, train_target, train_real)
                stats_train_file_output += "{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}\t{:.3e}\t{:
.3f}\t{:.5f}\t{}\t{:.5f}\t{:.5f}".format(avg_train_acc, avg_train_loss, end - start, f, gnorm,
 alpha, actred/prered, self.config._lambda, CGiter, actred, prered)
            else:
                stats_train_file_output = "{}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}\t{:.3e}\
t{:.3f}\t{:.5f}\t{}\t{:.5f}\t{:.5f}".format(k, avg_train_acc, avg_train_loss, end - start, f,
gnorm, alpha, actred/prered, self.config._lambda, CGiter, actred, prered)
                stats_train_output = stats_train_file_output

            print(stats_train_file_output,  file=metrics_train_file)

            # Force to write immediately to file
            metrics_train_file.flush()

            # Val;idation Statistics
            avg_val_loss, avg_val_acc, val_real = predict(
                self.sess, network, val_batch, self.config.bsize)
            val_target = np.argmax(val_batch[1], axis=1)

            # Print validation metrics in a text file friendly for Excel
            if num_classes == 2:
                stats_valid_output, stats_valid_file_output = stats_output(
                    "Validation", k, val_target, val_real)
```

```
                stats_valid_file_output += "{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}\t{:.3e}\t{:.3f}\t{:
.5f}\t{}\t{:.5f}\t{:.5f}".format(
                    avg_val_loss, end - start, f, gnorm, alpha, actred/prered,
                    self.config._lambda, CGiter, actred, prered)
            elif num_classes == 3:
                stats_valid_output, stats_valid_file_output = stats_output3(
                    "Validation", k, val_target, val_real)
                stats_valid_file_output += "{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}\t{:.3e}\t{:
.3f}\t{:.5f}\t{}\t{:.5f}\t{:.5f}".format(
                    avg_val_acc, avg_val_loss, end - start, f, gnorm, alpha, actred/prered,
                    self.config._lambda, CGiter, actred, prered)
            else:
                stats_valid_file_output = "{}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.5f}\t{:.3e}\
\t{:.3f}\t{:.5f}\t{}\t{:.5f}\t{:.5f}".format(
                    k, avg_val_acc, avg_val_loss, end - start, f, gnorm, alpha, actred/prered,
                    self.config._lambda, CGiter, actred, prered)
                stats_valid_output = stats_valid_file_output

            print(stats_valid_file_output, file=metrics_valid_file)

            # Force to write immediately to file
            metrics_valid_file.flush()

            # Print all statistics
            print(output_str)
            print(stats_train_output)
            print(stats_valid_output)
            if not self.config.screen_log_only:
                print(output_str, file=log_file)
                print(stats_train_output, file=log_file)
                print(stats_valid_output, file=log_file)

                # Force to write immediately to file
                log_file.flush()

            if val_batch is not None:
                # Evaluate the performance after every Newton Step
                if test_network == None:
                    val_loss, val_acc, _ = predict(
                        self.sess,
                        network=(x, y, self.loss, outputs),
                        test_batch=val_batch,
                        bsize=self.config.bsize,
                    )
                else:
                    # A separat test network part has not been done...
                    val_loss, val_acc, _ = predict(
                        self.sess,
                        network=test_network,
                        test_batch=val_batch,
                        bsize=self.config.bsize
                    )

                output_str = '\r\n {}-iter val_acc: {:.3f}% val_loss {:.3f}\r\n'.\
                    format(k, val_acc * 100, val_loss)

                print(output_str)
                if not self.config.screen_log_only:
                    print(output_str, file=log_file)

                    # Force to write immediately to file
                    log_file.flush()

                if val_acc > best_acc:
                    best_acc = val_acc
                    checkpoint_path = self.config.model_file
                    save_path = saver.save(self.sess, checkpoint_path)
                    print('Best model saved in {}\r\n'.format(save_path))

        if val_batch is None:
            checkpoint_path = self.config.model_file
            save_path = saver.save(self.sess, checkpoint_path)
            print('Model at the last iteration saved in {}\r\n'.format(save_path))
```

```
            output_str = 'total_#CG {} | total running time {:.3f}s'.format(
                total_CG, total_running_time)
        else:
            output_str = 'Final acc: {:.3f}% | best acc {:.3f}% | total_#CG {} | total running
 time {:.3f}s'.\
                format(val_acc * 100, best_acc * 100,
                       total_CG, total_running_time)

        print(output_str)
        if not self.config.screen_log_only:
            print(output_str, file=log_file)
            # Force to write immediately to file
            log_file.flush()
            log_file.close()
```

Code Snippet B.2. Implementation of the NewtonCG for the CNNs.

B.3 utilities.py

This file contains useful utilities that are needed from other files. Also, it contains the stats_output method that is responsible for what statistics will be saved in the output file, such as TP, TN, FP, FN, accuracy, sensitivity, etc.

```
import numpy as np
import math
import scipy.io as sio
import os
import math
import pdb
from sklearn.metrics import confusion_matrix, classification_report


class ConfigClass(object):
    def __init__(self, args, num_data, num_cls):
        super(ConfigClass, self).__init__()
        self.args = args
        self.iter_max = args.iter_max

        # Different notations of regularization term:
        # In SGD, weight decay:
        #   weight_decay <- lr/(C*num_of_training_samples)
        # In Newton method:
        #   C <- C * num_of_training_samples

        self.seed = args.seed

        if self.seed is None:
            print('You choose not to specify a random seed.' +
                  'A different result is produced after each run.')
        elif isinstance(self.seed, int) and self.seed >= 0:
            print('You specify random seed {}.'.format(self.seed))
        else:
            raise ValueError('Only accept None type or nonnegative integers for' +
                             ' random seed argument!')

        self.train_set = args.train_set
        self.val_set = args.val_set
        self.num_cls = num_cls
        self.dim = args.dim

        self.num_data = num_data
        self.GNsize = min(args.GNsize, self.num_data)
        self.C = args.C * self.num_data
        self.net = args.net

        self.xi = 0.1
```

```
        self.CGmax = args.CGmax
        self._lambda = args._lambda
        self.drop = args.drop
        self.boost = args.boost
        self.eta = args.eta
        self.lr = args.lr
        self.lr_decay = args.lr_decay

        self.bsize = args.bsize
        if args.momentum < 0:
            raise ValueError('Momentum needs to be larger than 0!')
        self.momentum = args.momentum

        self.loss = args.loss
        if self.loss not in ('MSELoss', 'CrossEntropy'):
            raise ValueError('Unrecognized loss type!')
        self.optim = args.optim
        if self.optim not in ('SGD', 'NewtonCG', 'Adam'):
            raise ValueError('Only support SGD, Adam & NewtonCG optimizer!')

        self.log_file = args.log_file
        self.model_file = args.model_file
        self.screen_log_only = args.screen_log_only

        # if self.screen_log_only:
        #     print('You choose not to store running log. Only store model to {}'.format(
        #         self.log_file))
        # else:
        #     print('Saving log to: {}'.format(self.log_file))
        #     dir_name, _ = os.path.split(self.log_file)
        #     if not os.path.isdir(dir_name):
        #         os.makedirs(dir_name, exist_ok=True)

        # dir_name, _ = os.path.split(self.model_file)
        # if not os.path.isdir(dir_name):
        #     os.makedirs(dir_name, exist_ok=True)

        self.elapsed_time = 0.0


def read_data(filename, dim, label_enum=None):
    """
    args:
            filename: the path where .mat files are stored
            label_enum (default None): the list that stores the original labels.
                    If label_enum is None, the function will generate a new list which stores
the
                    original labels in a sequence, and map original labels to [0, 1, ... numbe
r_of_classes-1].
                    If label_enum is a list, the function will use it to convert
                    original labels to [0, 1,..., number_of_classes-1].
    """

    mat_contents = sio.loadmat(filename)
    images, labels = mat_contents['Z'], mat_contents['y']

    labels = labels.reshape(-1)

    images = images.reshape(images.shape[0], -1)

    if (len(dim) == 3):
        _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS = dim
        zero_to_append = np.zeros((images.shape[0],
                                   _IMAGE_CHANNELS*_IMAGE_HEIGHT*_IMAGE_WIDTH-
np.prod(images.shape[1:])))
    elif(len(dim) == 4):
        _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_DEPTH, _IMAGE_CHANNELS = dim
        zero_to_append = np.zeros((images.shape[0],
                                   _IMAGE_CHANNELS*_IMAGE_HEIGHT*_IMAGE_WIDTH*_IMAGE_DEPTH-
np.prod(images.shape[1:])))

    images = np.append(images, zero_to_append, axis=1)
```

```
    # check data validity
    if label_enum is None:
        label_enum, labels = np.unique(labels, return_inverse=True)
        num_cls = labels.max() + 1

        if len(label_enum) != num_cls:
            raise ValueError('The number of classes is not equal to the number of\
                             labels in dataset. Please verify them.')
    else:
        num_cls = len(label_enum)
        forward_map = dict(zip(label_enum, np.arange(num_cls)))
        labels = np.expand_dims(labels, axis=1)
        labels = np.apply_along_axis(
            lambda x: forward_map[x[0]], axis=1, arr=labels)

    # convert groundtruth to one-hot encoding
    labels = np.eye(num_cls)[labels]
    labels = labels.astype('float32')

    return [images, labels], num_cls, label_enum


def normalize_and_reshape(images, dim, mean_tr=None):
    if (len(dim) == 3):
        _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS = dim
        images_shape = [images.shape[0],
                        _IMAGE_CHANNELS, _IMAGE_HEIGHT, _IMAGE_WIDTH]
    elif (len(dim) == 4):
        _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_DEPTH, _IMAGE_CHANNELS = dim
        images_shape = [images.shape[0],
                        _IMAGE_CHANNELS, _IMAGE_HEIGHT, _IMAGE_WIDTH,  _IMAGE_DEPTH]

    # images normalization and zero centering
    images = images.reshape(images_shape[0], -1)

    # images = images/255.0

    if mean_tr is None:
        print('No mean of data provided! Normalize images by their own mean.')
        # if no mean_tr is provided, we calculate it according to the current data
        mean_tr = images.mean(axis=0)
    else:
        print('Normalzie images according to the provided mean.')
        if np.prod(mean_tr.shape) != np.prod(dim):
            raise ValueError(
               'Dimension of provided mean does not agree with the data! Please verify them!v)

    images = images - mean_tr

    images = images.reshape(images_shape)

    if (len(dim) == 3):
        # Tensorflow accepts data shape: B x H x W x C
        images = np.transpose(images, (0, 2, 3, 1))
    elif (len(dim) == 4):
        # Tensorflow accepts data shape: B x H x W x D x C
        images = np.transpose(images, (0, 2, 3, 4, 1))

    return images, mean_tr


def predict(sess, network, test_batch, bsize):
    x, y, loss, outputs = network

    test_inputs, test_labels = test_batch
    batch_size = bsize

    num_data = test_labels.shape[0]
    num_batches = math.ceil(num_data/batch_size)

    results = np.zeros(shape=num_data, dtype=np.int)
```

```
    infer_loss = 0.0

    for i in range(num_batches):
        batch_idx = np.arange(i*batch_size, min((i+1)*batch_size, num_data))

        batch_input = test_inputs[batch_idx]
        batch_labels = test_labels[batch_idx]

        net_outputs, _loss = sess.run(
            [outputs, loss], feed_dict={x: batch_input, y: batch_labels}
        )

        # net_outputs are float values, no softmax is being applies
        # np.argmax returns the index of the max value
        results[batch_idx] = np.argmax(net_outputs, axis=1)
        # note that _loss was summed over batches
        infer_loss = infer_loss + _loss

    avg_acc = (np.argmax(test_labels, axis=1) == results).mean()
    avg_loss = infer_loss/num_data

    return avg_loss, avg_acc, results


def stats_output(stats_type, epoch, target, real):

    conf_matrix = confusion_matrix(target, real)

    # Correct
    tn = conf_matrix[0][0]
    fp = conf_matrix[0][1]
    fn = conf_matrix[1][0]
    tp = conf_matrix[1][1]

    # Incorrect!
    # tp = conf_matrix[0][0]
    # fp = conf_matrix[0][1]
    # fn = conf_matrix[1][0]
    # tn = conf_matrix[1][1]

    total = len(target)

    # print(classification_report(target, real))

    acc = "{:.3f}".format((tp + tn) / total)

    # Positive Predictive Value - PPV (Precision
    if (not ((tp + fp) == 0)):
        ppv = "{:.3f}".format(tp / (tp + fp))
    else:
        ppv = "nan"

    # Negative Predictive Value - NPV
    if(not ((tn + fn) == 0)):
        npv = "{:.3f}".format(tn / (tn + fn))
    else:
        npv = "nan"

    # Sensitivity (True Positive Rate)
    if(not ((tp + fn) == 0)):
        sensitivity = "{:.3f}".format(tp/(tp + fn))
    else:
        sensitivity = "nan"

    # Specificity (True Negative Rate)
    if(not ((tn + fp) == 0)):
        specificity = "{:.3f}".format(tn / (tn + fp))
    else:
        specificity = "nan"

    # Balanced Accuracy
    balanced_acc = "{:.3f}".format((tp / (tp + fn) + tn / (tn + fp)) / 2)
```

```
    output = '{}: epoch {} | tn {} | fp {} | fn {} | tp {} | acc {} | ppv {} | npv {} | sensit
ivity {} | specificity {} | b_acc {}'\
        .format(stats_type, epoch, tn, fp, fn, tp, acc, ppv, npv, sensitivity, specificity, ba
lanced_acc)

    # Output metrics for file
    output_file = "{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t".format(
        epoch, tn, fp, fn, tp, acc, ppv, npv, sensitivity, specificity, balanced_acc)
    return output, output_file


def stats_output3(stats_type, epoch, target, real):
    # For multi-class classifiacation, you may use one against all approach.
    # Suppose there are three classes: C1, C2, and C3

    # To find these four terms of C2 or C3 you can replace C1 with C2 or C3.
    # target_names = ['CN', 'MCI', 'AD']
    # labels = [0, 1, 2]
    # # Recall  == Sensitivity
    # # Precision == PPV
    # print(classification_report(target, real, labels=labels,
    #                             target_names=target_names, output_dict=True))
    conf_matrix = confusion_matrix(target, real)

    FP = conf_matrix.sum(axis=0) - np.diag(conf_matrix)
    FN = conf_matrix.sum(axis=1) - np.diag(conf_matrix)
    TP = np.diag(conf_matrix)
    TN = conf_matrix.sum() - (FP + FN + TP)

    # Sensitivity, hit rate, recall, or true positive rate
    TPR = TP/(TP+FN)
    # Specificity or true negative rate
    TNR = TN/(TN+FP)
    # Precision or positive predictive value
    PPV = TP/(TP+FP)
    # Negative predictive value
    NPV = TN/(TN+FN)
    # Fall out or false positive rate
    # FPR = FP/(FP+TN)
    # # False negative rate
    # FNR = FN/(TP+FN)
    # # False discovery rate
    # FDR = FP/(TP+FP)

    # Overall accuracy
    ACC = (TP + TN) / (TP + FP + FN + TN)

    # Balanced Accuracy
    BALANCED_ACC = (TP / (TP + FN) + TN / (TN + FP)) / 2

    acc_0 = "{:.3f}".format(ACC[0])
    acc_1 = "{:.3f}".format(ACC[1])
    acc_2 = "{:.3f}".format(ACC[2])

    # Positive Predictive Value - PPV (Precision
    if (not ((TP[0] + FP[0]) == 0)):
        ppv_0 = "{:.3f}".format(PPV[0])
    else:
        ppv_0 = "nan"

    if (not ((TP[1] + FP[1]) == 0)):
        ppv_1 = "{:.3f}".format(PPV[1])
    else:
        ppv_1 = "nan"

    if (not ((TP[2] + FP[2]) == 0)):
        ppv_2 = "{:.3f}".format(PPV[2])
    else:
        ppv_2 = "nan"

    # Negative Predictive Value - NPV
```

```
    if(not ((TN[0] + FN[0]) == 0)):
        npv_0 = "{:.3f}".format(NPV[0])
    else:
        npv_0 = "nan"

    if(not ((TN[1] + FN[1]) == 0)):
        npv_1 = "{:.3f}".format(NPV[1])
    else:
        npv_1 = "nan"

    if(not ((TN[2] + FN[2]) == 0)):
        npv_2 = "{:.3f}".format(NPV[2])
    else:
        npv_2 = "nan"

    # Sensitivity (True Positive Rate)
    if(not ((TP[0] + FN[0]) == 0)):
        sensitivity_0 = "{:.3f}".format(TPR[0])
    else:
        sensitivity_0 = "nan"

    if(not ((TP[1] + FN[1]) == 0)):
        sensitivity_1 = "{:.3f}".format(TPR[1])
    else:
        sensitivity_1 = "nan"

    if(not ((TP[2] + FN[2]) == 0)):
        sensitivity_2 = "{:.3f}".format(TPR[2])
    else:
        sensitivity_2 = "nan"

    # Specificity (True Negative Rate)
    if(not ((TN[0] + FP[0]) == 0)):
        specificity_0 = "{:.3f}".format(TNR[0])
    else:
        specificity_0 = "nan"

    if(not ((TN[1] + FP[1]) == 0)):
        specificity_1 = "{:.3f}".format(TNR[1])
    else:
        specificity_1 = "nan"

    if (not ((TN[2] + FP[2]) == 0)):
        specificity_2 = "{:.3f}".format(TNR[2])
    else:
        specificity_2 = "nan"

    b_acc_0 = "{:.3f}".format(BALANCED_ACC[0])
    b_acc_1 = "{:.3f}".format(BALANCED_ACC[1])
    b_acc_2 = "{:.3f}".format(BALANCED_ACC[2])

    output = '{}: epoch {} | tn_0 {} | fp_0 {} | fn_0 {} | tp_0 {} | acc_0 {} | ppv_0 {} | npv
_0 {} | sensitivity_0 {} | specificity_0 {} | b_acc_0 {} | tn_1 {} | fp_1 {} | fn_1 {} | tp_1
{} | acc_1 {} | ppv_1 {} | npv_1 {} | sensitivity_1 {} | specificity_1 {} | b_acc_1 {} | tn_2
{} | fp_2 {} | fn_2 {} | tp_2 {} | acc_2 {} | ppv_2 {} | npv_2 {} | sensitivity_2 {} | specifi
city_2 {} | b_acc_2 {}'\
        .format(stats_type, epoch, TN[0], FP[0], FN[0], TP[0], acc_0, ppv_0, npv_0, sensitivit
y_0, specificity_0, b_acc_0, TN[1], FP[1], FN[1], TP[1], acc_1, ppv_1, npv_1, sensitivity_1, s
pecificity_1, b_acc_1, TN[2], FP[2], FN[2], TP[2], acc_2, ppv_2, npv_2, sensitivity_2, specifi
city_2, b_acc_2)

    # Output metrics for file
    output_file = "{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\
t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t".format(
        epoch,  TN[0], FP[0], FN[0], TP[0], acc_0, ppv_0, npv_0, sensitivity_0, specificity_0,
 b_acc_0, TN[1], FP[1], FN[1], TP[1], acc_1, ppv_1, npv_1, sensitivity_1, specificity_1, b_acc
_1, TN[2], FP[2], FN[2], TP[2], acc_2, ppv_2, npv_2, sensitivity_2, specificity_2, b_acc_2)
    return output, output_file
```

Code Snippet B.3. Utility functions used in the CNN implementation.

B.4 predict.py

The predict.py file runs a single forward pass the network. Is being used during training but it can also be used for the test set as well.

```
import numpy as np
import pdb
import argparse
from net.net import CNN
from utilities import predict, read_data, normalize_and_reshape, stats_output, stats_output3
import tensorflow as tf
tf.compat.v1.disable_eager_execution()


def parse_args():
    parser = argparse.ArgumentParser(description='prediction')
    parser.add_argument('--test_set', dest='test_set',
                        help='provide the directory of .mat file for testing',
                        default='data/mnist-demo.t.mat', type=str)
    parser.add_argument('--model', dest='model_file',
                        help='provide file storing network parameters, i.e. ./dir/model.ckpt',
                        default='./saved_model/model.ckpt', type=str)
    parser.add_argument('--bsize', dest='bsize',
                        help='batch size',
                        default=1024, type=int)
    parser.add_argument('--loss', dest='loss',
                        help='which loss function to use: MSELoss or CrossEntropy',
                        default='MSELoss', type=str)
    parser.add_argument('--dim', dest='dim', nargs='+', help='input dimension of data,' +
                        'shape must be:  height width num_channels',
                                    default=[32, 32, 3], type=int)
    args = parser.parse_args()
    return args


if __name__ == '__main__':
    # Number of maximum cores to use in the server
    cores_to_use = 16
    args = parse_args()

    sess_config = tf.compat.v1.ConfigProto(
        allow_soft_placement=True, intra_op_parallelism_threads=cores_to_use, inter_op_paralle
lism_threads=cores_to_use)
    sess_config.gpu_options.allow_growth = True

    with tf.compat.v1.Session(config=sess_config) as sess:
        graph_address = args.model_file + '.meta'
        imported_graph = tf.compat.v1.train.import_meta_graph(graph_address)
        imported_graph.restore(sess, args.model_file)
        mean_param = [v for v in tf.compat.v1.global_variables()
                      if 'mean_tr:0' in v.name][0]
        label_enum_var = [
            v for v in tf.compat.v1.global_variables() if 'label_enum:0' in v.name][0]

        sess.run(tf.compat.v1.variables_initializer(
            [mean_param, label_enum_var]))
        mean_tr = sess.run(mean_param)
        label_enum = sess.run(label_enum_var)

        test_batch, num_cls, _ = read_data(
            args.test_set, dim=args.dim, label_enum=label_enum)
        test_batch[0], _ = normalize_and_reshape(
            test_batch[0], dim=args.dim, mean_tr=mean_tr)

        x = tf.compat.v1.get_default_graph().get_tensor_by_name(
            'main_params/input_of_net:0')
        y = tf.compat.v1.get_default_graph().get_tensor_by_name('main_params/labels:0')
        outputs = tf.compat.v1.get_default_graph().get_tensor_by_name('output_of_net:0')
```

```
        if args.loss == 'MSELoss':
            loss = tf.reduce_sum(input_tensor=tf.pow(outputs-y, 2))
        else:
            loss = tf.reduce_sum(input_tensor=tf.nn.softmax_cross_entropy_with_logits(
                logits=outputs, labels=tf.stop_gradient(y)))

        network = (x, y, loss, outputs)

        avg_loss, avg_acc, results = predict(
            sess, network, test_batch, args.bsize)

        # convert results back to the original labels
        inverse_map = dict(zip(np.arange(num_cls), label_enum))
        results = np.expand_dims(results, axis=1)
        results = np.apply_along_axis(
            lambda x: inverse_map[x[0]], axis=1, arr=results)

        target = np.argmax(test_batch[1], axis=1)
        real = results - 1  # Convert  from [1, 2, 2,] tp [0, 1, 1,]

        # Print train metrics in a text file friendly for Excel
        metrics_test_file = open(
            args.model_file[:-23] + "/metrics_test_file.txt", 'w')

        num_classes = max(np.argmax(test_batch[1], axis=1)) + 1
        if num_classes == 2:
            stats_test_output, stats_test_file_output = stats_output(
                "Testing", 1, target, real)
            print("epoch\tn\fp\fn\tp\acc\ppv\npv\sensitivity\specificity\b_acc\loss",
                  file=metrics_test_file)
            stats_test_file_output += "{:.3f}".format(avg_loss)

        elif num_classes == 3:
            stats_test_output, stats_test_file_output = stats_output3(
                "Testing", 1, target, real)
            print("epoch\ttn_0\tfp_0\tfn_0\ttp_0\tacc_0\tppv_0\tnpv_0\tsensitivity_0\tspecific
ity_0\tb_acc_0\ttn_1\tfp_1\tfn_1\ttp_1\tacc_1\tppv_1\tnpv_1\tsensitivity_1\tspecificity_1\tb_a
cc_1\ttn_2\tfp_2\tfn_2\ttp_2\tacc_2\tppv_2\tnpv_2\tsensitivity_2\tspecificity_2\tb_acc_2\tacc\
tloss",
                  file=metrics_test_file)

            stats_test_output += "\tacc: {:.3f}\tloss: {:.3f}".format(
                avg_acc, avg_loss)
            stats_test_file_output += "{:.3f}\t{:.3f}".format(
                avg_acc, avg_loss)
        else:
            print("epoch\tacc\tloss",
                  file=metrics_test_file)
            stats_test_file_output = "1\t{:.3f}\t{:.3f}".format(
                avg_acc, avg_loss)
            stats_test_output = stats_test_file_output

        print(stats_test_output)
        print(stats_test_file_output, file=metrics_test_file)

        # Force to write immediately to file
        metrics_test_file.flush()

        print('In test phase, average loss: {:.3f} | average accuracy: {:.3f}%'.
              format(avg_loss, avg_acc*100))
```

Code Snippet B.4. Forward propagation in the CNN implementations

B.5 net.py

The algorithm to run requires a net.py file that defines the network configuration. This file should be modified between experiments. Some examples of different networks used

with Dropout, L1 & L2 Regularization, Spatial Dropout, batch Normalization, deep, shallow, narrow, and wide networks can be found in the following subsections.

B.5.1 4-layer 2D & 3D CNNs, Single Dense Layer

```
from net.vgg import *
import numpy as np
from tensorflow.python.client import device_lib
import pdb
import math
import tensorflow.compat.v1 as tf
from keras.regularizers import l2
tf.disable_v2_behavior()


def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls
    if (len(dim) == 3):
        with tf.variable_scope('conv1', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv2D(
                filters=32,
                kernel_size=[3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(x_image)
            pool = tf.keras.layers.MaxPool2D(
                pool_size=[2, 2], strides=None, padding='valid')(conv)

        with tf.variable_scope('conv2', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv2D(
                filters=32,
                kernel_size=[3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(pool)
            pool = tf.keras.layers.MaxPool2D(
                pool_size=[2, 2], strides=None, padding='valid')(conv)

        with tf.variable_scope('conv3', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv2D(
                filters=64,
                kernel_size=[3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(pool)
            pool = tf.keras.layers.MaxPool2D(
                pool_size=[2, 2], strides=None, padding='valid')(conv)

        with tf.variable_scope('fully_connected', reuse=reuse) as scope:
            dim = np.prod(pool.shape[1:])
            flat = tf.reshape(pool, [-1, dim])
            outputs = tf.keras.layers.Dense(
                units=_NUM_CLASSES, name=scope.name)(flat)

        return outputs

    elif (len(dim) == 4):
        with tf.variable_scope('conv1', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv3D(
                filters=8,
                kernel_size=[5, 5, 5],
                padding='SAME',
                activation=tf.nn.relu
            )(x_image)
            pool = tf.keras.layers.MaxPooling3D(
                pool_size=[3, 3, 3], strides=2, padding='valid')(conv)

        with tf.variable_scope('conv2', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv3D(
```

```
                filters=16,
                kernel_size=[3, 3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(pool)
            pool = tf.keras.layers.MaxPooling3D(
                pool_size=[2, 2, 2], strides=2, padding='valid')(conv)

        with tf.variable_scope('conv3', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv3D(
                filters=32,
                kernel_size=[3, 3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(pool)
            pool = tf.keras.layers.MaxPooling3D(
                pool_size=[2, 2, 2], strides=2, padding='valid')(conv)

        with tf.variable_scope('fully_connected', reuse=reuse) as scope:
            dim = np.prod(pool.shape[1:])
            flat = tf.reshape(pool, [-1, dim])
            outputs = tf.keras.layers.Dense(
                units=_NUM_CLASSES, name=scope.name)(flat)

        return outputs

. . .

def CNN(net, num_cls, dim):

    _NUM_CLASSES = num_cls
    if (len(dim) == 3):
        _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS = dim
    elif (len(dim) == 4):
        _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_DEPTH, _IMAGE_CHANNELS = dim

    with tf.name_scope('main_params'):
        if (len(dim) == 3):
            x = tf.placeholder(tf.float32, shape=[
                None, _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS], name='input_of_net')
        elif (len(dim) == 4):
            x = tf.placeholder(tf.float32, shape=[
                None, _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_DEPTH, _IMAGE_CHANNELS], name='input
_of_net')
        y = tf.placeholder(tf.float32, shape=[
                           None, _NUM_CLASSES], name='labels')
        print(y)

    # call CNN structure according to string net
    outputs = globals()[net](x, _NUM_CLASSES, dim)
    outputs = tf.identity(outputs, name='output_of_net')

    return (x, y, outputs)
```

Code Snippet B.5. 4-layer networks for the 2D & 3D CNN implementation with a single dense layer.

B.5.2 4-layer / 7-layer 2D CNN & 4-layer 3D CNN, Dropout, Single Dense Layer

```
from net.vgg import *
import numpy as np
from tensorflow.python.client import device_lib
import pdb
import math
import tensorflow.compat.v1 as tf
from keras.regularizers import l2
tf.disable_v2_behavior()


def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls
```

```
    if (len(dim) == 3):
        with tf.variable_scope('conv1', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv2D(
                filters=32,
                kernel_size=[3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(x_image)
            pool = tf.keras.layers.MaxPool2D(
                pool_size=[2, 2], strides=None, padding='valid')(conv)
            drop = tf.keras.layers.Dropout(0.3)(pool)

        with tf.variable_scope('conv2', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv2D(
                filters=32,
                kernel_size=[3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(drop)
            pool = tf.keras.layers.MaxPool2D(
                pool_size=[2, 2], strides=None, padding='valid')(conv)
            drop = tf.keras.layers.Dropout(0.5)(pool)

        with tf.variable_scope('conv3', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv2D(
                filters=64,
                kernel_size=[3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(drop)
            pool = tf.keras.layers.MaxPool2D(
                pool_size=[2, 2], strides=None, padding='valid')(conv)
            drop = tf.keras.layers.Dropout(0.5)(pool)

        with tf.variable_scope('fully_connected', reuse=reuse) as scope:
            dim = np.prod(drop.shape[1:])
            flat = tf.reshape(drop, [-1, dim])
            outputs = tf.keras.layers.Dense(
                units=_NUM_CLASSES, name=scope.name)(flat)

        return outputs

    elif (len(dim) == 4):
        with tf.variable_scope('conv1', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv3D(
                filters=32,
                kernel_size=[3, 3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(x_image)
            pool = tf.keras.layers.MaxPooling3D(
                pool_size=[2, 2, 2], strides=2, padding='valid')(conv)
            drop = tf.keras.layers.Dropout(0.3)(pool)

        with tf.variable_scope('conv2', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv3D(
                filters=32,
                kernel_size=[3, 3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(drop)
            pool = tf.keras.layers.MaxPooling3D(
                pool_size=[2, 2, 2], strides=2, padding='valid')(conv)
            drop = tf.keras.layers.Dropout(0.5)(pool)

        with tf.variable_scope('conv3', reuse=reuse) as scope:
            conv = tf.keras.layers.Conv3D(
                filters=64,
                kernel_size=[3, 3, 3],
                padding='SAME',
                activation=tf.nn.relu
            )(drop)
            pool = tf.keras.layers.MaxPooling3D(
```

```
                pool_size=[2, 2, 2], strides=2, padding='valid')(conv)
            drop = tf.keras.layers.Dropout(0.5)(pool)

        with tf.variable_scope('fully_connected', reuse=reuse) as scope:
            dim = np.prod(drop.shape[1:])
            flat = tf.reshape(drop, [-1, dim])
            outputs = tf.keras.layers.Dense(
                units=_NUM_CLASSES, name=scope.name)(flat)

        return outputs


def CNN_7layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=16,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        drop = tf.keras.layers.Dropout(0.3)(conv)
        conv = tf.keras.layers.Conv2D(
            filters=16,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.3)(pool)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=16,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        drop = tf.keras.layers.Dropout(0.3)(conv)
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        drop = tf.keras.layers.Dropout(0.5)(conv)
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)
        # pool = tf.layers.dropout(pool, rate=0.25, name=scope.name)
```

```
    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

    . . .



def CNN(net, num_cls, dim):

    _NUM_CLASSES = num_cls
    if (len(dim) == 3):
        _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS = dim
    elif (len(dim) == 4):
        _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_DEPTH, _IMAGE_CHANNELS = dim

    with tf.name_scope('main_params'):
        if (len(dim) == 3):
            x = tf.placeholder(tf.float32, shape=[
                None, _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS], name='input_of_net')
        elif (len(dim) == 4):
            x = tf.placeholder(tf.float32, shape=[
                None, _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_DEPTH, _IMAGE_CHANNELS], name='input
_of_net')
        y = tf.placeholder(tf.float32, shape=[
                           None, _NUM_CLASSES], name='labels')
        print(y)

    # call CNN structure according to string net
    outputs = globals()[net](x, _NUM_CLASSES, dim)
    outputs = tf.identity(outputs, name='output_of_net')

    return (x, y, outputs)
```

Code Snippet B.6. Parts of 4-layer / 7-layer 2D CNN & 4-layer 3D CNN, with Dropout and a single dense layer.

B.5.3 5-layer 2D CNN, Two Dense Layers

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(pool)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
```

```
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(pool)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)

    with tf.variable_scope('fully_connected_layer_1', reuse=reuse) as scope:
        dim = np.prod(pool.shape[1:])
        flat1 = tf.reshape(pool, [-1, dim])
        flat2 = tf.keras.layers.Dense(
            units=256, name=scope.name)(flat1)

    with tf.variable_scope('fully_connected_layer_2', reuse=reuse) as scope:
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat2)

    return outputs

. . .
```

Code Snippet B.7. Parts of 4-layer 2D CNN with Dropout and two dense layers.

B.5.4 4-layer 2D CNN, Dropout, L1 & L2 Regularization

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
            kernel_regularizer=tf.keras.regularizers.l1_l2(
                l1=1e-5, l2=1e-4),
            bias_regularizer=tf.keras.regularizers.l2(1e-4),
            activity_regularizer=tf.keras.regularizers.l2(1e-5)
        )(x_image)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.3)(pool)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
            kernel_regularizer=tf.keras.regularizers.l1_l2(
                l1=1e-5, l2=1e-4),
            bias_regularizer=tf.keras.regularizers.l2(1e-4),
            activity_regularizer=tf.keras.regularizers.l2(1e-5)
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
            kernel_regularizer=tf.keras.regularizers.l1_l2(
```

```
                l1=1e-5, l2=1e-4),
            bias_regularizer=tf.keras.regularizers.l2(1e-4),
            activity_regularizer=tf.keras.regularizers.l2(1e-5)
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name,
            kernel_regularizer=tf.keras.regularizers.l1_l2(
                l1=1e-5, l2=1e-4),
            bias_regularizer=tf.keras.regularizers.l2(1e-4),
            activity_regularizer=tf.keras.regularizers.l2(1e-5))(flat)

    return outputs

. . .
```

Code Snippet B.8. Parts of 4-layer 2D CNN with Dropout and L1 & L2 Regularization.

B.5.5 4-layer 2D CNN, Dropout, Sigmoid

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.sigmoid
        )(x_image)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.3)(pool)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.sigmoid
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.sigmoid
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)
```

```
    return outputs

. . .
```

Code Snippet B.9. Parts of 4-layer 2D CNN with Dropout and the Sigmoid activation function.

B.5.6 4-layer 2D CNN, Spatial Dropout

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(pool)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(pool)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(pool.shape[1:])
        flat = tf.reshape(pool, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

. . .
```

Code Snippet B.10. Parts of 4-layer 2D CNN with Spatial Dropout.

B.5.7 4-layer 2D CNN, Spatial Dropout, Dropout

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .
```

```
    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
        )(x_image)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)
        drop = tf.keras.layers.Dropout(0.3)(pool)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
        )(drop)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
        )(drop)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

. . .
```

Code Snippet B.11. Parts of 4-layer 2D CNN with Spatial Dropout and Dropout.

B.5.8 4-layer 2D CNN, Spatial Dropout, SoftMax

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
        )(x_image)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
```

```
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
        )(pool)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu,
        )(pool)
        drop = tf.keras.layers.SpatialDropout2D(0.5)(conv)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(drop)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(pool.shape[1:])
        flat = tf.reshape(pool, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name, activation='softmax')(flat)

    return outputs

. . .
```

Code Snippet B.12. Parts of 4-layer 2D CNN with Spatial Dropout and SoftMax in the output layer.

B.5.9 4-layer 2D CNN, Dropout, Batch Normalization

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.3)(pool)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
```

```
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)
        bnorm = tf.keras.layers.BatchNormalization()(drop)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(bnorm.shape[1:])
        flat = tf.reshape(bnorm, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

. . .
```

Code Snippet B.13. Parts of 4-layer 2D CNN with Dropout and Batch Normalization

B.5.10 3-layer 2D CNN, Shallow-Wide

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

     . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.3)(pool)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=128,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

. . .
```

Code Snippet B.14. 3-layer 2D CNN implementation. (Shallow-Wide)

B.5.11 3-layer 2D CNN, Shallow-Narrow

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

     . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
```

```
        conv = tf.keras.layers.Conv2D(
            filters=16,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.3)(pool)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPool2D(
            pool_size=[2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

. . .
```

Code Snippet B.15. 3-layer 2D CNN implementation. (Shallow-Narrow)

B.5.12 4-layer 3D CNN, No Max-Pooling

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

     . . .

     with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=32,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        drop = tf.keras.layers.Dropout(0.3)(conv)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=32,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        drop = tf.keras.layers.Dropout(0.5)(conv)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=64,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        drop = tf.keras.layers.Dropout(0.5)(conv)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
```

```
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

. . .
```

Code Snippet B.16. 4-layer 3D CNN implementation without Max-Pooling.

B.5.13 4-layer 3D CNN, Single Layer Max-Pooling

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .

    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=32,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        drop = tf.keras.layers.Dropout(0.3)(conv)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=32,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPooling3D(
            pool_size=[2, 2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=64,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        drop = tf.keras.layers.Dropout(0.5)(conv)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

. . .
```

Code Snippet B.17. 4-layer 3D CNN implementation with a single layer of Max-Pooling.

B.5.14 5-layer 3D CNN, Dropout

```
. . .

def CNN_4layers(x_image, num_cls, dim, reuse=False):
    _NUM_CLASSES = num_cls

    . . .
```

```
    with tf.variable_scope('conv1', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=32,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(x_image)
        pool = tf.keras.layers.MaxPooling3D(
            pool_size=[2, 2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.3)(pool)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=32,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPooling3D(
            pool_size=[2, 2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('conv3', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=64,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPooling3D(
            pool_size=[2, 2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('conv4', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv3D(
            filters=64,
            kernel_size=[3, 3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(drop)
        pool = tf.keras.layers.MaxPooling3D(
            pool_size=[2, 2, 2], strides=None, padding='valid')(conv)
        drop = tf.keras.layers.Dropout(0.5)(pool)

    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
        dim = np.prod(drop.shape[1:])
        flat = tf.reshape(drop, [-1, dim])
        outputs = tf.keras.layers.Dense(
            units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

. . .
```

Code Snippet B.18. 5-layer 3D CNN implementation with Dropout

B.6 vgg.py

The vgg.py contains the VGG networks that are very deep, with 11, 13, 16, or 19 layers. You can specify which network architecture to use in the arguments passed during execution.

```
"""
Codes are modified from PyTorch and Tensorflow Versions of VGG:
https://github.com/pytorch/vision/blob/master/torchvision/models/vgg.py, and
https://github.com/keras-team/keras-applications/blob/master/keras_applications/vgg16.py
```

```
"""

import numpy as np
from tensorflow.keras.applications.vgg19 import VGG19 as vgg19
from tensorflow.keras.applications.vgg16 import VGG16 as vgg16
import pdb
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

__all__ = ['VGG11', 'VGG13', 'VGG16', 'VGG19']


def VGG(feature, num_cls, dim):

    with tf.variable_scope('fully_connected') as scope:
        dim = np.prod(feature.shape[1:])
        x = tf.reshape(feature, [-1, dim])

        x = tf.keras.layers.Dense(
            units=4096, activation='relu', name=scope.name)(x)
        x = tf.keras.layers.Dense(
            units=4096, activation='relu', name=scope.name)(x)
        x = tf.keras.layers.Dense(units=num_cls, name=scope.name)(x)

    return x


def make_layers(x, cfg, dim):
    if (len(dim) == 3):
        for v in cfg:
            if v == 'M':
                x = tf.keras.layers.MaxPool2D(
                    pool_size=[2, 2], strides=2, padding='valid')(x)
            elif v == 'D':
                x = tf.keras.layers.Dropout(0.3)(x)
            else:
                x = tf.keras.layers.Conv2D(
                    filters=v,
                    kernel_size=[3, 3],
                    padding='SAME',
                    activation=tf.nn.relu
                )(x)
    elif (len(dim) == 4):
        for v in cfg:
            if v == 'M':
                x = tf.keras.layers.MaxPool2D(
                    pool_size=[2, 2, 2], strides=2, padding='valid')(x)
            elif v == 'D':
                x = tf.keras.layers.Dropout(0.3)(x)
            else:
                x = tf.keras.layers.Conv2D(
                    filters=v,
                    kernel_size=[3, 3, 3],
                    padding='SAME',
                    activation=tf.nn.relu
                )(x)

    return x


cfg = {
    # 'A': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'A': [8, 'M', 16, 'M', 16, 16, 'M', 32, 32, 'M', 32, 32, 'M'],
    'B': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'D': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, '
M'],
    # 'E': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M',
    #     512, 512, 512, 512, 'M'],
    'E': [16, 16, 'M', 'D', 32, 32, 'M', 'D', 32, 32, 32, 32, 'M', 'D', 64, 64, 64, 64, 'M', '
D',
          64, 64, 64, 64, 'M', 'D'],
}
```

```
def VGG11(x_images, num_cls, dim):
    feature = make_layers(x_images, cfg['A'], dim)
    return VGG(feature, num_cls, dim)


def VGG13(x_images,  num_cls, dim):
    feature = make_layers(x_images, cfg['B'], dim)
    return VGG(feature, num_cls, dim)


def VGG16(x_images,  num_cls, dim):
    feature = make_layers(x_images, cfg['D'], dim)
    return VGG(feature, num_cls, dim)


def VGG19(x_images,  num_cls, dim):
    feature = make_layers(x_images, cfg['E'], dim)
    return VGG(feature, num_cls, dim)
```

Code Snippet B.19. VGG networks implementation.

Appendix C

Datasets Creation

C.1 create_B_2D_S.py

This create_B_2D_S.py reads the first excel with the patients' IDs, target classes, and genders, matched the IDS with the content of the folder with the MRI scans to create the B_2D_S dataset. Only a single slice of a single MRI scan per patient is being added to the dataset. The dataset has an equal number of AD, and NC patients.

```
import os
import scipy.io
import skimage.measure
import numpy as np
import pandas as pd
import nibabel as nib
import matplotlib.pyplot as plt
from skimage.transform import resize
import random

max_shape = (174, 174)
target_classes = ["AD", "CN"]
nc_counter = 0
ad_counter = 0
random.seed(5)


def data_info(path, data_Y_AD, data_Y_NC, data_Z, mri_dim):
    # unique_Y, counts_Y = np.unique(data_Y, return_counts=True)
    # dict_data_Y = dict(zip(unique_Y, counts_Y))
    # perc_Y = np.round(100.*(counts_Y/sum(counts_Y)), 1)
    # dict_data_perc_Y = dict(zip(unique_Y, perc_Y))

    unique_Z, counts_Z = np.unique(data_Z, return_counts=True)
    dict_data_Z = dict(zip(unique_Z, counts_Z))
    perc_Z = np.round(100.*(counts_Z/sum(counts_Z)), 1)
    dict_data_perc_Z = dict(zip(unique_Z, perc_Z))
    total_samples = len(data_Y_AD) + len(data_Y_NC)

    print("\nBalanced: ")
    print("AD Samples: ", str(len(data_Y_AD)))
    print("CN Samples: ", str(len(data_Y_NC)))
    print("Count per Gender: ", dict_data_Z)
    print("AD Percentage: {:.2f}%".format(100*len(data_Y_AD)/total_samples))
    print("CN Percentage: {:.2f}%".format(100*len(data_Y_NC)/total_samples))
    print("Gender Percentage.: ", dict_data_perc_Z)
    print("Total Set Size: ", str(total_samples))
    print('Image Dimensions: ', mri_dim)
    with open(path + 'data-info.txt', 'a') as info_file:
        print("AD Samples: ", str(len(data_Y_AD)), file=info_file)
        print("CN Samples: ", str(len(data_Y_NC)), file=info_file)
        print("Count per Gender: ", dict_data_Z, file=info_file)
        print("AD Percentage: {:.2f}%".format(
            100*len(data_Y_AD)/total_samples),  file=info_file)
        print("CN Percentage: {:.2f}%".format(
            100*len(data_Y_NC)/total_samples), file=info_file)
        print("Gender Percentage: ", dict_data_perc_Z, file=info_file)
        print("Total Set Size: ", str(total_samples), file=info_file)
        print('Image Dimensions: ', mri_dim, file=info_file)
```

```
def add_padding(i):
    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1]] = i
    return i_w_pad


def plot_slices(image):
    index_midle = round(len(image[0, :, 0, 0]) / 2)

    img1 = add_padding(image[:, index_midle - 40, :, 0])
    img2 = add_padding(image[:, index_midle - 30, :, 0])
    img3 = add_padding(image[:, index_midle - 20, :, 0])
    img4 = add_padding(image[:, index_midle - 10, :, 0])
    img5 = add_padding(image[:, index_midle, :, 0])
    img6 = add_padding(image[:, index_midle + 10, :, 0])
    img7 = add_padding(image[:, index_midle + 20, :, 0])
    img8 = add_padding(image[:, index_midle + 30, :, 0])
    img9 = add_padding(image[:, index_midle + 40, :, 0])

    fig = plt.figure(figsize=[8, 8])

    ax1 = fig.add_subplot(331)
    ax2 = fig.add_subplot(332)
    ax3 = fig.add_subplot(333)
    ax4 = fig.add_subplot(334)
    ax5 = fig.add_subplot(335)
    ax6 = fig.add_subplot(336)
    ax7 = fig.add_subplot(337)
    ax8 = fig.add_subplot(338)
    ax9 = fig.add_subplot(339)

    ax1.title.set_text(str(index_midle - 40) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax2.title.set_text(str(index_midle - 30) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax3.title.set_text(str(index_midle - 20) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax4.title.set_text(str(index_midle - 10) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax5.title.set_text(str(index_midle) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax6.title.set_text(str(index_midle + 10) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax7.title.set_text(str(index_midle + 20) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax8.title.set_text(str(index_midle + 30) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax9.title.set_text(str(index_midle + 40) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')

    ax1.imshow(img1, cmap="gray")
    ax2.imshow(img2, cmap="gray")
    ax3.imshow(img3, cmap="gray")
    ax4.imshow(img4, cmap="gray")
    ax5.imshow(img5, cmap="gray")
    ax6.imshow(img6, cmap="gray")
    ax7.imshow(img7, cmap="gray")
    ax8.imshow(img8, cmap="gray")
    ax9.imshow(img9, cmap="gray")

    ax1.axis("off")
    ax2.axis("off")
    ax3.axis("off")
    ax4.axis("off")
    ax5.axis("off")
    ax6.axis("off")
    ax7.axis("off")
    ax8.axis("off")
    ax9.axis("off")
```

```
    plt.show()


def read_patient(c, patient_folder, patient_id, patient_class, patient_sex):
    # data_list_X = []
    # data_list_Y = []
    data_list_Z = []
    data_list_X_AD = []
    data_list_Y_AD = []
    group_AD = []
    data_list_X_NC = []
    data_list_Y_NC = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    global ad_counter
    b = 1
    for file in os.listdir(patient_folder):

        if ((patient_class == "CN") & (nc_counter == 199)):
            break

        if (patient_class == "CN"):
            nc_counter += 1
        elif (patient_class == "AD"):
            ad_counter += 1

        print("Reading Patient: ", c, "\tPatient ID: ", patient_id, "\tBrain Scan: ", b,
              "\tNC counter: ", nc_counter, "\tAD counter: ", ad_counter)

        # try:
        image = nib.load(os.path.join(
            patient_folder, file)).get_fdata()

        # Display slices of brain
        # plot_slices(image)

        # Remove the last extra dimension
        index_midle = round(len(image[0, :, 0, 0]) / 2)
        image = image[:, index_midle, :, 0]

        # x.append(i_shape[0])
        # y.append(i_shape[1])
        # z.append(i_shape[2])

        # # Add zeros pading to make all the images the same size
        image = add_padding(image)

        # Reduce size to 1/4 of each dimension
        # image = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)

        # Display in Plot Single Slice
        # plt.figure(figsize=[6, 6])
        # plt.subplot(111)
        # plt.imshow(image, cmap="gray")
        # plt.show()

        # # Display in Plot Slices
        # plt.figure(figsize=[12, 6])
        # plt.subplot(121)
        # plt.imshow(image, cmap="gray")

        # image_shrinked = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)
        # plt.subplot(122)
        # plt.imshow(image_shrinked, cmap="gray")
        # plt.show()
```

```
        # # Convert to float
        image = image.astype('float32')

        # # Normalize data
        # # Normalize images between 0 and 255
        image = (image / np.max(image)) * 255

        # data_list_X.append(image.flatten())
        # data_list_Y.append(
        #     [target_classes.index(patient_class) + 1])  # Add the index
        data_list_Z.append([patient_sex])

        if (patient_class == "CN"):
            data_list_X_NC.append(image.flatten())
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            group_NC.append(patient_id)

        if (patient_class == "AD"):
            data_list_X_AD.append(image.flatten())
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            group_AD.append(patient_id)

        i_shape = image.shape

        mri_dim = [i_shape[0], i_shape[1]]
        # # To balance the data sets of the classes
        # if((patient_class == "MCI") & (c >= 1)):
        #     if(random.randint(1, 10) > 2):
        #         break
        #     else:
        #         continue

        b += 1
        break
        # except:
        #     print("Corrupted: ", os.path.join(patient_folder, file))
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, group_AD, group_NC
    # return data_list_X, data_list_Y, data_list_Z,  x, y, z, data_list_X_AD, data_list_Y_AD,
data_list_X_NC, data_list_Y_NC


def load_excel(excel_path):
    print(excel_path)
    df = pd.read_csv(excel_path, error_bad_lines=False)
    df.columns = [column.replace(" ", "_") for column in df.columns]
    print("Total Samples: ", len(df))
    # df.query(
    #     '(Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN") and (
Sex == "M" or Sex == "F")', inplace=True)
    # df.query(
    #     'Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN"', inpla
ce=True)
    df.query(
        '(Research_Group == "AD" or Research_Group == "CN") and (Sex == "M" or Sex == "F")', i
nplace=True)
    df.query(
        'Research_Group == "AD" or Research_Group == "CN"', inplace=True)
    df = df[['Subject_ID', 'Research_Group', 'Sex']]
    print("Total Samples (AD, CN): ", len(df))
    unique_patients = df.drop_duplicates(
        ['Subject_ID', 'Research_Group'], keep='last')
    df = pd.DataFrame(unique_patients)
    excel_data = list(df.itertuples(index=False, name=None))

    print("Unique Patients: ", len(excel_data))
    return excel_data
```

```
def read_data(excel_path, input_path):
    excel_data = load_excel(excel_path)
    print("Reading Excel Finidhed")
    # patients_X = []
    # patients_Y = []
    patients_Z = []
    patients_X_AD = []
    patients_Y_AD = []
    patients_X_NC = []
    patients_Y_NC = []
    group_AD = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    nc_counter = 0

    global ad_counter
    ad_counter = 0

    c = 1
    for patient in excel_data:
        # try:
        patient_id = patient[0]
        patient_class = patient[1]
        patient_sex = patient[2]

        if ((patient_class == "CN") & (nc_counter == 199)):
            print("Skip patient: ", patient_id)
            continue

        patient_folder = input_path + patient_id + "/"
        # p_X, p_Y, p_Z, p_x, p_y, p_z, p_X_AD, p_Y_AD, p_X_NC, p_Y_NC = read_patient(patient_
folder,
        #                                                                                 patient_
id, patient_class, patient_sex)
        p_X_AD, p_Y_AD, p_X_NC, p_Y_NC, p_Z, mri_dim, p_group_AD, p_group_NC = read_patient(c,
                                                                                            pa
tient_folder,   patient_id, patient_class, patient_sex)

        # patients_X = patients_X + p_X
        # patients_Y = patients_Y + p_Y
        patients_Z += p_Z
        patients_X_AD += p_X_AD
        patients_Y_AD += p_Y_AD
        patients_X_NC += p_X_NC
        patients_Y_NC += p_Y_NC
        group_AD += p_group_AD
        group_NC += p_group_NC

        # x = x + p_x
        # y = y + p_y
        # z = z + p_z
        # except:
        #     print("Corrupted Patient: ", patient)
        c += 1
        # if (c == 10):
        #     break

    # Max Dimensions: 35 34 50
    # print("Original Max Dimensions: ", max(x), max(y), max(z))

    # data_list_X = np.array(patients_X)
    # data_list_Y = np.array(patients_Y)
    data_list_Z = np.array(patients_Z)
    data_list_X_AD = np.array(patients_X_AD)
    data_list_Y_AD = np.array(patients_Y_AD)
    data_list_X_NC = np.array(patients_X_NC)
    data_list_Y_NC = np.array(patients_Y_NC)
    data_list_group_AD = np.array(group_AD)
    data_list_group_NC = np.array(group_NC)
```

```
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, data_list_group_AD, data_list_group_NC
    # return data_list_X, data_list_Y, data_list_Z, data_list_X_AD, data_list_Y_AD, data_list_
X_NC, data_list_Y_NC


def main():
    print("Start Reading ...")
    excel_path = "../../raw_data/brains/ADNI_brain.csv"
    input_path = "../../raw_data/brains/ADNI_brain/"
    # matrix_X, matrix_Y, matrix_Z, matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC = read_
data(
    #     excel_path, input_path)
    matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC, matrix_Z, mri_dim, group_AD, group_NC
= read_data(
        excel_path, input_path)

    print("... Reading Finished")

    # Normalize data
    # Normalize images between 0 and 255
    # matrix_X = (matrix_X / matrix_X.max())*255
    matrix_X_AD = (matrix_X_AD / matrix_X_AD.max())*255
    matrix_X_NC = (matrix_X_NC / matrix_X_NC.max())*255

    # Convert to float
    # matrix_X = matrix_X.astype('float32')
    matrix_X_AD = matrix_X_AD.astype('float32')
    matrix_X_NC = matrix_X_NC.astype('float32')

    output_path = "../../data_no_fold/brains-2D-AD-NC-single/"
    os.makedirs(output_path)

    # Save into .mat file
    # mdic = {"Z": matrix_X, "y": matrix_Y}
    # scipy.io.savemat(output_path + 'brains.mat', mdic)
    mdic = {"Z": matrix_X_AD, "y": matrix_Y_AD, "group": group_AD}
    scipy.io.savemat(output_path + 'brains_AD.mat', mdic)
    mdic = {"Z": matrix_X_NC, "y": matrix_Y_NC, "group": group_NC}
    scipy.io.savemat(output_path + 'brains_NC.mat', mdic)

    data_info(output_path, matrix_Y_AD, matrix_Y_NC, matrix_Z, mri_dim)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet C.1. Creation of the B_2D_S dataset and convert it to two .mat files, one with the AD and the other with the NC patients, without splitting it into a training, validation, or test set.

C.2 create_B_2D_5S.py

This create_B_2D_5S.py reads the first excel with the patients' IDs, target classes, and genders, matched the IDS with the content of the folder with the MRI scans to create the B_2D_5S dataset. Five slices of a single MRI scan per patient are being added to the dataset. The dataset has an equal number of AD, and NC patients.

```
import os
import scipy.io
import skimage.measure
import numpy as np
import pandas as pd
```

```
import nibabel as nib
import matplotlib.pyplot as plt
from skimage.transform import resize
import random

max_shape = (174, 174)
target_classes = ["AD", "CN"]
nc_counter = 0
ad_counter = 0
random.seed(5)


def data_info(path, data_Y_AD, data_Y_NC, data_Z, mri_dim):
    # unique_Y, counts_Y = np.unique(data_Y, return_counts=True)
    # dict_data_Y = dict(zip(unique_Y, counts_Y))
    # perc_Y = np.round(100.*(counts_Y/sum(counts_Y)), 1)
    # dict_data_perc_Y = dict(zip(unique_Y, perc_Y))

    unique_Z, counts_Z = np.unique(data_Z, return_counts=True)
    dict_data_Z = dict(zip(unique_Z, counts_Z))
    perc_Z = np.round(100.*(counts_Z/sum(counts_Z)), 1)
    dict_data_perc_Z = dict(zip(unique_Z, perc_Z))
    total_samples = len(data_Y_AD) + len(data_Y_NC)

    print("\nBalanced: ")
    print("AD Samples: ", str(len(data_Y_AD)))
    print("CN Samples: ", str(len(data_Y_NC)))
    print("Count per Gender: ", dict_data_Z)
    print("AD Percentage: {:.2f}%".format(100*len(data_Y_AD)/total_samples))
    print("CN Percentage: {:.2f}%".format(100*len(data_Y_NC)/total_samples))
    print("Gender Percentage.: ", dict_data_perc_Z)
    print("Total Set Size: ", str(total_samples))
    print('Image Dimensions: ', mri_dim)
    with open(path + 'data-info.txt', 'a') as info_file:
        print("AD Samples: ", str(len(data_Y_AD)), file=info_file)
        print("CN Samples: ", str(len(data_Y_NC)), file=info_file)
        print("Count per Gender: ", dict_data_Z, file=info_file)
        print("AD Percentage: {:.2f}%".format(
            100*len(data_Y_AD)/total_samples),  file=info_file)
        print("CN Percentage: {:.2f}%".format(
            100*len(data_Y_NC)/total_samples), file=info_file)
        print("Gender Percentage: ", dict_data_perc_Z, file=info_file)
        print("Total Set Size: ", str(total_samples), file=info_file)
        print('Image Dimensions: ', mri_dim, file=info_file)


def add_padding(i):
    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1]] = i
    return i_w_pad


def plot_slices(image):
    index_midle = round(len(image[0, :, 0, 0]) / 2)

    img1 = add_padding(image[:, index_midle - 40, :, 0])
    img2 = add_padding(image[:, index_midle - 30, :, 0])
    img3 = add_padding(image[:, index_midle - 20, :, 0])
    img4 = add_padding(image[:, index_midle - 10, :, 0])
    img5 = add_padding(image[:, index_midle, :, 0])
    img6 = add_padding(image[:, index_midle + 10, :, 0])
    img7 = add_padding(image[:, index_midle + 20, :, 0])
    img8 = add_padding(image[:, index_midle + 30, :, 0])
    img9 = add_padding(image[:, index_midle + 40, :, 0])

    fig = plt.figure(figsize=[8, 8])

    ax1 = fig.add_subplot(331)
    ax2 = fig.add_subplot(332)
    ax3 = fig.add_subplot(333)
    ax4 = fig.add_subplot(334)
```

```
    ax5 = fig.add_subplot(335)
    ax6 = fig.add_subplot(336)
    ax7 = fig.add_subplot(337)
    ax8 = fig.add_subplot(338)
    ax9 = fig.add_subplot(339)

    ax1.title.set_text(str(index_midle - 40) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax2.title.set_text(str(index_midle - 30) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax3.title.set_text(str(index_midle - 20) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax4.title.set_text(str(index_midle - 10) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax5.title.set_text(str(index_midle) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax6.title.set_text(str(index_midle + 10) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax7.title.set_text(str(index_midle + 20) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax8.title.set_text(str(index_midle + 30) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')
    ax9.title.set_text(str(index_midle + 40) + " / " +
                       str(len(image[0, :, 0, 0])) + ' Slice')

    ax1.imshow(img1, cmap="gray")
    ax2.imshow(img2, cmap="gray")
    ax3.imshow(img3, cmap="gray")
    ax4.imshow(img4, cmap="gray")
    ax5.imshow(img5, cmap="gray")
    ax6.imshow(img6, cmap="gray")
    ax7.imshow(img7, cmap="gray")
    ax8.imshow(img8, cmap="gray")
    ax9.imshow(img9, cmap="gray")

    ax1.axis("off")
    ax2.axis("off")
    ax3.axis("off")
    ax4.axis("off")
    ax5.axis("off")
    ax6.axis("off")
    ax7.axis("off")
    ax8.axis("off")
    ax9.axis("off")

    plt.show()


def read_patient(c, patient_folder, patient_id, patient_class, patient_sex):
    # data_list_X = []
    # data_list_Y = []
    data_list_Z = []
    data_list_X_AD = []
    data_list_Y_AD = []
    group_AD = []
    data_list_X_NC = []
    data_list_Y_NC = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    global ad_counter
    b = 1
    for file in os.listdir(patient_folder):

        if ((patient_class == "CN") & (nc_counter == 199)):
            break

        if (patient_class == "CN"):
            nc_counter += 1
```

```
        elif (patient_class == "AD"):
            ad_counter += 1

        print("Reading Patient: ", c, "\tPatient ID: ", patient_id, "\tBrain Scan: ", b,
              "\tNC counter: ", nc_counter, "\tAD counter: ", ad_counter)

        # try:
        image = nib.load(os.path.join(
            patient_folder, file)).get_fdata()

        # Display slices of brain
        # plot_slices(image)

        # Remove the last extra dimension
        index_midle = round(len(image[0, :, 0, 0]) / 2)
        image1 = image[:, index_midle, :, 0]
        image2 = image[:, index_midle - 2, :, 0]
        image3 = image[:, index_midle - 4, :, 0]
        image4 = image[:, index_midle + 2, :, 0]
        image5 = image[:, index_midle + 4, :, 0]

        # x.append(i_shape[0])
        # y.append(i_shape[1])
        # z.append(i_shape[2])

        # # Add zeros pading to make all the images the same size
        image1 = add_padding(image1)
        image2 = add_padding(image2)
        image3 = add_padding(image3)
        image4 = add_padding(image4)
        image5 = add_padding(image5)

        # Reduce size to 1/4 of each dimension
        # image = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)

        # Display in Plot Single Slice
        # plt.figure(figsize=[6, 6])
        # plt.subplot(111)
        # plt.imshow(image, cmap="gray")
        # plt.show()

        # # Display in Plot Slices
        # plt.figure(figsize=[12, 6])
        # plt.subplot(121)
        # plt.imshow(image, cmap="gray")

        # image_shrinked = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)
        # plt.subplot(122)
        # plt.imshow(image_shrinked, cmap="gray")
        # plt.show()

        # # Convert to float
        image1 = image1.astype('float32')
        image2 = image2.astype('float32')
        image3 = image3.astype('float32')
        image4 = image4.astype('float32')
        image5 = image5.astype('float32')

        # # Normalize data
        # # Normalize images between 0 and 255
        image1 = (image1 / np.max(image1)) * 255
        image2 = (image2 / np.max(image2)) * 255
        image3 = (image3 / np.max(image3)) * 255
        image4 = (image4 / np.max(image4)) * 255
        image5 = (image5 / np.max(image5)) * 255

        # data_list_X.append(image.flatten())
        # data_list_Y.append(
        #     [target_classes.index(patient_class) + 1])  # Add the index
        data_list_Z.append([patient_sex])
```

```
        if (patient_class == "CN"):

            data_list_X_NC.append(image1.flatten())
            data_list_X_NC.append(image2.flatten())
            data_list_X_NC.append(image3.flatten())
            data_list_X_NC.append(image4.flatten())
            data_list_X_NC.append(image5.flatten())

            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])

            group_NC.append(patient_id)
            group_NC.append(patient_id)
            group_NC.append(patient_id)
            group_NC.append(patient_id)
            group_NC.append(patient_id)

        if (patient_class == "AD"):
            data_list_X_AD.append(image1.flatten())
            data_list_X_AD.append(image2.flatten())
            data_list_X_AD.append(image3.flatten())
            data_list_X_AD.append(image4.flatten())
            data_list_X_AD.append(image5.flatten())

            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])

            group_AD.append(patient_id)
            group_AD.append(patient_id)
            group_AD.append(patient_id)
            group_AD.append(patient_id)
            group_AD.append(patient_id)

        i_shape = image1.shape

        mri_dim = [i_shape[0], i_shape[1]]
        # # To balance the data sets of the classes
        # if((patient_class == "MCI") & (c >= 1)):
        #     if(random.randint(1, 10) > 2):
        #         break
        #     else:
        #         continue

        b += 1
        break
        # except:
        #     print("Corrupted: ", os.path.join(patient_folder, file))
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, group_AD, group_NC
    # return data_list_X, data_list_Y, data_list_Z,  x, y, z, data_list_X_AD, data_list_Y_AD,
data_list_X_NC, data_list_Y_NC


def load_excel(excel_path):
    print(excel_path)
    df = pd.read_csv(excel_path, error_bad_lines=False)
```

```
    df.columns = [column.replace(" ", "_") for column in df.columns]
    print("Total Samples: ", len(df))
    # df.query(
    #     '(Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN") and (
Sex == "M" or Sex == "F")', inplace=True)
    # df.query(
    #     'Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN"', inpla
ce=True)
    df.query(
        '(Research_Group == "AD" or Research_Group == "CN") and (Sex == "M" or Sex == "F")', i
nplace=True)
    df.query(
        'Research_Group == "AD" or Research_Group == "CN"', inplace=True)
    df = df[['Subject_ID', 'Research_Group', 'Sex']]
    print("Total Samples (AD, CN): ", len(df))
    unique_patients = df.drop_duplicates(
        ['Subject_ID', 'Research_Group'], keep='last')
    df = pd.DataFrame(unique_patients)
    excel_data = list(df.itertuples(index=False, name=None))

    print("Unique Patients: ", len(excel_data))
    return excel_data


def read_data(excel_path, input_path):
    excel_data = load_excel(excel_path)
    print("Reading Excel Finidhed")
    # patients_X = []
    # patients_Y = []
    patients_Z = []
    patients_X_AD = []
    patients_Y_AD = []
    patients_X_NC = []
    patients_Y_NC = []
    group_AD = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    nc_counter = 0

    global ad_counter
    ad_counter = 0

    c = 1
    for patient in excel_data:
        # try:
        patient_id = patient[0]
        patient_class = patient[1]
        patient_sex = patient[2]

        if ((patient_class == "CN") & (nc_counter == 199)):
            print("Skip patient: ", patient_id)
            continue

        patient_folder = input_path + patient_id + "/"
        # p_X, p_Y, p_Z, p_x, p_y, p_z, p_X_AD, p_Y_AD, p_X_NC, p_Y_NC = read_patient(patient_
folder,
        #                                                                                 patient_
id, patient_class, patient_sex)
        p_X_AD, p_Y_AD, p_X_NC, p_Y_NC, p_Z, mri_dim, p_group_AD, p_group_NC = read_patient(c,
                                                                                            pa
tient_folder,   patient_id, patient_class, patient_sex)

        # patients_X = patients_X + p_X
        # patients_Y = patients_Y + p_Y
        patients_Z += p_Z
        patients_X_AD += p_X_AD
        patients_Y_AD += p_Y_AD
        patients_X_NC += p_X_NC
```

```
        patients_Y_NC += p_Y_NC
        group_AD += p_group_AD
        group_NC += p_group_NC

        # x = x + p_x
        # y = y + p_y
        # z = z + p_z
        # except:
        #     print("Corrupted Patient: ", patient)
        c += 1
        # if (c == 10):
        #     break

    # Max Dimensions: 35 34 50
    # print("Original Max Dimensions: ", max(x), max(y), max(z))

    # data_list_X = np.array(patients_X)
    # data_list_Y = np.array(patients_Y)
    data_list_Z = np.array(patients_Z)
    data_list_X_AD = np.array(patients_X_AD)
    data_list_Y_AD = np.array(patients_Y_AD)
    data_list_X_NC = np.array(patients_X_NC)
    data_list_Y_NC = np.array(patients_Y_NC)
    data_list_group_AD = np.array(group_AD)
    data_list_group_NC = np.array(group_NC)

    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, data_list_group_AD, data_list_group_NC
    # return data_list_X, data_list_Y, data_list_Z, data_list_X_AD, data_list_Y_AD, data_list_
X_NC, data_list_Y_NC


def main():
    print("Start Reading ...")
    excel_path = "../../raw_data/brains/ADNI_brain.csv"
    input_path = "../../raw_data/brains/ADNI_brain/"
    # matrix_X, matrix_Y, matrix_Z, matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC = read_
data(
    #     excel_path, input_path)
    matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC, matrix_Z, mri_dim, group_AD, group_NC
= read_data(
        excel_path, input_path)

    print("... Reading Finished")

    # Normalize data
    # Normalize images between 0 and 255
    # matrix_X = (matrix_X / matrix_X.max())*255
    matrix_X_AD = (matrix_X_AD / matrix_X_AD.max())*255
    matrix_X_NC = (matrix_X_NC / matrix_X_NC.max())*255

    # Convert to float
    # matrix_X = matrix_X.astype('float32')
    matrix_X_AD = matrix_X_AD.astype('float32')
    matrix_X_NC = matrix_X_NC.astype('float32')

    output_path = "../../data_no_fold/brains-2D-AD-NC-single-5-slices/"
    os.makedirs(output_path)

    # Save into .mat file
    # mdic = {"Z": matrix_X, "y": matrix_Y}
    # scipy.io.savemat(output_path + 'brains.mat', mdic)
    mdic = {"Z": matrix_X_AD, "y": matrix_Y_AD, "group": group_AD}
    scipy.io.savemat(output_path + 'brains_AD.mat', mdic)
    mdic = {"Z": matrix_X_NC, "y": matrix_Y_NC, "group": group_NC}
    scipy.io.savemat(output_path + 'brains_NC.mat', mdic)

    data_info(output_path, matrix_Y_AD, matrix_Y_NC, matrix_Z, mri_dim)


if __name__ == "__main__":
    # execute only if run as a script
```

```
    main()
```

Code Snippet C.2. Creation of the B_2D_5S dataset and convert it to two .mat files, one with the AD and the other with the NC patients, without splitting it into a training, validation, or test set.

C.3 create_B_2D_M.py

This create_B_2D_M.py reads the first excel with the patients' IDs, target classes, and genders, matched the IDS with the content of the folder with the MRI scans to create the B_2D_M dataset. Only a single slice per scan, of multiple MRI scans per patient, are being added to the dataset. The dataset has an equal number of AD, and NC patients.

```
import os
import scipy.io
import skimage.measure
import numpy as np
import pandas as pd
import nibabel as nib
import matplotlib.pyplot as plt
from skimage.transform import resize
import random

max_shape = (174, 174)
target_classes = ["AD", "CN"]
nc_counter = 0
ad_counter = 0
random.seed(5)


def data_info(path, data_Y_AD, data_Y_NC, data_Z, mri_dim):
    # unique_Y, counts_Y = np.unique(data_Y, return_counts=True)
    # dict_data_Y = dict(zip(unique_Y, counts_Y))
    # perc_Y = np.round(100.*(counts_Y/sum(counts_Y)), 1)
    # dict_data_perc_Y = dict(zip(unique_Y, perc_Y))

    unique_Z, counts_Z = np.unique(data_Z, return_counts=True)
    dict_data_Z = dict(zip(unique_Z, counts_Z))
    perc_Z = np.round(100.*(counts_Z/sum(counts_Z)), 1)
    dict_data_perc_Z = dict(zip(unique_Z, perc_Z))
    total_samples = len(data_Y_AD) + len(data_Y_NC)

    print("\nBalanced: ")
    print("AD Samples: ", str(len(data_Y_AD)))
    print("CN Samples: ", str(len(data_Y_NC)))
    print("Count per Gender: ", dict_data_Z)
    print("AD Percentage: {:.2f}%".format(100*len(data_Y_AD)/total_samples))
    print("CN Percentage: {:.2f}%".format(100*len(data_Y_NC)/total_samples))
    print("Gender Percentage.: ", dict_data_perc_Z)
    print("Total Set Size: ", str(total_samples))
    print('Image Dimensions: ', mri_dim)
    with open(path + 'data-info.txt', 'a') as info_file:
        print("AD Samples: ", str(len(data_Y_AD)), file=info_file)
        print("CN Samples: ", str(len(data_Y_NC)), file=info_file)
        print("Count per Gender: ", dict_data_Z, file=info_file)
        print("AD Percentage: {:.2f}%".format(
            100*len(data_Y_AD)/total_samples),  file=info_file)
        print("CN Percentage: {:.2f}%".format(
            100*len(data_Y_NC)/total_samples), file=info_file)
        print("Gender Percentage: ", dict_data_perc_Z, file=info_file)
        print("Total Set Size: ", str(total_samples), file=info_file)
        print('Image Dimensions: ', mri_dim, file=info_file)


def add_padding(i):
```

```
    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1]] = i
    return i_w_pad


def read_patient(c, patient_folder, patient_id, patient_class, patient_sex):
    # data_list_X = []
    # data_list_Y = []
    data_list_Z = []
    data_list_X_AD = []
    data_list_Y_AD = []
    group_AD = []
    data_list_X_NC = []
    data_list_Y_NC = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    global ad_counter
    b = 1
    for file in os.listdir(patient_folder):
        if ((patient_class == "CN") & (nc_counter >= 602)):
            break

        if ((patient_class == "CN") & (b == 4)):
            break

        if (patient_class == "CN"):
            nc_counter += 1
        elif (patient_class == "AD"):
            ad_counter += 1

        print("Reading Patient: ", c, "\tPatient ID: ", patient_id, "\tBrain Scan: ", b,
              "\tNC counter: ", nc_counter, "\tAD counter: ", ad_counter)

        # try:
        image = nib.load(os.path.join(
            patient_folder, file)).get_fdata()

        # Remove the last extra dimension
        index_midle = round(len(image[0, :, 0, 0]) / 2)
        image = image[:, index_midle, :, 0]

        # x.append(i_shape[0])
        # y.append(i_shape[1])
        # z.append(i_shape[2])

        # # Add zeros pading to make all the images the same size
        image = add_padding(image)

        # Reduce size to 1/4 of each dimension
        # image = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)

        # # Display in Plot Slices
        # plt.figure(figsize=[12, 6])
        # plt.subplot(121)
        # plt.imshow(image, cmap="gray")

        # image_shrinked = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)
        # plt.subplot(122)
        # plt.imshow(image_shrinked, cmap="gray")
        # plt.show()

        # # Convert to float
        image = image.astype('float32')

        # # Normalize data
```

```
        # # Normalize images between 0 and 255
        image = (image / np.max(image)) * 255

        # data_list_X.append(image.flatten())
        # data_list_Y.append(
        #     [target_classes.index(patient_class) + 1])  # Add the index
        data_list_Z.append([patient_sex])

        if (patient_class == "CN"):
            data_list_X_NC.append(image.flatten())
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            group_NC.append(patient_id)

        if (patient_class == "AD"):
            data_list_X_AD.append(image.flatten())
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            group_AD.append(patient_id)

        i_shape = image.shape

        mri_dim = [i_shape[0], i_shape[1]]
        # # To balance the data sets of the classes
        # if((patient_class == "MCI") & (c >= 1)):
        #     if(random.randint(1, 10) > 2):
        #         break
        #     else:
        #         continue

        b += 1
        # except:
        #     print("Corrupted: ", os.path.join(patient_folder, file))
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, group_AD, group_NC
    # return data_list_X, data_list_Y, data_list_Z,  x, y, z, data_list_X_AD, data_list_Y_AD,
data_list_X_NC, data_list_Y_NC


def load_excel(excel_path):
    print(excel_path)
    df = pd.read_csv(excel_path, error_bad_lines=False)
    df.columns = [column.replace(" ", "_") for column in df.columns]
    print("Total Samples: ", len(df))
    # df.query(
    #     '(Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN") and (
Sex == "M" or Sex == "F")', inplace=True)
    # df.query(
    #     'Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN"', inpla
ce=True)
    df.query(
        '(Research_Group == "AD" or Research_Group == "CN") and (Sex == "M" or Sex == "F")', i
nplace=True)
    df.query(
        'Research_Group == "AD" or Research_Group == "CN"', inplace=True)
    df = df[['Subject_ID', 'Research_Group', 'Sex']]
    print("Total Samples (AD, CN): ", len(df))
    unique_patients = df.drop_duplicates(
        ['Subject_ID', 'Research_Group'], keep='last')
    df = pd.DataFrame(unique_patients)
    excel_data = list(df.itertuples(index=False, name=None))

    print("Unique Patients: ", len(excel_data))
    return excel_data


def read_data(excel_path, input_path):
    excel_data = load_excel(excel_path)
    print("Reading Excel Finidhed")
    # patients_X = []
    # patients_Y = []
    patients_Z = []
```

```
    patients_X_AD = []
    patients_Y_AD = []
    patients_X_NC = []
    patients_Y_NC = []
    group_AD = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    nc_counter = 0

    global ad_counter
    ad_counter = 0

    c = 1
    for patient in excel_data:
        # try:
        patient_id = patient[0]
        patient_class = patient[1]
        patient_sex = patient[2]

        if ((patient_class == "CN") & (nc_counter >= 602)):
            print("Skip patient: ", patient_id)
            continue

        patient_folder = input_path + patient_id + "/"
        # p_X, p_Y, p_Z, p_x, p_y, p_z, p_X_AD, p_Y_AD, p_X_NC, p_Y_NC = read_patient(patient_
folder,
        #                                                                                 patient_
id, patient_class, patient_sex)
        p_X_AD, p_Y_AD, p_X_NC, p_Y_NC, p_Z, mri_dim, p_group_AD, p_group_NC = read_patient(c,
                                                                                            pa
tient_folder,   patient_id, patient_class, patient_sex)

        # patients_X = patients_X + p_X
        # patients_Y = patients_Y + p_Y
        patients_Z += p_Z
        patients_X_AD += p_X_AD
        patients_Y_AD += p_Y_AD
        patients_X_NC += p_X_NC
        patients_Y_NC += p_Y_NC
        group_AD += p_group_AD
        group_NC += p_group_NC

        # x = x + p_x
        # y = y + p_y
        # z = z + p_z
        # except:
        #     print("Corrupted Patient: ", patient)
        c += 1
        # if (c == 10):
        #     break

    # Max Dimensions: 35 34 50
    # print("Original Max Dimensions: ", max(x), max(y), max(z))

    # data_list_X = np.array(patients_X)
    # data_list_Y = np.array(patients_Y)
    data_list_Z = np.array(patients_Z)
    data_list_X_AD = np.array(patients_X_AD)
    data_list_Y_AD = np.array(patients_Y_AD)
    data_list_X_NC = np.array(patients_X_NC)
    data_list_Y_NC = np.array(patients_Y_NC)
    data_list_group_AD = np.array(group_AD)
    data_list_group_NC = np.array(group_NC)

    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, data_list_group_AD, data_list_group_NC
    # return data_list_X, data_list_Y, data_list_Z, data_list_X_AD, data_list_Y_AD, data_list_
X_NC, data_list_Y_NC
```

```
def main():
    print("Start Reading ...")
    excel_path = "../../raw_data/brains/ADNI_brain.csv"
    input_path = "../../raw_data/brains/ADNI_brain/"
    # matrix_X, matrix_Y, matrix_Z, matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC = read_
data(
    #     excel_path, input_path)
    matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC, matrix_Z, mri_dim, group_AD, group_NC
= read_data(
        excel_path, input_path)

    print("... Reading Finished")

    # Normalize data
    # Normalize images between 0 and 255
    # matrix_X = (matrix_X / matrix_X.max())*255
    matrix_X_AD = (matrix_X_AD / matrix_X_AD.max())*255
    matrix_X_NC = (matrix_X_NC / matrix_X_NC.max())*255

    # Convert to float
    # matrix_X = matrix_X.astype('float32')
    matrix_X_AD = matrix_X_AD.astype('float32')
    matrix_X_NC = matrix_X_NC.astype('float32')

    output_path = "../../data_no_fold/brains-2D-AD-NC-multiple/"
    os.makedirs(output_path)

    # Save into .mat file
    # mdic = {"Z": matrix_X, "y": matrix_Y}
    # scipy.io.savemat(output_path + 'brains.mat', mdic)
    mdic = {"Z": matrix_X_AD, "y": matrix_Y_AD, "group": group_AD}
    scipy.io.savemat(output_path + 'brains_AD.mat', mdic)
    mdic = {"Z": matrix_X_NC, "y": matrix_Y_NC, "group": group_NC}
    scipy.io.savemat(output_path + 'brains_NC.mat', mdic)

    data_info(output_path, matrix_Y_AD, matrix_Y_NC, matrix_Z, mri_dim)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet C.3. Creation of the B_2D_M dataset and convert it to two .mat files, one with the AD and the other with the NC patients, without splitting it to a training, validation, or test set.

C.4 create_B_2D_7M.py

This create_B_2D_7M.py reads the first excel with the patients' IDs, target classes, and genders, matched the IDS with the content of the folder with the MRI scans to create the B_2D_7M dataset. Seven slices per scan, of multiple MRI scans per patient, are being added to the dataset. The dataset has an equal number of AD, and NC patients.

```
import os
import scipy.io
import skimage.measure
import numpy as np
import pandas as pd
import nibabel as nib
import matplotlib.pyplot as plt
from skimage.transform import resize
import random
```

```
max_shape = (174, 174)
target_classes = ["AD", "CN"]
nc_counter = 0
ad_counter = 0
random.seed(5)
# target_classes = ["AD", "MCI", "NC"]


def data_info(path, data_Y_AD, data_Y_NC, data_Z, mri_dim):
    # unique_Y, counts_Y = np.unique(data_Y, return_counts=True)
    # dict_data_Y = dict(zip(unique_Y, counts_Y))
    # perc_Y = np.round(100.*(counts_Y/sum(counts_Y)), 1)
    # dict_data_perc_Y = dict(zip(unique_Y, perc_Y))

    unique_Z, counts_Z = np.unique(data_Z, return_counts=True)
    dict_data_Z = dict(zip(unique_Z, counts_Z))
    perc_Z = np.round(100.*(counts_Z/sum(counts_Z)), 1)
    dict_data_perc_Z = dict(zip(unique_Z, perc_Z))
    total_samples = len(data_Y_AD) + len(data_Y_NC)

    print("\nBalanced: ")
    print("AD Samples: ", str(len(data_Y_AD)))
    print("CN Samples: ", str(len(data_Y_NC)))
    print("Count per Gender: ", dict_data_Z)
    print("AD Percentage: {:.2f}%".format(100*len(data_Y_AD)/total_samples))
    print("CN Percentage: {:.2f}%".format(100*len(data_Y_NC)/total_samples))
    print("Gender Percentage.: ", dict_data_perc_Z)
    print("Total Set Size: ", str(total_samples))
    print('Image Dimensions: ', mri_dim)
    with open(path + 'data-info.txt', 'a') as info_file:
        print("AD Samples: ", str(len(data_Y_AD)), file=info_file)
        print("CN Samples: ", str(len(data_Y_NC)), file=info_file)
        print("Count per Gender: ", dict_data_Z, file=info_file)
        print("AD Percentage: {:.2f}%".format(
            100*len(data_Y_AD)/total_samples),  file=info_file)
        print("CN Percentage: {:.2f}%".format(
            100*len(data_Y_NC)/total_samples), file=info_file)
        print("Gender Percentage: ", dict_data_perc_Z, file=info_file)
        print("Total Set Size: ", str(total_samples), file=info_file)
        print('Image Dimensions: ', mri_dim, file=info_file)


def add_padding(i):
    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1]] = i
    return i_w_pad


def read_patient(c, patient_folder, patient_id, patient_class, patient_sex):
    # data_list_X = []
    # data_list_Y = []
    data_list_Z = []
    data_list_X_AD = []
    data_list_Y_AD = []
    group_AD = []
    data_list_X_NC = []
    data_list_Y_NC = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    global ad_counter
    b = 1
    for file in os.listdir(patient_folder):
        if ((patient_class == "CN") & (nc_counter >= 602)):
            break

        if ((patient_class == "CN") & (b == 4)):
            break
```

```
        if (patient_class == "CN"):
            nc_counter += 1
        elif (patient_class == "AD"):
            ad_counter += 1

        print("Reading Patient: ", c, "\tPatient ID: ", patient_id, "\tBrain Scan: ", b,
              "\tNC counter: ", nc_counter, "\tAD counter: ", ad_counter)

        # try:
        image = nib.load(os.path.join(
            patient_folder, file)).get_fdata()

        # Remove the last extra dimension
        index_midle = round(len(image[0, :, 0, 0]) / 2)
        image1 = image[:, index_midle, :, 0]
        image2 = image[:, index_midle - 2, :, 0]
        image3 = image[:, index_midle - 4, :, 0]
        image4 = image[:, index_midle - 6, :, 0]
        image5 = image[:, index_midle + 2, :, 0]
        image6 = image[:, index_midle + 4, :, 0]
        image7 = image[:, index_midle + 6, :, 0]

        # x.append(i_shape[0])
        # y.append(i_shape[1])
        # z.append(i_shape[2])

        # # Add zeros pading to make all the images the same size
        image1 = add_padding(image1)
        image2 = add_padding(image2)
        image3 = add_padding(image3)
        image4 = add_padding(image4)
        image5 = add_padding(image5)
        image6 = add_padding(image6)
        image7 = add_padding(image7)

        # Reduce size to 1/4 of each dimension
        # image = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)

        # # Display in Plot Slices
        # plt.figure(figsize=[12, 6])
        # plt.subplot(121)
        # plt.imshow(image, cmap="gray")

        # image_shrinked = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)
        # plt.subplot(122)
        # plt.imshow(image_shrinked, cmap="gray")
        # plt.show()

        # # Convert to float
        image1 = image1.astype('float32')
        image2 = image2.astype('float32')
        image3 = image3.astype('float32')
        image4 = image4.astype('float32')
        image5 = image5.astype('float32')
        image6 = image6.astype('float32')
        image7 = image7.astype('float32')

        # # Normalize data
        # # Normalize images between 0 and 255
        image1 = (image1 / np.max(image1)) * 255
        image2 = (image2 / np.max(image2)) * 255
        image3 = (image3 / np.max(image3)) * 255
        image4 = (image4 / np.max(image4)) * 255
        image5 = (image5 / np.max(image5)) * 255
        image6 = (image6 / np.max(image6)) * 255
        image7 = (image7 / np.max(image7)) * 255

        # data_list_X.append(image.flatten())
        # data_list_Y.append(
        #     [target_classes.index(patient_class) + 1])  # Add the index
```

```
        data_list_Z.append([patient_sex])

        if (patient_class == "CN"):
            data_list_X_NC.append(image1.flatten())
            data_list_X_NC.append(image2.flatten())
            data_list_X_NC.append(image3.flatten())
            data_list_X_NC.append(image4.flatten())
            data_list_X_NC.append(image5.flatten())
            data_list_X_NC.append(image6.flatten())
            data_list_X_NC.append(image7.flatten())

            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])

            group_NC.append(patient_id)
            group_NC.append(patient_id)
            group_NC.append(patient_id)
            group_NC.append(patient_id)
            group_NC.append(patient_id)
            group_NC.append(patient_id)
            group_NC.append(patient_id)

        if (patient_class == "AD"):
            data_list_X_AD.append(image1.flatten())
            data_list_X_AD.append(image2.flatten())
            data_list_X_AD.append(image3.flatten())
            data_list_X_AD.append(image4.flatten())
            data_list_X_AD.append(image5.flatten())
            data_list_X_AD.append(image6.flatten())
            data_list_X_AD.append(image7.flatten())

            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])

            group_AD.append(patient_id)
            group_AD.append(patient_id)
            group_AD.append(patient_id)
            group_AD.append(patient_id)
            group_AD.append(patient_id)
            group_AD.append(patient_id)
            group_AD.append(patient_id)

        i_shape = image1.shape

        mri_dim = [i_shape[0], i_shape[1]]
        # # To balance the data sets of the classes
        # if((patient_class == "MCI") & (c >= 1)):
        #     if(random.randint(1, 10) > 2):
        #         break
```

```
        #     else:
        #         continue

        b += 1
        # except:
        #     print("Corrupted: ", os.path.join(patient_folder, file))
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, group_AD, group_NC
    # return data_list_X, data_list_Y, data_list_Z,  x, y, z, data_list_X_AD, data_list_Y_AD,
data_list_X_NC, data_list_Y_NC


def load_excel(excel_path):
    print(excel_path)
    df = pd.read_csv(excel_path, error_bad_lines=False)
    df.columns = [column.replace(" ", "_") for column in df.columns]
    print("Total Samples: ", len(df))
    # df.query(
    #     '(Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN") and (
Sex == "M" or Sex == "F")', inplace=True)
    # df.query(
    #     'Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN"', inpla
ce=True)
    df.query(
        '(Research_Group == "AD" or Research_Group == "CN") and (Sex == "M" or Sex == "F")', i
nplace=True)
    df.query(
        'Research_Group == "AD" or Research_Group == "CN"', inplace=True)
    df = df[['Subject_ID', 'Research_Group', 'Sex']]
    print("Total Samples (AD, CN): ", len(df))
    unique_patients = df.drop_duplicates(
        ['Subject_ID', 'Research_Group'], keep='last')
    df = pd.DataFrame(unique_patients)
    excel_data = list(df.itertuples(index=False, name=None))

    print("Unique Patients: ", len(excel_data))
    return excel_data


def read_data(excel_path, input_path):
    excel_data = load_excel(excel_path)
    print("Reading Excel Finidhed")
    # patients_X = []
    # patients_Y = []
    patients_Z = []
    patients_X_AD = []
    patients_Y_AD = []
    patients_X_NC = []
    patients_Y_NC = []
    group_AD = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    nc_counter = 0

    global ad_counter
    ad_counter = 0

    c = 1
    for patient in excel_data:
        # try:
        patient_id = patient[0]
        patient_class = patient[1]
        patient_sex = patient[2]

        if ((patient_class == "CN") & (nc_counter >= 602)):
            print("Skip patient: ", patient_id)
            continue
```

```
        patient_folder = input_path + patient_id + "/"
        # p_X, p_Y, p_Z, p_x, p_y, p_z, p_X_AD, p_Y_AD, p_X_NC, p_Y_NC = read_patient(patient_
folder,
        #                                                                                 patient_
id, patient_class, patient_sex)
        p_X_AD, p_Y_AD, p_X_NC, p_Y_NC, p_Z, mri_dim, p_group_AD, p_group_NC = read_patient(c,
                                                                                          pa
tient_folder,   patient_id, patient_class, patient_sex)

        # patients_X = patients_X + p_X
        # patients_Y = patients_Y + p_Y
        patients_Z += p_Z
        patients_X_AD += p_X_AD
        patients_Y_AD += p_Y_AD
        patients_X_NC += p_X_NC
        patients_Y_NC += p_Y_NC
        group_AD += p_group_AD
        group_NC += p_group_NC

        # x = x + p_x
        # y = y + p_y
        # z = z + p_z
        # except:
        #     print("Corrupted Patient: ", patient)
        c += 1
        # if (c == 10):
        #     break

    # Max Dimensions: 35 34 50
    # print("Original Max Dimensions: ", max(x), max(y), max(z))

    # data_list_X = np.array(patients_X)
    # data_list_Y = np.array(patients_Y)
    data_list_Z = np.array(patients_Z)
    data_list_X_AD = np.array(patients_X_AD)
    data_list_Y_AD = np.array(patients_Y_AD)
    data_list_X_NC = np.array(patients_X_NC)
    data_list_Y_NC = np.array(patients_Y_NC)
    data_list_group_AD = np.array(group_AD)
    data_list_group_NC = np.array(group_NC)

    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, data_list_group_AD, data_list_group_NC
    # return data_list_X, data_list_Y, data_list_Z, data_list_X_AD, data_list_Y_AD, data_list_
X_NC, data_list_Y_NC


def main():
    print("Start Reading ...")
    excel_path = "../../raw_data/brains/ADNI_brain.csv"
    input_path = "../../raw_data/brains/ADNI_brain/"
    # matrix_X, matrix_Y, matrix_Z, matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC = read_
data(
    #     excel_path, input_path)
    matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC, matrix_Z, mri_dim, group_AD, group_NC
= read_data(
        excel_path, input_path)

    print("... Reading Finished")

    # Normalize data
    # Normalize images between 0 and 255
    # matrix_X = (matrix_X / matrix_X.max())*255
    matrix_X_AD = (matrix_X_AD / matrix_X_AD.max())*255
    matrix_X_NC = (matrix_X_NC / matrix_X_NC.max())*255

    # Convert to float
    # matrix_X = matrix_X.astype('float32')
    matrix_X_AD = matrix_X_AD.astype('float32')
    matrix_X_NC = matrix_X_NC.astype('float32')

    output_path = "../../data_no_fold/7-slices-brains-2D-AD-NC-multiple/"
```

```
    os.makedirs(output_path)

    # Save into .mat file
    # mdic = {"Z": matrix_X, "y": matrix_Y}
    # scipy.io.savemat(output_path + 'brains.mat', mdic)
    mdic = {"Z": matrix_X_AD, "y": matrix_Y_AD, "group": group_AD}
    scipy.io.savemat(output_path + 'brains_AD.mat', mdic)
    mdic = {"Z": matrix_X_NC, "y": matrix_Y_NC, "group": group_NC}
    scipy.io.savemat(output_path + 'brains_NC.mat', mdic)

    data_info(output_path, matrix_Y_AD, matrix_Y_NC, matrix_Z, mri_dim)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet C.4. Creation of the B_2D_7M dataset and convert it to two .mat files, one with the AD and the other with the NC patients, without splitting it into a training, validation, or test set.

C.5 create_B_3D_S.py

This create_B_3D_S.py reads the first excel with the patients' IDs, target classes, and genders, matched the IDS with the content of the folder with the MRI scans to create the B_3D_S dataset. Only a single MRI scan per patient is being added to the dataset, which is being shrunk previously by taking the mean of each $4 \times 4 \times 4$ block. The dataset has an equal number of AD, and NC patients.

```
import os
import scipy.io
import skimage.measure
import numpy as np
import pandas as pd
import nibabel as nib
import matplotlib.pyplot as plt
from skimage.transform import resize
import random

max_shape = (174, 190, 174)
target_classes = ["AD", "CN"]
nc_counter = 0
ad_counter = 0
random.seed(5)


def data_info(path, data_Y_AD, data_Y_NC, data_Z, mri_dim):
    # unique_Y, counts_Y = np.unique(data_Y, return_counts=True)
    # dict_data_Y = dict(zip(unique_Y, counts_Y))
    # perc_Y = np.round(100.*(counts_Y/sum(counts_Y)), 1)
    # dict_data_perc_Y = dict(zip(unique_Y, perc_Y))

    unique_Z, counts_Z = np.unique(data_Z, return_counts=True)
    dict_data_Z = dict(zip(unique_Z, counts_Z))
    perc_Z = np.round(100.*(counts_Z/sum(counts_Z)), 1)
    dict_data_perc_Z = dict(zip(unique_Z, perc_Z))
    total_samples = len(data_Y_AD) + len(data_Y_NC)

    print("\nBalanced: ")
    print("AD Samples: ", str(len(data_Y_AD)))
    print("CN Samples: ", str(len(data_Y_NC)))
    print("Count per Gender: ", dict_data_Z)
    print("AD Percentage: {:.2f}%".format(100*len(data_Y_AD)/total_samples))
    print("CN Percentage: {:.2f}%".format(100*len(data_Y_NC)/total_samples))
```

```
    print("Gender Percentage.: ", dict_data_perc_Z)
    print("Total Set Size: ", str(total_samples))
    print('Image Dimensions: ', mri_dim)
    with open(path + 'data-info.txt', 'a') as info_file:
        print("AD Samples: ", str(len(data_Y_AD)), file=info_file)
        print("CN Samples: ", str(len(data_Y_NC)), file=info_file)
        print("Count per Gender: ", dict_data_Z, file=info_file)
        print("AD Percentage: {:.2f}%".format(
            100*len(data_Y_AD)/total_samples),  file=info_file)
        print("CN Percentage: {:.2f}%".format(
            100*len(data_Y_NC)/total_samples), file=info_file)
        print("Gender Percentage: ", dict_data_perc_Z, file=info_file)
        print("Total Set Size: ", str(total_samples), file=info_file)
        print('Image Dimensions: ', mri_dim, file=info_file)


def add_padding(i):
    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1], :i.shape[2]] = i
    return i_w_pad


def read_patient(c, patient_folder, patient_id, patient_class, patient_sex):
    # data_list_X = []
    # data_list_Y = []
    data_list_Z = []
    data_list_X_AD = []
    data_list_Y_AD = []
    group_AD = []
    data_list_X_NC = []
    data_list_Y_NC = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    global ad_counter
    b = 1
    for file in os.listdir(patient_folder):
        if ((patient_class == "CN") & (nc_counter == 199)):
            break

        if (patient_class == "CN"):
            nc_counter += 1
        elif (patient_class == "AD"):
            ad_counter += 1

        print("Reading Patient: ", c, "\tPatient ID: ", patient_id, "\tBrain Scan: ", b,
              "\tNC counter: ", nc_counter, "\tAD counter: ", ad_counter)

        # try:
        image = nib.load(os.path.join(
            patient_folder, file)).get_fdata()

        # Remove the last extra dimension
        image = image[:, :, :, 0]

        # x.append(i_shape[0])
        # y.append(i_shape[1])
        # z.append(i_shape[2])

        # # Add zeros pading to make all the images the same size
        image = add_padding(image)

        # Reduce size to 1/4 of each dimension
        image = skimage.measure.block_reduce(
            image, (4, 4, 4), np.mean)

        # # Display in Plot Slices
        # plt.figure(figsize=[12, 6])
```

```
        # plt.subplot(121)
        # image_slice = image[:, 120, :]
        # plt.imshow(image_slice, cmap="gray")

        # image_shrinked = skimage.measure.block_reduce(
        #     image, (4, 4, 4), np.mean)
        # plt.subplot(122)
        # image_shrinked_slice = image_shrinked[:, 30, :]
        # plt.imshow(image_shrinked_slice, cmap="gray")
        # plt.show()

        # # Convert to float
        image = image.astype('float32')

        # # Normalize data
        # # Normalize images between 0 and 255
        image = (image / np.max(image)) * 255

        # data_list_X.append(image.flatten())
        # data_list_Y.append(
        #     [target_classes.index(patient_class) + 1])  # Add the index
        data_list_Z.append([patient_sex])

        if (patient_class == "CN"):
            data_list_X_NC.append(image.flatten())
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            group_NC.append(patient_id)

        if (patient_class == "AD"):
            data_list_X_AD.append(image.flatten())
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            group_AD.append(patient_id)

        i_shape = image.shape

        mri_dim = [i_shape[0], i_shape[1], i_shape[2]]
        # # To balance the data sets of the classes
        # if((patient_class == "MCI") & (c >= 1)):
        #     if(random.randint(1, 10) > 2):
        #         break
        #     else:
        #         continue

        b += 1
        break
        # except:
        #     print("Corrupted: ", os.path.join(patient_folder, file))
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, group_AD, group_NC
    # return data_list_X, data_list_Y, data_list_Z,  x, y, z, data_list_X_AD, data_list_Y_AD,
data_list_X_NC, data_list_Y_NC


def load_excel(excel_path):
    print(excel_path)
    df = pd.read_csv(excel_path, error_bad_lines=False)
    df.columns = [column.replace(" ", "_") for column in df.columns]
    print("Total Samples: ", len(df))
    # df.query(
    #     '(Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN") and (
Sex == "M" or Sex == "F")', inplace=True)
    # df.query(
    #     'Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN"', inpla
ce=True)
    df.query(
        '(Research_Group == "AD" or Research_Group == "CN") and (Sex == "M" or Sex == "F")', i
nplace=True)
    df.query(
        'Research_Group == "AD" or Research_Group == "CN"', inplace=True)
    df = df[['Subject_ID', 'Research_Group', 'Sex']]
```

```
    print("Total Samples (AD, CN): ", len(df))
    unique_patients = df.drop_duplicates(
        ['Subject_ID', 'Research_Group'], keep='last')
    df = pd.DataFrame(unique_patients)
    excel_data = list(df.itertuples(index=False, name=None))

    print("Unique Patients: ", len(excel_data))
    return excel_data


def read_data(excel_path, input_path):
    excel_data = load_excel(excel_path)
    print("Reading Excel Finidhed")
    # patients_X = []
    # patients_Y = []
    patients_Z = []
    patients_X_AD = []
    patients_Y_AD = []
    patients_X_NC = []
    patients_Y_NC = []
    group_AD = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    nc_counter = 0

    global ad_counter
    ad_counter = 0

    c = 1
    for patient in excel_data:
        # try:
        patient_id = patient[0]
        patient_class = patient[1]
        patient_sex = patient[2]

        if ((patient_class == "CN") & (nc_counter == 199)):
            print("Skip patient: ", patient_id)
            continue

        patient_folder = input_path + patient_id + "/"
        # p_X, p_Y, p_Z, p_x, p_y, p_z, p_X_AD, p_Y_AD, p_X_NC, p_Y_NC = read_patient(patient_
folder,
        #                                                                                 patient_
id, patient_class, patient_sex)
        p_X_AD, p_Y_AD, p_X_NC, p_Y_NC, p_Z, mri_dim, p_group_AD, p_group_NC = read_patient(c,
                                                                                            pa
tient_folder,   patient_id, patient_class, patient_sex)

        # patients_X = patients_X + p_X
        # patients_Y = patients_Y + p_Y
        patients_Z += p_Z
        patients_X_AD += p_X_AD
        patients_Y_AD += p_Y_AD
        patients_X_NC += p_X_NC
        patients_Y_NC += p_Y_NC
        group_AD += p_group_AD
        group_NC += p_group_NC

        # x = x + p_x
        # y = y + p_y
        # z = z + p_z
        # except:
        #     print("Corrupted Patient: ", patient)
        c += 1
        # if (c == 10):
        #     break

    # Max Dimensions: 35 34 50
```

```
    # print("Original Max Dimensions: ", max(x), max(y), max(z))

    # data_list_X = np.array(patients_X)
    # data_list_Y = np.array(patients_Y)
    data_list_Z = np.array(patients_Z)
    data_list_X_AD = np.array(patients_X_AD)
    data_list_Y_AD = np.array(patients_Y_AD)
    data_list_X_NC = np.array(patients_X_NC)
    data_list_Y_NC = np.array(patients_Y_NC)
    data_list_group_AD = np.array(group_AD)
    data_list_group_NC = np.array(group_NC)

    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, data_list_group_AD, data_list_group_NC
    # return data_list_X, data_list_Y, data_list_Z, data_list_X_AD, data_list_Y_AD, data_list_
X_NC, data_list_Y_NC


def main():
    print("Start Reading ...")
    excel_path = "../../raw_data/shrunk_brains/ADNI_brain.csv"
    input_path = "../../raw_data/shrunk_brains/ADNI_brain/"
    # matrix_X, matrix_Y, matrix_Z, matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC = read_
data(
    #     excel_path, input_path)
    matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC, matrix_Z, mri_dim, group_AD, group_NC
= read_data(
        excel_path, input_path)

    print("... Reading Finished")

    # Normalize data
    # Normalize images between 0 and 255
    # matrix_X = (matrix_X / matrix_X.max())*255
    matrix_X_AD = (matrix_X_AD / matrix_X_AD.max())*255
    matrix_X_NC = (matrix_X_NC / matrix_X_NC.max())*255

    # Convert to float
    # matrix_X = matrix_X.astype('float32')
    matrix_X_AD = matrix_X_AD.astype('float32')
    matrix_X_NC = matrix_X_NC.astype('float32')

    output_path = "../../data_no_fold/shrunk-brains-3D-AD-NC-single/"
    os.makedirs(output_path)

    # Save into .mat file
    # mdic = {"Z": matrix_X, "y": matrix_Y}
    # scipy.io.savemat(output_path + 'brains.mat', mdic)
    mdic = {"Z": matrix_X_AD, "y": matrix_Y_AD, "group": group_AD}
    scipy.io.savemat(output_path + 'brains_AD.mat', mdic)
    mdic = {"Z": matrix_X_NC, "y": matrix_Y_NC, "group": group_NC}
    scipy.io.savemat(output_path + 'brains_NC.mat', mdic)

    data_info(output_path, matrix_Y_AD, matrix_Y_NC, matrix_Z, mri_dim)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet C.5. Creation of the B_3D_S dataset and convert it to two .mat files, one with the AD and the other with the NC patients, without splitting it into a training, validation, or test set.

C.6 create_CB_3D_S.py

This create_CB_3D_S.py reads the first excel with the patients' IDs, target classes, and genders, matched the IDS with the content of the folder with the MRI scans to create the

CB_3D_S dataset. Only a single MRI scan per patient is being added to the dataset, which is being cropped previously in the area of the left hippocampus. The dataset has an equal number of AD, and NC patients.

```
import os
import scipy.io
import skimage.measure
import numpy as np
import pandas as pd
import nibabel as nib
import matplotlib.pyplot as plt
from skimage.transform import resize
import random

max_shape = (174, 190, 174)
target_classes = ["AD", "CN"]
nc_counter = 0
ad_counter = 0
random.seed(5)
# target_classes = ["AD", "MCI", "NC"]


def data_info(path, data_Y_AD, data_Y_NC, data_Z, mri_dim):
    # unique_Y, counts_Y = np.unique(data_Y, return_counts=True)
    # dict_data_Y = dict(zip(unique_Y, counts_Y))
    # perc_Y = np.round(100.*(counts_Y/sum(counts_Y)), 1)
    # dict_data_perc_Y = dict(zip(unique_Y, perc_Y))

    unique_Z, counts_Z = np.unique(data_Z, return_counts=True)
    dict_data_Z = dict(zip(unique_Z, counts_Z))
    perc_Z = np.round(100.*(counts_Z/sum(counts_Z)), 1)
    dict_data_perc_Z = dict(zip(unique_Z, perc_Z))
    total_samples = len(data_Y_AD) + len(data_Y_NC)

    print("\nBalanced: ")
    print("AD Samples: ", str(len(data_Y_AD)))
    print("CN Samples: ", str(len(data_Y_NC)))
    print("Count per Gender: ", dict_data_Z)
    print("AD Percentage: {:.2f}%".format(100*len(data_Y_AD)/total_samples))
    print("CN Percentage: {:.2f}%".format(100*len(data_Y_NC)/total_samples))
    print("Gender Percentage.: ", dict_data_perc_Z)
    print("Total Set Size: ", str(total_samples))
    print('Image Dimensions: ', mri_dim)
    with open(path + 'data-info.txt', 'a') as info_file:
        print("AD Samples: ", str(len(data_Y_AD)), file=info_file)
        print("CN Samples: ", str(len(data_Y_NC)), file=info_file)
        print("Count per Gender: ", dict_data_Z, file=info_file)
        print("AD Percentage: {:.2f}%".format(
            100*len(data_Y_AD)/total_samples),  file=info_file)
        print("CN Percentage: {:.2f}%".format(
            100*len(data_Y_NC)/total_samples), file=info_file)
        print("Gender Percentage: ", dict_data_perc_Z, file=info_file)
        print("Total Set Size: ", str(total_samples), file=info_file)
        print('Image Dimensions: ', mri_dim, file=info_file)


def add_padding(i):
    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1], :i.shape[2]] = i
    return i_w_pad


def plot_slices_z(patient_id, mode, image):
    index_midle = round(len(image[0, 0, :]) / 2)

    img1 = image[:, :, index_midle - 20]
    img2 = image[:, :, index_midle - 15]
```

```
    img3 = image[:, :, index_midle - 10]
    img4 = image[:, :, index_midle - 5]
    img5 = image[:, :, index_midle]
    img6 = image[:, :, index_midle + 5]
    img7 = image[:, :, index_midle + 10]
    img8 = image[:, :, index_midle + 15]
    img9 = image[:, :, index_midle + 20]

    fig = plt.figure(figsize=[8, 8])

    ax1 = fig.add_subplot(331)
    ax2 = fig.add_subplot(332)
    ax3 = fig.add_subplot(333)
    ax4 = fig.add_subplot(334)
    ax5 = fig.add_subplot(335)
    ax6 = fig.add_subplot(336)
    ax7 = fig.add_subplot(337)
    ax8 = fig.add_subplot(338)
    ax9 = fig.add_subplot(339)

    ax1.title.set_text(str(index_midle - 20) + " / " +
                       str(len(image[0,  0, :])) + ' Slice')
    ax2.title.set_text(str(index_midle - 15) + " / " +
                       str(len(image[0,  0, :])) + ' Slice')
    ax3.title.set_text(str(index_midle - 10) + " / " +
                       str(len(image[0,  0, :])) + ' Slice')
    ax4.title.set_text(str(index_midle - 5) + " / " +
                       str(len(image[0, 0, :])) + ' Slice')
    ax5.title.set_text(str(index_midle) + " / " +
                       str(len(image[0, 0, :])) + ' Slice')
    ax6.title.set_text(str(index_midle + 5) + " / " +
                       str(len(image[0,  0, :])) + ' Slice')
    ax7.title.set_text(str(index_midle + 10) + " / " +
                       str(len(image[0,  0, :])) + ' Slice')
    ax8.title.set_text(str(index_midle + 15) + " / " +
                       str(len(image[0,  0, :])) + ' Slice')
    ax9.title.set_text(str(index_midle + 20) + " / " +
                       str(len(image[0, 0, :])) + ' Slice')

    ax1.imshow(img1, cmap="gray")
    ax2.imshow(img2, cmap="gray")
    ax3.imshow(img3, cmap="gray")
    ax4.imshow(img4, cmap="gray")
    ax5.imshow(img5, cmap="gray")
    ax6.imshow(img6, cmap="gray")
    ax7.imshow(img7, cmap="gray")
    ax8.imshow(img8, cmap="gray")
    ax9.imshow(img9, cmap="gray")

    ax1.axis("off")
    ax2.axis("off")
    ax3.axis("off")
    ax4.axis("off")
    ax5.axis("off")
    ax6.axis("off")
    ax7.axis("off")
    ax8.axis("off")
    ax9.axis("off")

    fig_folder = "../3D_brains_cropped/" + patient_id + "/"
    if not os.path.exists(fig_folder):
        os.makedirs(fig_folder)
    plt.savefig(fig_folder + "x_y_" + mode + ".png")
    plt.show()


def plot_slices_y(patient_id, mode, image):
    index_midle = round(len(image[0, :, 0]) / 2)

    img1 = image[:, index_midle - 20, :]
    img2 = image[:, index_midle - 15, :]
    img3 = image[:, index_midle - 10, :]
```

```
    img4 = image[:, index_midle - 5, :]
    img5 = image[:, index_midle, :]
    img6 = image[:, index_midle + 5, :]
    img7 = image[:, index_midle + 10, :]
    img8 = image[:, index_midle + 15, :]
    img9 = image[:, index_midle + 20, :]

    fig = plt.figure(figsize=[8, 8])

    ax1 = fig.add_subplot(331)
    ax2 = fig.add_subplot(332)
    ax3 = fig.add_subplot(333)
    ax4 = fig.add_subplot(334)
    ax5 = fig.add_subplot(335)
    ax6 = fig.add_subplot(336)
    ax7 = fig.add_subplot(337)
    ax8 = fig.add_subplot(338)
    ax9 = fig.add_subplot(339)

    ax1.title.set_text(str(index_midle - 20) + " / " +
                       str(len(image[0, : 0])) + ' Slice')
    ax2.title.set_text(str(index_midle - 15) + " / " +
                       str(len(image[0, :, 0])) + ' Slice')
    ax3.title.set_text(str(index_midle - 10) + " / " +
                       str(len(image[0, :, 0])) + ' Slice')
    ax4.title.set_text(str(index_midle - 5) + " / " +
                       str(len(image[0, :, 0])) + ' Slice')
    ax5.title.set_text(str(index_midle) + " / " +
                       str(len(image[0, :, 0])) + ' Slice')
    ax6.title.set_text(str(index_midle + 5) + " / " +
                       str(len(image[0, :, 0])) + ' Slice')
    ax7.title.set_text(str(index_midle + 10) + " / " +
                       str(len(image[0, :, 0])) + ' Slice')
    ax8.title.set_text(str(index_midle + 15) + " / " +
                       str(len(image[0, :, 0])) + ' Slice')
    ax9.title.set_text(str(index_midle + 20) + " / " +
                       str(len(image[0, :, 0])) + ' Slice')

    ax1.imshow(img1, cmap="gray")
    ax2.imshow(img2, cmap="gray")
    ax3.imshow(img3, cmap="gray")
    ax4.imshow(img4, cmap="gray")
    ax5.imshow(img5, cmap="gray")
    ax6.imshow(img6, cmap="gray")
    ax7.imshow(img7, cmap="gray")
    ax8.imshow(img8, cmap="gray")
    ax9.imshow(img9, cmap="gray")

    ax1.axis("off")
    ax2.axis("off")
    ax3.axis("off")
    ax4.axis("off")
    ax5.axis("off")
    ax6.axis("off")
    ax7.axis("off")
    ax8.axis("off")
    ax9.axis("off")

    fig_folder = "../3D_brains_cropped/" + patient_id + "/"
    if not os.path.exists(fig_folder):
        os.makedirs(fig_folder)
    plt.savefig(fig_folder + "x_z_" + mode + ".png")
    plt.show()


def plot_slices_x(patient_id, mode, image):
    index_midle = round(len(image[:, 0, 0]) / 2)

    img1 = image[index_midle - 20, :, :]
    img2 = image[index_midle - 15, :, :]
    img3 = image[index_midle - 10, :, :]
    img4 = image[index_midle - 5, :, :]
```

```
    img5 = image[index_midle, :, :]
    img6 = image[index_midle + 5, :, :]
    img7 = image[index_midle + 10, :, :]
    img8 = image[index_midle + 15, :, :]
    img9 = image[index_midle + 20, :, :]

    fig = plt.figure(figsize=[8, 8])

    ax1 = fig.add_subplot(331)
    ax2 = fig.add_subplot(332)
    ax3 = fig.add_subplot(333)
    ax4 = fig.add_subplot(334)
    ax5 = fig.add_subplot(335)
    ax6 = fig.add_subplot(336)
    ax7 = fig.add_subplot(337)
    ax8 = fig.add_subplot(338)
    ax9 = fig.add_subplot(339)

    ax1.title.set_text(str(index_midle - 20) + " / " +
                       str(len(image[:0,  0])) + ' Slice')
    ax2.title.set_text(str(index_midle - 15) + " / " +
                       str(len(image[:, 0,  0])) + ' Slice')
    ax3.title.set_text(str(index_midle - 10) + " / " +
                       str(len(image[:, 0, 0])) + ' Slice')
    ax4.title.set_text(str(index_midle - 5) + " / " +
                       str(len(image[:, 0, 0])) + ' Slice')
    ax5.title.set_text(str(index_midle) + " / " +
                       str(len(image[:, 0,  0])) + ' Slice')
    ax6.title.set_text(str(index_midle + 5) + " / " +
                       str(len(image[:, 0,  0])) + ' Slice')
    ax7.title.set_text(str(index_midle + 10) + " / " +
                       str(len(image[:, 0,  0])) + ' Slice')
    ax8.title.set_text(str(index_midle + 15) + " / " +
                       str(len(image[:, 0,  0])) + ' Slice')
    ax9.title.set_text(str(index_midle + 20) + " / " +
                       str(len(image[:, 0, 0])) + ' Slice')

    ax1.imshow(img1, cmap="gray")
    ax2.imshow(img2, cmap="gray")
    ax3.imshow(img3, cmap="gray")
    ax4.imshow(img4, cmap="gray")
    ax5.imshow(img5, cmap="gray")
    ax6.imshow(img6, cmap="gray")
    ax7.imshow(img7, cmap="gray")
    ax8.imshow(img8, cmap="gray")
    ax9.imshow(img9, cmap="gray")

    ax1.axis("off")
    ax2.axis("off")
    ax3.axis("off")
    ax4.axis("off")
    ax5.axis("off")
    ax6.axis("off")
    ax7.axis("off")
    ax8.axis("off")
    ax9.axis("off")

    fig_folder = "../3D_brains_cropped/" + patient_id + "/"
    if not os.path.exists(fig_folder):
        os.makedirs(fig_folder)
    plt.savefig(fig_folder + "y_z_" + mode + ".png")
    plt.show()


def read_patient(c, patient_folder, patient_id, patient_class, patient_sex):
    # data_list_X = []
    # data_list_Y = []
    data_list_Z = []
    data_list_X_AD = []
    data_list_Y_AD = []
    group_AD = []
    data_list_X_NC = []
```

```
    data_list_Y_NC = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    global ad_counter
    b = 1
    for file in os.listdir(patient_folder):
        if ((patient_class == "CN") & (nc_counter == 199)):
            break

        if (patient_class == "CN"):
            nc_counter += 1
        elif (patient_class == "AD"):
            ad_counter += 1

        print("Reading Patient: ", c, "\tPatient ID: ", patient_id, "\tBrain Scan: ", b,
              "\tNC counter: ", nc_counter, "\tAD counter: ", ad_counter)

        # try:
        image = nib.load(os.path.join(
            patient_folder, file)).get_fdata()

        # Remove the last extra dimension
        image = image[:, :, :, 0]

        # # Display in Plot Slices
        # plt.figure(figsize=[12, 6])
        # plt.subplot(131)
        # i1 = image[int(image.shape[0] / 2), :, :]
        # plt.imshow(i1, cmap="gray")

        # plt.subplot(132)
        # i2 = image[:, int(image.shape[1] / 2), :]
        # plt.imshow(i2, cmap="gray")

        # plt.subplot(133)
        # i3 = image[:, :, int(image.shape[2] / 2)]
        # plt.imshow(i3, cmap="gray")
        # plt.show()

        # print(image.shape)
        # plot_slices_x(patient_id, "original", image)
        # plot_slices_y(patient_id, "original", image)
        # plot_slices_z(patient_id, "original", image)
        # image_cropped = image[:70, 50:125, 25:100]
        image = image[:70, 20:80, 30:90]

        # plot_slices_x(patient_id, "cropped", image_cropped)
        # plot_slices_y(patient_id, "cropped", image_cropped)
        # plot_slices_z(patient_id, "cropped", image_cropped)
        #   :int(image.shape[2] / 2)]
        # print(image_cropped.shape)

        # x.append(i_shape[0])
        # y.append(i_shape[1])
        # z.append(i_shape[2])

        # # Add zeros pading to make all the images the same size
        # image = add_padding(image)

        # Reduce size to 1/4 of each dimension
        # image = skimage.measure.block_reduce(
        #     image, (4, 4, 4), np.mean)

        # # Convert to float
        image = image.astype('float32')

        # # Normalize data
        # # Normalize images between 0 and 255
        image = (image / np.max(image)) * 255
```

```
        # data_list_X.append(image.flatten())
        # data_list_Y.append(
        #     [target_classes.index(patient_class) + 1])  # Add the index
        data_list_Z.append([patient_sex])

        if (patient_class == "CN"):
            data_list_X_NC.append(image.flatten())
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            group_NC.append(patient_id)

        if (patient_class == "AD"):
            data_list_X_AD.append(image.flatten())
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            group_AD.append(patient_id)

        i_shape = image.shape

        mri_dim = [i_shape[0], i_shape[1], i_shape[2]]
        # # To balance the data sets of the classes
        # if((patient_class == "MCI") & (c >= 1)):
        #     if(random.randint(1, 10) > 2):
        #         break
        #     else:
        #         continue

        b += 1
        break
        # except:
        #     print("Corrupted: ", os.path.join(patient_folder, file))
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, group_AD, group_NC
    # return data_list_X, data_list_Y, data_list_Z,  x, y, z, data_list_X_AD, data_list_Y_AD,
data_list_X_NC, data_list_Y_NC


def load_excel(excel_path):
    print(excel_path)
    df = pd.read_csv(excel_path, error_bad_lines=False)
    df.columns = [column.replace(" ", "_") for column in df.columns]
    print("Total Samples: ", len(df))
    # df.query(
    #     '(Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN") and (
Sex == "M" or Sex == "F")', inplace=True)
    # df.query(
    #     'Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN"', inpla
ce=True)
    df.query(
        '(Research_Group == "AD" or Research_Group == "CN") and (Sex == "M" or Sex == "F")', i
nplace=True)
    df.query(
        'Research_Group == "AD" or Research_Group == "CN"', inplace=True)
    df = df[['Subject_ID', 'Research_Group', 'Sex']]
    print("Total Samples (AD, CN): ", len(df))
    unique_patients = df.drop_duplicates(
        ['Subject_ID', 'Research_Group'], keep='last')
    df = pd.DataFrame(unique_patients)
    excel_data = list(df.itertuples(index=False, name=None))

    print("Unique Patients: ", len(excel_data))
    return excel_data


def read_data(excel_path, input_path):
    excel_data = load_excel(excel_path)
    print("Reading Excel Finidhed")
    # patients_X = []
    # patients_Y = []
    patients_Z = []
    patients_X_AD = []
```

```
    patients_Y_AD = []
    patients_X_NC = []
    patients_Y_NC = []
    group_AD = []
    group_NC = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    nc_counter = 0

    global ad_counter
    ad_counter = 0

    c = 1
    for patient in excel_data:
        # try:
        patient_id = patient[0]
        patient_class = patient[1]
        patient_sex = patient[2]

        if ((patient_class == "CN") & (nc_counter == 199)):
            print("Skip patient: ", patient_id)
            continue

        patient_folder = input_path + patient_id + "/"
        # p_X, p_Y, p_Z, p_x, p_y, p_z, p_X_AD, p_Y_AD, p_X_NC, p_Y_NC = read_patient(patient_
folder,
        #                                                                                 patient_
id, patient_class, patient_sex)
        p_X_AD, p_Y_AD, p_X_NC, p_Y_NC, p_Z, mri_dim, p_group_AD, p_group_NC = read_patient(c,
                                                                                            pa
tient_folder,   patient_id, patient_class, patient_sex)

        # patients_X = patients_X + p_X
        # patients_Y = patients_Y + p_Y
        patients_Z += p_Z
        patients_X_AD += p_X_AD
        patients_Y_AD += p_Y_AD
        patients_X_NC += p_X_NC
        patients_Y_NC += p_Y_NC
        group_AD += p_group_AD
        group_NC += p_group_NC

        # x = x + p_x
        # y = y + p_y
        # z = z + p_z
        # except:
        #     print("Corrupted Patient: ", patient)
        c += 1
        # if (c == 10):
        #     break

    # Max Dimensions: 35 34 50
    # print("Original Max Dimensions: ", max(x), max(y), max(z))

    # data_list_X = np.array(patients_X)
    # data_list_Y = np.array(patients_Y)
    data_list_Z = np.array(patients_Z)
    data_list_X_AD = np.array(patients_X_AD)
    data_list_Y_AD = np.array(patients_Y_AD)
    data_list_X_NC = np.array(patients_X_NC)
    data_list_Y_NC = np.array(patients_Y_NC)
    data_list_group_AD = np.array(group_AD)
    data_list_group_NC = np.array(group_NC)

    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_Z, mri_di
m, data_list_group_AD, data_list_group_NC
    # return data_list_X, data_list_Y, data_list_Z, data_list_X_AD, data_list_Y_AD, data_list_
X_NC, data_list_Y_NC
```

```
def main():
    print("Start Reading ...")
    excel_path = "../../raw_data/shrinked_brains/ADNI_brain.csv"
    input_path = "../../raw_data/shrinked_brains/ADNI_brain/"
    # matrix_X, matrix_Y, matrix_Z, matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC = read_
data(
    #     excel_path, input_path)
    matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC, matrix_Z, mri_dim, group_AD, group_NC
= read_data(
        excel_path, input_path)

    print("... Reading Finished")

    # Normalize data
    # Normalize images between 0 and 255
    # matrix_X = (matrix_X / matrix_X.max())*255
    matrix_X_AD = (matrix_X_AD / matrix_X_AD.max())*255
    matrix_X_NC = (matrix_X_NC / matrix_X_NC.max())*255

    # Convert to float
    # matrix_X = matrix_X.astype('float32')
    matrix_X_AD = matrix_X_AD.astype('float32')
    matrix_X_NC = matrix_X_NC.astype('float32')

    output_path = "../../data_no_fold/brains-3D-AD-NC-cropped-single/"
    os.makedirs(output_path)

    # Save into .mat file
    # mdic = {"Z": matrix_X, "y": matrix_Y}
    # scipy.io.savemat(output_path + 'brains.mat', mdic)
    mdic = {"Z": matrix_X_AD, "y": matrix_Y_AD, "group": group_AD}
    scipy.io.savemat(output_path + 'brains_AD.mat', mdic)
    mdic = {"Z": matrix_X_NC, "y": matrix_Y_NC, "group": group_NC}
    scipy.io.savemat(output_path + 'brains_NC.mat', mdic)

    data_info(output_path, matrix_Y_AD, matrix_Y_NC, matrix_Z, mri_dim)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet C.6. Creation of the CB_3D_S dataset and convert it to two .mat files, one with the AD and the other with the NC patients, without splitting it into a training, validation, or test set.

C.7 create_LH_3D_S.py

This create_LH_3D_S.py reads the content of the folder with the Left Hippocampus of the patients to create the B_3D_S dataset. Only a single scan of the Left Hippocampus per patient is being added to the dataset. The dataset has an equal number of AD, and NC patients.

```
import os
import scipy.io
import numpy as np
import matplotlib.pyplot as plt
from skimage.transform import resize
from sklearn.model_selection import train_test_split

max_shape = (37, 32, 50)
target_classes = ["AD", "NC"]
```

```
def add_padding(i):
    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1], :i.shape[2]] = i
    return i_w_pad


def read_files(input_path):
    data_list_X_AD = []
    data_list_Y_AD = []
    group_AD = []
    data_list_X_NC = []
    data_list_Y_NC = []
    group_NC = []
    x = []
    y = []
    z = []

    nc_count = 0
    ad_count = 0

    for file in os.listdir(input_path):
        file_parts = file.split("_")  # Remove "_"
        patient_class = file_parts[-1][:-4]  # Get file's category

        is_left = (file_parts[-3] == "left")  # Is the file left
        is_hippo = (file_parts[-2] == "hippo")  # Is the file hippo or not

        patient_id = file_parts[0] + "_" + file_parts[1] + \
            "_" + file_parts[2]  # Read patients id

        if (is_hippo & is_left):

            if ((nc_count >= 148) & (patient_class == "NC")):
                continue

            # Ignore MCIc
            if ((patient_class == "MCIc") | (patient_class == "MCI")):
                continue

            print("Patient ID: ", patient_id, "\tNC counter: ",
                  nc_count, "\tAD counter: ", ad_count)

            mat = scipy.io.loadmat(input_path + file)
            image = mat['data'].copy()
            i_shape = image.shape

            if ((i_shape[0] == 0) | (i_shape[1] == 0) | (i_shape[2] == 0)):
                continue

            x.append(i_shape[0])
            y.append(i_shape[1])
            z.append(i_shape[2])

            image_w_pad = add_padding(image)

            if (patient_class == "NC"):
                data_list_X_NC.append(image_w_pad.flatten())
                data_list_Y_NC.append(
                    [target_classes.index(patient_class) + 1])
                group_NC.append(patient_id)
                nc_count += 1

            if (patient_class == "AD"):
                data_list_X_AD.append(image_w_pad.flatten())
                data_list_Y_AD.append(
                    [target_classes.index(patient_class) + 1])
                group_AD.append(patient_id)
                ad_count += 1
```

```
            # image_slice = image_w_pad[:, 12, :]
            # plt.imshow(image_slice, cmap="gray")
            # plt.show()

    # Max Dimensions: 35 34 50
    print("Original Max Dimensions: ", max(x), max(y), max(z))
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, group_AD, group_NC


def main():
    print("Start Reading ...")

    input_path = "../../raw_data/hippocampus_single/"
    output_path = "../../data_no_fold/hippo-AD-NC-left-single/"

    AD_X, AD_Y, NC_X, NC_Y, group_AD, group_NC = read_files(input_path)

    matrix_X_AD = np.array(AD_X)
    matrix_Y_AD = np.array(AD_Y)
    matrix_X_NC = np.array(NC_X)
    matrix_Y_NC = np.array(NC_Y)
    data_list_group_AD = np.array(group_AD)
    data_list_group_NC = np.array(group_NC)

    print("... Reading Finished")

    # Normalize data
    # Normalize images between 0 and 255
    matrix_X_AD = (matrix_X_AD / matrix_X_AD.max())*255
    matrix_X_NC = (matrix_X_NC / matrix_X_NC.max())*255

    # Convert to float
    matrix_X_AD = matrix_X_AD.astype('float32')
    matrix_X_NC = matrix_X_NC.astype('float32')

    os.makedirs(output_path)

    mri_dim = [max_shape[0], max_shape[1], max_shape[2]]

    # path = "C:\\Users\\mario\\OneDrive - University of Cyprus\\Thesis\\3D_mat_files\\"
    # Save into .mat file
    mdic = {"Z": matrix_X_AD, "y": matrix_Y_AD, "group": group_AD}
    scipy.io.savemat(output_path + 'hippo_AD.mat', mdic)
    mdic = {"Z": matrix_X_NC, "y": matrix_Y_NC, "group": group_NC}
    scipy.io.savemat(output_path + 'hippo_NC.mat', mdic)
    # mdic = {"Z": test_X, "y": test_Y}
    # scipy.io.savemat(path + 'hippo-test.mat', mdic)

    print('Image dimensions: ', mri_dim)
    print("Total Set Size: AD:", len(matrix_Y_AD), " NC:", len(matrix_Y_NC))
    with open(output_path + 'hippo-info.txt', 'a') as f:
        print('Image dimensions: ', mri_dim, file=f)
        print("Total Set Size: AD:", len(matrix_Y_AD),
              " NC:", len(matrix_Y_NC), file=f)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet C.7. Creation of the LH_3D_S dataset and convert it to two .mat files, one with the AD and the other with the NC patients, without splitting it into a training, validation, or test set.

C.8 create_B_2D_M_AD-MCI-NC.py

This create_B_2D_M_AD-MCI-NC.py reads the first excel with the patients' IDs, target classes, and genders, matched the IDS with the content of the folder with the MRI scans

to create the B_2D_M [AD, MCI, NC] dataset. Only a single slice per scan, of multiple MRI scans per patient, are being added to the dataset. The dataset has an equal number of AD, MCI, and NC patients.

```
import os
import scipy.io
import skimage.measure
import numpy as np
import pandas as pd
import nibabel as nib
import matplotlib.pyplot as plt
from skimage.transform import resize
import random

max_shape = (174, 174)
target_classes = ["AD", "MCI", "CN"]
nc_counter = 0
ad_counter = 0
mci_counter = 0
random.seed(5)


def data_info(path, data_Y_AD, data_Y_NC, data_Y_MCI, data_Z, mri_dim):
    # unique_Y, counts_Y = np.unique(data_Y, return_counts=True)
    # dict_data_Y = dict(zip(unique_Y, counts_Y))
    # perc_Y = np.round(100.*(counts_Y/sum(counts_Y)), 1)
    # dict_data_perc_Y = dict(zip(unique_Y, perc_Y))

    unique_Z, counts_Z = np.unique(data_Z, return_counts=True)
    dict_data_Z = dict(zip(unique_Z, counts_Z))
    perc_Z = np.round(100.*(counts_Z/sum(counts_Z)), 1)
    dict_data_perc_Z = dict(zip(unique_Z, perc_Z))
    total_samples = len(data_Y_AD) + len(data_Y_NC) + len(data_Y_MCI)

    print("\nBalanced: ")
    print("AD Samples: ", str(len(data_Y_AD)))
    print("CN Samples: ", str(len(data_Y_NC)))
    print("MCI Samples: ", str(len(data_Y_MCI)))
    print("Count per Gender: ", dict_data_Z)
    print("AD Percentage: {:.2f}%".format(100*len(data_Y_AD)/total_samples))
    print("CN Percentage: {:.2f}%".format(100*len(data_Y_NC)/total_samples))
    print("MCI Percentage: {:.2f}%".format(100*len(data_Y_MCI)/total_samples))
    print("Gender Percentage.: ", dict_data_perc_Z)
    print("Total Set Size: ", str(total_samples))
    print('Image Dimensions: ', mri_dim)
    with open(path + 'data-info.txt', 'a') as info_file:
        print("AD Samples: ", str(len(data_Y_AD)), file=info_file)
        print("CN Samples: ", str(len(data_Y_NC)), file=info_file)
        print("MCI Samples: ", str(len(data_Y_MCI)), file=info_file)
        print("Count per Gender: ", dict_data_Z, file=info_file)
        print("AD Percentage: {:.2f}%".format(
            100*len(data_Y_AD)/total_samples),  file=info_file)
        print("CN Percentage: {:.2f}%".format(
            100*len(data_Y_NC)/total_samples), file=info_file)
        print("MCI Percentage: {:.2f}%".format(
            100*len(data_Y_MCI)/total_samples), file=info_file)
        print("Gender Percentage: ", dict_data_perc_Z, file=info_file)
        print("Total Set Size: ", str(total_samples), file=info_file)
        print('Image Dimensions: ', mri_dim, file=info_file)


def add_padding(i):
    # Resize the MRI scans to specific shape by adding padding with 0s
    i_w_pad = np.zeros(max_shape)
    i_w_pad[:i.shape[0], :i.shape[1]] = i
    return i_w_pad
```

```
def read_patient(c, patient_folder, patient_id, patient_class, patient_sex):
    # data_list_X = []
    # data_list_Y = []
    data_list_Z = []
    data_list_X_AD = []
    data_list_Y_AD = []
    group_AD = []
    data_list_X_NC = []
    data_list_Y_NC = []
    group_NC = []
    data_list_X_MCI = []
    data_list_Y_MCI = []
    group_MCI = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    global ad_counter
    global mci_counter
    b = 1
    for file in os.listdir(patient_folder):
        # To balance the sets
        if ((patient_class == "CN") & (nc_counter >= 602)):
            break

        if ((patient_class == "MCI") & (mci_counter >= 602)):
            break

        # To limit the samples from the same patient
        if ((patient_class == "CN") & (b == 4)):
            break

        if ((patient_class == "MCI") & (b == 4)):
            break

        if (patient_class == "CN"):
            nc_counter += 1
        elif (patient_class == "AD"):
            ad_counter += 1
        elif (patient_class == "MCI"):
            mci_counter += 1

        print("Reading Patient: ", c, "\tPatient ID: ", patient_id, "\tPatient Class: ", patie
nt_class, "\tBrain Scan: ", b,
              "\tNC counter: ", nc_counter, "\tAD counter: ", ad_counter, "\tMCI counter: ", m
ci_counter)

        # try:
        image = nib.load(os.path.join(
            patient_folder, file)).get_fdata()

        # Remove the last extra dimension
        index_midle = round(len(image[0, :, 0, 0]) / 2)
        image = image[:, index_midle, :, 0]

        # x.append(i_shape[0])
        # y.append(i_shape[1])
        # z.append(i_shape[2])

        # # Add zeros pading to make all the images the same size
        image = add_padding(image)

        # Reduce size to 1/4 of each dimension
        # image = skimage.measure.block_reduce(
        #     image, (2, 2), np.mean)

        # # Display in Plot Slices
        # plt.figure(figsize=[12, 6])
        # plt.subplot(121)
        # plt.imshow(image, cmap="gray")

        # image_shrinked = skimage.measure.block_reduce(
```

```
        #     image, (2, 2), np.mean)
        # plt.subplot(122)
        # plt.imshow(image_shrinked, cmap="gray")
        # plt.show()

        # # Convert to float
        image = image.astype('float32')

        # # Normalize data
        # # Normalize images between 0 and 255
        image = (image / np.max(image)) * 255

        # data_list_X.append(image.flatten())
        # data_list_Y.append(
        #     [target_classes.index(patient_class) + 1])  # Add the index
        data_list_Z.append([patient_sex])

        if (patient_class == "CN"):
            data_list_X_NC.append(image.flatten())
            data_list_Y_NC.append(
                [target_classes.index(patient_class) + 1])
            group_NC.append(patient_id)

        if (patient_class == "AD"):
            data_list_X_AD.append(image.flatten())
            data_list_Y_AD.append(
                [target_classes.index(patient_class) + 1])
            group_AD.append(patient_id)

        if (patient_class == "MCI"):
            data_list_X_MCI.append(image.flatten())
            data_list_Y_MCI.append(
                [target_classes.index(patient_class) + 1])
            group_MCI.append(patient_id)

        i_shape = image.shape

        mri_dim = [i_shape[0], i_shape[1]]
        # # To balance the data sets of the classes
        # if((patient_class == "MCI") & (c >= 1)):
        #     if(random.randint(1, 10) > 2):
        #         break
        #     else:
        #         continue

        b += 1
        # except:
        #     print("Corrupted: ", os.path.join(patient_folder, file))
    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_X_MCI, da
ta_list_Y_MCI, data_list_Z, mri_dim, group_AD, group_NC, group_MCI
    # return data_list_X, data_list_Y, data_list_Z,  x, y, z, data_list_X_AD, data_list_Y_AD,
data_list_X_NC, data_list_Y_NC


def load_excel(excel_path):
    print(excel_path)
    df = pd.read_csv(excel_path, error_bad_lines=False)
    df.columns = [column.replace(" ", "_") for column in df.columns]
    print("Total Samples: ", len(df))
    # df.query(
    #     '(Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN") and (
Sex == "M" or Sex == "F")', inplace=True)
    # df.query(
    #     'Research_Group == "MCI" or Research_Group == "AD" or Research_Group == "CN"', inpla
ce=True)
    df.query(
        '(Research_Group == "AD" or Research_Group == "CN" or Research_Group == "MCI") and (Se
x == "M" or Sex == "F")', inplace=True)
    df.query(
        'Research_Group == "AD" or Research_Group == "CN" or Research_Group == "MCI"', inplace
=True)
    df = df[['Subject_ID', 'Research_Group', 'Sex']]
```

```
    print("Total Samples (AD, CN, MCI): ", len(df))
    unique_patients = df.drop_duplicates(
        ['Subject_ID', 'Research_Group'], keep='last')
    df = pd.DataFrame(unique_patients)
    excel_data = list(df.itertuples(index=False, name=None))

    print("Unique Patients: ", len(excel_data))
    return excel_data


def read_data(excel_path, input_path):
    excel_data = load_excel(excel_path)
    print("Reading Excel Finidhed")
    # patients_X = []
    # patients_Y = []
    patients_Z = []
    patients_X_AD = []
    patients_Y_AD = []
    patients_X_NC = []
    patients_Y_NC = []
    patients_X_MCI = []
    patients_Y_MCI = []
    group_AD = []
    group_NC = []
    group_MCI = []
    # x = []
    # y = []
    # z = []

    global nc_counter
    nc_counter = 0

    global ad_counter
    ad_counter = 0

    global mci_counter
    mci_counter = 0

    c = 1
    for patient in excel_data:
        # try:
        patient_id = patient[0]
        patient_class = patient[1]
        patient_sex = patient[2]

        if ((patient_class == "CN") & (nc_counter >= 602)):
            print("Skip CN patient: ", patient_id)
            continue

        if ((patient_class == "MCI") & (mci_counter >= 602)):
            print("Skip MCI patient: ", patient_id)
            continue

        patient_folder = input_path + patient_id + "/"
        # p_X, p_Y, p_Z, p_x, p_y, p_z, p_X_AD, p_Y_AD, p_X_NC, p_Y_NC = read_patient(patient_
folder,
        #                                                                                 patient_
id, patient_class, patient_sex)
        p_X_AD, p_Y_AD, p_X_NC, p_Y_NC, p_X_MCI, p_Y_MCI, p_Z, mri_dim, p_group_AD, p_group_NC
, p_group_MCI = read_patient(c,

                                patient_folder,   patient_id, patient_class, patient_sex)

        # patients_X = patients_X + p_X
        # patients_Y = patients_Y + p_Y
        patients_Z += p_Z
        patients_X_AD += p_X_AD
        patients_Y_AD += p_Y_AD
        patients_X_NC += p_X_NC
        patients_Y_NC += p_Y_NC
        patients_X_MCI += p_X_MCI
        patients_Y_MCI += p_Y_MCI
```

```
        group_AD += p_group_AD
        group_NC += p_group_NC
        group_MCI += p_group_MCI

        # x = x + p_x
        # y = y + p_y
        # z = z + p_z
        # except:
        #     print("Corrupted Patient: ", patient)
        c += 1
        # if (c == 10):
        #     break

    # Max Dimensions: 35 34 50
    # print("Original Max Dimensions: ", max(x), max(y), max(z))

    # data_list_X = np.array(patients_X)
    # data_list_Y = np.array(patients_Y)
    data_list_Z = np.array(patients_Z)
    data_list_X_AD = np.array(patients_X_AD)
    data_list_Y_AD = np.array(patients_Y_AD)
    data_list_X_NC = np.array(patients_X_NC)
    data_list_Y_NC = np.array(patients_Y_NC)
    data_list_X_MCI = np.array(patients_X_MCI)
    data_list_Y_MCI = np.array(patients_Y_MCI)
    data_list_group_AD = np.array(group_AD)
    data_list_group_NC = np.array(group_NC)
    data_list_group_MCI = np.array(group_MCI)

    return data_list_X_AD, data_list_Y_AD, data_list_X_NC, data_list_Y_NC, data_list_X_MCI, da
ta_list_Y_MCI, data_list_Z, mri_dim, data_list_group_AD, data_list_group_NC, data_list_group_M
CI
    # return data_list_X, data_list_Y, data_list_Z, data_list_X_AD, data_list_Y_AD, data_list_
X_NC, data_list_Y_NC


def main():
    print("Start Reading ...")
    excel_path = "../../raw_data/brains/ADNI_brain.csv"
    input_path = "../../raw_data/brains/ADNI_brain/"
    # matrix_X, matrix_Y, matrix_Z, matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC = read_
data(
    #     excel_path, input_path)
    matrix_X_AD, matrix_Y_AD, matrix_X_NC, matrix_Y_NC, matrix_X_MCI, matrix_Y_MCI, matrix_Z,
mri_dim, group_AD, group_NC, group_MCI = read_data(
        excel_path, input_path)

    print("... Reading Finished")

    # Normalize data
    # Normalize images between 0 and 255
    # matrix_X = (matrix_X / matrix_X.max())*255
    matrix_X_AD = (matrix_X_AD / matrix_X_AD.max())*255
    matrix_X_NC = (matrix_X_NC / matrix_X_NC.max())*255
    matrix_X_MCI = (matrix_X_MCI / matrix_X_MCI.max())*255

    # Convert to float
    # matrix_X = matrix_X.astype('float32')
    matrix_X_AD = matrix_X_AD.astype('float32')
    matrix_X_NC = matrix_X_NC.astype('float32')
    matrix_X_MCI = matrix_X_MCI.astype('float32')

    output_path = "../../data_no_fold/brains-2D-AD-MCI-NC-multiple/"
    os.makedirs(output_path)

    # Save into .mat file
    mdic = {"Z": matrix_X_AD, "y": matrix_Y_AD, "group": group_AD}
    scipy.io.savemat(output_path + 'brains_AD.mat', mdic)
    mdic = {"Z": matrix_X_NC, "y": matrix_Y_NC, "group": group_NC}
    scipy.io.savemat(output_path + 'brains_NC.mat', mdic)
    mdic = {"Z": matrix_X_MCI, "y": matrix_Y_MCI, "group": group_MCI}
    scipy.io.savemat(output_path + 'brains_MCI.mat', mdic)
```

```
    data_info(output_path, matrix_Y_AD, matrix_Y_NC,
              matrix_Y_MCI,  matrix_Z, mri_dim)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet C.8. Creation of the B_2D_m [AD, MCI, NC] dataset and convert it to three .mat files, one with the AD, one with the MCI, and the other with the NC patients, without splitting it into a training, validation, or test set.

Appendix D

KFold Cross-validation

D.1 stratified_group_k_fold.py

The stratified_group_k_fold.py had the code for the StratifiedGroupKFold which is not available in the scikit-learn library. For the StratifiedKFold, and the GroupKFold, the implementations that are used are from the scikit-learn library.

```
from collections import Counter, defaultdict

import numpy as np

from sklearn.model_selection._split import _BaseKFold, _RepeatedSplits
from sklearn.utils.validation import check_random_state

class StratifiedGroupKFold(_BaseKFold):
    """Stratified K-Folds iterator variant with non-overlapping groups.

    This cross-validation object is a variation of StratifiedKFold that returns
    stratified folds with non-overlapping groups. The folds are made by
    preserving the percentage of samples for each class.

    The same group will not appear in two different folds (the number of
    distinct groups have to be at least equal to the number of folds).

    The difference between GroupKFold and StratifiedGroupKFold is that
    the former attempts to create balanced folds such that the number of
    distinct groups are approximately the same in each fold, whereas
    StratifiedGroupKFold attempts to create folds that preserve the
    percentage of samples for each class.

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    ----------
    n_splits : int, default=5
        Number of folds. Must be at least 2.

    shuffle : bool, default=False
        Whether to shuffle each class's samples before splitting into batches.
        Note that the samples within each split will not be shuffled.

    random_state : int or RandomState instance, default=None
        When `shuffle` is True, `random_state` affects the ordering of the
        indices, which controls the randomness of each fold for each class.
        Otherwise, leave `random_state` as `None`.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    Examples
    --------
    >>> import numpy as np
    >>> from sklearn.model_selection import StratifiedGroupKFold
    >>> X = np.ones((17, 2))
    >>> y = np.array([0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0])
    >>> groups = np.array([1, 1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 6, 6, 7, 8, 8])
    >>> cv = StratifiedGroupKFold(n_splits=3)
    >>> for train_idxs, test_idxs in cv.split(X, y, groups):
    ...     print("TRAIN:", groups[train_idxs])
    ...     print("      ", y[train_idxs])
```

```
...     print(" TEST:", groups[test_idxs])
...     print("      ", y[test_idxs])
TRAIN: [2 2 4 5 5 5 5 6 6 7]
       [1 1 1 0 0 0 0 0 0 0]
 TEST: [1 1 3 3 3 8 8]
       [0 0 1 1 1 0 0]
TRAIN: [1 1 3 3 3 4 5 5 5 5 8 8]
       [0 0 1 1 1 1 0 0 0 0 0 0]
 TEST: [2 2 6 6 7]
       [1 1 0 0 0]
TRAIN: [1 1 2 2 3 3 3 6 6 7 8 8]
       [0 0 1 1 1 1 1 0 0 0 0 0]
 TEST: [4 5 5 5 5]
       [1 0 0 0 0]

See also
--------
StratifiedKFold: Takes class information into account to build folds which
    retain class distributions (for binary or multiclass classification
    tasks).

GroupKFold: K-fold iterator variant with non-overlapping groups.
"""

def __init__(self, n_splits=5, shuffle=False, random_state=None):
    super().__init__(n_splits=n_splits, shuffle=shuffle,
                     random_state=random_state)

# Implementation based on this kaggle kernel:
# https://www.kaggle.com/jakubwasikowski/stratified-group-k-fold-cross-validation
def _iter_test_indices(self, X, y, groups):
    y = np.reshape(y, len(y)) - 1
    groups = np.reshape(groups, len(groups))

    print(X.shape)
    print(y.shape)
    print(groups.shape)

    labels_num = np.max(y) + 1
    y_counts_per_group = defaultdict(lambda: np.zeros(labels_num))
    y_distr = Counter()
    for label, group in zip(y, groups):
        y_counts_per_group[group][label] += 1
        y_distr[label] += 1

    y_counts_per_fold = defaultdict(lambda: np.zeros(labels_num))
    groups_per_fold = defaultdict(set)

    groups_and_y_counts = list(y_counts_per_group.items())
    rng = check_random_state(self.random_state)
    if self.shuffle:
        rng.shuffle(groups_and_y_counts)

    for group, y_counts in sorted(groups_and_y_counts,
                                  key=lambda x: -np.std(x[1])):
        best_fold = None
        min_eval = None
        for i in range(self.n_splits):
            y_counts_per_fold[i] += y_counts
            std_per_label = []
            for label in range(labels_num):
                std_per_label.append(np.std(
                    [y_counts_per_fold[j][label] / y_distr[label]
                     for j in range(self.n_splits)]))
            y_counts_per_fold[i] -= y_counts
            fold_eval = np.mean(std_per_label)
            if min_eval is None or fold_eval < min_eval:
                min_eval = fold_eval
                best_fold = i
        y_counts_per_fold[best_fold] += y_counts
        groups_per_fold[best_fold].add(group)

    for i in range(self.n_splits):
```

```
            test_indices = [idx for idx, group in enumerate(groups)
                            if group in groups_per_fold[i]]
            yield test_indices


class RepeatedStratifiedGroupKFold(_RepeatedSplits):
    """Repeated Stratified K-Fold cross validator.
    Repeats Stratified K-Fold with non-overlapping groups n times with
    different randomization in each repetition.

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    ----------
    n_splits : int, default=5
        Number of folds. Must be at least 2.

    n_repeats : int, default=10
        Number of times cross-validator needs to be repeated.

    random_state : int or RandomState instance, default=None
        Controls the generation of the random states for each repetition.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    Examples
    --------
    >>> import numpy as np
    >>> from sklearn.model_selection import RepeatedStratifiedGroupKFold
    >>> X = np.ones((17, 2))
    >>> y = np.array([0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0])
    >>> groups = np.array([1, 1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 6, 6, 7, 8, 8])
    >>> cv = RepeatedStratifiedGroupKFold(n_splits=2, n_repeats=2,
    ...                                   random_state=36851234)
    >>> for train_index, test_index in cv.split(X, y, groups):
    ...     print("TRAIN:", groups[train_idxs])
    ...     print("      ", y[train_idxs])
    ...     print(" TEST:", groups[test_idxs])
    ...     print("      ", y[test_idxs])
    TRAIN: [2 2 4 5 5 5 5 8 8]
           [1 1 1 0 0 0 0 0 0]
     TEST: [1 1 3 3 3 6 6 7]
           [0 0 1 1 1 0 0 0]
    TRAIN: [1 1 3 3 3 6 6 7]
           [0 0 1 1 1 0 0 0]
     TEST: [2 2 4 5 5 5 5 8 8]
           [1 1 1 0 0 0 0 0 0]
    TRAIN: [3 3 3 4 7 8 8]
           [1 1 1 1 0 0 0]
     TEST: [1 1 2 2 5 5 5 5 6 6]
           [0 0 1 1 0 0 0 0 0 0]
    TRAIN: [1 1 2 2 5 5 5 5 6 6]
           [0 0 1 1 0 0 0 0 0 0]
     TEST: [3 3 3 4 7 8 8]
           [1 1 1 1 0 0 0]

    Notes
    -----
    Randomized CV splitters may return different results for each call of
    split. You can make the results identical by setting `random_state`
    to an integer.

    See also
    --------
    RepeatedStratifiedKFold: Repeats Stratified K-Fold n times.
    """

    def __init__(self, n_splits=5, n_repeats=10, random_state=None):
        super().__init__(StratifiedGroupKFold, n_splits=n_splits,
                         n_repeats=n_repeats, random_state=random_state)
```

Code Snippet D.1. StratifiedGroupKFold implementation.

D.2 5-folds

The following codes perform the 5-fold splits in the experiments B_2D_5S, B_2D_7M, B_3D_S, CB_3D_S, LH_3D_S.

D.2.1 5_fold_B_2D_5S.py

```
import random
import scipy.io
import numpy as np
from model_selection.stratified_group_k_fold import StratifiedGroupKFold
import os


def shuffle_in_unison(a, b, c):
    n_elem = a.shape[0]
    indeces = np.random.choice(n_elem, size=n_elem, replace=False)
    return a[indeces], b[indeces], c[indeces]


def is_balanced(train_Y, valid_Y):
    unique, counts = np.unique(train_Y, return_counts=True)
    dict_train = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_train_perc = dict(zip(unique, perc))

    unique, counts = np.unique(valid_Y, return_counts=True)
    dict_valid = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_valid_perc = dict(zip(unique, perc))

    print("\nBalanced Sets: ")
    print("Train: ", dict_train)
    print("Valid: ", dict_valid)
    print("Train_Perc.: ", dict_train_perc)
    print("Valid_Perc.: ", dict_valid_perc)


def create_test_set(c, Z, y, g):
    test_Z = []
    test_y = []
    test_g = []

    elements_to_remove = []

    i = 0
    while (c != 0):
        # Count how many patients are in the same class
        count = 0
        same = True
        current = g[i]
        while (same):
            if (g[i + 1] != current):
                same = False
            current = g[i + 1]
            count += 1
            i += 1
        # If patient has less or equal scans than the needed scans add them in the test set
        if (count <= c):
            # Add this elements in the test set
            for j in range(i - count, i):
                test_Z.append(Z[j])
                test_y.append(y[j])
                test_g.append(g[j])
                elements_to_remove.append(j)
                c -= 1
```

```
    test_Z = np.array(test_Z)
    test_y = np.array(test_y)
    test_g = np.array(test_g)

    # Remove these element from the global set
    print(elements_to_remove)
    Z = np.delete(Z, elements_to_remove, axis=0)
    y = np.delete(y, elements_to_remove, axis=0)
    g = np.delete(g, elements_to_remove, axis=0)
    return test_Z, test_y, test_g, Z, y, g


def split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, path):
    # Subjects per category for testing
    N = 45
    print("Initial Size AD: Z:", len(AD_Z), " y:", len(
        AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("Initial Size NC: Z:", len(NC_Z), " y:", len(
        NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("Initial Size AD: Z:", len(AD_Z), " y:", len(
            AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("Initial Size NC: Z:", len(NC_Z), " y:", len(
            NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Create thhe test set
    test_AD_Z, test_AD_y, test_AD_g, AD_Z, AD_y, AD_g = create_test_set(
        N, AD_Z, AD_y, AD_g)
    test_NC_Z, test_NC_y, test_NC_g,  NC_Z, NC_y, NC_g = create_test_set(
        N, NC_Z, NC_y, NC_g)

    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g.sha
pe)
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g.sha
pe)
    print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
          " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape)
    print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
          " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g)
,
              " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g
.shape, file=f)
        print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g)
,
              " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g
.shape, file=f)
        print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
              " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape, file=f)
        print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
              " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape, file=f)

    # Merge the test data
    test_X = np.concatenate((test_AD_Z, test_NC_Z), axis=0)
    test_y = np.concatenate((test_AD_y, test_NC_y), axis=0)
    test_g = np.concatenate((test_AD_g, test_NC_g), axis=0)

    # Save groups with class
    file_w = open(path + "test_file_class_with_ids.txt", 'w')
    for i in range(len(test_y)):
        print(test_g[i], "\t", test_y[i][0], file=file_w)

    # Save into .mat file
    mdic = {"Z": test_X, "y": test_y}
    scipy.io.savemat(
        path + '5-slices-brains-2D-AD-NC-test.mat', mdic)

    # Merge the two classes
    X = np.concatenate((AD_Z, NC_Z), axis=0)
```

```
    y = np.concatenate((AD_y, NC_y), axis=0)
    groups = np.concatenate((AD_g, NC_g), axis=0)

    # Shuffle them in unison
    X, y, groups = shuffle_in_unison(X, y, groups)

    # Check if they are balanced

    count = 1
    k = 5
    AD = 1
    NC = 2

    gkf = StratifiedGroupKFold(n_splits=k)

    for train_index, valid_index in gkf.split(X, y, groups=groups):
        # select rows
        train_X, valid_X = X[train_index], X[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition
        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f:
            print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f)

        # Save into .mat file
        mdic = {"Z": train_X, "y": train_y}
        scipy.io.savemat(path + '5-slices-brains-2D-AD-NC-train' +
                         str(count) + '.mat', mdic)
        mdic = {"Z": valid_X, "y": valid_y}
        scipy.io.savemat(path + '5-slices-brains-2D-AD-NC-valid' +
                         str(count) + '.mat', mdic)
        count += 1

    count = 1
    # To store the groups
    for train_index, valid_index in gkf.split(X, y, groups=groups):
        # select rows
        train_g, valid_g = groups[train_index], groups[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition

        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f2:
            print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f2)

        # Save into .mat file
        file_w1 = open(path + "train_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(train_y)):
            print(train_g[i], "\t", train_y[i][0], file=file_w1)

        file_w2 = open(path + "valid_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(valid_y)):
            print(valid_g[i], "\t", valid_y[i][0], file=file_w2)

        count += 1

    print("Total samples: ", len(y))
    with open(path + 'split-info.txt', 'a') as f:
        print("Total samples: ", len(y), file=f)
```

```
def main():
    input_path = "../../data_no_fold/brains-2D-AD-NC-single-5-slices/"

    mat_AD = scipy.io.loadmat(input_path + "brains_AD.mat")
    mat_NC = scipy.io.loadmat(input_path + "brains_NC.mat")
    AD_Z = mat_AD['Z']
    AD_y = mat_AD['y']
    AD_g = mat_AD['group']
    NC_Z = mat_NC['Z']
    NC_y = mat_NC['y']
    NC_g = mat_NC['group']

    output_path = "../../data_with_5_fold/brains-2D-AD-NC-single-5-slices/"
    os.makedirs(output_path)

    split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, output_path)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet D.2. 5-fold Cross-validation with StratifiedGroupKFold (Appendix D.1) for the B_2D_5S dataset.

D.2.2 5_fold_B_2D_7M.py

```
import random
import scipy.io
import numpy as np
from model_selection.stratified_group_k_fold import StratifiedGroupKFold
import os


def shuffle_in_unison(a, b, c):
    n_elem = a.shape[0]
    indeces = np.random.choice(n_elem, size=n_elem, replace=False)
    return a[indeces], b[indeces], c[indeces]


def is_balanced(train_Y, valid_Y):
    unique, counts = np.unique(train_Y, return_counts=True)
    dict_train = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_train_perc = dict(zip(unique, perc))

    unique, counts = np.unique(valid_Y, return_counts=True)
    dict_valid = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_valid_perc = dict(zip(unique, perc))

    print("\nBalanced Sets: ")
    print("Train: ", dict_train)
    print("Valid: ", dict_valid)
    print("Train_Perc.: ", dict_train_perc)
    print("Valid_Perc.: ", dict_valid_perc)



def create_test_set(c, Z, y, g):
    test_Z = []
    test_y = []
    test_g = []

    elements_to_remove = []

    i = 0
    while (c != 0):
```

```
        # Count how many patients are in the same class
        count = 0
        same = True
        current = g[i]
        while (same):
            if (g[i + 1] != current):
                same = False
            current = g[i + 1]
            count += 1
            i += 1
        # If patient has less or equal scans than the needed scans add them in the test set
        if (count <= c):
            # Add this elements in the test set
            for j in range(i - count, i):
                test_Z.append(Z[j])
                test_y.append(y[j])
                test_g.append(g[j])
                elements_to_remove.append(j)
                c -= 1

    test_Z = np.array(test_Z)
    test_y = np.array(test_y)
    test_g = np.array(test_g)

    # Remove these element from the global set
    print(elements_to_remove)
    Z = np.delete(Z, elements_to_remove, axis=0)
    y = np.delete(y, elements_to_remove, axis=0)
    g = np.delete(g, elements_to_remove, axis=0)
    return test_Z, test_y, test_g, Z, y, g


def split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, path):
    # Subjects per category for testing
    N = 154
    print("Initial Size AD: Z:", len(AD_Z), " y:", len(
        AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("Initial Size NC: Z:", len(NC_Z), " y:", len(
        NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("Initial Size AD: Z:", len(AD_Z), " y:", len(
            AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("Initial Size NC: Z:", len(NC_Z), " y:", len(
            NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Create thhe test set
    test_AD_Z, test_AD_y, test_AD_g, AD_Z, AD_y, AD_g = create_test_set(
        N, AD_Z, AD_y, AD_g)
    test_NC_Z, test_NC_y, test_NC_g,  NC_Z, NC_y, NC_g = create_test_set(
        N, NC_Z, NC_y, NC_g)

    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g.sha
pe)
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g.sha
pe)
    print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
          " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape)
    print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
          " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g)
,
              " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g
.shape, file=f)
        print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g)
,
              " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g
.shape, file=f)
        print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
              " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape, file=f)
```

```
        print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
              " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape, file=f)

    # Merge the test data
    test_X = np.concatenate((test_AD_Z, test_NC_Z), axis=0)
    test_y = np.concatenate((test_AD_y, test_NC_y), axis=0)
    test_g = np.concatenate((test_AD_g, test_NC_g), axis=0)

    # Save groups with class
    file_w = open(path + "test_file_class_with_ids.txt", 'w')
    for i in range(len(test_y)):
        print(test_g[i], "\t", test_y[i][0], file=file_w)

    # Save into .mat file
    mdic = {"Z": test_X, "y": test_y}
    scipy.io.savemat(
        path + '7-slices-brains-2D-AD-NC-test.mat', mdic)

    # Merge the two classes
    X = np.concatenate((AD_Z, NC_Z), axis=0)
    y = np.concatenate((AD_y, NC_y), axis=0)
    groups = np.concatenate((AD_g, NC_g), axis=0)

    # Shuffle them in unison
    X, y, groups = shuffle_in_unison(X, y, groups)

    # Check if they are balanced

    count = 1
    k = 5
    AD = 1
    NC = 2

    gkf = StratifiedGroupKFold(n_splits=k)

    for train_index, valid_index in gkf.split(X, y, groups=groups):
        # select rows
        train_X, valid_X = X[train_index], X[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition
        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f:
            print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f)

        # Save into .mat file
        mdic = {"Z": train_X, "y": train_y}
        scipy.io.savemat(path + '7-slices-brains-2D-AD-NC-train' +
                         str(count) + '.mat', mdic)
        mdic = {"Z": valid_X, "y": valid_y}
        scipy.io.savemat(path + '7-slices-brains-2D-AD-NC-valid' +
                         str(count) + '.mat', mdic)
        count += 1

    count = 1
    # To store the groups
    for train_index, valid_index in gkf.split(X, y, groups=groups):
        # select rows
        train_g, valid_g = groups[train_index], groups[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition

        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
```

```
        with open(path + 'split-info.txt', 'a') as f2:
            print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f2)

        # Save into .mat file
        file_w1 = open(path + "train_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(train_y)):
            print(train_g[i], "\t", train_y[i][0], file=file_w1)

        file_w2 = open(path + "valid_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(valid_y)):
            print(valid_g[i], "\t", valid_y[i][0], file=file_w2)

        count += 1

    print("Total samples: ", len(y))
    with open(path + 'split-info.txt', 'a') as f:
        print("Total samples: ", len(y), file=f)


def main():
    input_path = "../../data_no_fold/7-slices-brains-2D-AD-NC-multiple/"

    mat_AD = scipy.io.loadmat(input_path + "brains_AD.mat")
    mat_NC = scipy.io.loadmat(input_path + "brains_NC.mat")
    AD_Z = mat_AD['Z']
    AD_y = mat_AD['y']
    AD_g = mat_AD['group']
    NC_Z = mat_NC['Z']
    NC_y = mat_NC['y']
    NC_g = mat_NC['group']

    output_path = "../../data_with_5_fold/7-slices-brains-2D-AD-NC-multiple/"
    os.makedirs(output_path)

    split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, output_path)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet D.3. 5-fold Cross-validation with StratifiedGroupKFold (Appendix D.1) for the B_2D_7M dataset.

D.2.3 5_fold_B_3D_S.py

```
import random
import scipy.io
import numpy as np
from sklearn.model_selection import StratifiedKFold
import os


def shuffle_in_unison(a, b, c):
    n_elem = a.shape[0]
    indeces = np.random.choice(n_elem, size=n_elem, replace=False)
    return a[indeces], b[indeces], c[indeces]


def is_balanced(train_Y, valid_Y):
    unique, counts = np.unique(train_Y, return_counts=True)
    dict_train = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_train_perc = dict(zip(unique, perc))

    unique, counts = np.unique(valid_Y, return_counts=True)
    dict_valid = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
```

```
    dict_valid_perc = dict(zip(unique, perc))

    print("\nBalanced Sets: ")
    print("Train: ", dict_train)
    print("Valid: ", dict_valid)
    print("Train_Perc.: ", dict_train_perc)
    print("Valid_Perc.: ", dict_valid_perc)


def create_test_set(c, Z, y, g):
    test_Z = []
    test_y = []
    test_g = []

    elements_to_remove = []

    i = 0
    while (c != 0):
        # Count how many patients are in the same class
        count = 0
        same = True
        current = g[i]
        while (same):
            if (g[i + 1] != current):
                same = False
            current = g[i + 1]
            count += 1
            i += 1
        # If patient has less or equal scans than the needed scans add them in the test set
        if (count <= c):
            # Add this elements in the test set
            for j in range(i - count, i):
                test_Z.append(Z[j])
                test_y.append(y[j])
                test_g.append(g[j])
                elements_to_remove.append(j)
                c -= 1

    test_Z = np.array(test_Z)
    test_y = np.array(test_y)
    test_g = np.array(test_g)

    # Remove these element from the global set
    print(elements_to_remove)
    Z = np.delete(Z, elements_to_remove, axis=0)
    y = np.delete(y, elements_to_remove, axis=0)
    g = np.delete(g, elements_to_remove, axis=0)
    return test_Z, test_y, test_g, Z, y, g


def split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, path):
    # Subjects per category for testing
    N = 19
    print("Initial Size AD: Z:", len(AD_Z), " y:", len(
        AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("Initial Size NC: Z:", len(NC_Z), " y:", len(
        NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("Initial Size AD: Z:", len(AD_Z), " y:", len(
            AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("Initial Size NC: Z:", len(NC_Z), " y:", len(
            NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Create thhe test set
    test_AD_Z, test_AD_y, test_AD_g, AD_Z, AD_y, AD_g = create_test_set(
        N, AD_Z, AD_y, AD_g)
    test_NC_Z, test_NC_y, test_NC_g,  NC_Z, NC_y, NC_g = create_test_set(
        N, NC_Z, NC_y, NC_g)

    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g.sha
pe)
```

```
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g.sha
pe)
    print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
          " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape)
    print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
          " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g)
,
              " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g
.shape, file=f)
        print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g)
,
              " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g
.shape, file=f)
        print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
              " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape, file=f)
        print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
              " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape, file=f)

    # Merge the test data
    test_X = np.concatenate((test_AD_Z, test_NC_Z), axis=0)
    test_y = np.concatenate((test_AD_y, test_NC_y), axis=0)
    test_g = np.concatenate((test_AD_g, test_NC_g), axis=0)

    # Save groups with class
    file_w = open(path + "test_file_class_with_ids.txt", 'w')
    for i in range(len(test_y)):
        print(test_g[i], "\t", test_y[i][0], file=file_w)

    # Save into .mat file
    mdic = {"Z": test_X, "y": test_y}
    scipy.io.savemat(
        path + 'shrunk-brains-3D-AD-NC-test.mat', mdic)

    # Merge the two classes
    X = np.concatenate((AD_Z, NC_Z), axis=0)
    y = np.concatenate((AD_y, NC_y), axis=0)
    groups = np.concatenate((AD_g, NC_g), axis=0)

    # Shuffle them in unison
    X, y, groups = shuffle_in_unison(X, y, groups)

    # Check if they are balanced

    count = 1
    k = 5
    AD = 1
    NC = 2

    kfold = StratifiedKFold(n_splits=k, shuffle=True, random_state=1)

    for train_index, valid_index in kfold.split(X, y, groups=groups):
        # select rows
        train_X, valid_X = X[train_index], X[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition
        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f:
            print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f)

        # Save into .mat file
        mdic = {"Z": train_X, "y": train_y}
        scipy.io.savemat(path + 'shrunk-brains-3D-AD-NC-train' +
                         str(count) + '.mat', mdic)
        mdic = {"Z": valid_X, "y": valid_y}
```

```
        scipy.io.savemat(path + 'shrunk-brains-3D-AD-NC-valid' +
                         str(count) + '.mat', mdic)
        count += 1

    count = 1
    # To store the groups
    for train_index, valid_index in kfold.split(groups, y):
        # select rows
        train_g, valid_g = groups[train_index], groups[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition

        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f2:
            print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f2)

        # Save into .mat file
        file_w1 = open(path + "train_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(train_y)):
            print(train_g[i], "\t", train_y[i][0], file=file_w1)

        file_w2 = open(path + "valid_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(valid_y)):
            print(valid_g[i], "\t", valid_y[i][0], file=file_w2)

        count += 1

    print("Total samples: ", len(y))
    with open(path + 'split-info.txt', 'a') as f:
        print("Total samples: ", len(y), file=f)


def main():
    input_path = "../../data_no_fold/shrunk-brains-3D-AD-NC-single/"

    mat_AD = scipy.io.loadmat(input_path + "brains_AD.mat")
    mat_NC = scipy.io.loadmat(input_path + "brains_NC.mat")
    AD_Z = mat_AD['Z']
    AD_y = mat_AD['y']
    AD_g = mat_AD['group']
    NC_Z = mat_NC['Z']
    NC_y = mat_NC['y']
    NC_g = mat_NC['group']

    output_path = "../../data_with_5_fold/shrunk-brains-3D-AD-NC-single/"
    os.makedirs(output_path)

    split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, output_path)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet D.4. 5-fold Cross-validation with StratifiedKFold for the B_3D_S dataset.

D.2.4 5_fold_CB_3D_S.py

```
import random
import scipy.io
import numpy as np
from sklearn.model_selection import StratifiedKFold
```

```
import os

def shuffle_in_unison(a, b, c):
    n_elem = a.shape[0]
    indeces = np.random.choice(n_elem, size=n_elem, replace=False)
    return a[indeces], b[indeces], c[indeces]

def is_balanced(train_Y, valid_Y):
    unique, counts = np.unique(train_Y, return_counts=True)
    dict_train = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_train_perc = dict(zip(unique, perc))

    unique, counts = np.unique(valid_Y, return_counts=True)
    dict_valid = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_valid_perc = dict(zip(unique, perc))

    print("\nBalanced Sets: ")
    print("Train: ", dict_train)
    print("Valid: ", dict_valid)
    print("Train_Perc.: ", dict_train_perc)
    print("Valid_Perc.: ", dict_valid_perc)


def create_test_set(c, Z, y, g):
    test_Z = []
    test_y = []
    test_g = []

    elements_to_remove = []

    i = 0
    while (c != 0):
        # Count how many patients are in the same class
        count = 0
        same = True
        current = g[i]
        while (same):
            if (g[i + 1] != current):
                same = False
            current = g[i + 1]
            count += 1
            i += 1
        # If  patient has less or equal scans than the needed scans add them in the test set
        if (count <= c):
            # Add this elements in the test set
            for j in range(i - count, i):
                test_Z.append(Z[j])
                test_y.append(y[j])
                test_g.append(g[j])
                elements_to_remove.append(j)
                c -= 1

    test_Z = np.array(test_Z)
    test_y = np.array(test_y)
    test_g = np.array(test_g)

    # Remove these element from the global set
    print(elements_to_remove)
    Z = np.delete(Z, elements_to_remove, axis=0)
    y = np.delete(y, elements_to_remove, axis=0)
    g = np.delete(g, elements_to_remove, axis=0)
    return test_Z, test_y, test_g, Z, y, g


def split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, path):
    # Subjects per category for testing
    N = 19
    print("Initial Size AD: Z:", len(AD_Z), " y:", len(
        AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("Initial Size NC: Z:", len(NC_Z), " y:", len(
```

```
        NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("Initial Size AD: Z:", len(AD_Z), " y:", len(
            AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("Initial Size NC: Z:", len(NC_Z), " y:", len(
            NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Create thhe test set
    test_AD_Z, test_AD_y, test_AD_g, AD_Z, AD_y, AD_g = create_test_set(
        N, AD_Z, AD_y, AD_g)
    test_NC_Z, test_NC_y, test_NC_g,  NC_Z, NC_y, NC_g = create_test_set(
        N, NC_Z, NC_y, NC_g)

    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g.sha
pe)
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g.sha
pe)
    print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
          " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape)
    print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
          " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g)
,
              " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g
.shape, file=f)
        print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g)
,
              " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g
.shape, file=f)
        print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
              " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape, file=f)
        print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
              " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape, file=f)

    # Merge the test data
    test_X = np.concatenate((test_AD_Z, test_NC_Z), axis=0)
    test_y = np.concatenate((test_AD_y, test_NC_y), axis=0)
    test_g = np.concatenate((test_AD_g, test_NC_g), axis=0)

    # Save groups with class
    file_w = open(path + "test_file_class_with_ids.txt", 'w')
    for i in range(len(test_y)):
        print(test_g[i], "\t", test_y[i][0], file=file_w)

    # Save into .mat file
    mdic = {"Z": test_X, "y": test_y}
    scipy.io.savemat(
        path + 'cropped-brains-3D-AD-NC-test.mat', mdic)

    # Merge the two classes
    X = np.concatenate((AD_Z, NC_Z), axis=0)
    y = np.concatenate((AD_y, NC_y), axis=0)
    groups = np.concatenate((AD_g, NC_g), axis=0)

    # Shuffle them in unison
    X, y, groups = shuffle_in_unison(X, y, groups)

    # Check if they are balanced

    count = 1
    k = 5
    AD = 1
    NC = 2

    kfold = StratifiedKFold(n_splits=k, shuffle=True, random_state=1)

    for train_index, valid_index in kfold.split(X, y, groups=groups):
        # select rows
        train_X, valid_X = X[train_index], X[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
```

```
        # summarize train and test composition
        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f:
            print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f)

        # Save into .mat file
        mdic = {"Z": train_X, "y": train_y}
        scipy.io.savemat(path + 'cropped-brains-3D-AD-NC-train' +
                         str(count) + '.mat', mdic)
        mdic = {"Z": valid_X, "y": valid_y}
        scipy.io.savemat(path + 'cropped-brains-3D-AD-NC-valid' +
                         str(count) + '.mat', mdic)
        count += 1

    count = 1
    # To store the groups
    for train_index, valid_index in kfold.split(groups, y):
        # select rows
        train_g, valid_g = groups[train_index], groups[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition

        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f2:
            print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f2)

        # Save into .mat file
        file_w1 = open(path + "train_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(train_y)):
            print(train_g[i], "\t", train_y[i][0], file=file_w1)

        file_w2 = open(path + "valid_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(valid_y)):
            print(valid_g[i], "\t", valid_y[i][0], file=file_w2)

        count += 1

    print("Total samples: ", len(y))
    with open(path + 'split-info.txt', 'a') as f:
        print("Total samples: ", len(y), file=f)


def main():
    input_path = "../../data_no_fold/brains-3D-AD-NC-cropped-single/"

    mat_AD = scipy.io.loadmat(input_path + "brains_AD.mat")
    mat_NC = scipy.io.loadmat(input_path + "brains_NC.mat")
    AD_Z = mat_AD['Z']
    AD_y = mat_AD['y']
    AD_g = mat_AD['group']
    NC_Z = mat_NC['Z']
    NC_y = mat_NC['y']
    NC_g = mat_NC['group']

    output_path = "../../data_with_5_fold/brains-3D-AD-NC-cropped-single/"
    os.makedirs(output_path)

    split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, output_path)
```

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet D.5. 5-fold Cross-validation with StratifiedKFold for the CB_3D_S dataset.

D.2.5 5_fold_LH_3D_S.py

```
import random
import scipy.io
import numpy as np
from sklearn.model_selection import StratifiedKFold
import os


def shuffle_in_unison(a, b, c):
    n_elem = a.shape[0]
    indeces = np.random.choice(n_elem, size=n_elem, replace=False)
    return a[indeces], b[indeces], c[indeces]


def is_balanced(train_Y, valid_Y):
    unique, counts = np.unique(train_Y, return_counts=True)
    dict_train = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_train_perc = dict(zip(unique, perc))

    unique, counts = np.unique(valid_Y, return_counts=True)
    dict_valid = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_valid_perc = dict(zip(unique, perc))

    print("\nBalanced Sets: ")
    print("Train: ", dict_train)
    print("Valid: ", dict_valid)
    print("Train_Perc.: ", dict_train_perc)
    print("Valid_Perc.: ", dict_valid_perc)


def split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, path):
    # Subjects per category for testing
    N = 18
    print("Initial Size AD: Z:", len(AD_Z), " y:", len(
        AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("Initial Size NC: Z:", len(NC_Z), " y:", len(
        NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("Initial Size AD: Z:", len(AD_Z), " y:", len(
            AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("Initial Size NC: Z:", len(NC_Z), " y:", len(
            NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Get test elements
    test_AD_Z = AD_Z[len(AD_Z)-N: len(AD_Z)]
    test_AD_y = AD_y[len(AD_y)-N: len(AD_y)]
    test_AD_g = AD_g[len(AD_g)-N: len(AD_g)]
    test_NC_Z = NC_Z[len(NC_Z)-N: len(NC_Z)]
    test_NC_y = NC_y[len(NC_y)-N: len(NC_y)]
    test_NC_g = NC_g[len(NC_g)-N: len(NC_g)]

    # Remove the test elements from the initial arrays
    AD_Z = AD_Z[:len(AD_Z)-N]
    AD_y = AD_y[:len(AD_y)-N]
    AD_g = AD_g[:len(AD_g)-N]
    NC_Z = NC_Z[:len(NC_Z)-N]
    NC_y = NC_y[:len(NC_y)-N]
    NC_g = NC_g[:len(NC_g)-N]
```

```
    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g.sha
pe)
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g.sha
pe)
    print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
          " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape)
    print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
          " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y), " g:", len(test_AD_g)
,
              " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, "shape g:", test_AD_g
.shape, file=f)
        print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y), " g:", len(test_NC_g)
,
              " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, "shape g:", test_NC_g
.shape, file=f)
        print("AD size for 5-fold: Z:", len(AD_Z), " y:", len(AD_y), " g:", len(AD_g),
              " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, "shape g:", AD_g.shape, file=f)
        print("NC size for 5-fold: Z:", len(NC_Z), " y:", len(NC_y), " g:", len(NC_g),
              " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, "shape g:", NC_g.shape, file=f)

    # Merge the test data
    test_X = np.concatenate((test_AD_Z, test_NC_Z), axis=0)
    test_y = np.concatenate((test_AD_y, test_NC_y), axis=0)
    test_g = np.concatenate((test_AD_g, test_NC_g), axis=0)

    # Save groups with class
    file_w = open(path + "test_file_class_with_ids.txt", 'w')
    for i in range(len(test_y)):
        print(test_g[i], "\t", test_y[i][0], file=file_w)

    # Save into .mat file
    mdic = {"Z": test_X, "y": test_y}
    scipy.io.savemat(path + 'hippo-3D-left-test.mat', mdic)

    # Merge the two classes
    X = np.concatenate((AD_Z, NC_Z), axis=0)
    y = np.concatenate((AD_y, NC_y), axis=0)
    groups = np.concatenate((AD_g, NC_g), axis=0)

    # Shuffle them in unison
    X, y, groups = shuffle_in_unison(X, y, groups)

    # Check if they are balanced

    count = 1
    k = 5
    AD = 1
    NC = 2

    kfold = StratifiedKFold(n_splits=k, shuffle=True, random_state=1)

    for train_index, valid_index in kfold.split(X, y):
        # select rows
        train_X, valid_X = X[train_index], X[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition

        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f1:
            print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f1)

        # Save into .mat file
        mdic = {"Z": train_X, "y": train_y}
```

```
        scipy.io.savemat(path + 'hippo-3D-left-train' +
                         str(count) + '.mat', mdic)
        mdic = {"Z": valid_X, "y": valid_y}
        scipy.io.savemat(path + 'hippo-3D-left-valid' +
                         str(count) + '.mat', mdic)
        count += 1

    count = 1
    # To store the groups
    for train_index, valid_index in kfold.split(groups, y):
        # select rows
        train_g, valid_g = groups[train_index], groups[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition

        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f2:
            print('>Groups: Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f2)

        # Save into .mat file
        file_w1 = open(path + "train_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(train_y)):
            print(train_g[i], "\t", train_y[i][0], file=file_w1)

        file_w2 = open(path + "valid_file_class_with_ids" +
                       str(count) + ".txt", 'w')
        for i in range(len(valid_y)):
            print(valid_g[i], "\t", valid_y[i][0], file=file_w2)

        count += 1

    print("Total samples: ", len(y))
    with open(path + 'split-info.txt', 'a') as f:
        print("Total samples: ", len(y), file=f)


def main():
    input_path = "../../data_no_fold/hippo-AD-NC-left-single/"

    mat_AD = scipy.io.loadmat(
        input_path + "hippo_AD.mat")
    mat_NC = scipy.io.loadmat(
        input_path + "hippo_NC.mat")
    AD_Z = mat_AD['Z']
    AD_y = mat_AD['y']
    AD_g = mat_AD['group']
    NC_Z = mat_NC['Z']
    NC_y = mat_NC['y']
    NC_g = mat_NC['group']

    output_path = "../../data_with_5_fold/hippo-3D-left-single/"
    os.makedirs(output_path)

    split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g,  output_path)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet D.6. 5-fold Cross-validation with StratifiedKFold for the LH_3D_S dataset.

D.3 10-folds

The following codes perform the 5-fold splits in the experiments B_2D_S, B_2D_M, B_2D_M [AD, MCI, NC].

D.3.1 10_fold_B_2D_S.py

```
import random
import scipy.io
import numpy as np
from sklearn.model_selection import StratifiedKFold
import os


def shuffle_in_unison(a, b):
    rng_state = np.random.get_state()
    np.random.shuffle(a)
    np.random.set_state(rng_state)
    np.random.shuffle(b)
    return a, b


def is_balanced(train_Y, valid_Y):
    unique, counts = np.unique(train_Y, return_counts=True)
    dict_train = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_train_perc = dict(zip(unique, perc))

    unique, counts = np.unique(valid_Y, return_counts=True)
    dict_valid = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_valid_perc = dict(zip(unique, perc))

    print("\nBalanced Sets: ")
    print("Train: ", dict_train)
    print("Valid: ", dict_valid)
    print("Train_Perc.: ", dict_train_perc)
    print("Valid_Perc.: ", dict_valid_perc)


def create_test_set(c, Z, y, g):
    test_Z = []
    test_y = []
    test_g = []

    elements_to_remove = []

    i = 0
    while (c != 0):
        # Count how many patients are in the same class
        count = 0
        same = True
        current = g[i]
        while (same):
            if (g[i + 1] != current):
                same = False
            current = g[i + 1]
            count += 1
            i += 1
        # If patient has less or equal scans than the needed scans add them in the test set
        if (count <= c):
            # Add this elements in the test set
            for j in range(i - count, i):
                test_Z.append(Z[j])
                test_y.append(y[j])
                test_g.append(g[j])
                elements_to_remove.append(j)
                c -= 1

    test_Z = np.array(test_Z)
```

```
    test_y = np.array(test_y)
    test_g = np.array(test_g)

    # Remove these element from the global set
    print(elements_to_remove)
    Z = np.delete(Z, elements_to_remove, axis=0)
    y = np.delete(y, elements_to_remove, axis=0)
    g = np.delete(g, elements_to_remove, axis=0)
    return test_Z, test_y, test_g, Z, y, g


def split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, path):
    # Subjects per category for testing
    N = 9
    print("Initial Size AD: Z:", len(AD_Z), " y:", len(
        AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("Initial Size NC: Z:", len(NC_Z), " y:", len(
        NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("Initial Size AD: Z:", len(AD_Z), " y:", len(
            AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("Initial Size NC: Z:", len(NC_Z), " y:", len(
            NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Create the test set
    test_AD_Z = AD_Z[len(AD_Z)-N: len(AD_Z)]
    test_AD_y = AD_y[len(AD_y)-N: len(AD_y)]
    test_NC_Z = NC_Z[len(NC_Z)-N: len(NC_Z)]
    test_NC_y = NC_y[len(NC_y) - N: len(NC_y)]

    # Remove the test elements from the initial arrays
    AD_Z = AD_Z[:len(AD_Z)-N]
    AD_y = AD_y[:len(AD_y)-N]
    NC_Z = NC_Z[:len(NC_Z)-N]
    NC_y = NC_y[: len(NC_y) - N]

    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape)
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape)
    print("AD size for 10-fold: Z:", len(AD_Z), " y:", len(AD_y),
          " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("NC size for 10-fold: Z:", len(NC_Z), " y:", len(NC_y),
          " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y),
              " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, file=f)
        print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y),
              " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, file=f)
        print("AD size for 10-fold: Z:", len(AD_Z), " y:", len(AD_y),
              " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("NC size for 10-fold: Z:", len(NC_Z), " y:", len(NC_y),
              " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Merge the test data
    test_X = np.concatenate((test_AD_Z, test_NC_Z), axis=0)
    test_y = np.concatenate((test_AD_y, test_NC_y), axis=0)

    # Save into .mat file
    mdic = {"Z": test_X, "y": test_y}
    scipy.io.savemat(
        path + 'single-brains-2D-AD-NC-test.mat', mdic)

    # Merge the two classes
    X = np.concatenate((AD_Z, NC_Z), axis=0)
    y = np.concatenate((AD_y, NC_y), axis=0)

    # Shuffle them in unison
    X, y = shuffle_in_unison(X, y)

    # Check if they are balanced
```

```
    count = 1
    k = 10
    AD = 1
    NC = 2

    kfold = StratifiedKFold(n_splits=k, shuffle=True, random_state=1)

    for train_index, valid_index in kfold.split(X, y):
        # select rows
        train_X, valid_X = X[train_index], X[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition
        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f:
            print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f)

        # Save into .mat file
        mdic = {"Z": train_X, "y": train_y}
        scipy.io.savemat(path + 'single-brains-2D-AD-NC-train' +
                         str(count) + '.mat', mdic)
        mdic = {"Z": valid_X, "y": valid_y}
        scipy.io.savemat(path + 'single-brains-2D-AD-NC-valid' +
                         str(count) + '.mat', mdic)
        count += 1

    print("Total samples: ", len(y))
    with open(path + 'split-info.txt', 'a') as f:
        print("Total samples: ", len(y), file=f)


def main():
    input_path = "../../data_no_fold/brains-2D-AD-NC-single/"

    mat_AD = scipy.io.loadmat(input_path + "brains_AD.mat")
    mat_NC = scipy.io.loadmat(input_path + "brains_NC.mat")
    AD_Z = mat_AD['Z']
    AD_y = mat_AD['y']
    AD_g = mat_AD['group']
    NC_Z = mat_NC['Z']
    NC_y = mat_NC['y']
    NC_g = mat_NC['group']

    output_path = "../../data_with_fold/brains-2D-AD-NC-single/"
    os.makedirs(output_path)

    split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, output_path)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet D.7. 10-fold Cross-validation with StratifiedKFold for the B_2D_S dataset.

D.3.2 10_fold_B_2D_M.py

```
import random
import scipy.io
import numpy as np
from model_selection.stratified_group_k_fold import StratifiedGroupKFold
import os


def shuffle_in_unison(a, b, c):
```

```
    n_elem = a.shape[0]
    indeces = np.random.choice(n_elem, size=n_elem, replace=False)
    return a[indeces], b[indeces], c[indeces]


def is_balanced(train_Y, valid_Y):
    unique, counts = np.unique(train_Y, return_counts=True)
    dict_train = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_train_perc = dict(zip(unique, perc))

    unique, counts = np.unique(valid_Y, return_counts=True)
    dict_valid = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_valid_perc = dict(zip(unique, perc))

    print("\nBalanced Sets: ")
    print("Train: ", dict_train)
    print("Valid: ", dict_valid)
    print("Train_Perc.: ", dict_train_perc)
    print("Valid_Perc.: ", dict_valid_perc)


def create_test_set(c, Z, y, g):
    test_Z = []
    test_y = []
    test_g = []

    elements_to_remove = []

    i = 0
    while (c != 0):
        # Count how many patients are in the same class
        count = 0
        same = True
        current = g[i]
        while (same):
            if (g[i + 1] != current):
                same = False
            current = g[i + 1]
            count += 1
            i += 1
        # If patient has less or equal scans than the needed scans add them in the test set
        if (count <= c):
            # Add this elements in the test set
            for j in range(i - count, i):
                test_Z.append(Z[j])
                test_y.append(y[j])
                test_g.append(g[j])
                elements_to_remove.append(j)
                c -= 1

    test_Z = np.array(test_Z)
    test_y = np.array(test_y)
    test_g = np.array(test_g)

    # Remove these element from the global set
    print(elements_to_remove)
    Z = np.delete(Z, elements_to_remove, axis=0)
    y = np.delete(y, elements_to_remove, axis=0)
    g = np.delete(g, elements_to_remove, axis=0)
    return test_Z, test_y, test_g, Z, y, g


def split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, path):
    # Subjects per category for testing
    # N = 18  # For balanced
    N = 22  # For unbalanced
    print("Initial Size AD: Z:", len(AD_Z), " y:", len(
        AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("Initial Size NC: Z:", len(NC_Z), " y:", len(
        NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
```

```
    with open(path + 'split-info.txt', 'a') as f:
        print("Initial Size AD: Z:", len(AD_Z), " y:", len(
            AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("Initial Size NC: Z:", len(NC_Z), " y:", len(
            NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Create the test set
    test_AD_Z, test_AD_y, test_AD_g, AD_Z, AD_y, AD_g = create_test_set(
        N, AD_Z, AD_y, AD_g)
    test_NC_Z, test_NC_y, test_NC_g,  NC_Z, NC_y, NC_g = create_test_set(
        N, NC_Z, NC_y, NC_g)

    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape)
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape)
    print("AD size for 10-fold: Z:", len(AD_Z), " y:", len(AD_y),
          " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("NC size for 10-fold: Z:", len(NC_Z), " y:", len(NC_y),
          " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y),
              " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, file=f)
        print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y),
              " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, file=f)
        print("AD size for 10-fold: Z:", len(AD_Z), " y:", len(AD_y),
              " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("NC size for 10-fold: Z:", len(NC_Z), " y:", len(NC_y),
              " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)

    # Merge the test data
    test_X = np.concatenate((test_AD_Z, test_NC_Z), axis=0)
    test_y = np.concatenate((test_AD_y, test_NC_y), axis=0)

    # Save into .mat file
    mdic = {"Z": test_X, "y": test_y}
    scipy.io.savemat(
        path + 'multiple-brains-2D-AD-NC-test.mat', mdic)

    # Merge the two classes
    X = np.concatenate((AD_Z, NC_Z), axis=0)
    y = np.concatenate((AD_y, NC_y), axis=0)
    groups = np.concatenate((AD_g, NC_g), axis=0)

    # Shuffle them in unison
    X, y, groups = shuffle_in_unison(X, y, groups)

    # Check if they are balanced

    count = 1
    k = 10
    AD = 1
    NC = 2

    gkf = StratifiedGroupKFold(n_splits=k)

    for train_index, valid_index in gkf.split(X, y, groups=groups):
        # select rows
        train_X, valid_X = X[train_index], X[valid_index]
        train_y, valid_y = y[train_index], y[valid_index]
        # summarize train and test composition
        train_AD, train_NC = len(train_y[train_y == AD]), len(
            train_y[train_y == NC])
        test_AD, test_NC = len(valid_y[valid_y == AD]), len(
            valid_y[valid_y == NC])
        print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
              (train_AD, train_NC, test_AD, test_NC))
        with open(path + 'split-info.txt', 'a') as f:
            print('>Train: AD=%d, NC=%d, Validation: AD=%d, NC=%d' %
                  (train_AD, train_NC, test_AD, test_NC), file=f)

        # Save into .mat file
        mdic = {"Z": train_X, "y": train_y}
```

```
        scipy.io.savemat(path + 'multiple-brains-2D-AD-NC-train' +
                         str(count) + '.mat', mdic)
        mdic = {"Z": valid_X, "y": valid_y}
        scipy.io.savemat(path + 'multiple-brains-2D-AD-NC-valid' +
                         str(count) + '.mat', mdic)
        count += 1

    print("Total samples: ", len(y))
    with open(path + 'split-info.txt', 'a') as f:
        print("Total samples: ", len(y), file=f)


def main():
    input_path = "../../data_no_fold/brains-2D-AD-NC-multiple/"

    mat_AD = scipy.io.loadmat(input_path + "brains_AD.mat")
    mat_NC = scipy.io.loadmat(input_path + "brains_NC.mat")
    AD_Z = mat_AD['Z']
    AD_y = mat_AD['y']
    AD_g = mat_AD['group']
    NC_Z = mat_NC['Z']
    NC_y = mat_NC['y']
    NC_g = mat_NC['group']

    output_path = "../../data_with_fold/brains-2D-AD-NC-multiple/"
    os.makedirs(output_path)

    split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, output_path)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet D.8. 10-fold Cross-validation with StratifiedGroupKFold (Appendix D.1) for the B_2D_M dataset.

D.3.3 10_fold_B_2D_M_AD-MCI-NC.py

```
import random
import scipy.io
import numpy as np
from model_selection.stratified_group_k_fold import StratifiedGroupKFold
import os


def shuffle_in_unison(a, b, c):
    n_elem = a.shape[0]
    indeces = np.random.choice(n_elem, size=n_elem, replace=False)
    return a[indeces], b[indeces], c[indeces]


def is_balanced(train_Y, valid_Y):
    unique, counts = np.unique(train_Y, return_counts=True)
    dict_train = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_train_perc = dict(zip(unique, perc))

    unique, counts = np.unique(valid_Y, return_counts=True)
    dict_valid = dict(zip(unique, counts))
    perc = np.round(100.*(counts/sum(counts)), 1)
    dict_valid_perc = dict(zip(unique, perc))

    print("\nBalanced Sets: ")
    print("Train: ", dict_train)
    print("Valid: ", dict_valid)
    print("Train_Perc.: ", dict_train_perc)
    print("Valid_Perc.: ", dict_valid_perc)
```

```
def create_test_set(c, Z, y, g):
    test_Z = []
    test_y = []
    test_g = []

    elements_to_remove = []

    i = 0
    while (c != 0):
        # Count how many patients are in the same class
        count = 0
        same = True
        current = g[i]
        while (same):
            if (g[i + 1] != current):
                same = False
            current = g[i + 1]
            count += 1
            i += 1
        # If patient has less or equal scans than the needed scans add them in the test set
        if (count <= c):
            # Add this elements in the test set
            for j in range(i - count, i):
                test_Z.append(Z[j])
                test_y.append(y[j])
                test_g.append(g[j])
                elements_to_remove.append(j)
                c -= 1

    test_Z = np.array(test_Z)
    test_y = np.array(test_y)
    test_g = np.array(test_g)

    # Remove these element from the global set
    print(elements_to_remove)
    Z = np.delete(Z, elements_to_remove, axis=0)
    y = np.delete(y, elements_to_remove, axis=0)
    g = np.delete(g, elements_to_remove, axis=0)
    return test_Z, test_y, test_g, Z, y, g


def split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g, MCI_Z, MCI_y, MCI_g, path):
    # Subjects per category for testing
    # N = 18  # For balanced
    N = 22  # For unbalanced
    print("Initial Size AD: Z:", len(AD_Z), " y:", len(
        AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
    print("Initial Size NC: Z:", len(NC_Z), " y:", len(
        NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape)
    print("Initial Size MCI: Z:", len(MCI_Z), " y:", len(
        MCI_y), " shape Z:", MCI_Z.shape, "shape y:", MCI_y.shape)
    with open(path + 'split-info.txt', 'a') as f:
        print("Initial Size AD: Z:", len(AD_Z), " y:", len(
            AD_y), " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
        print("Initial Size NC: Z:", len(NC_Z), " y:", len(
            NC_y), " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)
        print("Initial Size MCI: Z:", len(MCI_Z), " y:", len(
            MCI_y), " shape Z:", MCI_Z.shape, "shape y:", MCI_y.shape, file=f)

    # Create the test set
    test_AD_Z, test_AD_y, test_AD_g, AD_Z, AD_y, AD_g = create_test_set(
        N, AD_Z, AD_y, AD_g)
    test_NC_Z, test_NC_y, test_NC_g,  NC_Z, NC_y, NC_g = create_test_set(
        N, NC_Z, NC_y, NC_g)
    test_MCI_Z, test_MCI_y, test_MCI_g,  MCI_Z, MCI_y, MCI_g = create_test_set(
        N, MCI_Z, MCI_y, MCI_g)

    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape)
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape)
    print("MCI test size: Z:", len(test_MCI_Z), " y:", len(test_MCI_y),
```

```
      " shape Z:", test_MCI_Z.shape, "shape y:", test_MCI_y.shape)
print("AD size for 10-fold: Z:", len(AD_Z), " y:", len(AD_y),
      " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
print("NC size for 10-fold: Z:", len(NC_Z), " y:", len(NC_y),
      " shape Z:", AD_Z.shape, "shape y:", AD_y.shape)
print("MCI size for 10-fold: Z:", len(MCI_Z), " y:", len(MCI_y),
      " shape Z:", MCI_Z.shape, "shape y:", MCI_y.shape)
with open(path + 'split-info.txt', 'a') as f:
    print("AD test size: Z:", len(test_AD_Z), " y:", len(test_AD_y),
          " shape Z:", test_AD_Z.shape, "shape y:", test_AD_y.shape, file=f)
    print("NC test size: Z:", len(test_NC_Z), " y:", len(test_NC_y),
          " shape Z:", test_NC_Z.shape, "shape y:", test_NC_y.shape, file=f)
    print("MCI test size: Z:", len(test_MCI_Z), " y:", len(test_MCI_y),
          " shape Z:", test_MCI_Z.shape, "shape y:", test_MCI_y.shape, file=f)
    print("AD size for 10-fold: Z:", len(AD_Z), " y:", len(AD_y),
          " shape Z:", AD_Z.shape, "shape y:", AD_y.shape, file=f)
    print("NC size for 10-fold: Z:", len(NC_Z), " y:", len(NC_y),
          " shape Z:", NC_Z.shape, "shape y:", NC_y.shape, file=f)
    print("MCI size for 10-fold: Z:", len(MCI_Z), " y:", len(MCI_y),
          " shape Z:", MCI_Z.shape, "shape y:", MCI_y.shape, file=f)

# Merge the test data
test_X = np.concatenate((test_AD_Z, test_NC_Z, test_MCI_Z), axis=0)
test_y = np.concatenate((test_AD_y, test_NC_y, test_MCI_y), axis=0)

# Save into .mat file
mdic = {"Z": test_X, "y": test_y}
scipy.io.savemat(
    path + 'shrinked-brains-2D-AD-NC-test.mat', mdic)

# Merge the two classes
X = np.concatenate((AD_Z, NC_Z, MCI_Z), axis=0)
y = np.concatenate((AD_y, NC_y, MCI_y), axis=0)
groups = np.concatenate((AD_g, NC_g, MCI_g), axis=0)

# Shuffle them in unison
X, y, groups = shuffle_in_unison(X, y, groups)

# Check if they are balanced

count = 1
k = 10
AD = 1
MCI = 2
NC = 3

gkf = StratifiedGroupKFold(n_splits=k)

for train_index, valid_index in gkf.split(X, y, groups=groups):
    # select rows
    train_X, valid_X = X[train_index], X[valid_index]
    train_y, valid_y = y[train_index], y[valid_index]
    # summarize train and test composition
    train_AD, train_NC, train_MCI = len(train_y[train_y == AD]), len(
        train_y[train_y == NC]), len(train_y[train_y == MCI])
    test_AD, test_NC, test_MCI = len(valid_y[valid_y == AD]), len(
        valid_y[valid_y == NC]), len(valid_y[valid_y == MCI])
    print('>Train: AD=%d, NC=%d, MCI=%d Validation: AD=%d, NC=%d, MCI=%d' %
          (train_AD, train_NC, train_MCI, test_AD, test_NC, test_MCI))
    with open(path + 'split-info.txt', 'a') as f:
        print('>Train: AD=%d, NC=%d, MCI=%d Validation: AD=%d, NC=%d, MCI=%d' %
              (train_AD, train_NC, train_MCI, test_AD, test_NC, test_MCI), file=f)

    # Save into .mat file
    mdic = {"Z": train_X, "y": train_y}
    scipy.io.savemat(path + 'brains-2D-AD-MCI-NC-train' +
                     str(count) + '.mat', mdic)
    mdic = {"Z": valid_X, "y": valid_y}
    scipy.io.savemat(path + 'brains-2D-AD-MCI-NC-valid' +
                     str(count) + '.mat', mdic)
    count += 1

print("Total samples: ", len(y))
```

```
    with open(path + 'split-info.txt', 'a') as f:
        print("Total samples: ", len(y), file=f)


def main():
    input_path = "../../data_no_fold/brains-2D-AD-MCI-NC-multiple/"

    mat_AD = scipy.io.loadmat(input_path + "brains_AD.mat")
    mat_NC = scipy.io.loadmat(input_path + "brains_NC.mat")
    mat_MCI = scipy.io.loadmat(input_path + "brains_MCI.mat")
    AD_Z = mat_AD['Z']
    AD_y = mat_AD['y']
    AD_g = mat_AD['group']
    NC_Z = mat_NC['Z']
    NC_y = mat_NC['y']
    NC_g = mat_NC['group']
    MCI_Z = mat_MCI['Z']
    MCI_y = mat_MCI['y']
    MCI_g = mat_MCI['group']

    output_path = "../../data_with_fold/brains-2D-AD-MCI-NC-multiple/"
    os.makedirs(output_path)

    split_sets(AD_Z, AD_y, AD_g, NC_Z, NC_y, NC_g,
               MCI_Z, MCI_y, MCI_g, output_path)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet D.9. 10-fold Cross-validation with StratifiedGroupKFold (Appendix D.1) for the B_2D_M [AD, MCI, NC] dataset.

Appendix E

Run in Arcadia

E.1 Create Virtual Environment

E.1.1 Extract Required Libraries

The following command extracts the required libraries for a Python project created in a virtual environment. If the project was not created in a virtual environment, all the libraries you have installed in your system will be included in the requiremenets.txt.

```
pip freeze > requirements.txt
```

The required libraries will be stored in the requirements.txt file having the following form.

```
pandas==1.1.5
nibabel==3.2.0
numpy==1.16.4
matplotlib==3.1.0
tensorflow==2.0.1
tensorflow-estimator==2.0.1
scipy==1.1.0
scikit-image==0.17.2
scikit-learn==0.21.3
Keras==2.3.1
Keras-Applications==1.0.8
Keras-Preprocessing==1.1.0
opencv-python==4.1.2.30
joblib==0.14.0
hessianfree==0.4.0
```

E.1.2 Create Virtual Environment and Install Libraries

To create a virtual environment and install your libraries to run your Python code in Arcadia, you have to execute the following commands in a terminal:

```
1. export https_proxy='http://proxy.cs.ucy.ac.cy:8008/'
2. export http_proxy='http://proxy.cs.ucy.ac.cy:8008/'
3. python -m venv <path/to/my-env-python/> --system-site-packages
4. . <path/to/my-env-python/>/bin/activate
5. pip3.6 install --user --upgrade pip
6. pip3.6 install -r requirements.txt
7. deactivate
```

Then, every time you connect with Arcadia, before your run your Python code, you have to activate the venv with the command:

```
. <path/to/my-env-python/>/bin/activate
```

E.2 Run MLP

For the MLP implementations, no special arguments are required. To modify the hyperparameters such as the alpha (α), learning rate, network configuration, CG iterations, etc. you have to do it in the code.

E.3 Run Adam with CNN

The following code executes the training with the Adam optimzier, for a 3D input, with dimensions $70 \times 60 \times 60$, a learning rate = 0.01, a regularization term (weight decay) C = 0.01, a batch size equal to 256, for 500 epochs. The training and validation set have to be given in a .mat file format.

```
python train.py --optim Adam --lr 0.01 --C 0.01 --net CNN_4layers --bsize 256 --train_set <path to train set ‘.mat’> --val_set  <path to valid set ‘.mat’> --dim 70 60 60 1 --epoch_max 500
```

To run your training in background and store the results in a file you can execute the following command:

```
nohup python train.py --optim Adam --lr 0.01 --C 0.01 --net CNN_4layers --bsize 256 --train_set <path to train set ‘.mat’> --val_set <path to valid set ‘.mat’> --dim 70 60 60 1 --epoch_max 500 &> <path to output folder ’.out’> &
```

E.4 Run NewtonCG with CNN

Very similarly you can execute a NewtonCG training. Keep in mind that now you have to define the arguments “iter_max” instead of “epoch_max”.

```
python train.py --optim NewtonCG --GNsize 50 --C 0.01 --net CNN_4layers --bsize 32 --train_set <path to train set ‘.mat’> --val_set <path to valid set ‘.mat’> --dim 174 174 1 --iter_max 100
```

E.5 Test/Predict CNN

To predict, you can execute the following command by providing the test set in the same format as the training and validation sets and folder where the based model was saved during training. The algorithm saves the best model by default in the path “/saved_model/model.ckpt”.

```
python predict.py --bsize 256 --test_set <path to test set ‘.mat’> --model <path to ‘/saved_model/model.ckpt’> --dim 174 174 1
```

E.6 Arguments for CNN

E.6.1 General

`--optim`: the optimization method used for training CNN. (NewtonCG, SGD or Adam)

Default: --optim NewtonCG

`--net`: network configuration (CNN_4layers, CNN_7layers, VGG11, VGG13, VGG16, and VGG19)

Default: --net CNN_4layers

`--train_set & --val_set`: provide the address of .mat file for training or validation (optional).

Default: --train_set data/mnist-demo.mat

`--model`: save the model to a file

Default: --model ./saved_model/model.ckpt

`--loss`: which loss function to use: MSELoss or CrossEntropy

Default: --loss MSELoss

`--bsize`: Split data into segments of size bsize so that each segment can fit into memory for evaluating Gv, stochastic gradient, and global gradient. If you encounter Out of Memory (OOM) during training, you may decrease the --bsize parameter.

Default: --bsize 1024

`--log`: saving log to a file

Default: --log ./running_log/logger.log

`--screen_log_only`: if specified, log printed on-screen only but not to the log file

Default: --screen_log_only

`--C`: Regularization term = 1/(2C × num_data) × L2_norm(weight)^2

Default: --C 0.01

`--dim`: input dimension of data. The shape must be: height width num_channels

Default: --dim 32 32 3

`--seed`: specify random seed to make results deterministic and reproducible.

Default: --seed 0

`--epoch_max`: the maximal number of SG epochs.

Default: -epoch_max 500

E.6.2 Newton Method

`--GNsize`: number of samples used in the subsampled Gauss-Newton matrix.

Default: --GNsize 4096

`--iter_max`: the maximal number of Newton iterations.

Default: --iter_max 100

`--xi`: the tolerance in the relative stopping condition for the conjugate gradient (CG) method.

Default: --xi 0.1

`--CGmax`: the maximal number of CG iterations.

Default: --CGmax 250

`--lambda`: the initial lambda for the Levenberg-Marquardt (LM) method.

Default: --lambda 1

`--drop/--boost`: the drop and boost constants for the LM method.

Default: --drop 2/3; --boost 3/2

`--eta`: the parameter for the line search stopping condition.

Default: --eta 0.0001

E.6.3 SGD

`--decay`: learning rate decay over each mini-batch update.

Default: --decay 0

`--momentum`: SGD + momentum

Default: --momentum 0

Appendix F

Check CNN Implementations

F.1 digit-recognizer-2D.py

Code for creating the sets to test the 2D CNN implementations with the MNIST dataset.

```
import csv
import numpy as np
import os
import scipy.io
from sklearn.model_selection import train_test_split


def shuffle_in_unison(a, b):
    n_elem = a.shape[0]
    indeces = np.random.choice(n_elem, size=n_elem, replace=False)
    return a[indeces], b[indeces]


def data_info(output_path, data_y, mode):

    unique_y, counts_y = np.unique(data_y, return_counts=True)
    dict_data_y = dict(zip(unique_y, counts_y))
    perc_y = np.round(100.*(counts_y/sum(counts_y)), 1)
    dict_data_perc_y = dict(zip(unique_y, perc_y))
    total_samples = len(data_y)

    print("Count per category " + mode + " : ", dict_data_y)
    print("Percentage " + mode + " : ", dict_data_perc_y)
    print("Total " + mode + " set size: ", str(total_samples))
    with open(output_path + 'data-info-' + mode + '.txt', 'w') as info_file:
        print("Count per category " + mode +
              " : ", dict_data_y, file=info_file)
        print("Percentage " + mode + " : ", dict_data_perc_y, file=info_file)
        print("Total " + mode + " set size: ",
              str(total_samples), file=info_file)


def read_data(path, output_path):

    data_Z = []
    data_y = []

    with open(path, newline='') as f:
        reader = csv.reader(f)
        count = 0
        for row in reader:
            if count == 0:
                count += 1
                continue

            Z = row[1:]
            y = int(row[0])

            data_Z.append(Z)
            data_y.append([y+1])
            count += 1

        data_Z = np.array(data_Z)
        data_y = np.array(data_y)

        data_Z = data_Z.astype('float32')
```

```
        data_y = data_y.astype('float32')

        print("Before Shuffle:")
        print("Z: ", data_Z)
        print("y: ", data_y)

        data_Z, data_y = shuffle_in_unison(data_Z, data_y)

        print("After Shuffle:")
        print("Z: ", data_Z)
        print("y: ", data_y)

        Z_train, Z_valid, y_train, y_valid = train_test_split(
            data_Z, data_y, test_size=0.25, random_state=0)

        print("Z_train: ", Z_train)
        print("y_train: ", y_train)
        print("Z_valid: ", Z_valid)
        print("y_valid: ", y_valid)

    mdic = {"Z": Z_train, "y": y_train}
    scipy.io.savemat(output_path + 'digits-2D-train.mat', mdic)
    mdic = {"Z": Z_valid, "y": y_valid}
    scipy.io.savemat(output_path + 'digits-2D-valid.mat', mdic)

    data_info(output_path, data_y,  "total")
    data_info(output_path, y_train,  "train")
    data_info(output_path,  y_valid,  "valid")


def main():
    path_train = "../../../testing_datasets/digit-recognizer-2D/train.csv"
    path_test = "../../../testing_datasets/digit-recognizer-2D/test.csv"

    output_path = "../../../testing_datasets/digit-recognizer-2D/mat_files/"
    os.makedirs(output_path)

    read_data(path_train, output_path)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet F.1. The code for testing the 2D CNN implementation with the MNIST dataset.

F.2 digit-recognizer-3D.py

Code for creating the sets to check the 3D CNN implementations with the 3D MNIST dataset.

```
import h5py
import numpy as np  # linear algebra
import pandas as pd  # data processing, CSV file I/O (e.g. pd.read_csv)
from matplotlib.pyplot import cm
import os
import scipy.io
import sys

np.set_printoptions(threshold=sys.maxsize)


def main():

    path_train = "../../../testing_datasets/digit-recognizer-3D/full_dataset_vectors.h5"
    # path_train = "../../../testing_datasets/digit-recognizer-3D/train_point_clouds.h5"
```

```
    # f = h5py.File(path_train, 'r')

    # print(list(f.keys()))

    # print(f['1'])

    # print(list(f['1'].keys()))

    # print(f['1']['img'])
    # print(f['1']['normals'])
    # print(f['1']['points'])

    with h5py.File(path_train, 'r') as dataset:
        x_train = dataset["X_train"][:]
        x_test = dataset["X_test"][:]
        y_train = dataset["y_train"][:]
        y_test = dataset["y_test"][:]

        print("x_train shape: ", x_train.shape)
        print("y_train shape: ", y_train.shape)

        print("x_test shape:  ", x_test.shape)
        print("y_test shape:  ", y_test.shape)

        # trasform to 3d
        xtrain = np.ndarray((x_train.shape[0], 4096, 3))
        xtest = np.ndarray((x_test.shape[0], 4096, 3))

        print("x_train shape: ", x_train.shape)
        print("x_test shape:  ", x_test.shape)

        data_y_train = []
        data_y_test = []

        def add_rgb_dimention(array):
            scaler_map = cm.ScalarMappable(cmap="Oranges")
            array = scaler_map.to_rgba(array)[:, : -1]
            return array

        for i in range(x_train.shape[0]):
            xtrain[i] = add_rgb_dimention(x_train[i])
            data_y_train.append([y_train[i]+1])

        for i in range(x_test.shape[0]):
            xtest[i] = add_rgb_dimention(x_test[i])
            data_y_test.append([y_test[i]+1])

        output_path = "../../../testing_datasets/digit-recognizer-3D/mat_files/"
        os.makedirs(output_path)

        mdic = {"Z": xtrain, "y": data_y_train}
        scipy.io.savemat(output_path + 'digits-2D-train.mat', mdic)
        mdic = {"Z": xtest, "y": data_y_test}
        scipy.io.savemat(output_path + 'digits-2D-valid.mat', mdic)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Code Snippet F.2. The code for testing the 3D CNN implementation with the 3D MNIST dataset.

Appendix G

Experiments Hippocampus Features [AD, NC]

G.1 MLP with Adam

G.1.1 Experiment HF_M_A1: α = 7, lr = 0.3, net = [10, 8, 8, 1]

G.1.1.1 Training

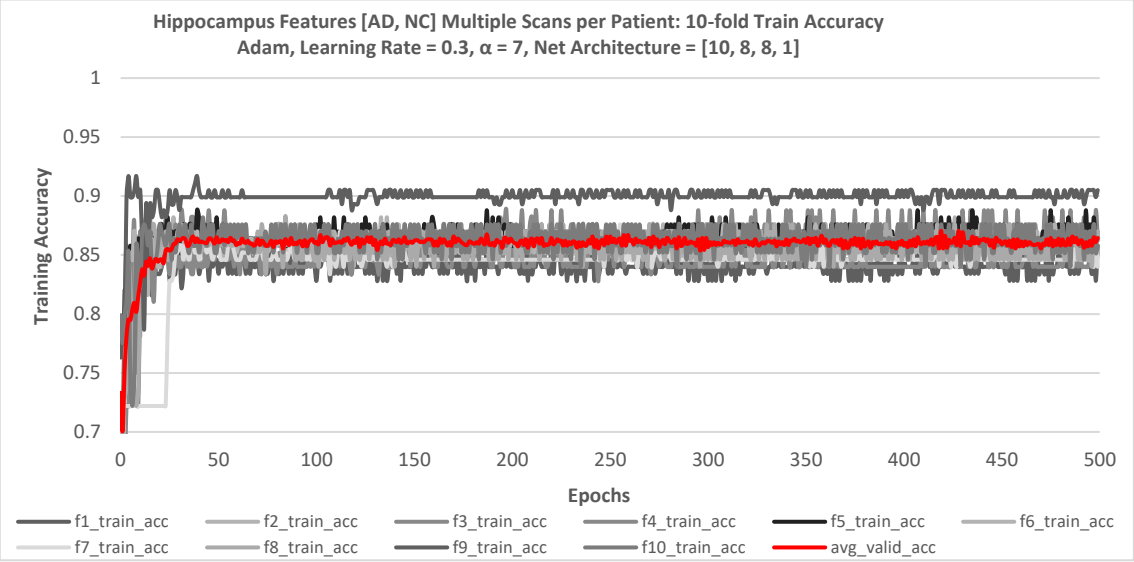


Figure G.1. Caption in the title.

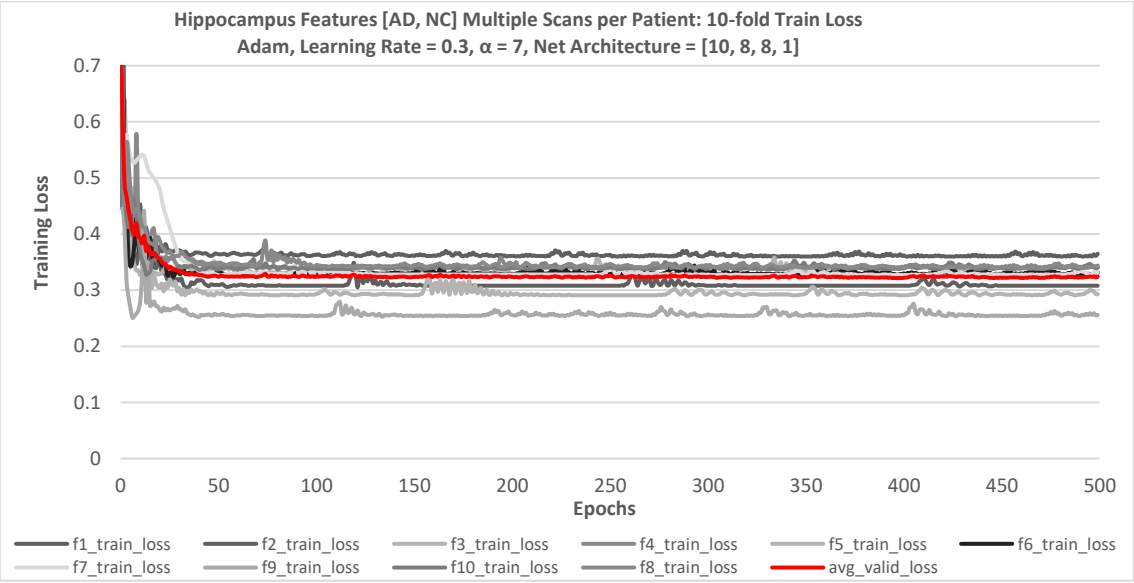


Figure G.2. Caption in the title.

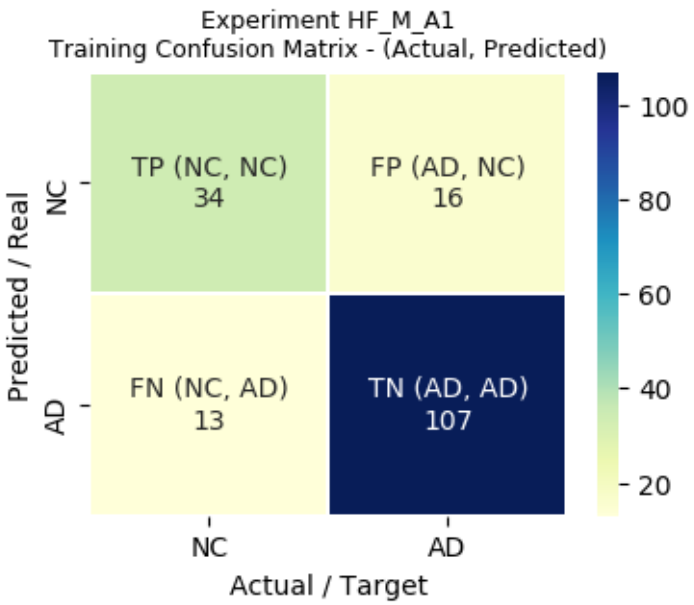


Figure G.3. Caption in the title.

G.1.1.2 Validation

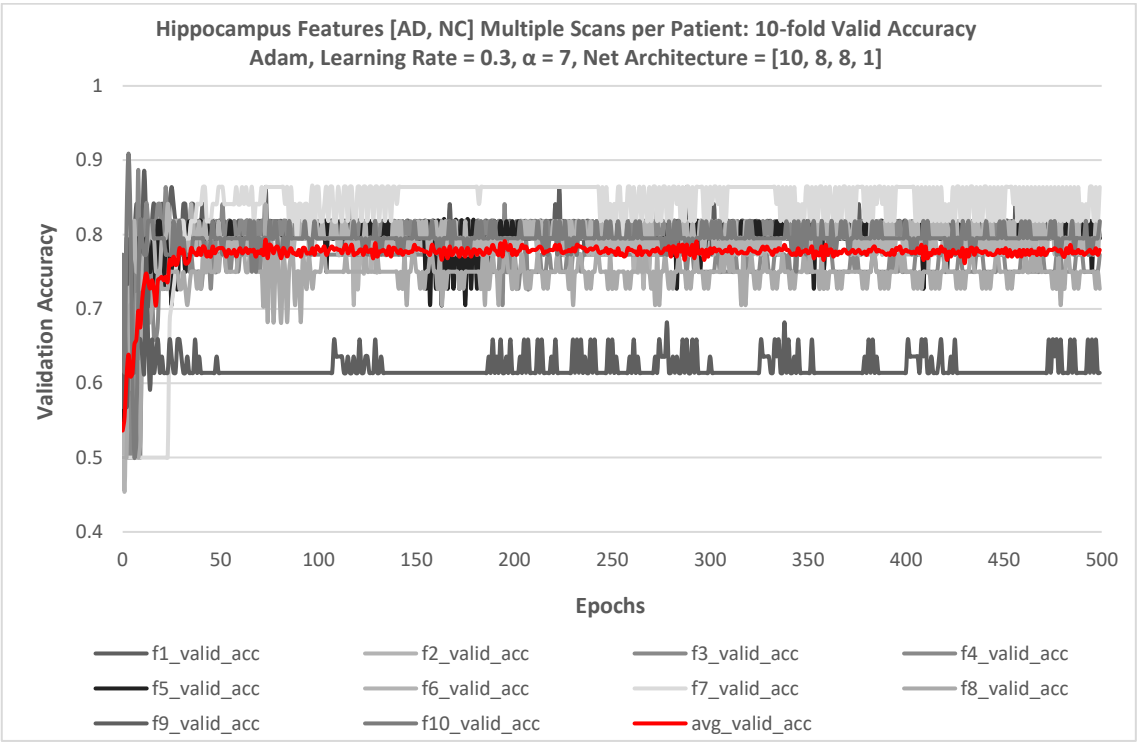


Figure G.4. Caption in the title.

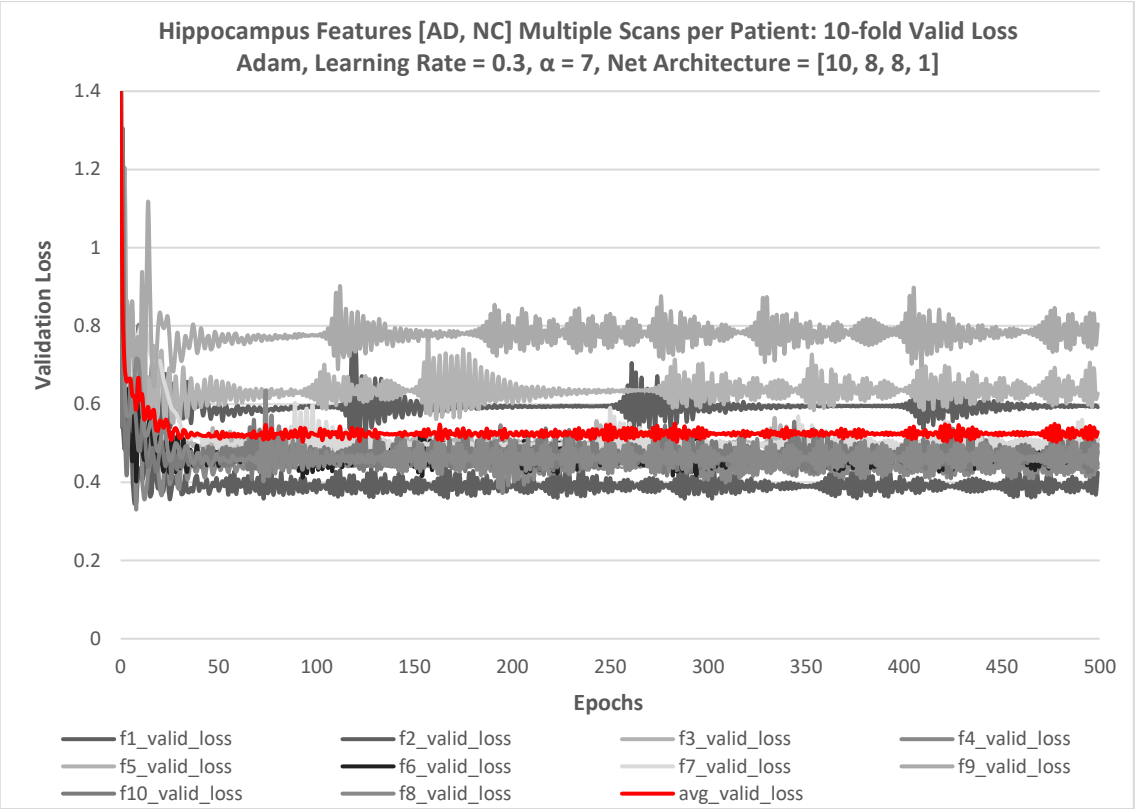


Figure G.5. Caption in the title.

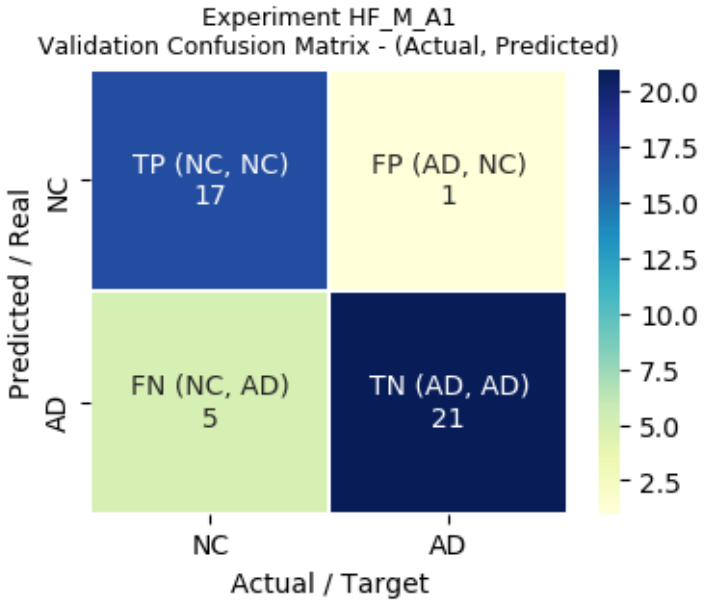


Figure G.6. Caption in the title.

G.1.2 Experiment HF_M_A2: α = 7, lr = 0.03, net = [10, 8, 8, 1]

G.1.2.1 Training

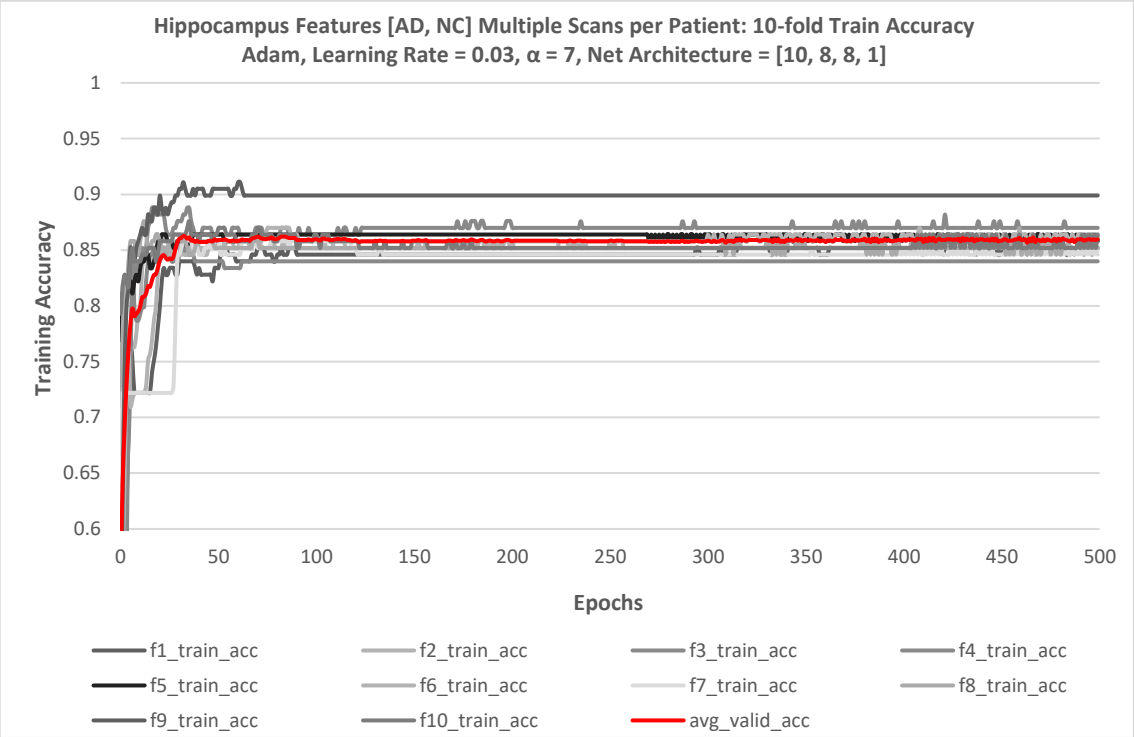
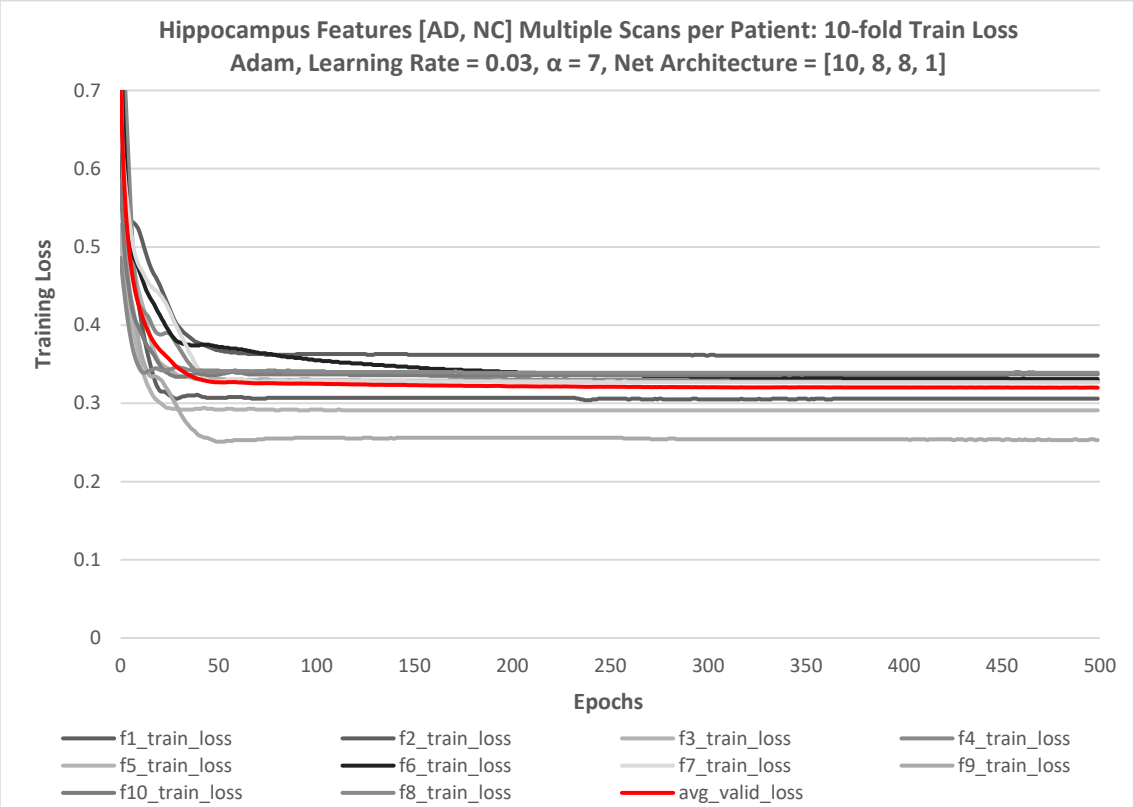


Figure G.7. Caption in the title.



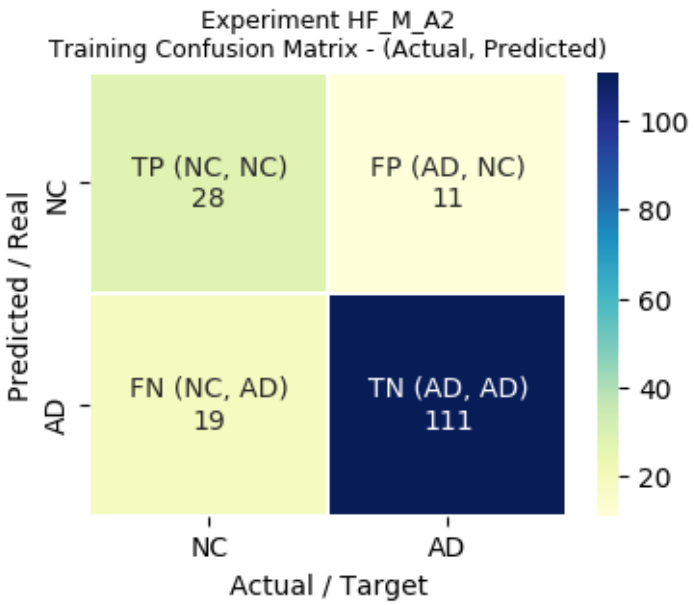


Figure G.8. Caption in the title.

G.1.2.2 Validation

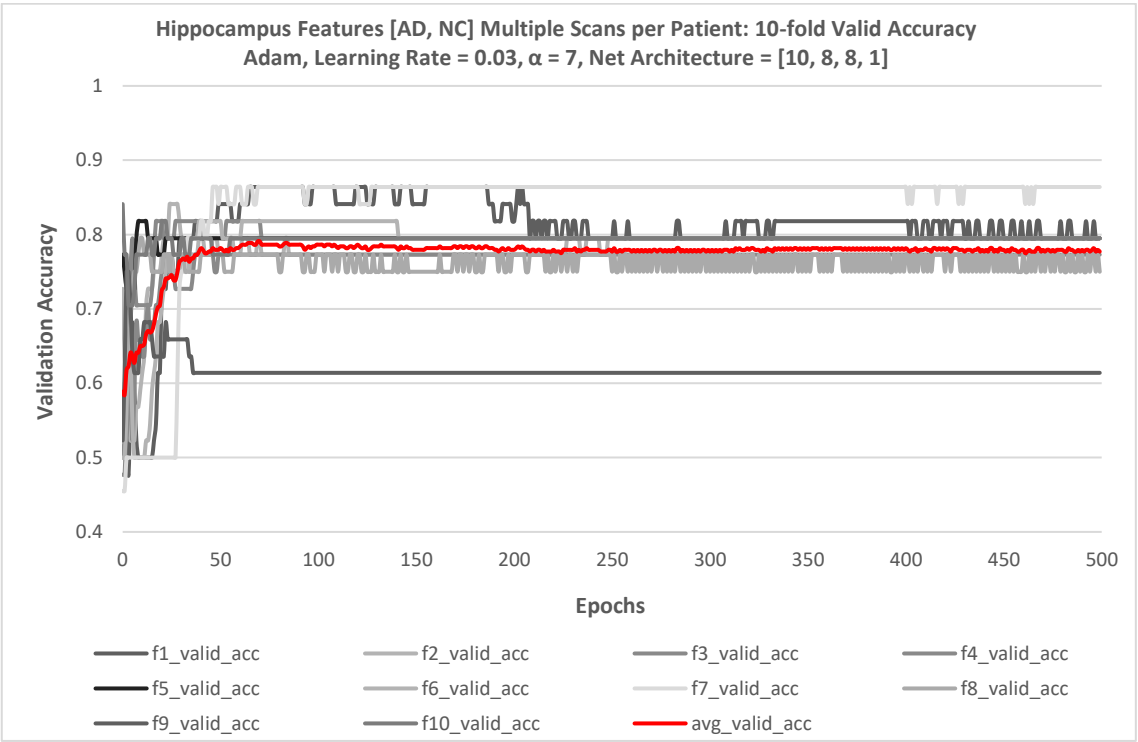


Figure G.9. Caption in the title.

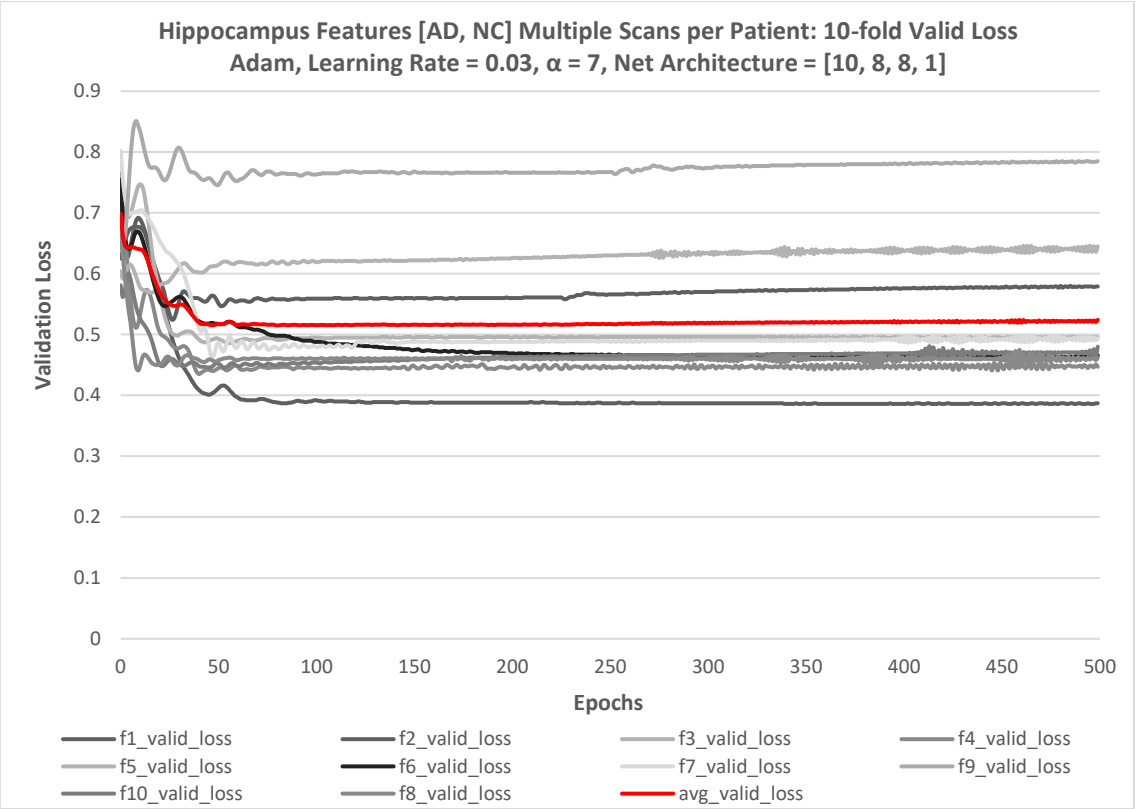


Figure G.10. Caption in the title.

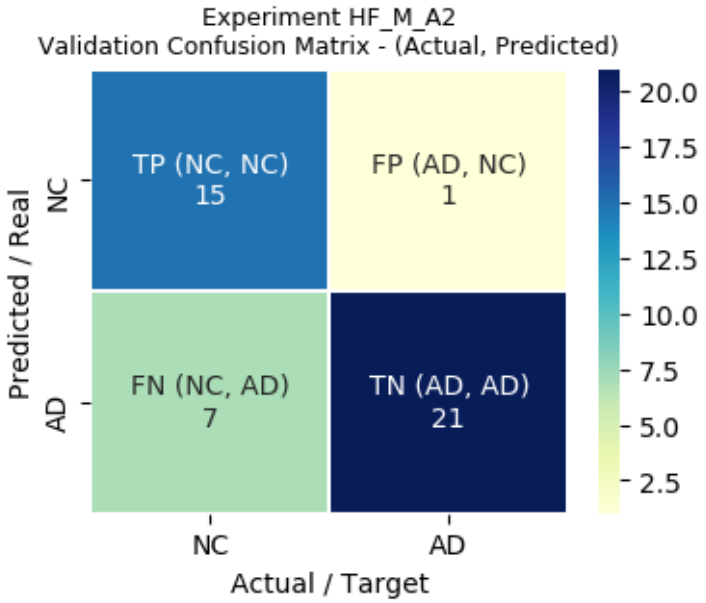


Figure G.11. Caption in the title.

G.1.3 Experiment HF_M_A3: α = 7, lr = 0.003, net = [10, 8, 8, 1]

G.1.3.1 Training

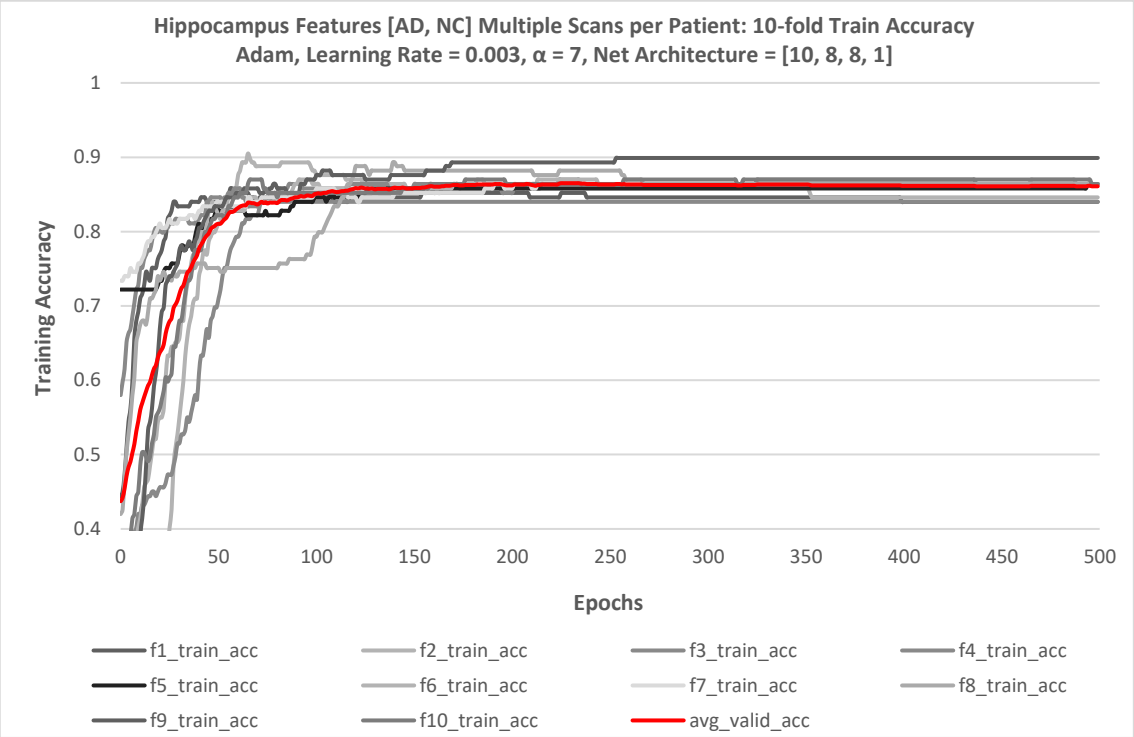


Figure G.12. Caption in the title.

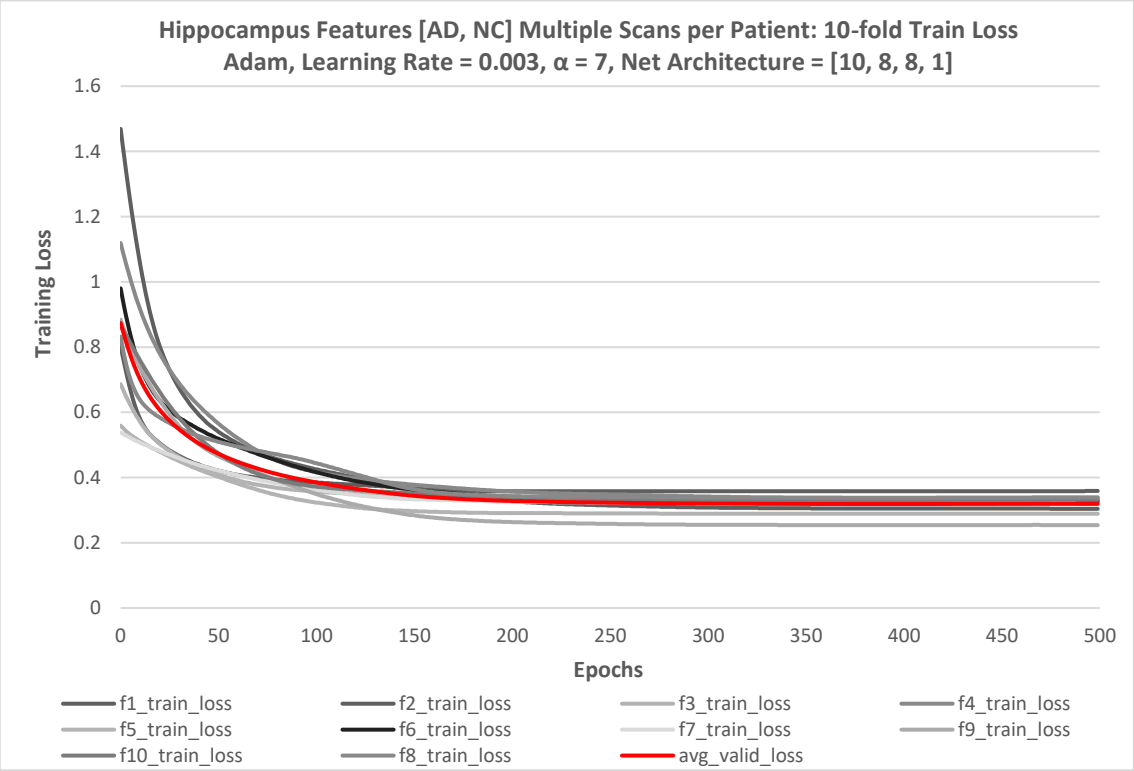


Figure G.13. Caption in the title.

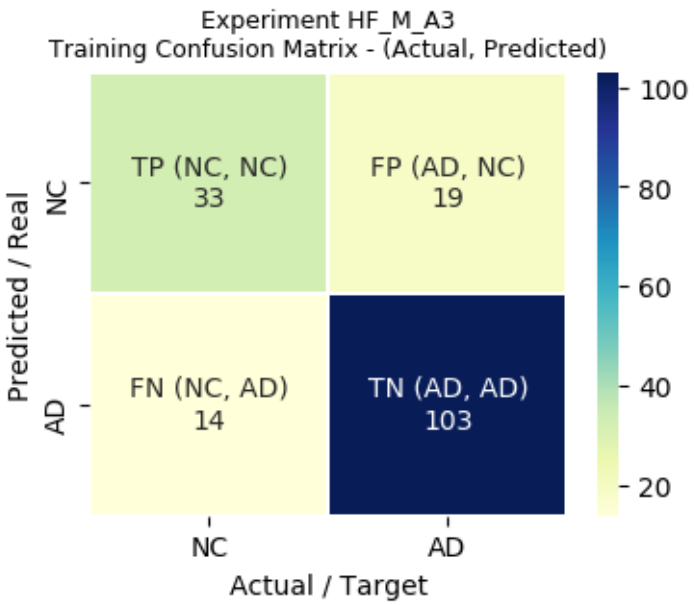


Figure G.14. Caption in the title.

G.1.3.2 Validation

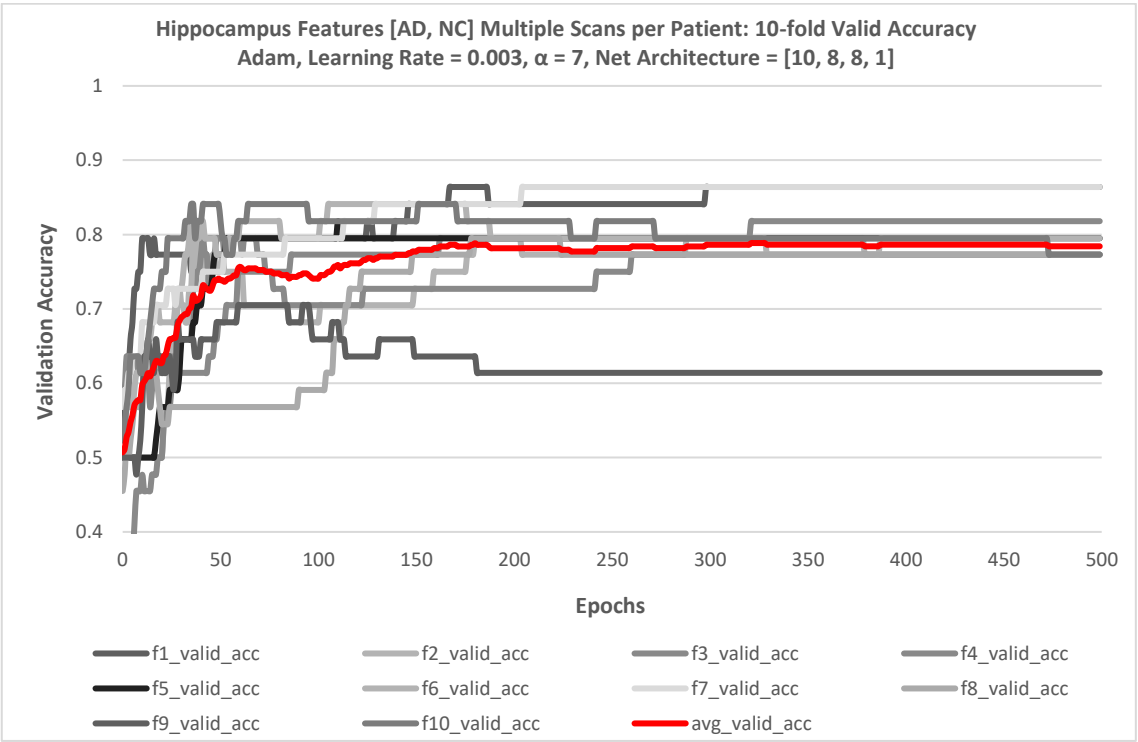


Figure G.15. Caption in the title.

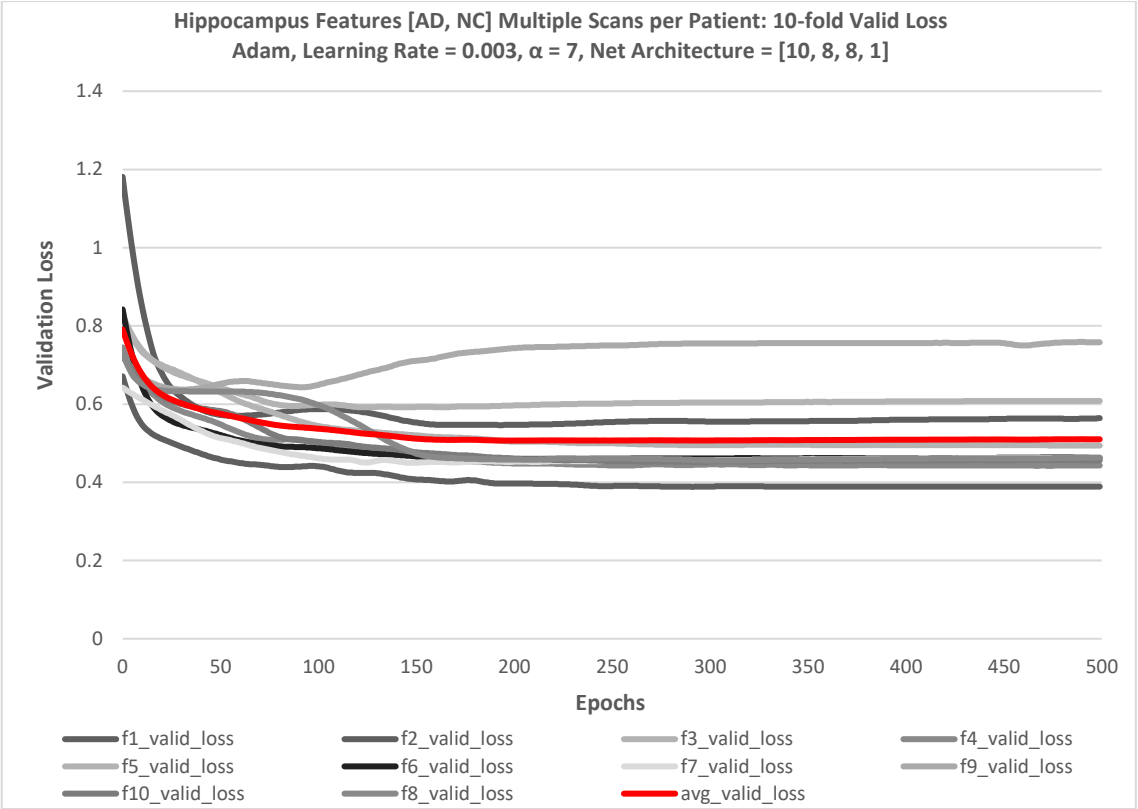


Figure G.16. Caption in the title.

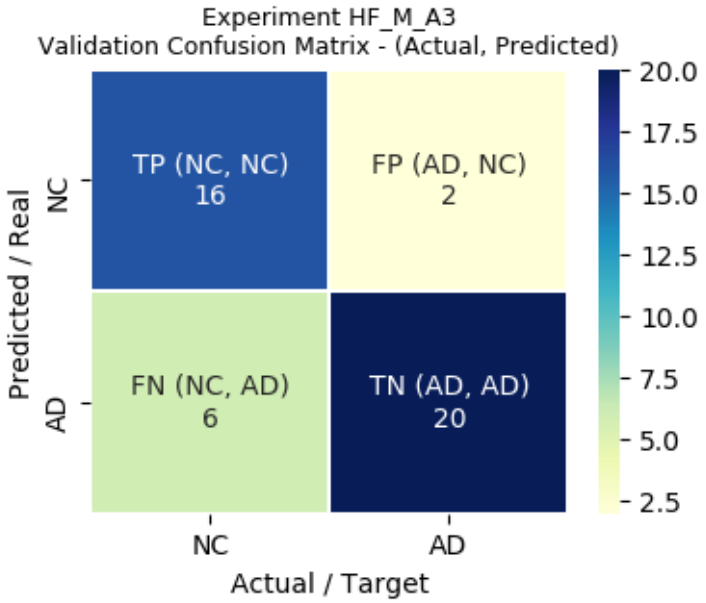


Figure G.17. Caption in the title.

G.1.4 Experiment HF_M_A4: α = 7, lr = 0.0003, net = [10, 8, 8, 1]

G.1.4.1 Training

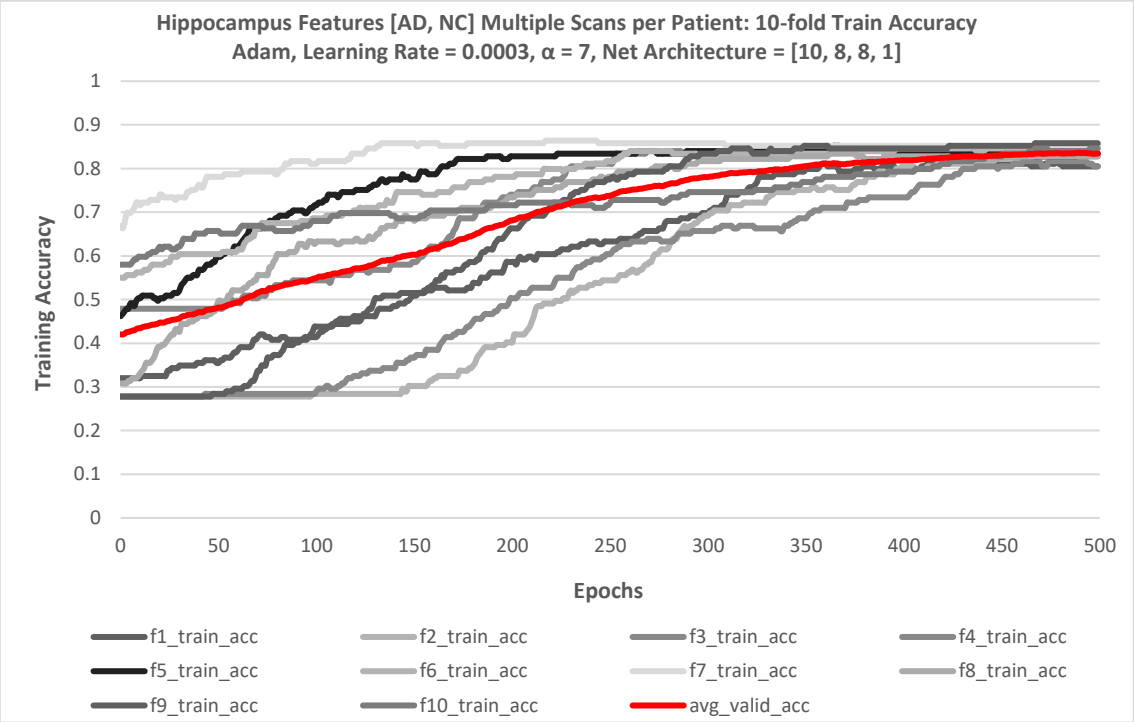


Figure G.18. Caption in the title.

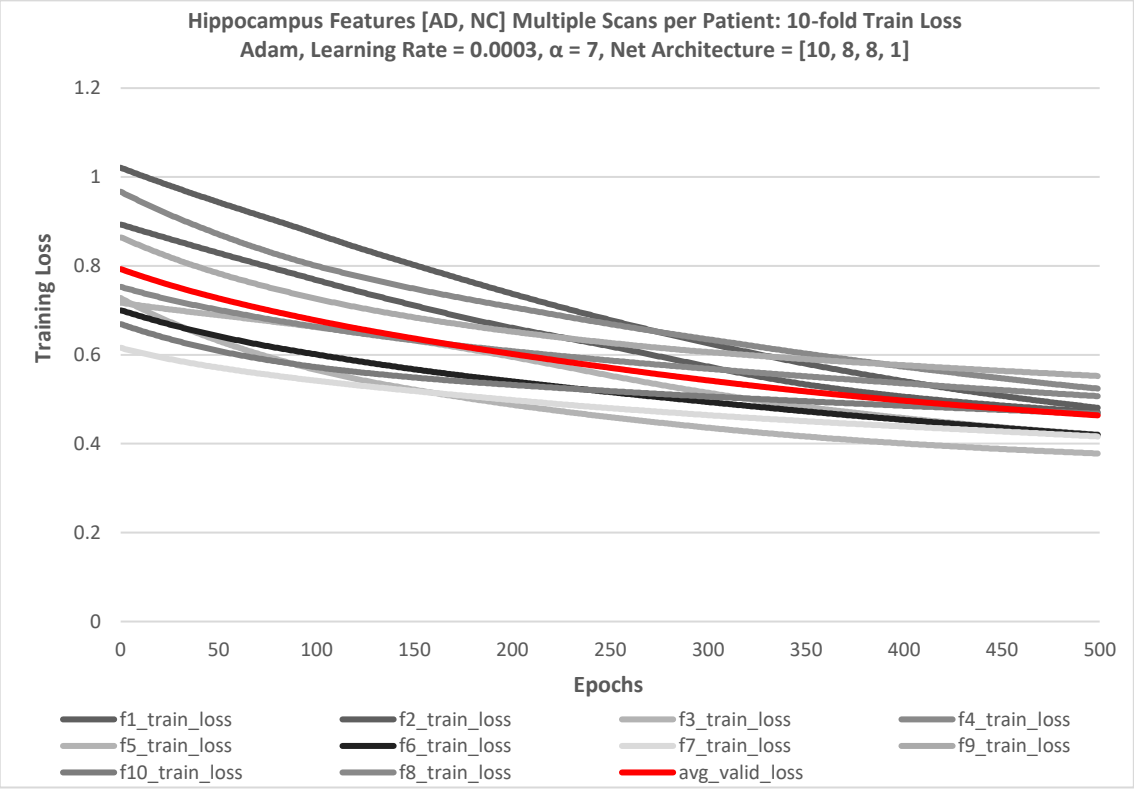


Figure G.19. Caption in the title.

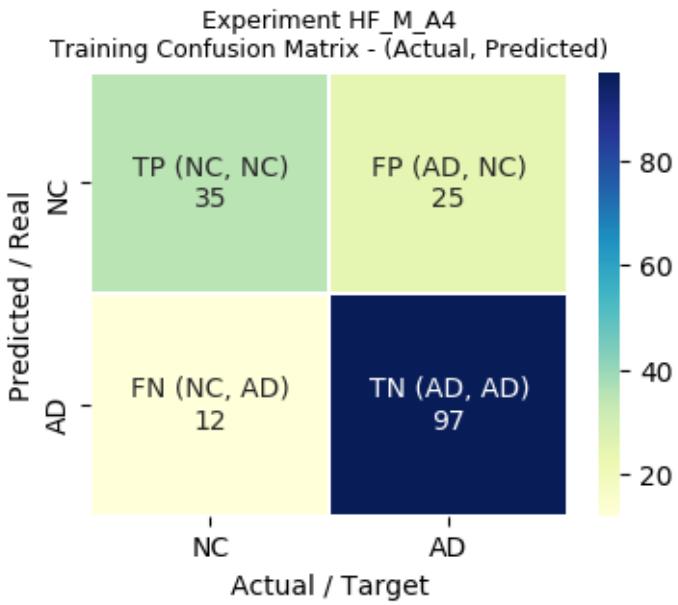


Figure G.20. Caption in the title.

G.1.4.2 Validation

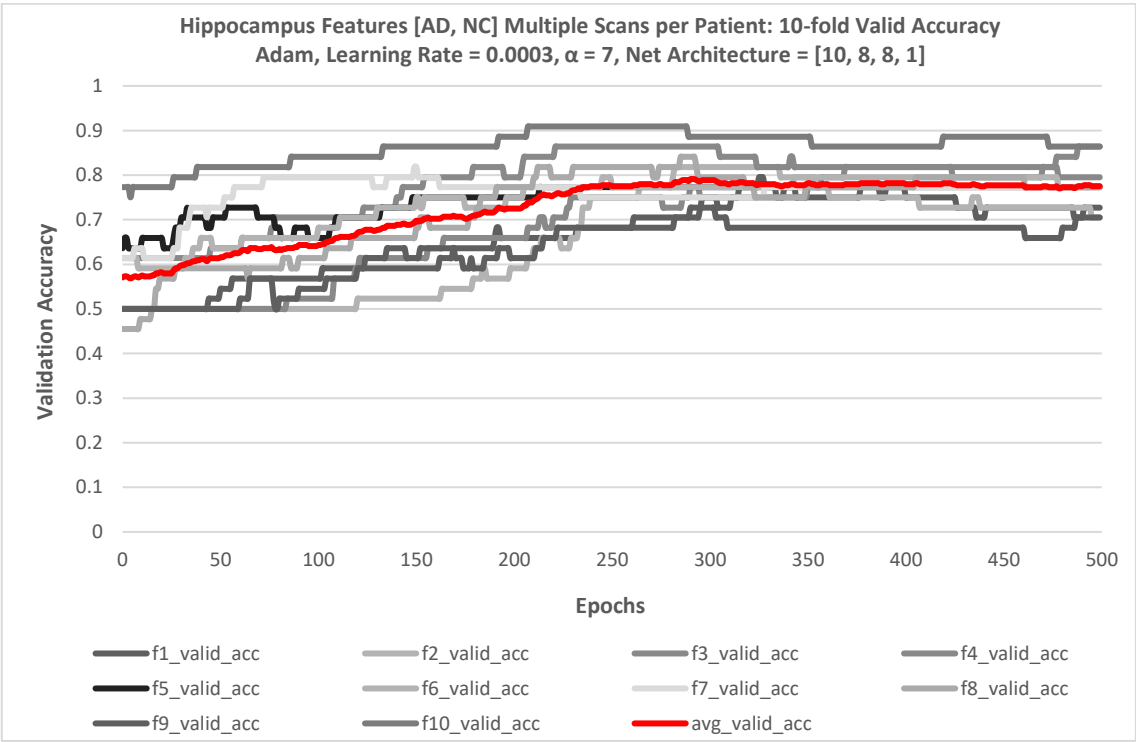


Figure G.21. Caption in the title.

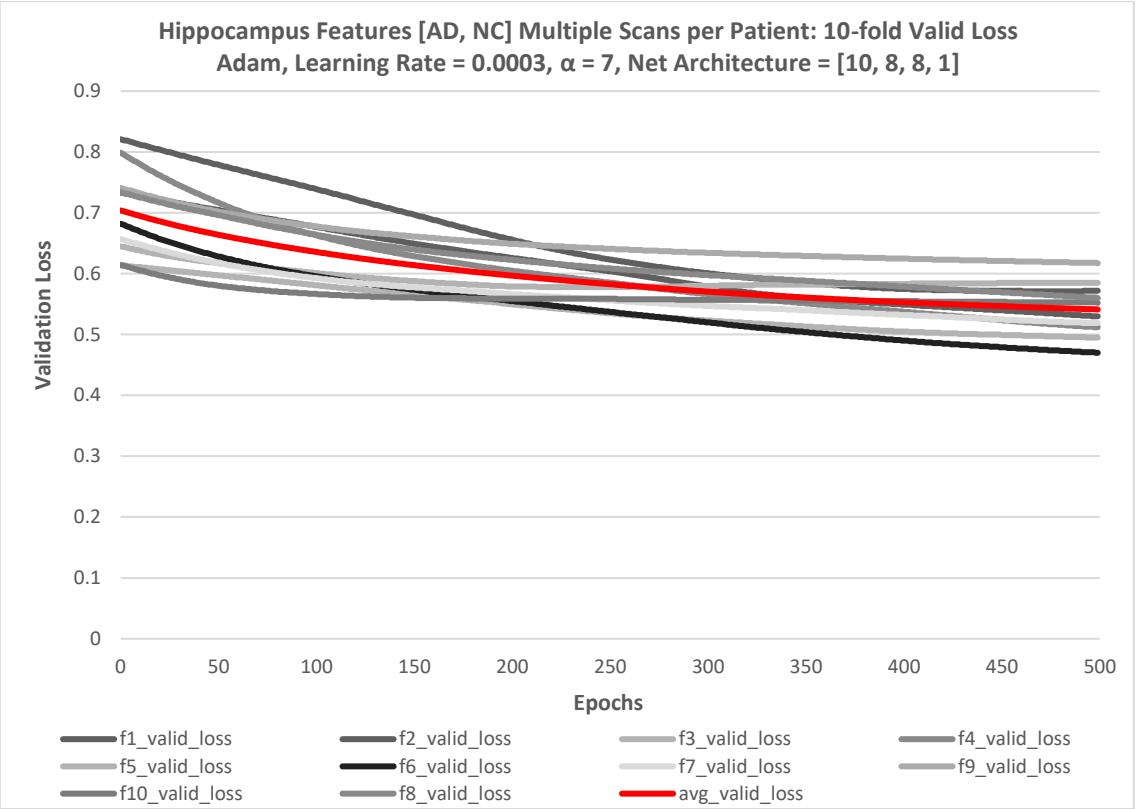


Figure G.22. Caption in the title.

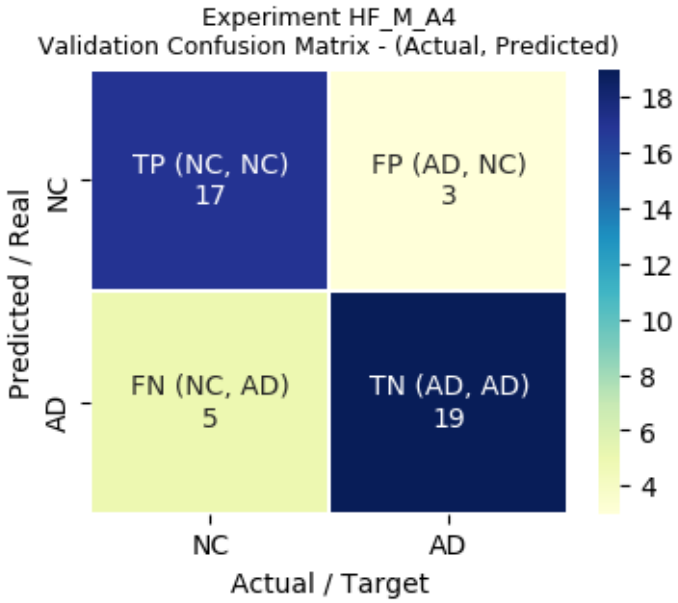


Figure G.23. Caption in the title.

G.1.5 Experiment HF_M_A5: α = 5, lr = 0.3, net = [10, 8, 8, 1]

G.1.5.1 Training

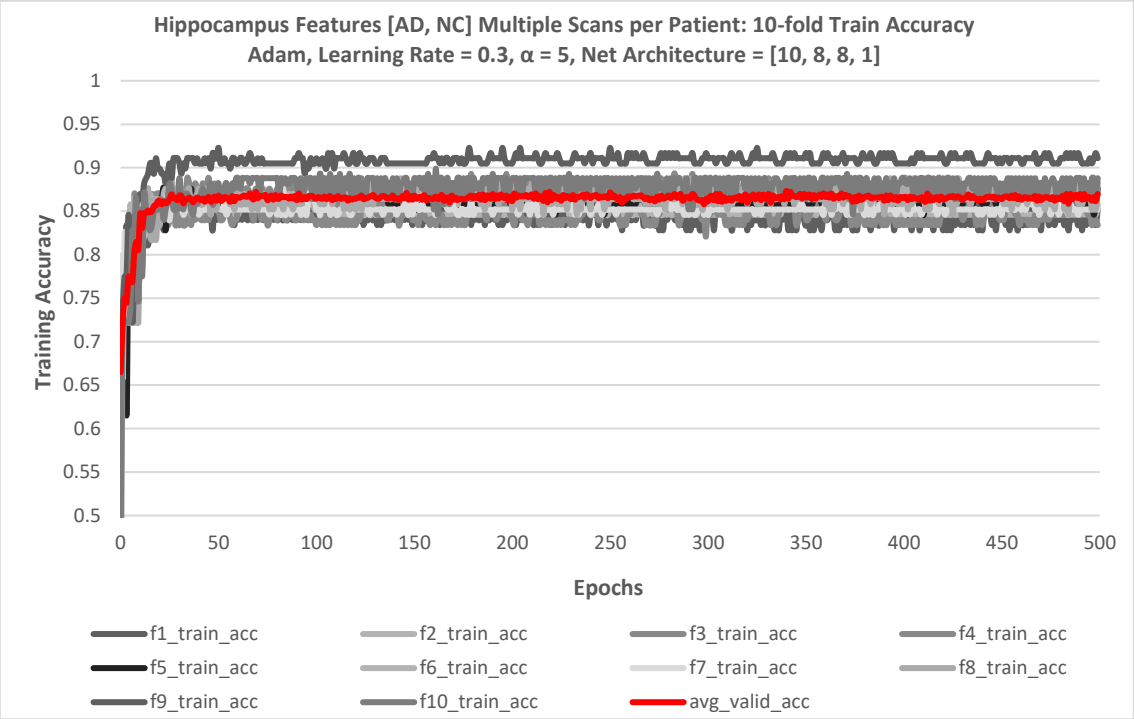


Figure G.24. Caption in the title.

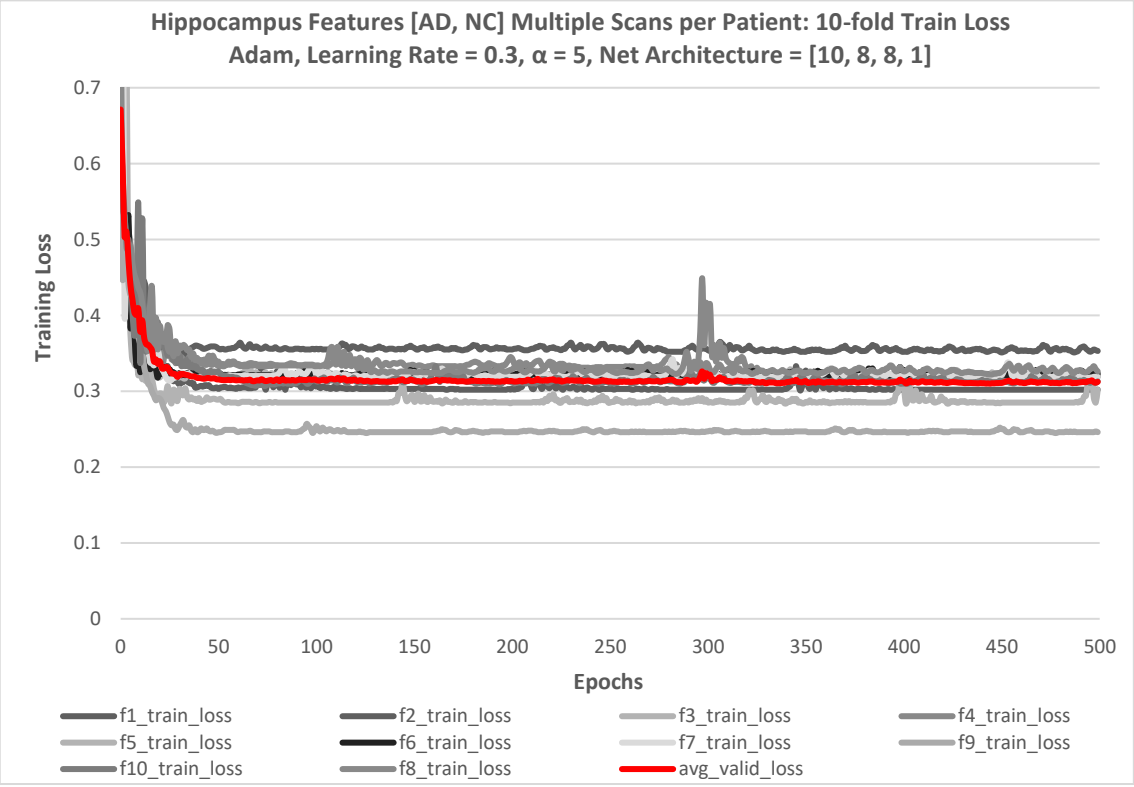


Figure G.25. Caption in the title.

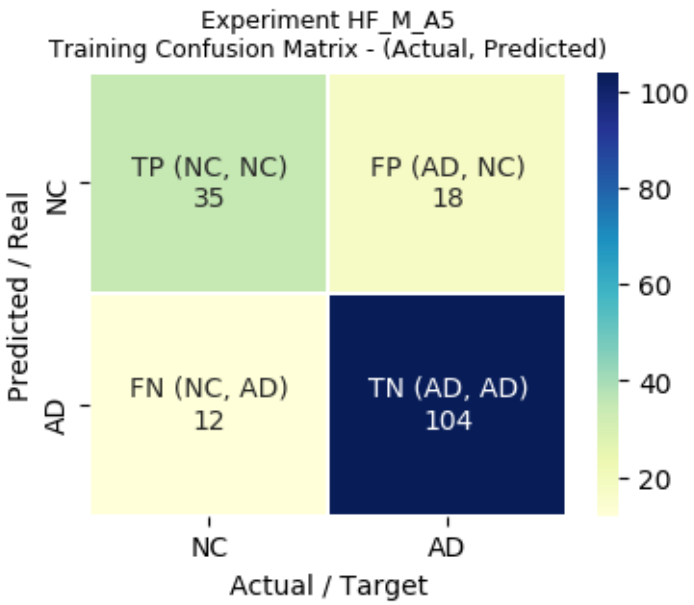


Figure G.26. Caption in the title.

G.1.5.2 Validation

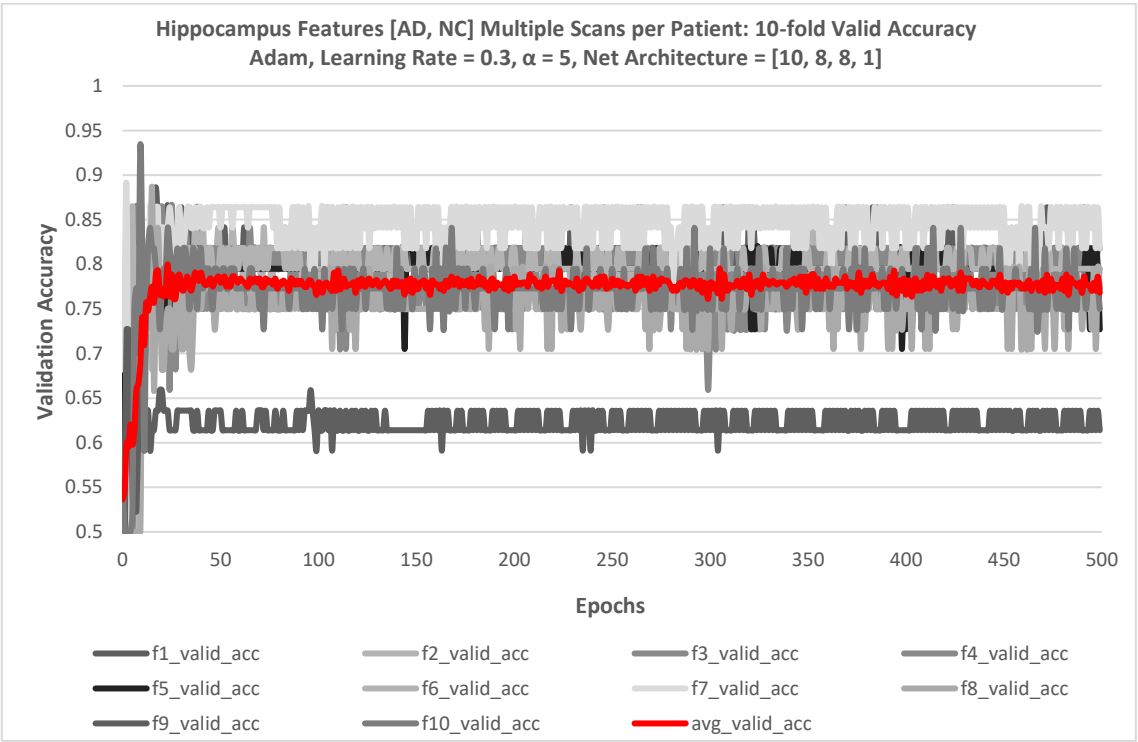


Figure G.27. Caption in the title.

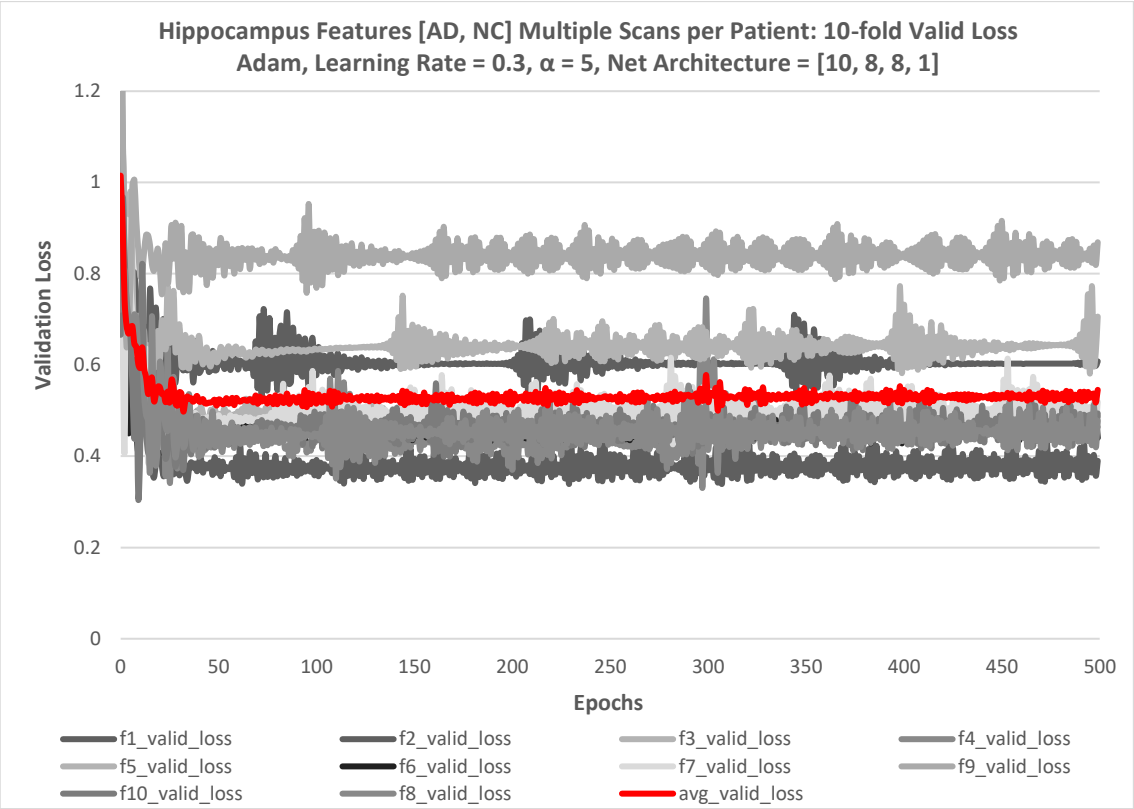


Figure G.28. Caption in the title.

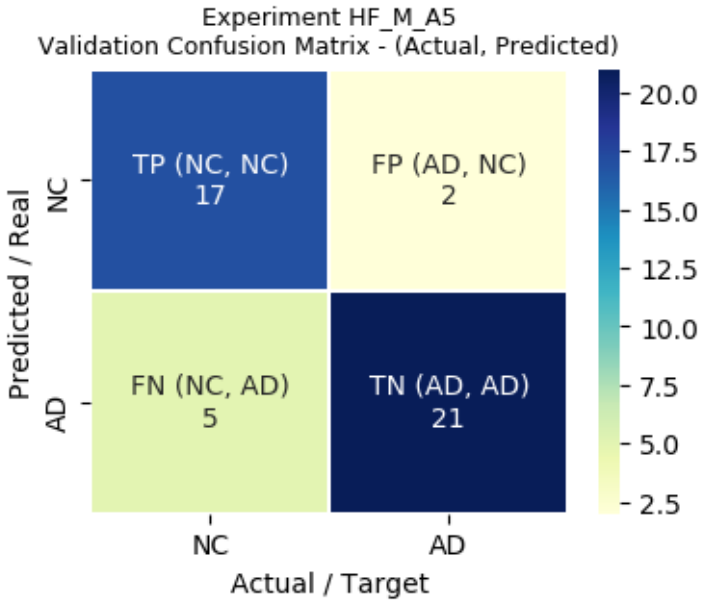


Figure G.29. Caption in the title.

G.1.6 Experiment HF_M_A6: α = 3, lr = 0.3, net = [10, 8, 8, 1]

G.1.6.1 Training

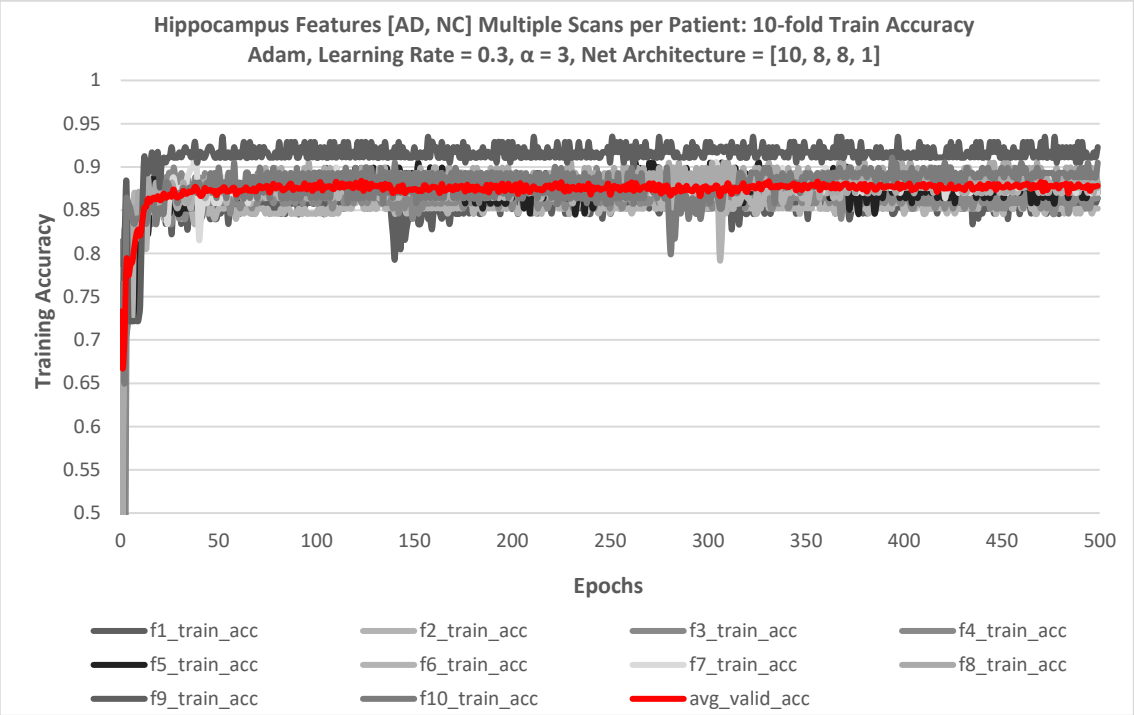


Figure G.30. Caption in the title.

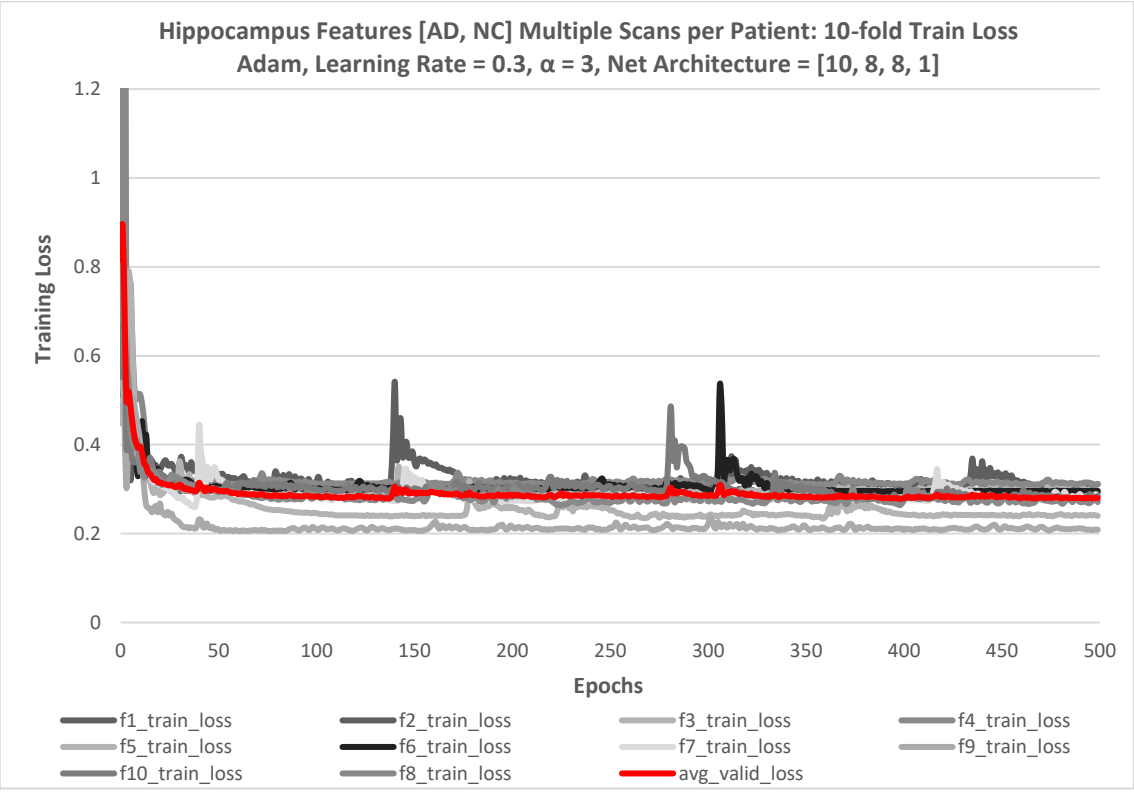


Figure G.31. Caption in the title.

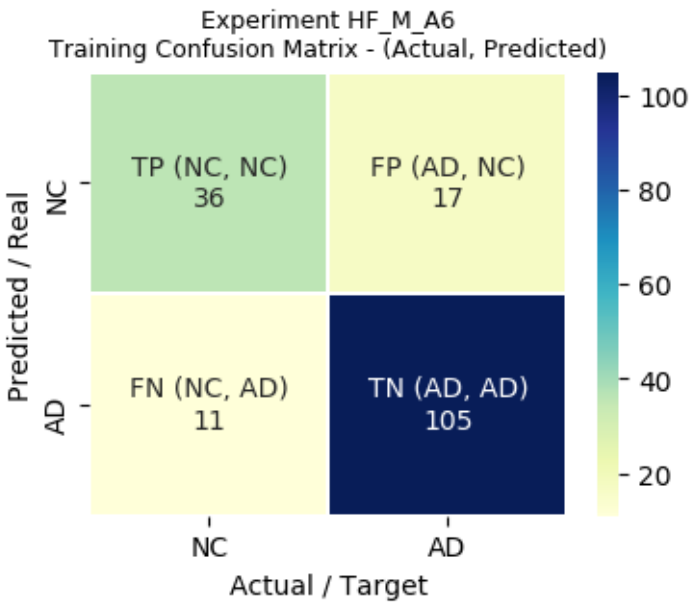


Figure G.32. Caption in the title.

G.1.6.2 Validation

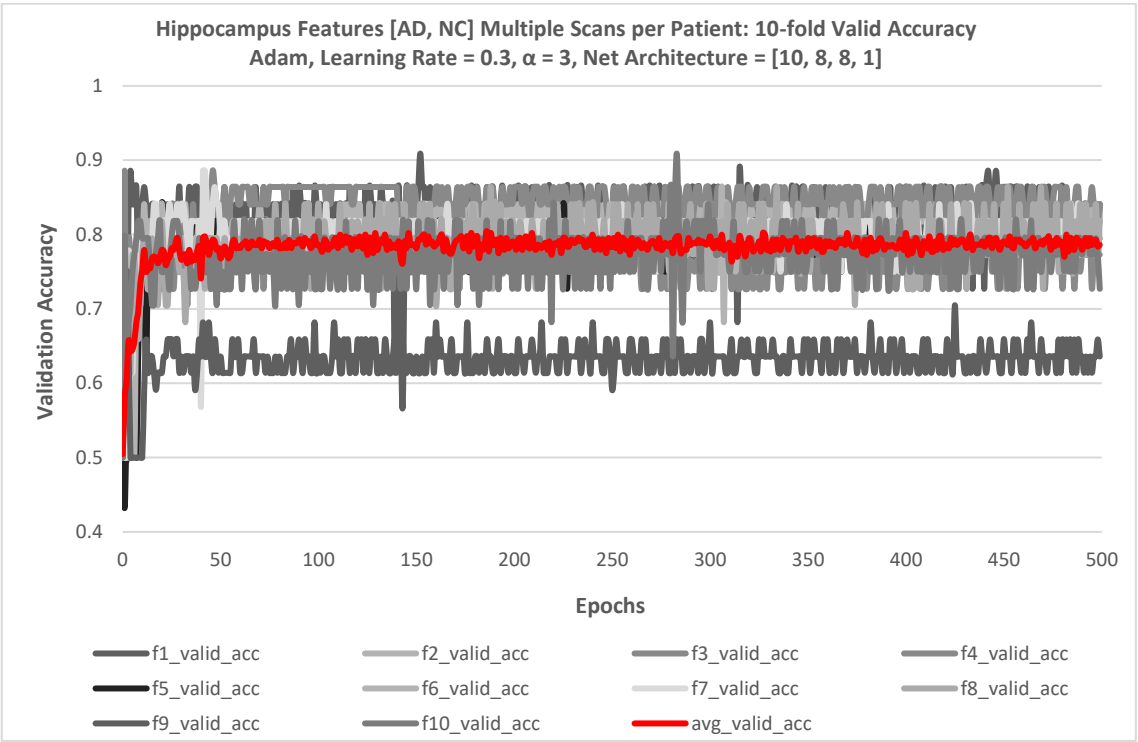


Figure G.33. Caption in the title.

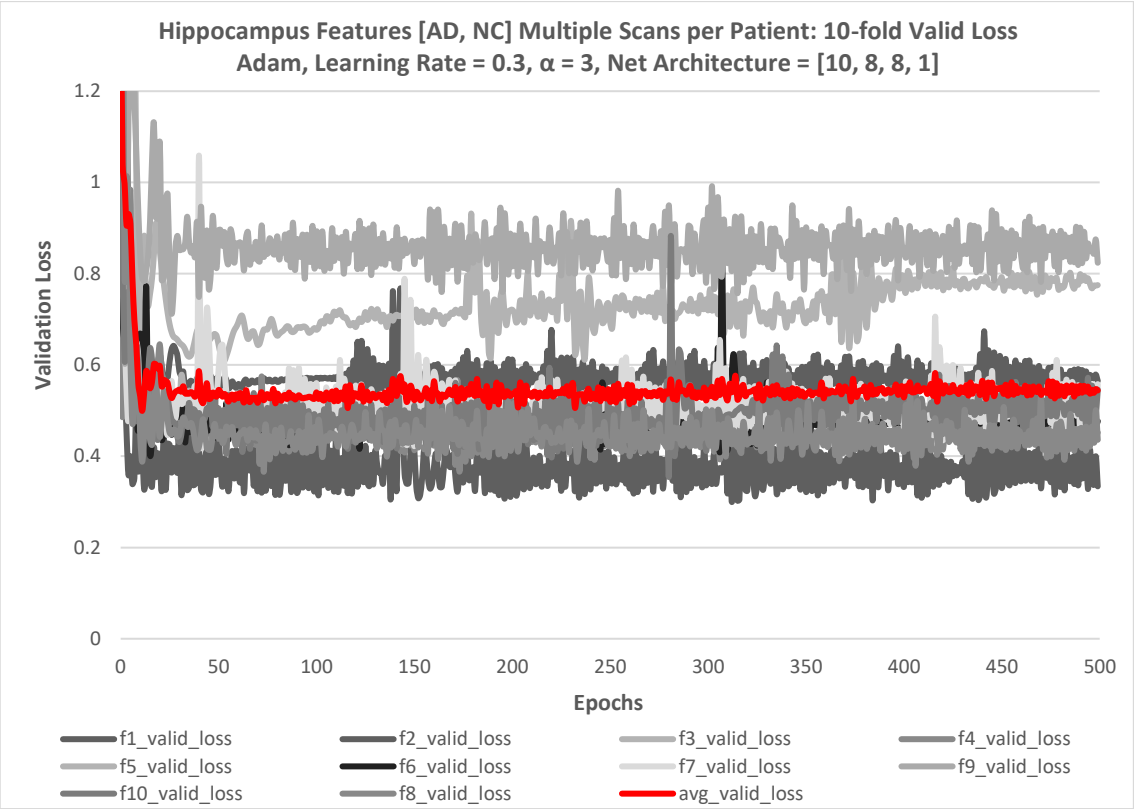


Figure G.34. Caption in the title.

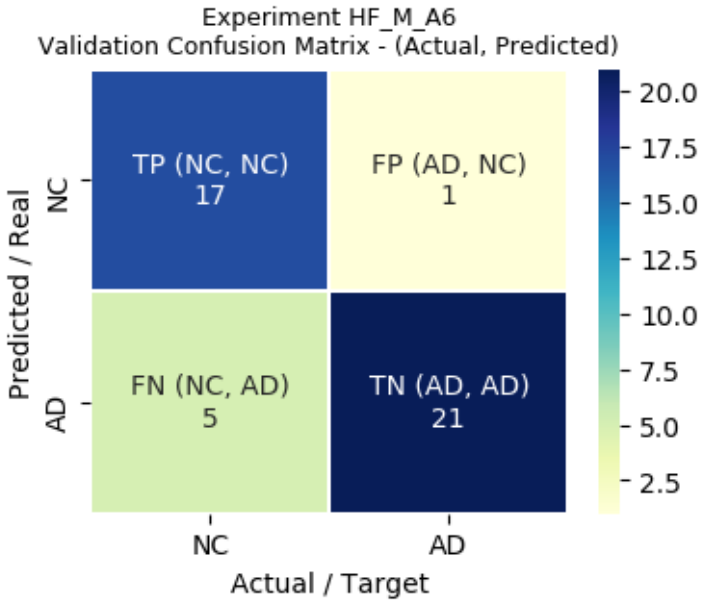


Figure G.35. Caption in the title.

G.1.7 Experiment HF_M_A7: α = 1, lr = 0.3, net = [10, 8, 8, 1]

G.1.7.1 Training

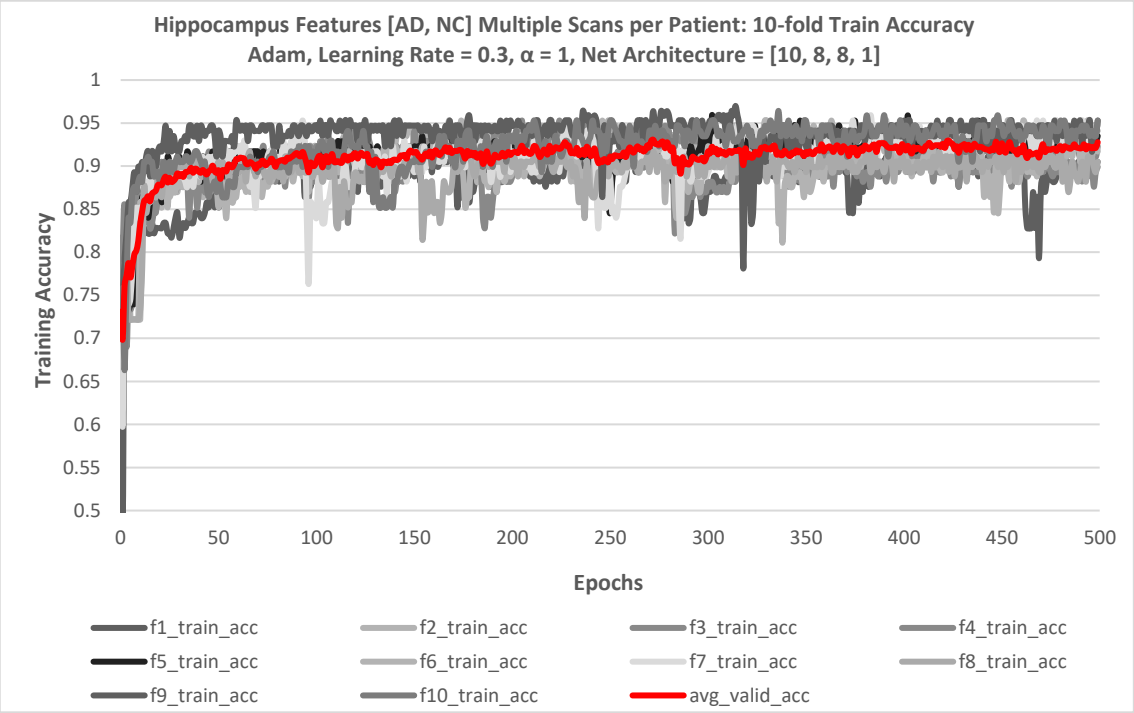


Figure G.36. Caption in the title.

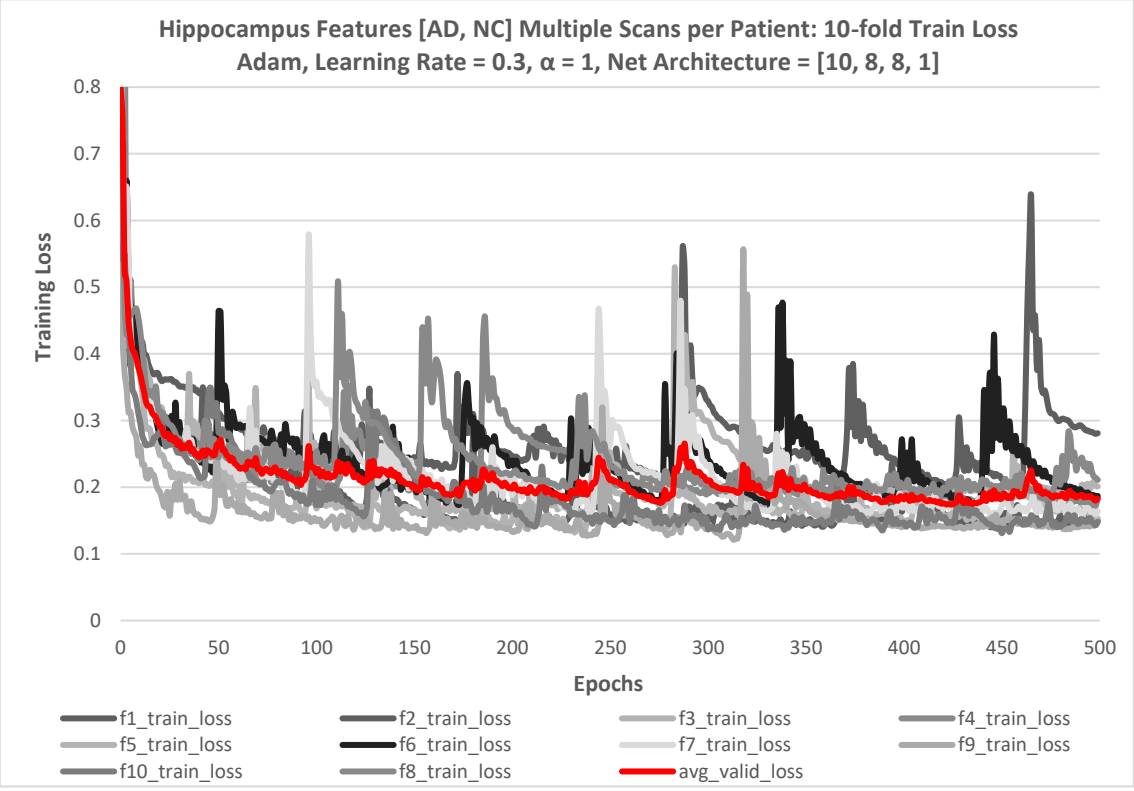


Figure G.37. Caption in the title.

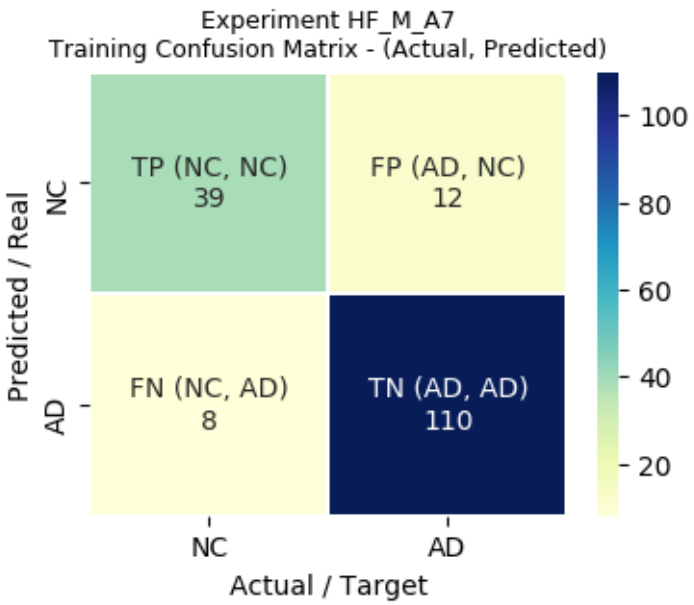


Figure G.38. Caption in the title.

G.1.7.2 Validation

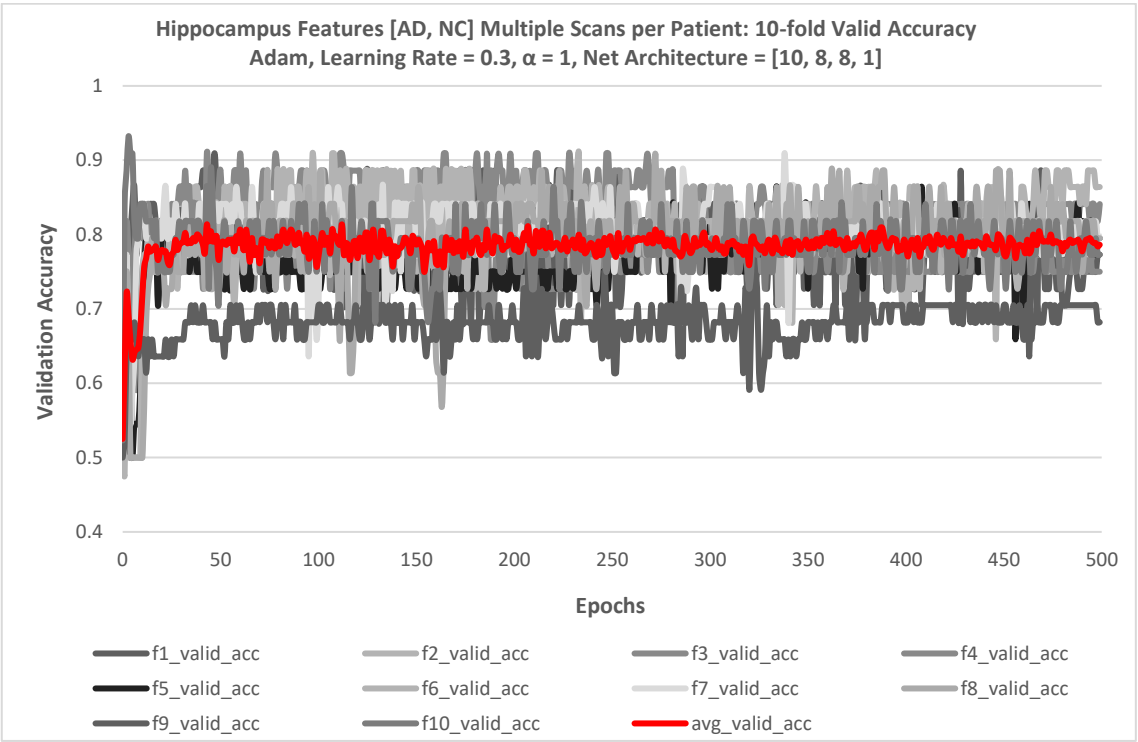


Figure G.39. Caption in the title.

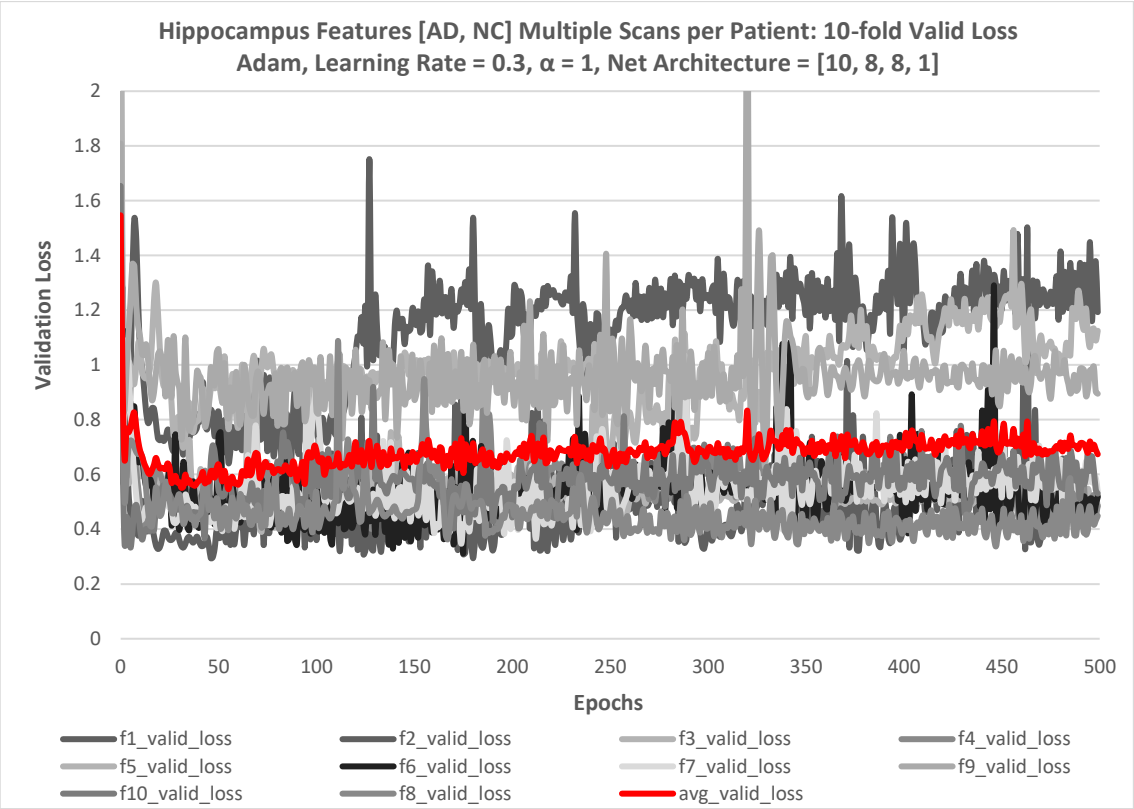


Figure G.40. Caption in the title.

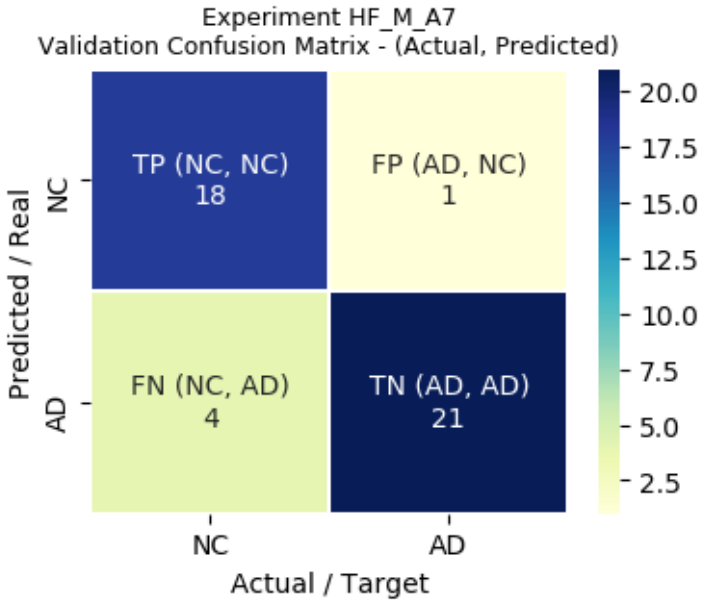


Figure G.41. Caption in the title.

G.1.8 Experiment HF_M_A8: α = 0.1, lr = 0.3, net = [10, 8, 8, 1]

G.1.8.1 Training

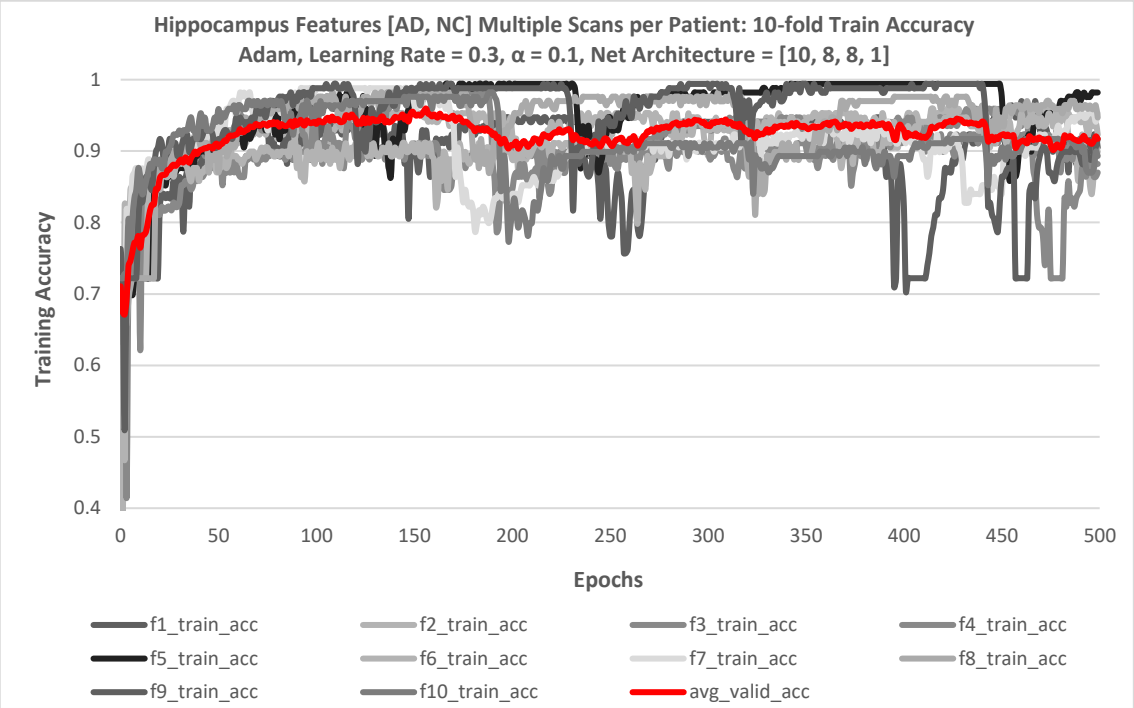


Figure G.42. Caption in the title.

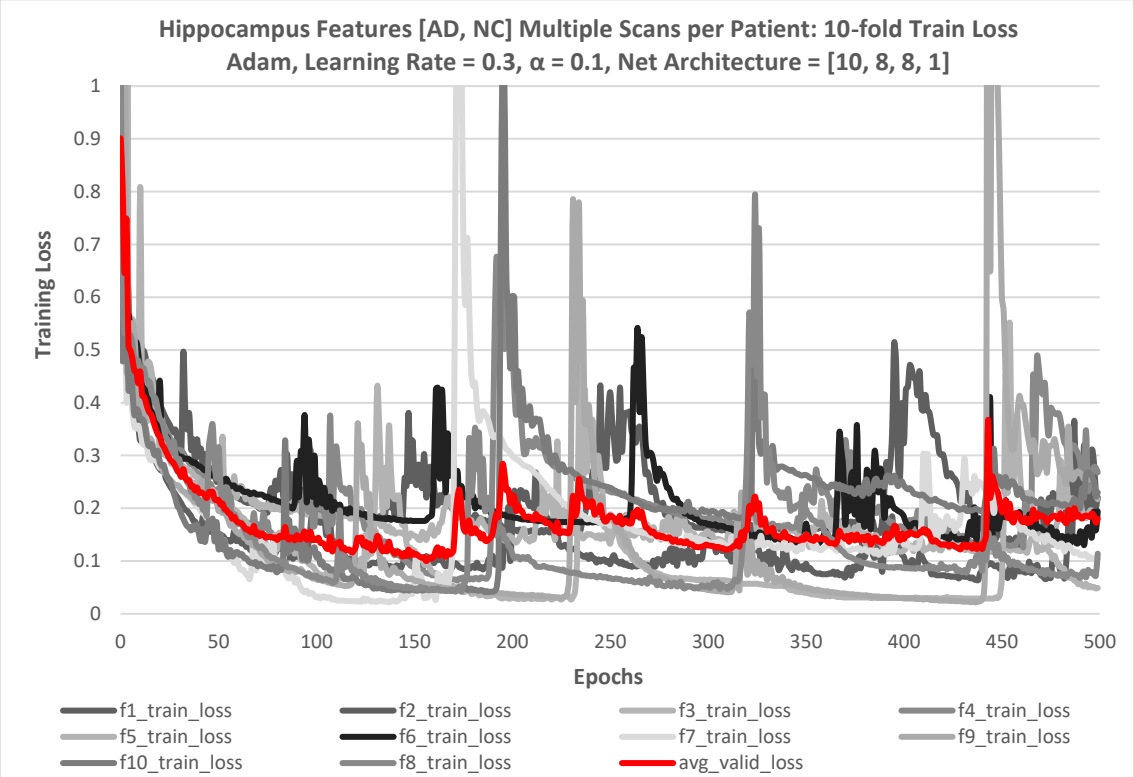


Figure G.43. Caption in the title.

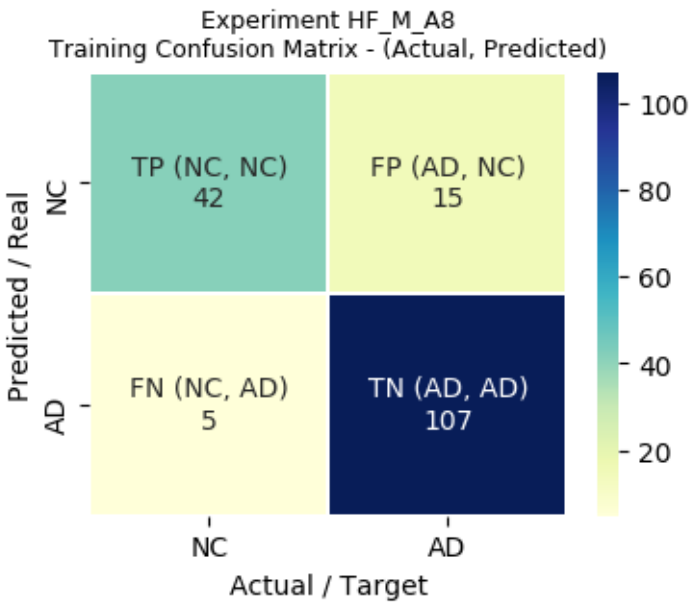


Figure G.44. Caption in the title.

G.1.8.2 Validation

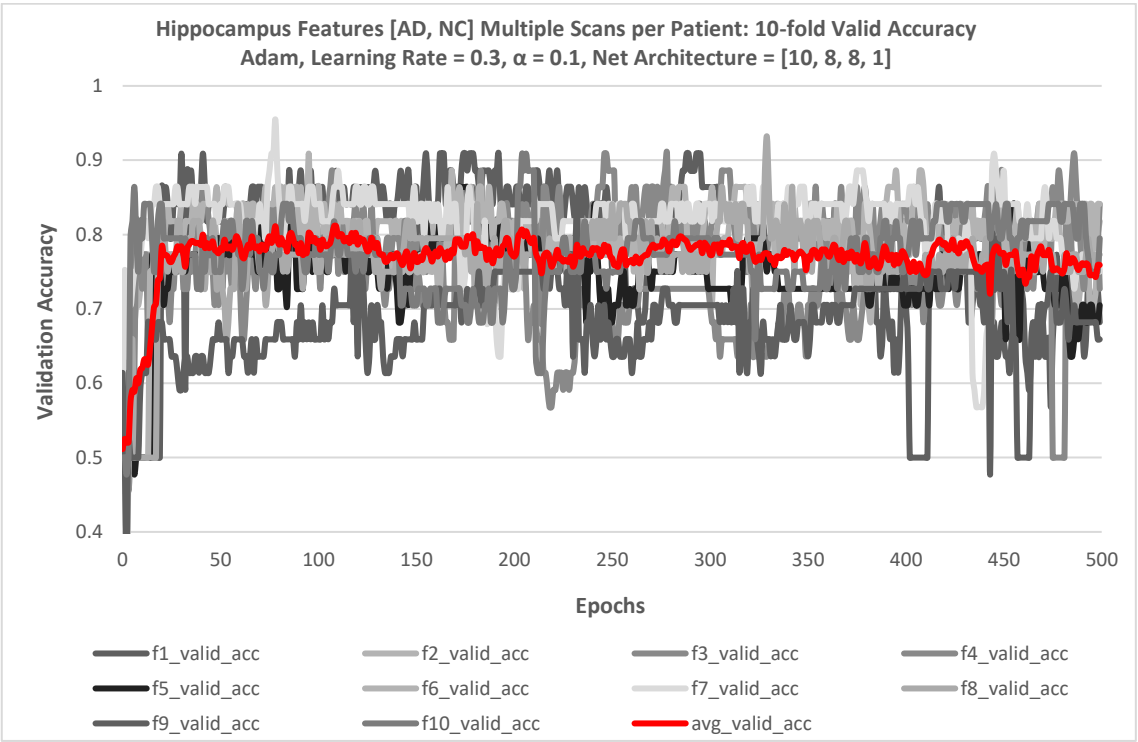


Figure G.45. Caption in the title.

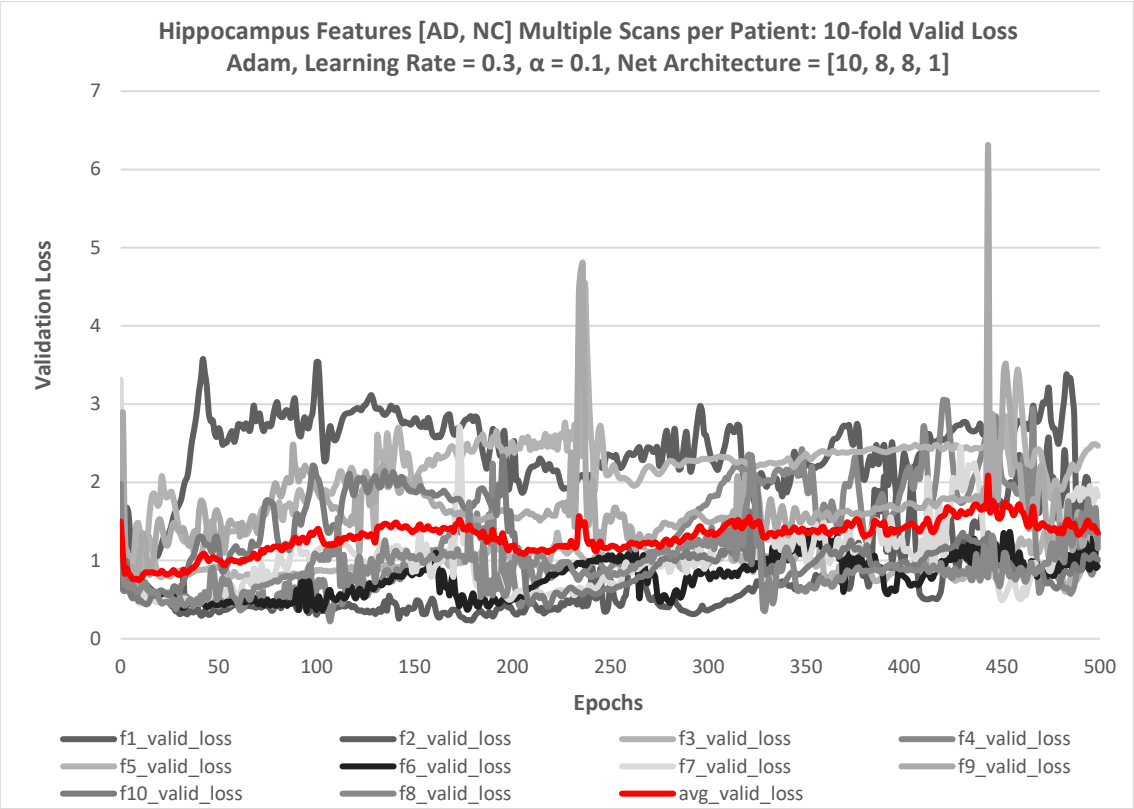


Figure G.46. Caption in the title.

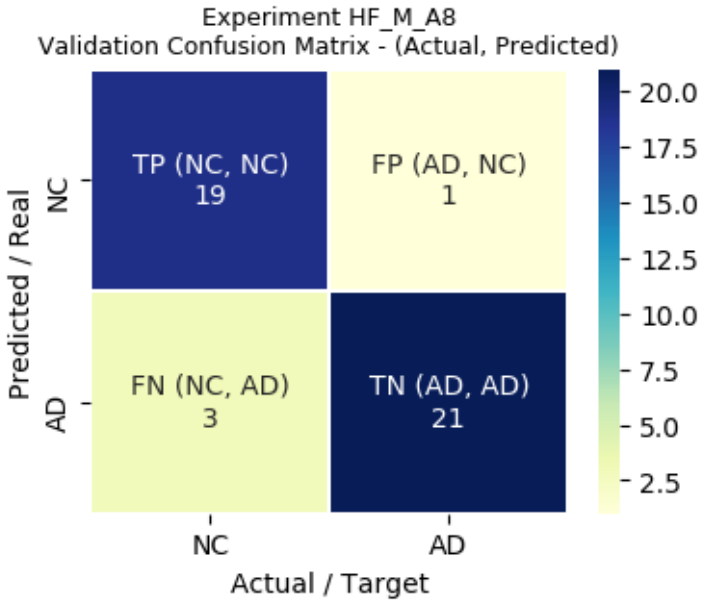


Figure G.47. Caption in the title.

G.1.9 Experiment HF_M_A9: α = 0.01, lr = 0.3, net = [10, 8, 8, 1]

G.1.9.1 Training

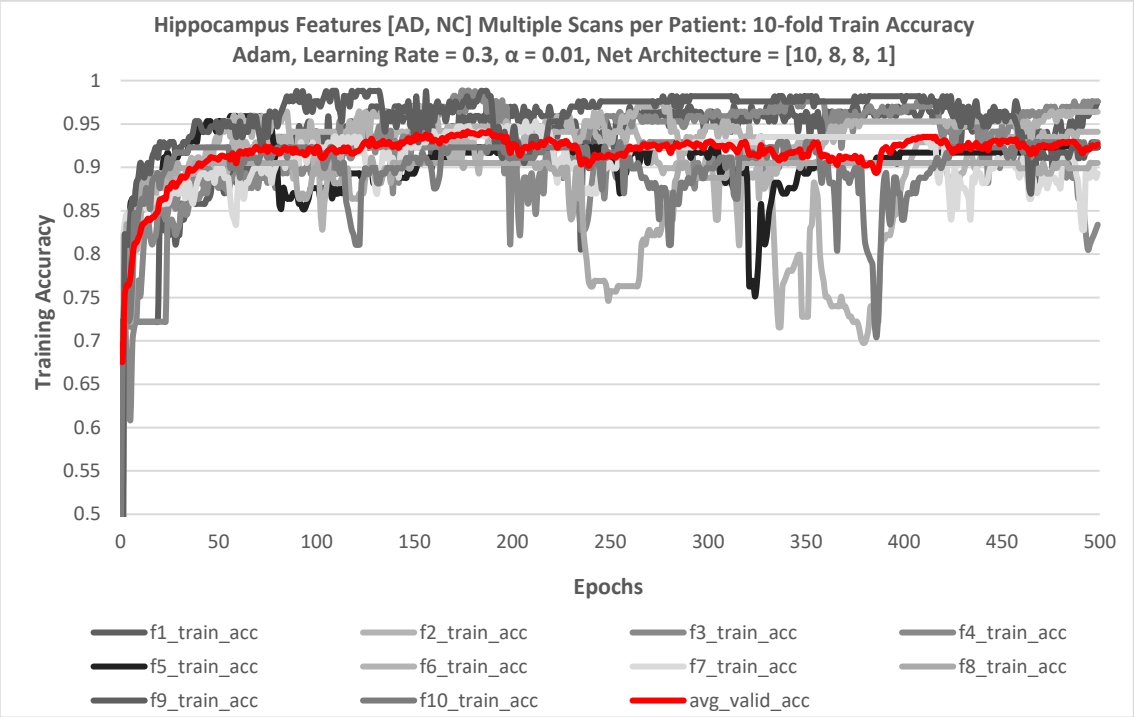


Figure G.48. Caption in the title.

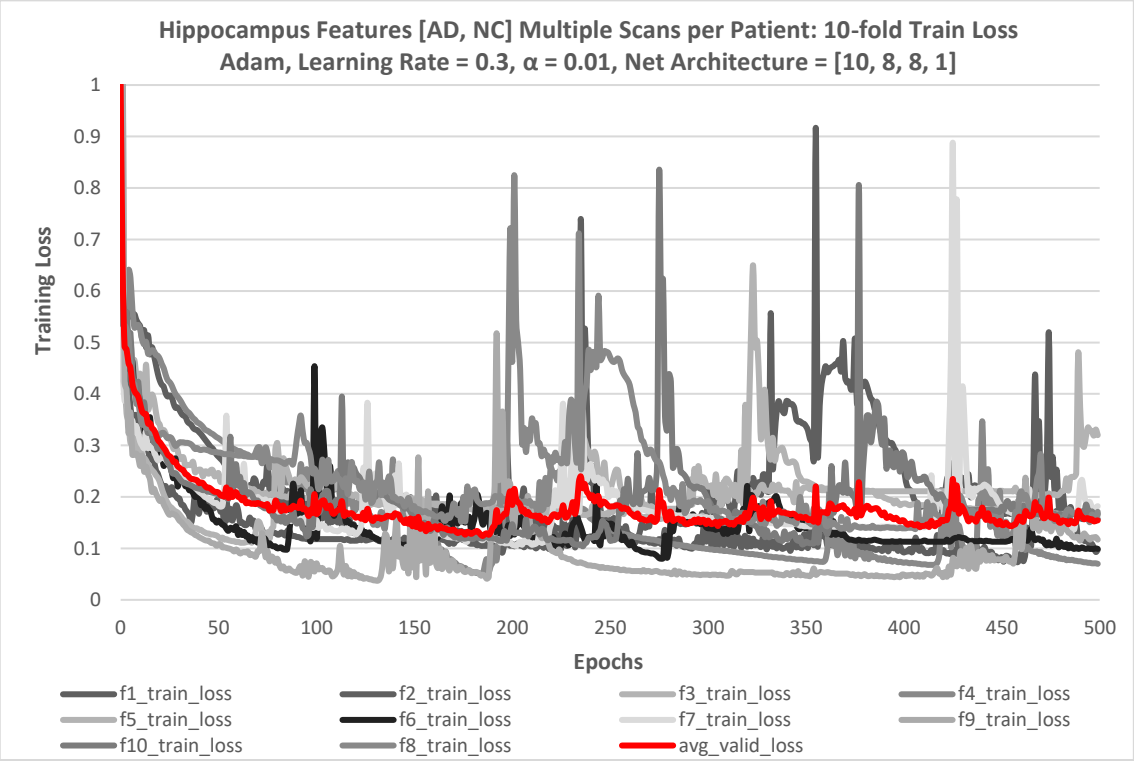


Figure G.49. Caption in the title.

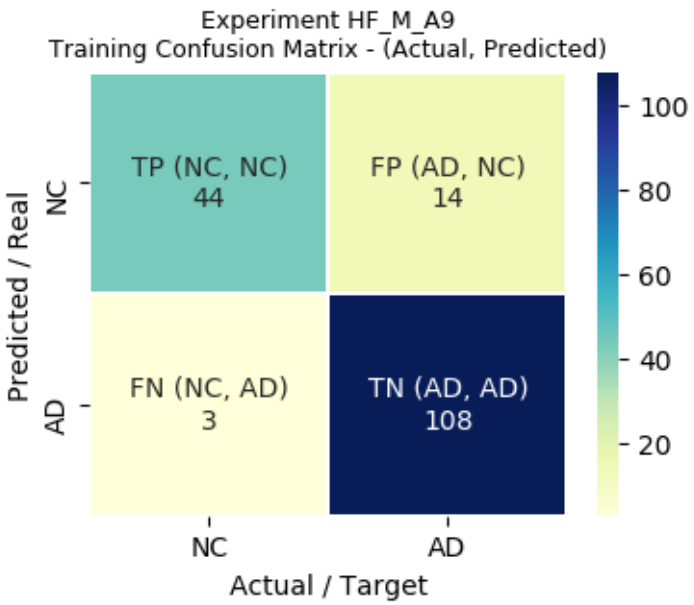


Figure G.50. Caption in the title.

G.1.9.2 Validation

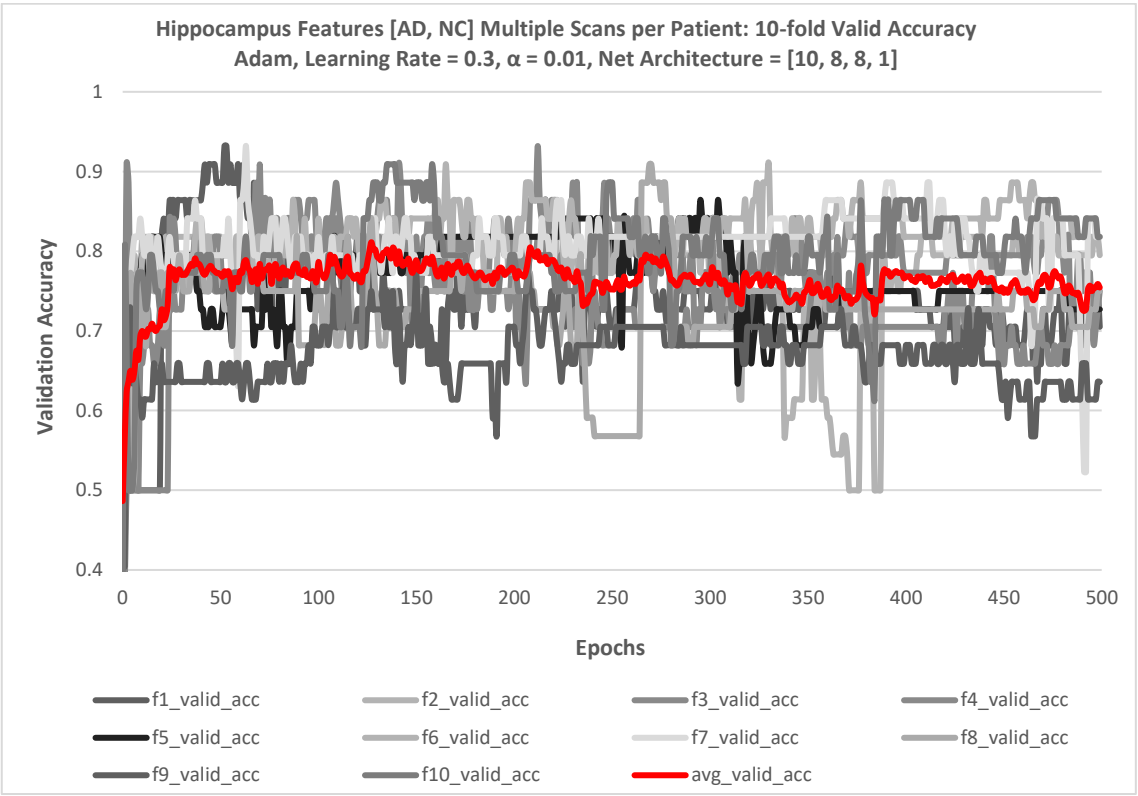


Figure G.51. Caption in the title.

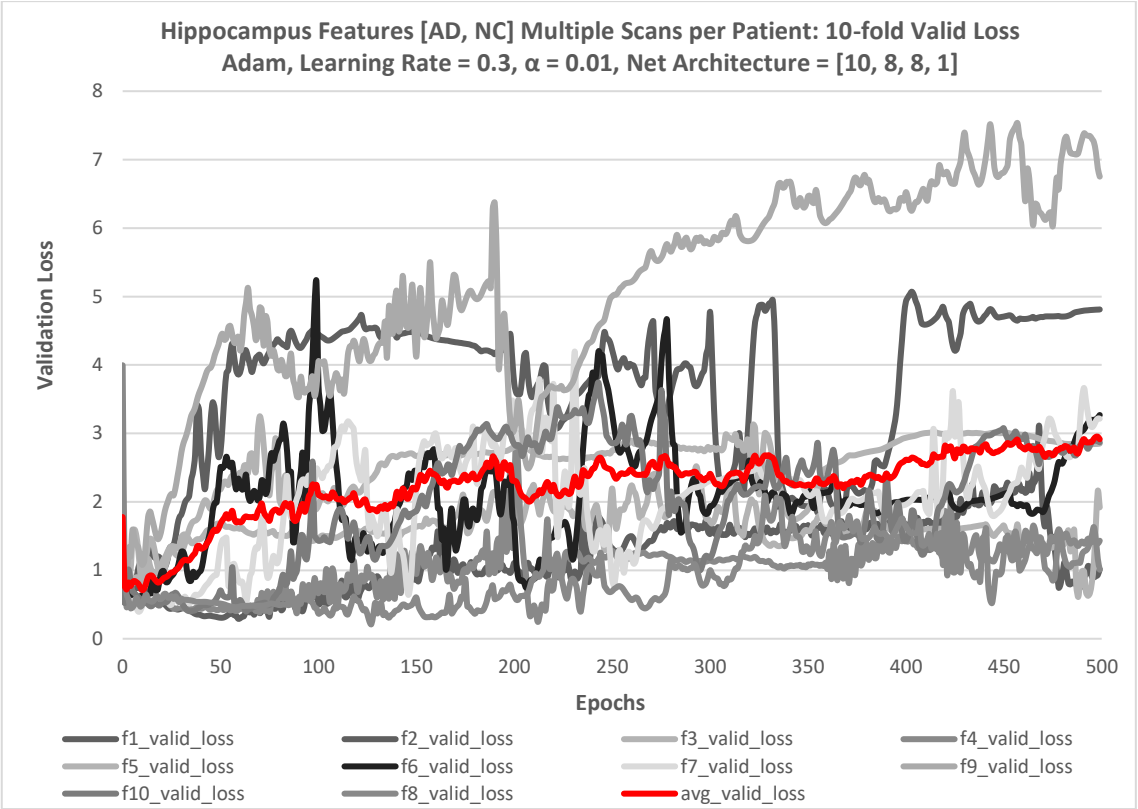


Figure G.52. Caption in the title.

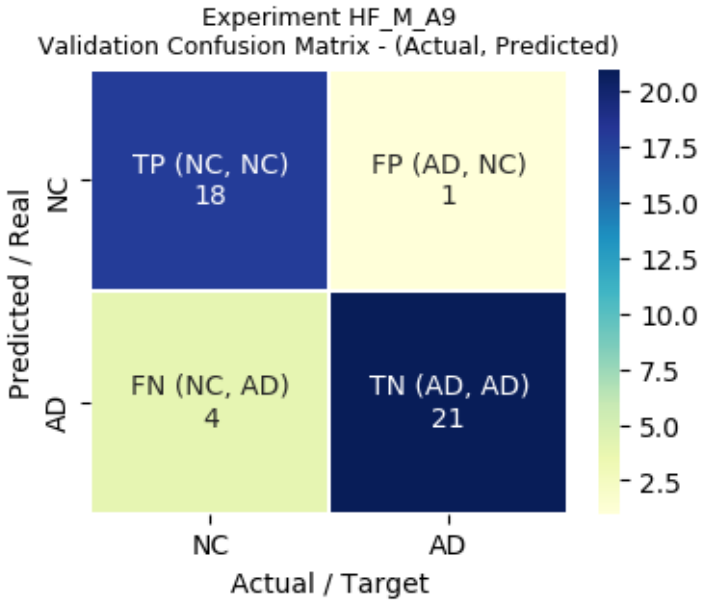


Figure G.53. Caption in the title.

G.1.10 Experiment HF_M_A10: α = 0.001, lr = 0.3, net = [10, 8, 8, 1]

G.1.10.1 Training

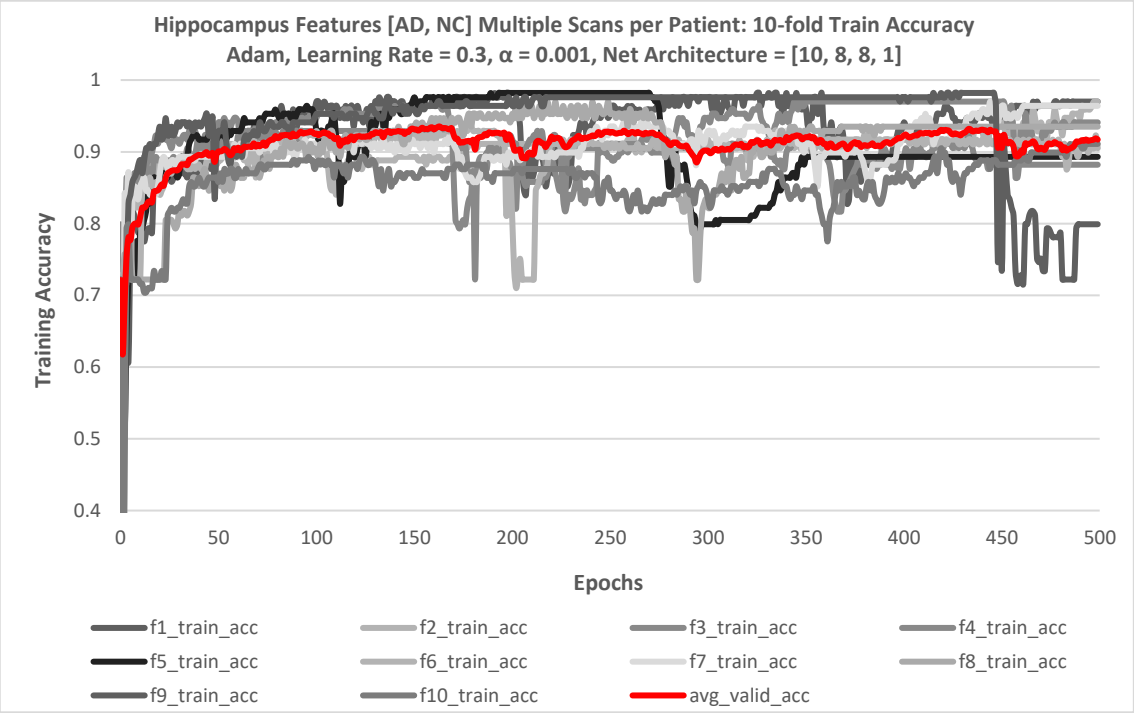


Figure G.54. Caption in the title.

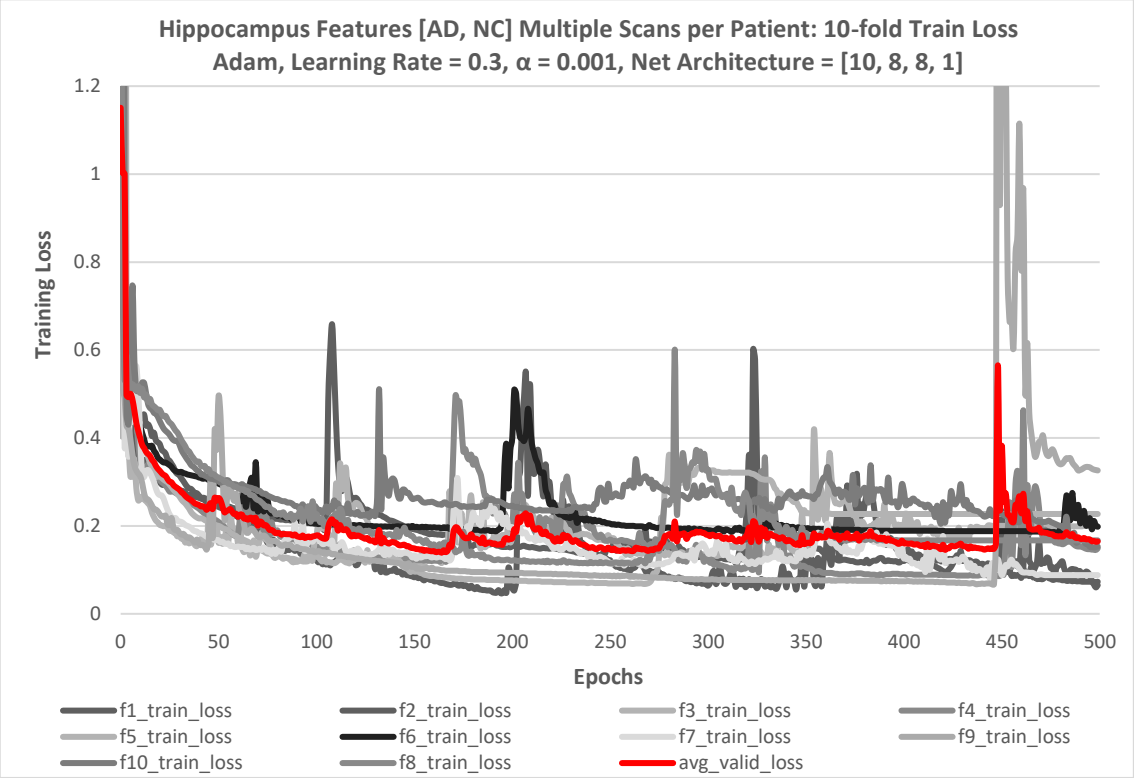


Figure G.55. Caption in the title.

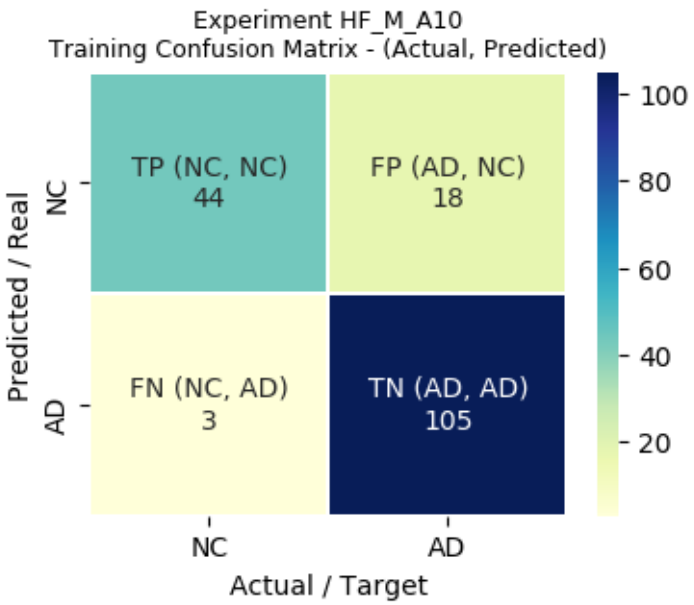


Figure G.56. Caption in the title.

G.1.10.2 Validation

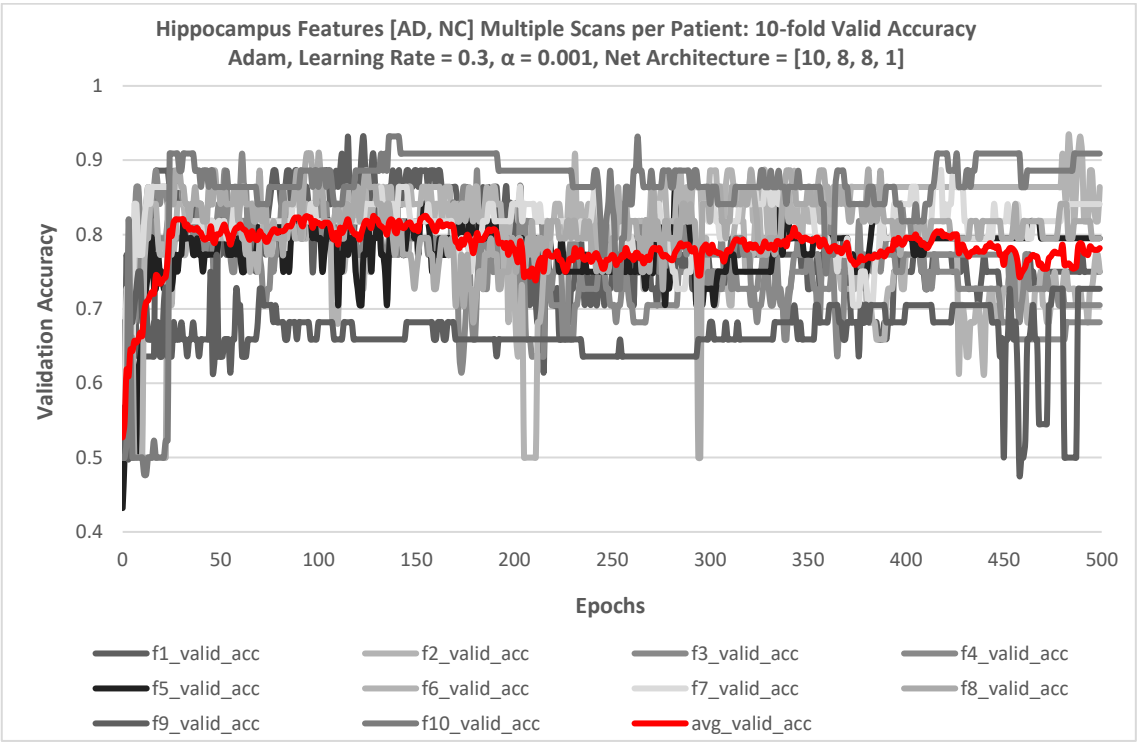


Figure G.57. Caption in the title.

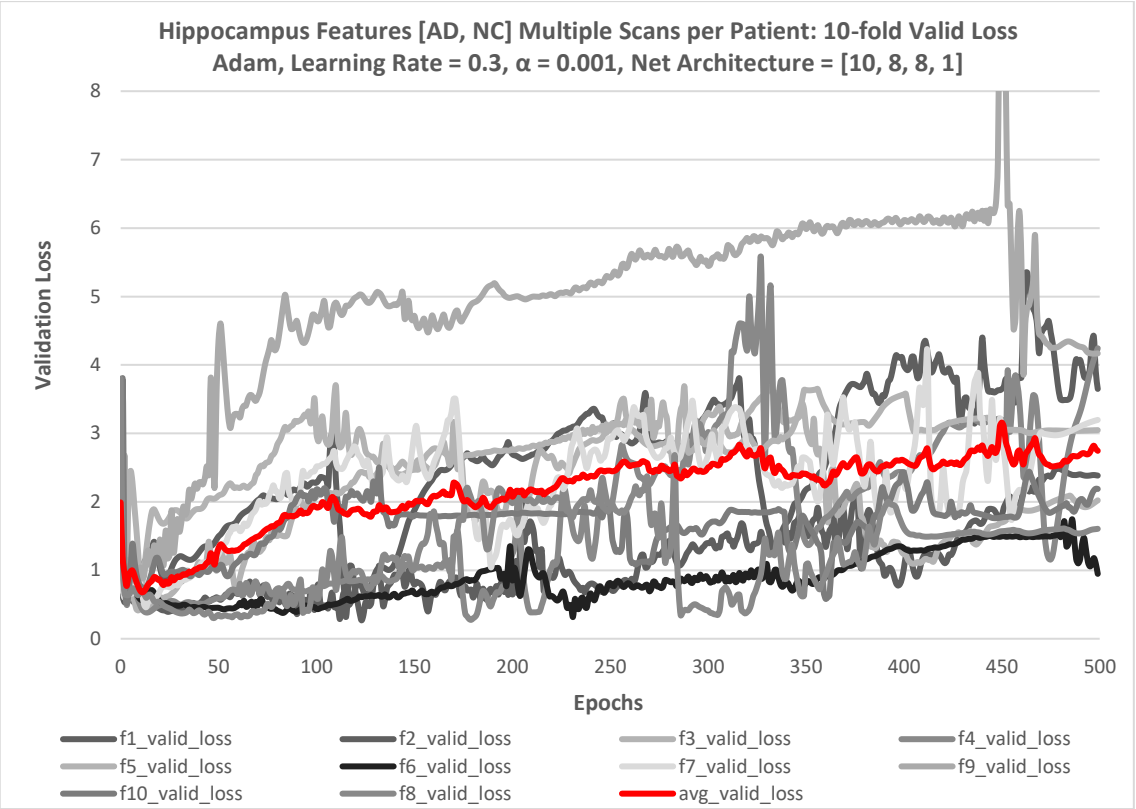


Figure G.58. Caption in the title.

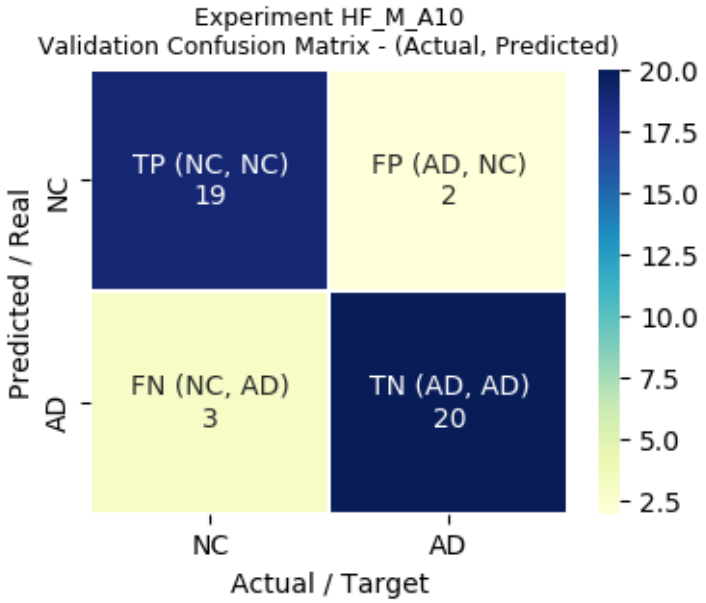


Figure G.59. Caption in the title.

G.1.11 Experiment HF_M_A11: α = 0.1, lr = 0.3, net = [10, 30, 1]

G.1.11.1 Training

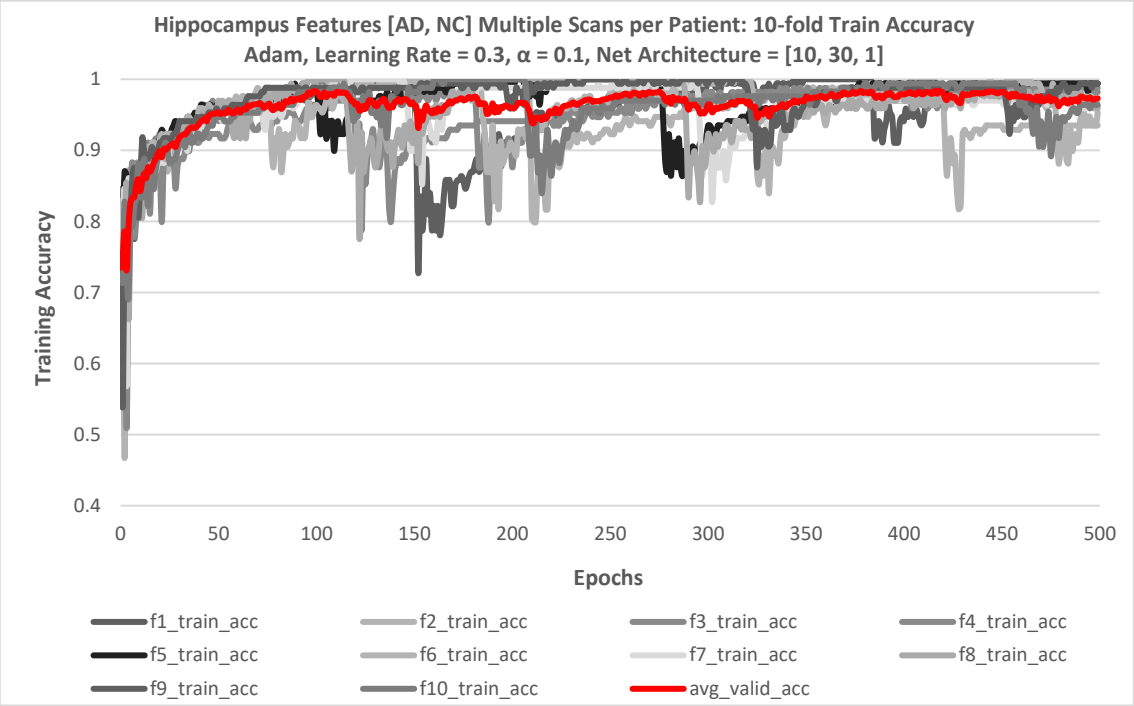


Figure G.60. Caption in the title.

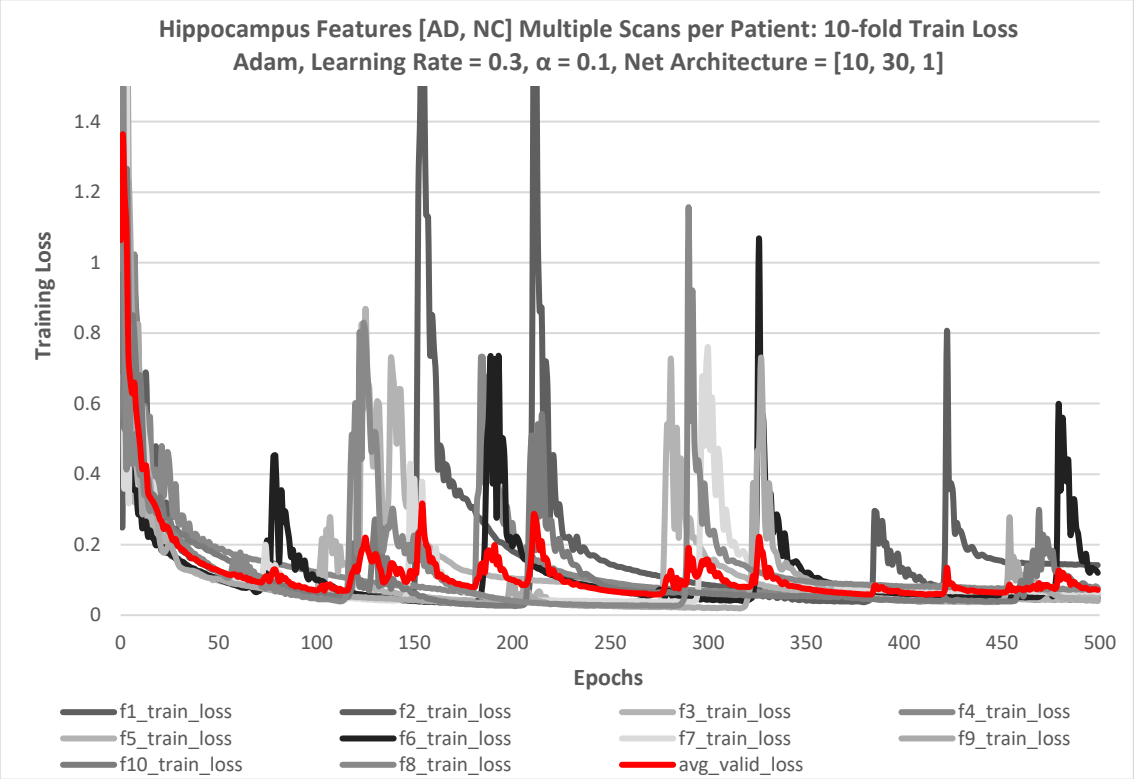


Figure G.61. Caption in the title.

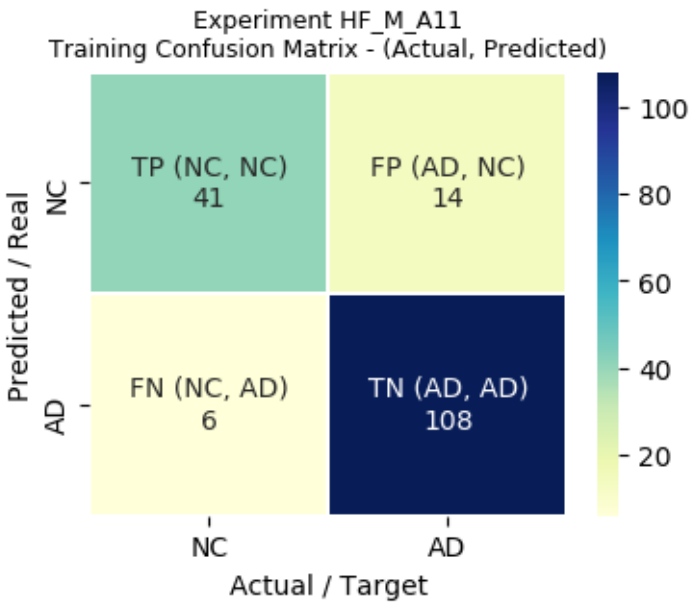


Figure G.62. Caption in the title.

G.1.11.2 Validation

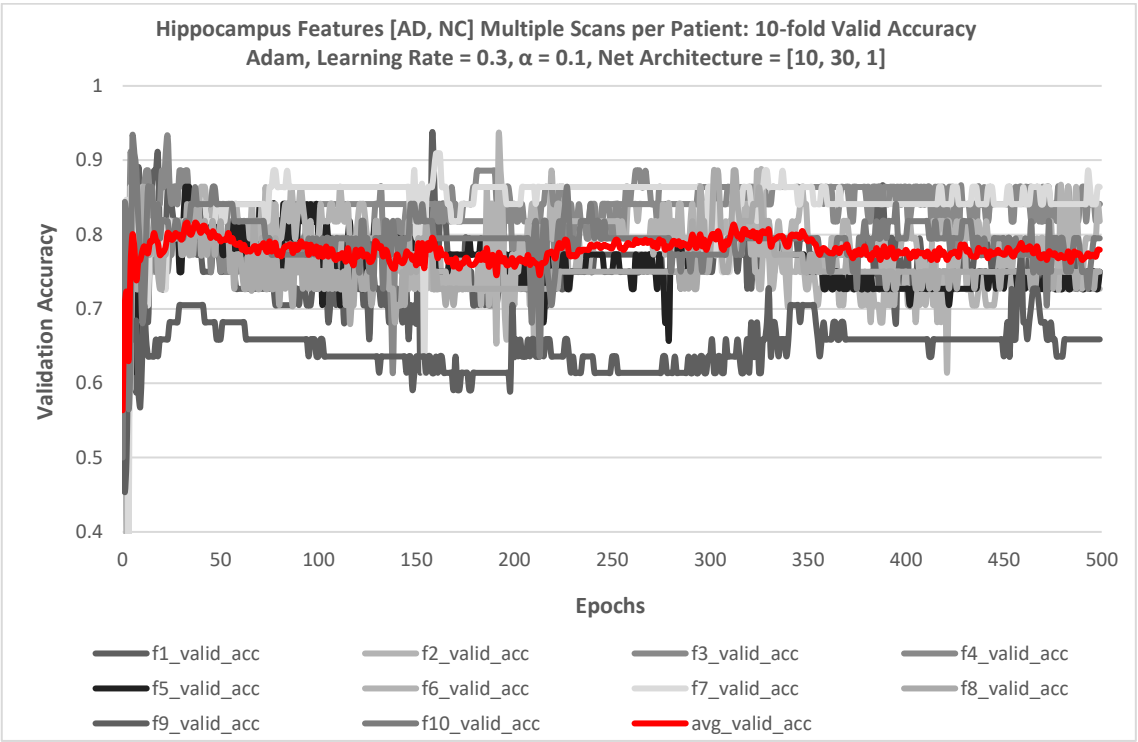


Figure G.63. Caption in the title.

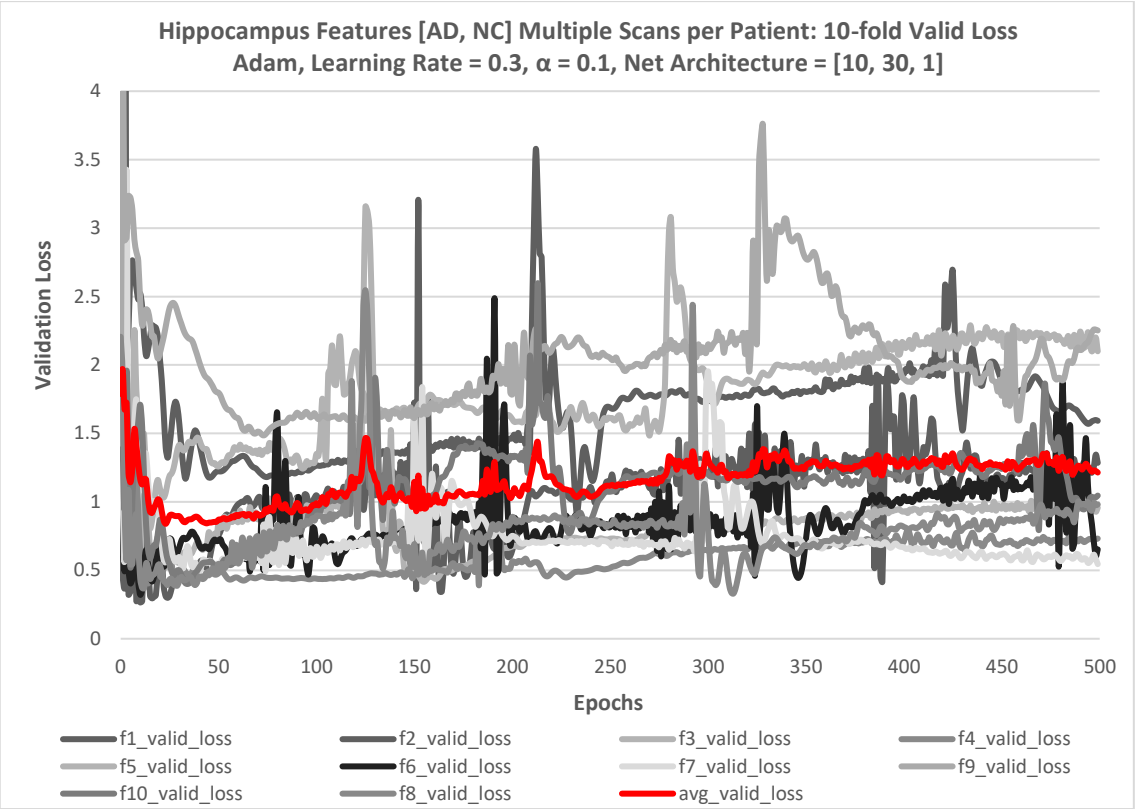


Figure G.64. Caption in the title.

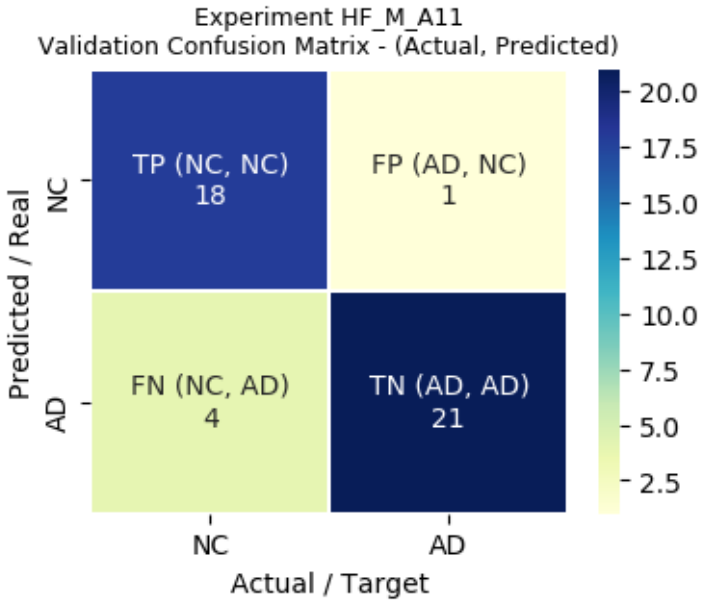


Figure G.65. Caption in the title.

G.1.12 Experiment HF_M_A12: α = 0.1, lr = 0.3, net = [10, 100, 1]

G.1.12.1 Training

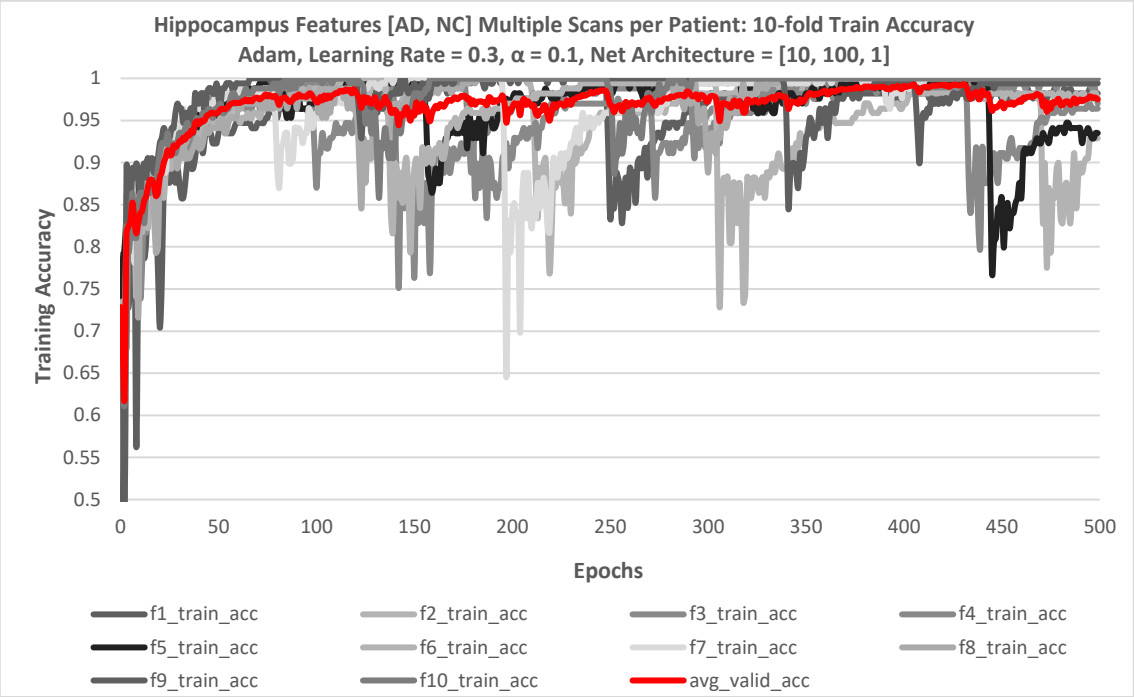


Figure G.66. Caption in the title.

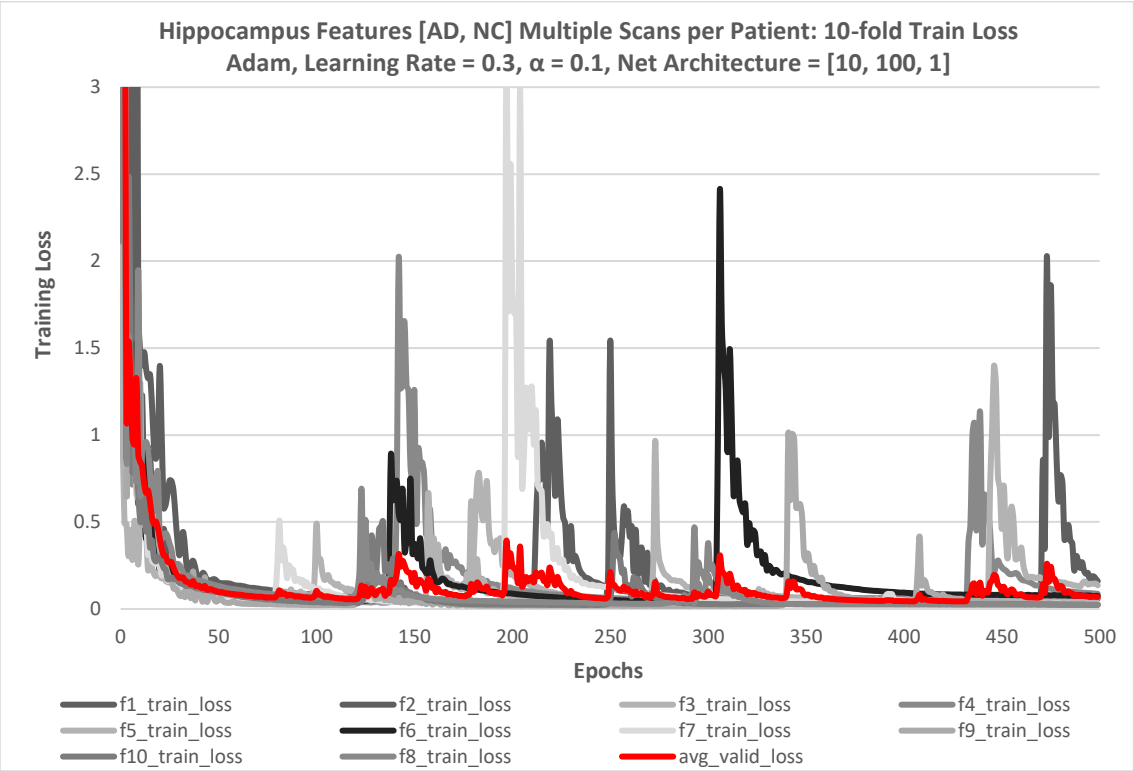


Figure G.67. Caption in the title.

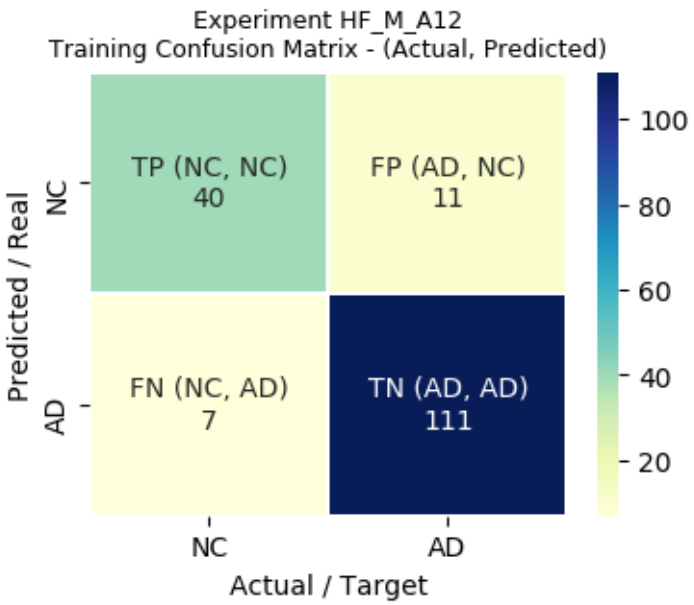


Figure G.68. Caption in the title.

G.1.12.2 Validation

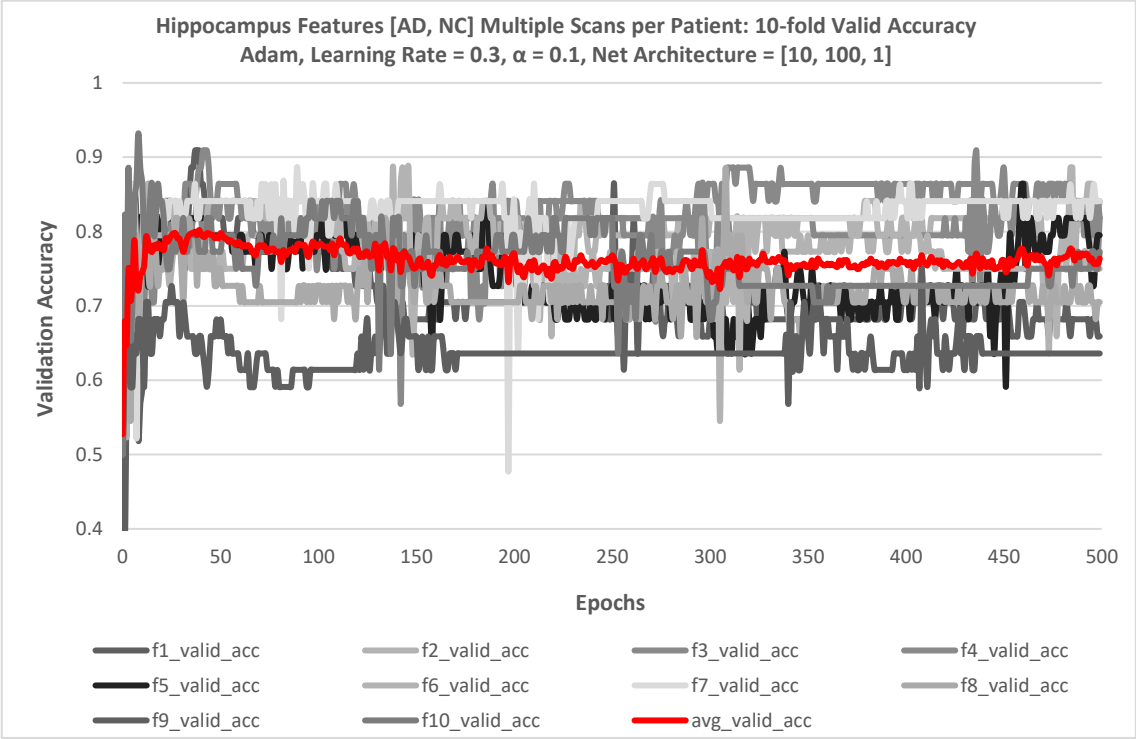


Figure G.69. Caption in the title.

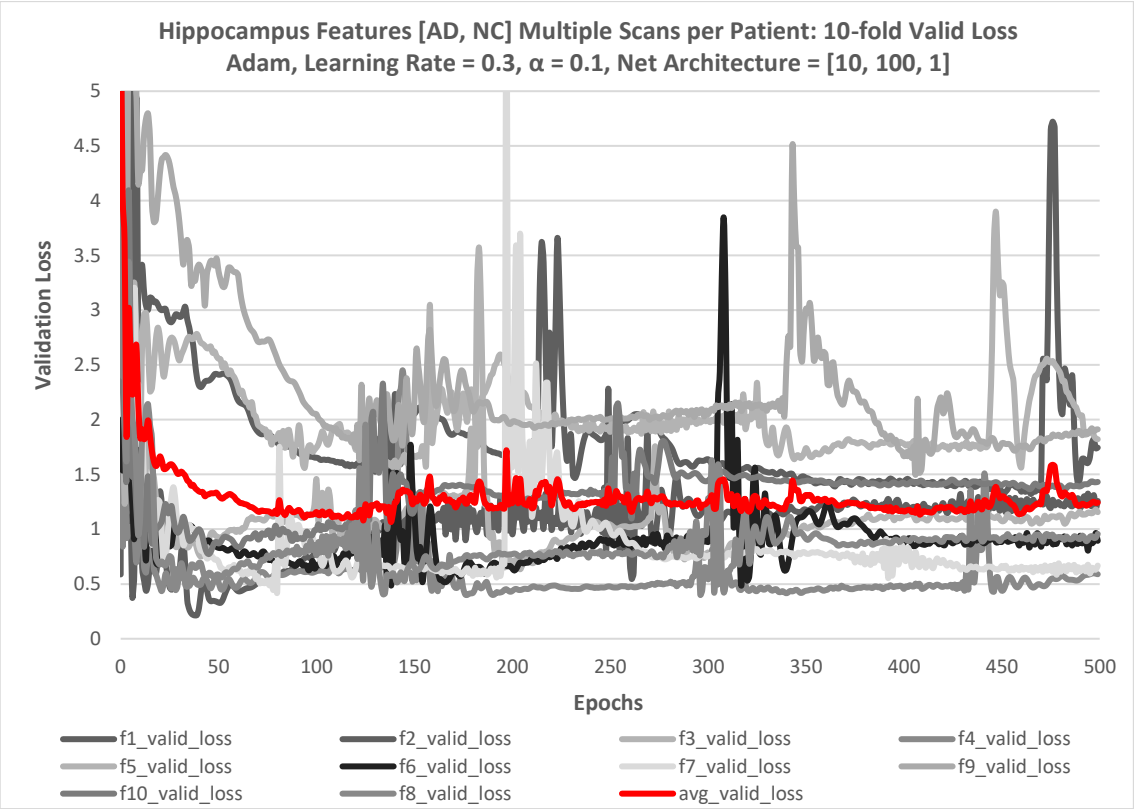


Figure G.70. Caption in the title.

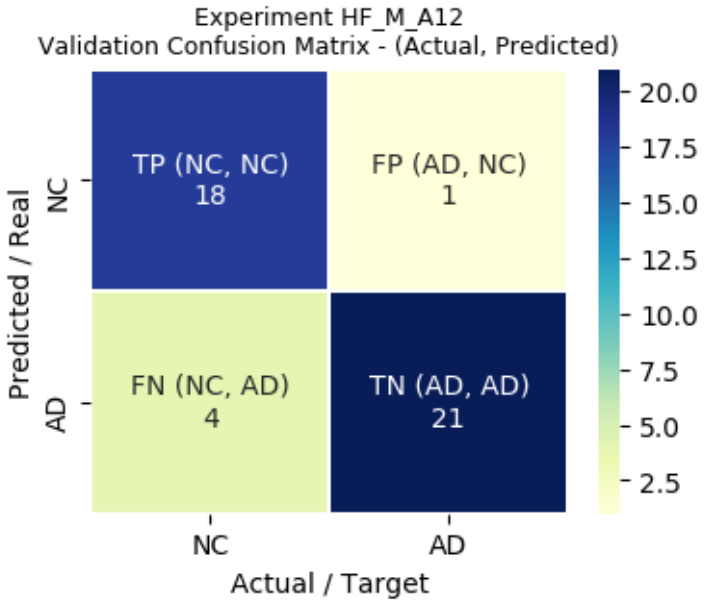


Figure G.71. Caption in the title.

G.1.13 Experiment HF_M_A13: α = 0.1, lr = 0.3, net = [10, 20, 20, 1]

G.1.13.1 Training

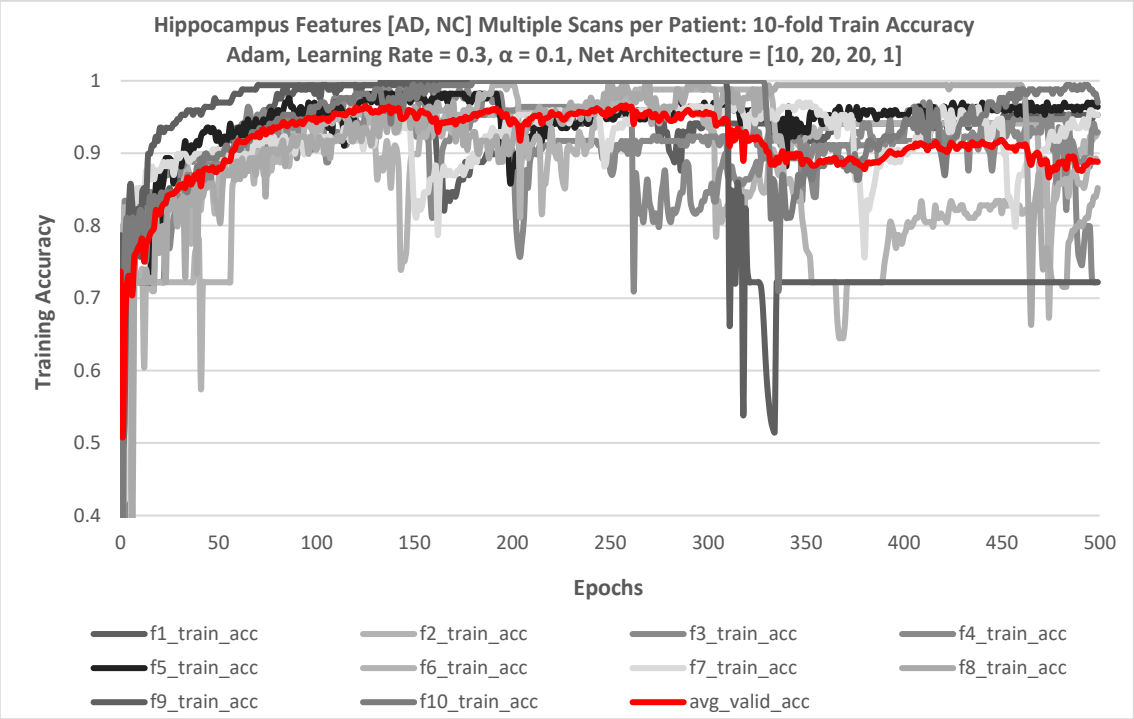


Figure G.72. Caption in the title.

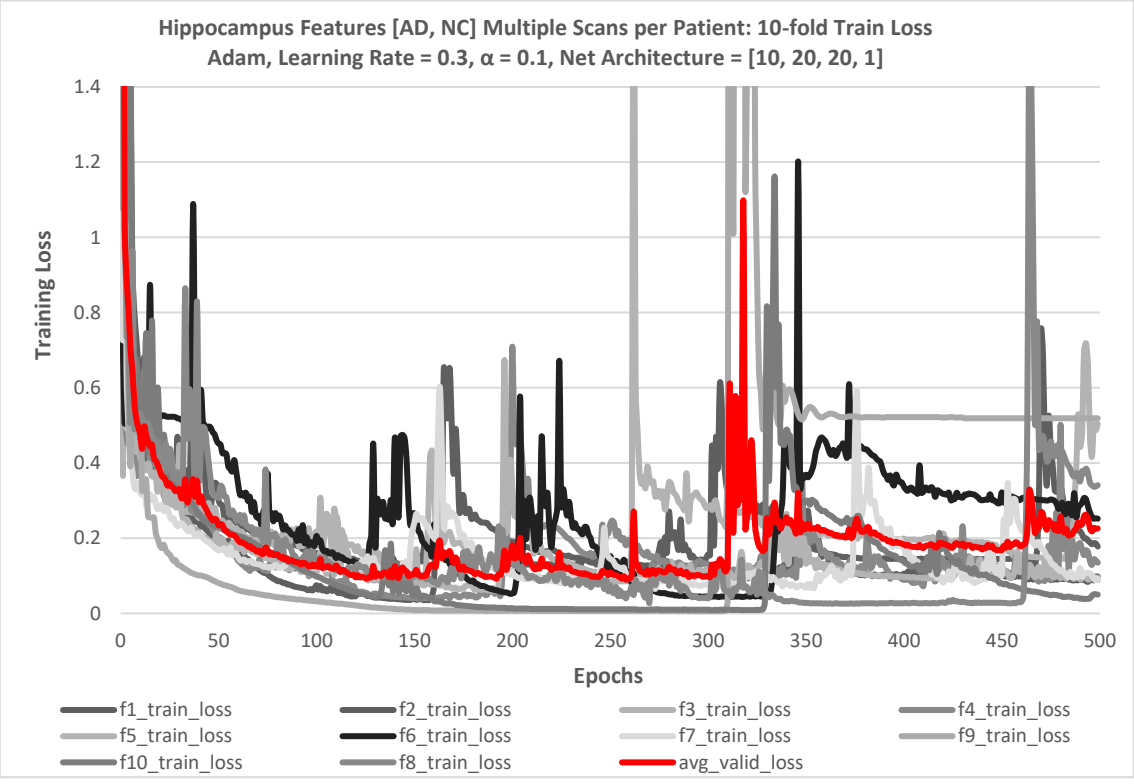


Figure G.73. Caption in the title.

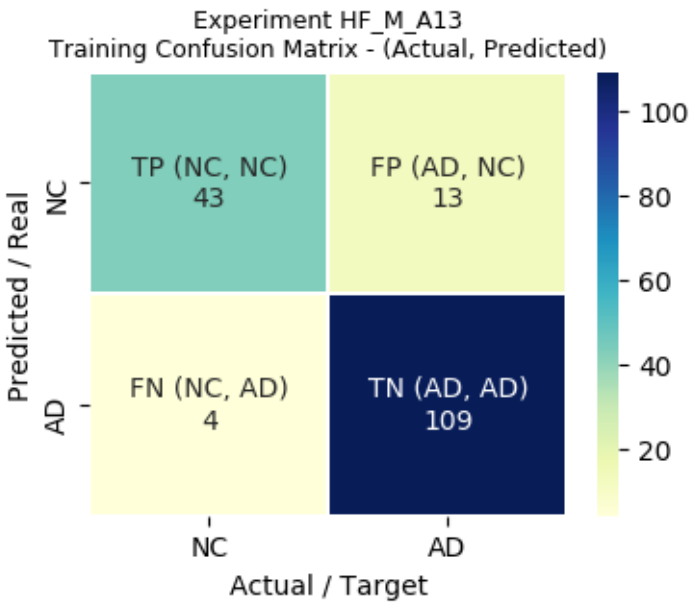


Figure G.74. Caption in the title.

G.1.13.2 Validation

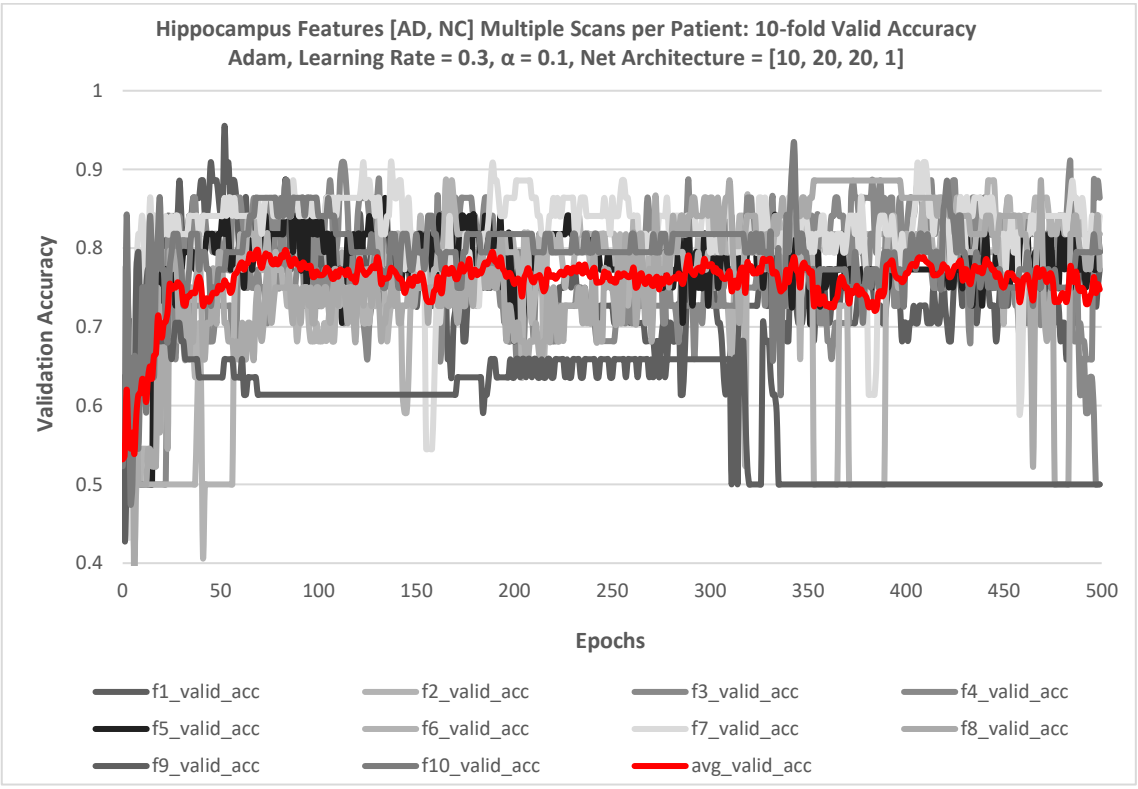


Figure G.75. Caption in the title.

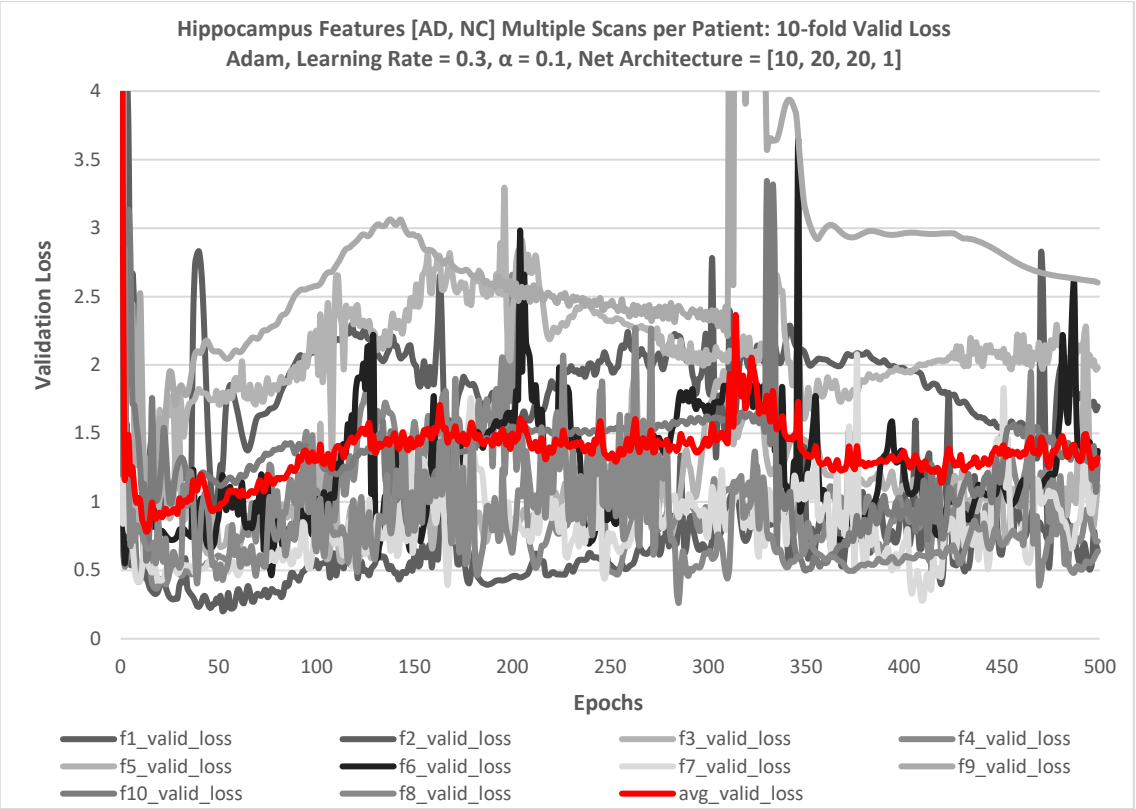


Figure G.76. Caption in the title.

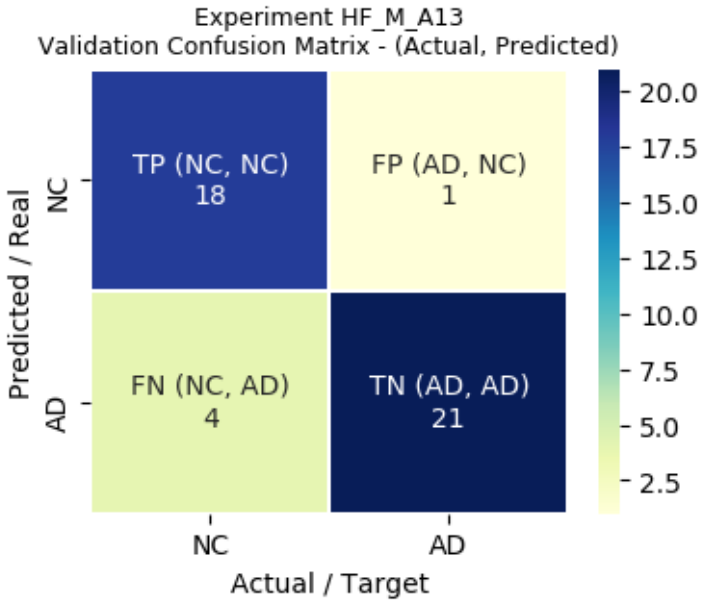


Figure G.77. Caption in the title.

G.2 MLP with SGD

G.2.1 Experiment HF_M_S1: α = 7, lr = 0.3, net = [10, 8, 8, 1]

G.2.1.1 Training

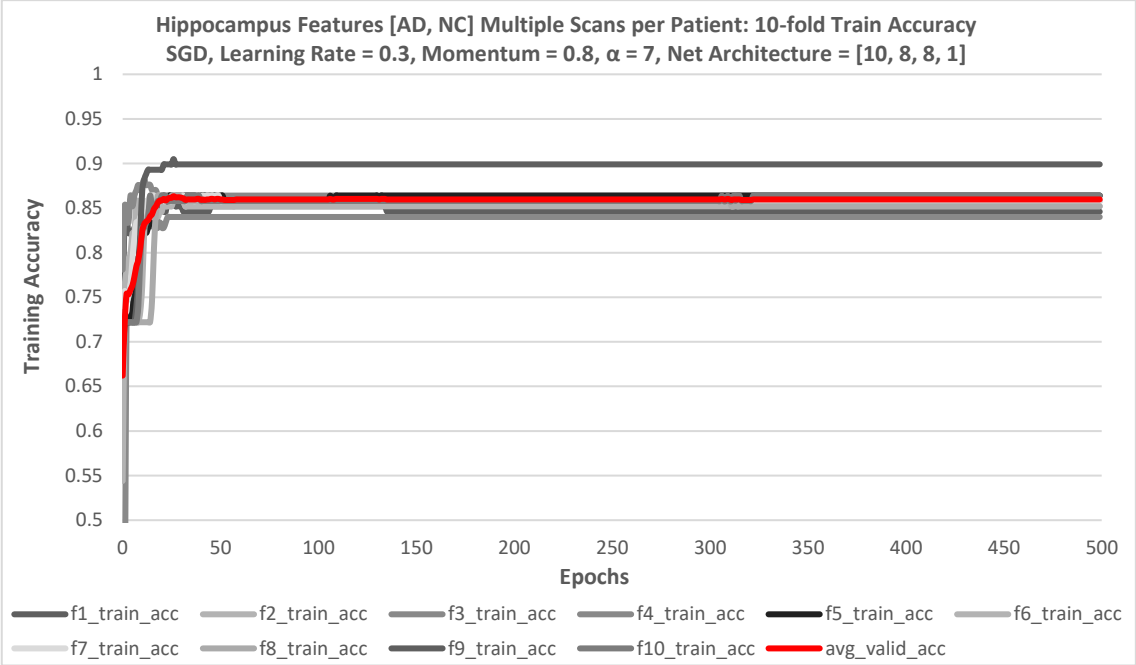


Figure G.78. Caption in the title.

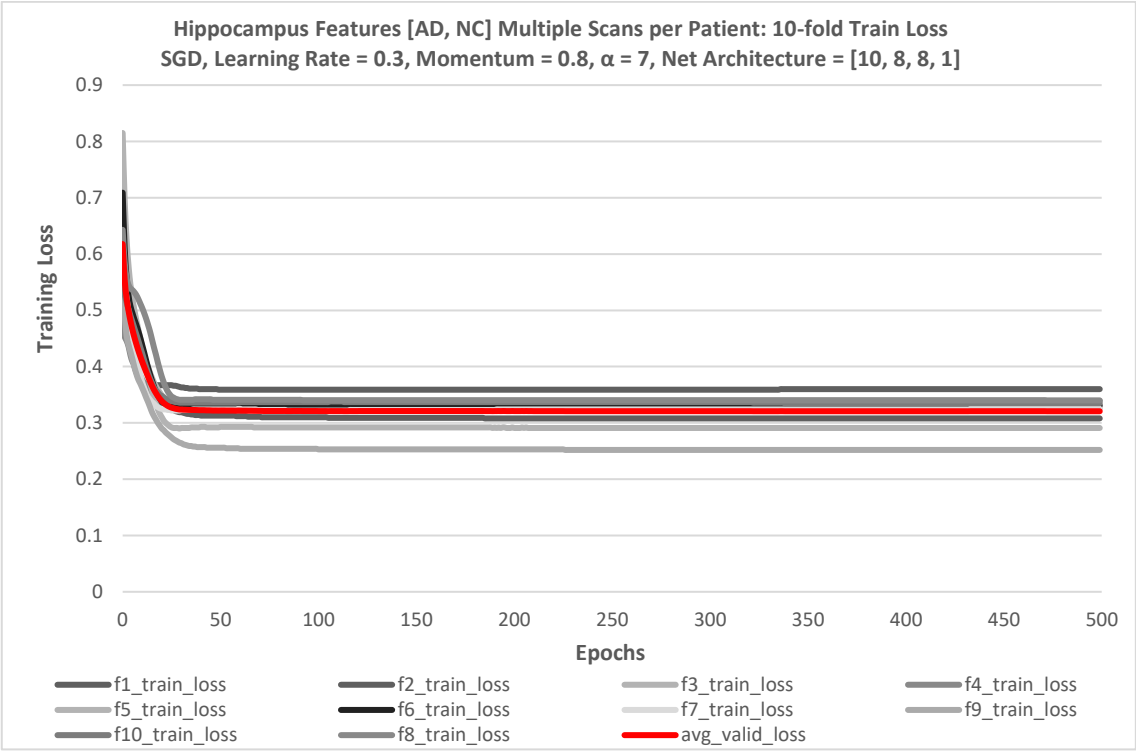


Figure G.79. Caption in the title.

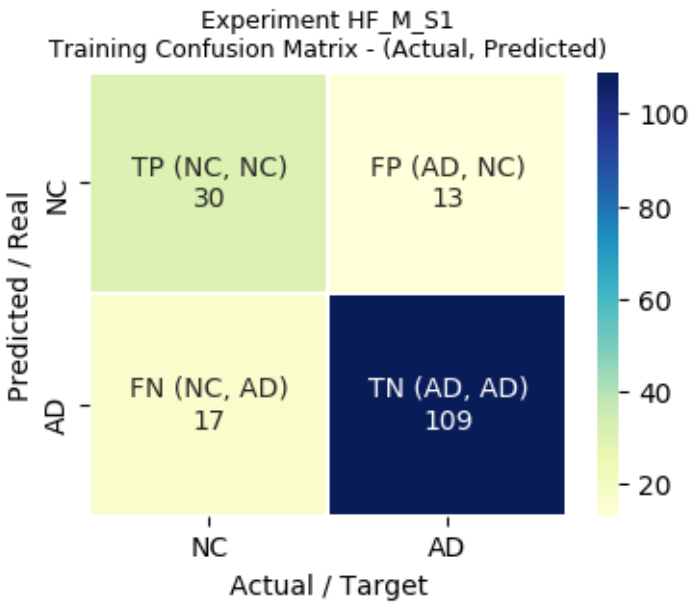


Figure G.80. Caption in the title.

G.2.1.2 Validation

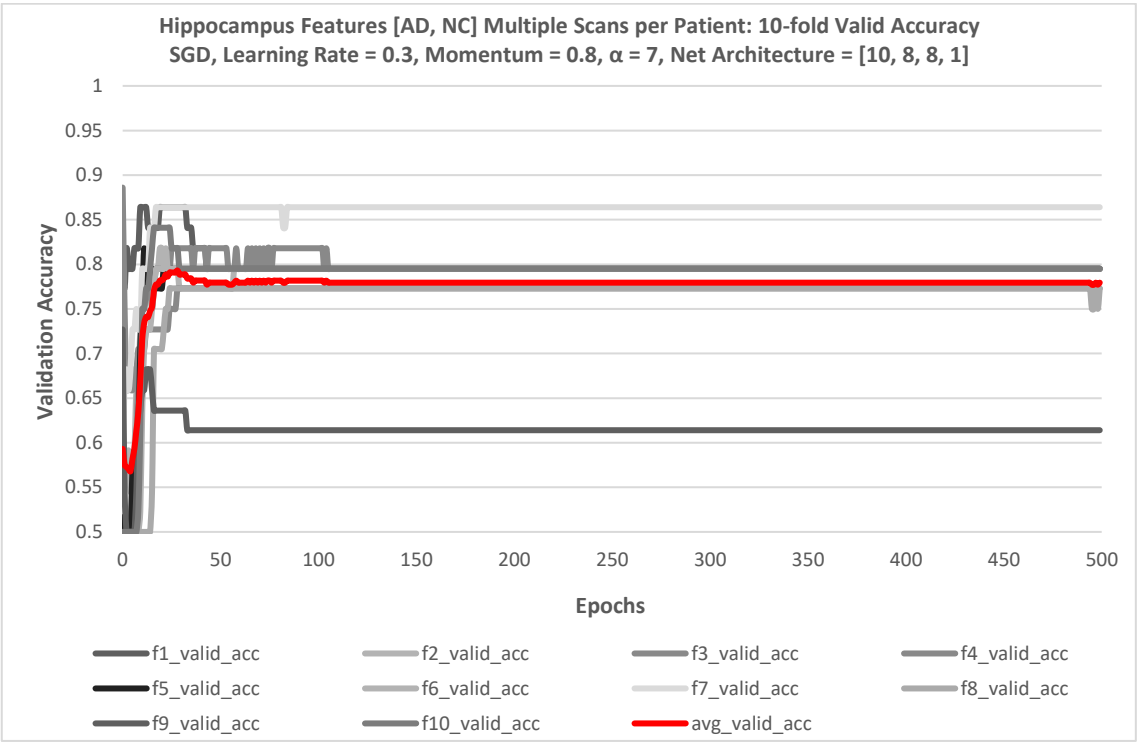


Figure G.81. Caption in the title.

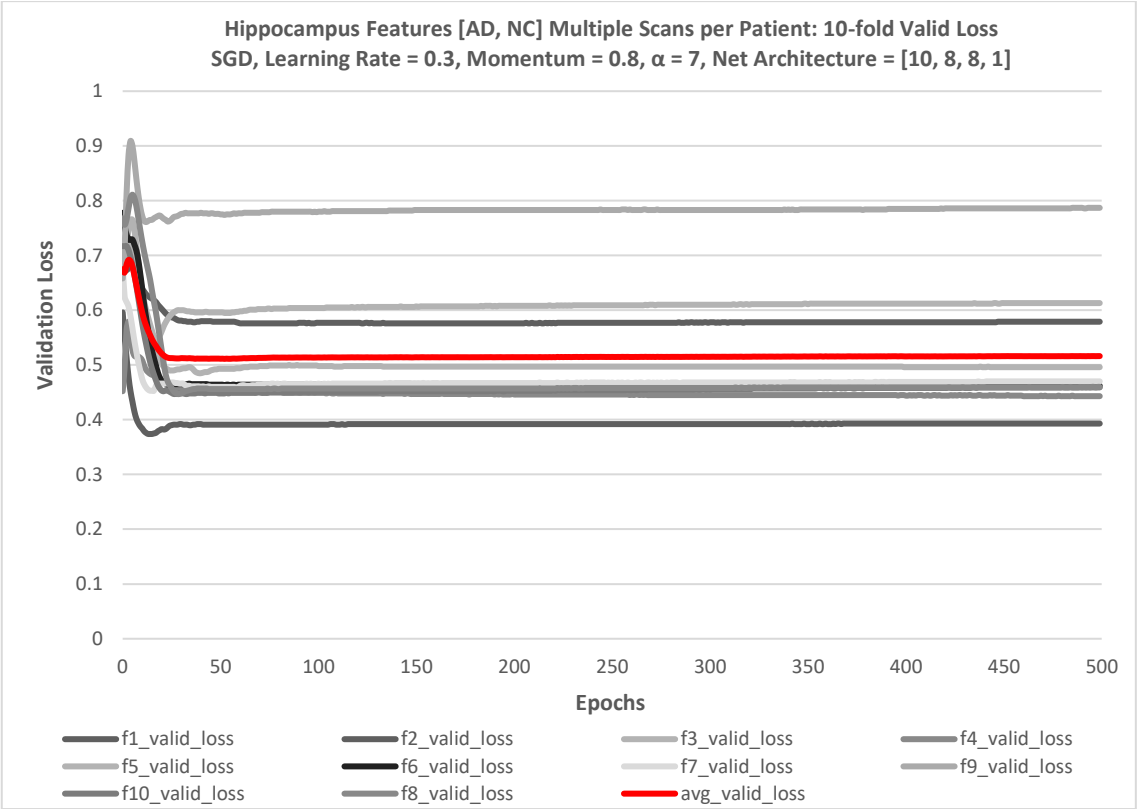


Figure G.82. Caption in the title.

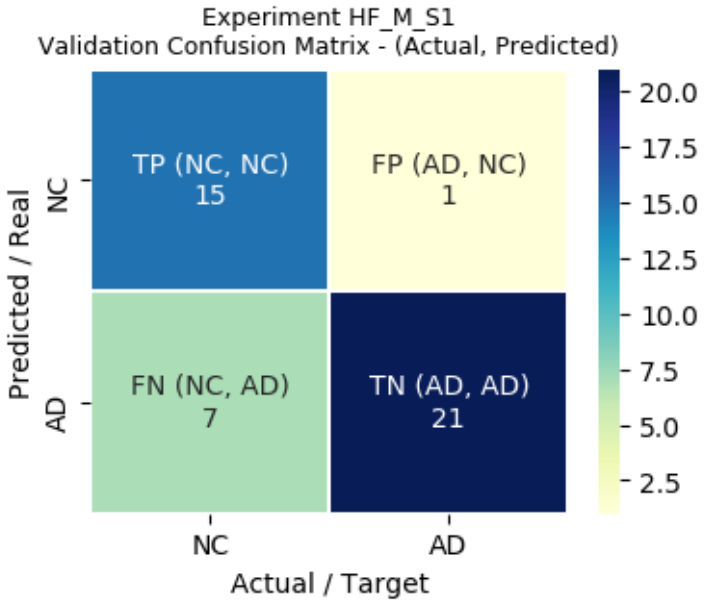


Figure G.83. Caption in the title.

G.2.2 Experiment HF_M_S2: α = 7, lr = 0.03, net = [10, 8, 8, 1]

G.2.2.1 Training

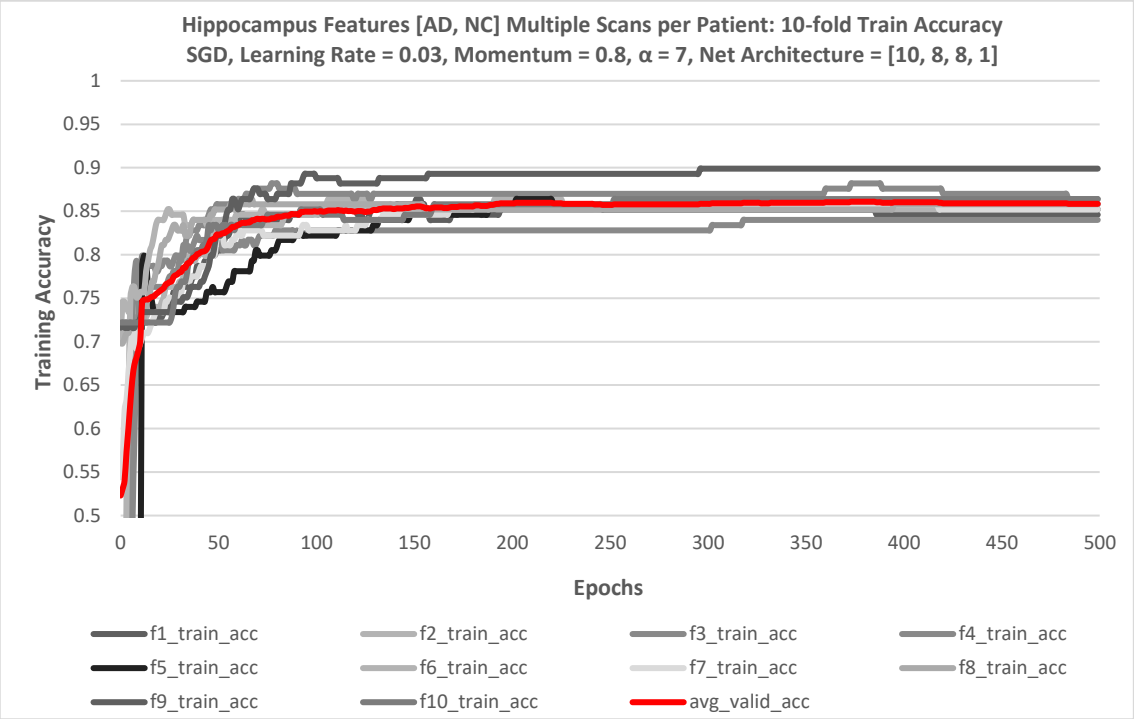


Figure G.84. Caption in the title.

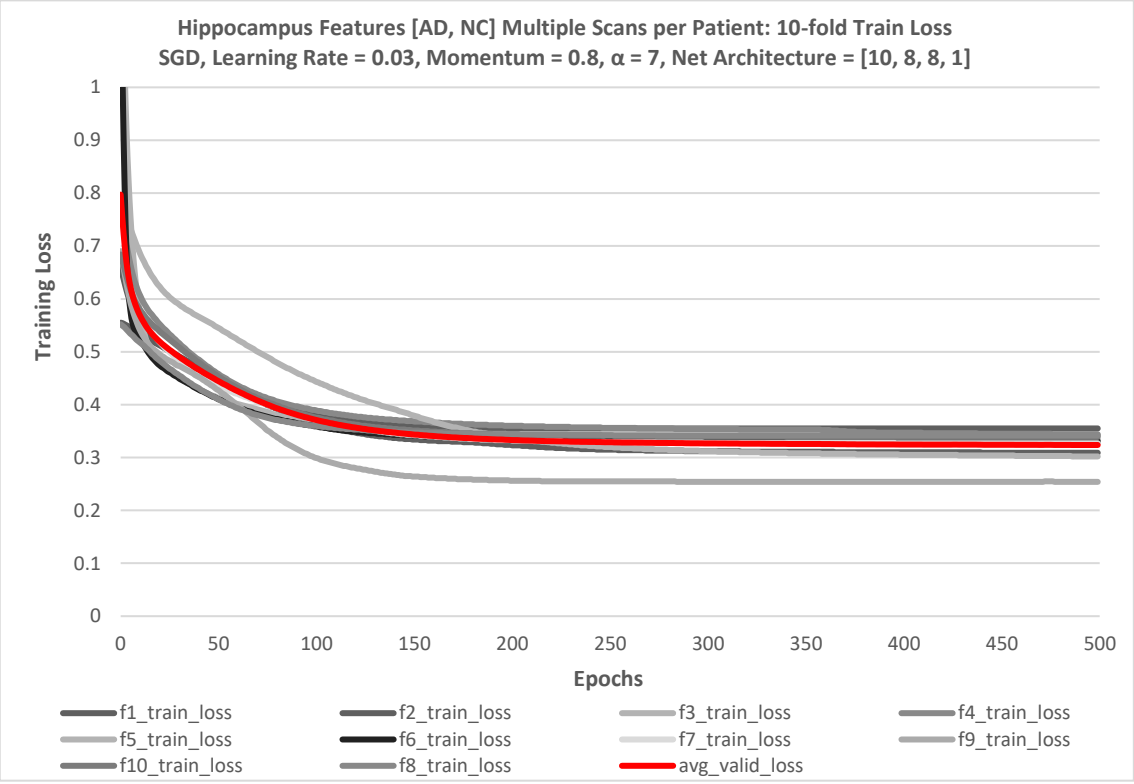


Figure G.85. Caption in the title.

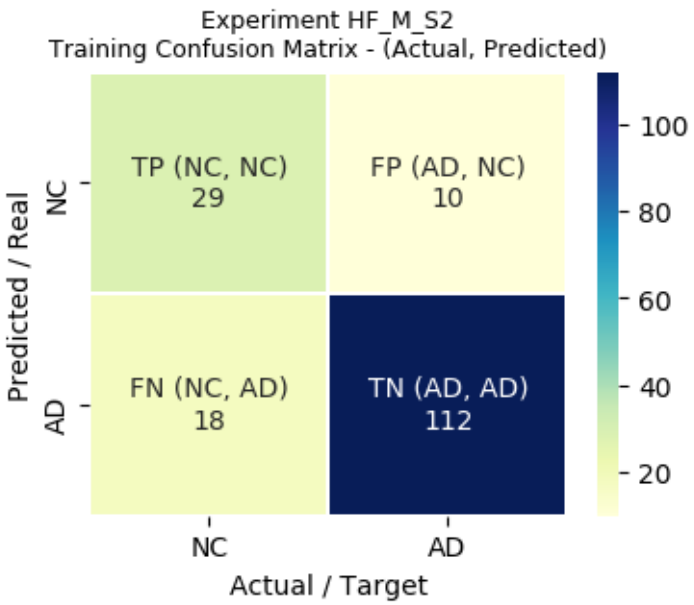


Figure G.86. Caption in the title.

G.2.2.2 Validation

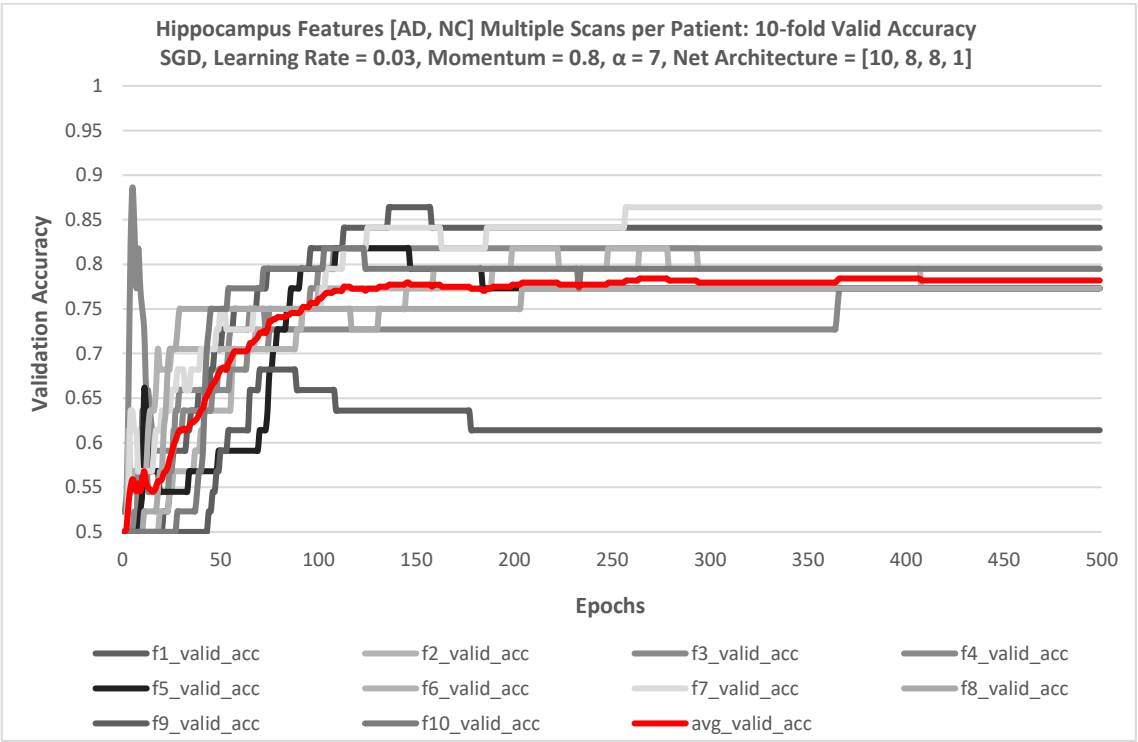


Figure G.87. Caption in the title.

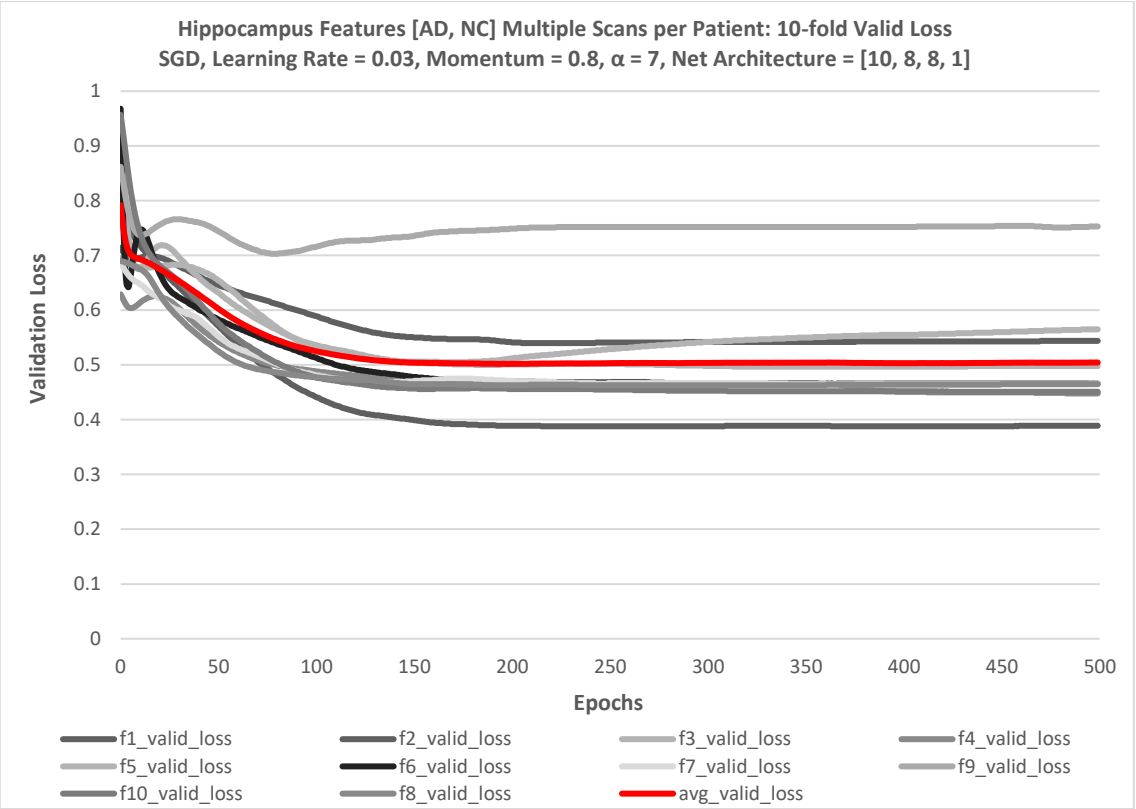


Figure G.88. Caption in the title.

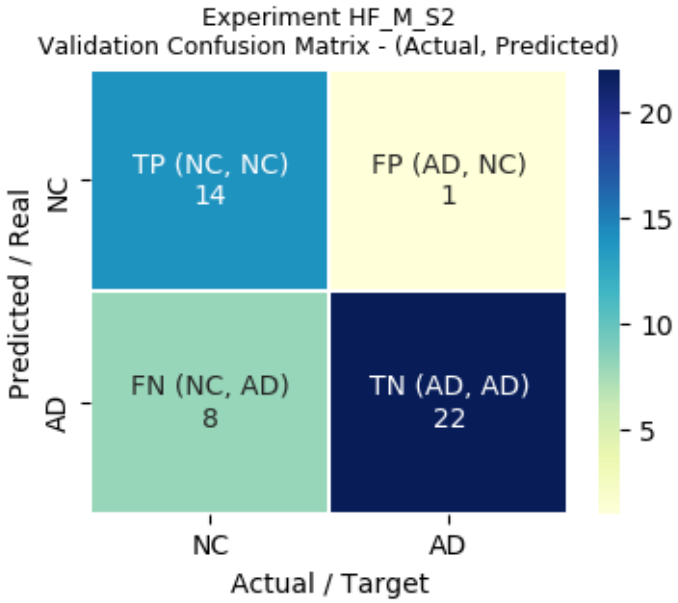


Figure G.89. Caption in the title.

G.2.3 Experiment HF_M_S3: α = 7, lr = 0.003, net = [10, 8, 8, 1]

G.2.3.1 Training

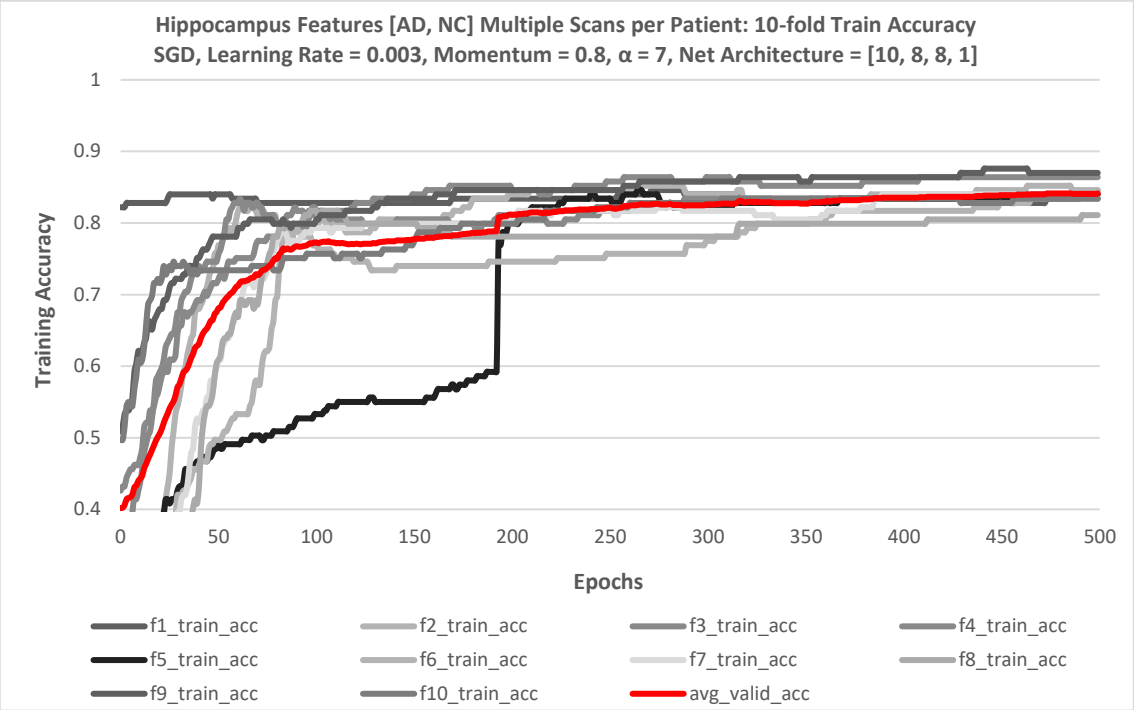


Figure G.90. Caption in the title.

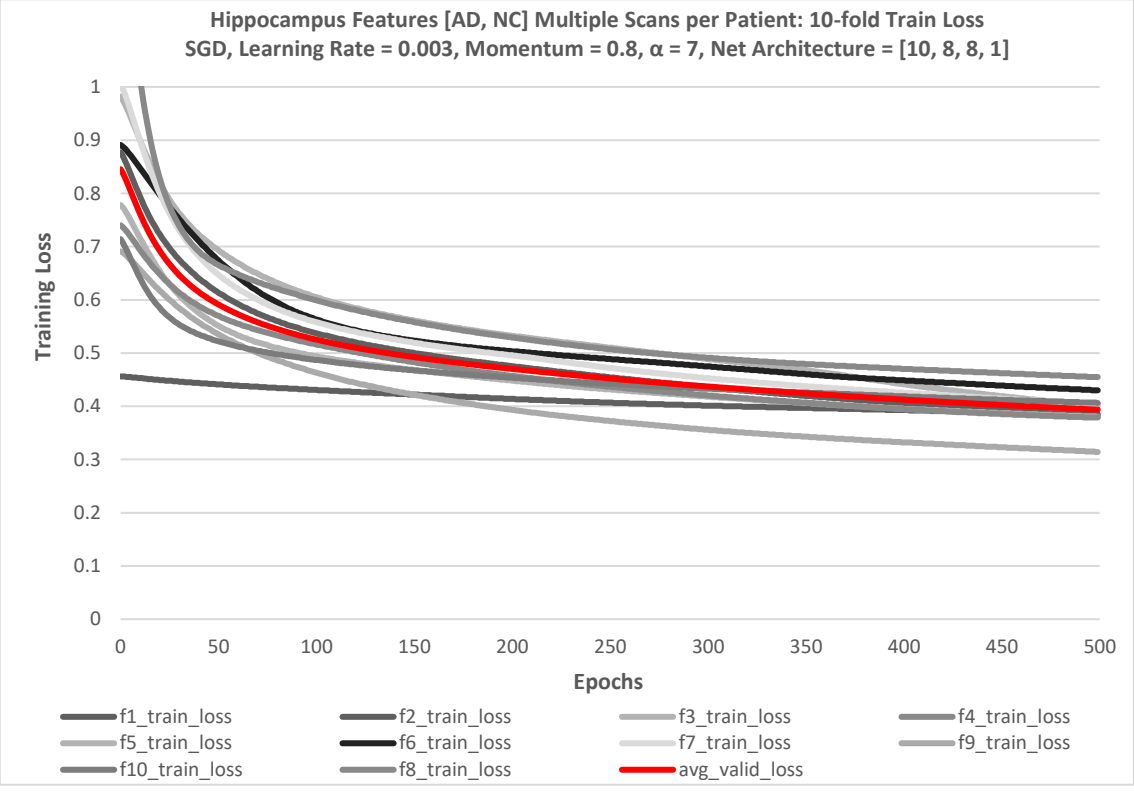


Figure G.91. Caption in the title.

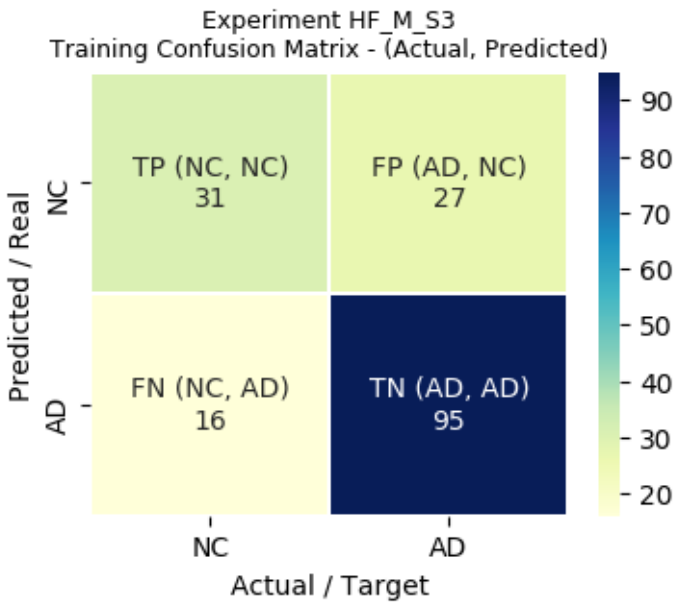


Figure G.92. Caption in the title.

G.2.3.2 Validation

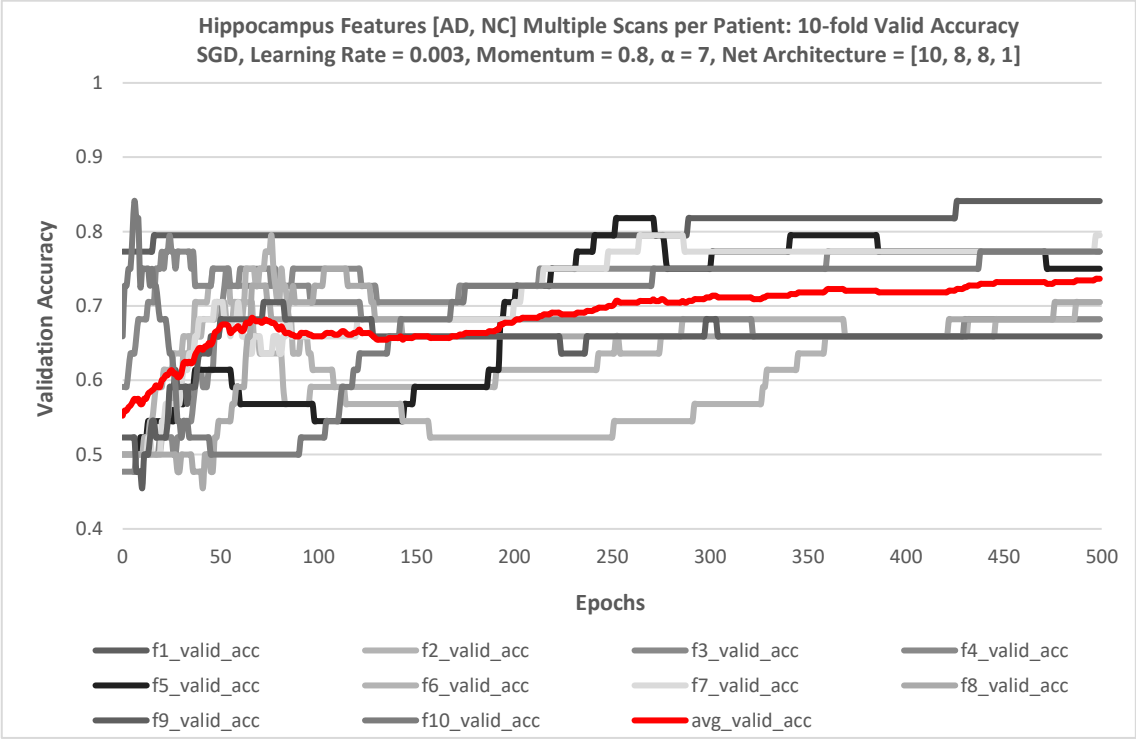


Figure G.93. Caption in the title.

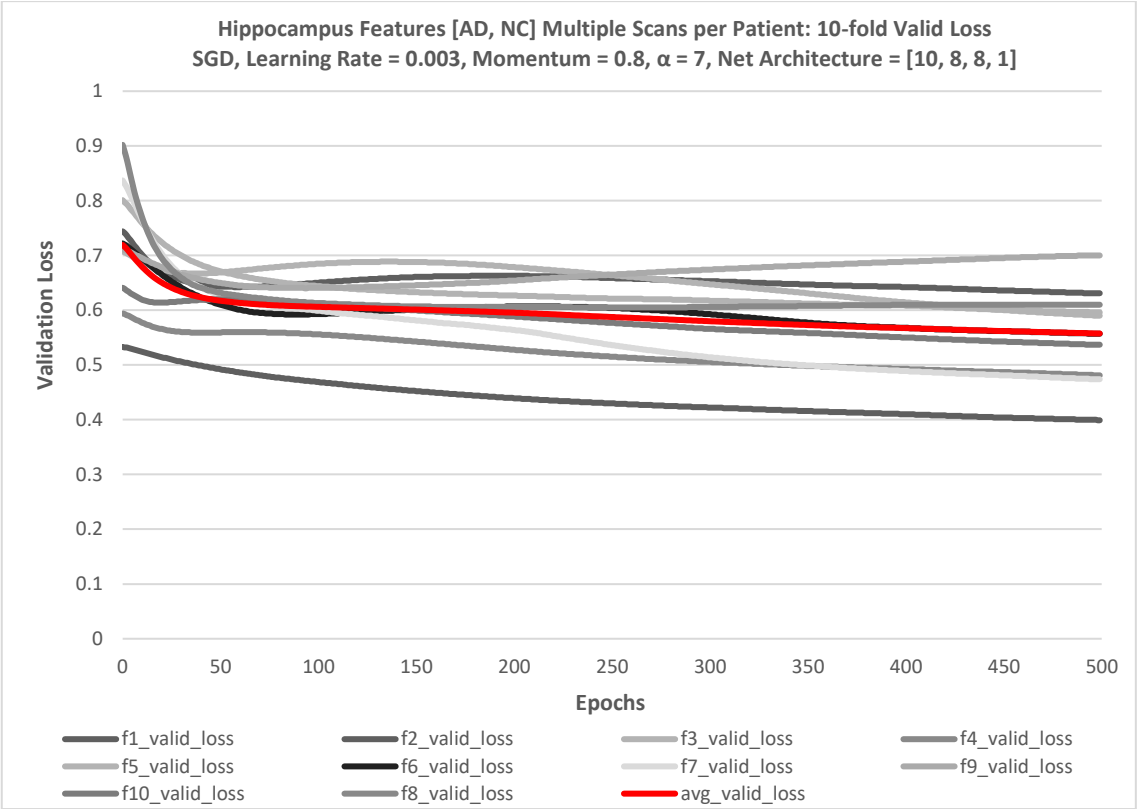


Figure G.94. Caption in the title.

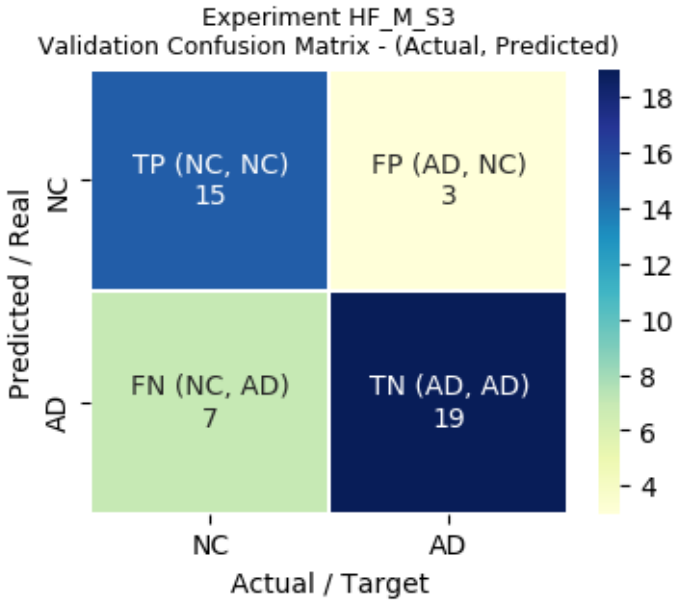


Figure G.95. Caption in the title.

G.2.4 Experiment HF_M_S4: α = 7, lr = 0.0003, net = [10, 8, 8, 1]

G.2.4.1 Training

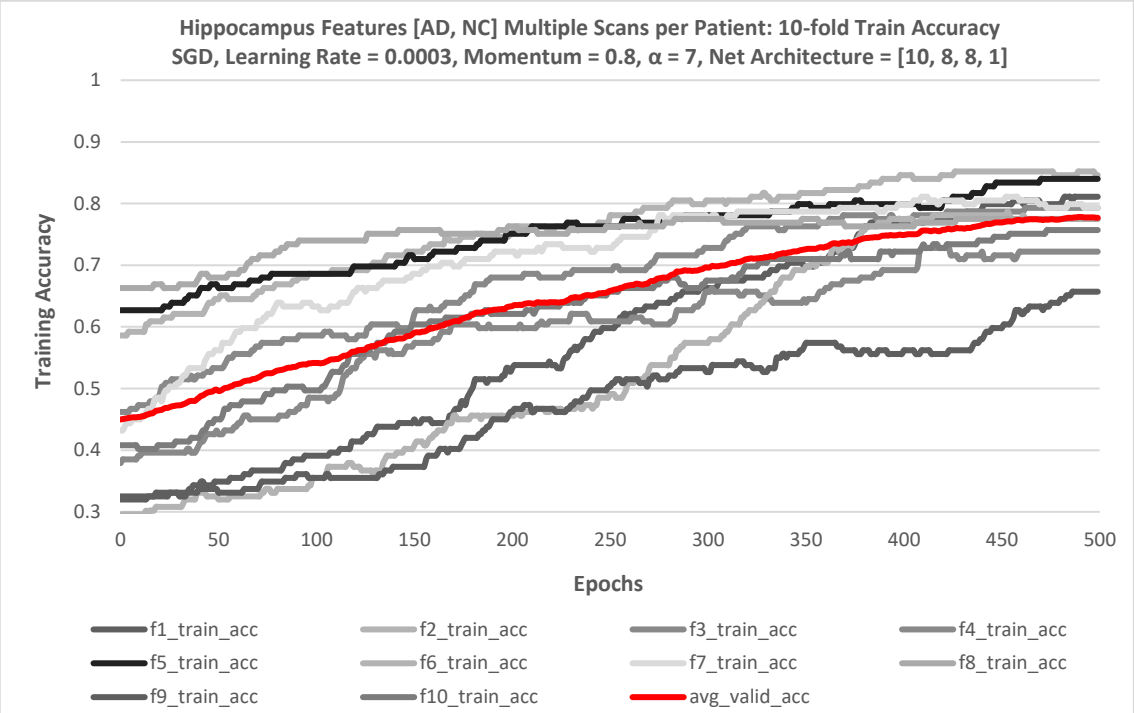


Figure G.96. Caption in the title.

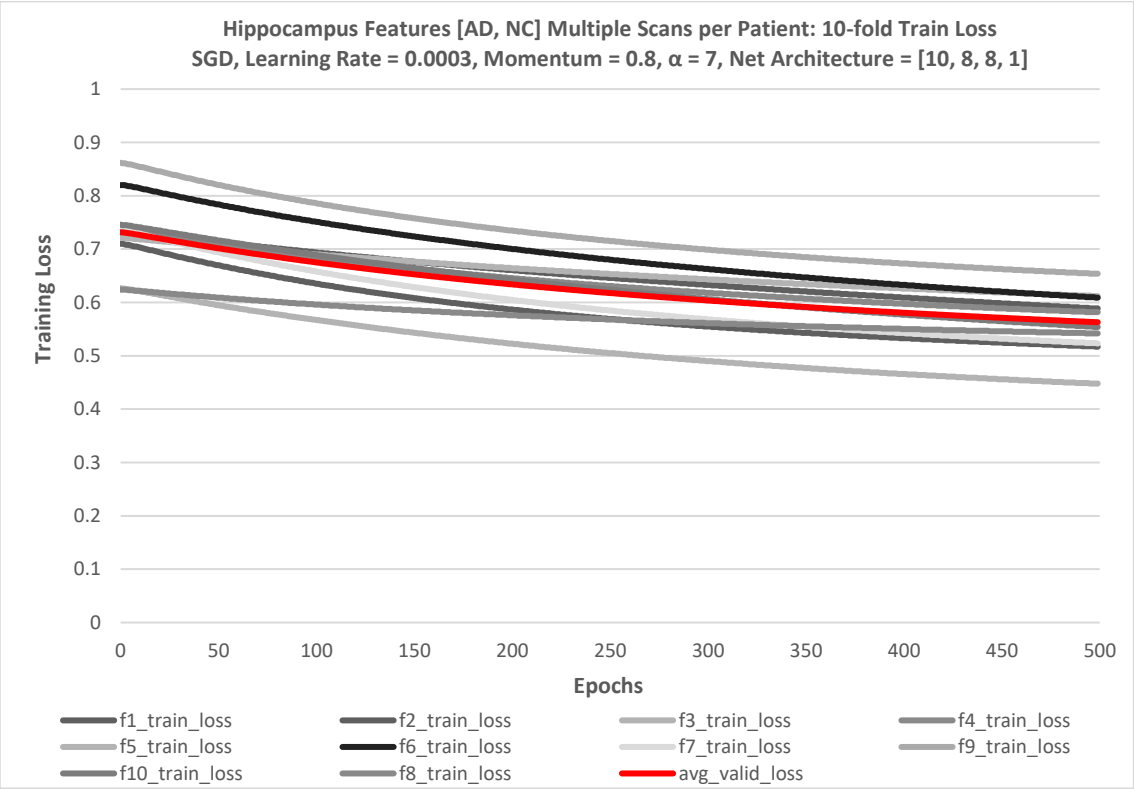


Figure G.97. Caption in the title.

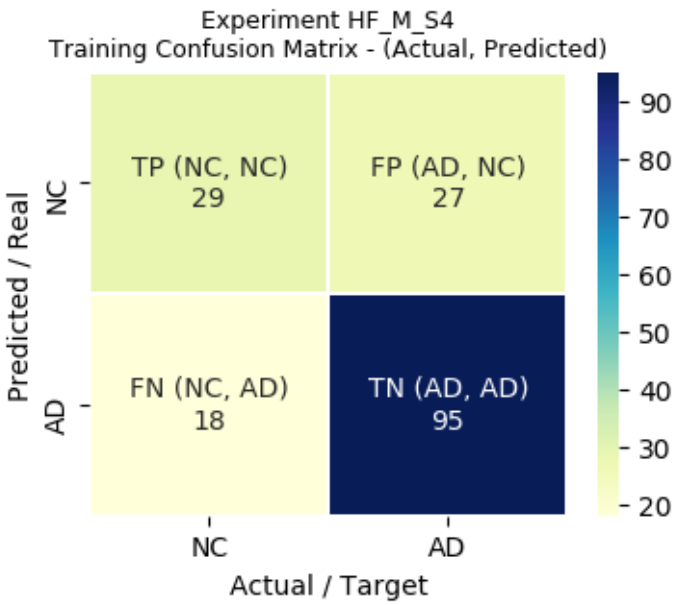


Figure G.98. Caption in the title.

G.2.4.2 Validation

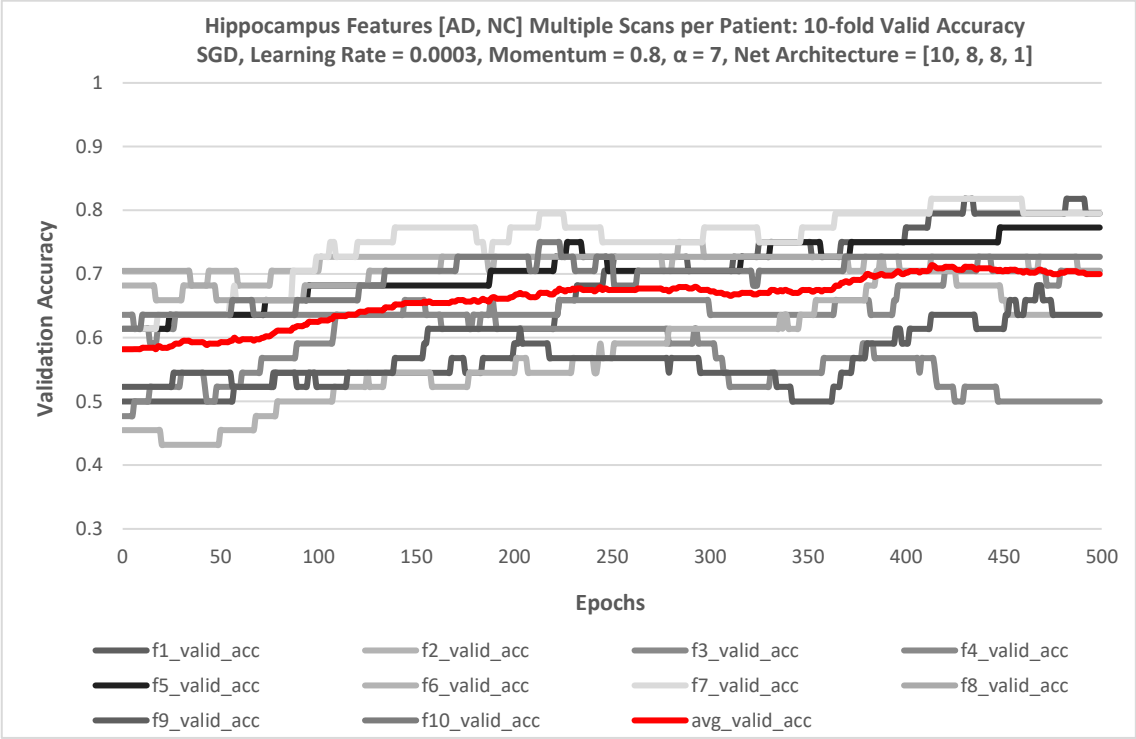


Figure G.99. Caption in the title.

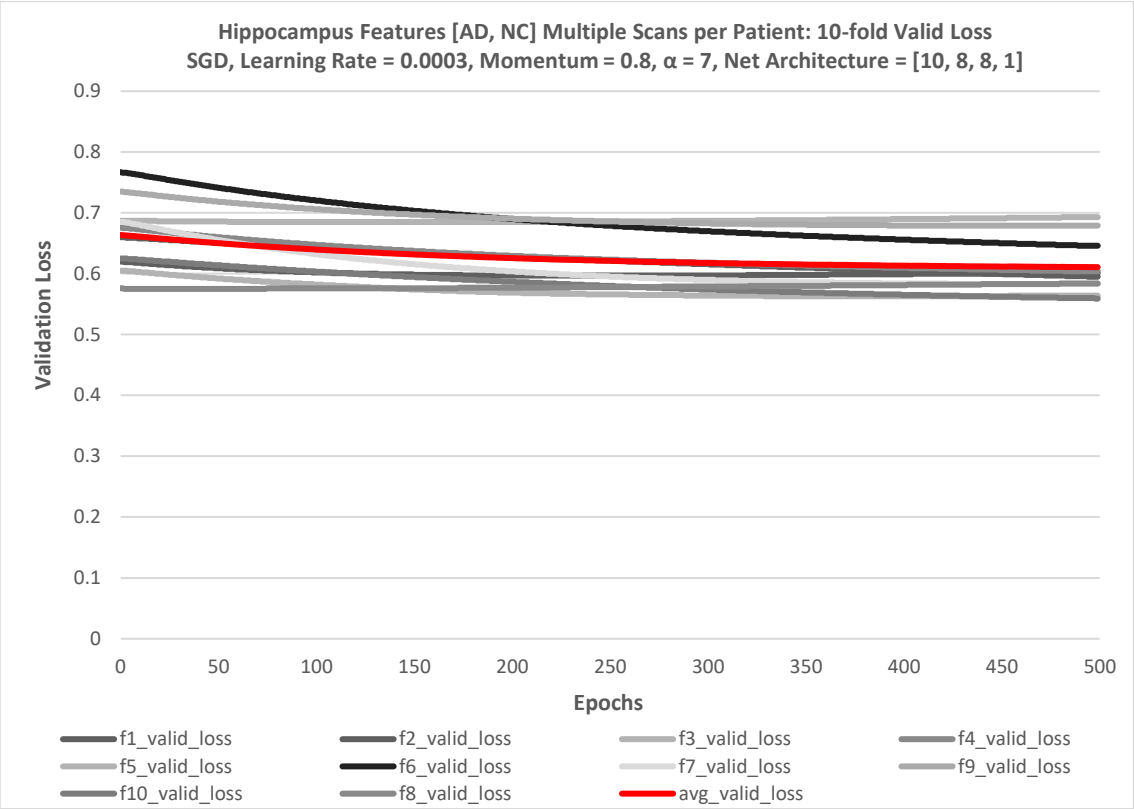


Figure G.100. Caption in the title.

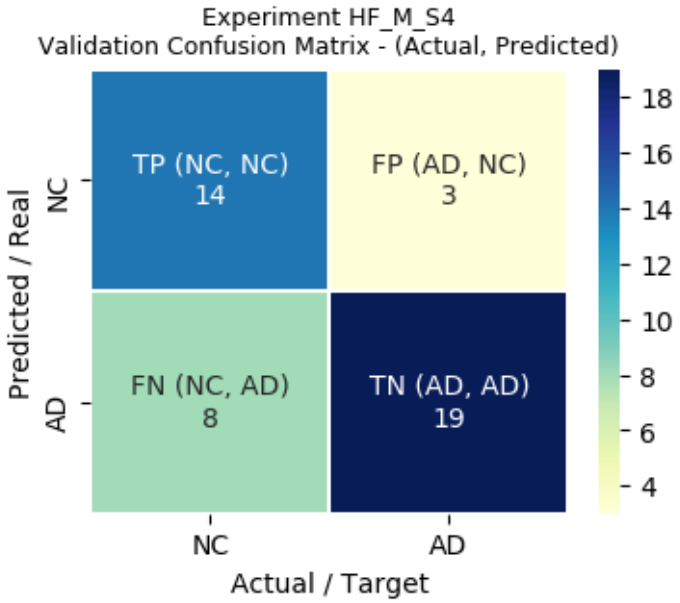


Figure G.101. Caption in the title.

G.2.5 Experiment HF_M_S5: α = 0.1, lr = 0.3, net = [10, 8, 8, 1]

G.2.5.1 Training

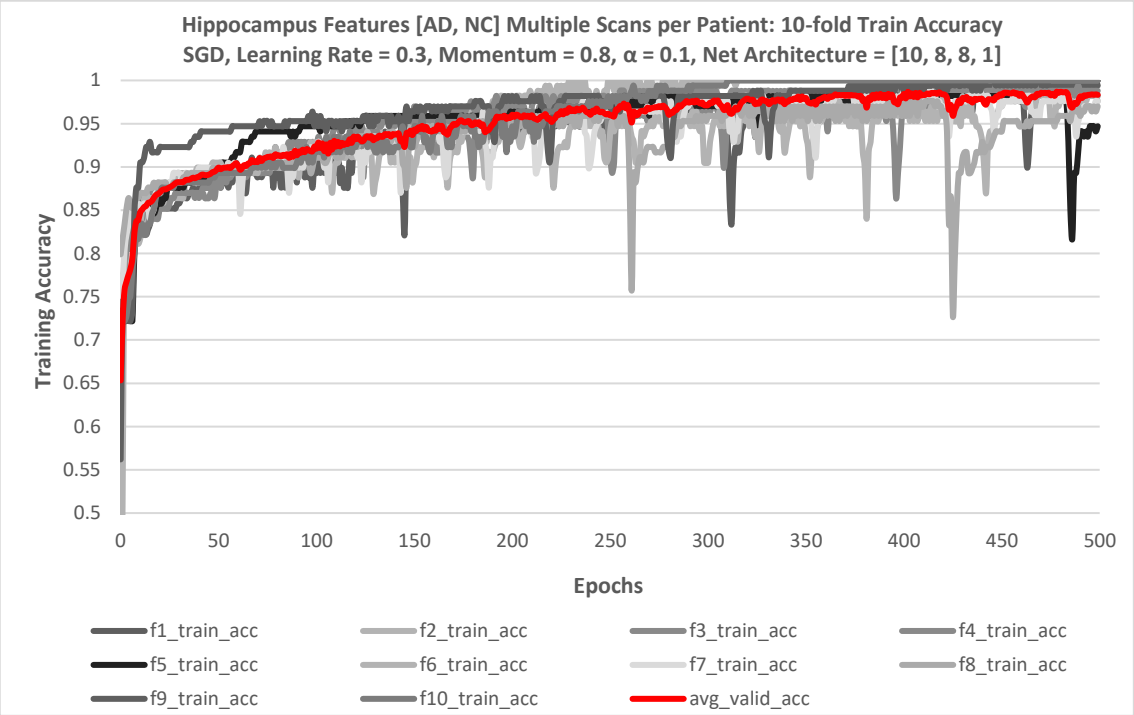


Figure G.102. Caption in the title.

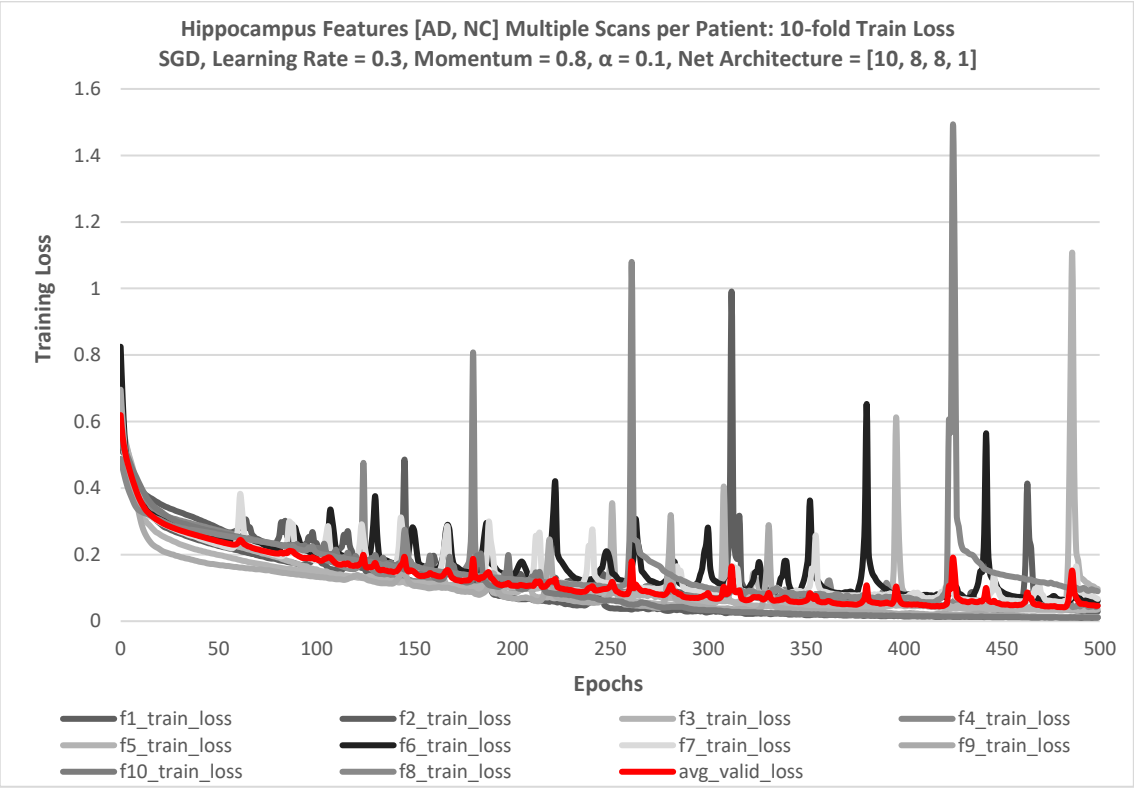


Figure G.103. Caption in the title.

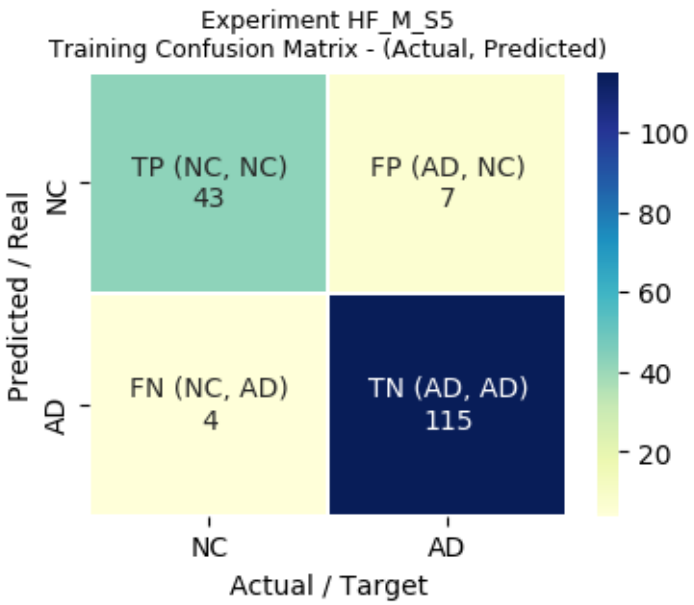


Figure G.104. Caption in the title.

G.2.5.2 Validation

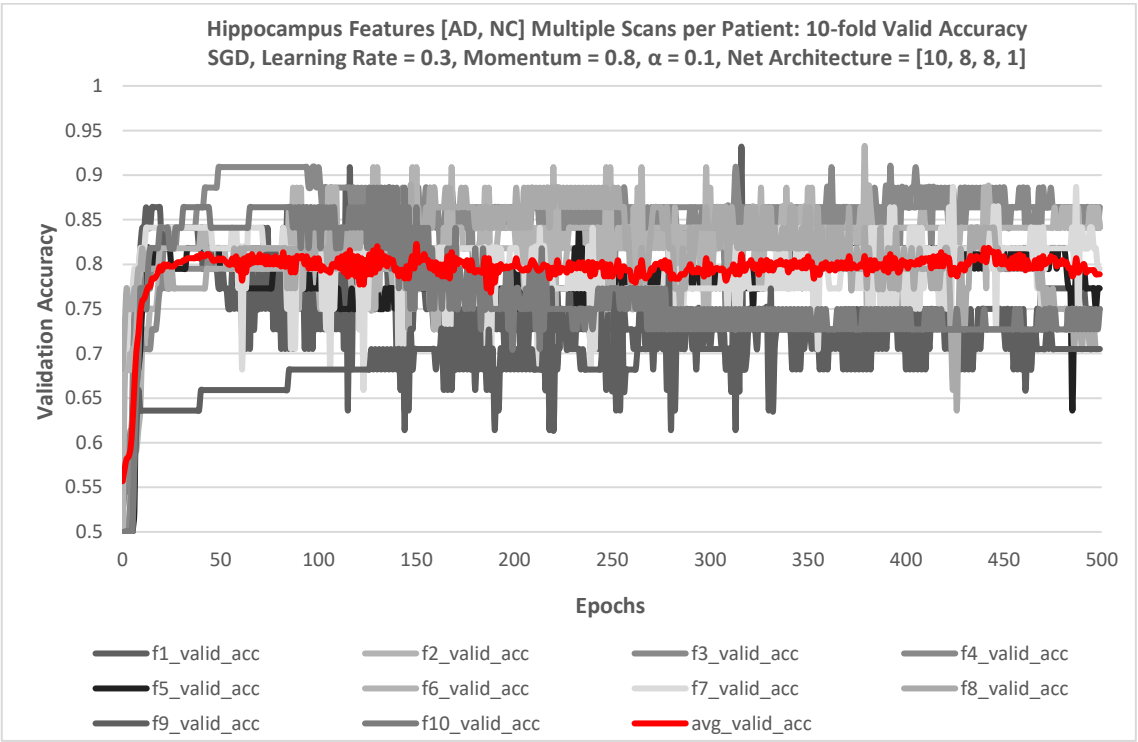


Figure G.105. Caption in the title.

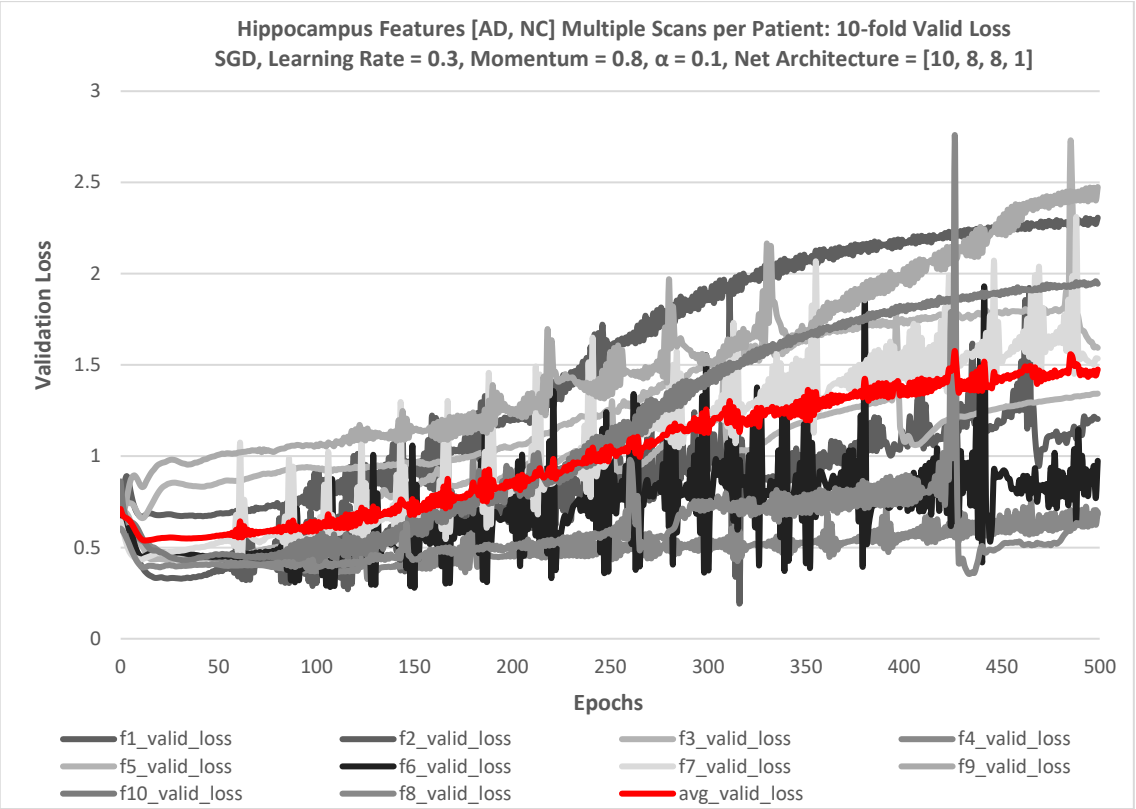


Figure G.106. Caption in the title.

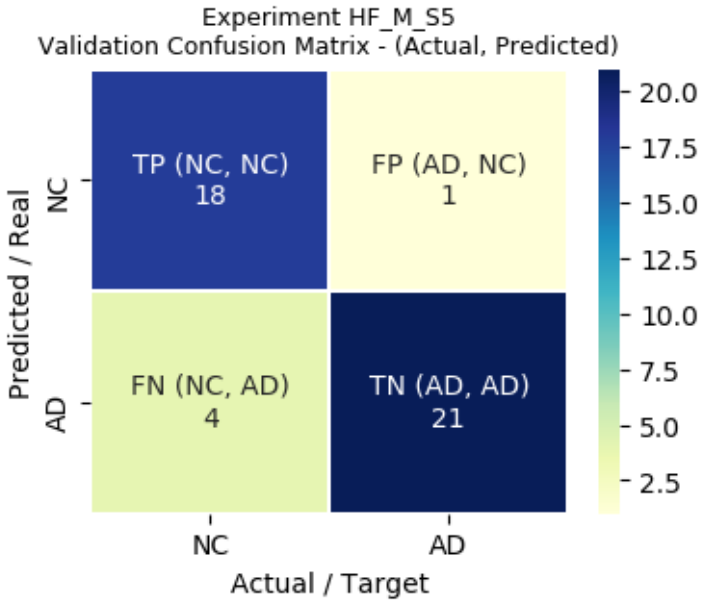


Figure G.107. Caption in the title.

G.3 MLP with HFO

G.3.1 Experiment HF_M_H1: CGiter = 1, net = [10, 30, 1]

G.3.1.1 Training

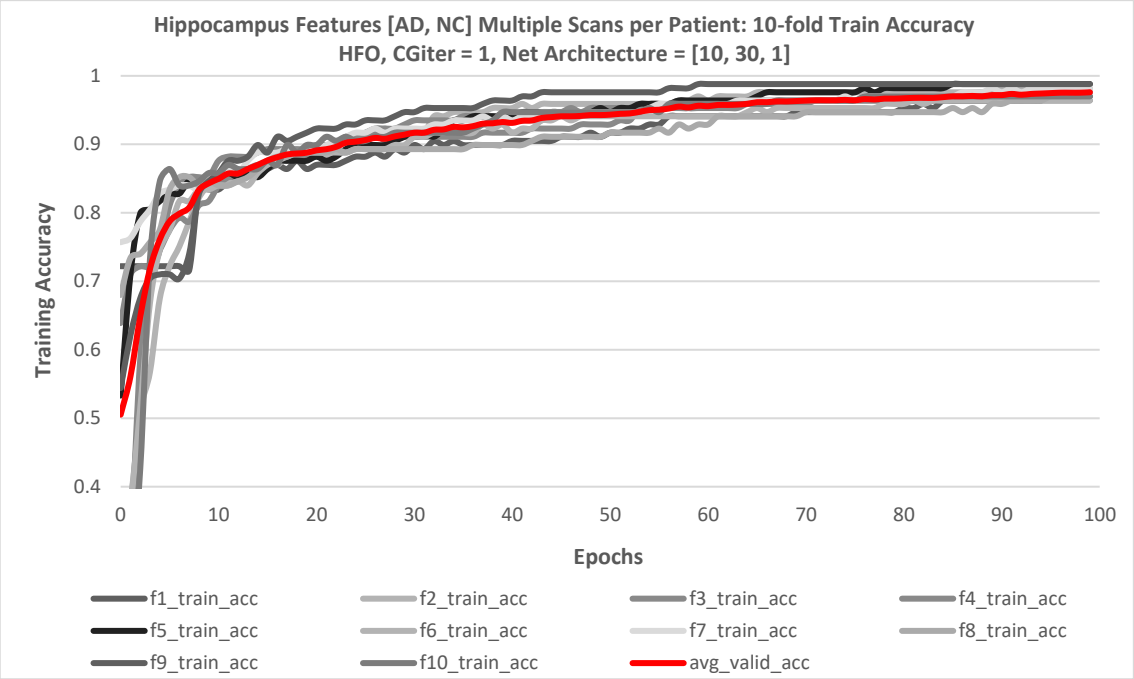


Figure G.108. Caption in the title.

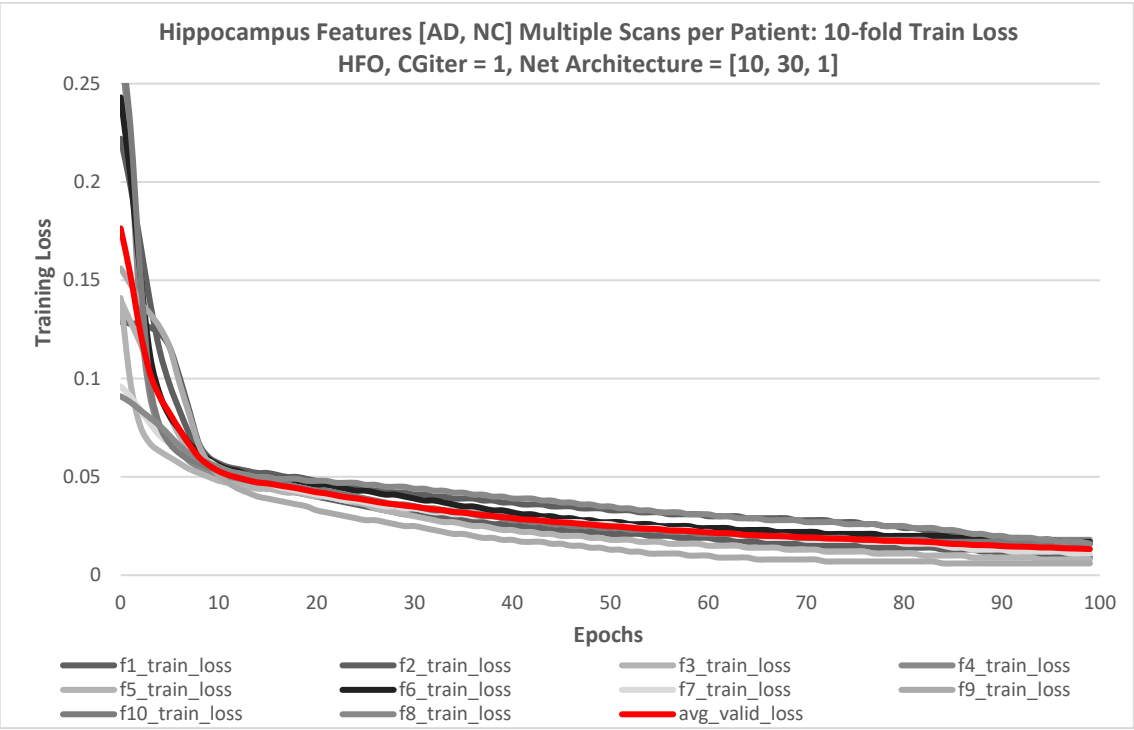


Figure G.109. Caption in the title.

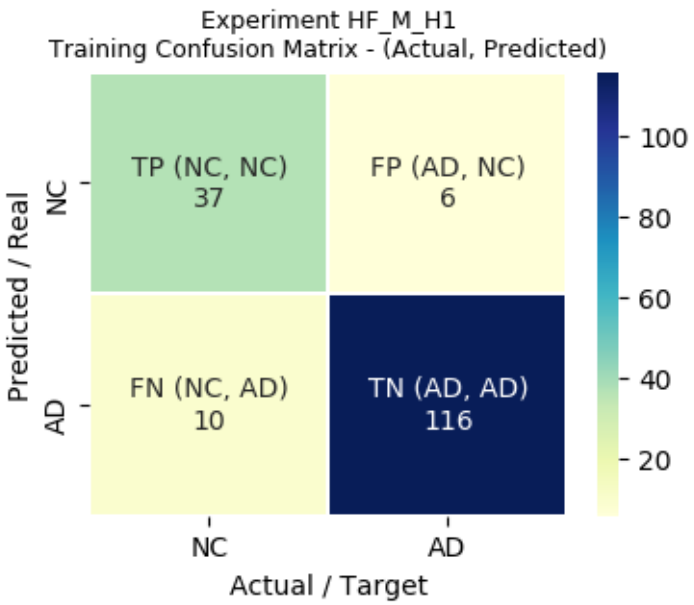


Figure G.110. Caption in the title.

G.3.1.2 Validation

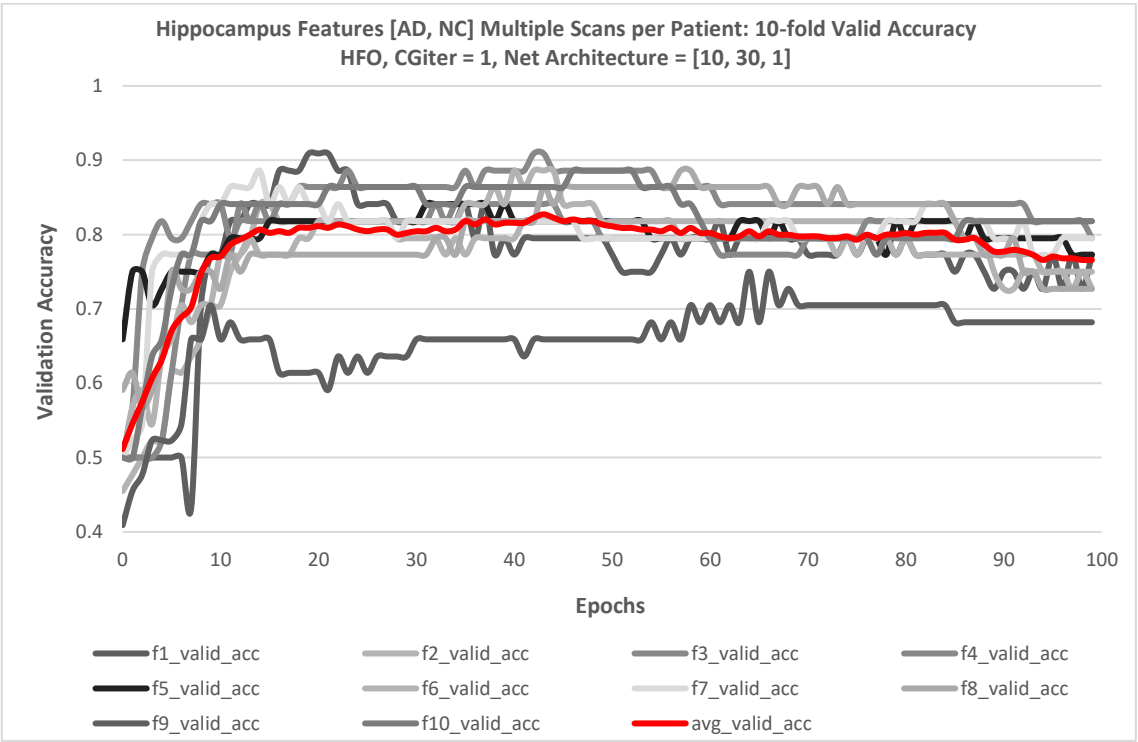


Figure G.111. Caption in the title.

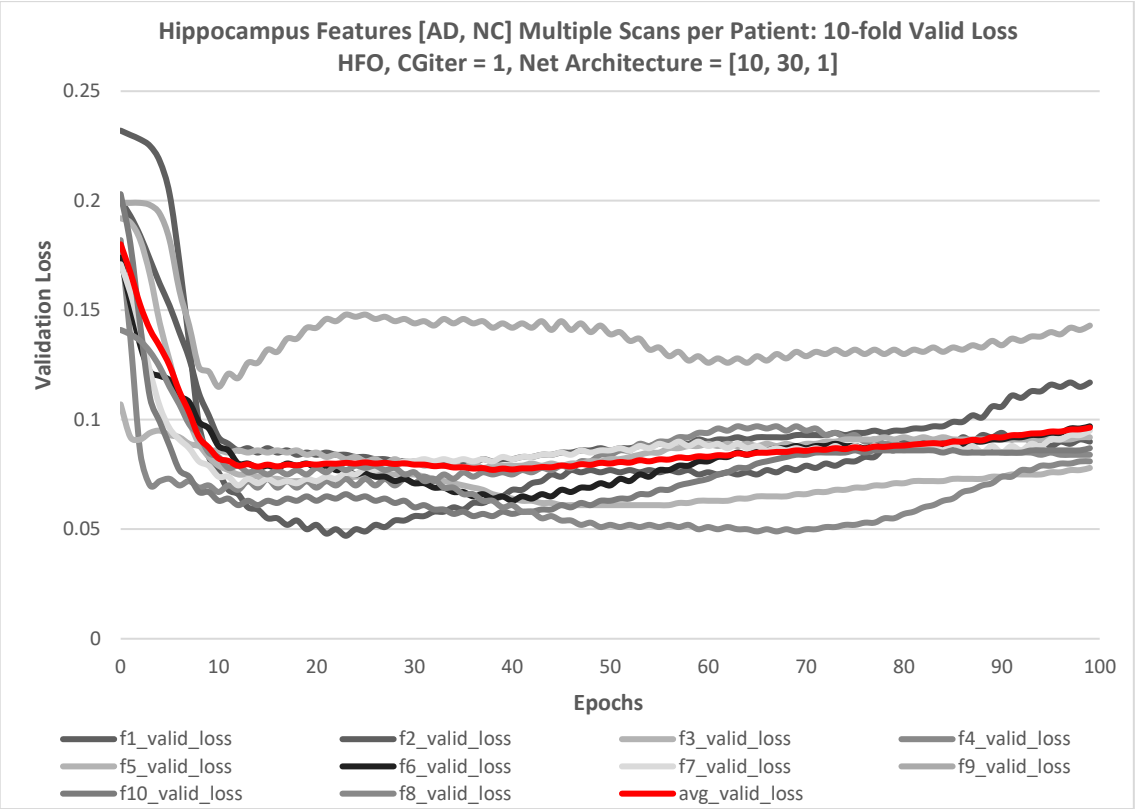


Figure G.112. Caption in the title.

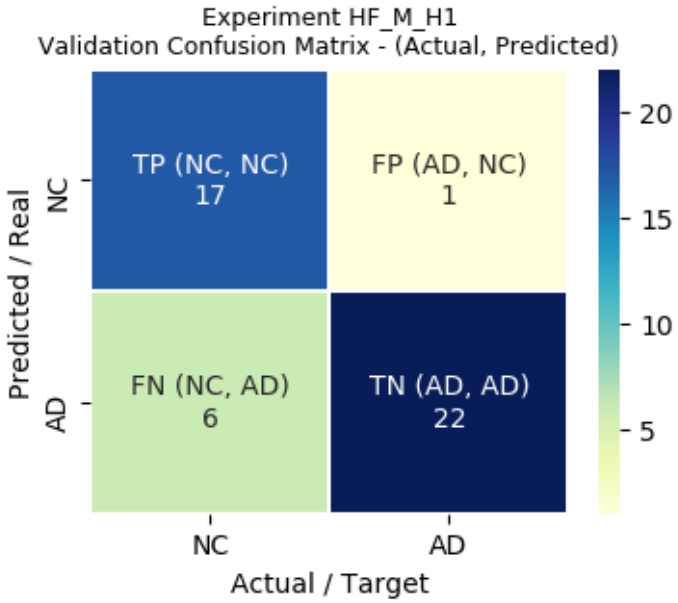


Figure G.113. Caption in the title.

G.3.2 Experiment HF_M_H2: CGiter = 2, net = [10, 30, 1]

G.3.2.1 Training

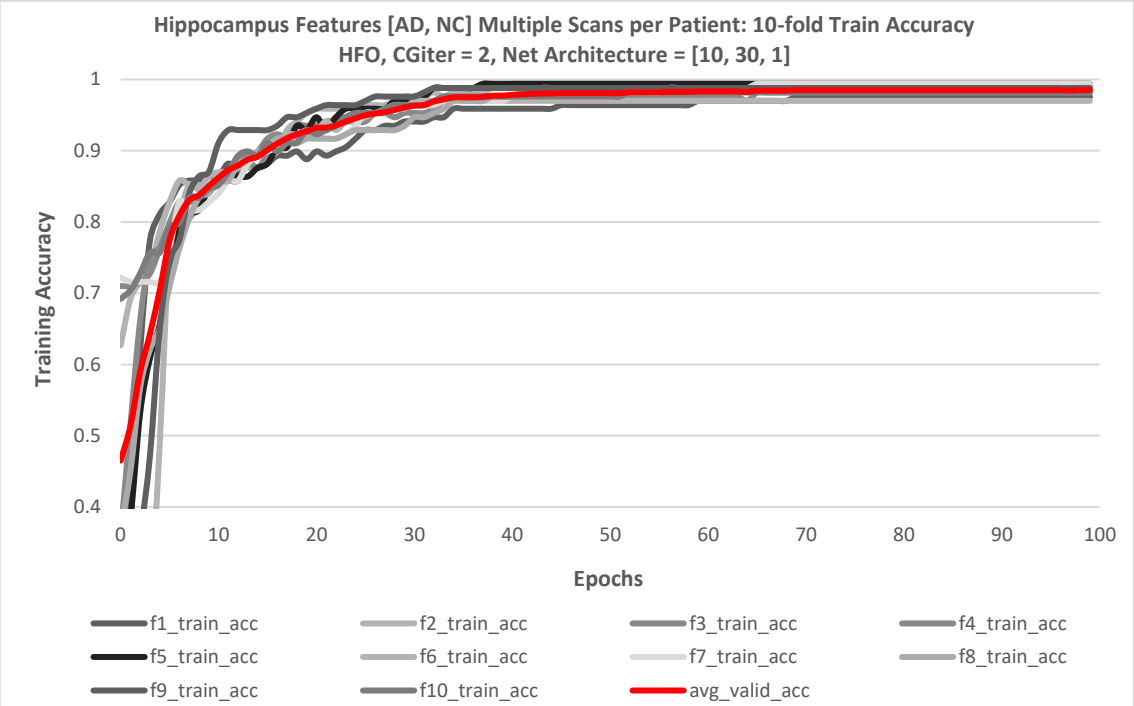


Figure G.114. Caption in the title.

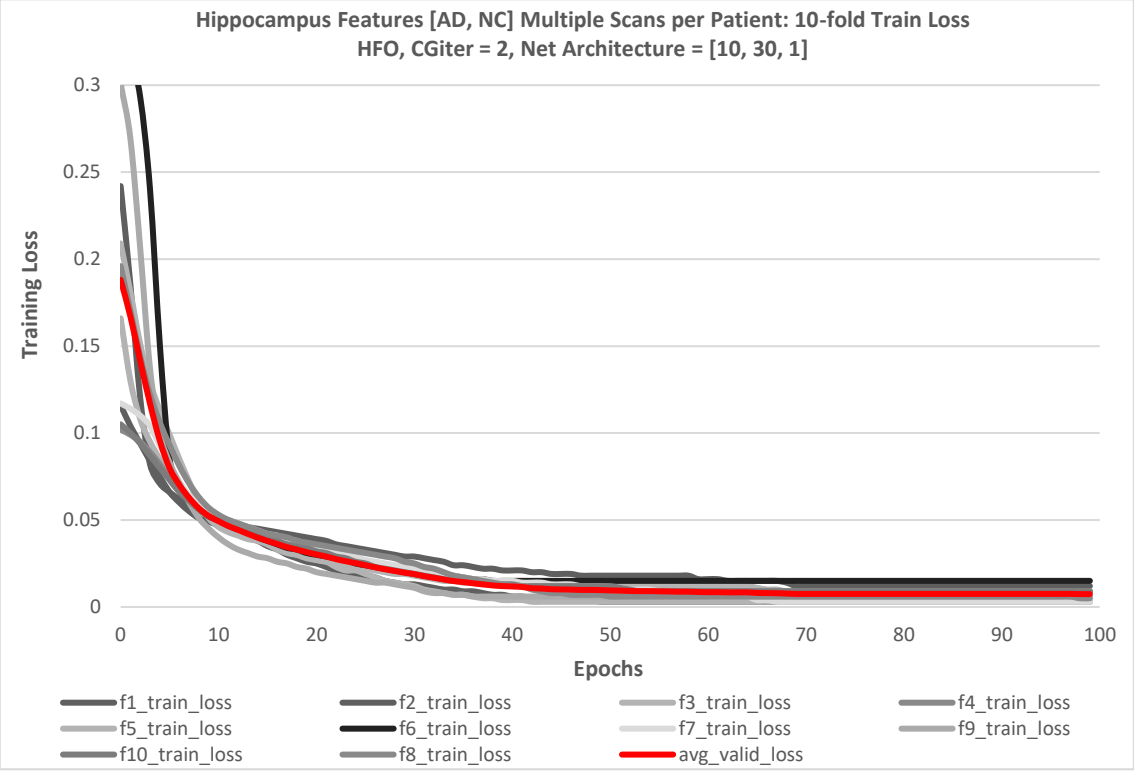


Figure G.115. Caption in the title.

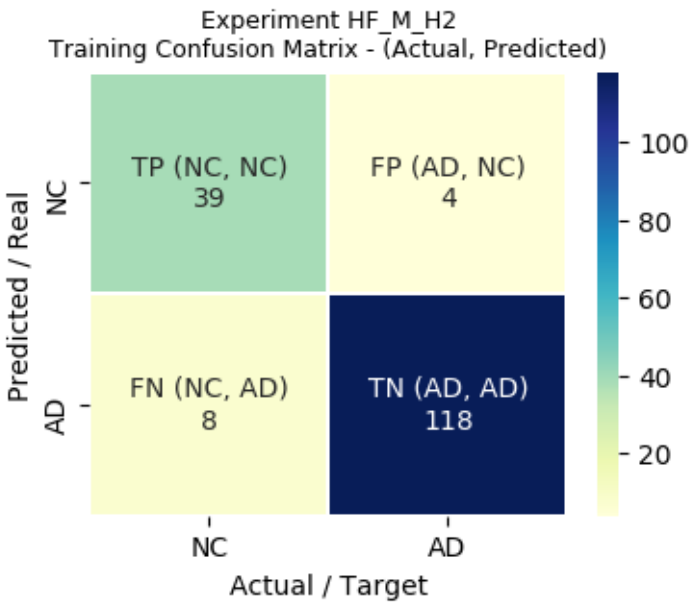


Figure G.116. Caption in the title.

G.3.2.2 Validation

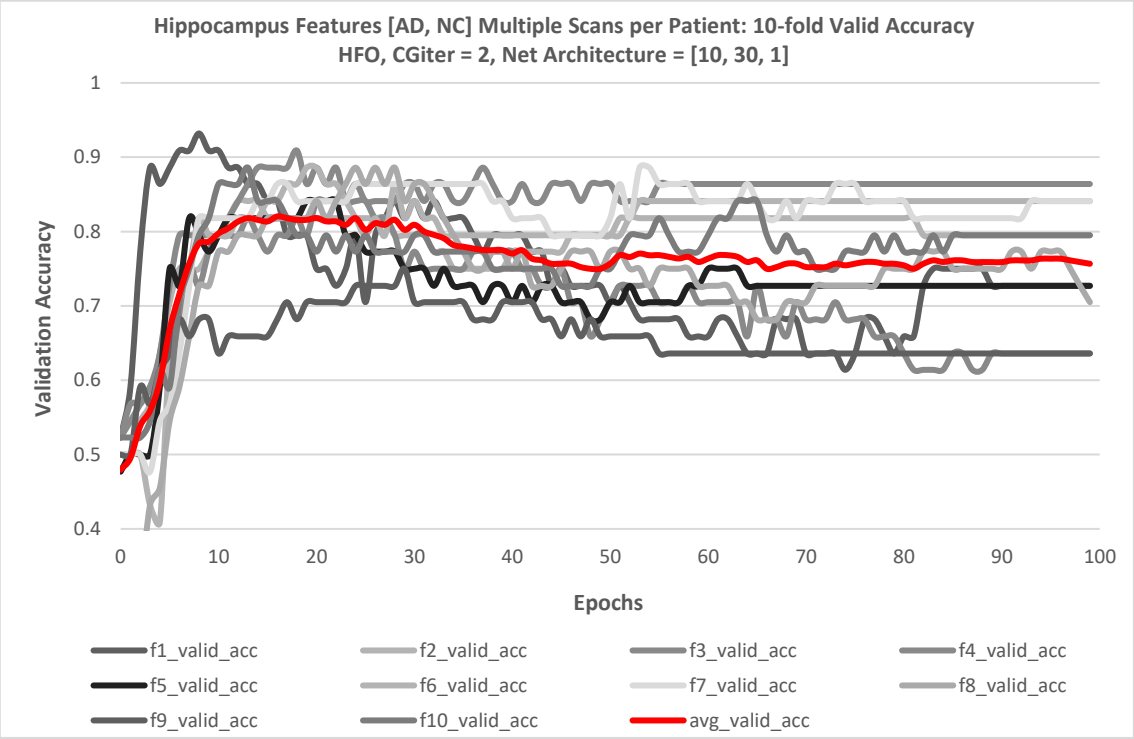


Figure G.117. Caption in the title.

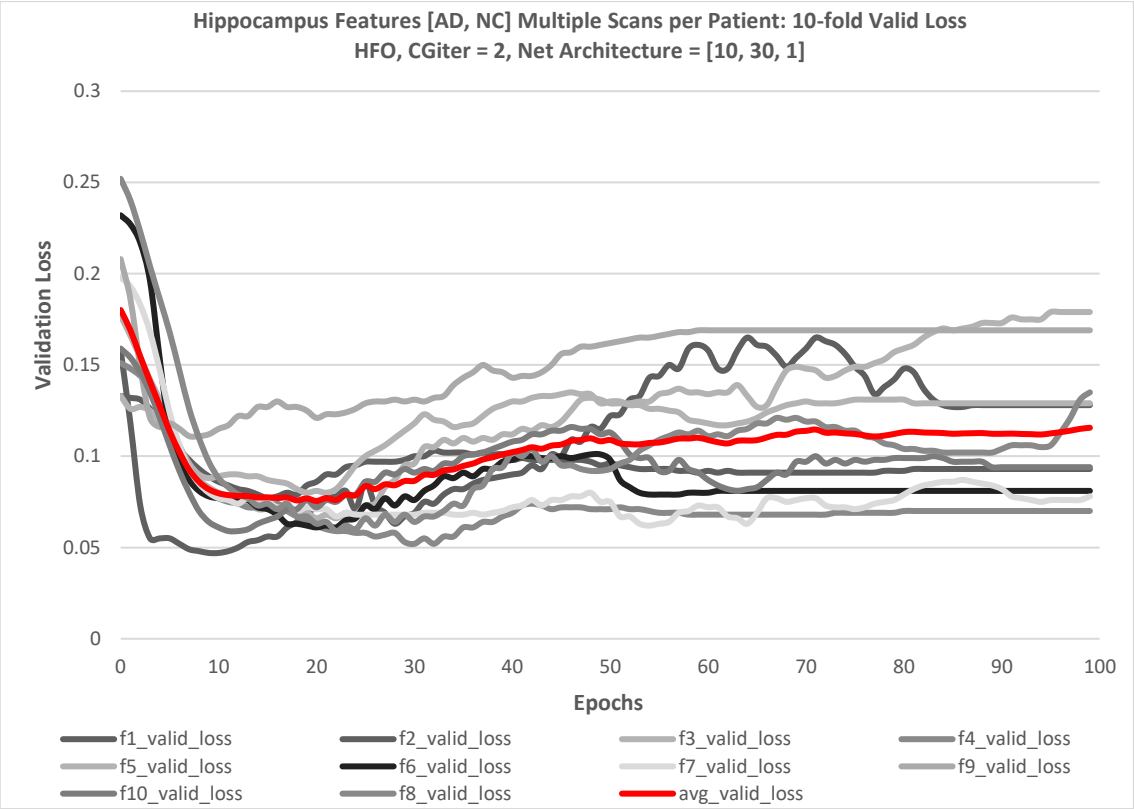


Figure G.118. Caption in the title.

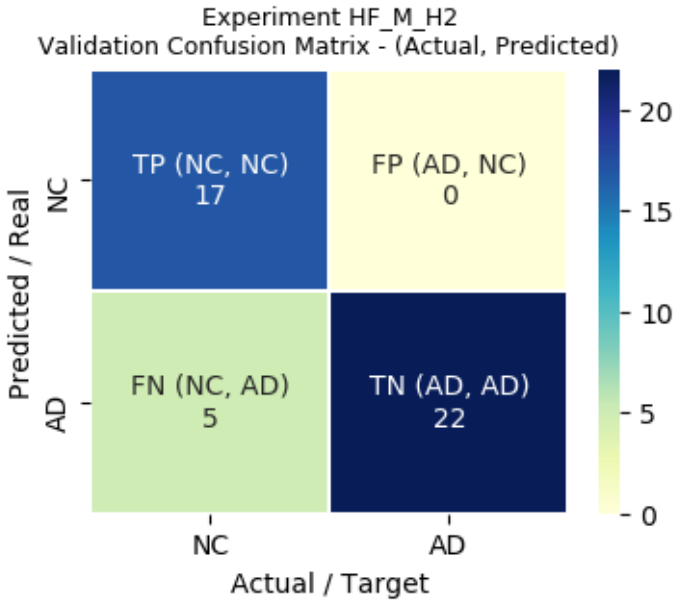


Figure G.119. Caption in the title.

G.3.3 Experiment HF_M_H3: CGiter = 4, net = [10, 30, 1]

G.3.3.1 Training

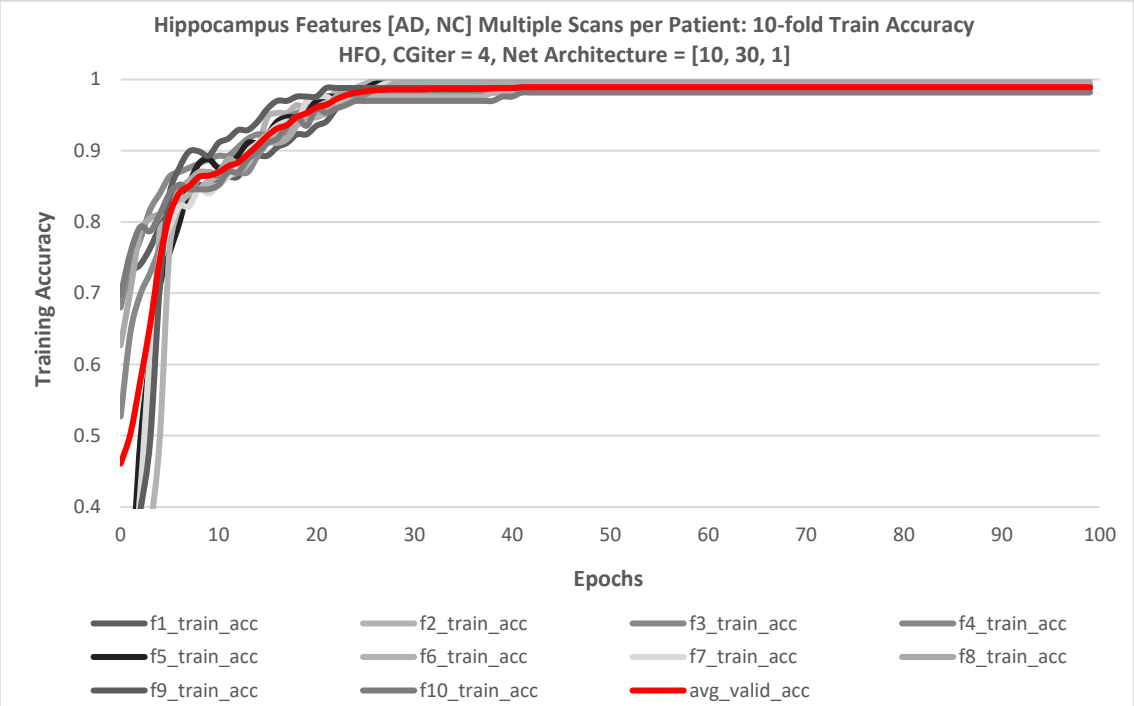


Figure G.120. Caption in the title.

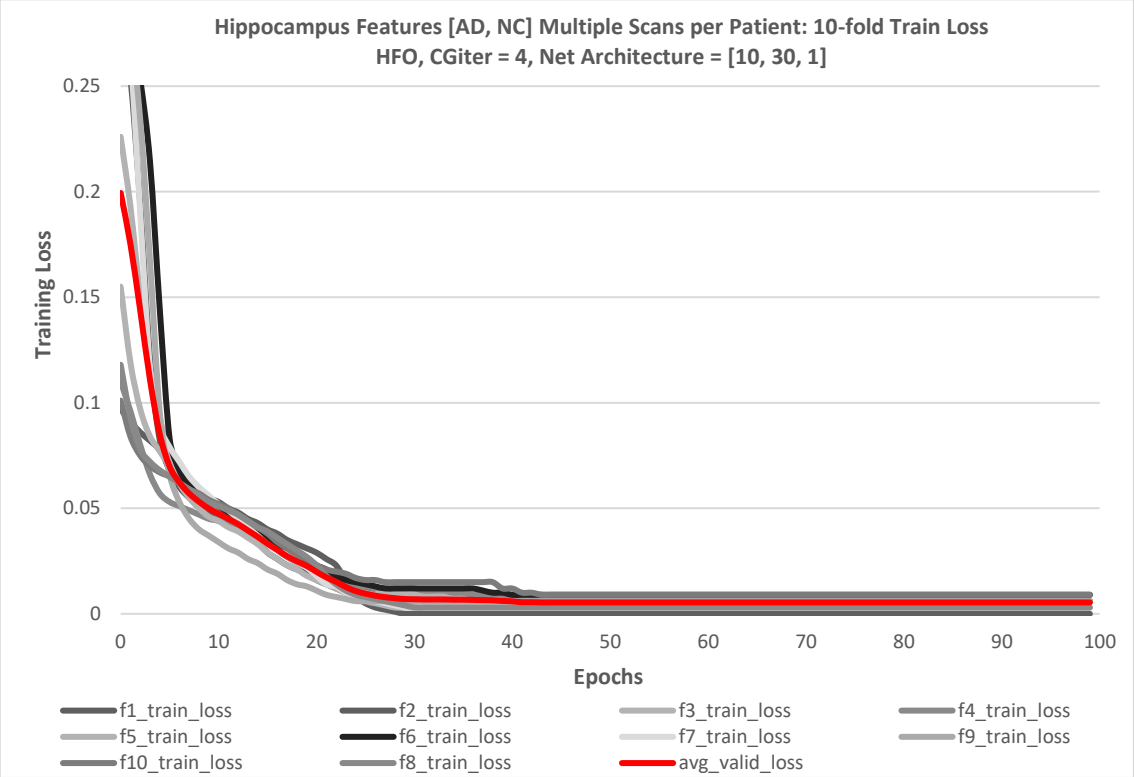


Figure G.121. Caption in the title.

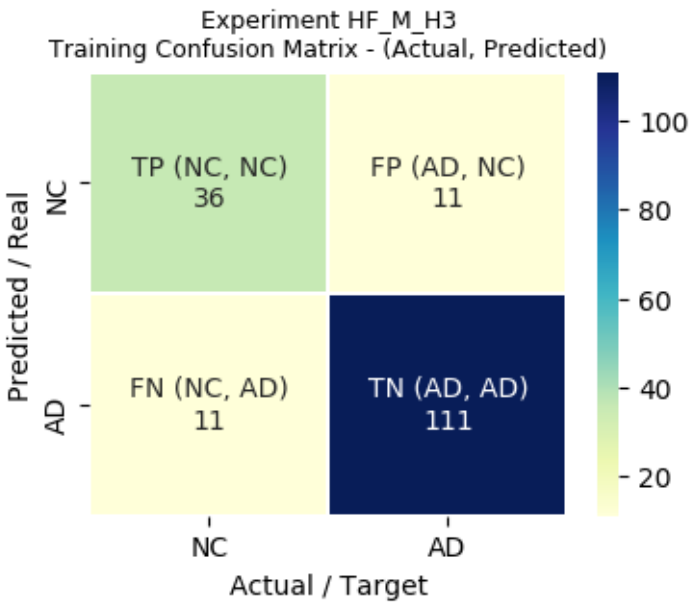


Figure G.122. Caption in the title.

G.3.3.2 Validation

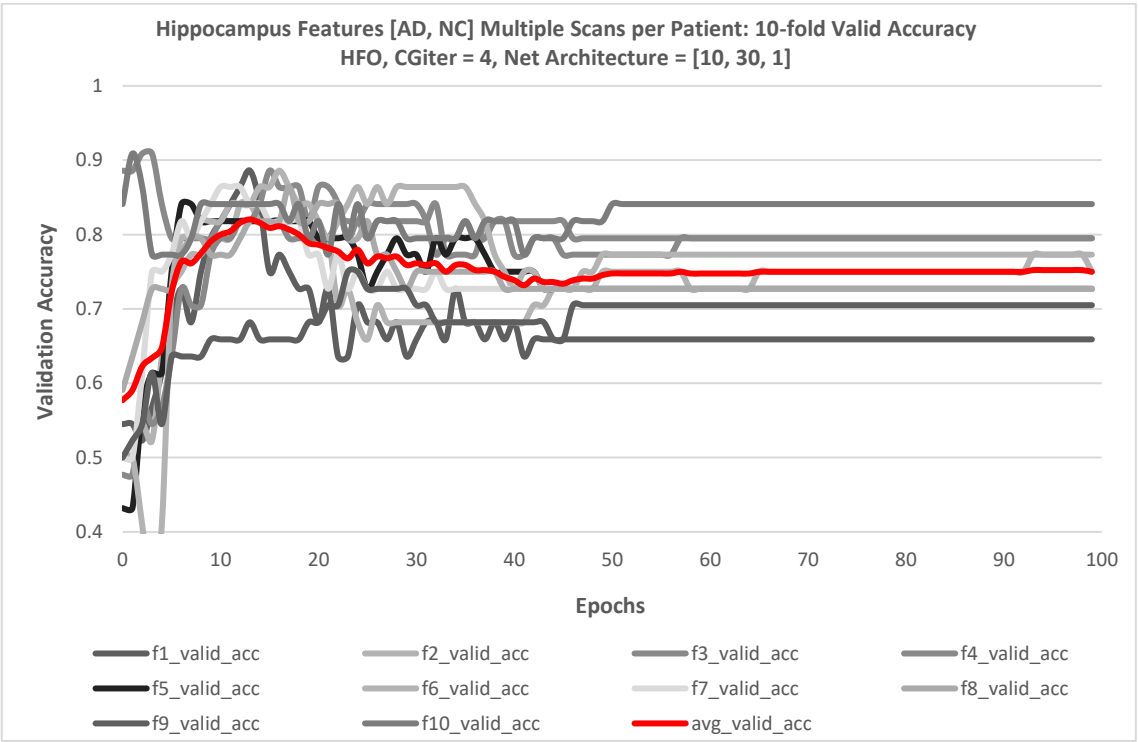


Figure G.123. Caption in the title.

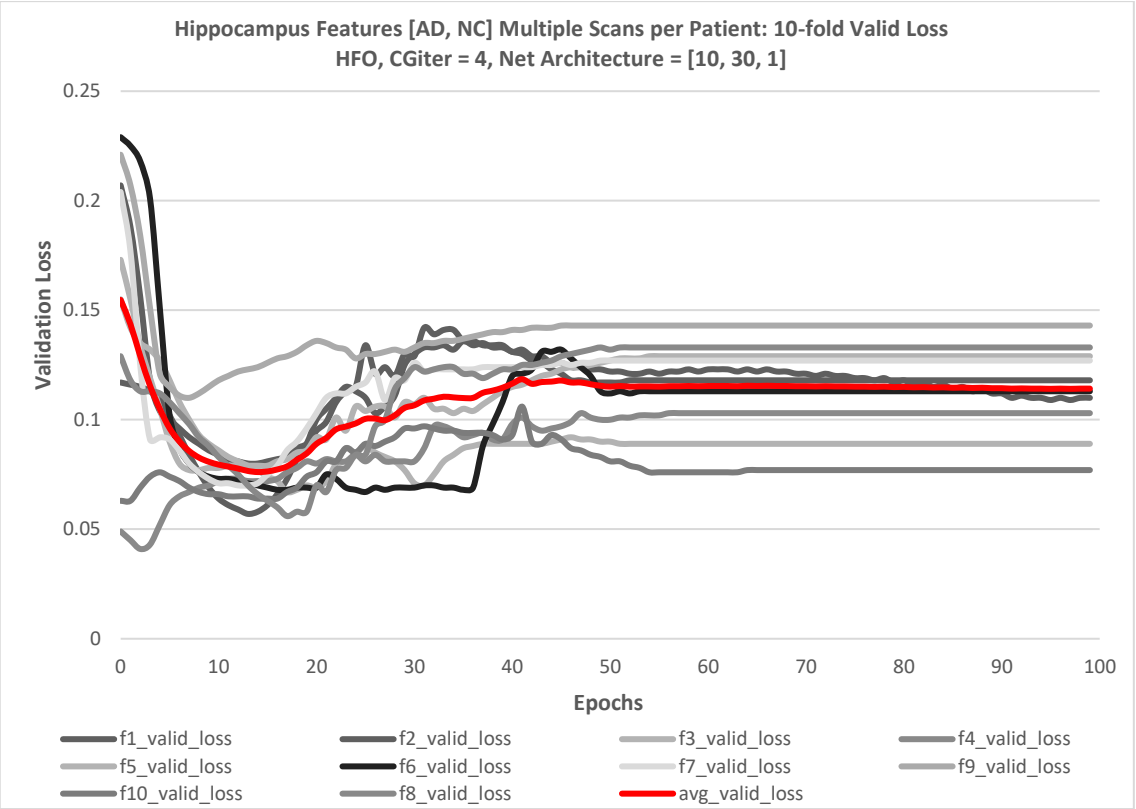


Figure G.124. Caption in the title.

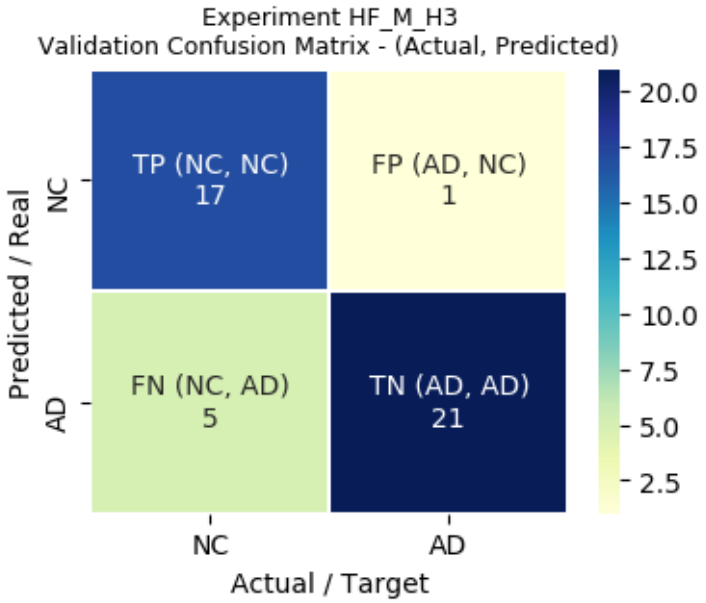


Figure G.125. Caption in the title.

G.3.4 Experiment HF_M_H4: CGiter = 8, net = [10, 30, 1]

G.3.4.1 Training

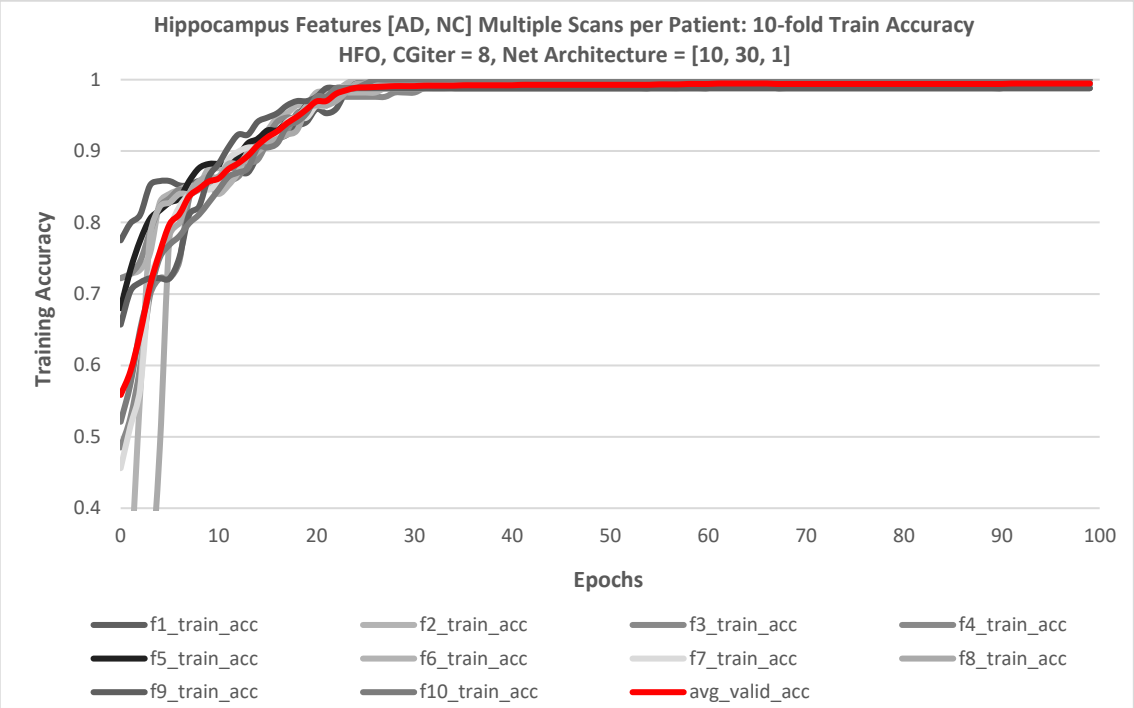


Figure G.126. Caption in the title.

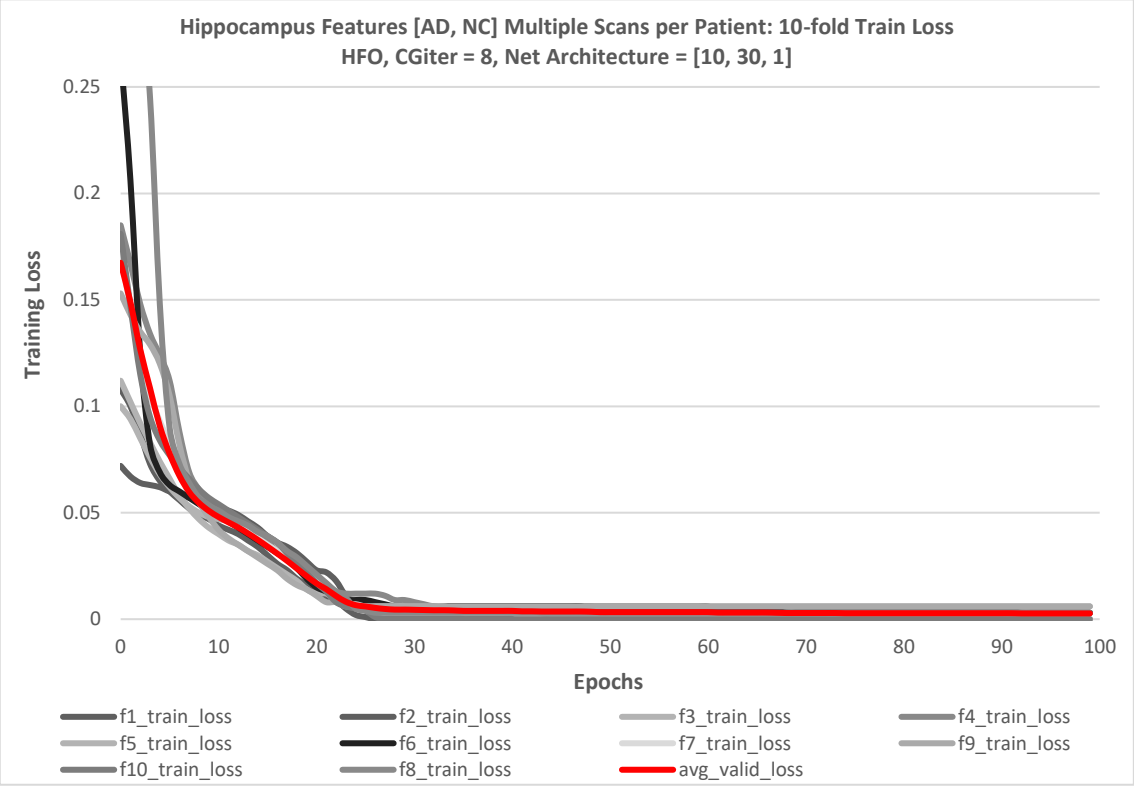


Figure G.127. Caption in the title.

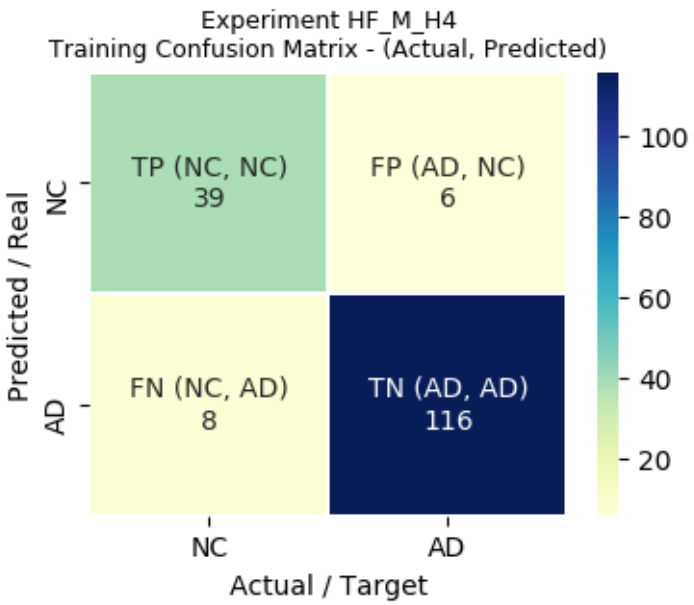


Figure G.128. Caption in the title.

G.3.4.2 Validation

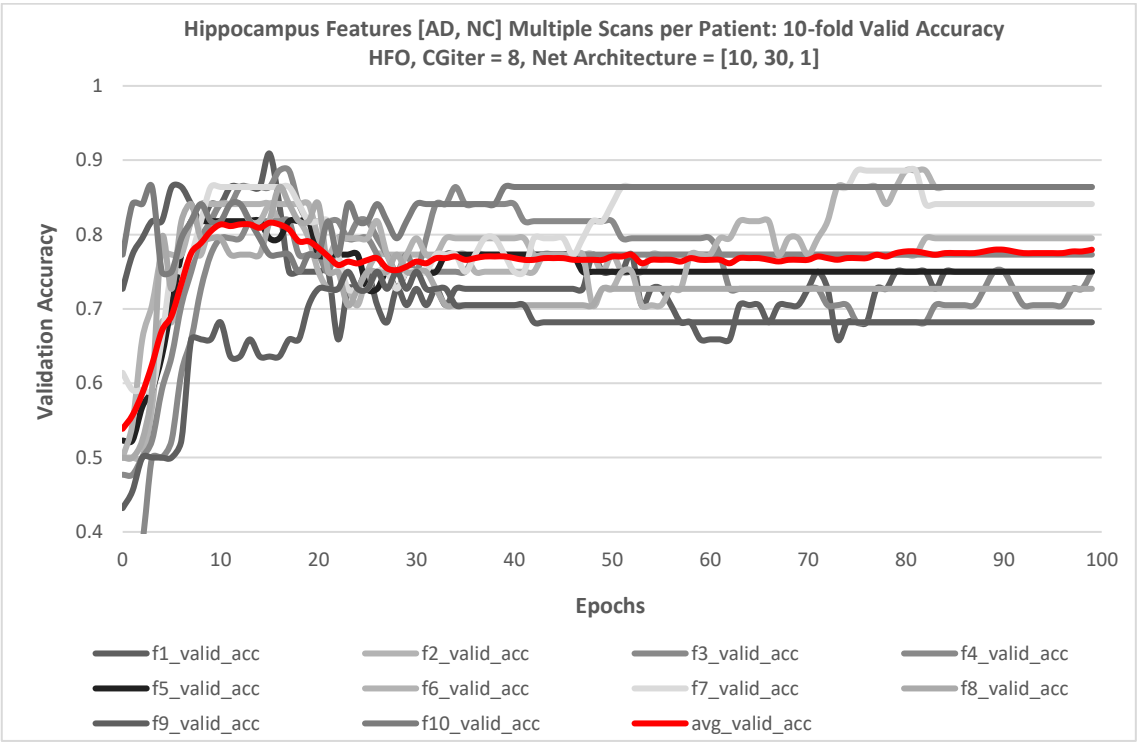


Figure G.129. Caption in the title.

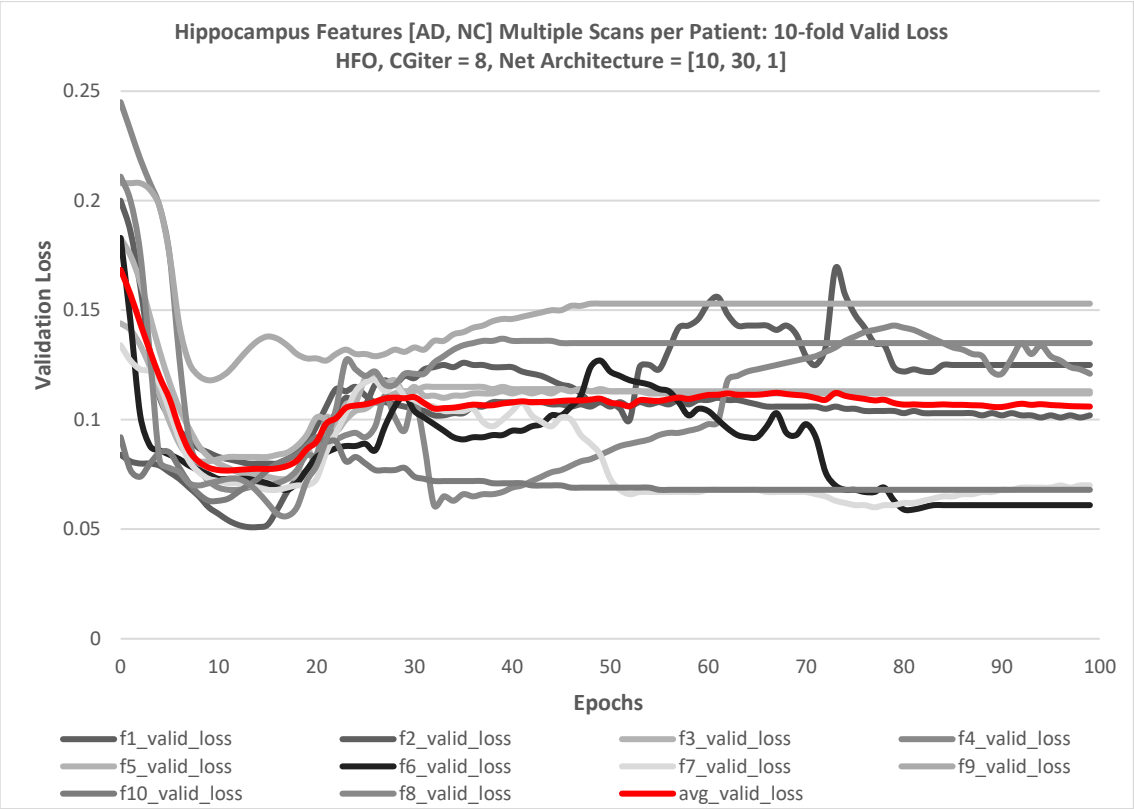


Figure G.130. Caption in the title.

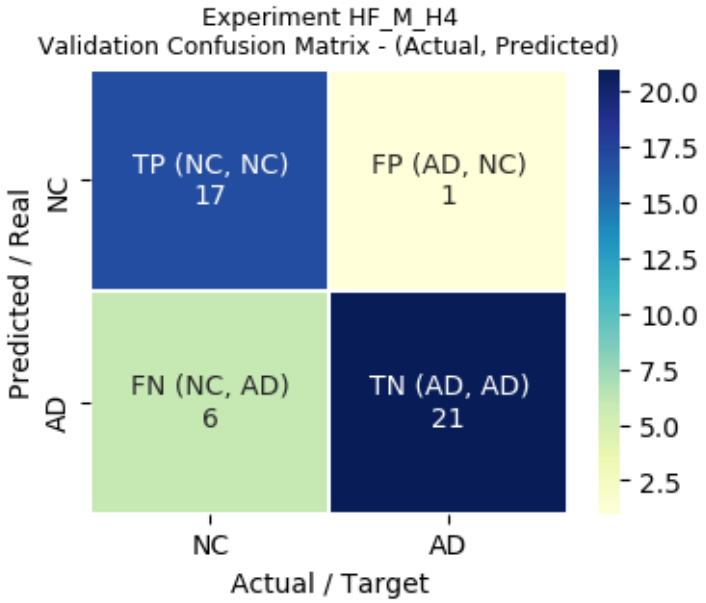


Figure G.131. Caption in the title.

G.3.5 Experiment HF_M_H5: CGiter = 16, net = [10, 30, 1]

G.3.5.1 Training

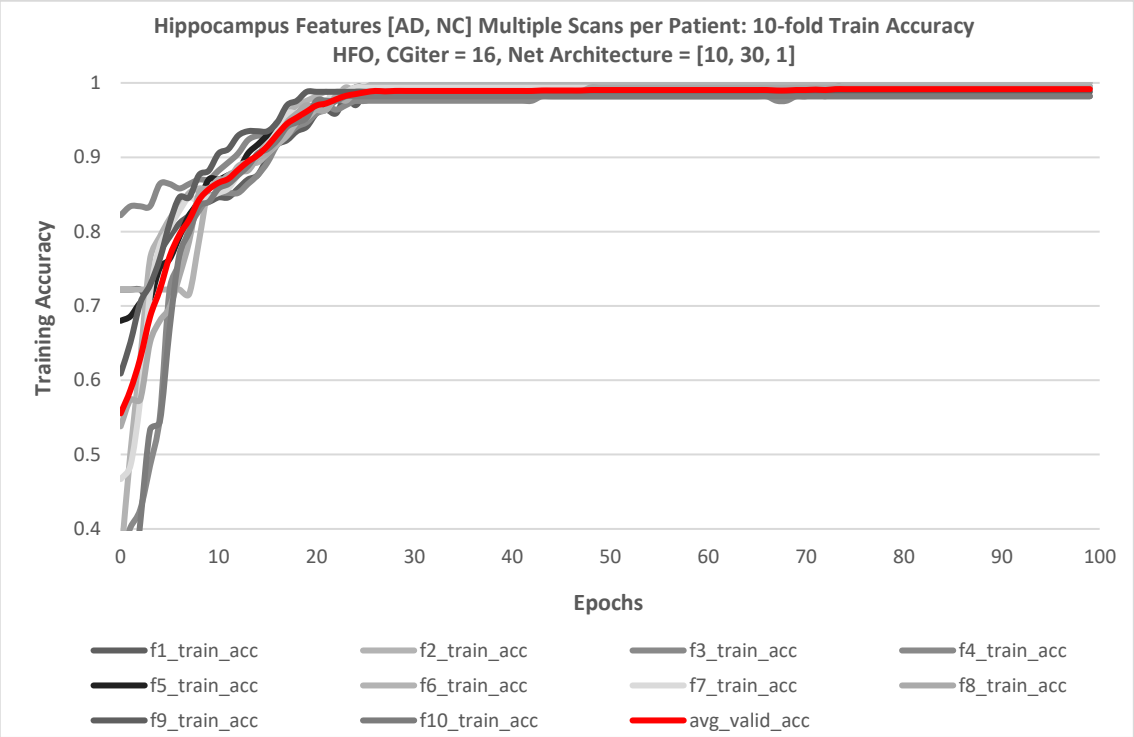


Figure G.132. Caption in the title.

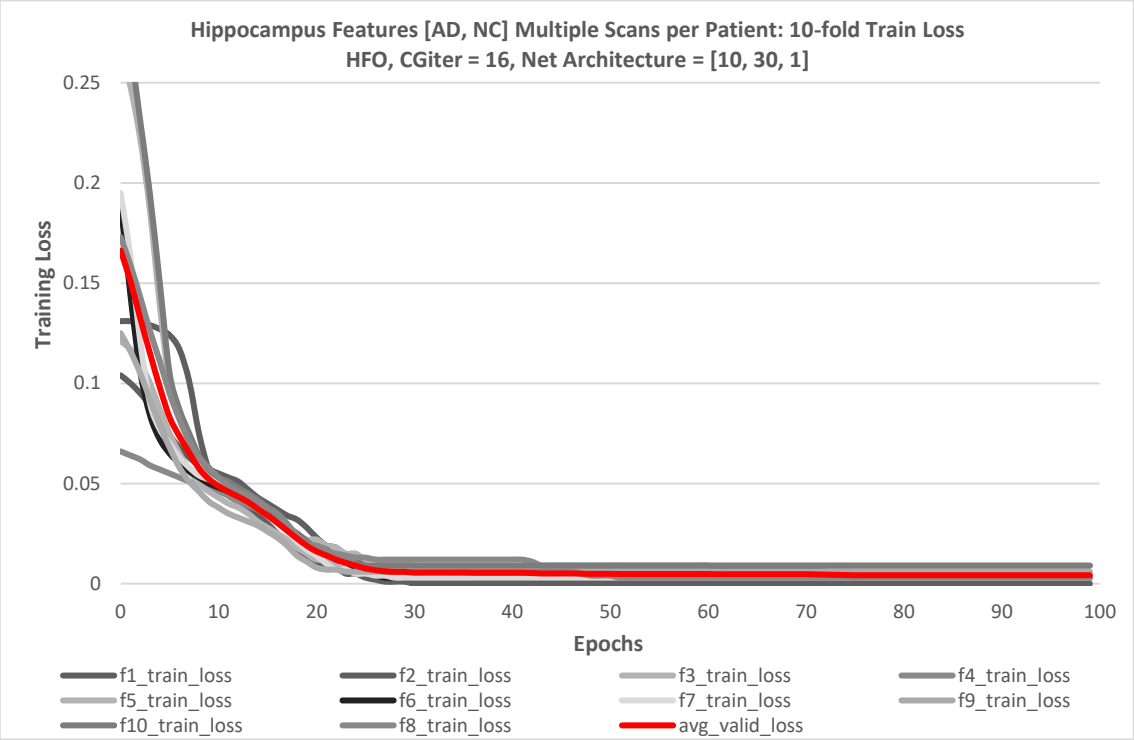


Figure G.133. Caption in the title.

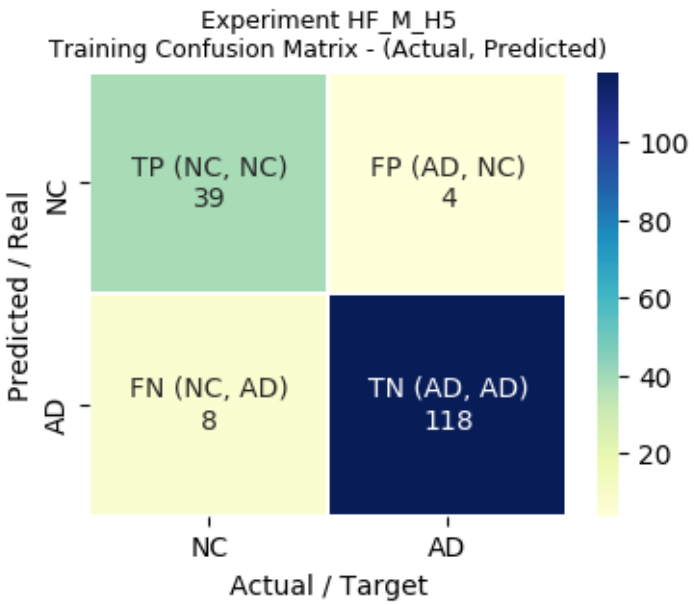


Figure G.134. Caption in the title.

G.3.5.2 Validation

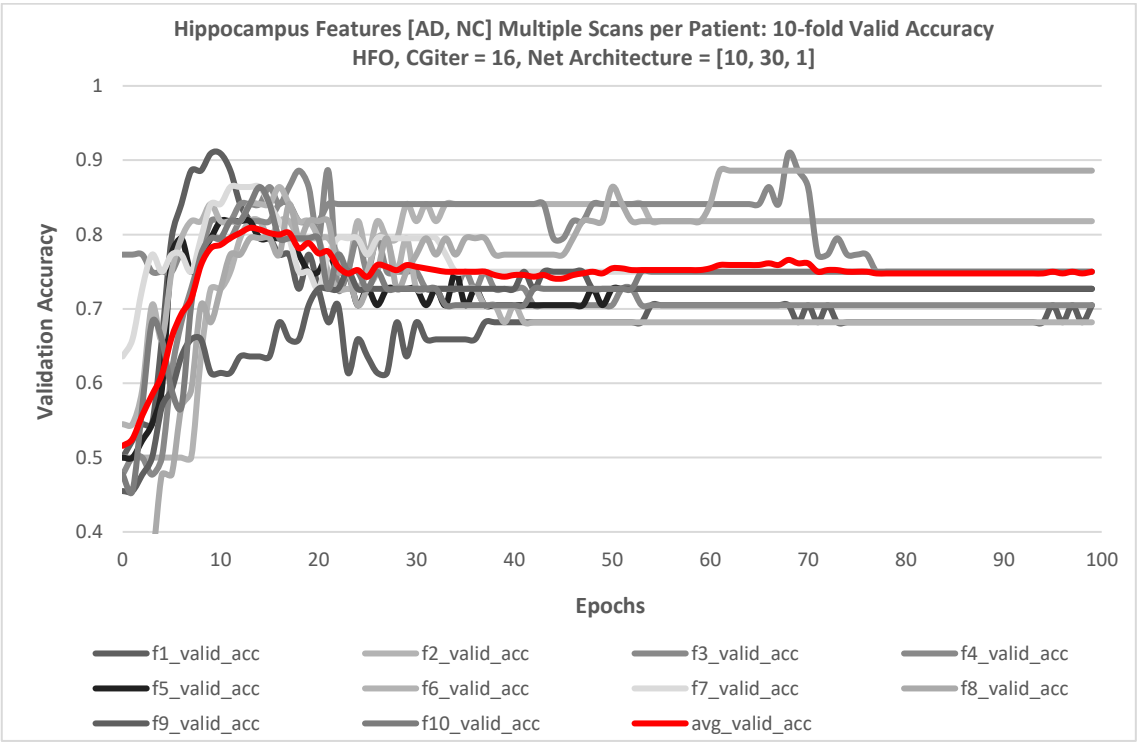


Figure G.135. Caption in the title.

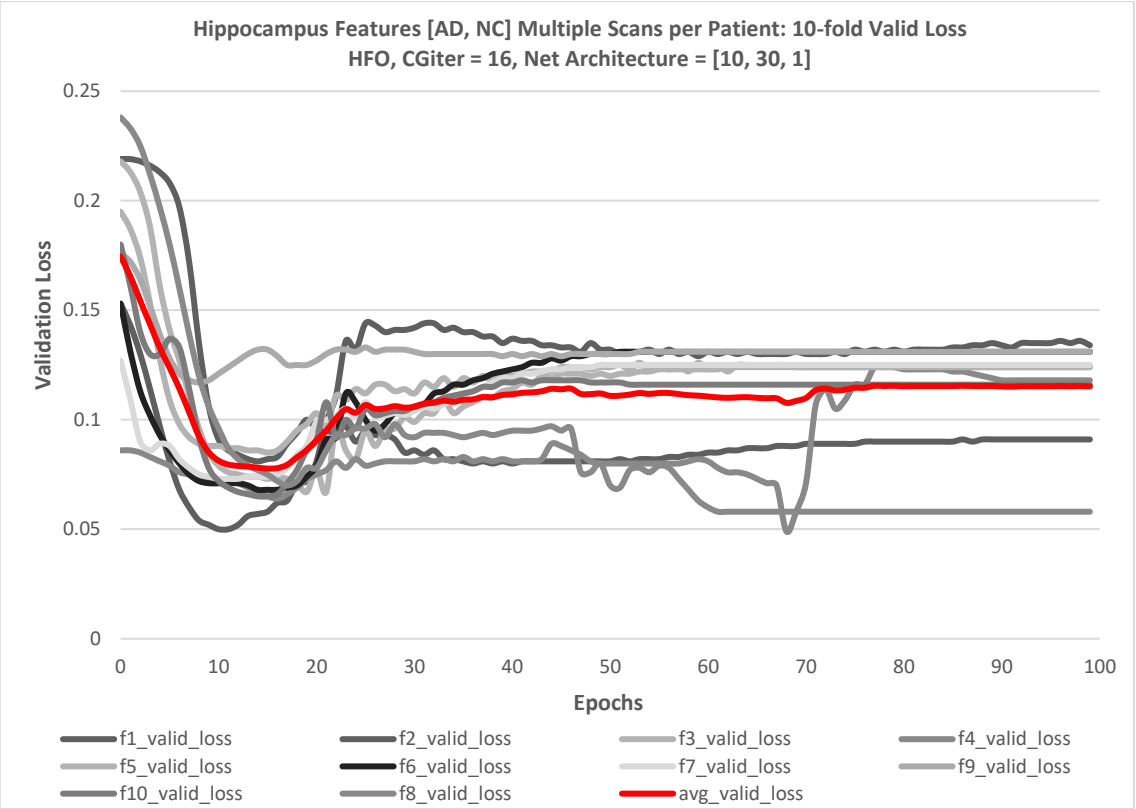


Figure G.136. Caption in the title.

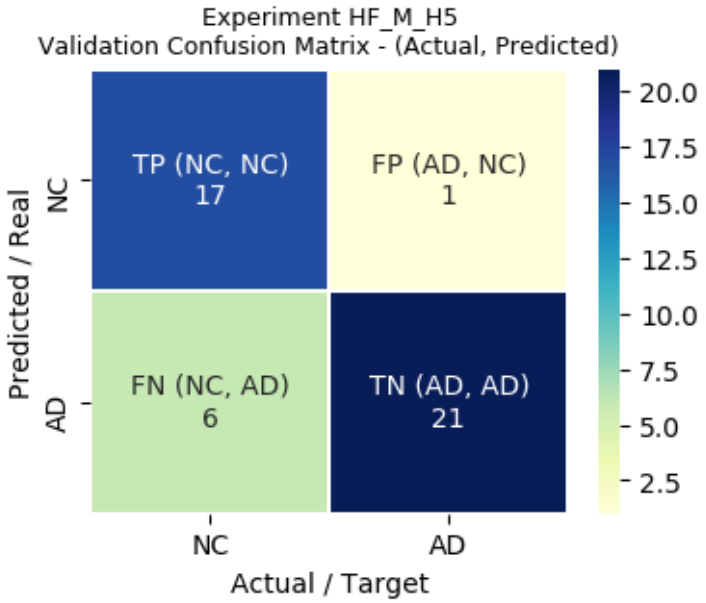


Figure G.137. Caption in the title.

G.3.6 Experiment HF_M_H6: CGiter = 32, net = [10, 30, 1]

G.3.6.1 Training

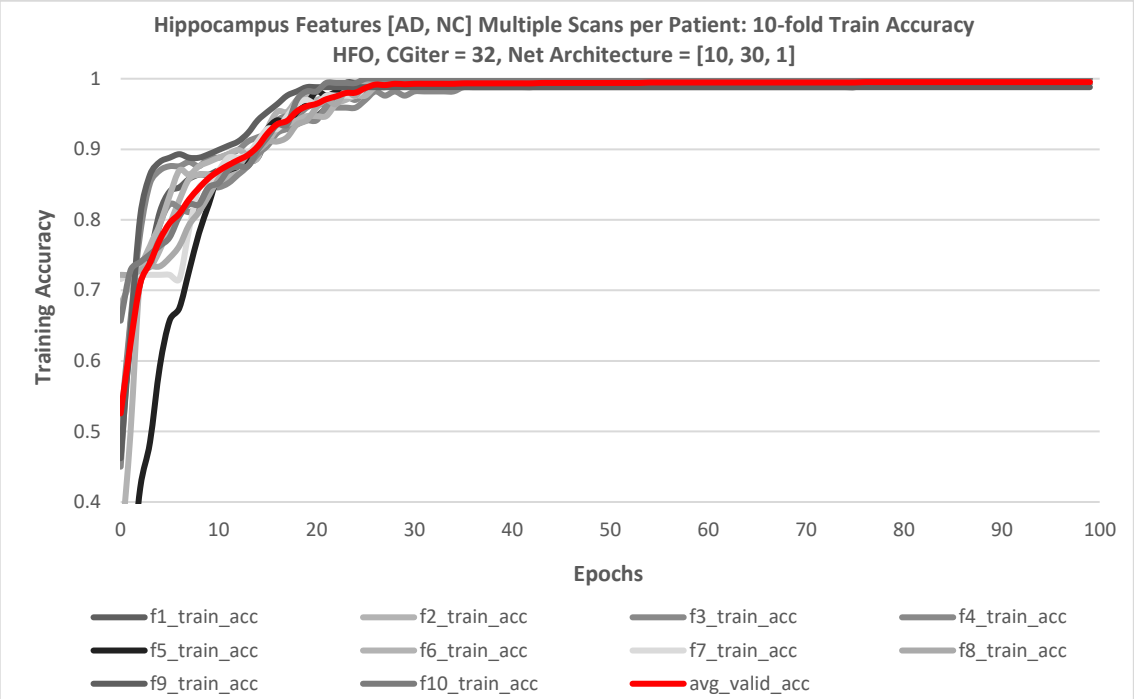


Figure G.138. Caption in the title.

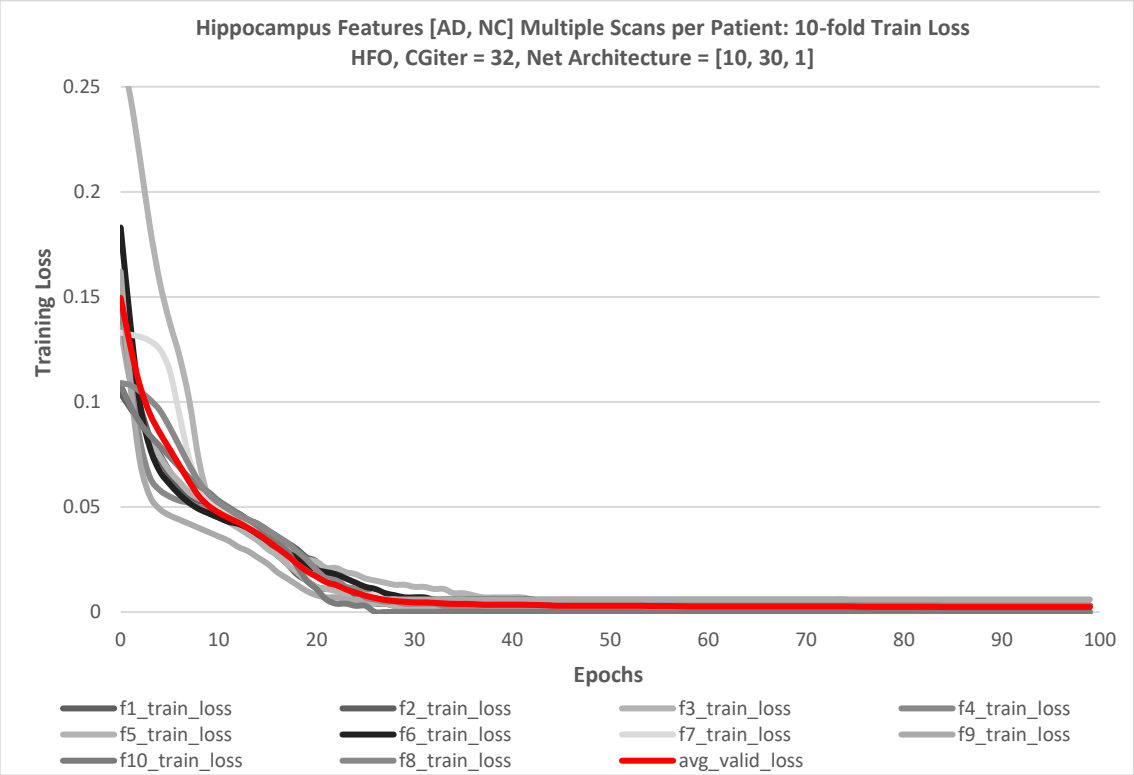


Figure G.139. Caption in the title.

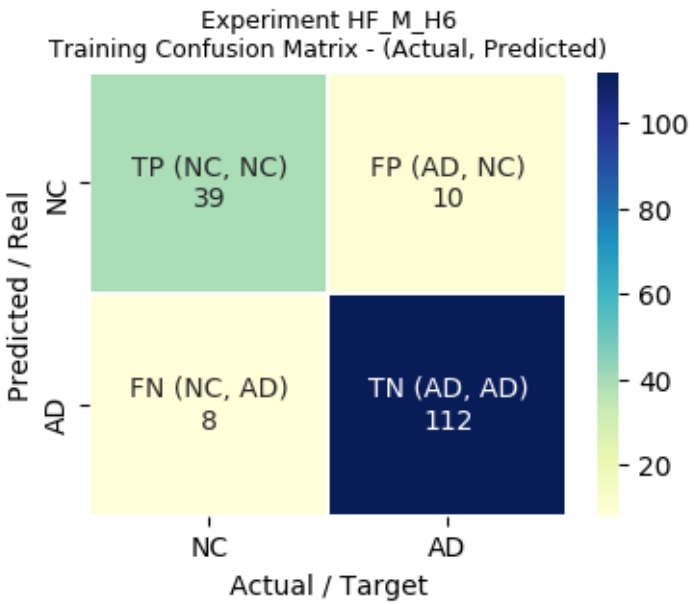


Figure G.140. Caption in the title.

G.3.6.2 Validation

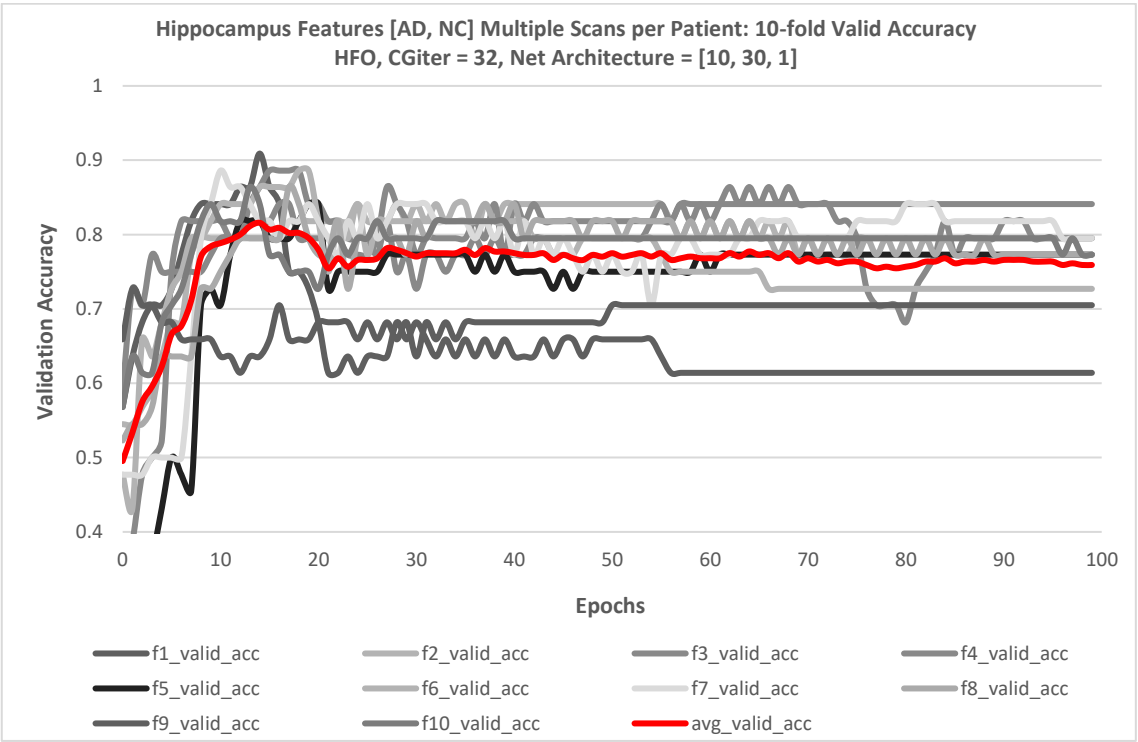


Figure G.141. Caption in the title.

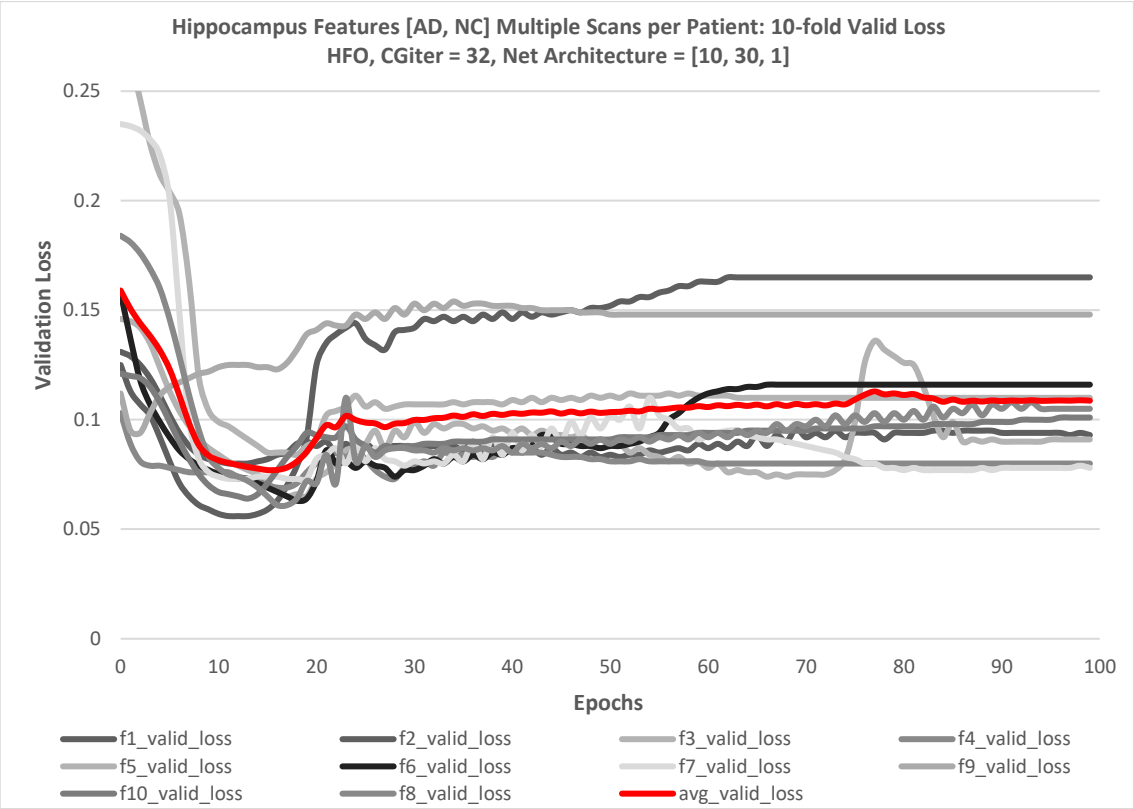


Figure G.142. Caption in the title.

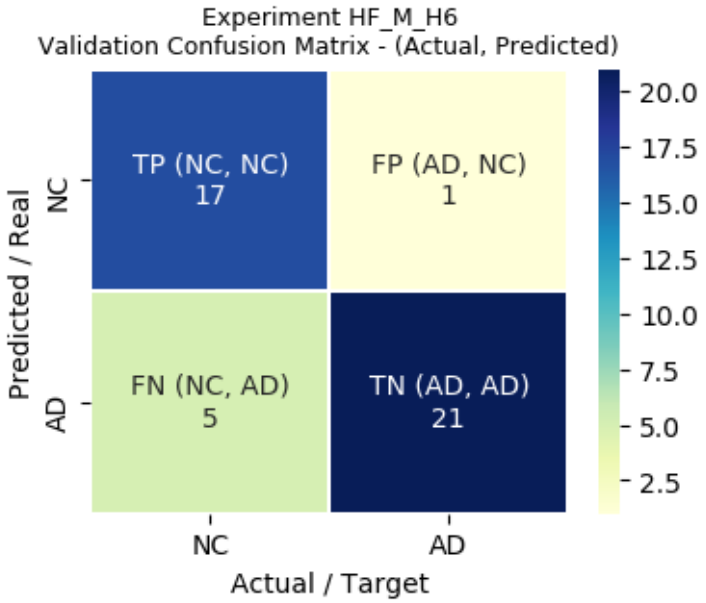


Figure G.143. Caption in the title.

G.3.7 Experiment HF_M_H7: CGiter = 2, net = [10, 20, 20, 1]

G.3.7.1 Training

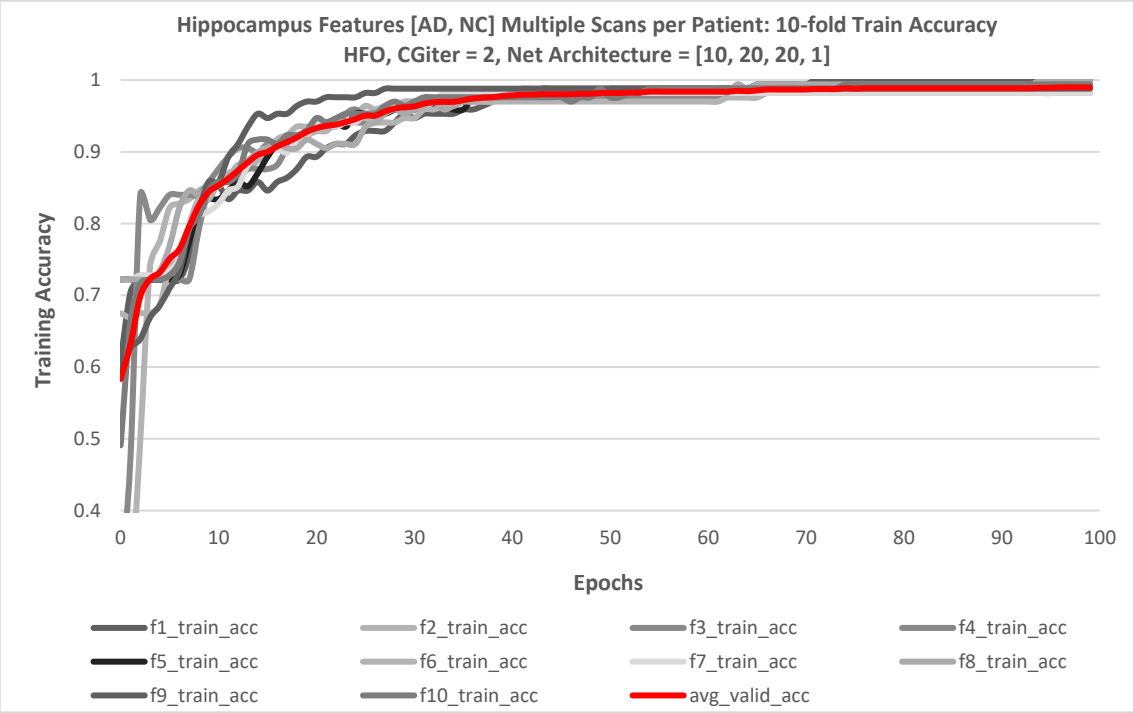


Figure G.144. Caption in the title.

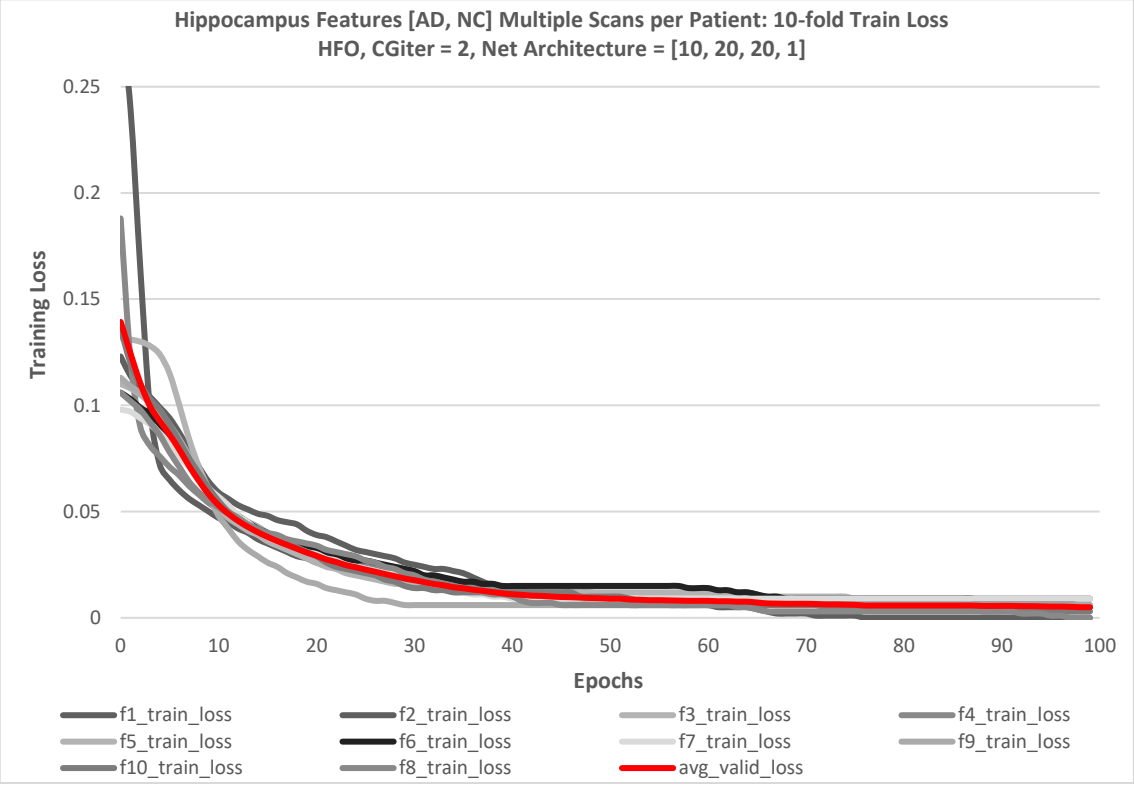


Figure G.145. Caption in the title.

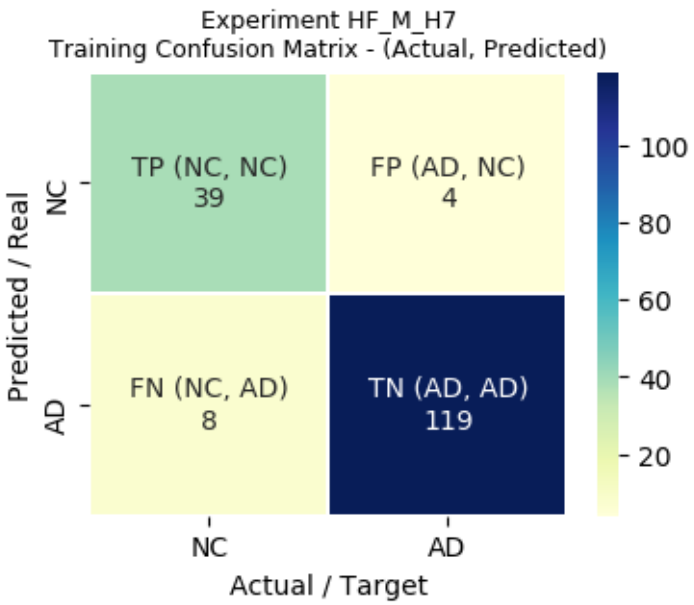


Figure G.146. Caption in the title.

G.3.7.2 Validation

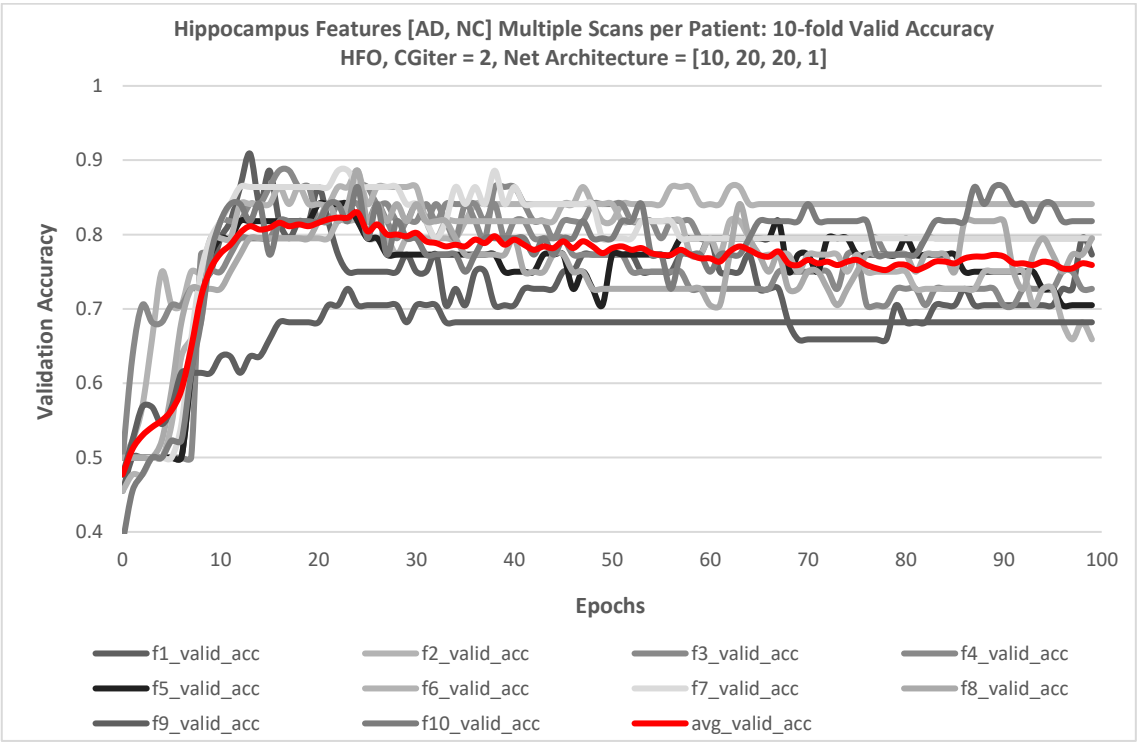


Figure G.147. Caption in the title.

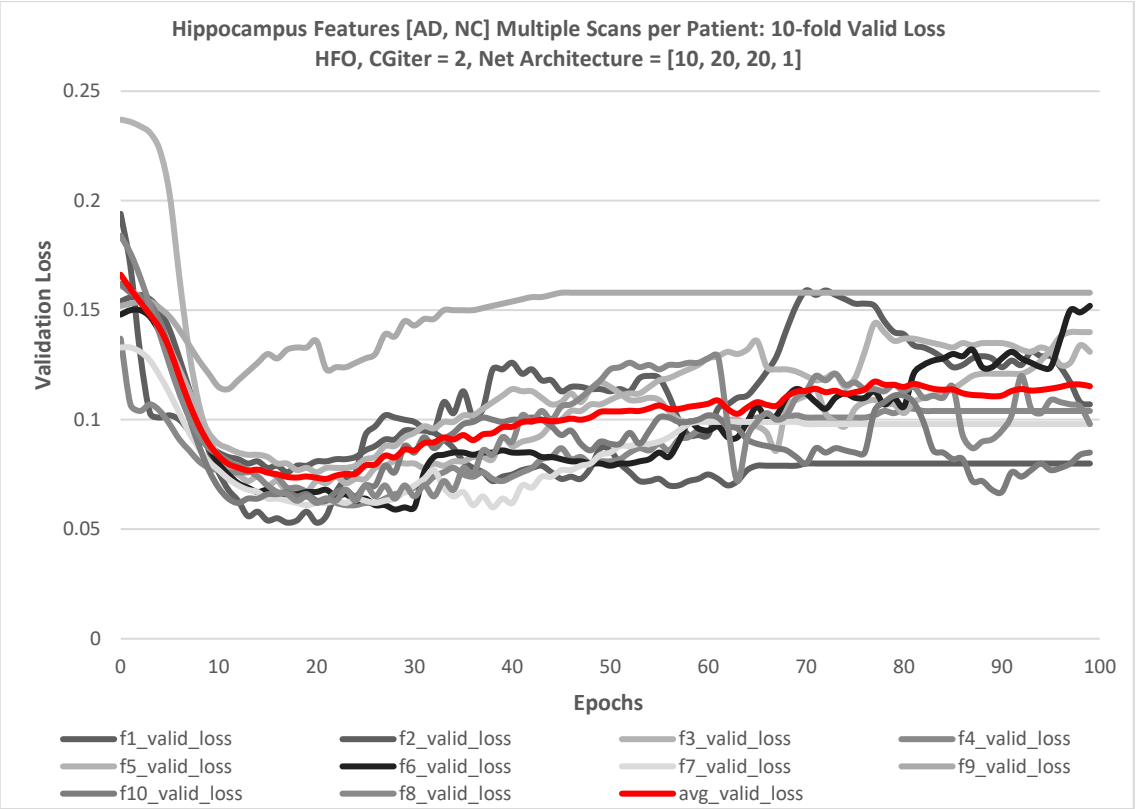


Figure G.148. Caption in the title.

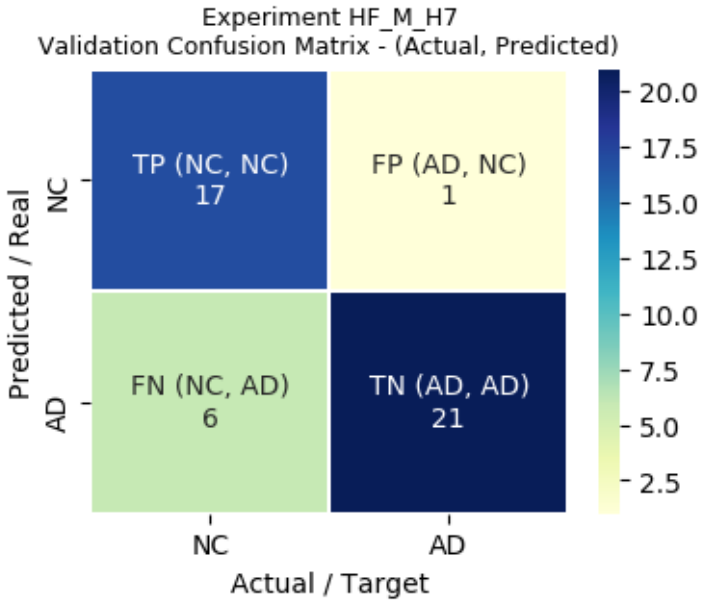


Figure G.149. Caption in the title.

G.3.8 Experiment HF_M_H8: CGiter = 2, net = [10, 100, 1]

G.3.8.1 Training

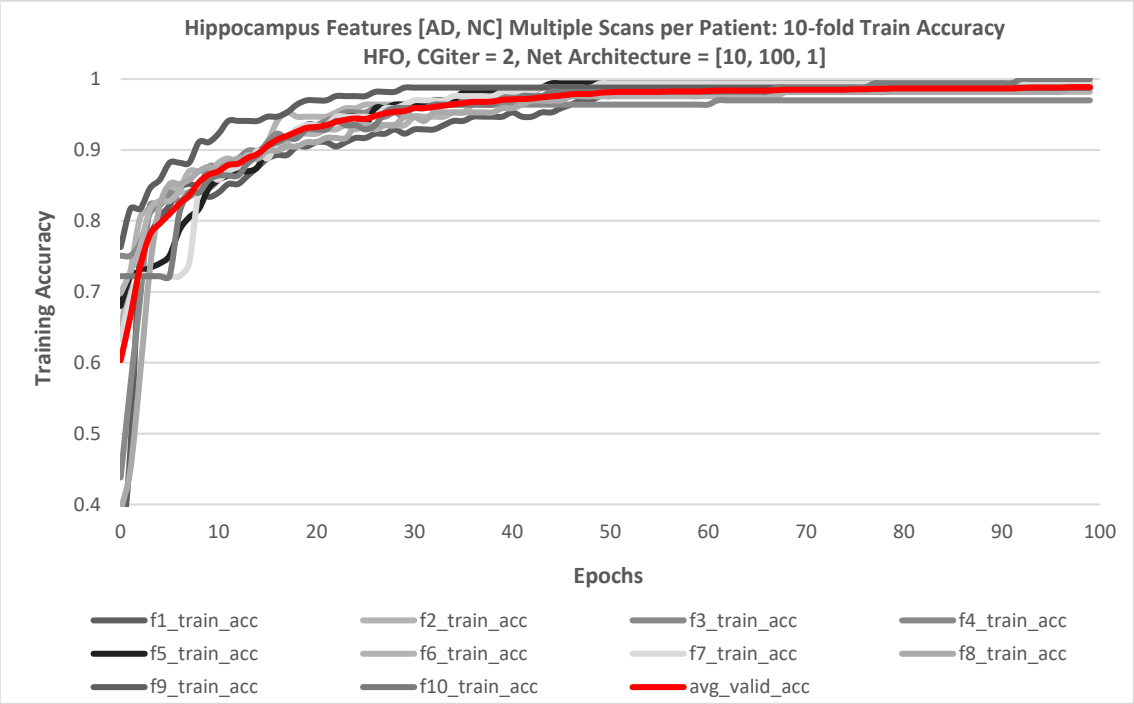


Figure G.150. Caption in the title.

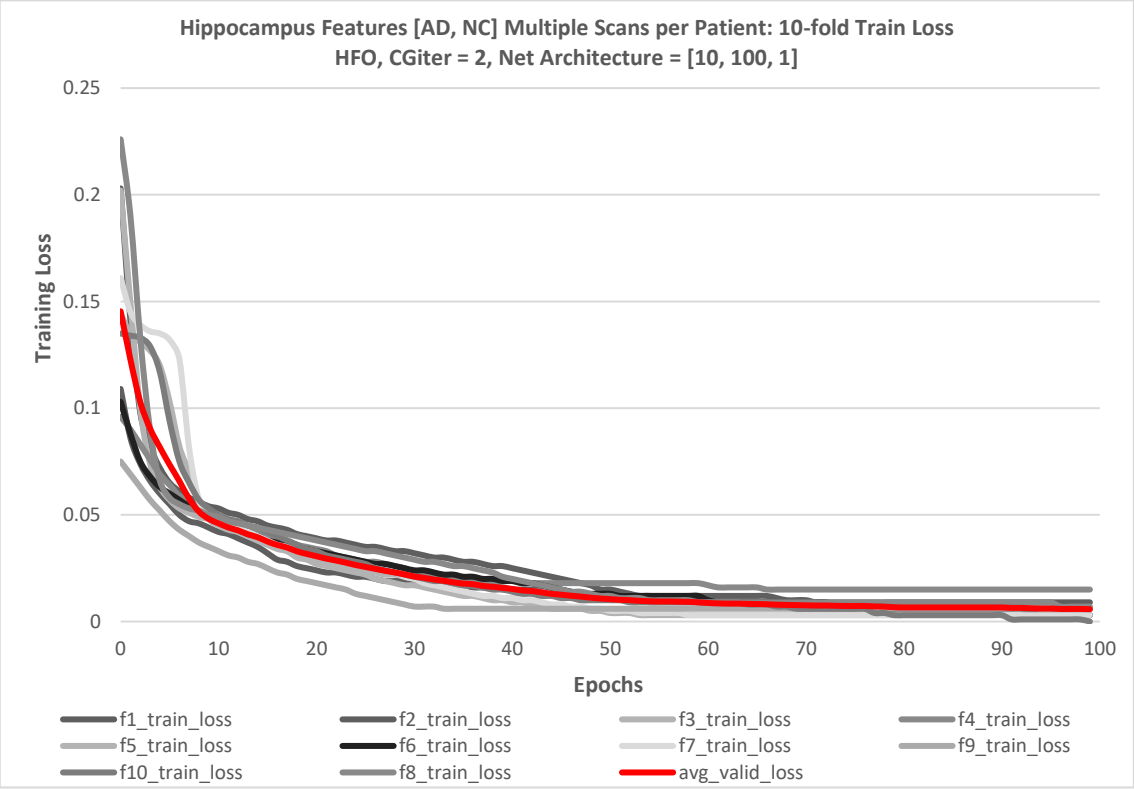


Figure G.151. Caption in the title.

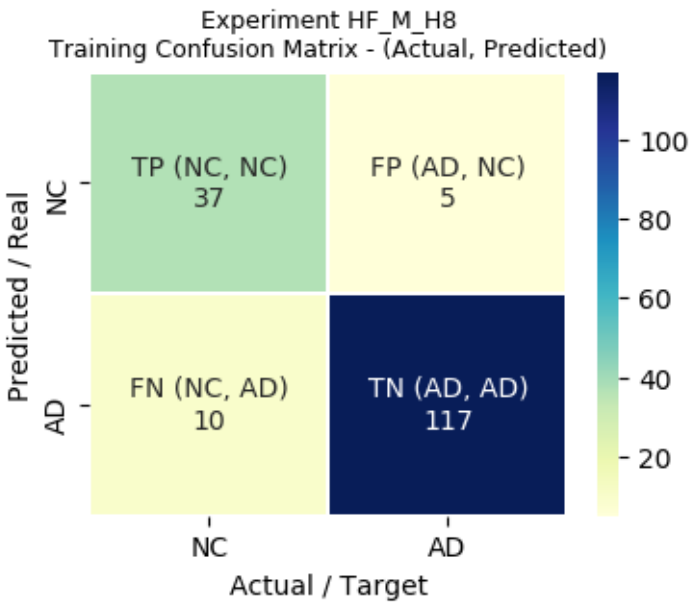


Figure G.152. Caption in the title.

G.3.8.2 Validation

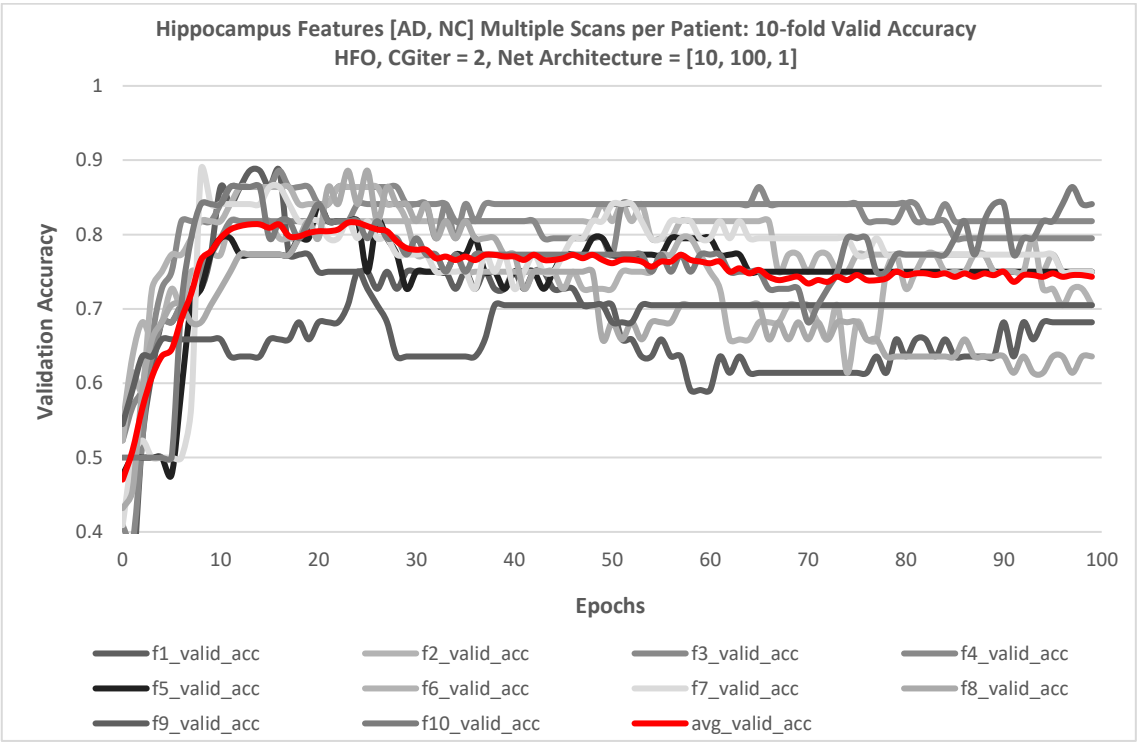


Figure G.153. Caption in the title.

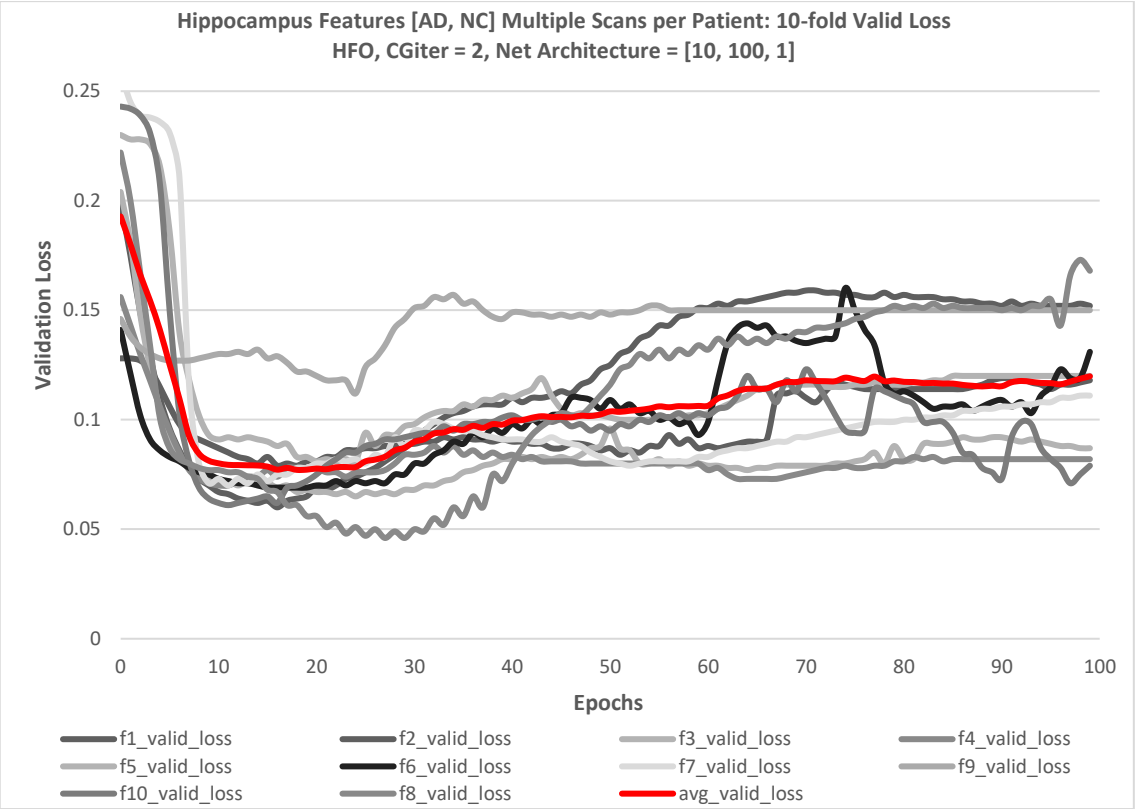


Figure G.154. Caption in the title.

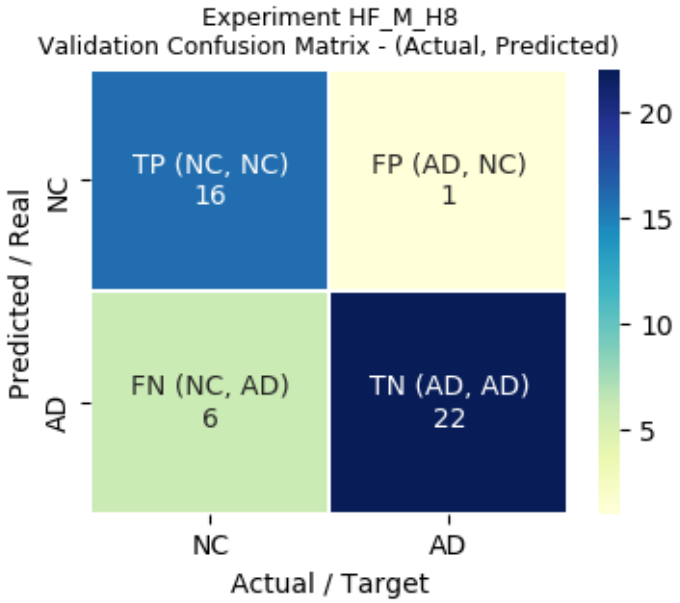


Figure G.155. Caption in the title.

G.3.9 Experiment HF_M_H9: CGiter = 2, net = [10, 8, 8, 1]

G.3.9.1 Training

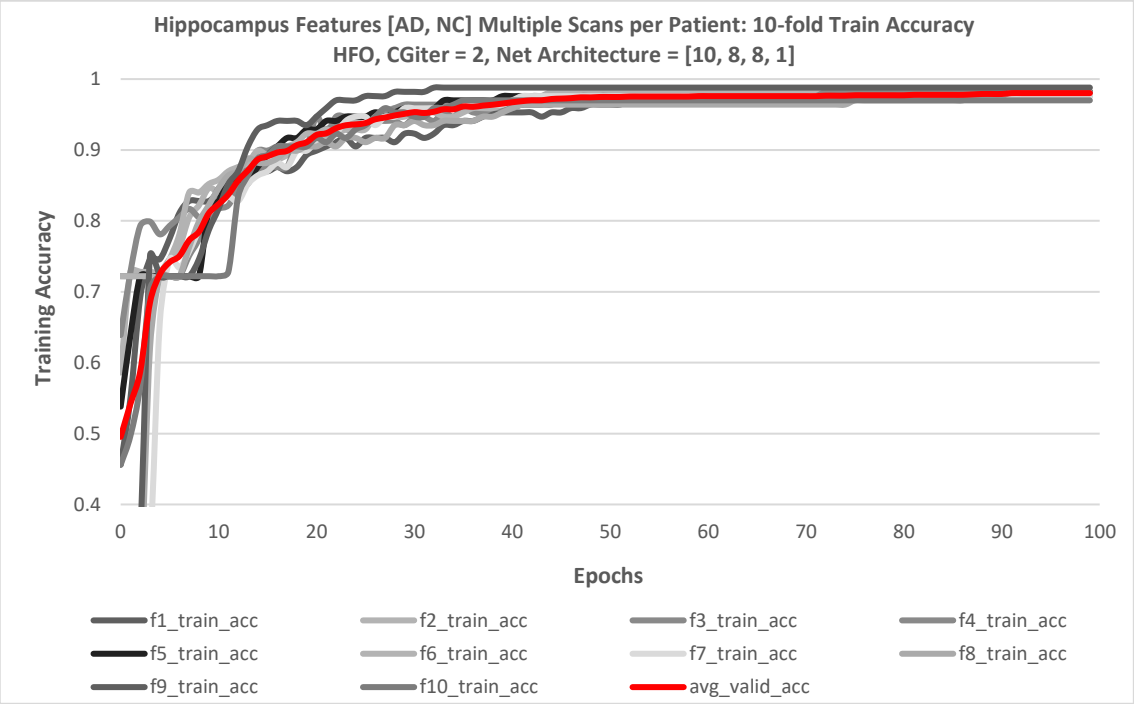


Figure G.156. Caption in the title.

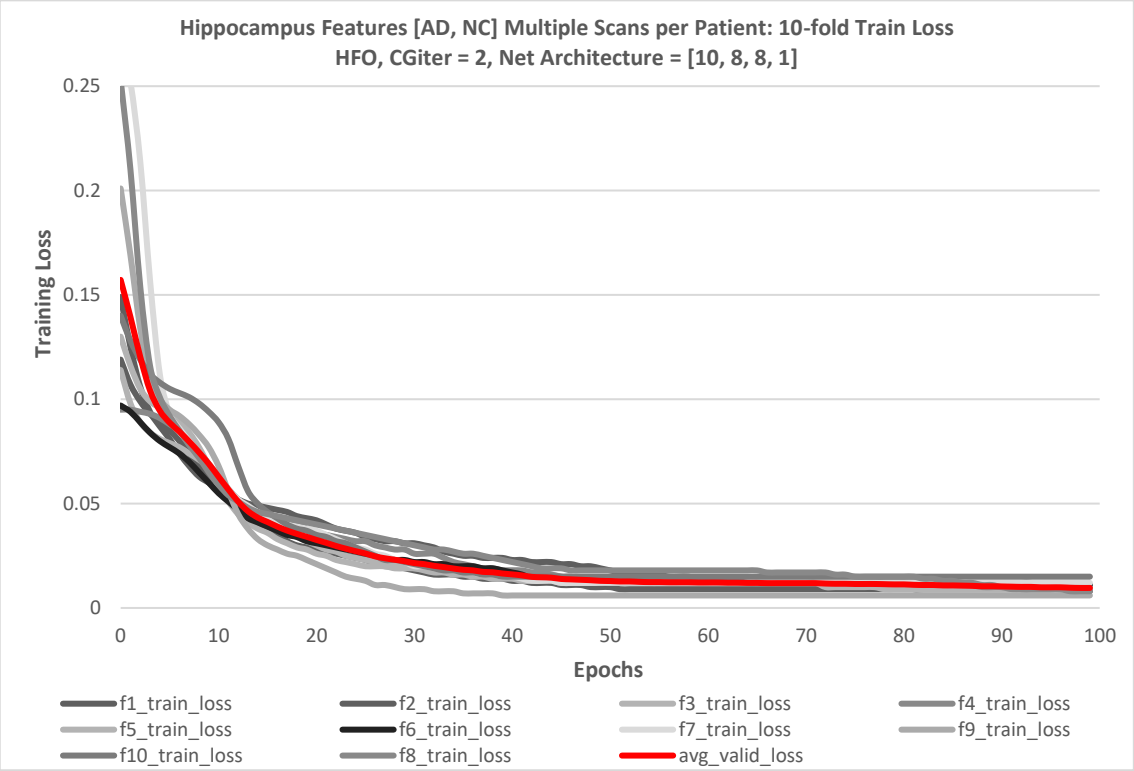


Figure G.157. Caption in the title.

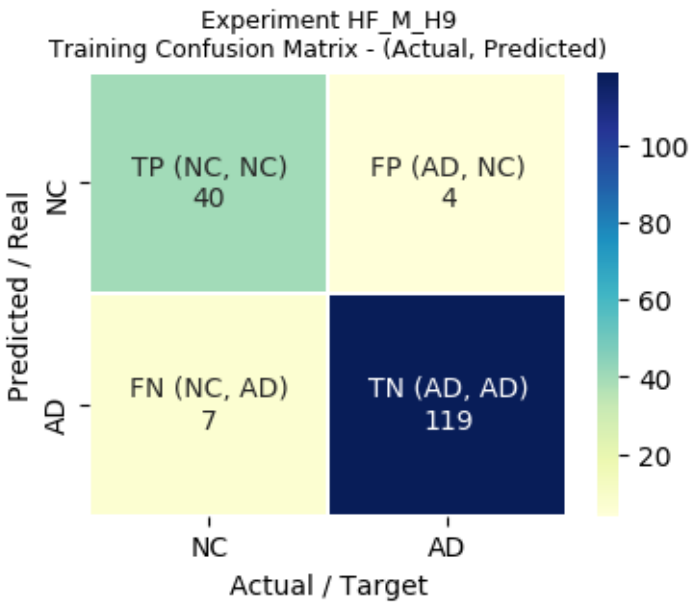


Figure G.158. Caption in the title.

G.3.9.2 Validation

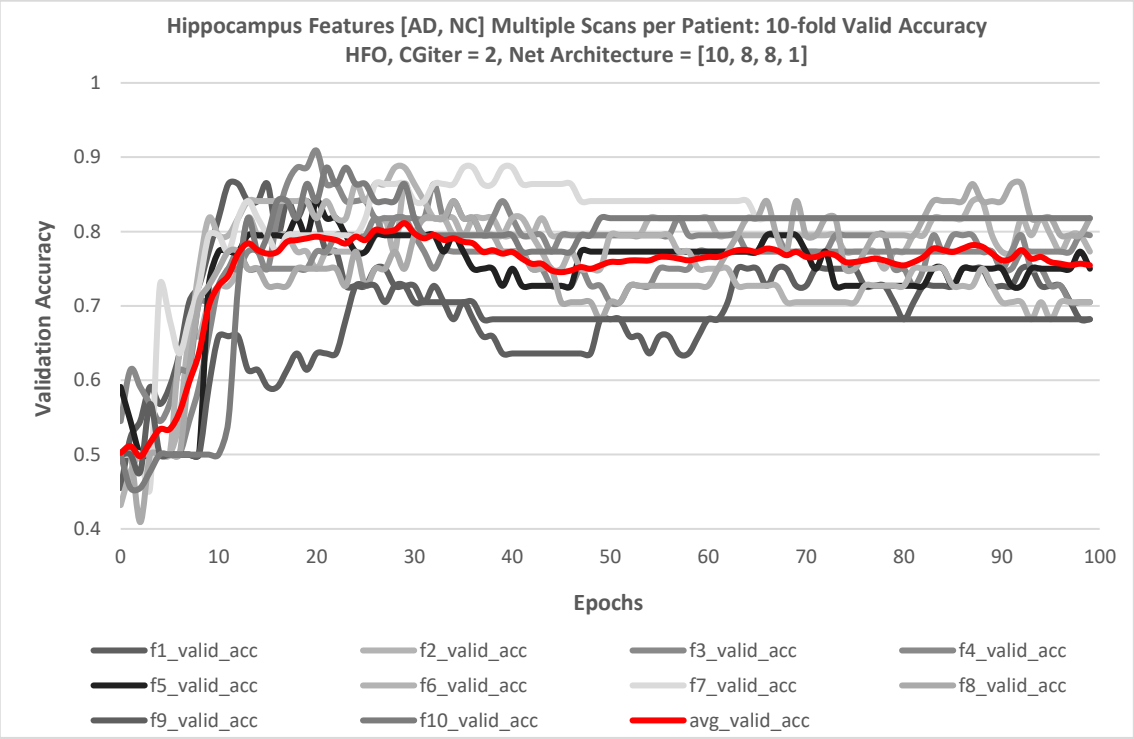


Figure G.159. Caption in the title.

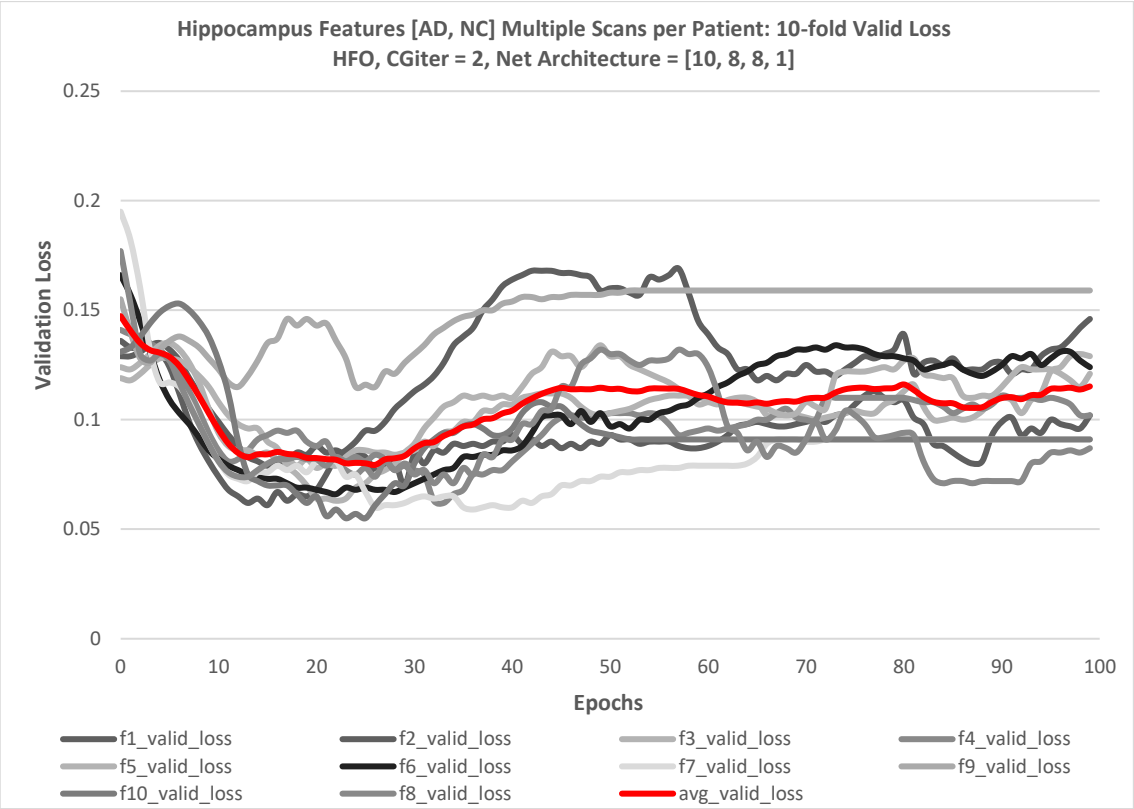


Figure G.160. Caption in the title.

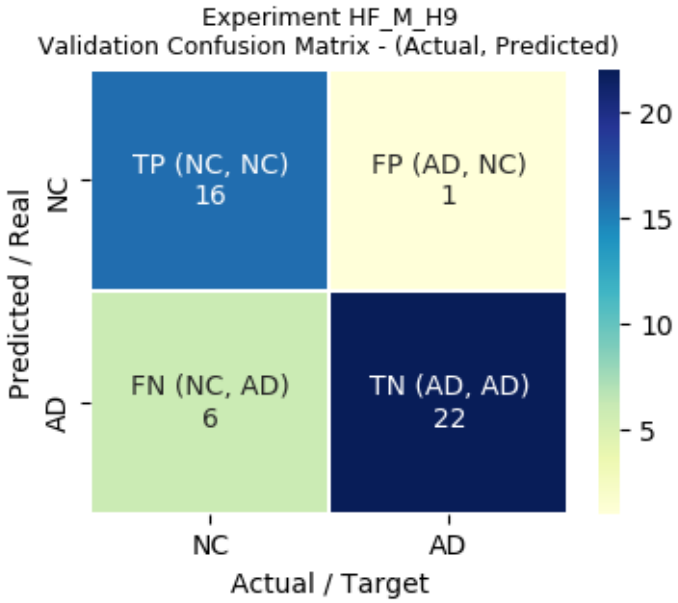


Figure G.161. Caption in the title.

Appendix H

Experiments 2D Brain Slices [AD, NC]: Multiple Scans per Patient – Single Slice per Scan (174 × 174)

H.1 CNN 4 Layers with Adam

H.1.1 Experiment B_2D_M_A1: C = 0.01, lr = 0.1, Dropout

H.1.1.1 General

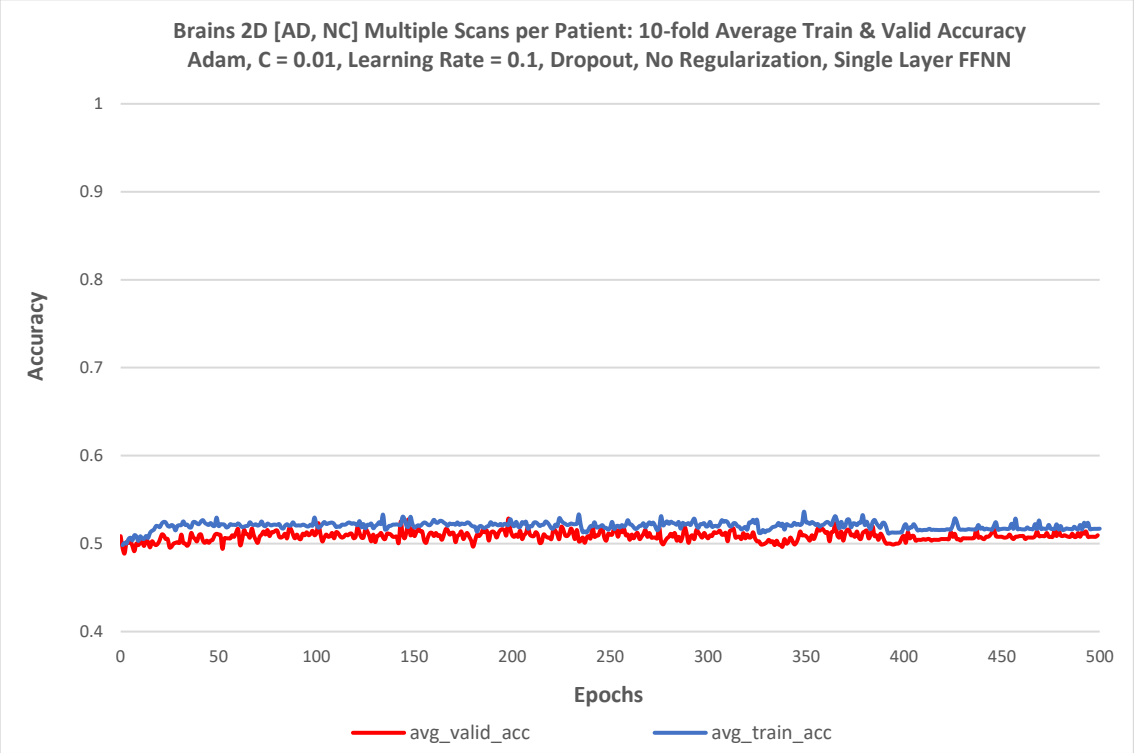


Figure H.1. Caption in the title.

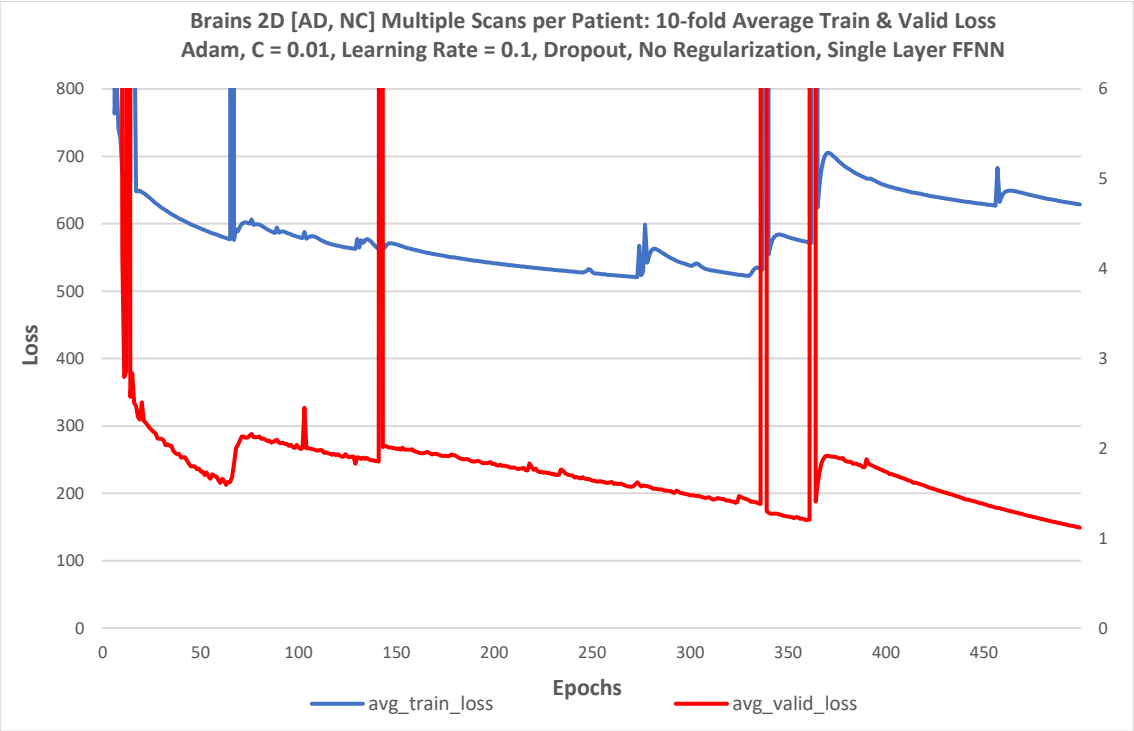


Figure H.2. Caption in the title.

H.1.1.2 Training

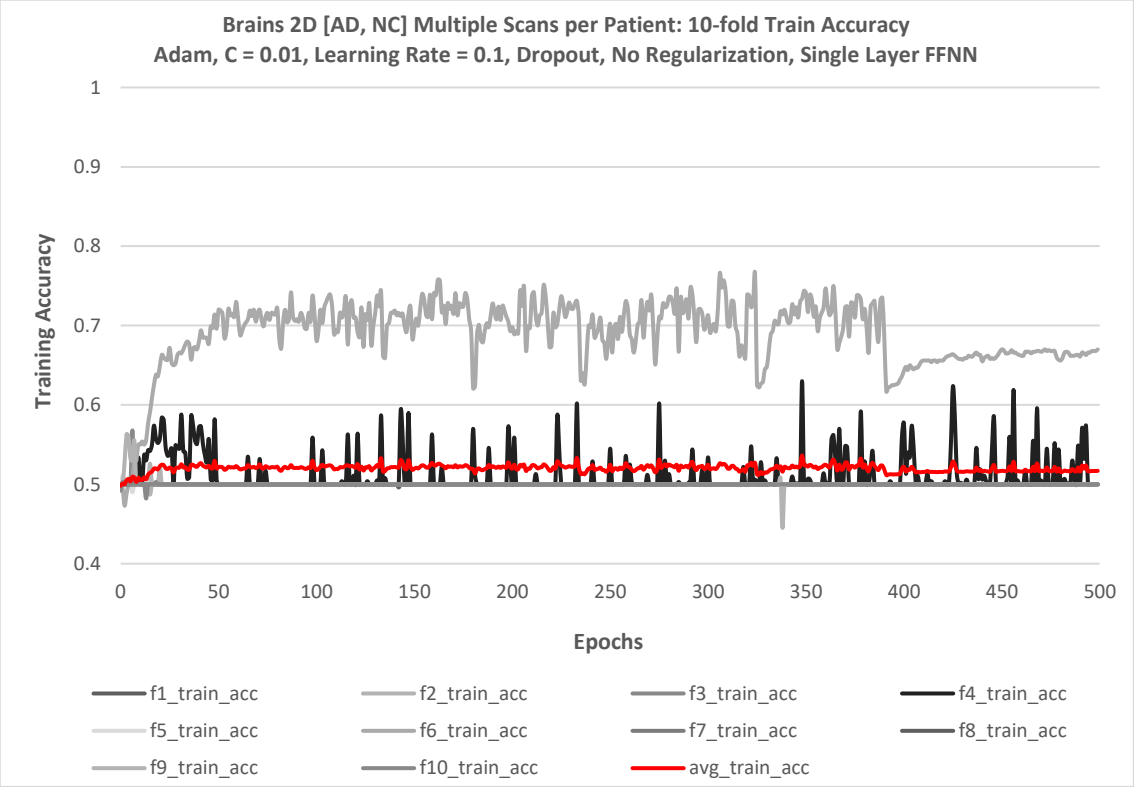


Figure H.3. Caption in the title.

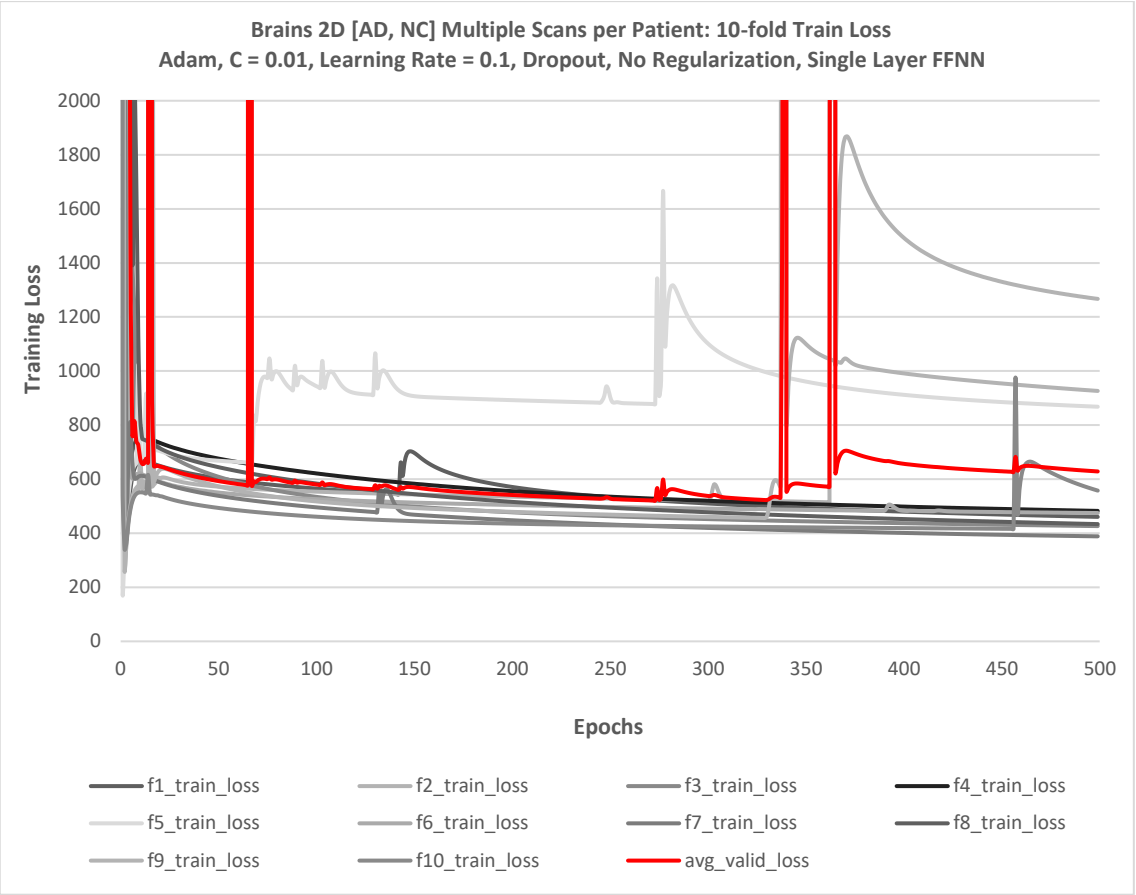


Figure H.4. Caption in the title.

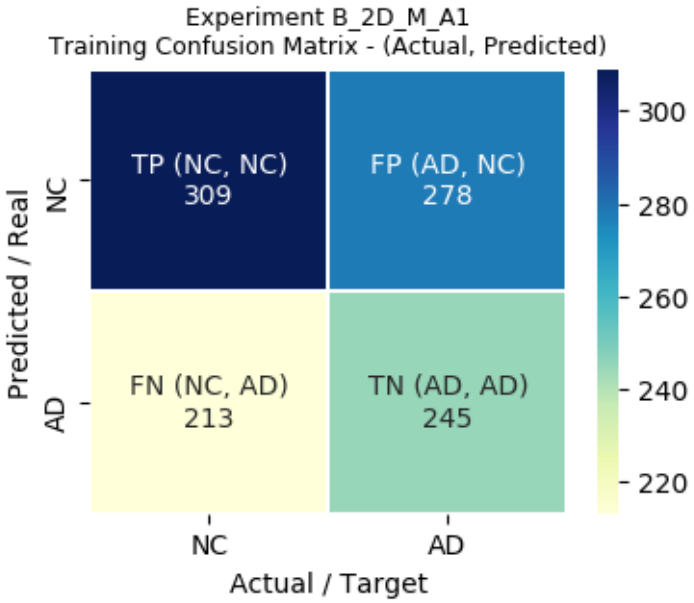


Figure H.5. Caption in the title.

H.1.1.3 Validation

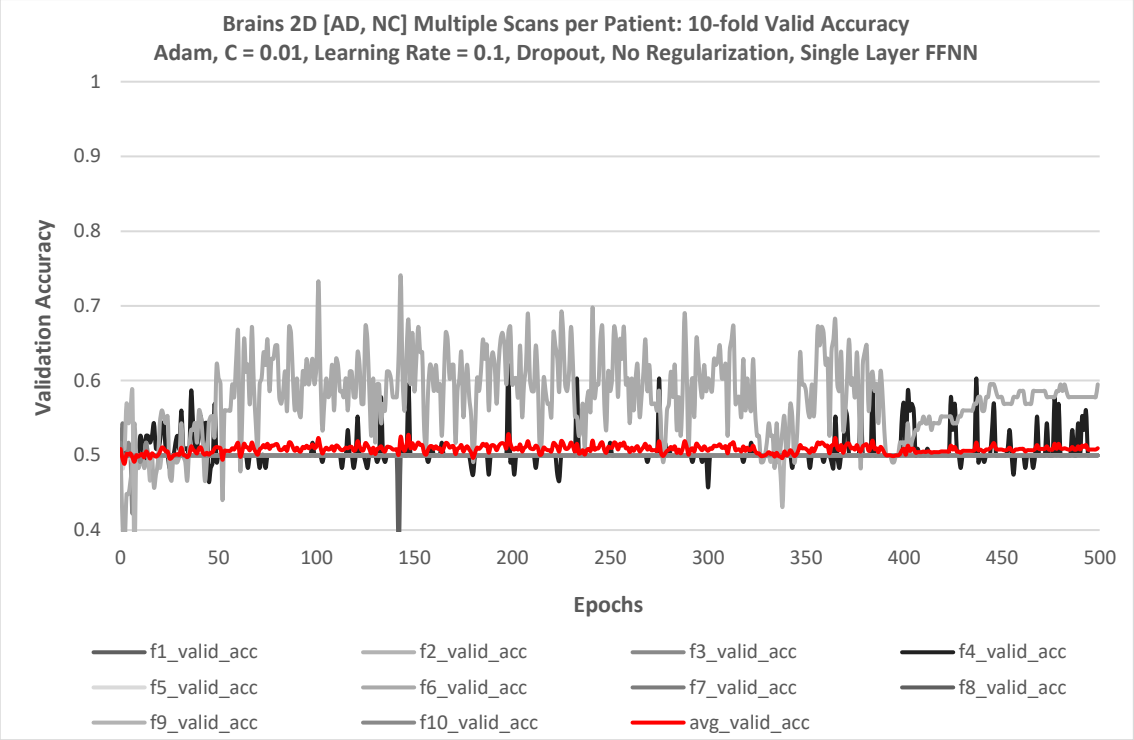


Figure H.6. Caption in the title.

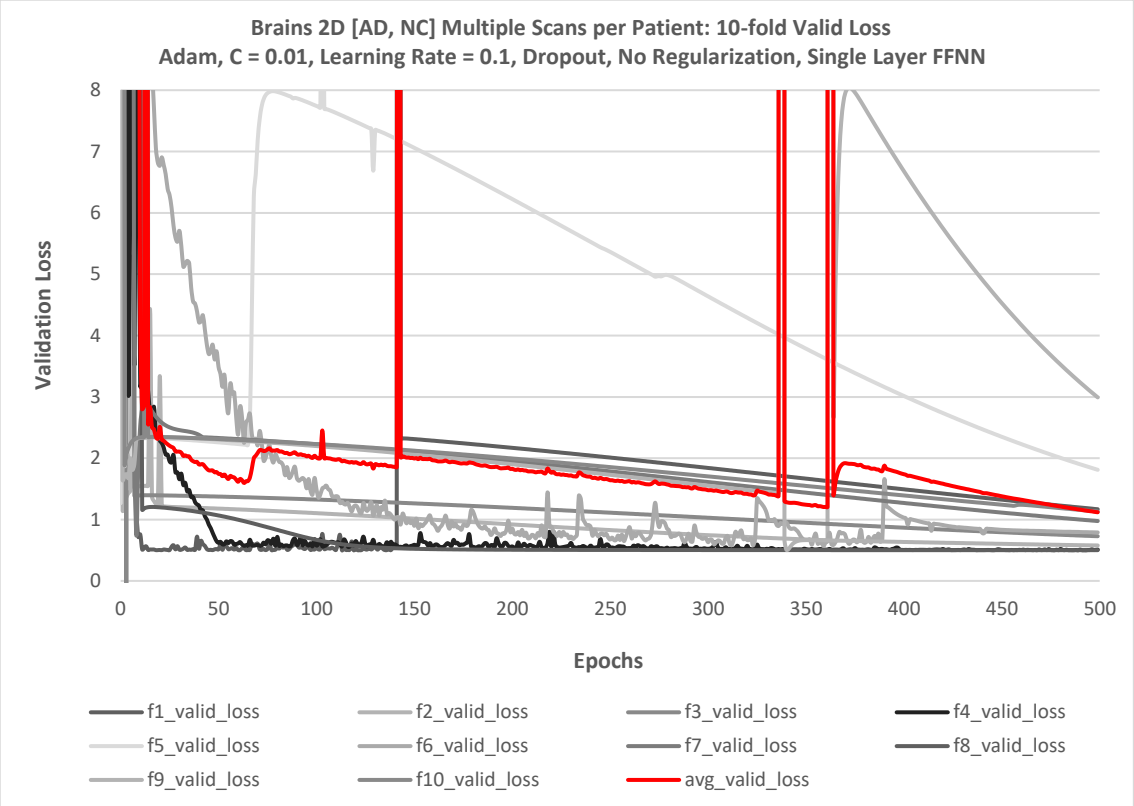


Figure H.7. Caption in the title.

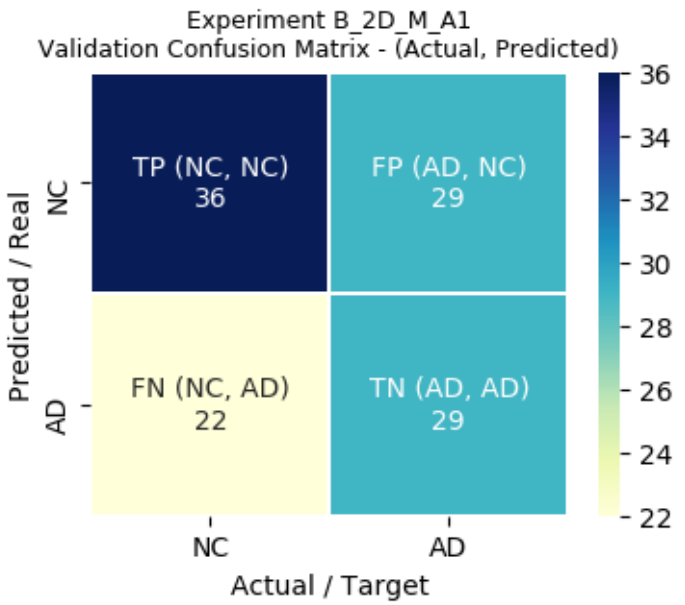


Figure H.8. Caption in the title.

H.1.1.4 Testing

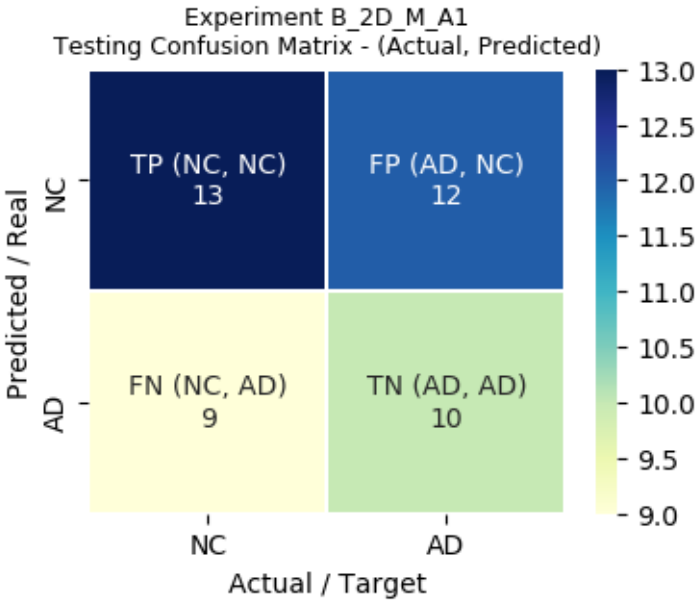


Figure H.9. Caption in the title.

H.1.2 Experiment B_2D_M_A2: C = 0.01, lr = 0.01, Dropout

H.1.2.1 General

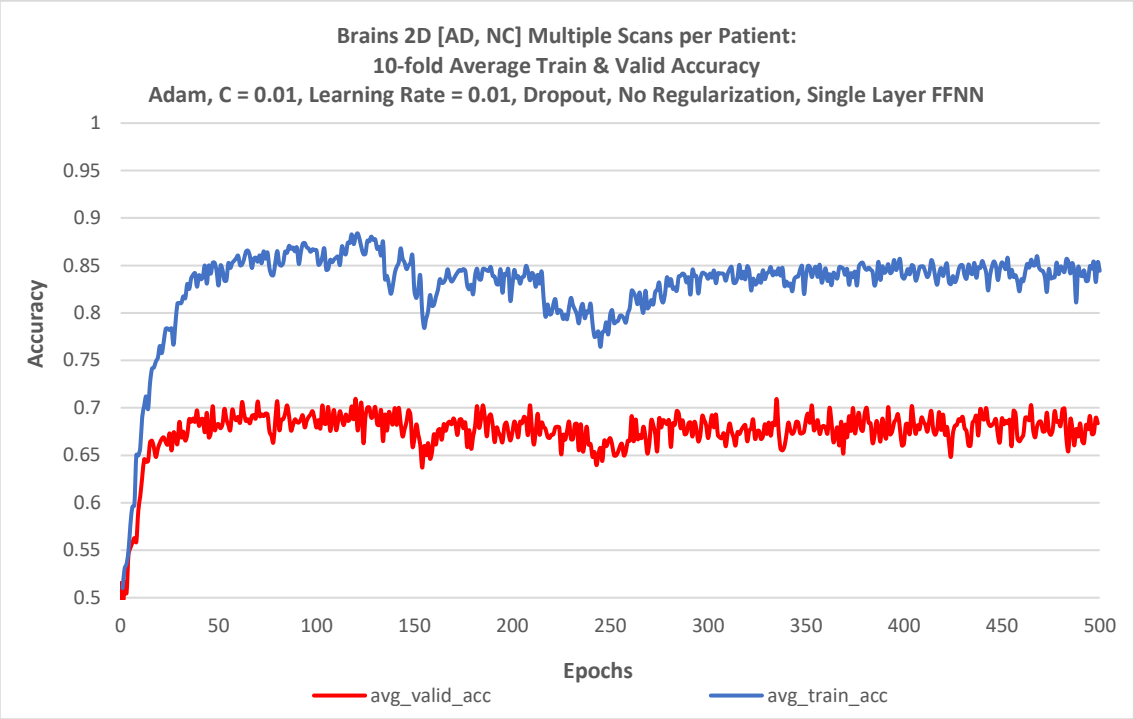


Figure H.10. Caption in the title.

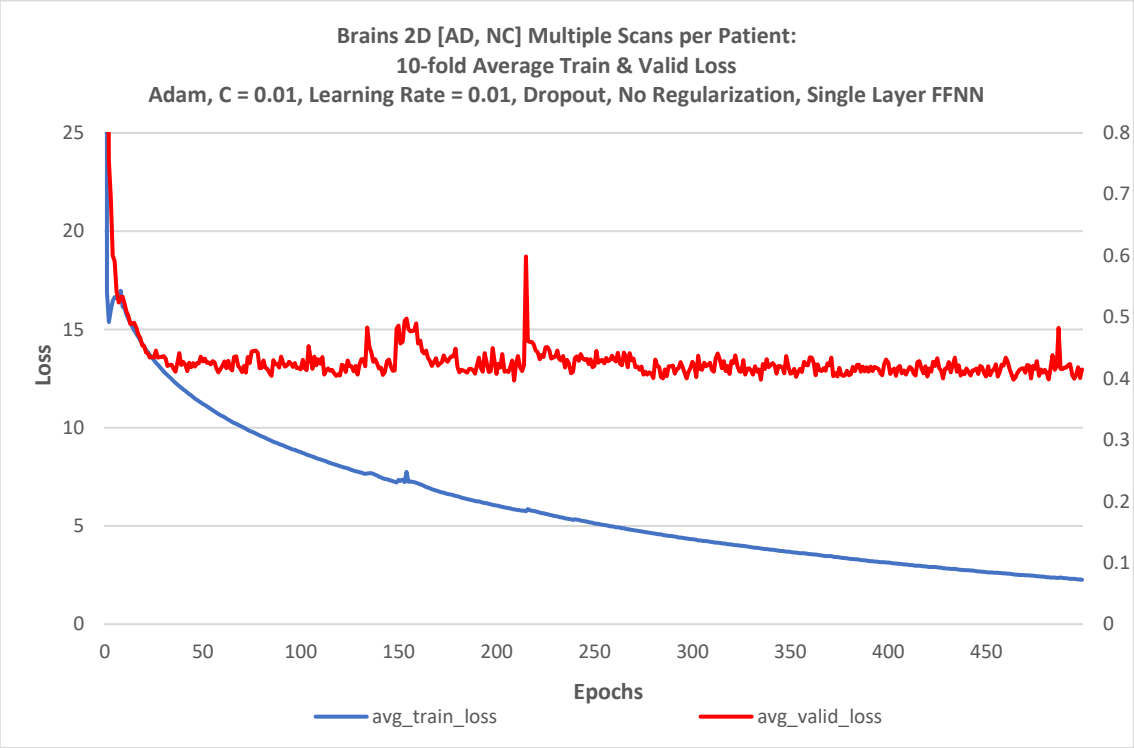


Figure H.11. Caption in the title.

H.1.2.2 Training

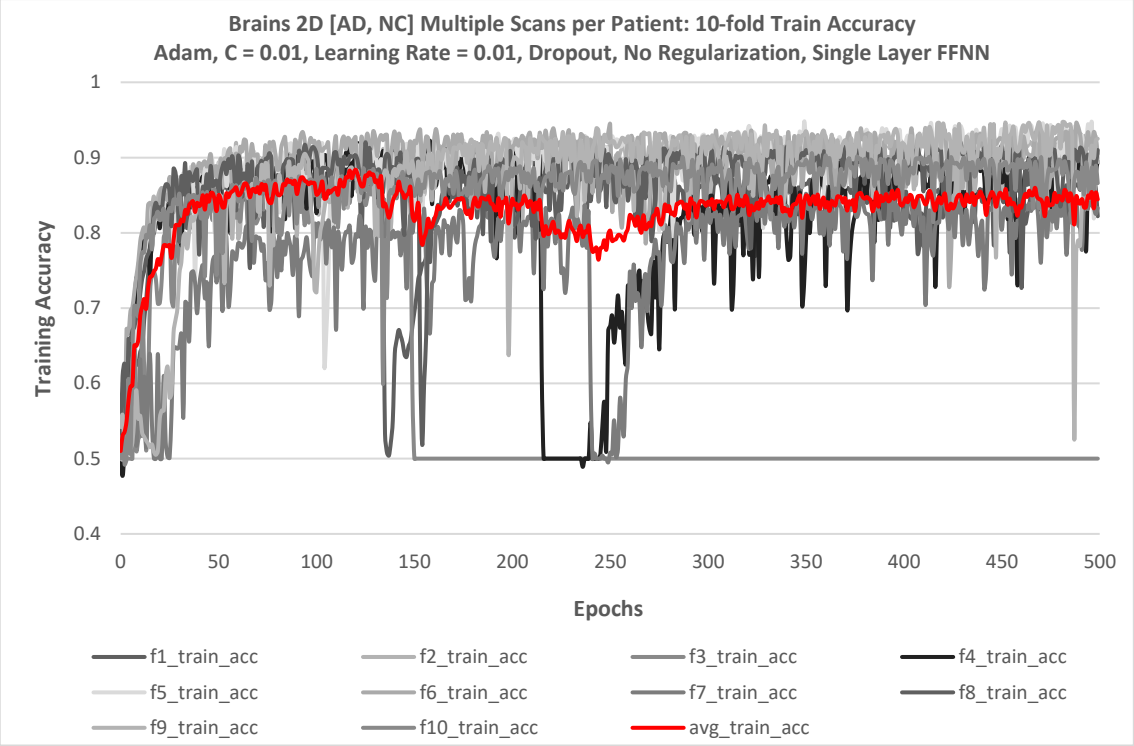


Figure H.12. Caption in the title.

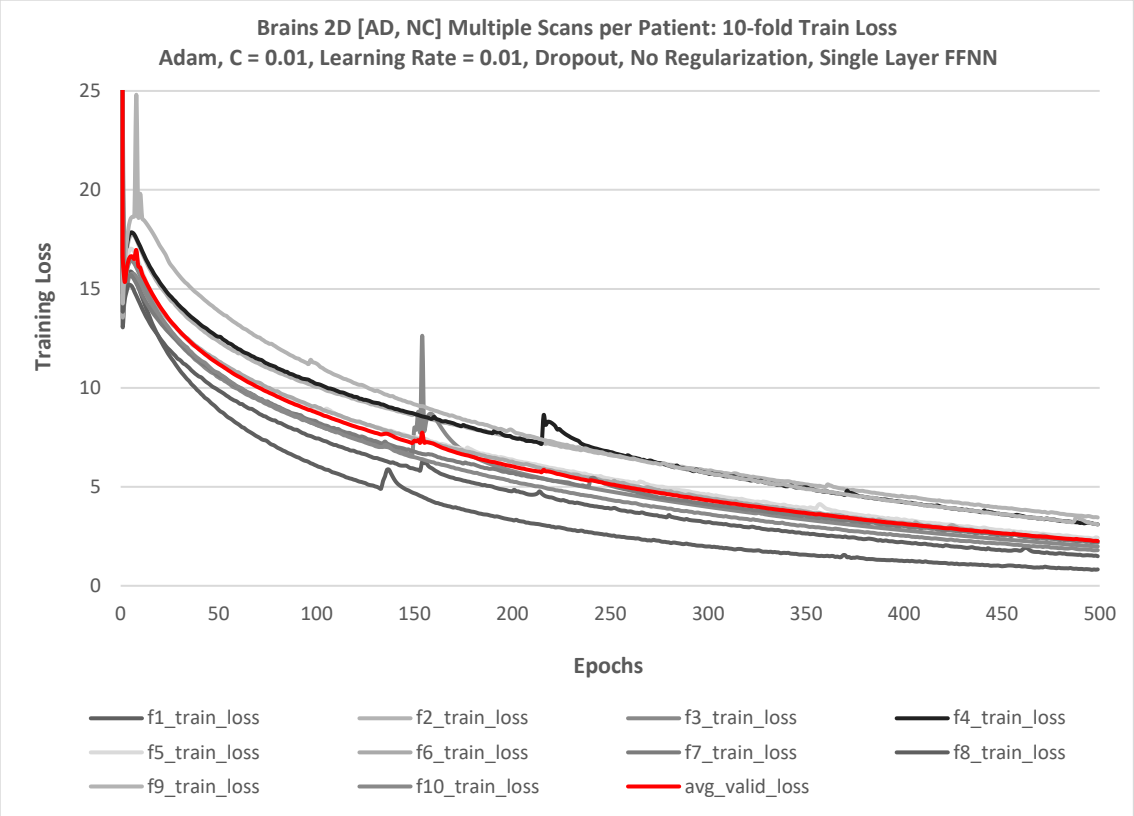


Figure H.13. Caption in the title.

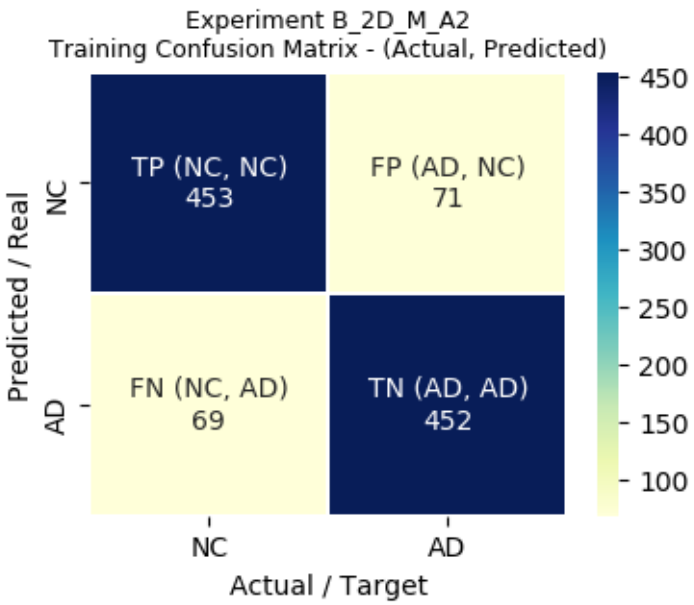


Figure H.14. Caption in the title.

H.1.2.3 Validation

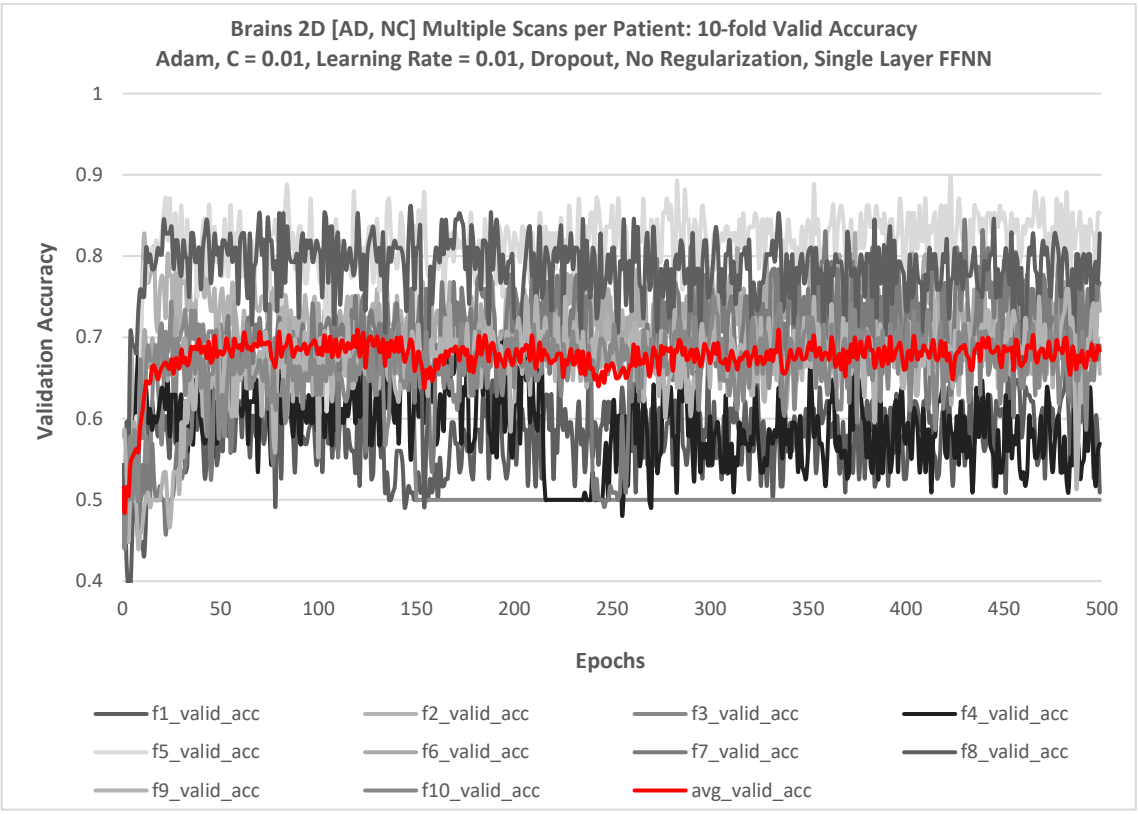


Figure H.15. Caption in the title.

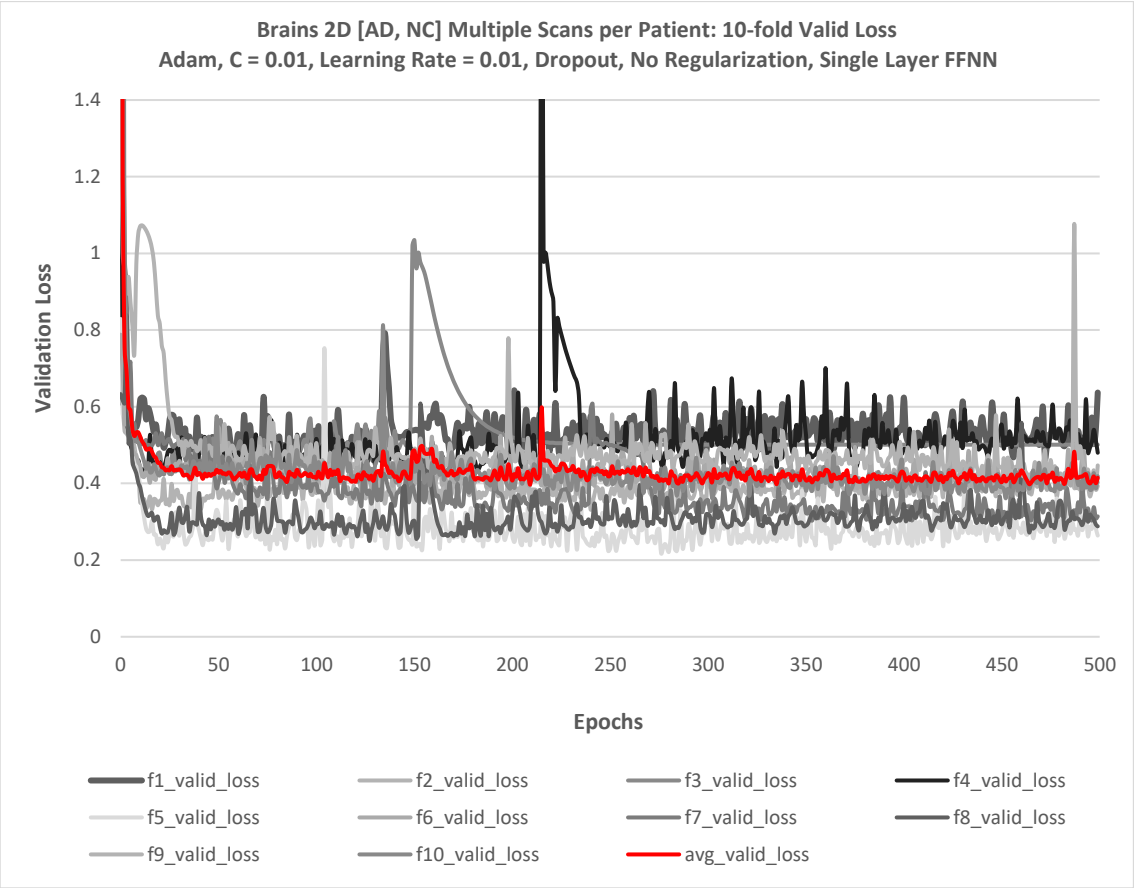


Figure H.16. Caption in the title.

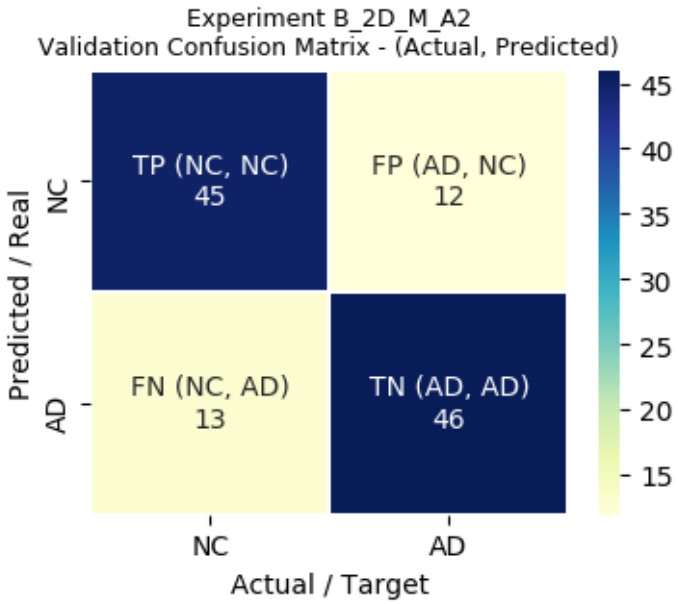


Figure H.17. Caption in the title.

H.1.2.4 Testing

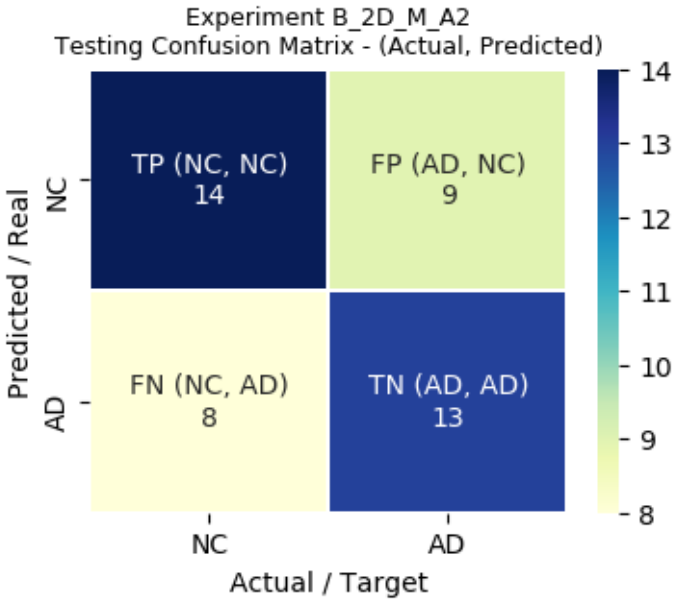


Figure H.18. Caption in the title.

H.1.3 Experiment B_2D_M_A3: C = 0.01, lr = 0.001, Dropout

H.1.3.1 General

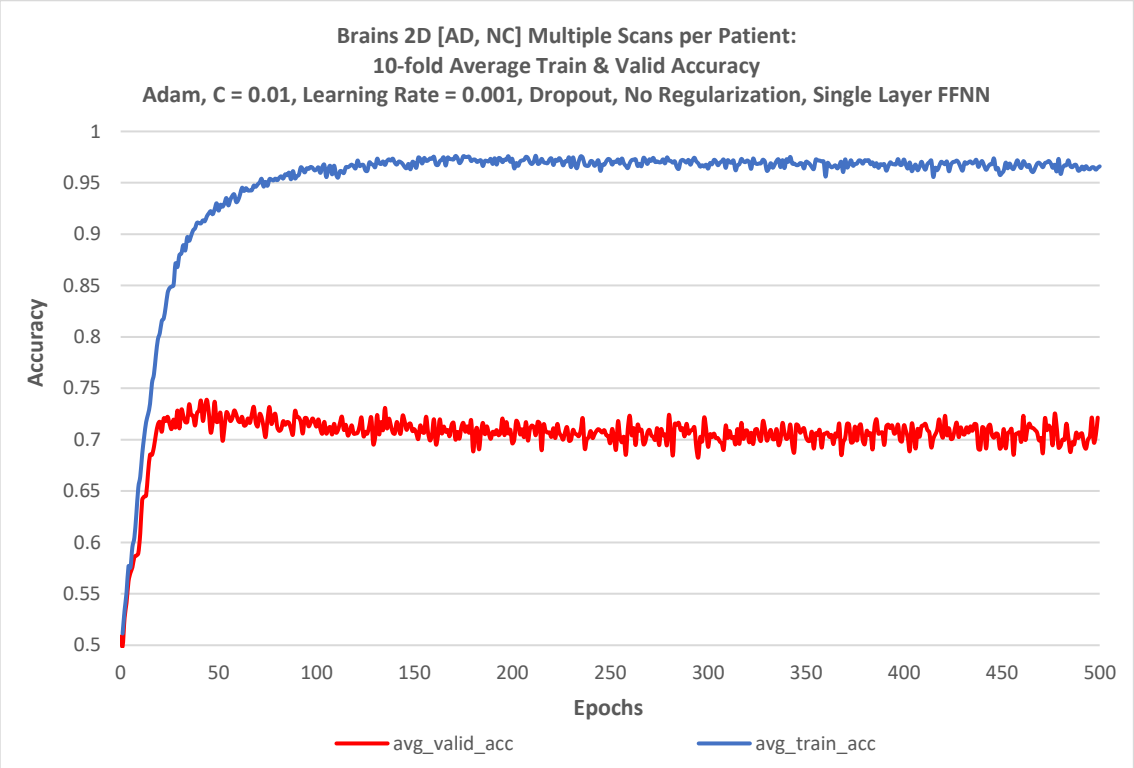


Figure H.19. Caption in the title.

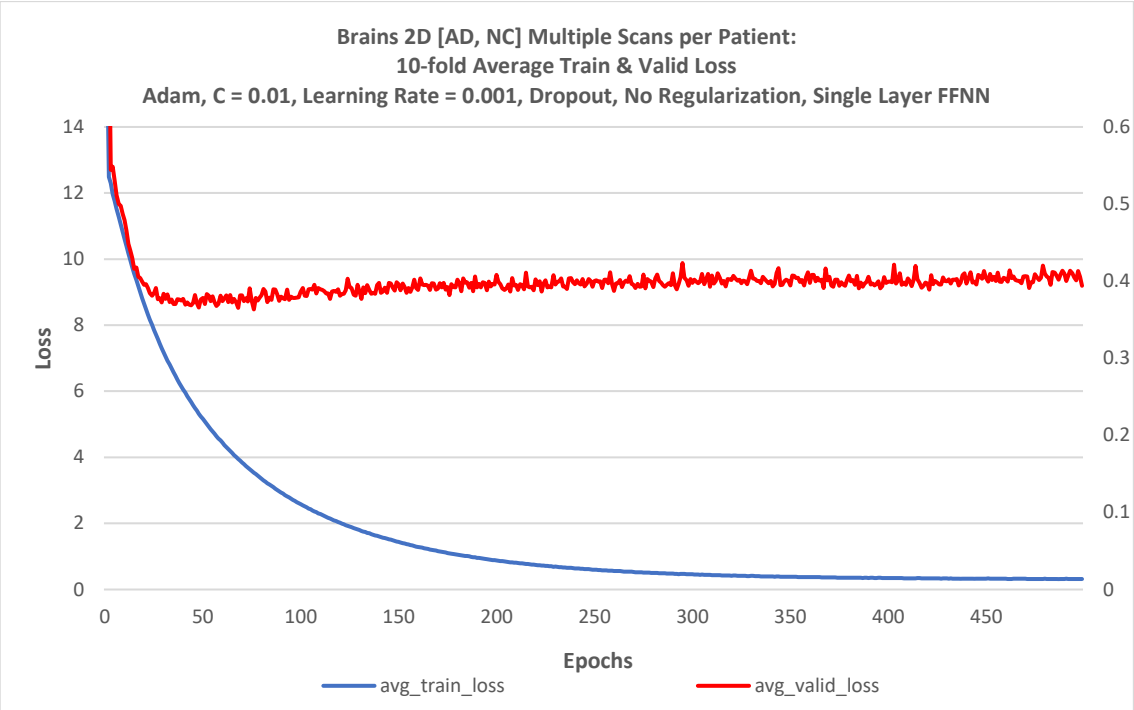


Figure H.20. Caption in the title.

H.1.3.2 Training

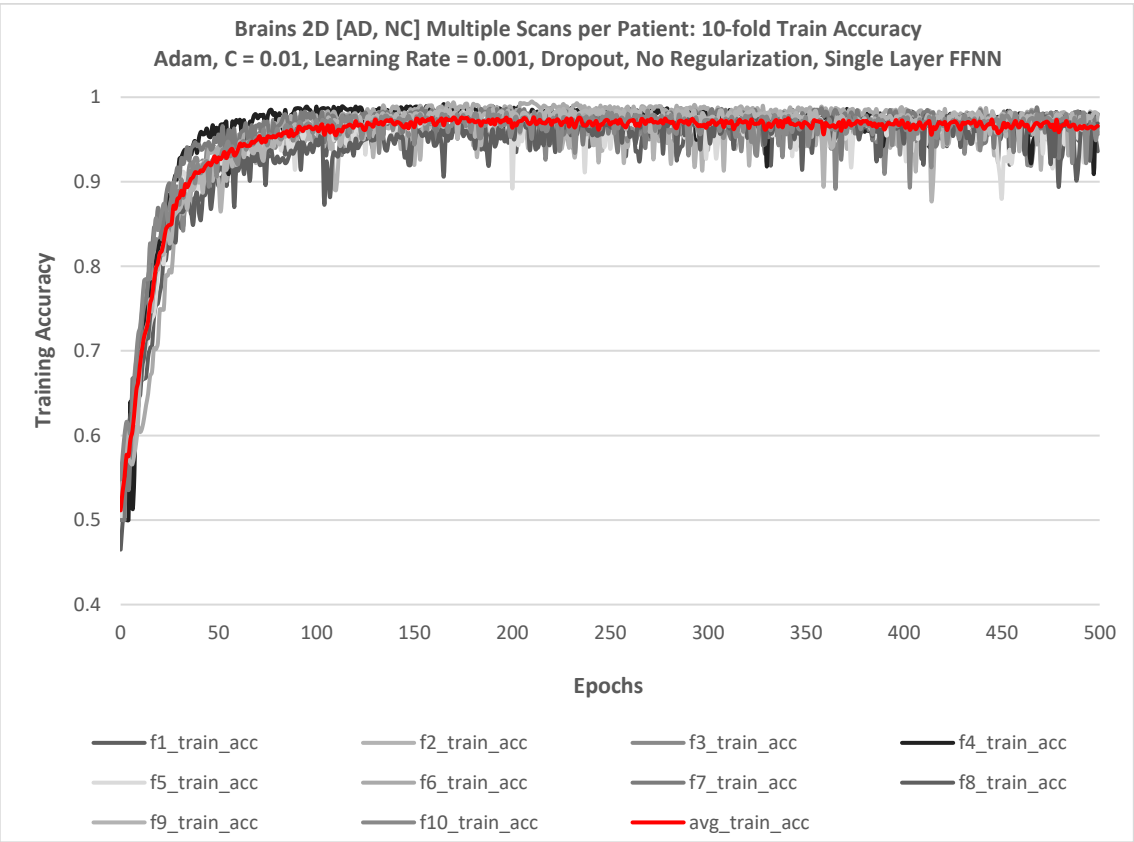


Figure H.21. Caption in the title.

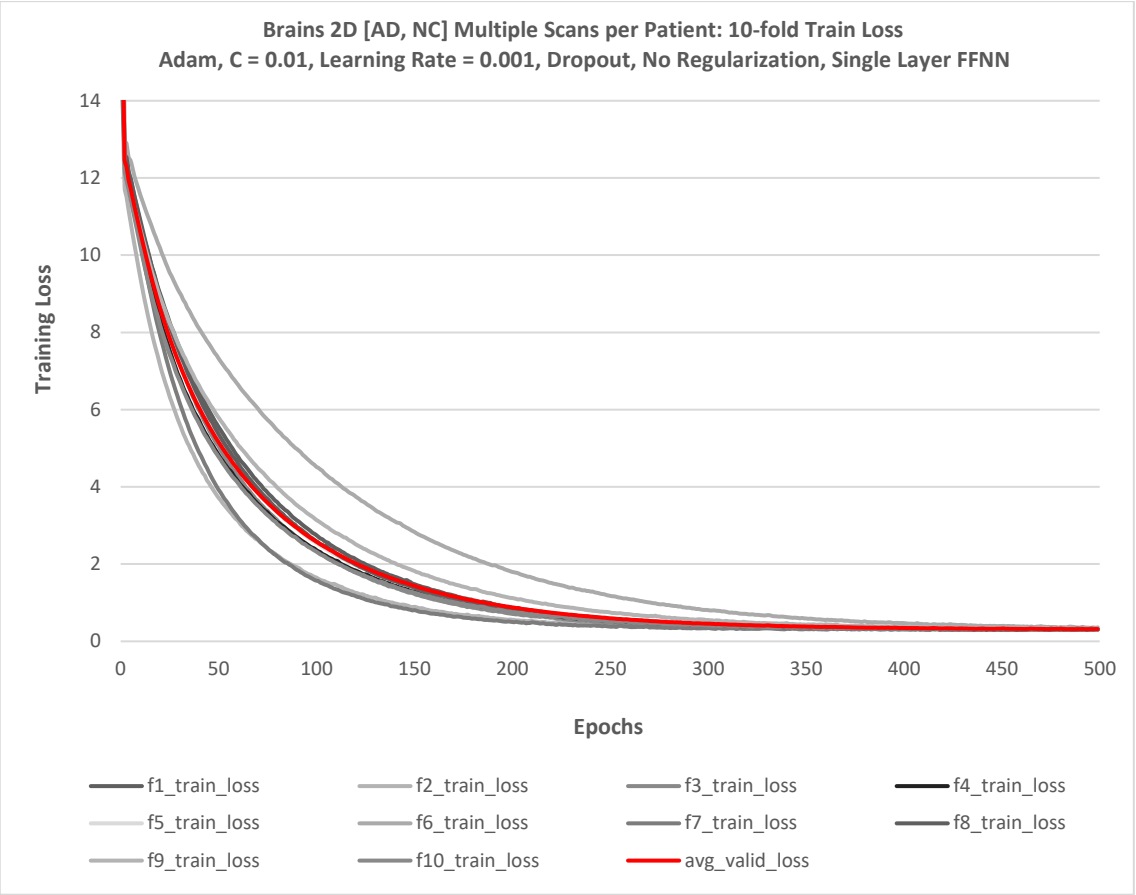


Figure H.22. Caption in the title.

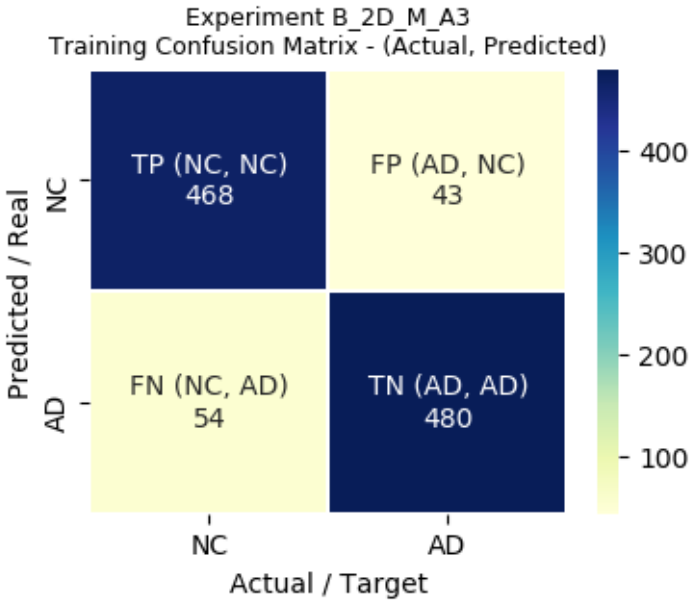


Figure H.23. Caption in the title.

H.1.3.3 Validation

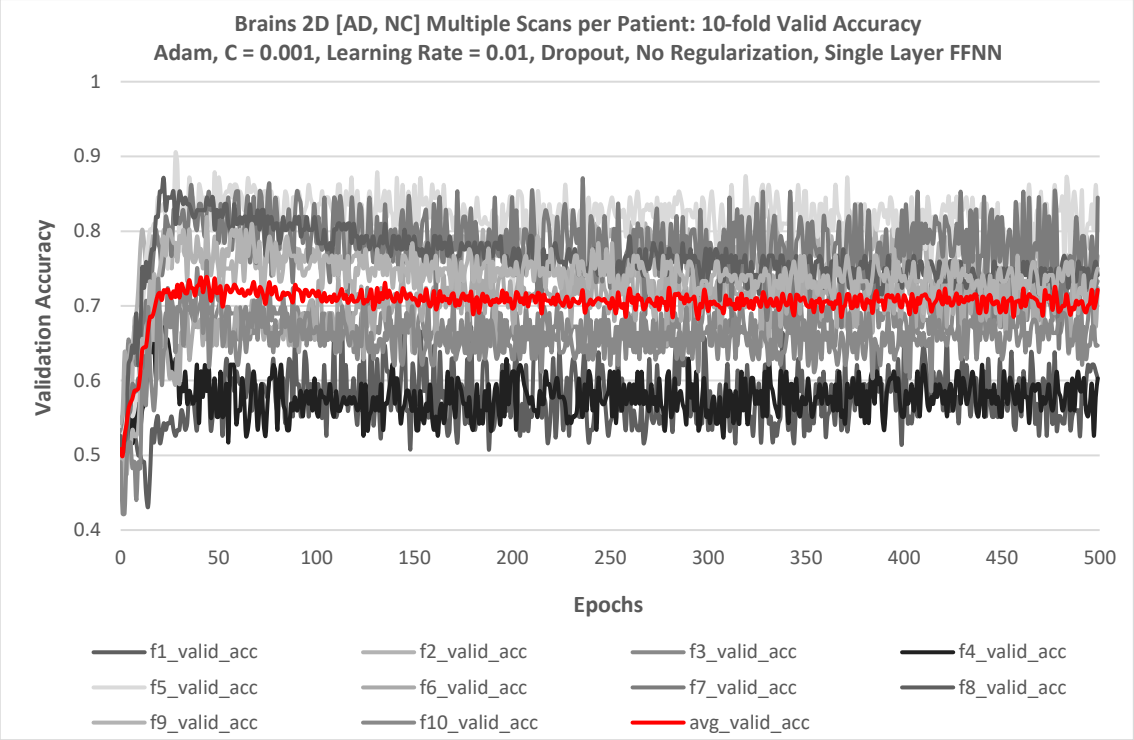


Figure H.24. Caption in the title.

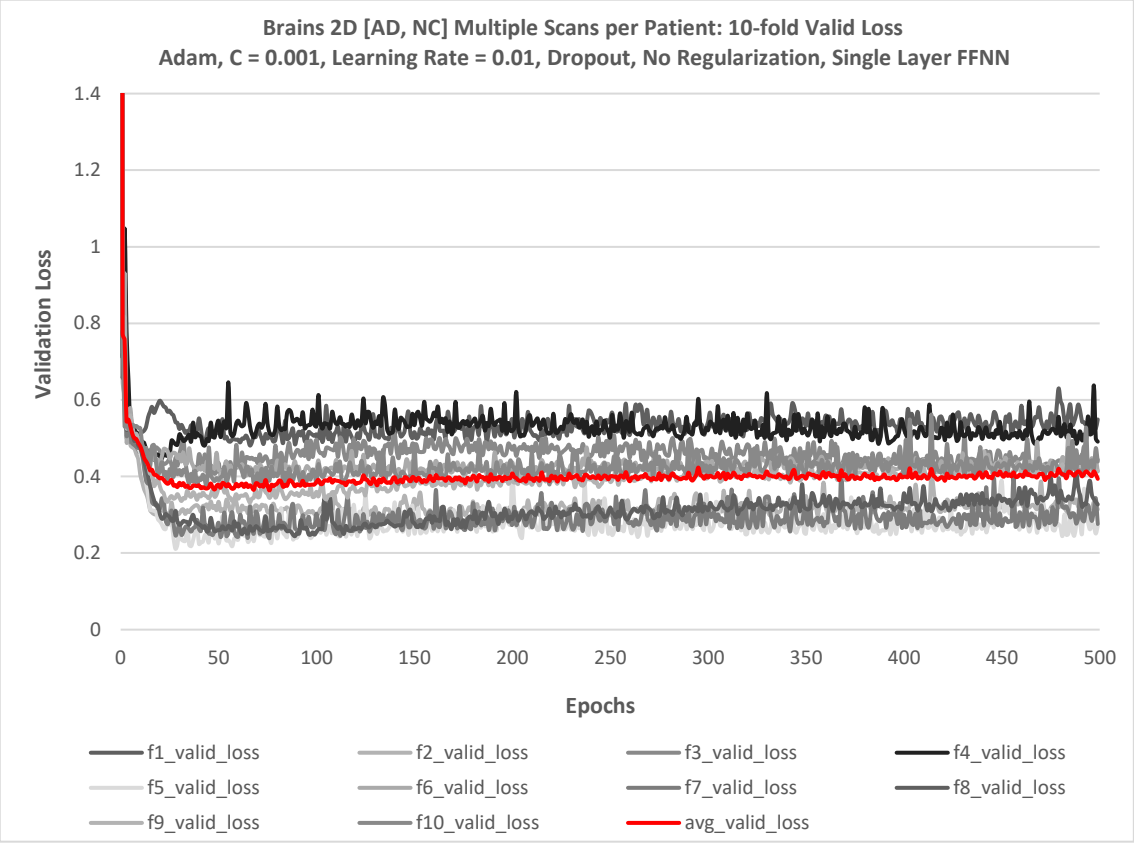


Figure H.25. Caption in the title.

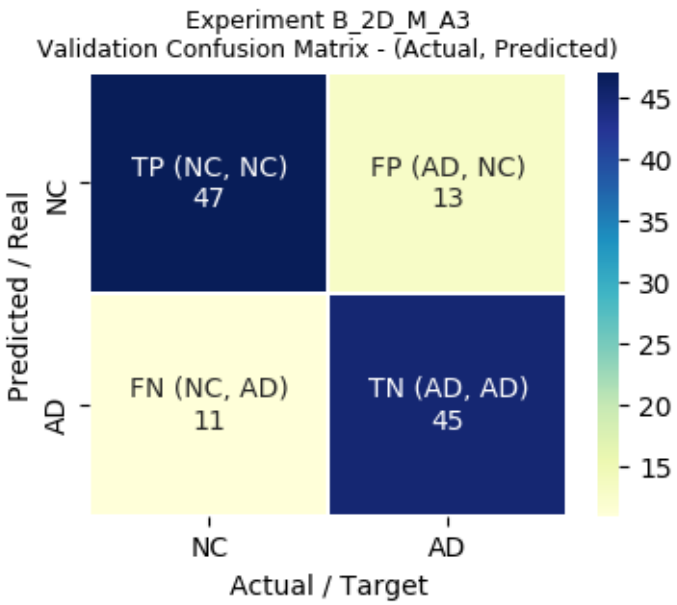


Figure H.26. Caption in the title.

H.1.3.4 Testing

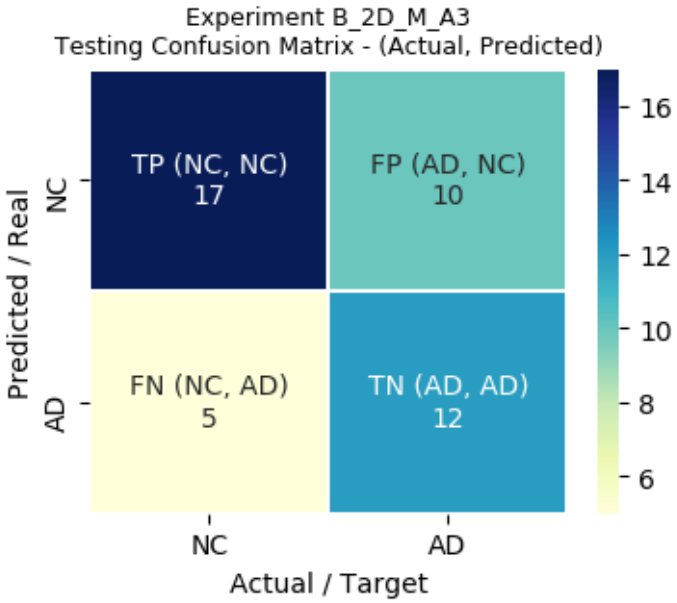


Figure H.27. Caption in the title.

H.2 CNN 4 Layers with NewtonCG

H.2.1 Experiment B_2D_M_N1: C = 0.01, GNsize = 5

H.2.1.1 General

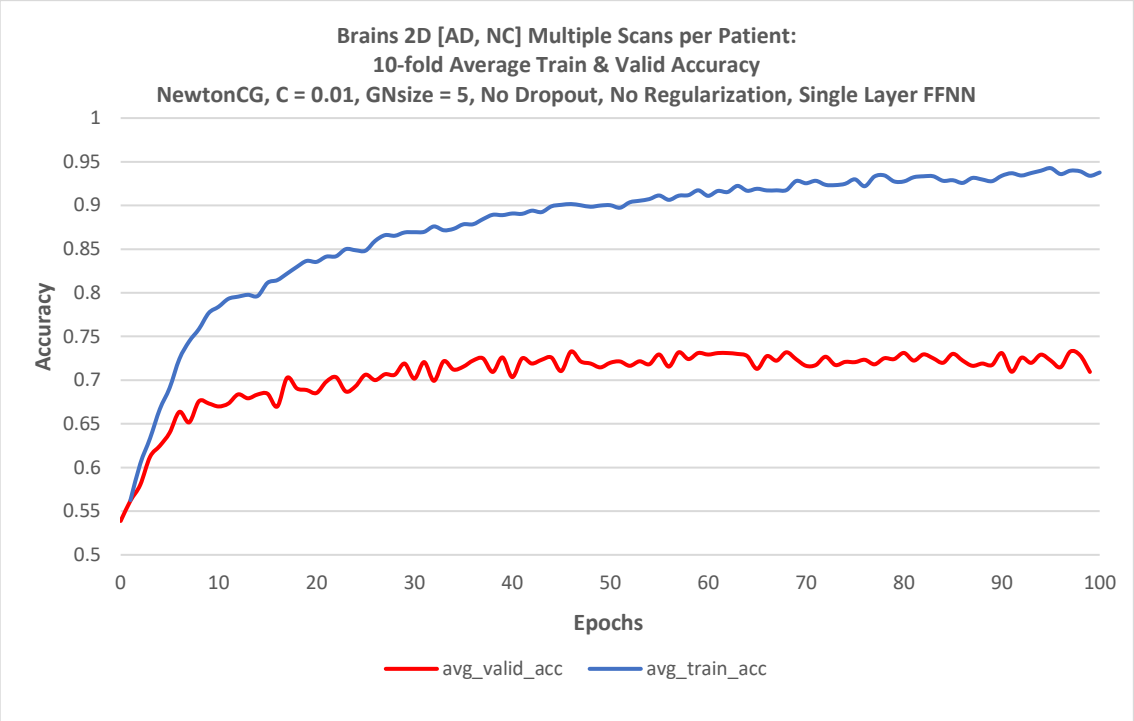


Figure H.28. Caption in the title.

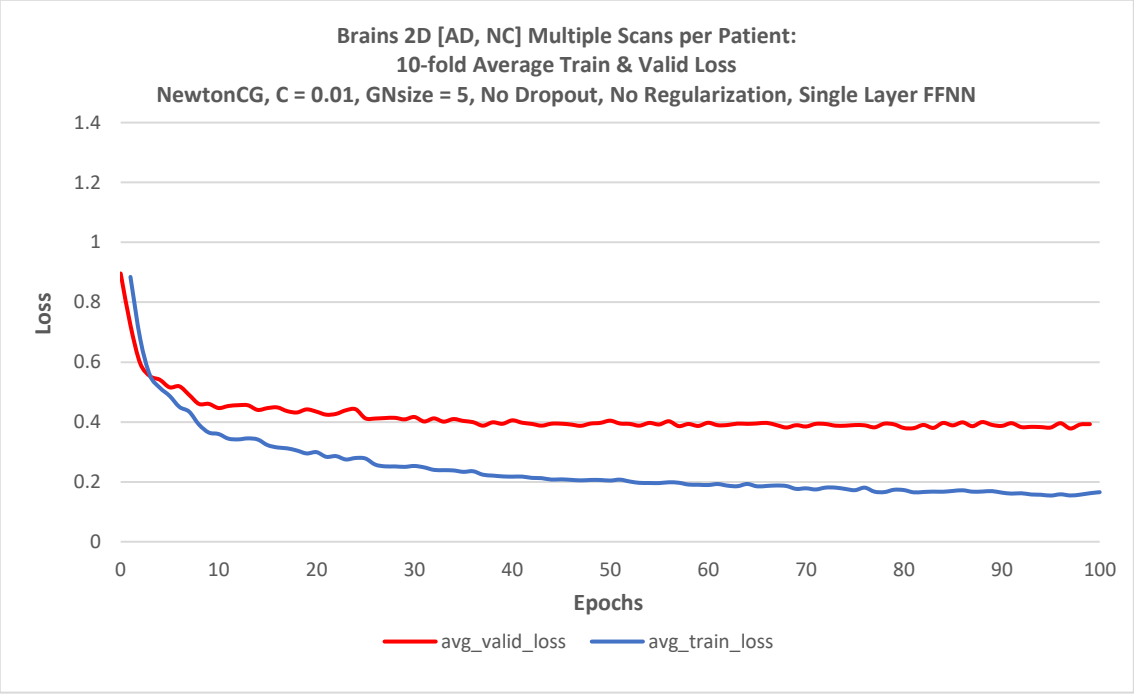


Figure H.29. Caption in the title.

H.2.1.2 Training

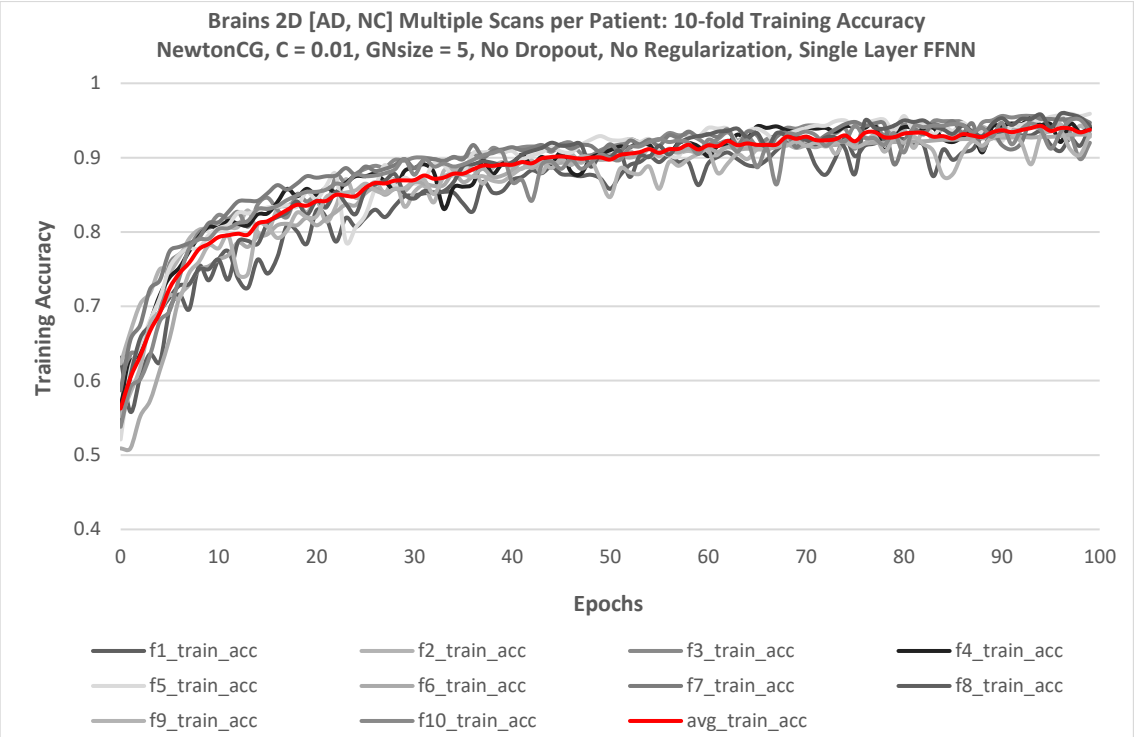


Figure H.30. Caption in the title.

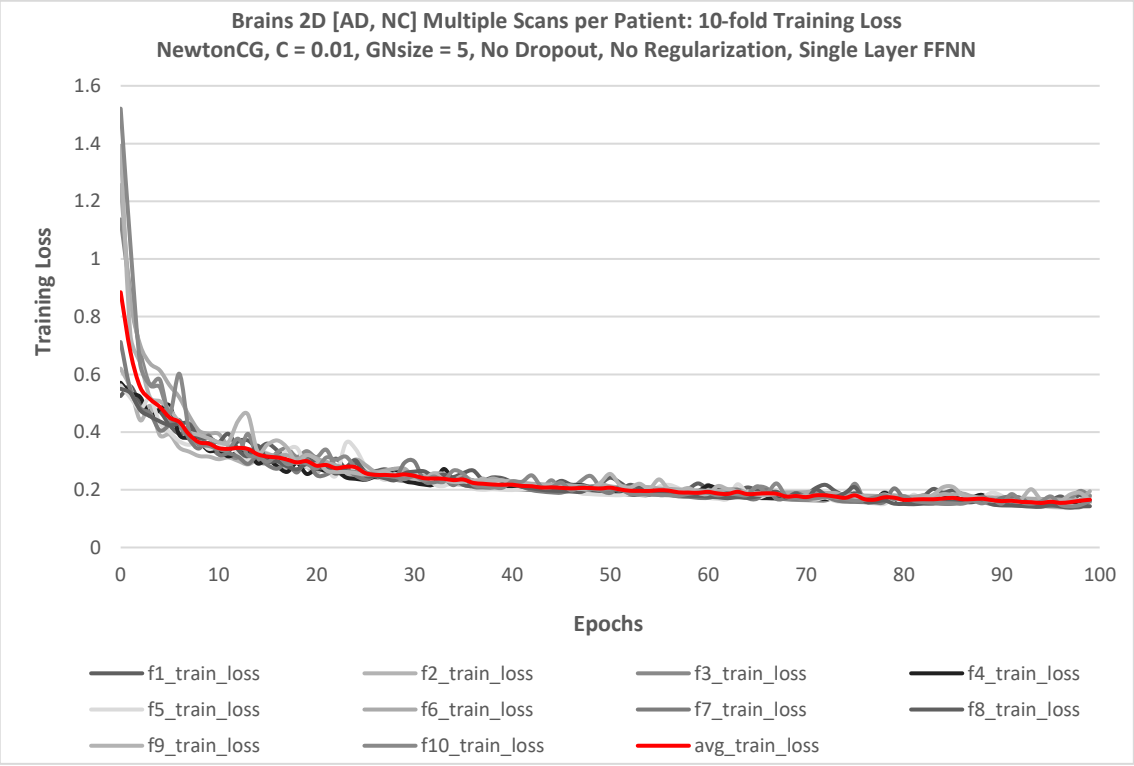


Figure H.31. Caption in the title.

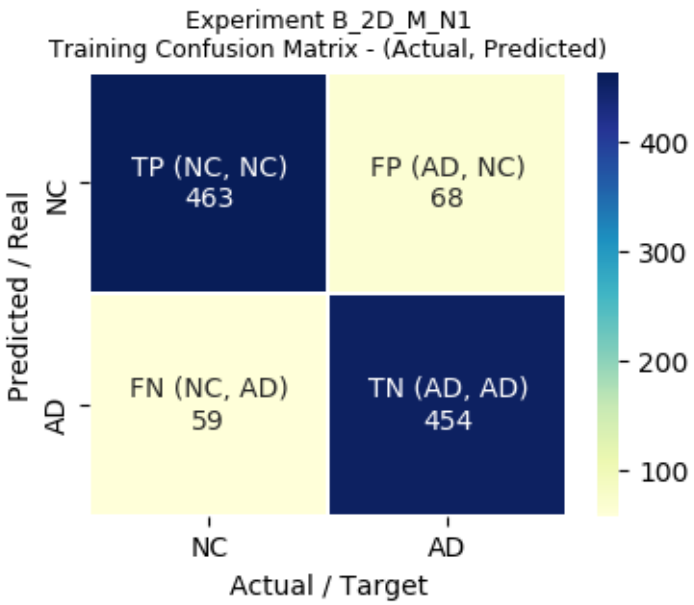


Figure H.32. Caption in the title.

H.2.1.3 Validation

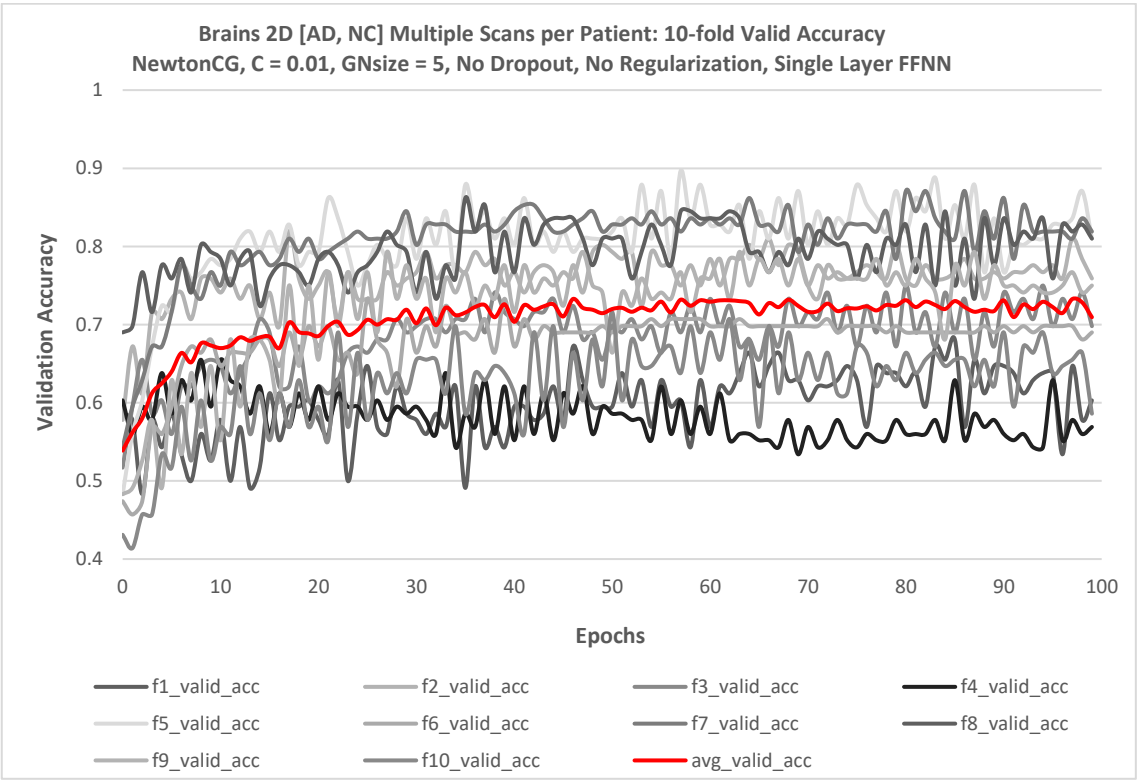


Figure H.33. Caption in the title.

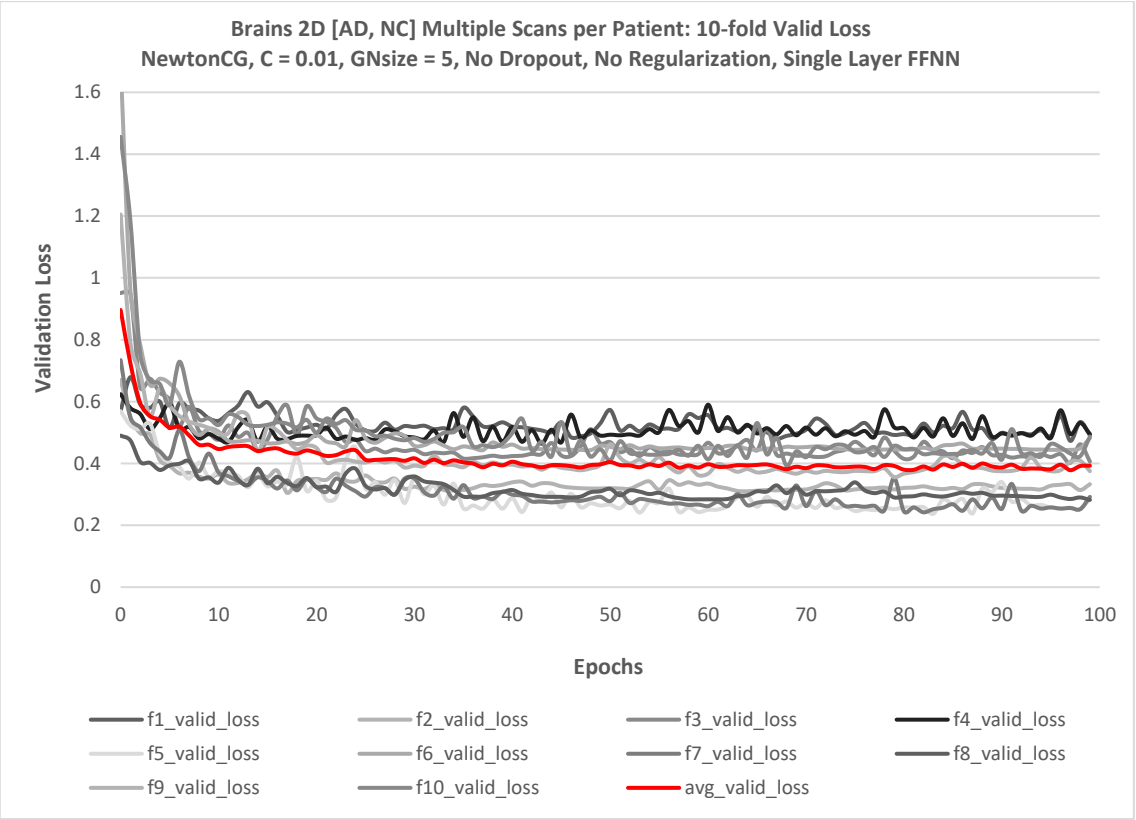


Figure H.34. Caption in the title.

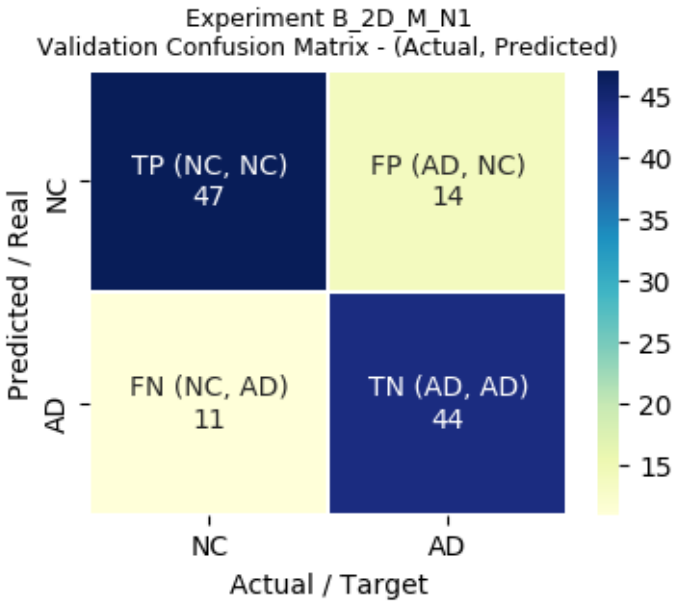


Figure H.35. Caption in the title.

H.2.1.4 Testing

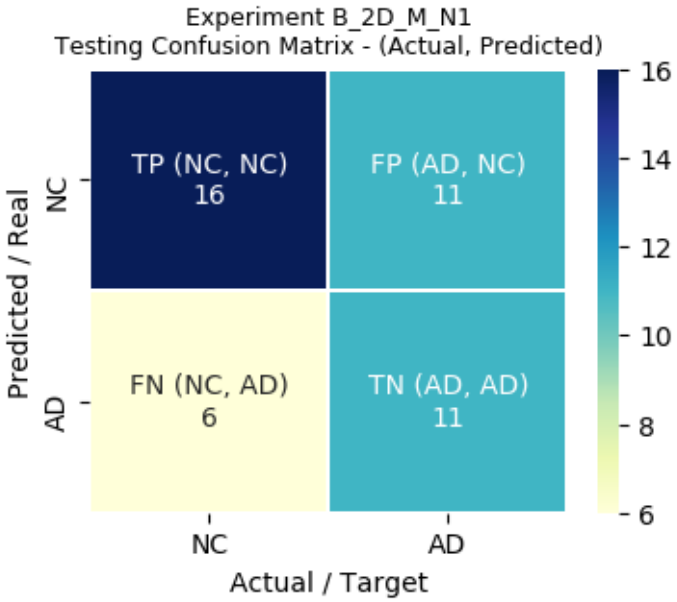


Figure H.36. Caption in the title.

H.2.2 Experiment B_2D_M_N2: C = 0.1, GNsize = 5

H.2.2.1 General

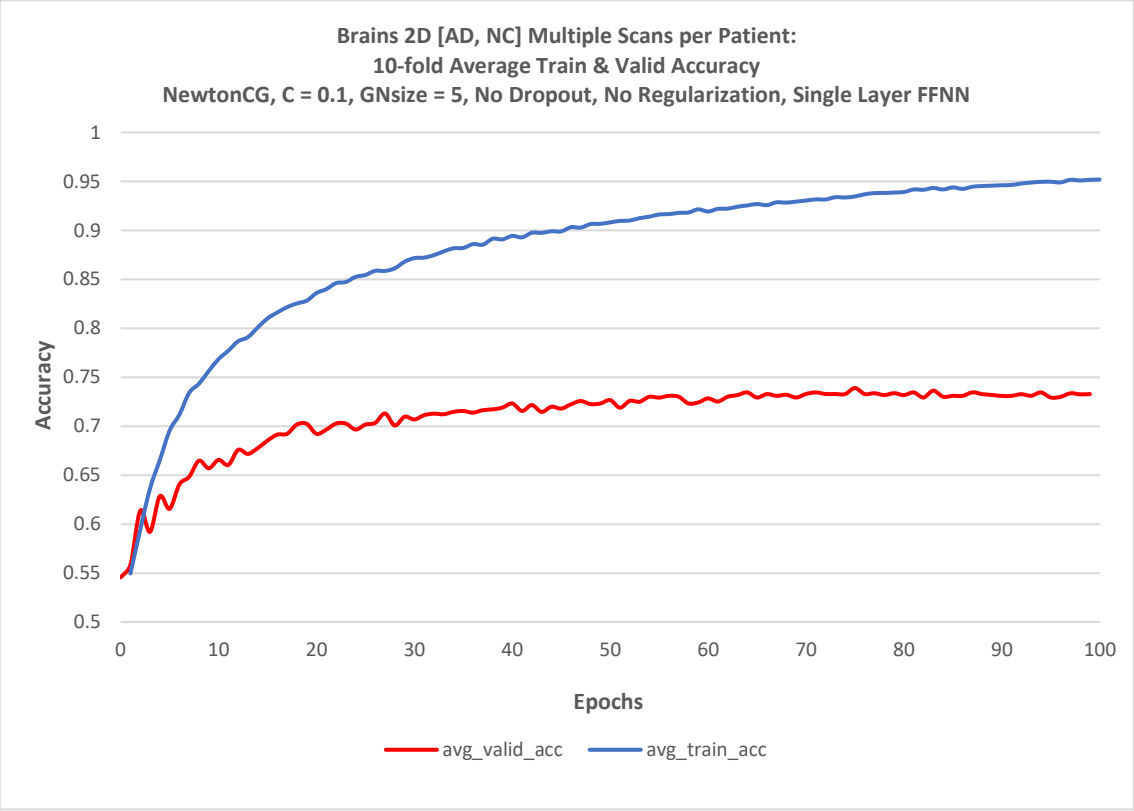


Figure H.37. Caption in the title.

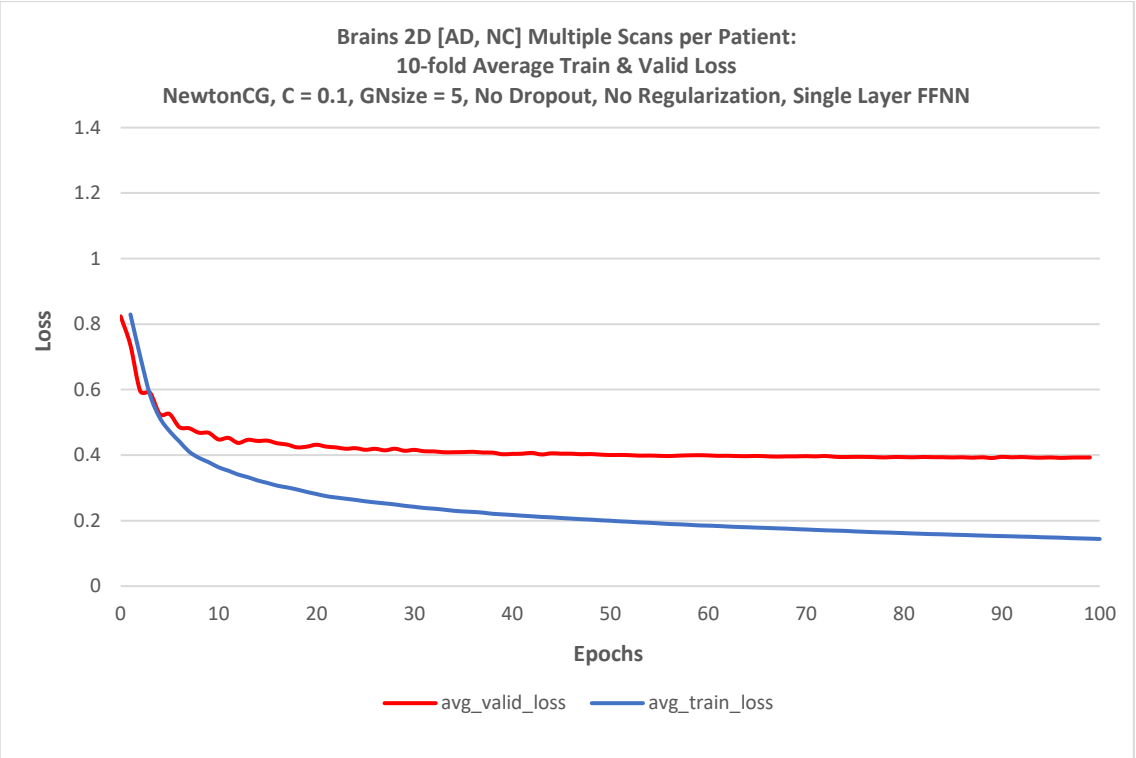


Figure H.38. Caption in the title.

H.2.2.2 Training

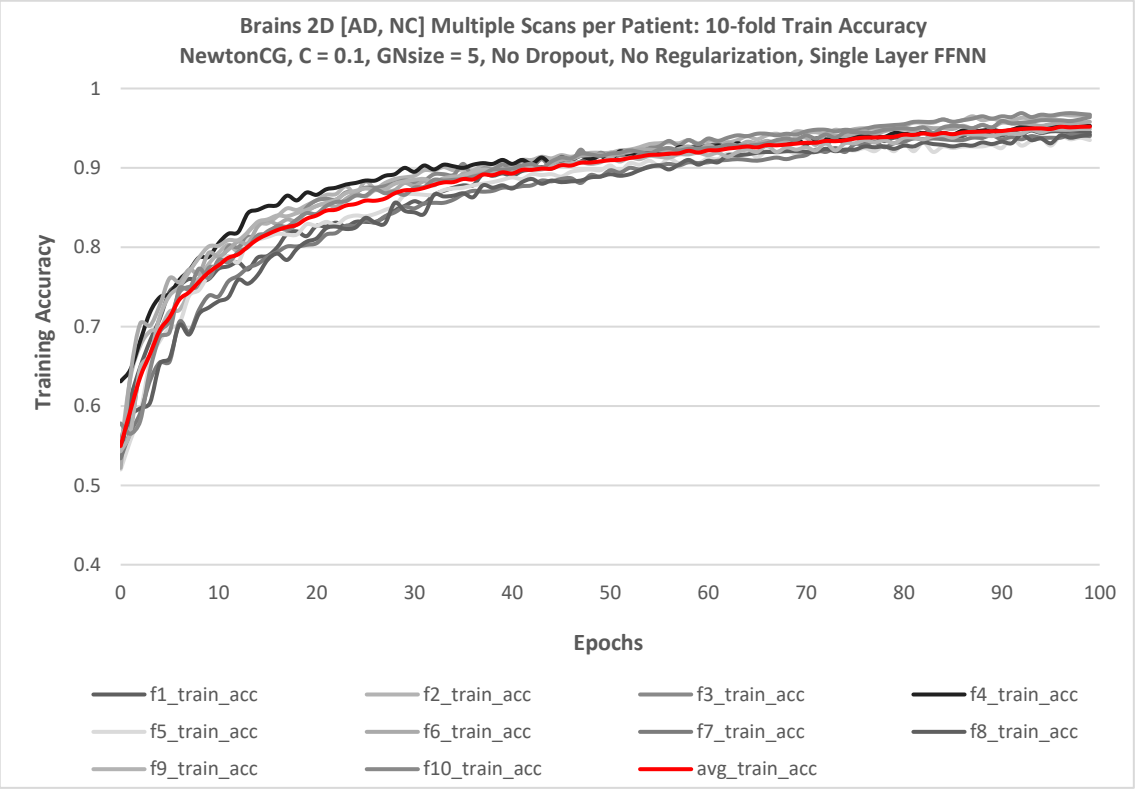


Figure H.39. Caption in the title.

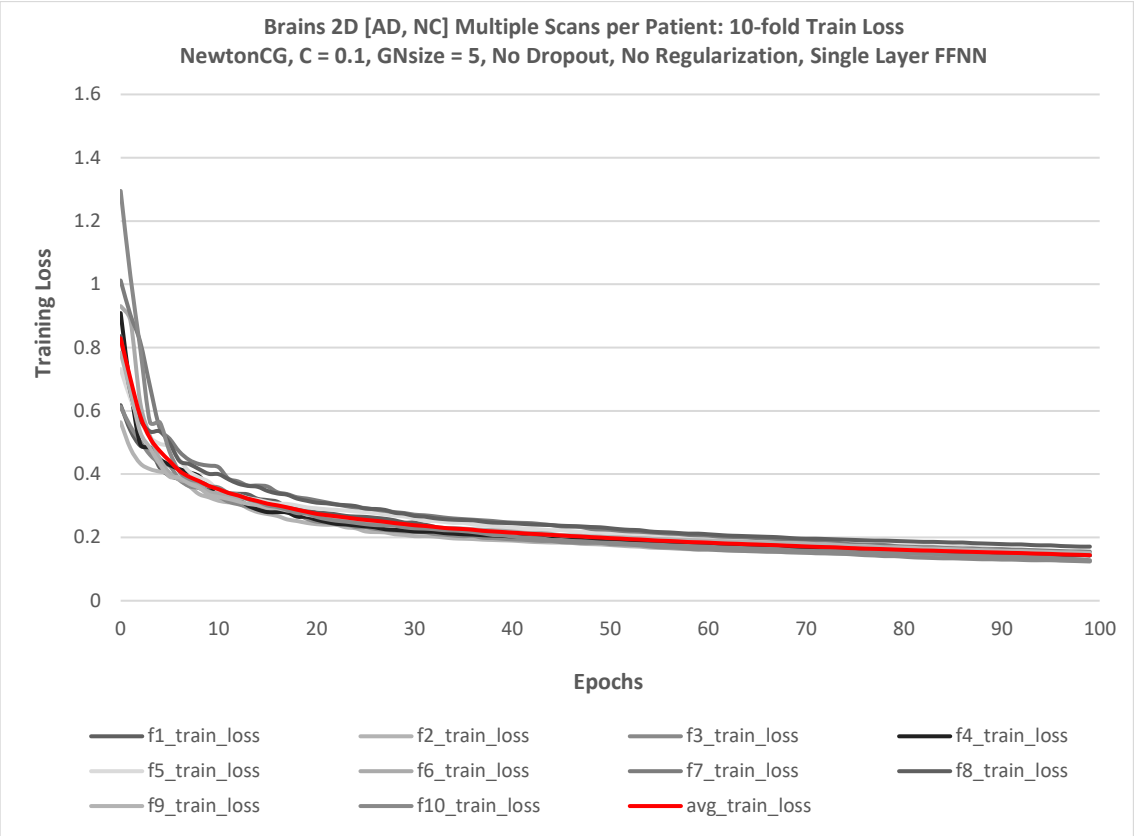


Figure H.40. Caption in the title.

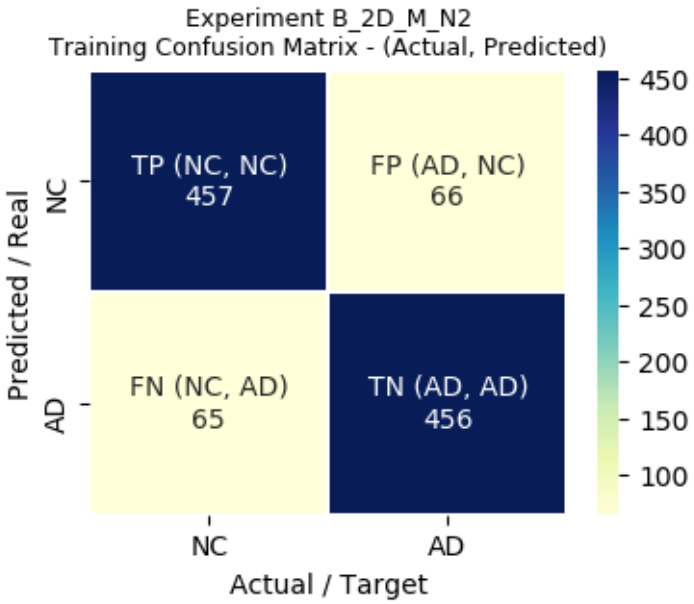


Figure H.41. Caption in the title.

H.2.2.3 Validation

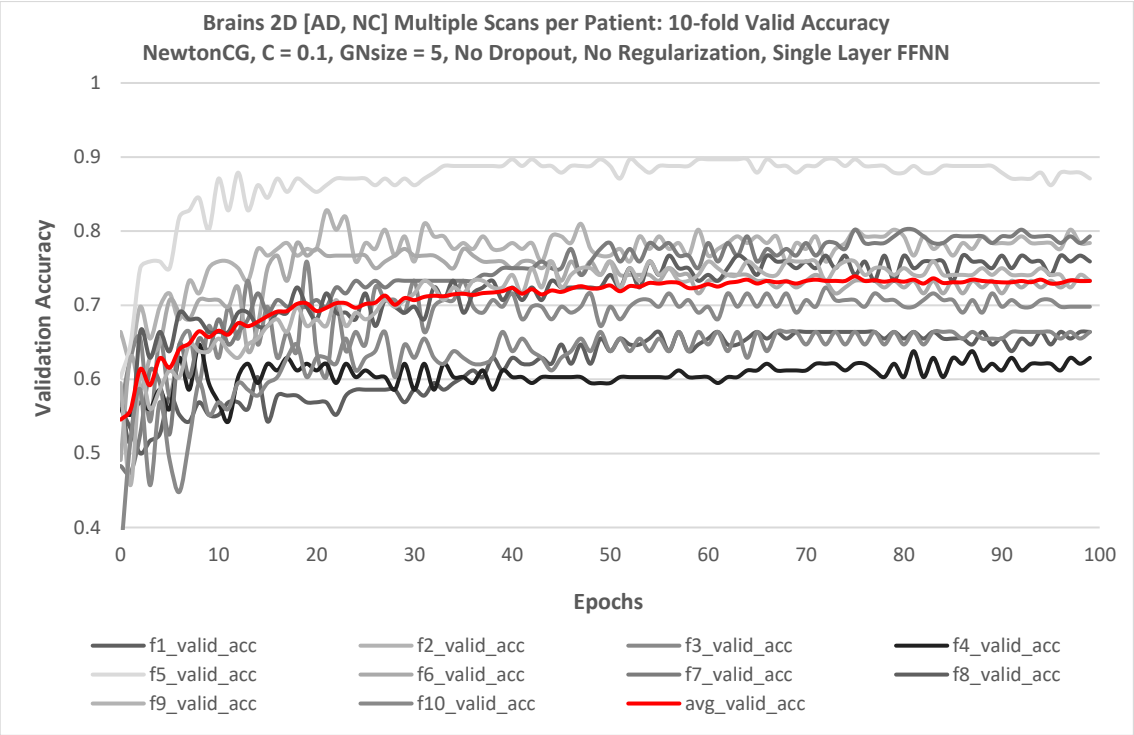


Figure H.42. Caption in the title.

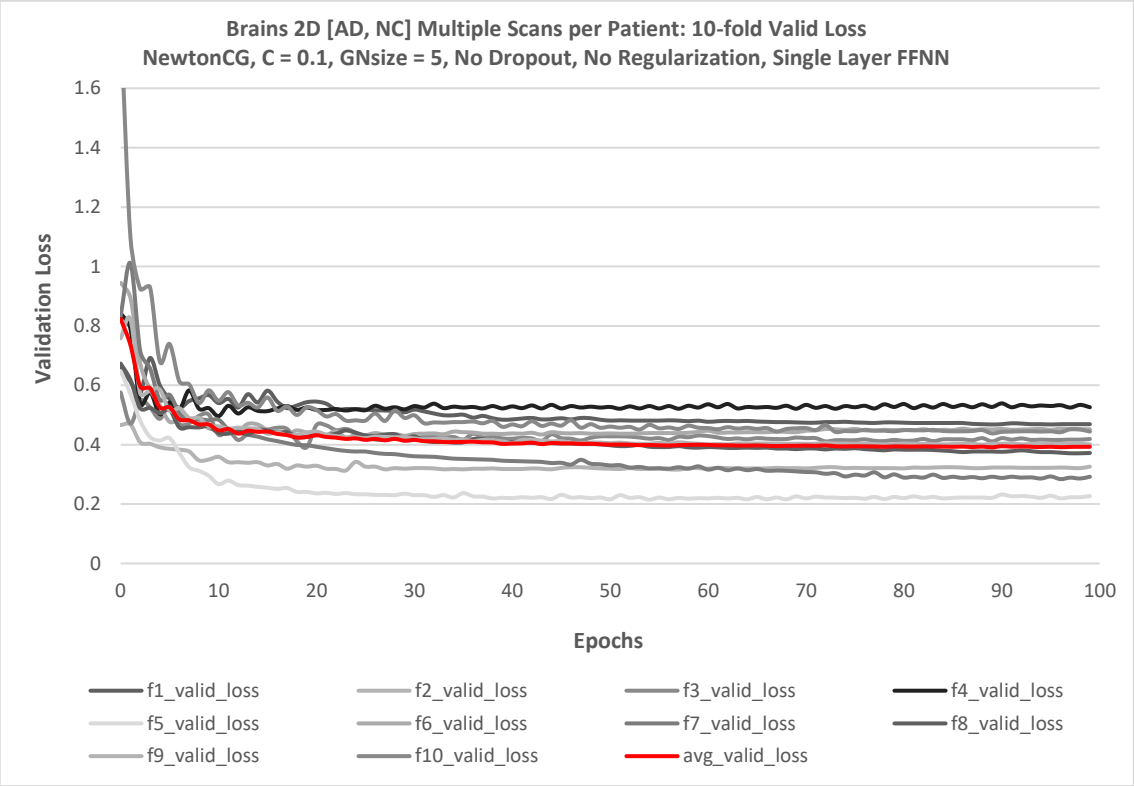


Figure H.43. Caption in the title.

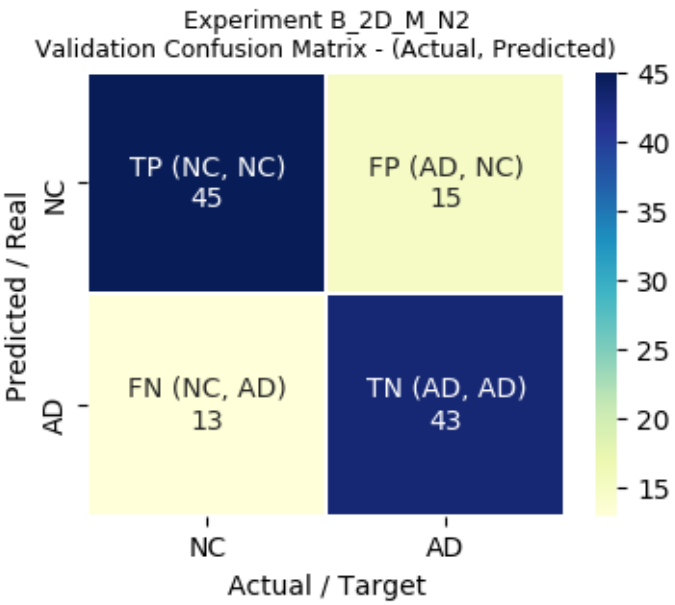


Figure H.44. Caption in the title.

H.2.2.4 Testing

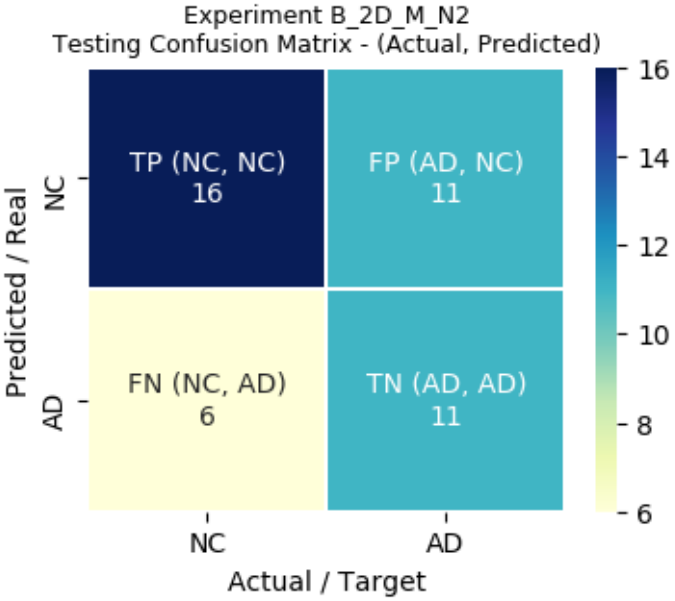


Figure H.45. Caption in the title.

H.2.3 Experiment B_2D_M_N3: C = 1, GNsize = 5

H.2.3.1 General

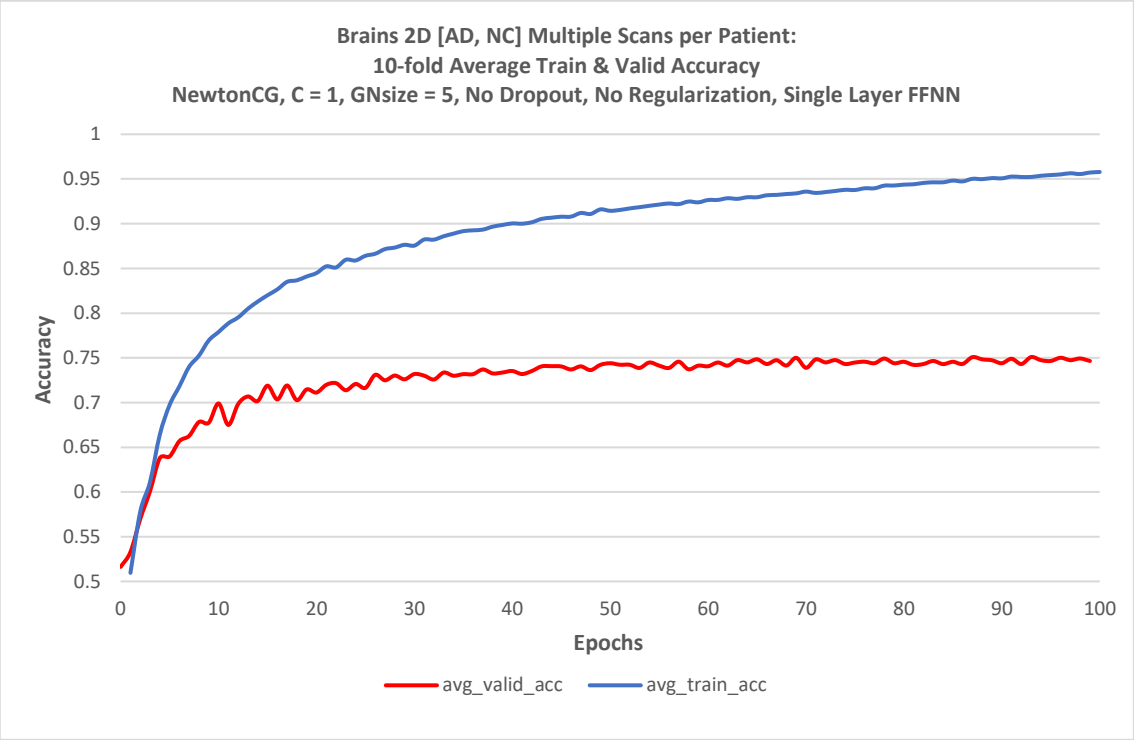


Figure H.46. Caption in the title.

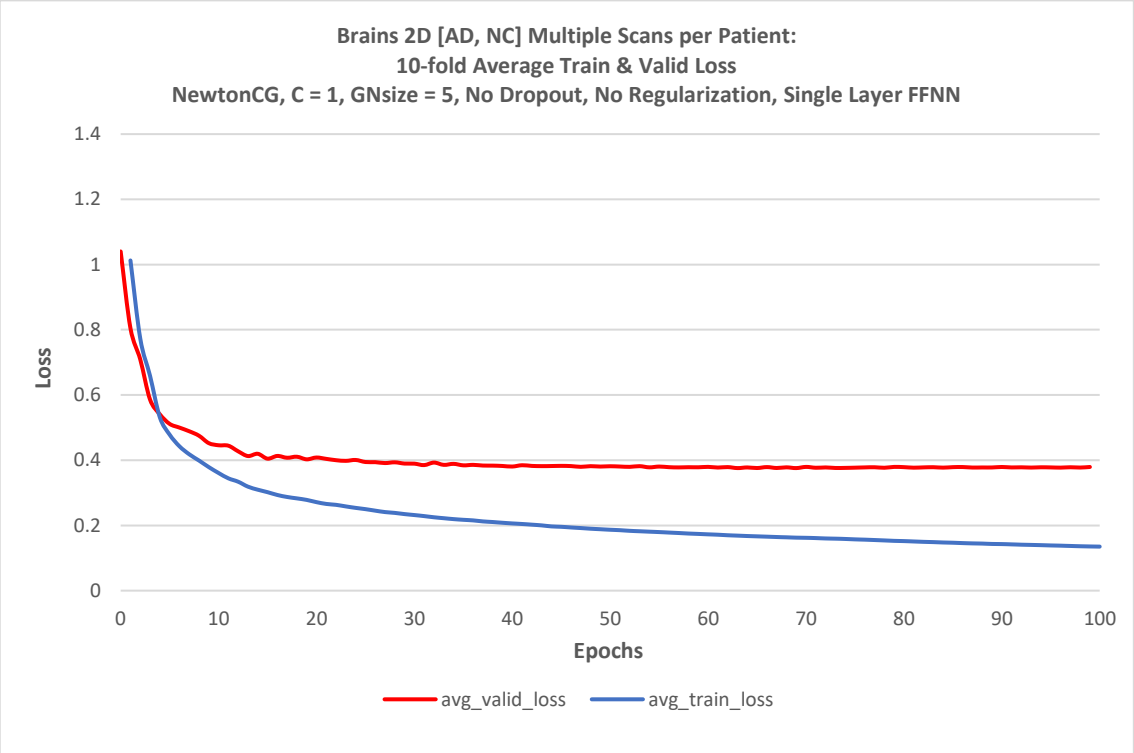


Figure H.47. Caption in the title.

H.2.3.2 Training

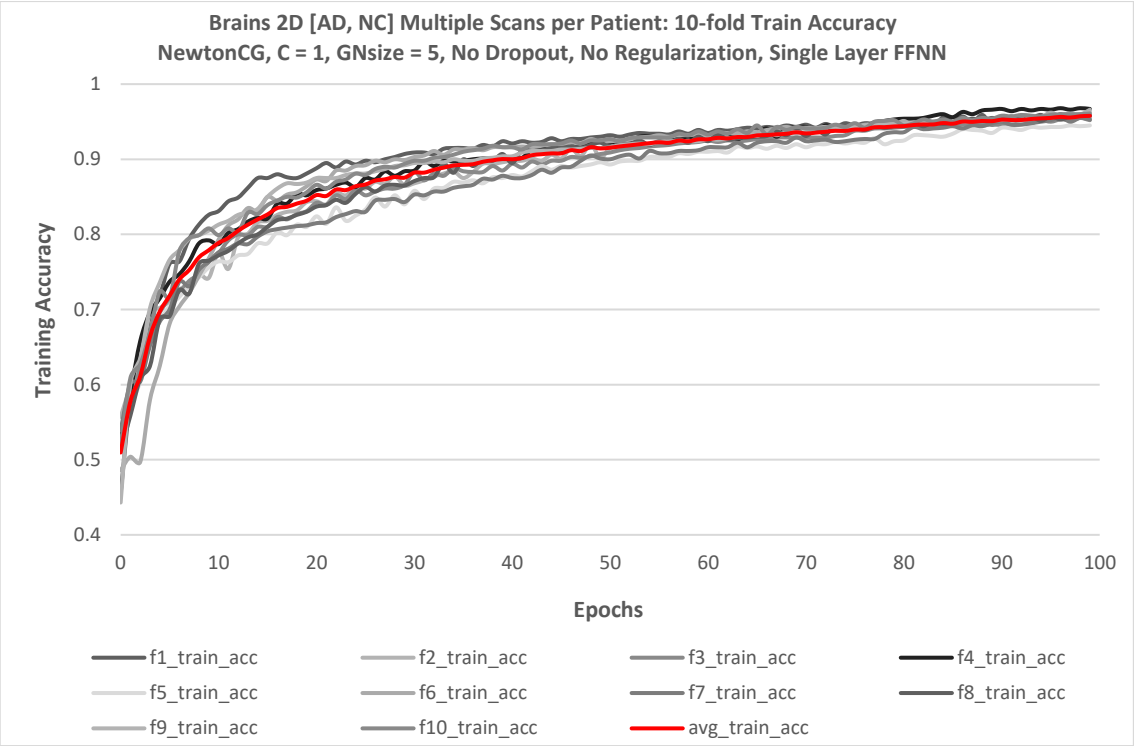


Figure H.48. Caption in the title.

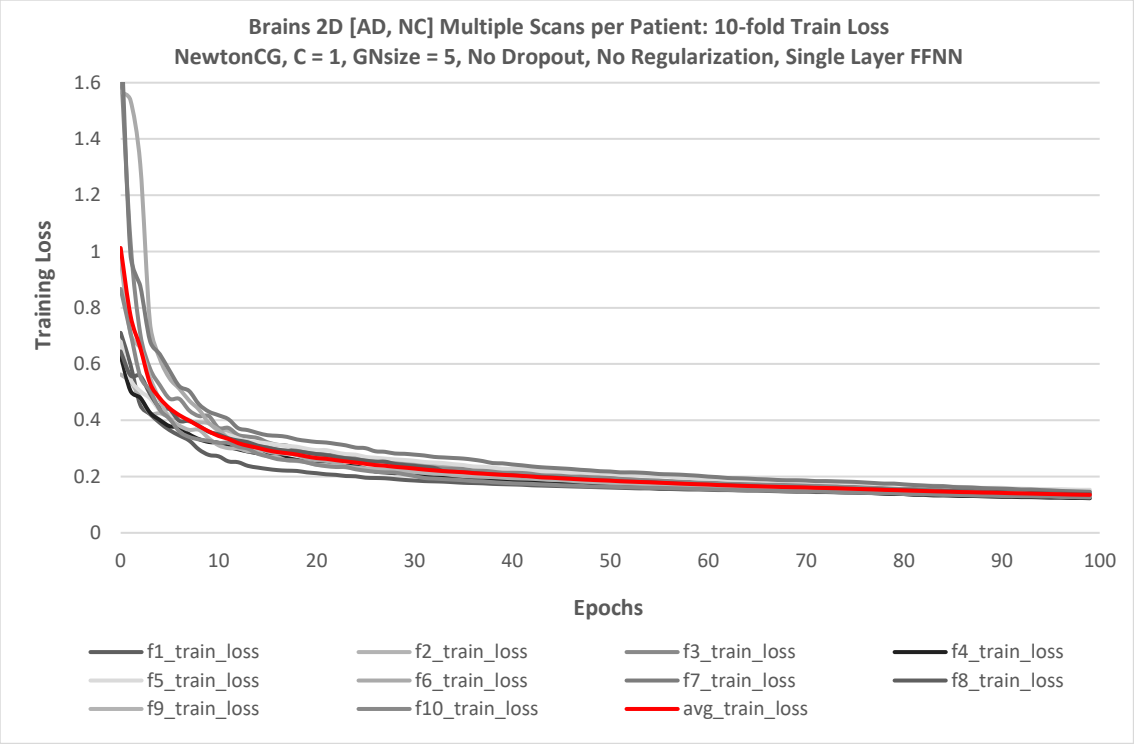


Figure H.49. Caption in the title.

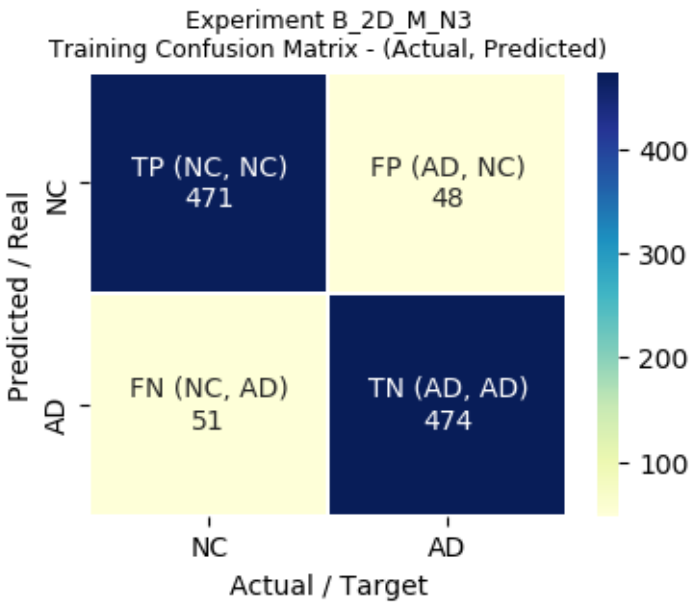


Figure H.50. Caption in the title.

H.2.3.3 Validation

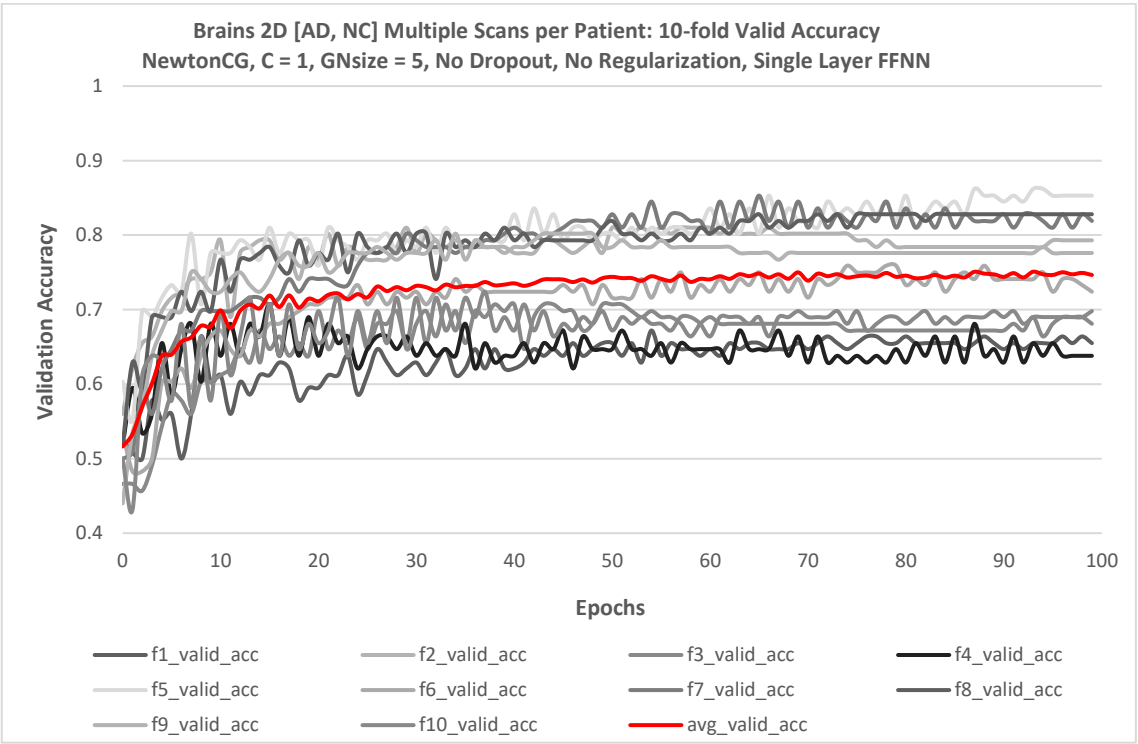


Figure H.51. Caption in the title.

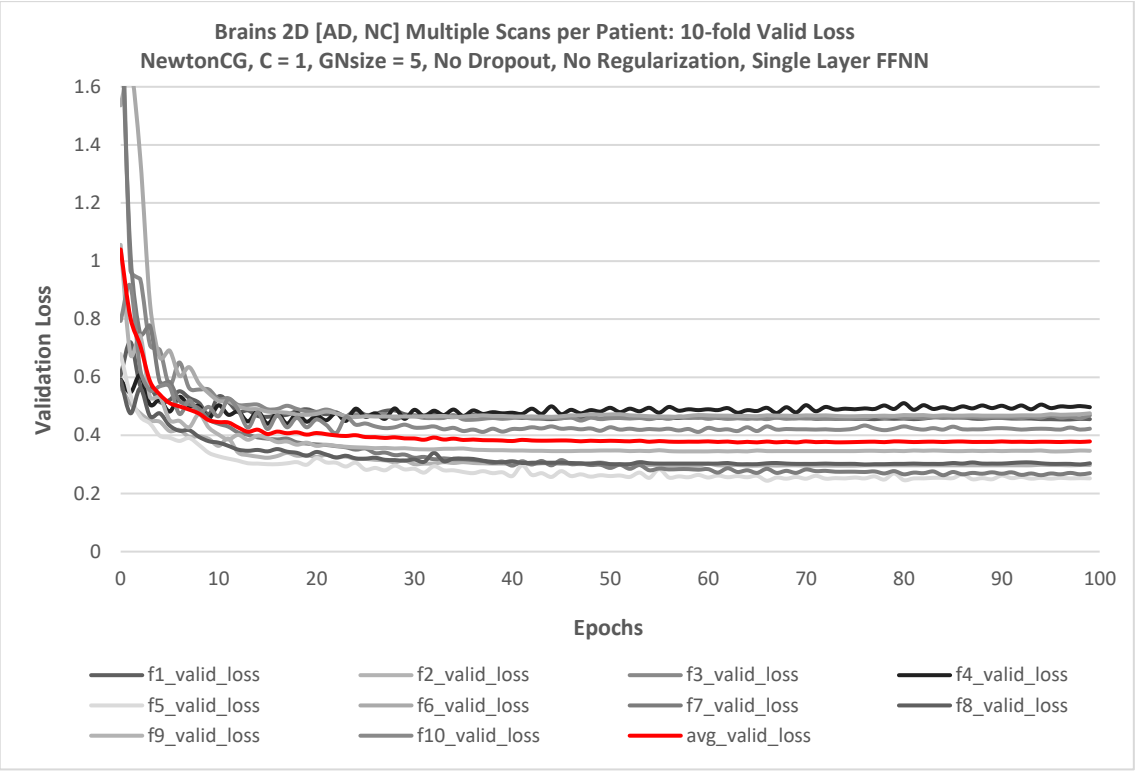


Figure H.52. Caption in the title.

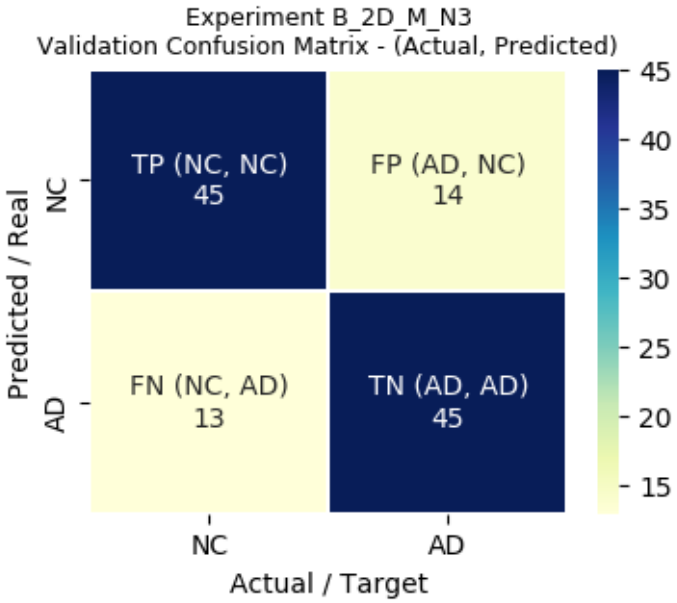


Figure H.53. Caption in the title.

H.2.3.4 Testing

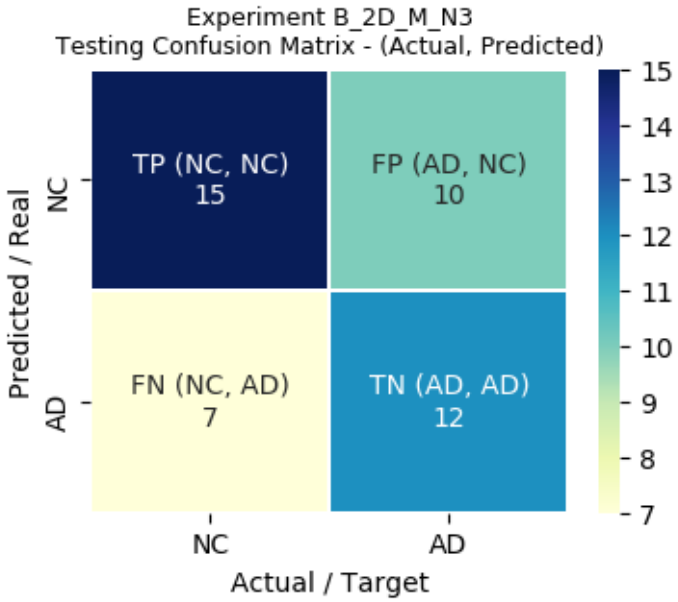


Figure H.54. Caption in the title.

H.2.4 Experiment B_2D_M_N4: C = 10, GNsize = 5

H.2.4.1 General

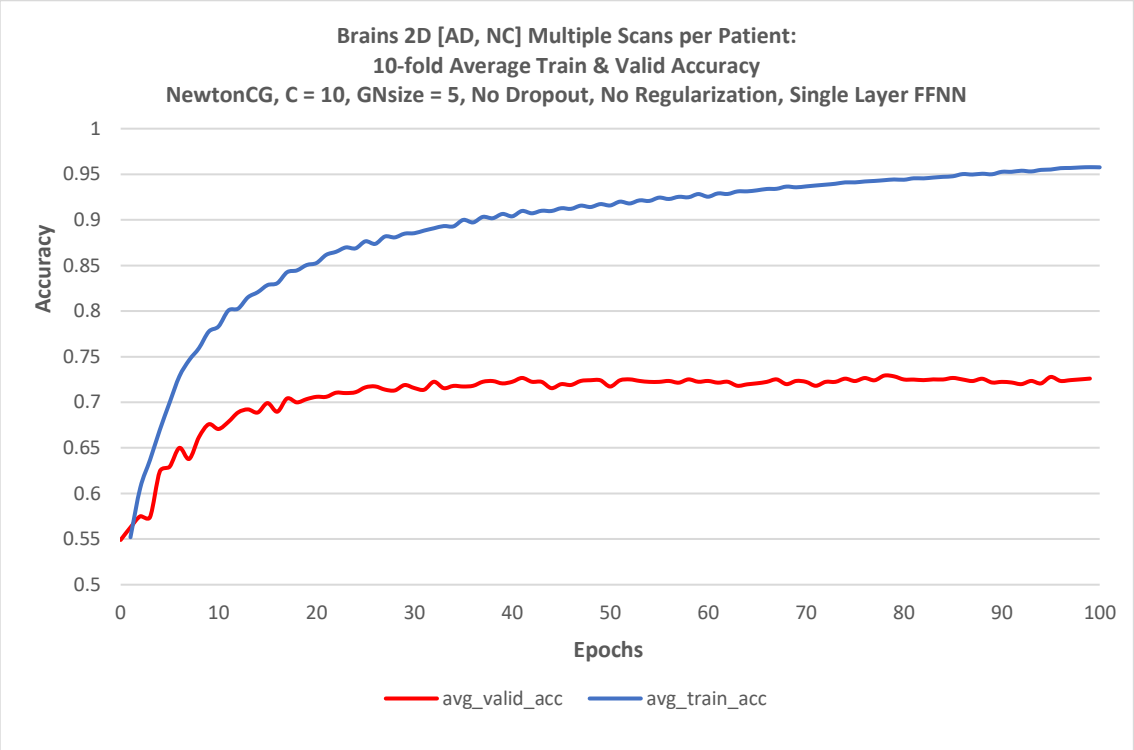


Figure H.55. Caption in the title.

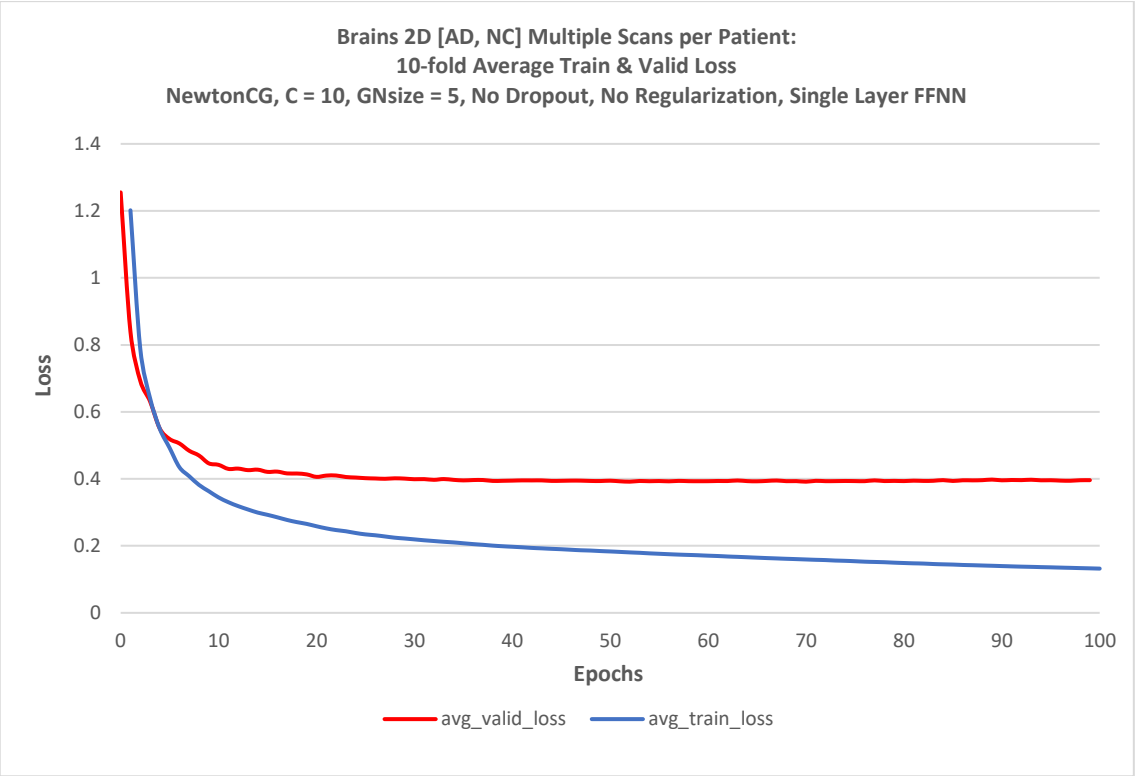


Figure H.56. Caption in the title.

H.2.4.2 Training

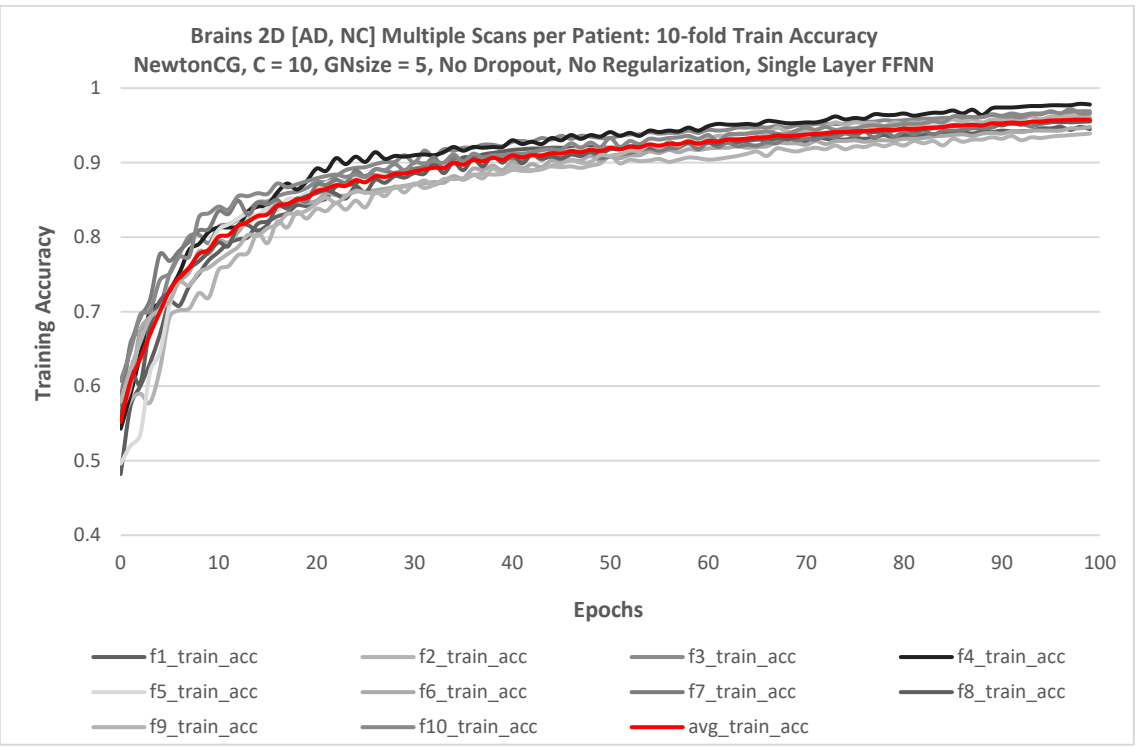


Figure H.57. Caption in the title.

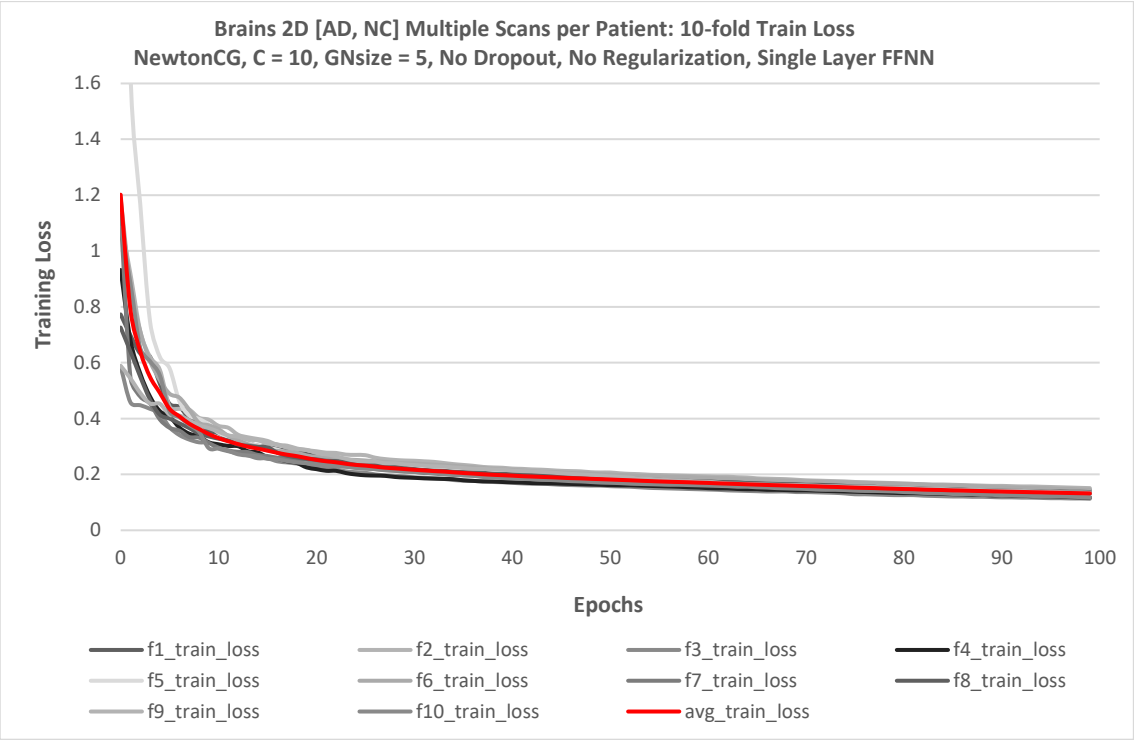


Figure H.58. Caption in the title.

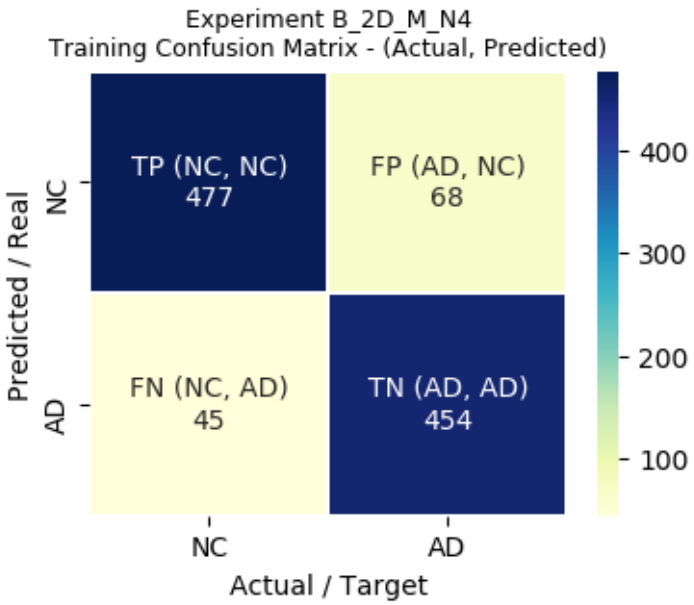


Figure H.59. Caption in the title.

H.2.4.3 Validation

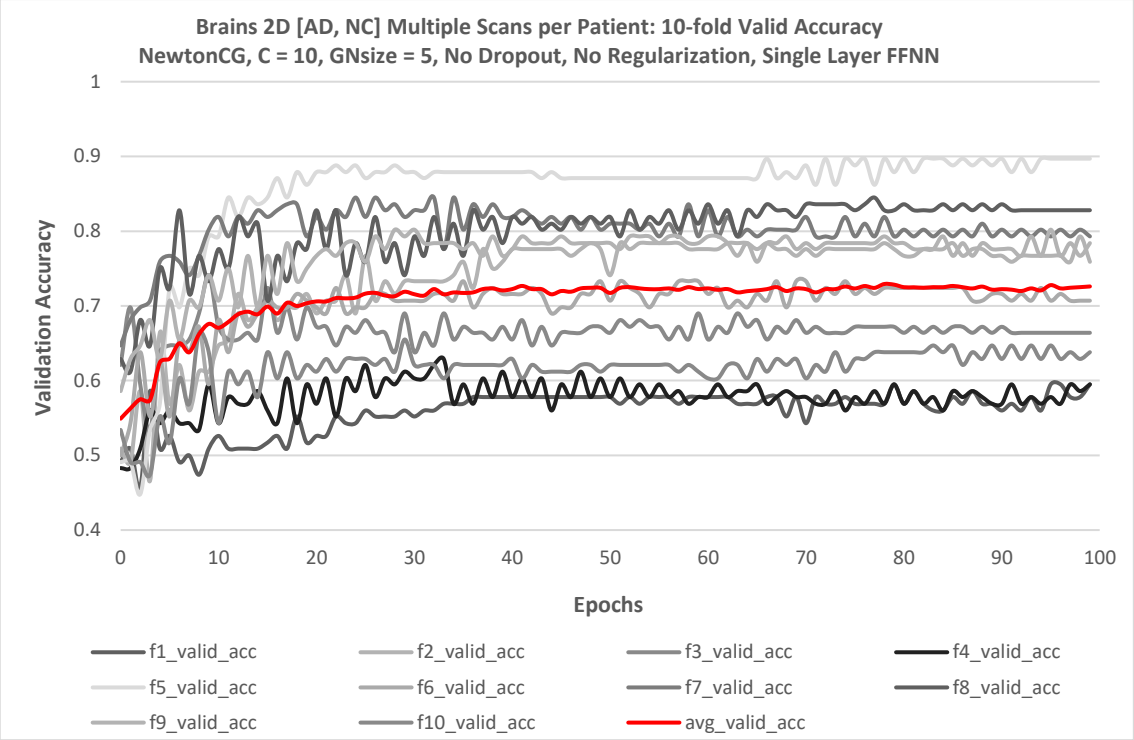


Figure H.60. Caption in the title.

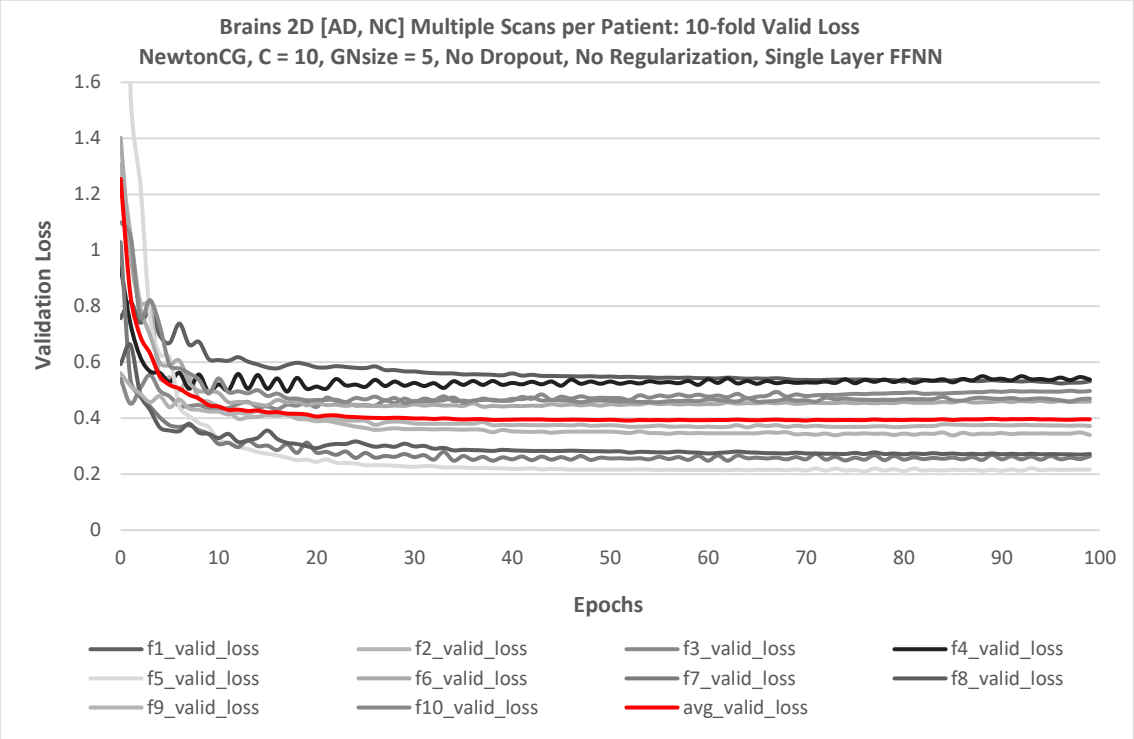


Figure H.61. Caption in the title.

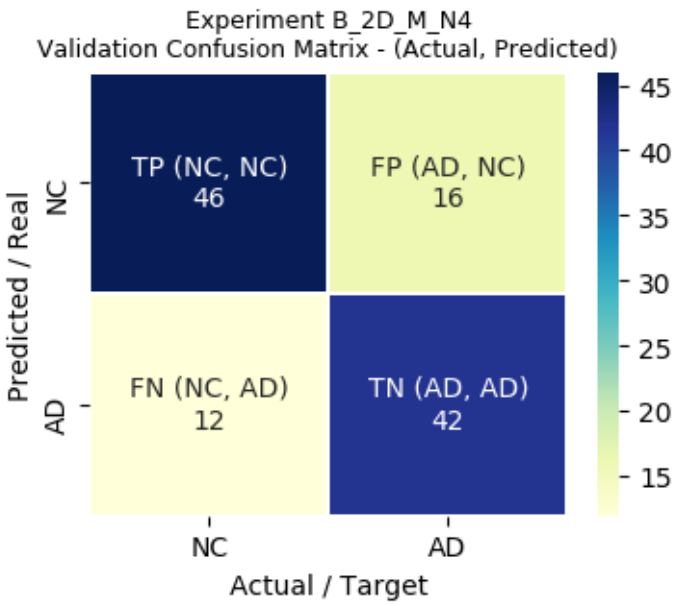


Figure H.62. Caption in the title.

H.2.4.4 Testing

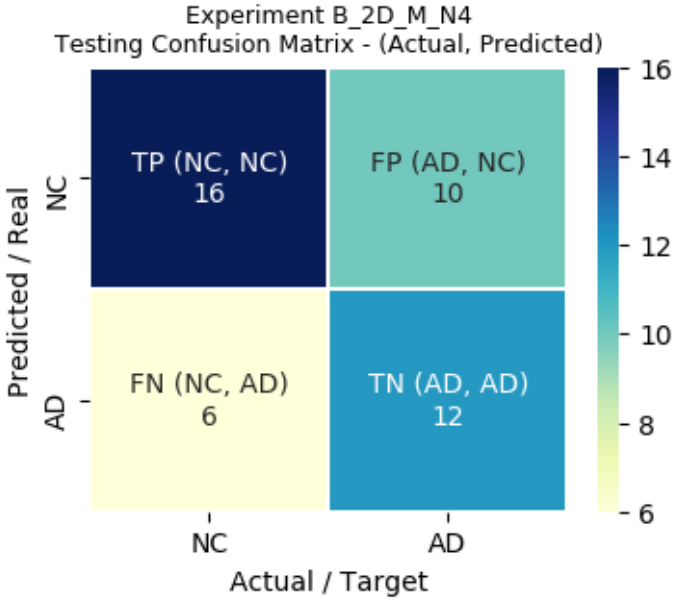


Figure H.63. Caption in the title.

H.2.5 Experiment B_2D_M_N5: C = 1, GNsize = 5, Two Layers FFNN

H.2.5.1 General

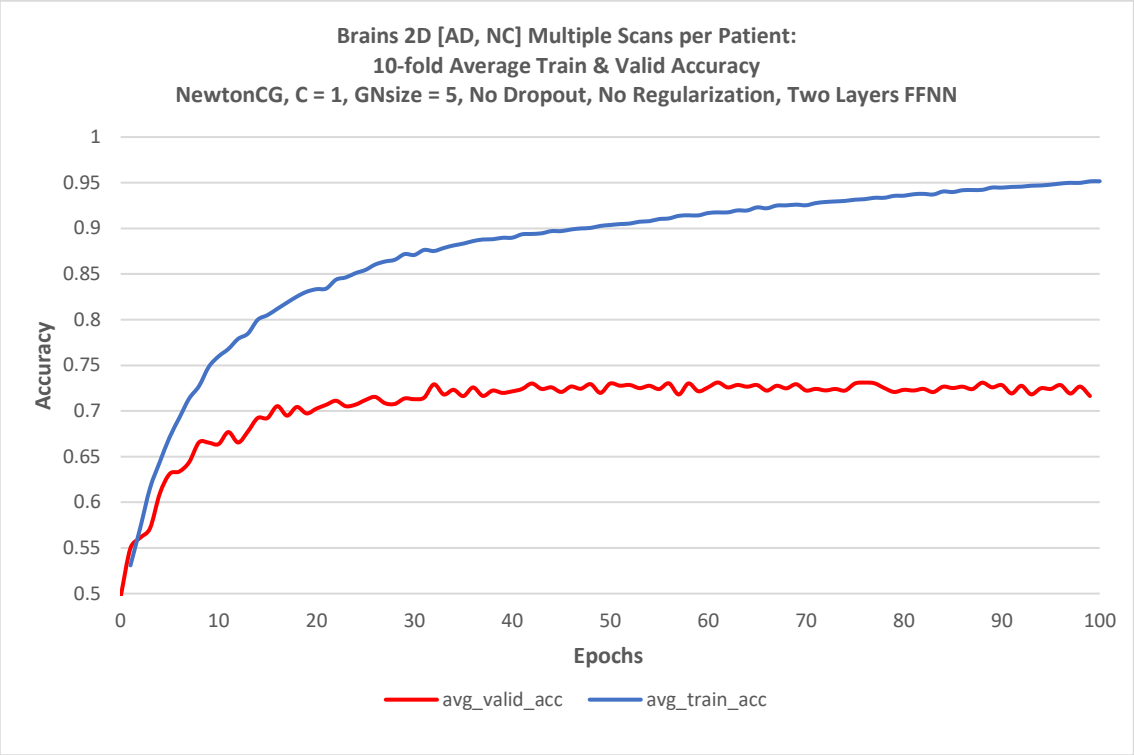


Figure H.64. Caption in the title.

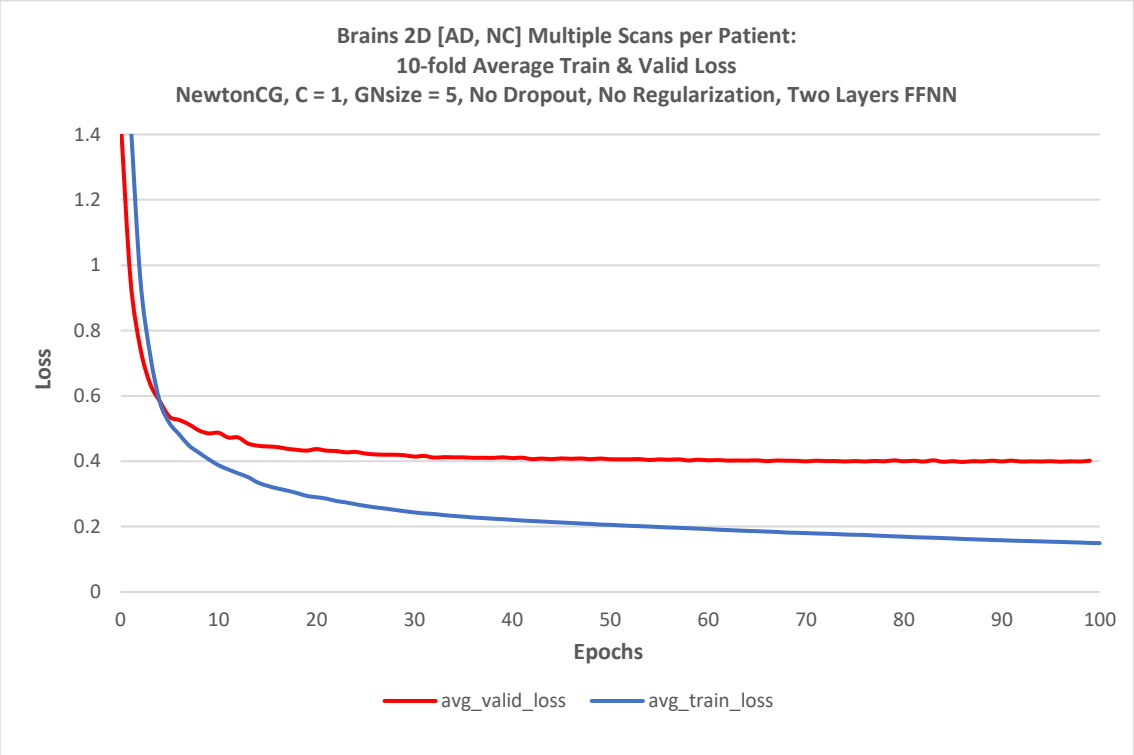


Figure H.65. Caption in the title.

H.2.5.2 Training

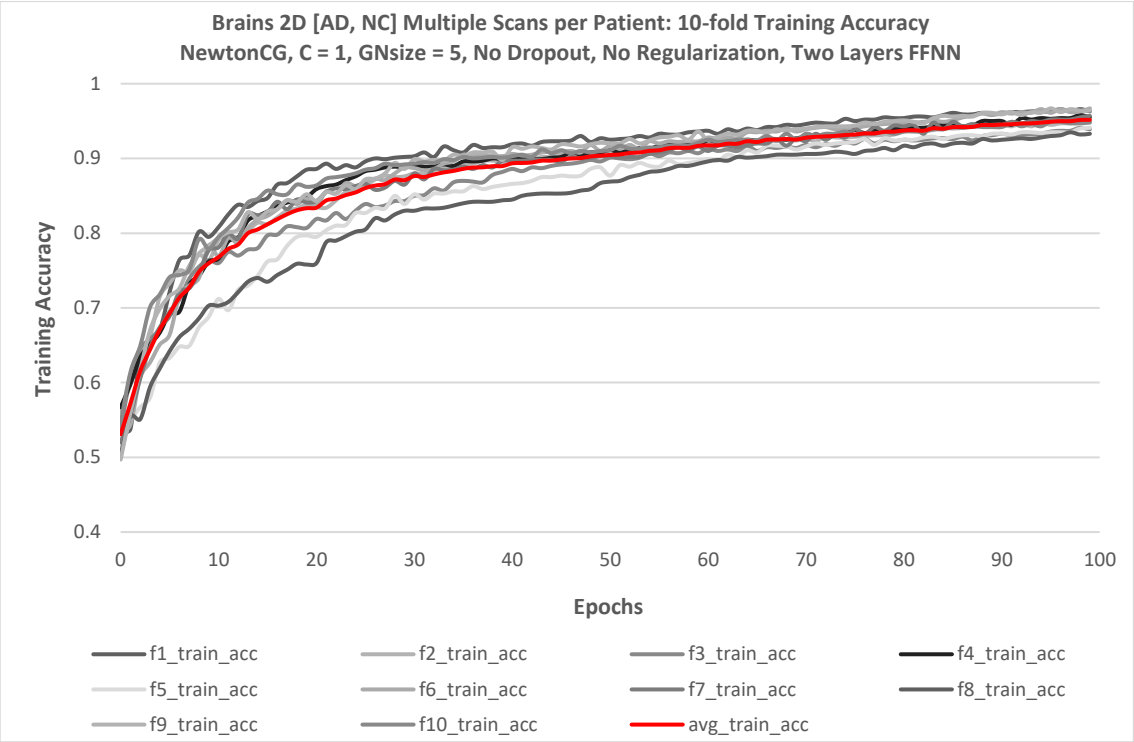


Figure H.66. Caption in the title.

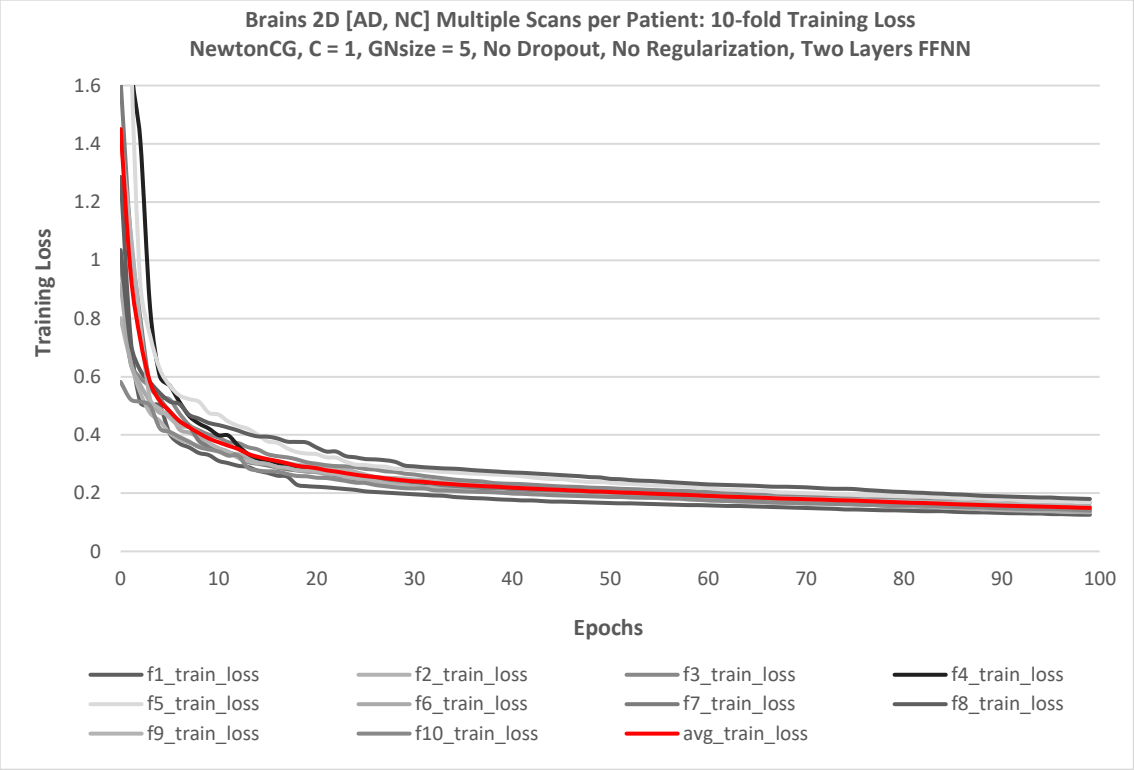


Figure H.67. Caption in the title.

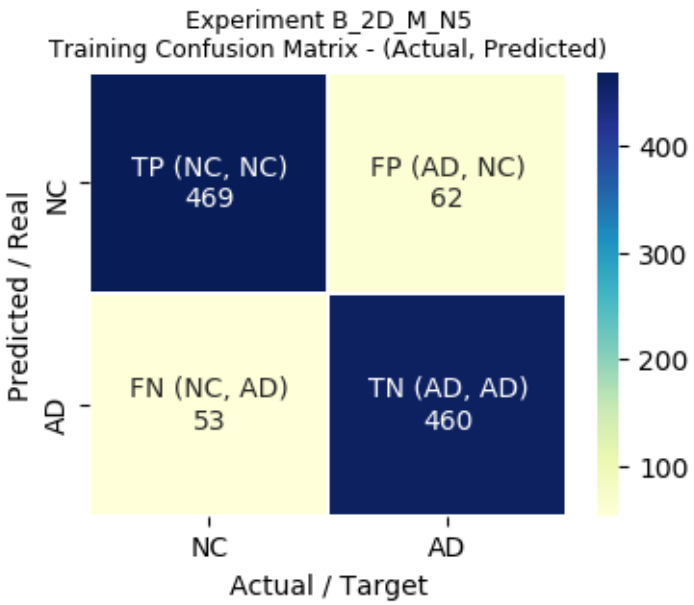


Figure H.68. Caption in the title.

H.2.5.3 Validation

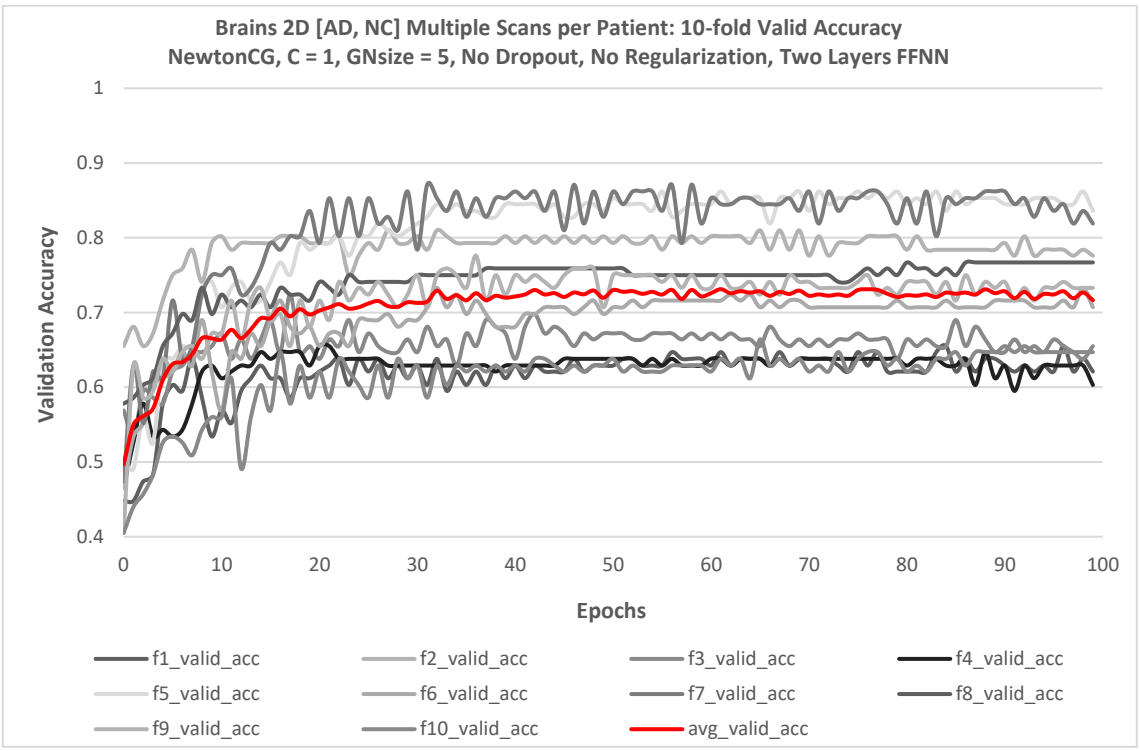


Figure H.69. Caption in the title.

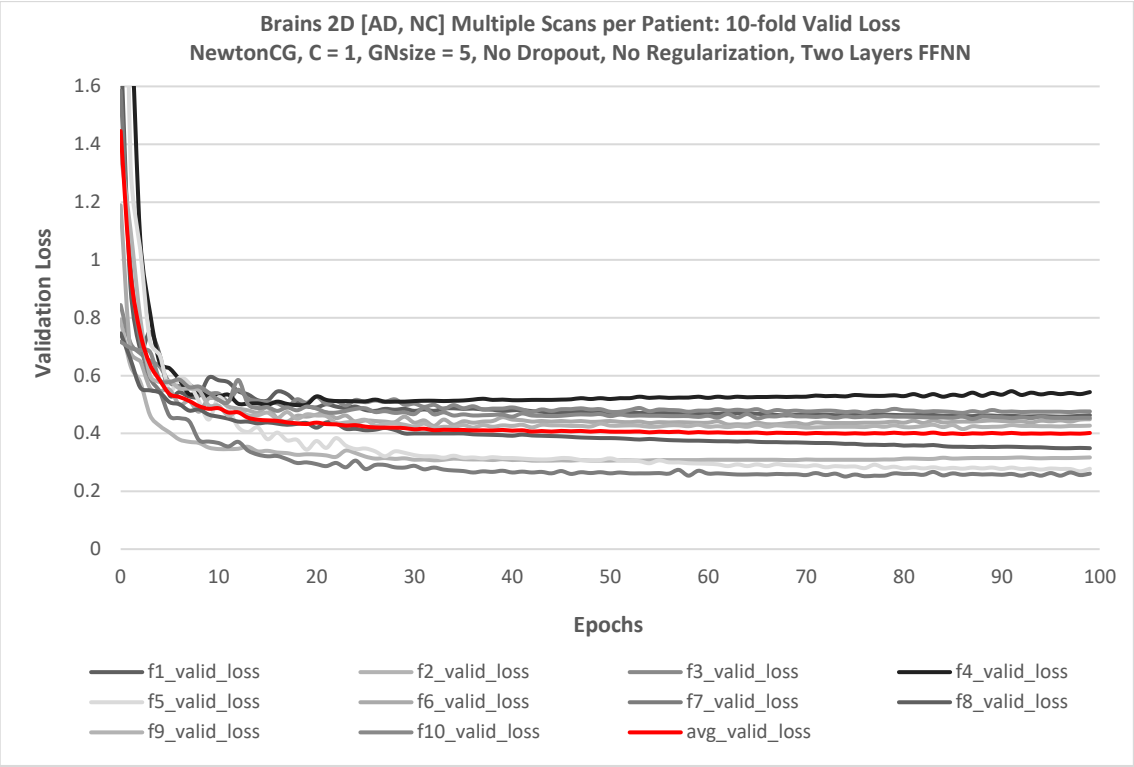


Figure H.70. Caption in the title.

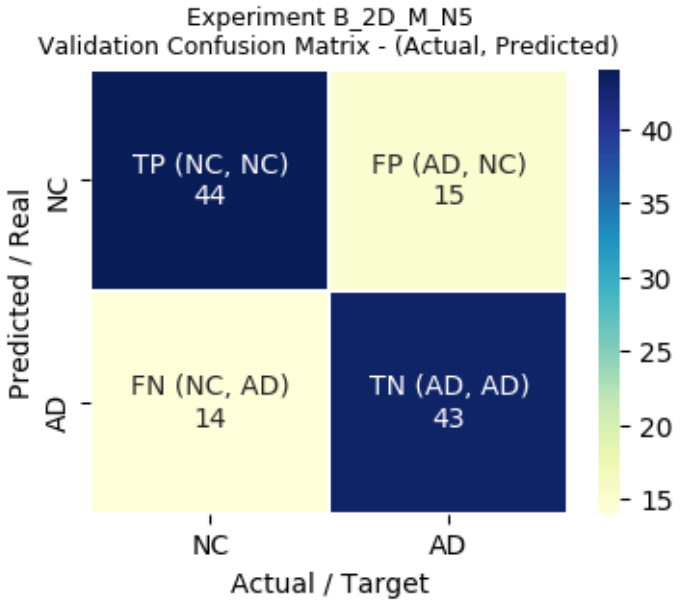


Figure H.71. Caption in the title.

H.2.5.4 Testing

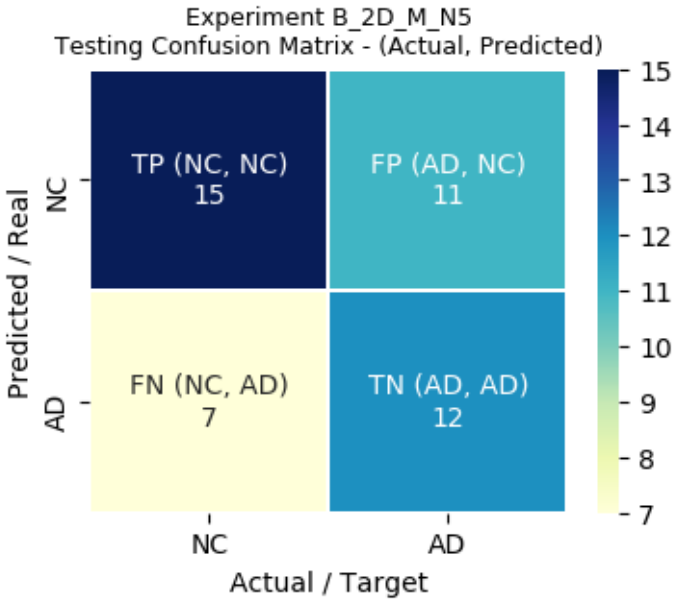


Figure H.72. Caption in the title.

H.2.6 Experiment B_2D_M_N6: C = 1, GNsize = 5, L2 Regularization

H.2.6.1 General

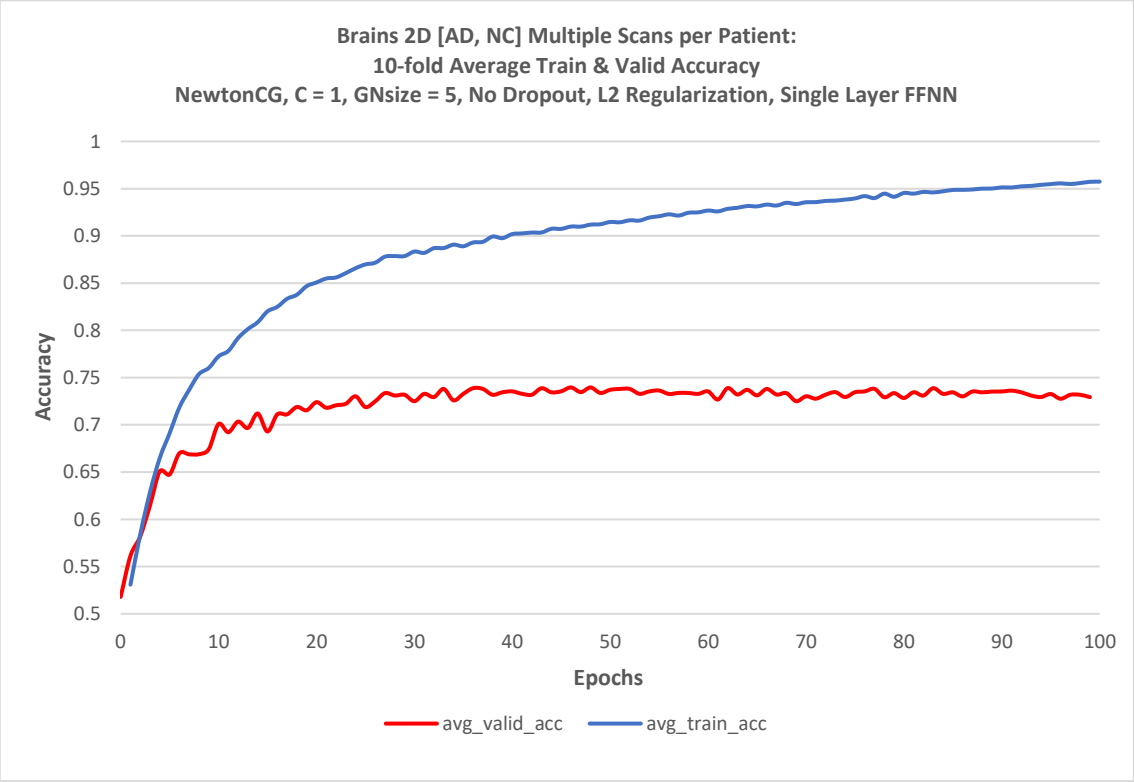


Figure H.73. Caption in the title.

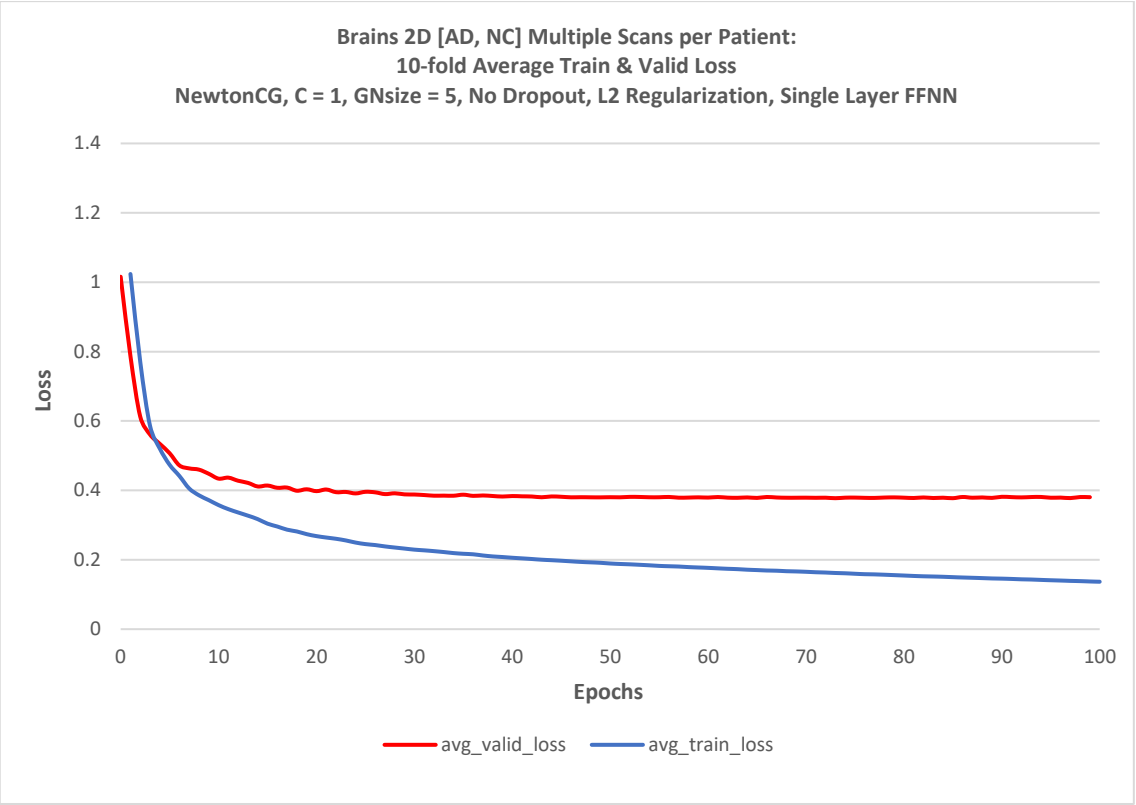


Figure H.74. Caption in the title.

H.2.6.2 Training

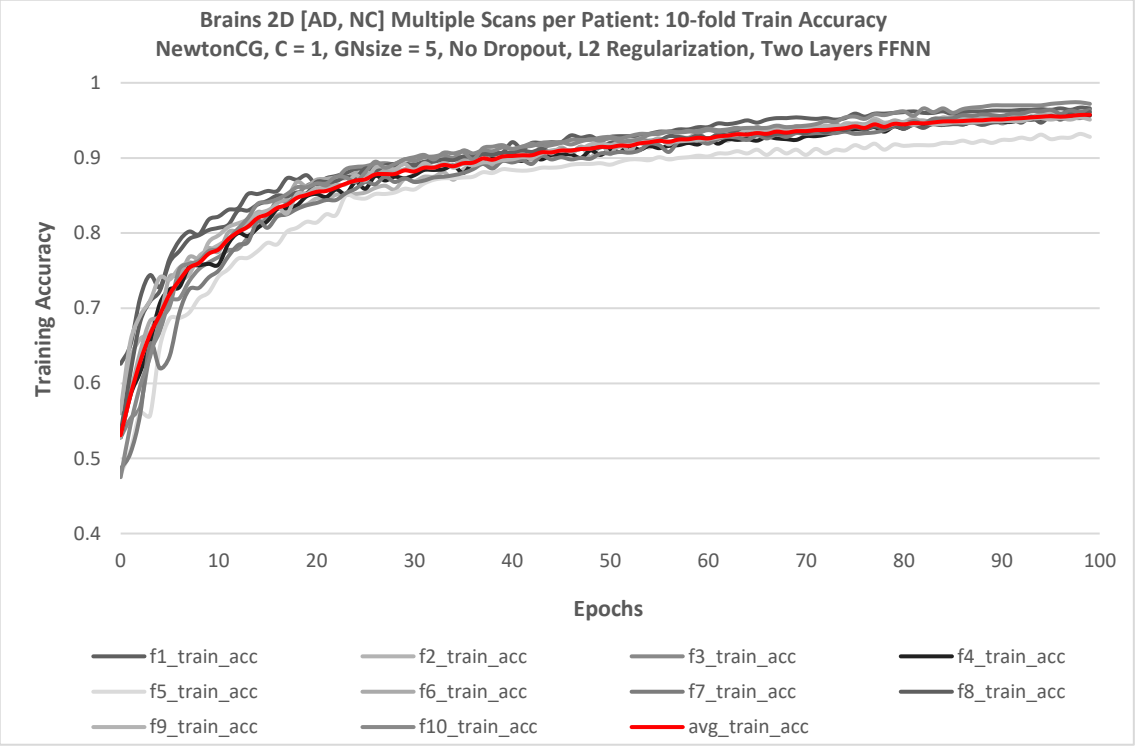


Figure H.75. Caption in the title.

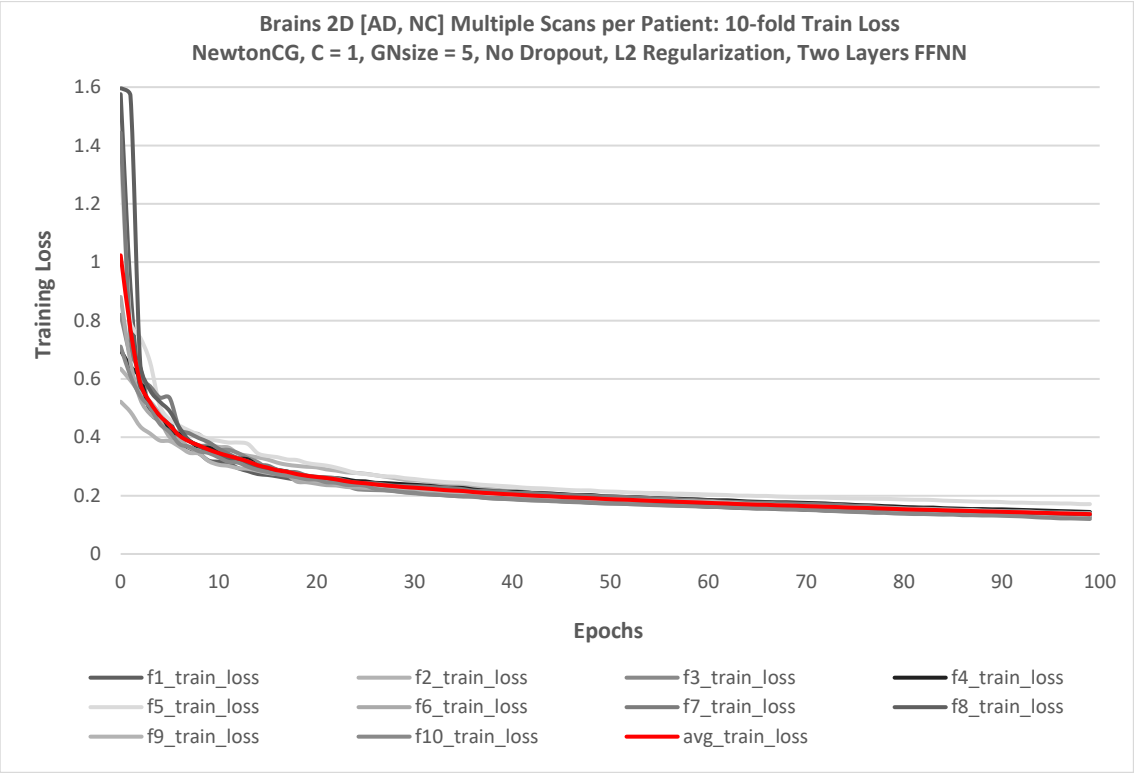


Figure H.76. Caption in the title.

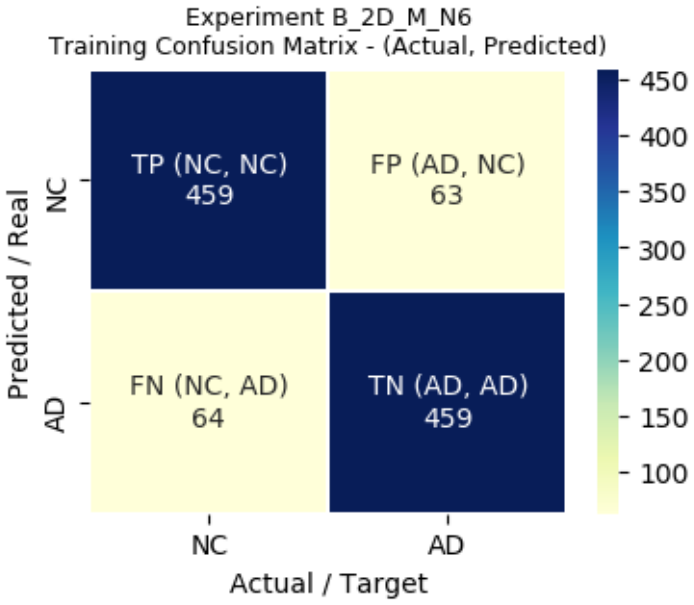


Figure H.77. Caption in the title.

H.2.6.3 Validation

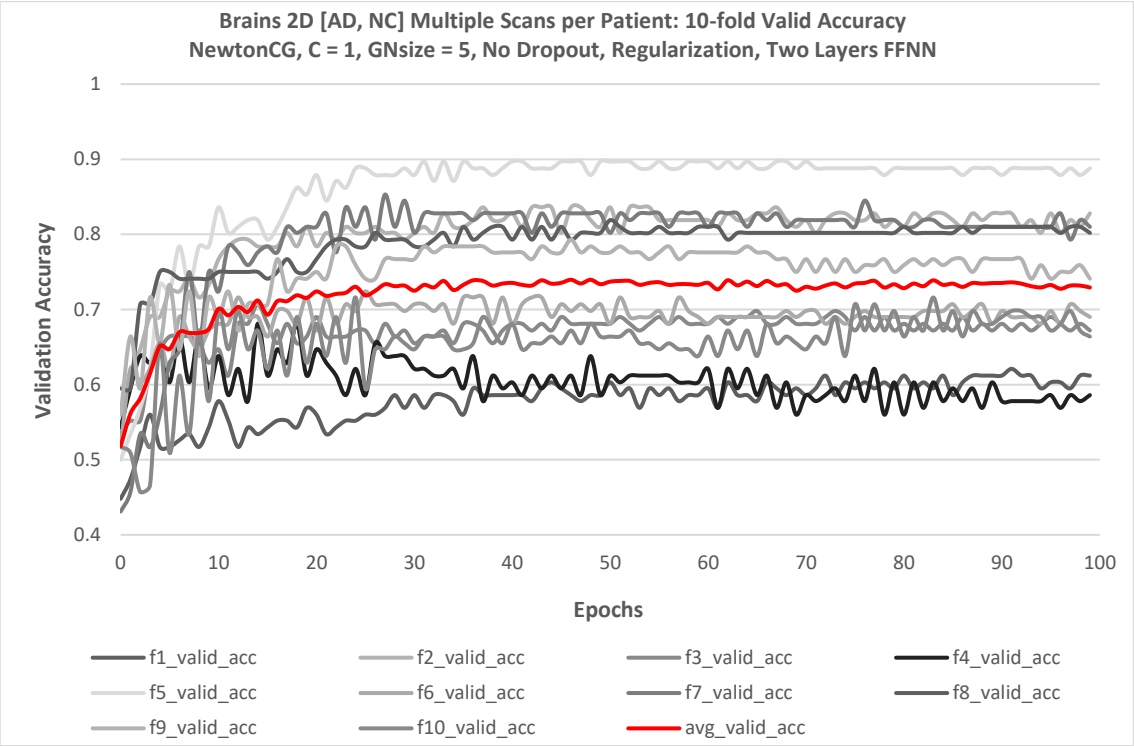


Figure H.78. Caption in the title.

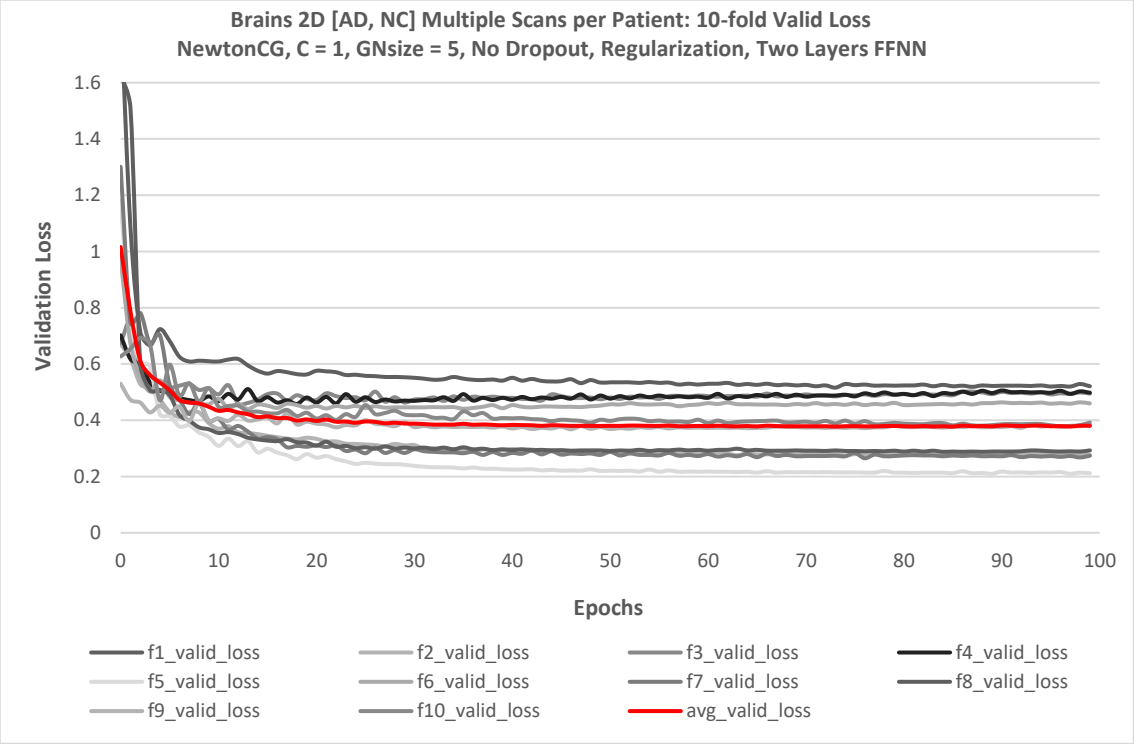


Figure H.79. Caption in the title.

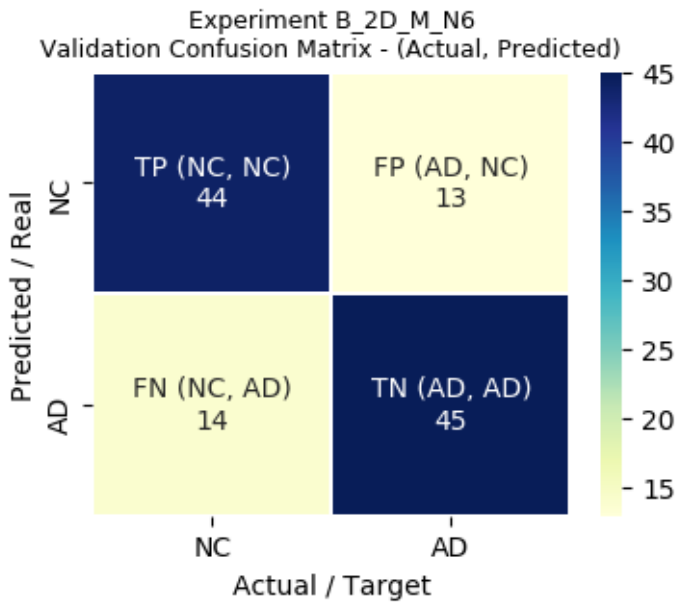


Figure H.80. Caption in the title.

H.2.6.4 Testing

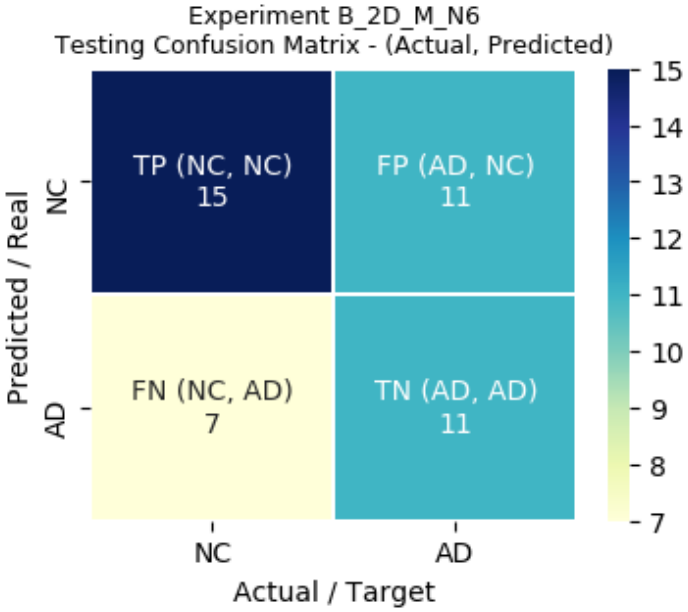


Figure H.81. Caption in the title.

H.2.7 Experiment B_2D_M_N7: C = 1, GNsize = 5, Dropout

H.2.7.1 General

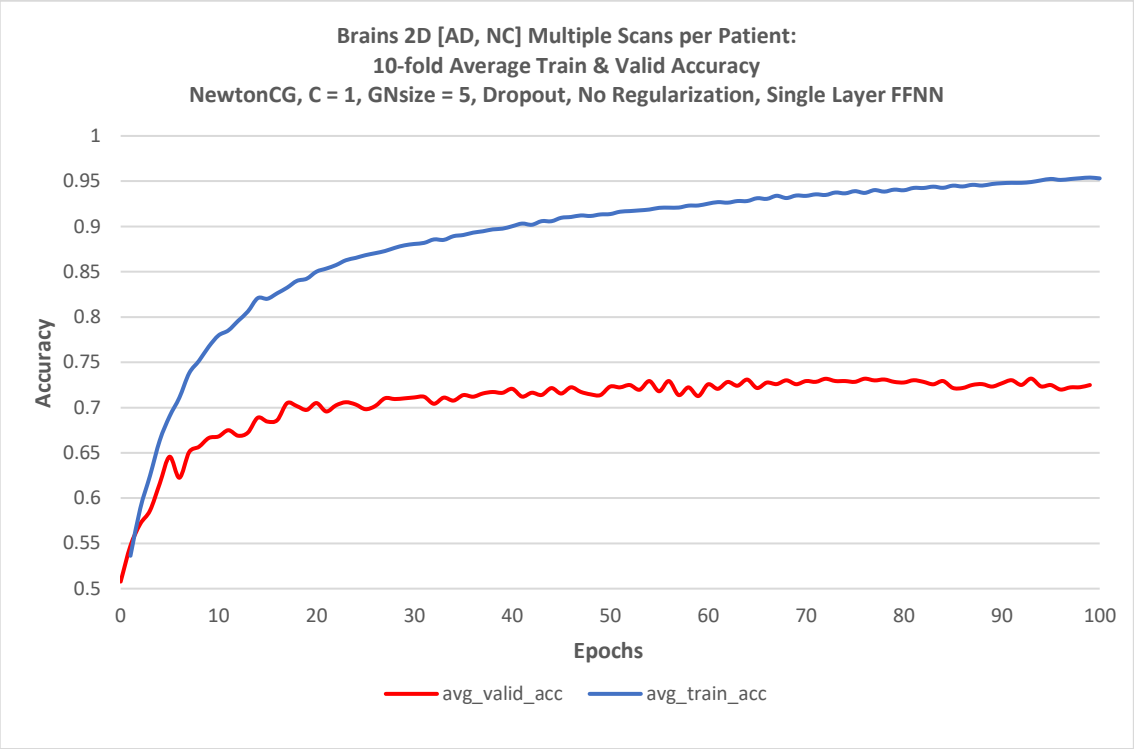


Figure H.82. Caption in the title.

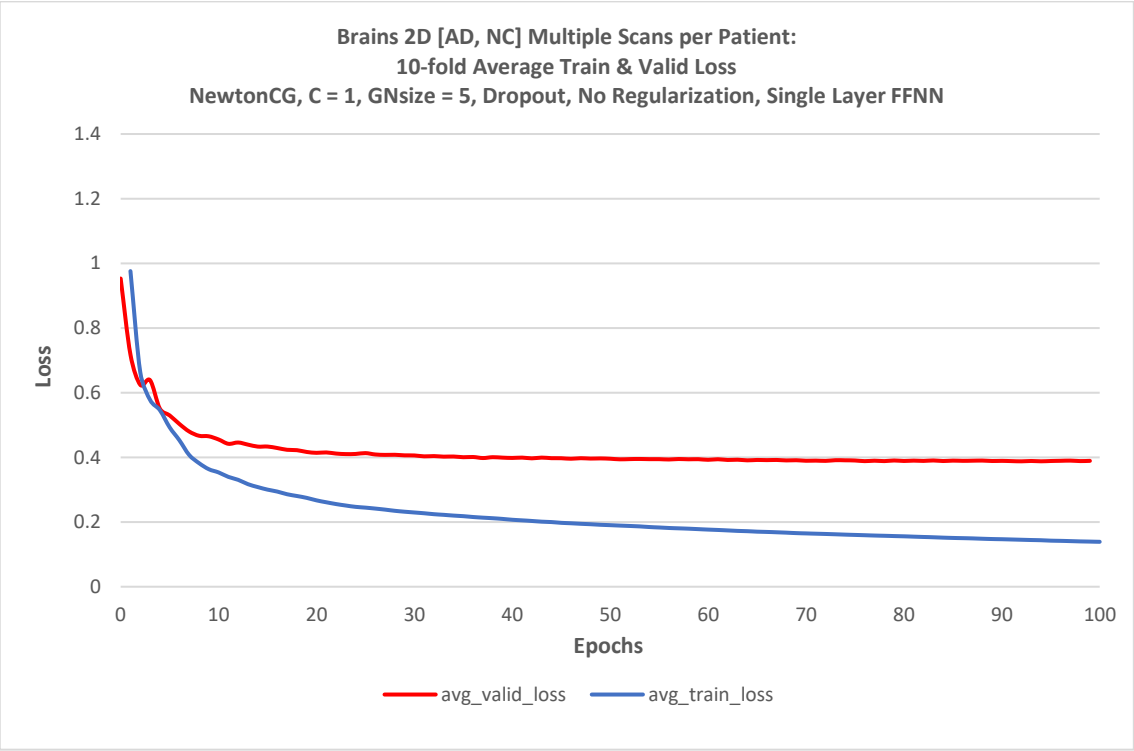


Figure H.83. Caption in the title.

H.2.7.2 Training

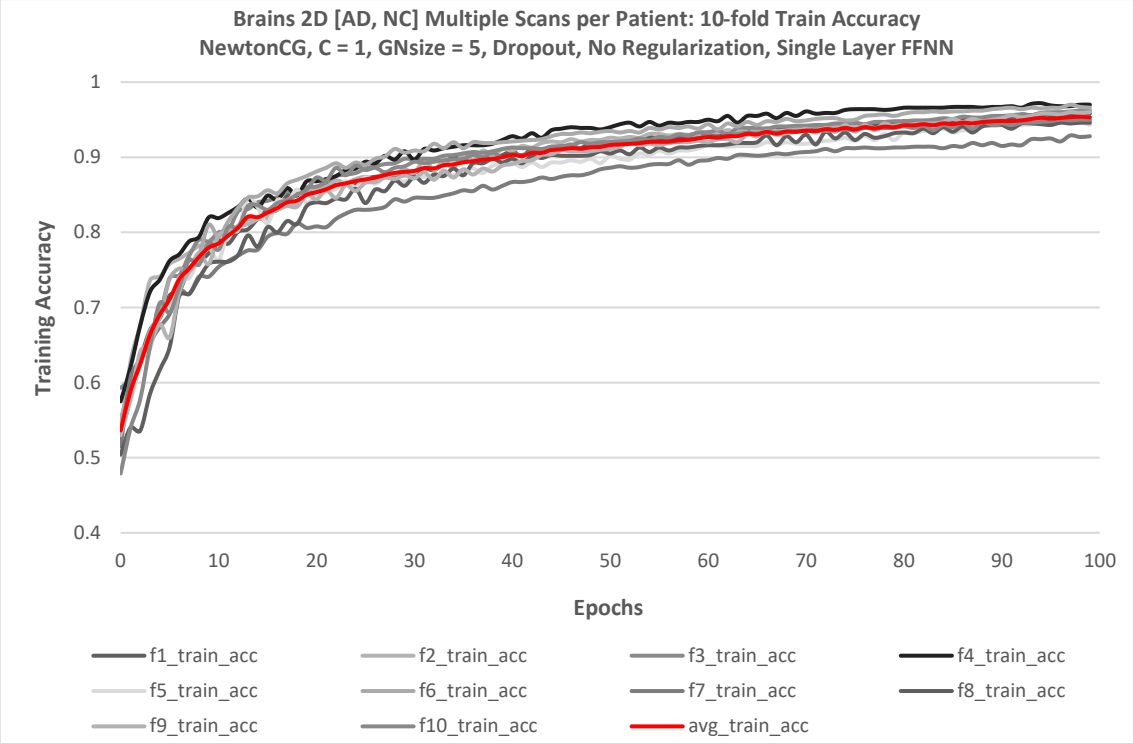


Figure H.84. Caption in the title.

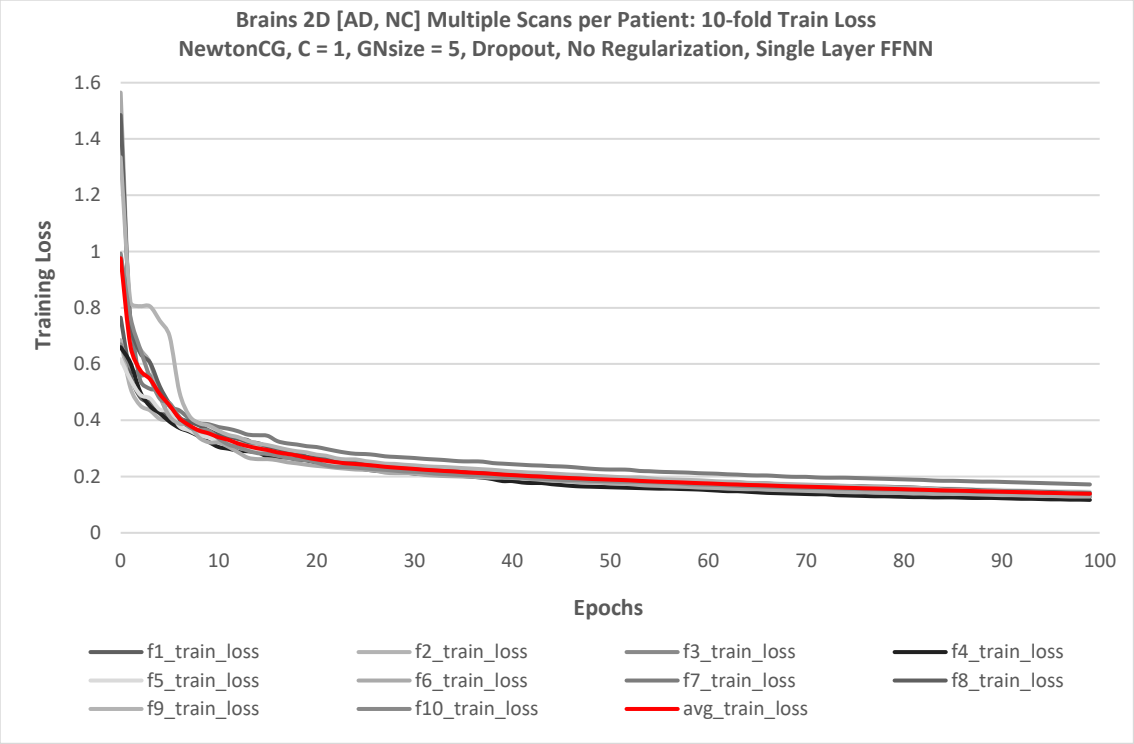


Figure H.85. Caption in the title.

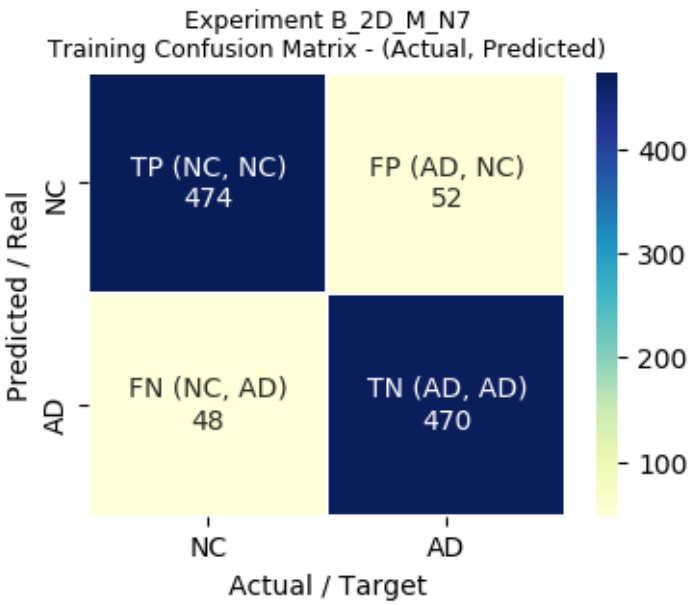


Figure H.86. Caption in the title.

H.2.7.3 Validation

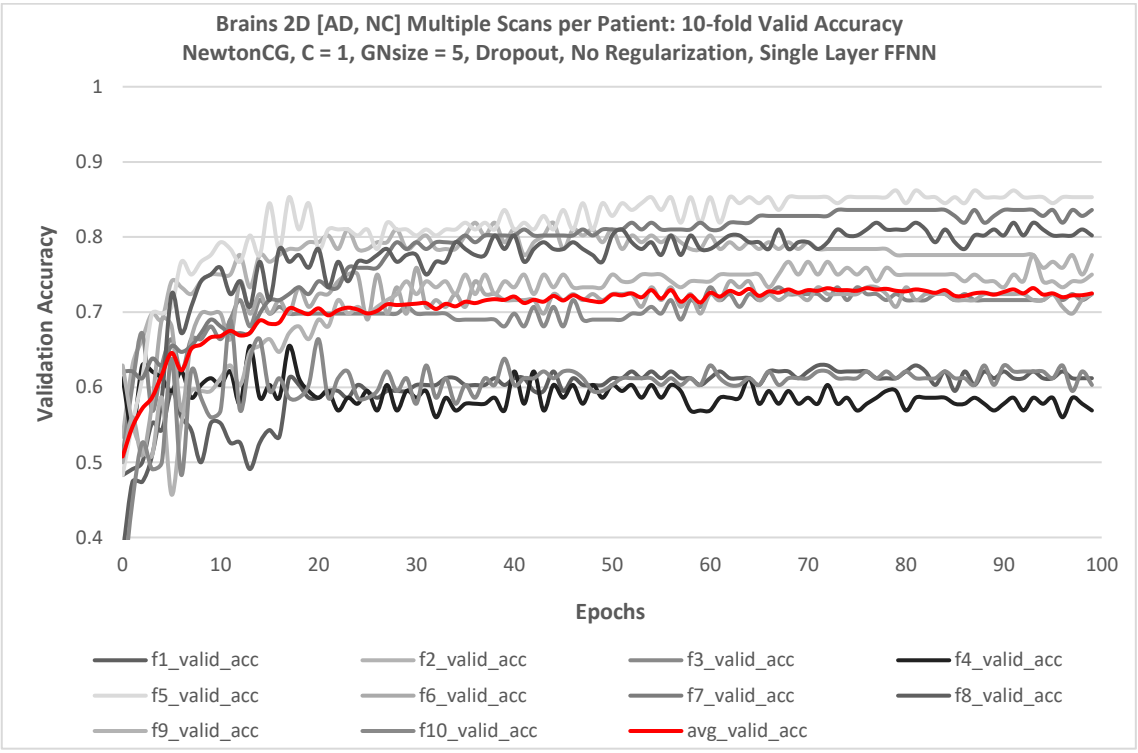


Figure H.87. Caption in the title.

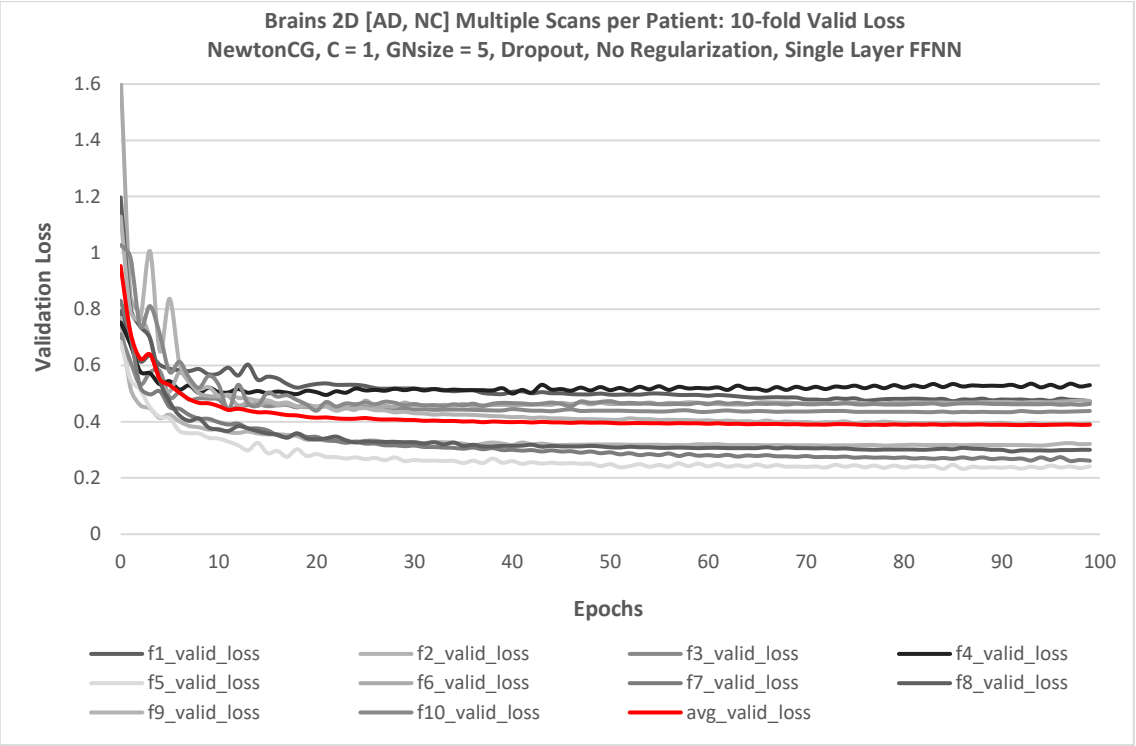


Figure H.88. Caption in the title.

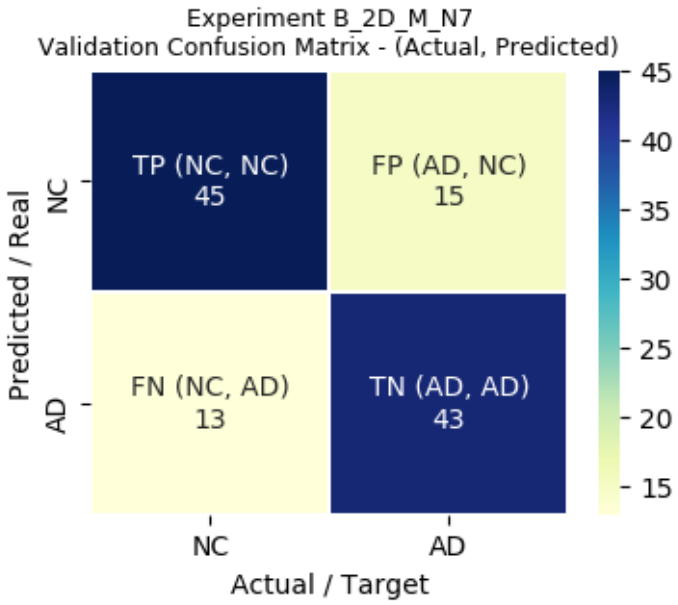


Figure H.89. Caption in the title.

H.2.7.4 Testing

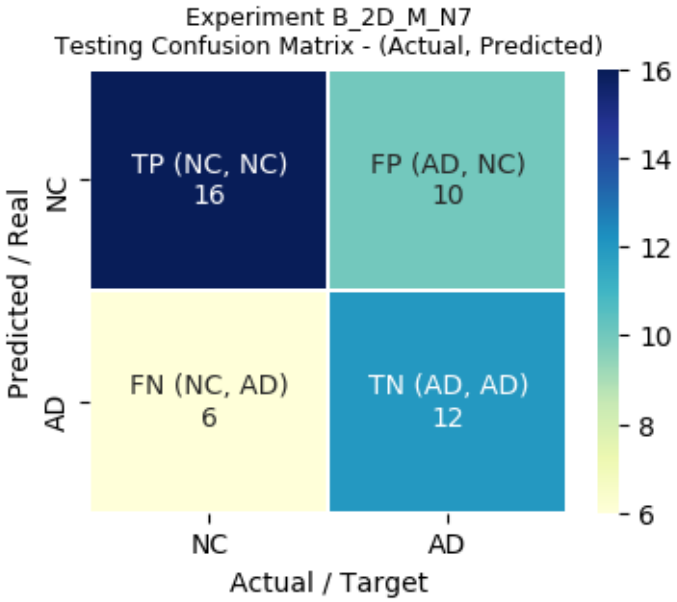


Figure H.90. Caption in the title.

H.2.8 Experiment B_2D_M_N8: C = 1, GNsize = 50

H.2.8.1 General

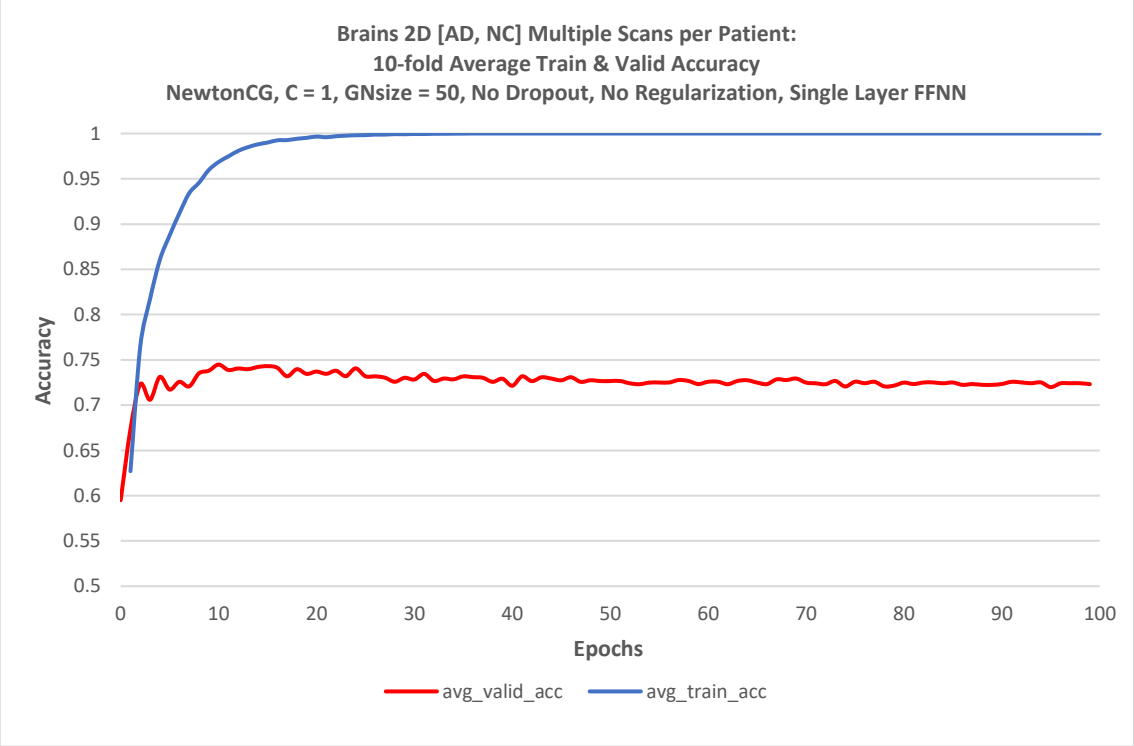


Figure H.91. Caption in the title.

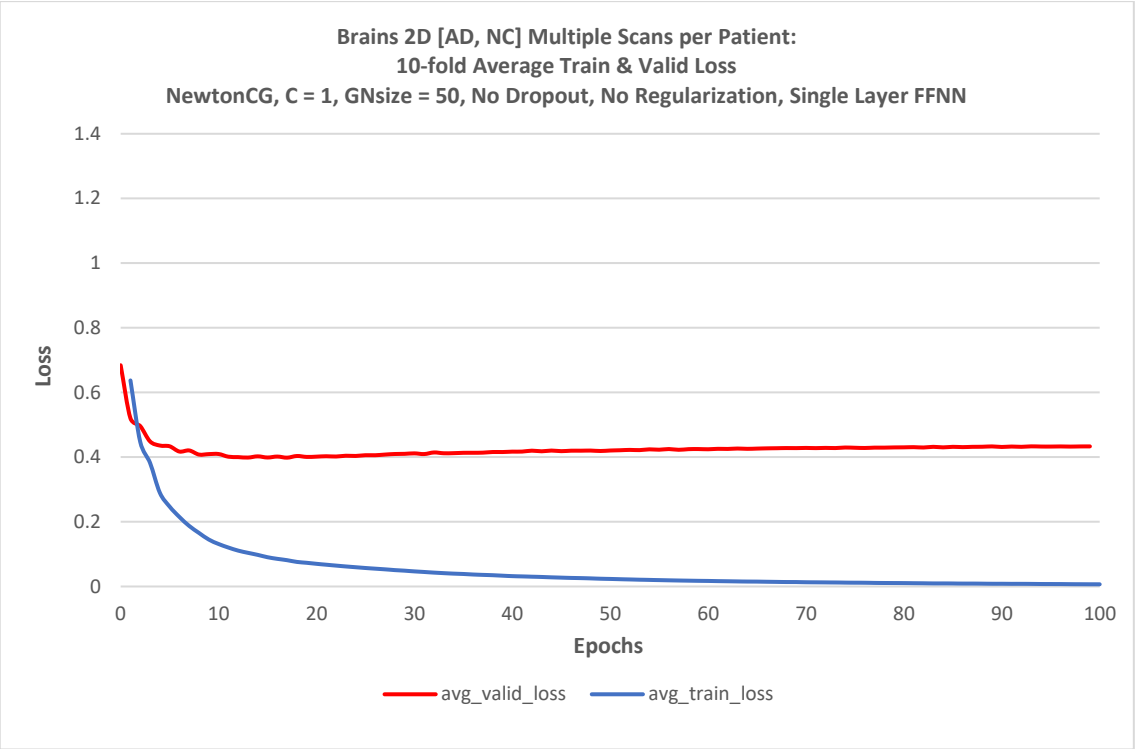


Figure H.92. Caption in the title.

H.2.8.2 Training

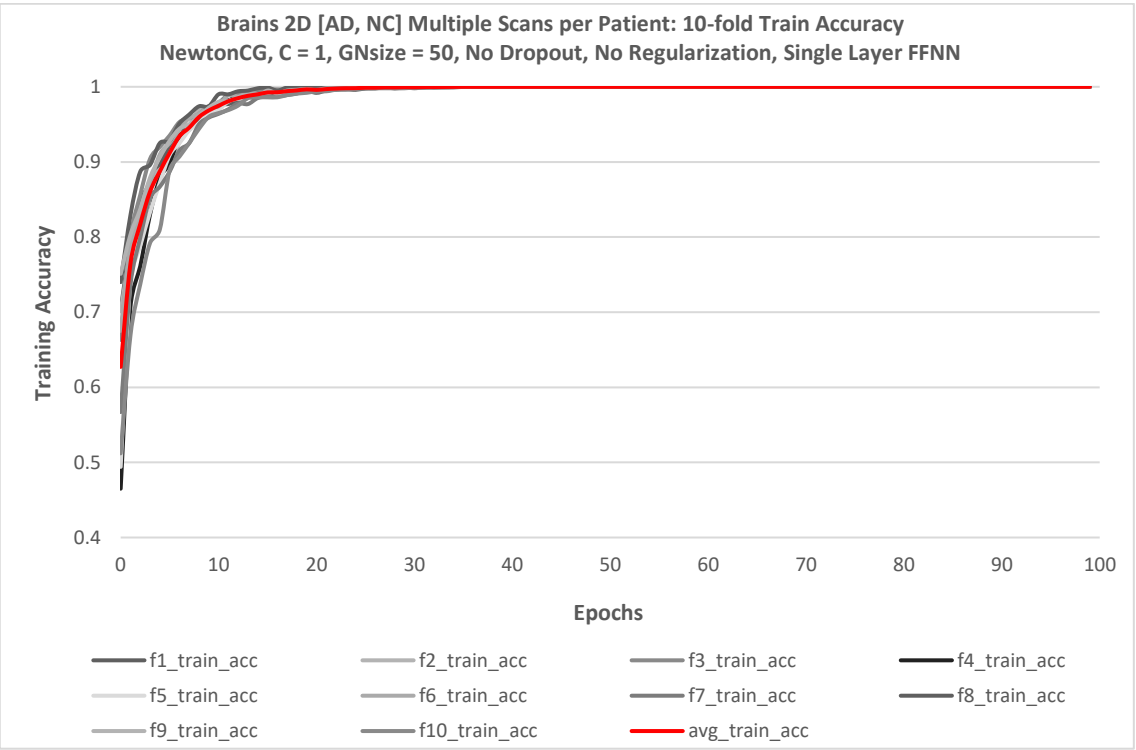


Figure H.93. Caption in the title.

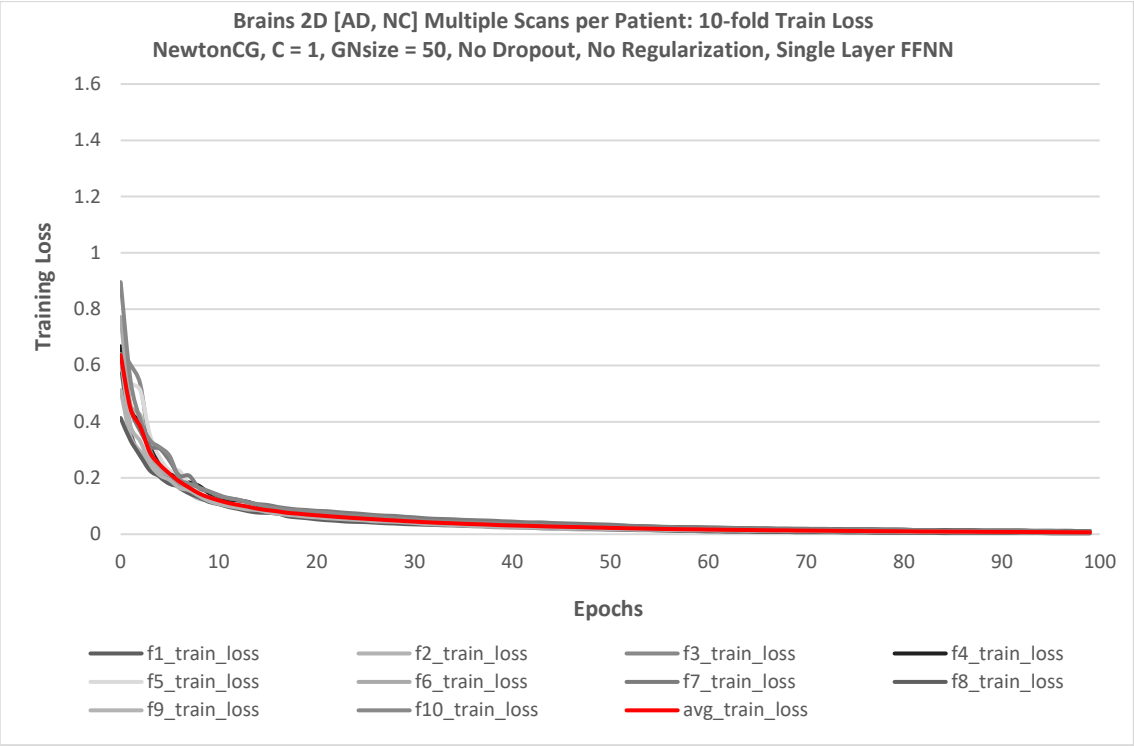


Figure H.94. Caption in the title.

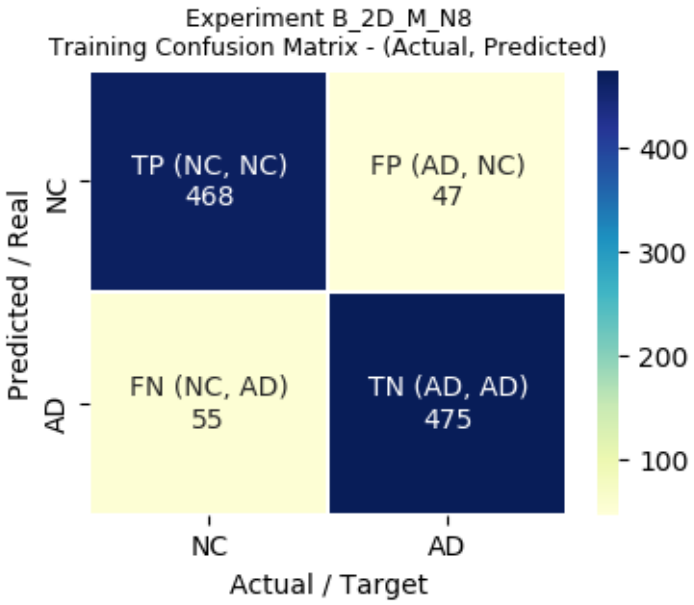


Figure H.95. Caption in the title.

H.2.8.3 Validation

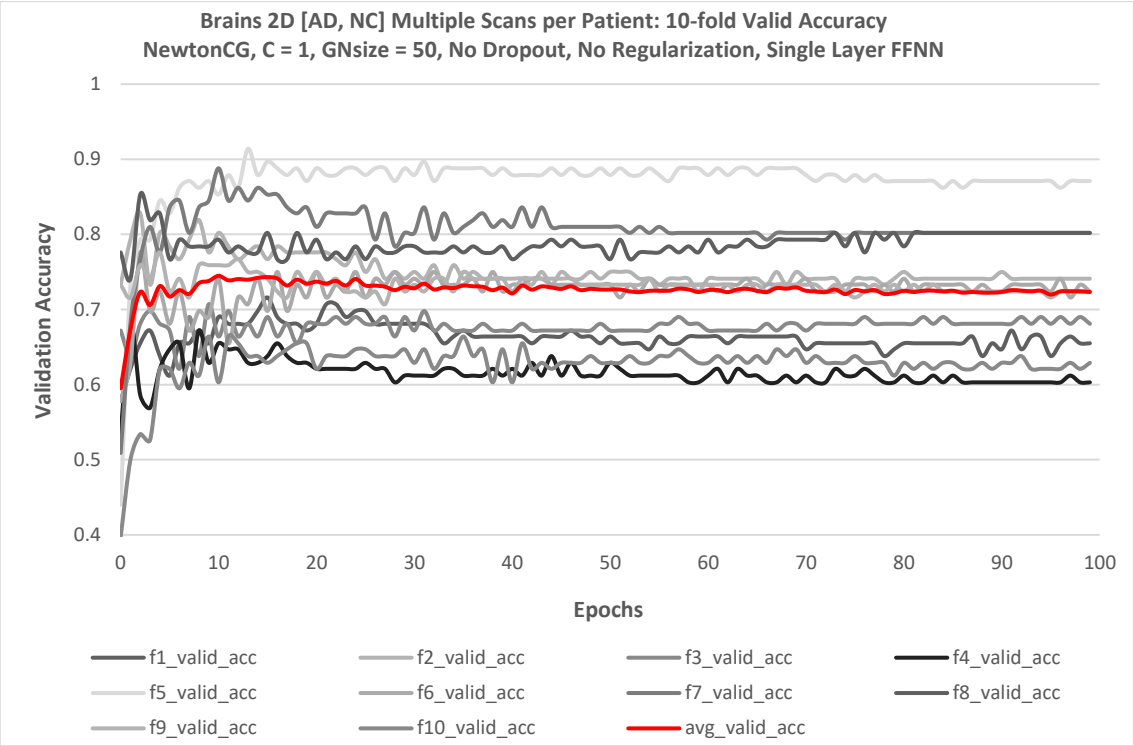


Figure H.96. Caption in the title.

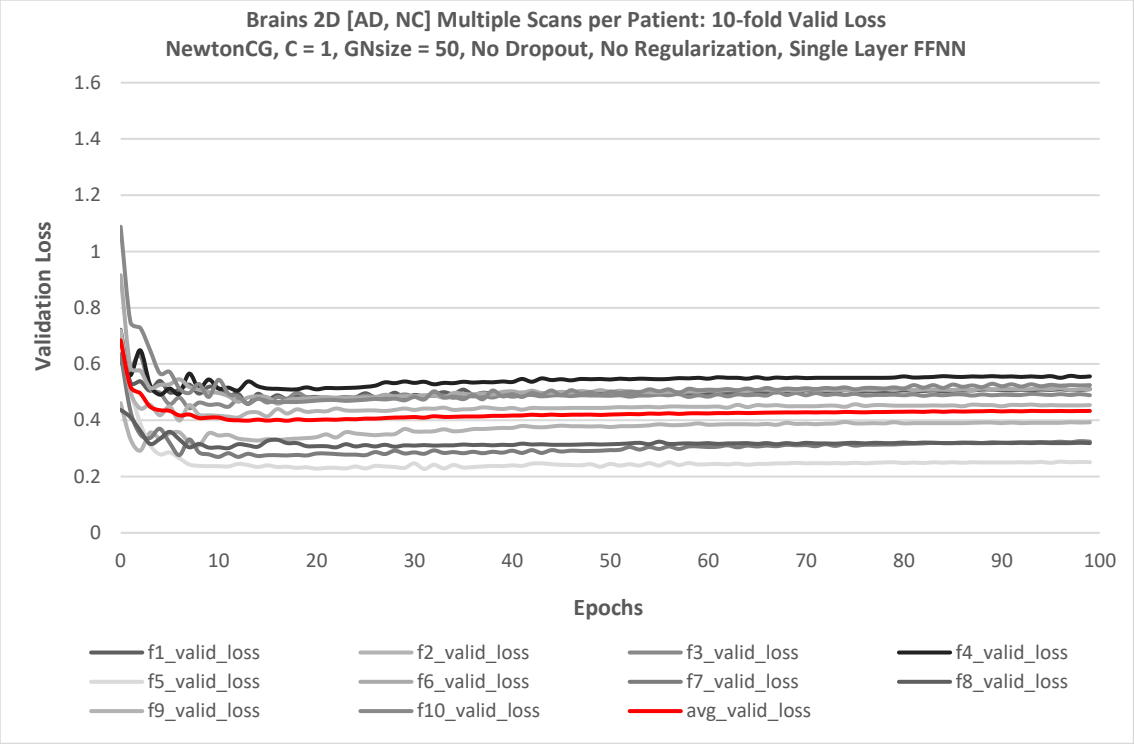


Figure H.97. Caption in the title.

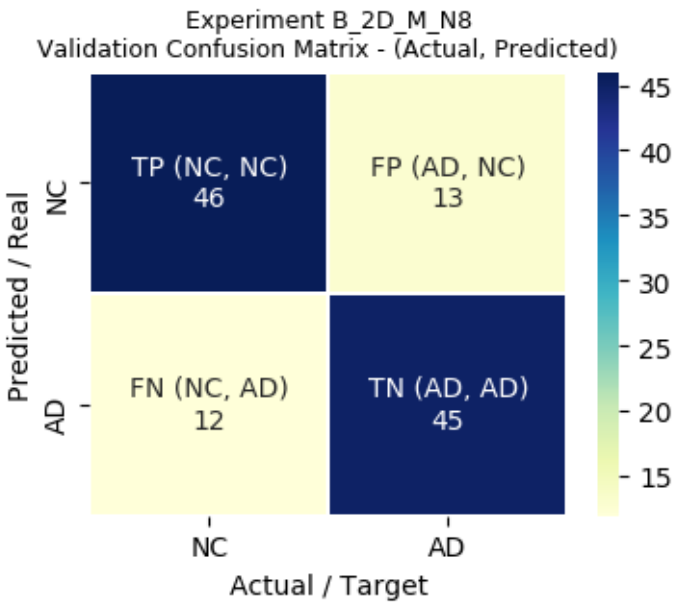


Figure H.98. Caption in the title.

H.2.8.4 Testing

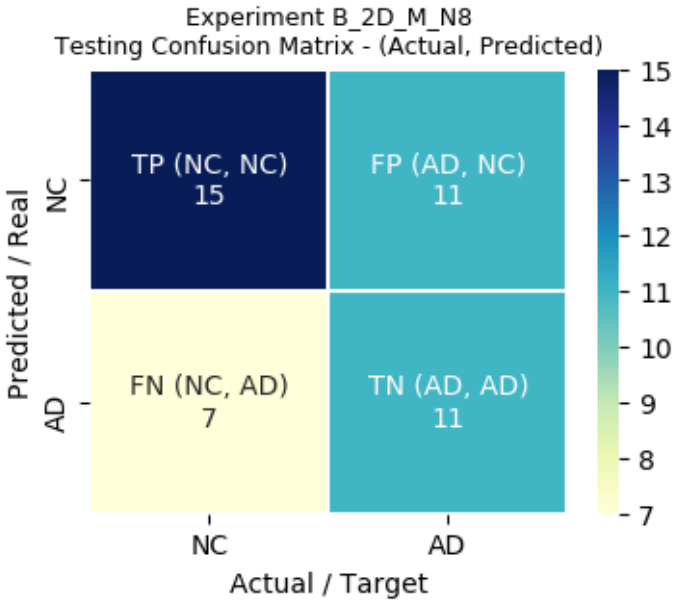


Figure H.99. Caption in the title.

H.2.9 Experiment B_2D_M_N9: C = 1, GNsize = 50, Dropout

H.2.9.1 General

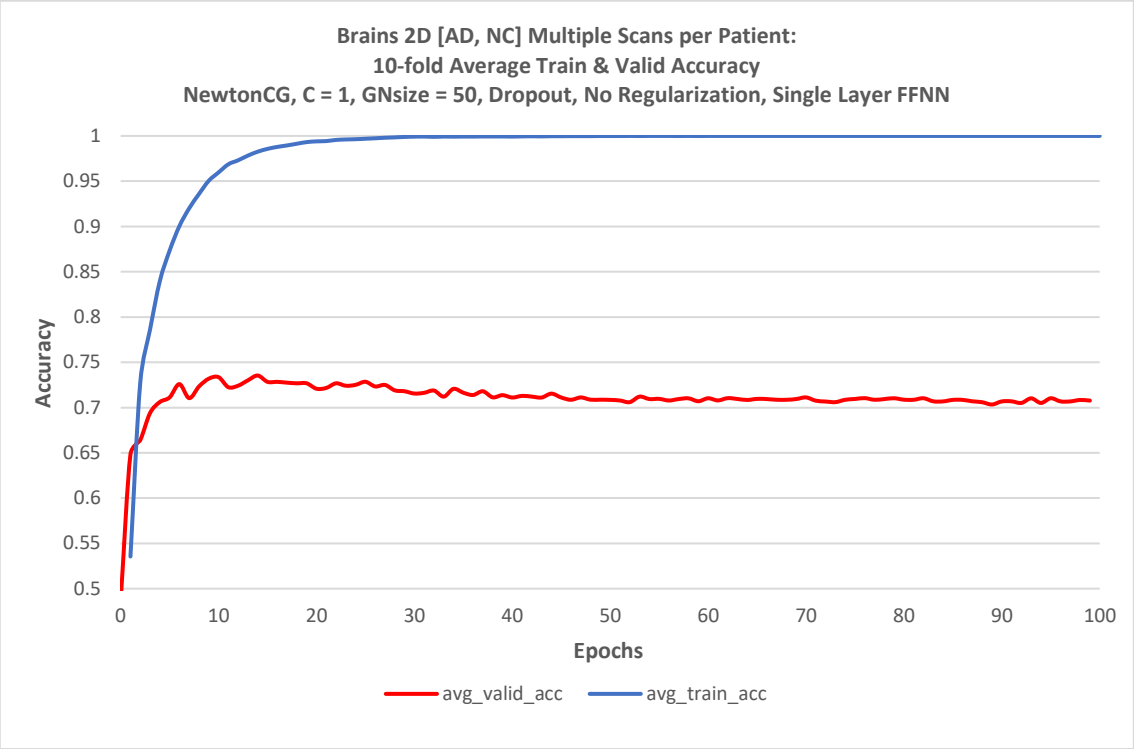


Figure H.100. Caption in the title.

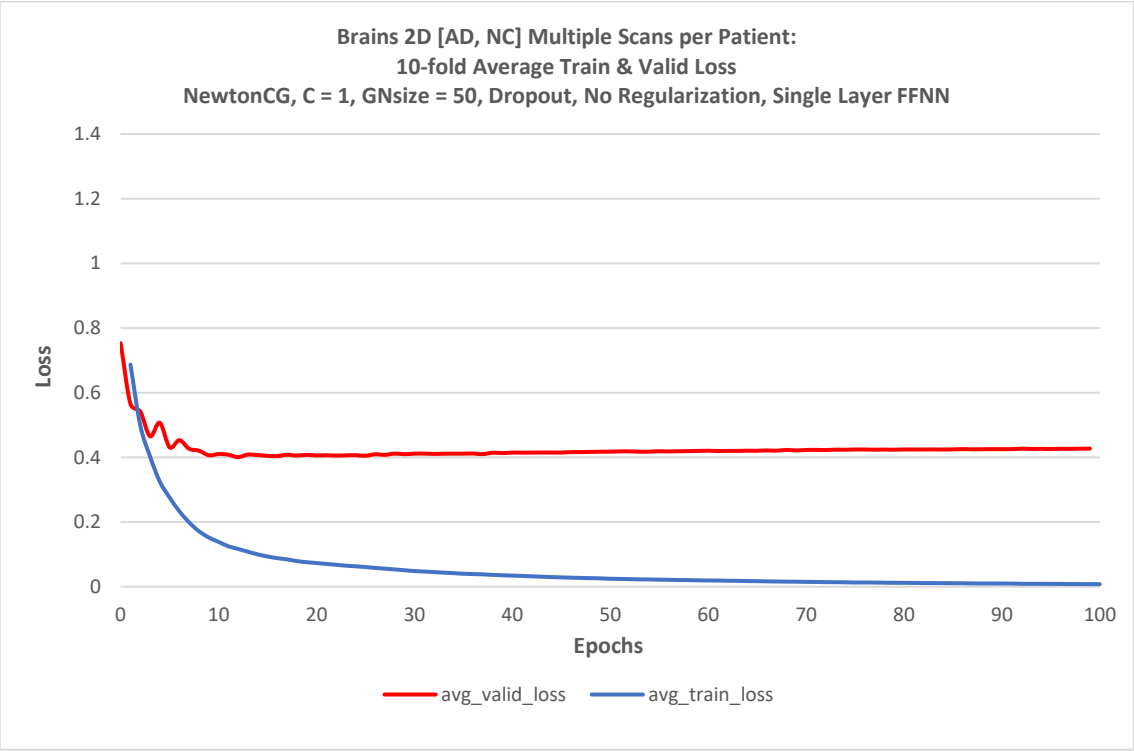


Figure H.101. Caption in the title.

H.2.9.2 Training

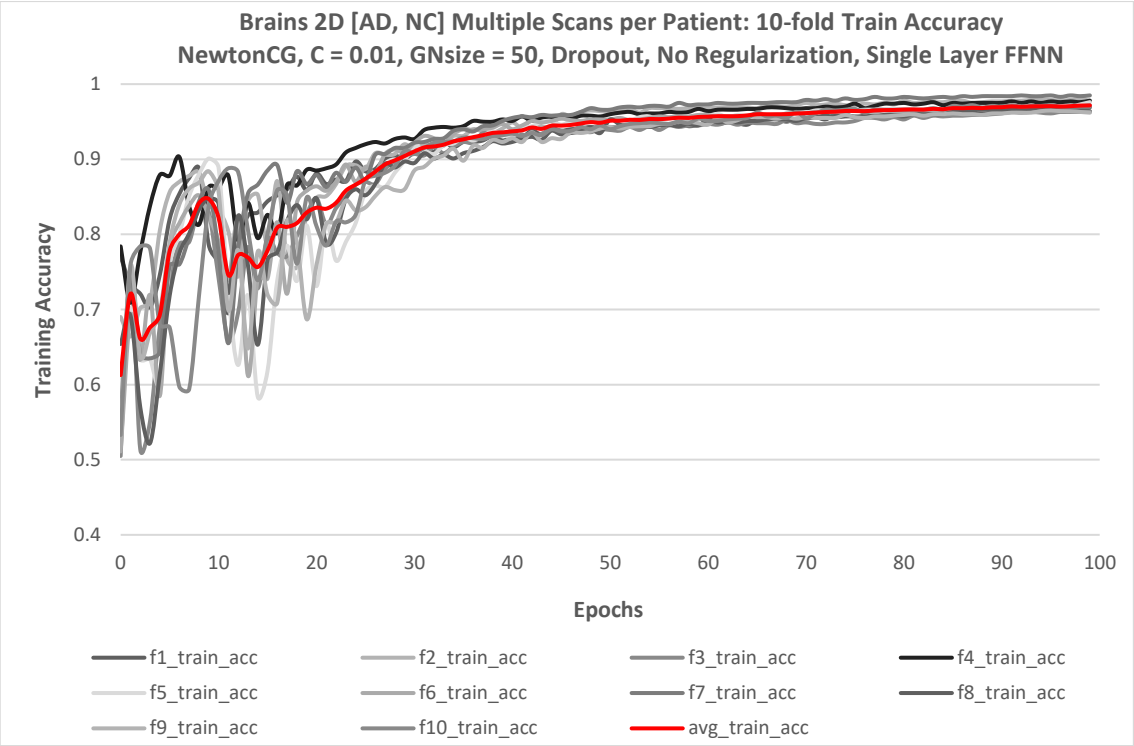


Figure H.102. Caption in the title.

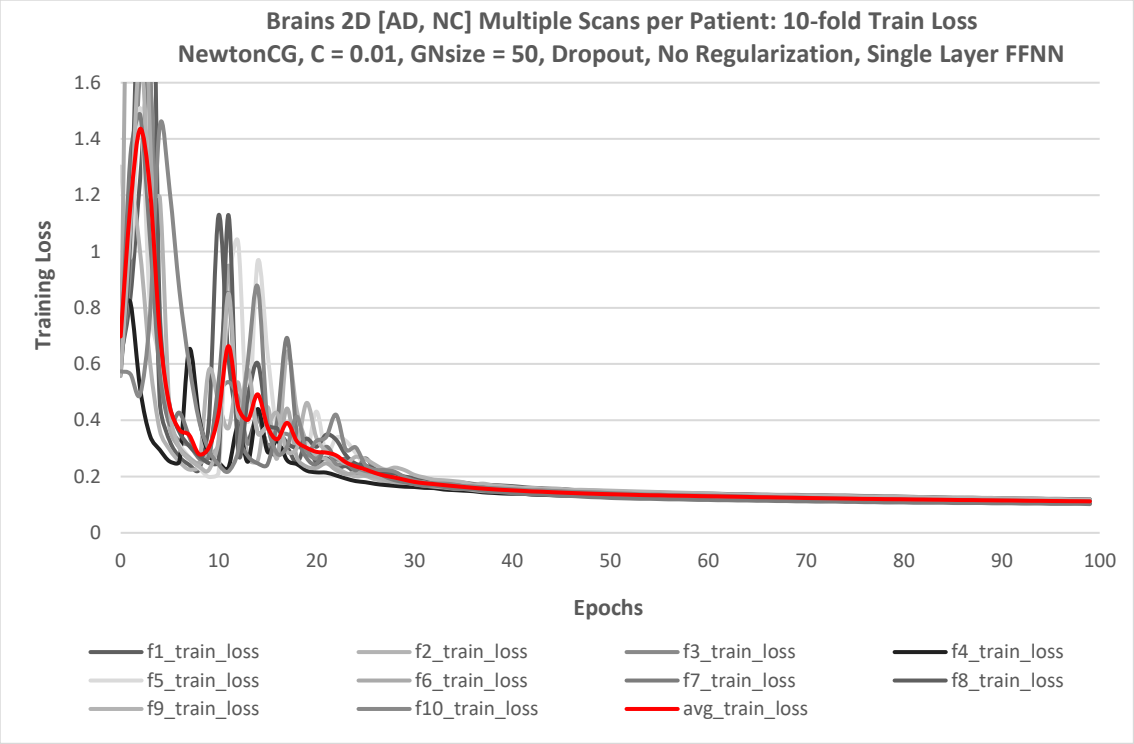


Figure H.103. Caption in the title.

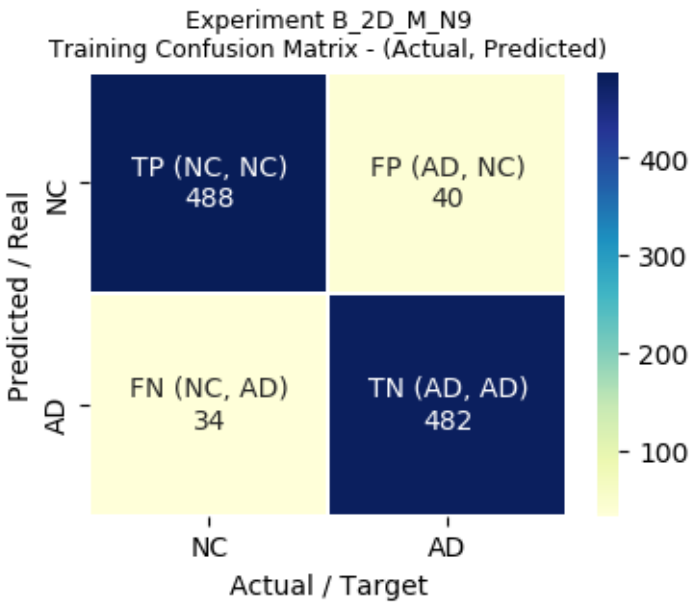


Figure H.104. Caption in the title.

H.2.9.3 Validation

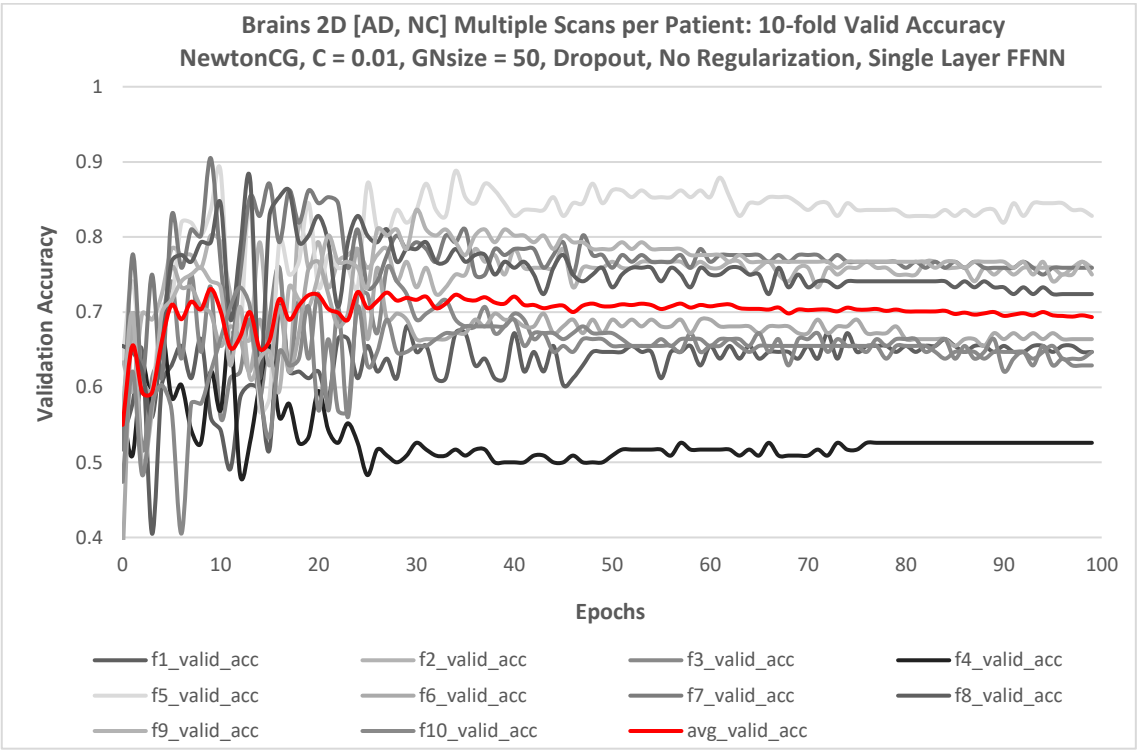


Figure H.105. Caption in the title.

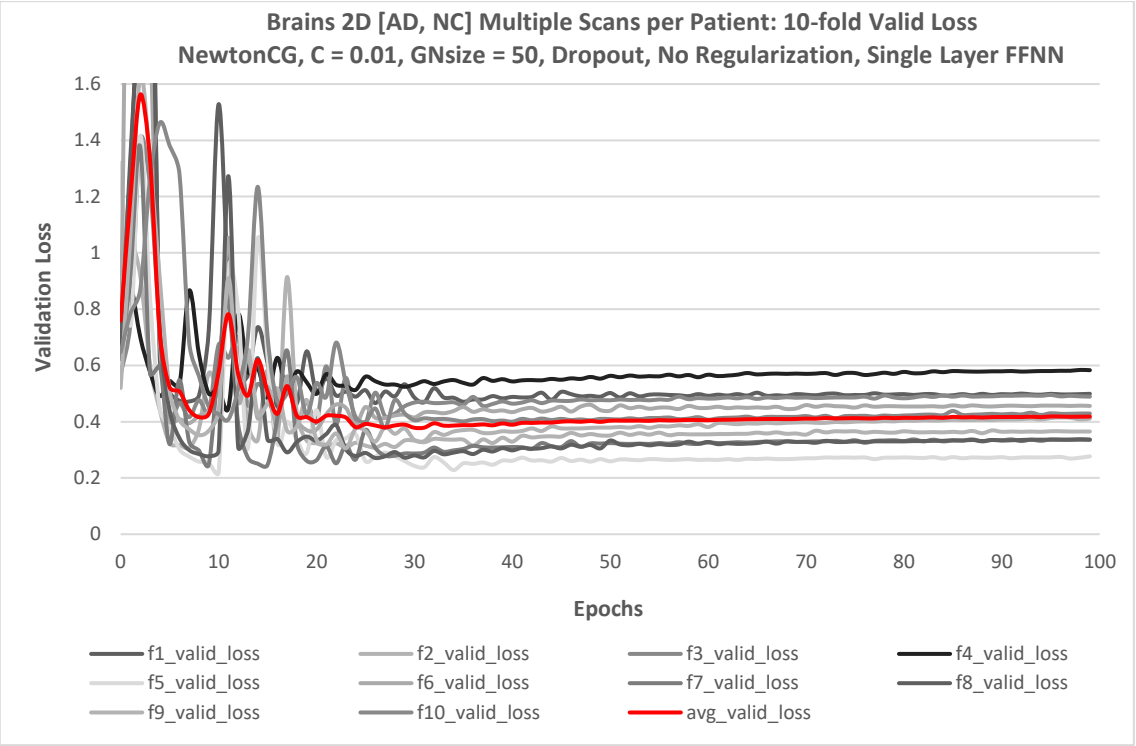


Figure H.106. Caption in the title.

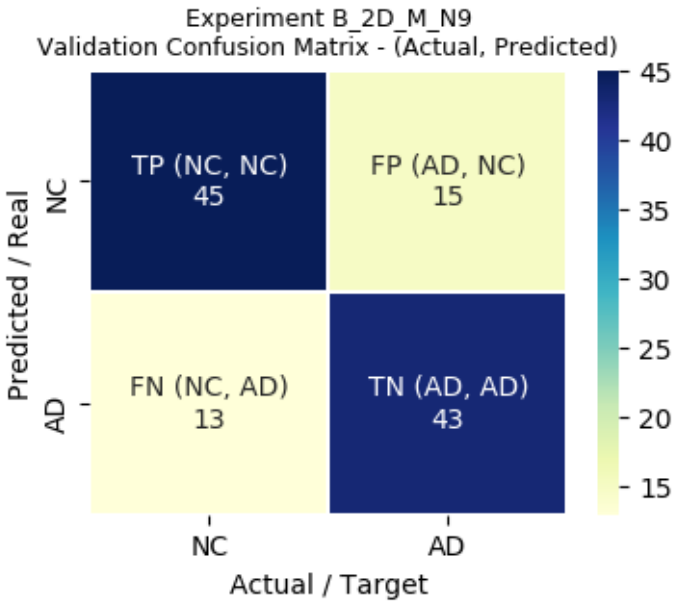


Figure H.107. Caption in the title.

H.2.9.4 Testing

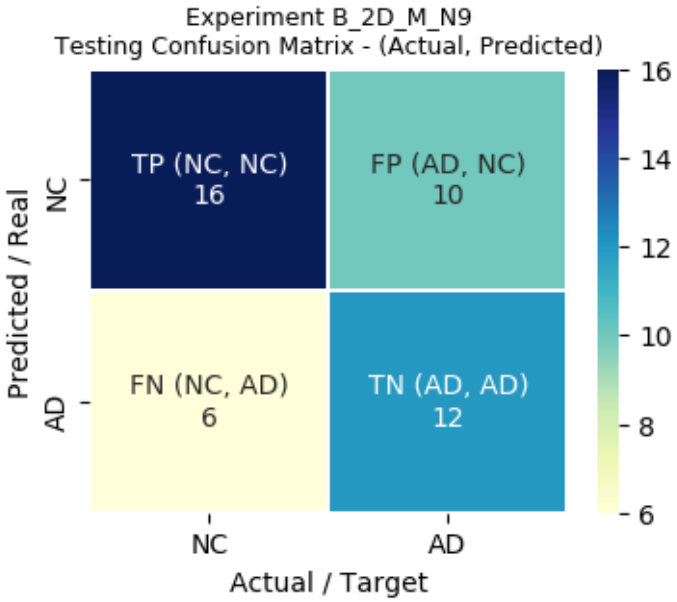


Figure H.108. Caption in the title.

H.2.10 Experiment B_2D_M_N10: C = 0.01, GNsize = 50, Dropout

H.2.10.1 General

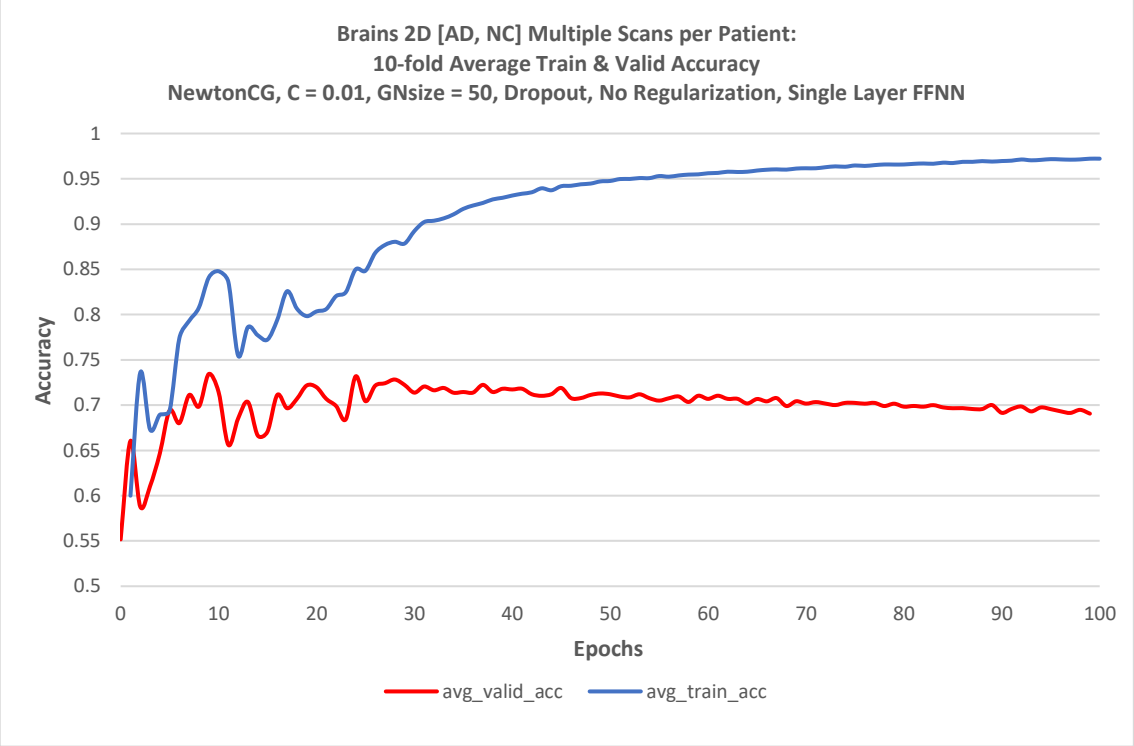


Figure H.109. Caption in the title.

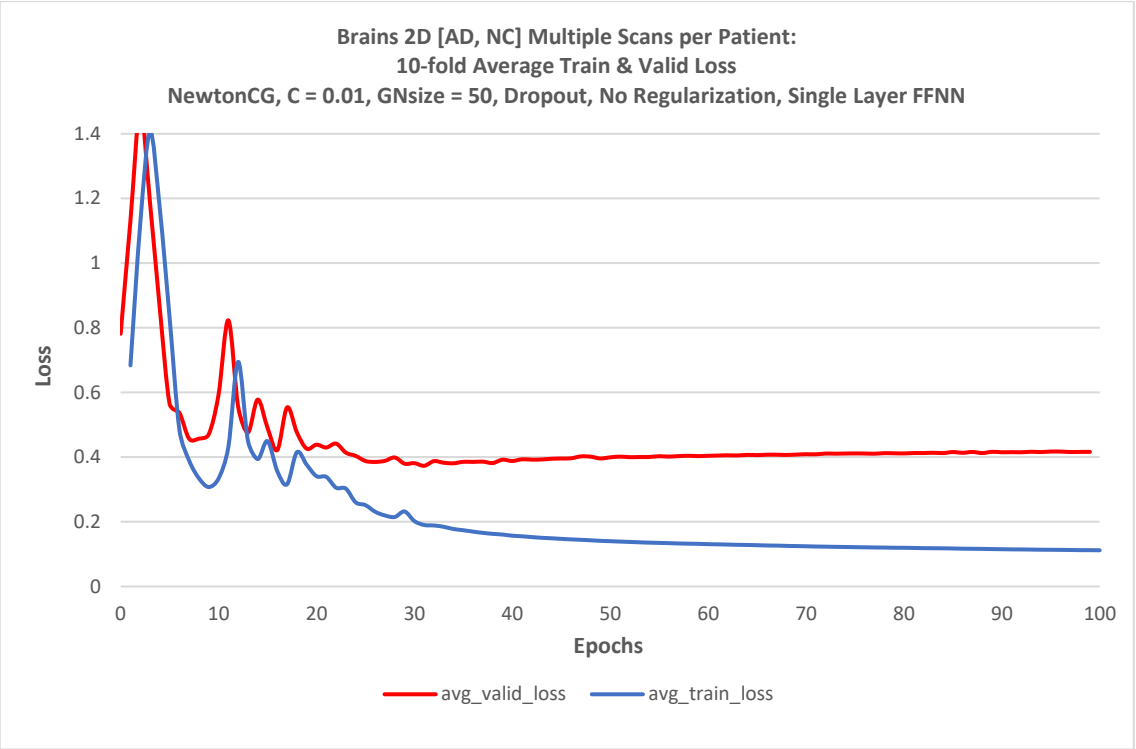


Figure H.110. Caption in the title.

H.2.10.2 Training

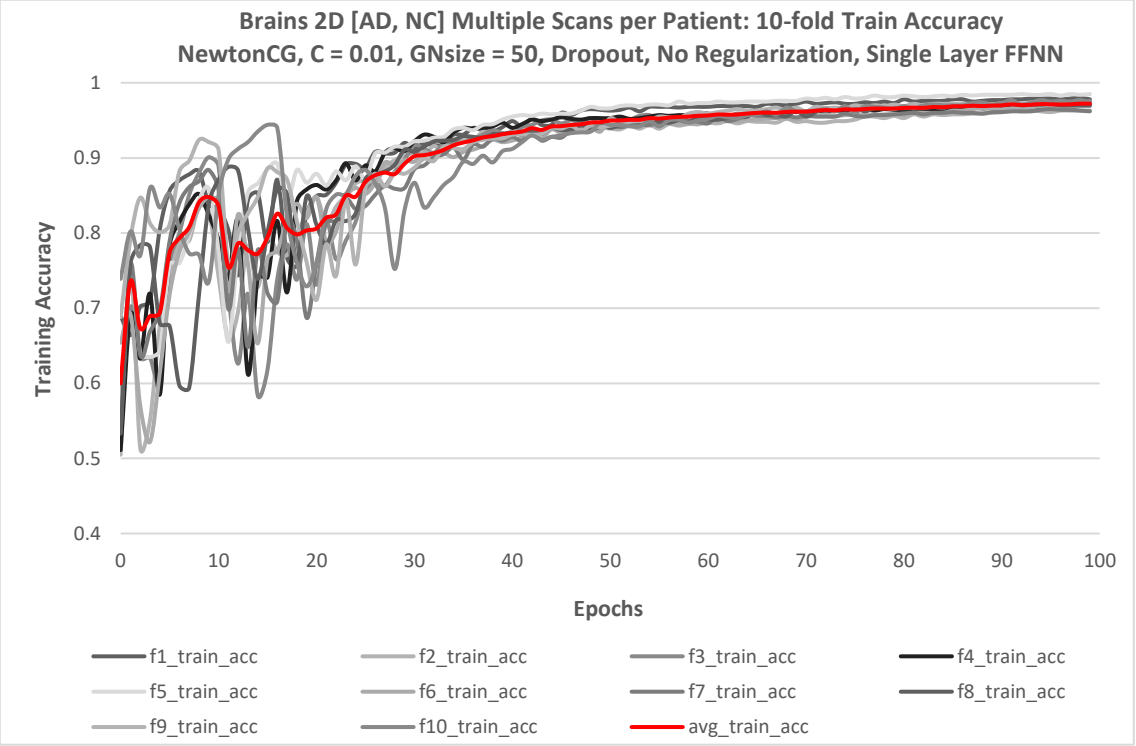


Figure H.111. Caption in the title.

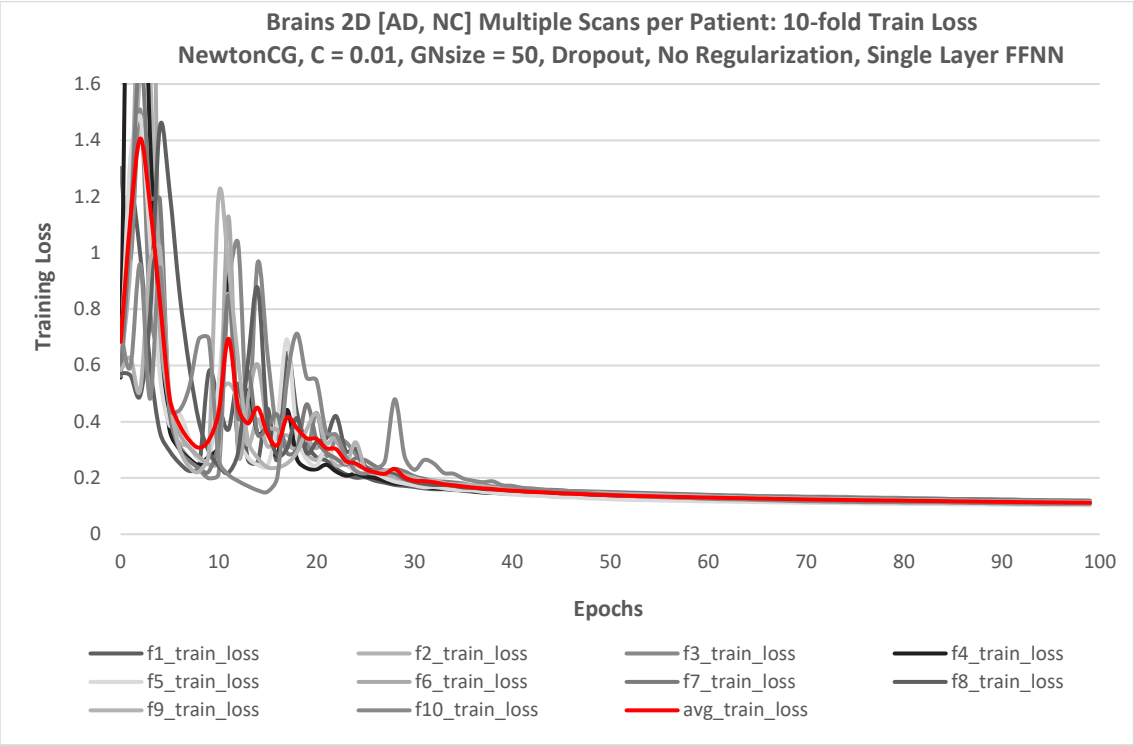


Figure H.112. Caption in the title.

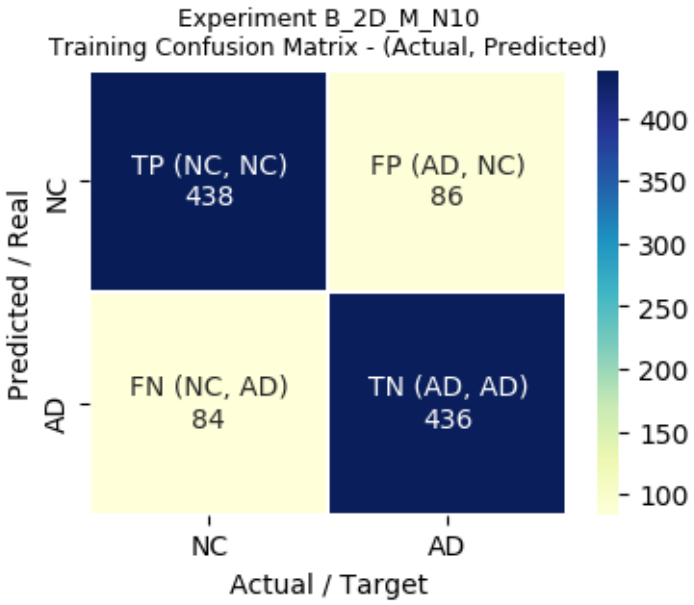


Figure H.113. Caption in the title.

H.2.10.3 Validation

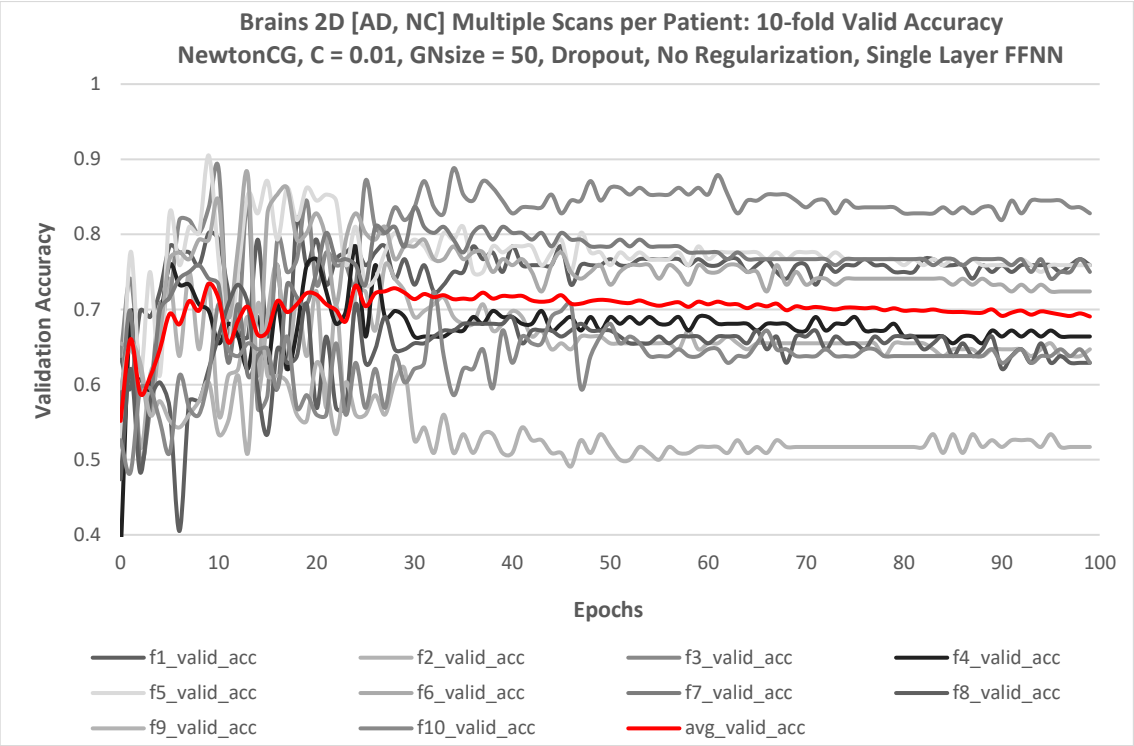


Figure H.114. Caption in the title.

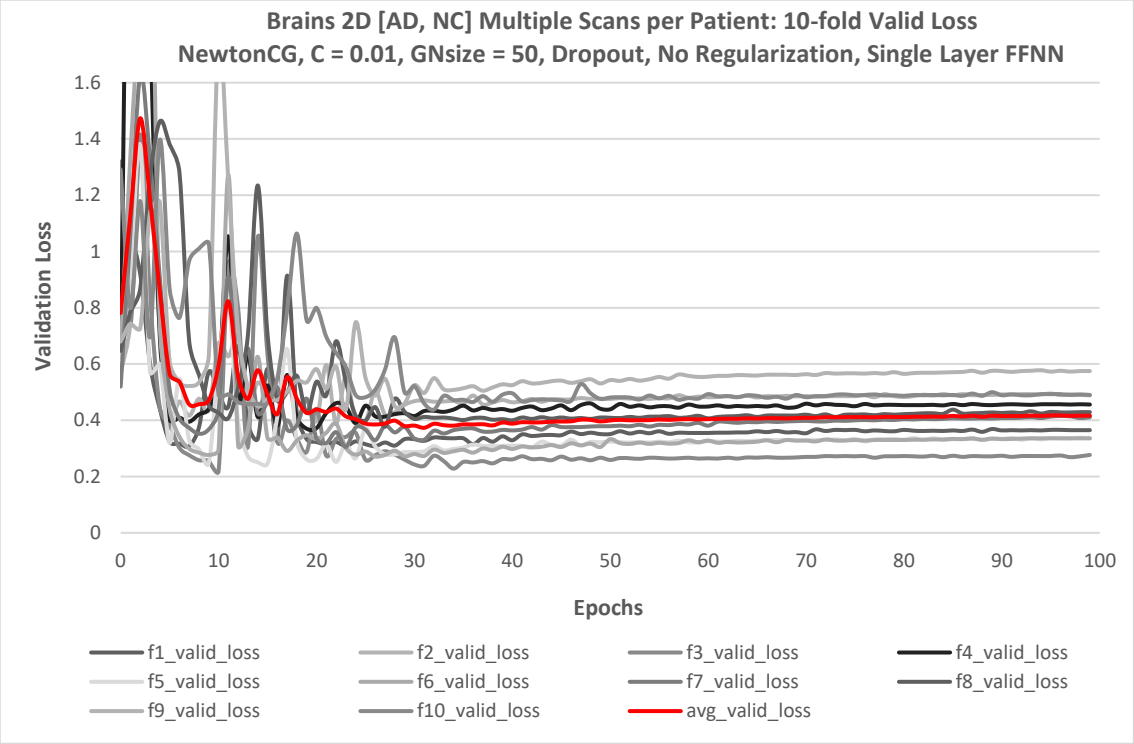


Figure H.115. Caption in the title.

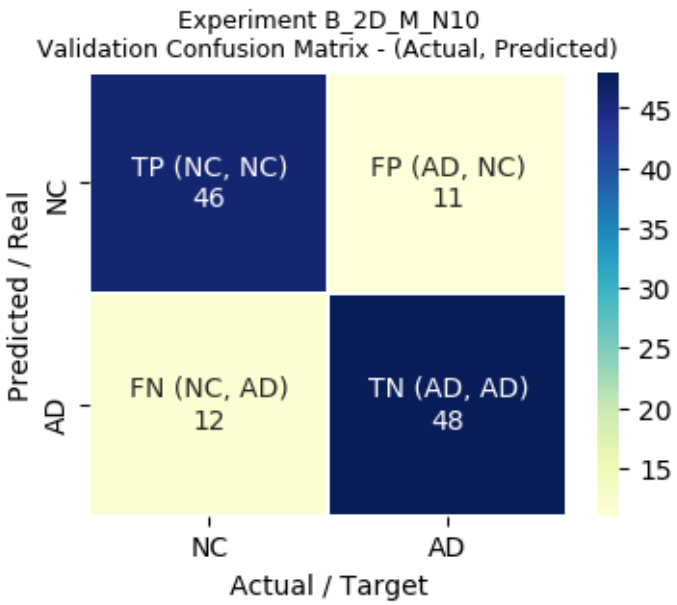


Figure H.116. Caption in the title.

H.2.10.4 Testing

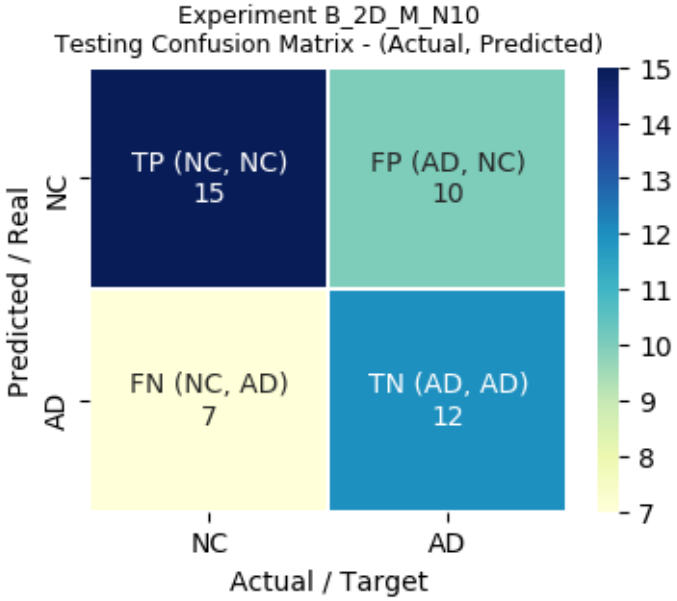


Figure H.117. Caption in the title.

H.2.11 Experiment B_2D_M_N11: C = 0.01, GNsize = 200, Dropout

H.2.11.1 General

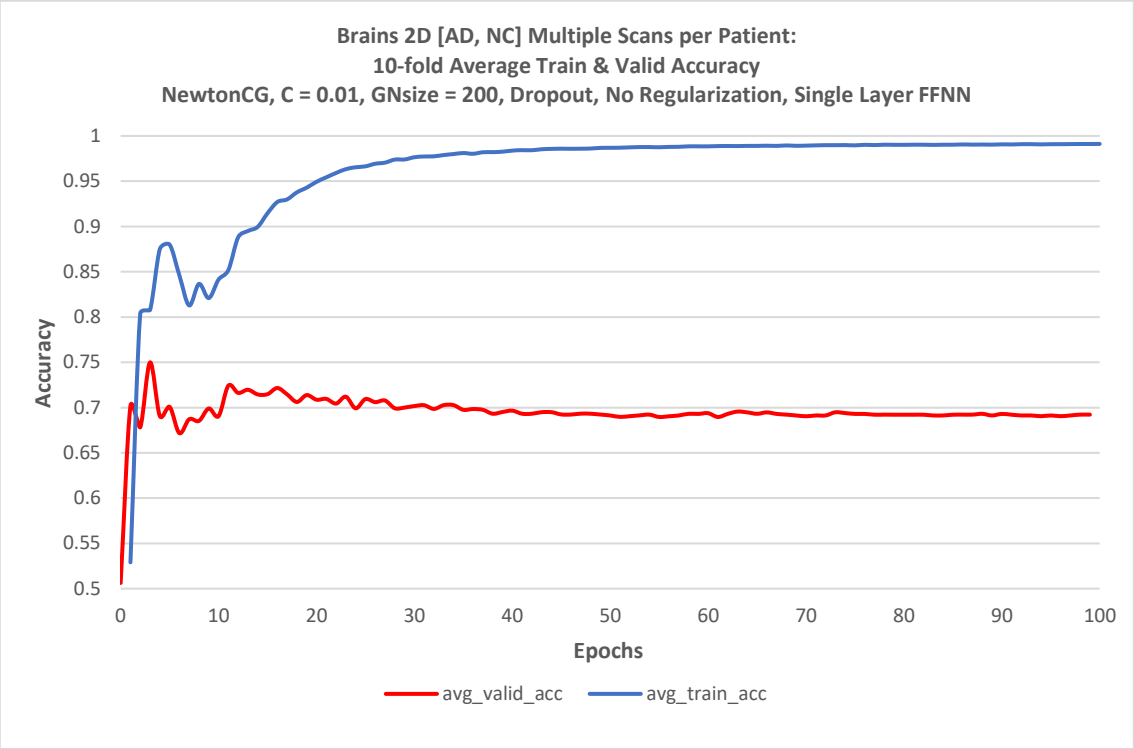


Figure H.118. Caption in the title.

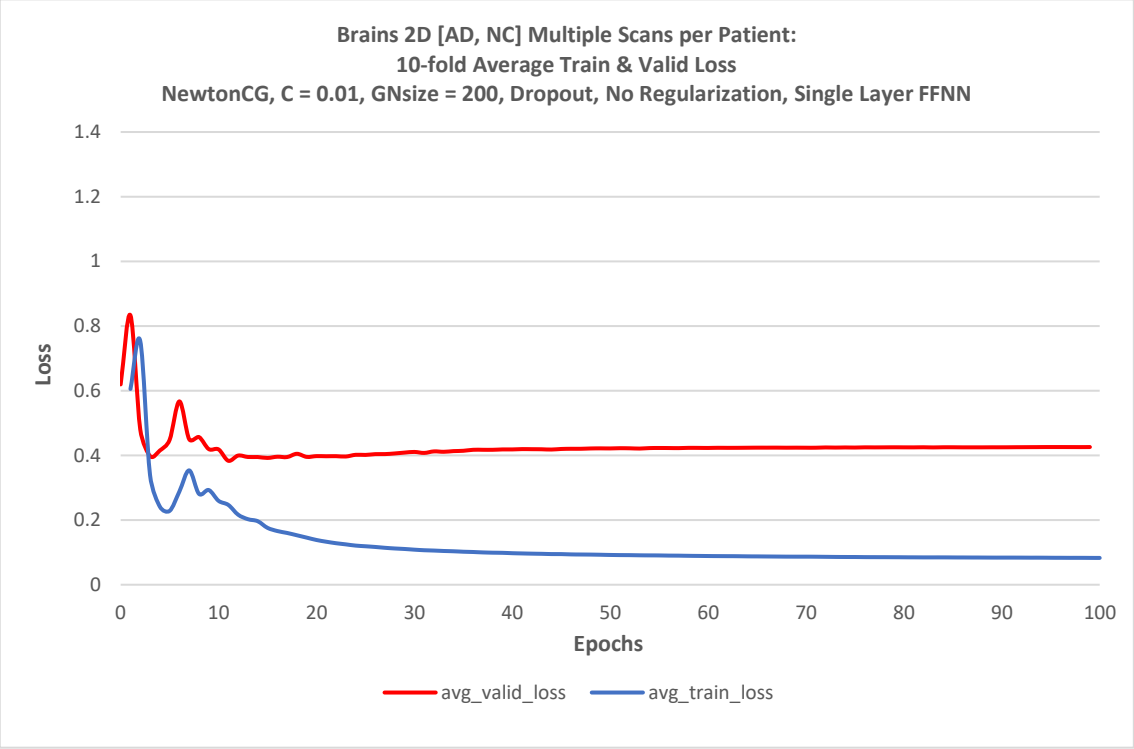


Figure H.119. Caption in the title.

H.2.11.2 Training

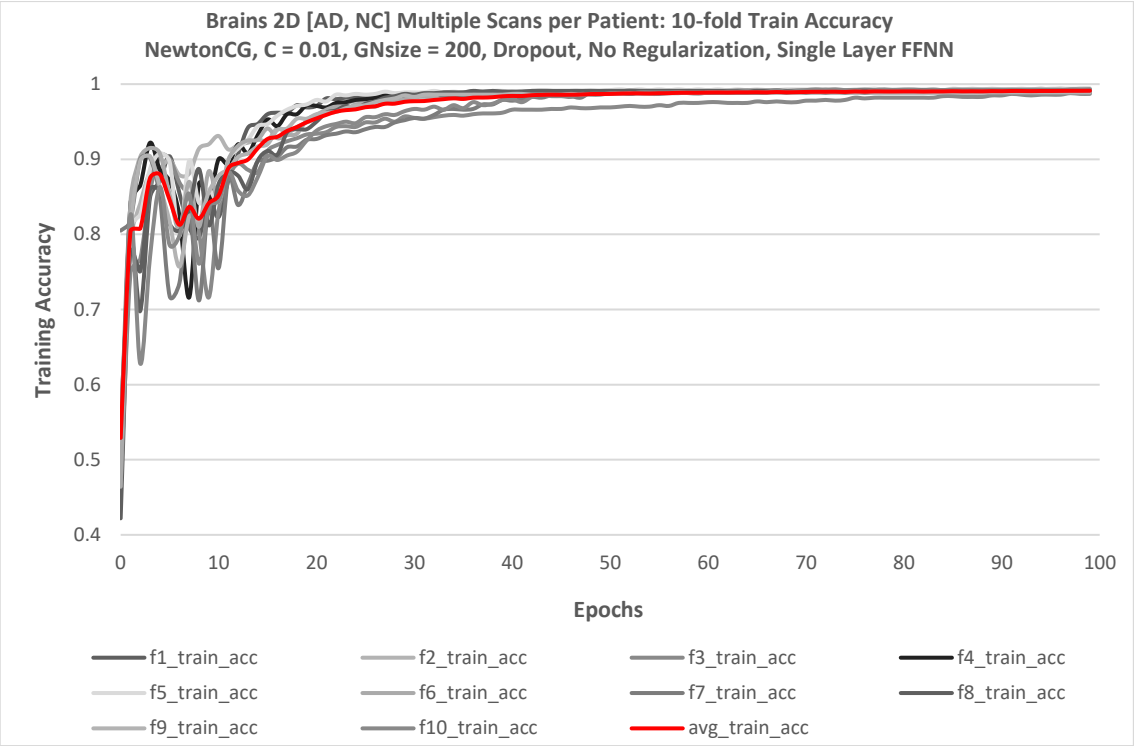


Figure H.120. Caption in the title.

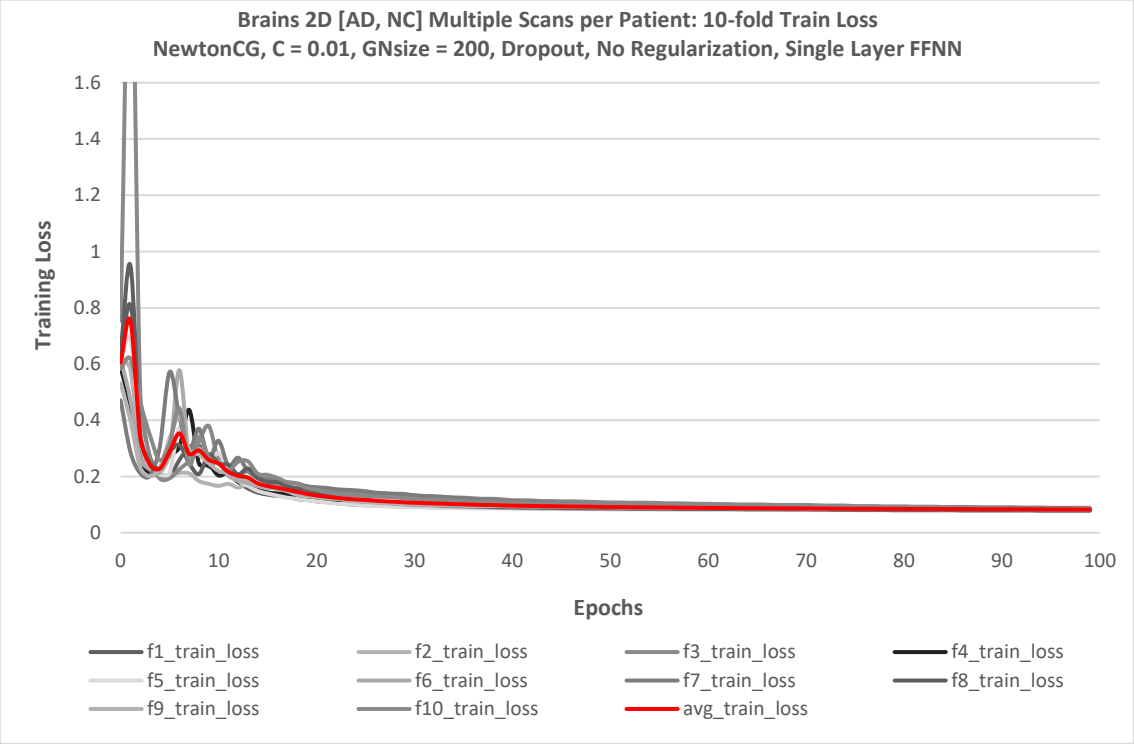


Figure H.121. Caption in the title.

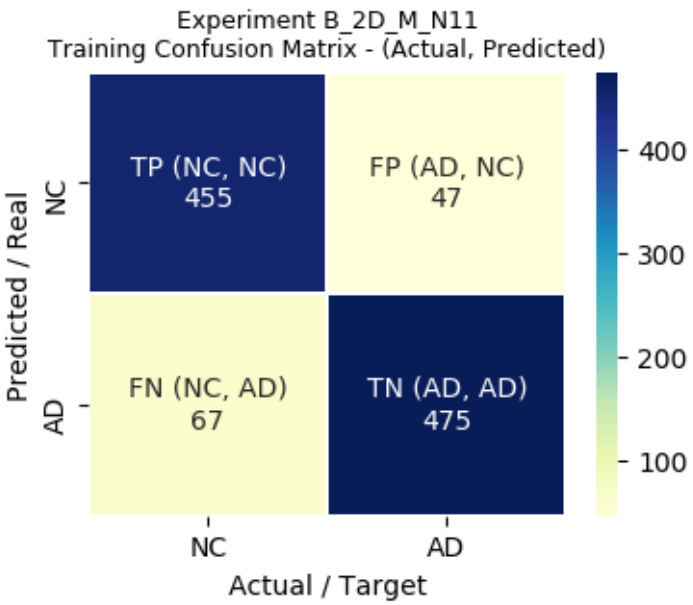


Figure H.122. Caption in the title.

H.2.11.3 Validation

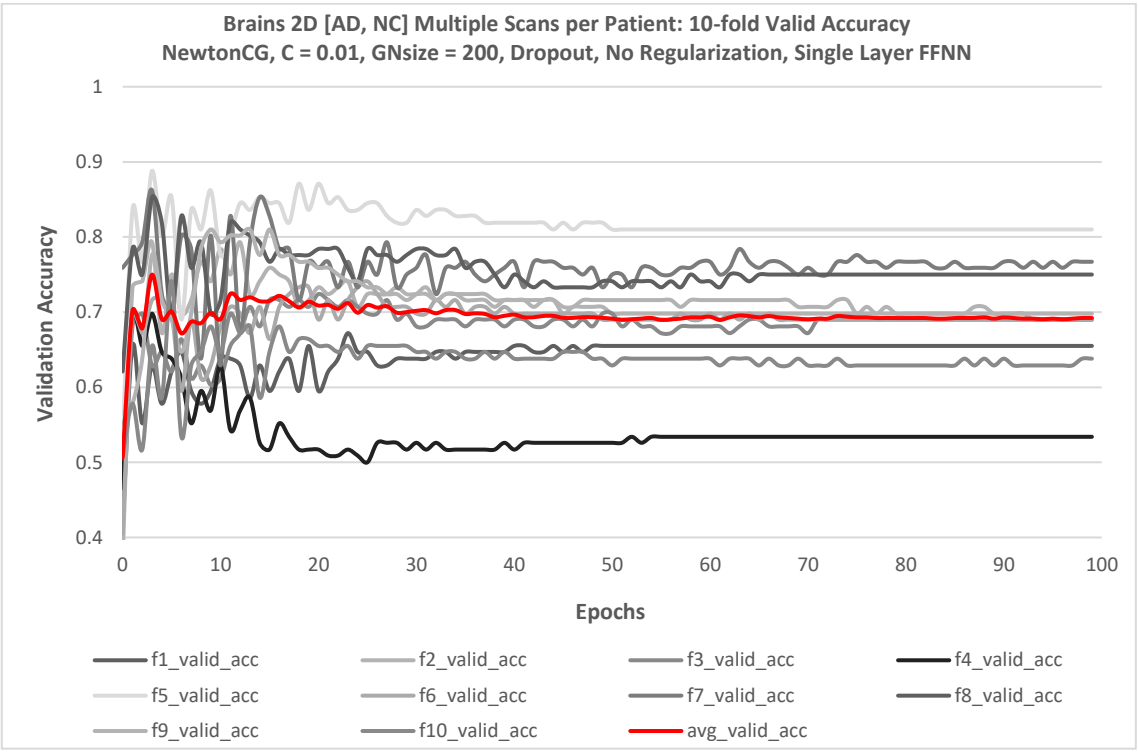


Figure H.123. Caption in the title.

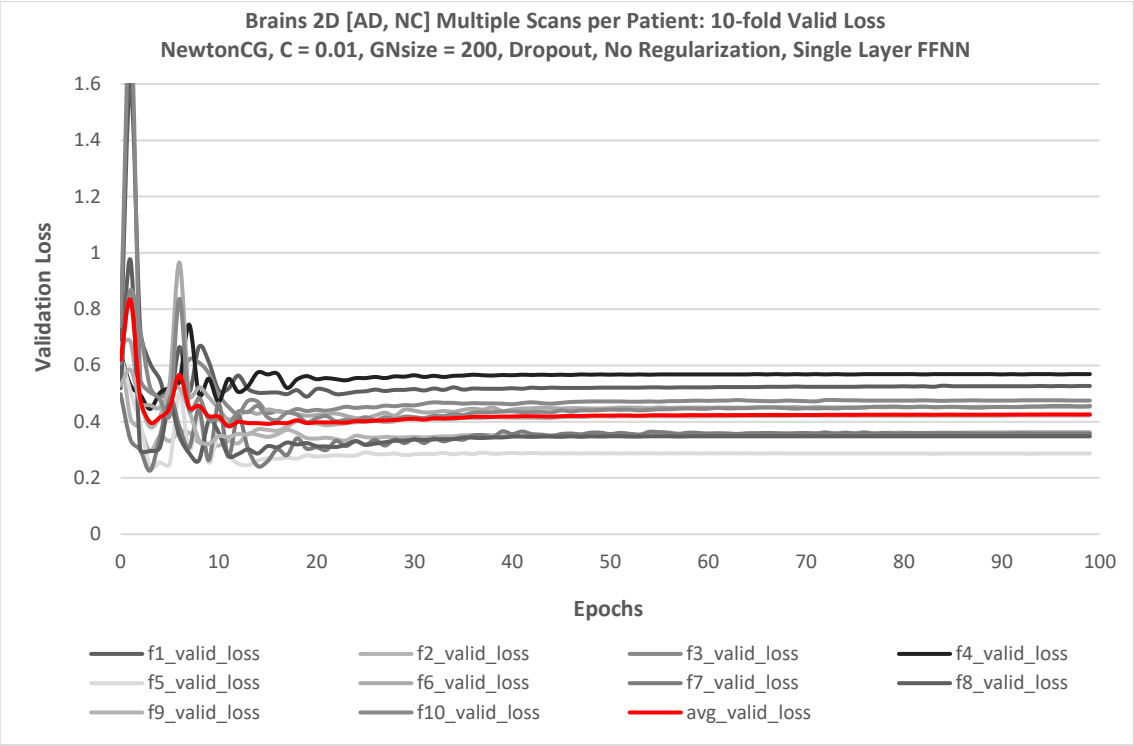


Figure H.124. Caption in the title.

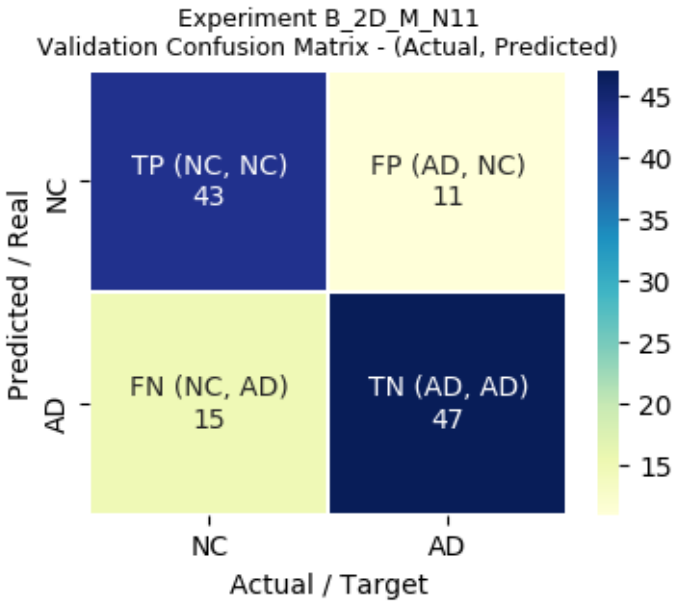


Figure H.125. Caption in the title.

H.2.11.4 Testing

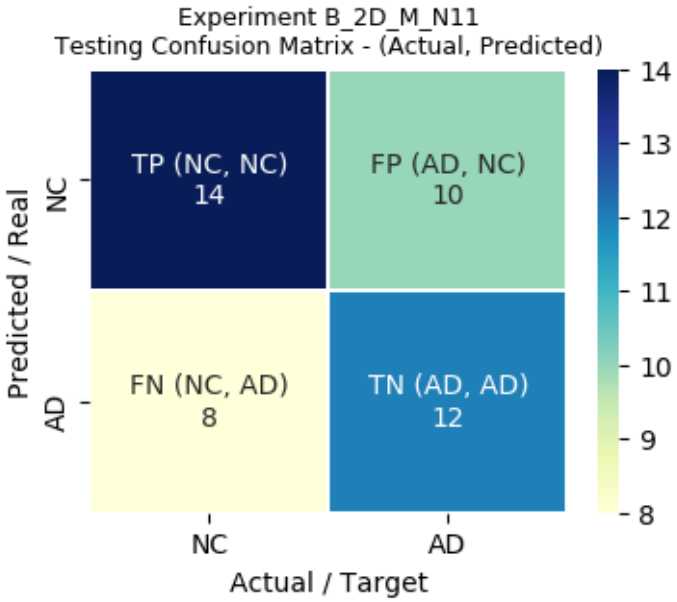


Figure H.126. Caption in the title.

H.2.12 Experiment B_2D_M_N12: C = 0.01, GNsize = 200

H.2.12.1 General

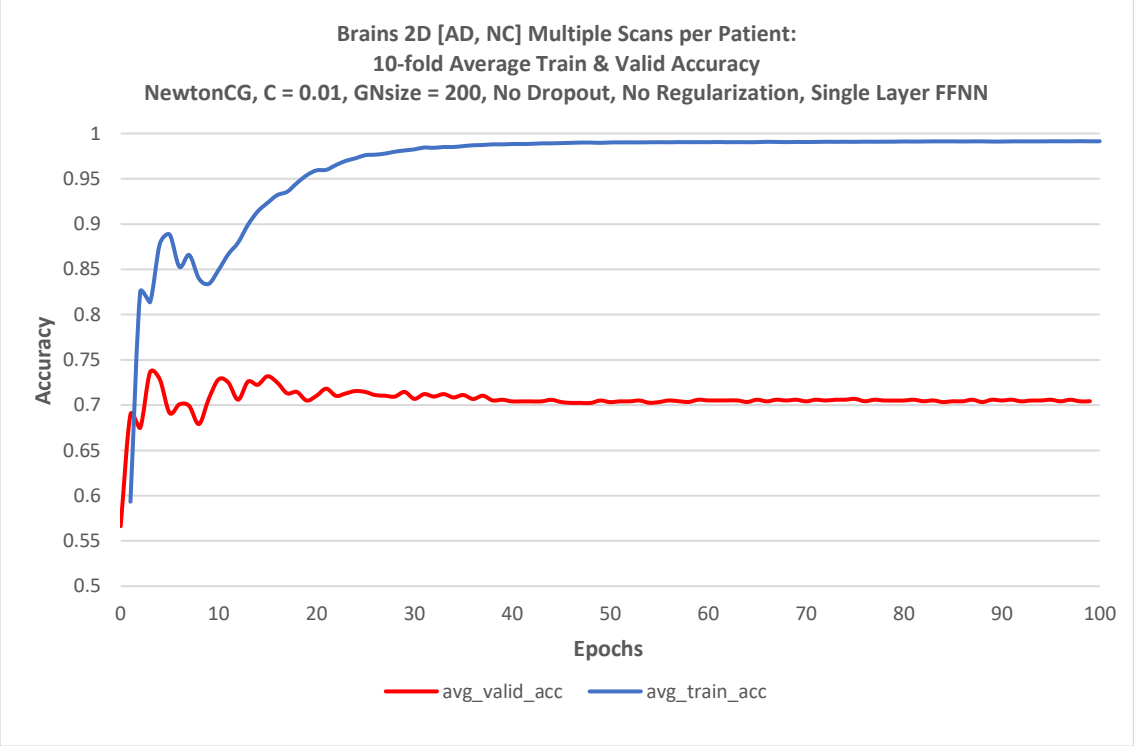


Figure H.127. Caption in the title.

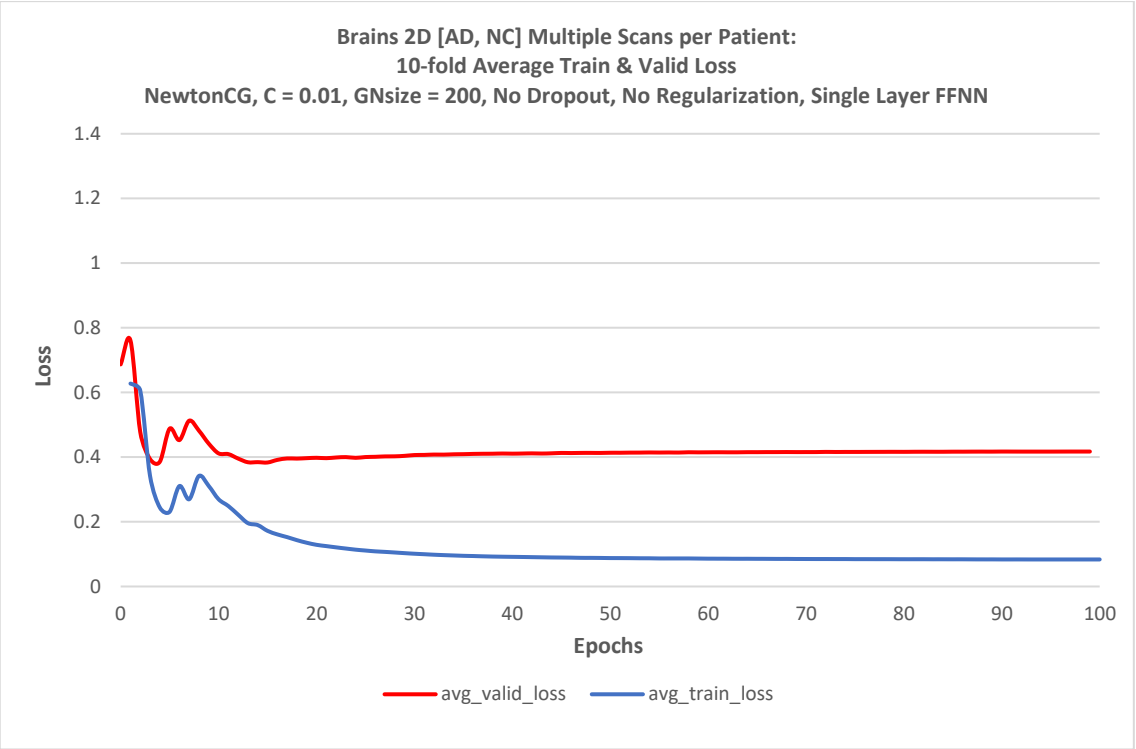


Figure H.128. Caption in the title.

H.2.12.2 Training

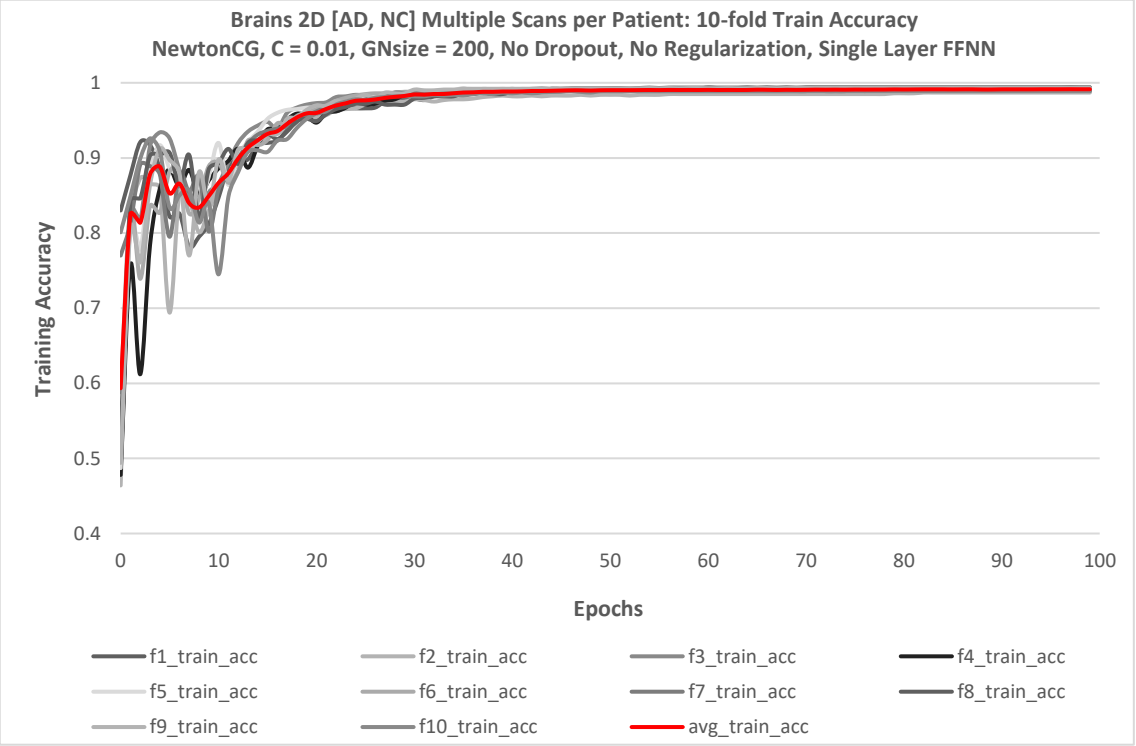


Figure H.129. Caption in the title.

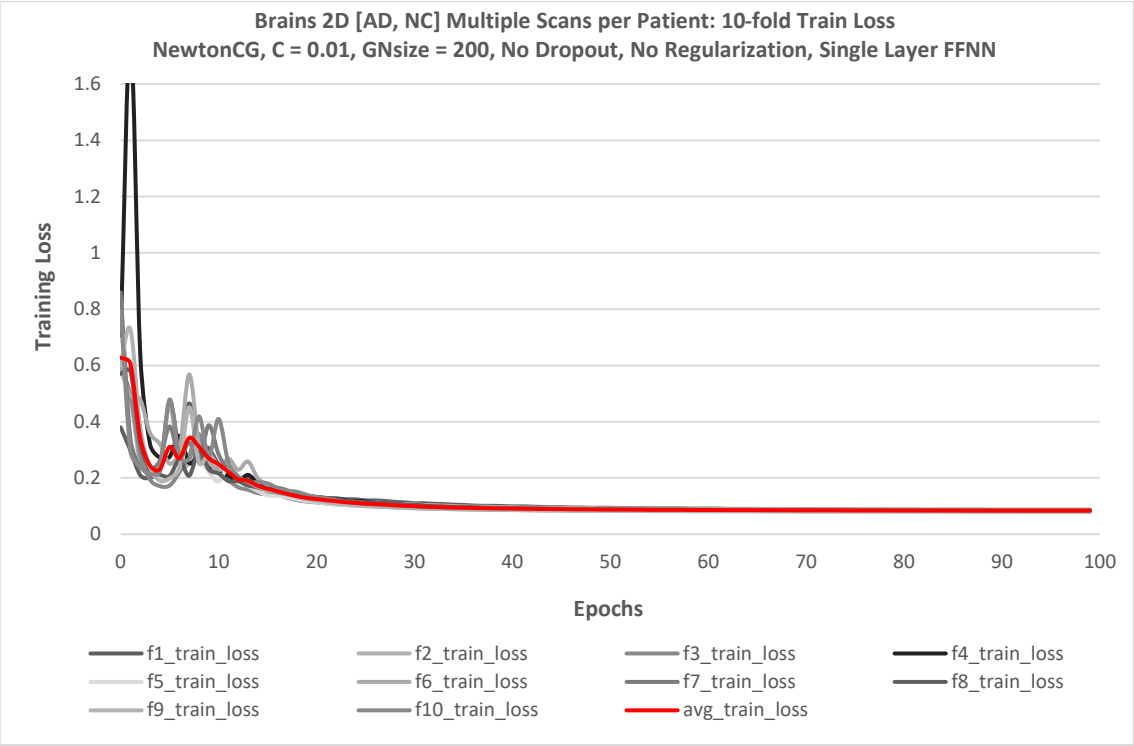


Figure H.130. Caption in the title.

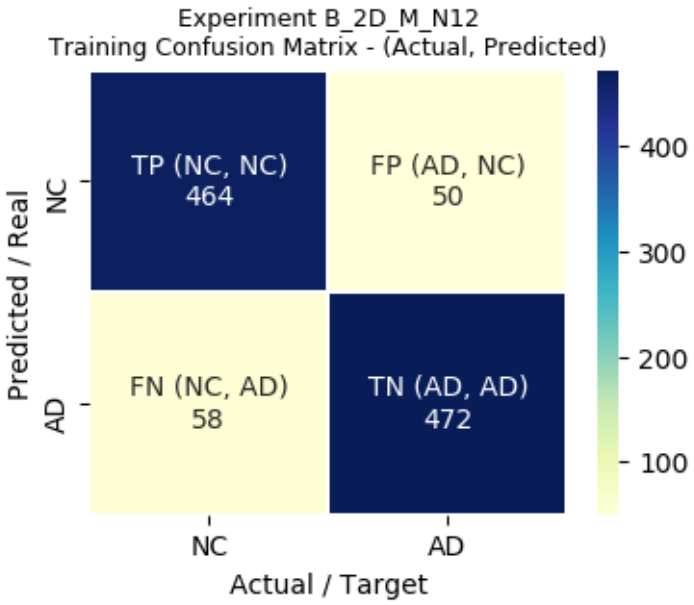


Figure H.131. Caption in the title.

H.2.12.3 Validation

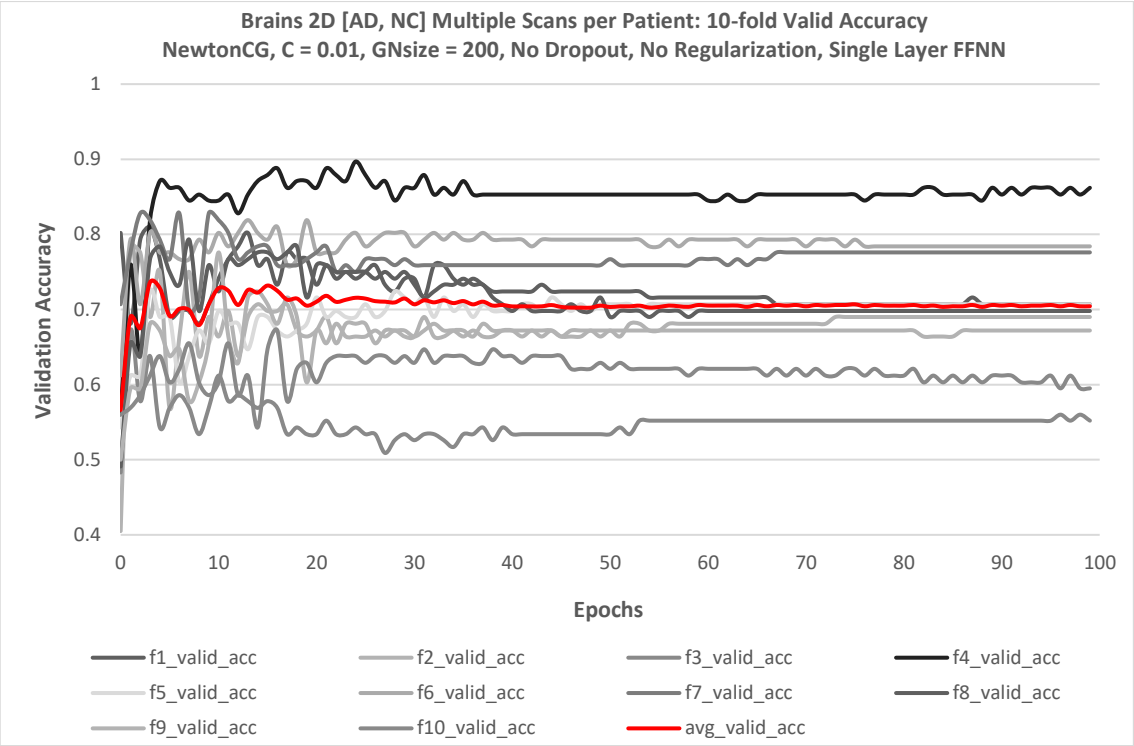


Figure H.132. Caption in the title.

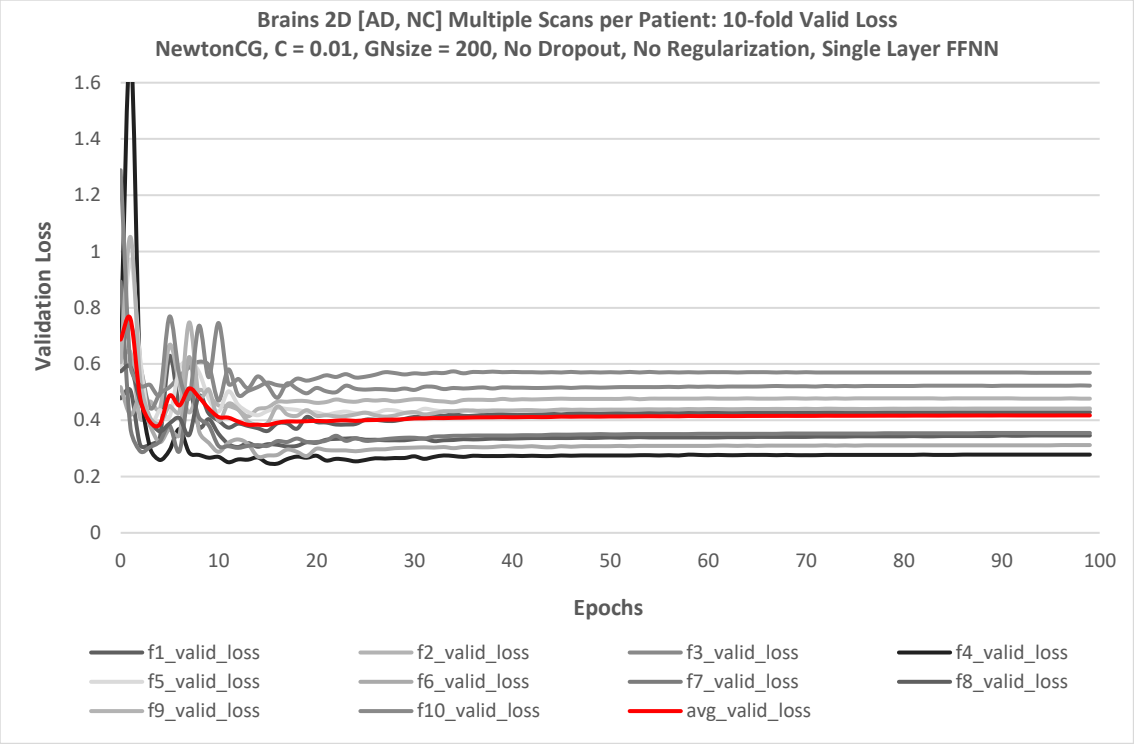


Figure H.133. Caption in the title.

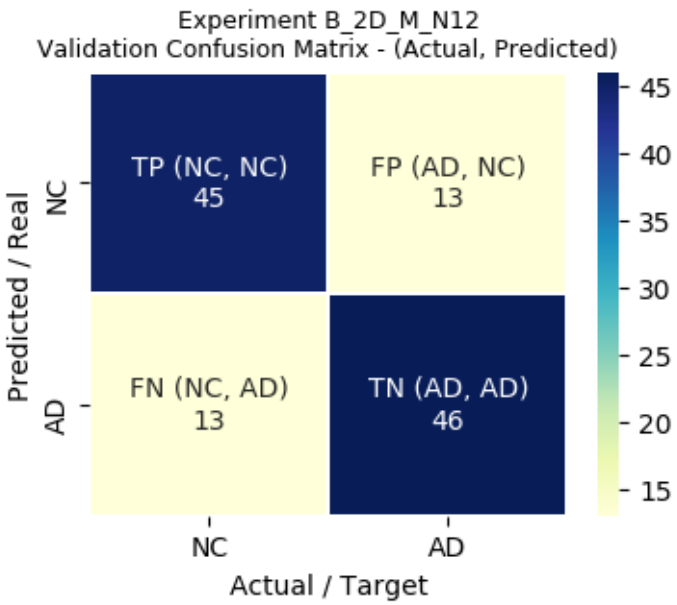


Figure H.134. Caption in the title.

H.2.12.4 Testing

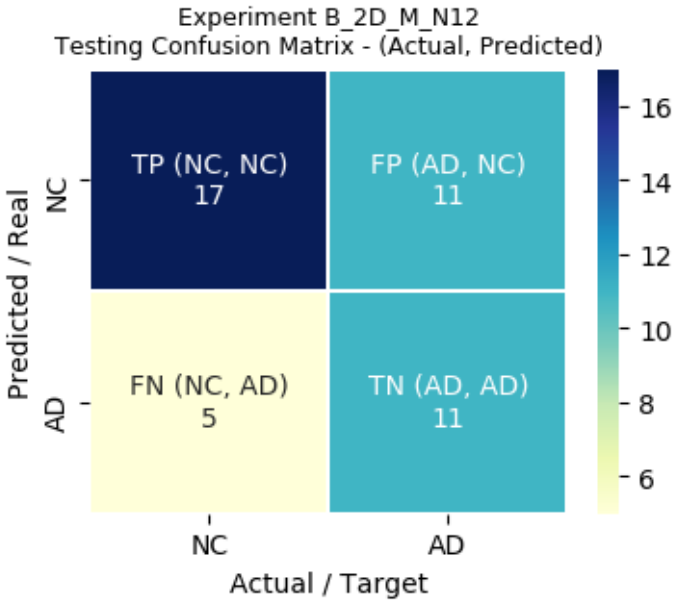


Figure H.135. Caption in the title.

H.2.13 Experiment B_2D_M_N13: C = 0.01, GNsize = 50

H.2.13.1 General

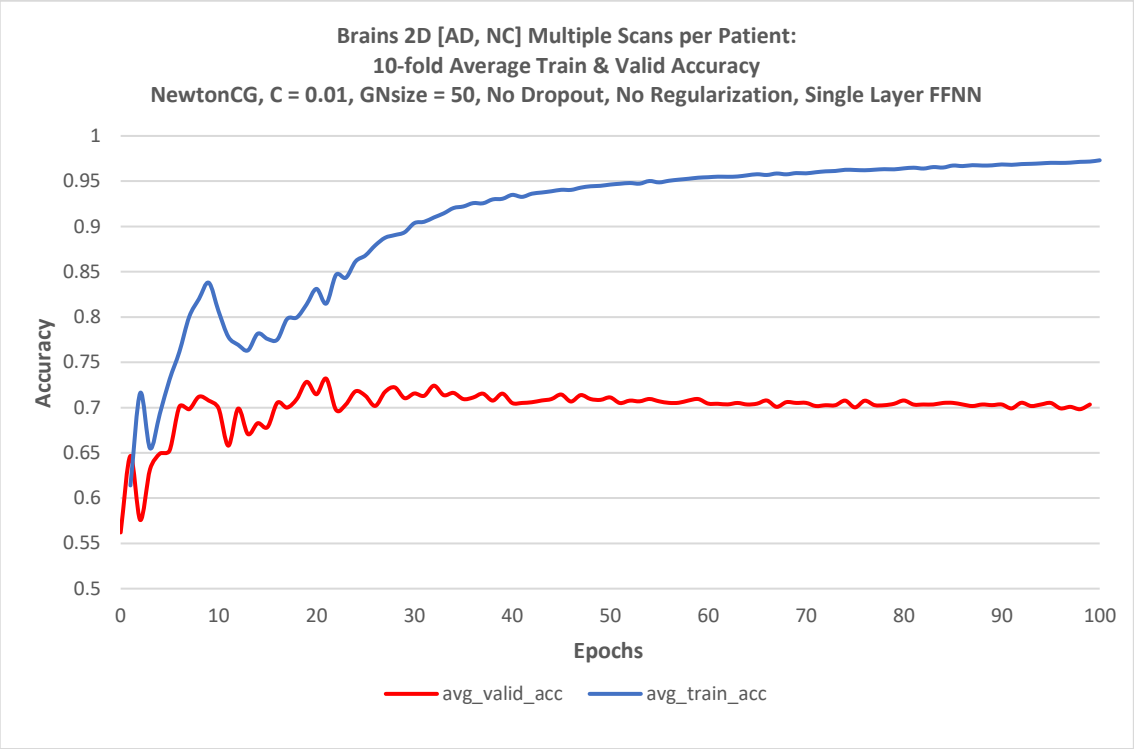


Figure H.136. Caption in the title.

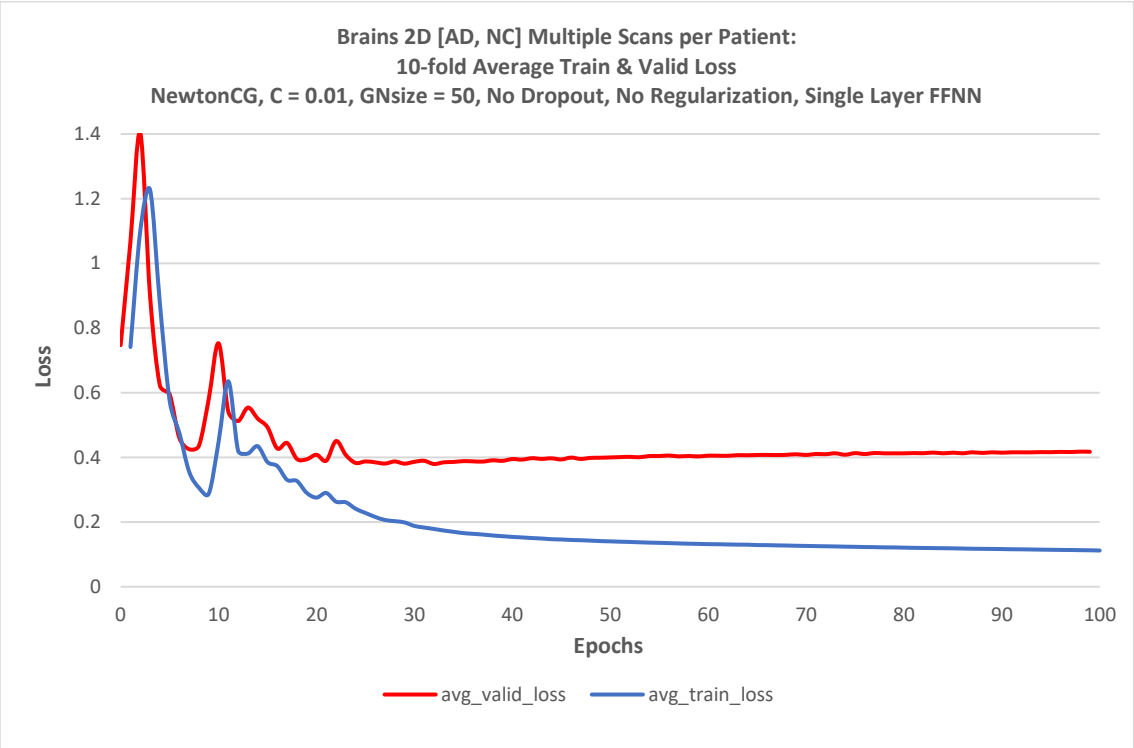


Figure H.137. Caption in the title.

H.2.13.2 Training

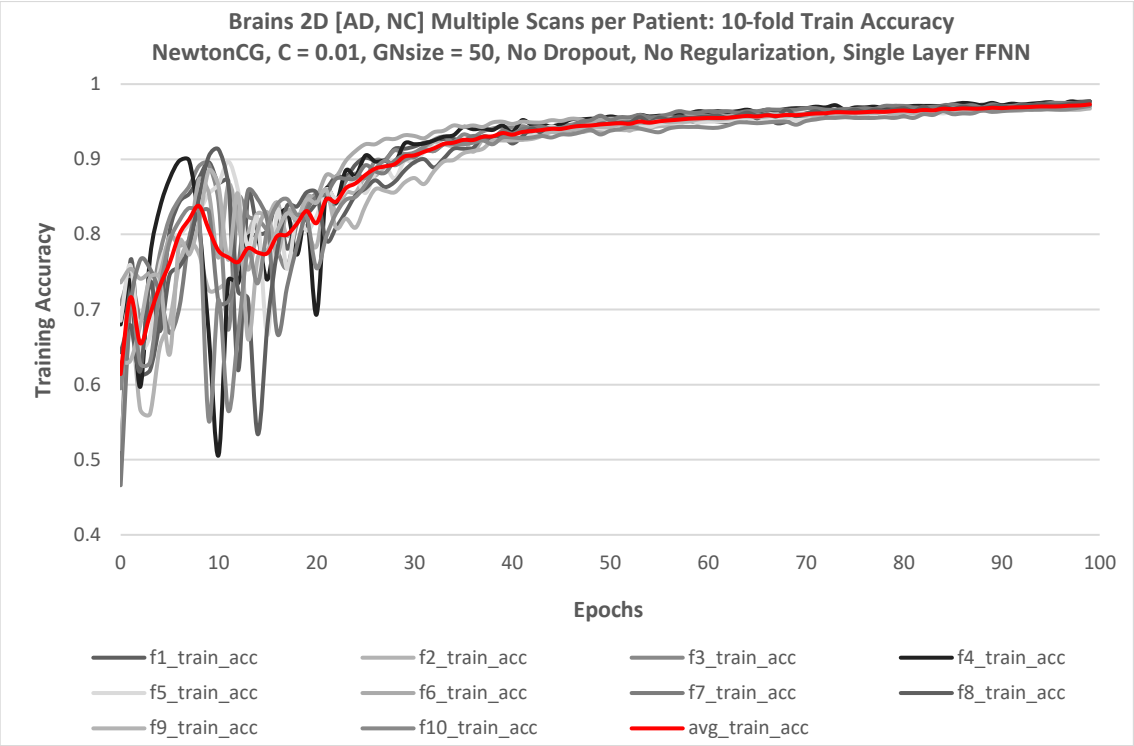


Figure H.138. Caption in the title.

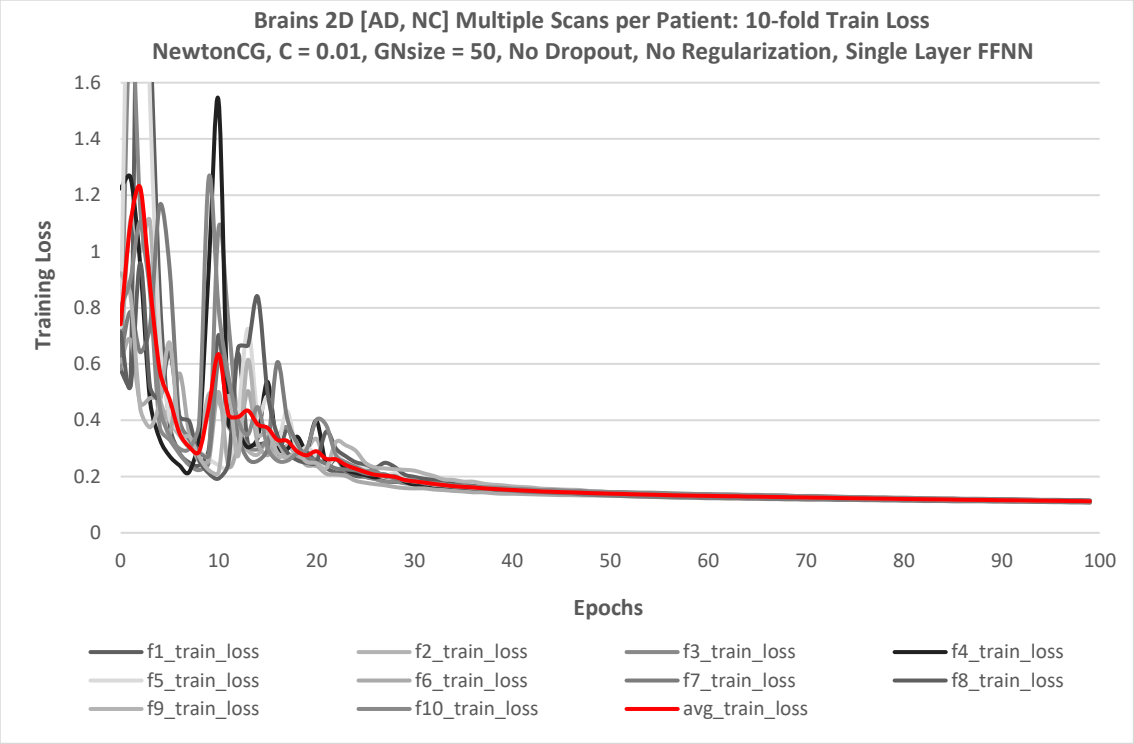


Figure H.139. Caption in the title.

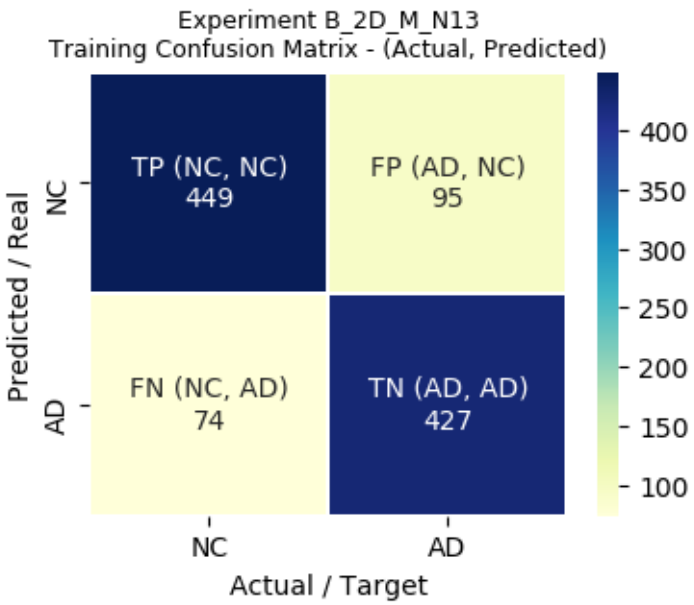


Figure H.140. Caption in the title.

H.2.13.3 Validation

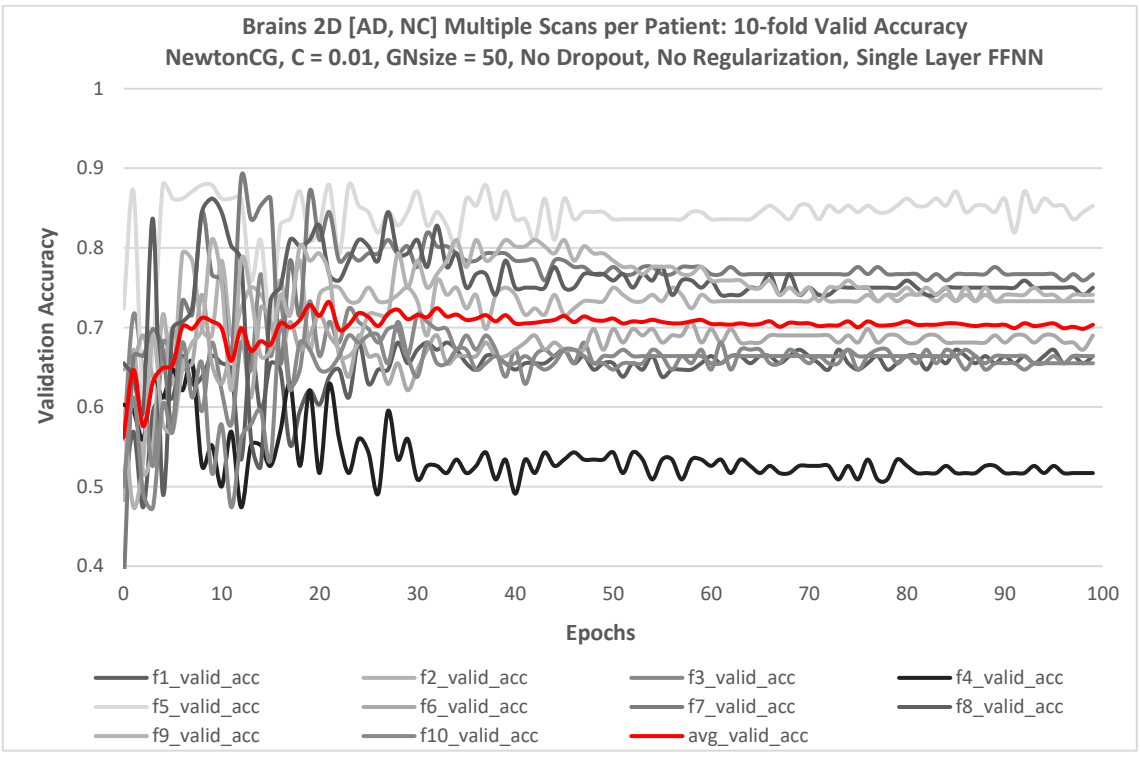


Figure H.141. Caption in the title.

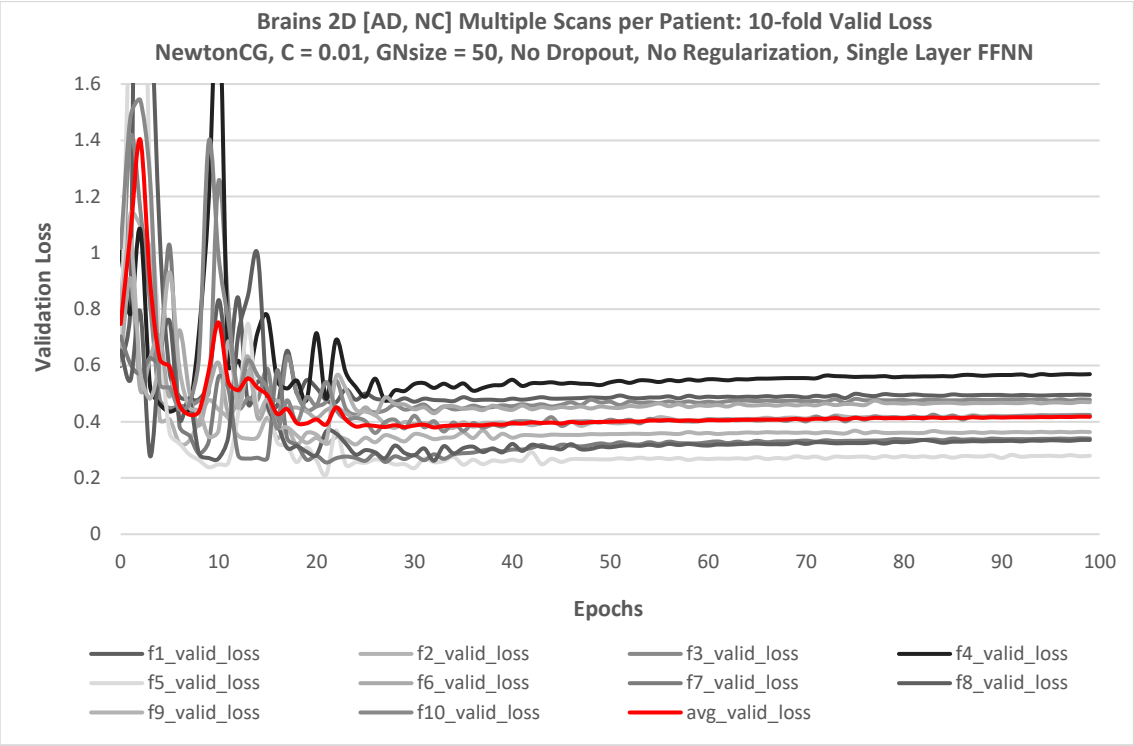


Figure H.142. Caption in the title.

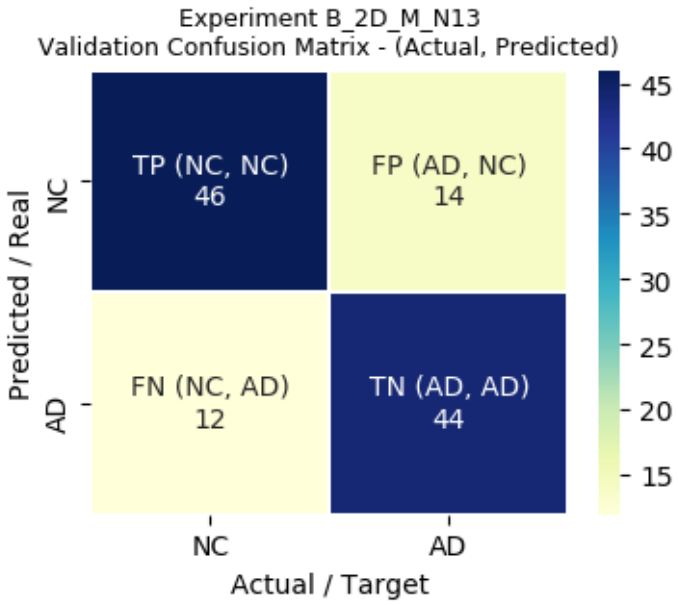


Figure H.143. Caption in the title.

H.2.13.4 Testing

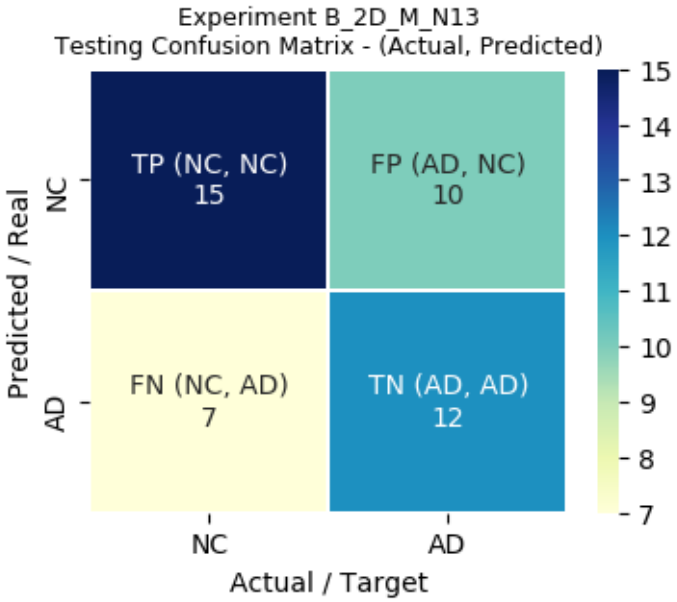


Figure H.144. Caption in the title.

H.2.14 Experiment B_2D_M_N14: C = 0.01, GNsize = 50, Dropout, No Max-Pooling

H.2.14.1 General

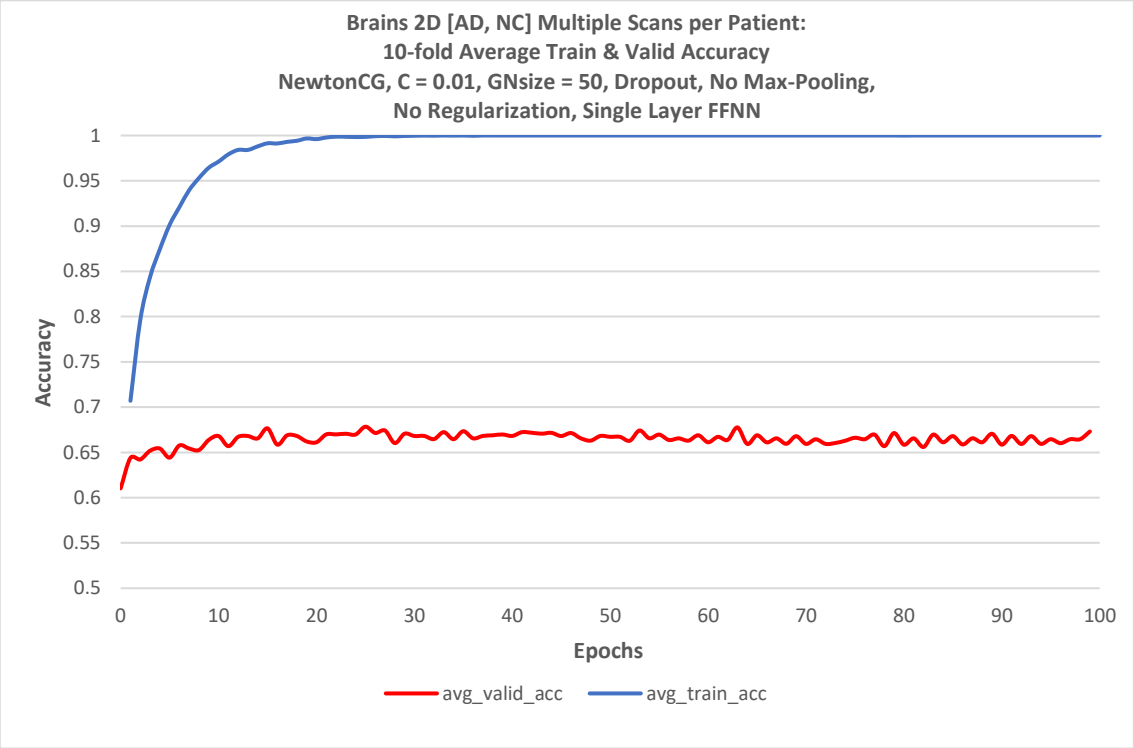


Figure H.145. Caption in the title.

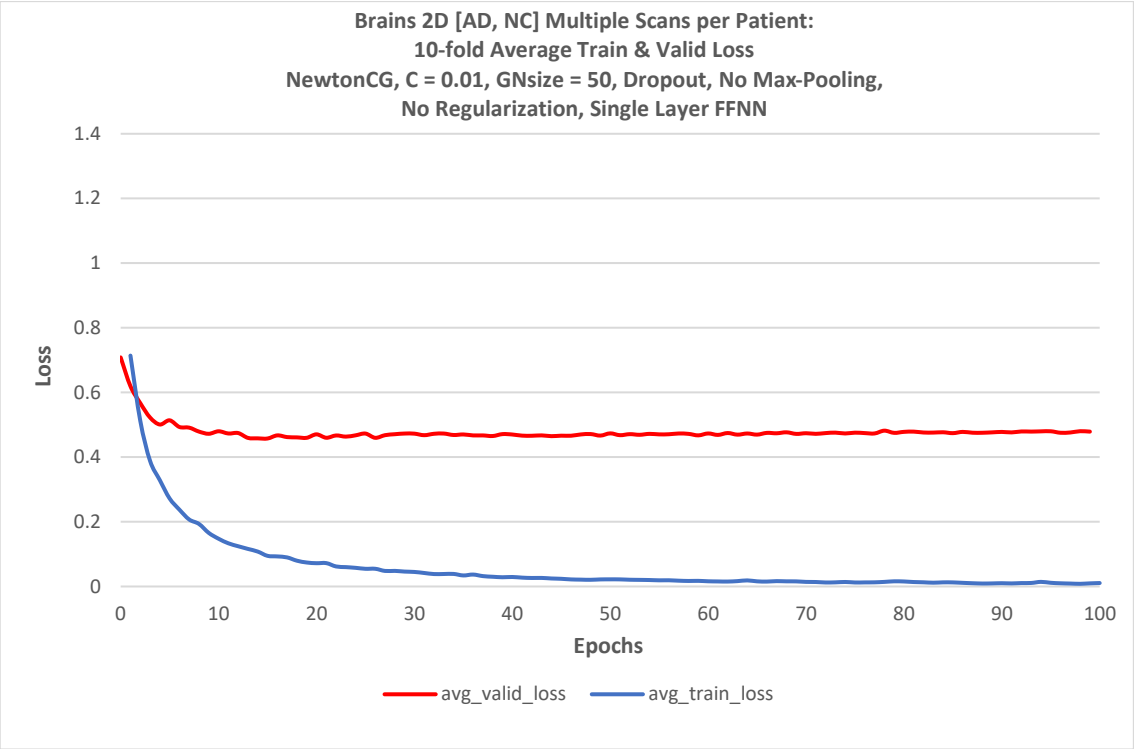


Figure H.146. Caption in the title.

H.2.14.2 Training

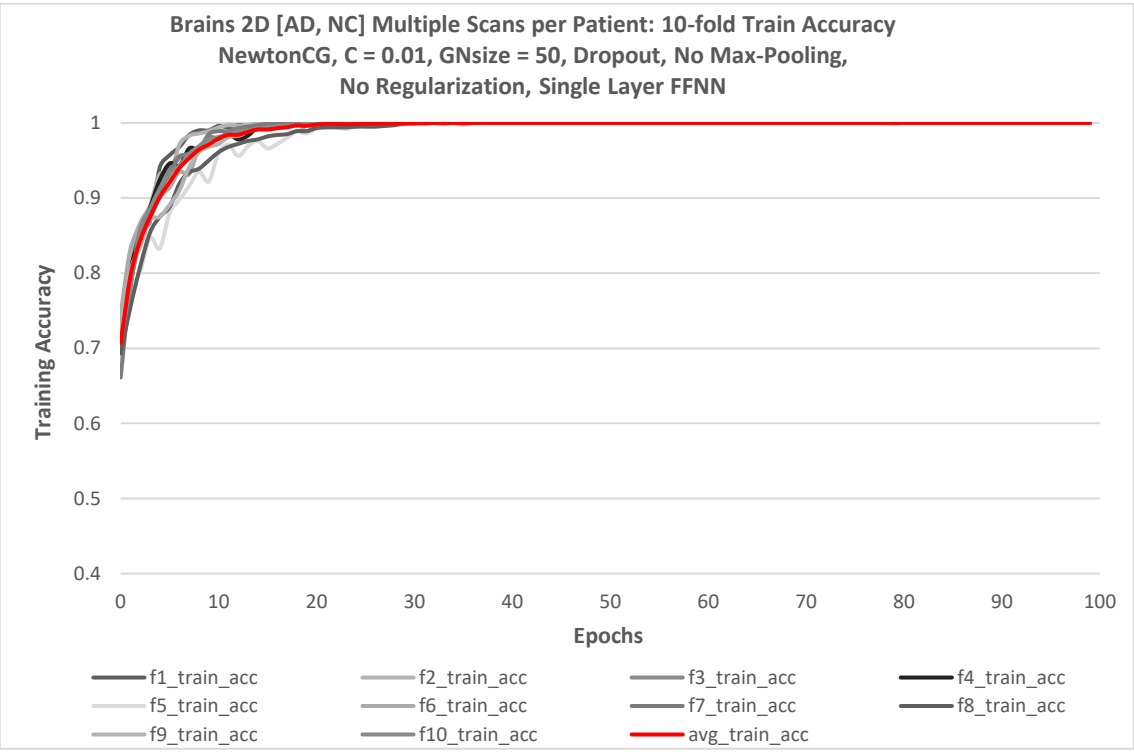


Figure H.147. Caption in the title.

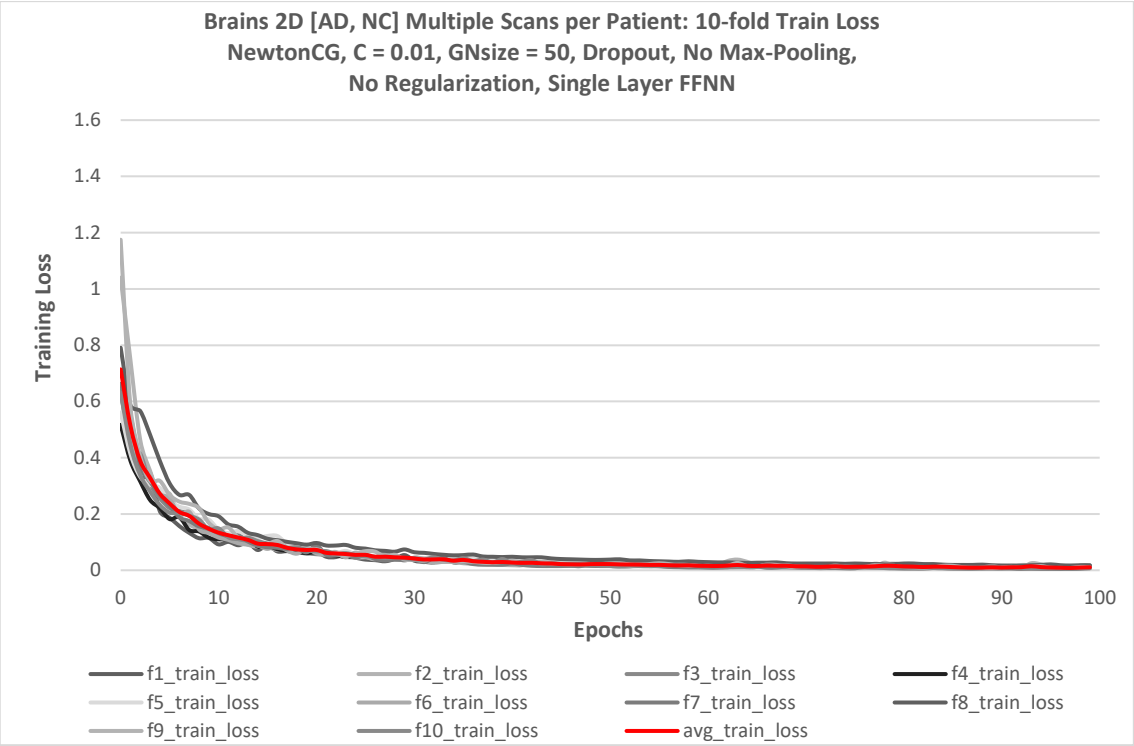


Figure H.148. Caption in the title.

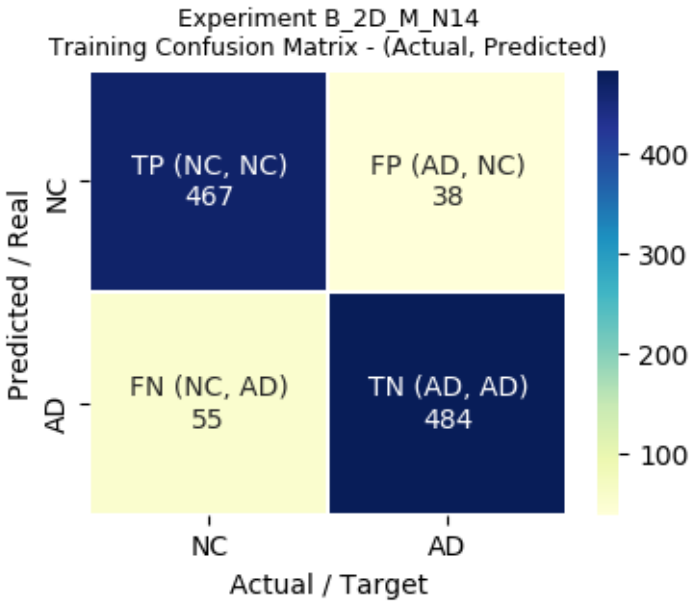


Figure H.149. Caption in the title.

H.2.14.3 Validation

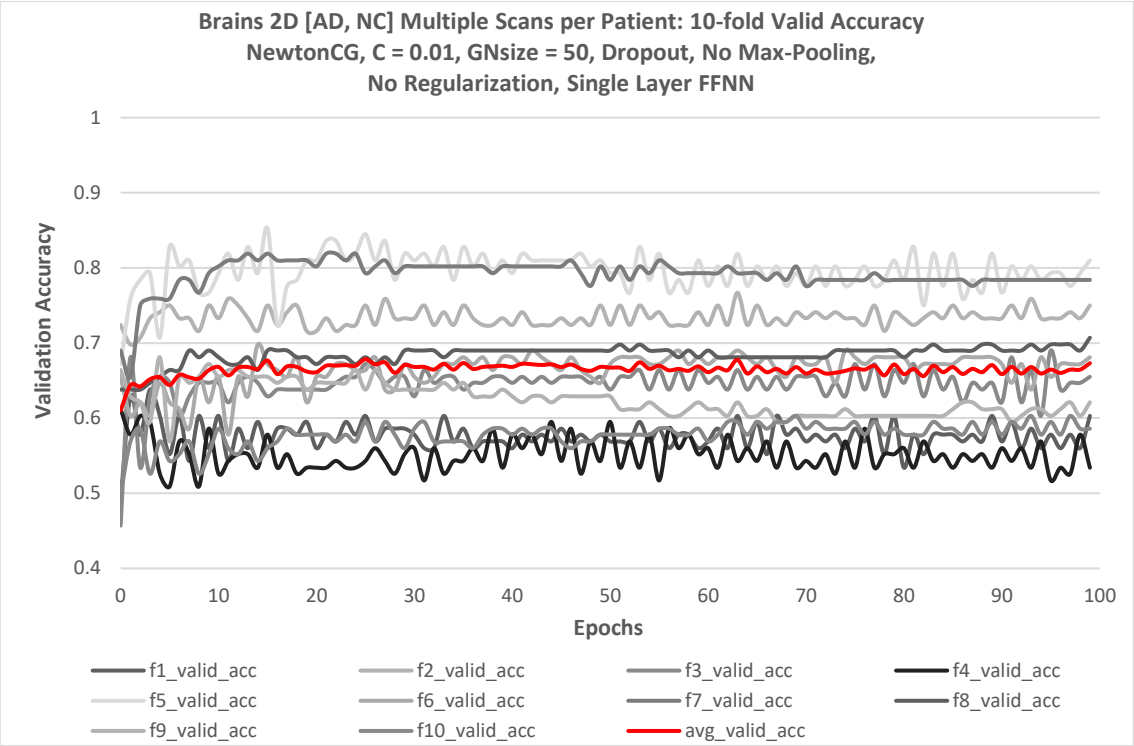


Figure H.150. Caption in the title.

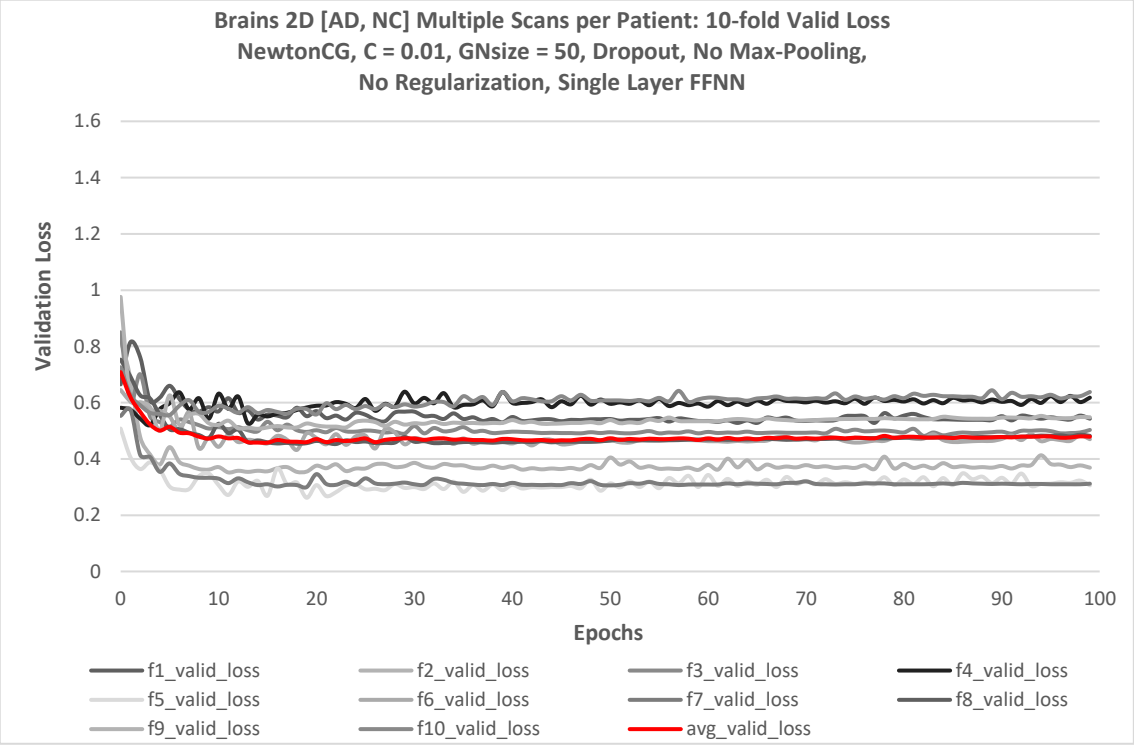


Figure H.151. Caption in the title.

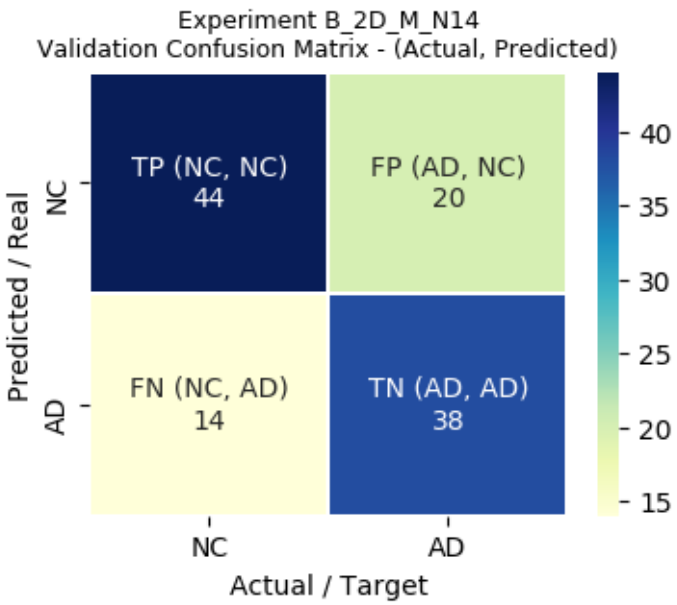


Figure H.152. Caption in the title.

H.2.14.4 Testing

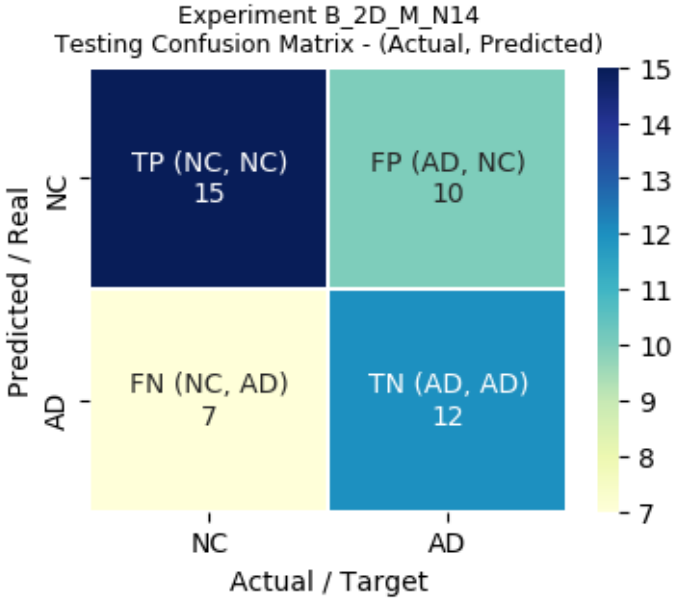


Figure H.153. Caption in the title.

H.2.15 Experiment B_2D_M_N16: C = 0.01, GNsize = 50, Dropout, Loss = Cross Entropy

H.2.15.1 General

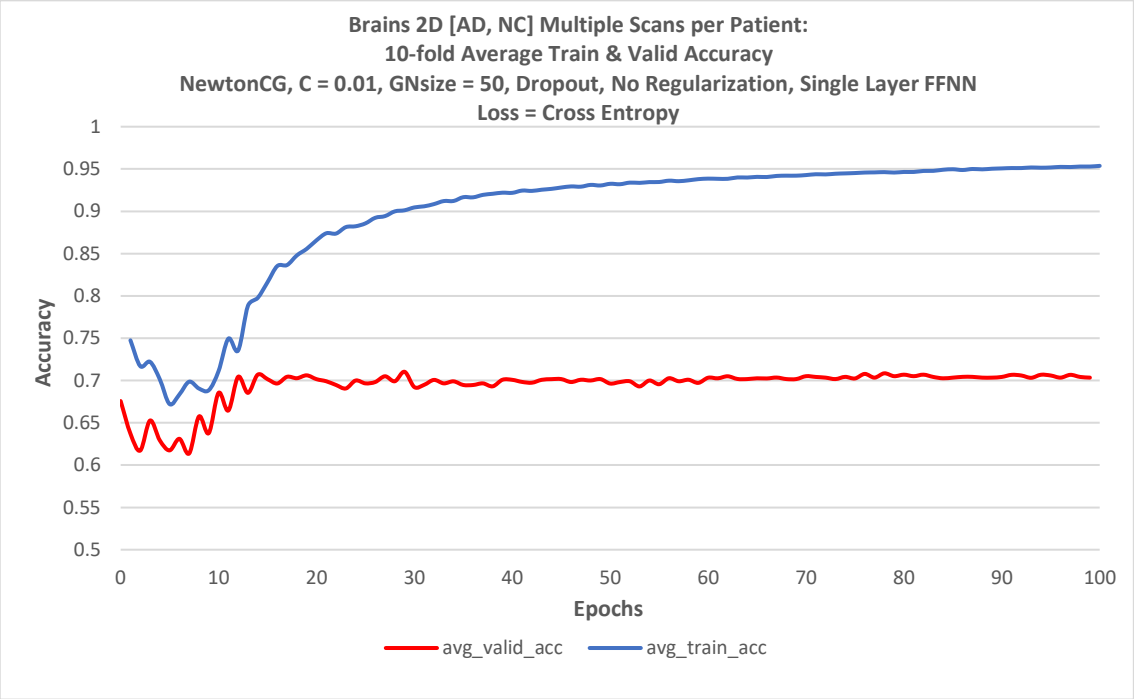


Figure H.154. Caption in the title.

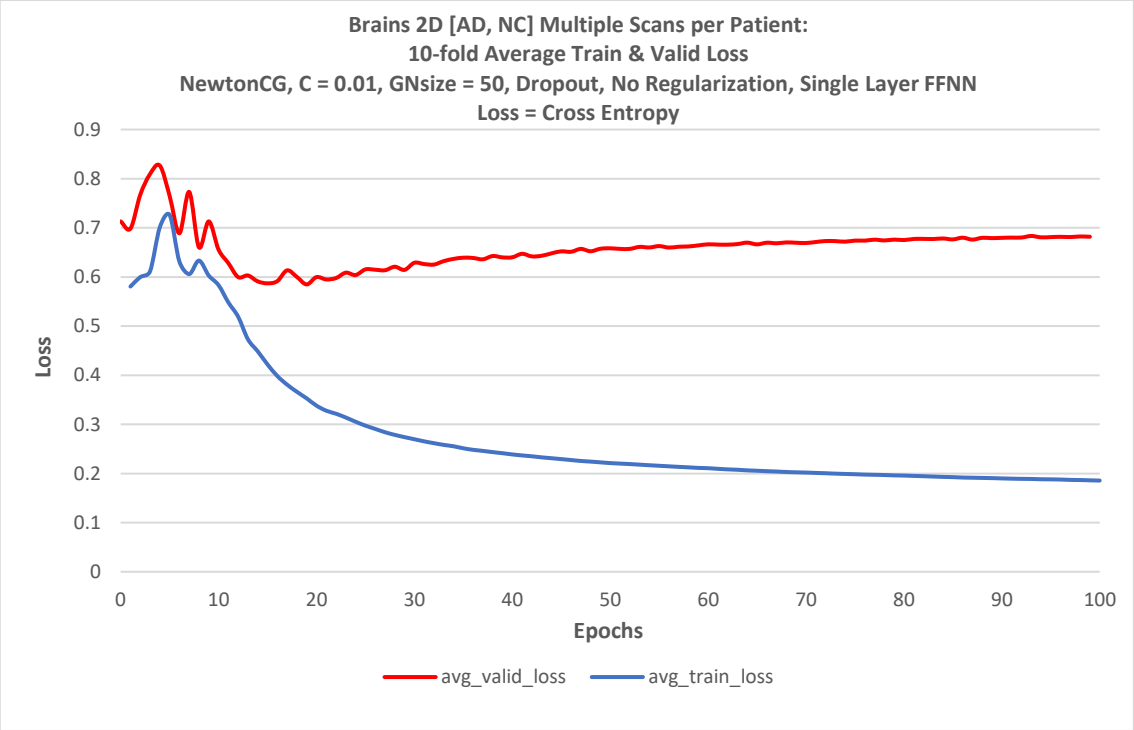


Figure H.155. Caption in the title.

H.2.15.2 Training

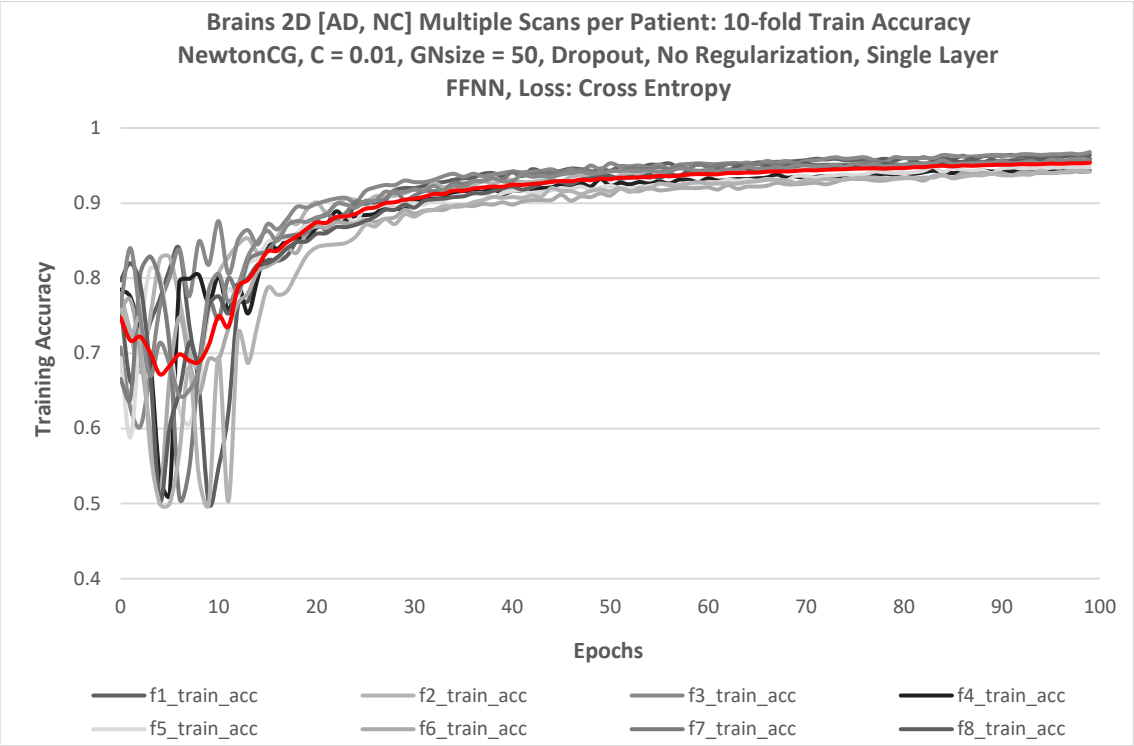


Figure H.156. Caption in the title.

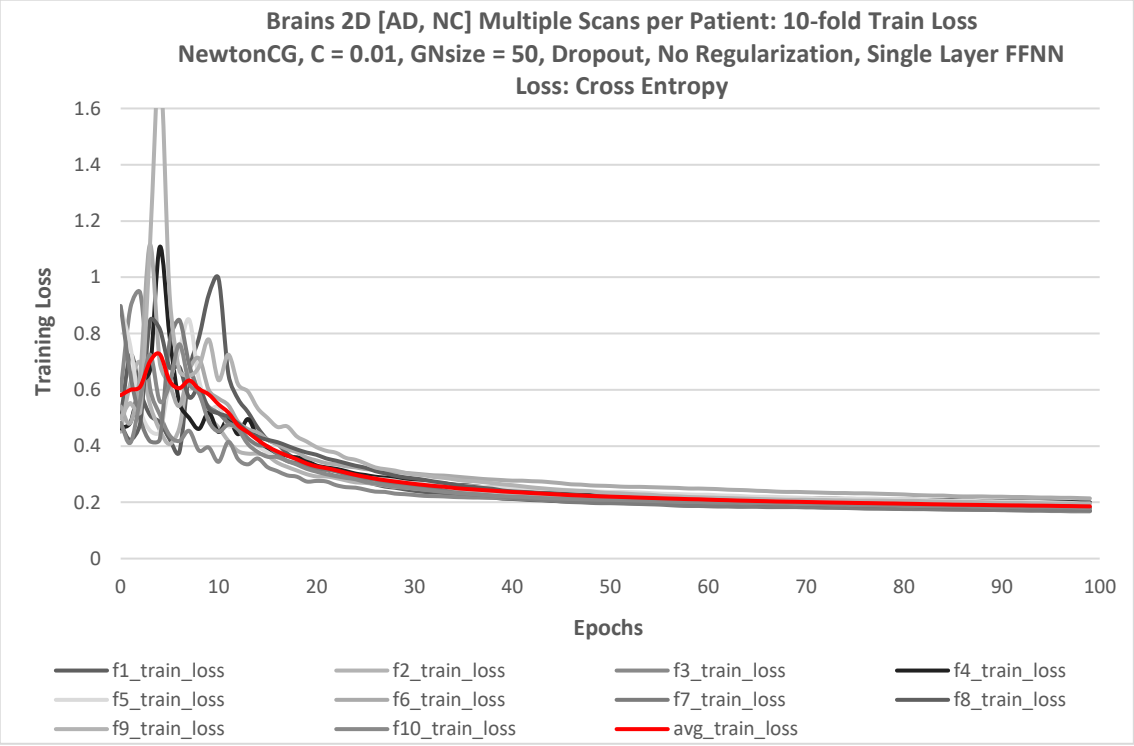


Figure H.157. Caption in the title.

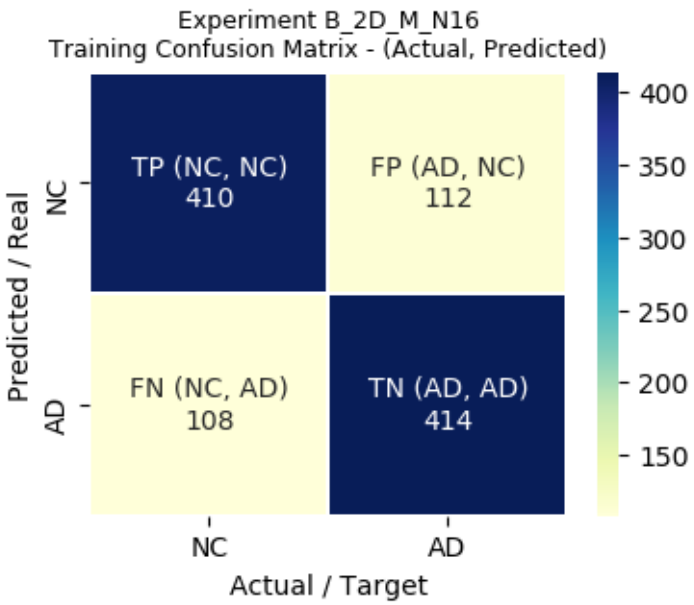


Figure H.158. Caption in the title.

H.2.15.3 Validation

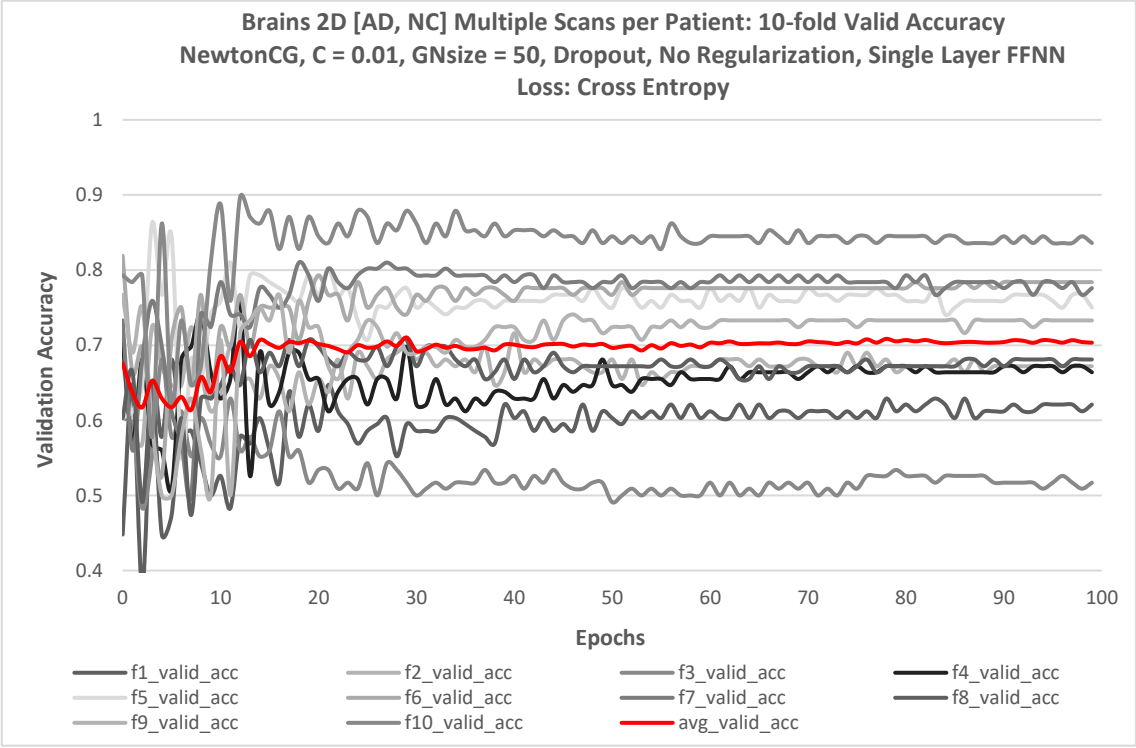


Figure H.159. Caption in the title.

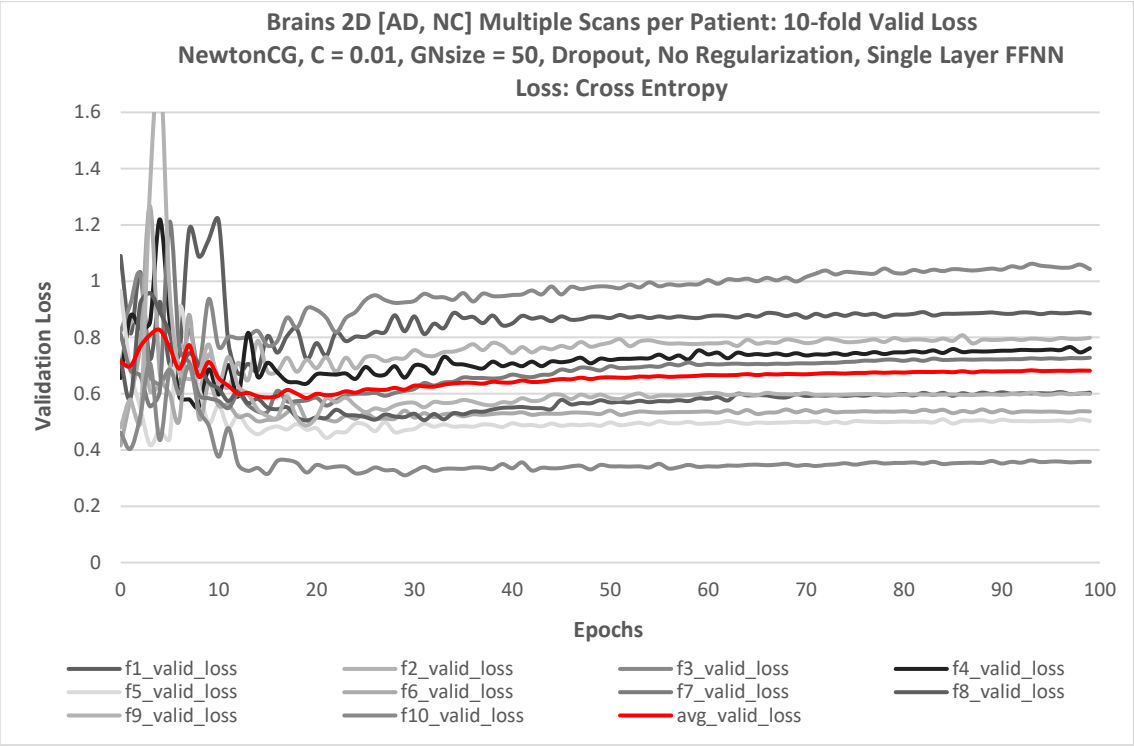


Figure H.160. Caption in the title.

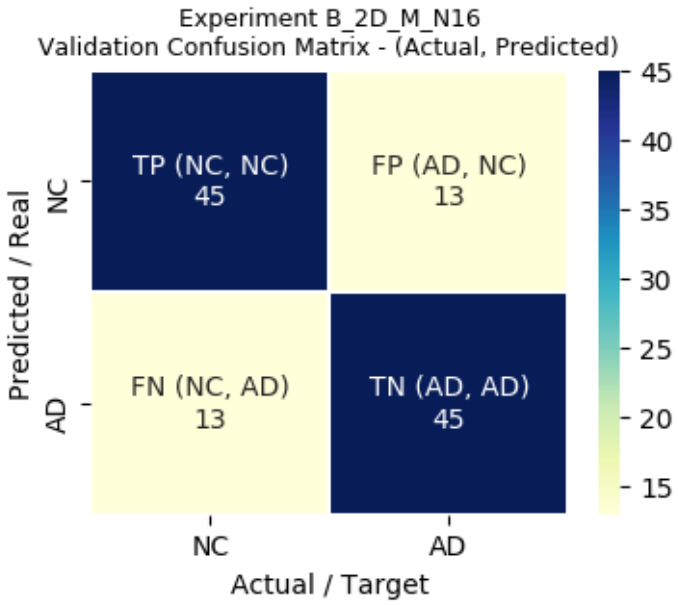


Figure H.161. Caption in the title.

H.2.15.4 Testing

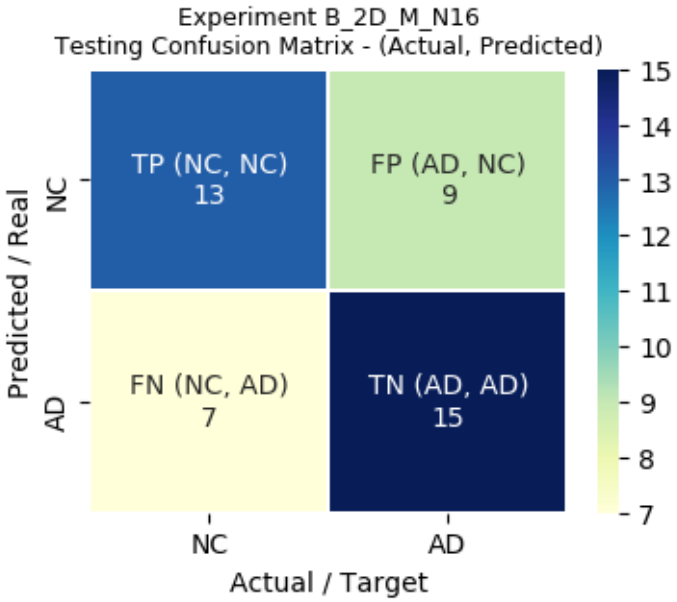


Figure H.162. Caption in the title.

H.2.16 Experiment B_2D_M_N17: C = 0.01, GNsize = 50, Dropout, L1 & L2 Regularization

H.2.16.1 General

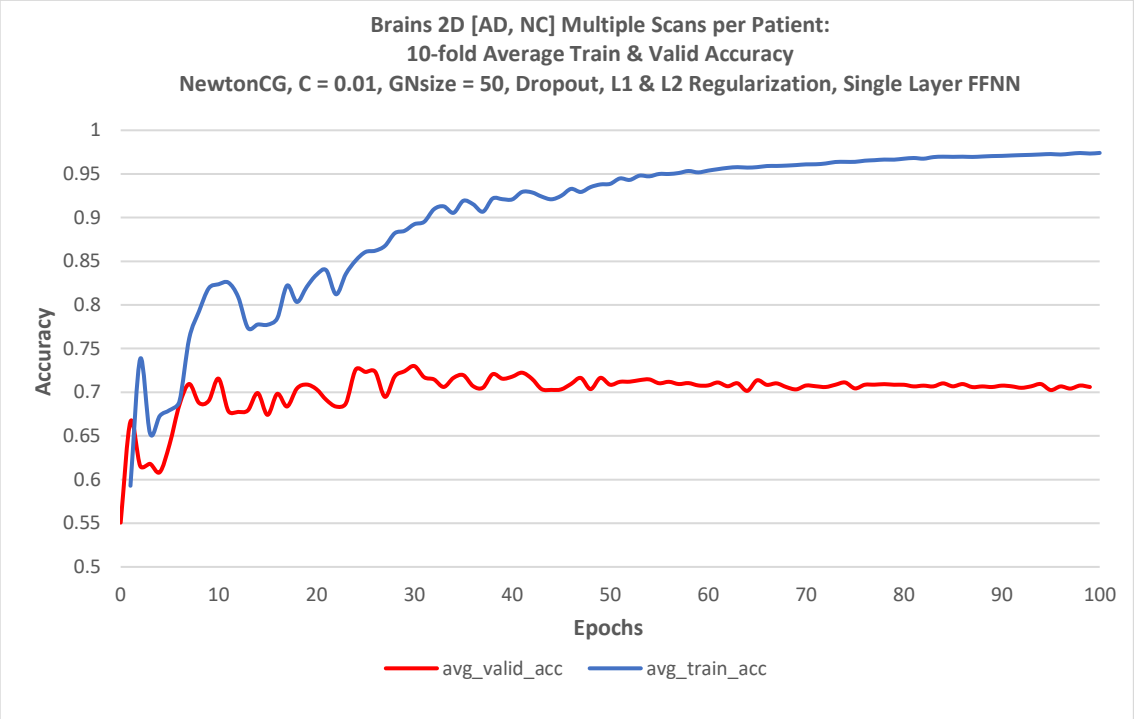


Figure H.163. Caption in the title.

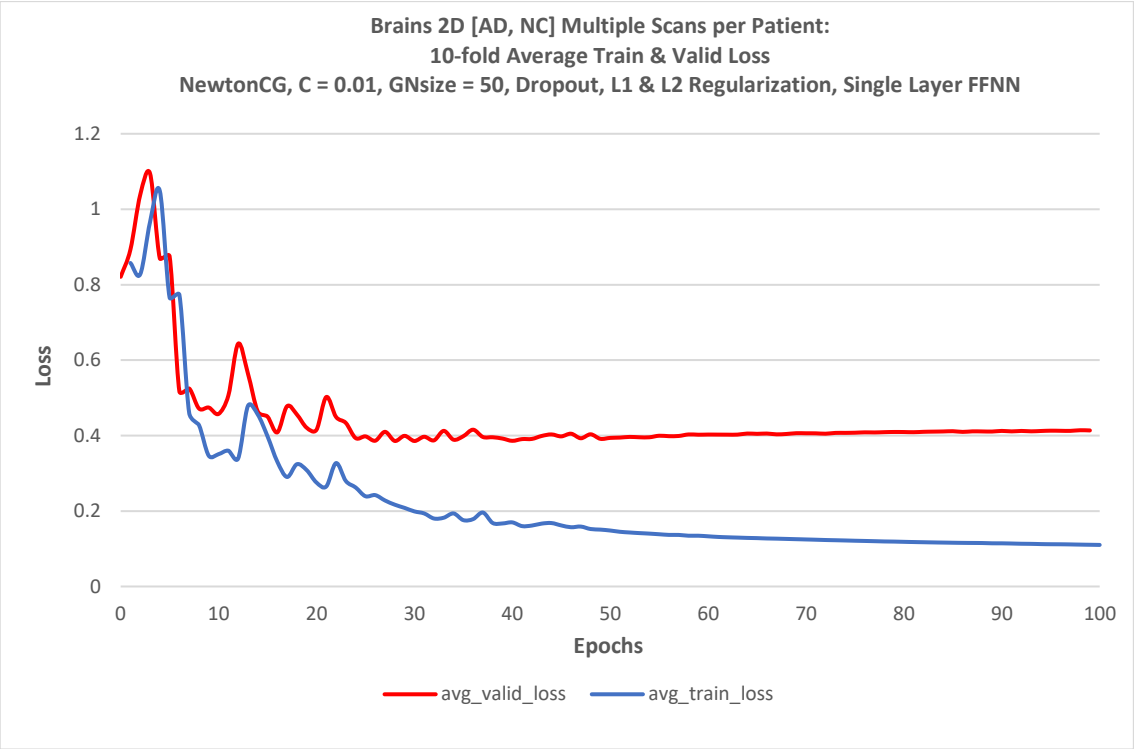


Figure H.164. Caption in the title.

H.2.16.2 Training

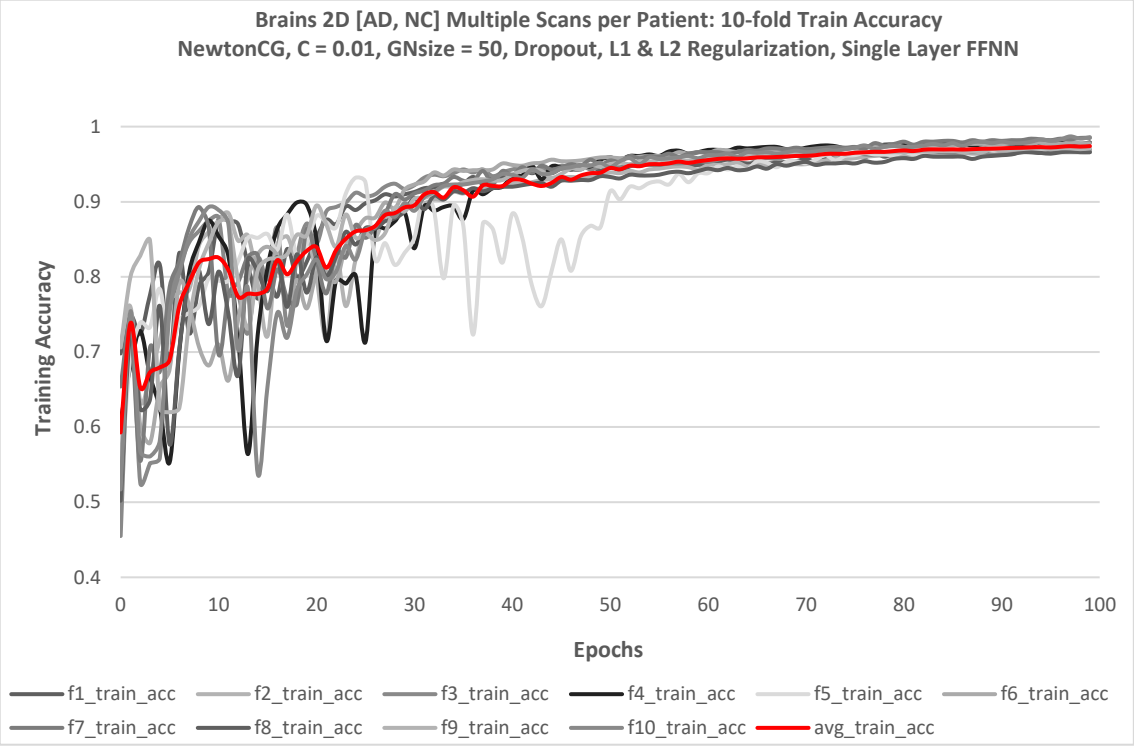


Figure H.165. Caption in the title.

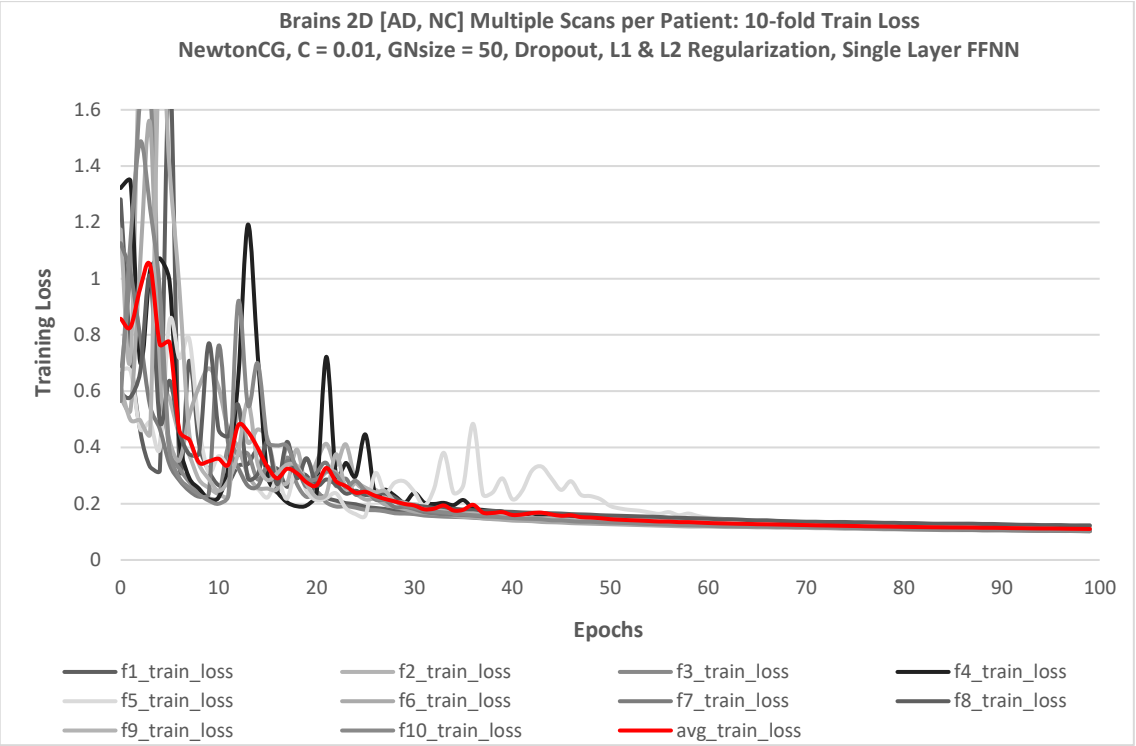


Figure H.166. Caption in the title.

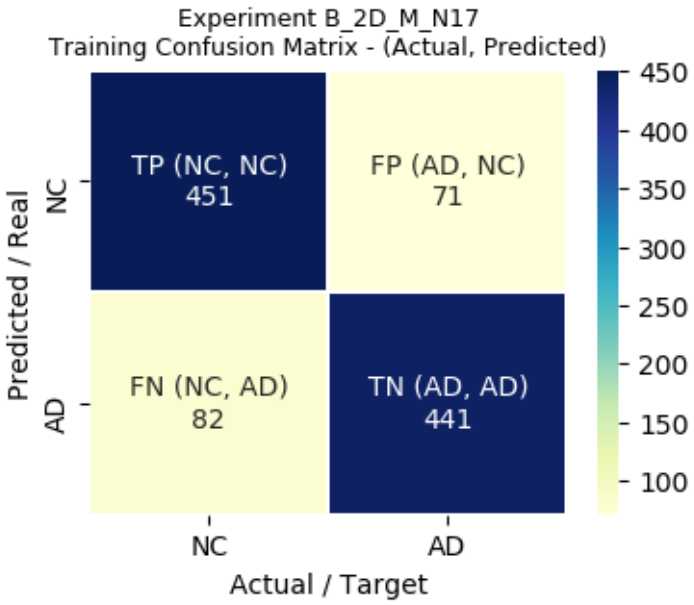


Figure H.167. Caption in the title.

H.2.16.3 Validation

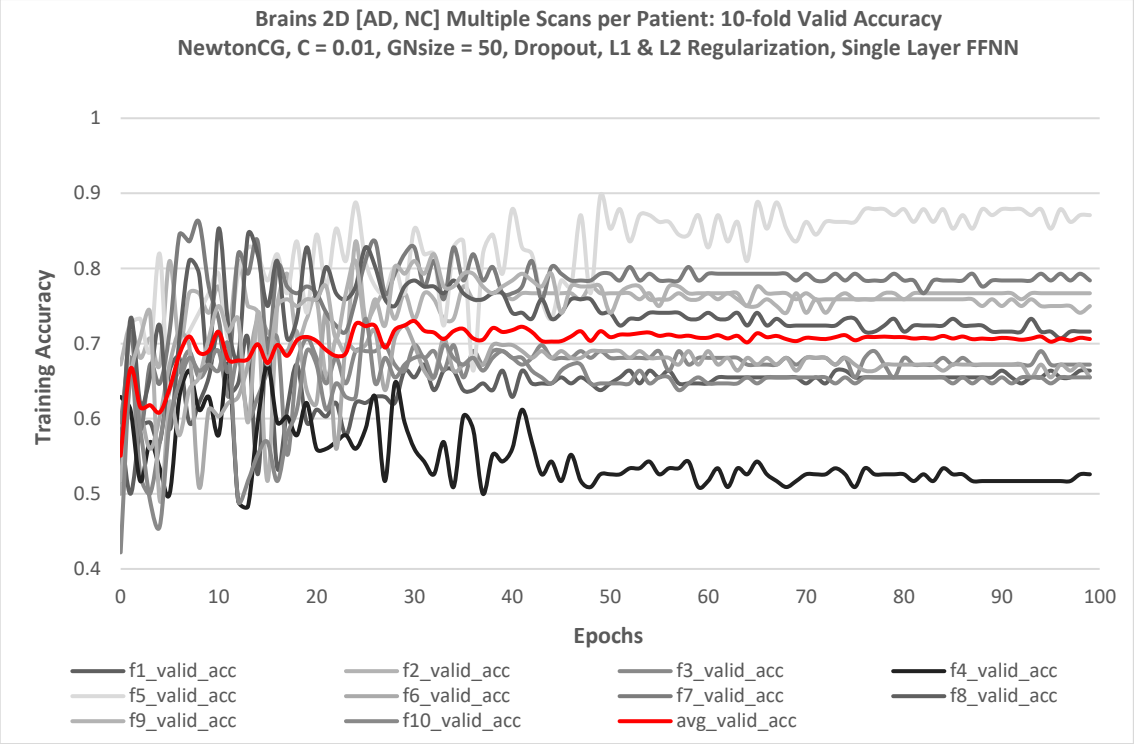


Figure H.168. Caption in the title.

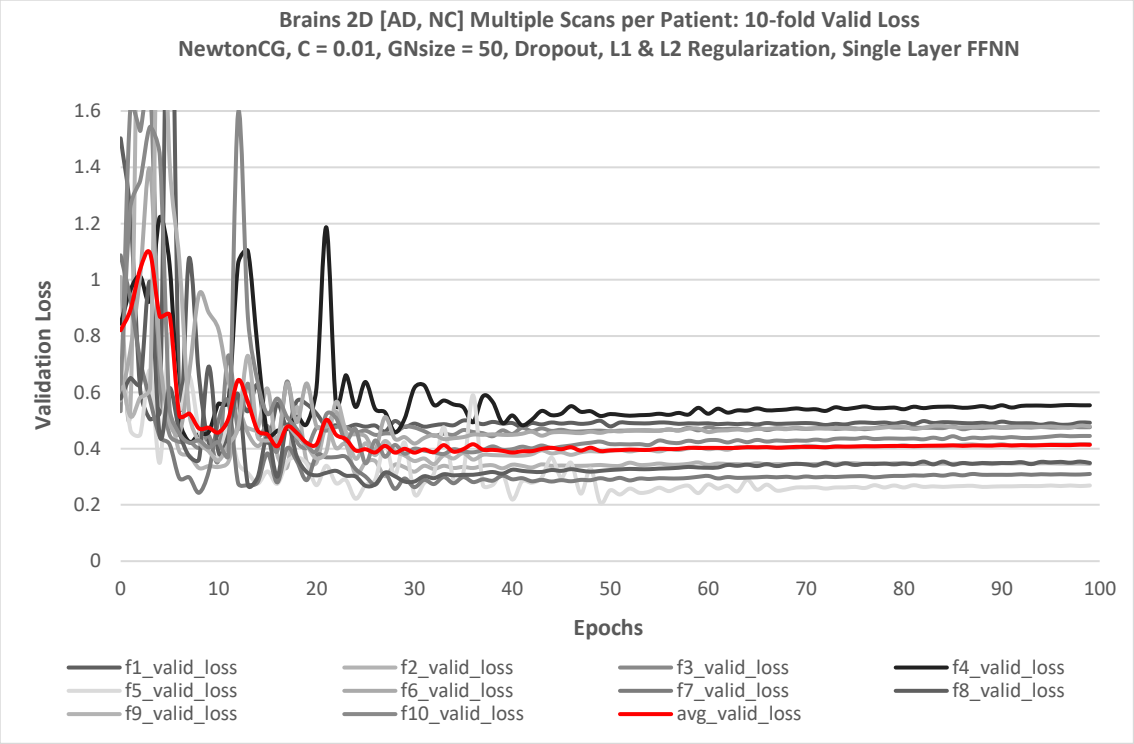


Figure H.169. Caption in the title.

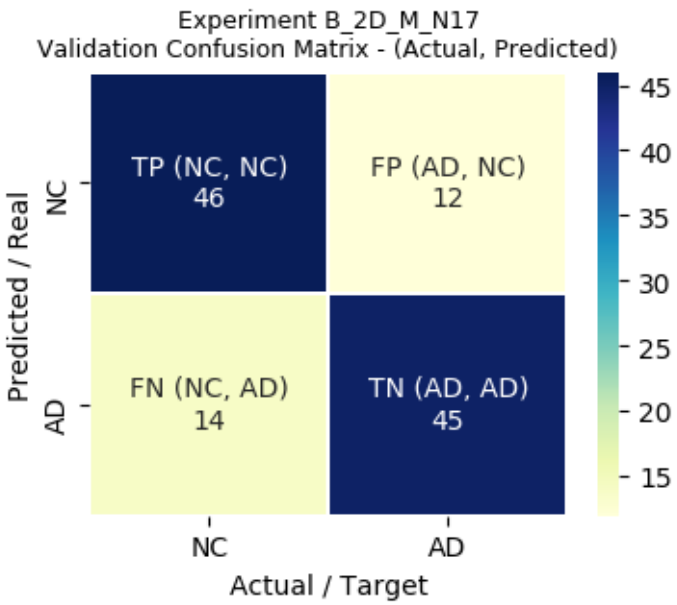


Figure H.170. Caption in the title.

H.2.16.4 Testing

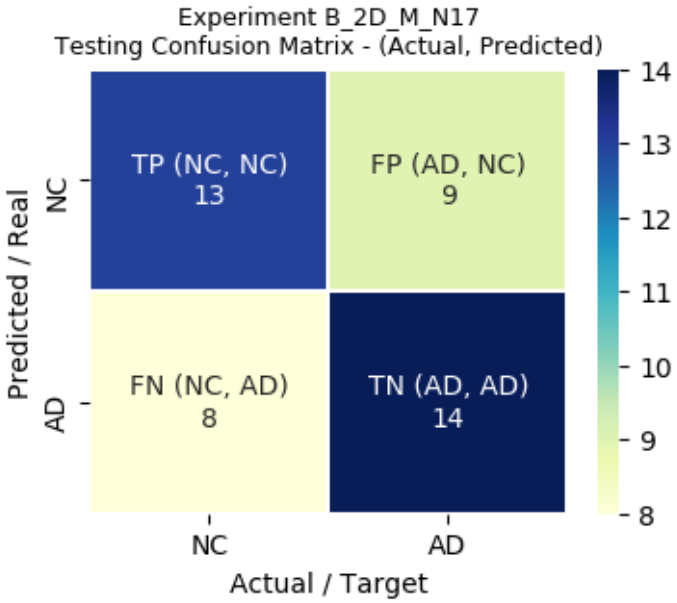


Figure H.171. Caption in the title.

H.2.17 Experiment B_2D_M_N18: C = 0.01, GNsize = 50, Dropout, Batch Normalization

H.2.17.1 General

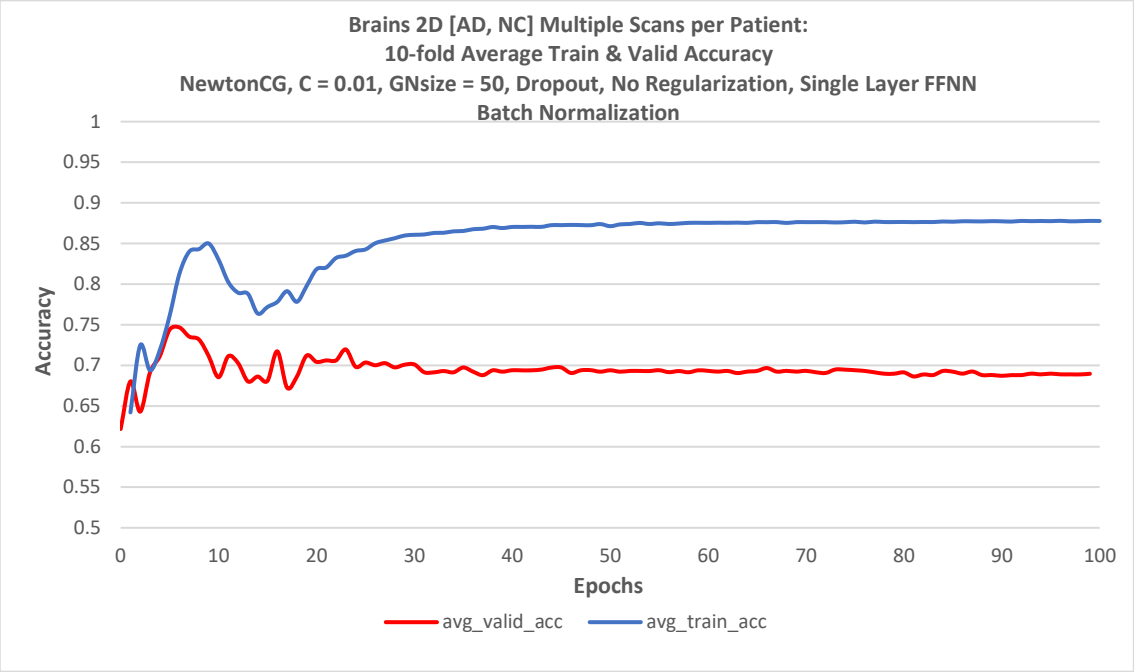


Figure H.172. Caption in the title.

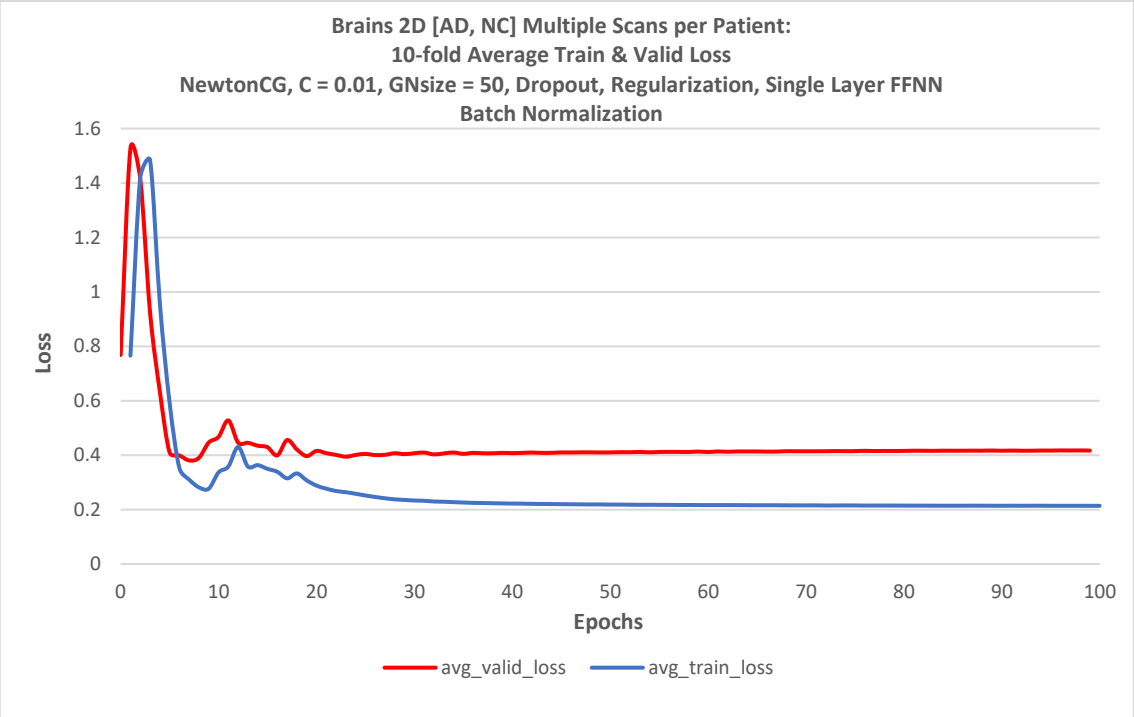


Figure H.173. Caption in the title.

H.2.17.2 Training

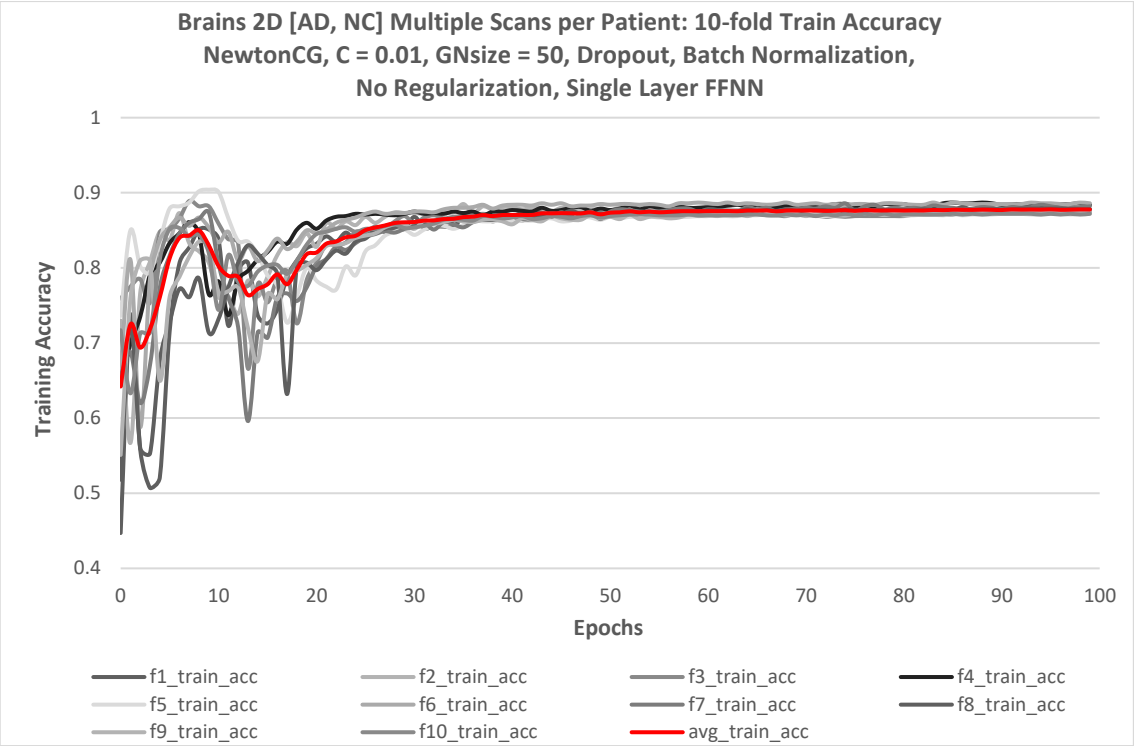


Figure H.174. Caption in the title.

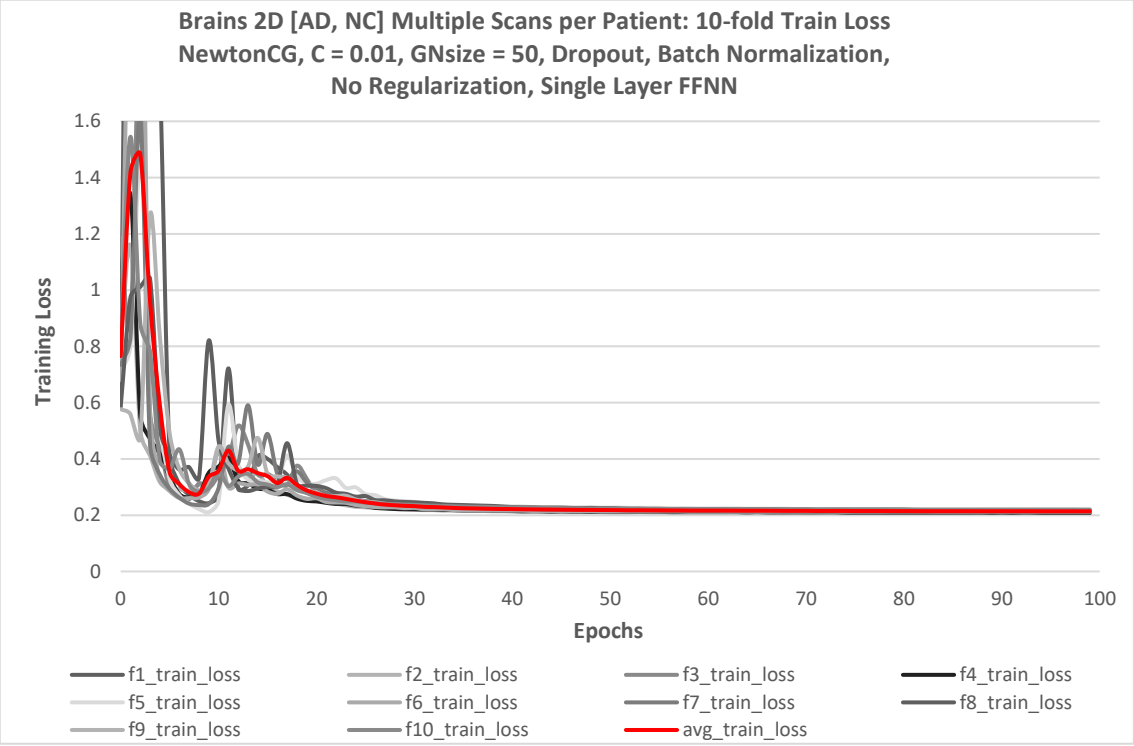


Figure H.175. Caption in the title.

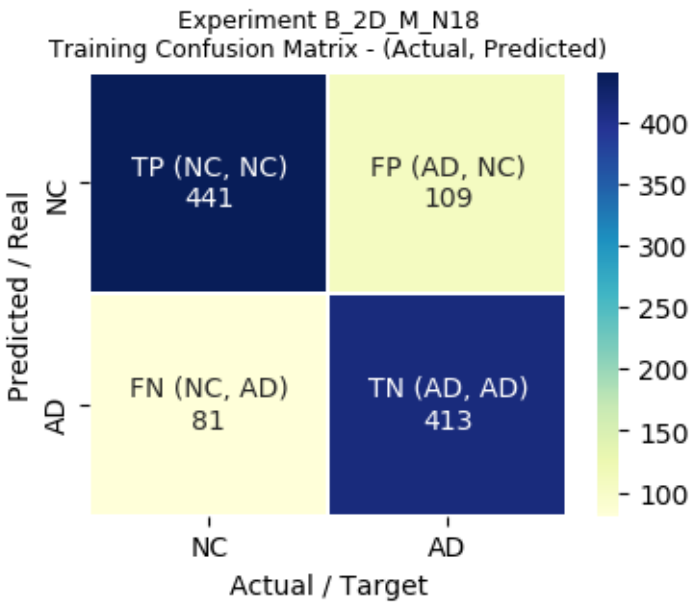


Figure H.176. Caption in the title.

H.2.17.3 Validation

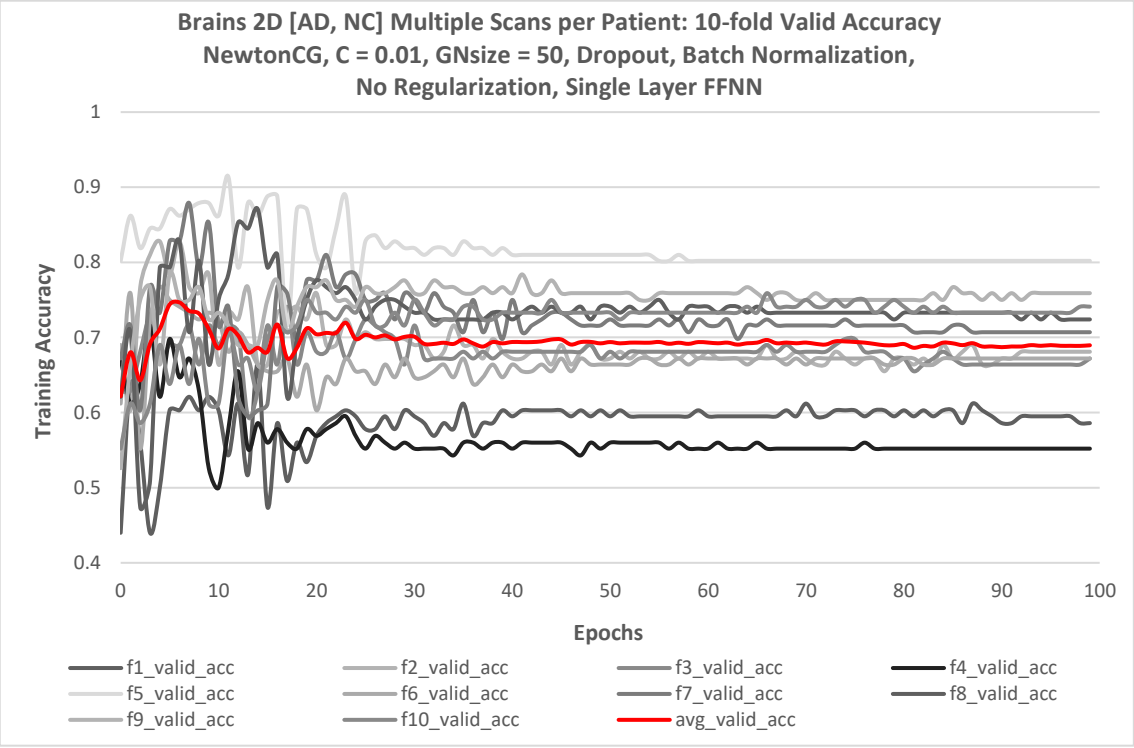


Figure H.177. Caption in the title.

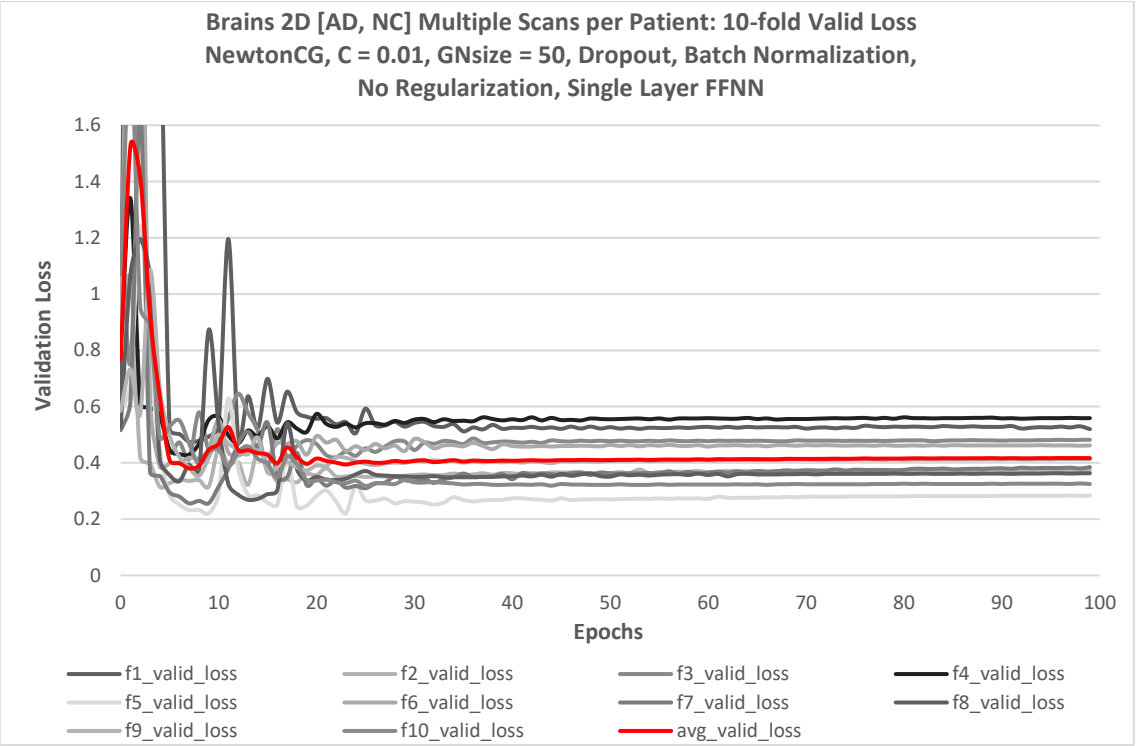


Figure H.178. Caption in the title.

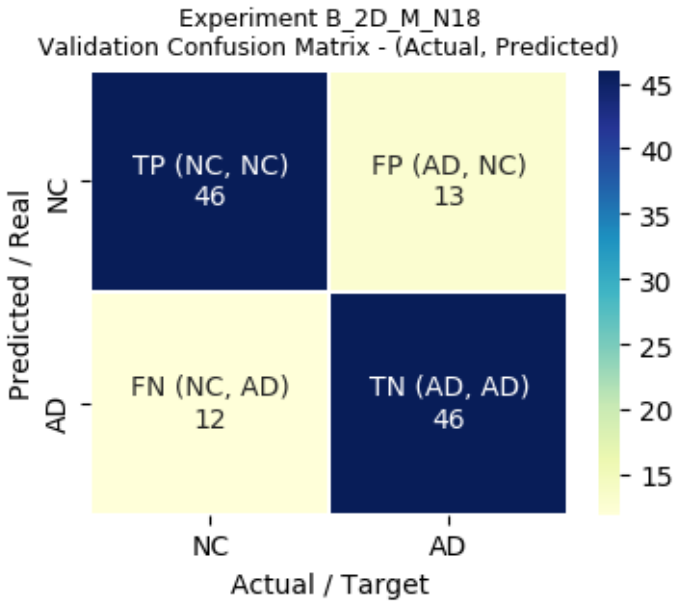


Figure H.179. Caption in the title.

H.2.18 Experiment B_2D_M_N19: C = 0.01, GNsize = 50, Dropout, Filters/Kernels Size = 5 × 5

H.2.18.1 General

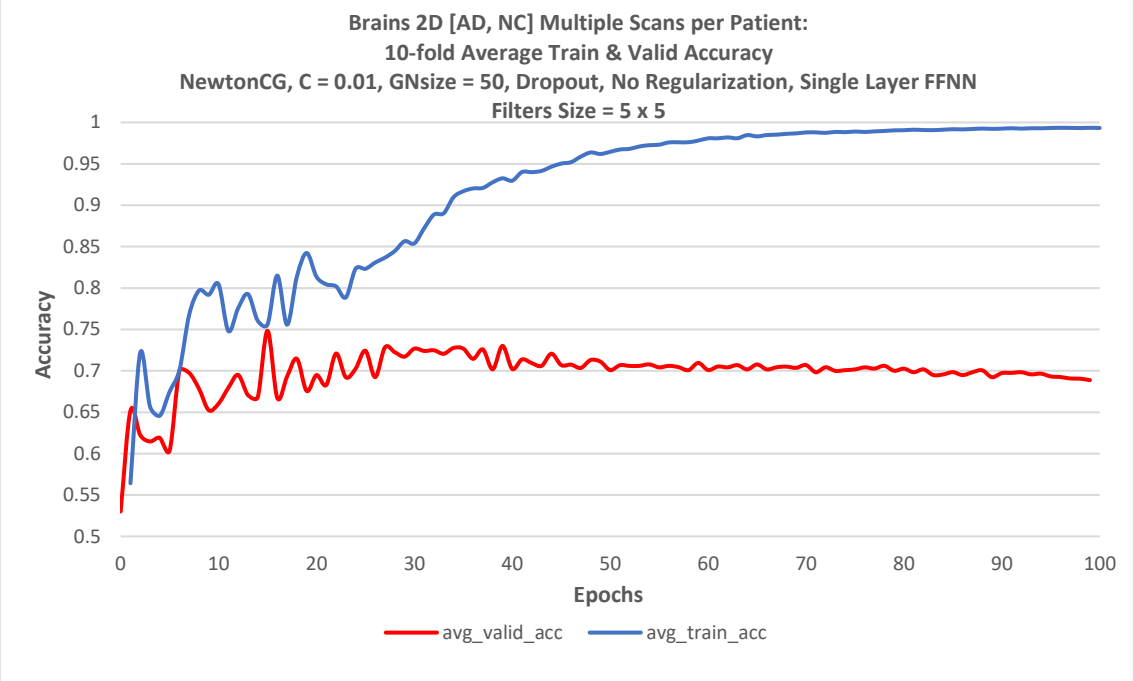


Figure H.180. Caption in the title.

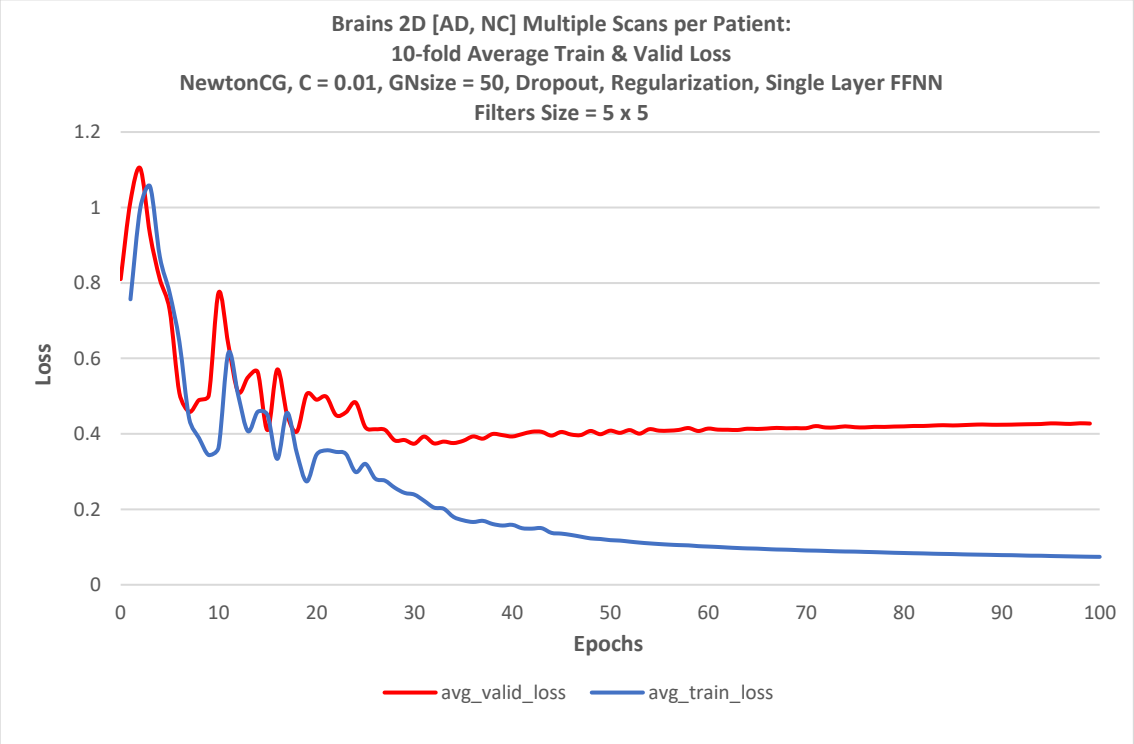


Figure H.181. Caption in the title.

H.2.18.2 Training

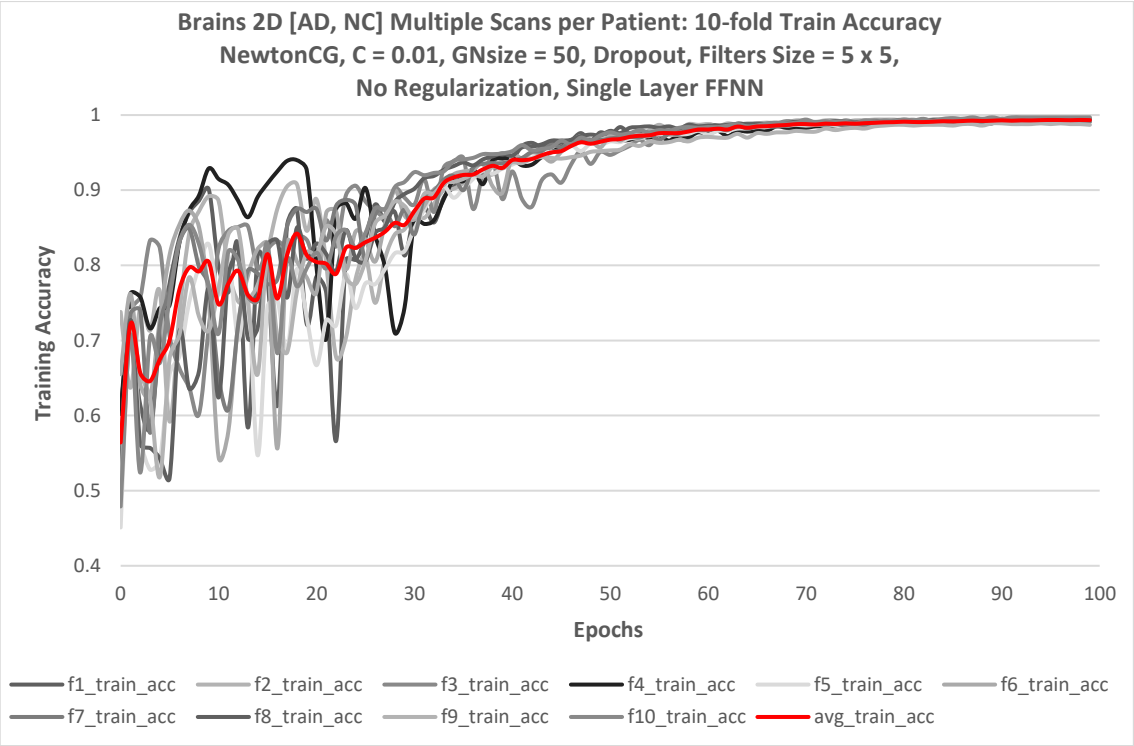


Figure H.182. Caption in the title.

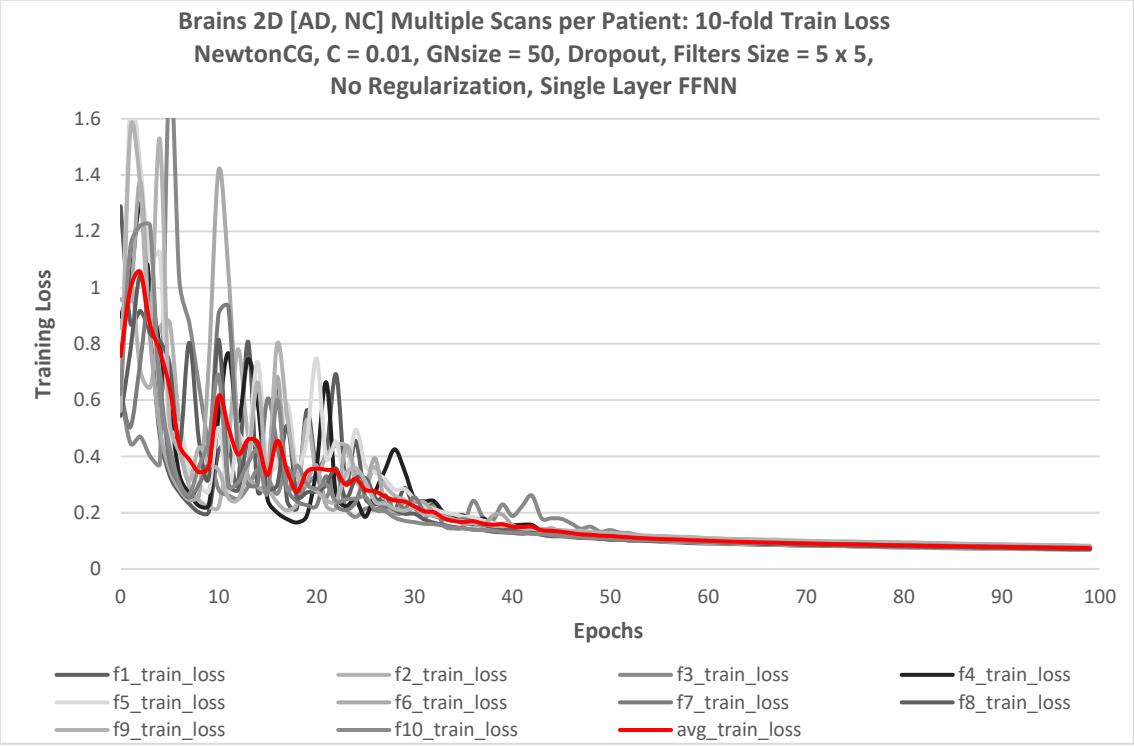


Figure H.183. Caption in the title.

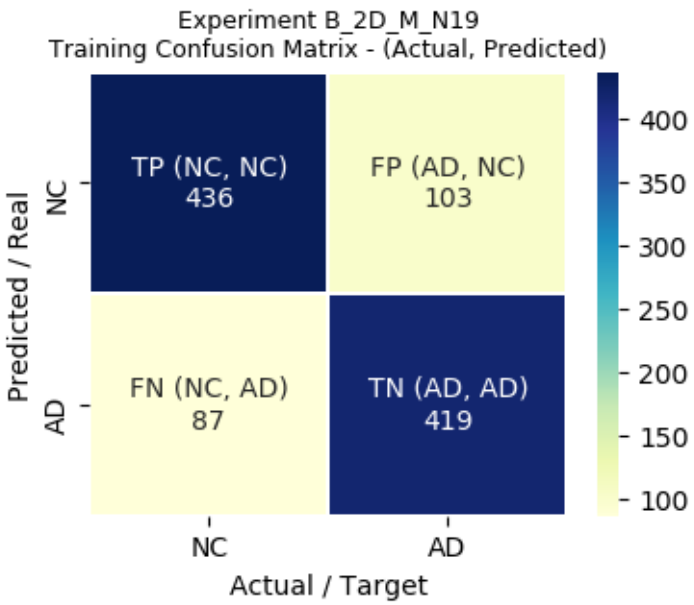


Figure H.184. Caption in the title.

H.2.18.3 Validation

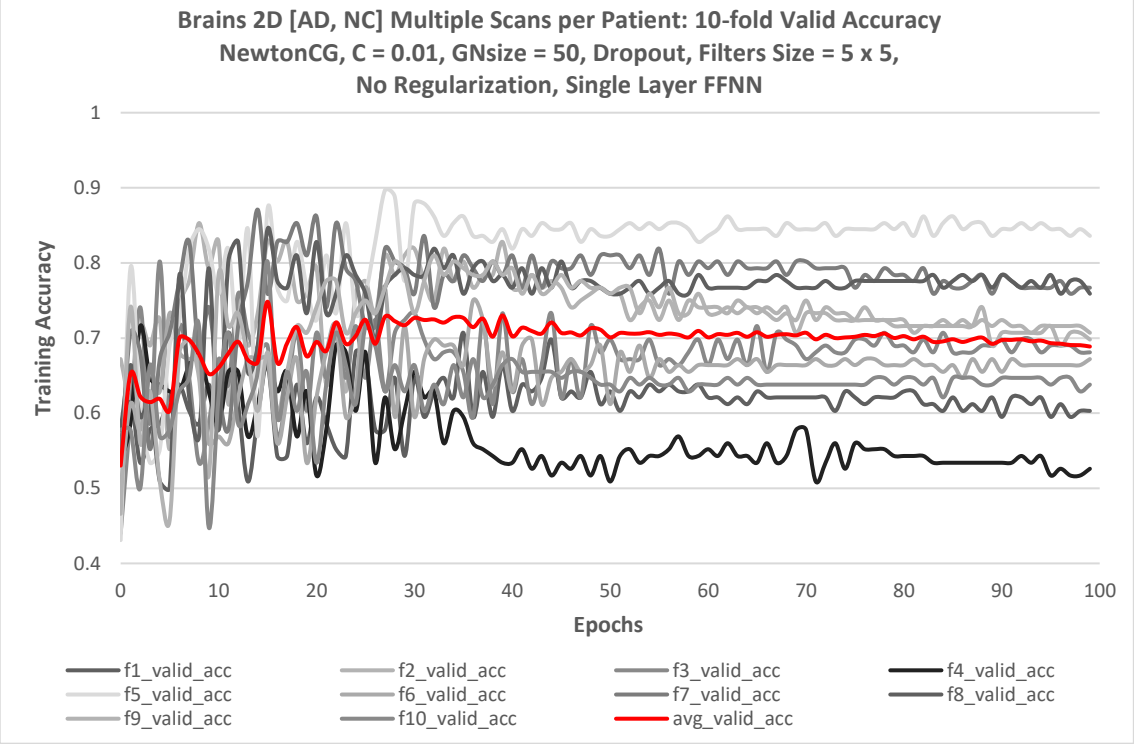


Figure H.185. Caption in the title.

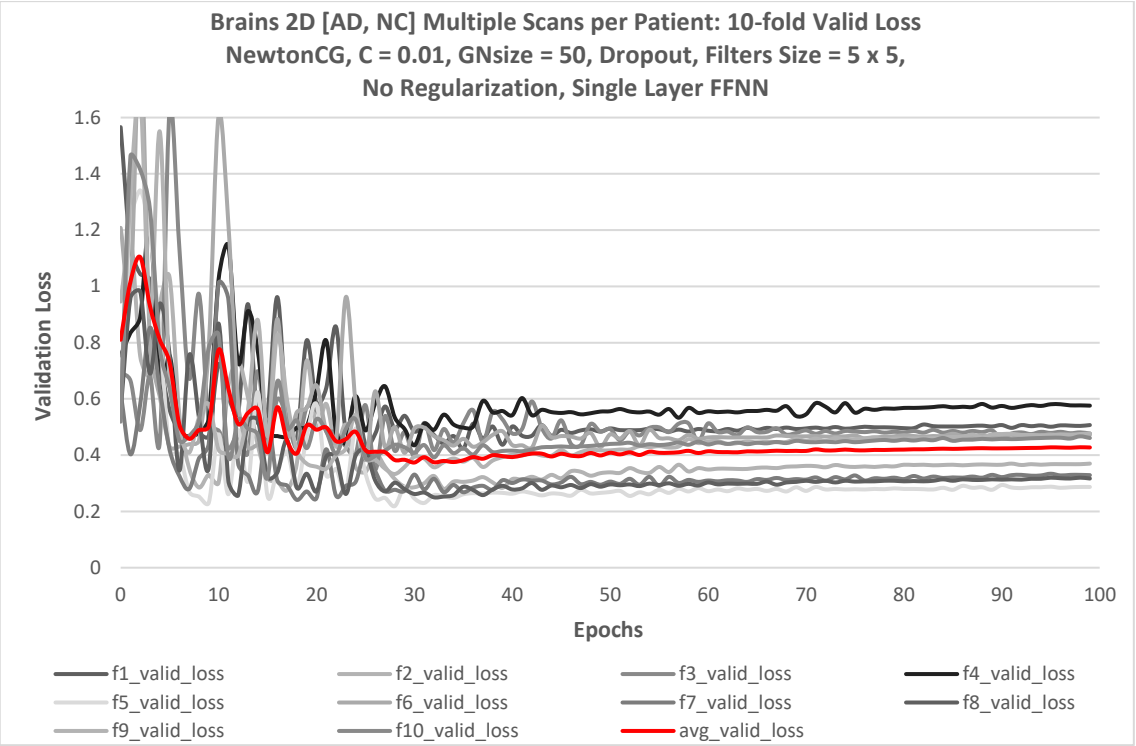


Figure H.186. Caption in the title.

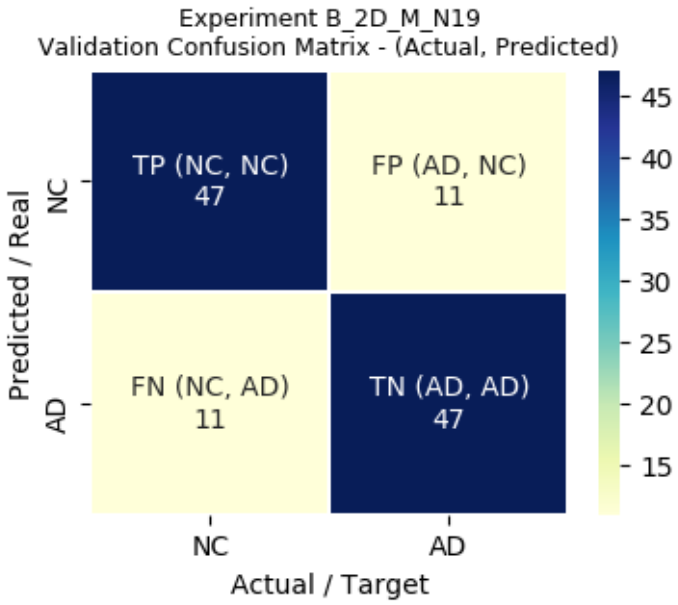


Figure H.187. Caption in the title.

H.2.18.4 Testing

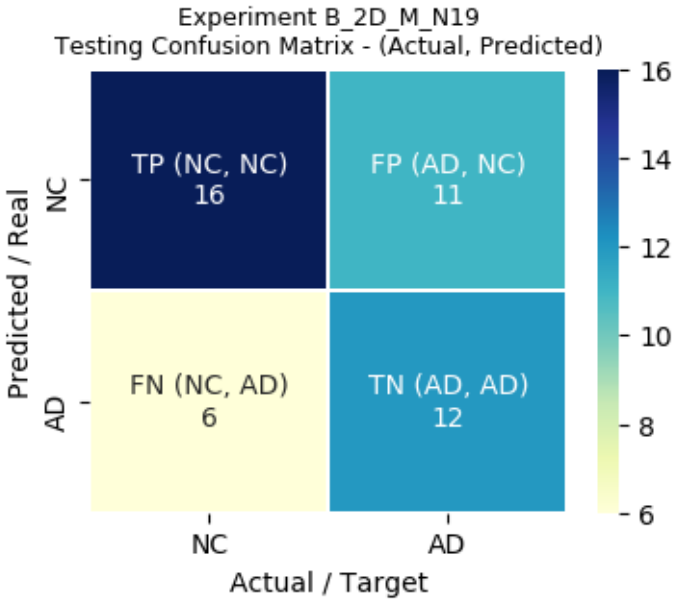


Figure H.188. Caption in the title.

H.2.19 Experiment B_2D_M_N20: C = 0.01, GNsize = 50, Dropout, Filters/Kernels Size = 7 × 7

H.2.19.1 General

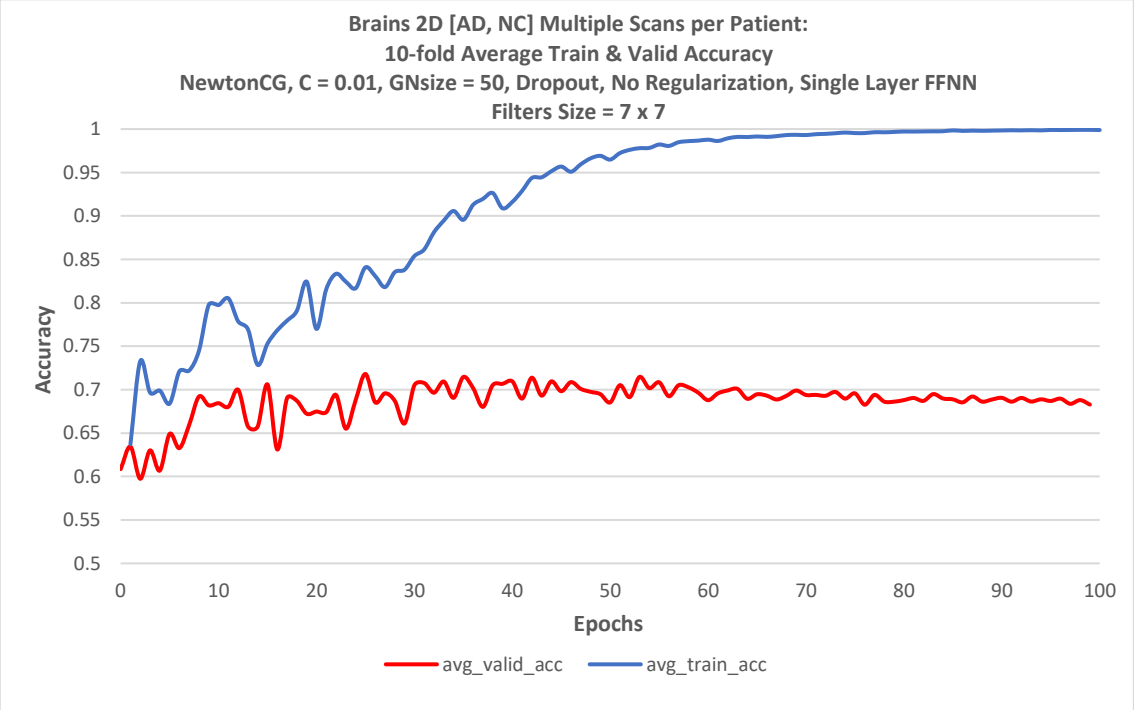


Figure H.189. Caption in the title.

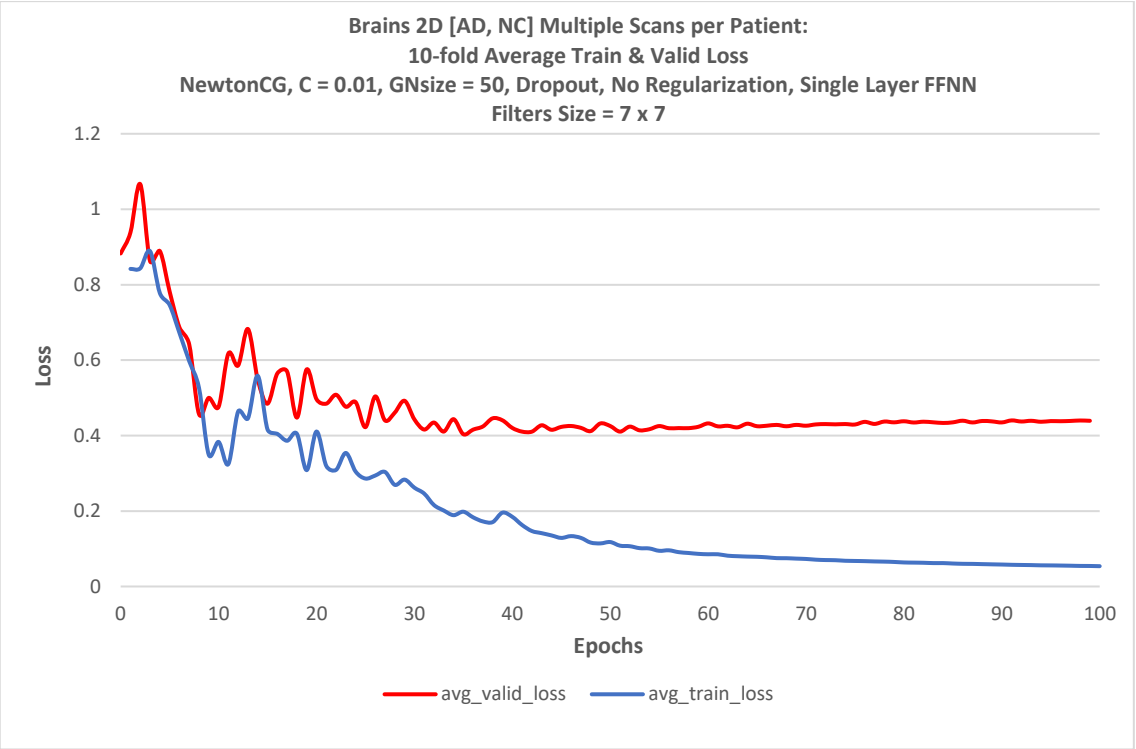


Figure H.190. Caption in the title.

H.2.19.2 Training

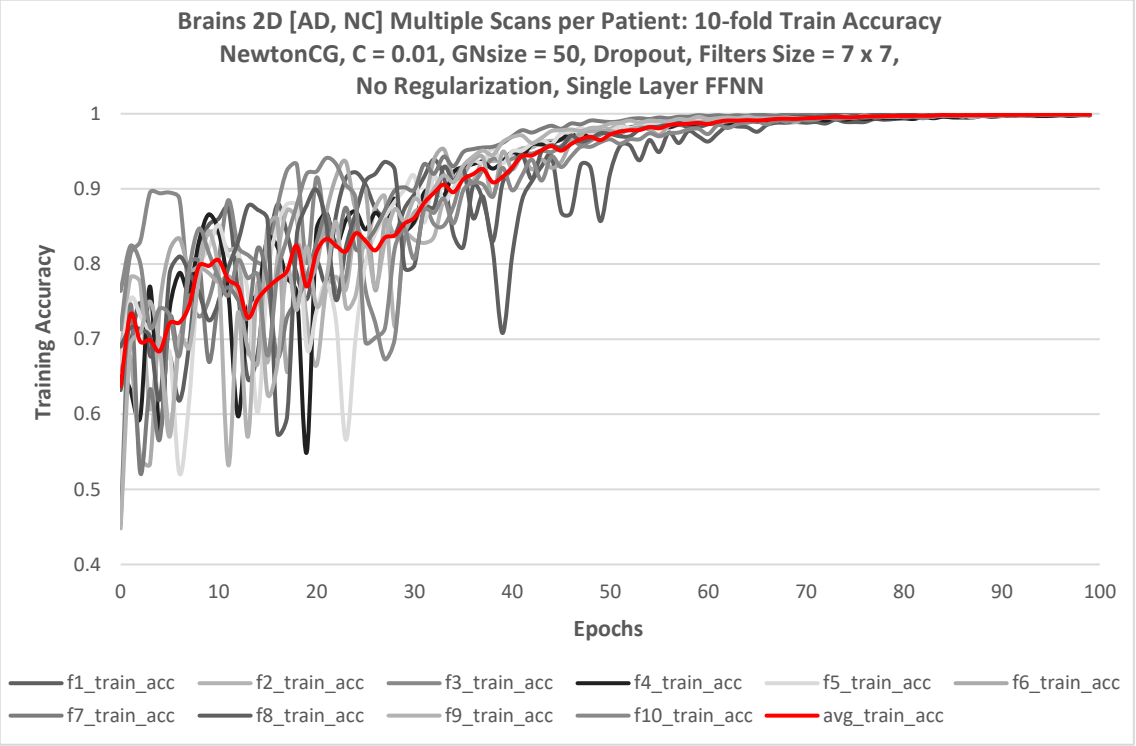


Figure H.191. Caption in the title.

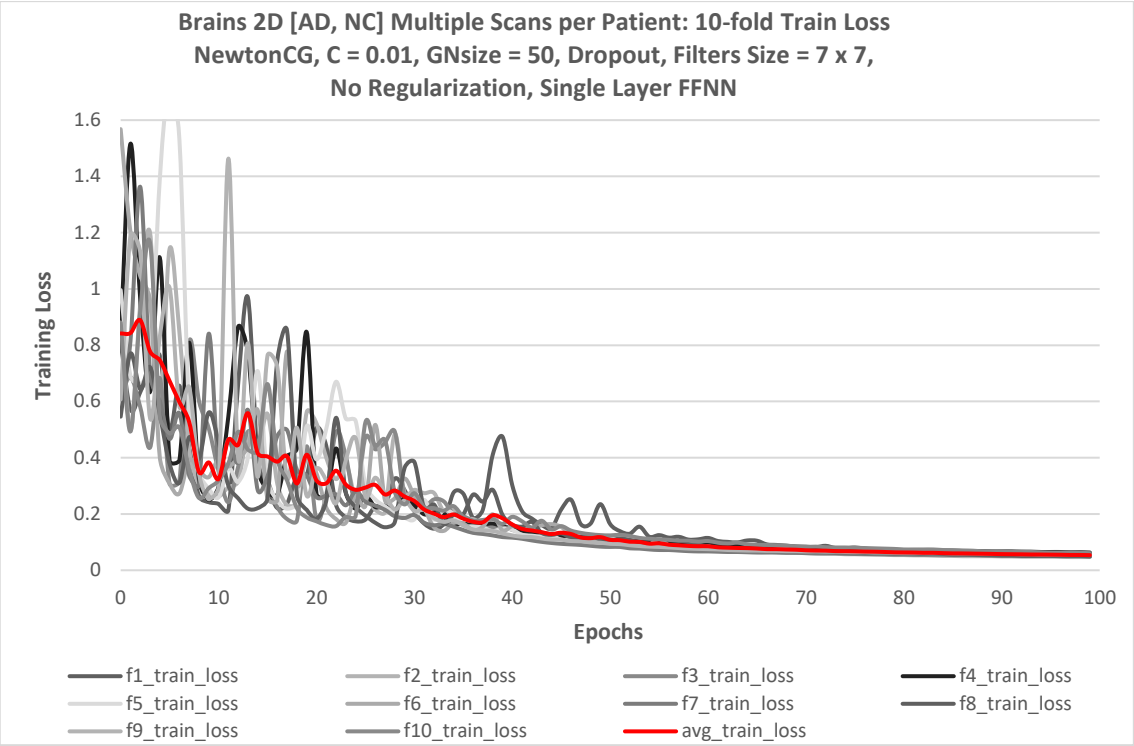


Figure H.192. Caption in the title.

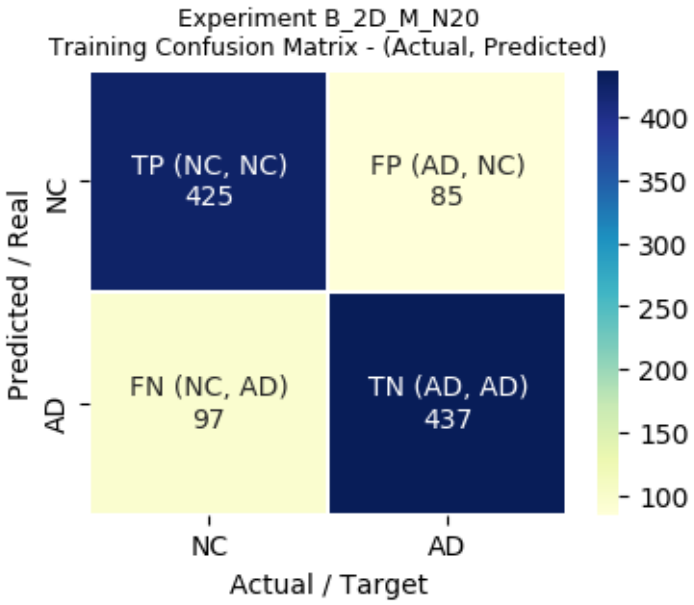


Figure H.193. Caption in the title.

H.2.19.3 Validation

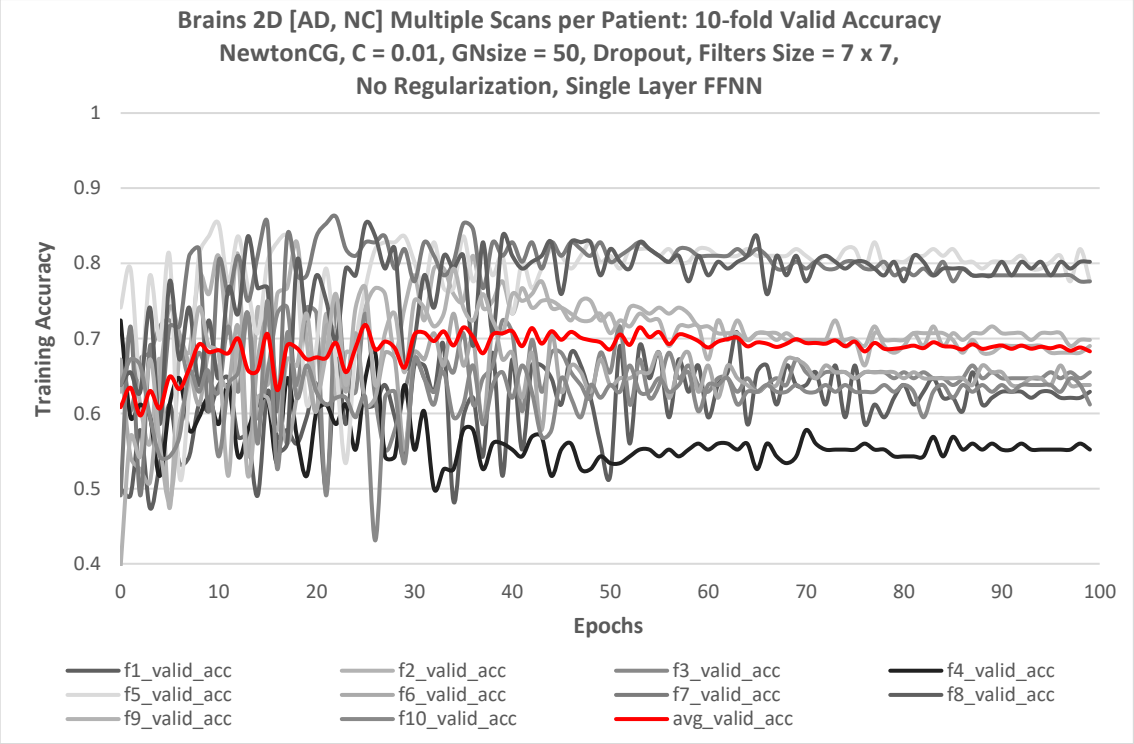


Figure H.194. Caption in the title.

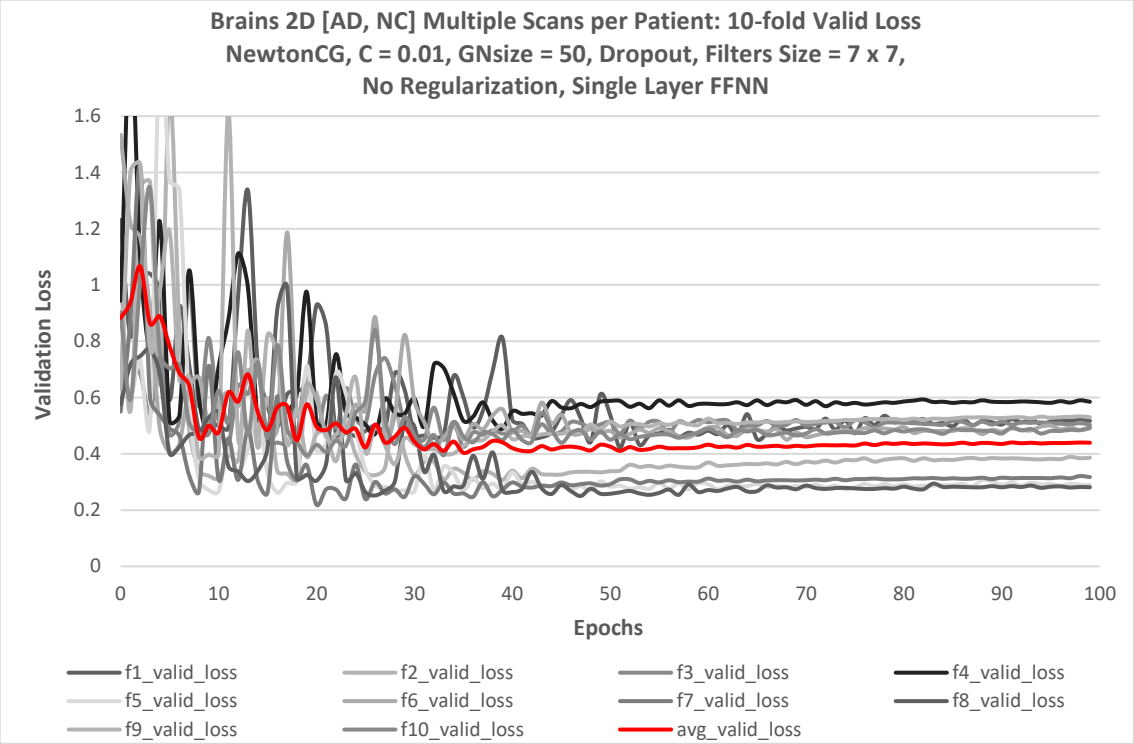


Figure H.195. Caption in the title.

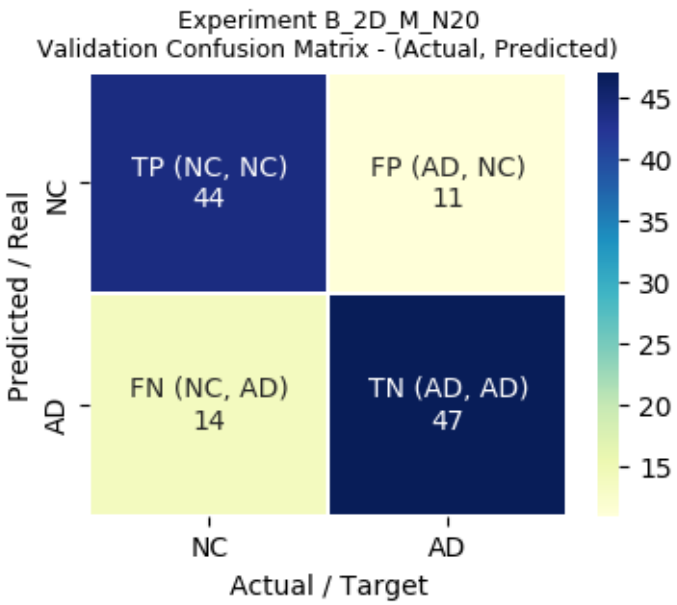


Figure H.196. Caption in the title.

H.2.19.4 Testing

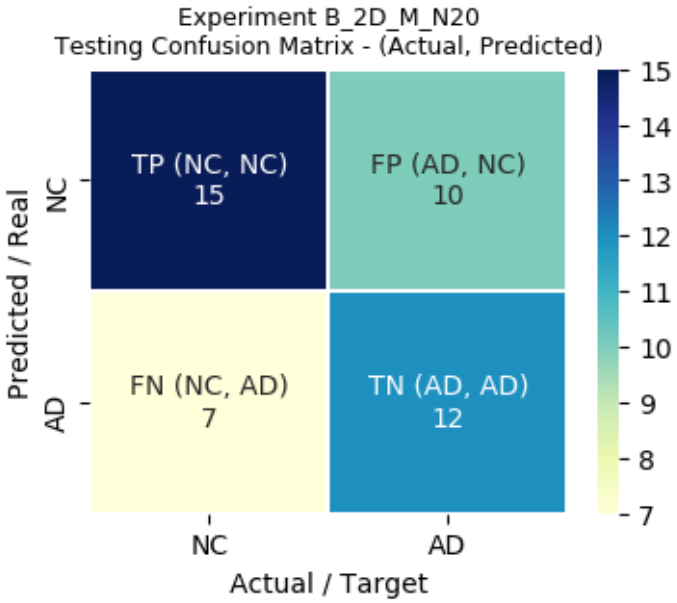


Figure H.197. Caption in the title.

H.2.20 Experiment B_2D_M_N21: C = 0.01, GNsize = 50, Spatial Dropout

H.2.20.1 General

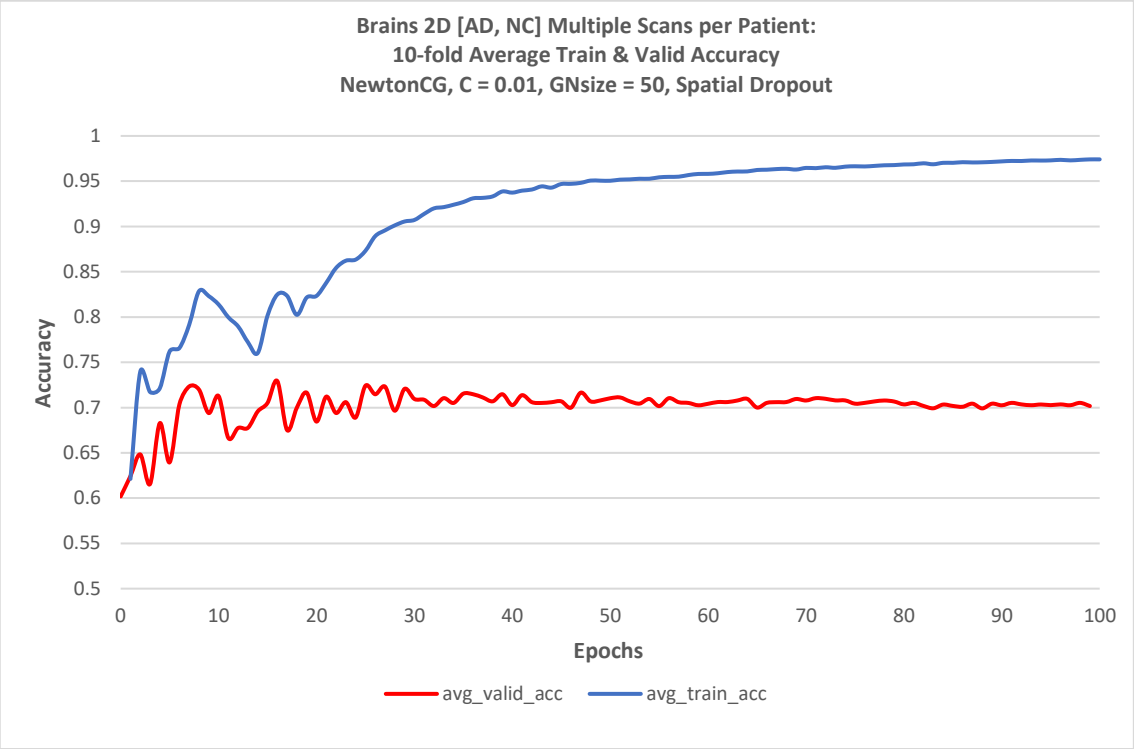


Figure H.198. Caption in the title.

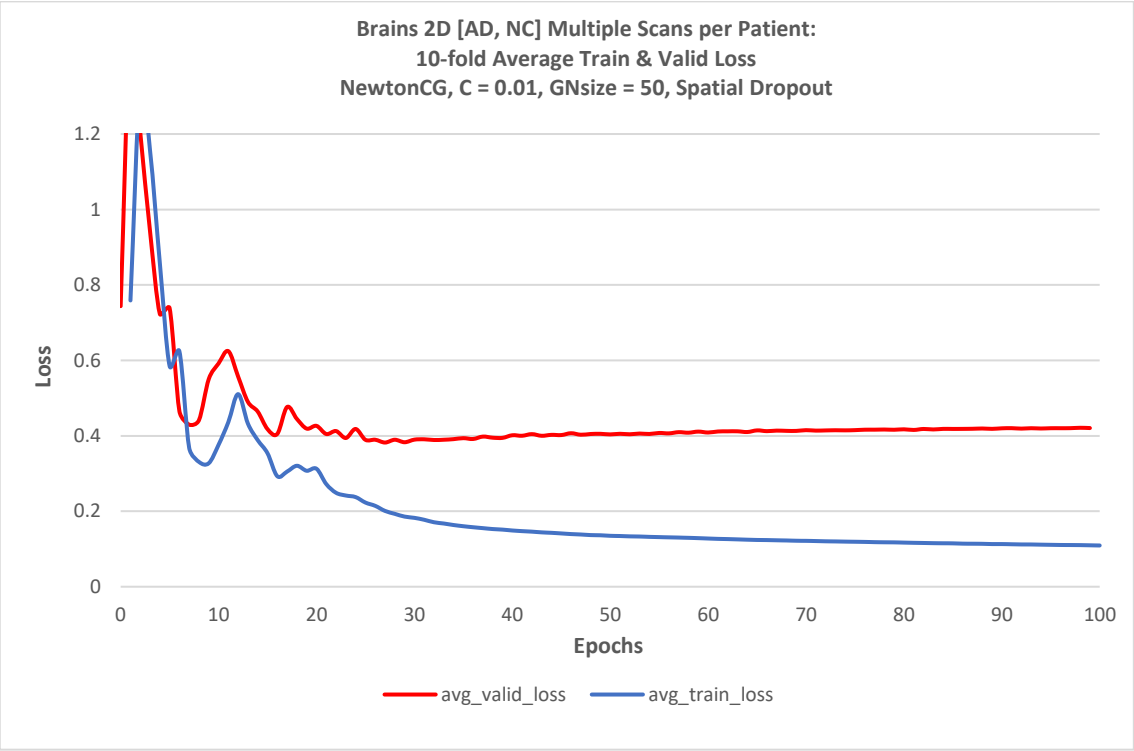


Figure H.199. Caption in the title.

H.2.20.2 Training

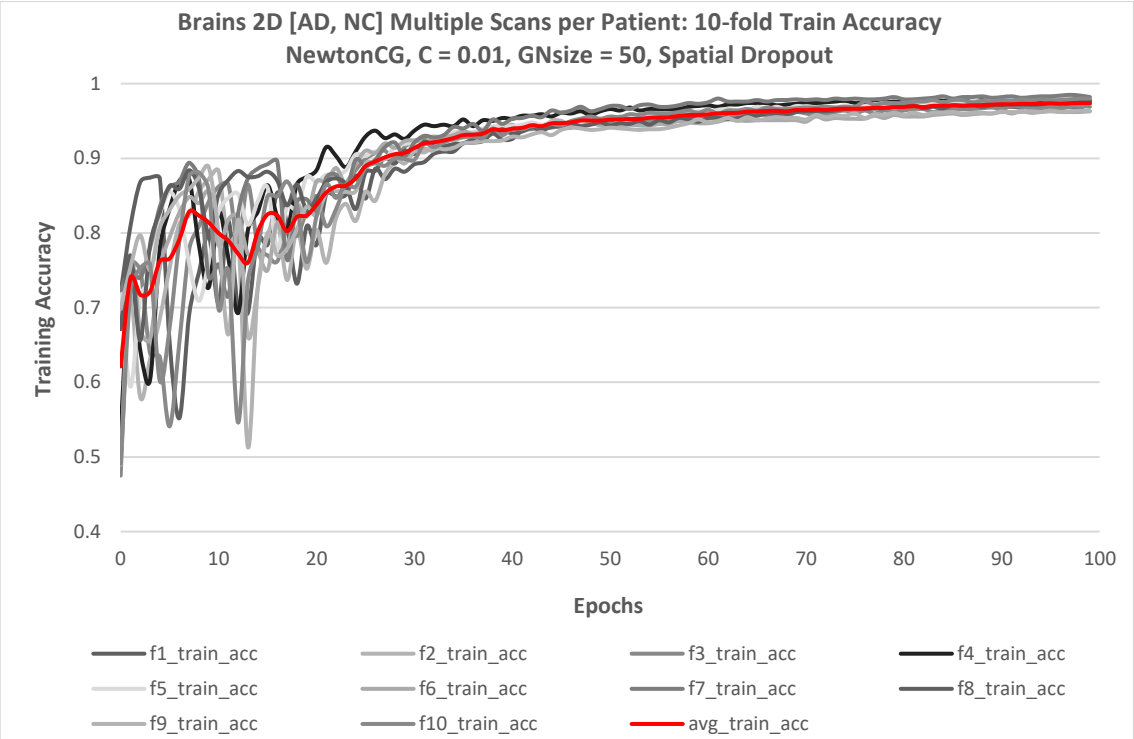


Figure H.200. Caption in the title.

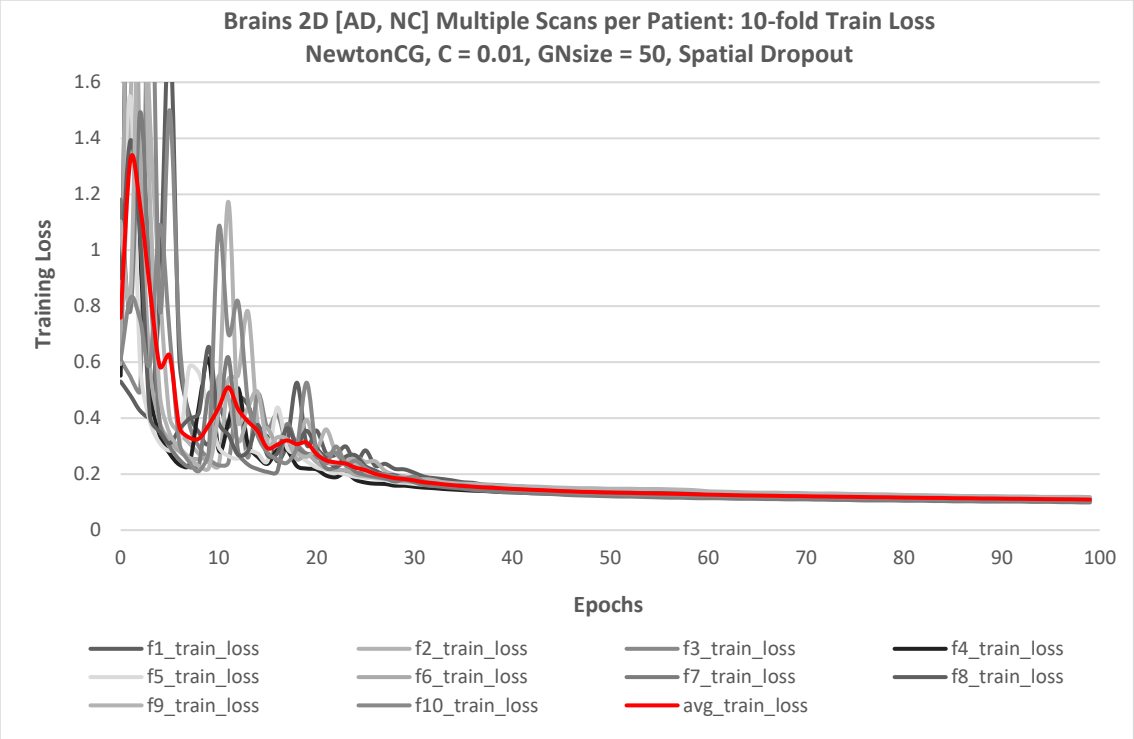


Figure H.201. Caption in the title.

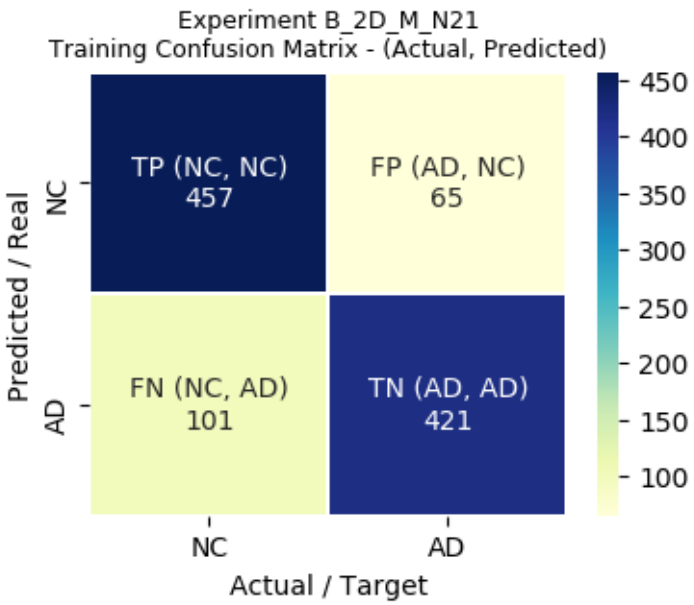


Figure H.202. Caption in the title.

H.2.20.3 Validation

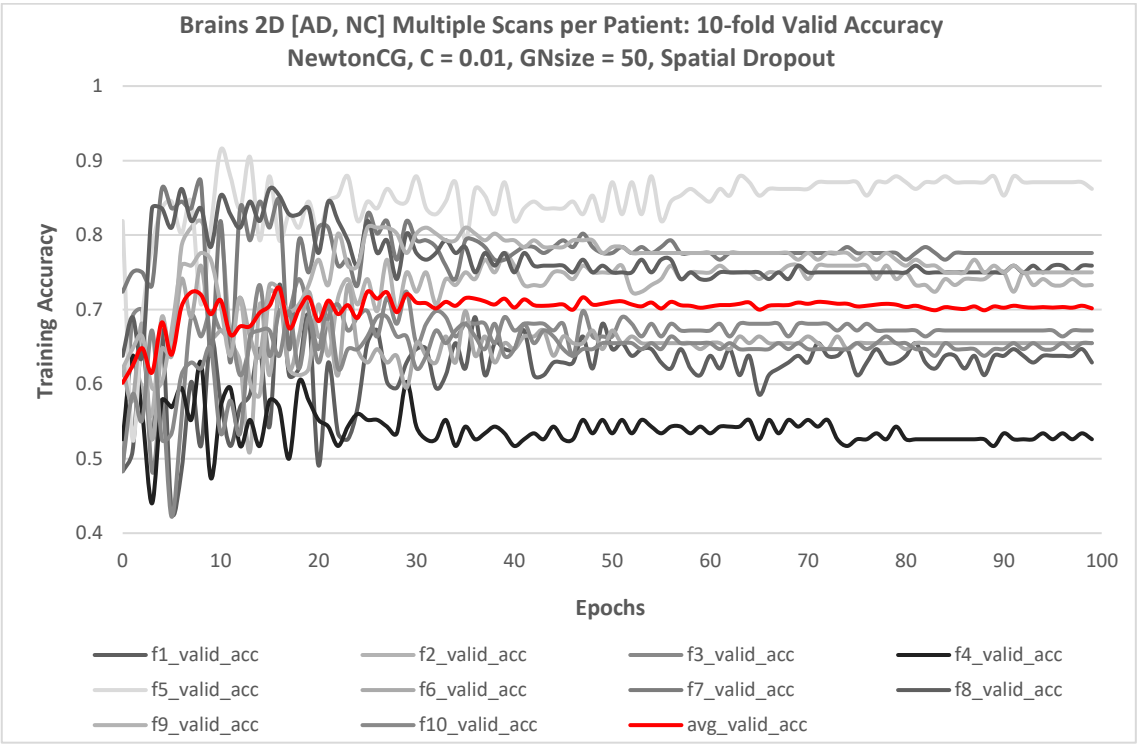


Figure H.203. Caption in the title.

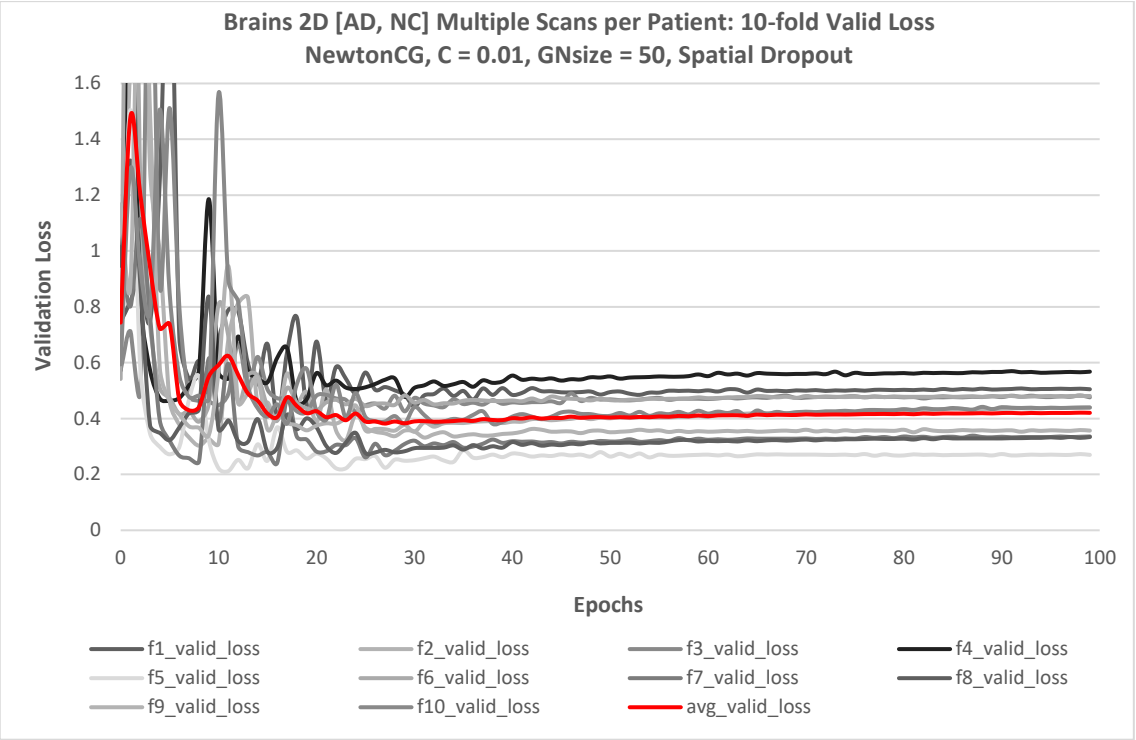


Figure H.204. Caption in the title.

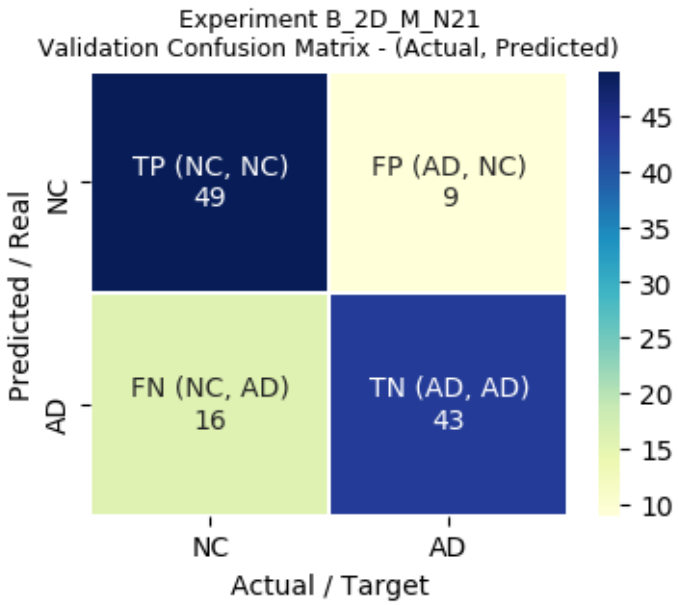


Figure H.205. Caption in the title.

H.2.20.4 Testing

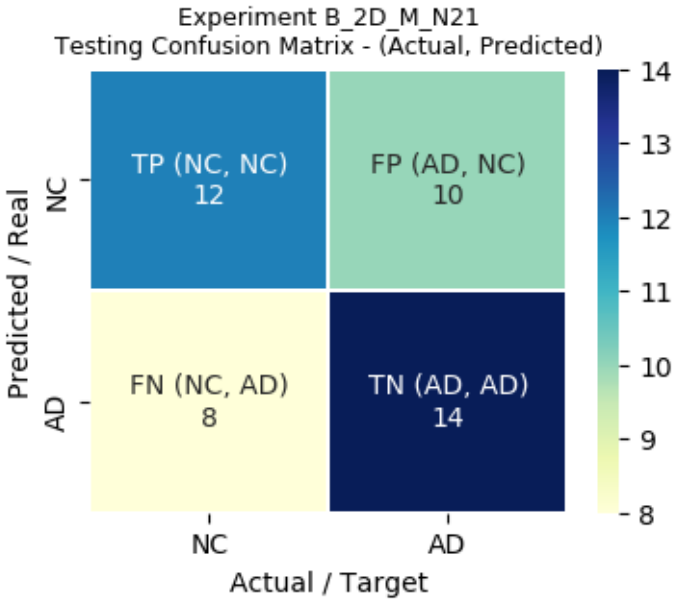


Figure H.206. Caption in the title.

H.2.21 Experiment B_2D_M_N22: C = 0.01, GNsize = 50, Spatial Dropout, Dropout

H.2.21.1 General

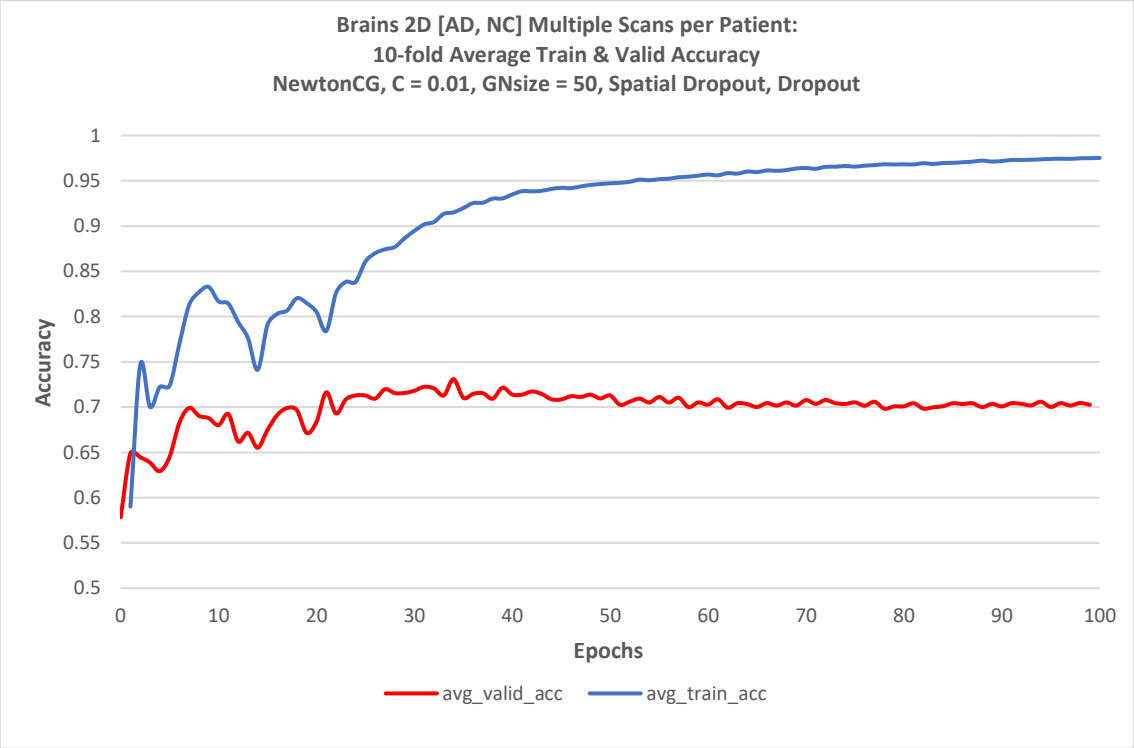


Figure H.207. Caption in the title.

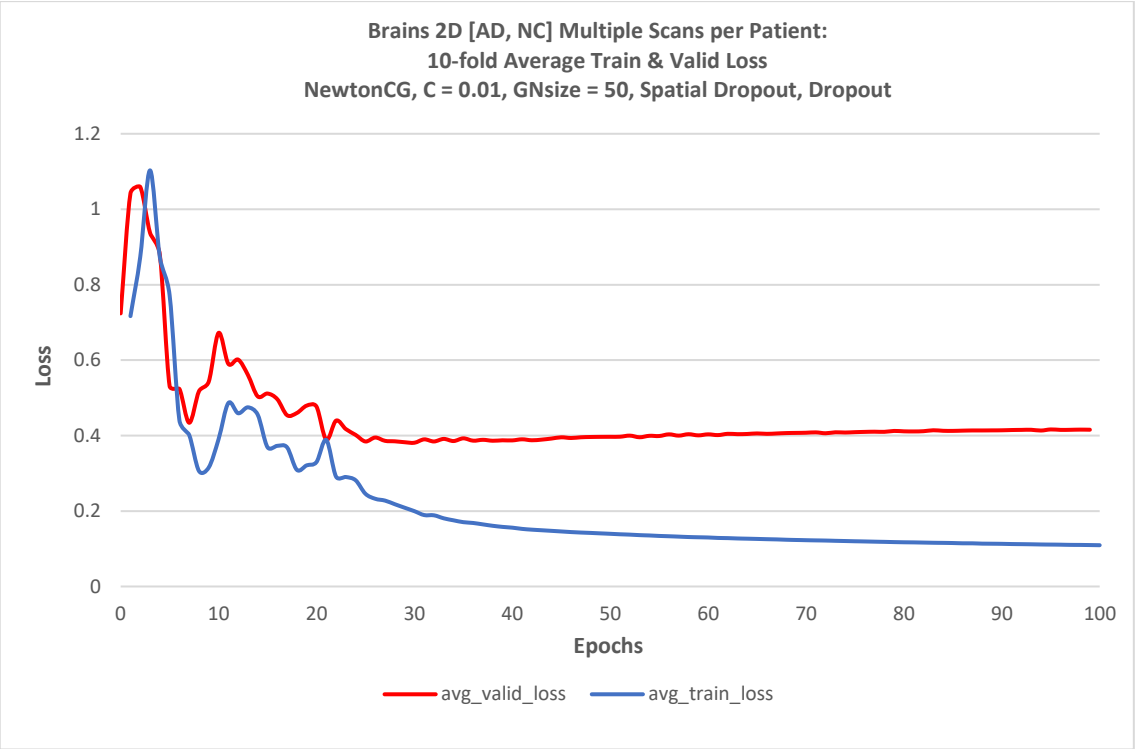


Figure H.208. Caption in the title.

H.2.21.2 Training

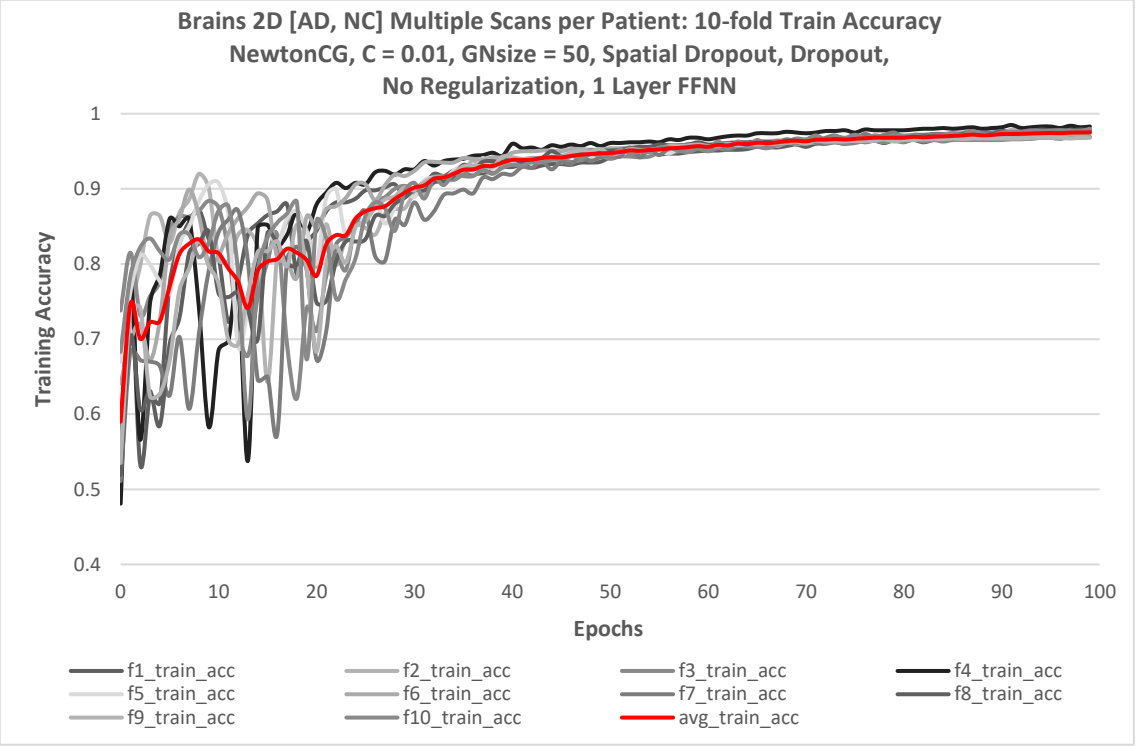


Figure H.209. Caption in the title.

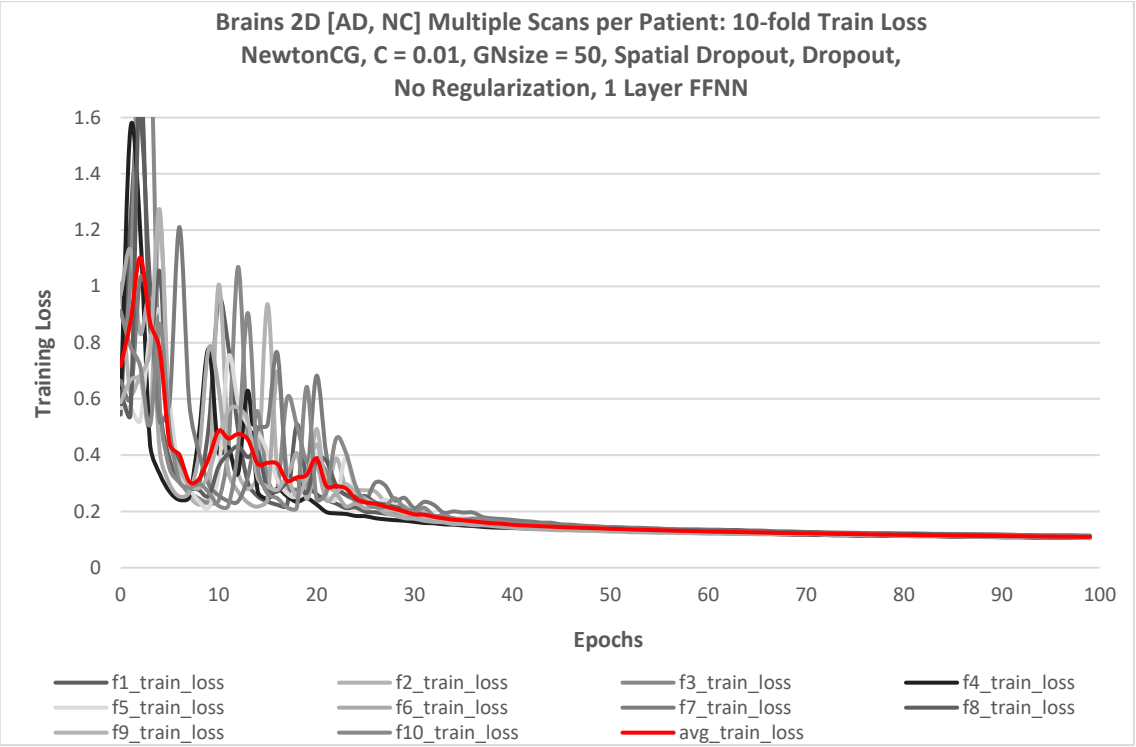


Figure H.210. Caption in the title.

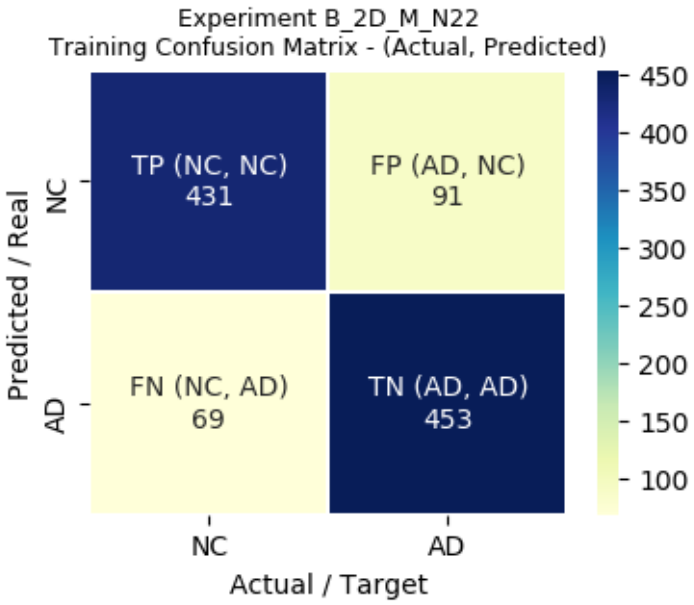


Figure H.211. Caption in the title.

H.2.21.3 Validation

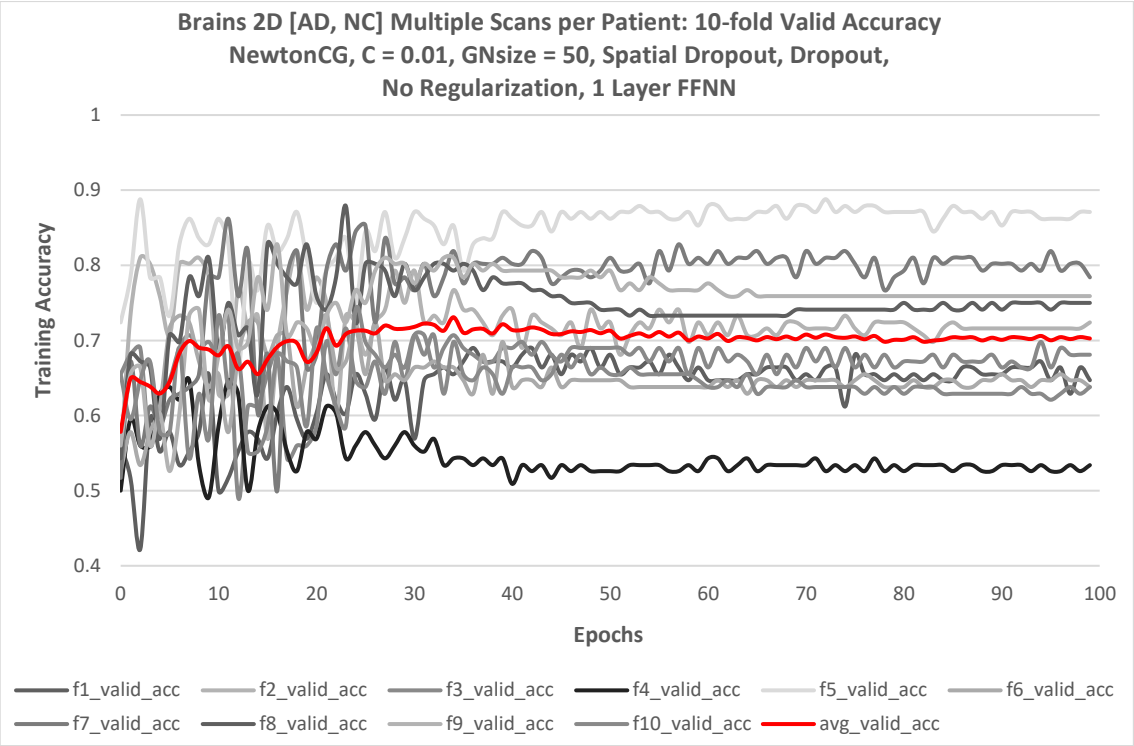


Figure H.212. Caption in the title.

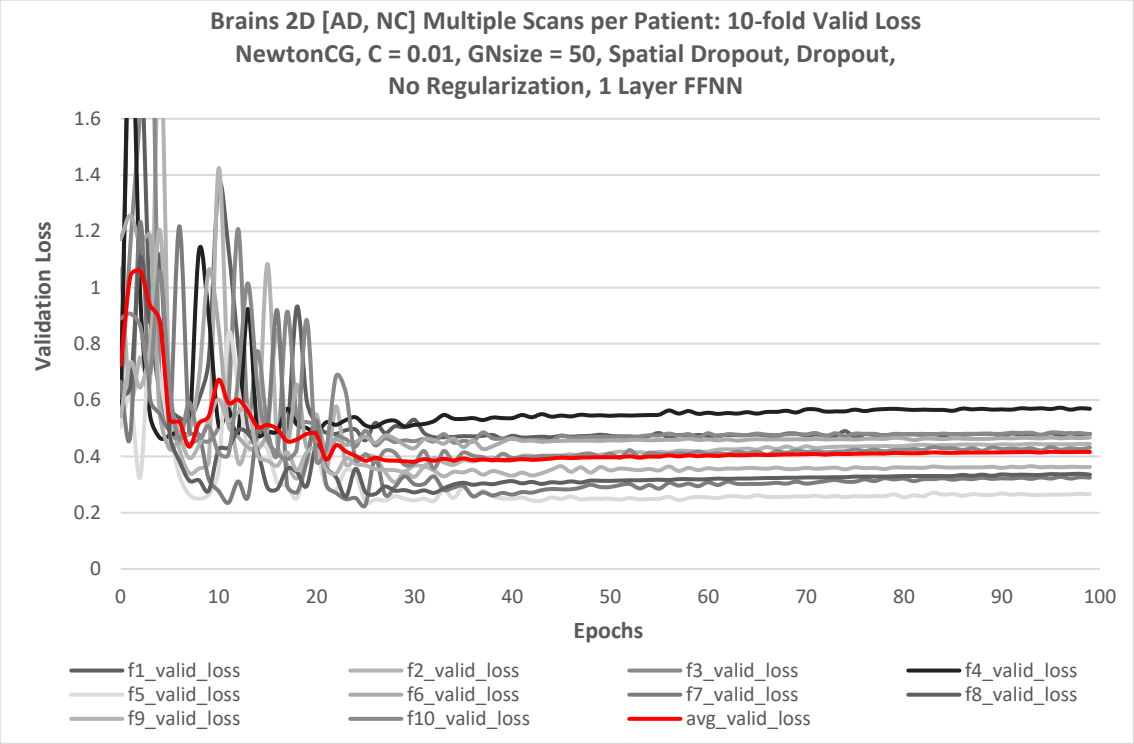


Figure H.213. Caption in the title.

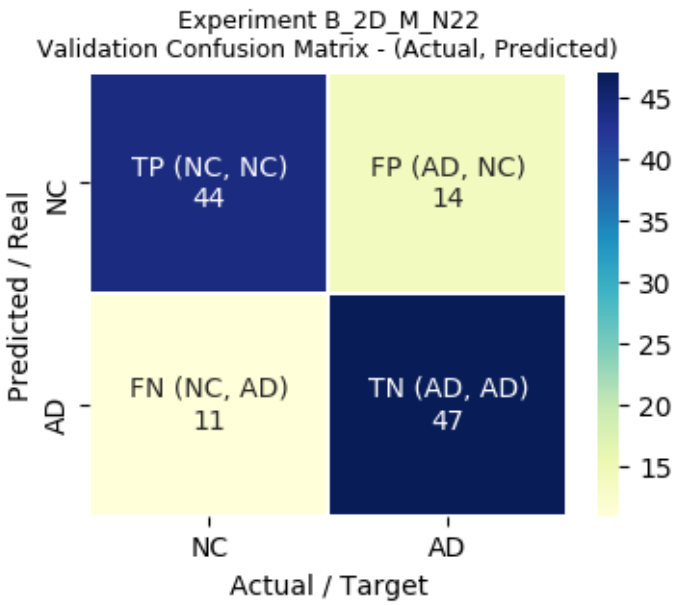


Figure H.214. Caption in the title.

H.2.21.4 Testing

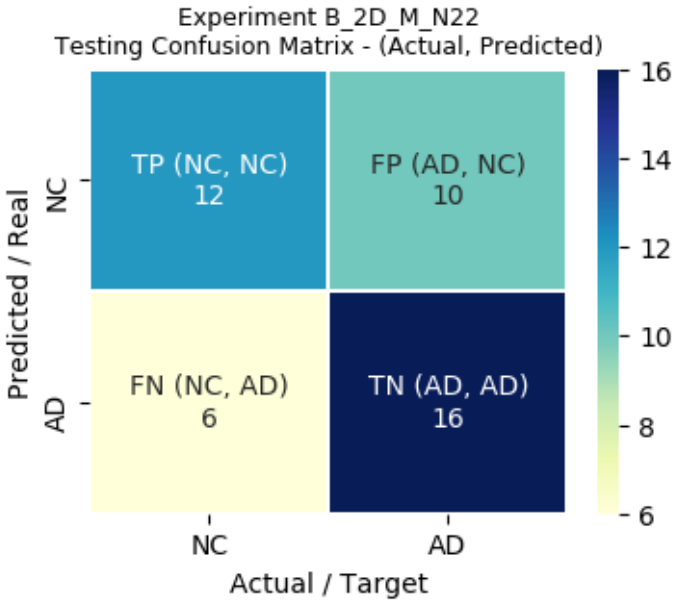


Figure H.215. Caption in the title.

H.2.22 Experiment B_2D_M_N23: C = 0.01, GNsize = 50, Spatial Dropout, L1 & L2 Regularization

H.2.22.1 General

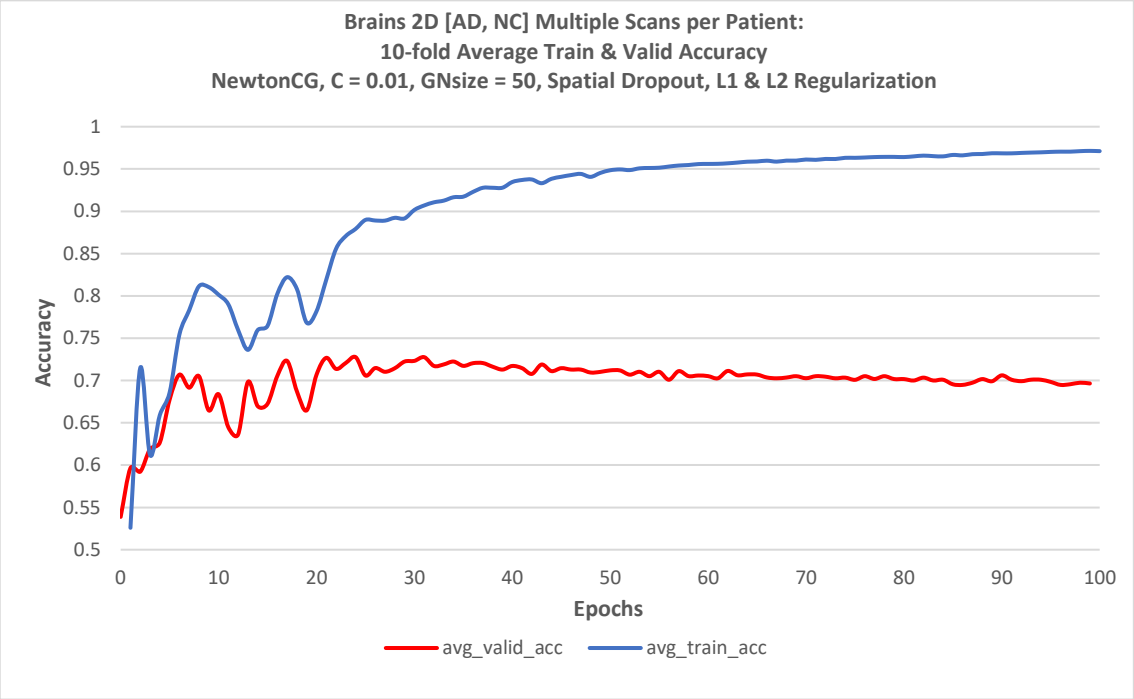


Figure H.216. Caption in the title.

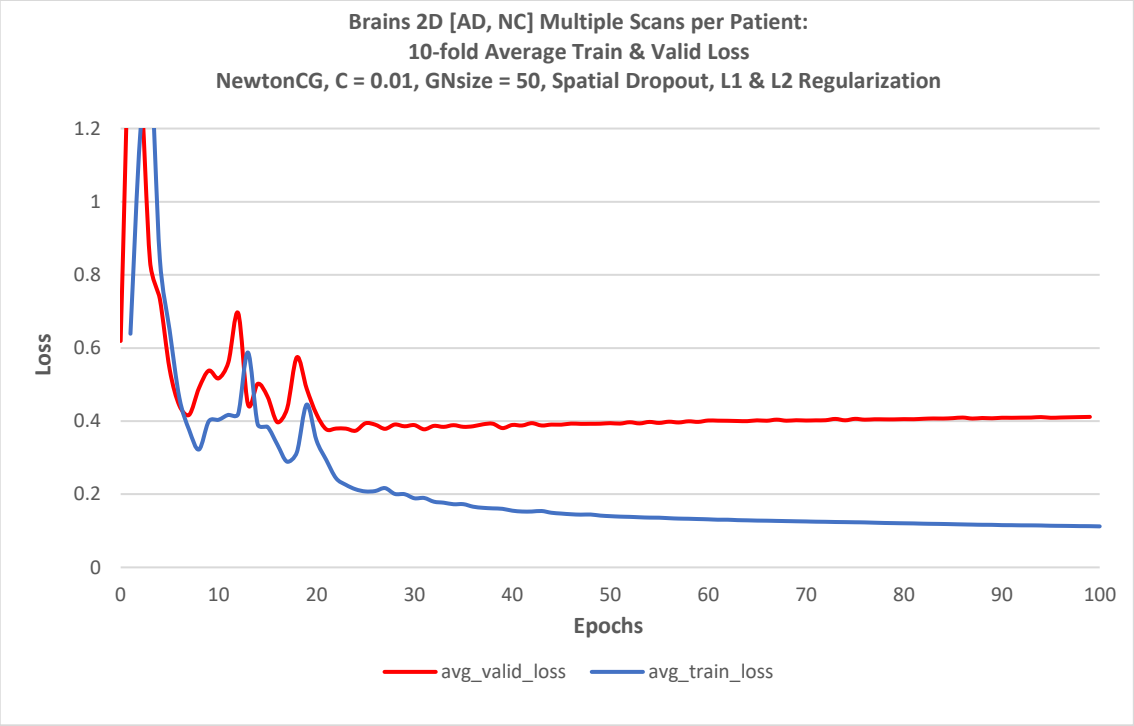


Figure H.217. Caption in the title.

H.2.22.2 Training

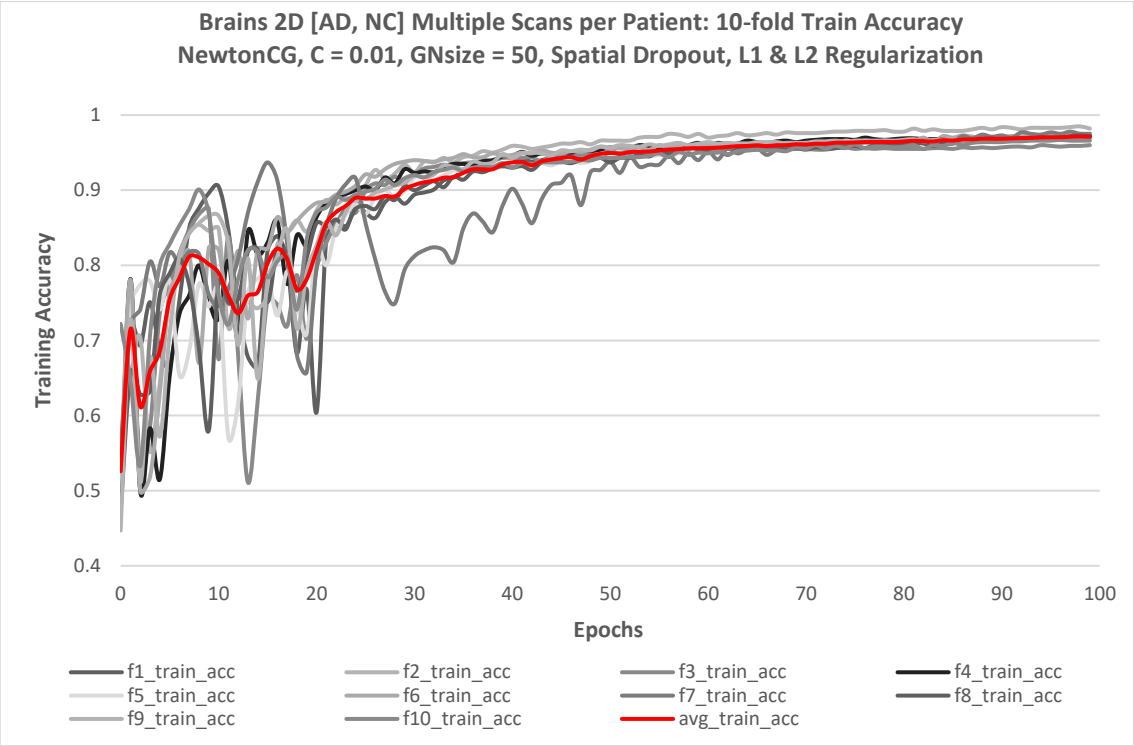


Figure H.218. Caption in the title.

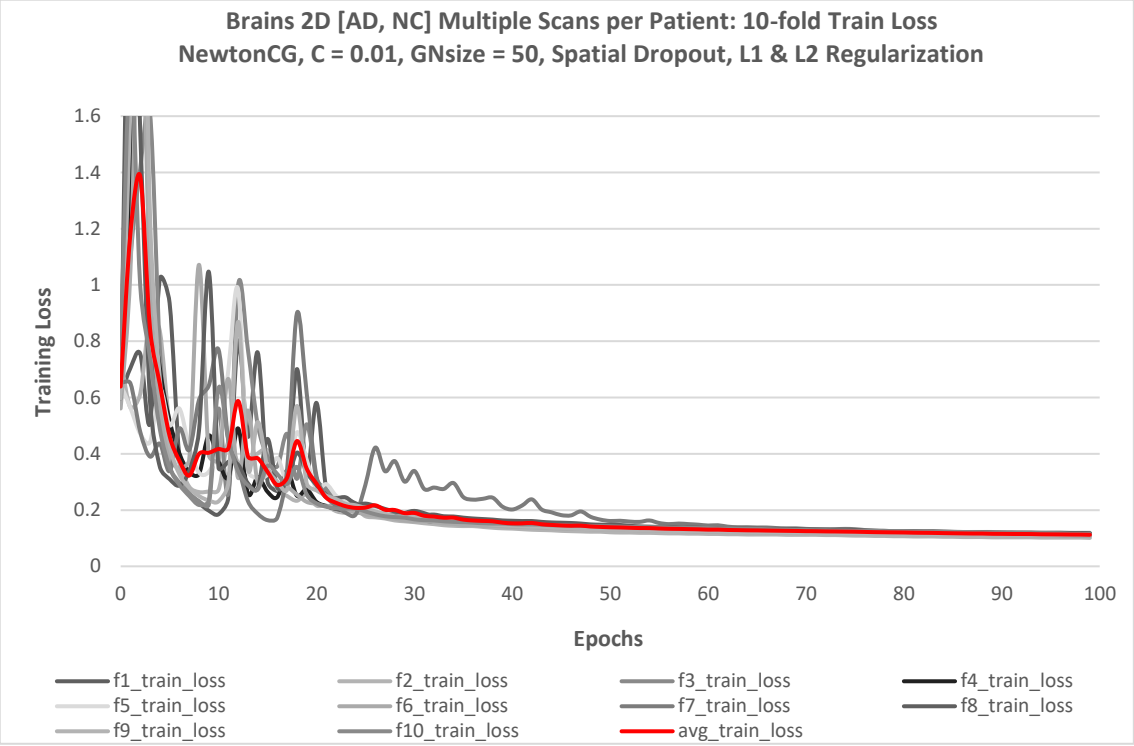


Figure H.219. Caption in the title.

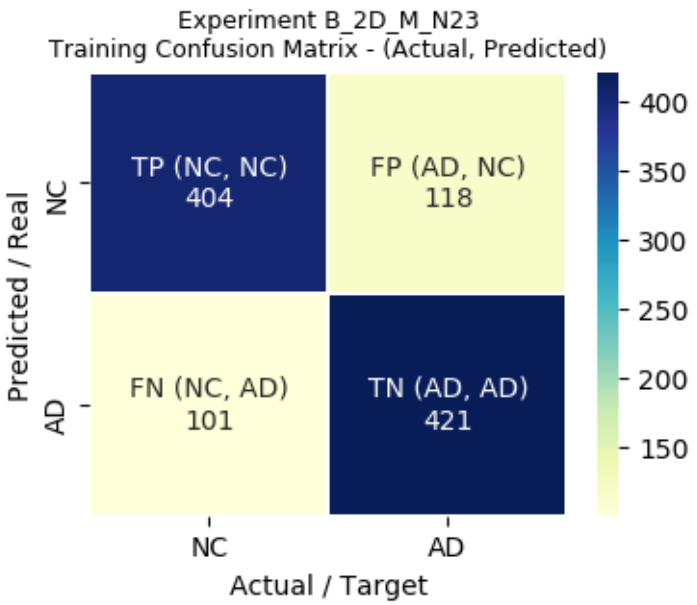


Figure H.220. Caption in the title.

H.2.22.3 Validation

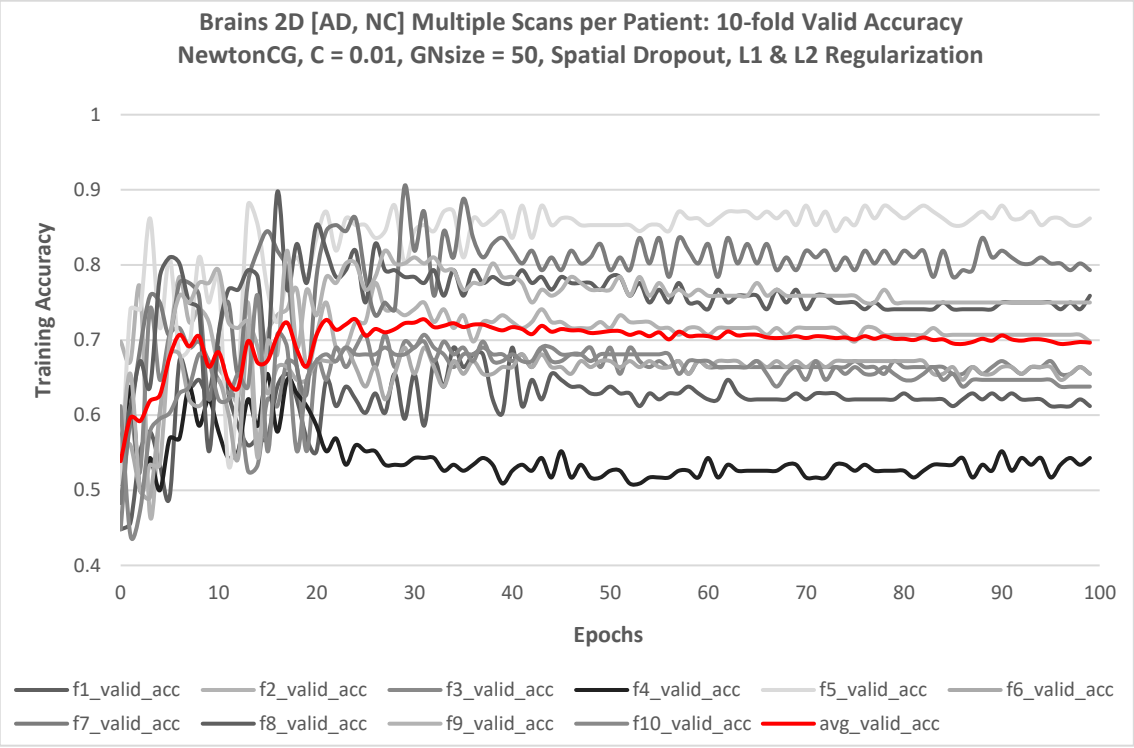


Figure H.221. Caption in the title.

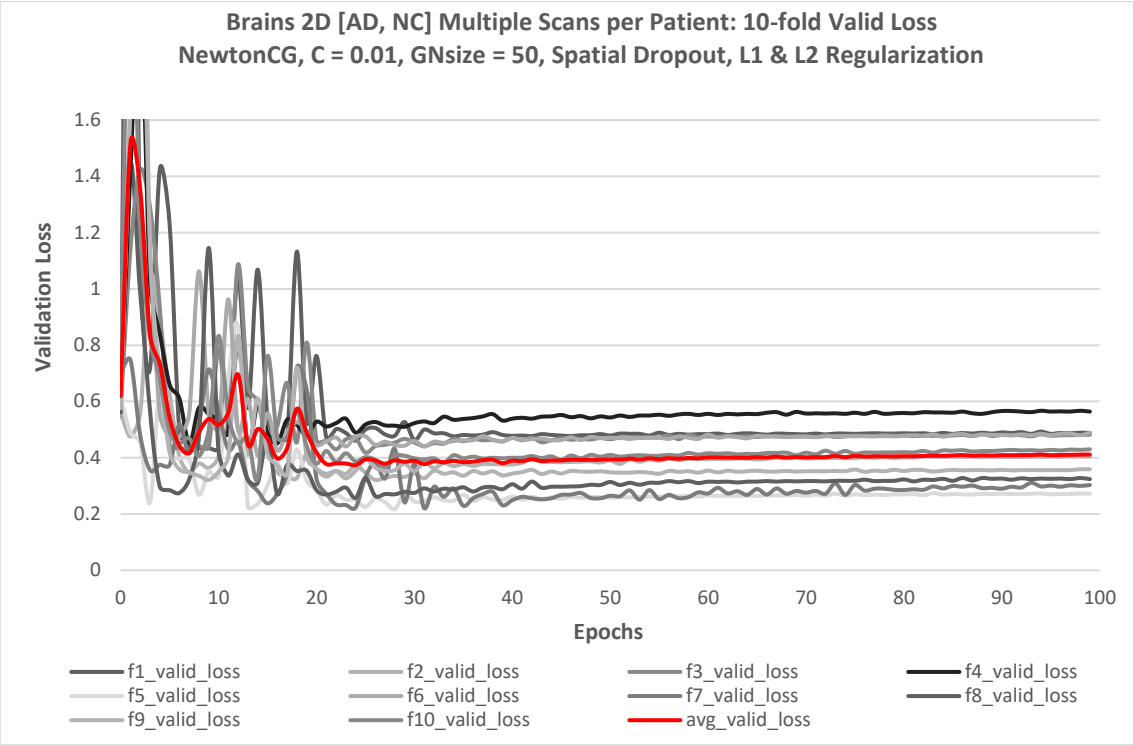


Figure H.222. Caption in the title.

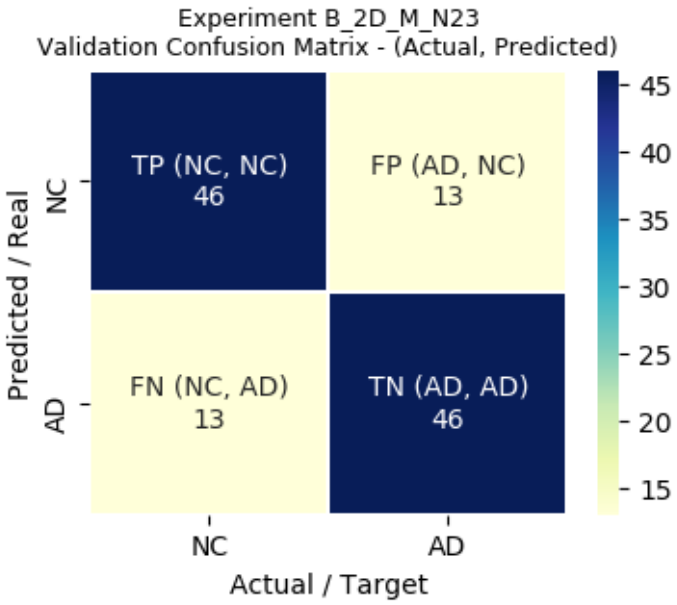


Figure H.223. Caption in the title.

H.2.22.4 Testing

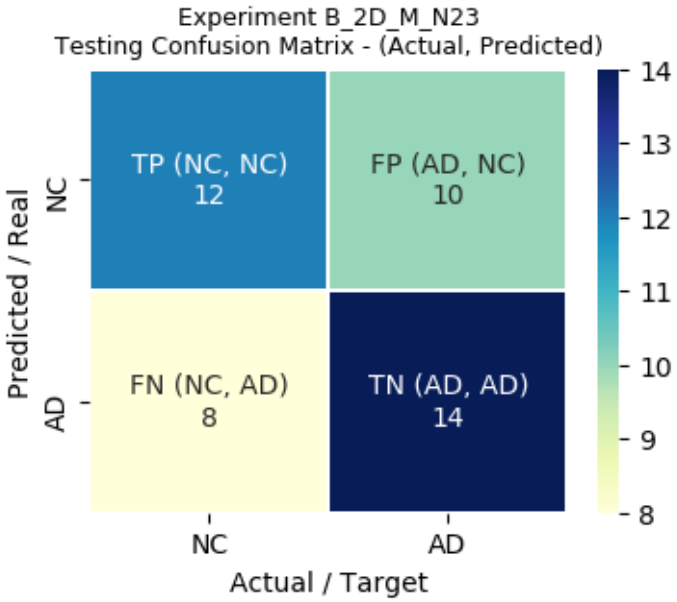


Figure H.224. Caption in the title.

H.2.23 Experiment B_2D_M_N24: C = 0.01, GNsize = 50, Spatial Dropout, Last Layer's Activation = SoftMax

H.2.23.1 General

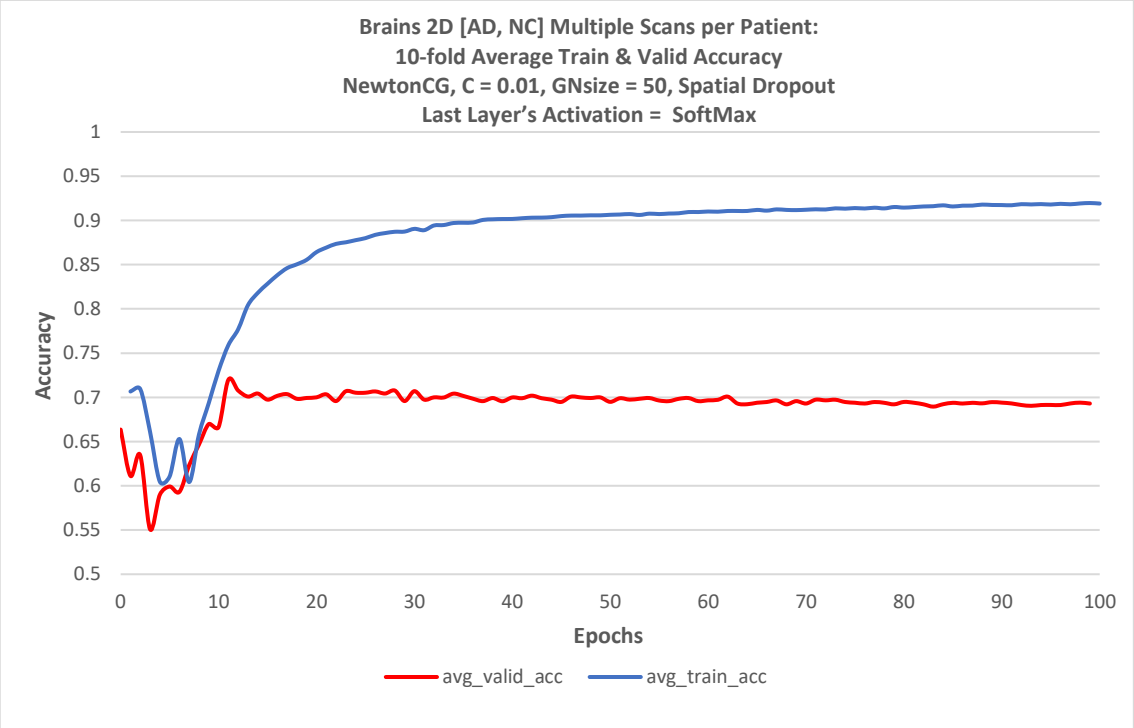


Figure H.225. Caption in the title.

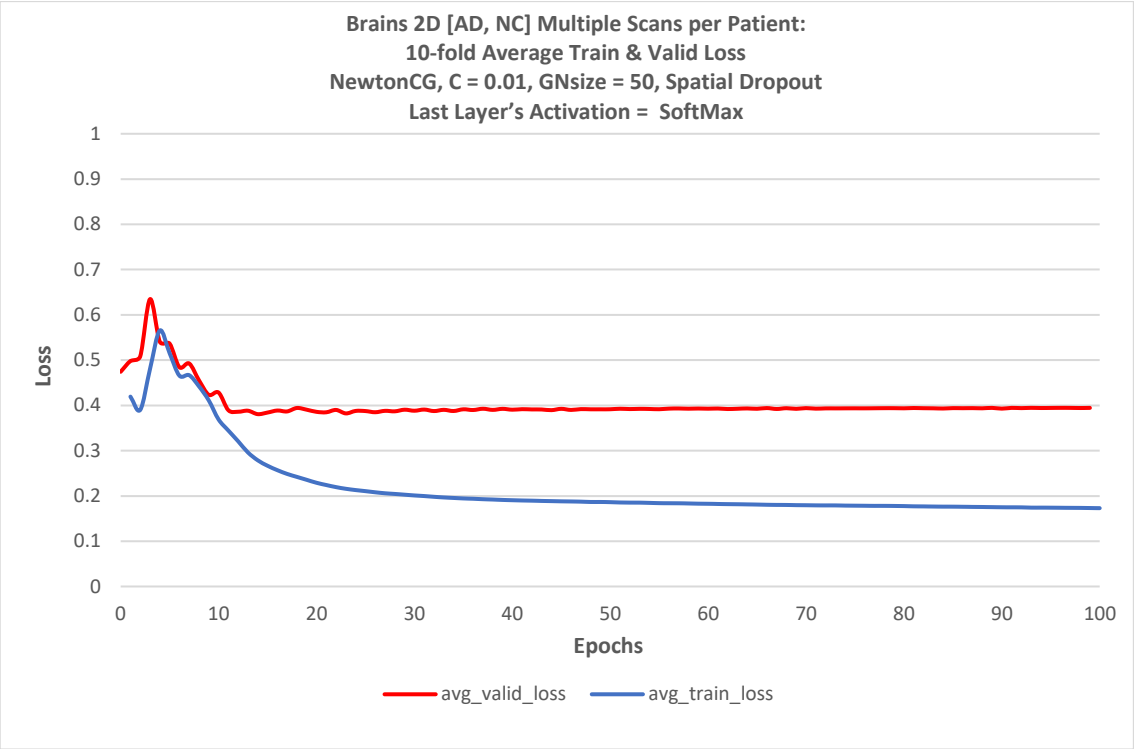


Figure H.226. Caption in the title.

H.2.23.2 Training

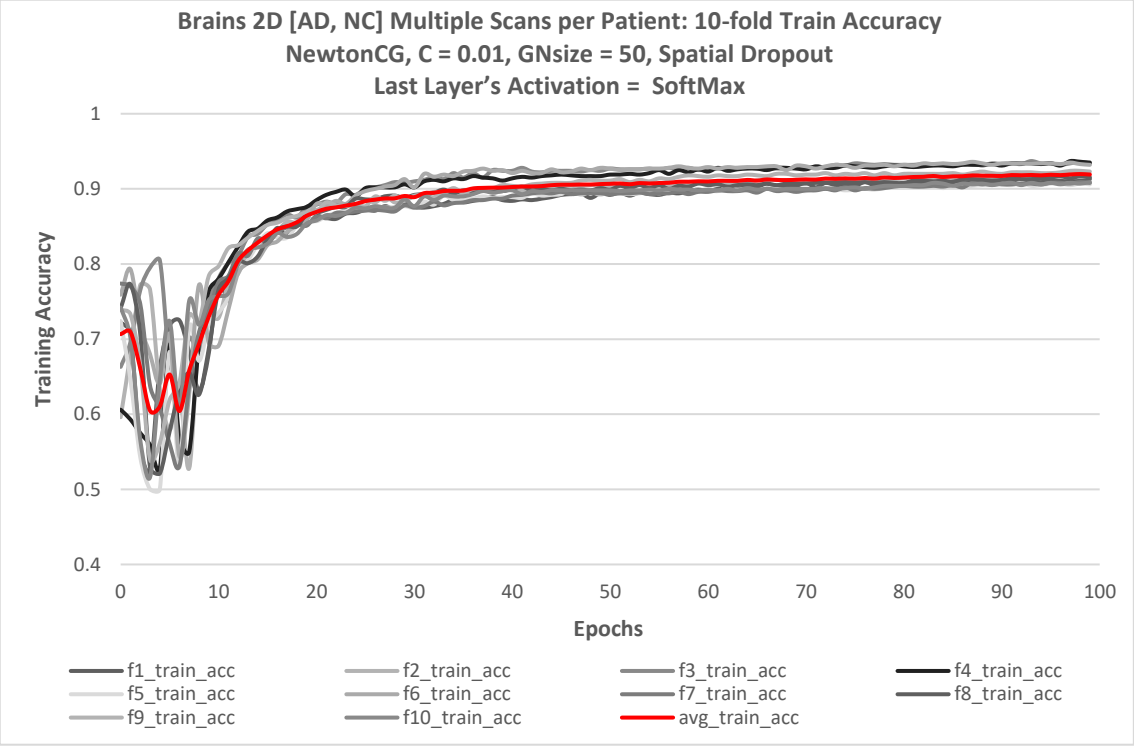


Figure H.227. Caption in the title.

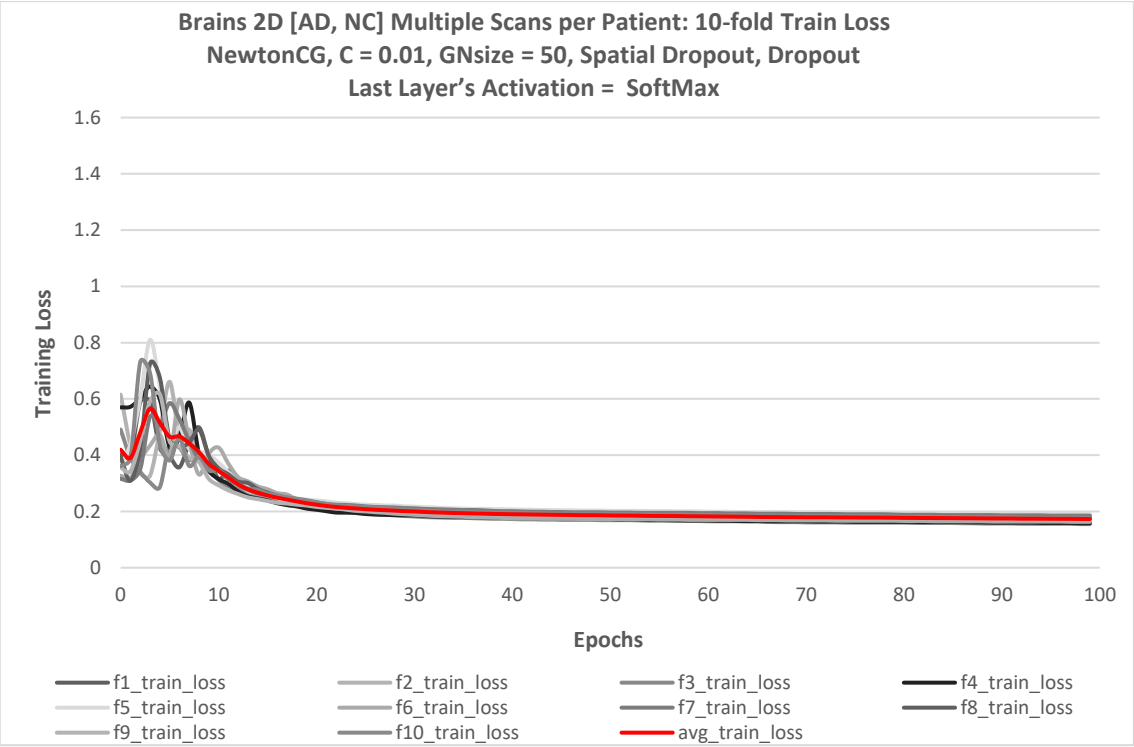


Figure H.228. Caption in the title.

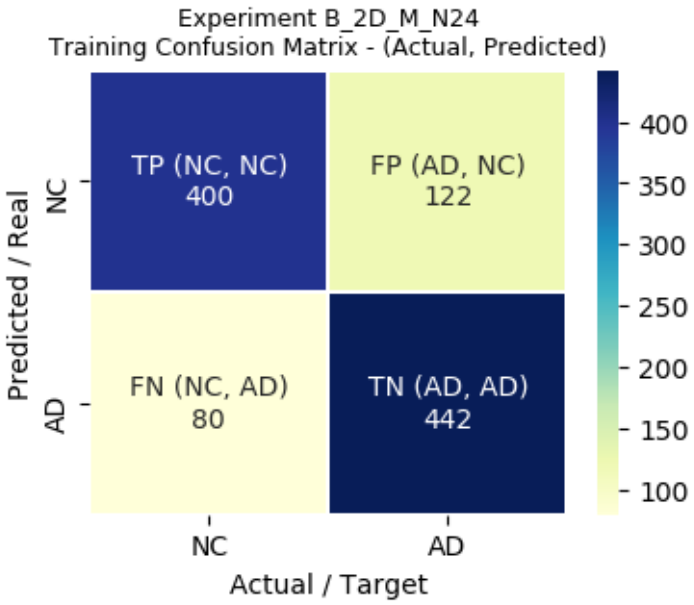


Figure H.229. Caption in the title.

H.2.23.3 Validation

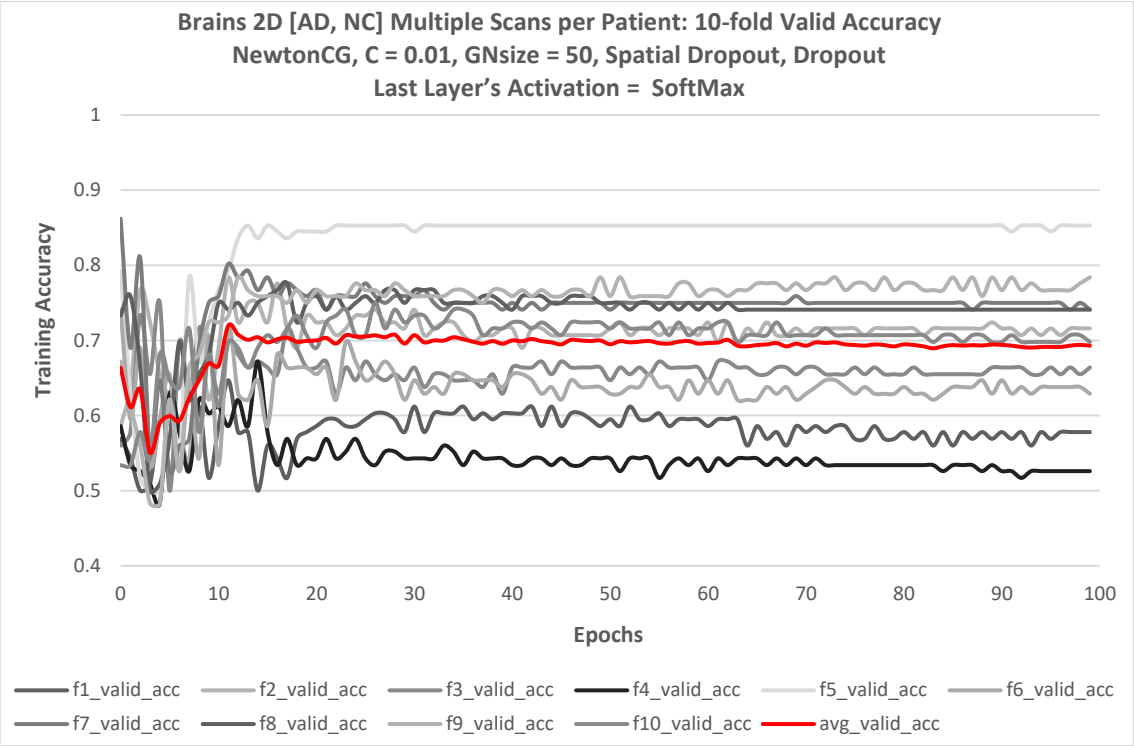


Figure H.230. Caption in the title.

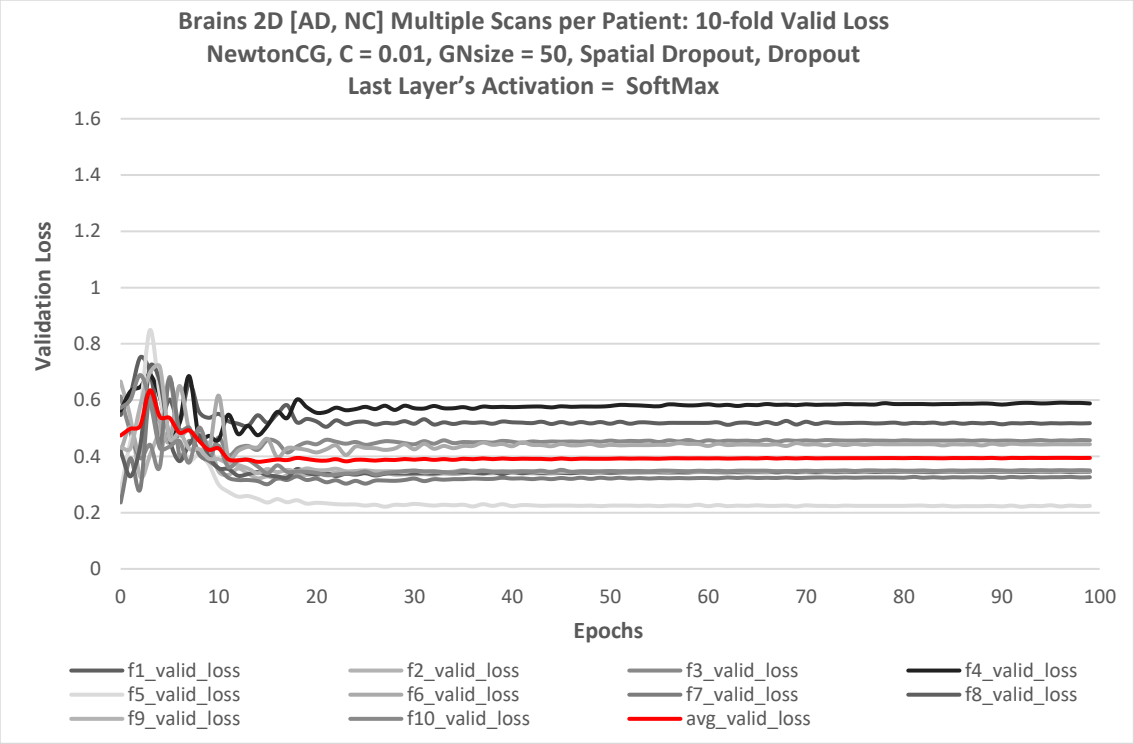


Figure H.231. Caption in the title.

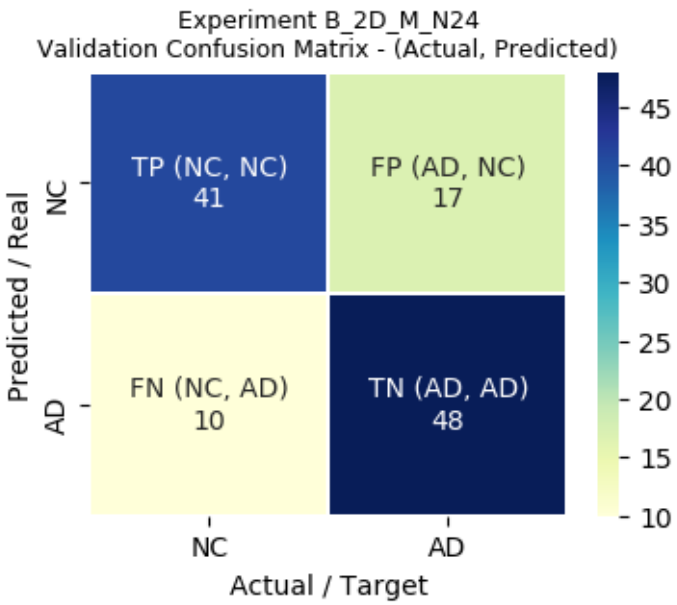


Figure H.232. Caption in the title.

H.2.23.4 Testing

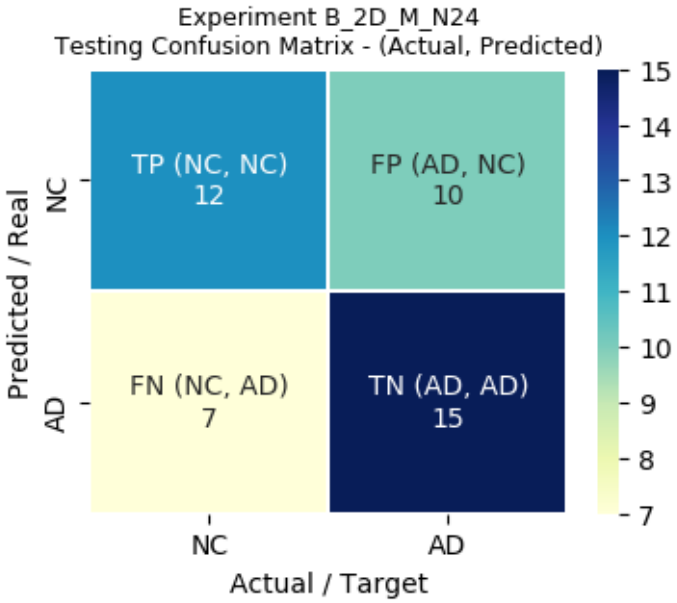


Figure H.233. Caption in the title.

H.2.24 Experiment B_2D_M_N25: C = 0.01, GNsize = 50, Spatial Dropout, Dropout, Last Layer's Activation = SoftMax

H.2.24.1 General

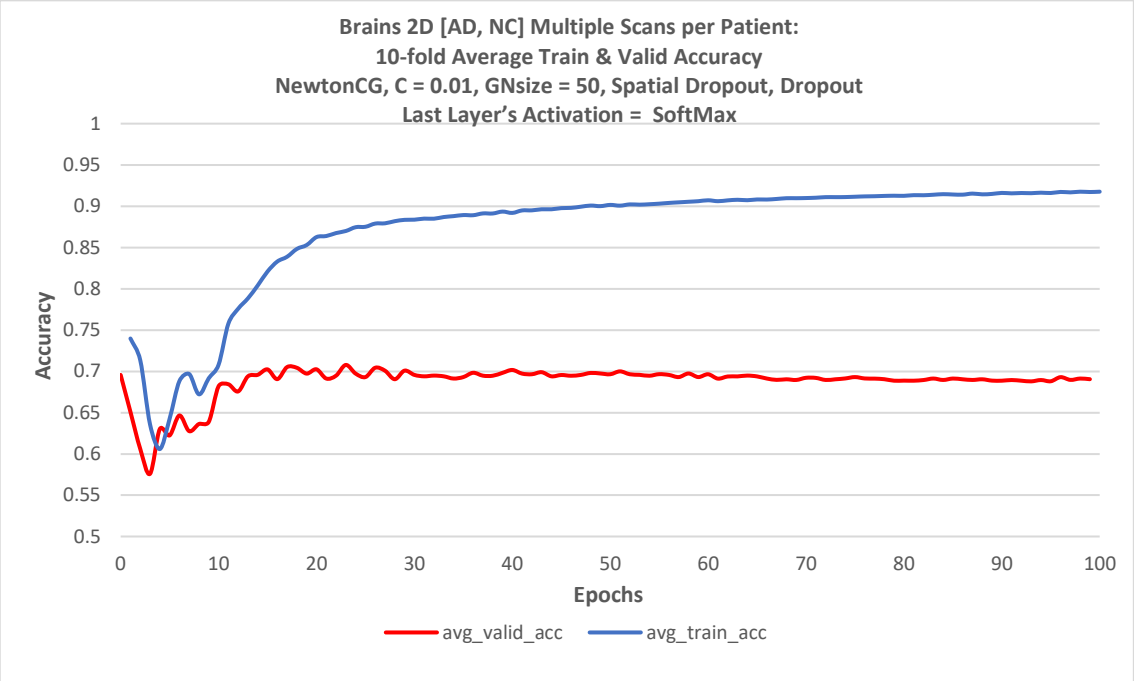


Figure H.234. Caption in the title.

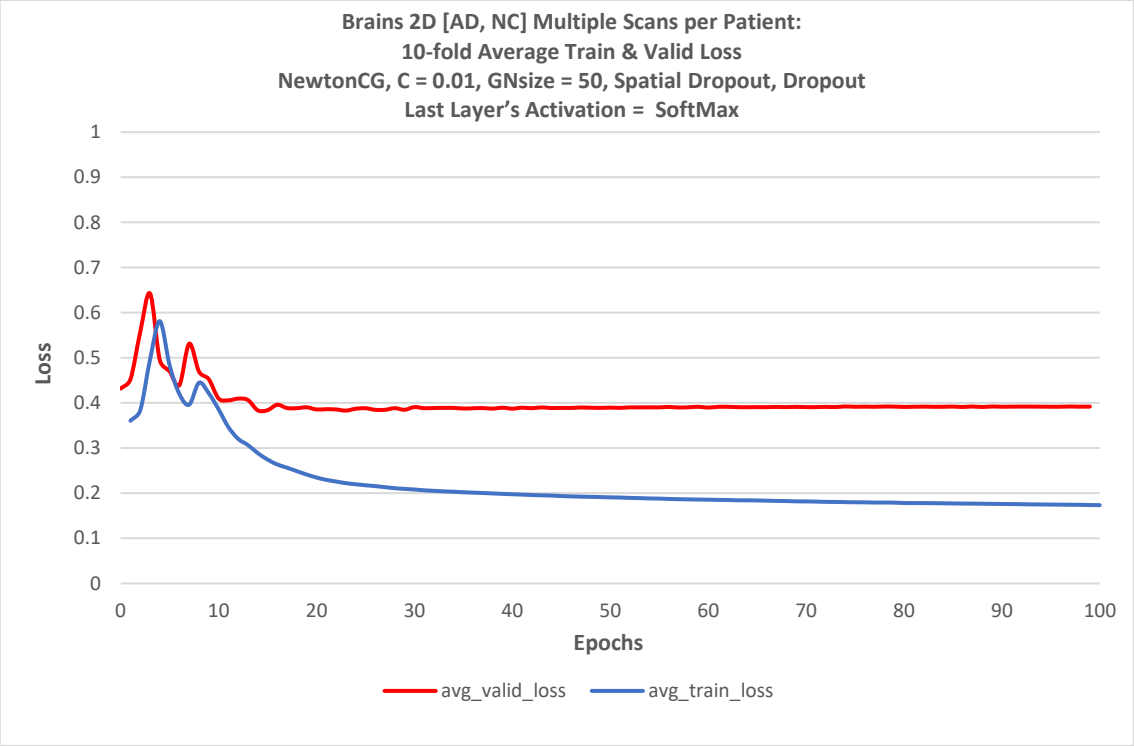


Figure H.235. Caption in the title.

H.2.24.2 Training

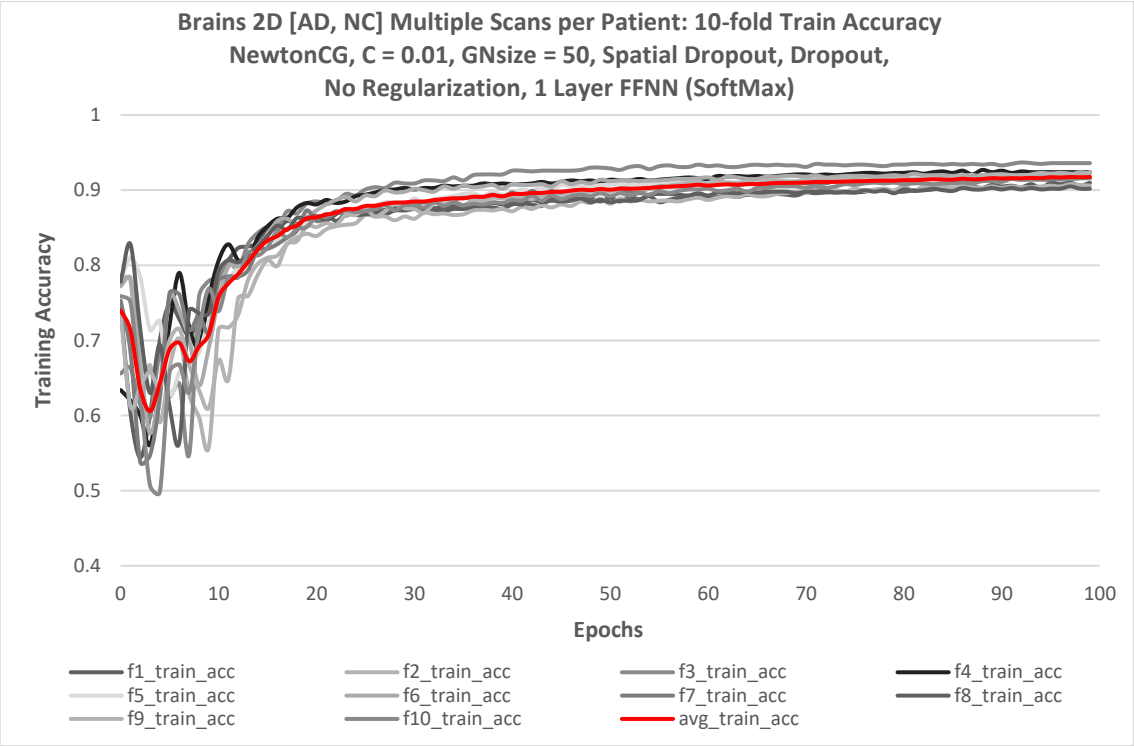


Figure H.236. Caption in the title.

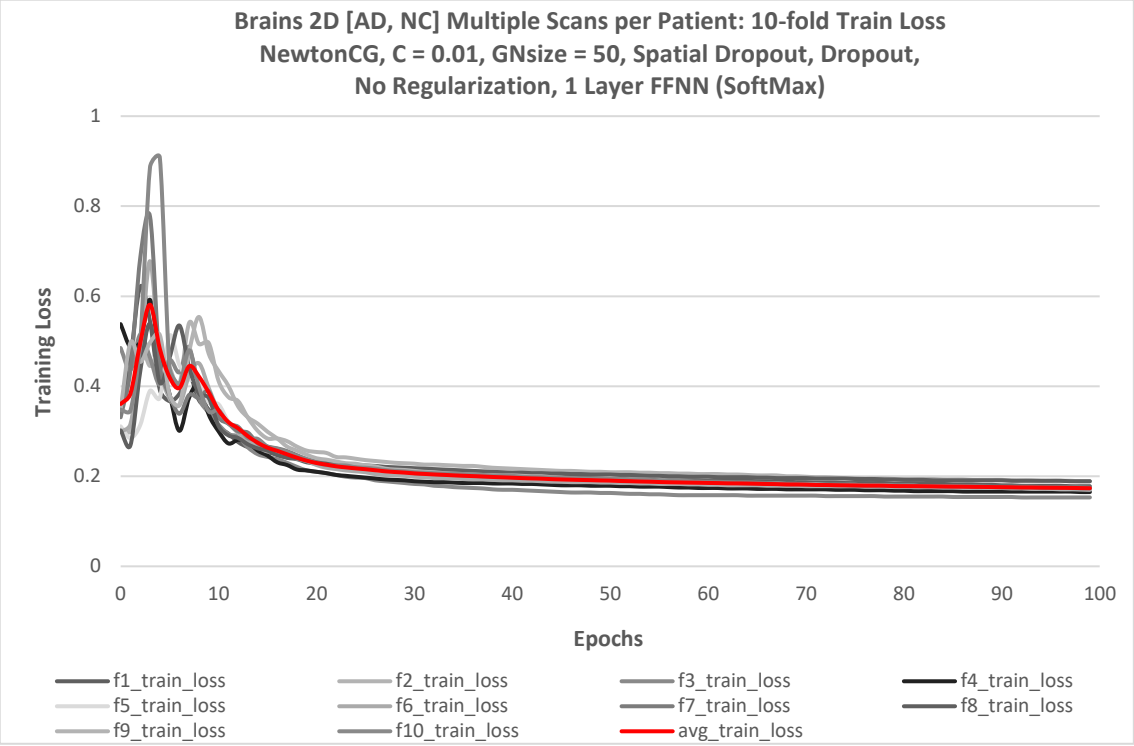


Figure H.237. Caption in the title.

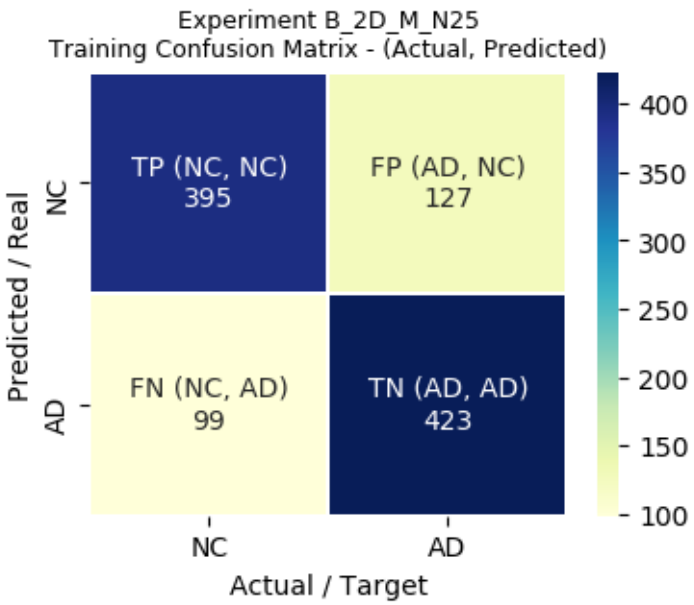


Figure H.238. Caption in the title.

H.2.24.3 Validation

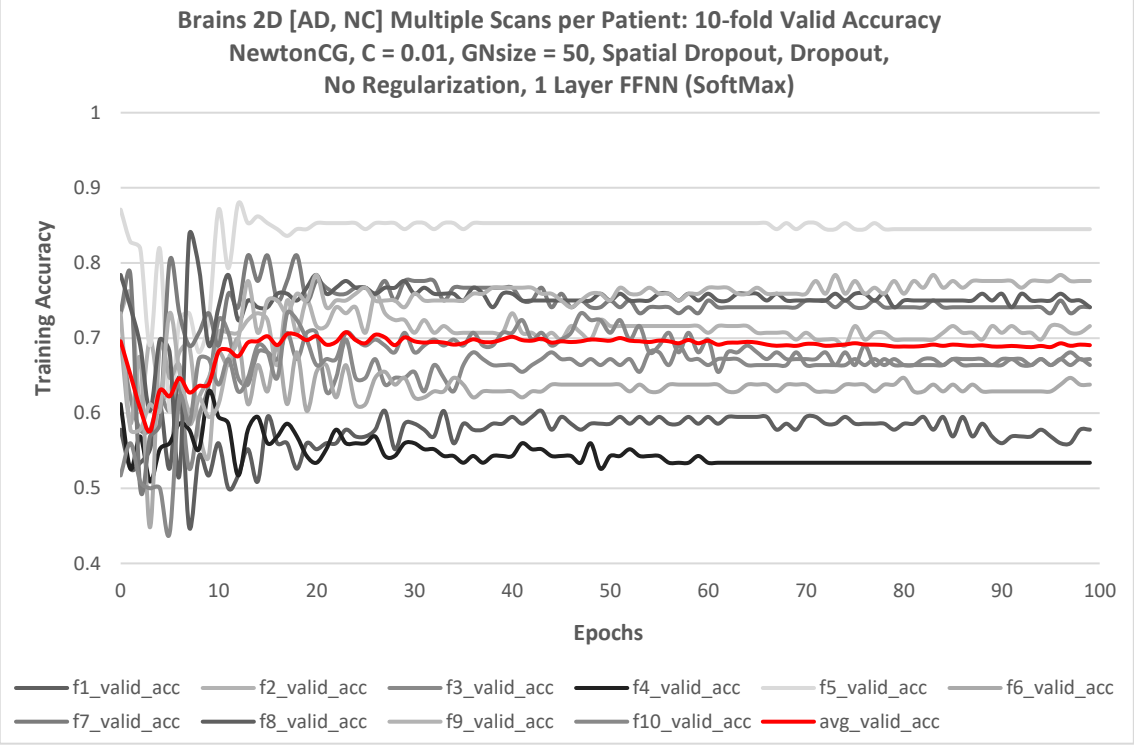


Figure H.239. Caption in the title.

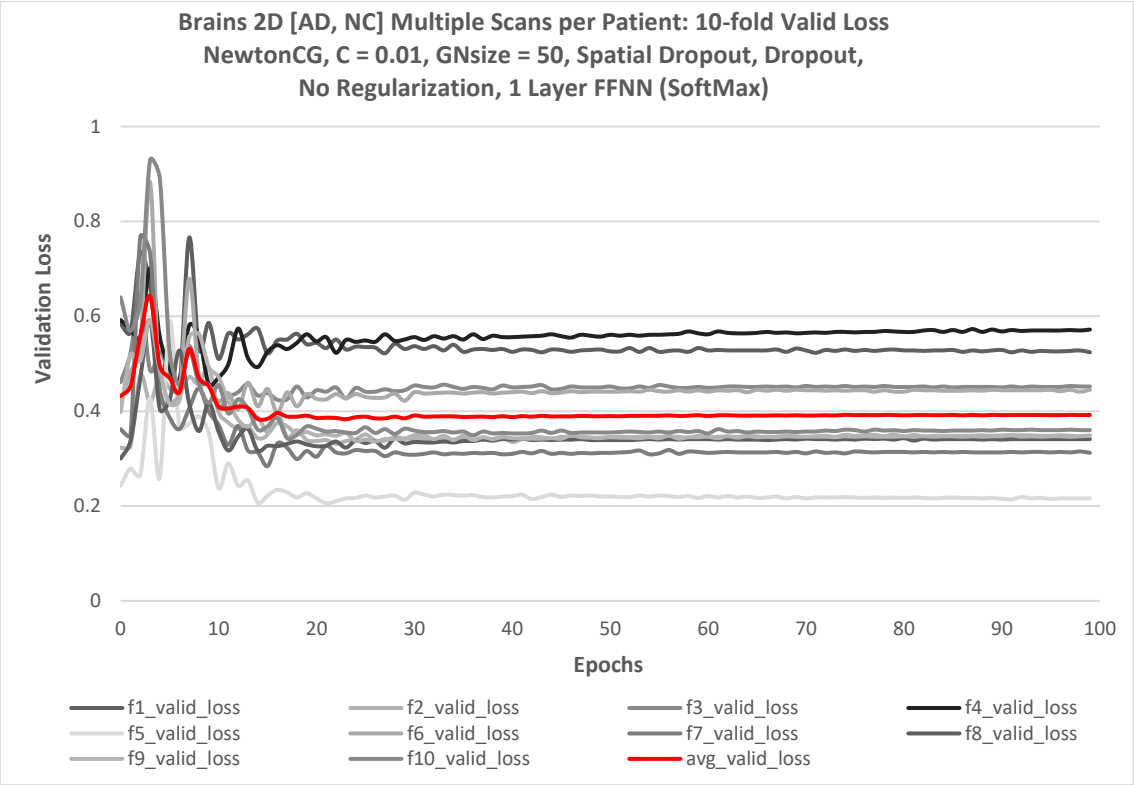


Figure H.240. Caption in the title.

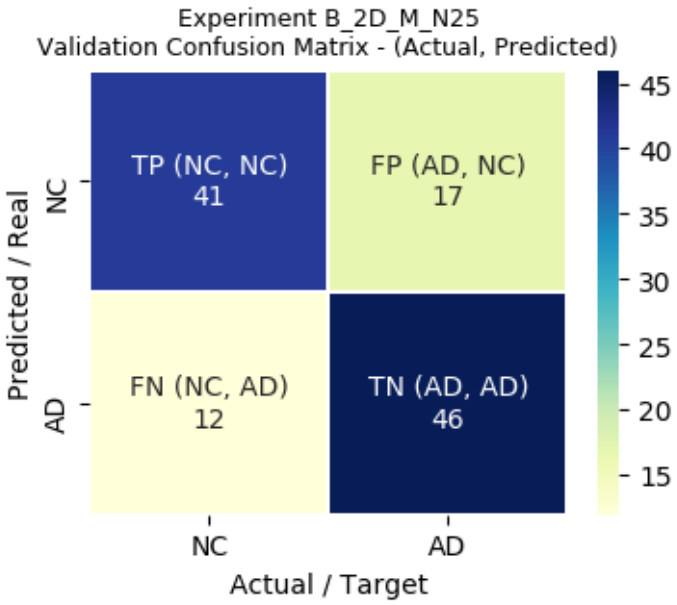


Figure H.241. Caption in the title.

H.2.24.4 Testing

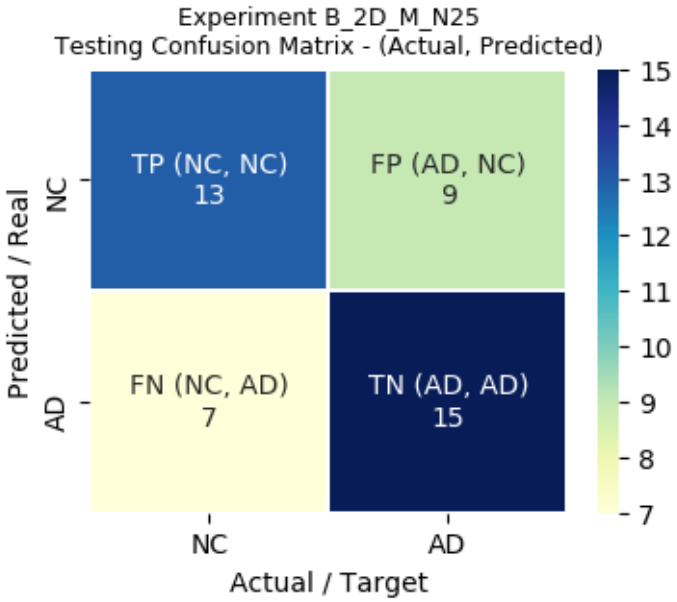


Figure H.242. Caption in the title.

H.2.25 Experiment B_2D_M_N26: C = 0.01, GNsize = 50, Spatial Dropout, Dropout, L1 & L2 Regularization, Last Layer's Activation = SoftMax

H.2.25.1 General

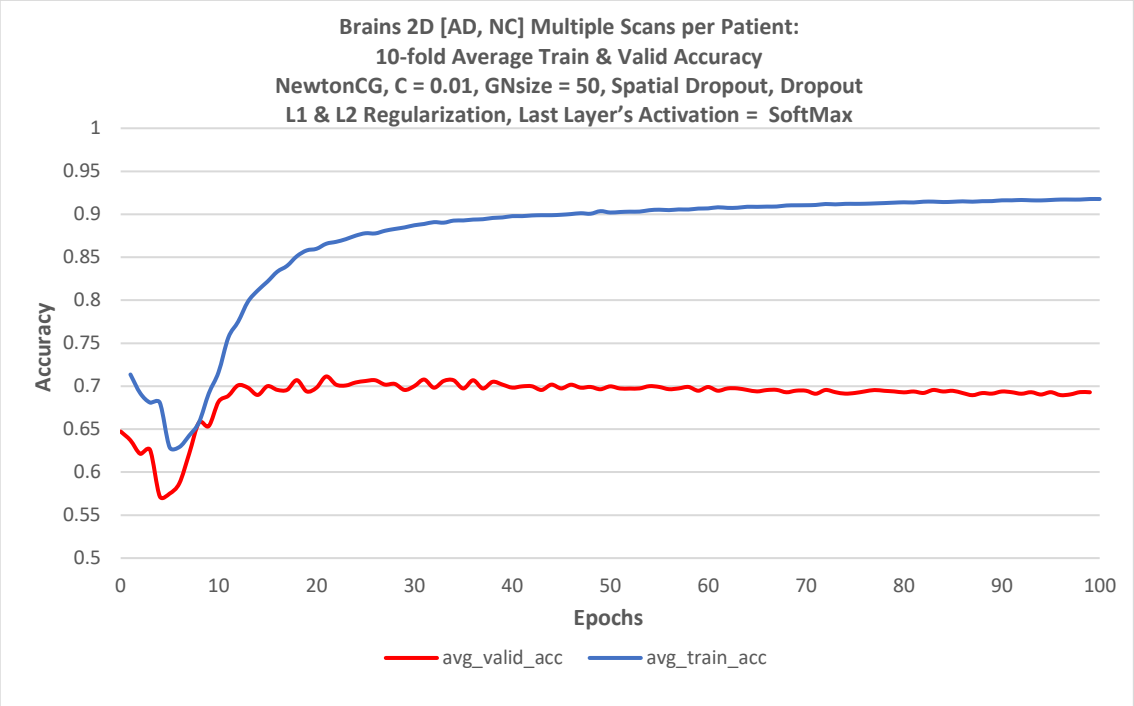


Figure H.243. Caption in the title.

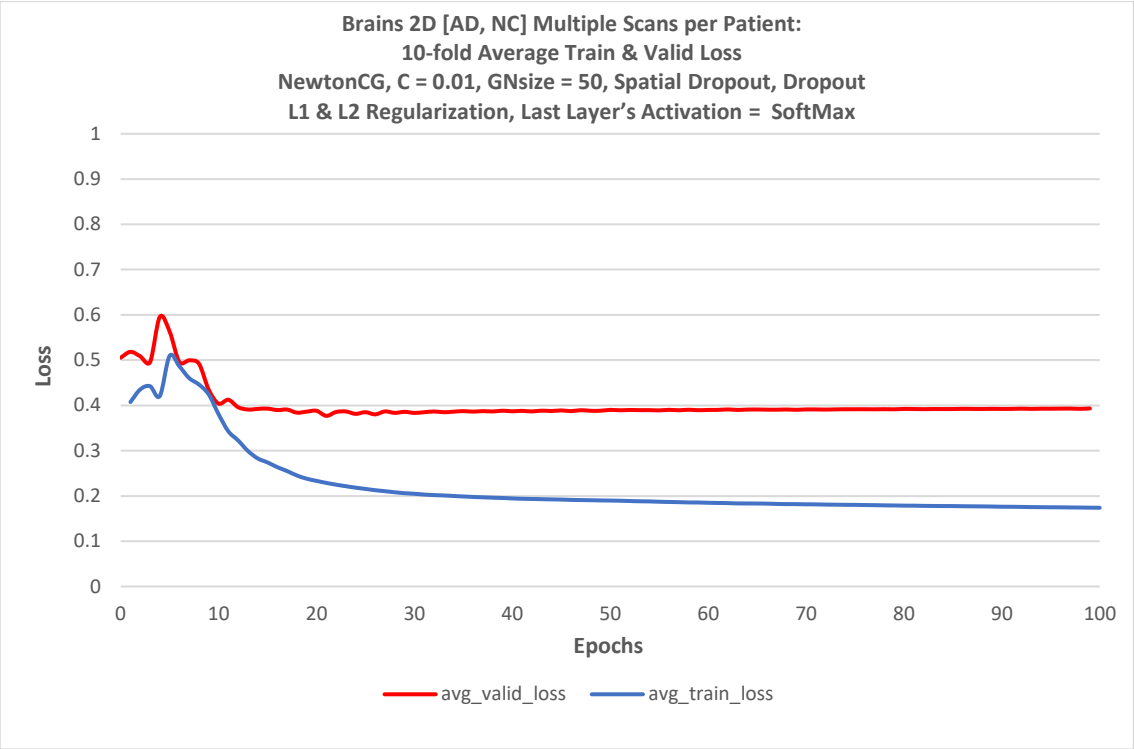


Figure H.244. Caption in the title.

H.2.25.2 Training

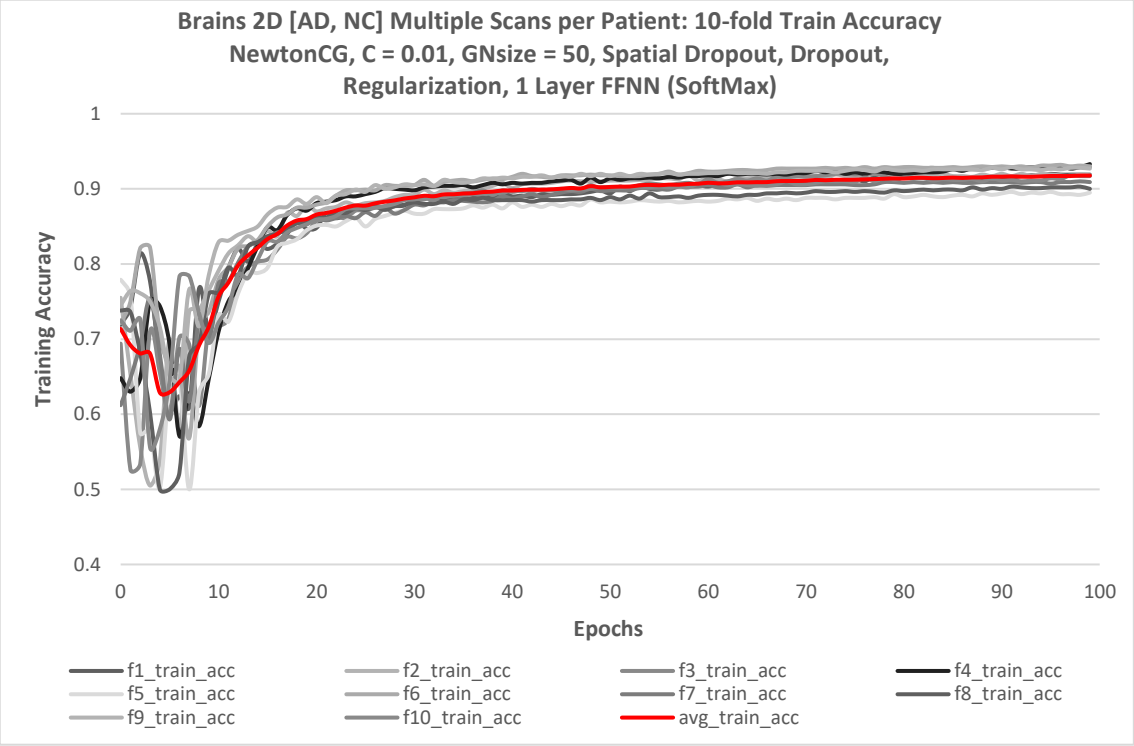


Figure H.245. Caption in the title.

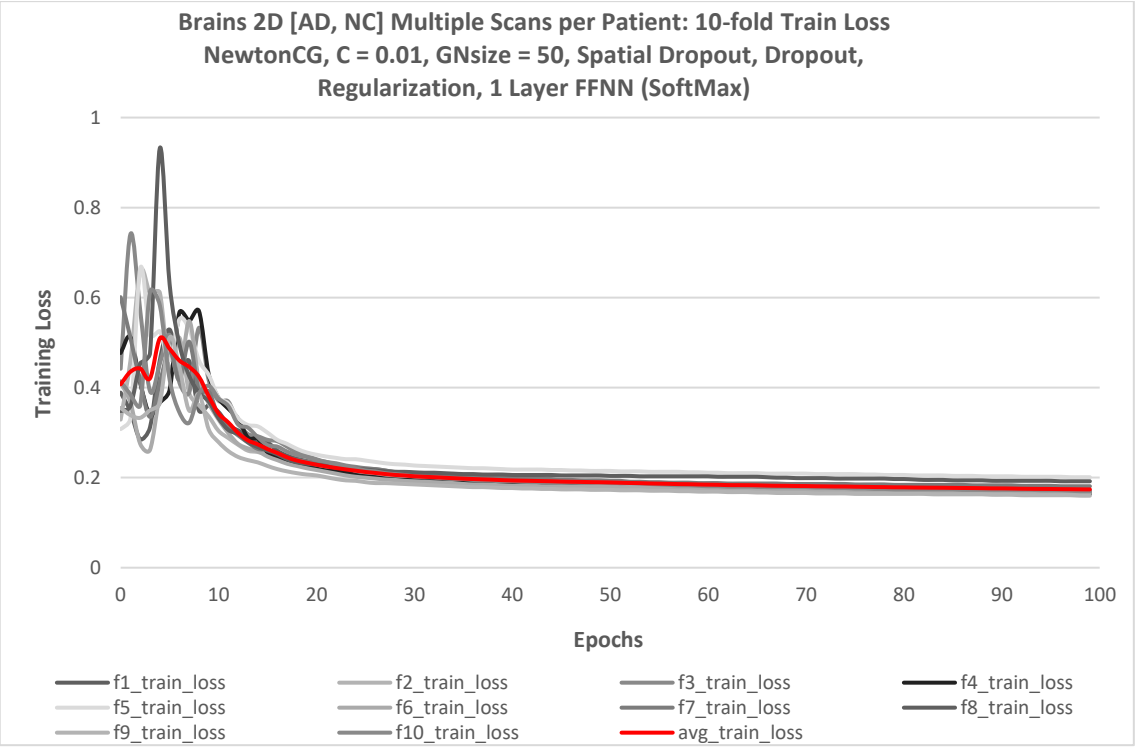


Figure H.246. Caption in the title.

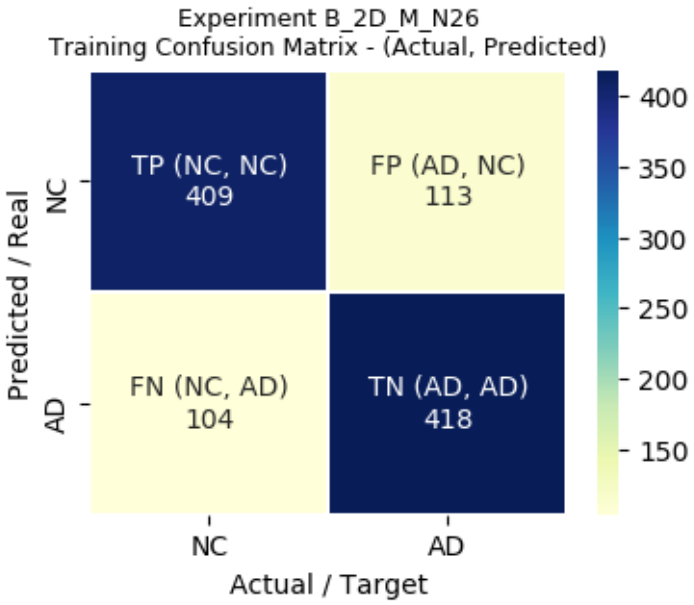


Figure H.247. Caption in the title.

H.2.25.3 Validation

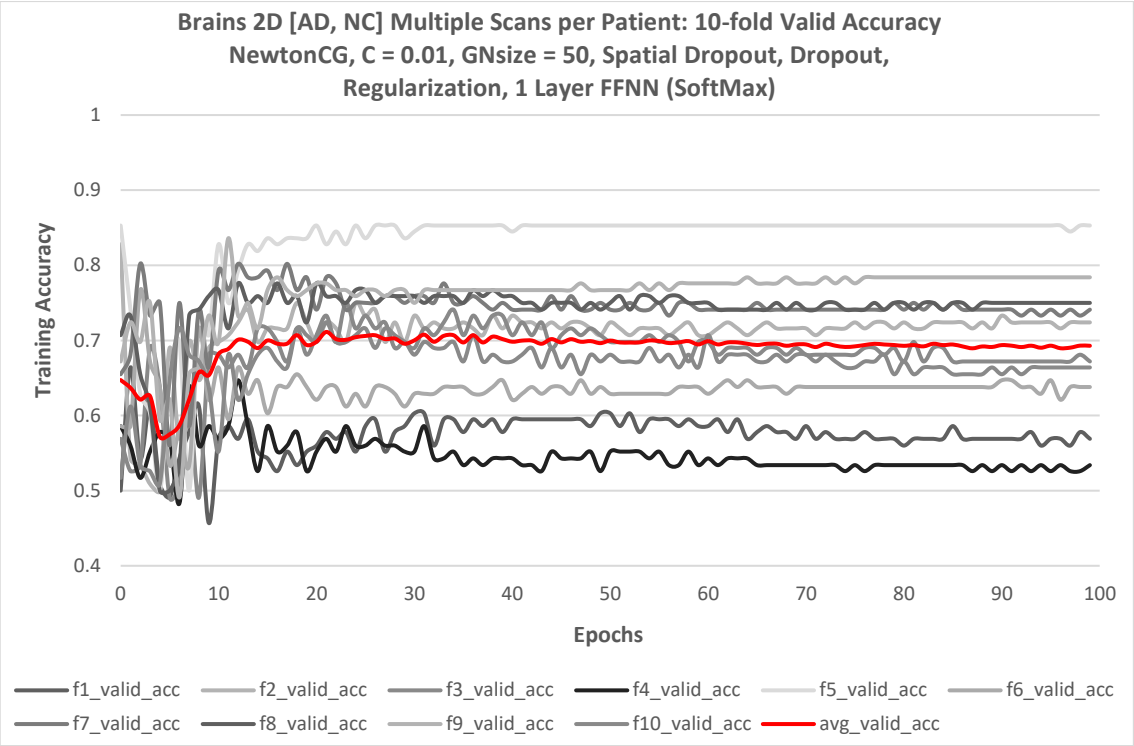


Figure H.248. Caption in the title.

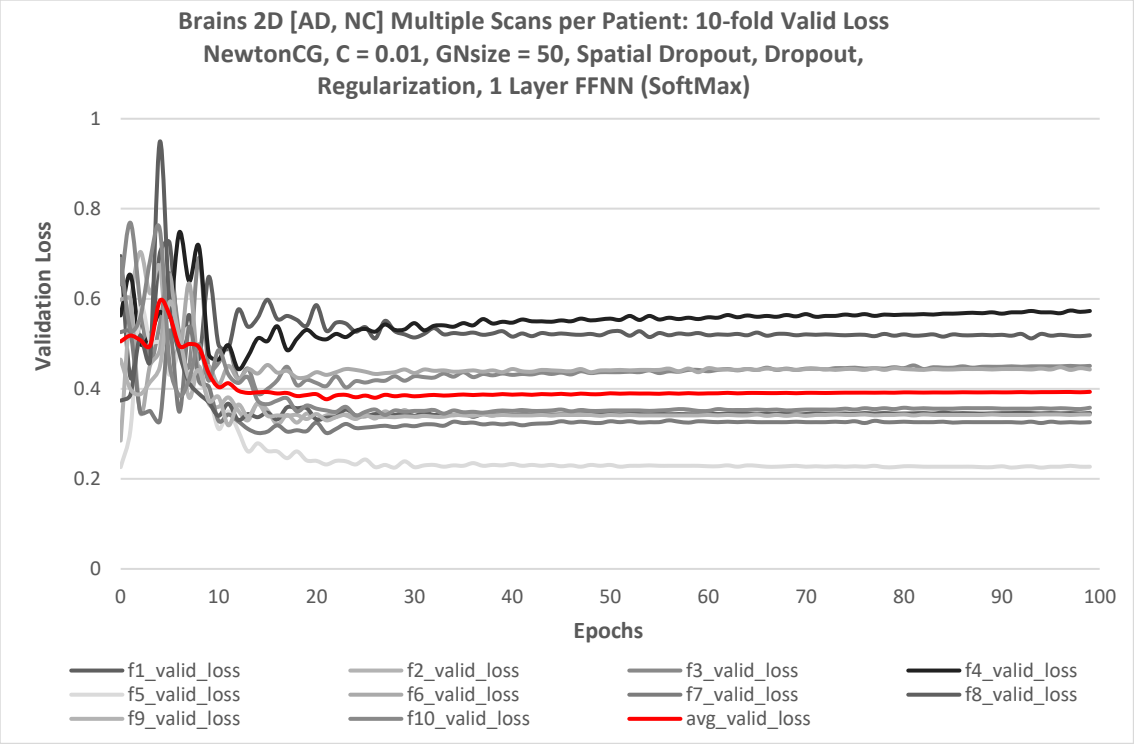


Figure H.249. Caption in the title.

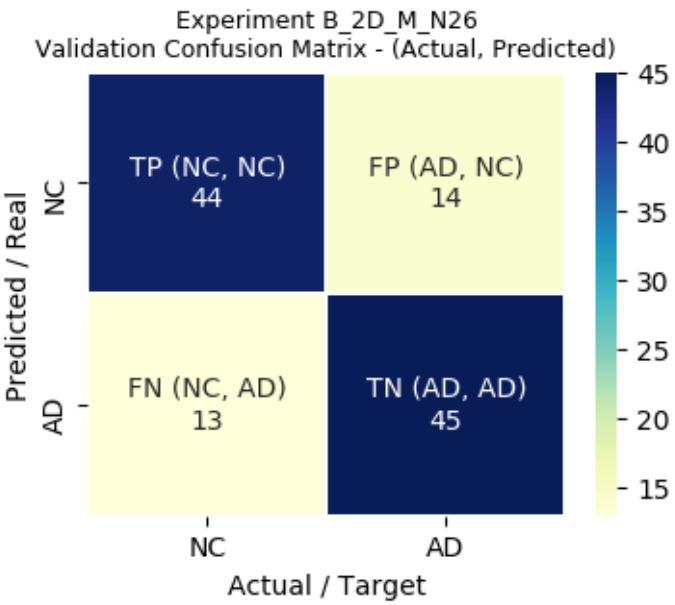


Figure H.250. Caption in the title.

H.2.25.4 Testing

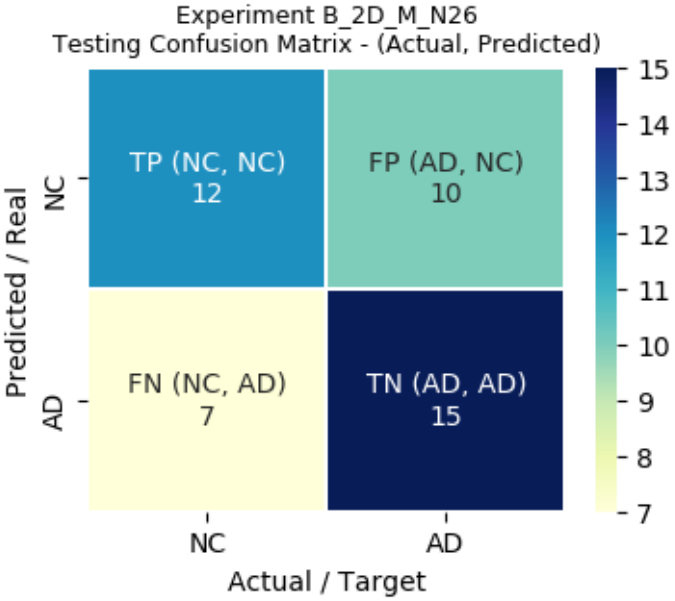


Figure H.251. Caption in the title.

H.2.26 Experiment B_2D_M_N27: C = 0.01, GNsize = 50, Spatial Dropout, Dropout, L1 & L2 Regularization, Two Layers FFNN, Last Layer's Activation = SoftMax

H.2.26.1 General

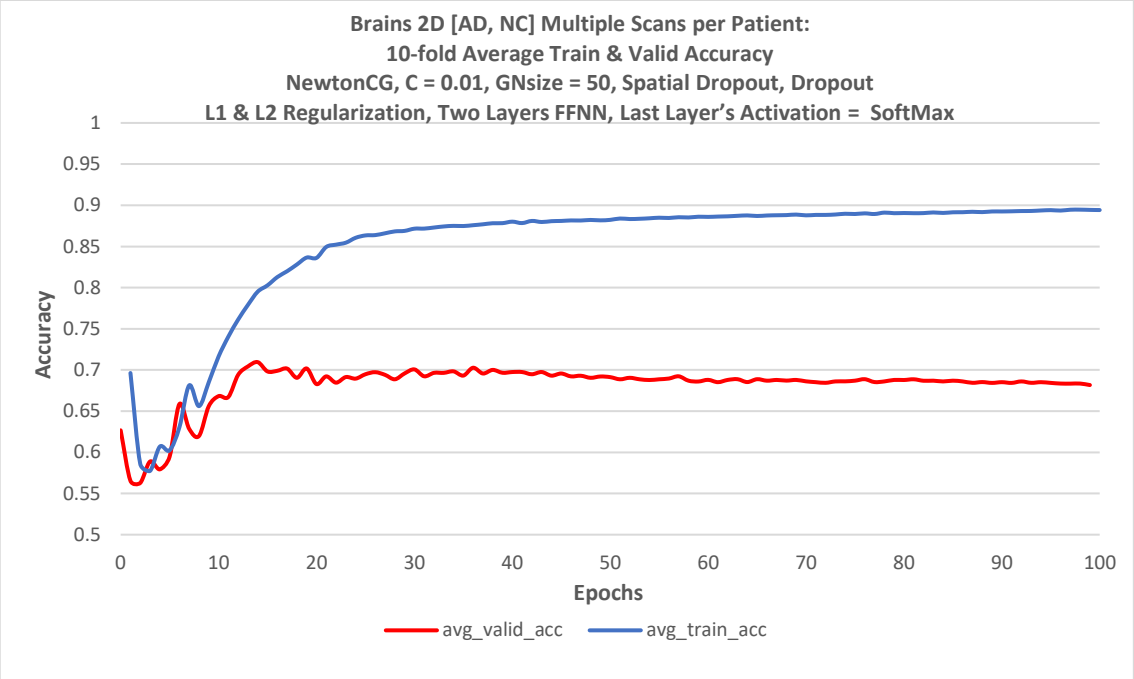


Figure H.252. Caption in the title.

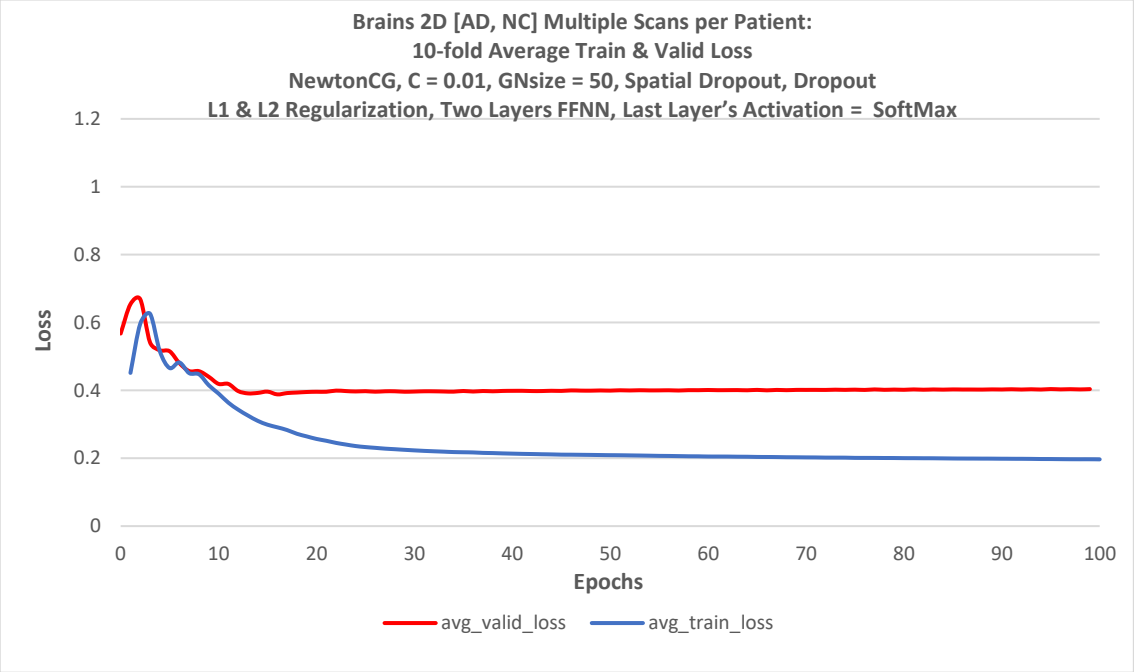


Figure H.253. Caption in the title.

H.2.26.2 Training

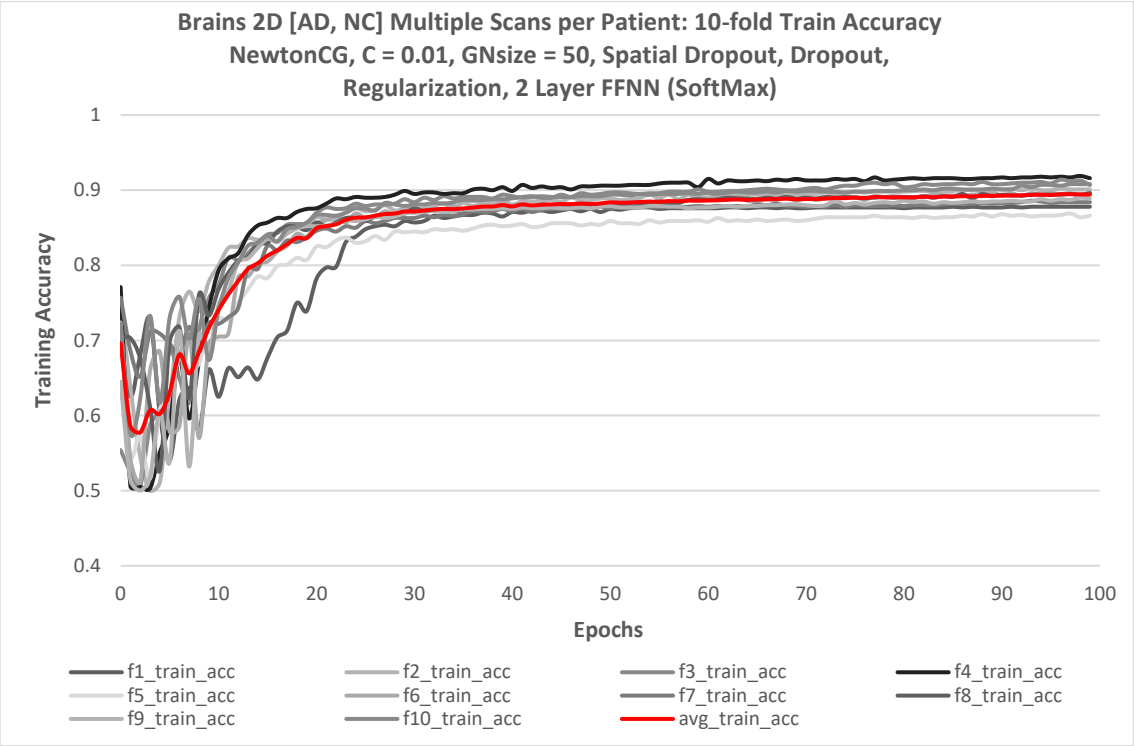


Figure H.254. Caption in the title.

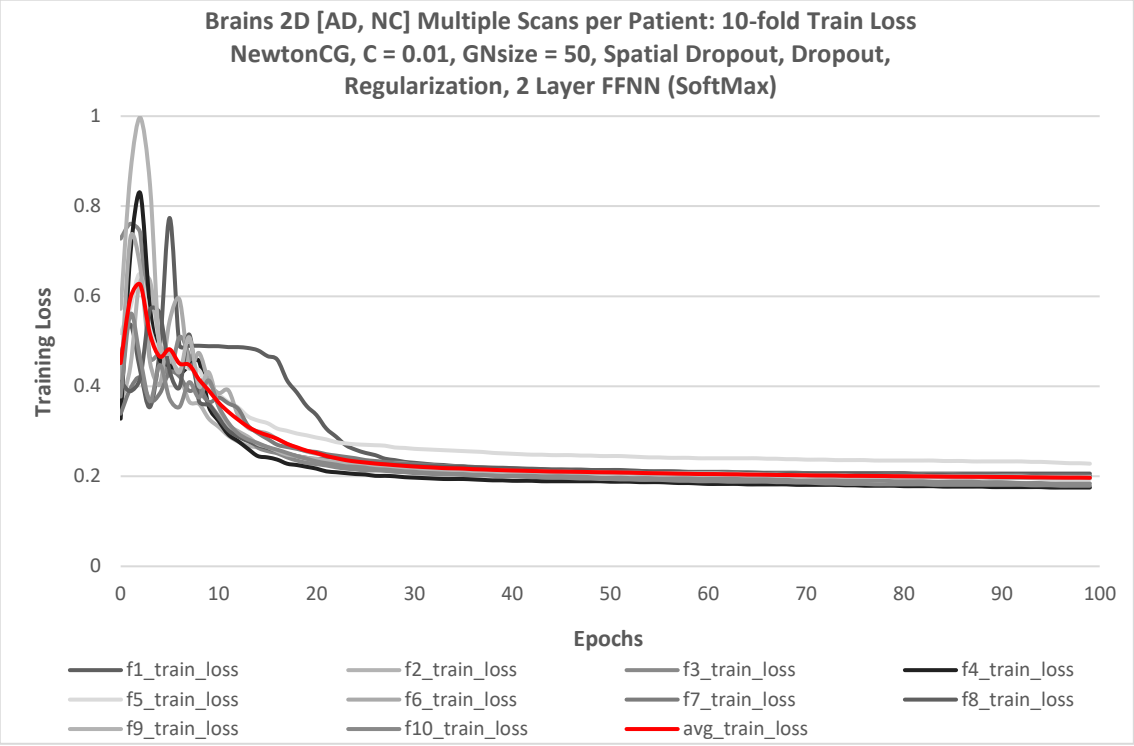


Figure H.255. Caption in the title.

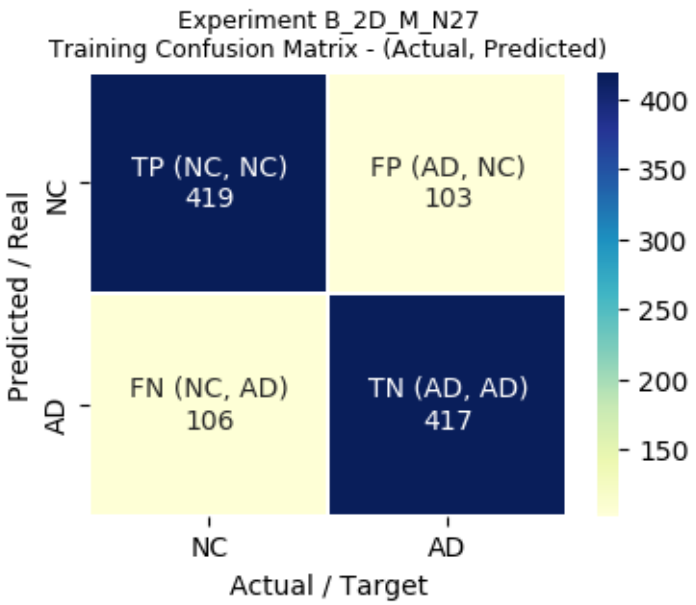


Figure H.256. Caption in the title.

H.2.26.3 Validation

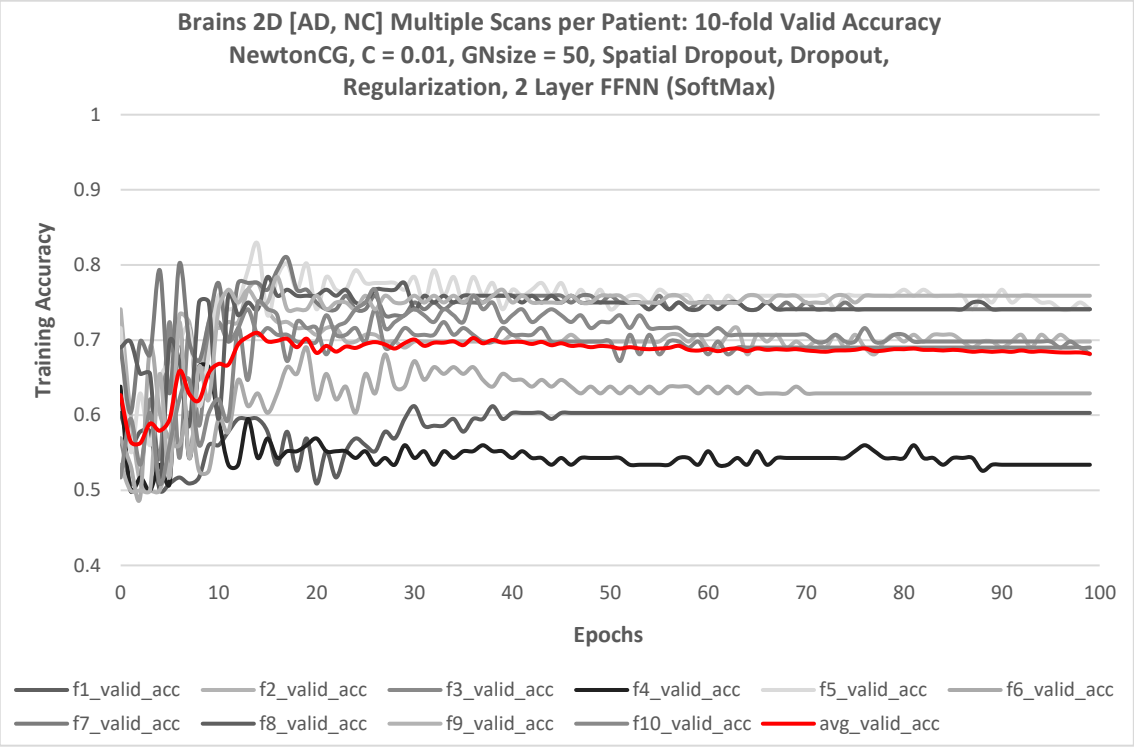


Figure H.257. Caption in the title.

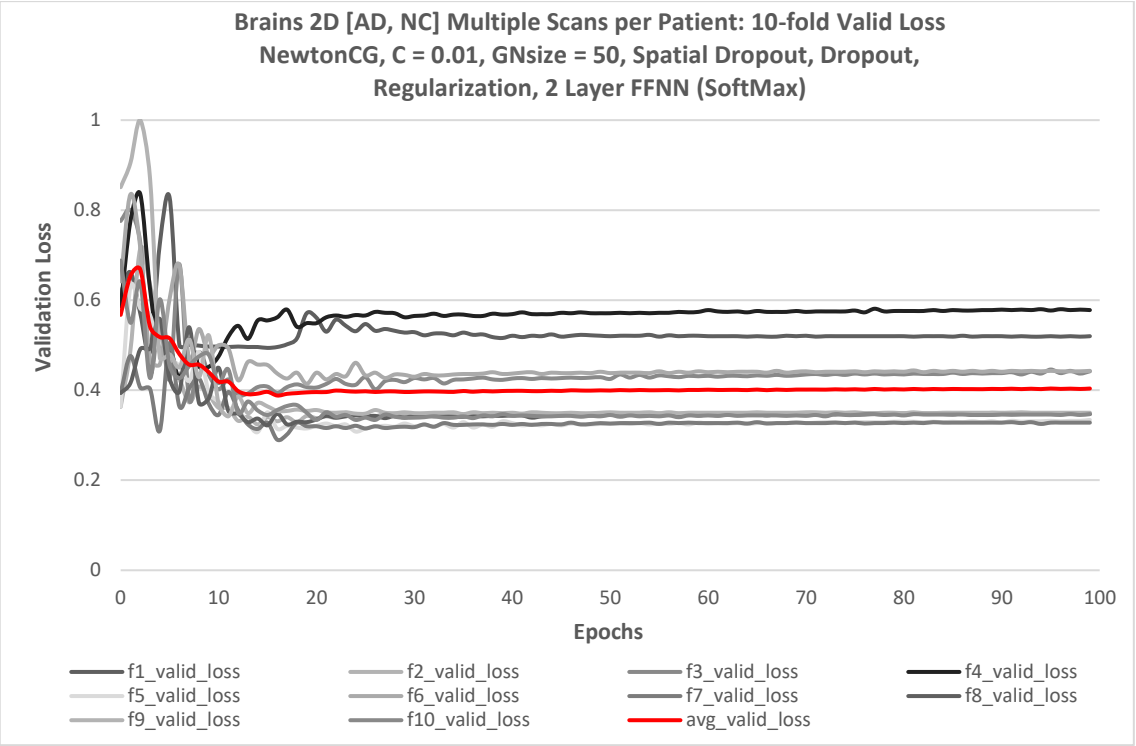


Figure H.258. Caption in the title.

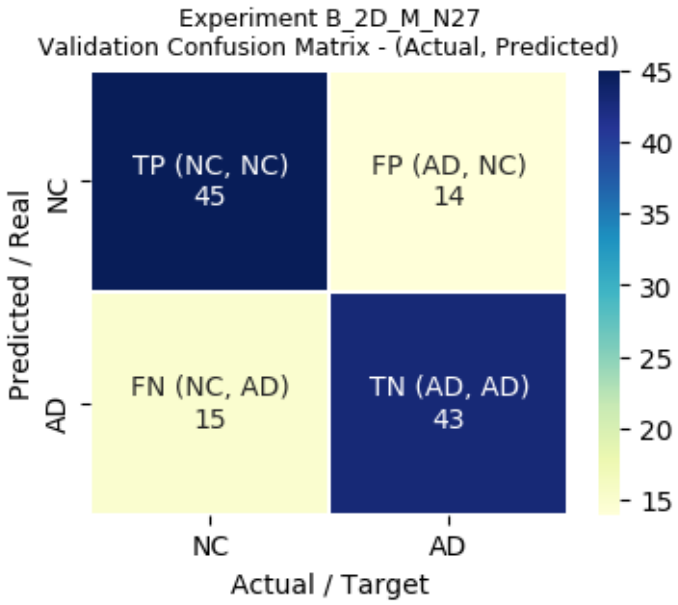


Figure H.259. Caption in the title.

H.2.26.4 Testing

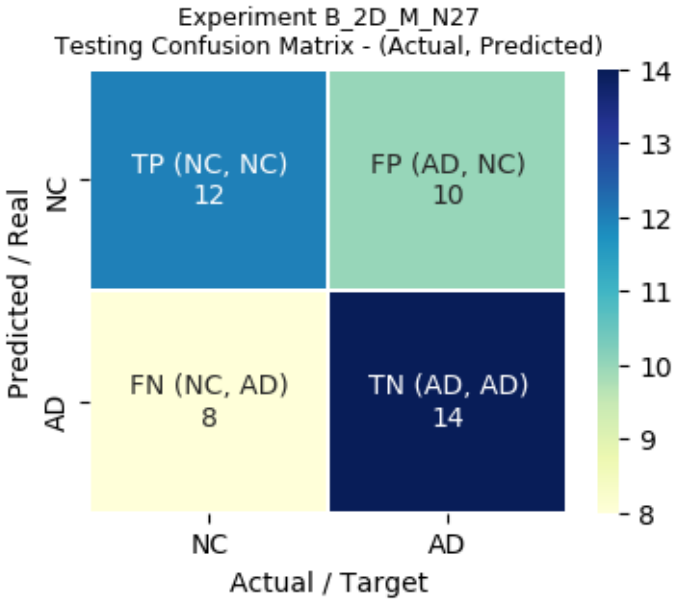


Figure H.260. Caption in the title.

H.2.27 Experiment B_2D_M_N28: C = 1, GNsize = 5, Maximum Epochs = 500

H.2.27.1 General

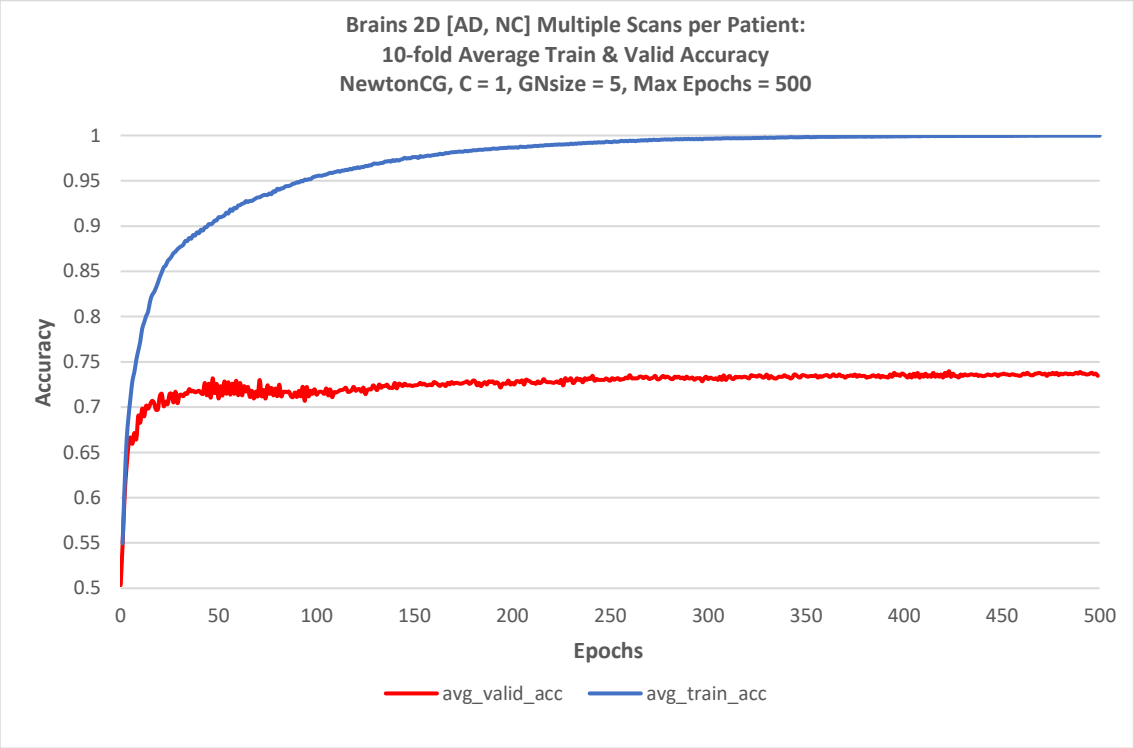


Figure H.261. Caption in the title.

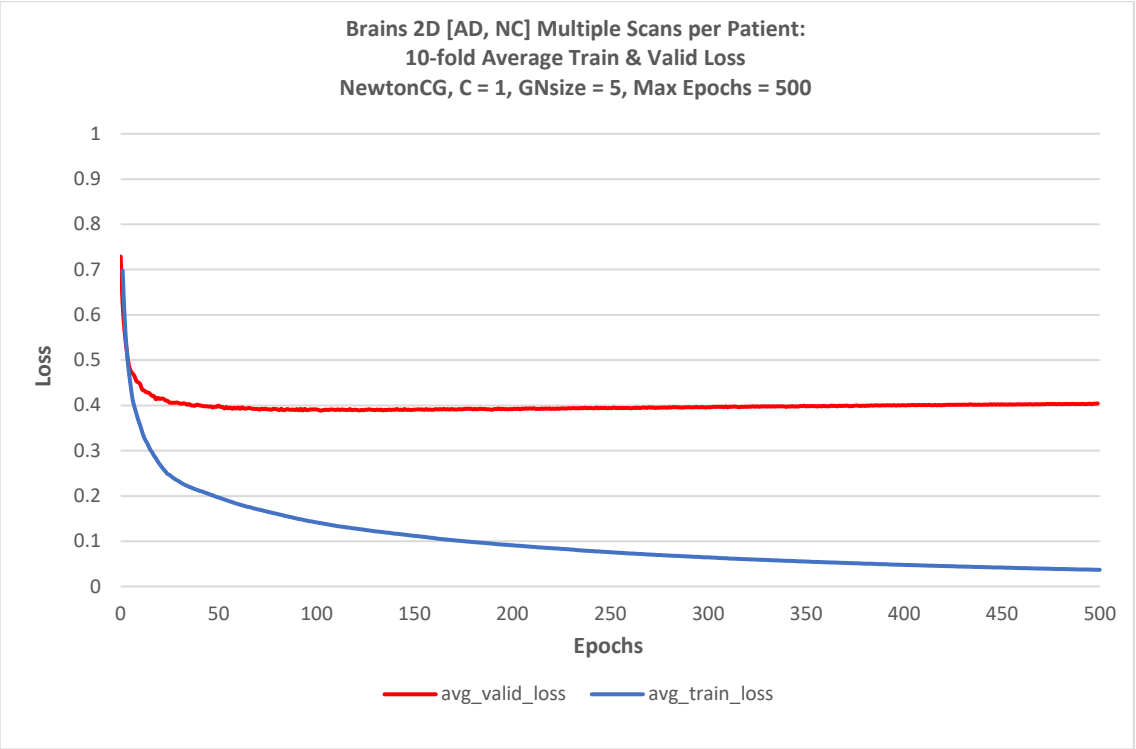


Figure H.262. Caption in the title.

H.2.27.2 Training

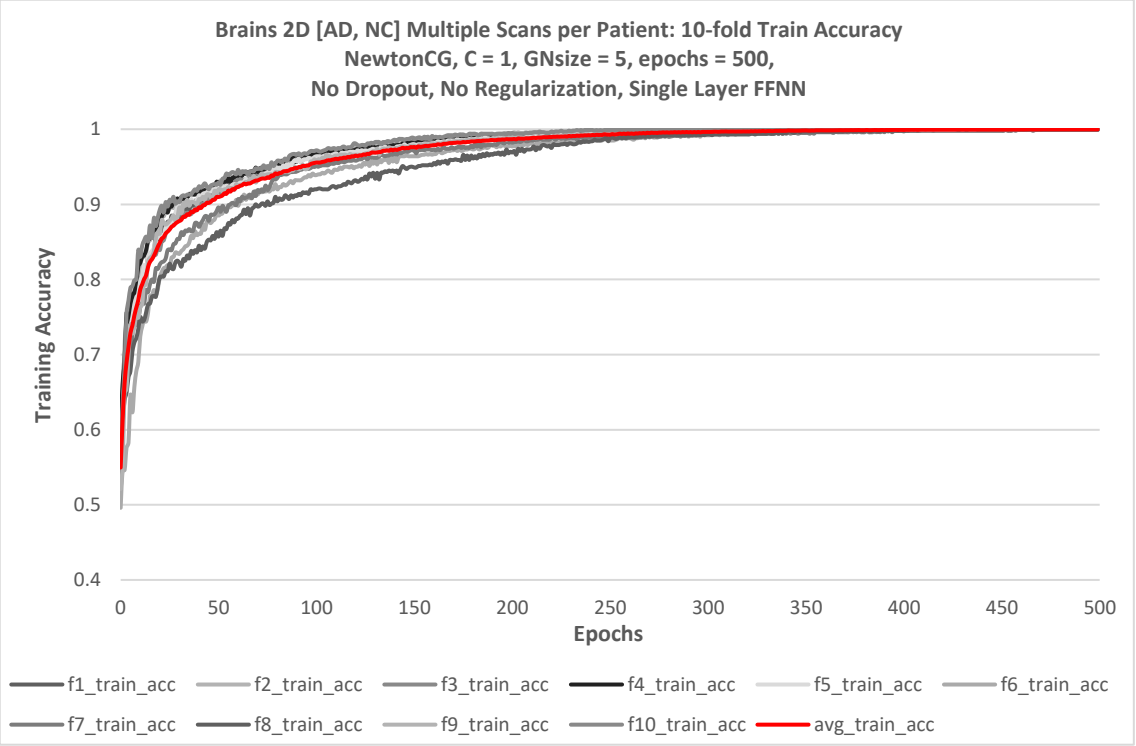


Figure H.263. Caption in the title.

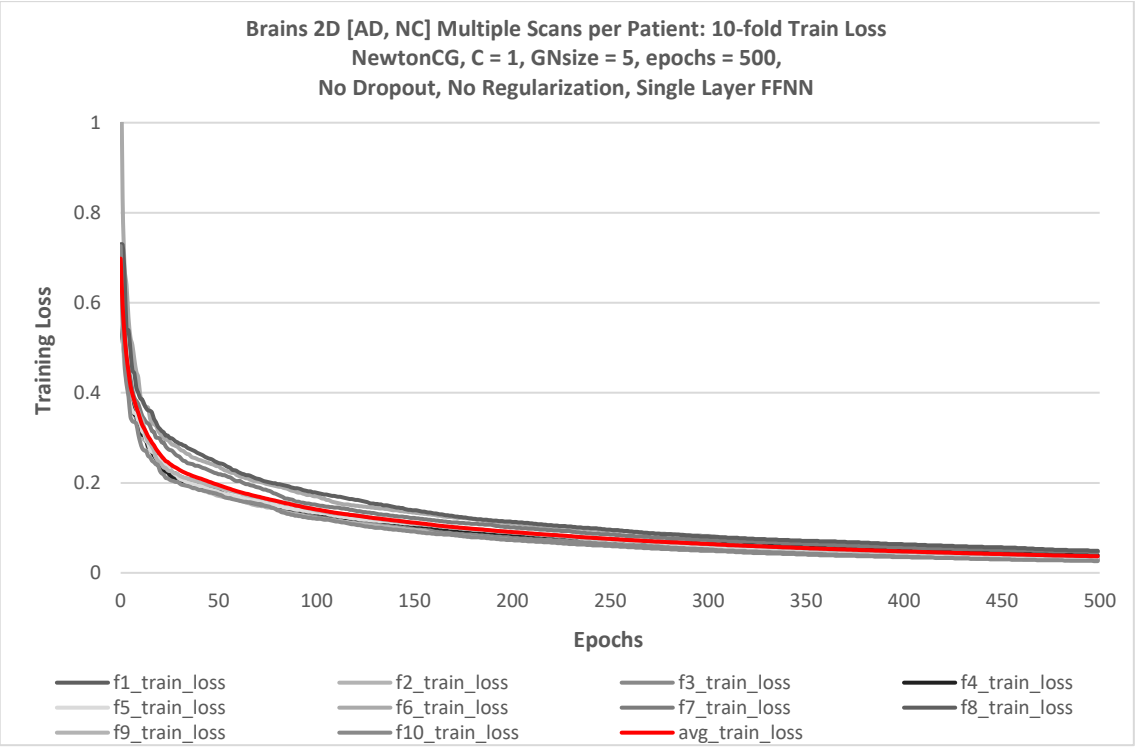


Figure H.264. Caption in the title.

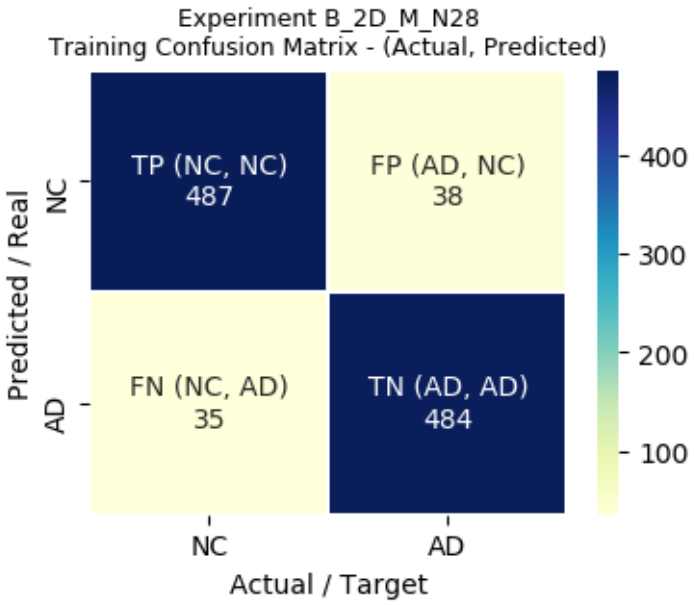


Figure H.265. Caption in the title.

H.2.27.3 Validation

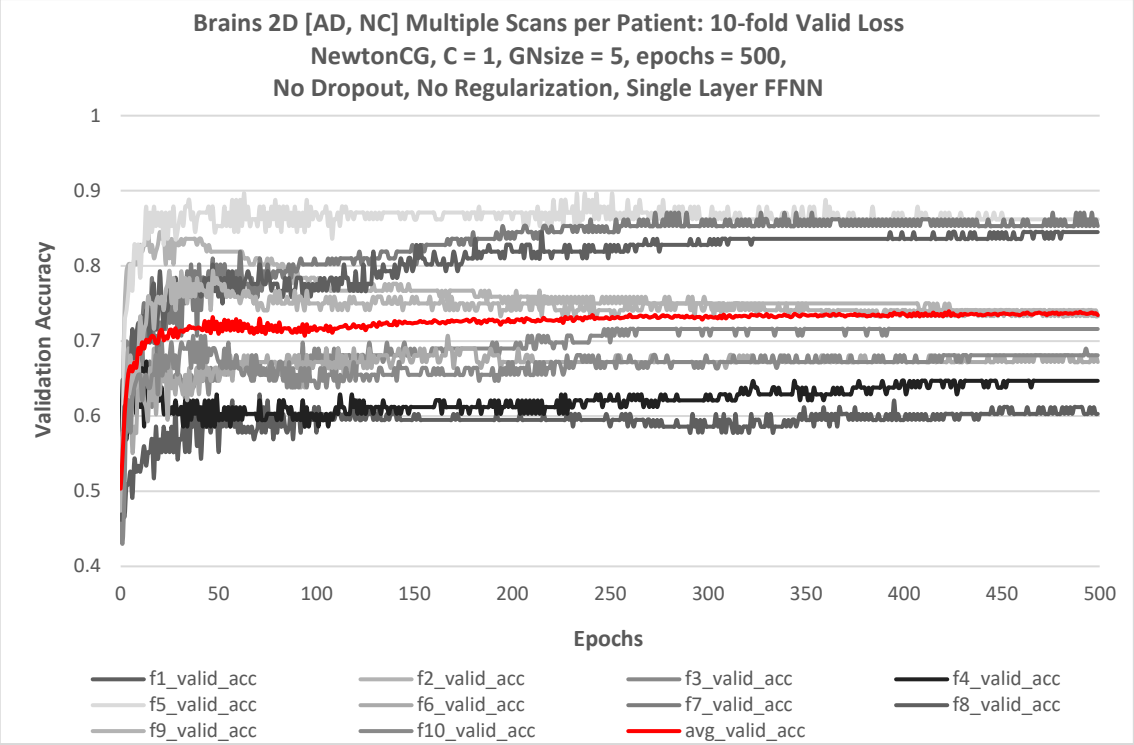


Figure H.266. Caption in the title.

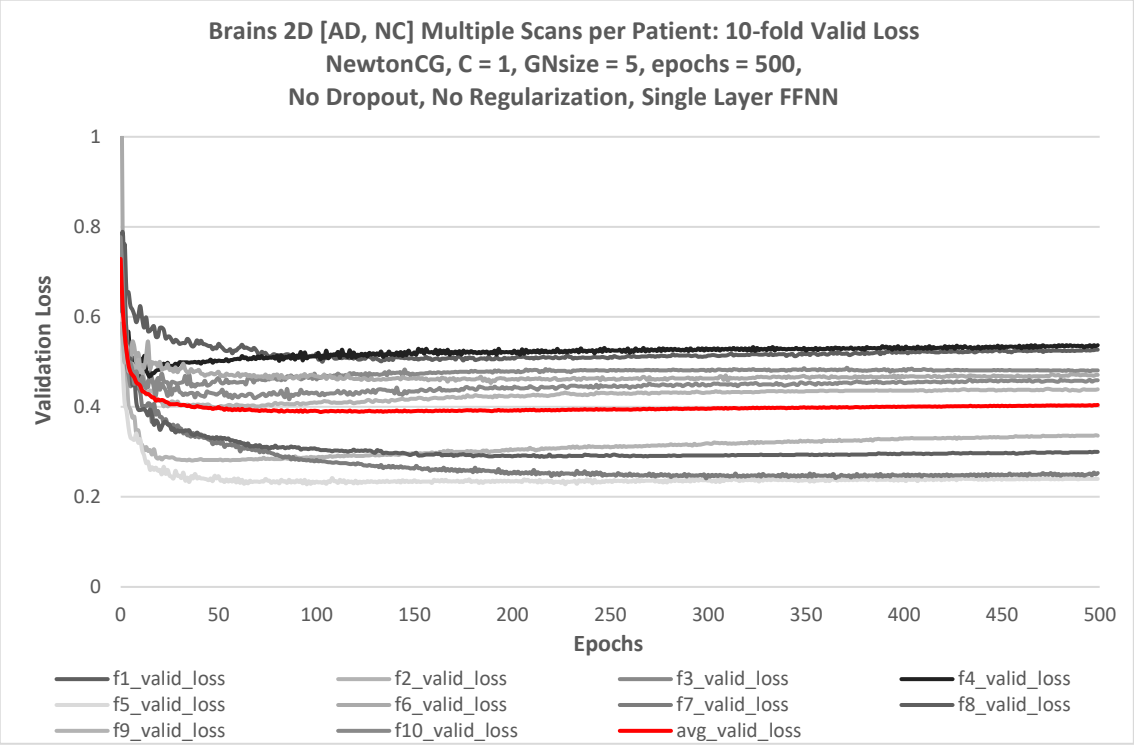


Figure H.267. Caption in the title.

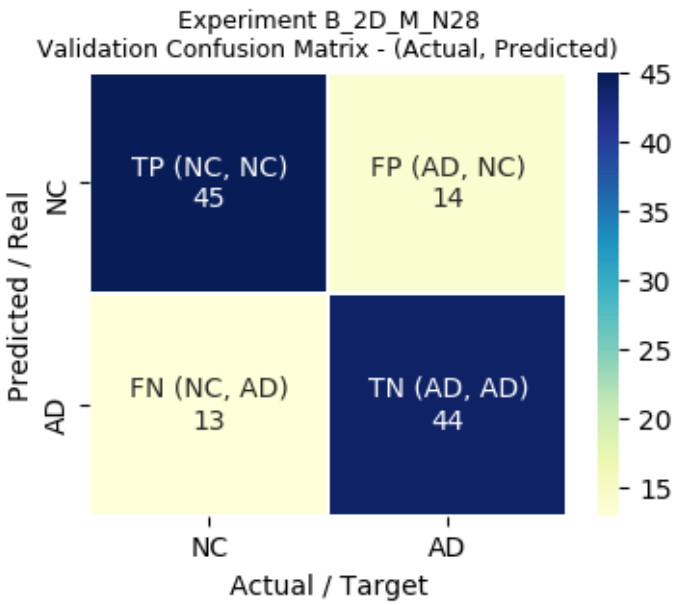


Figure H.268. Caption in the title.

H.2.27.4 Testing

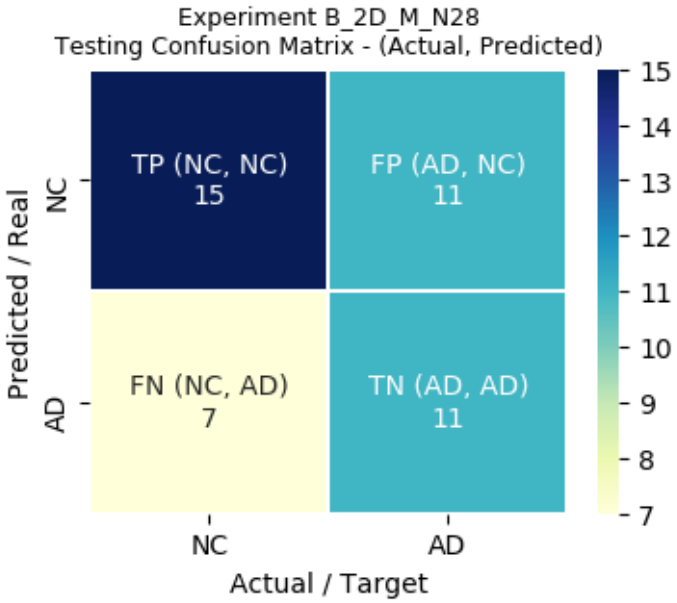


Figure H.269. Caption in the title.

H.3 CNN 7 Layers with NewtonCG

H.3.1 Experiment B_2D_M_N15: C = 0.01, GNsize = 50, Dropout

H.3.1.1 General

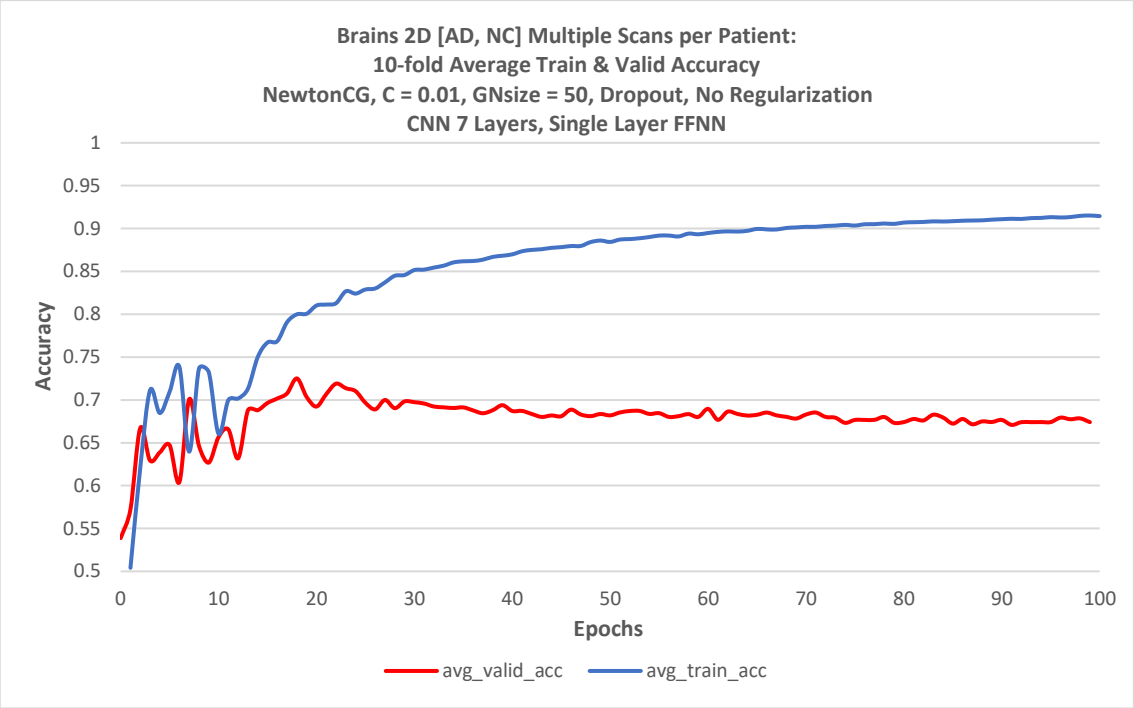


Figure H.270. Caption in the title.

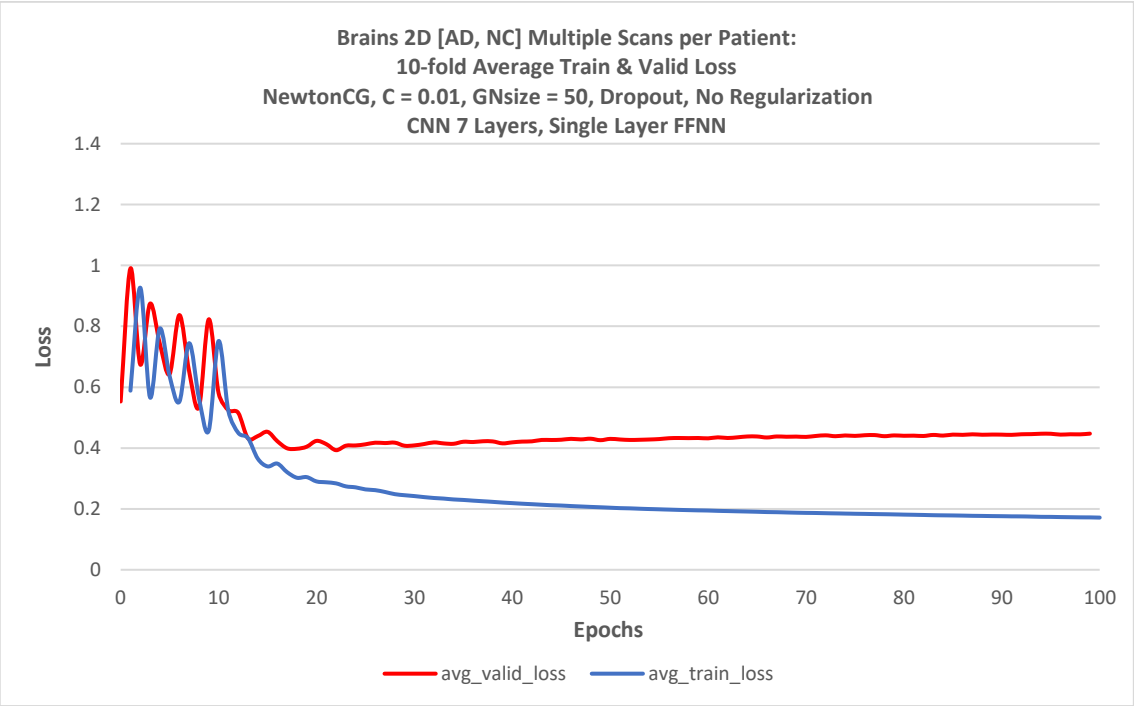


Figure H.271. Caption in the title.

H.3.1.2 Training

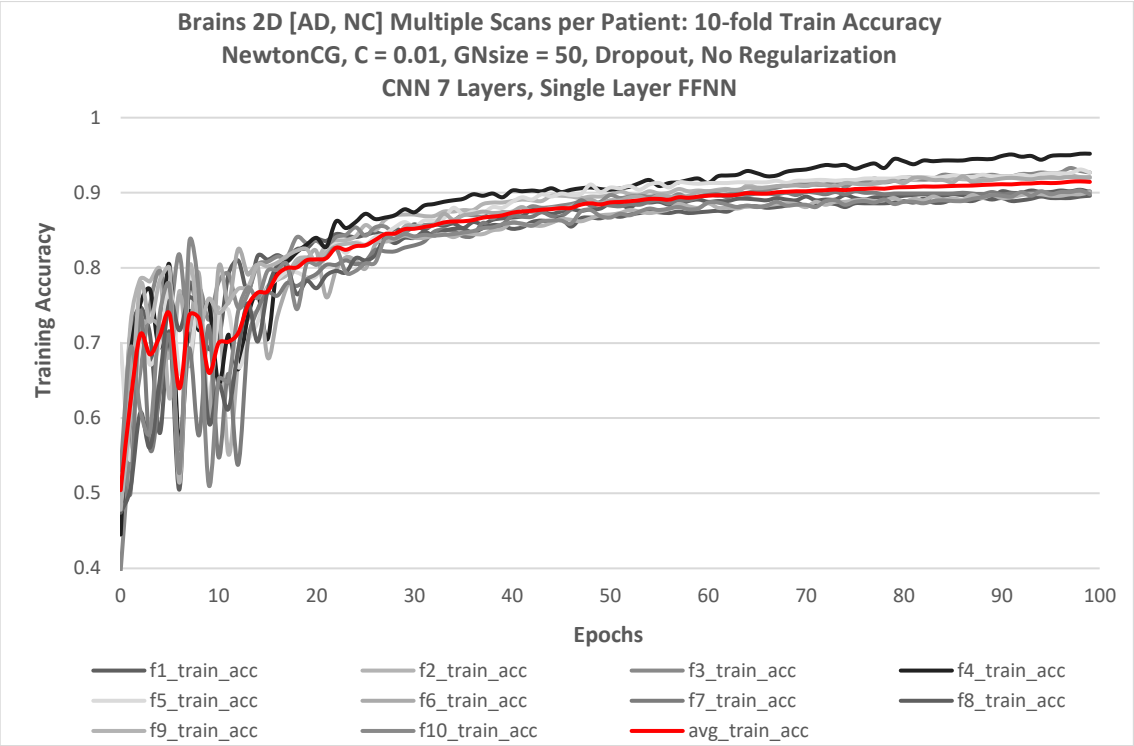


Figure H.272. Caption in the title.

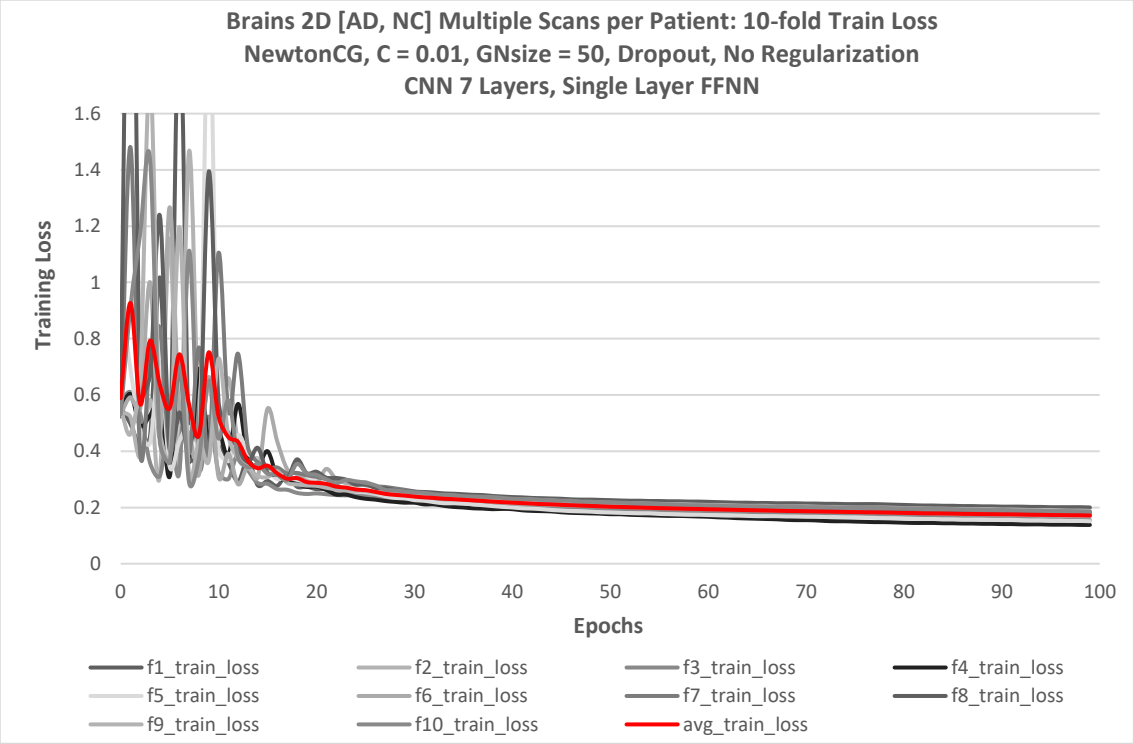


Figure H.273. Caption in the title.

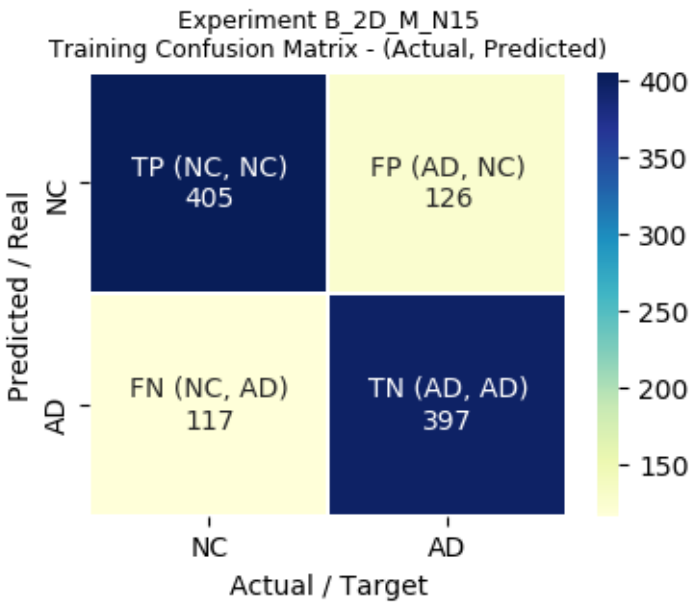


Figure H.274. Caption in the title.

H.3.1.3 Validation

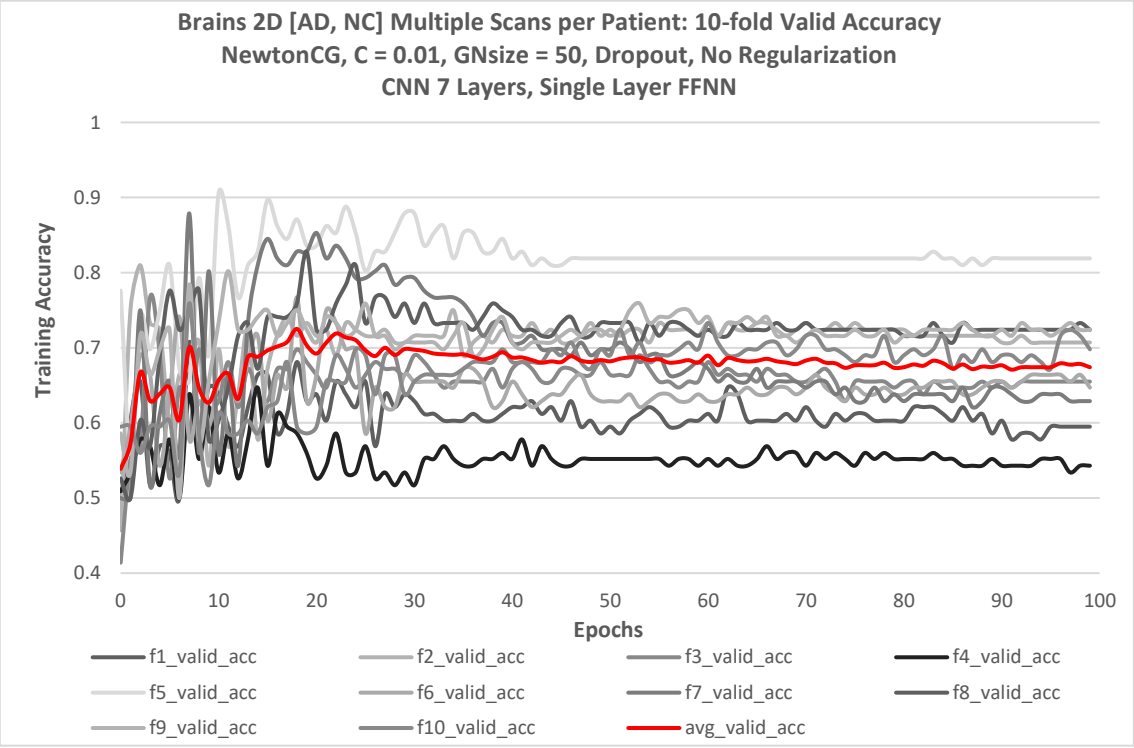


Figure H.275. Caption in the title.

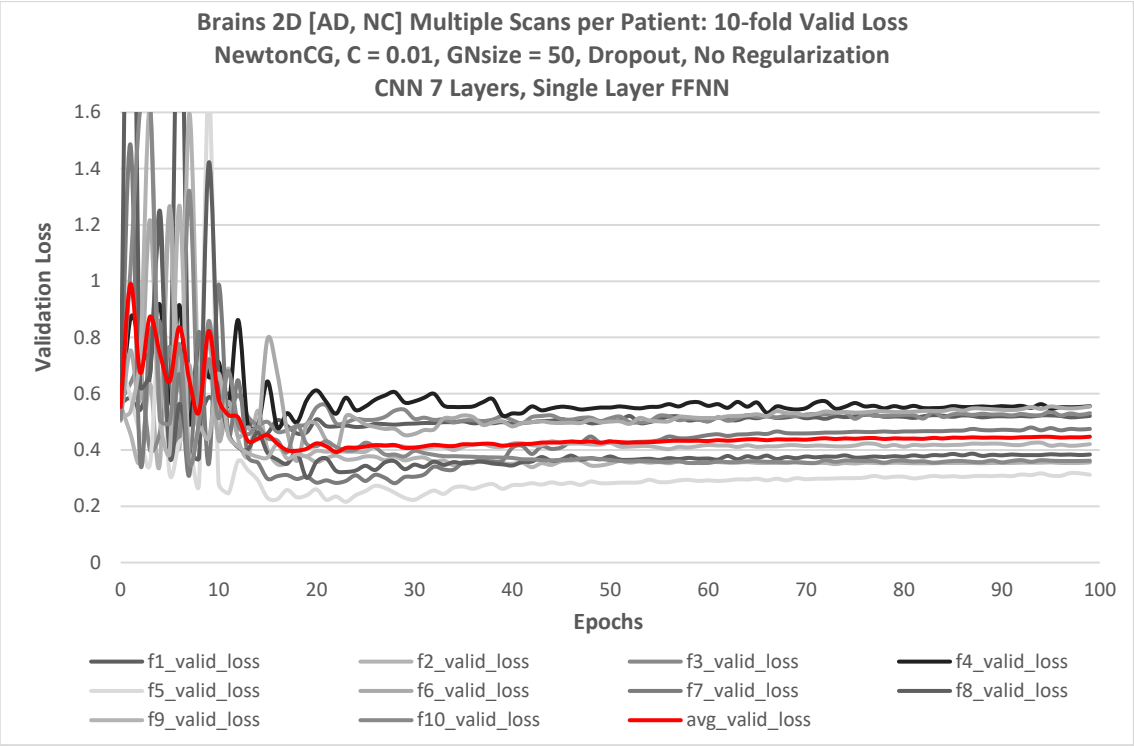


Figure H.276. Caption in the title.

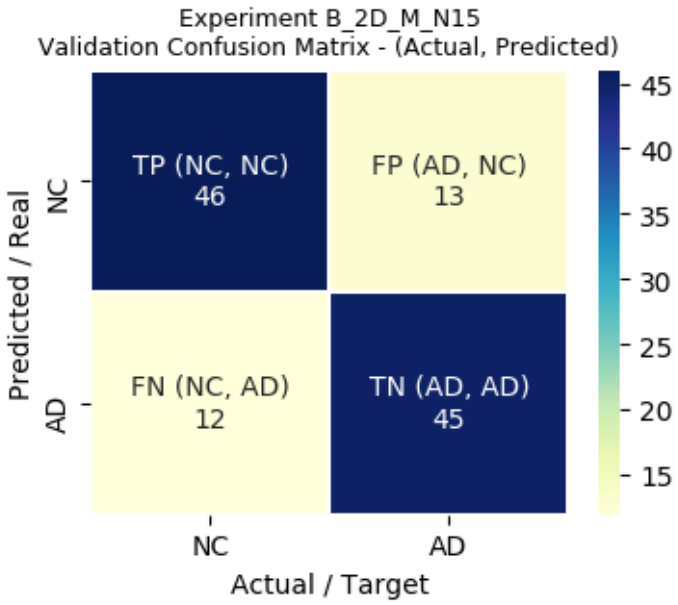


Figure H.277. Caption in the title.

H.3.1.4 Testing

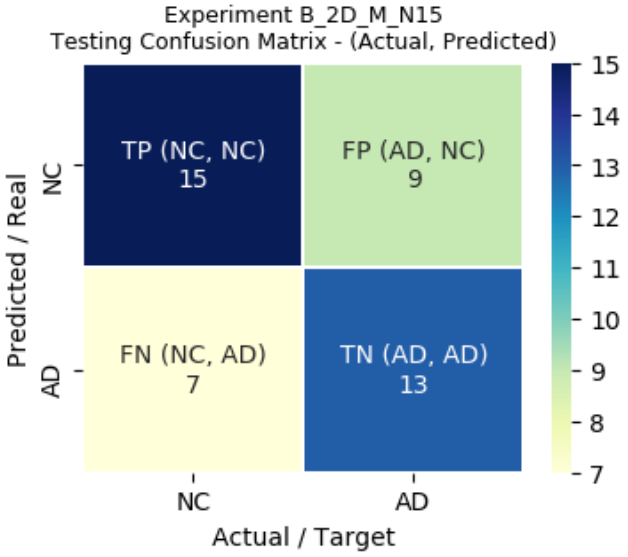


Figure H.278. Caption in the title.

Appendix I

Experiments 2D Brain Slices [AD, NC]: Single Scan per Patient – Single Slice per Scan (174 × 174)

I.1 CNN 4 Layer with NewtonCG

I.1.1 Experiment B_2D_S_N1: C = 1, GNsize = 5, Dropout

I.1.1.1 General

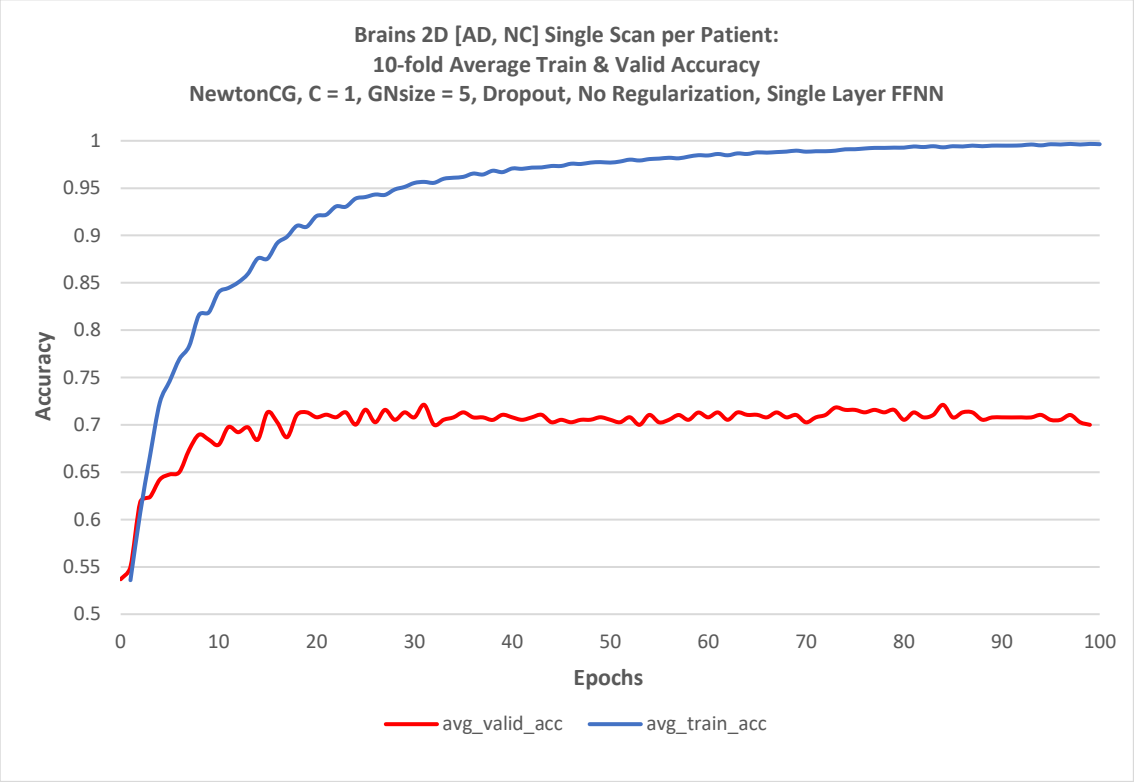


Figure I.1. Caption in the title.

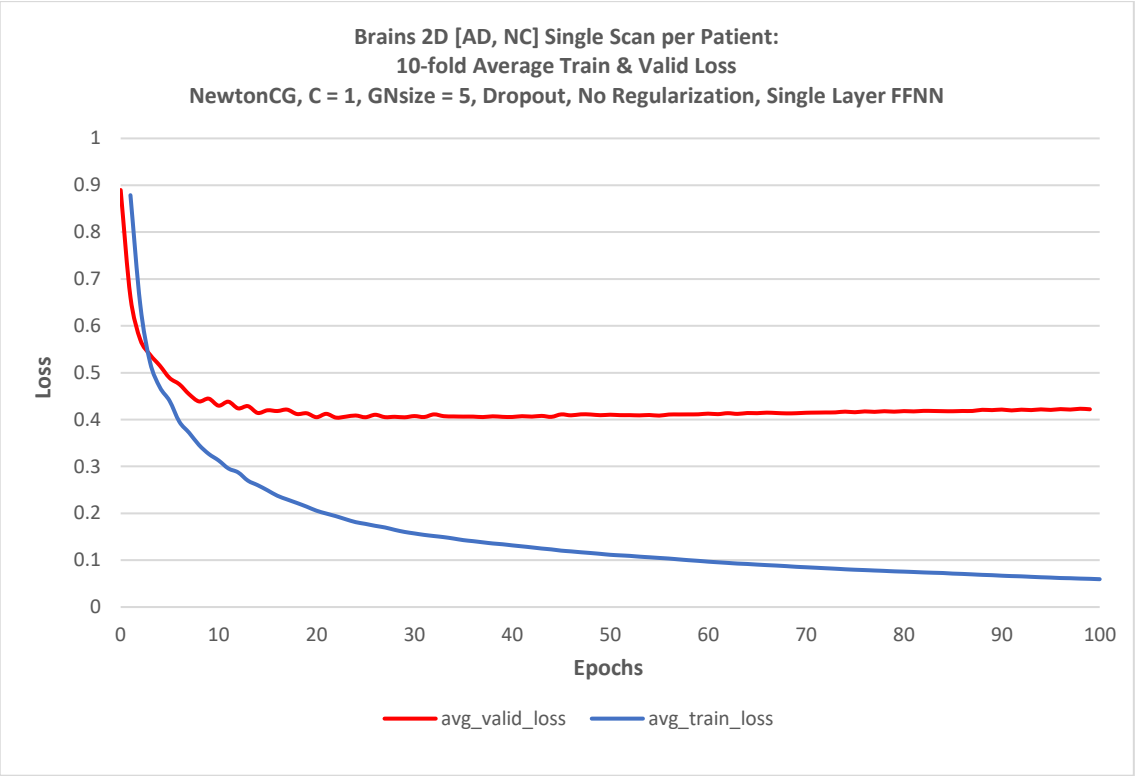


Figure I.2. Caption in the title.

I.1.1.2 Training

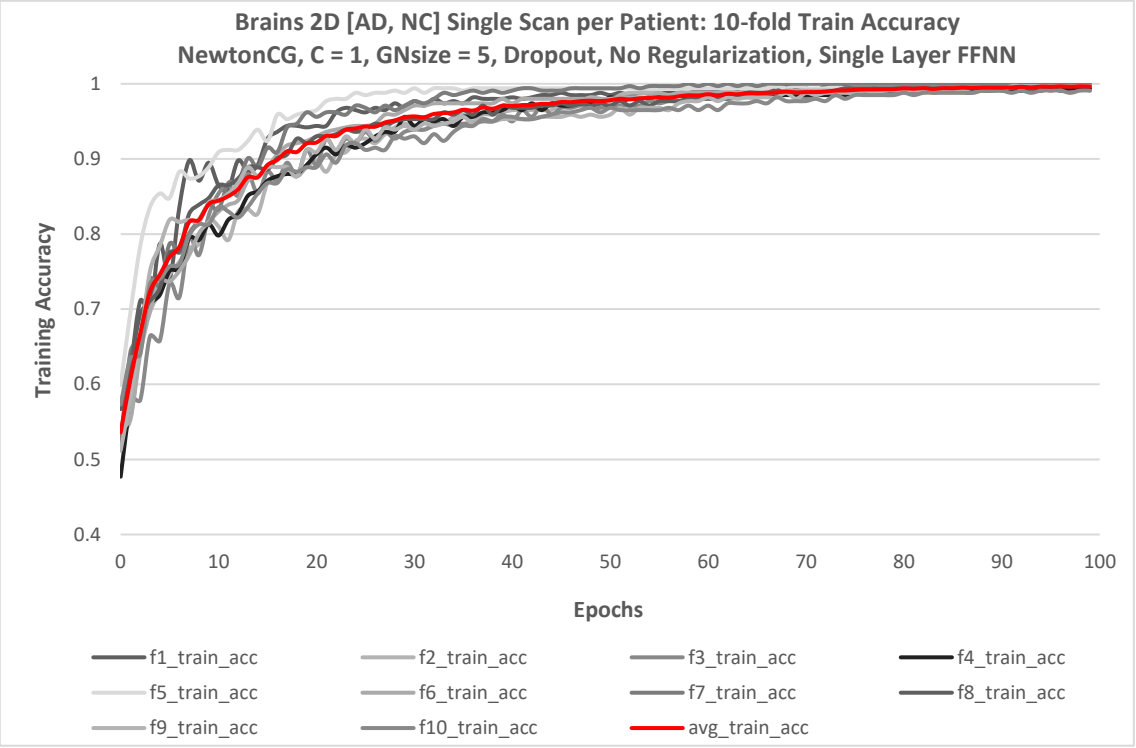


Figure I.3. Caption in the title.

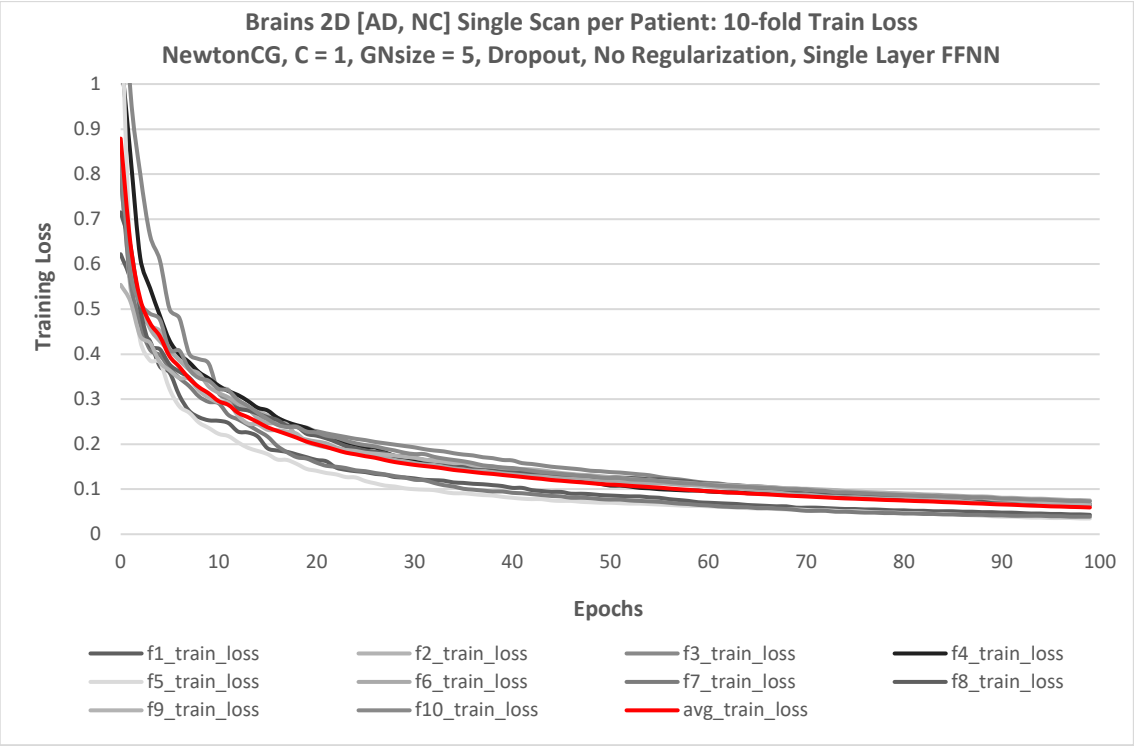


Figure I.4. Caption in the title.

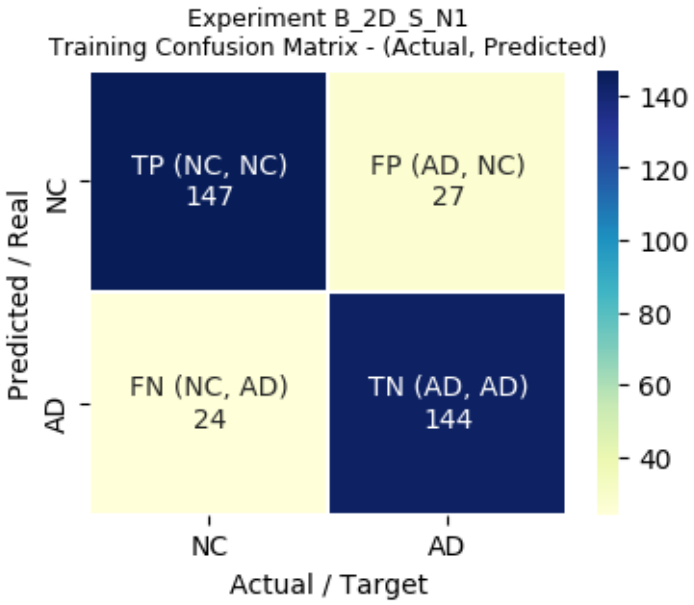


Figure I.5. Caption in the title.

I.1.1.3 Validation

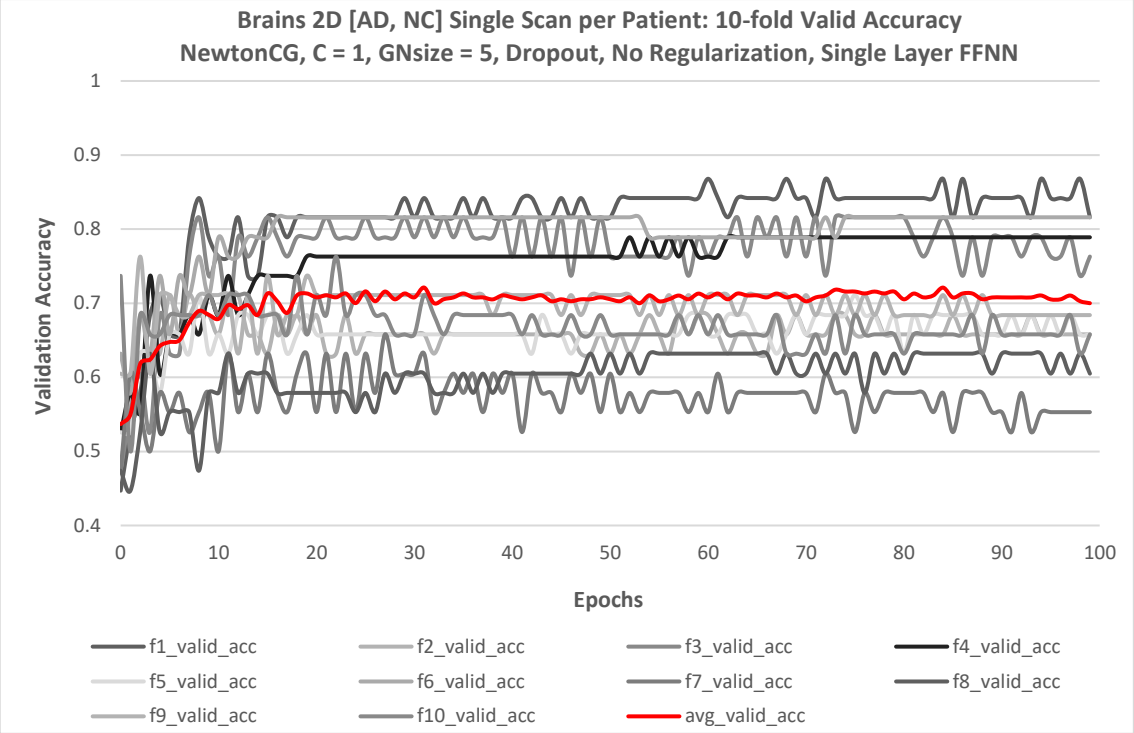


Figure I.6. Caption in the title.

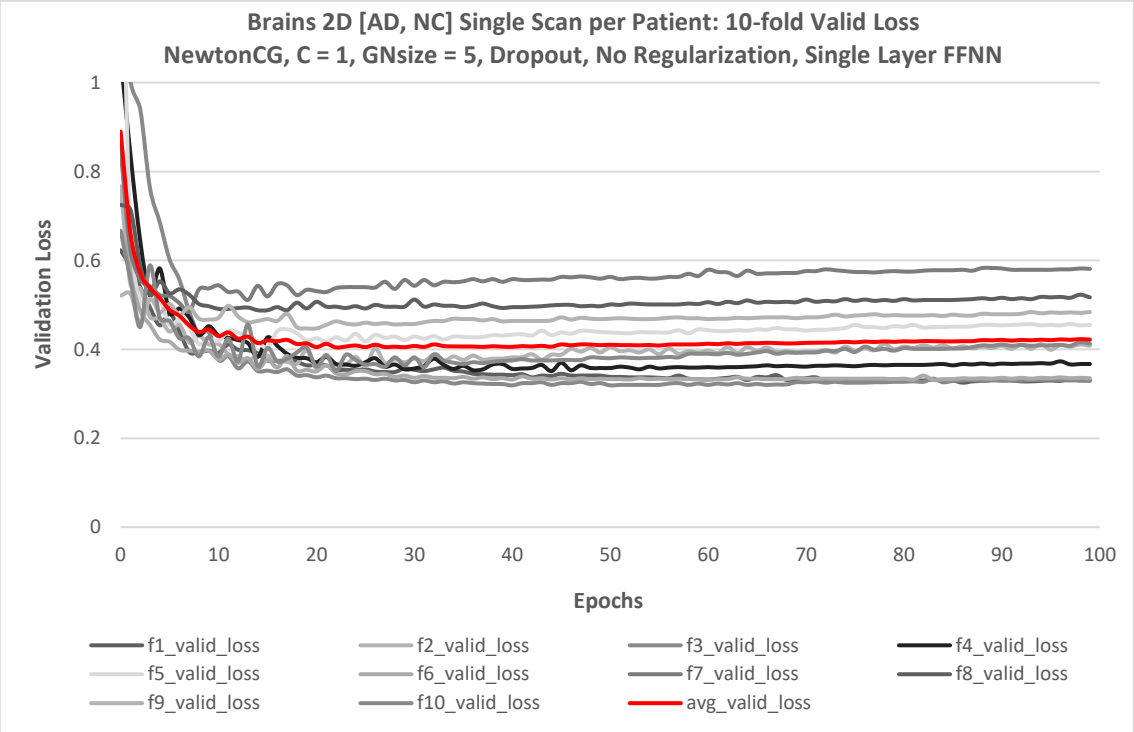


Figure I.7. Caption in the title.

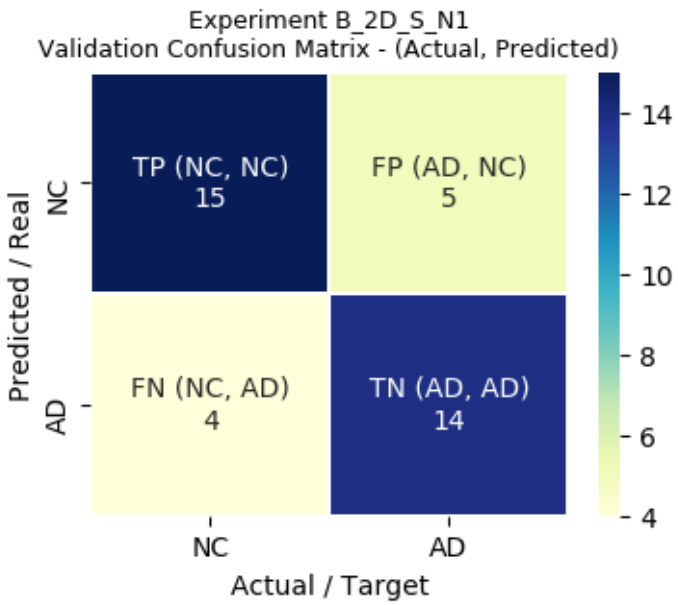


Figure I.8. Caption in the title.

I.1.1.4 Testing

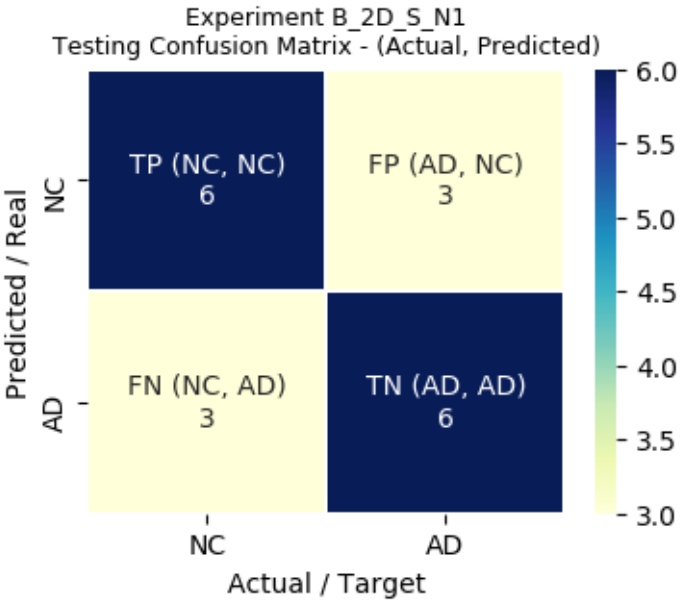


Figure I.9. Caption in the title.

I.1.2 Experiment B_2D_S_N2: C = 1, GNsize = 5, Dropout, L2 Regularization, Simple CNN Architecture

I.1.2.1 General

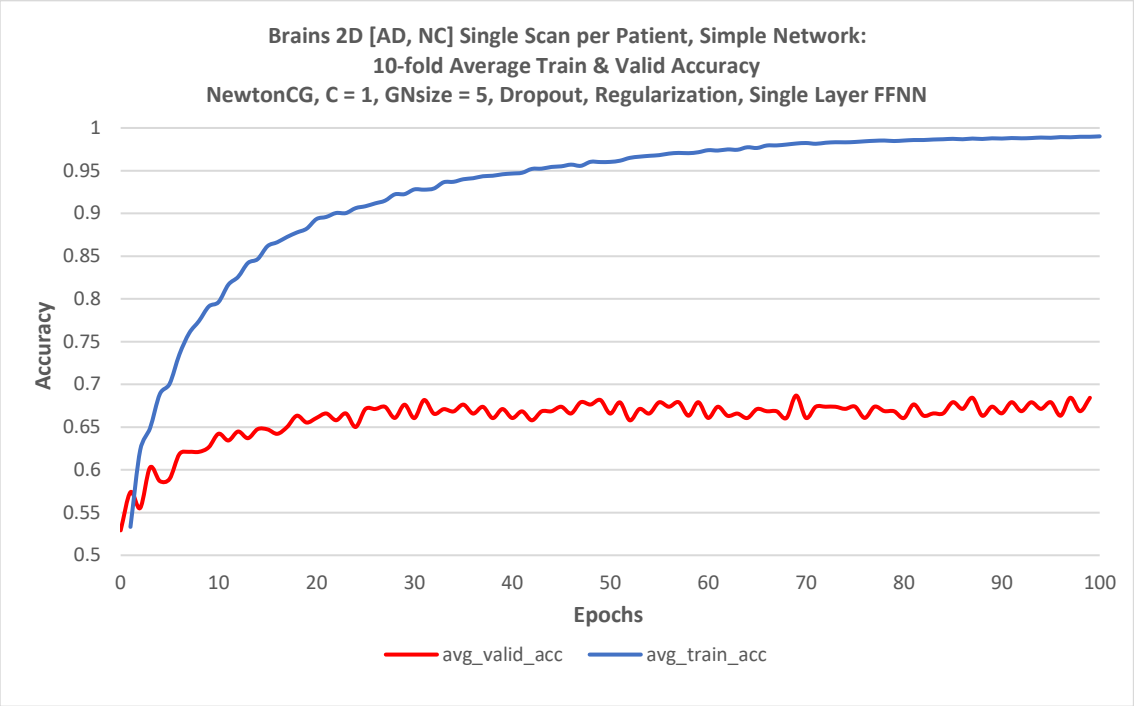


Figure I.10. Caption in the title.

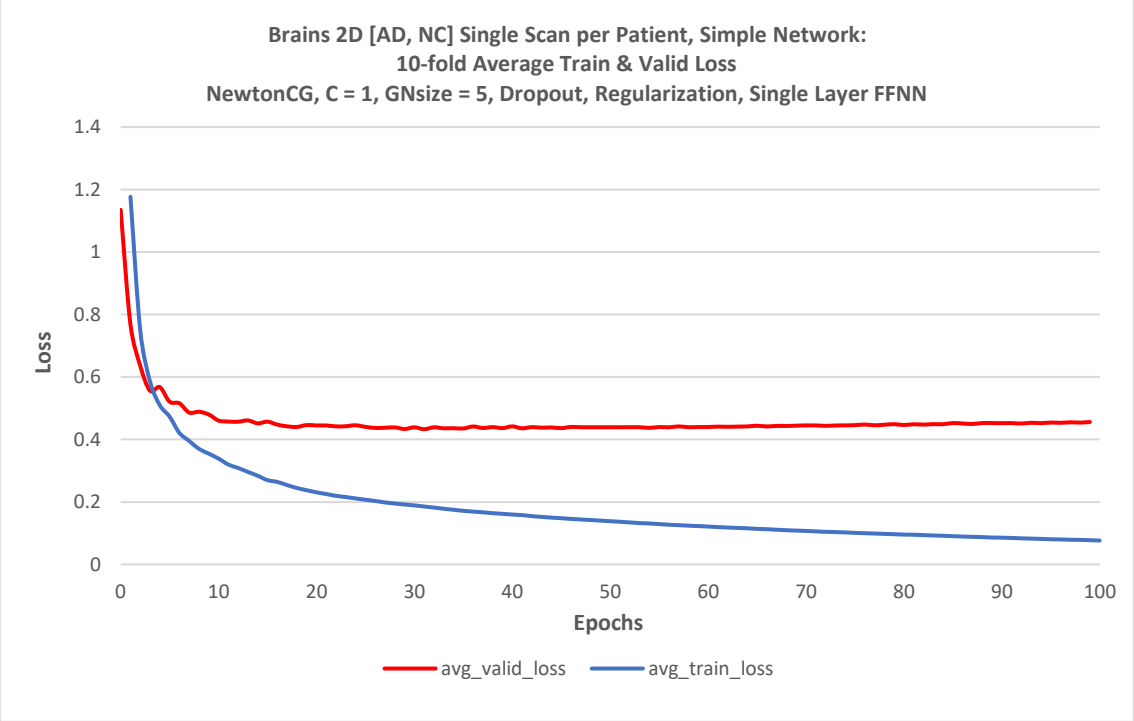


Figure I.11. Caption in the title.

I.1.2.2 Training

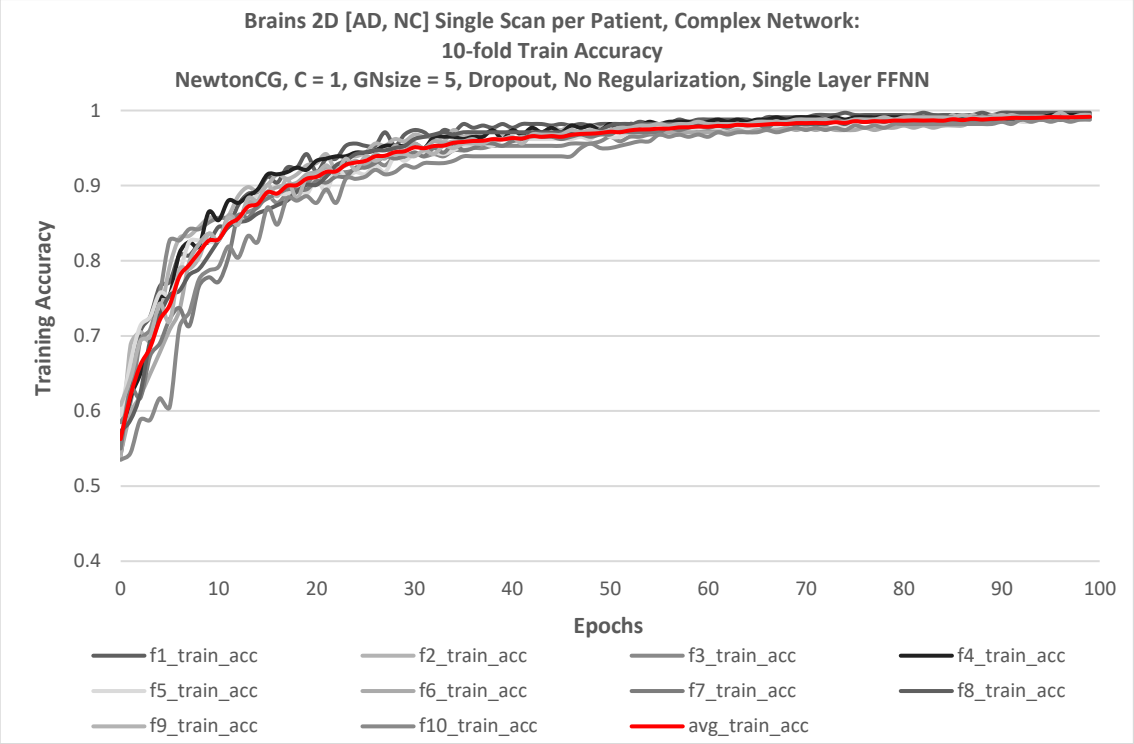


Figure I.12. Caption in the title.

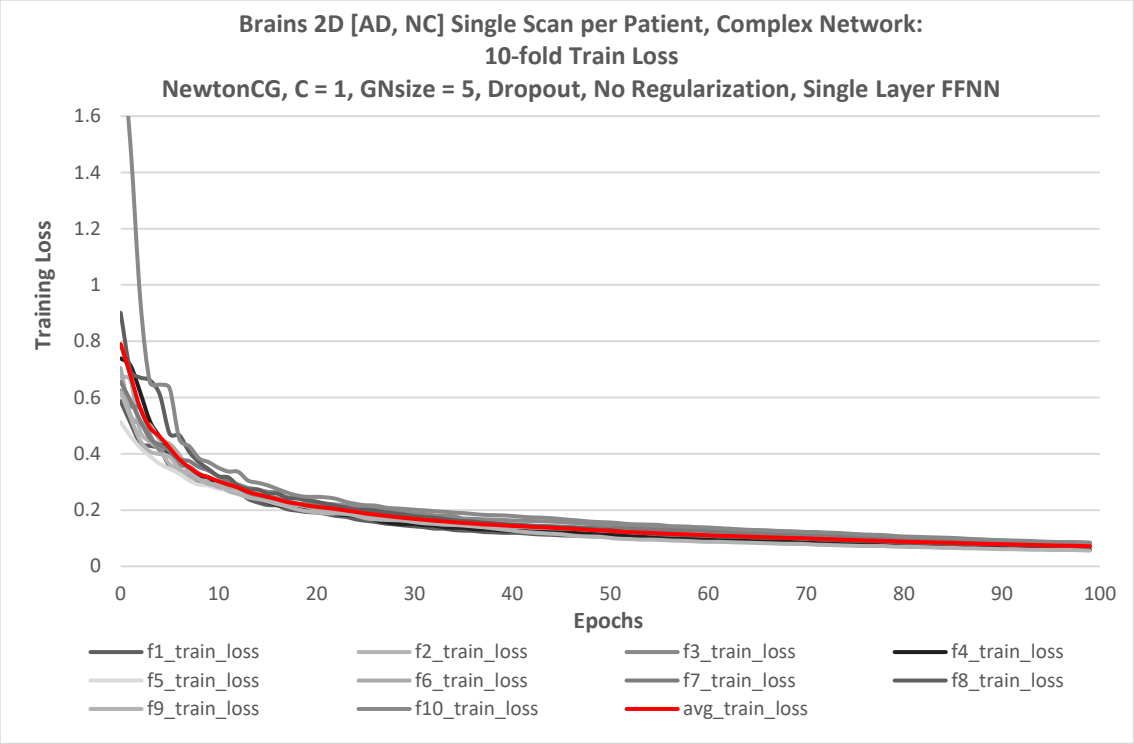


Figure I.13. Caption in the title.

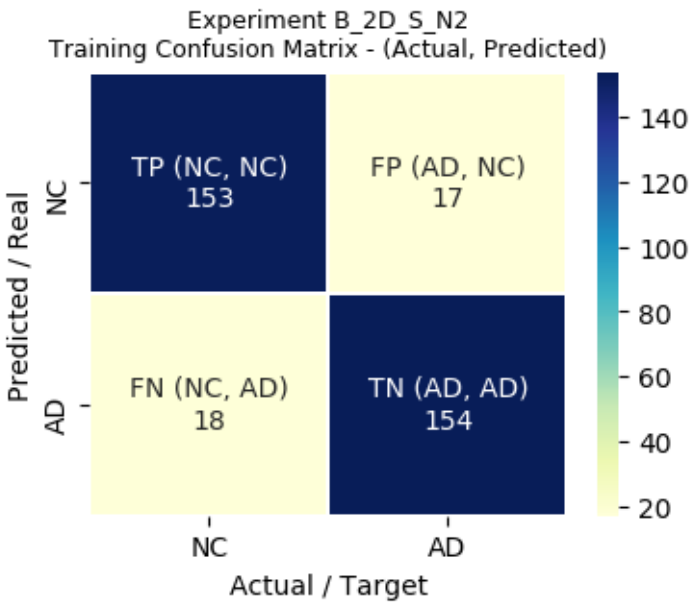


Figure I.14. Caption in the title.

I.1.2.3 Validation

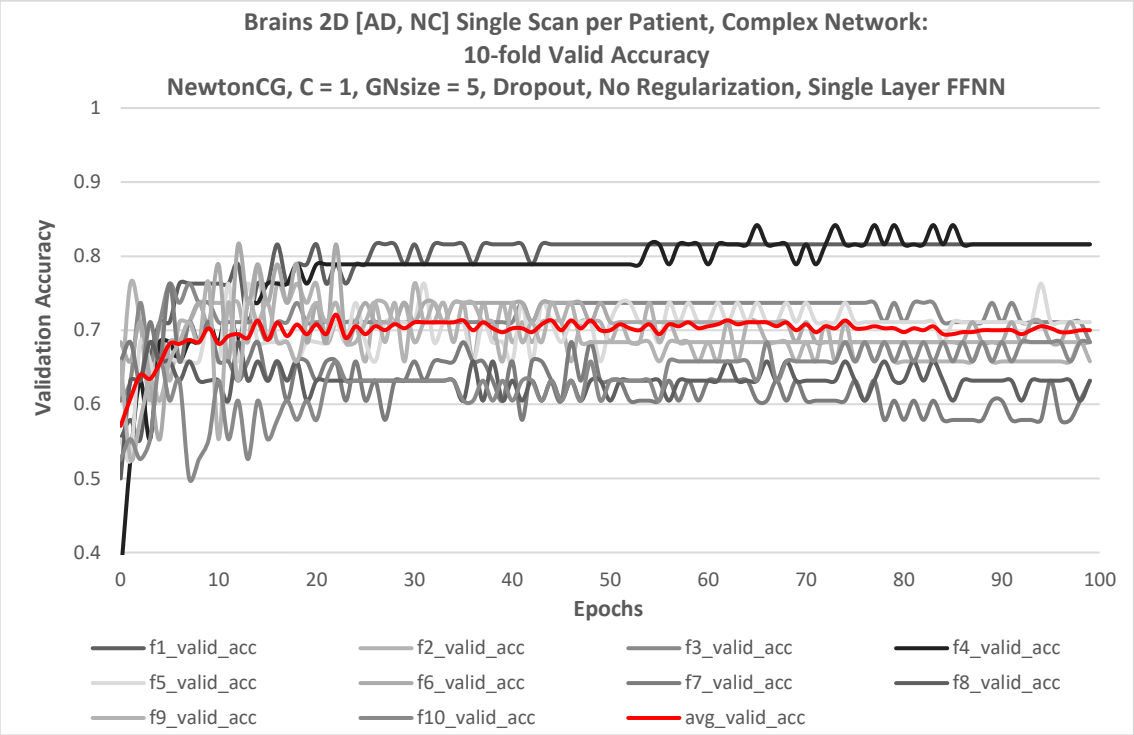


Figure I.15. Caption in the title.

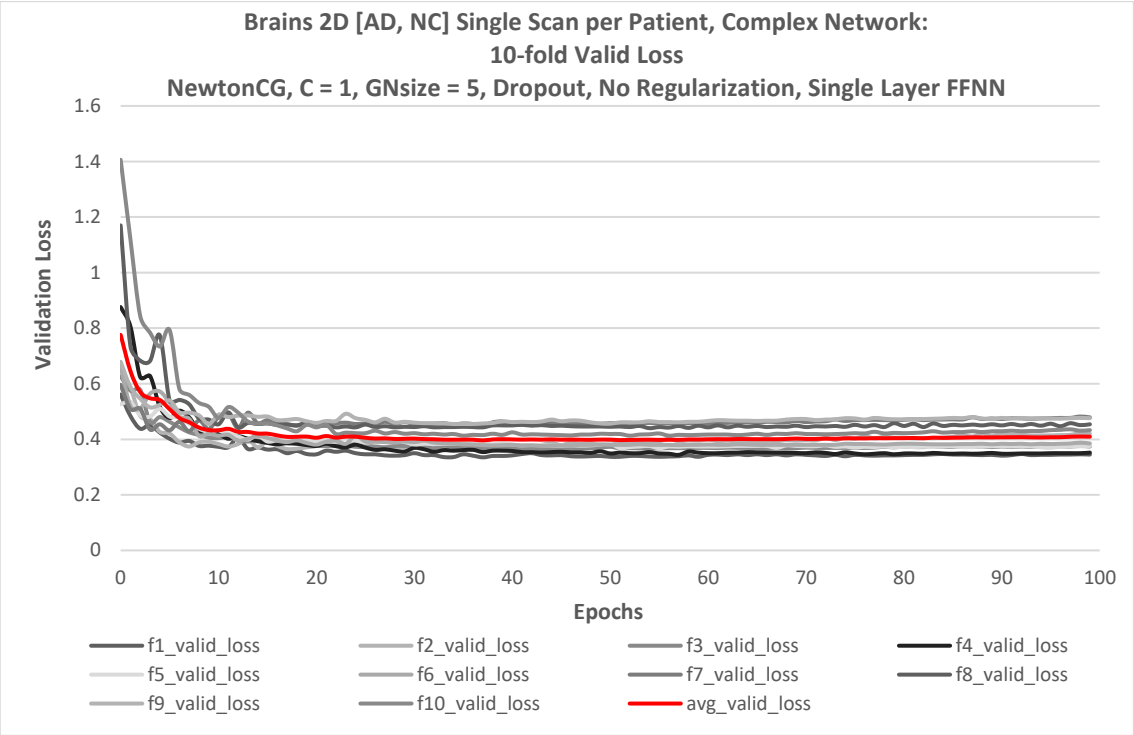


Figure I.16. Caption in the title.

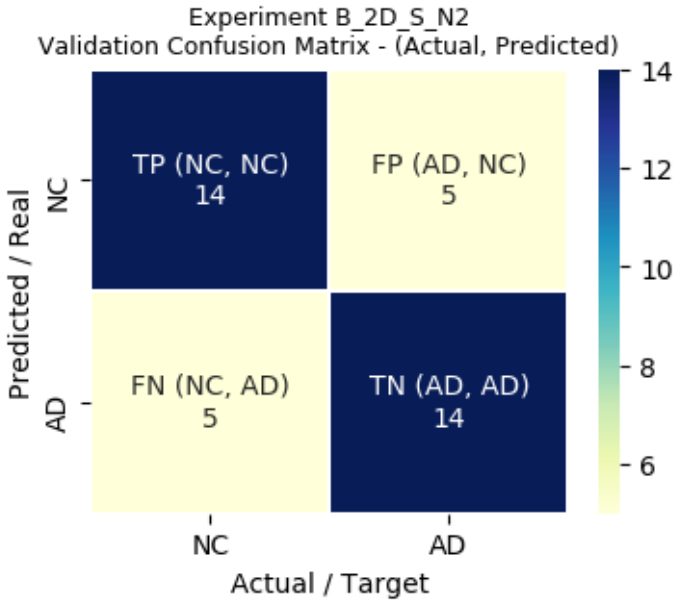


Figure I.17. Caption in the title.

I.1.2.4 Testing

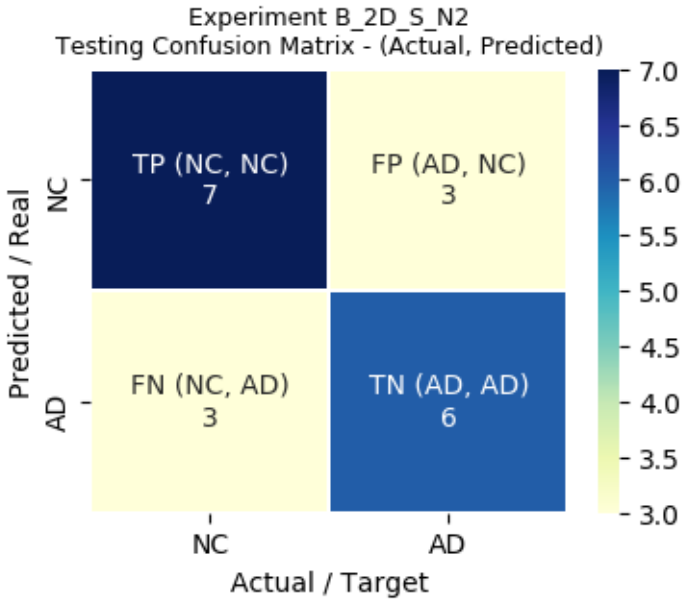


Figure I.18. Caption in the title.

I.1.3 Experiment B_2D_S_N3: C = 1, GNsize = 5, Dropout, Complex CNN Architecture

I.1.3.1 General

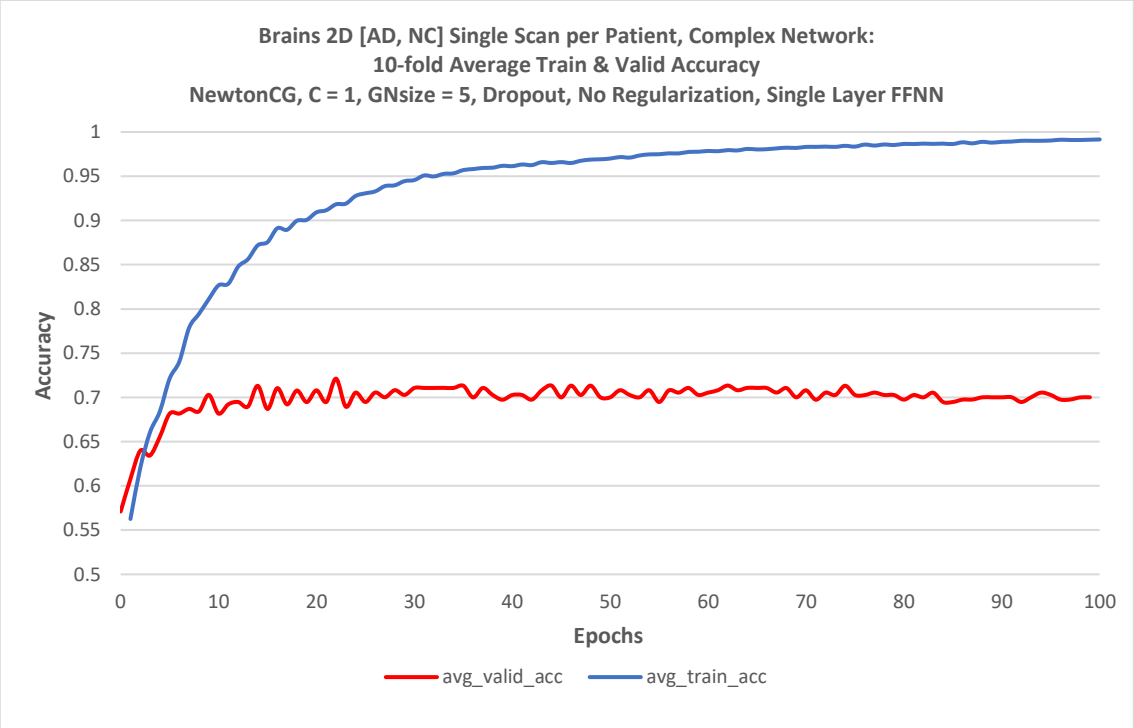


Figure I.19. Caption in the title.

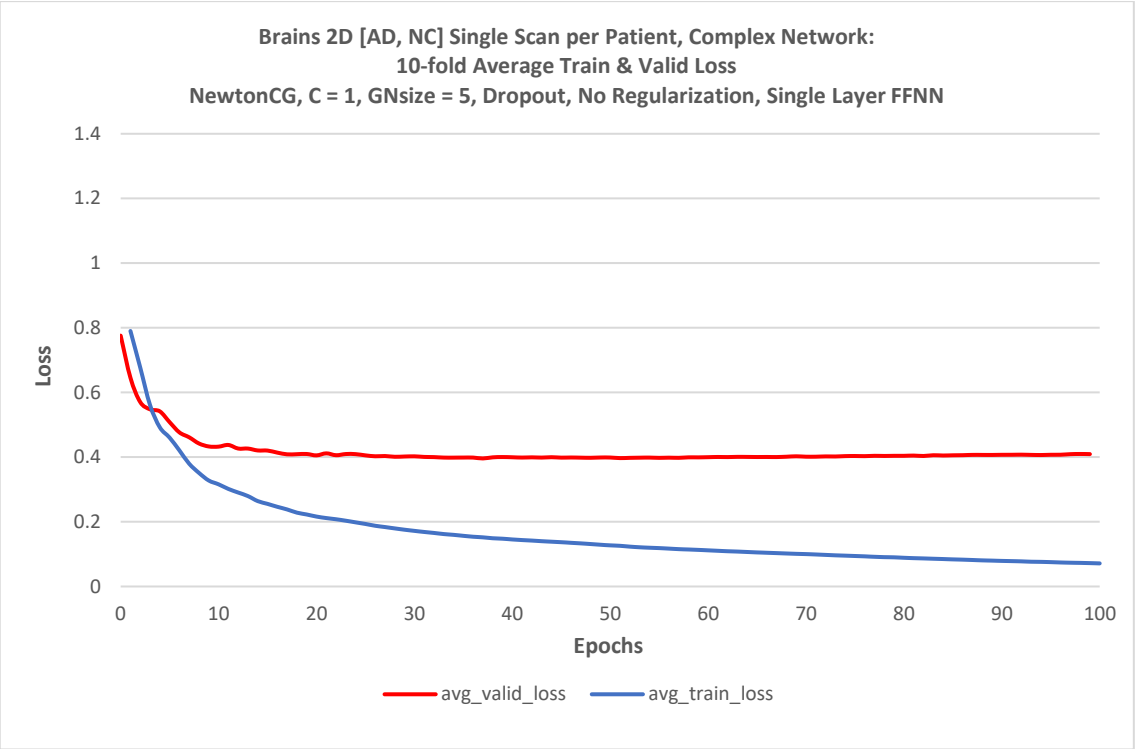


Figure I.20. Caption in the title.

I.1.3.2 Training

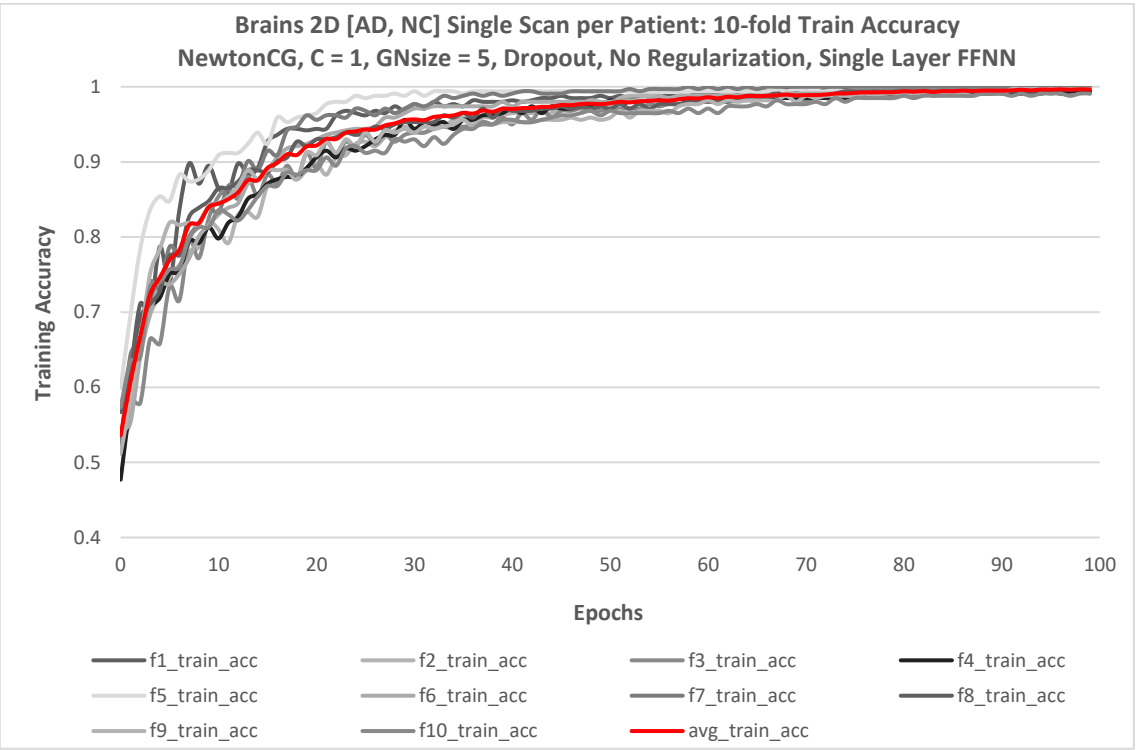


Figure I.21. Caption in the title.

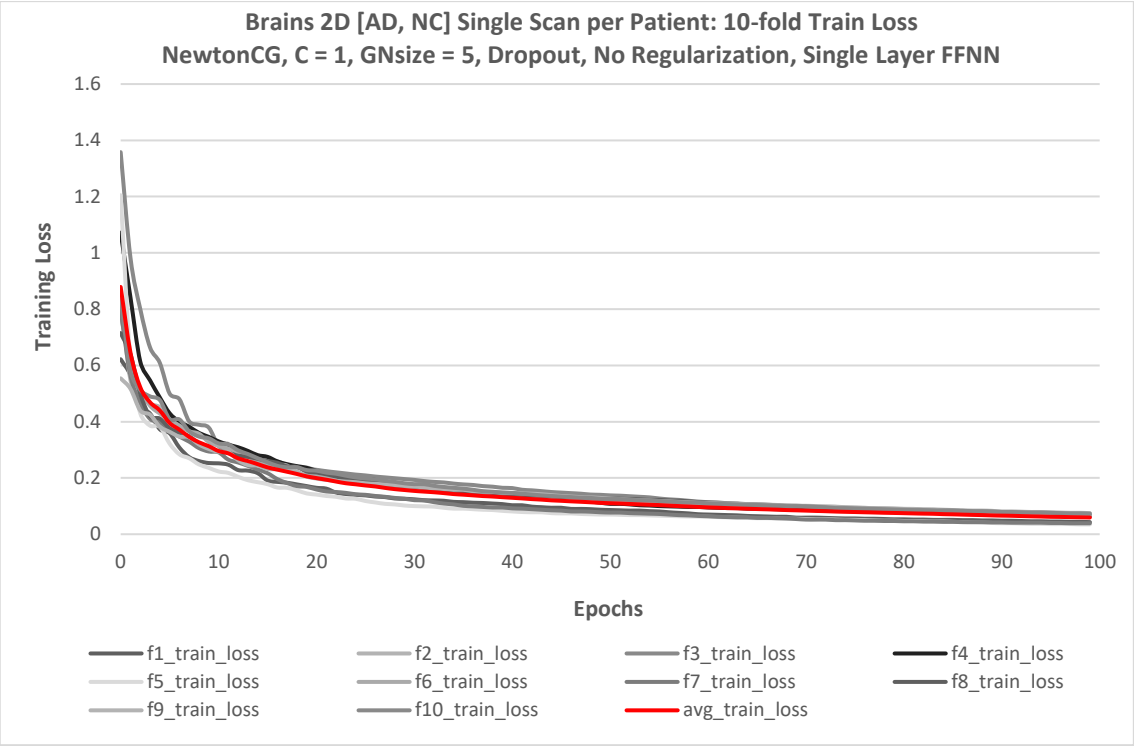


Figure I.22. Caption in the title.

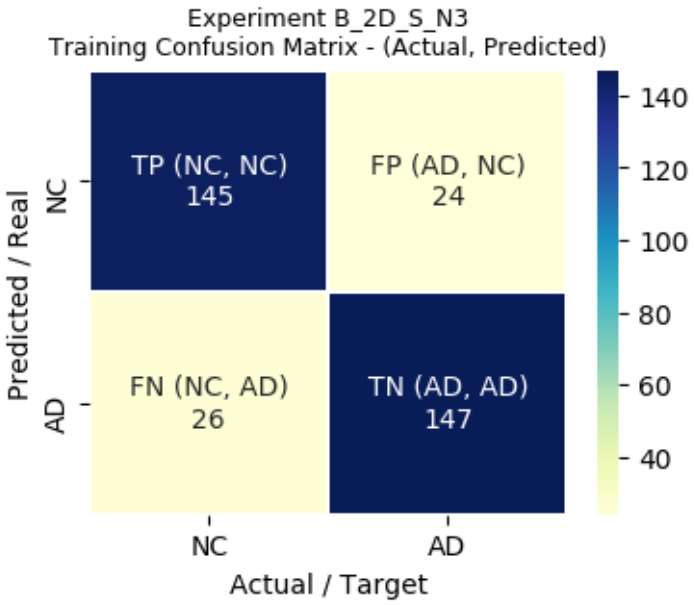


Figure I.23. Caption in the title.

I.1.3.3 Validation

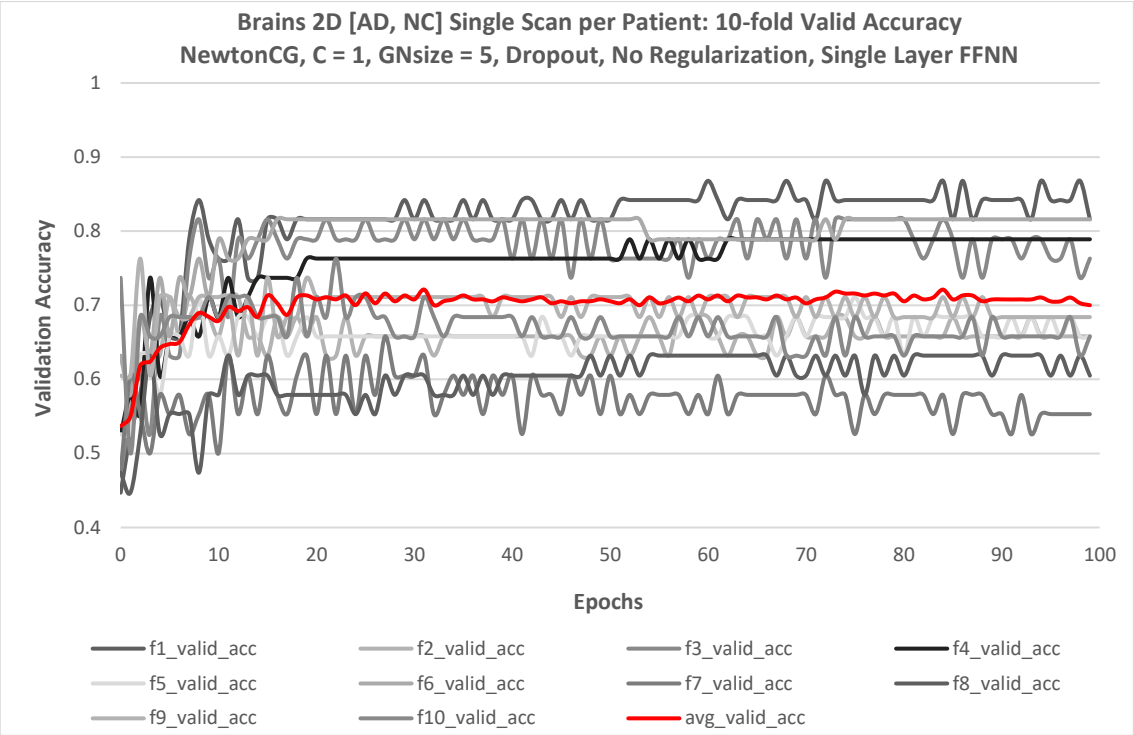


Figure I.24. Caption in the title.

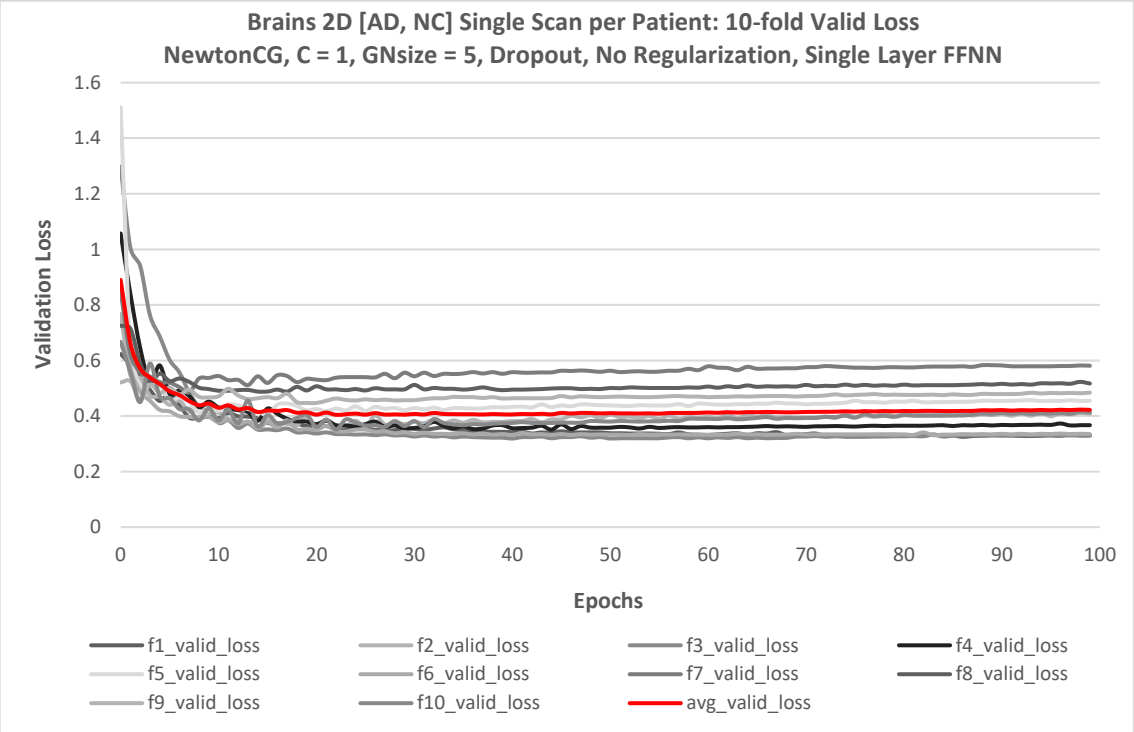


Figure I.25. Caption in the title.

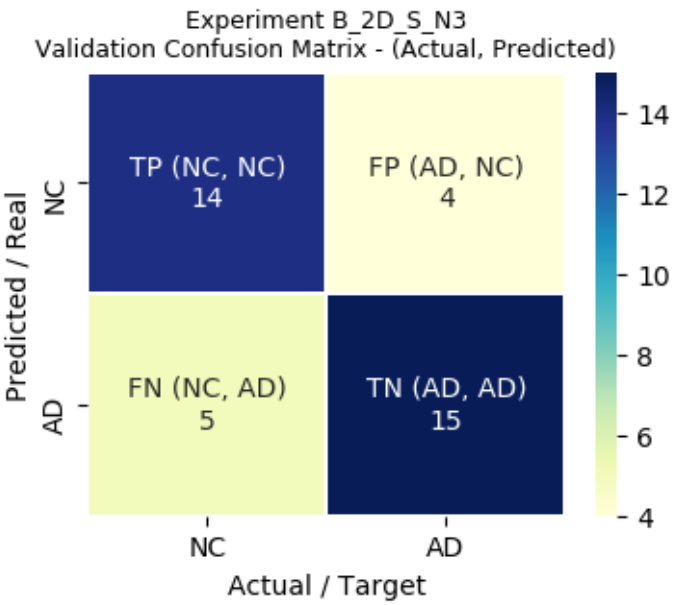


Figure I.26. Caption in the title.

I.1.3.4 Testing

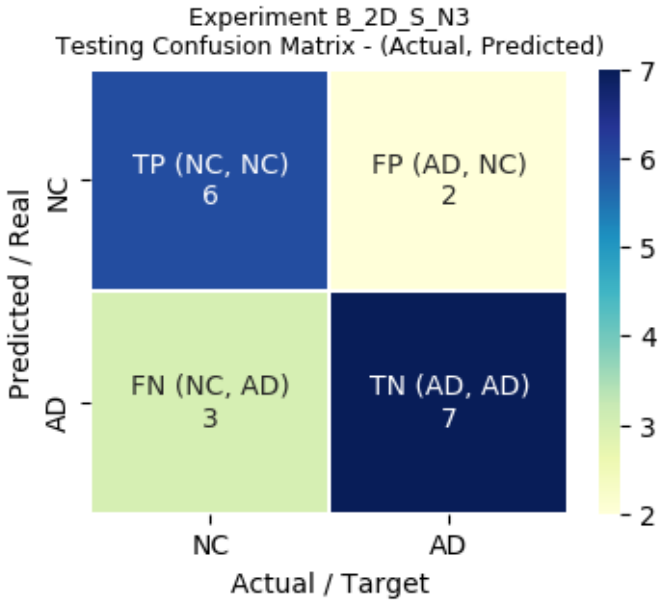


Figure I.27. Caption in the title.

Appendix J

Experiments 2D Brain Slices [AD, NC]: Multiple Scans per Patient – 7 Slices per Scan (174 × 174)

J.1 CNN 4 Layers with Adam

J.1.1 Experiment B_2D_7M_A1: C = 0.01, lr = 0.01, Dropout

J.1.1.1 General

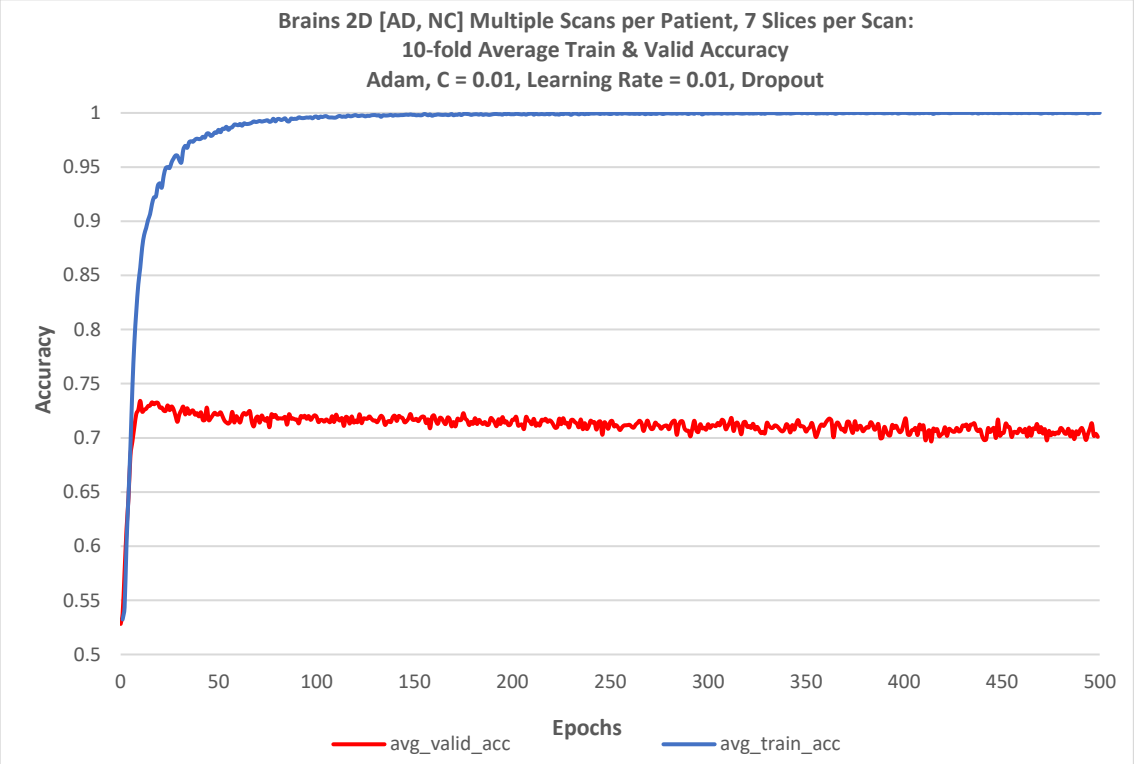


Figure J.1. Caption in the title.

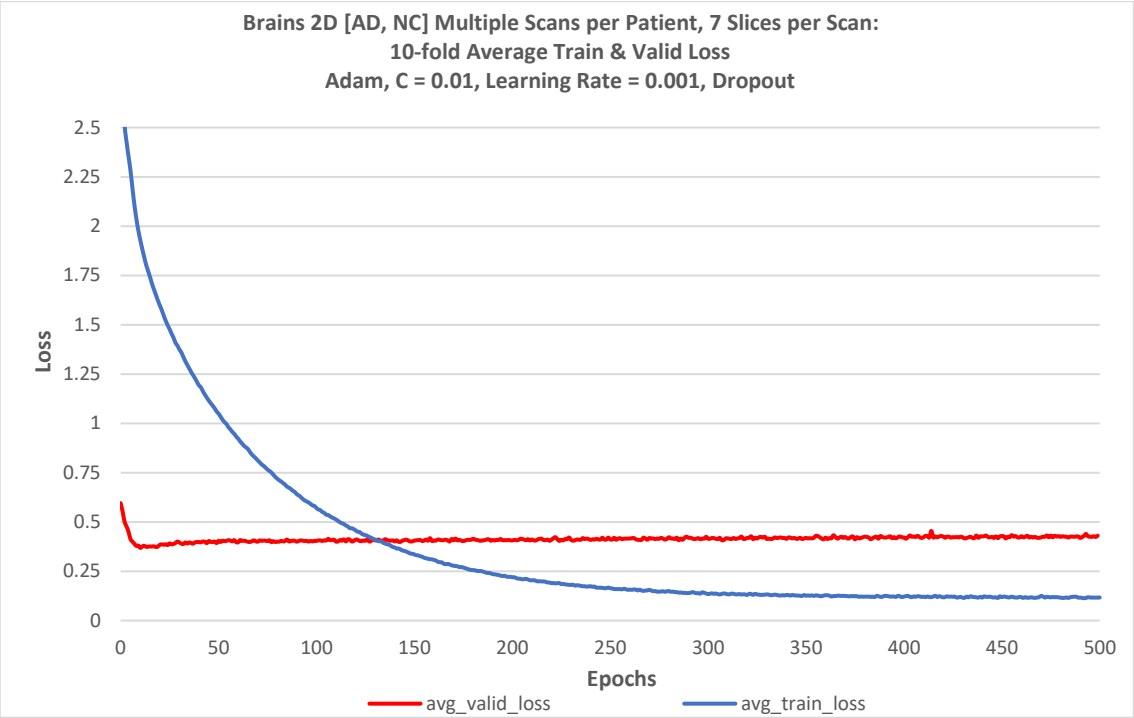


Figure J.2. Caption in the title.

J.1.1.2 Training

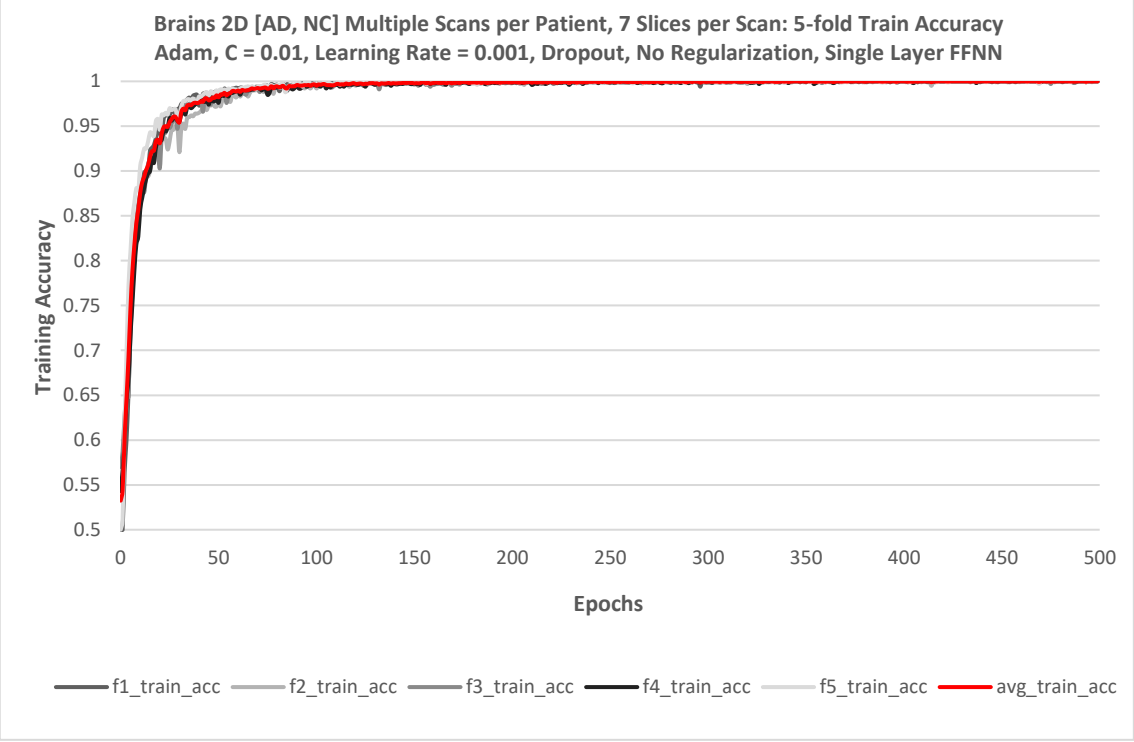


Figure J.3. Caption in the title.

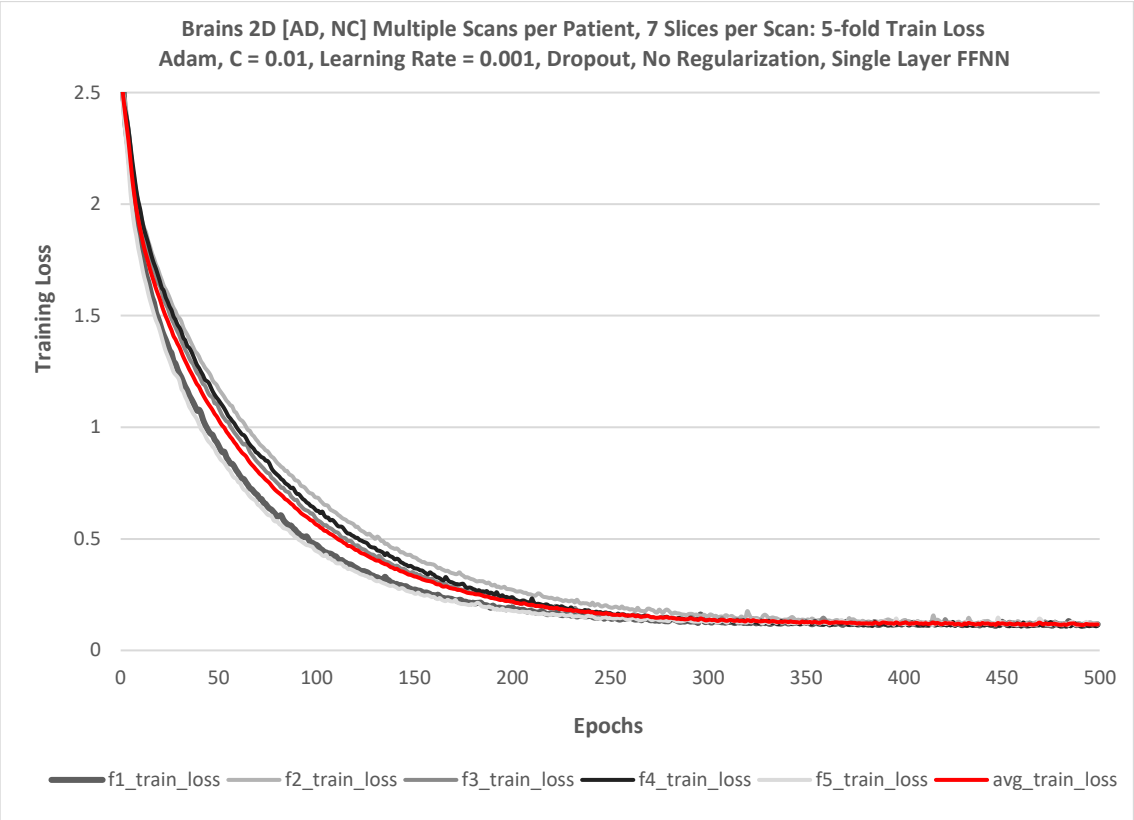


Figure J.4. Caption in the title.

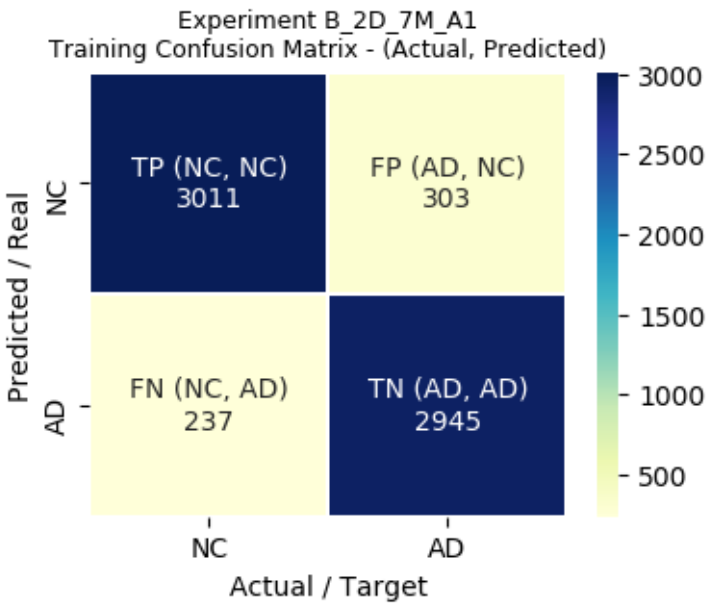


Figure J.5. Caption in the title.

J.1.1.3 Validation

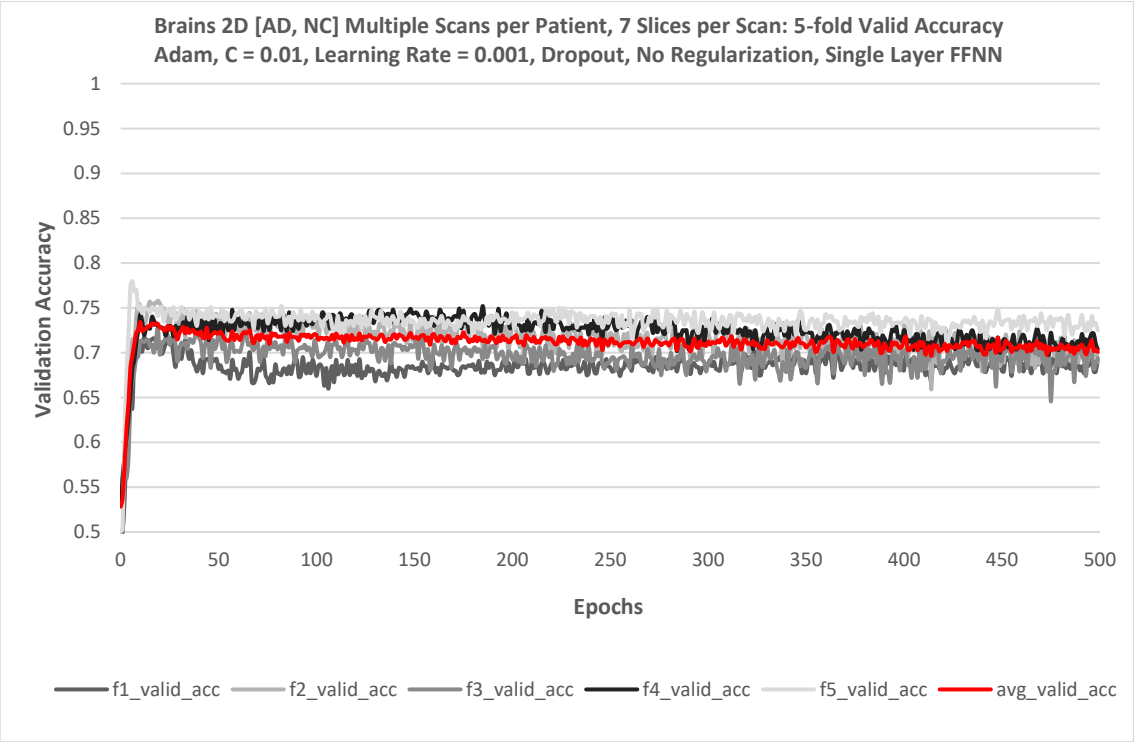


Figure J.6. Caption in the title.

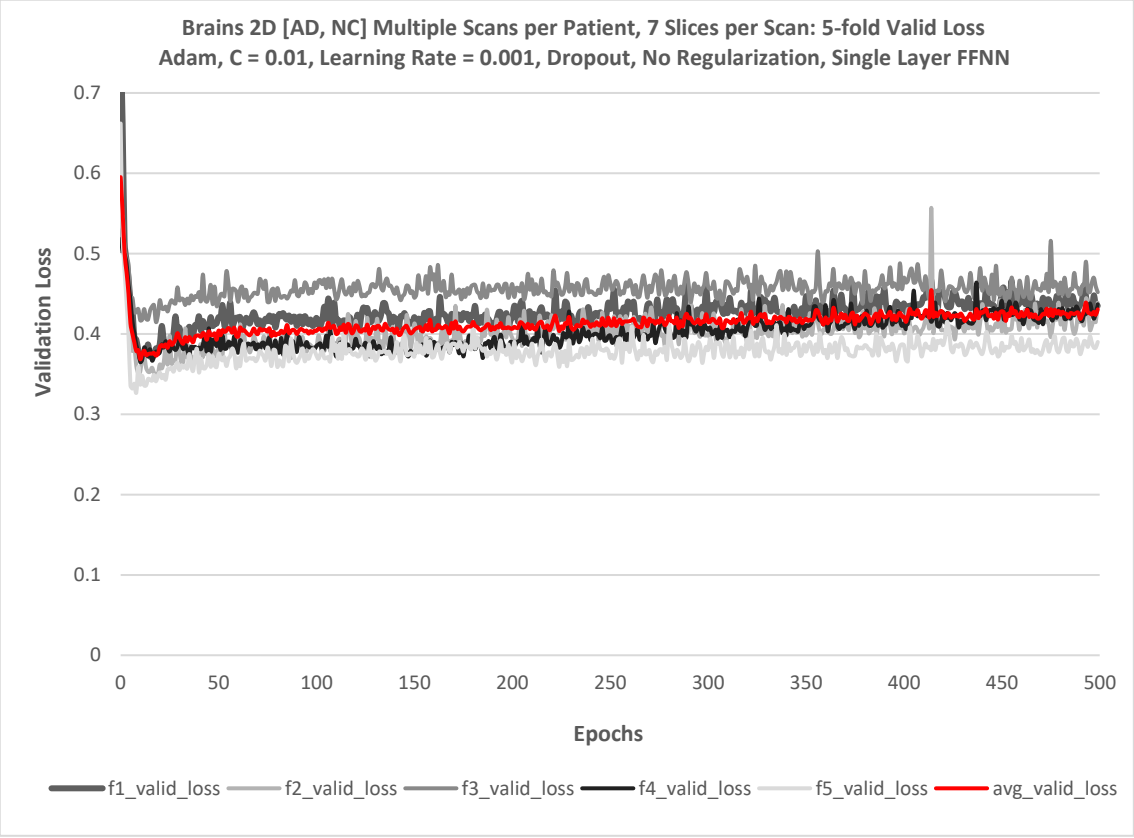


Figure J.7. Caption in the title.

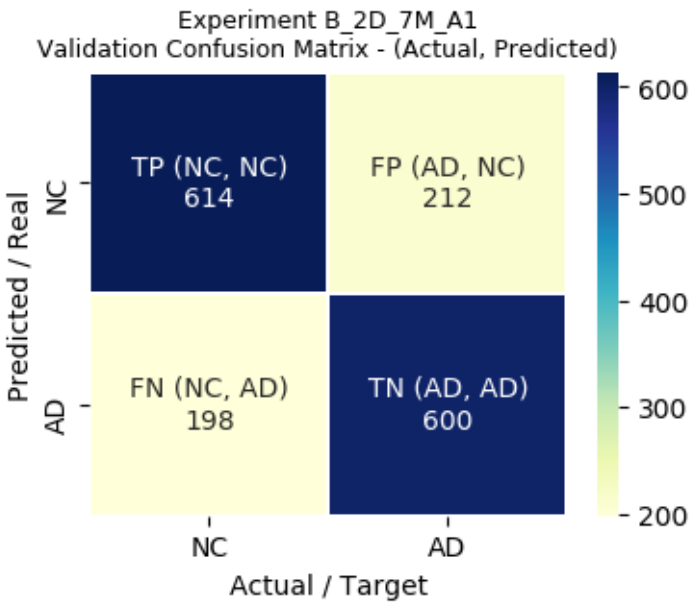


Figure J.8. Caption in the title.

J.1.1.4 Testing

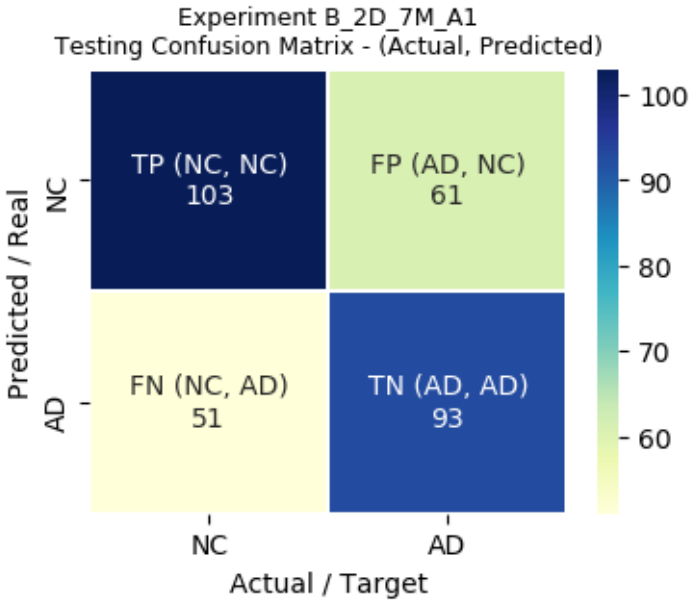


Figure J.9. Caption in the title.

J.2 CNN 4 Layers with NewtonCG

J.2.1 Experiment B_2D_7M_N1: C = 0.01, GNsize = 50, Dropout

J.2.1.1 General

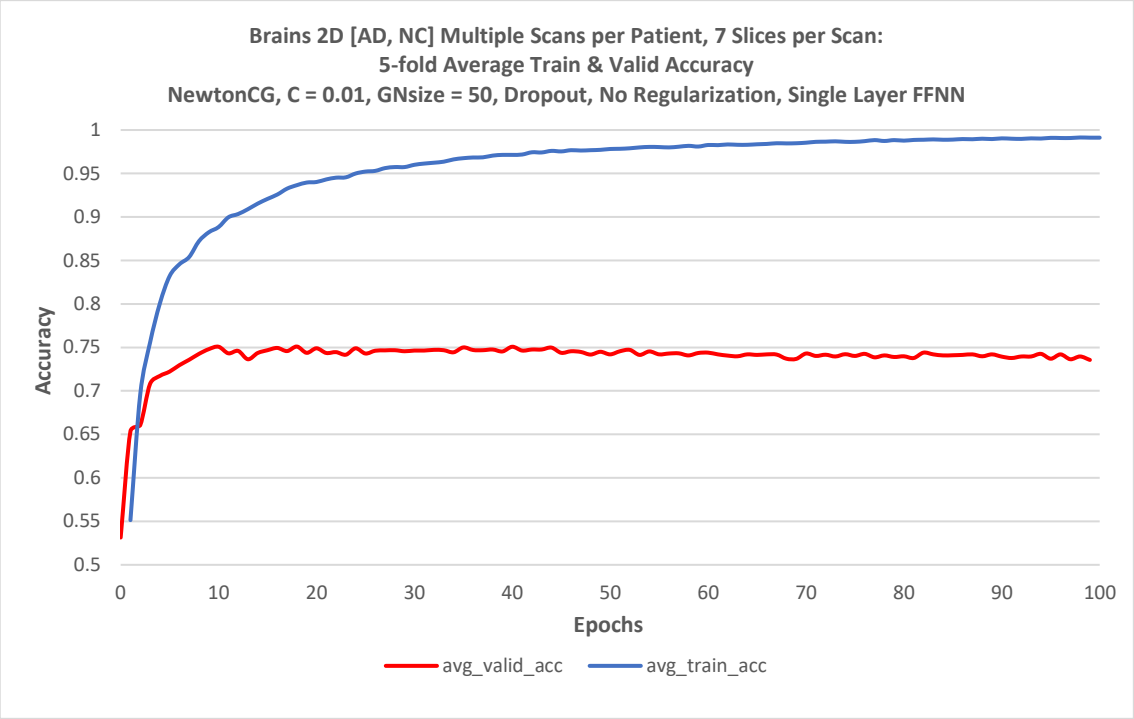


Figure J.10. Caption in the title.

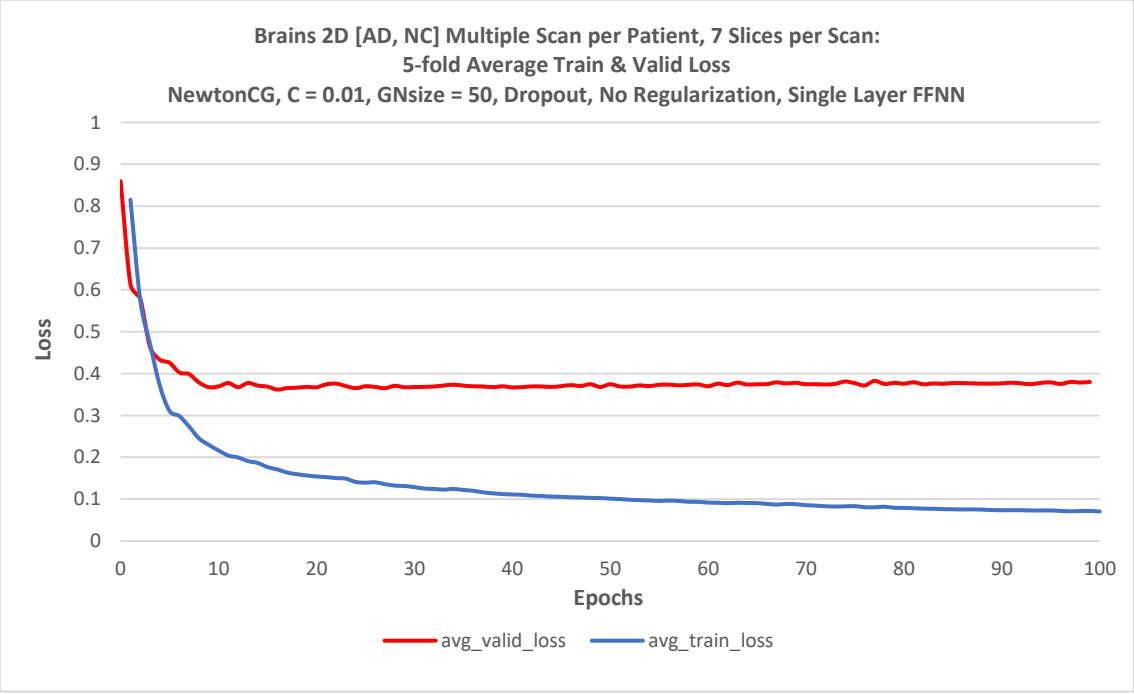


Figure J.11. Caption in the title.

J.2.1.2 Training

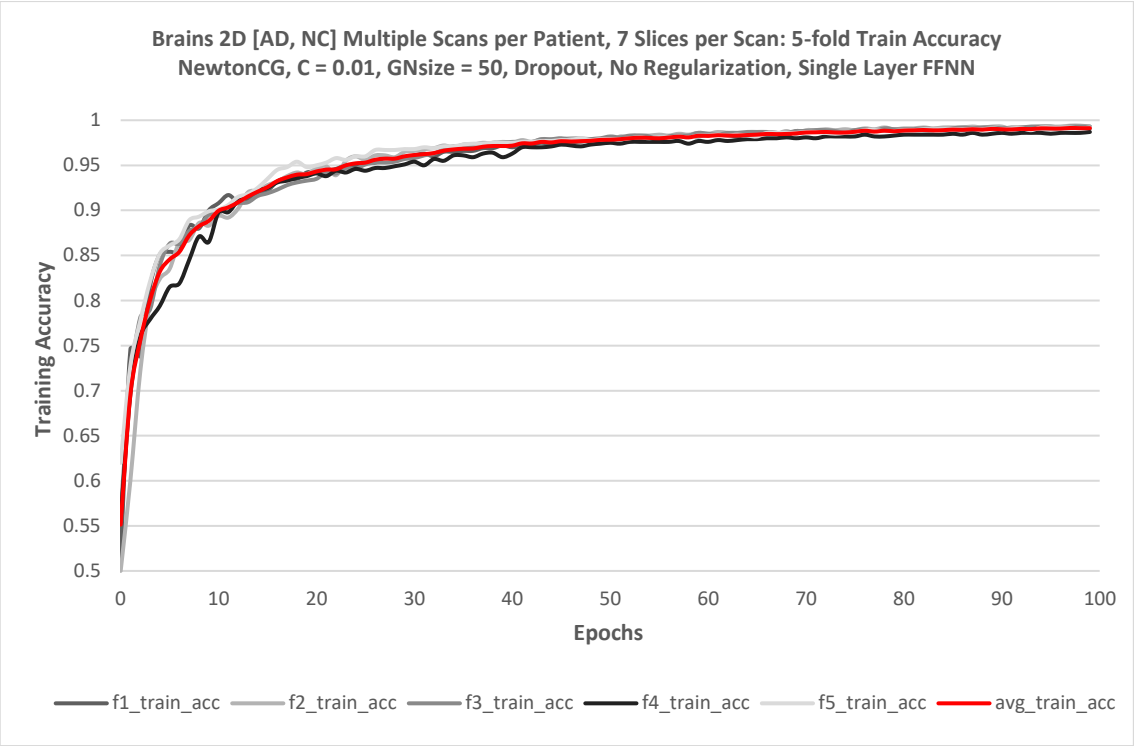


Figure J.12. Caption in the title.

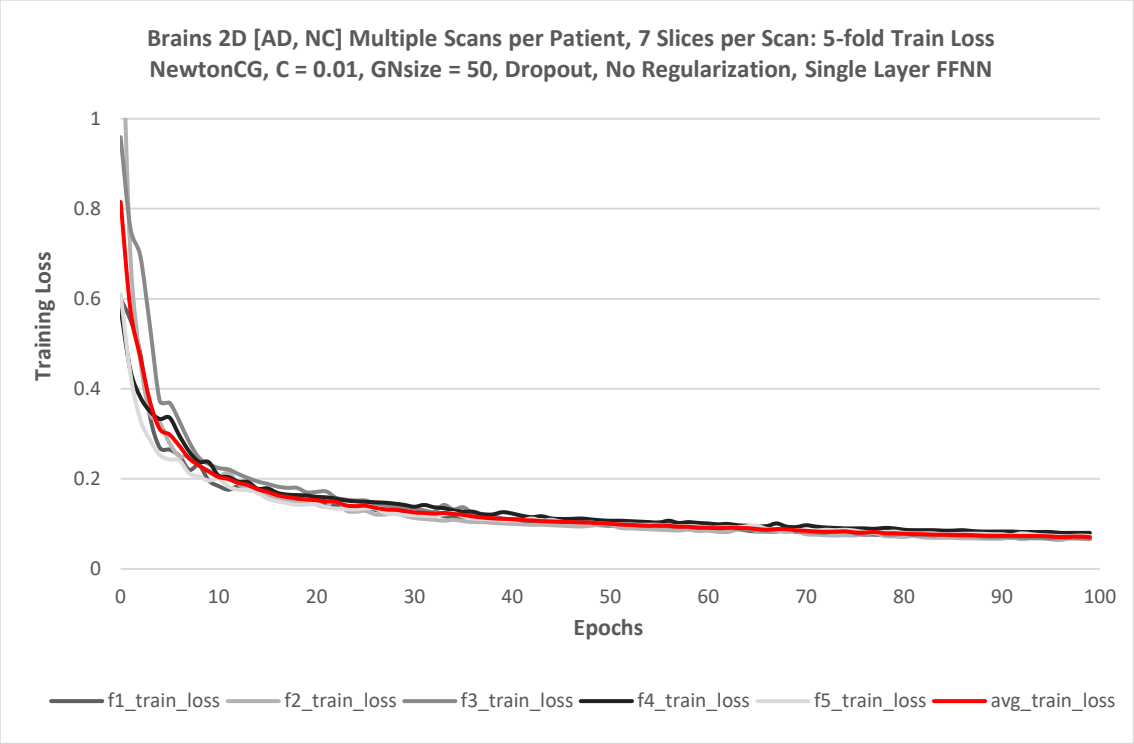


Figure J.13. Caption in the title.

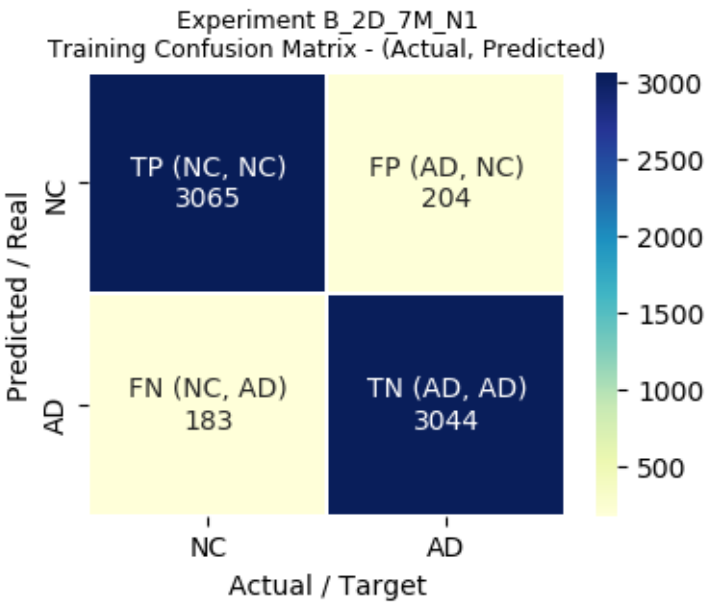


Figure J.14. Caption in the title.

J.2.1.3 Validation

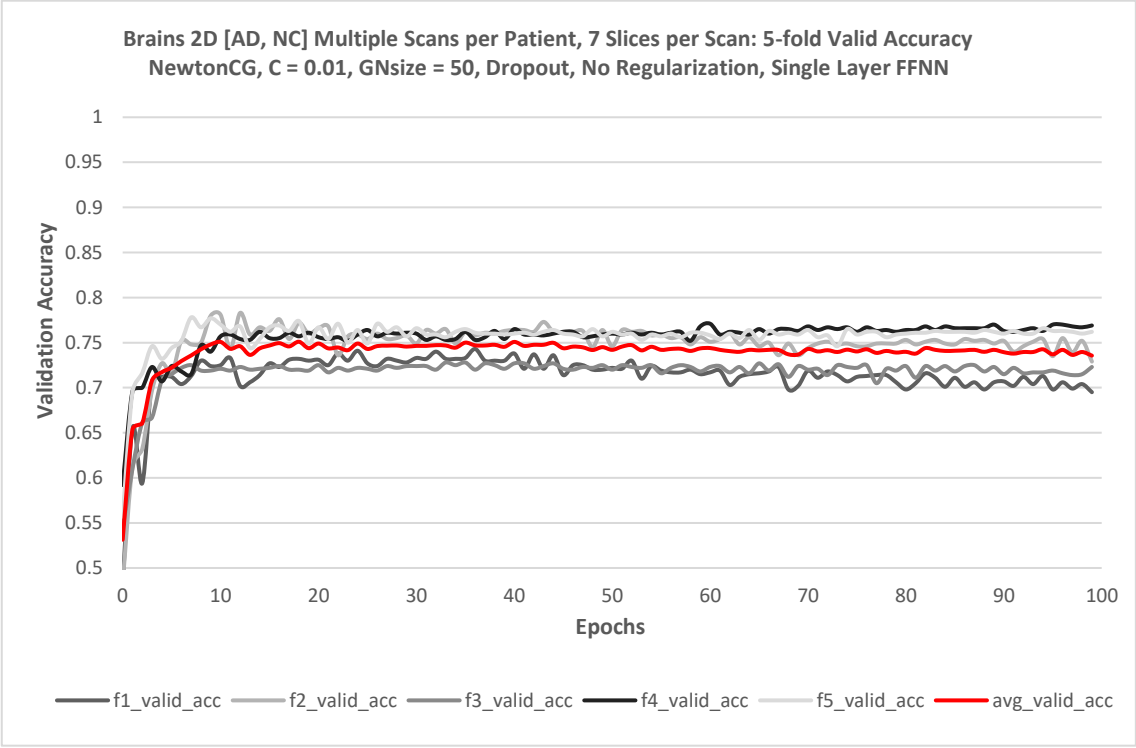


Figure J.15. Caption in the title.

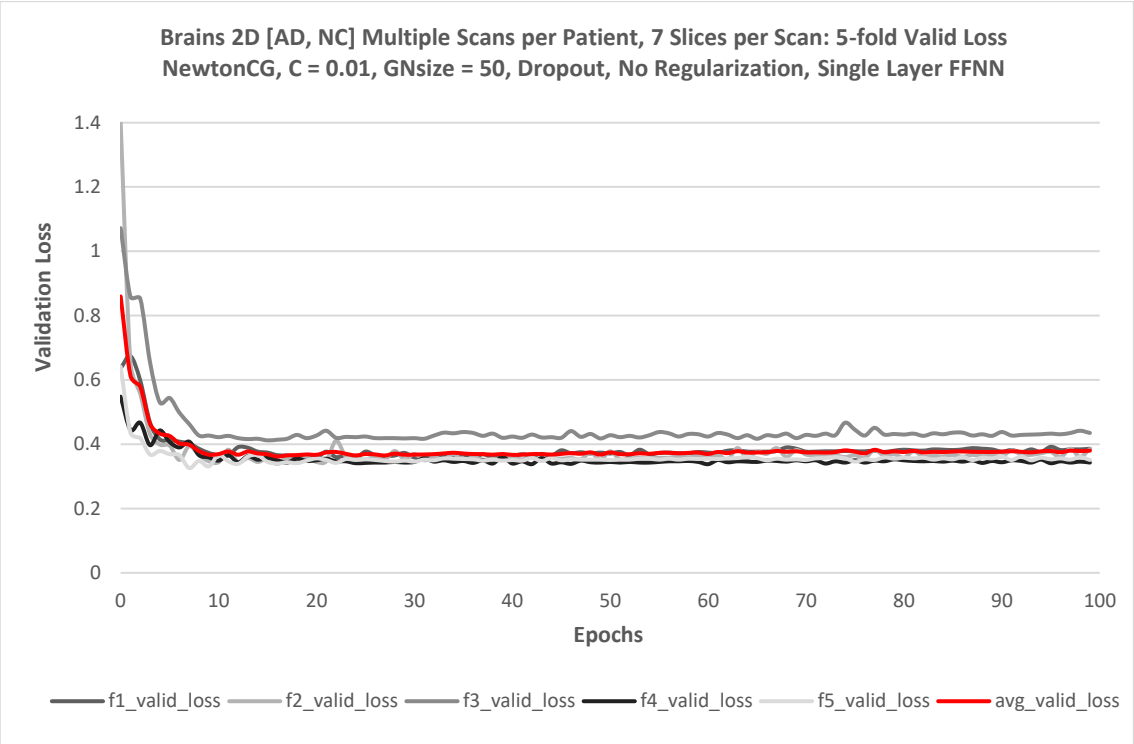


Figure J.16. Caption in the title.

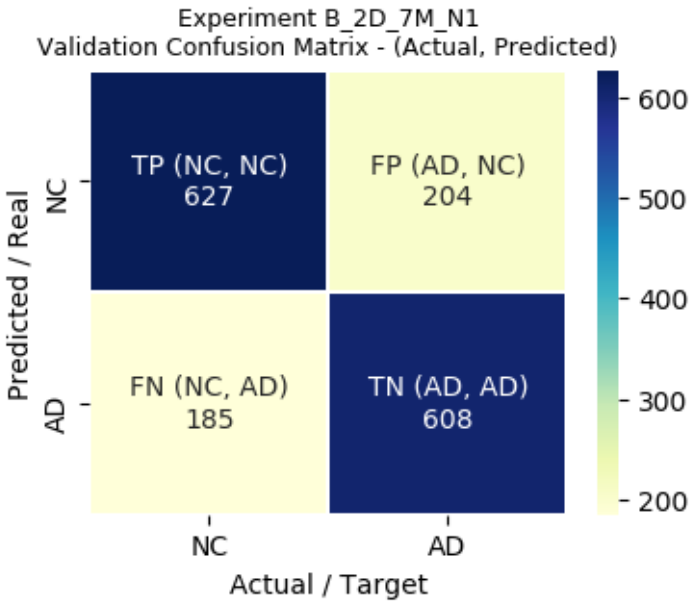


Figure J.17. Caption in the title.

J.2.1.4 Testing

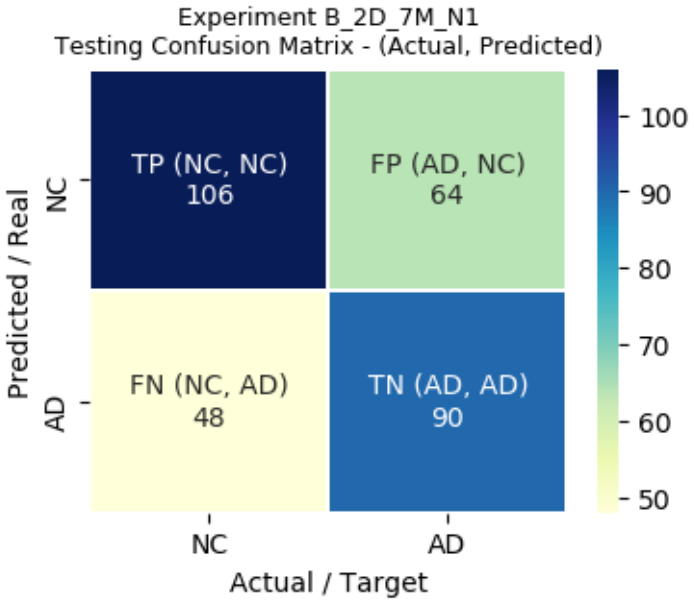


Figure J.18. Caption in the title.

J.3 CNN 3 Layers with NewtonCG

J.3.1 Experiment B_2D_7M_N2: C = 0.01, GNsize = 50, Dropout, Shallow & Wide Network

J.3.1.1 General

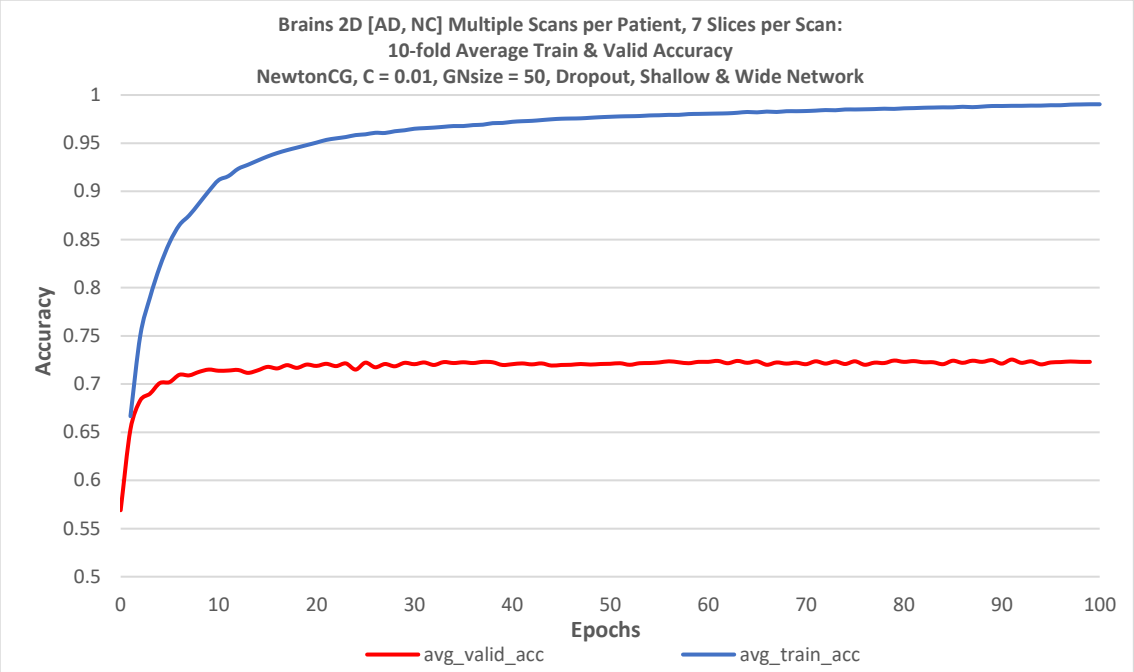


Figure J.19. Caption in the title.

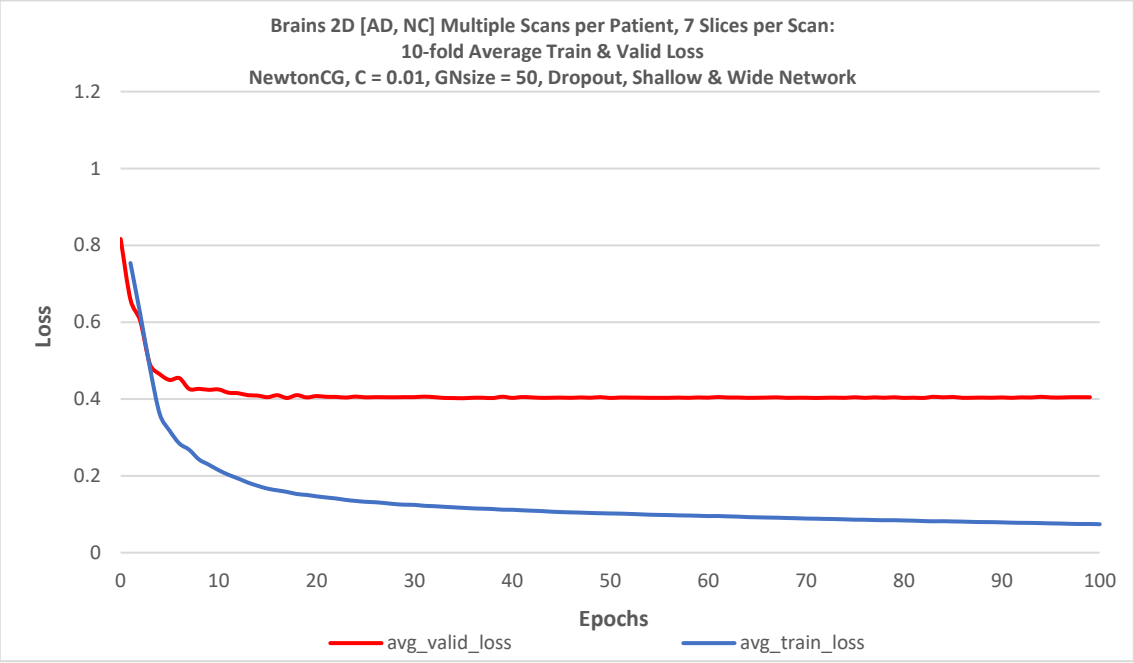


Figure J.20. Caption in the title.

J.3.1.2 Training

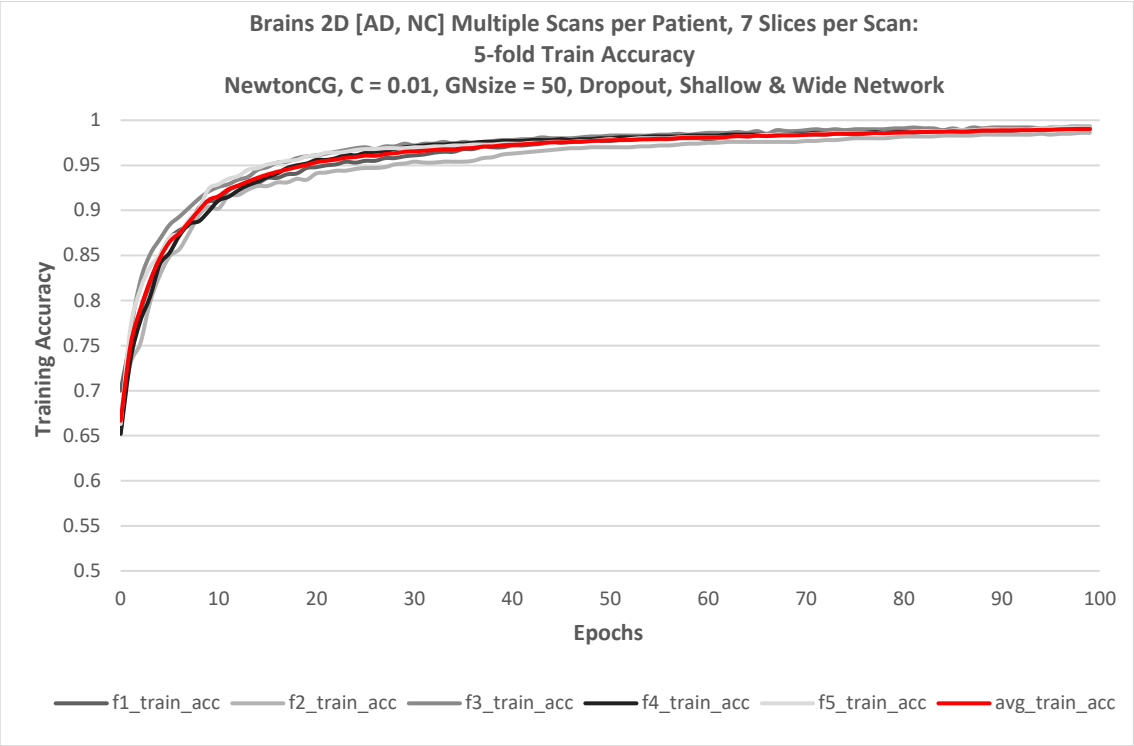


Figure J.21. Caption in the title.

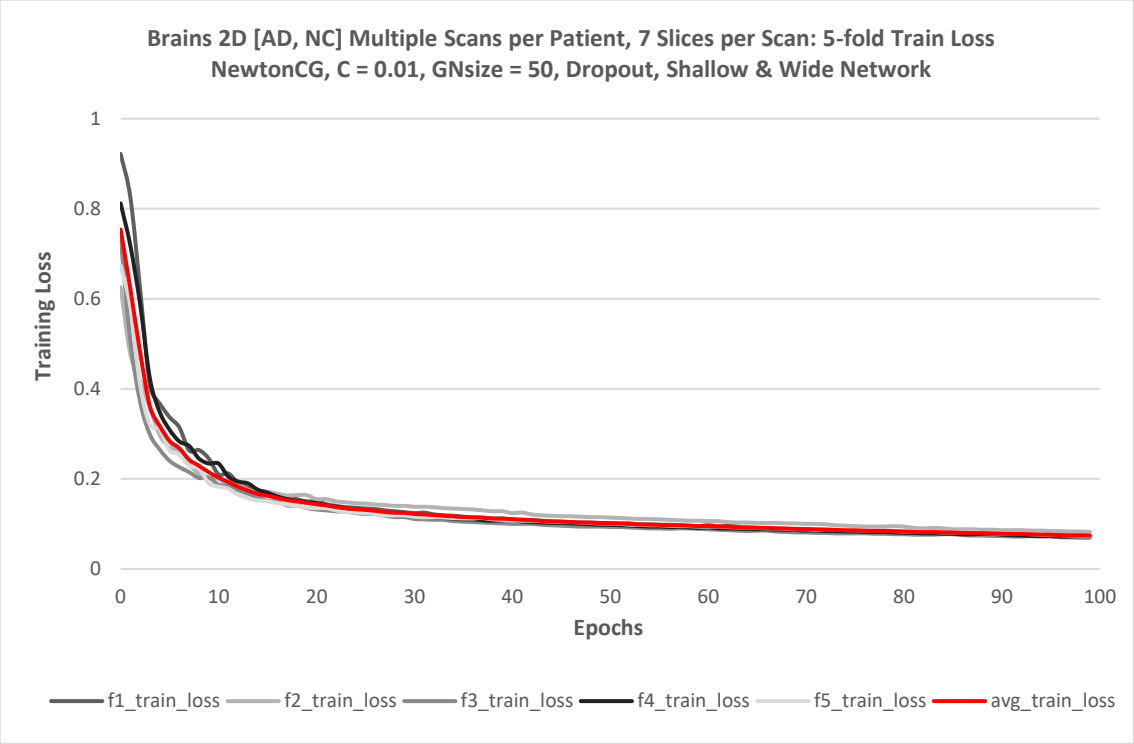


Figure J.22. Caption in the title.

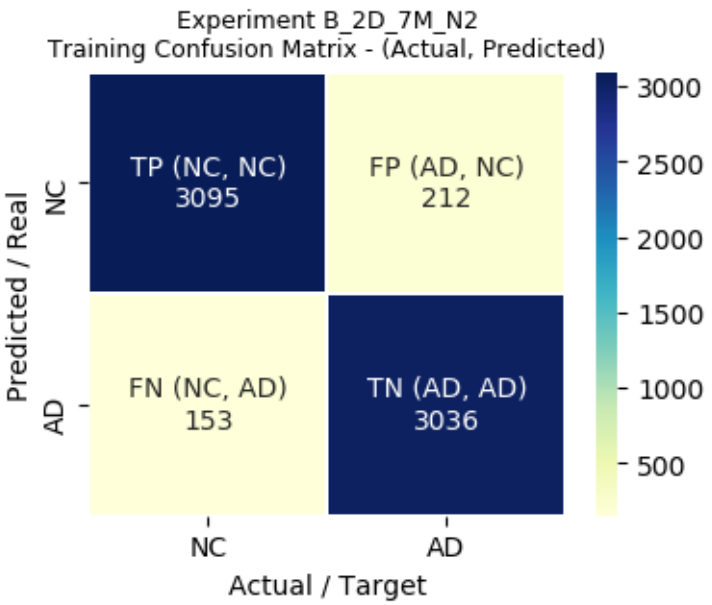


Figure J.23. Caption in the title.

J.3.1.3 Validation

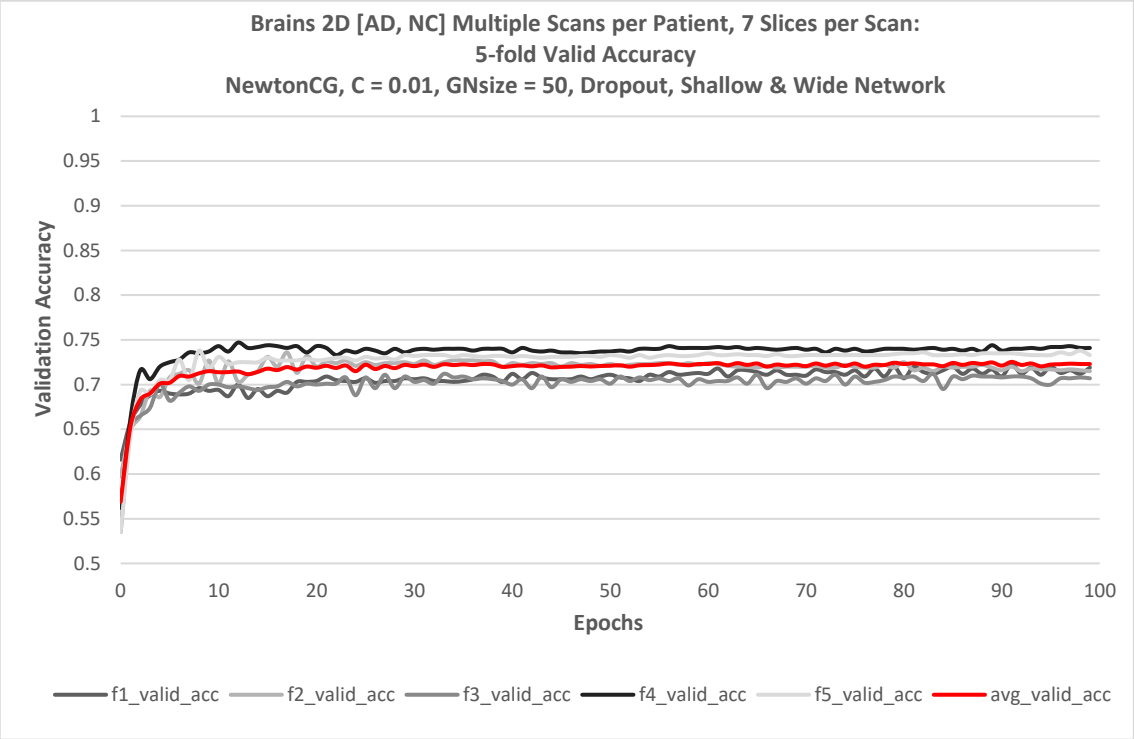


Figure J.24. Caption in the title.

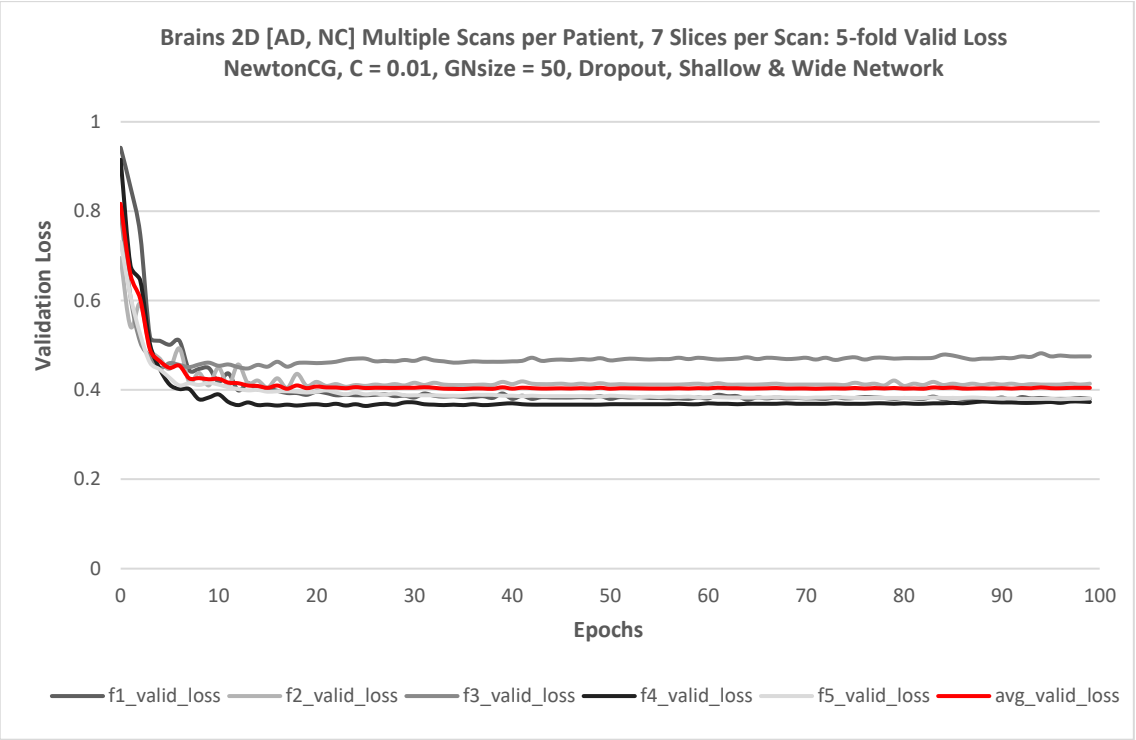


Figure J.25. Caption in the title.

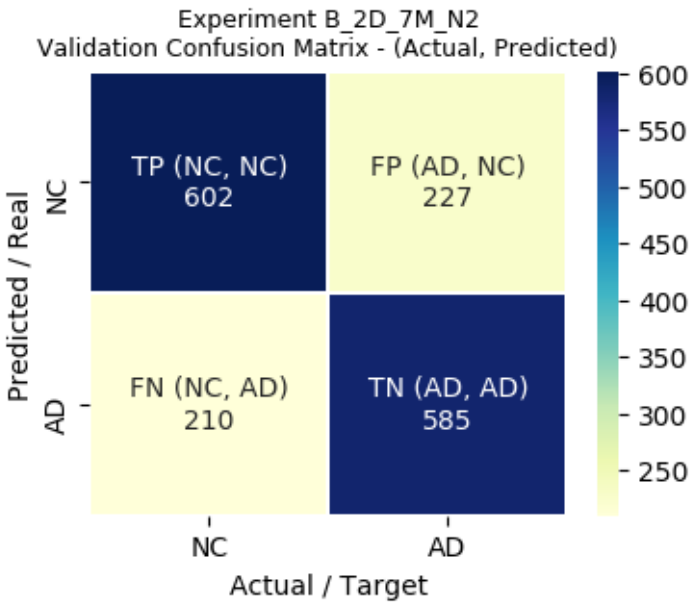


Figure J.26. Caption in the title.

J.3.1.4 Testing

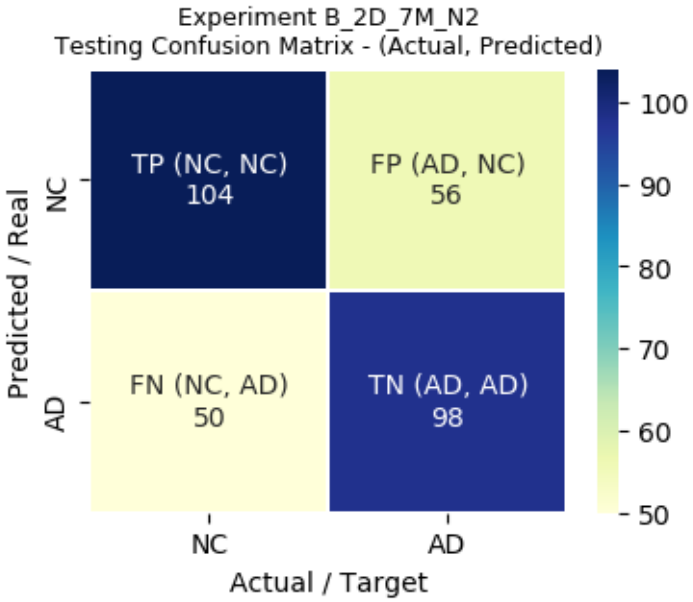


Figure J.27. Caption in the title.

J.3.2 Experiment B_2D_7M_N3: C = 0.01, GNsize = 50, Dropout, Shallow & Narrow Network

J.3.2.1 General

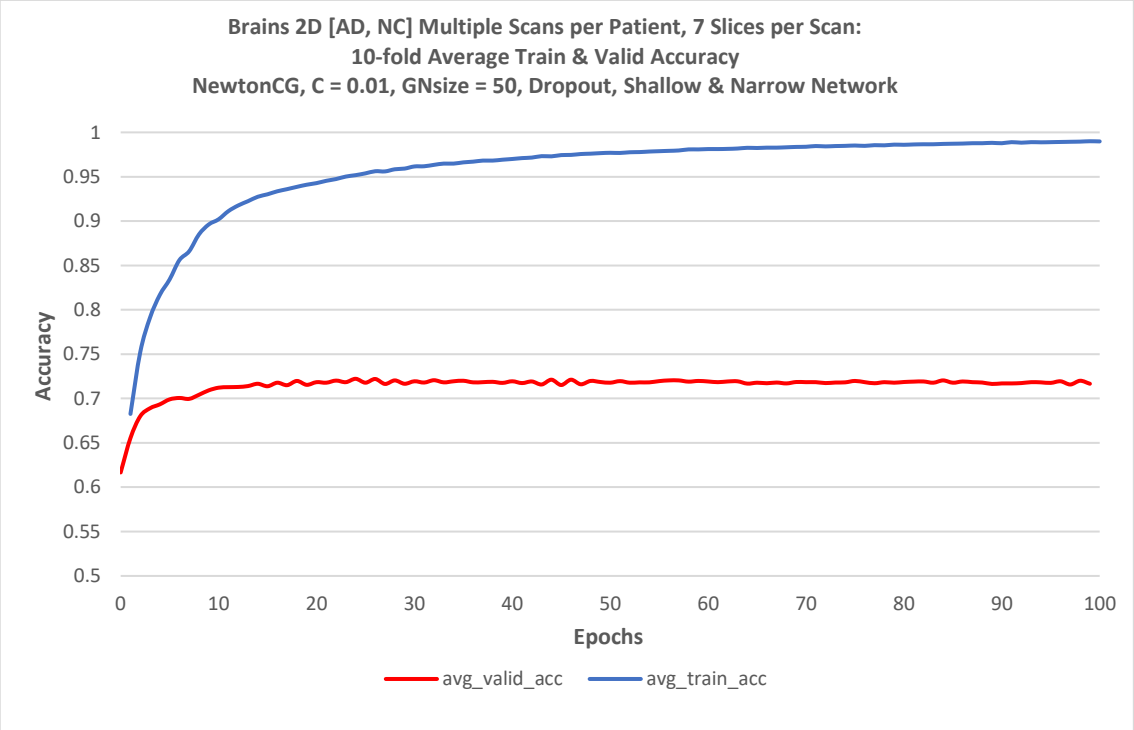


Figure J.28. Caption in the title.

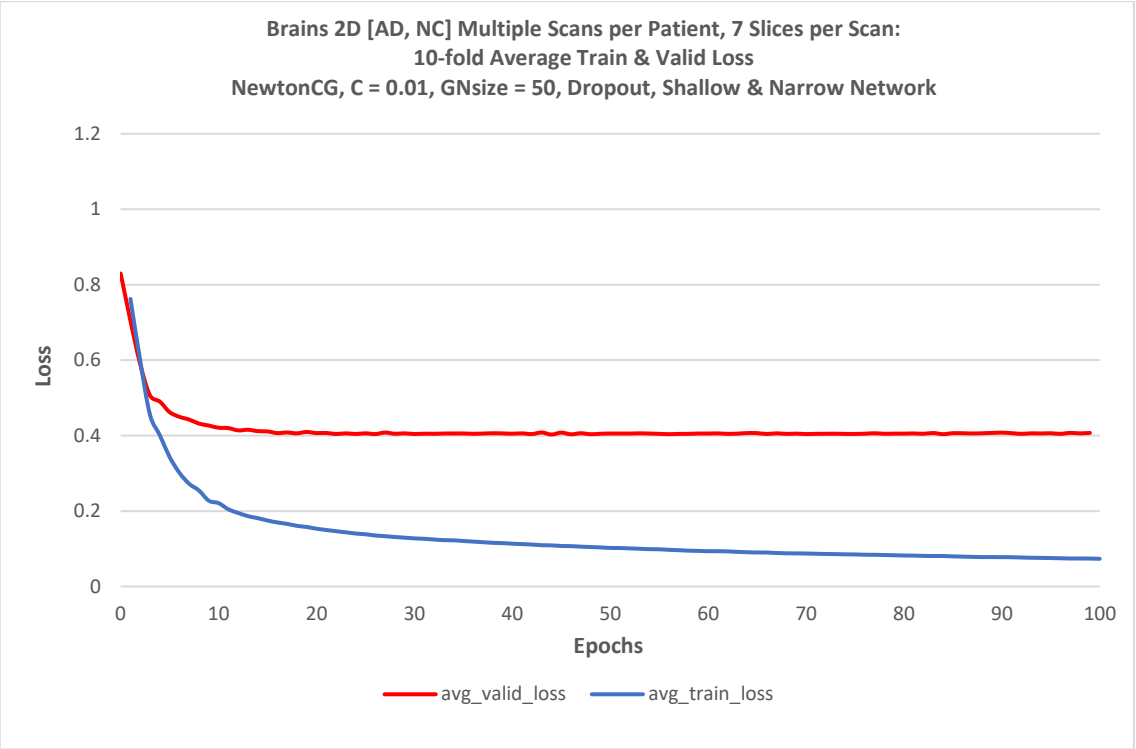


Figure J.29. Caption in the title.

J.3.2.2 Training

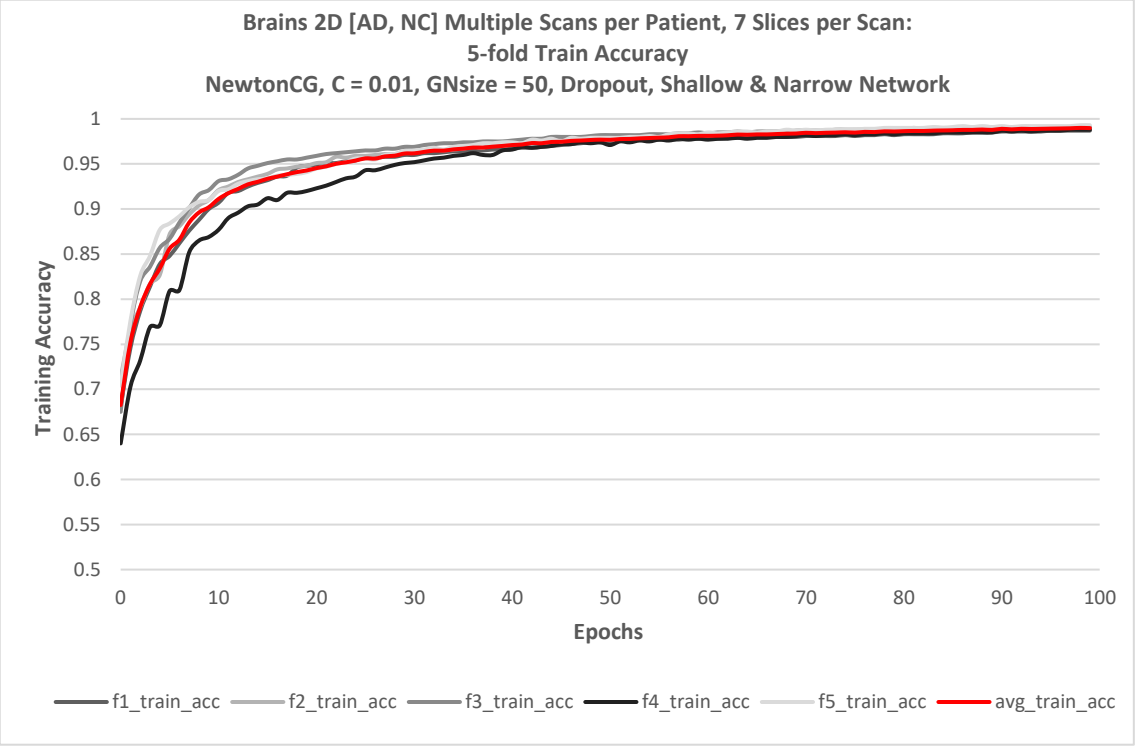


Figure J.30. Caption in the title.

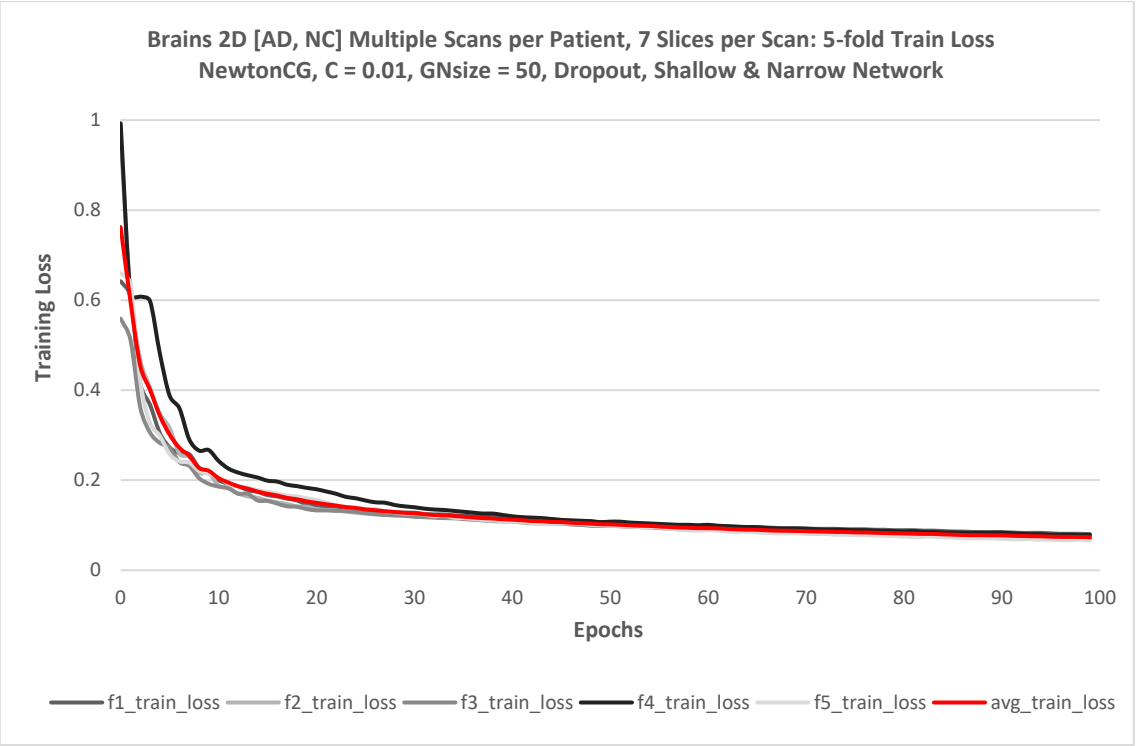


Figure J.31. Caption in the title.

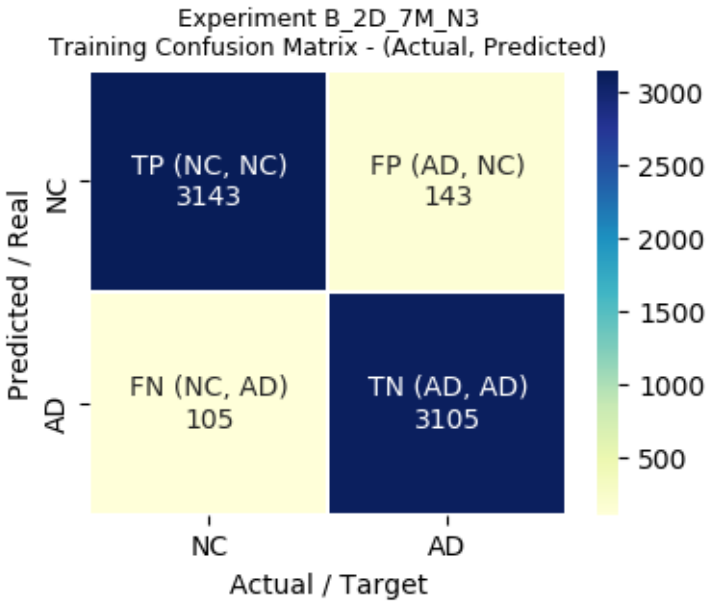


Figure J.32. Caption in the title.

J.3.2.3 Validation

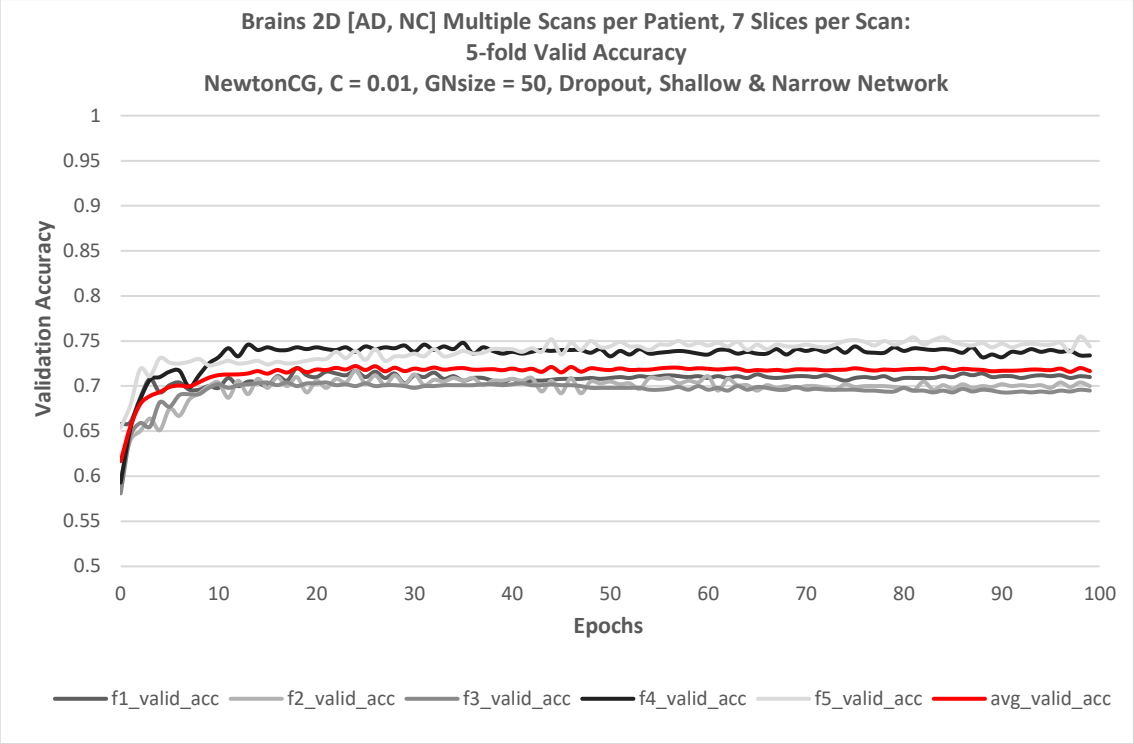


Figure J.33. Caption in the title.

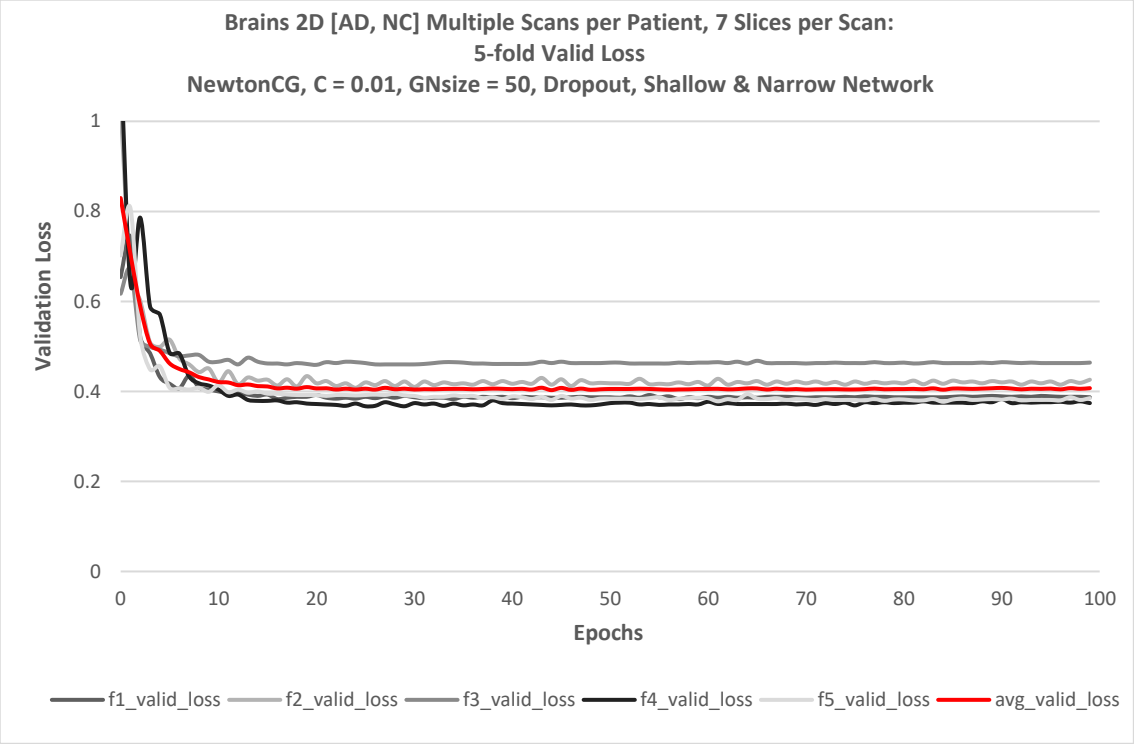


Figure J.34. Caption in the title.

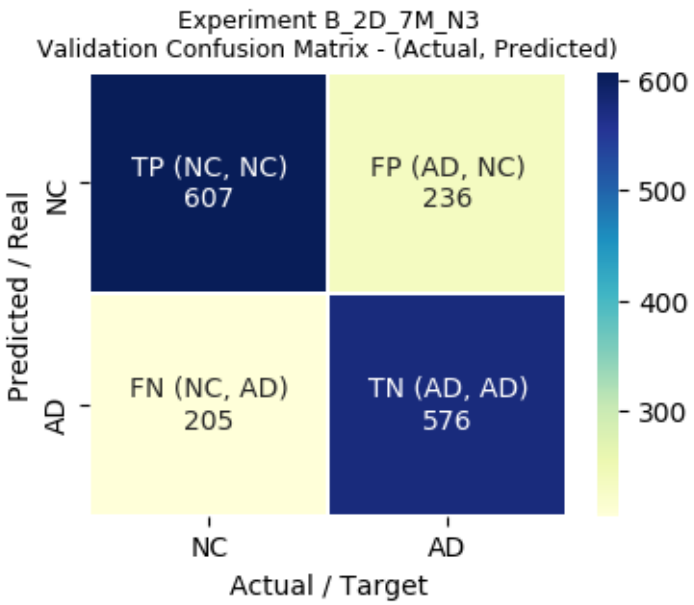


Figure J.35. Caption in the title.

J.3.2.4 Testing

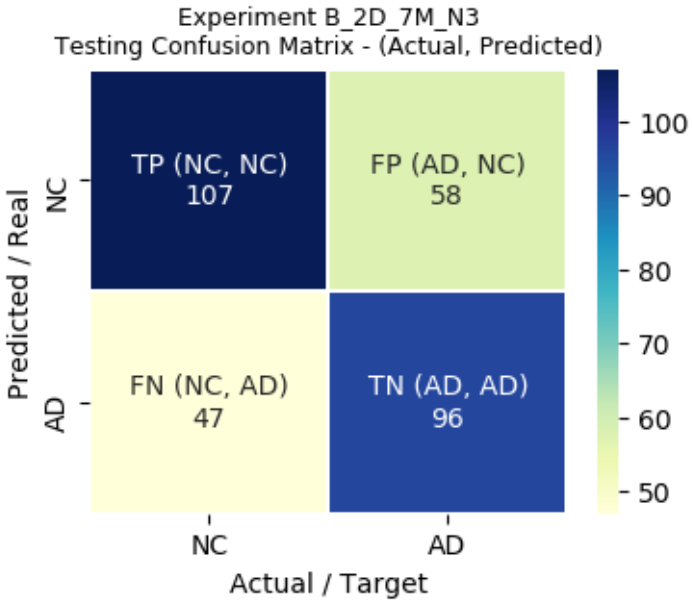


Figure J.36. Caption in the title.

Appendix K

Experiments 2D Brain Slices [AD, NC]: Single Scan per Patient – 5 Slices per Scan (174 × 174)

K.1 CNN 4 Layers with NewtonCG

K.1.1 Experiment B_2D_5S_N1: C = 1, GNsize = 50, Dropout

K.1.1.1 General

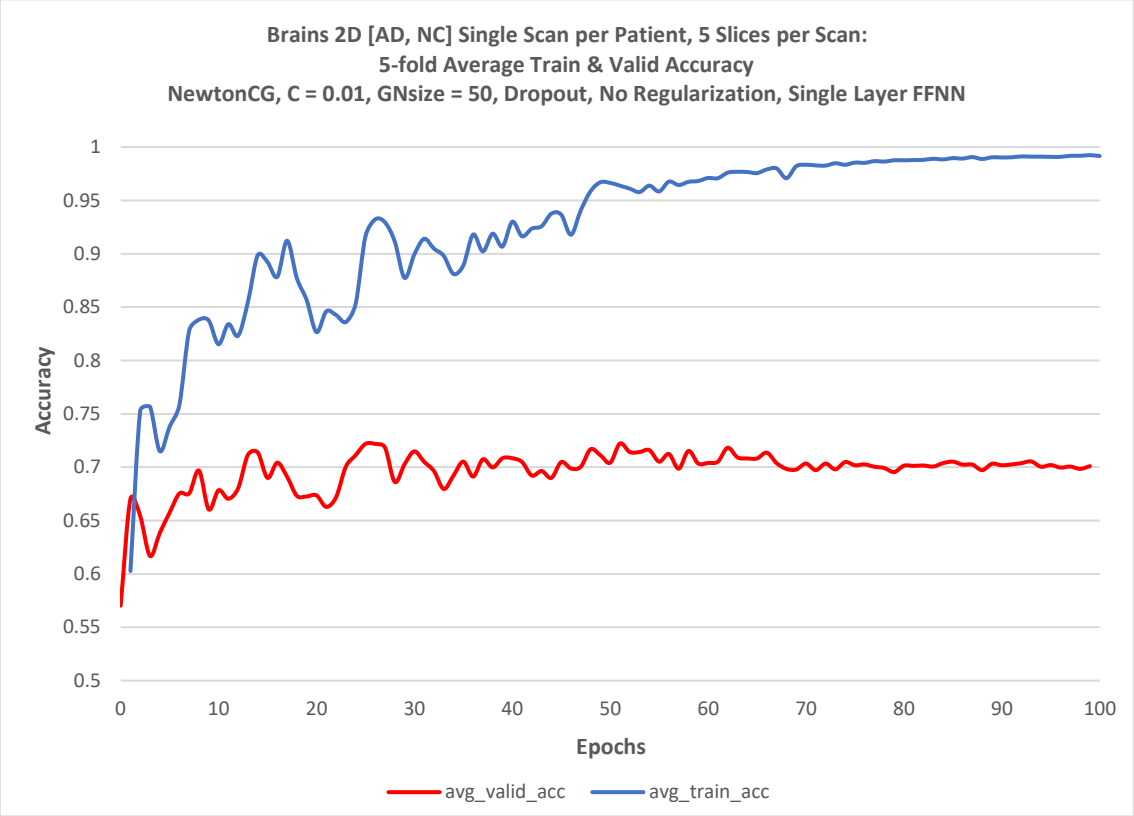


Figure K.1. Caption in the title.

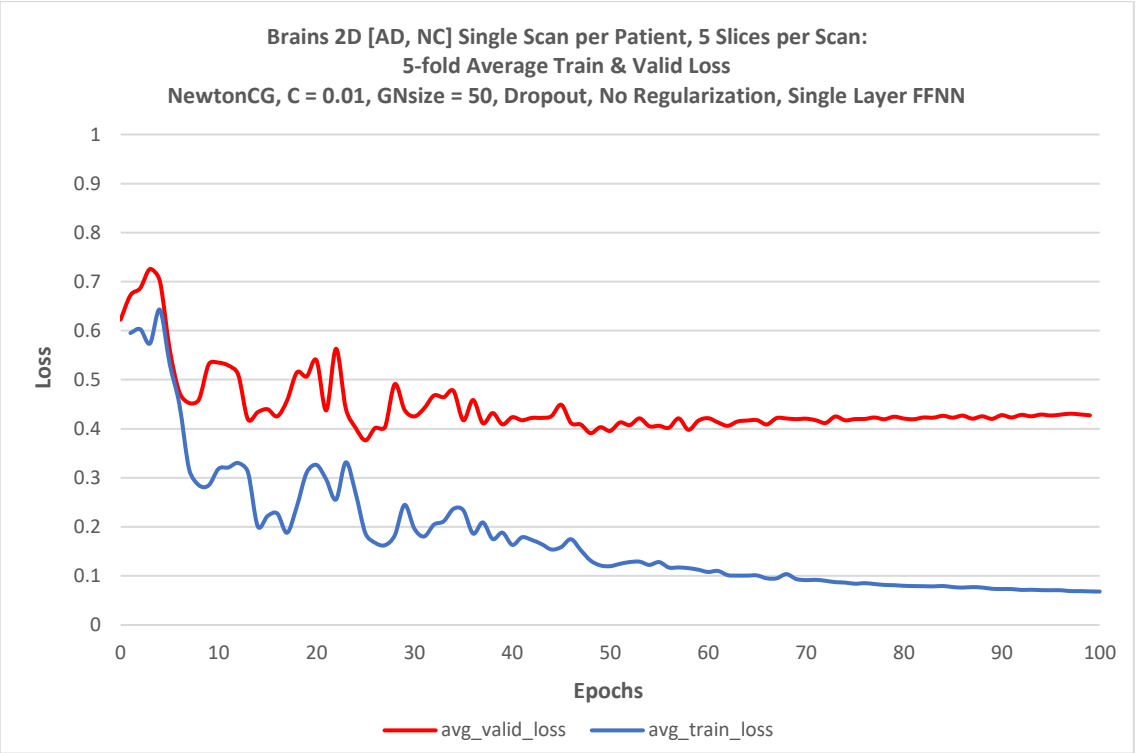


Figure K.2. Caption in the title.

K.1.1.2 Training

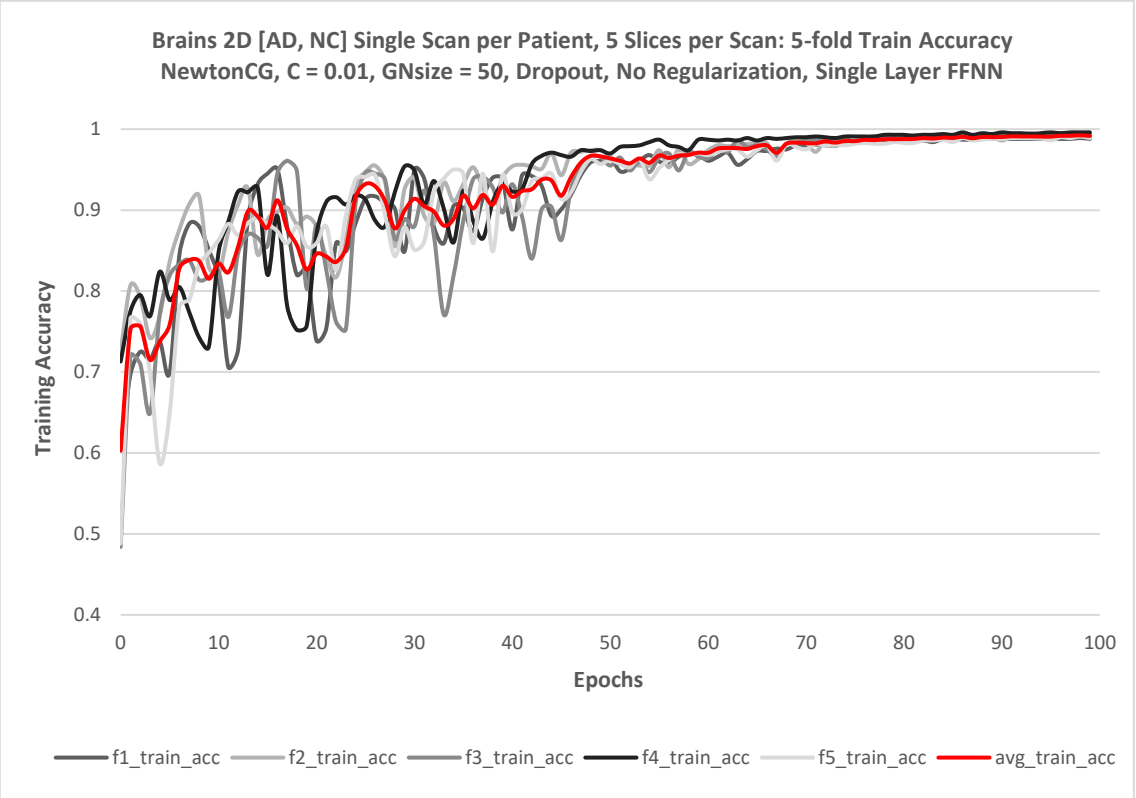


Figure K.3. Caption in the title.

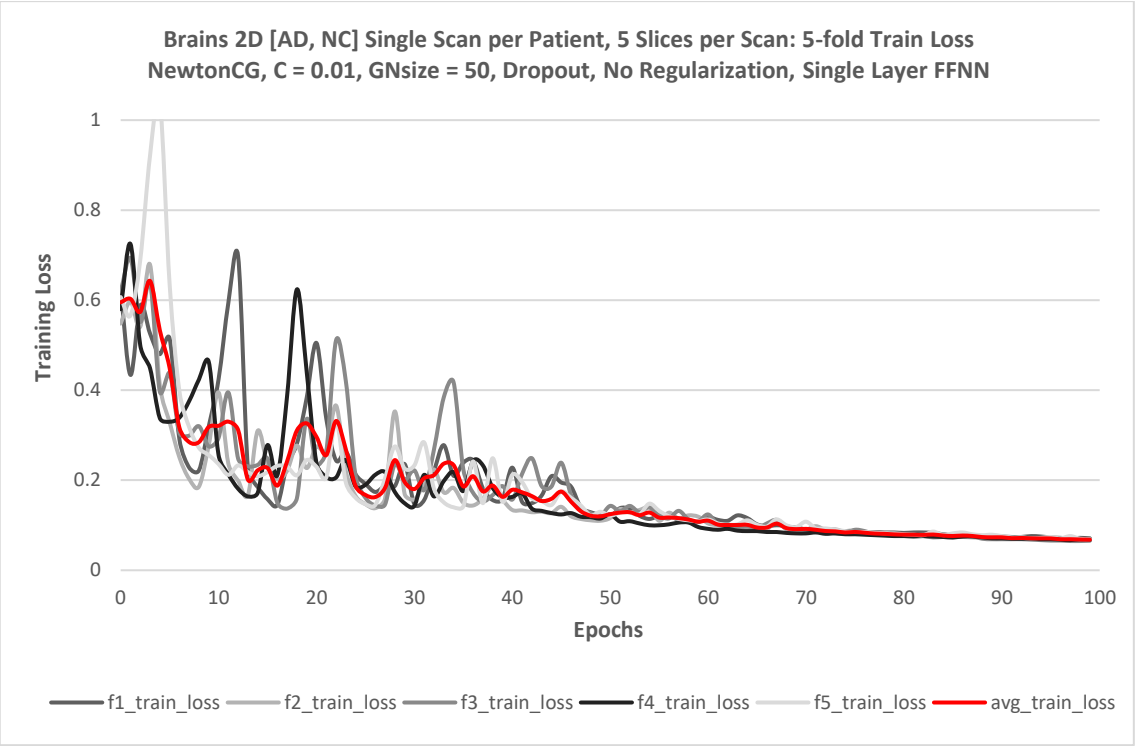


Figure K.4. Caption in the title.

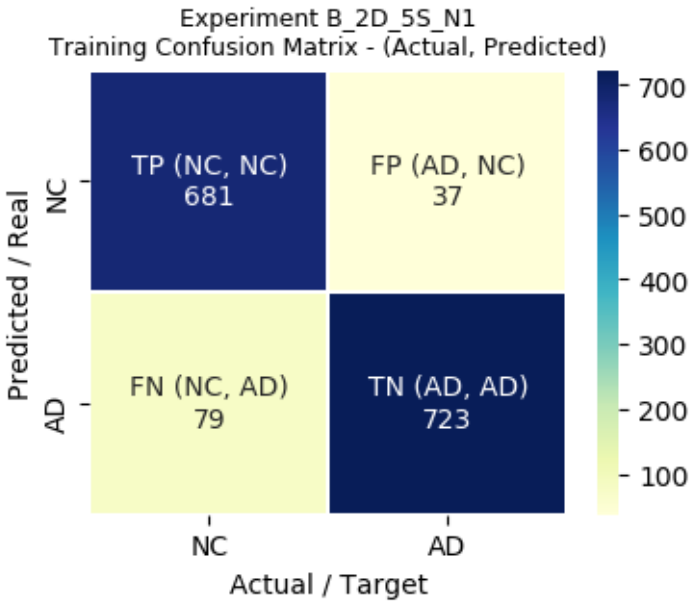


Figure K.5. Caption in the title.

K.1.1.3 Validation

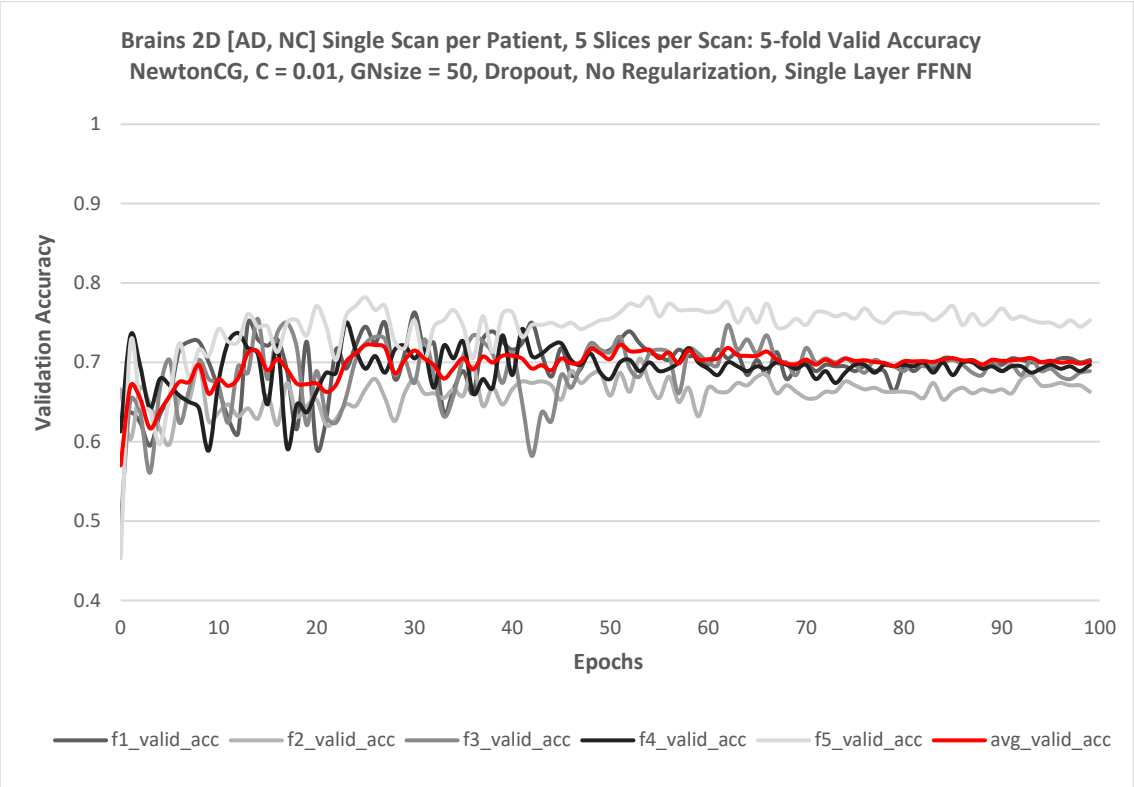


Figure K.6. Caption in the title.

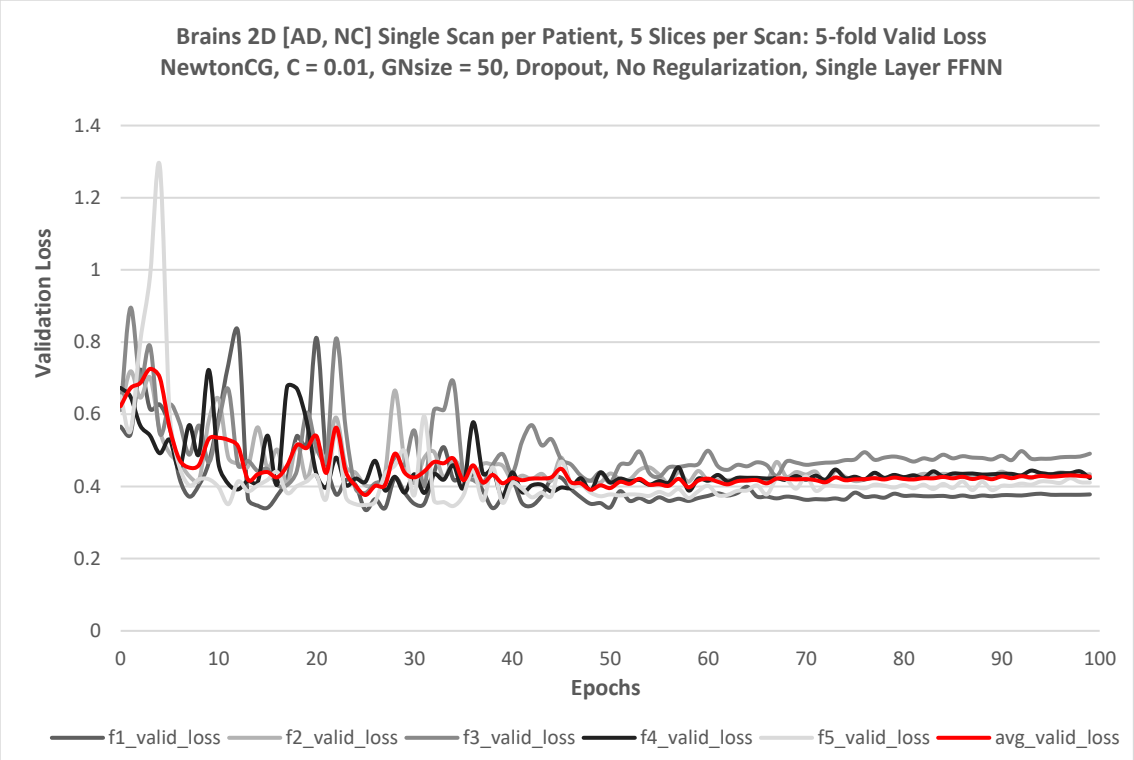


Figure K.7. Caption in the title.

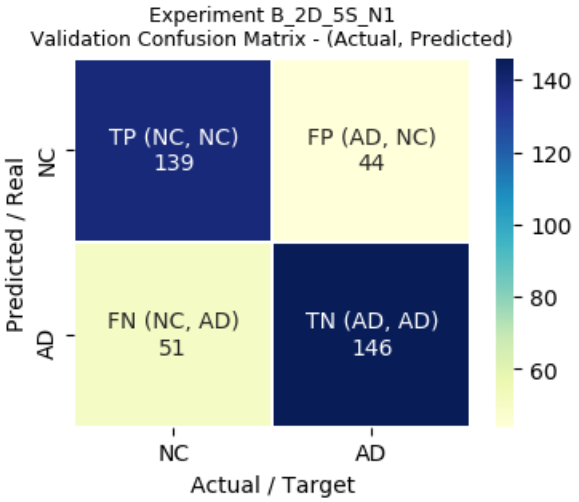


Figure K.8. Caption in the title.

K.1.1.4 Testing

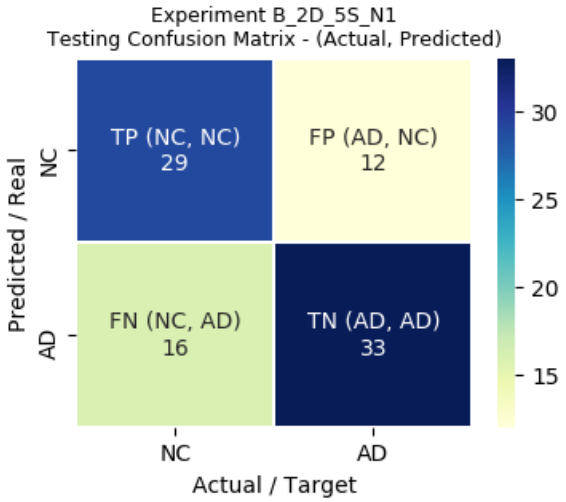


Figure K.9. Caption in the title.

Appendix L

Experiments 3D Left Hippocampus [AD, NC]: Single Scan per Patient (37 × 32 × 50)

L.1 CNN 4 Layer with Adam

L.1.1 Experiment LH_3D_S_A1: C = 0.01, lr = 0.001, Dropout

L.1.1.1 General

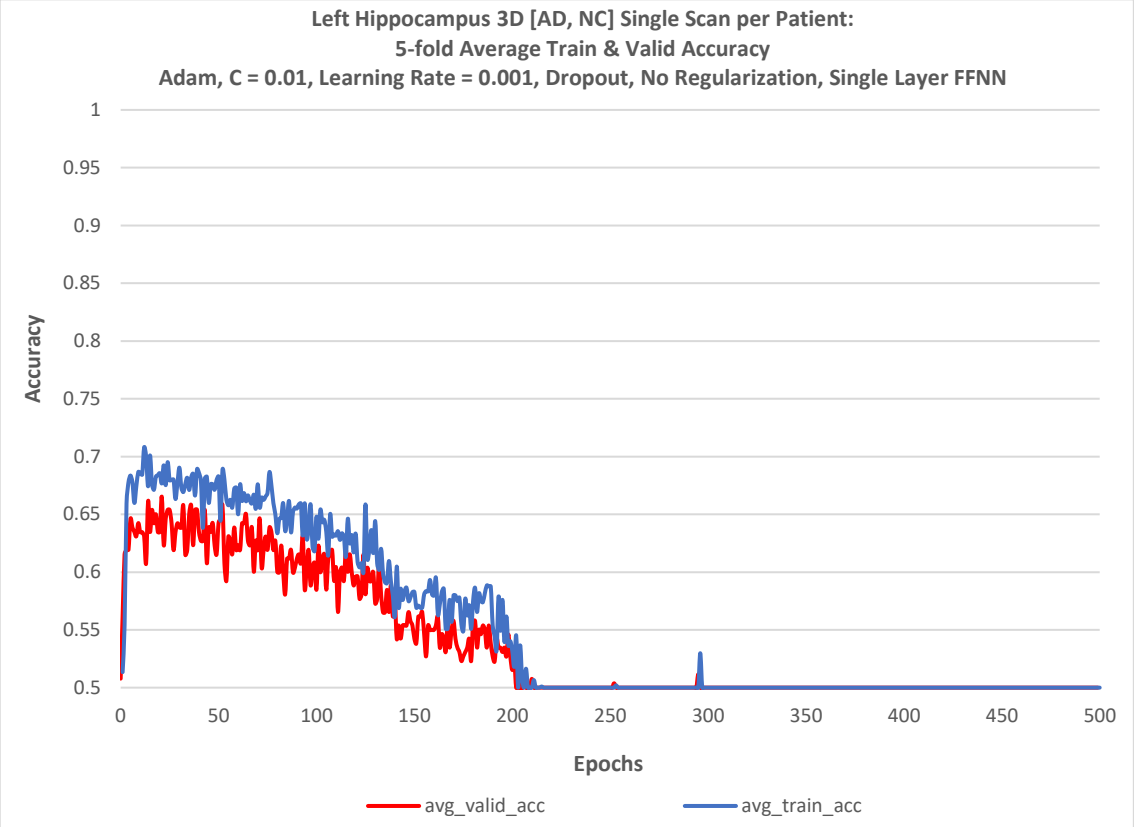


Figure L.1. Caption in the title.

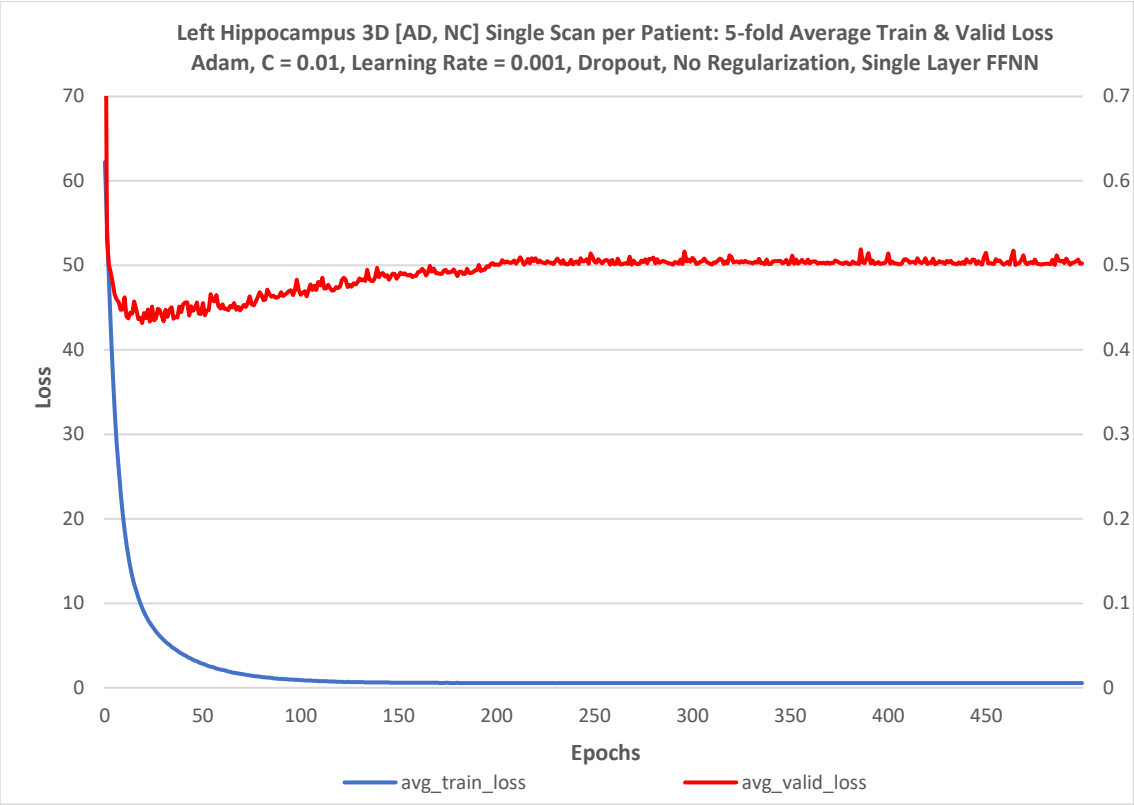


Figure L.2. Caption in the title.

L.1.1.2 Training

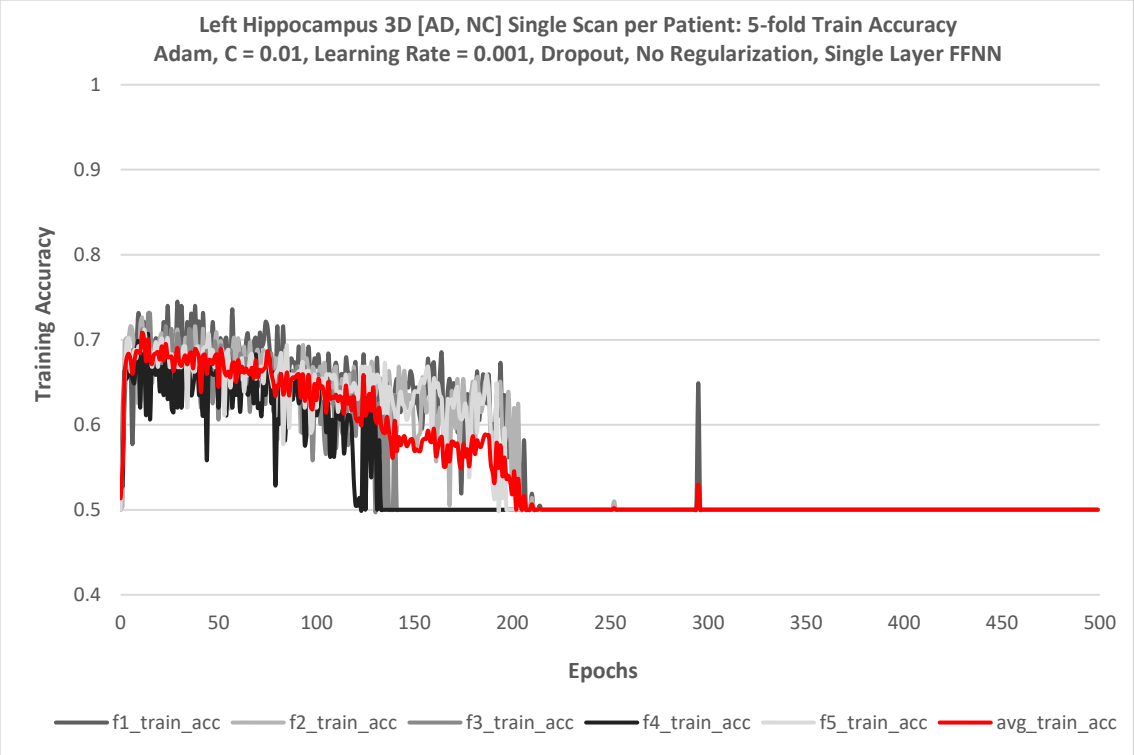


Figure L.3. Caption in the title.

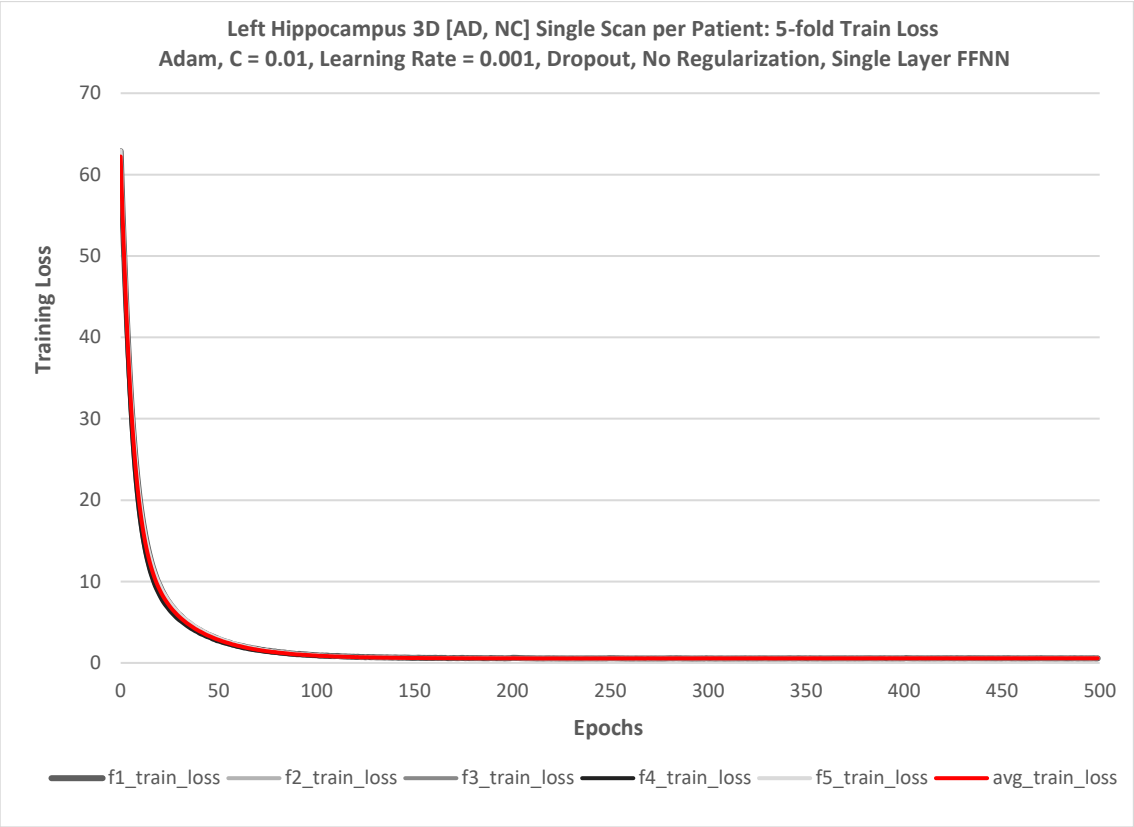


Figure L.4. Caption in the title.

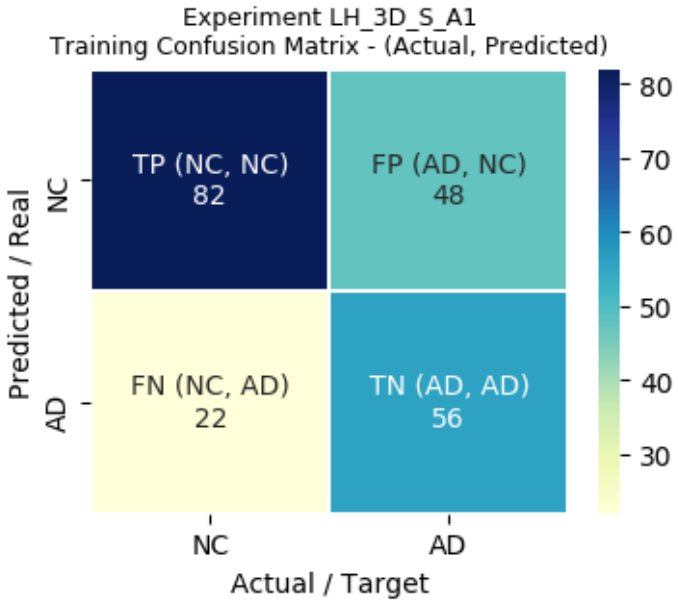


Figure L.5. Caption in the title.

L.1.1.3 Validation

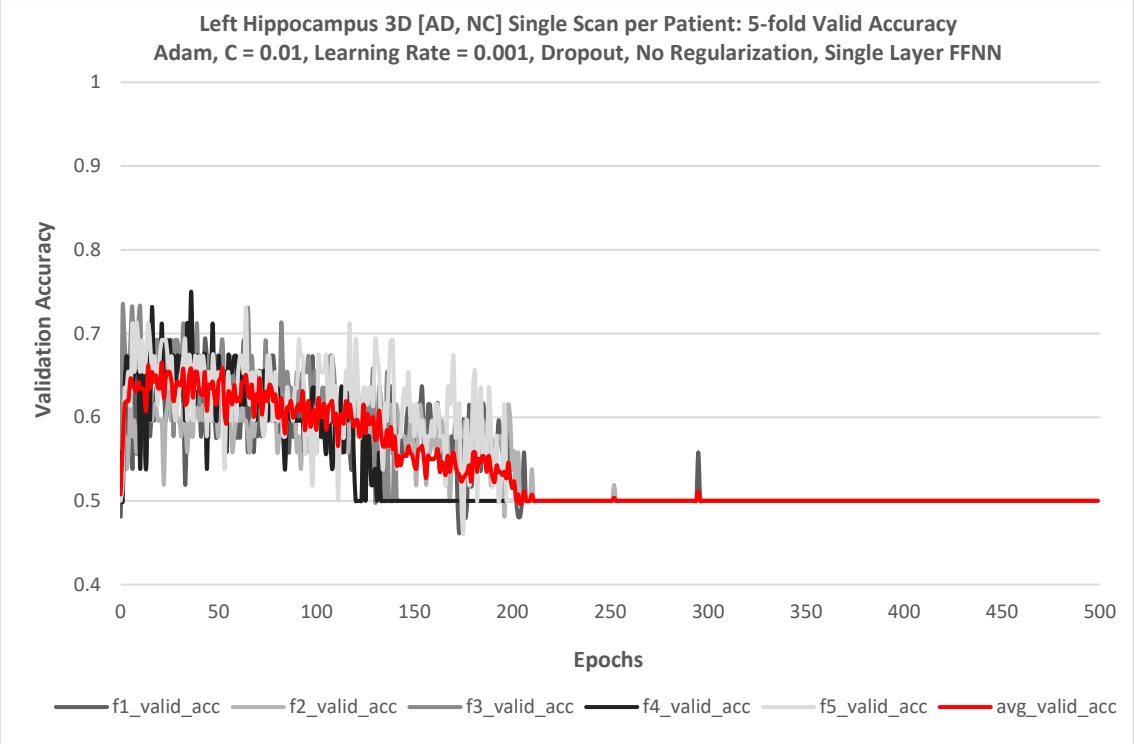


Figure L.6. Caption in the title.

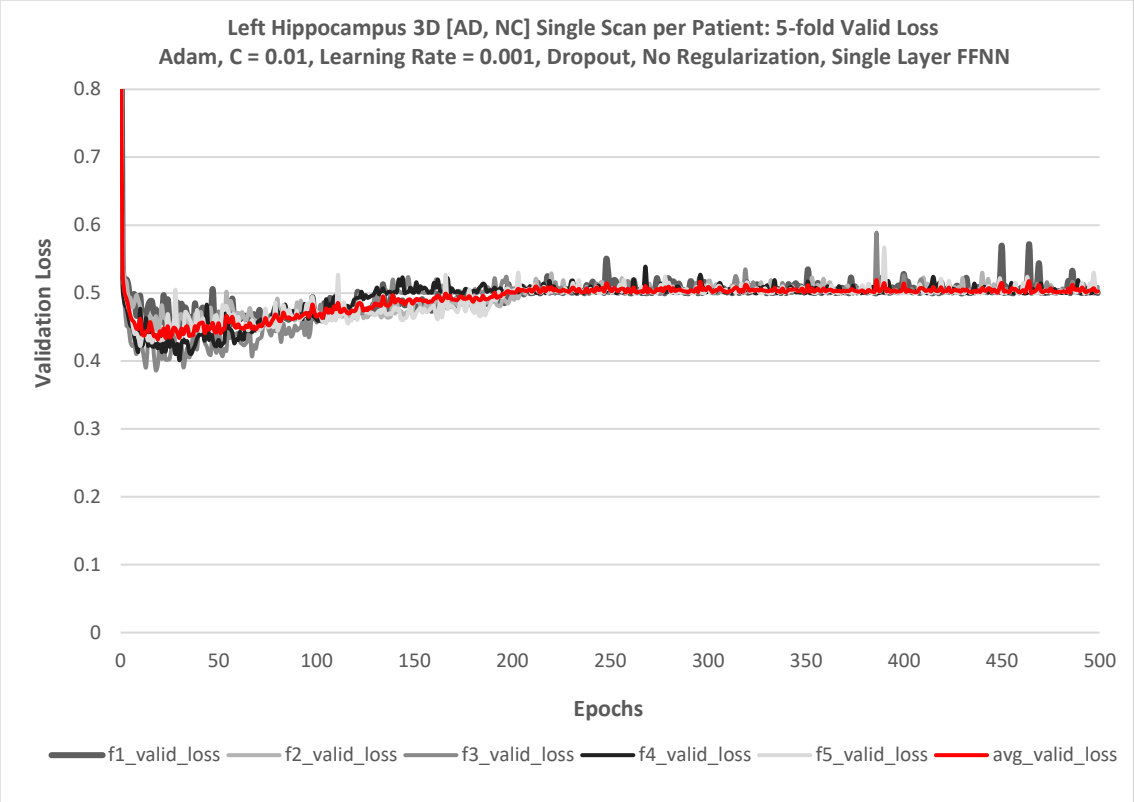


Figure L.7. Caption in the title.

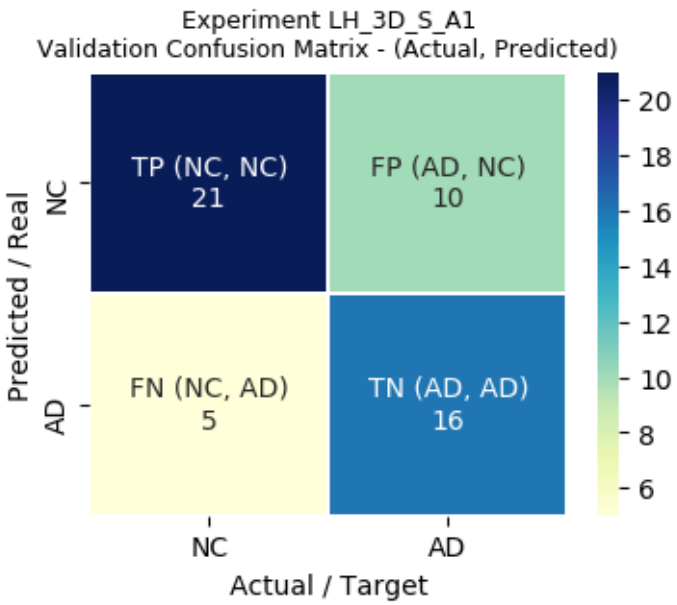


Figure L.8. Caption in the title.

L.1.1.4 Testing

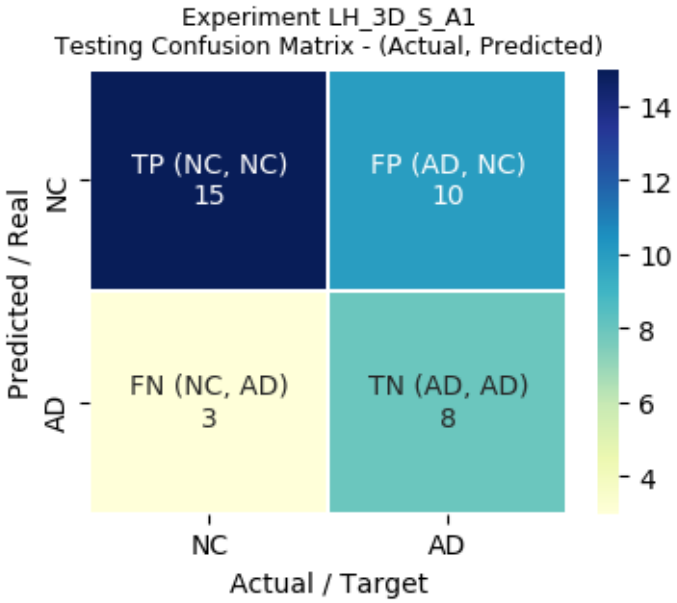


Figure L.9. Caption in the title.

L.1.2 Experiment LH_3D_S_A2: C = 0.01, lr = 0.0001, Dropout

L.1.2.1 General

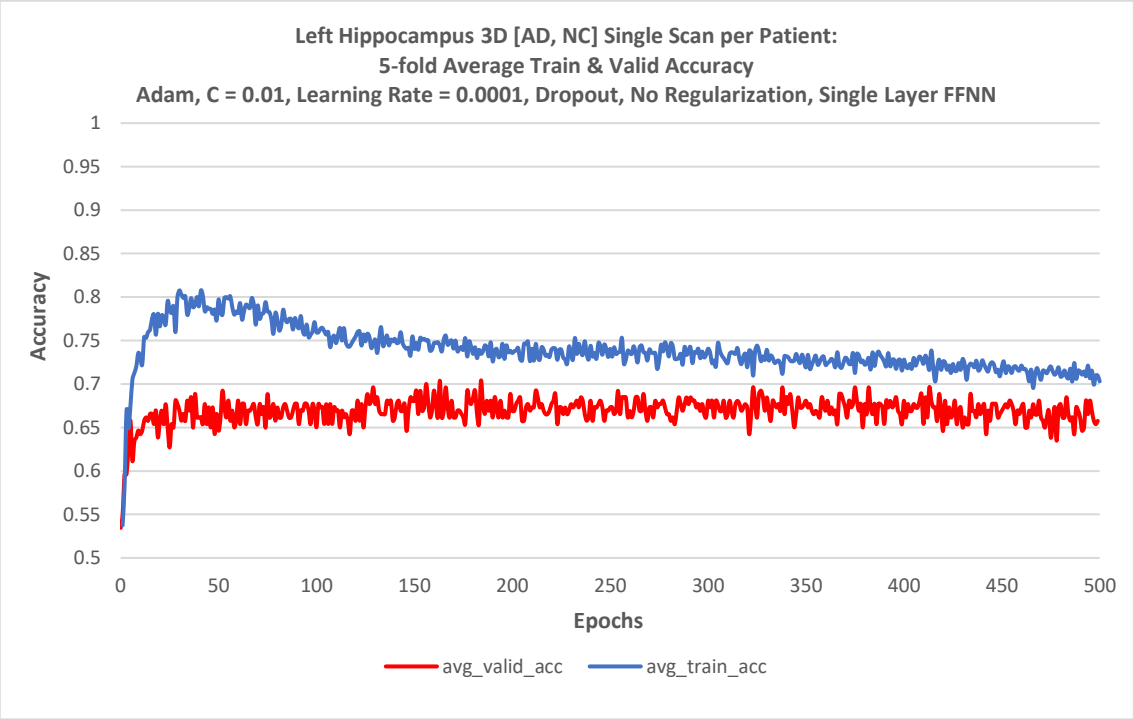


Figure L.10. Caption in the title.

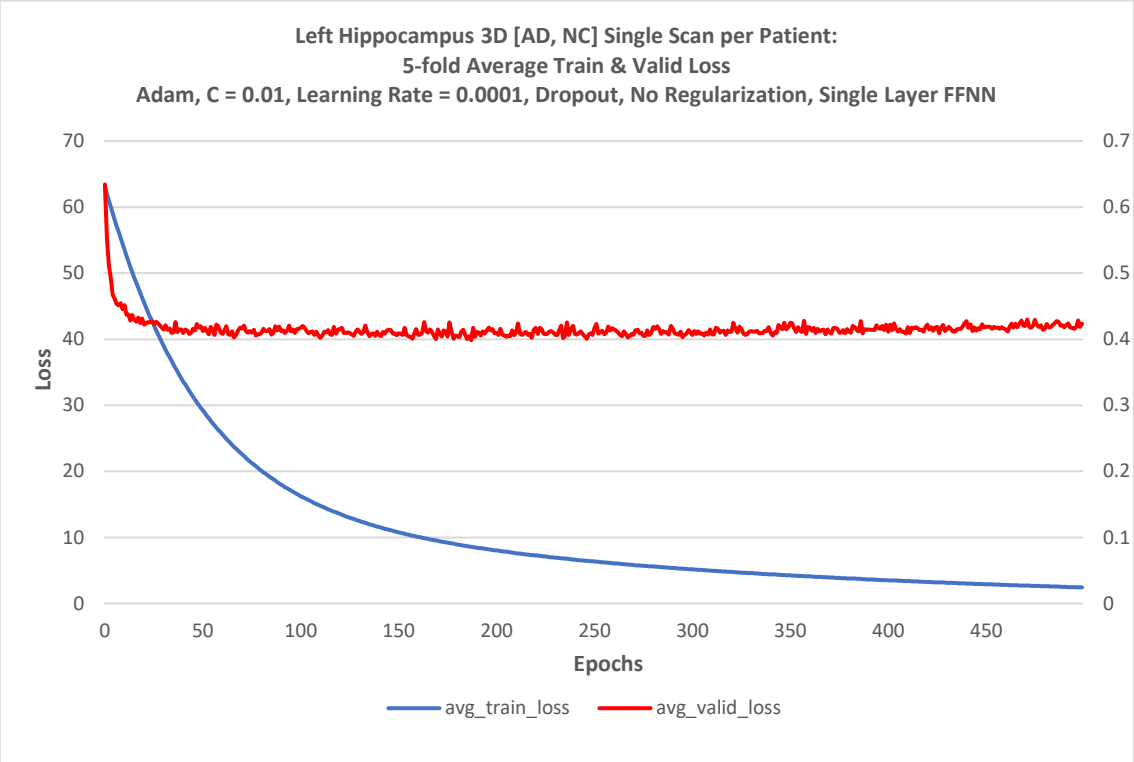


Figure L.11. Caption in the title.

L.1.2.2 Training

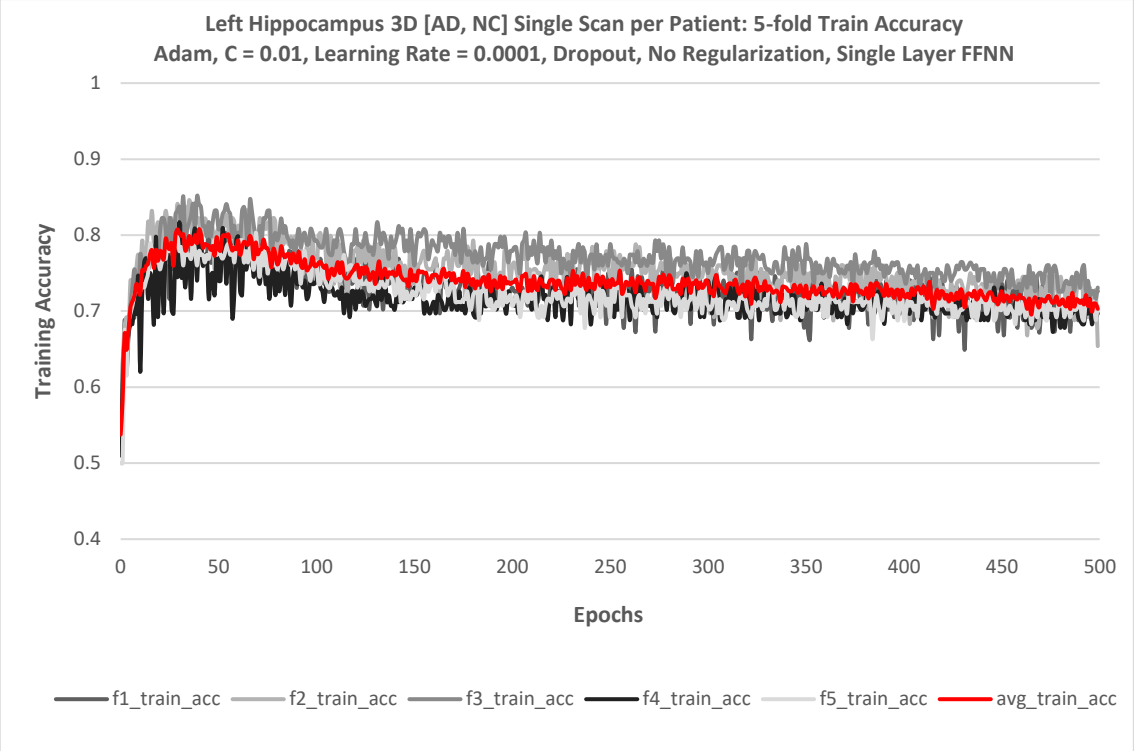


Figure L.12. Caption in the title.

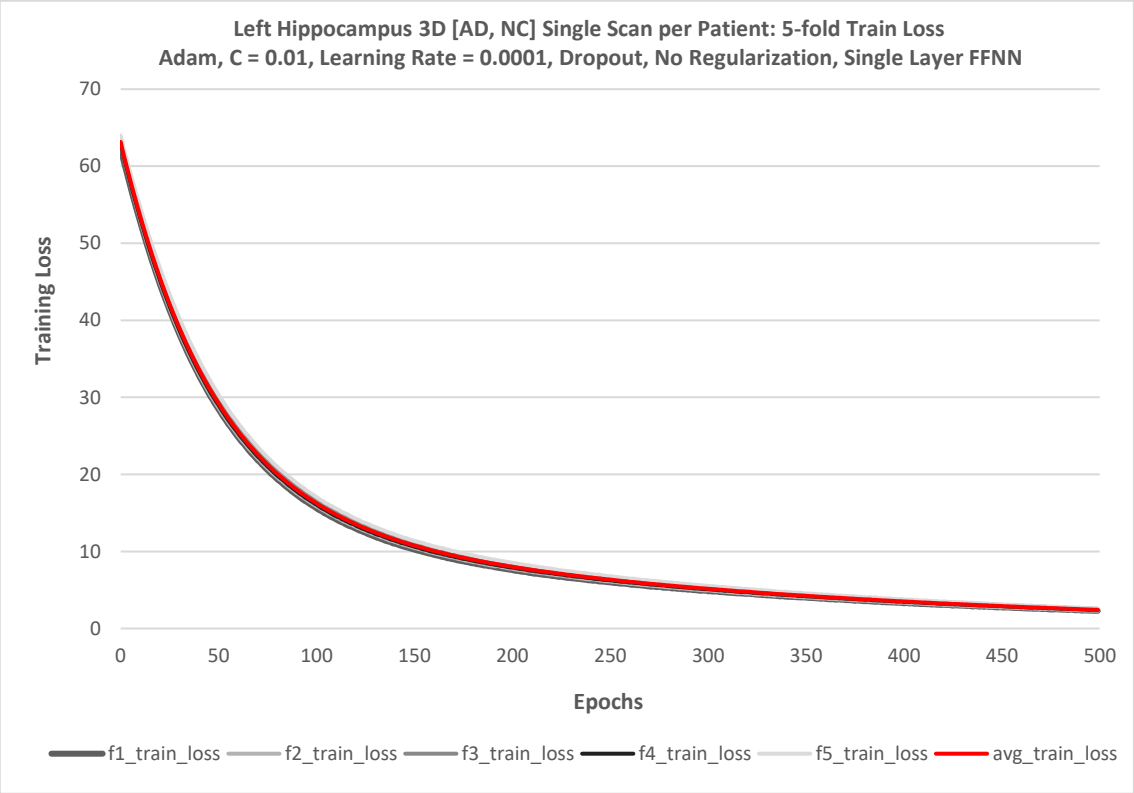


Figure L.13. Caption in the title.

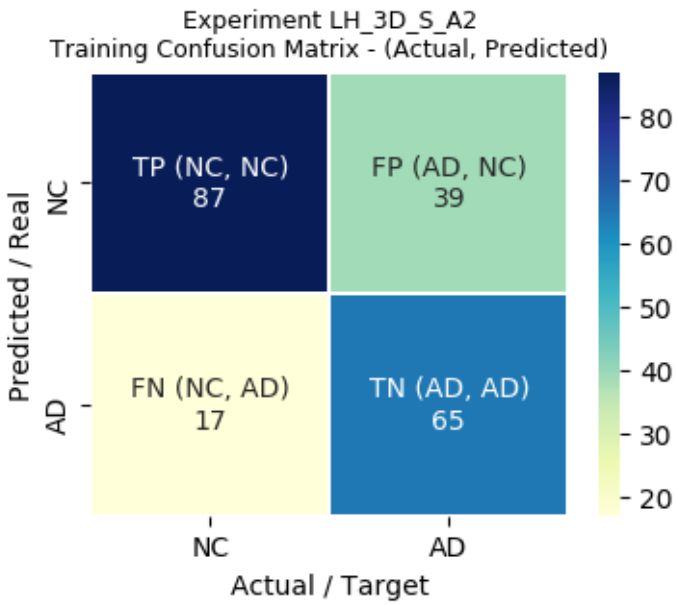


Figure L.14. Caption in the title.

L.1.2.3 Validation

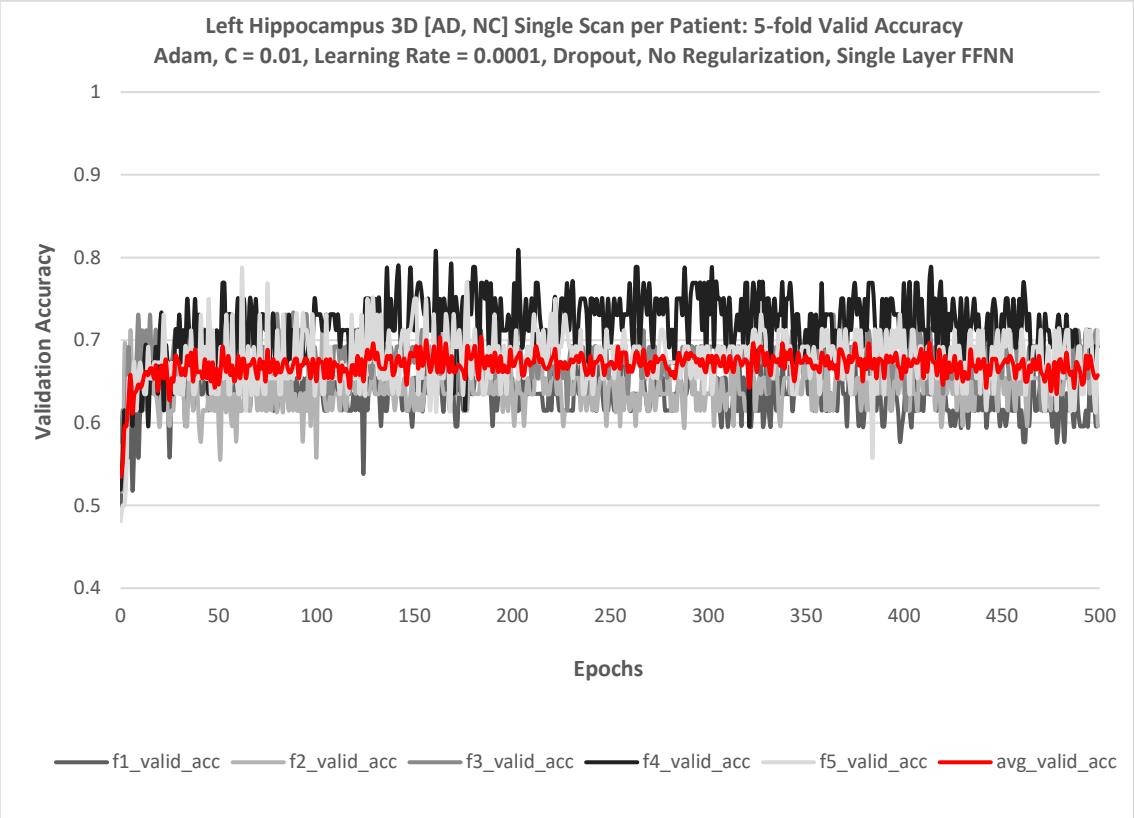


Figure L.15. Caption in the title.

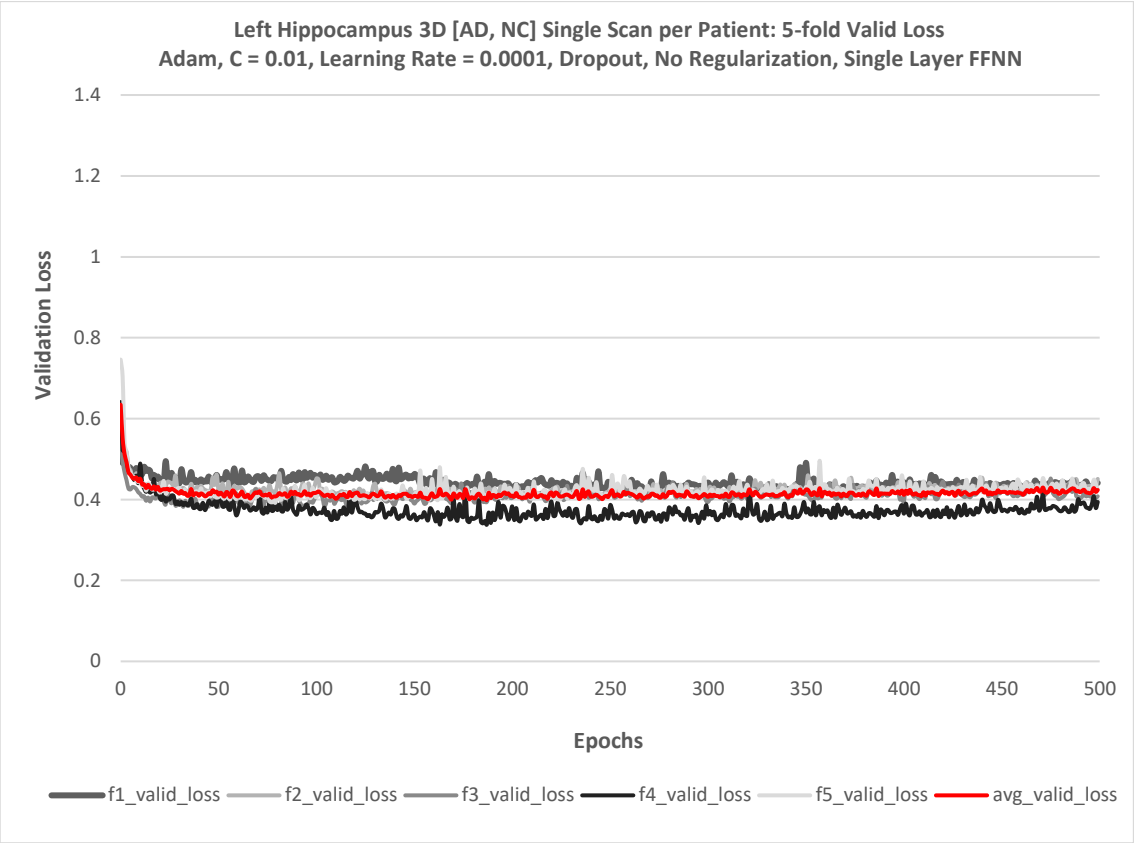


Figure L.16. Caption in the title.

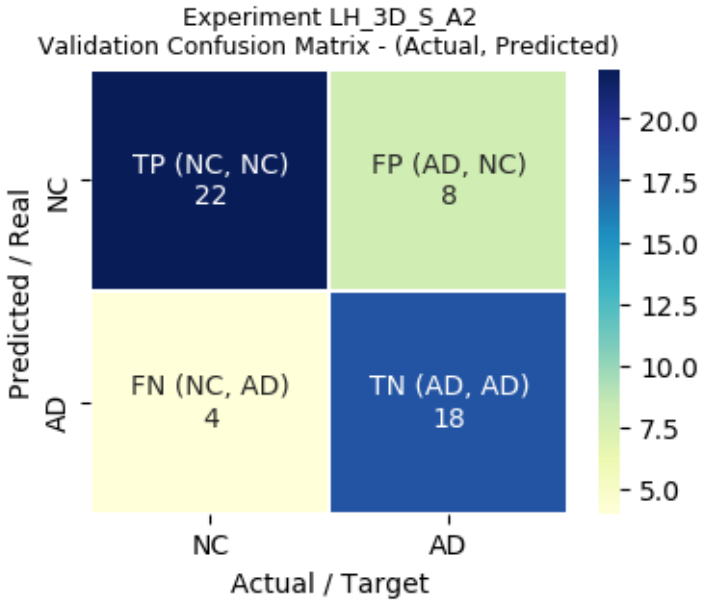


Figure L.17. Caption in the title.

L.1.2.4 Testing

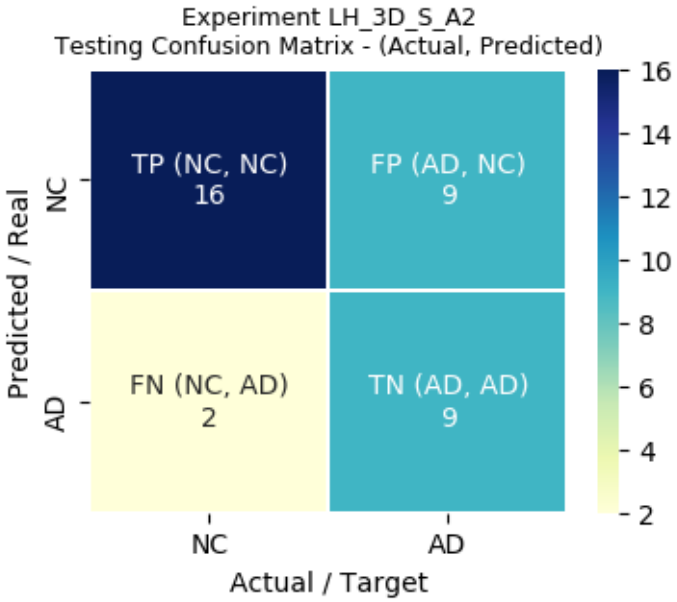


Figure L.18. Caption in the title.

L.1.3 Experiment LH_3D_S_A3: C = 0.01, lr = 0.00001, Dropout

L.1.3.1 General

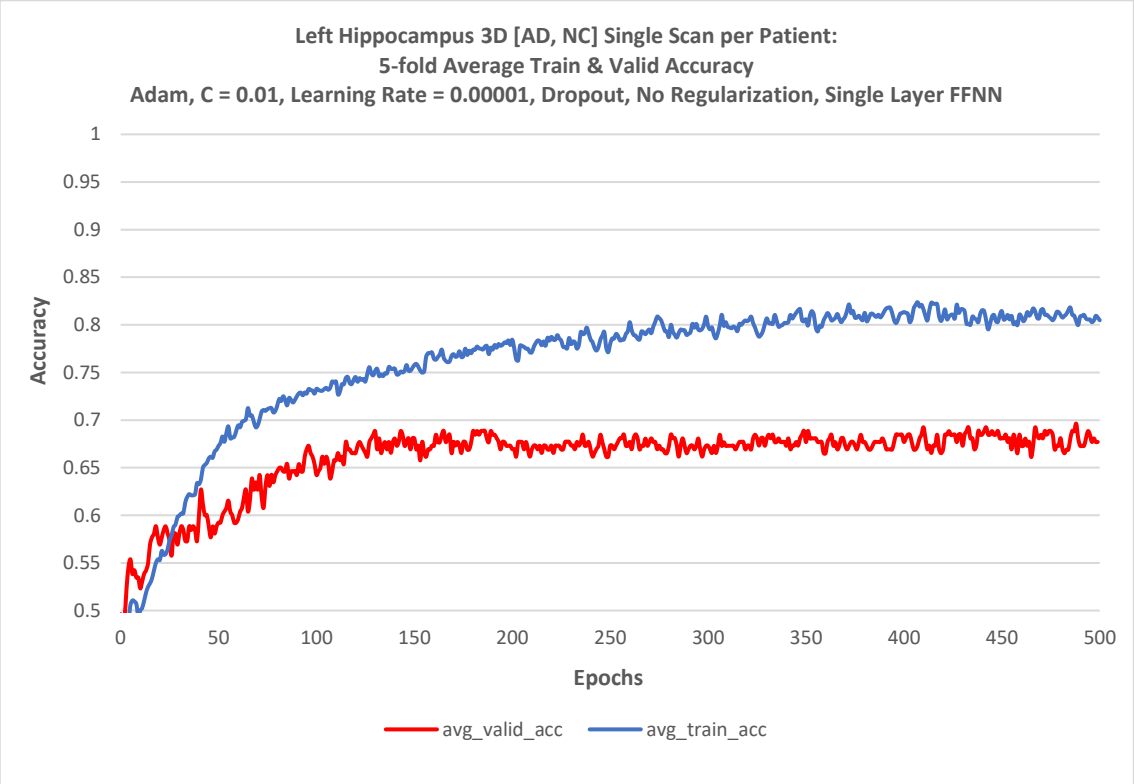


Figure L.19. Caption in the title.

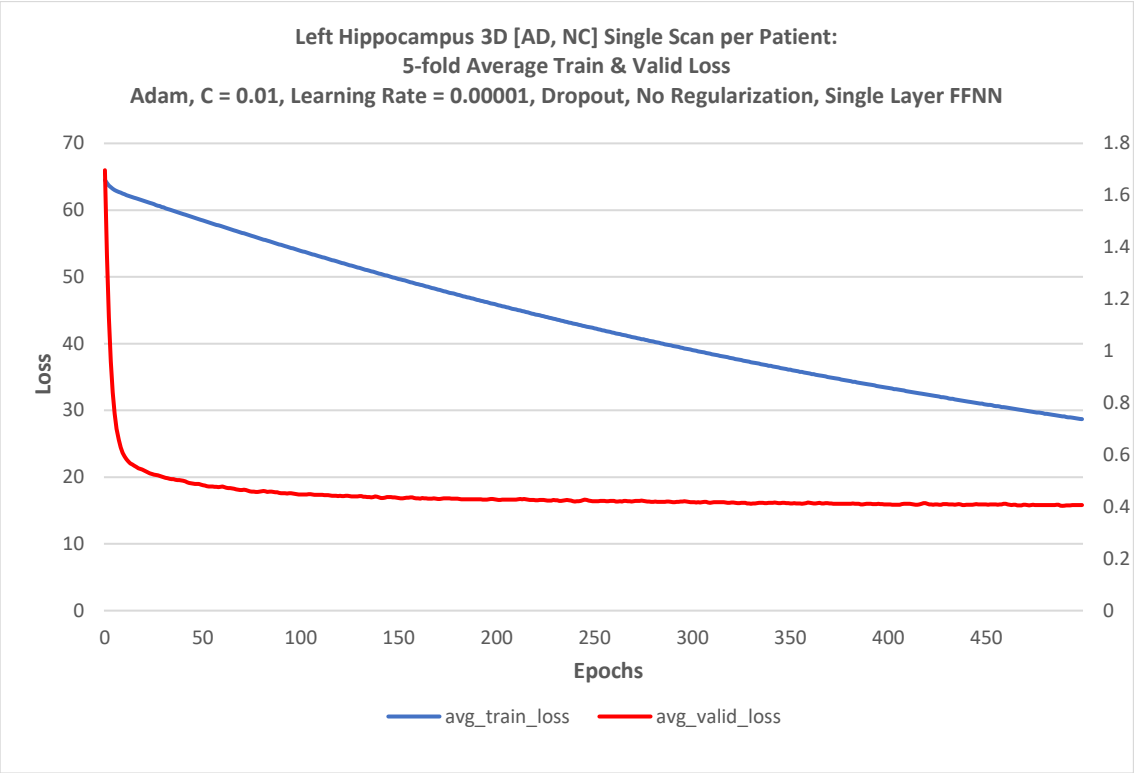


Figure L.20. Caption in the title.

L.1.3.2 Training

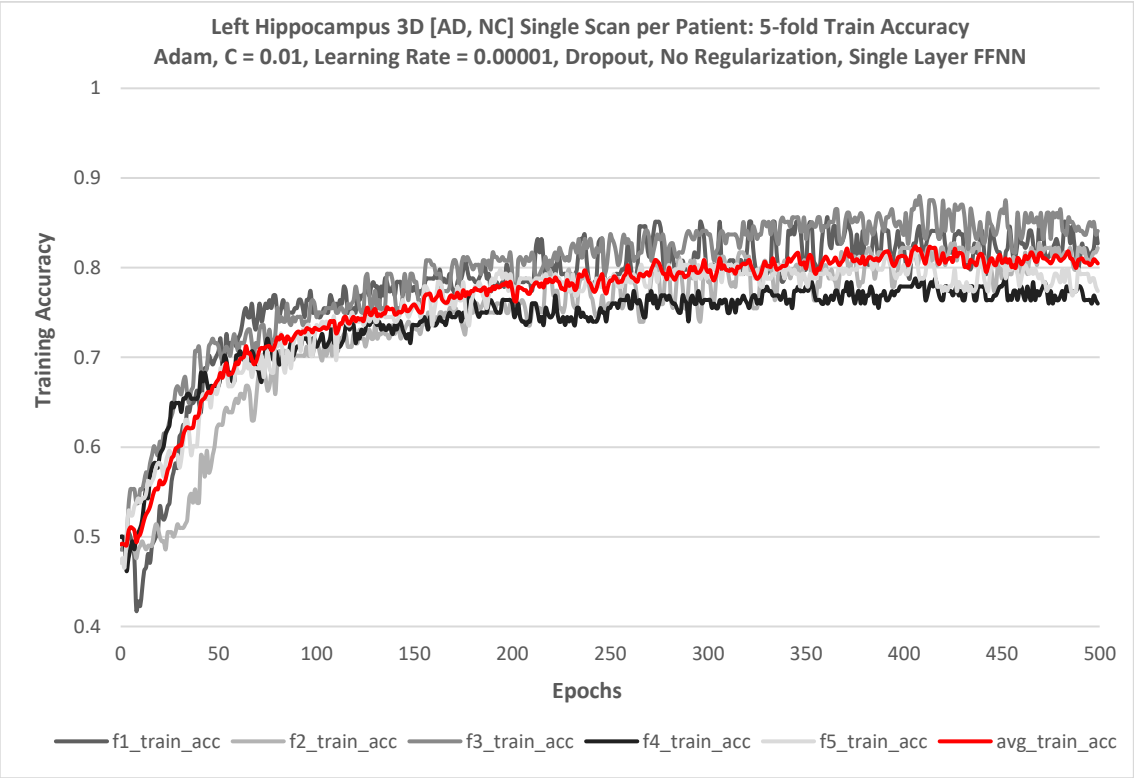


Figure L.21. Caption in the title.

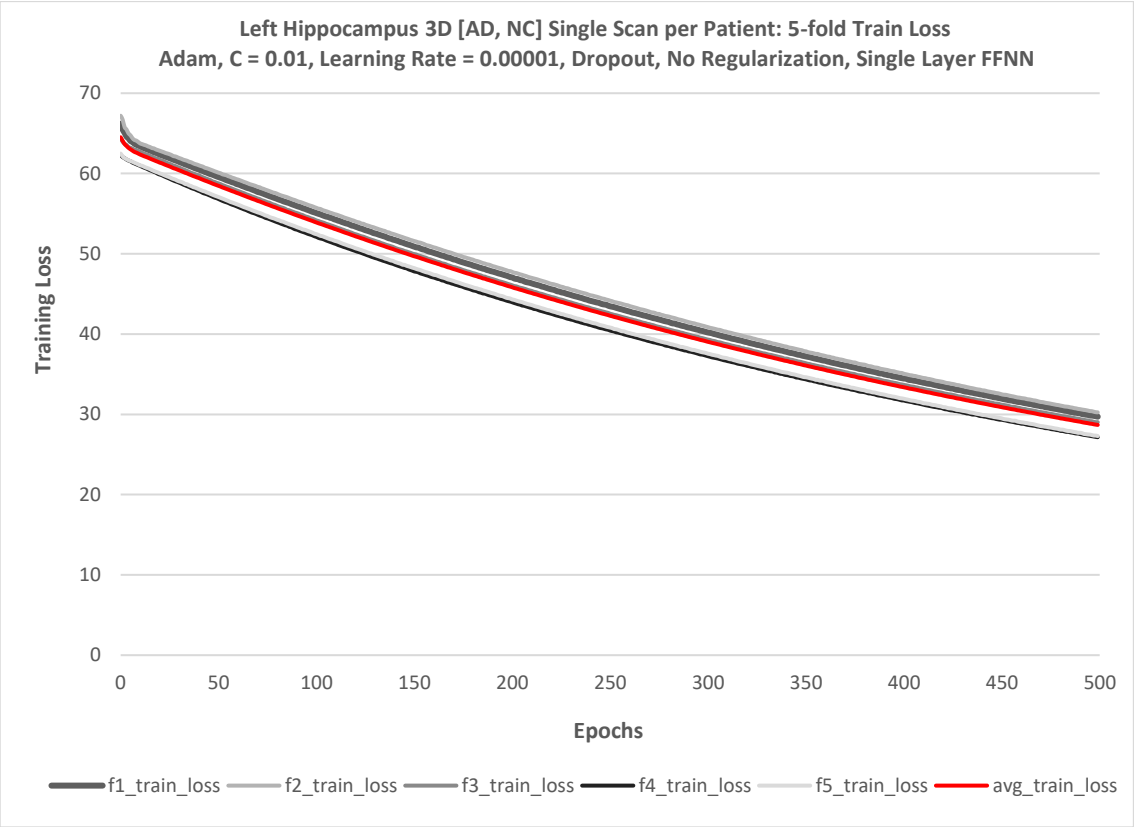


Figure L.22. Caption in the title.

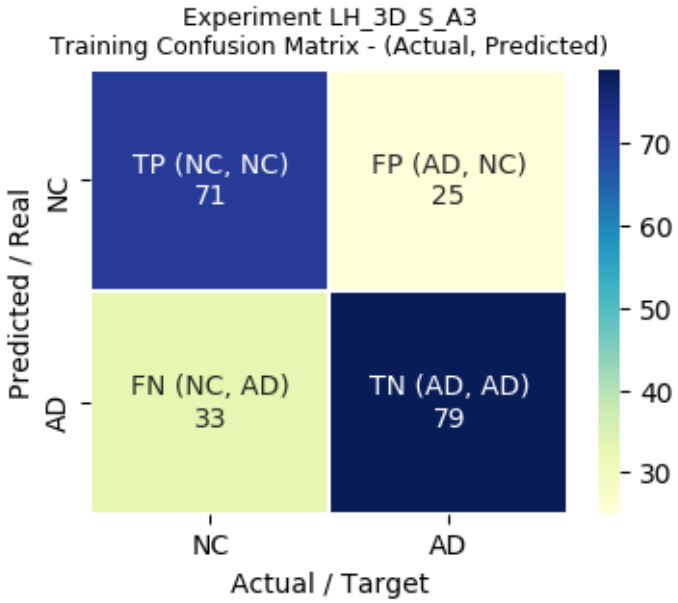


Figure L.23. Caption in the title.

L.1.3.3 Validation

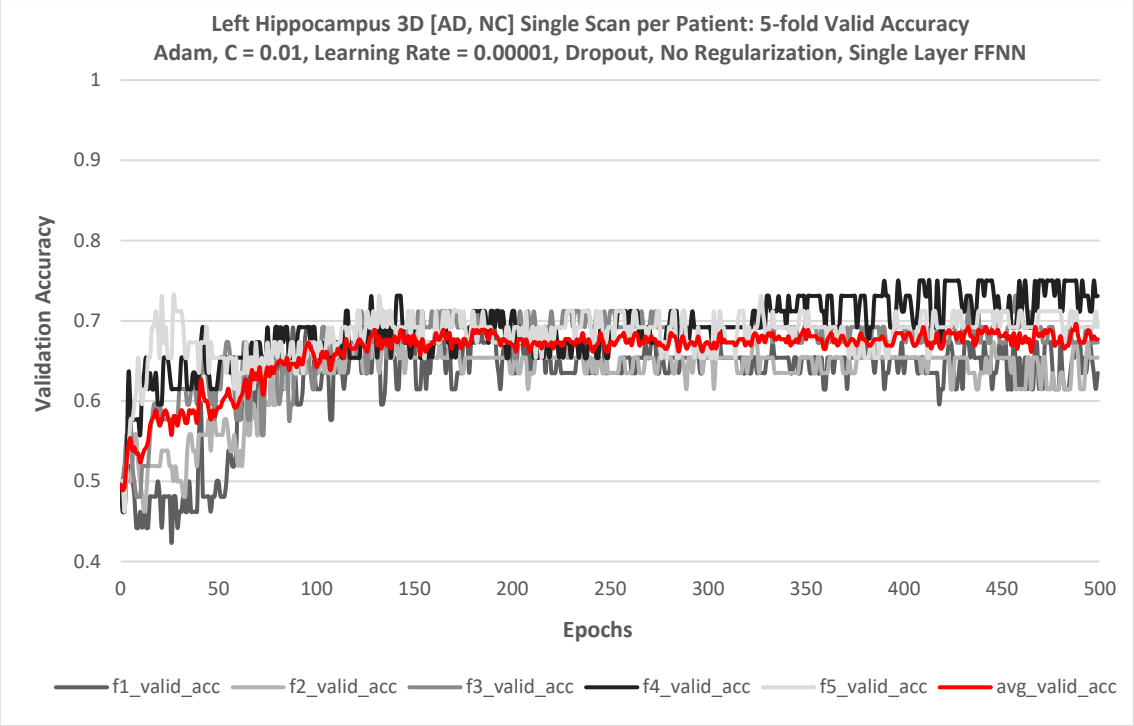


Figure L.24. Caption in the title.

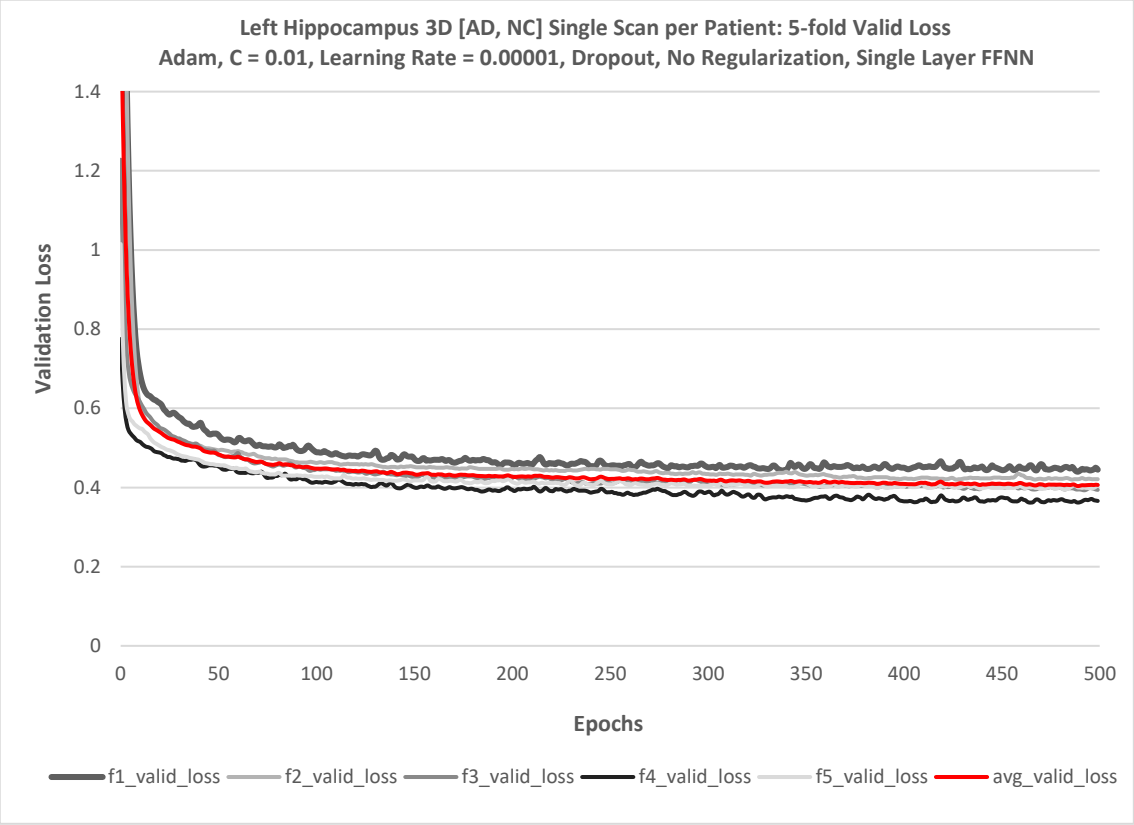


Figure L.25. Caption in the title.

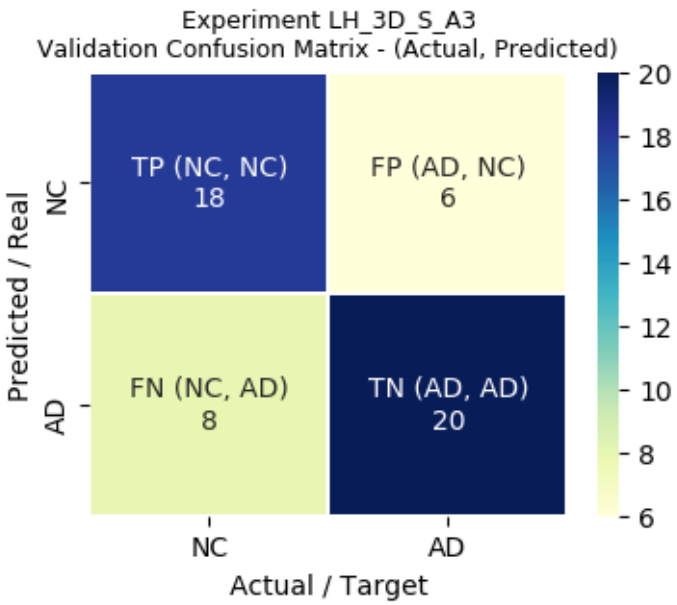


Figure L.26. Caption in the title.

L.1.3.4 Testing

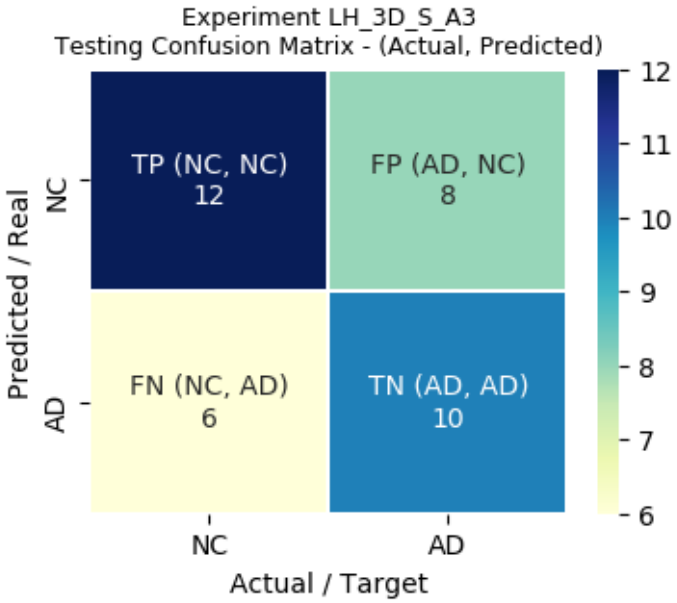


Figure L.27. Caption in the title.

L.1.4 Experiment LH_3D_S_A4: C = 0.01, lr = 0.0001, Dropout, No Max-Pooling

L.1.4.1 General

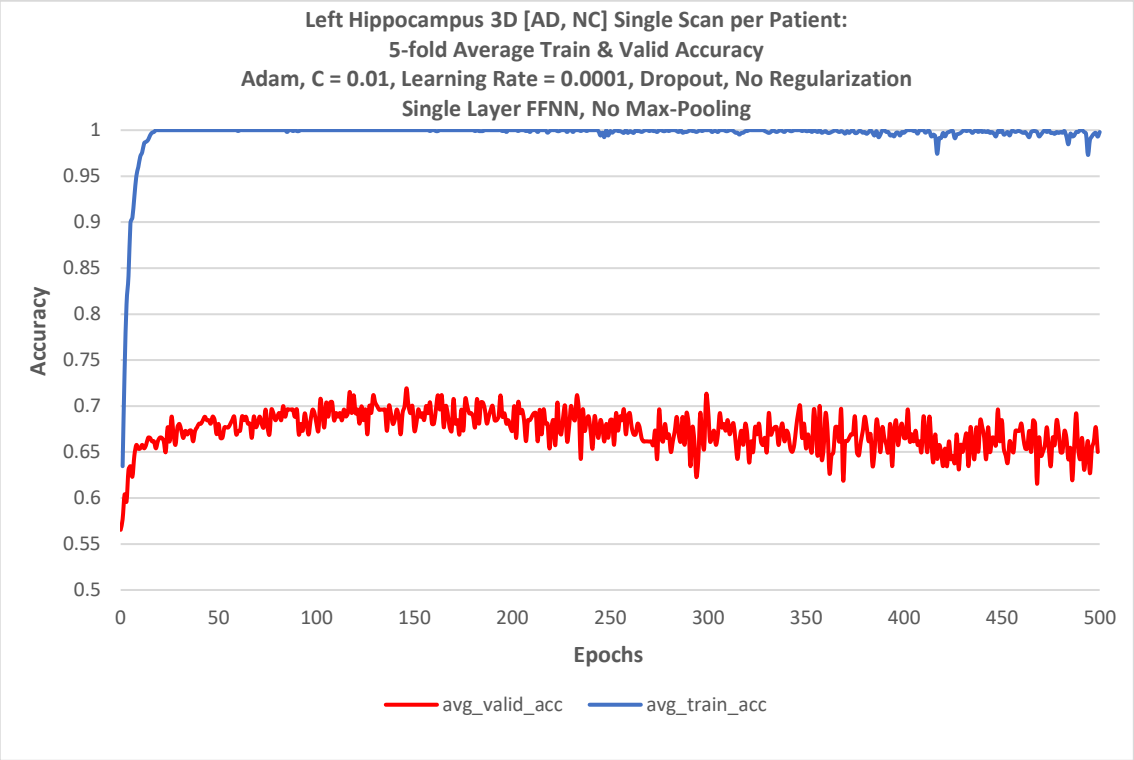


Figure L.28. Caption in the title.

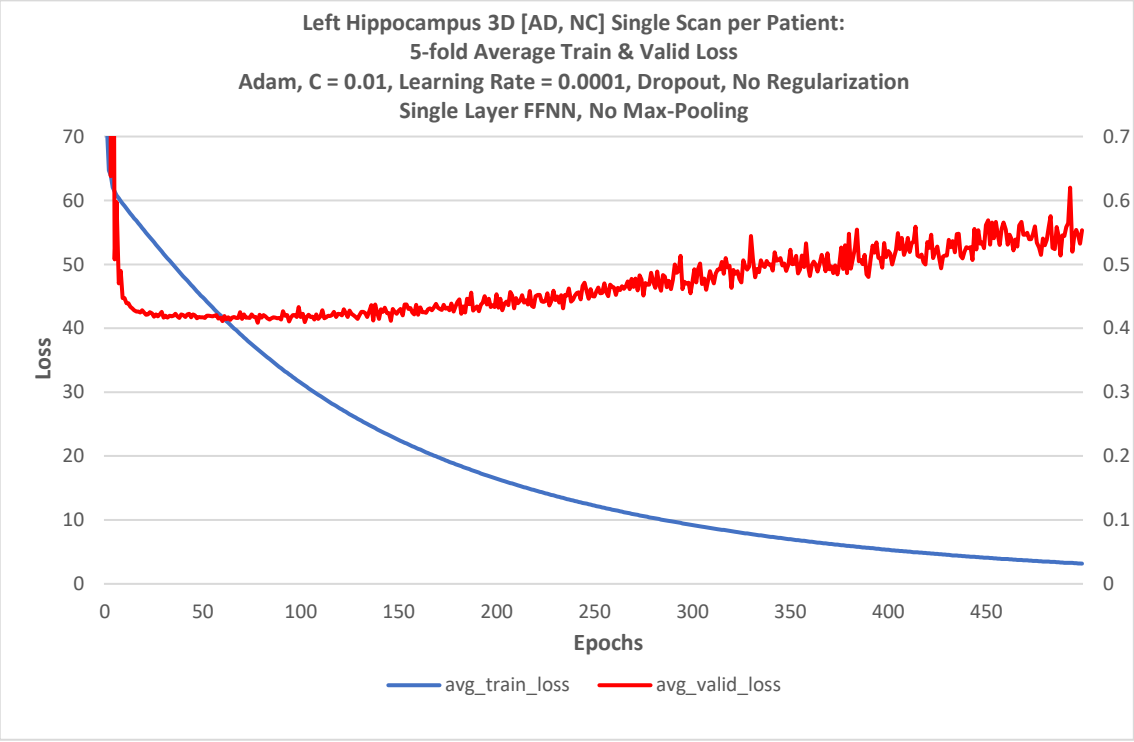


Figure L.29. Caption in the title.

L.1.4.2 Training

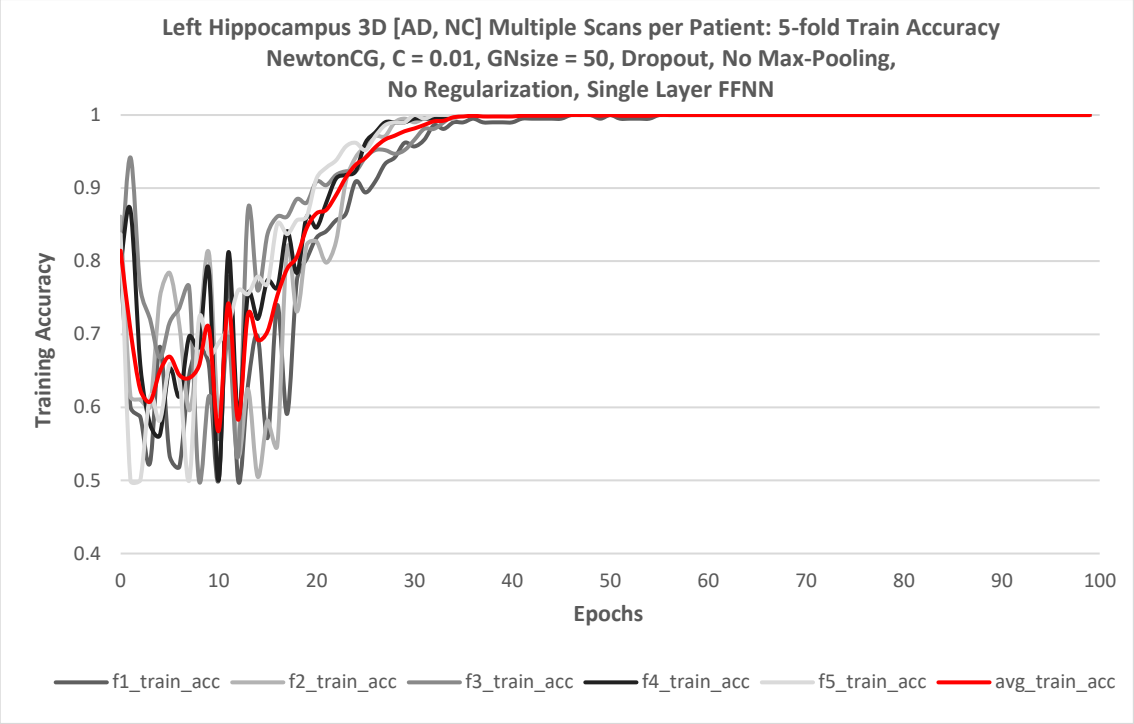


Figure L.30. Caption in the title.

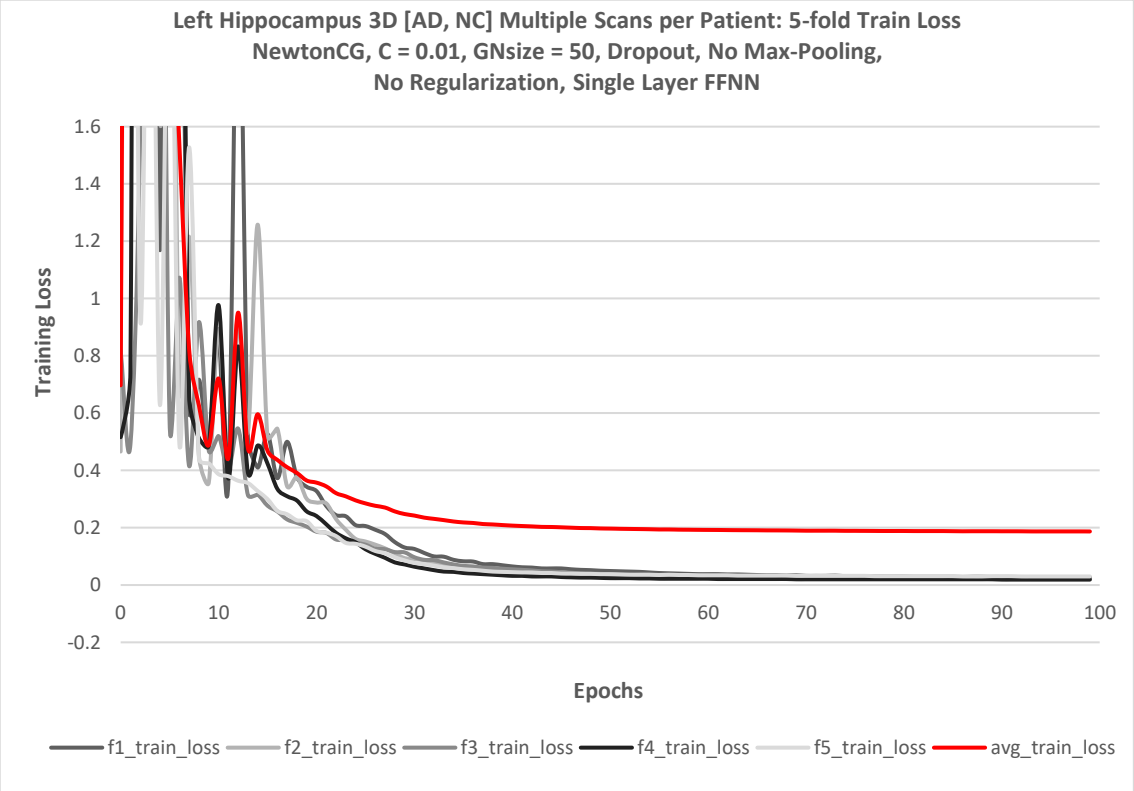


Figure L.31. Caption in the title.

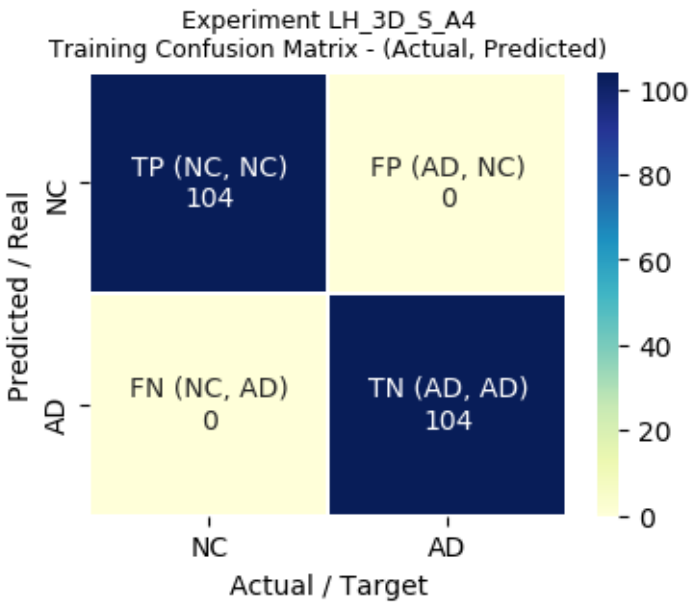


Figure L.32. Caption in the title.

L.1.4.3 Validation

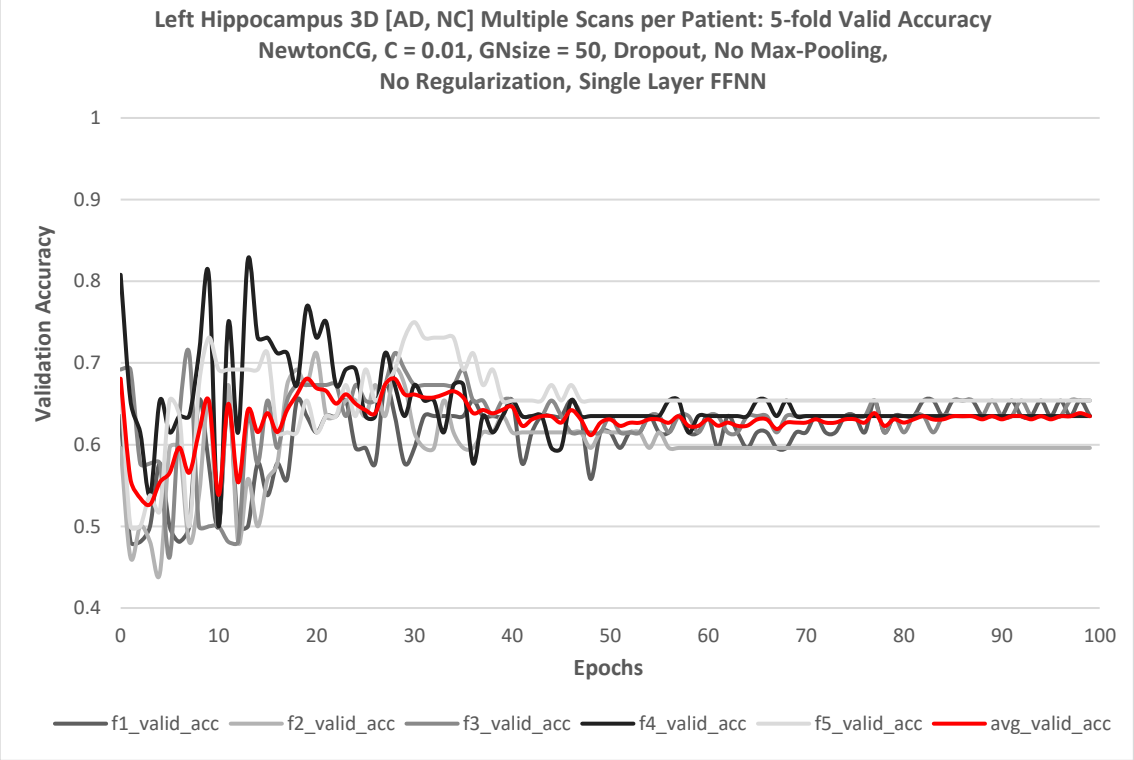


Figure L.33. Caption in the title.

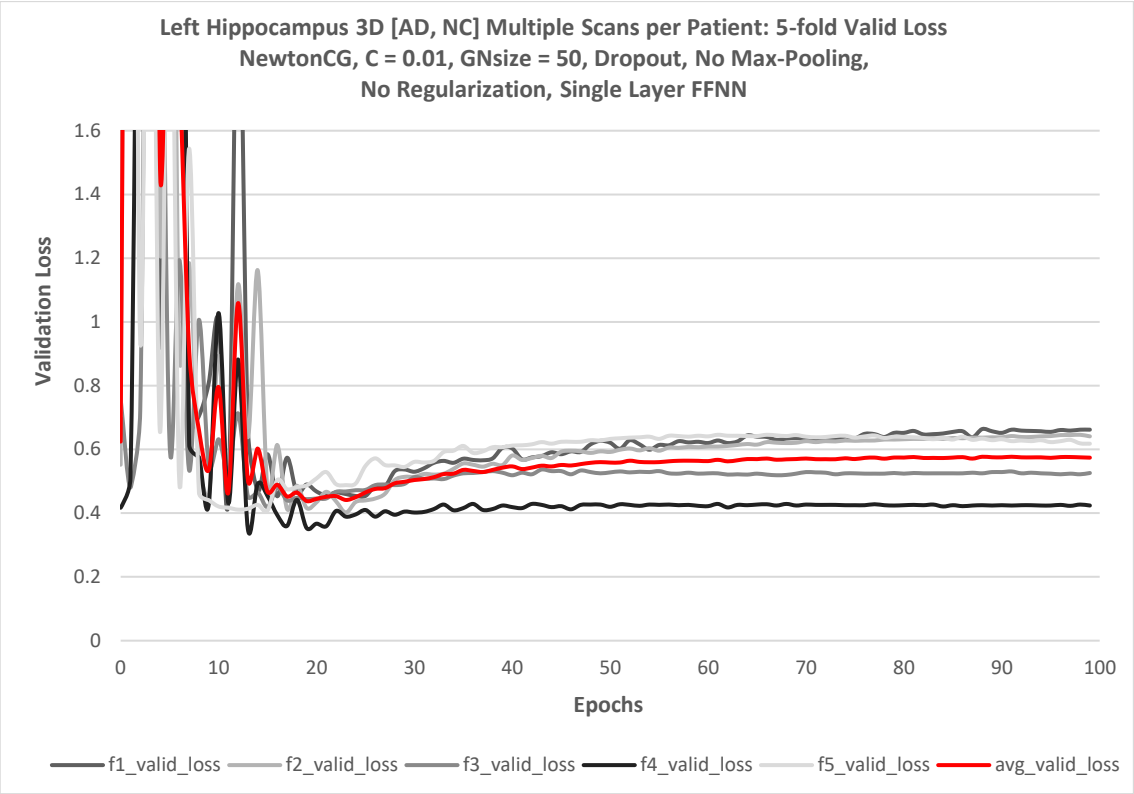


Figure L.34. Caption in the title.

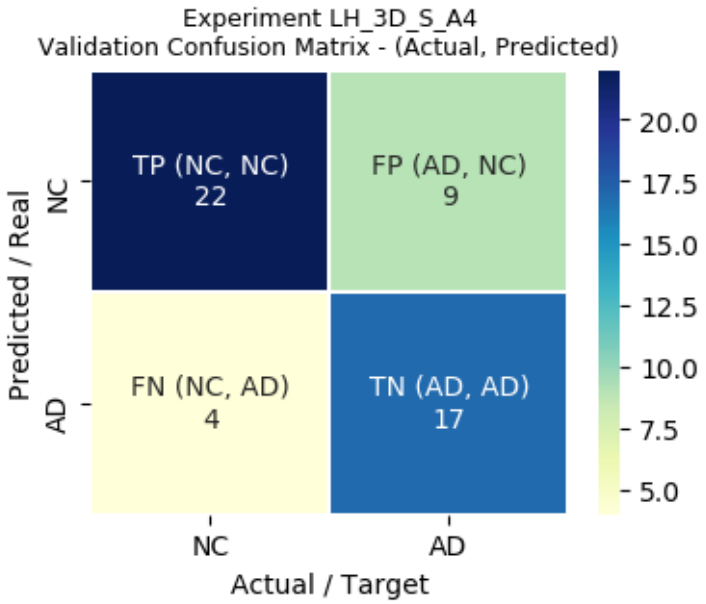


Figure L.35. Caption in the title.

L.1.4.4 Testing

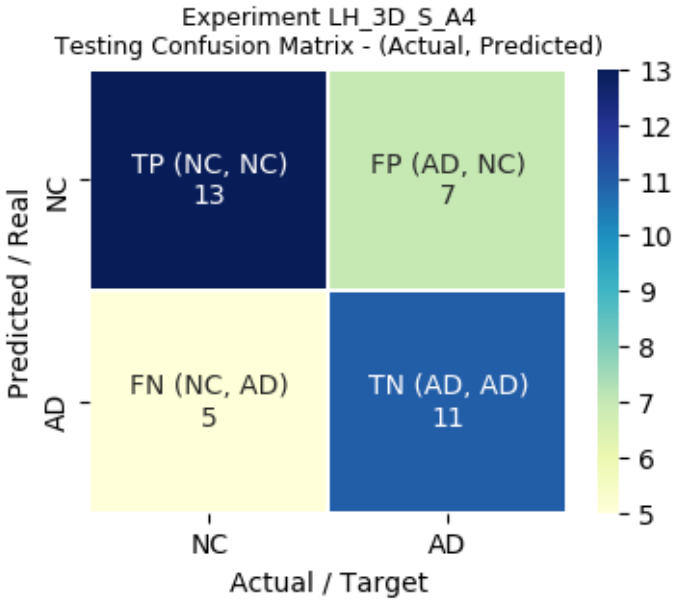


Figure L.36. Caption in the title.

L.2 CNN 4 Layer with NewtonCG

L.2.1 Experiment LH_3D_S_N1: C = 0.01, GNsize = 500, Dropout

L.2.1.1 General

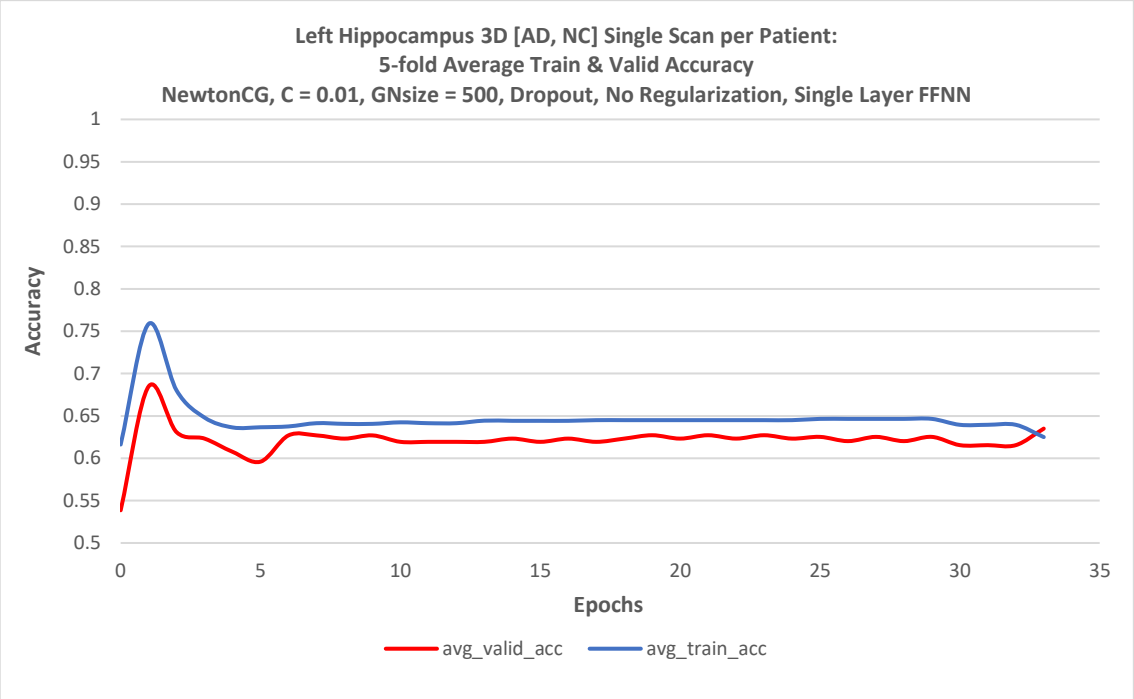


Figure L.37. Caption in the title.

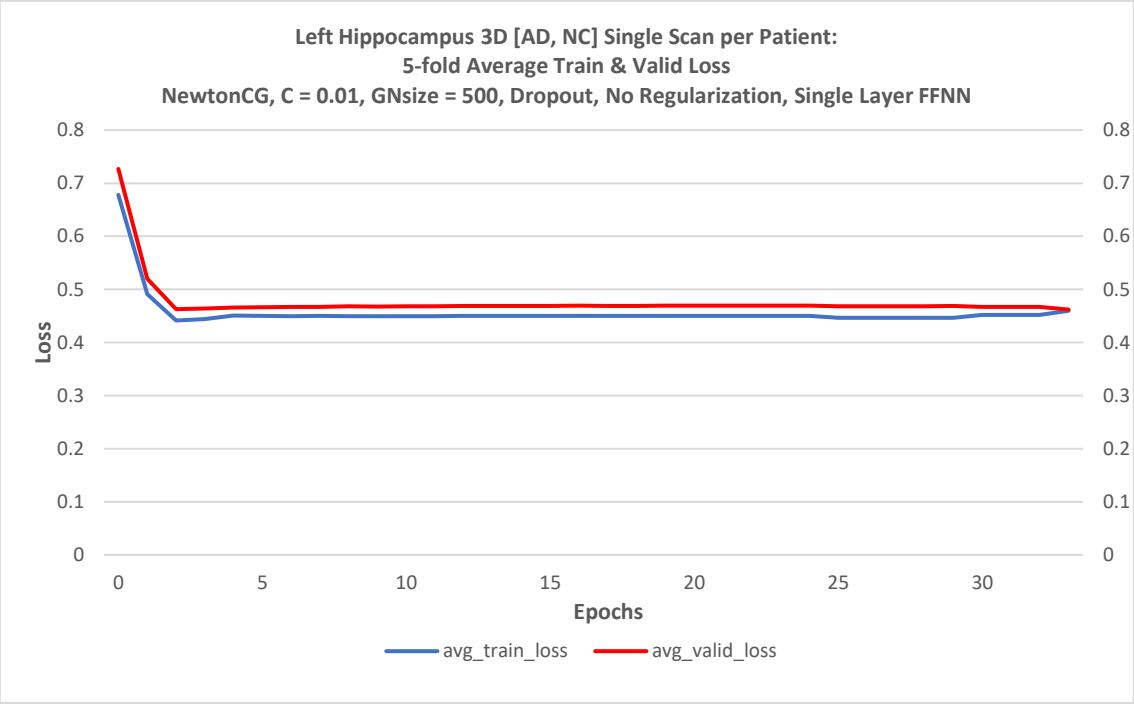


Figure L.38. Caption in the title.

L.2.1.2 Training

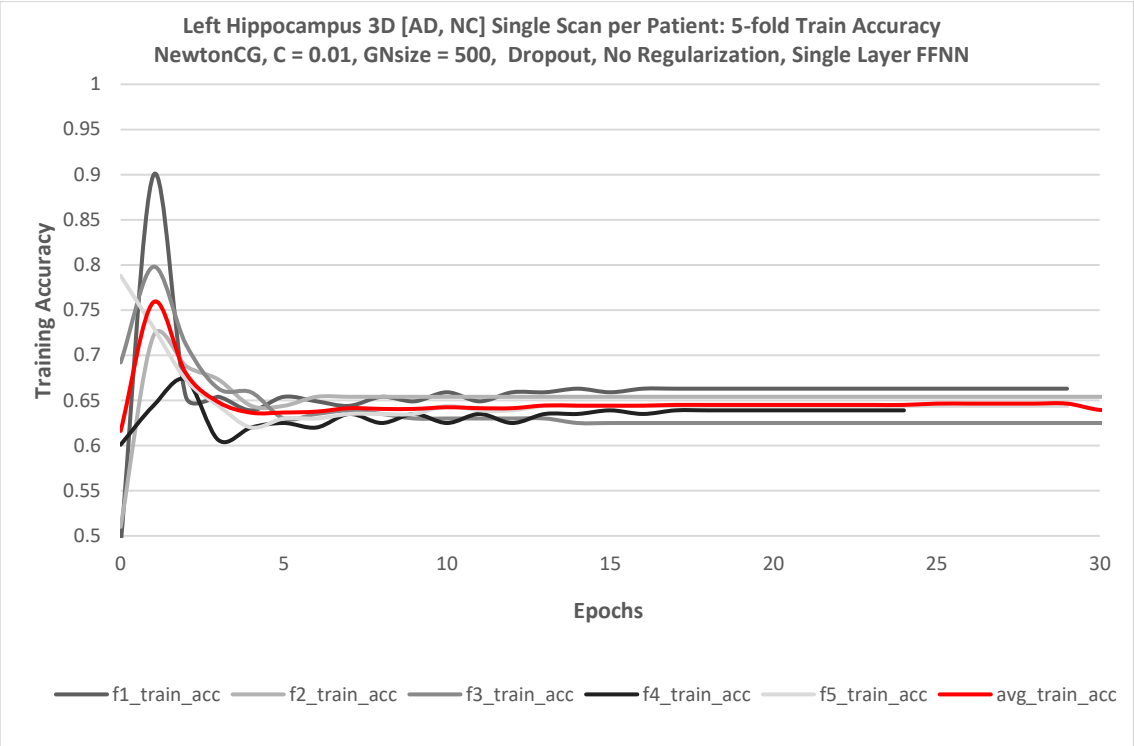


Figure L.39. Caption in the title.

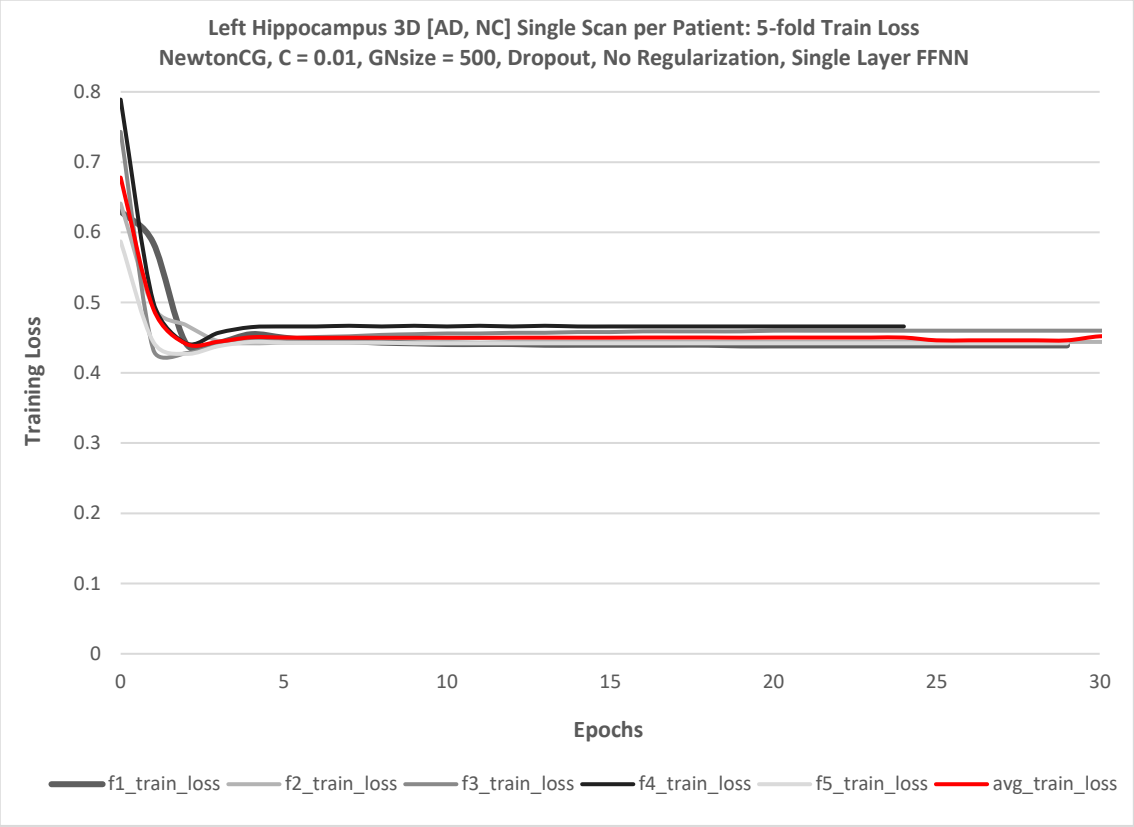


Figure L.40. Caption in the title.

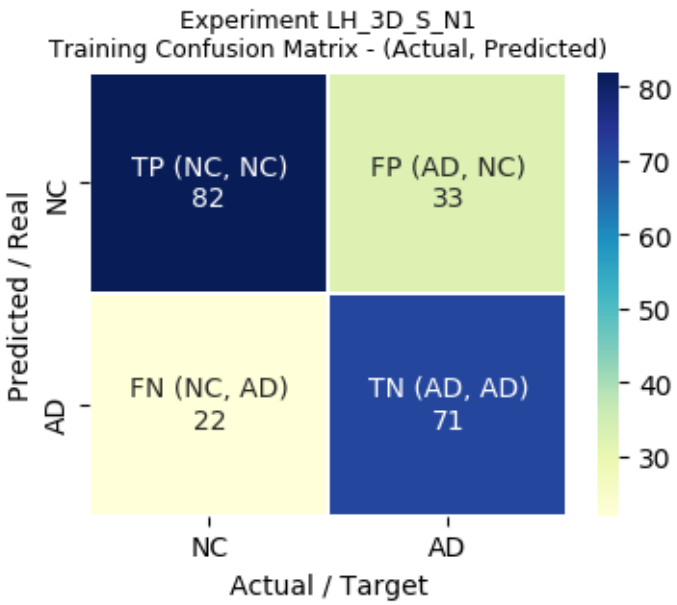


Figure L.41. Caption in the title.

L.2.1.3 Validation

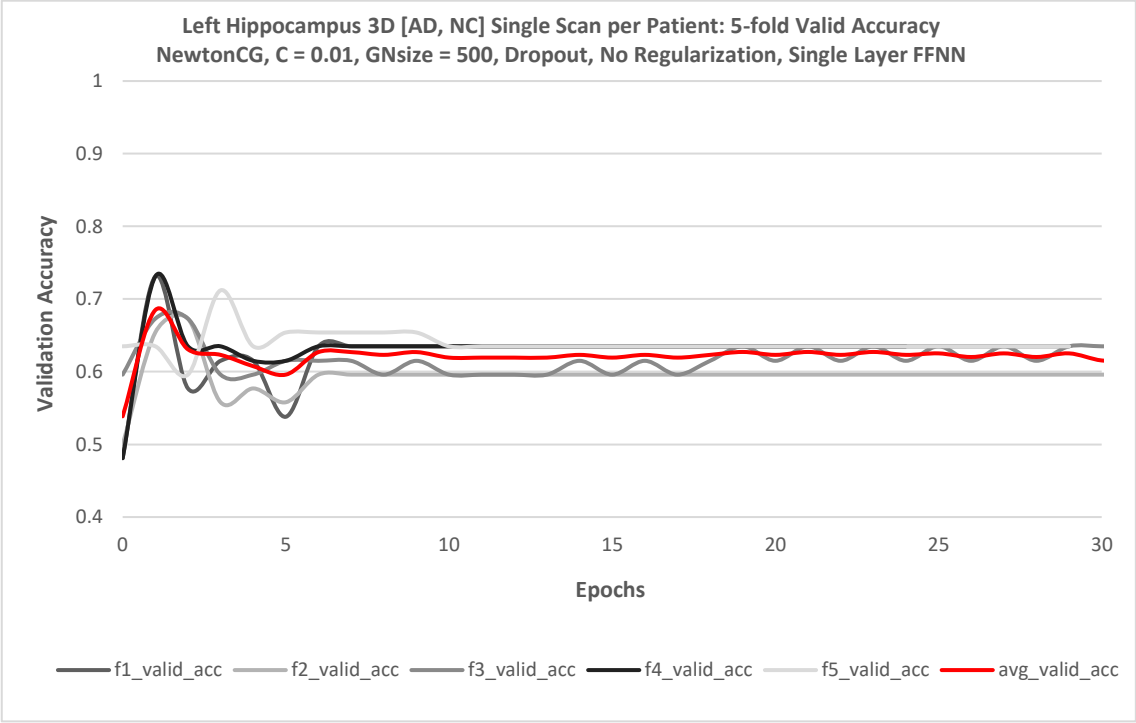


Figure L.42. Caption in the title.

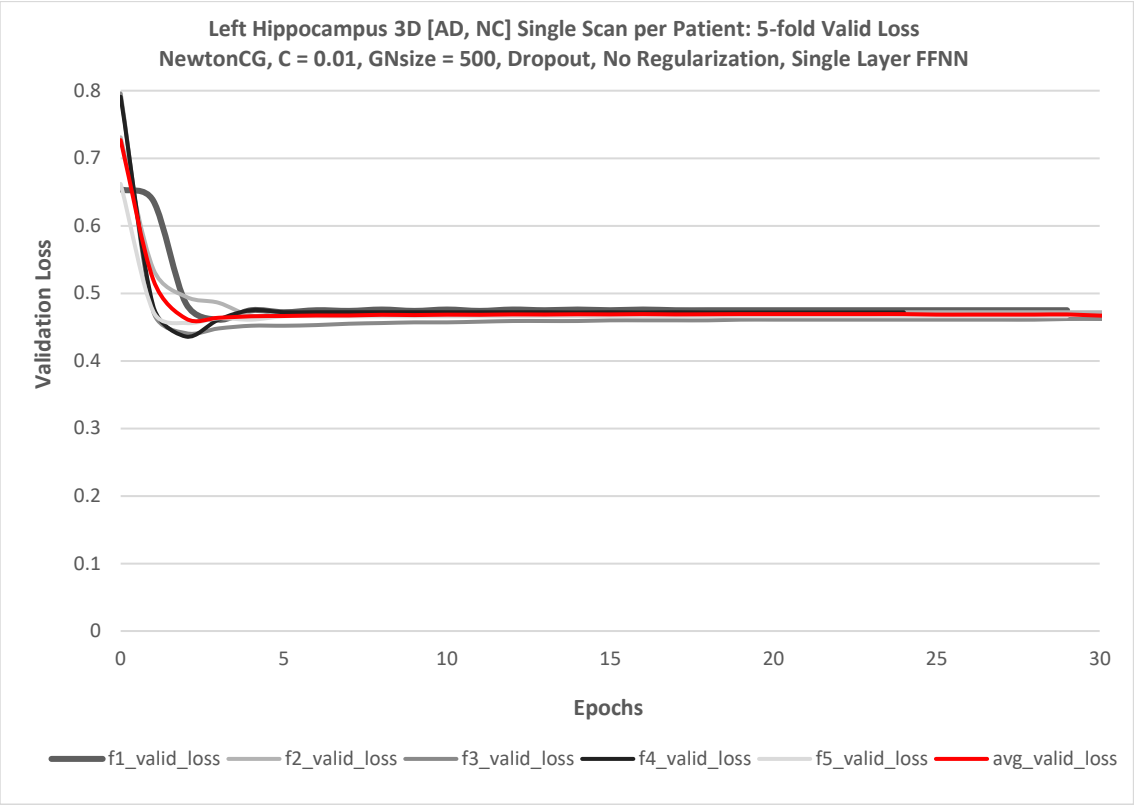


Figure L.43. Caption in the title.

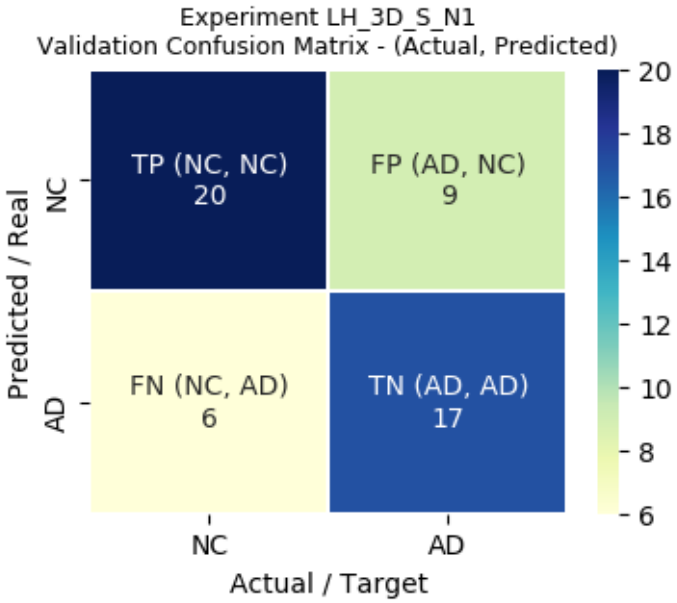


Figure L.44. Caption in the title.

L.2.1.4 Testing

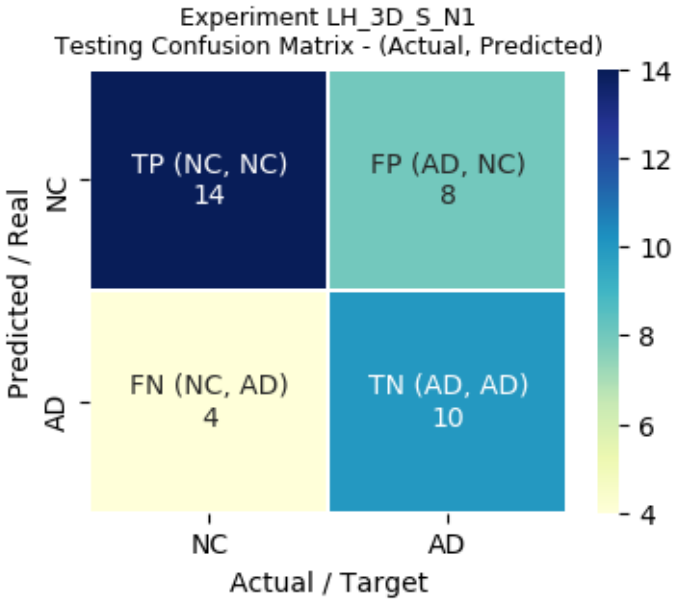


Figure L.45. Caption in the title.

L.2.2 Experiment LH_3D_S_N2: C = 0.01, GNsize = 50, Dropout, No Max-Pooling

L.2.2.1 General

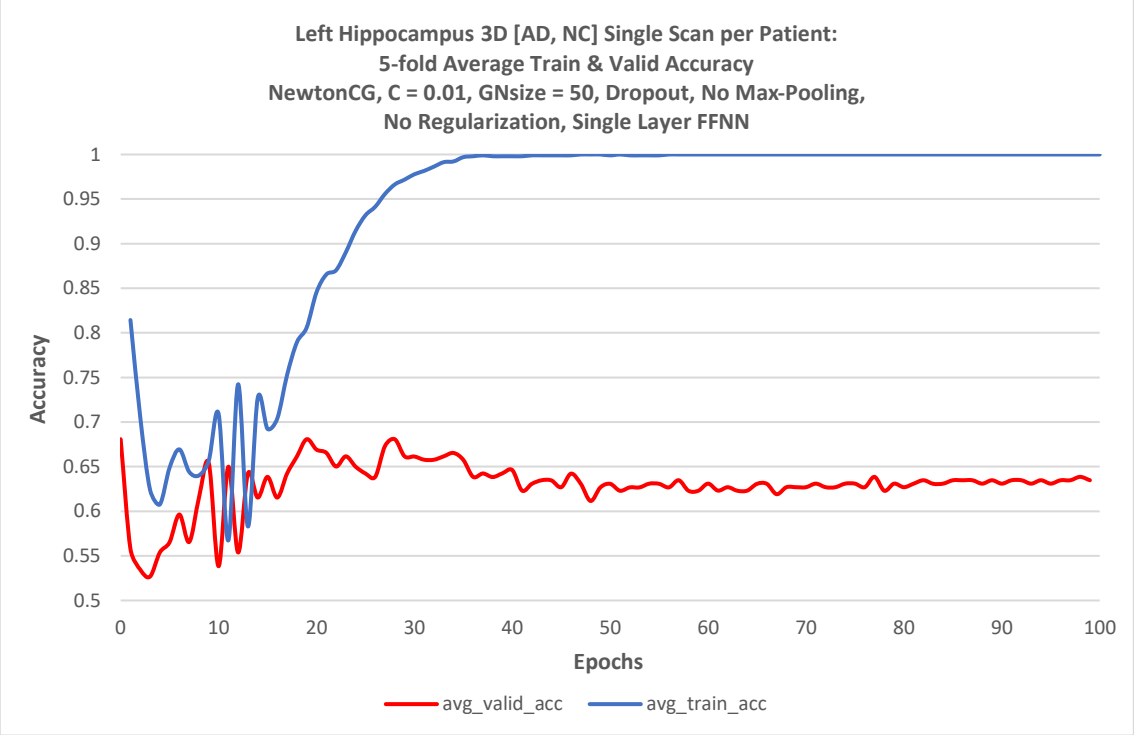


Figure L.46. Caption in the title.

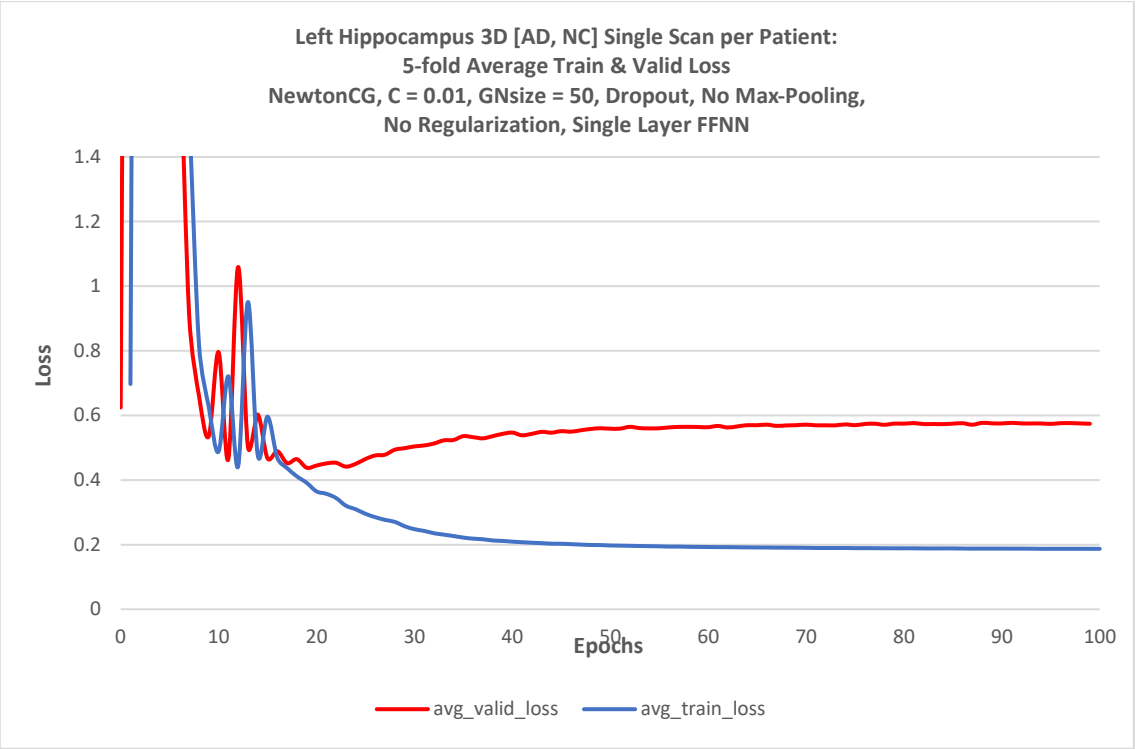


Figure L.47. Caption in the title.

L.2.2.2 Training

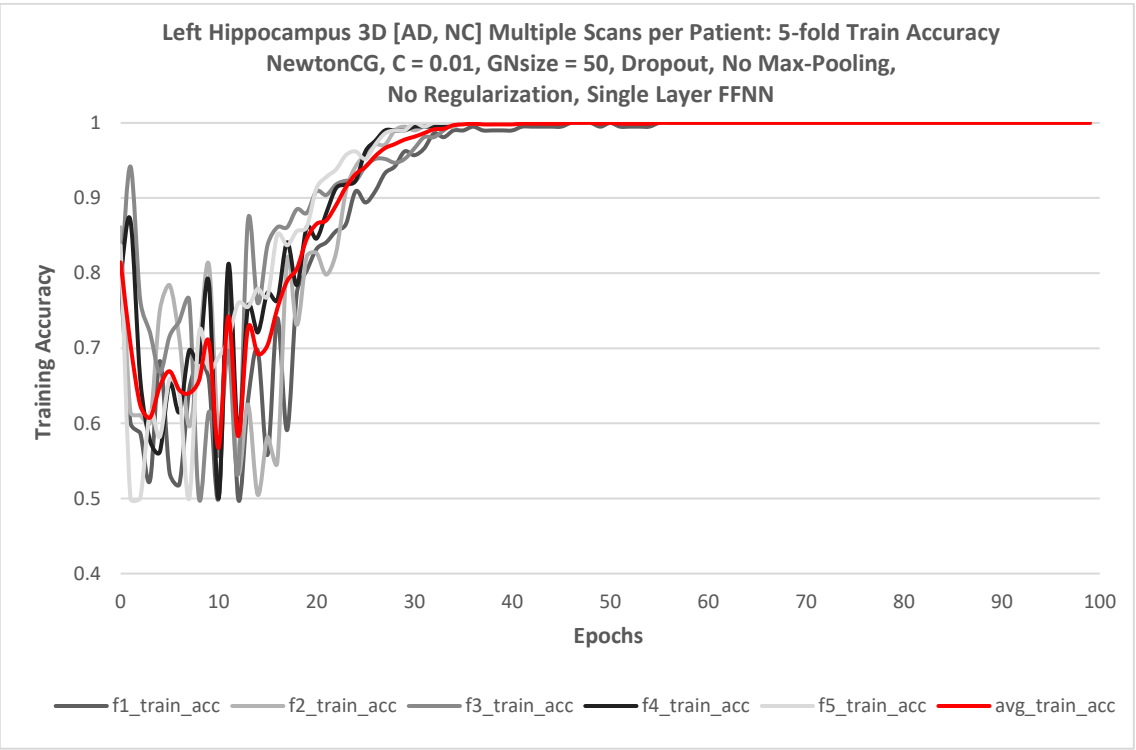


Figure L.48. Caption in the title.

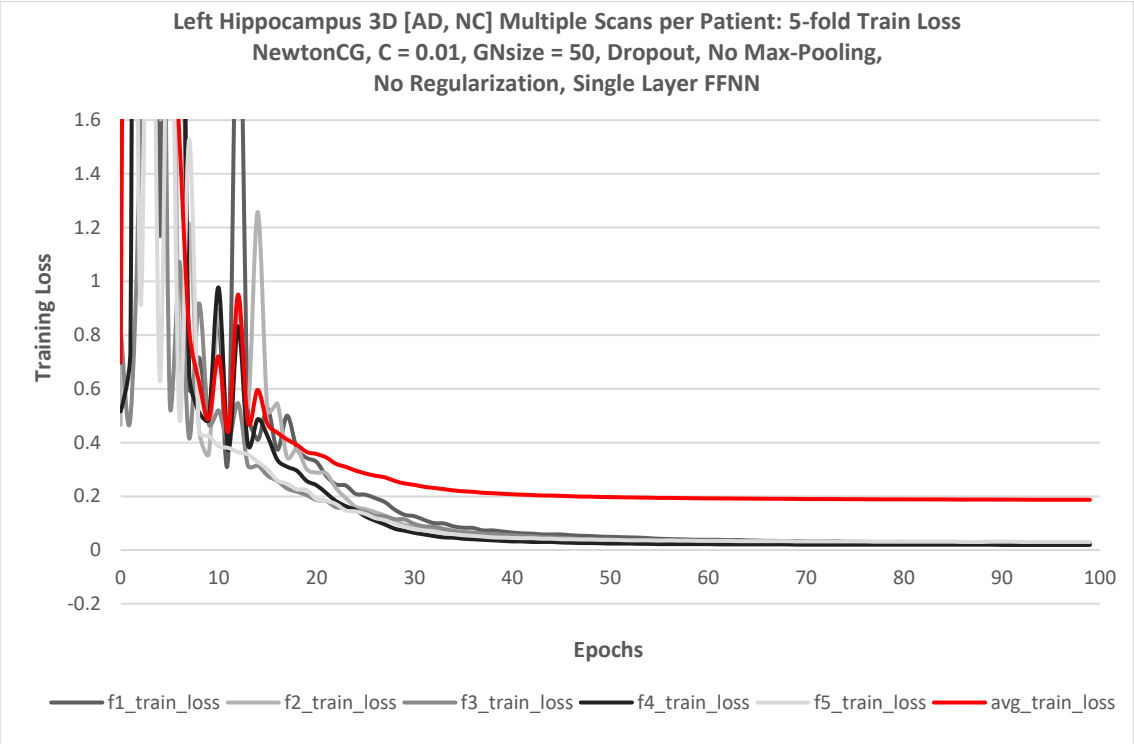


Figure L.49. Caption in the title.

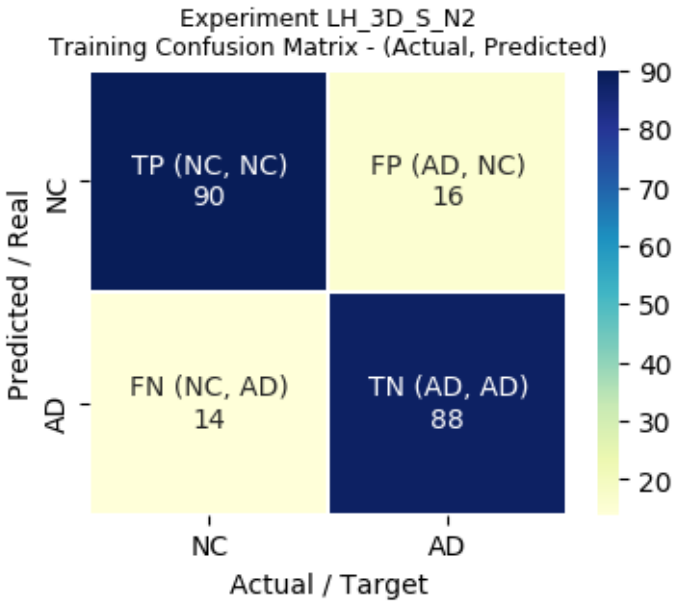


Figure L.50. Caption in the title.

L.2.2.3 Validation

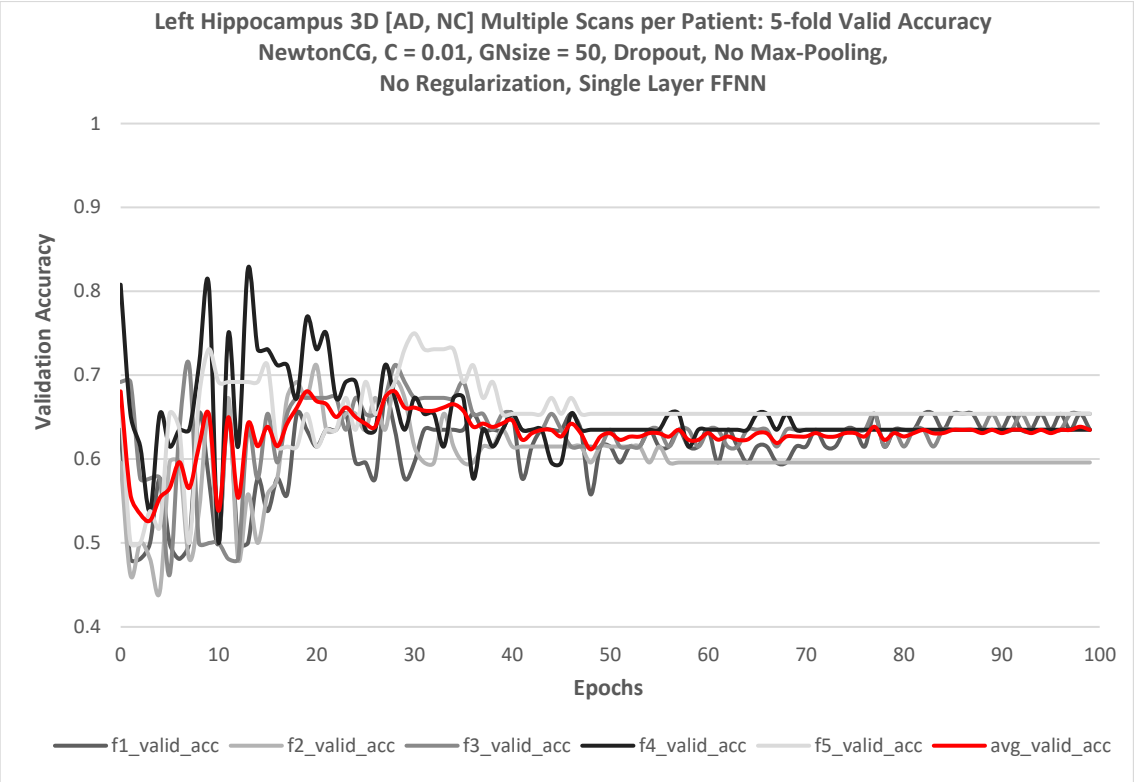


Figure L.51. Caption in the title.

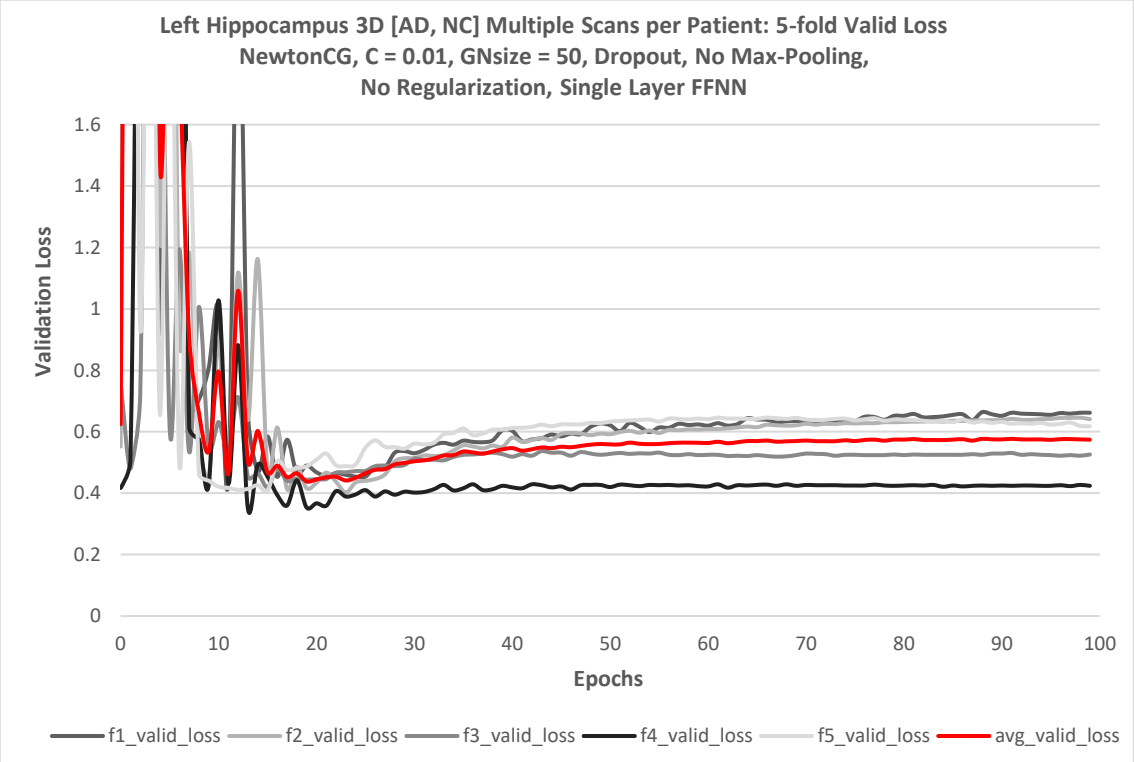


Figure L.52. Caption in the title.

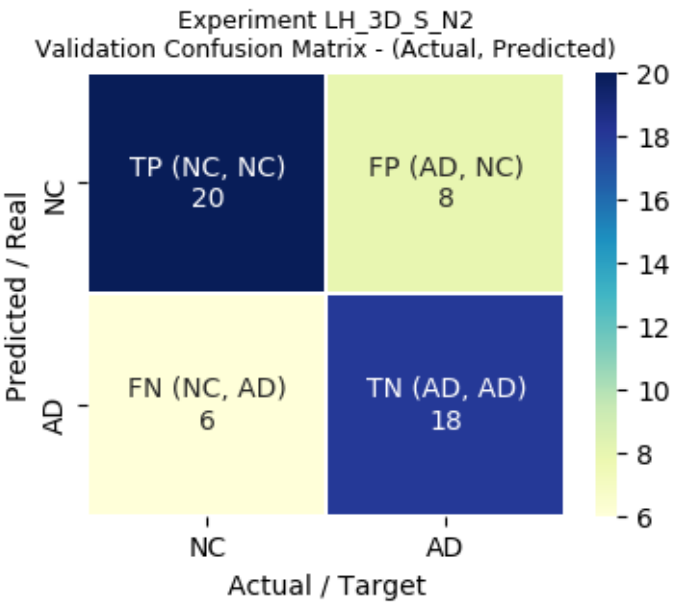


Figure L.53. Caption in the title.

L.2.2.4 Testing

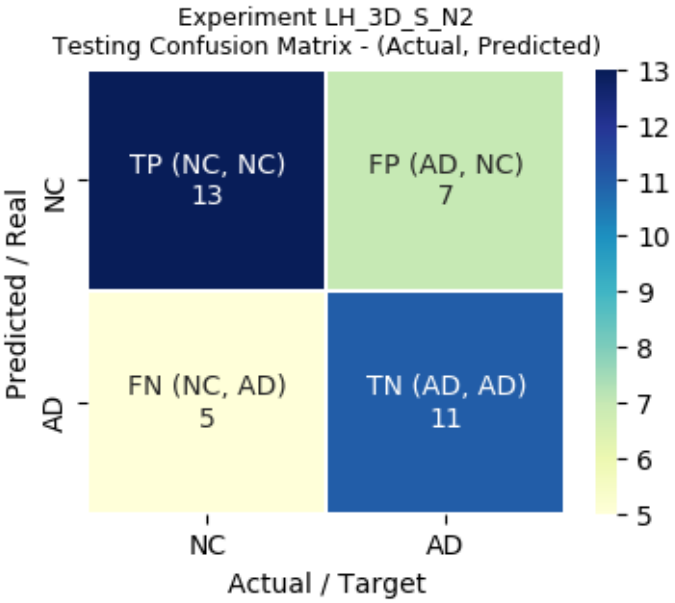


Figure L.54. Caption in the title.

Appendix M

3D Shrunk Brains [AD, NC] (44 × 48 × 44)

M.1 CNN 4 Layers with Adam

M.1.1 Experiment B_3D_S_A1: C = 0.01, lr = 0.0001, Dropout

M.1.1.1 General

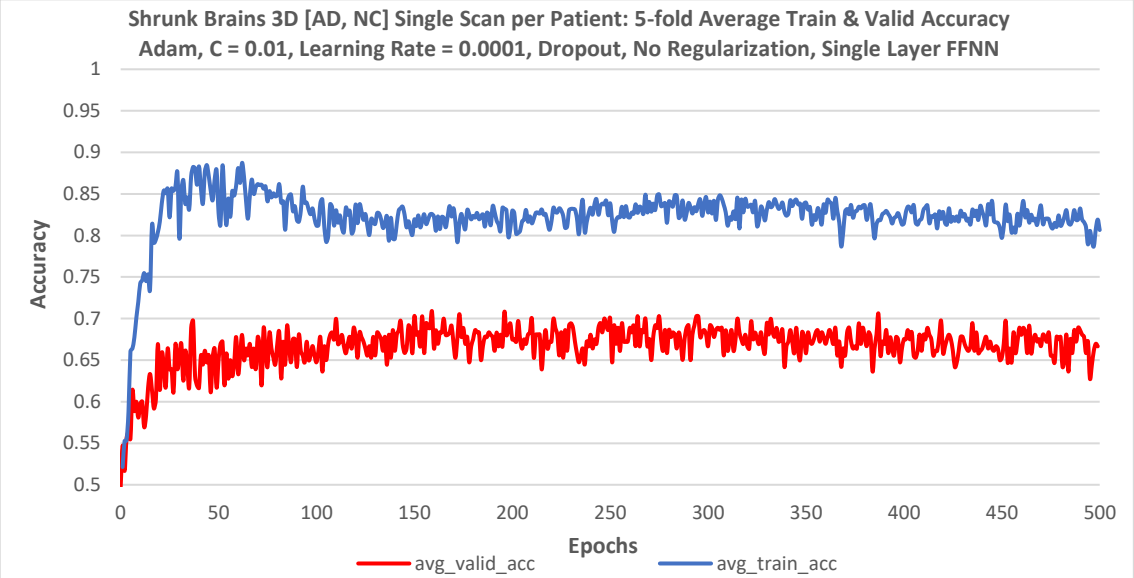


Figure M.1. Caption in the title.

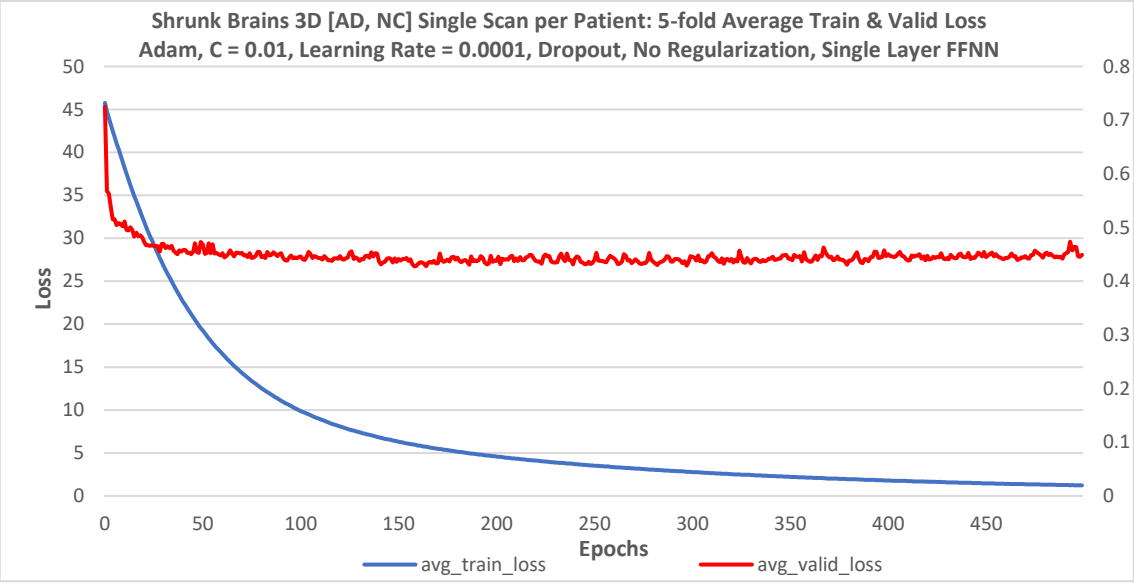


Figure M.2. Caption in the title.

M.1.1.2 Training

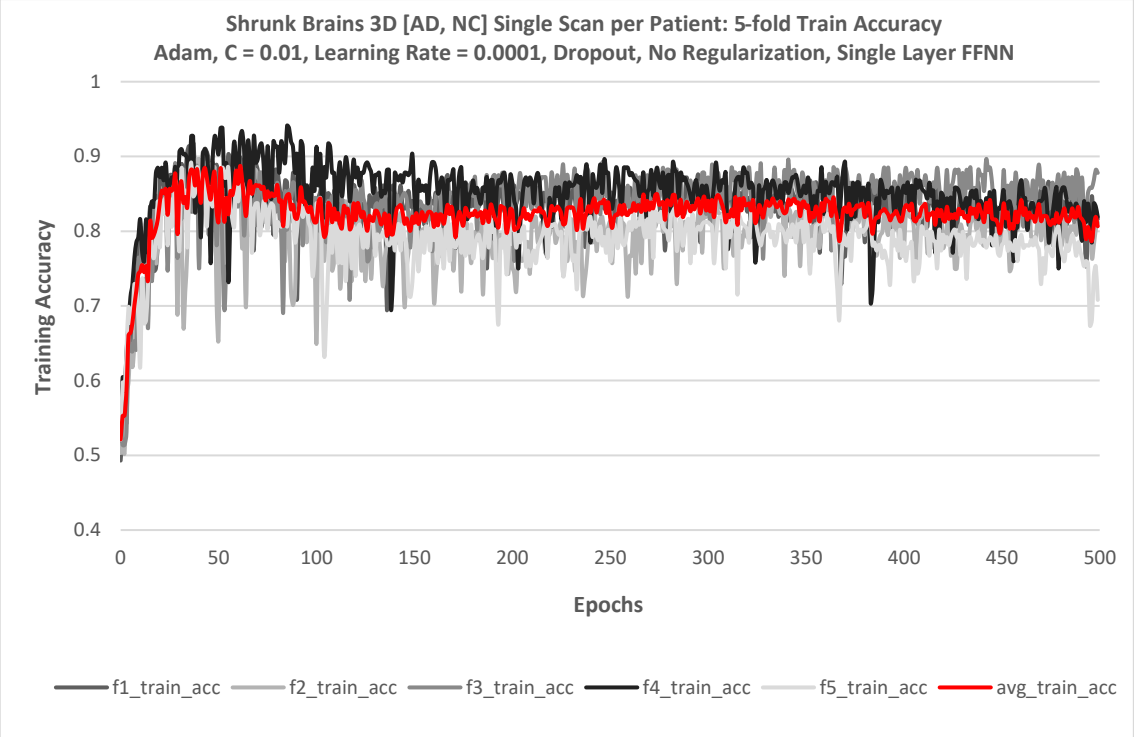


Figure M.3. Caption in the title.

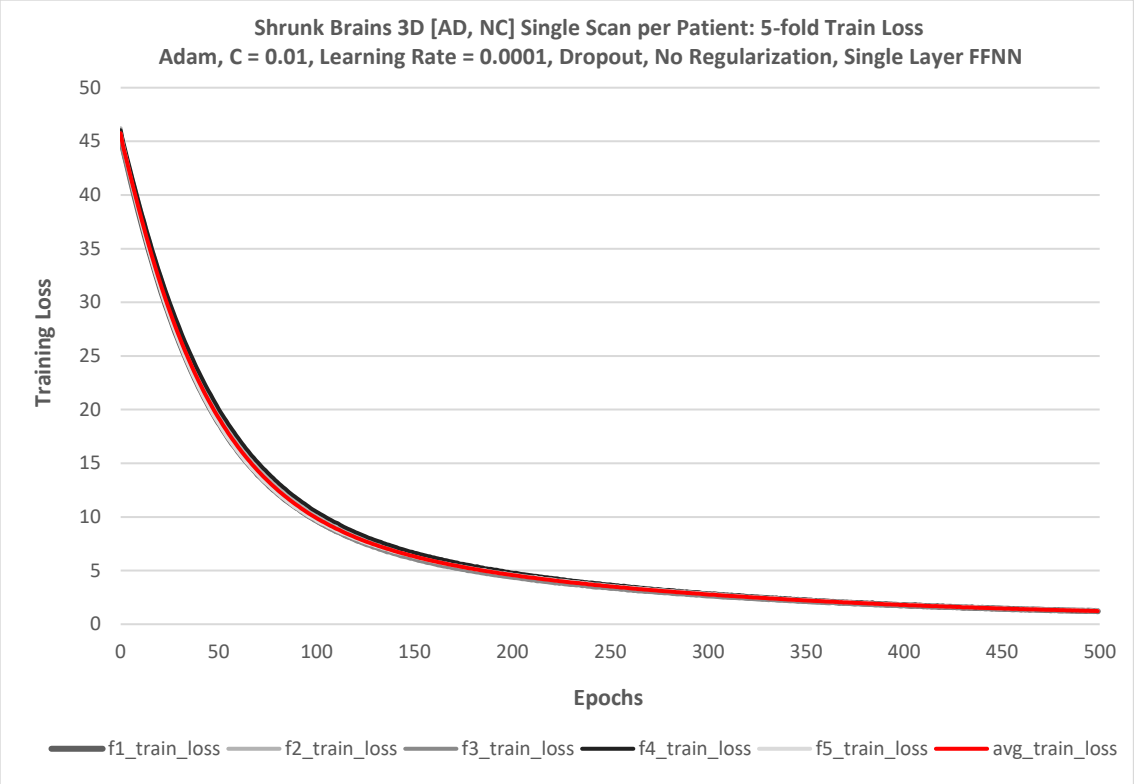


Figure M.4. Caption in the title.

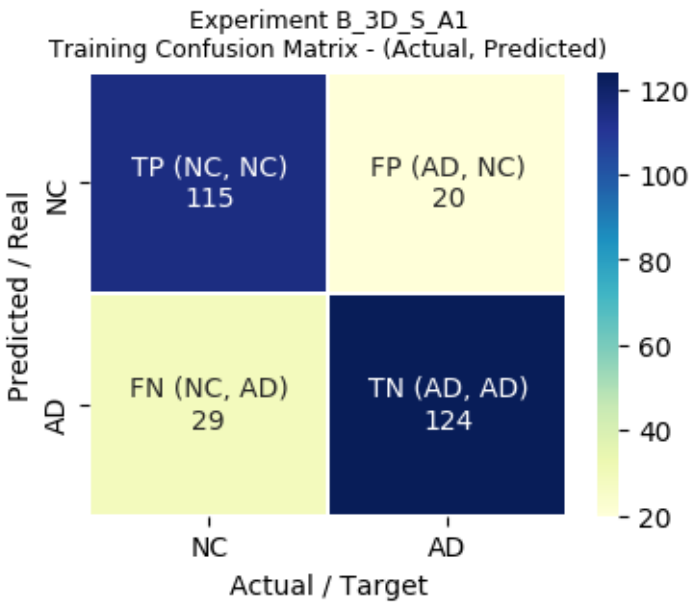


Figure M.5. Caption in the title.

M.1.1.3 Validation

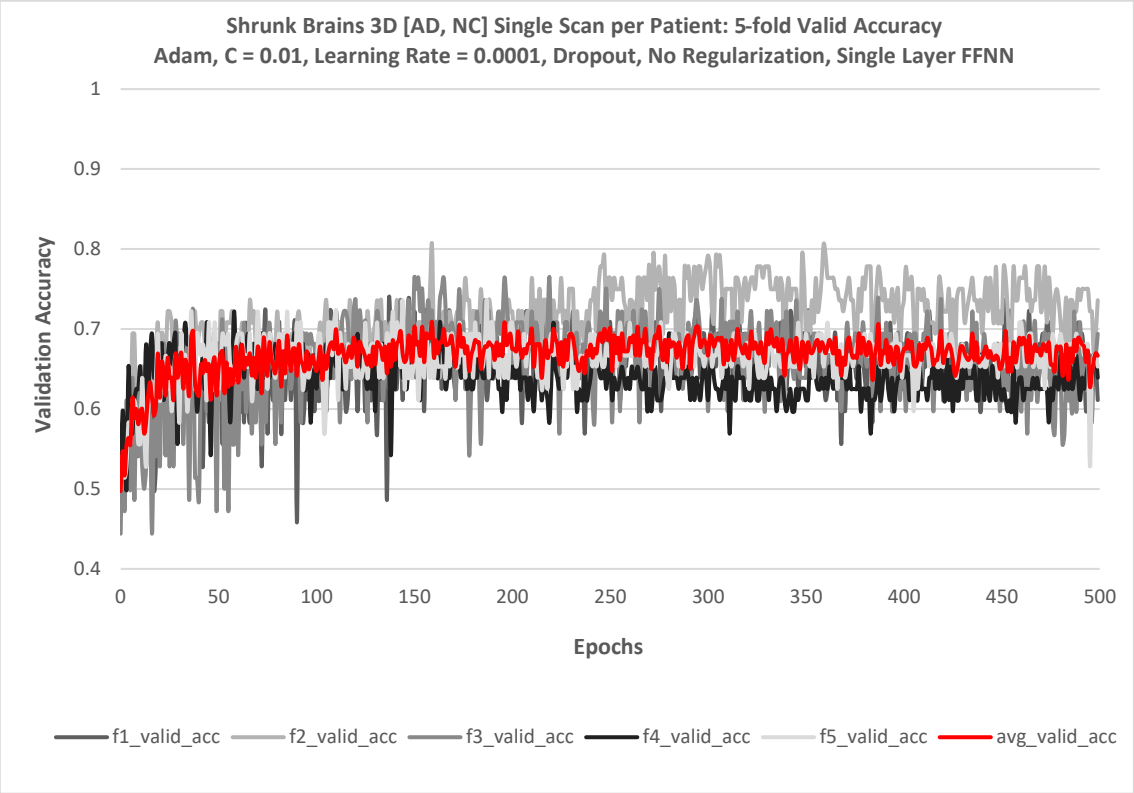


Figure M.6. Caption in the title.

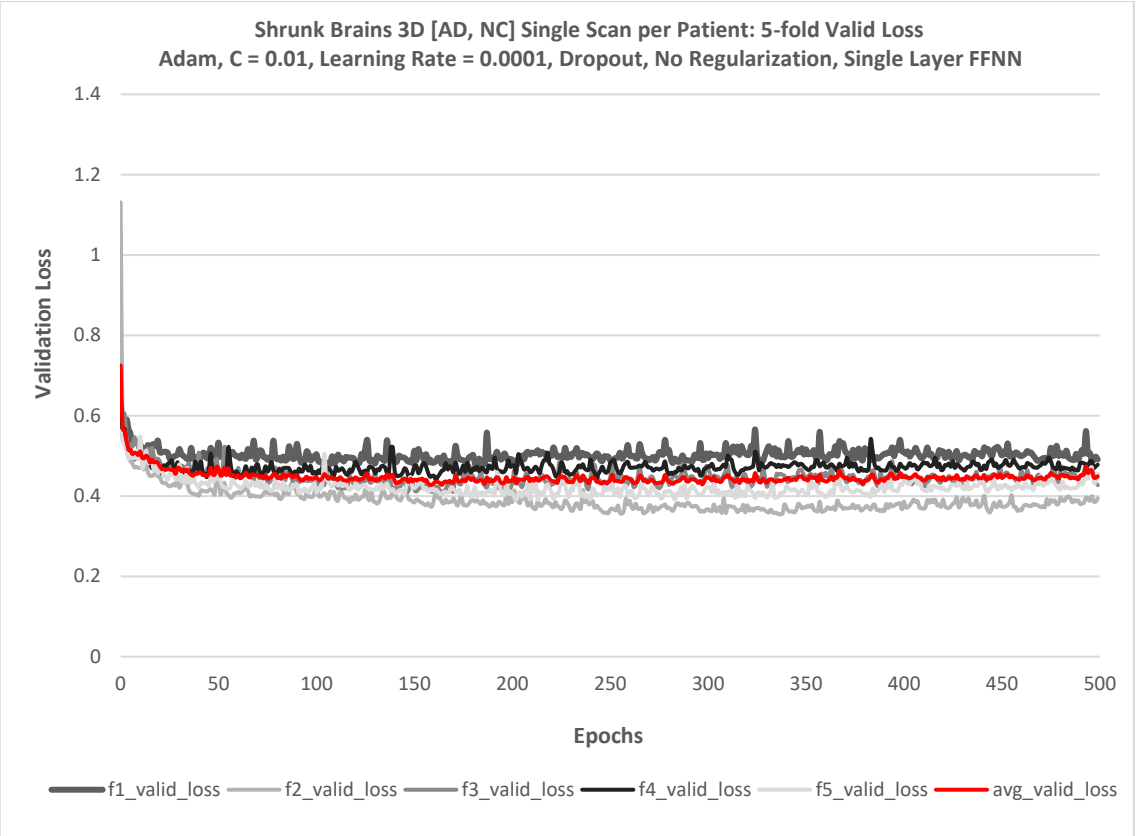


Figure M.7. Caption in the title.

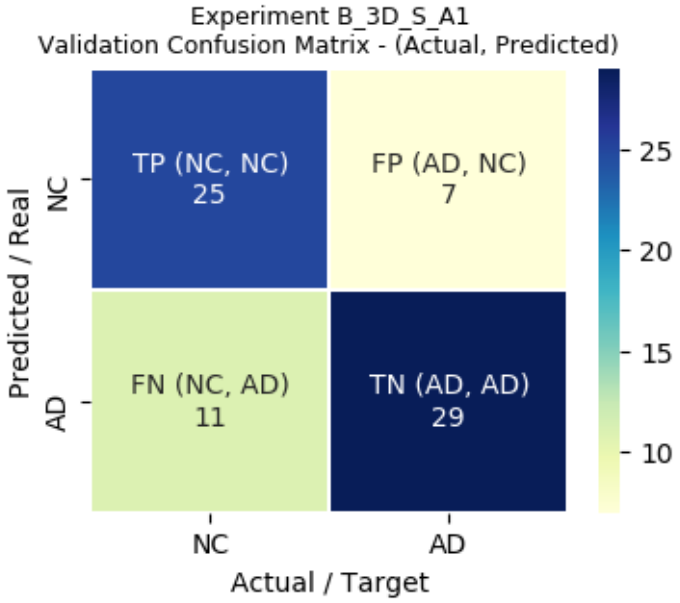


Figure M.8. Caption in the title.

M.1.1.4 Testing

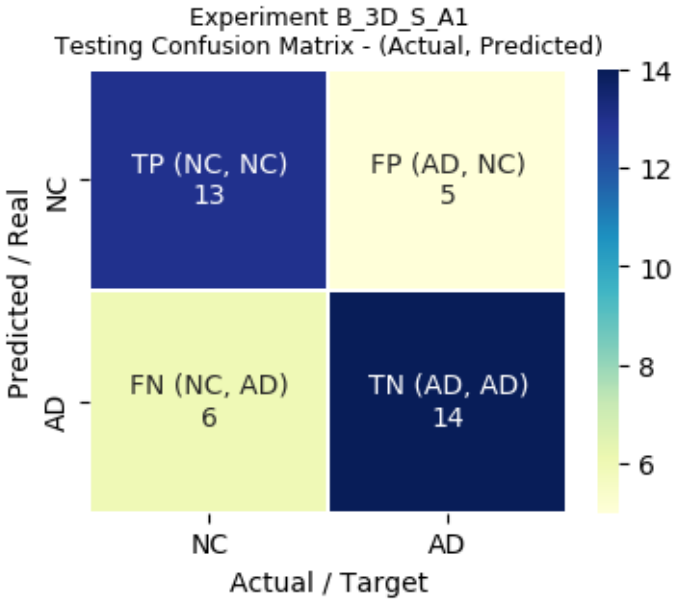


Figure M.9. Caption in the title.

M.2 CNN 4 Layers with NewtonCG

M.2.1 Experiment B_3D_S_N1: C = 0.01, GNsize = 200, Dropout

M.2.1.1 General

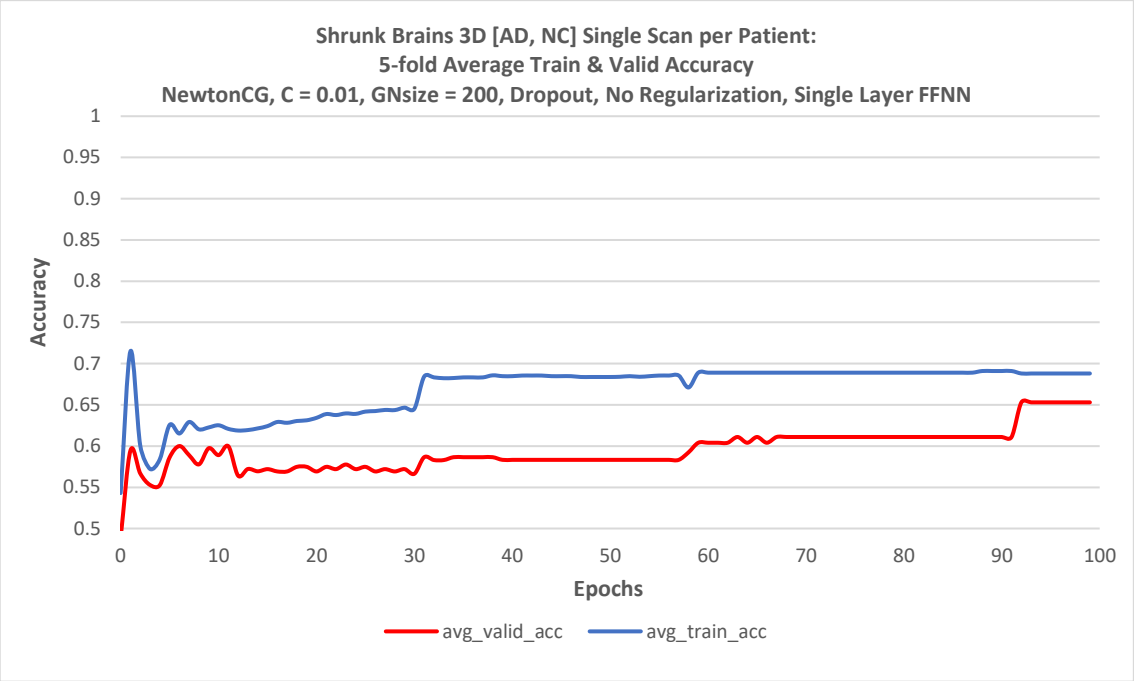


Figure M.10. Caption in the title.

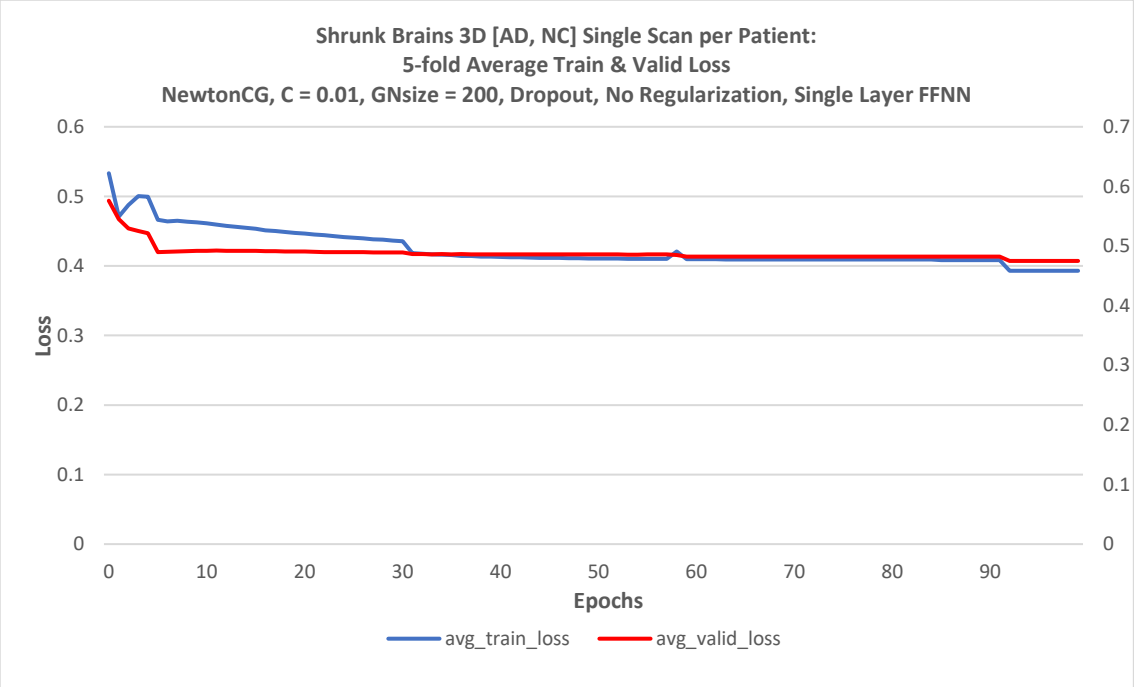


Figure M.11. Caption in the title.

M.2.1.2 Training

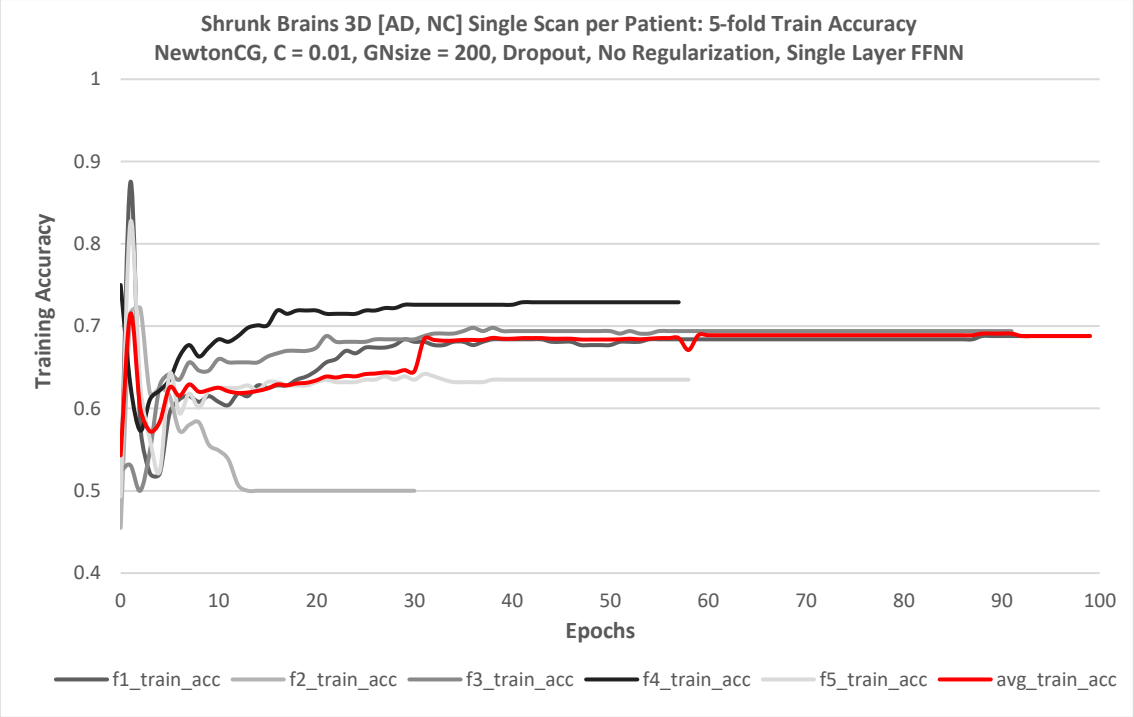


Figure M.12. Caption in the title.

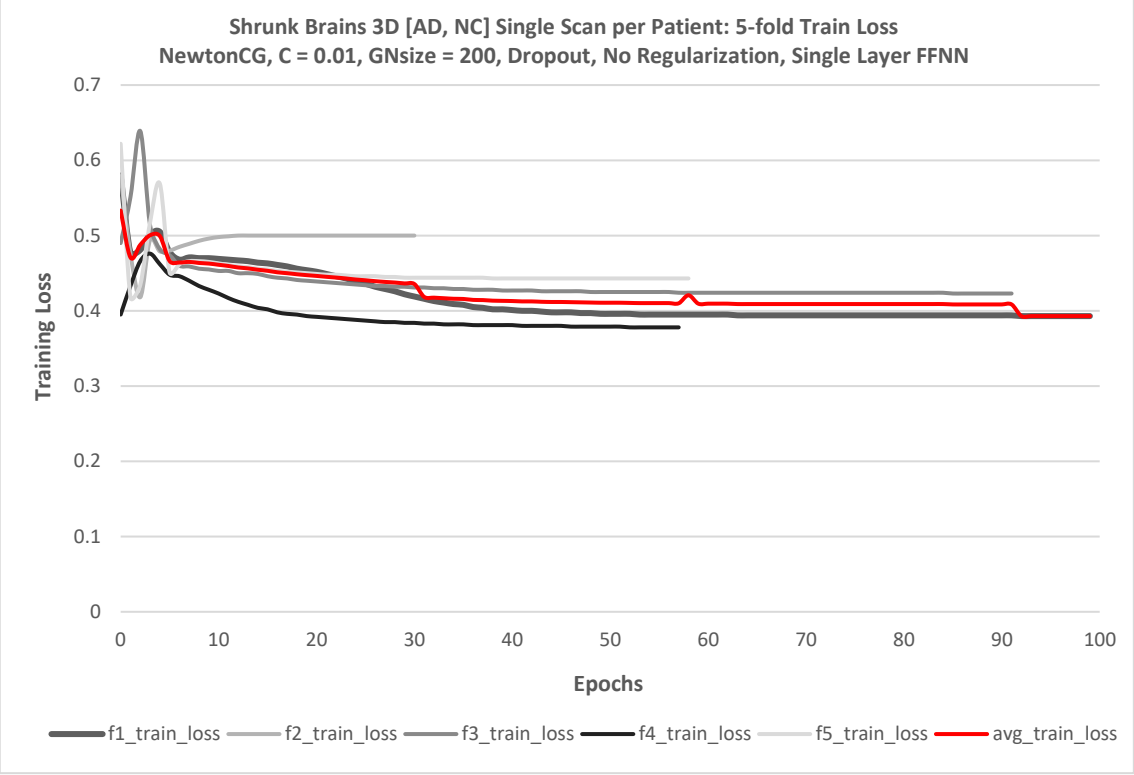


Figure M.13. Caption in the title.

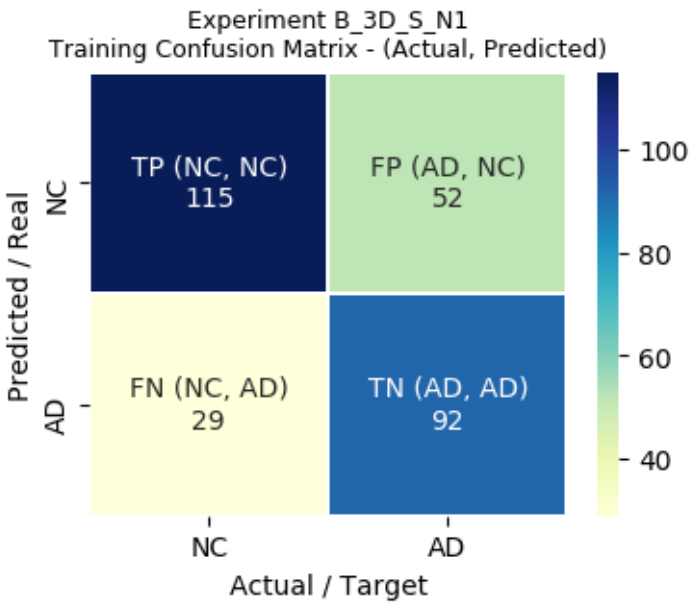


Figure M.14. Caption in the title.

M.2.1.3 Validation

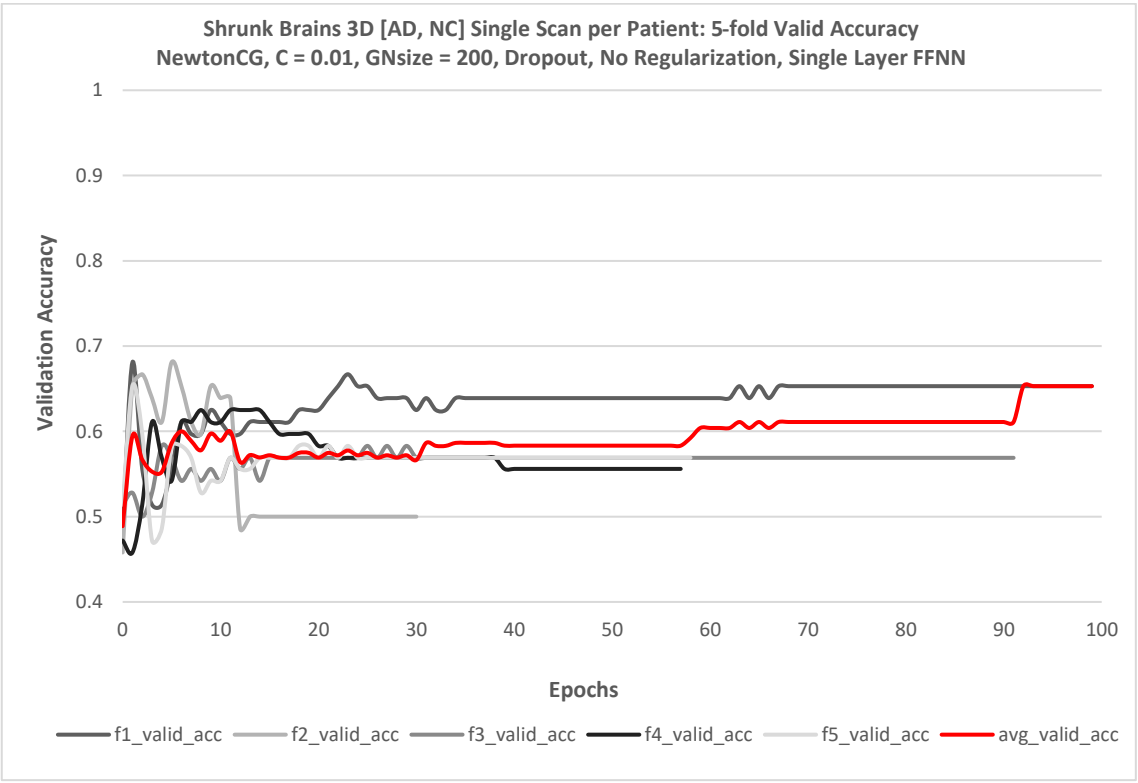


Figure M.15. Caption in the title.

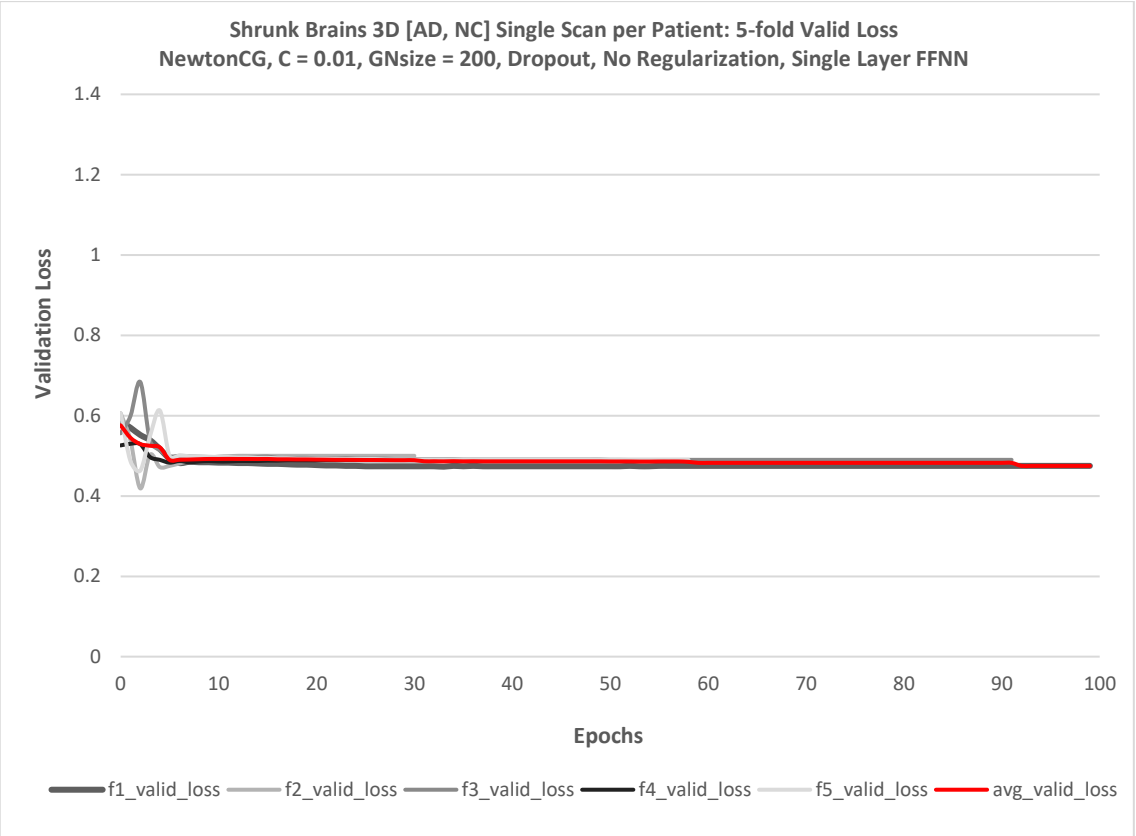


Figure M.16. Caption in the title.

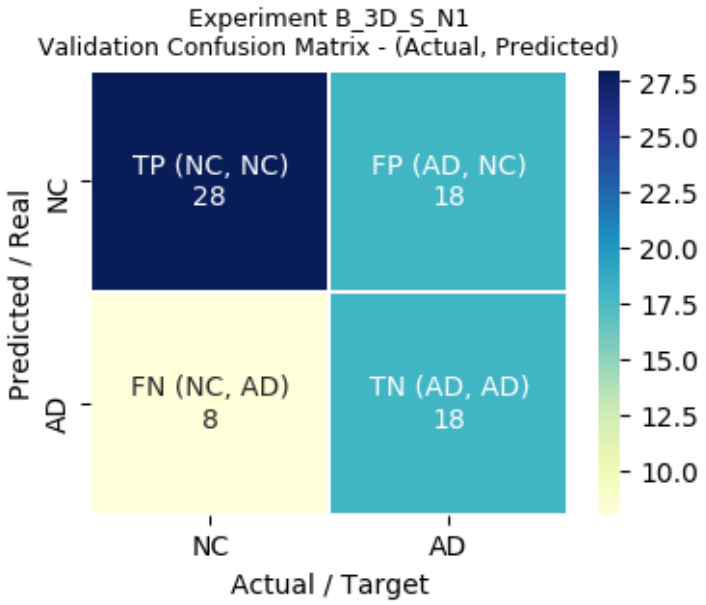


Figure M.17. Caption in the title.

M.2.1.4 Testing

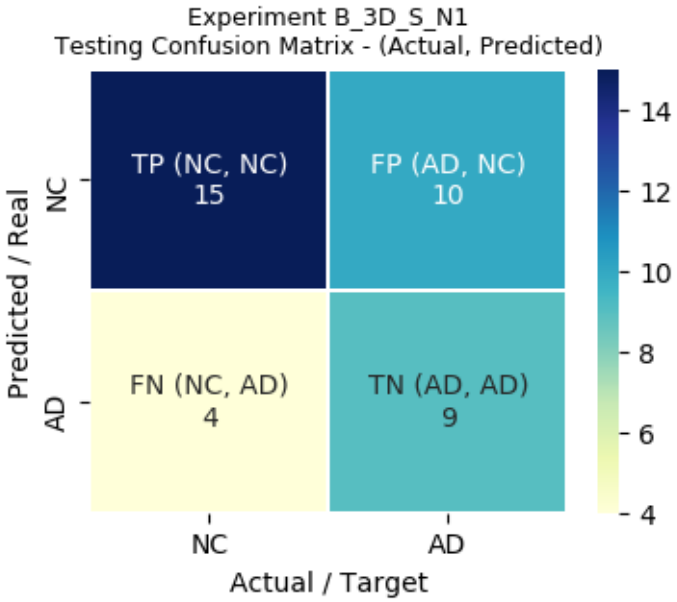


Figure M.18. Caption in the title.

Appendix N

3D Cropped Brains [AD, NC] (70 × 60 × 60)

N.1 CNN 4 Layers with Adam

N.1.1 Experiment CB_3D_S_A1: C = 0.01, lr = 0.0001, Dropout

N.1.1.1 General

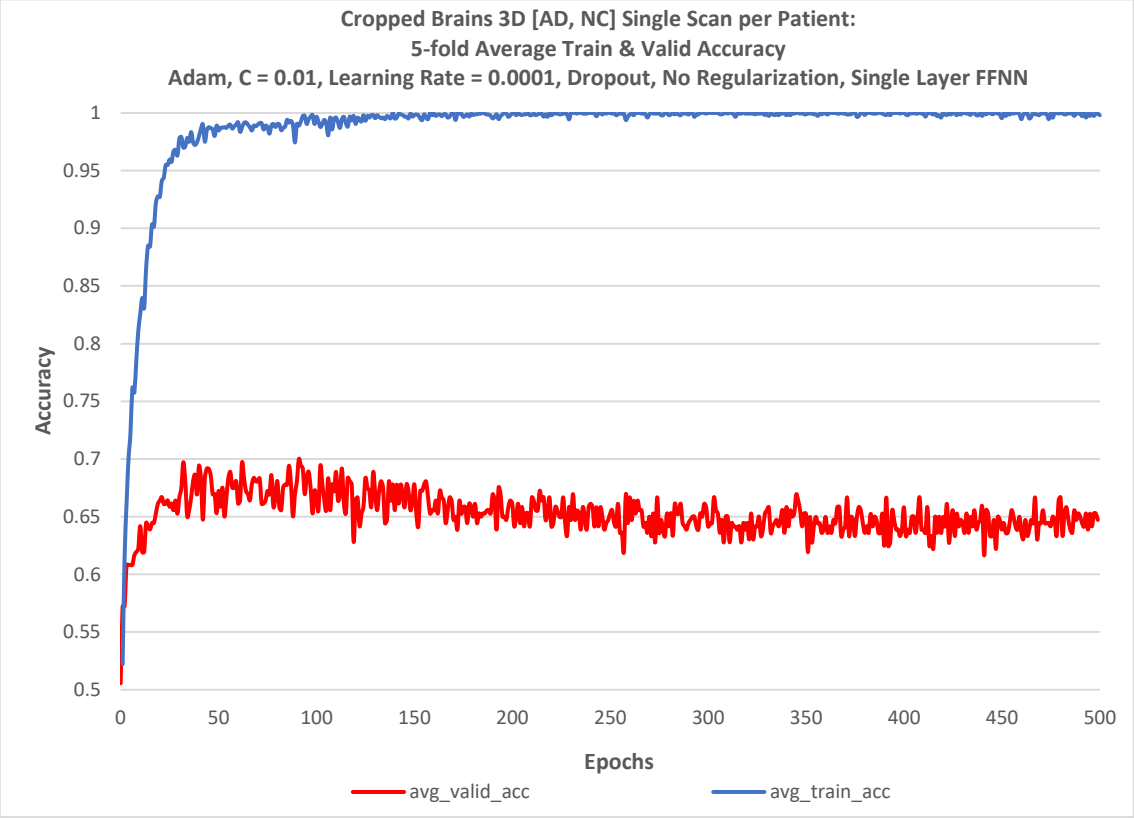


Figure N.1. Caption in the title.

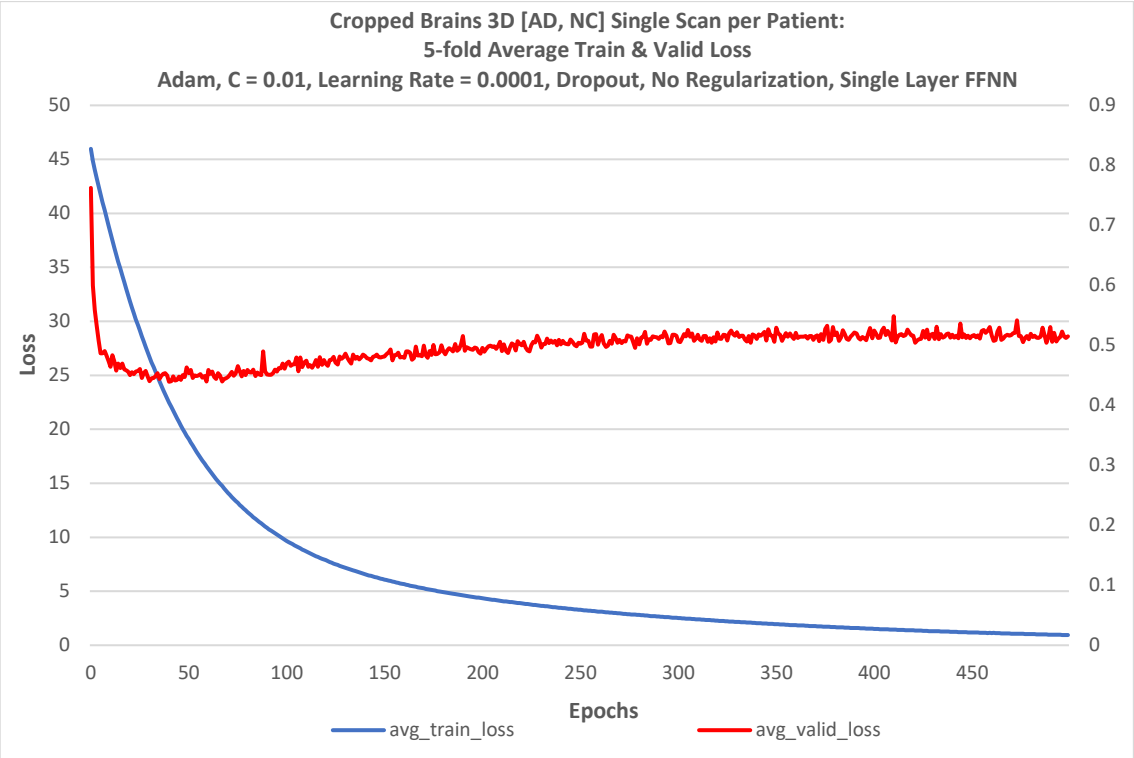


Figure N.2. Caption in the title.

N.1.1.2 Training

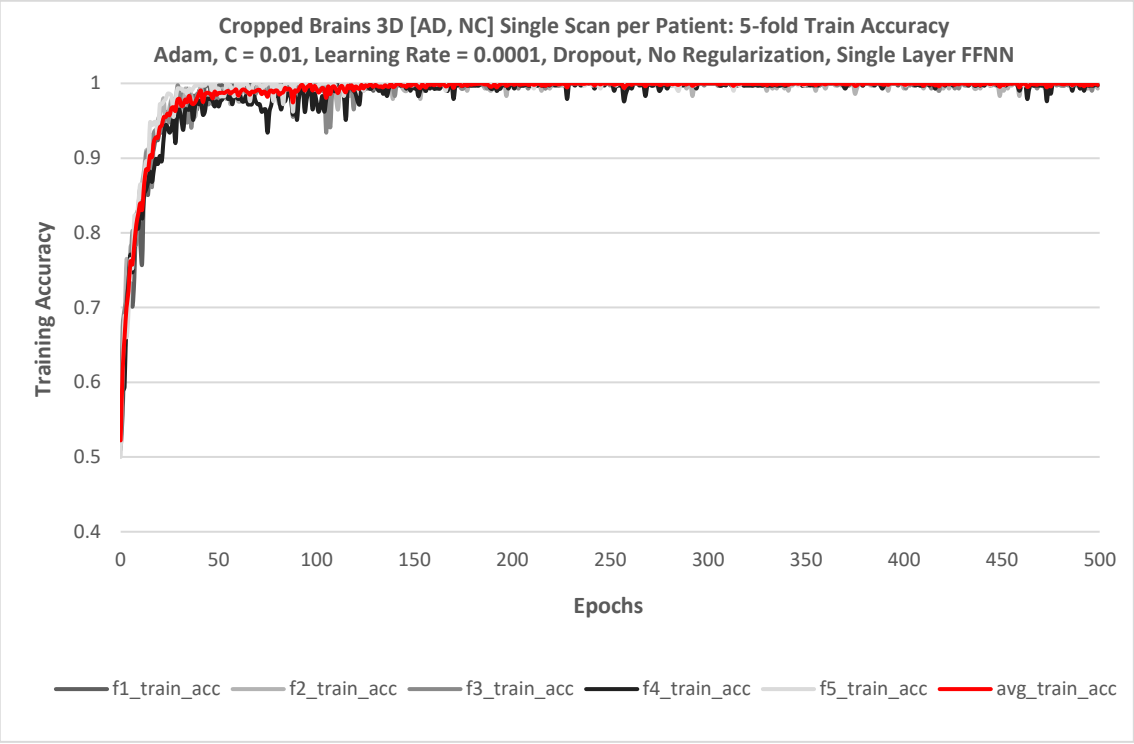


Figure N.3. Caption in the title.

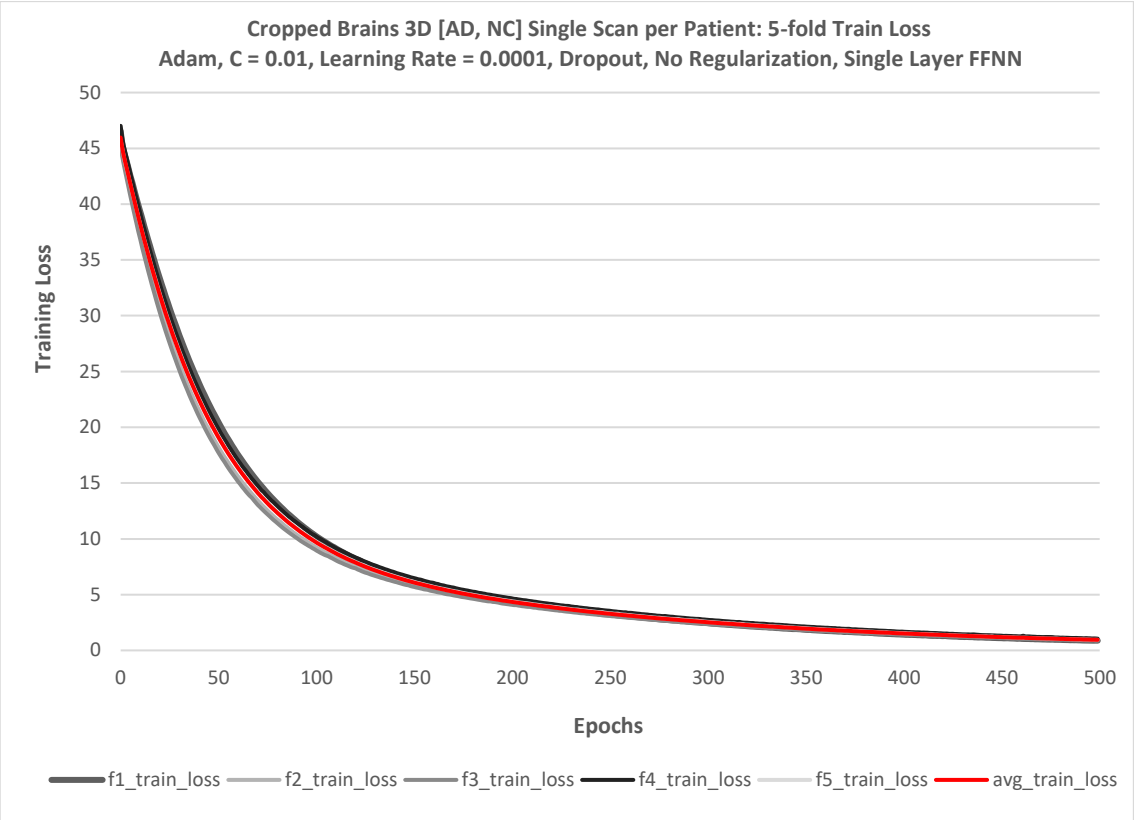
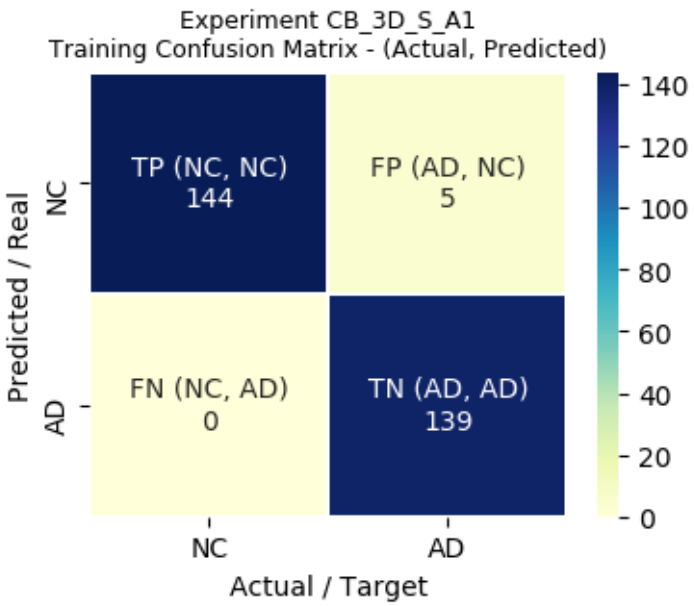


Figure N.4. Caption in the title.



N.1.1.3 Validation

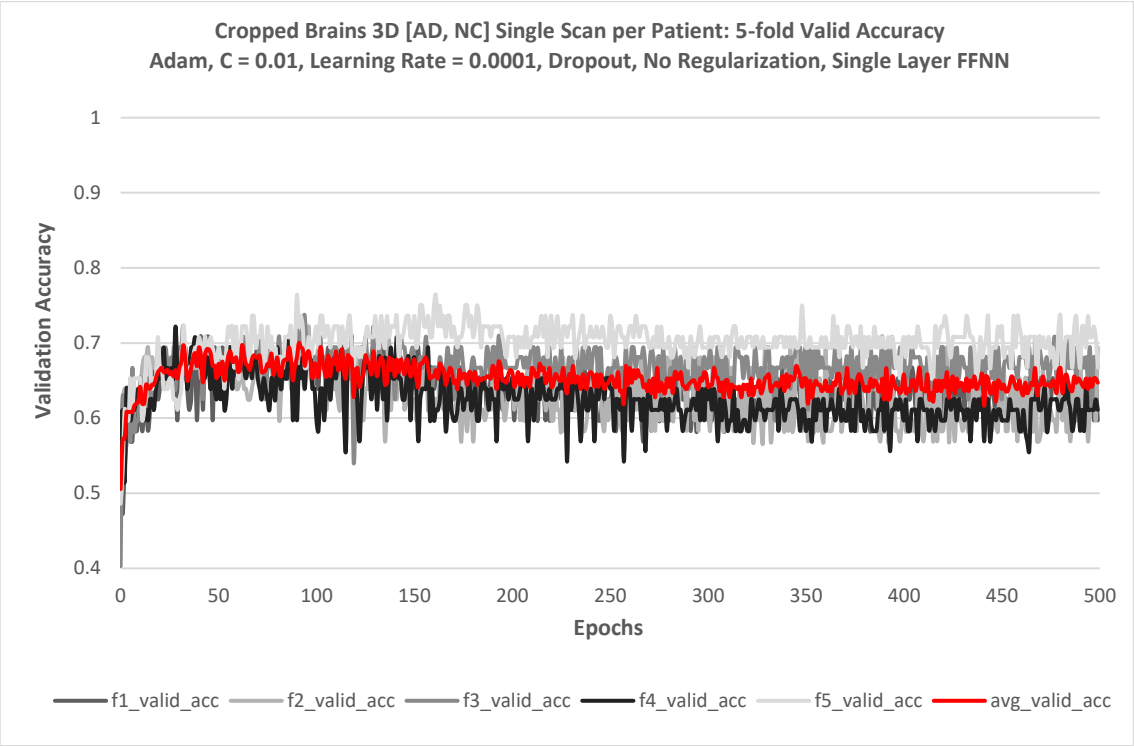


Figure N.5. Caption in the title.

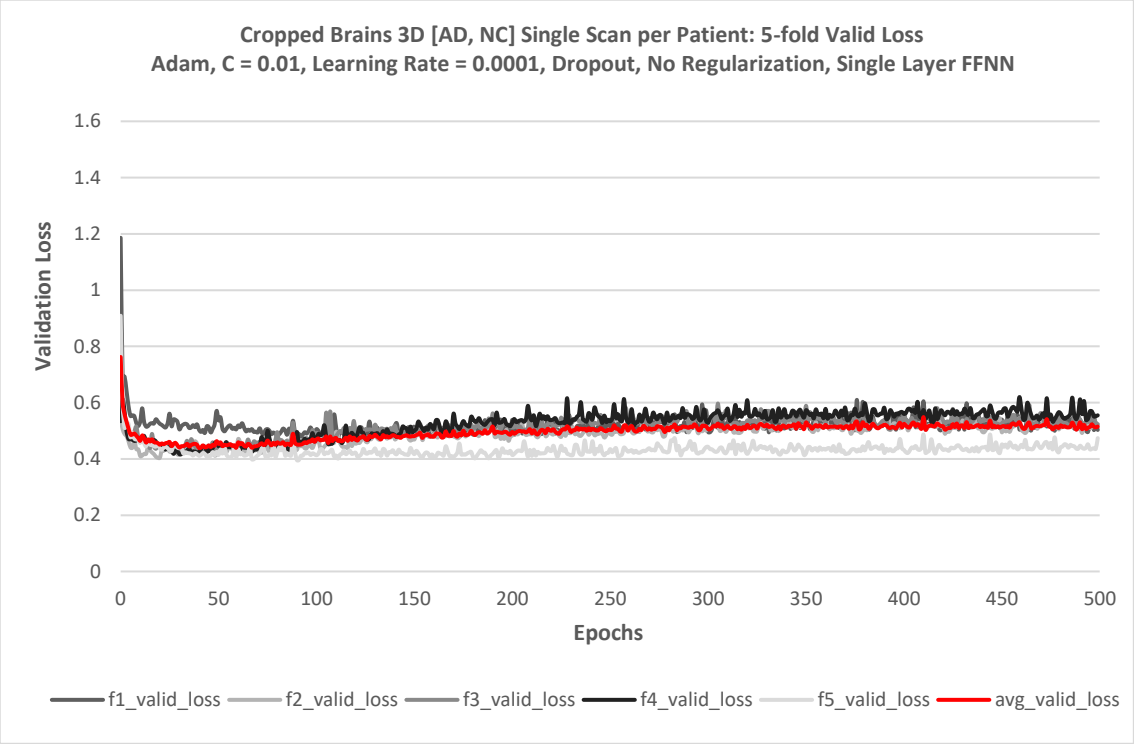


Figure N.6. Caption in the title.

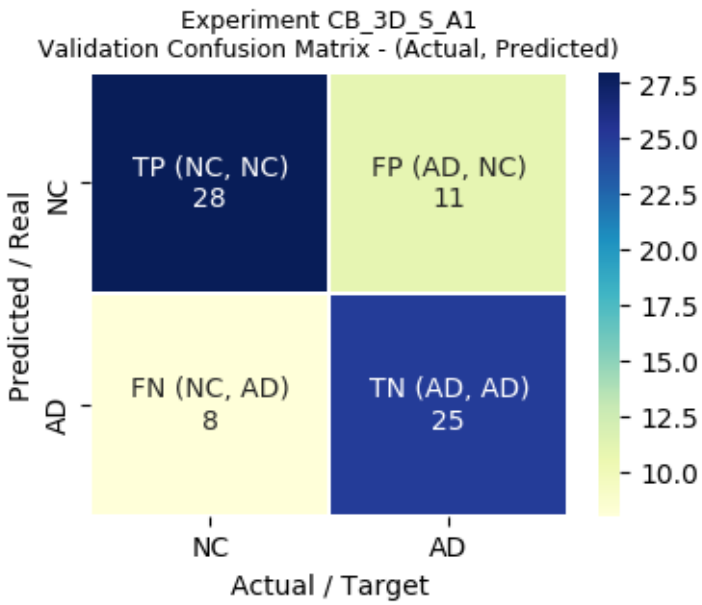


Figure N.7. Caption in the title.

N.1.1.4 Testing

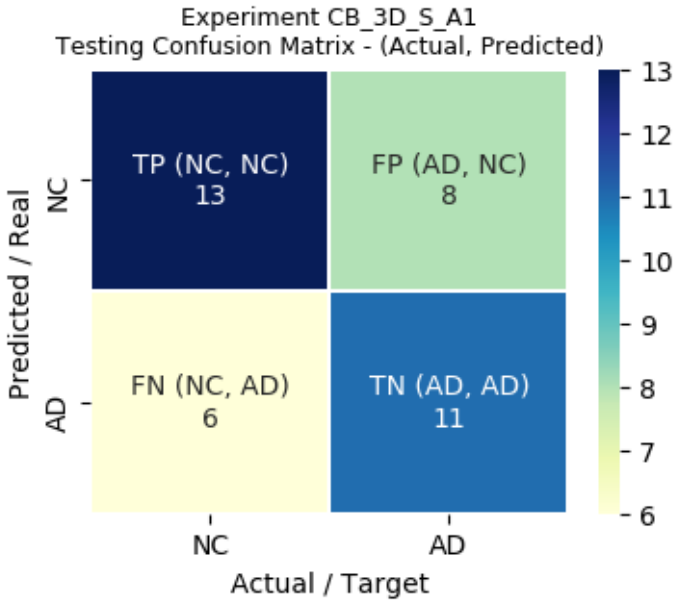


Figure N.8. Caption in the title.

N.2 CNN 4 Layers with NewtonCG

N.2.1 Experiment CB_3D_S_N1: C = 0.01, GNsize = 50, Dropout

N.2.1.1 General

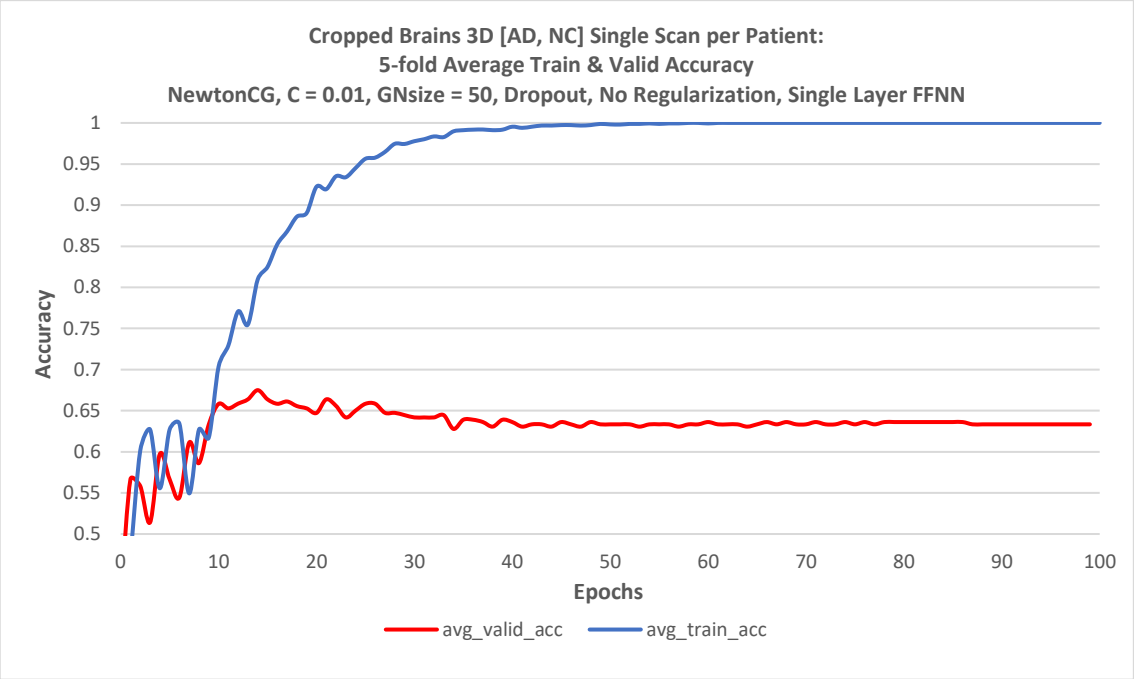


Figure N.9. Caption in the title.

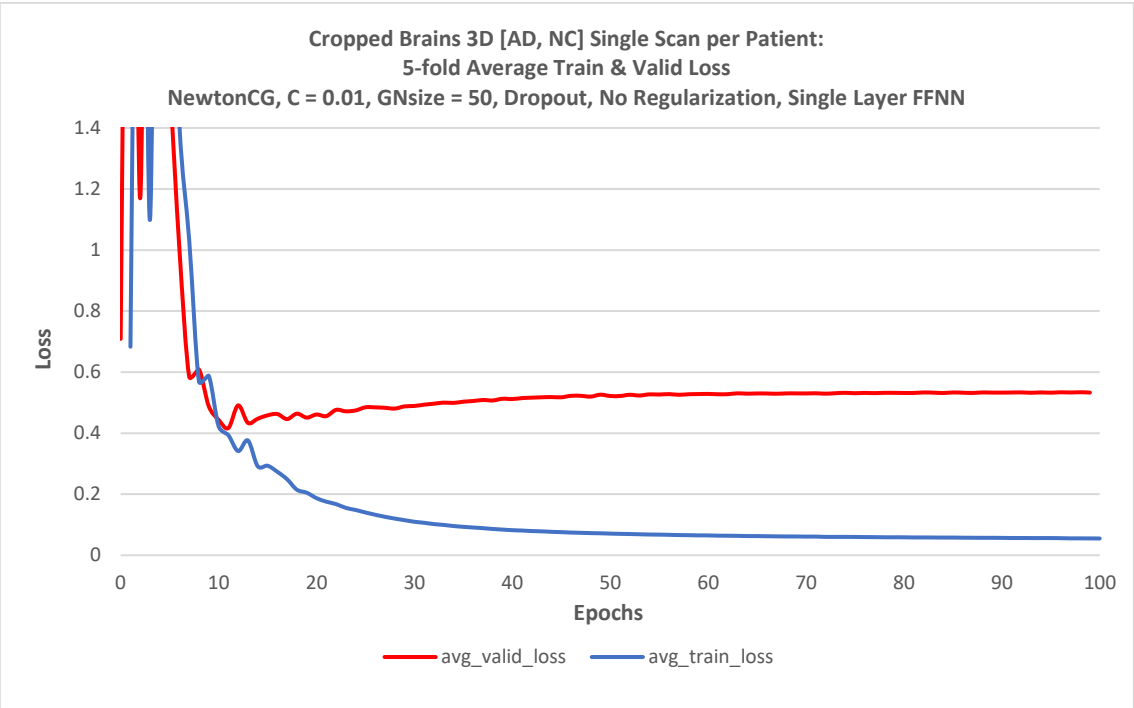


Figure N.10. Caption in the title.

N.2.1.2 Training

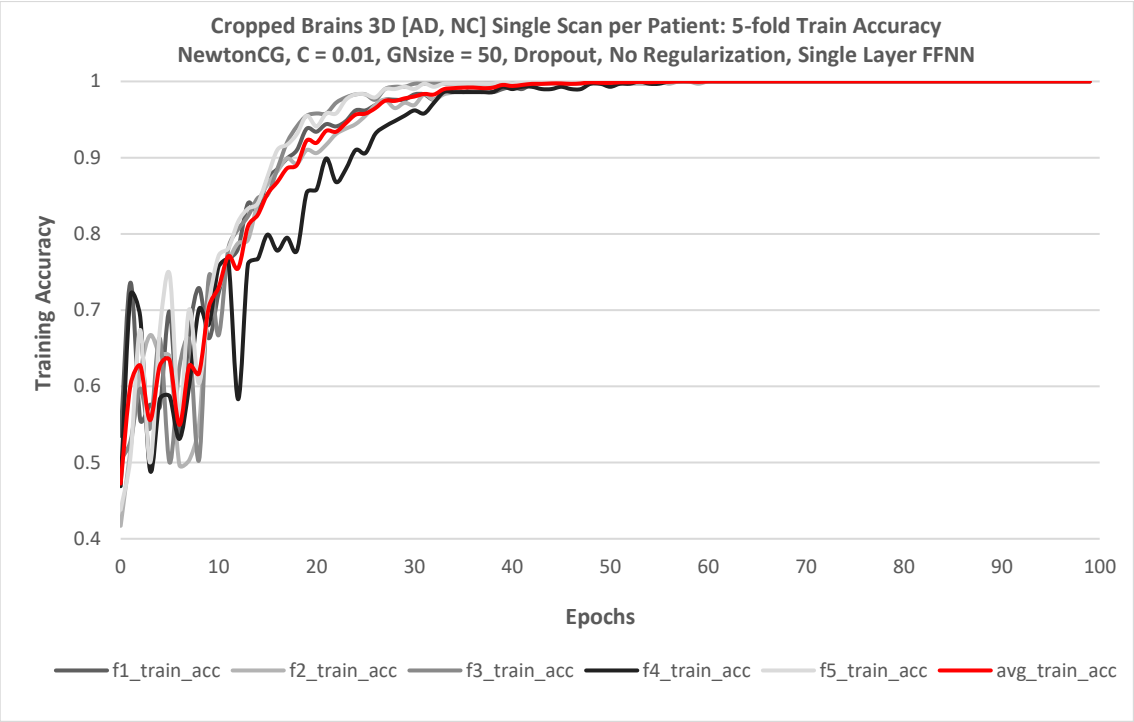


Figure N.11. Caption in the title.

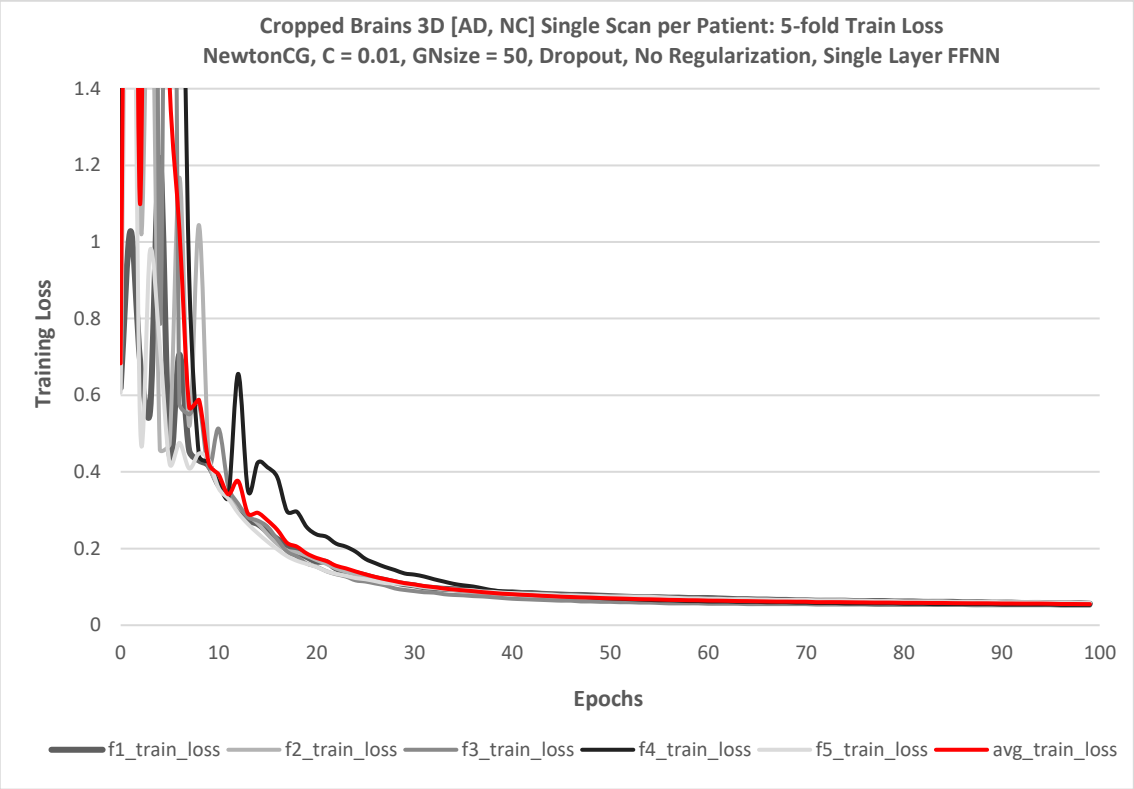


Figure N.12. Caption in the title.

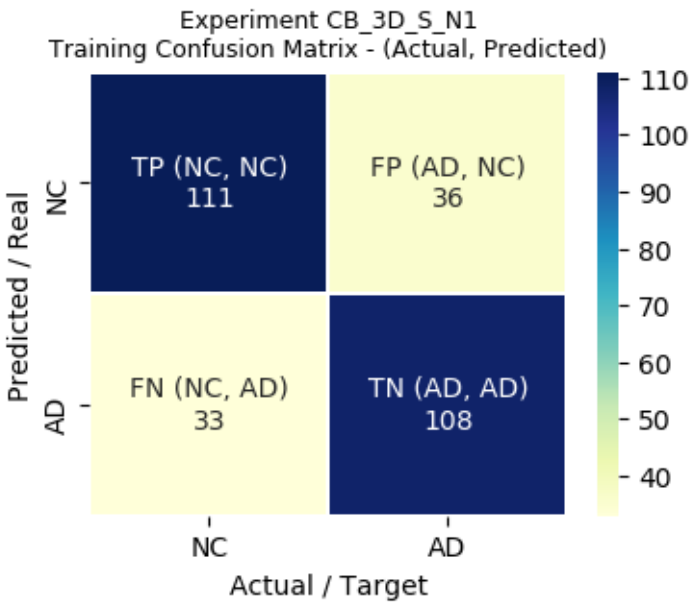


Figure N.13. Caption in the title.

N.2.1.3 Validation

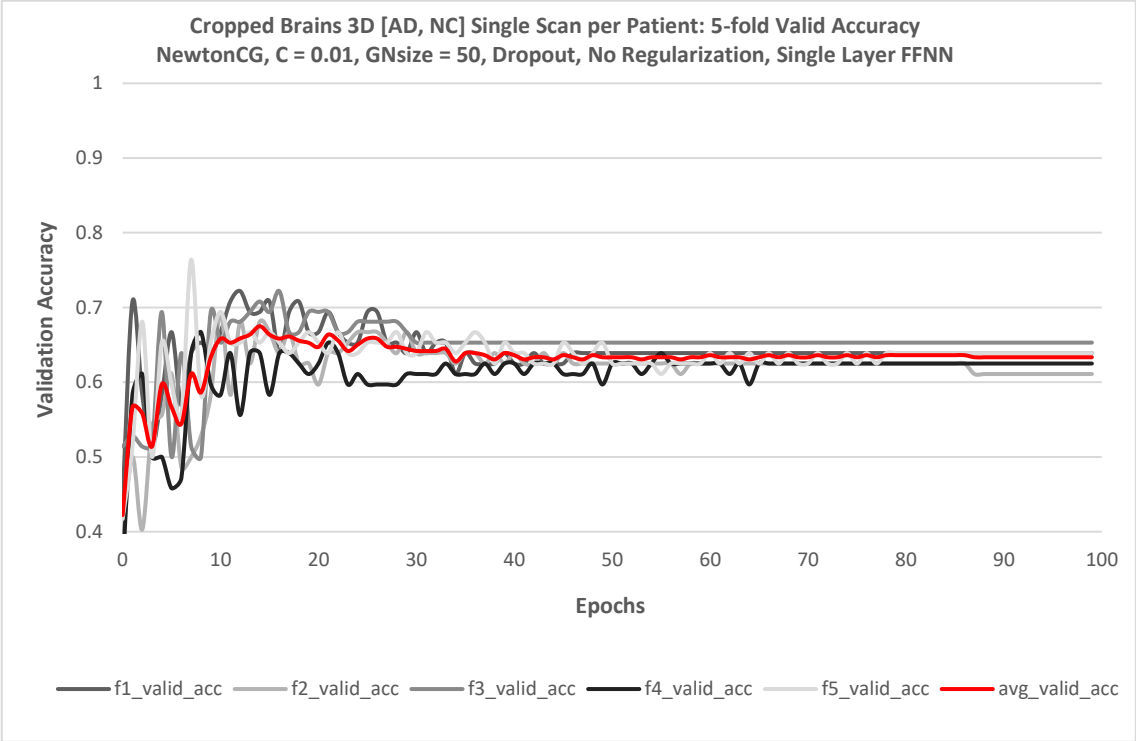


Figure N.14. Caption in the title.

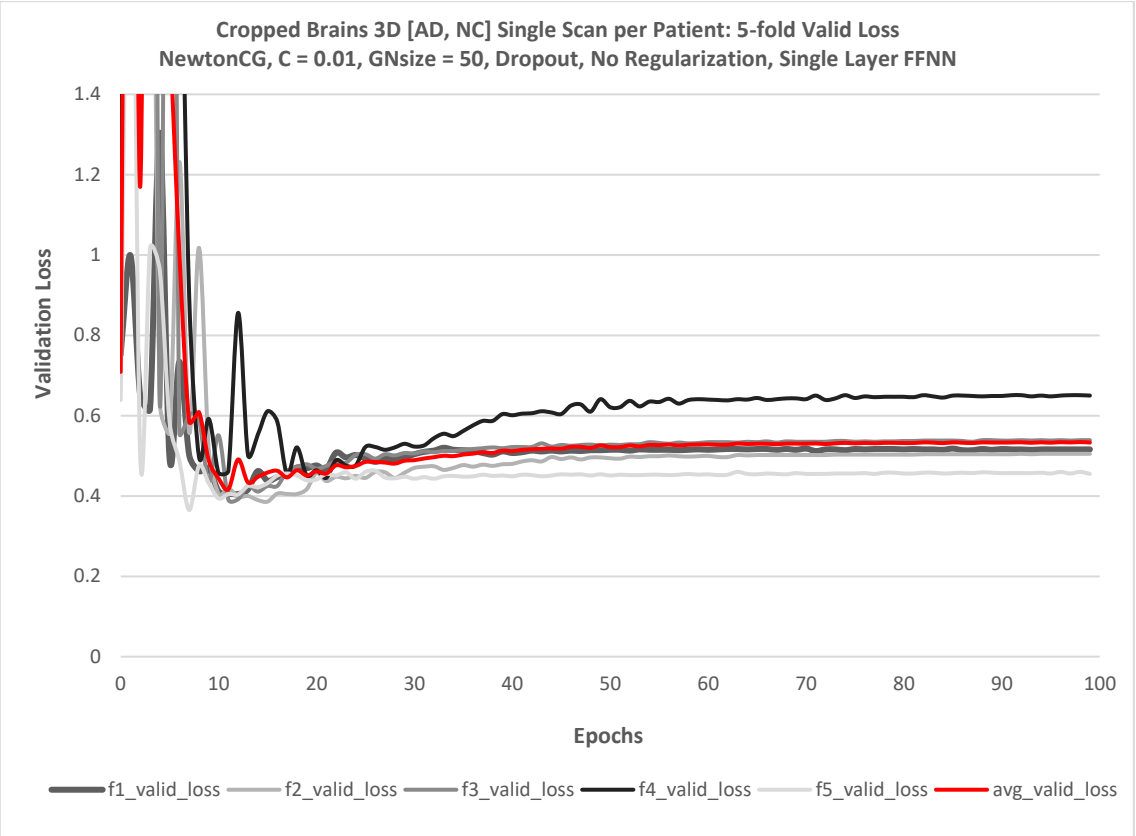


Figure N.15. Caption in the title.

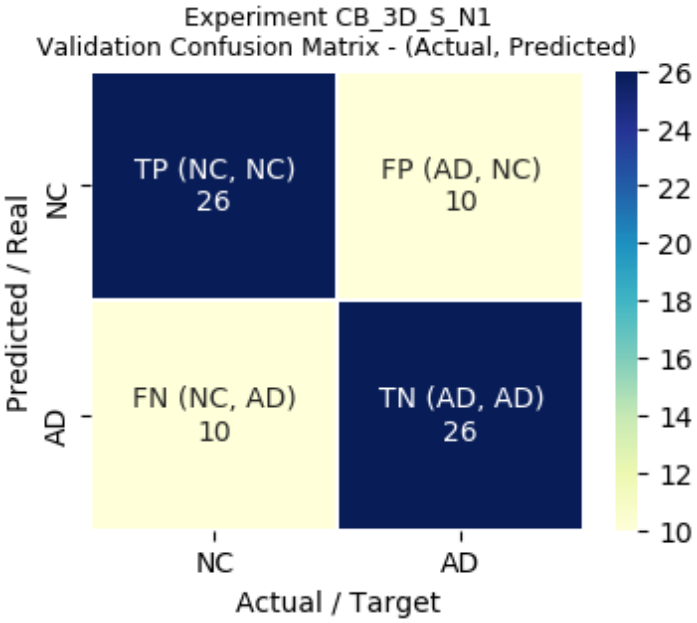


Figure N.16. Caption in the title.

N.2.1.4 Testing

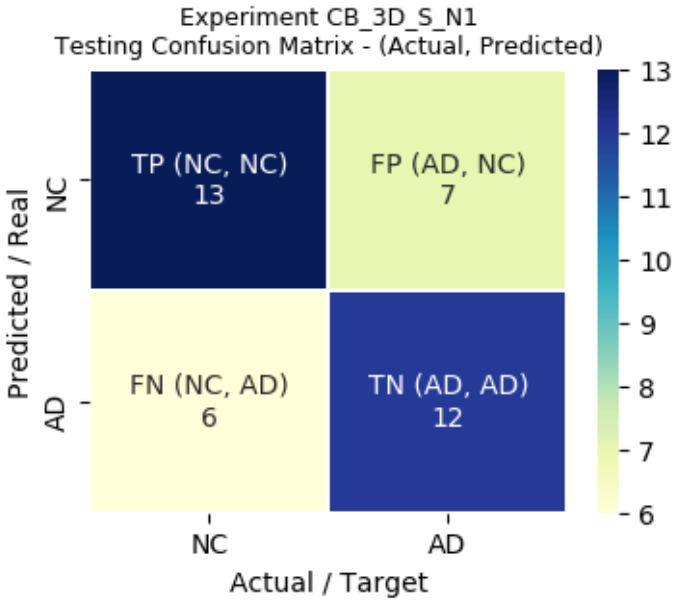


Figure N.17. Caption in the title.

N.3 CNN 5 Layers with NewtonCG

N.3.1 Experiment CB_3D_S_N2: C = 0.01, GNsize = 50, Dropout

N.3.1.1 General

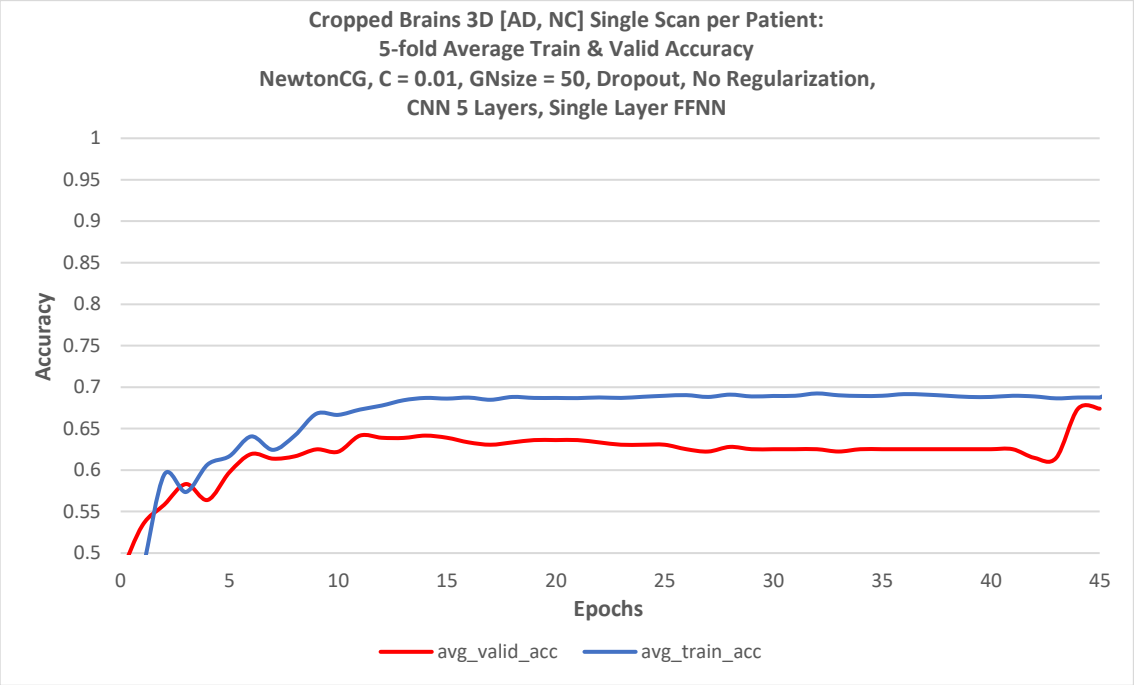


Figure N.18. Caption in the title.

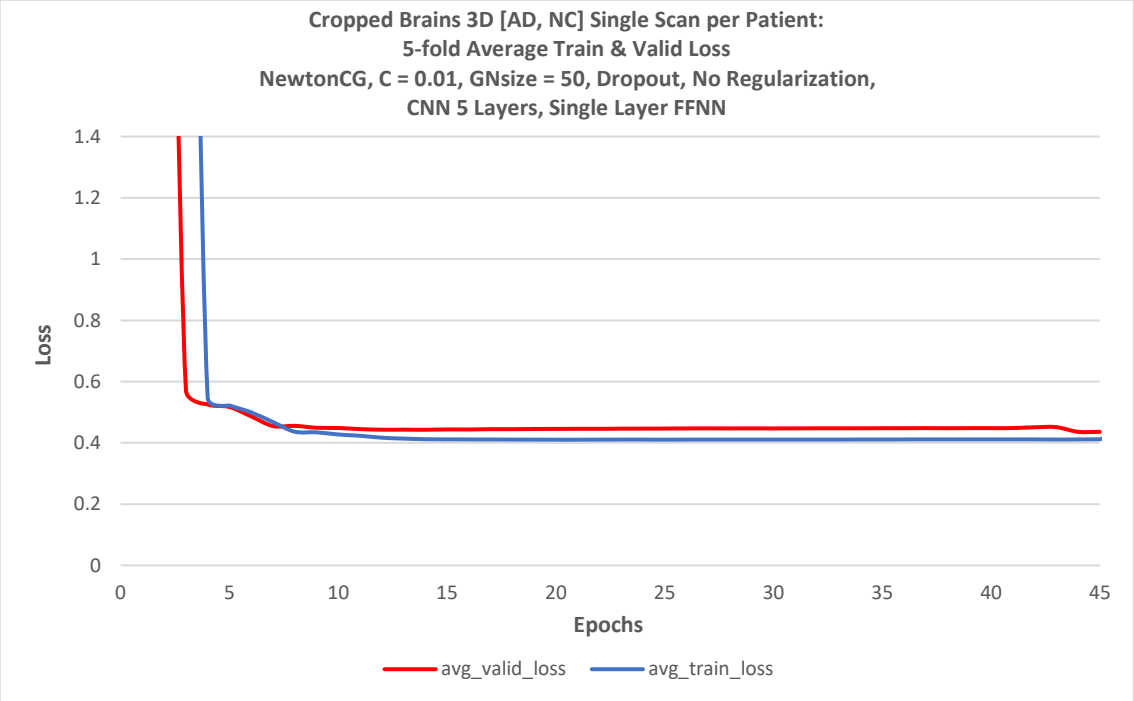


Figure N.19. Caption in the title.

N.3.1.2 Training

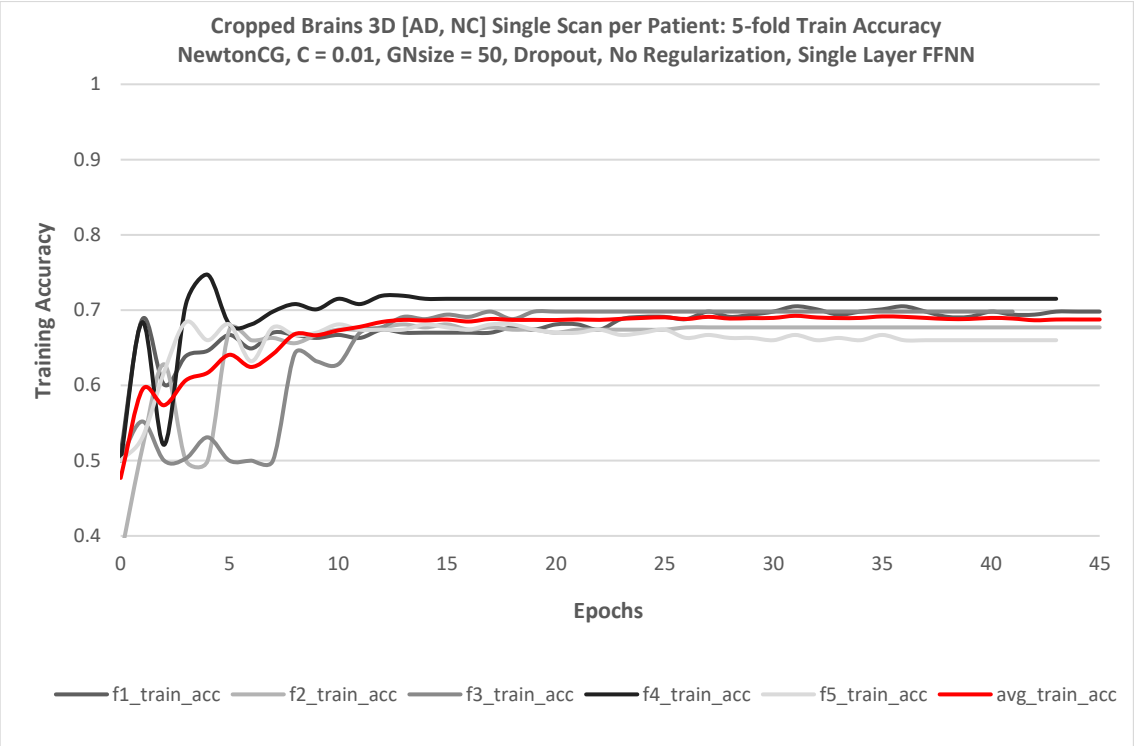


Figure N.20. Caption in the title.

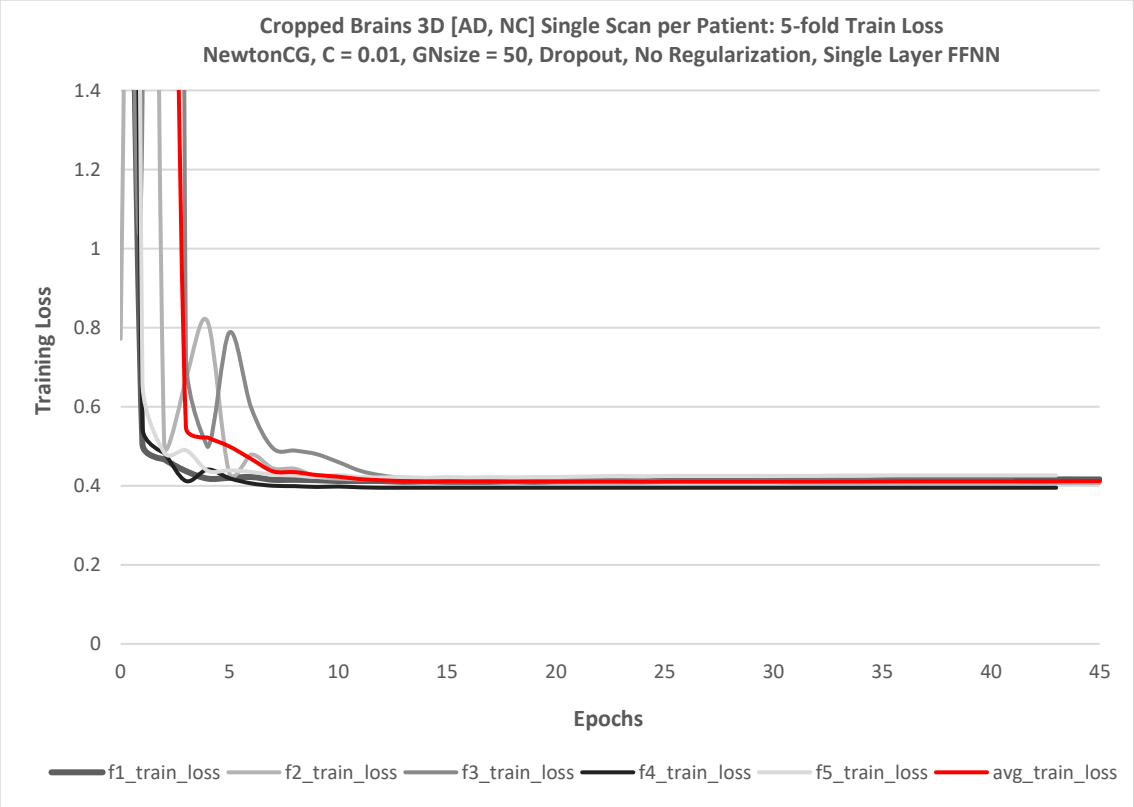


Figure N.21. Caption in the title.

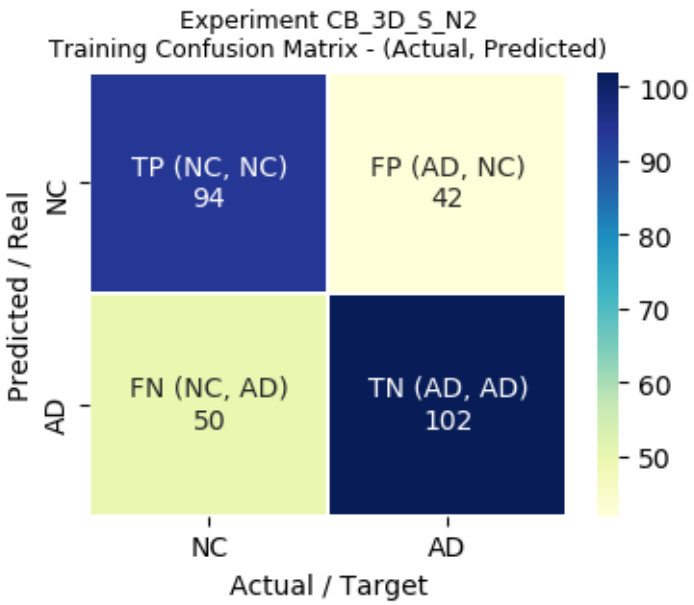


Figure N.22. Caption in the title.

N.3.1.3 Validation

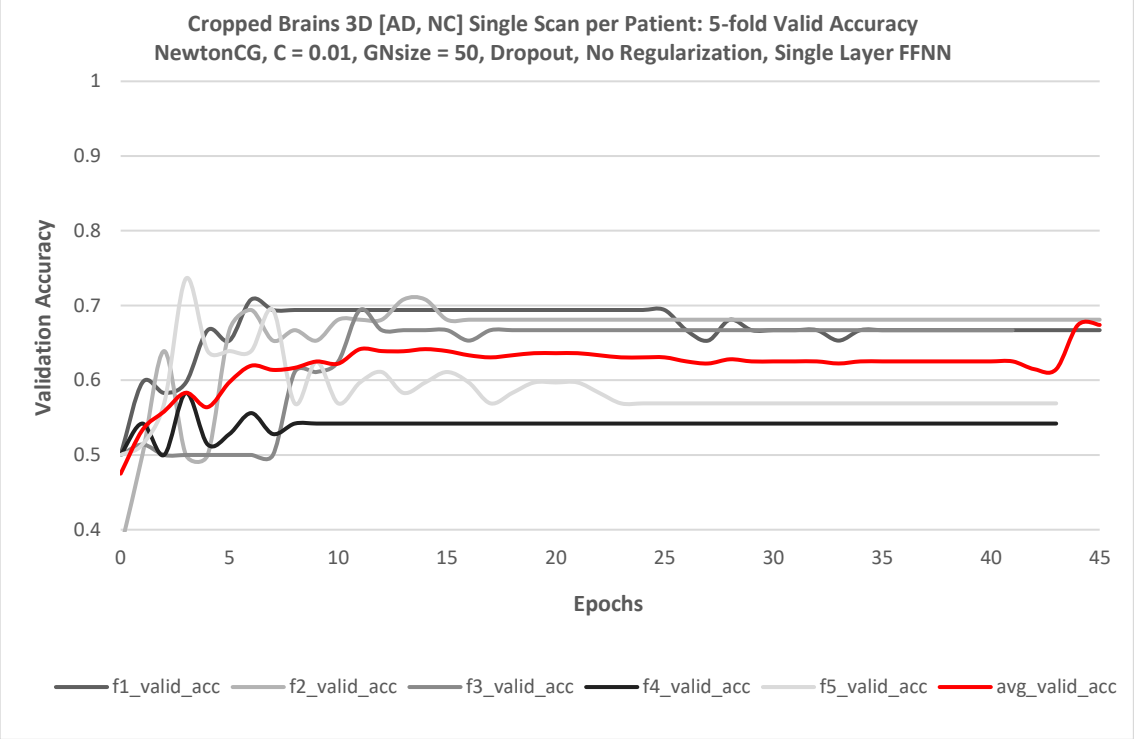


Figure N.23. Caption in the title.

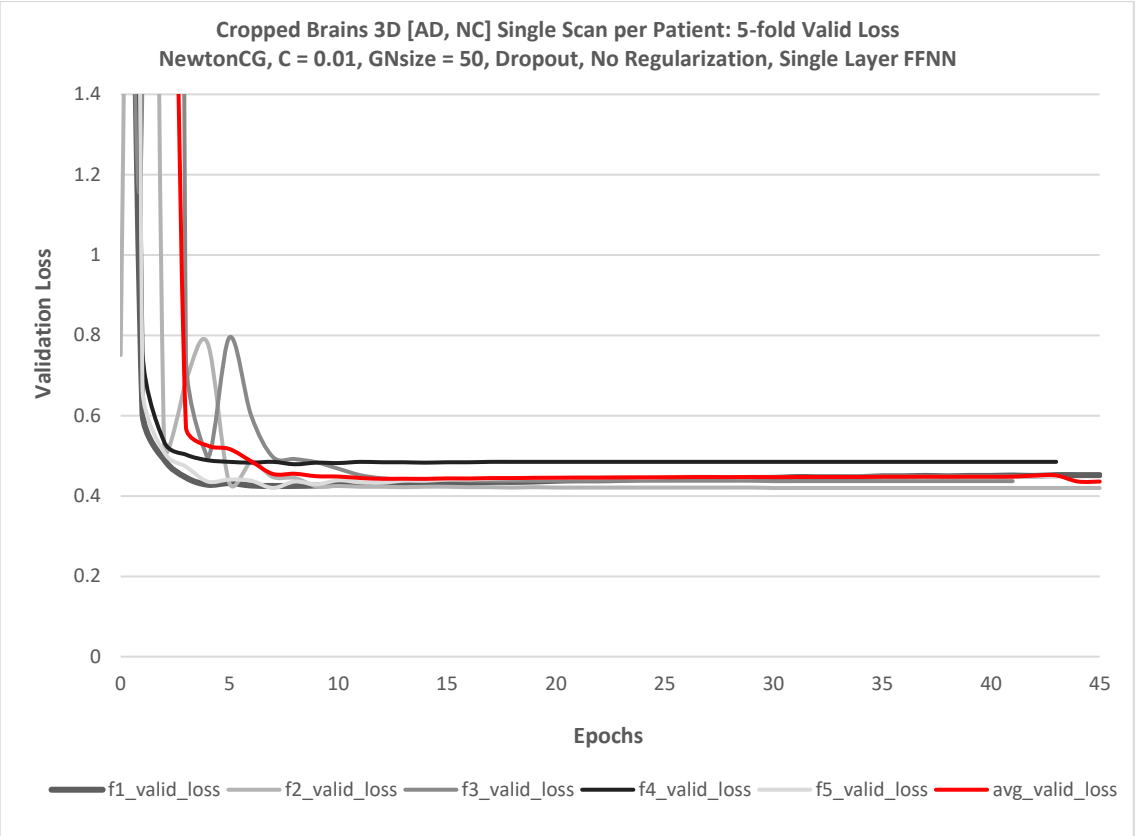


Figure N.24. Caption in the title.

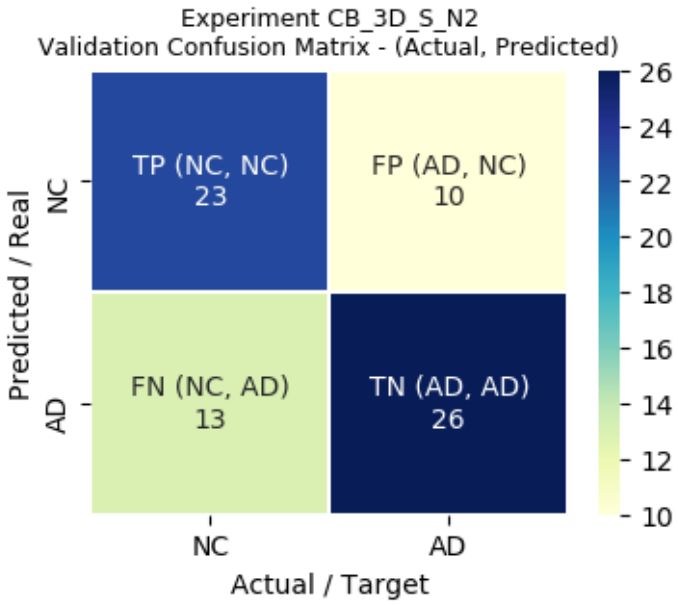


Figure N.25. Caption in the title.

N.3.1.4 Testing

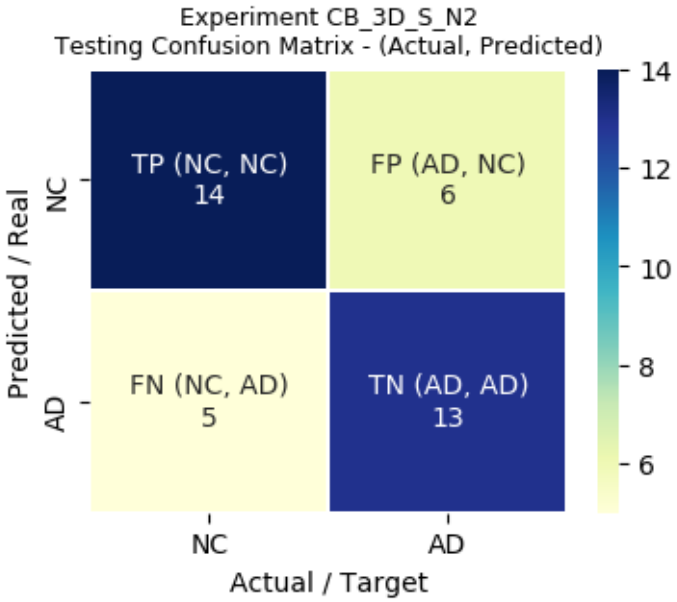


Figure N.26. Caption in the title.

Appendix O

2D Brain Slices [AD, MCI, NC]: Multiple Scans per Patient – Single Slice per Scan (174 × 174)

O.1 CNN 4 Layers with Adam

O.1.1 Experiment B_2D_M_A1 [AD, MCI, NC]: C = 0.01, lr = 0.001, Dropout

O.1.1.1 General

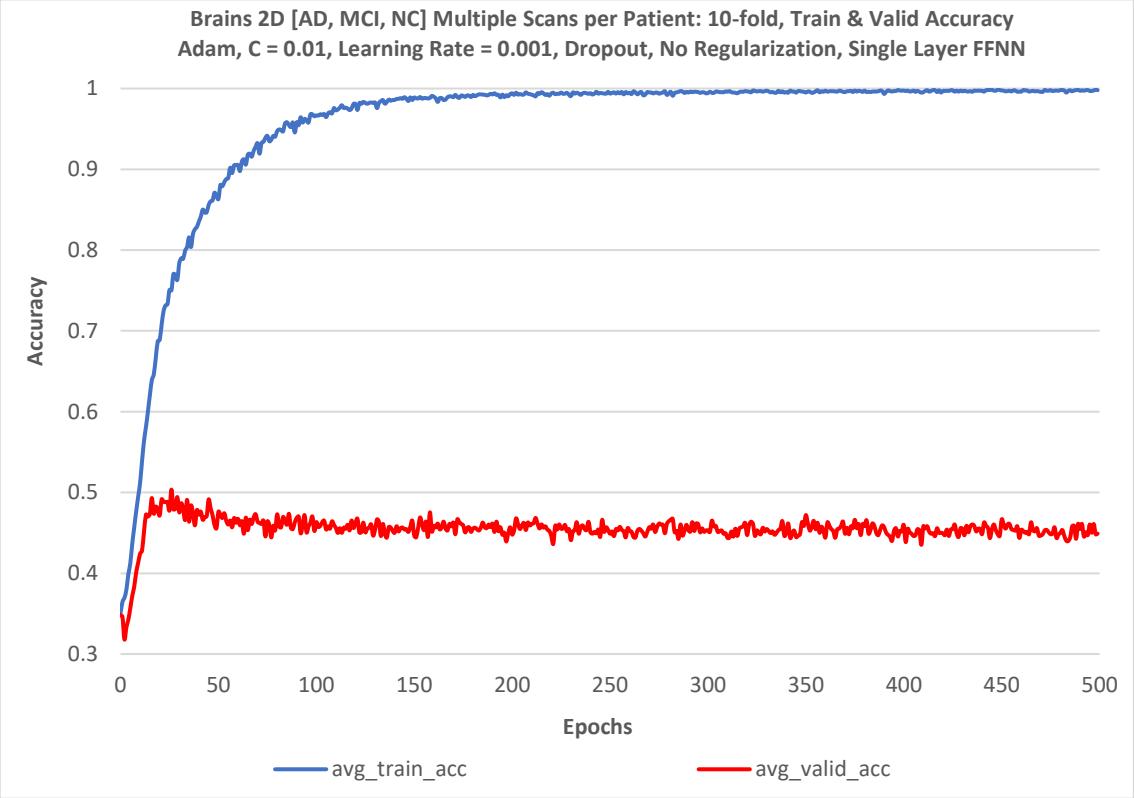


Figure O.1. Caption in the title.

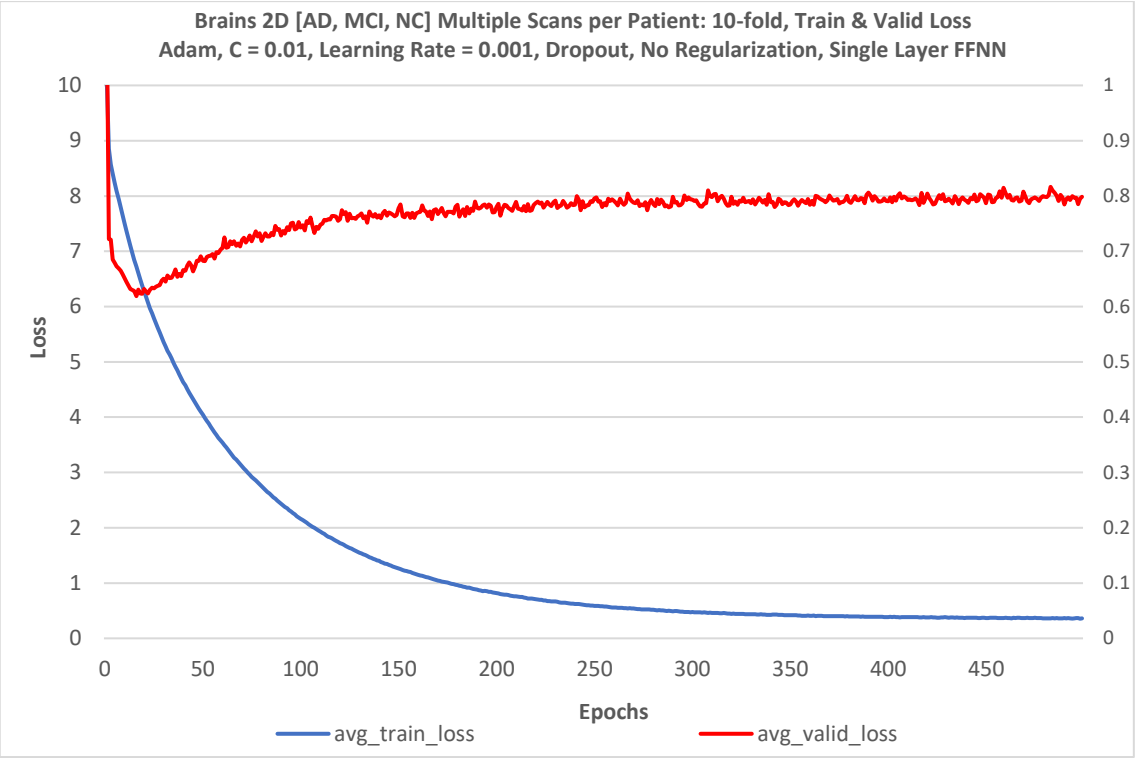


Figure O.2. Caption in the title.

O.1.1.2 Training

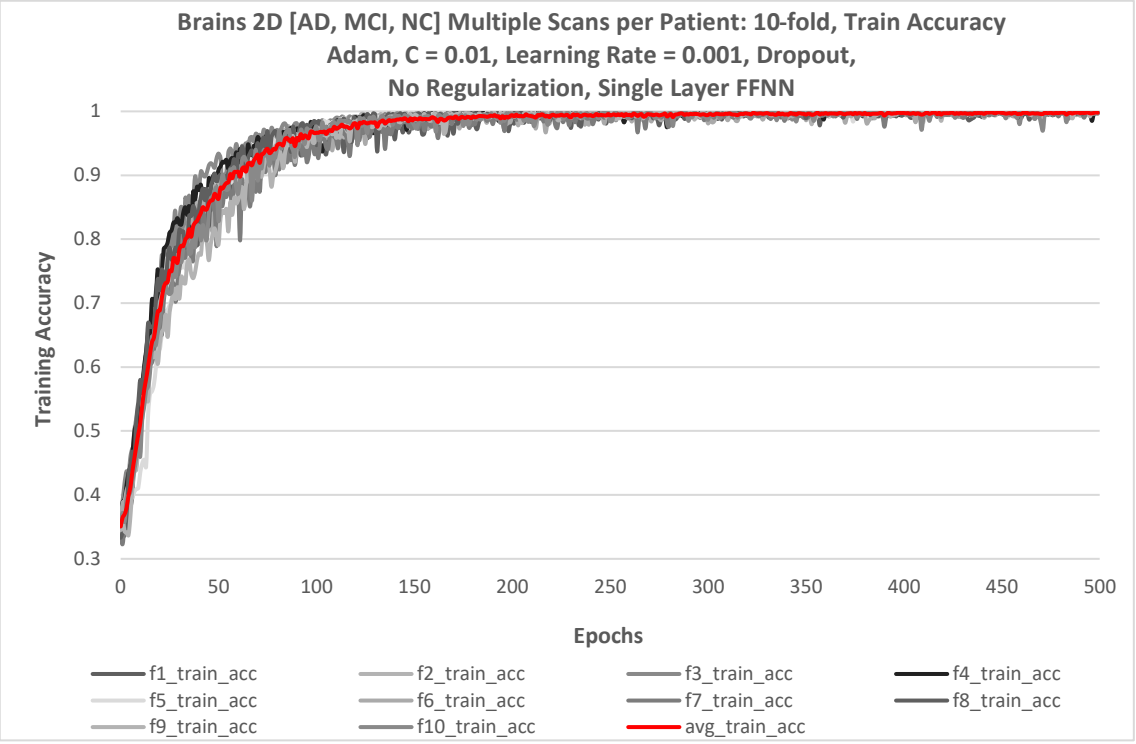


Figure O.3. Caption in the title.

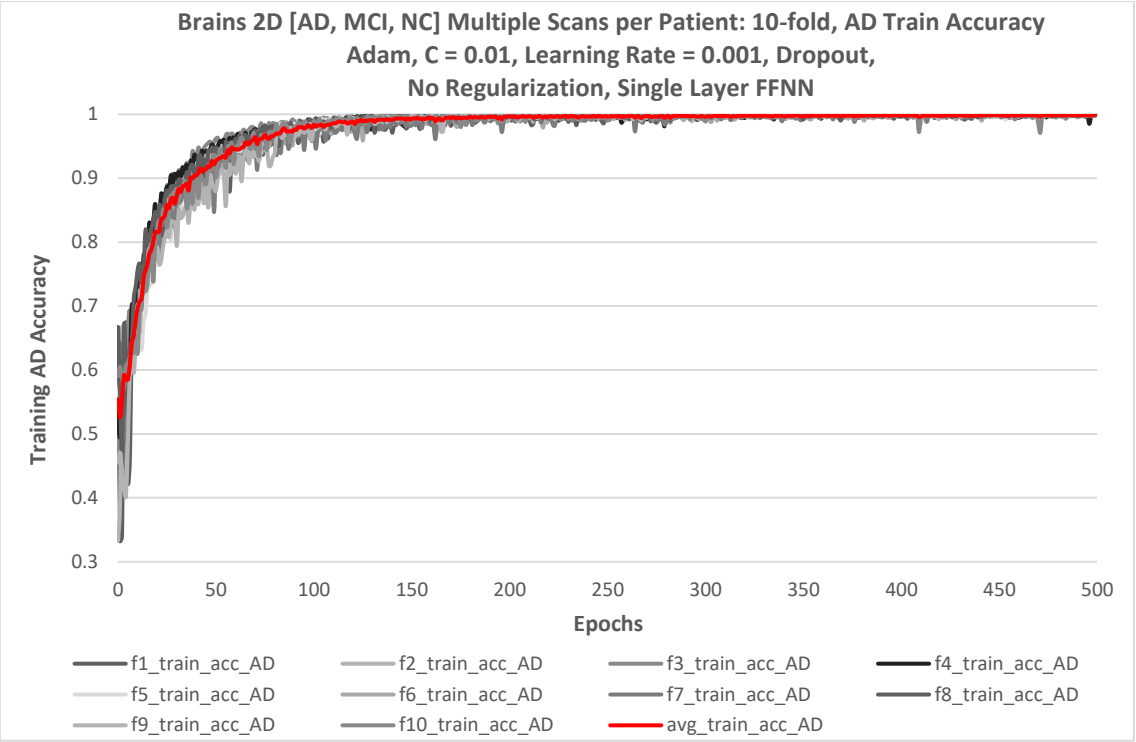


Figure O.4. Caption in the title.

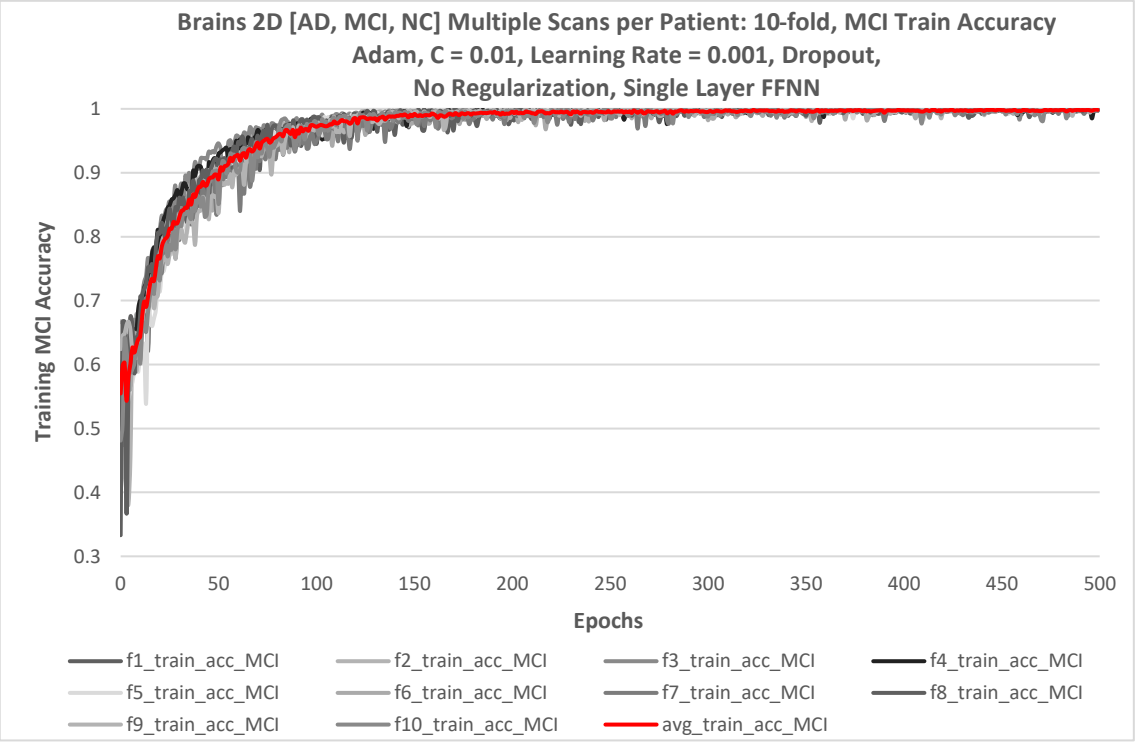


Figure O.5. Caption in the title.

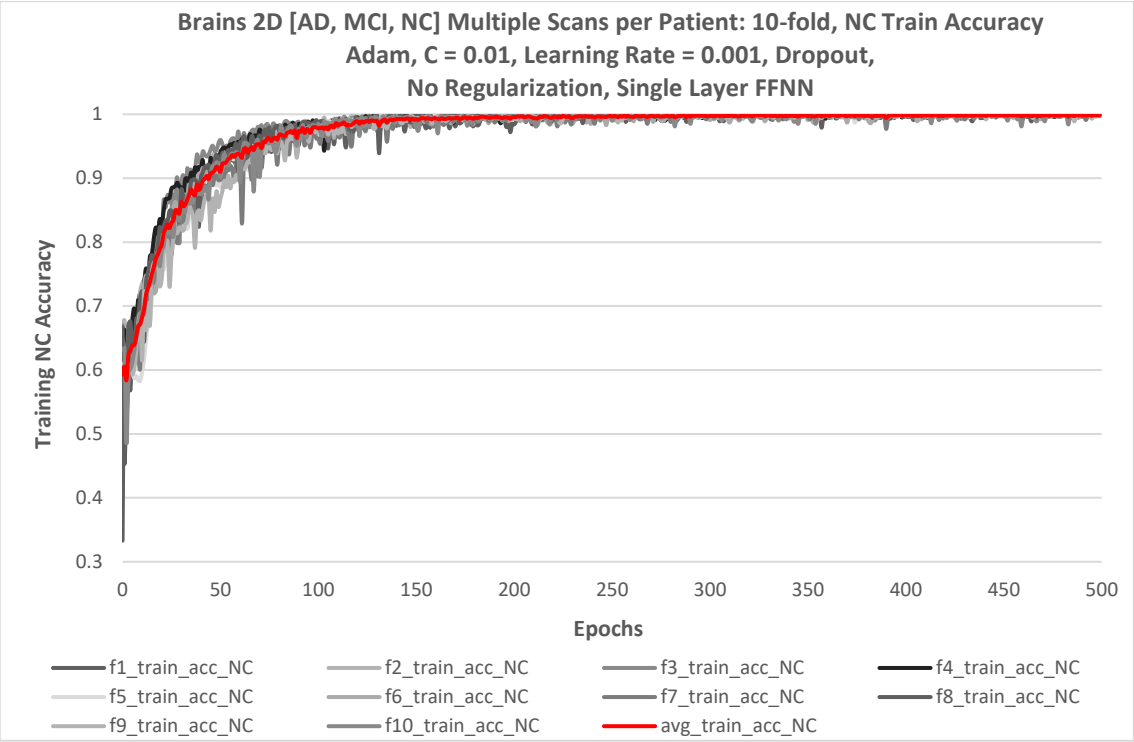


Figure O.6. Caption in the title.

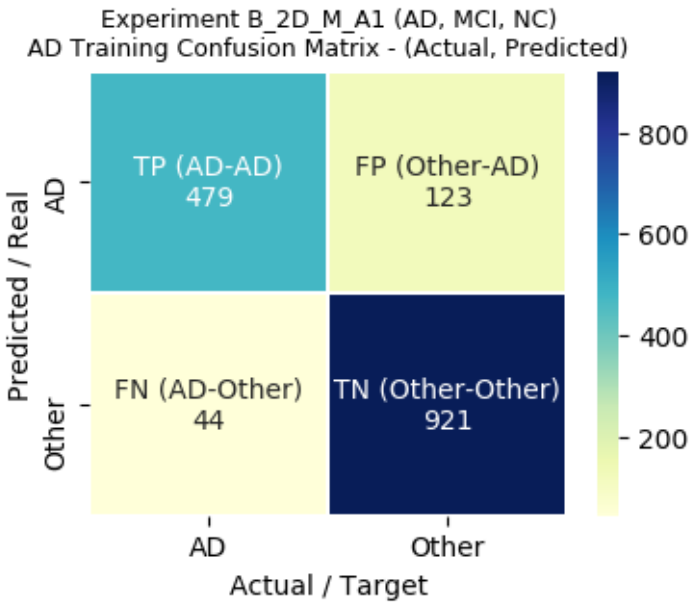


Figure O.7. Caption in the title.

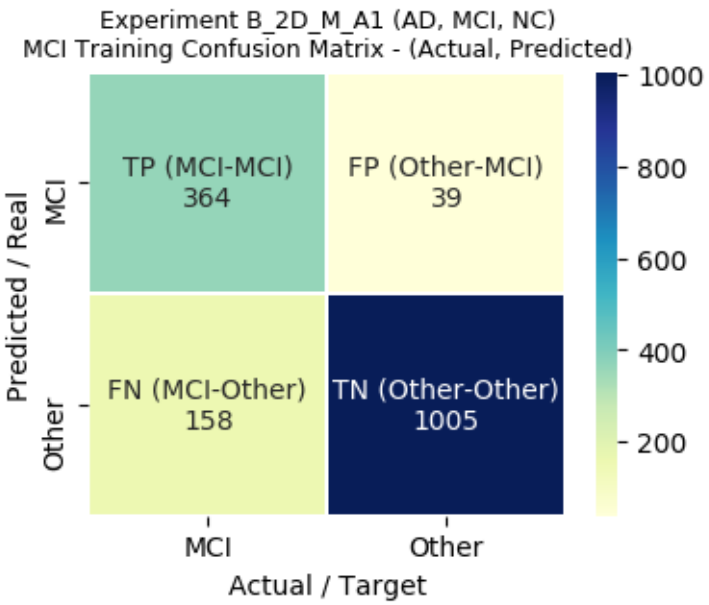


Figure O.8. Caption in the title.

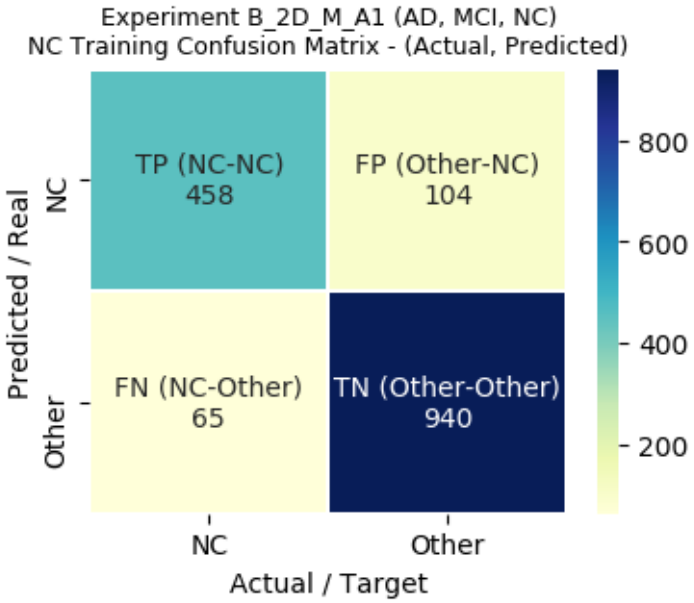


Figure O.9. Caption in the title.

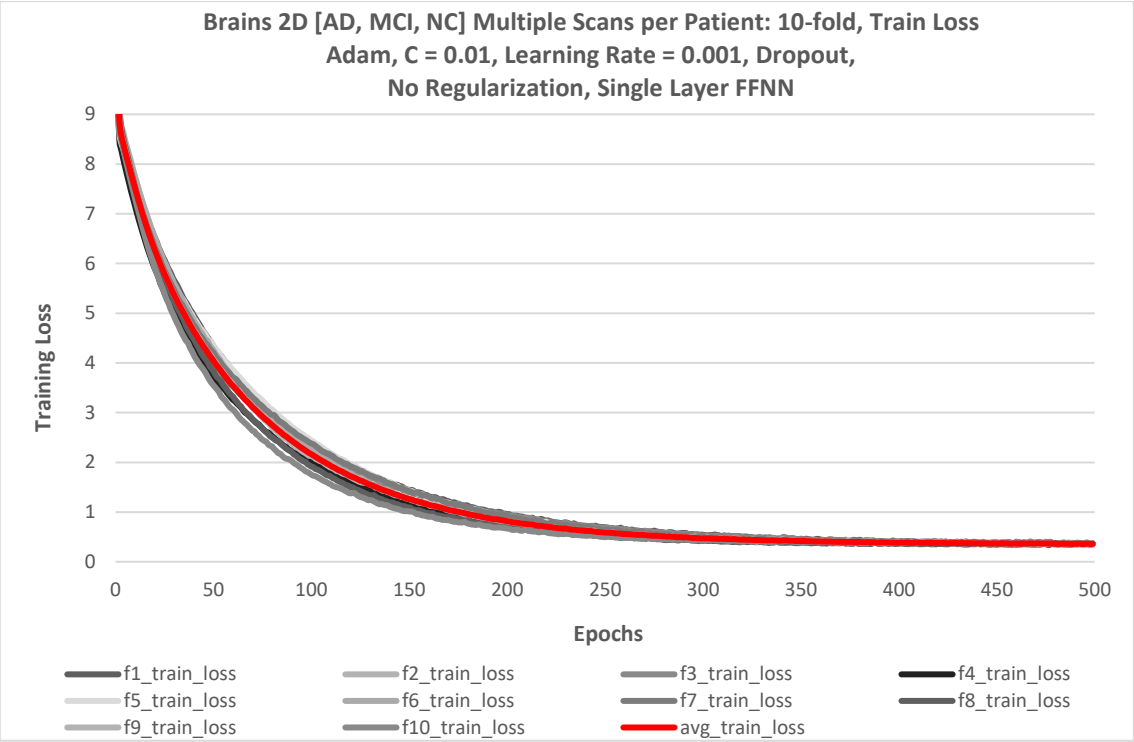


Figure O.10. Caption in the title.

O.1.1.3 Validation

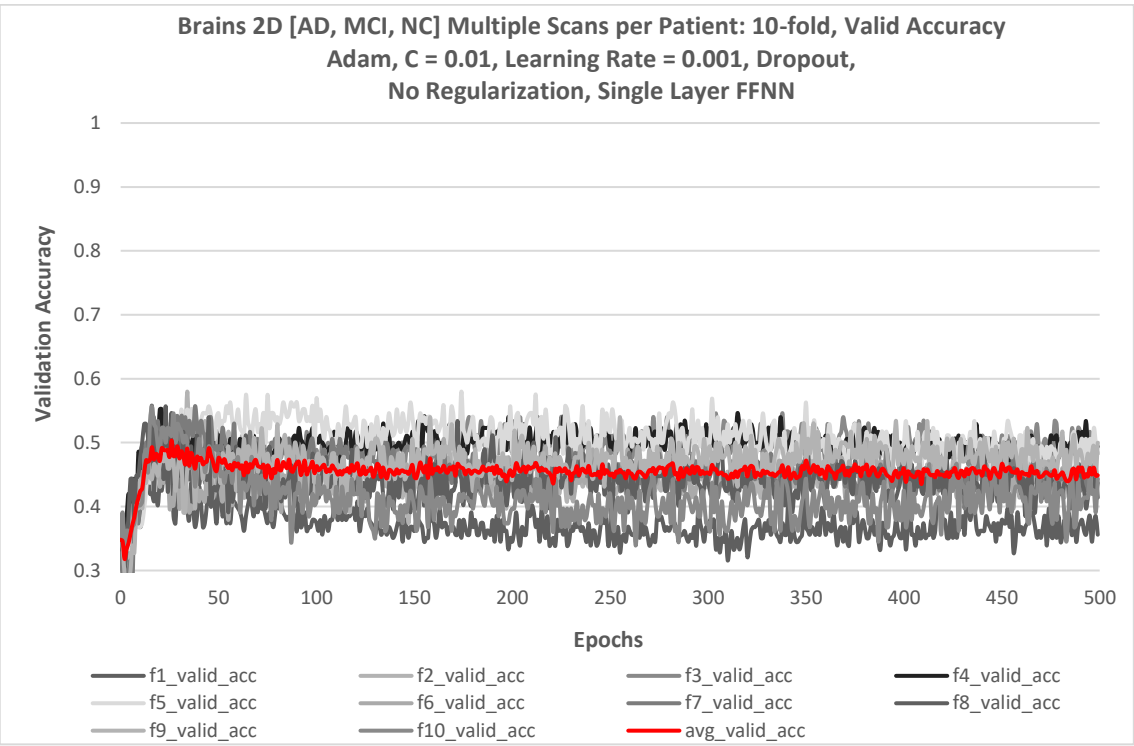


Figure O.11. Caption in the title.

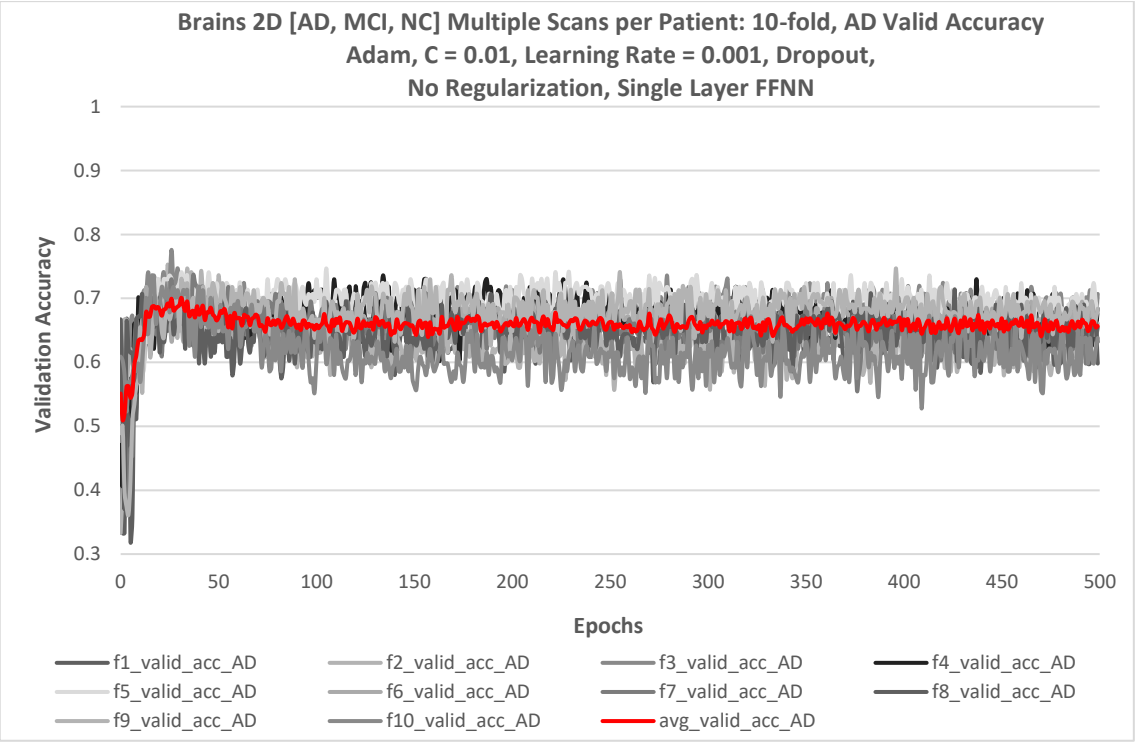


Figure O.12. Caption in the title.

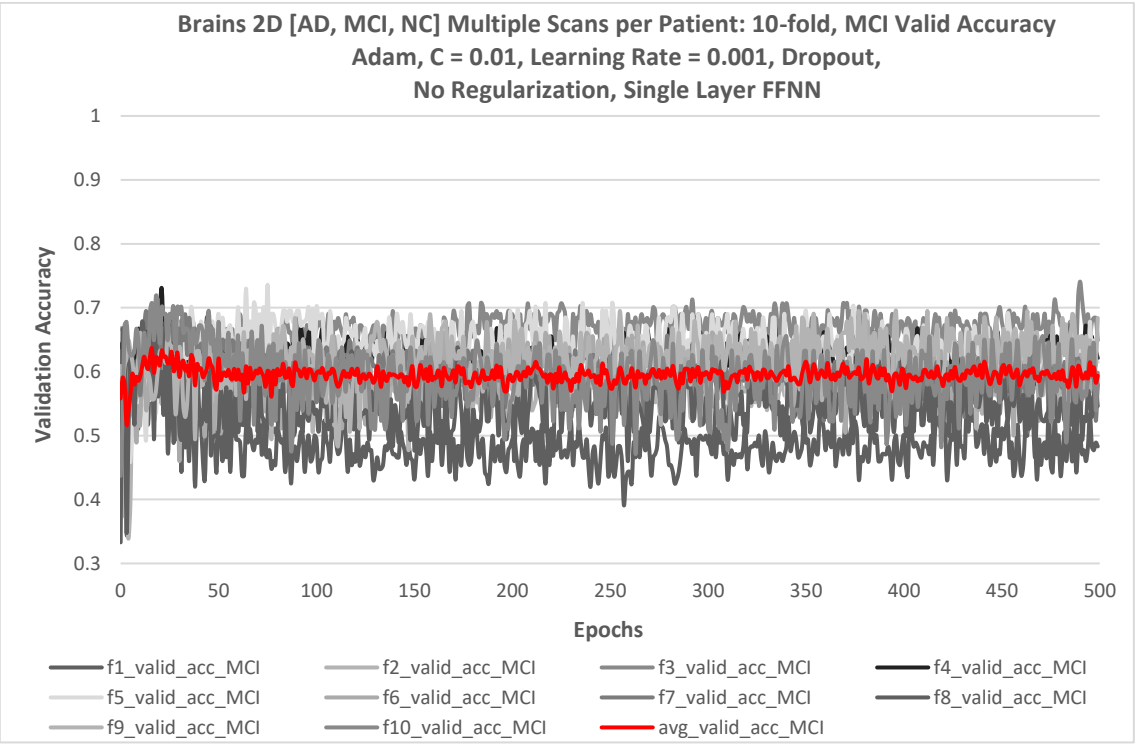


Figure O.13. Caption in the title.

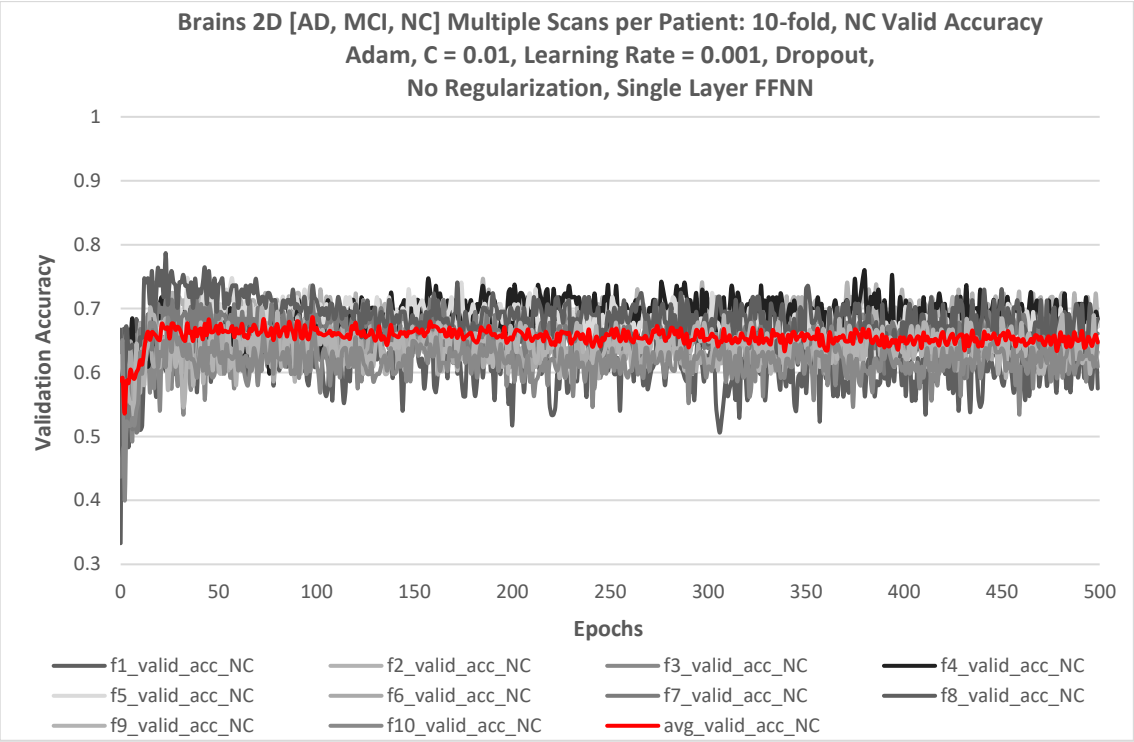


Figure O.14. Caption in the title.

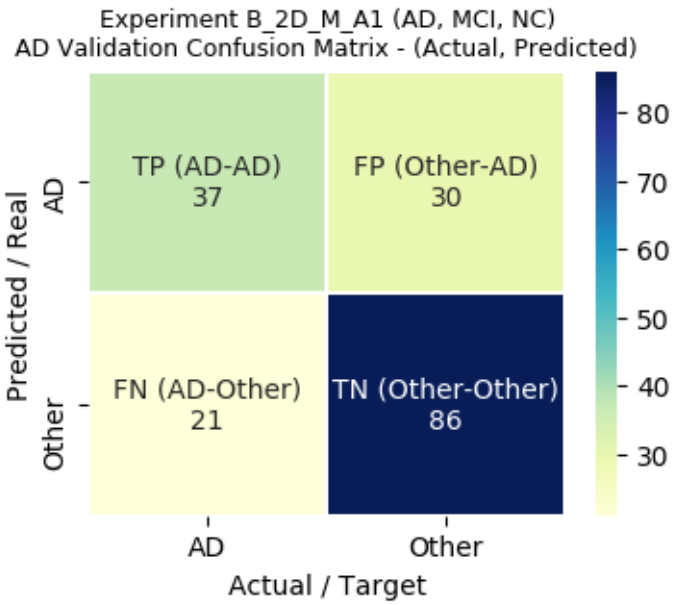


Figure O.15. Caption in the title.

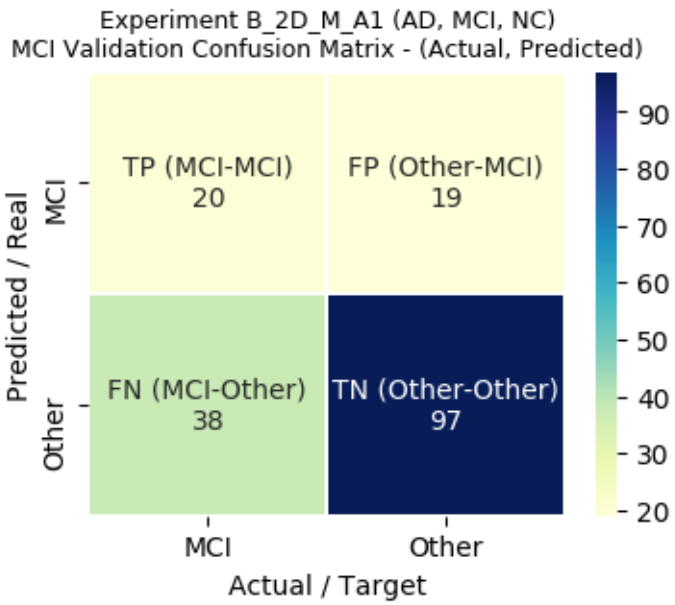


Figure O.16. Caption in the title.

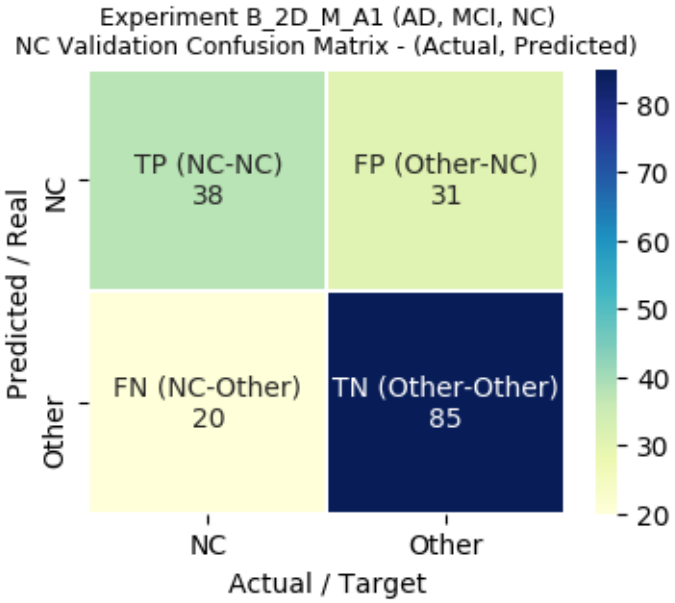


Figure O.17. Caption in the title.

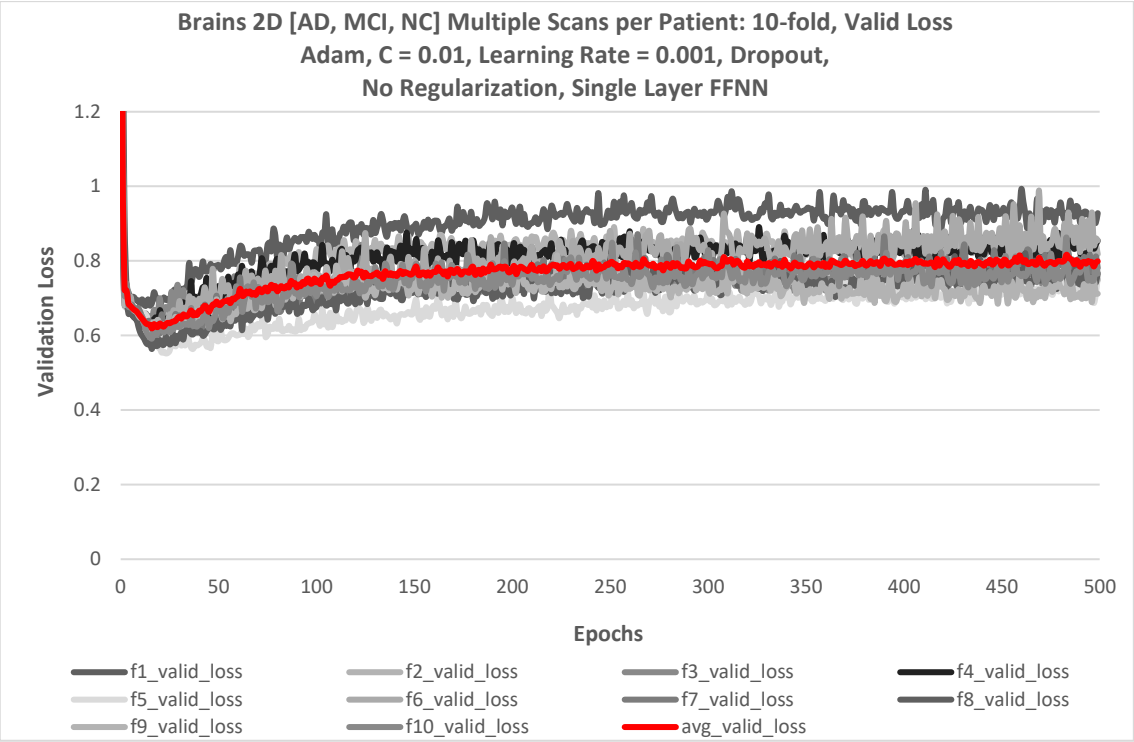


Figure O.18. Caption in the title.

O.1.1.4 Testing

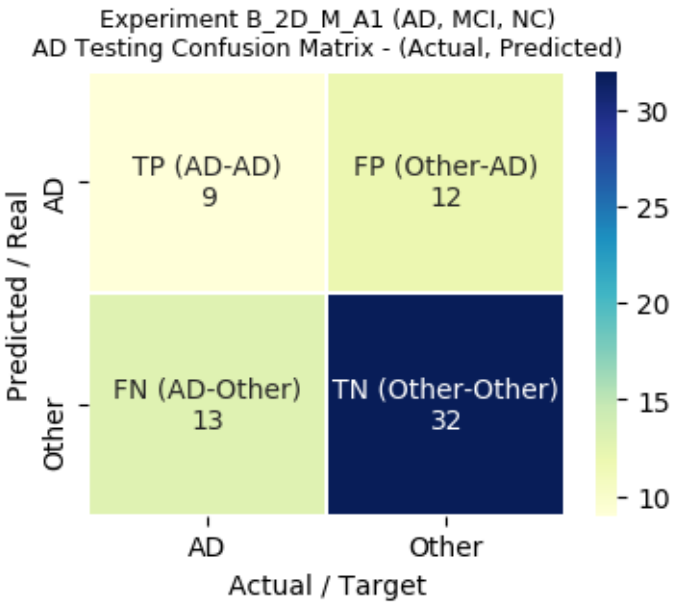


Figure O.19. Caption in the title.

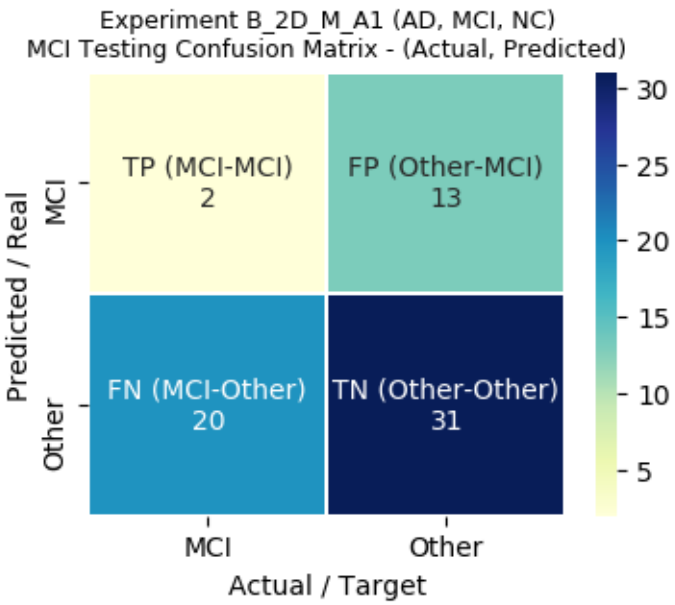


Figure O.20. Caption in the title.

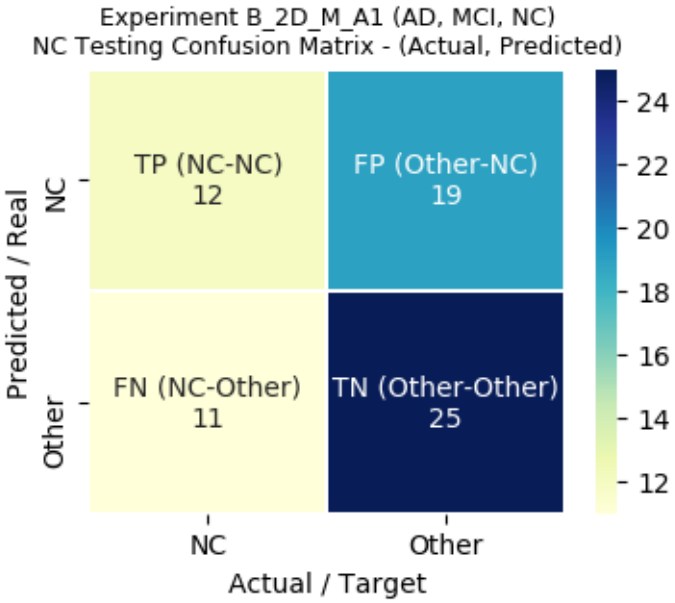


Figure O.21. Caption in the title.

O.1.2 Experiment B_2D_M_A2 [AD, MCI, NC]: C = 0.01, lr = 0.0001, Dropout

O.1.2.1 General

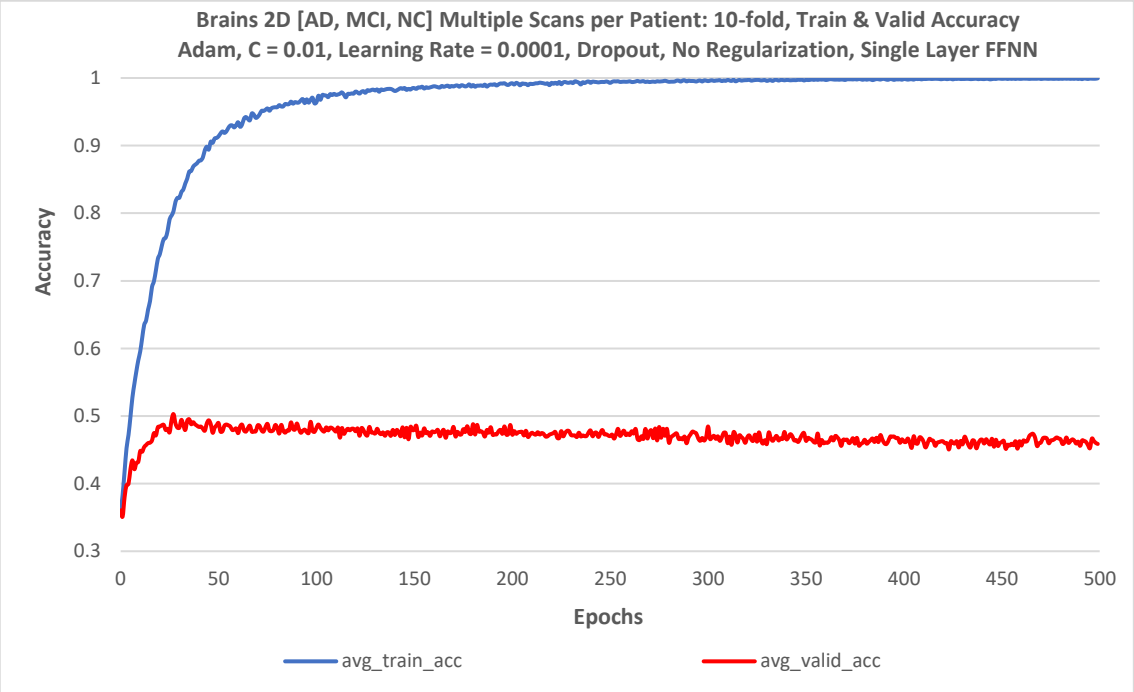


Figure O.22. Caption in the title.

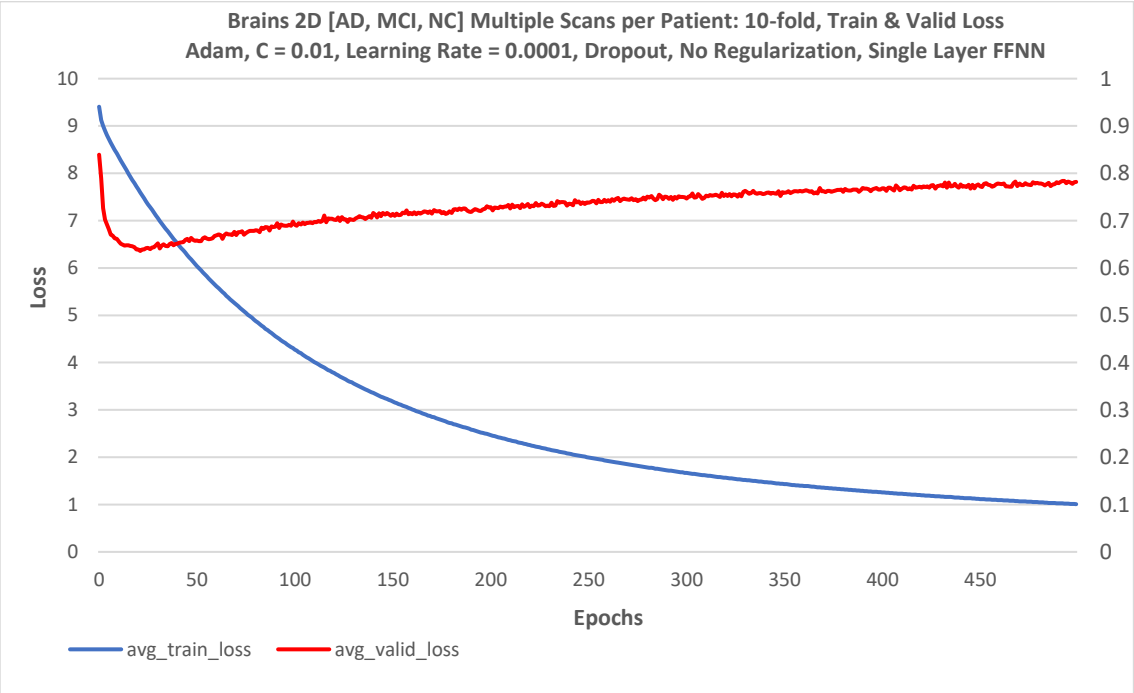


Figure O.23. Caption in the title.

O.1.2.2 Training

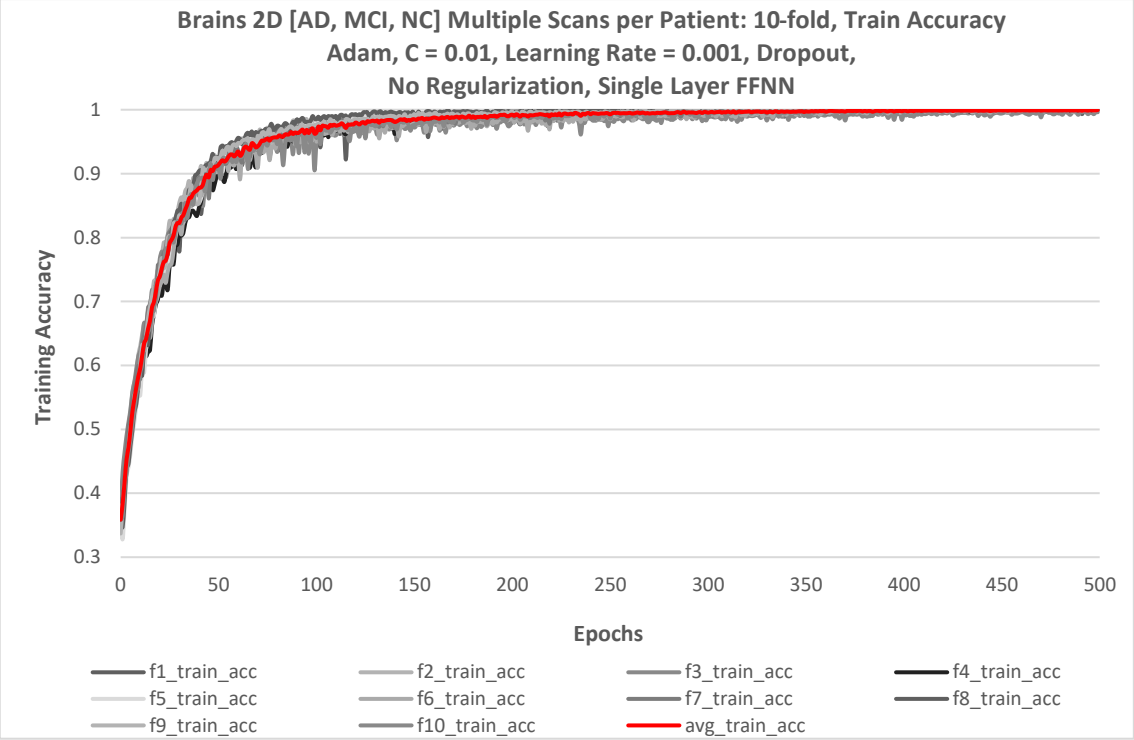


Figure O.24. Caption in the title.

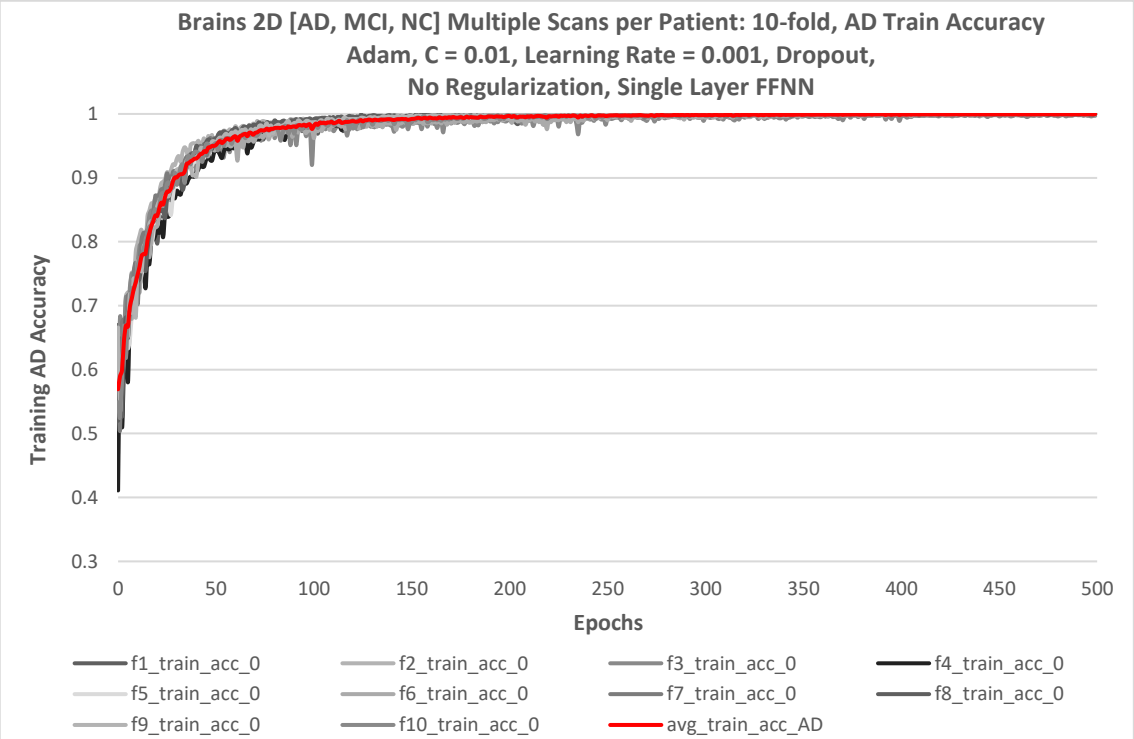


Figure O.25. Caption in the title.

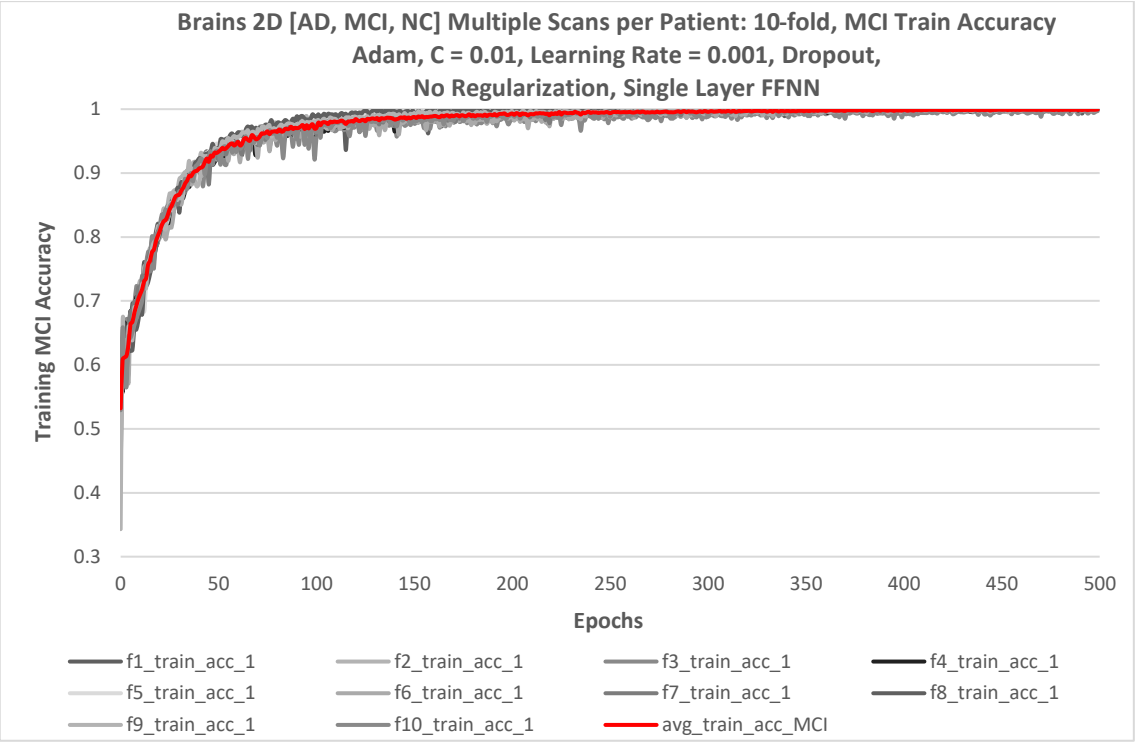


Figure O.26. Caption in the title.

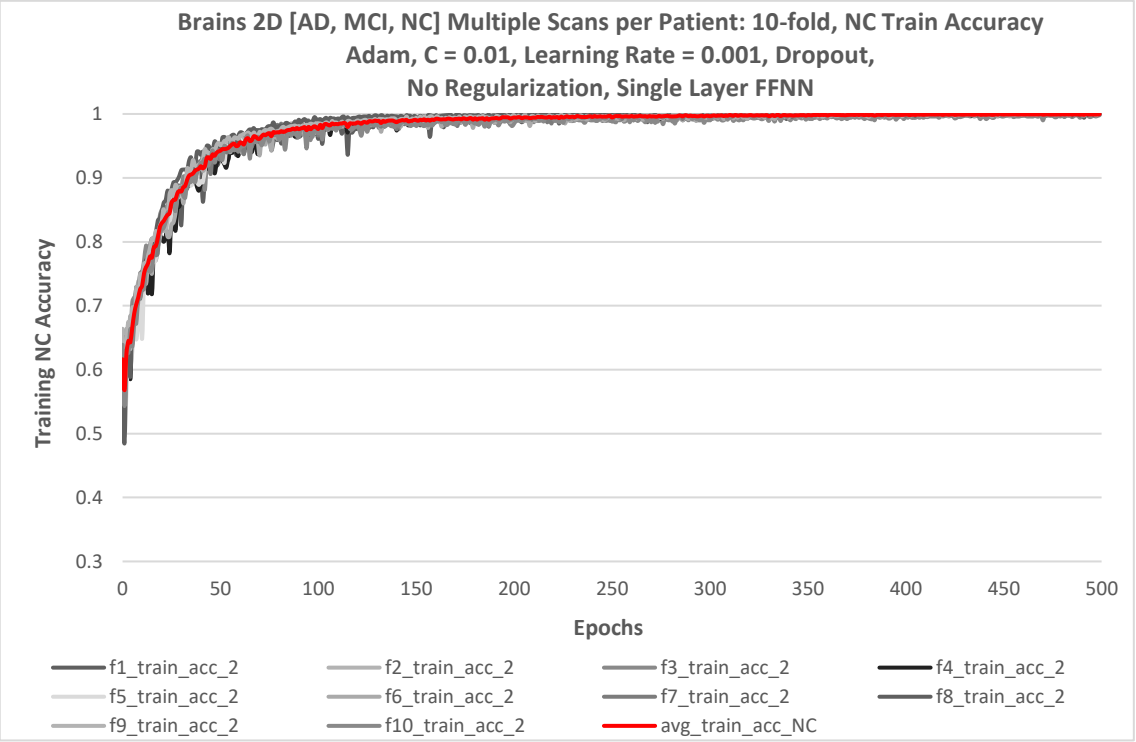


Figure O.27. Caption in the title.

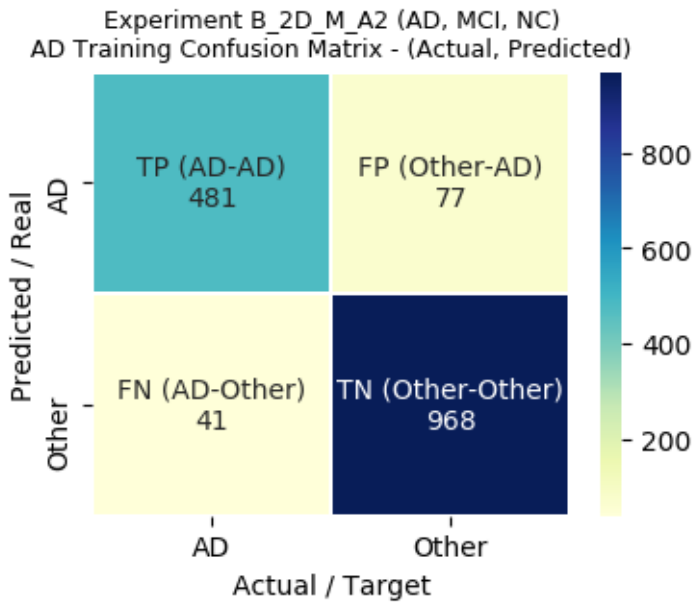


Figure O.28. Caption in the title.

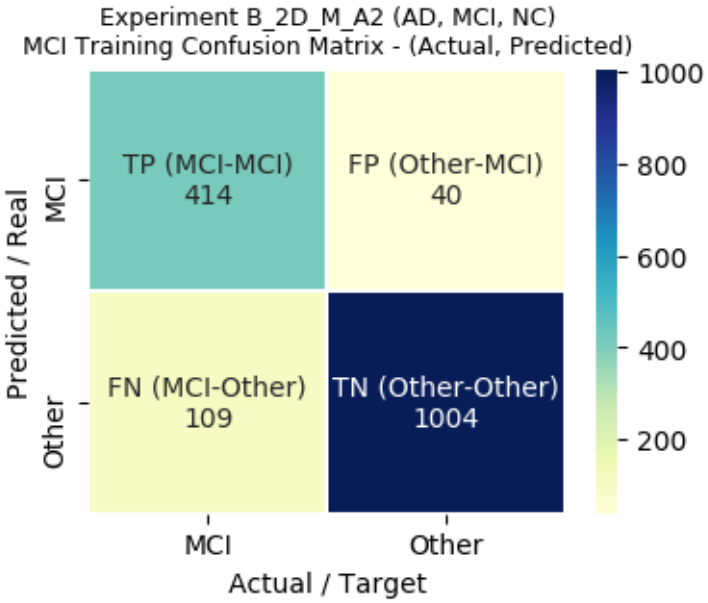


Figure O.29. Caption in the title.

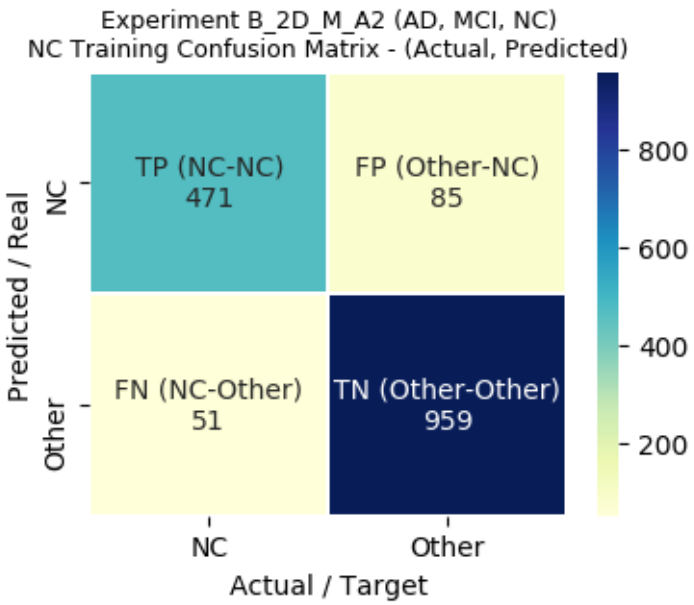


Figure O.30. Caption in the title.

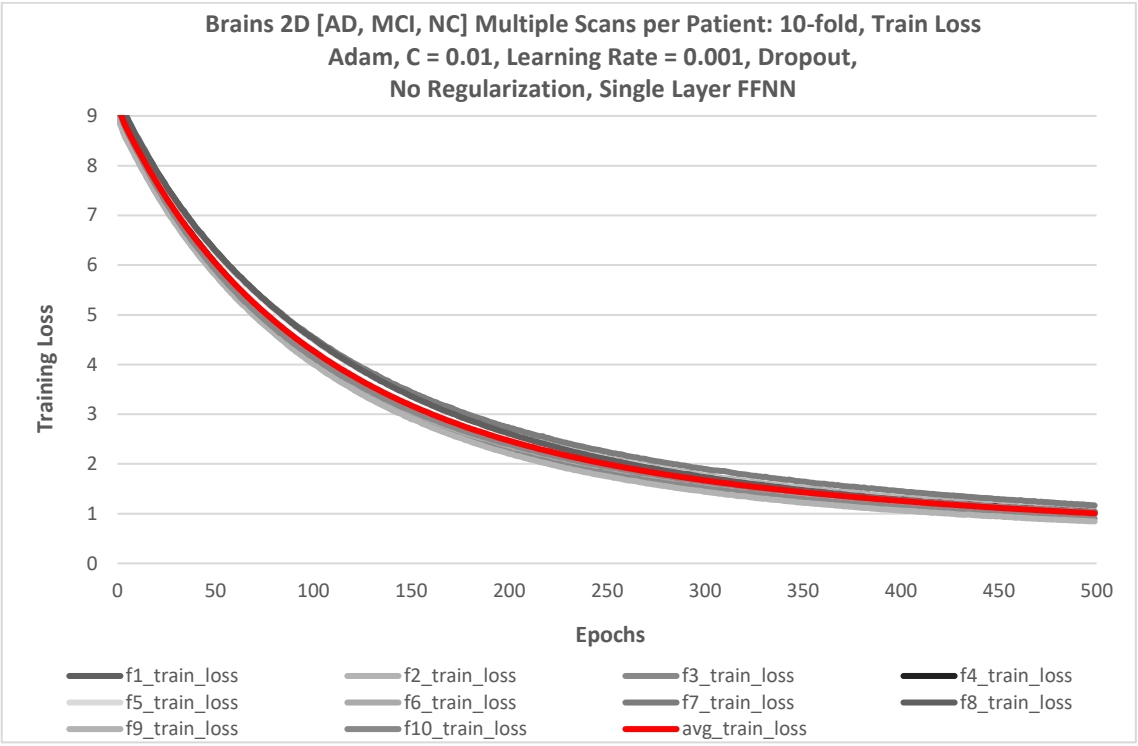


Figure O.31. Caption in the title.

O.1.2.3 Validation

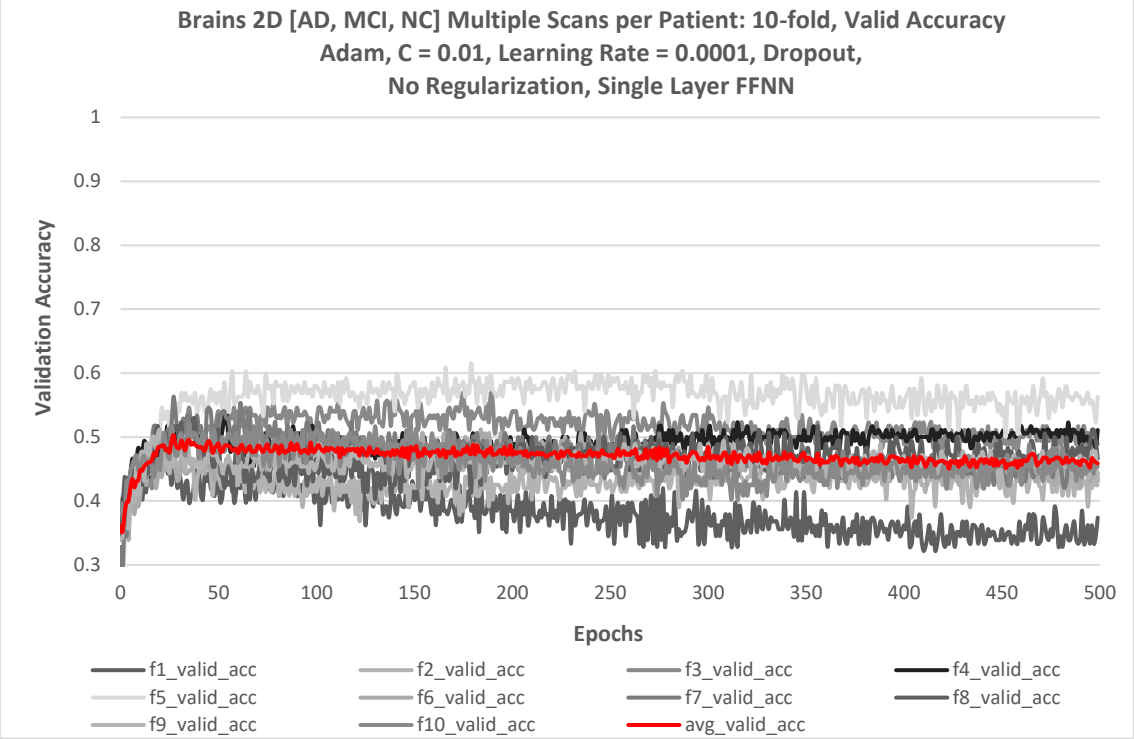


Figure O.32. Caption in the title.

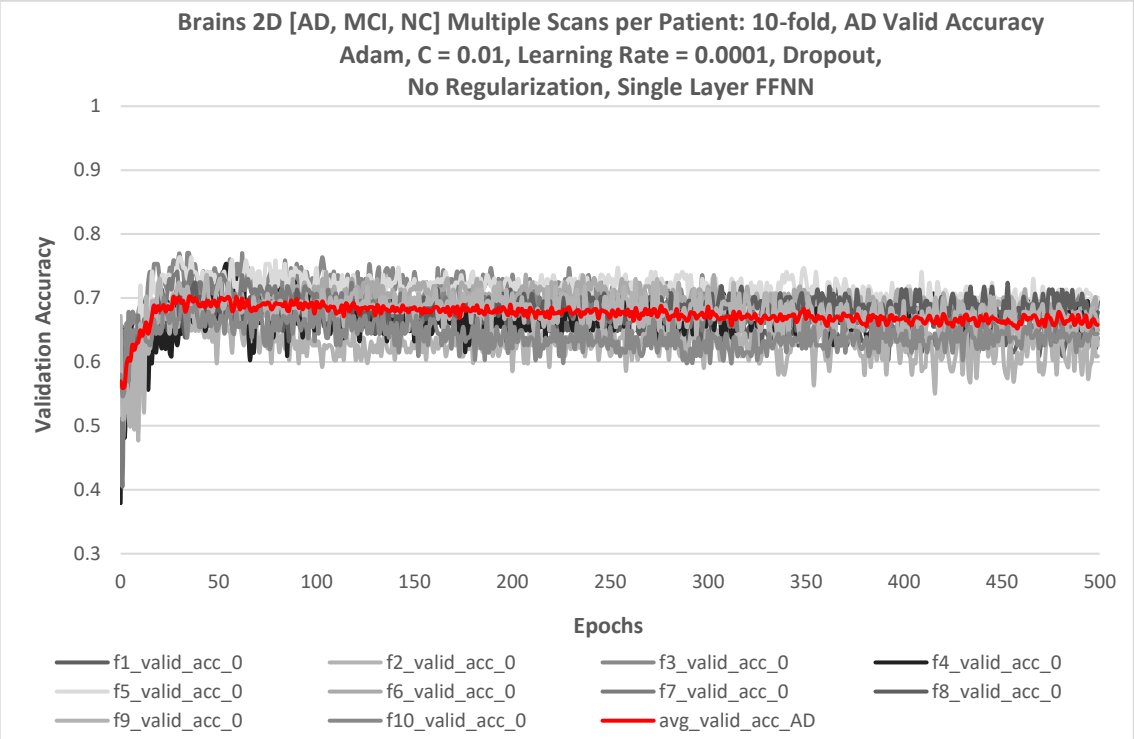


Figure O.33. Caption in the title.

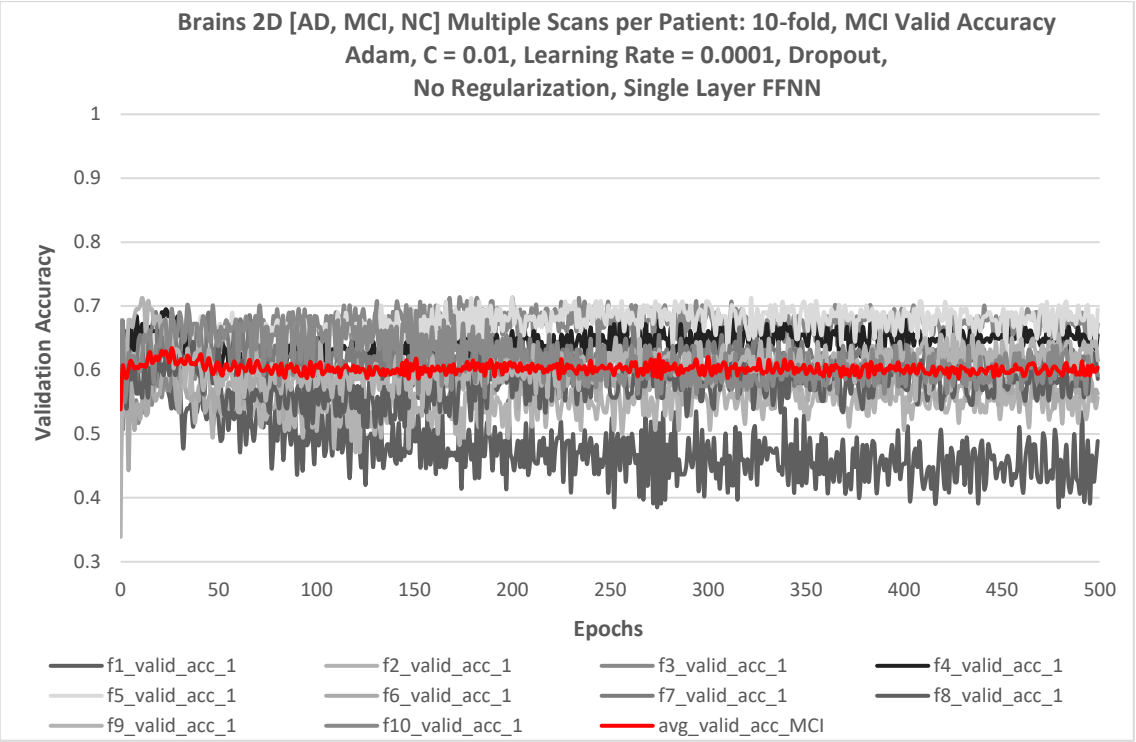


Figure O.34. Caption in the title.

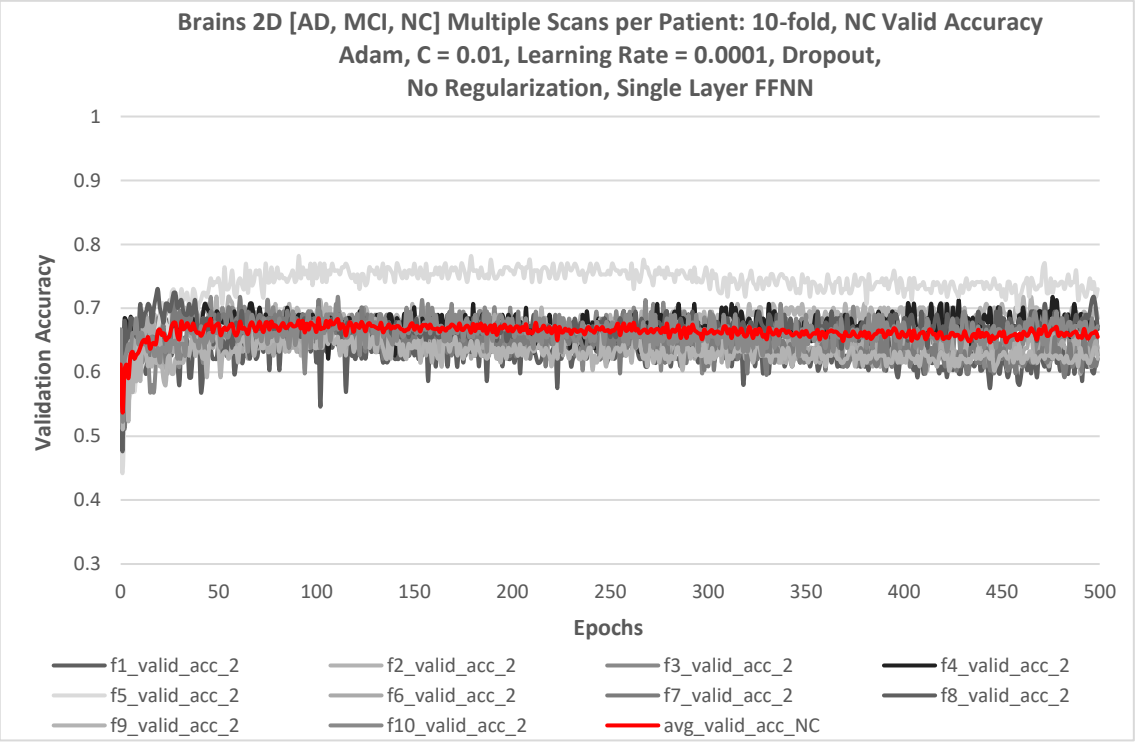


Figure O.35. Caption in the title.

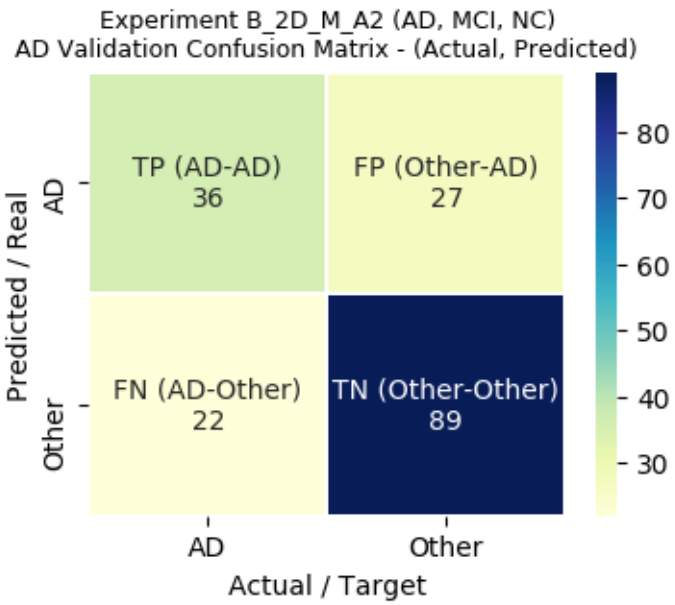


Figure O.36. Caption in the title.

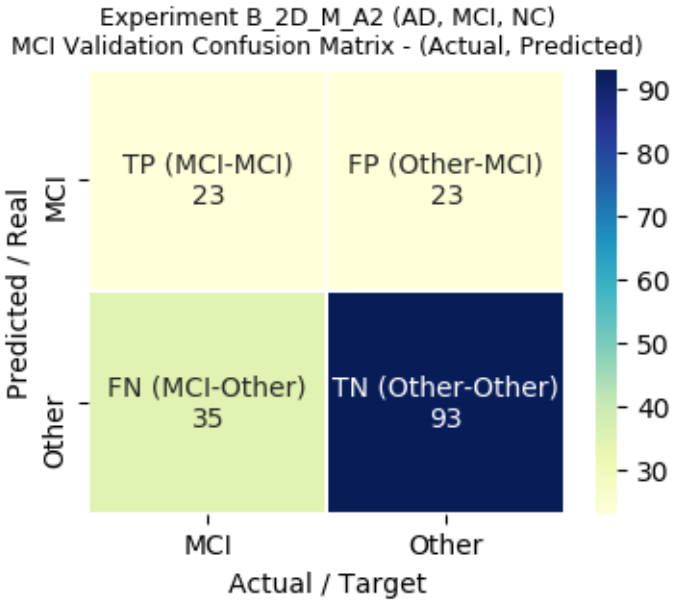


Figure O.37. Caption in the title.

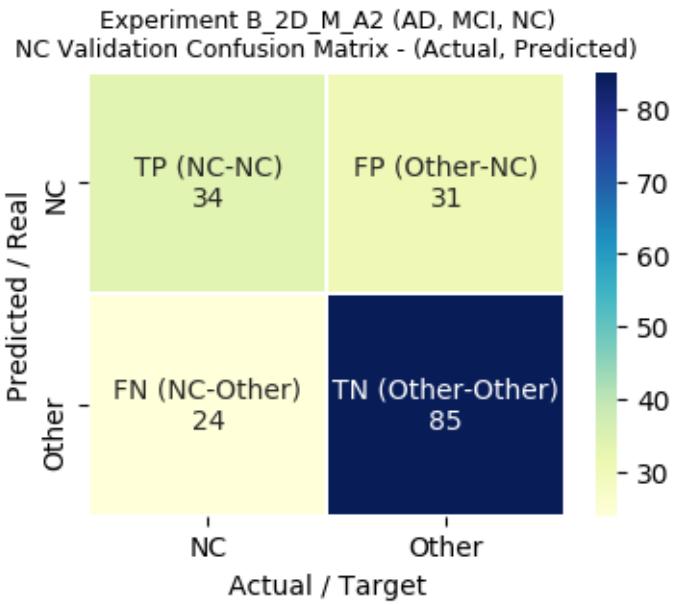


Figure O.38. Caption in the title.

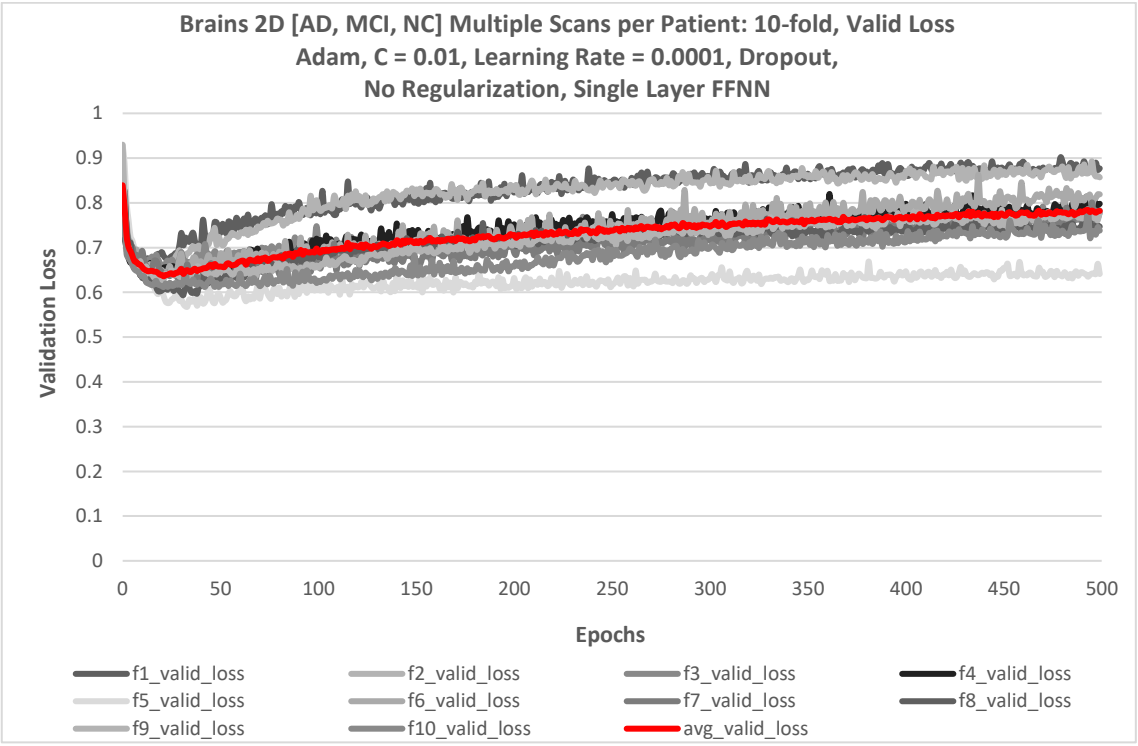


Figure O.39. Caption in the title.

O.1.2.4 Testing

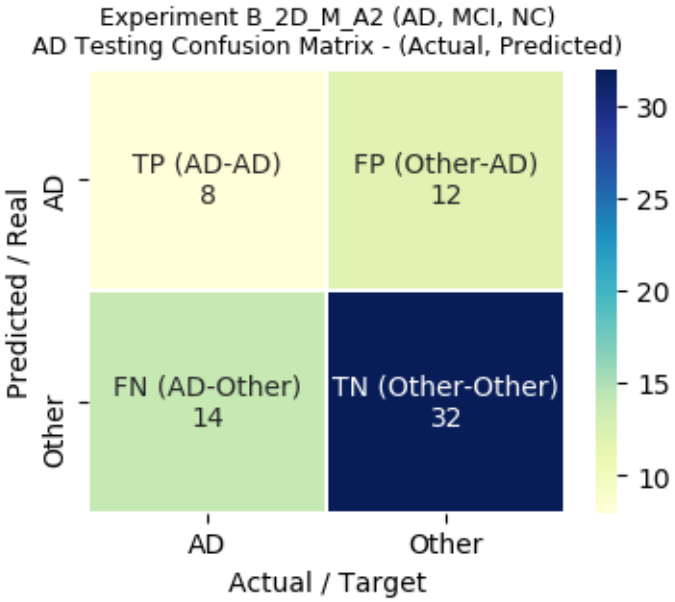


Figure O.40. Caption in the title.

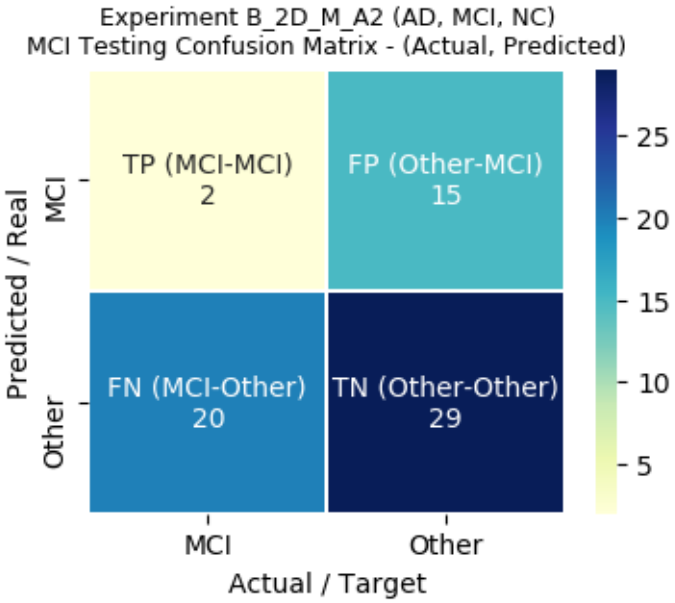


Figure O.41. Caption in the title.

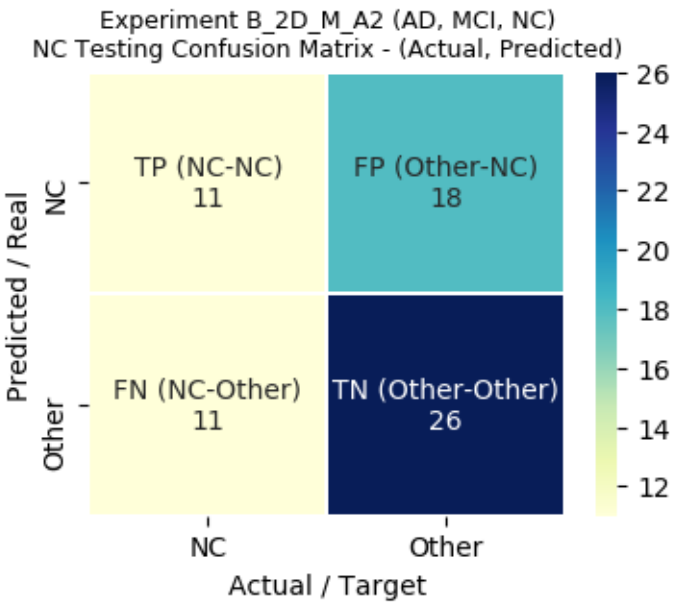


Figure O.42. Caption in the title.

O.2 CNN 4 Layers with NewtonCG

O.2.1 Experiment B_2D_M_N1 [AD, MCI, NC]: C = 0.01, GNsize = 50, Dropout

O.2.1.1 General

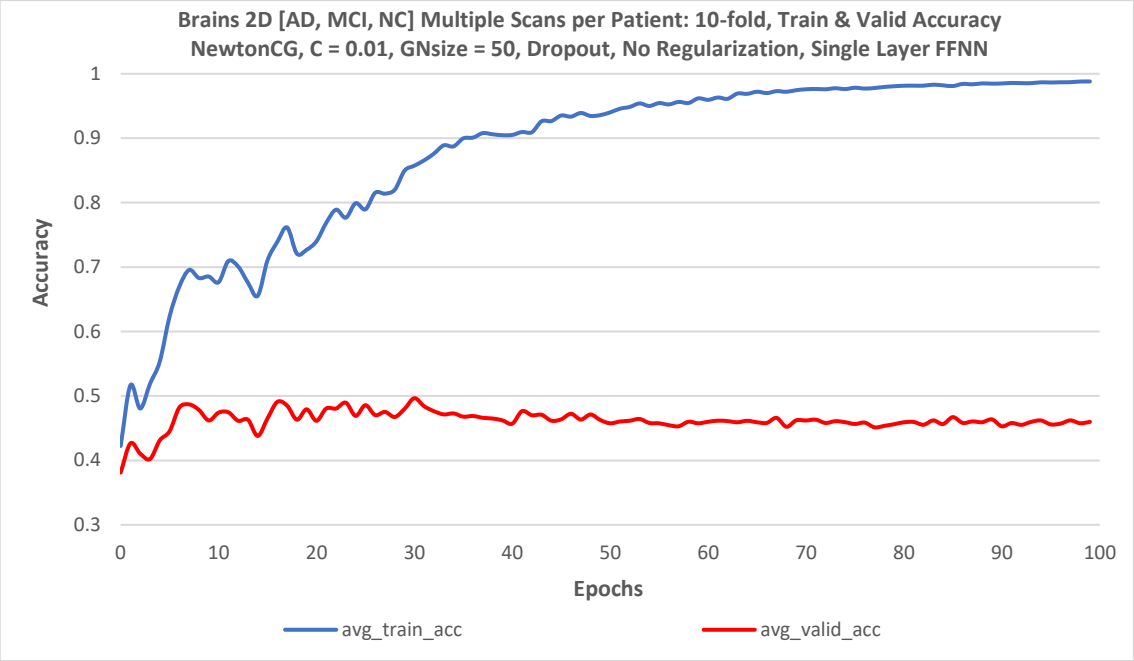


Figure O.43. Caption in the title.

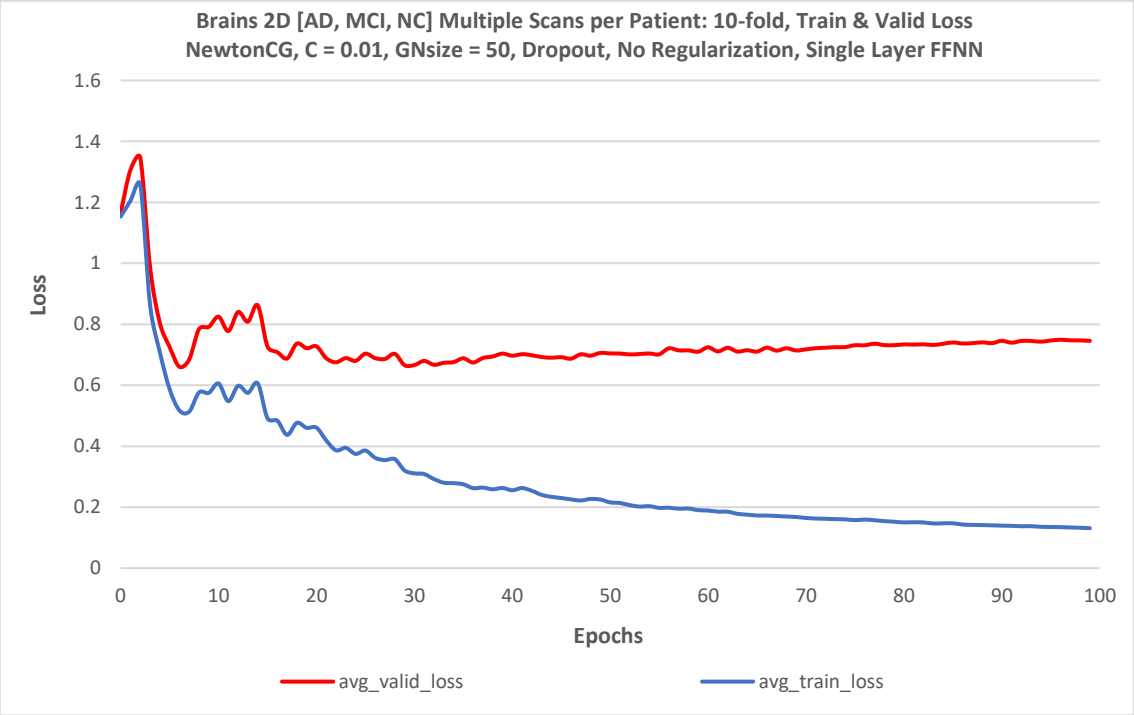


Figure O.44. Caption in the title.

O.2.1.2 Training

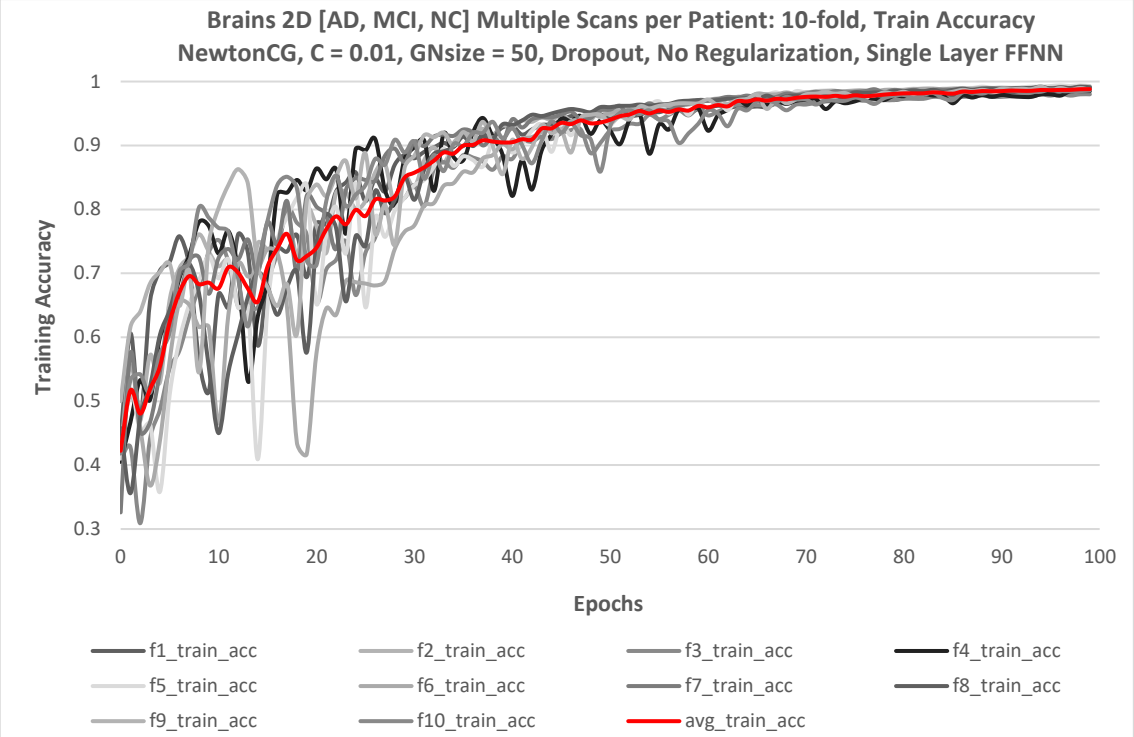


Figure O.45. Caption in the title.

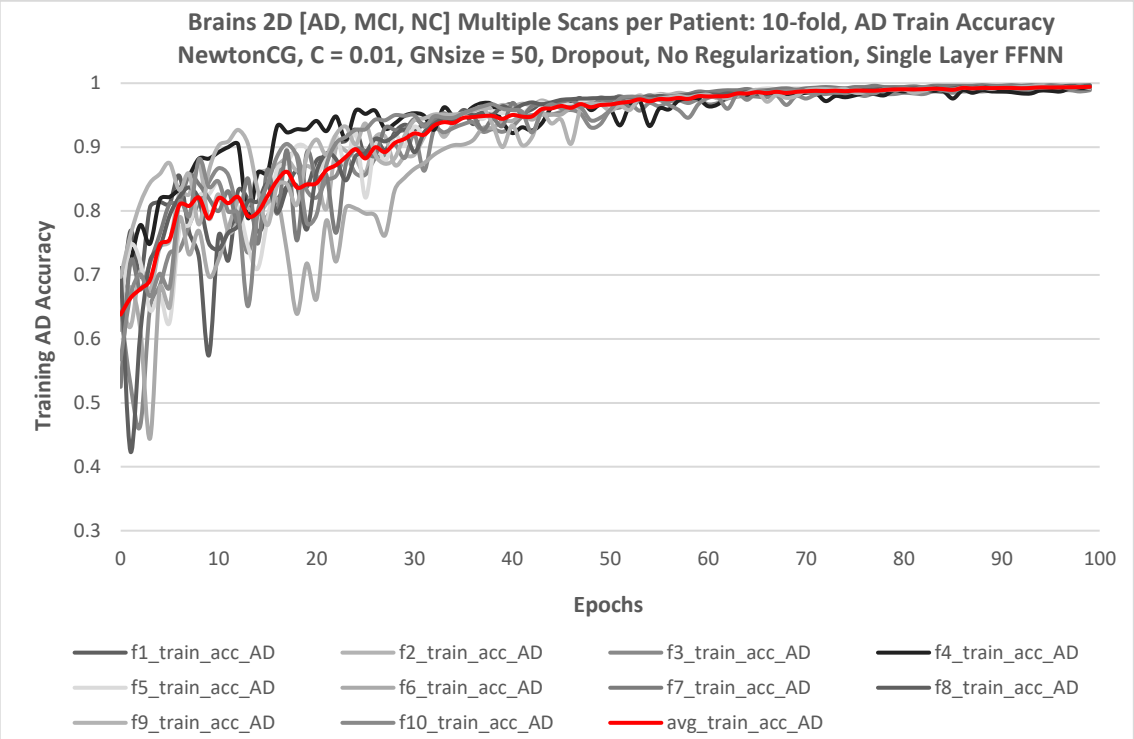


Figure O.46. Caption in the title.

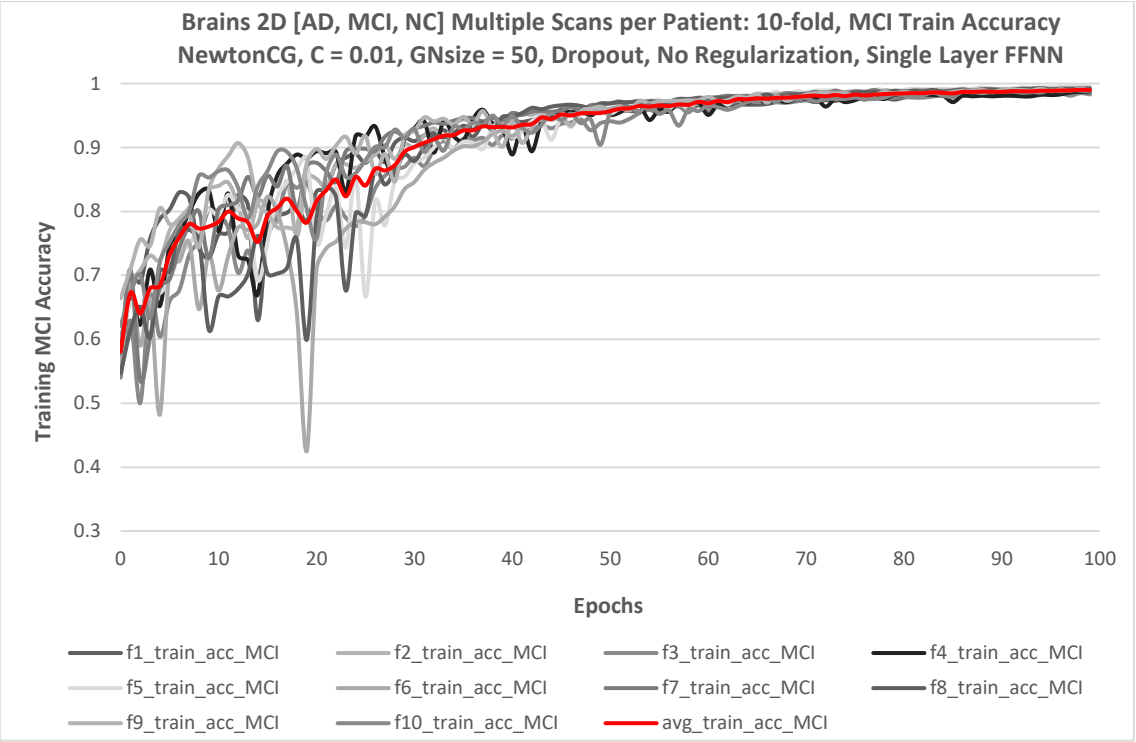


Figure O.47. Caption in the title.

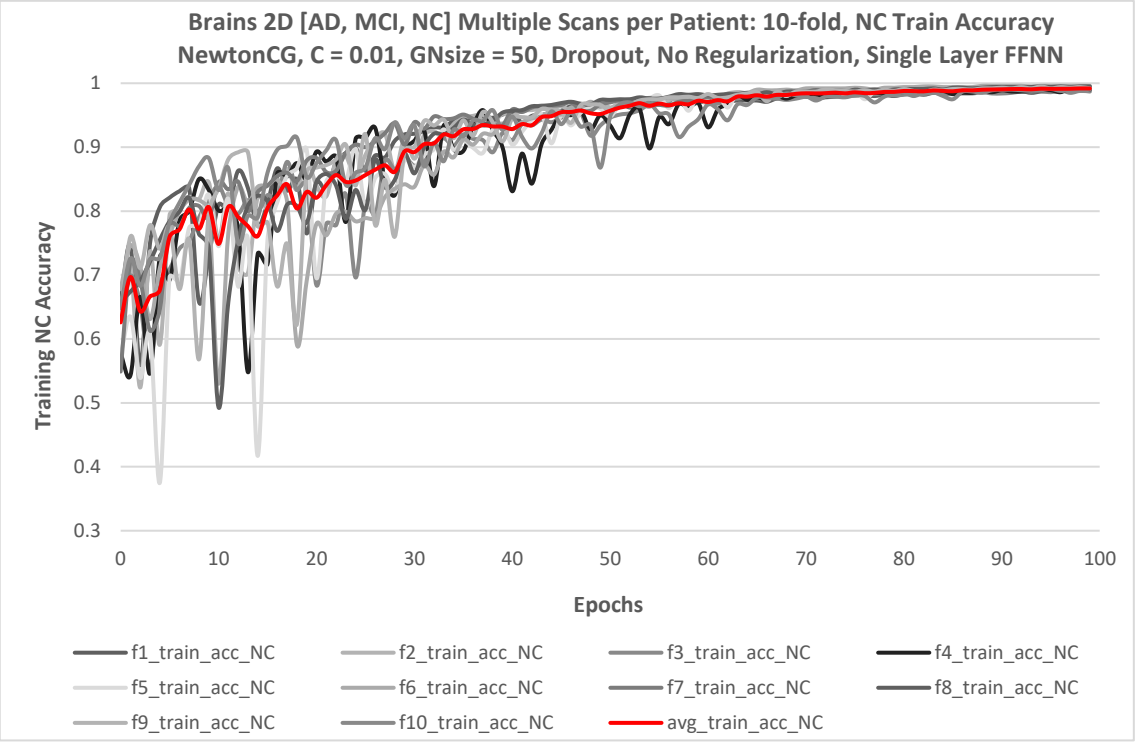


Figure O.48. Caption in the title.

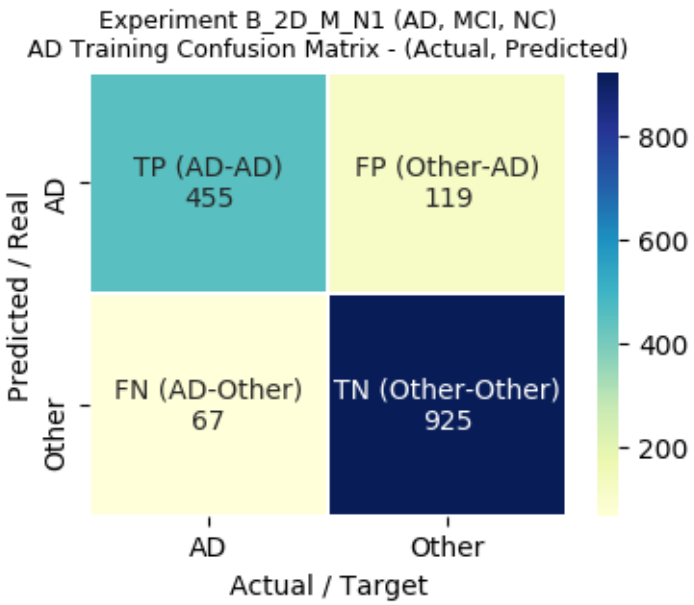


Figure O.49. Caption in the title.

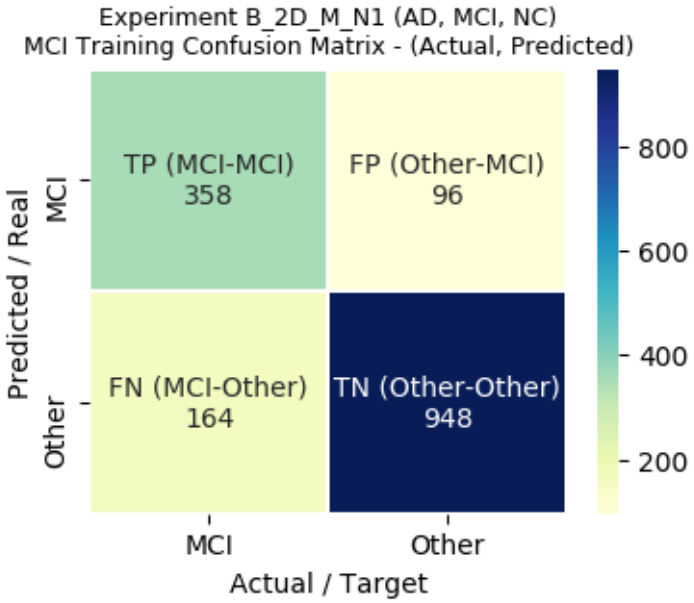


Figure O.50. Caption in the title.

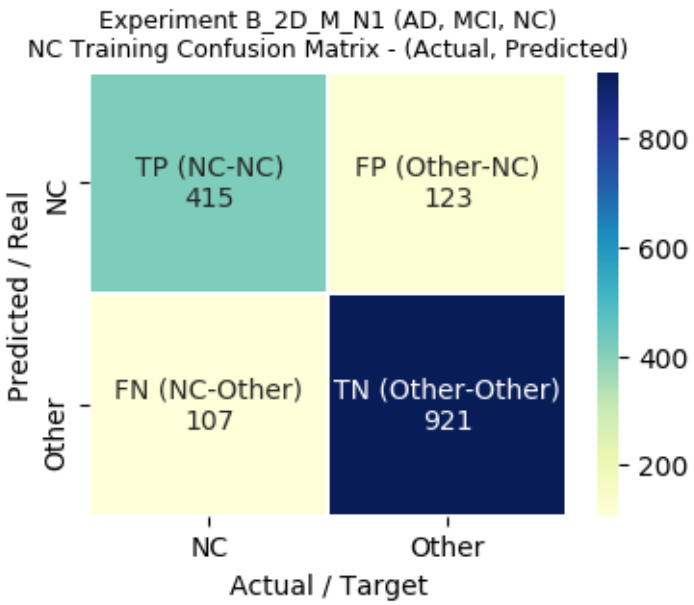


Figure O.51. Caption in the title.

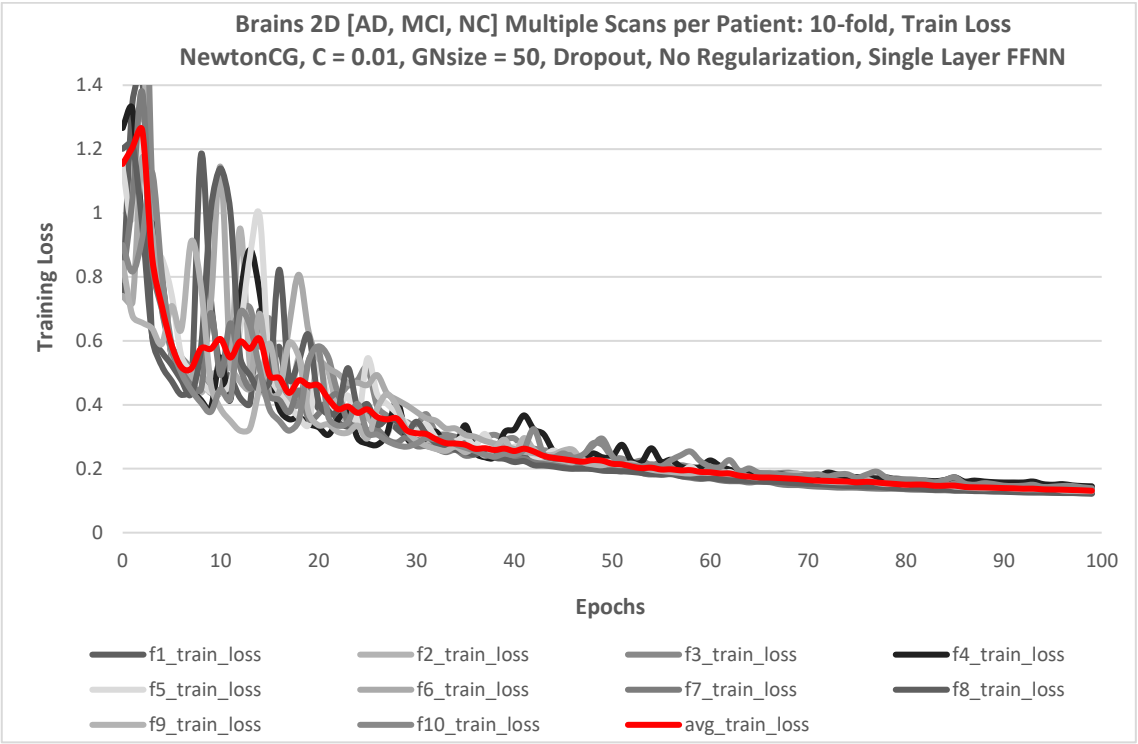


Figure O.52. Caption in the title.

O.2.1.3 Validation

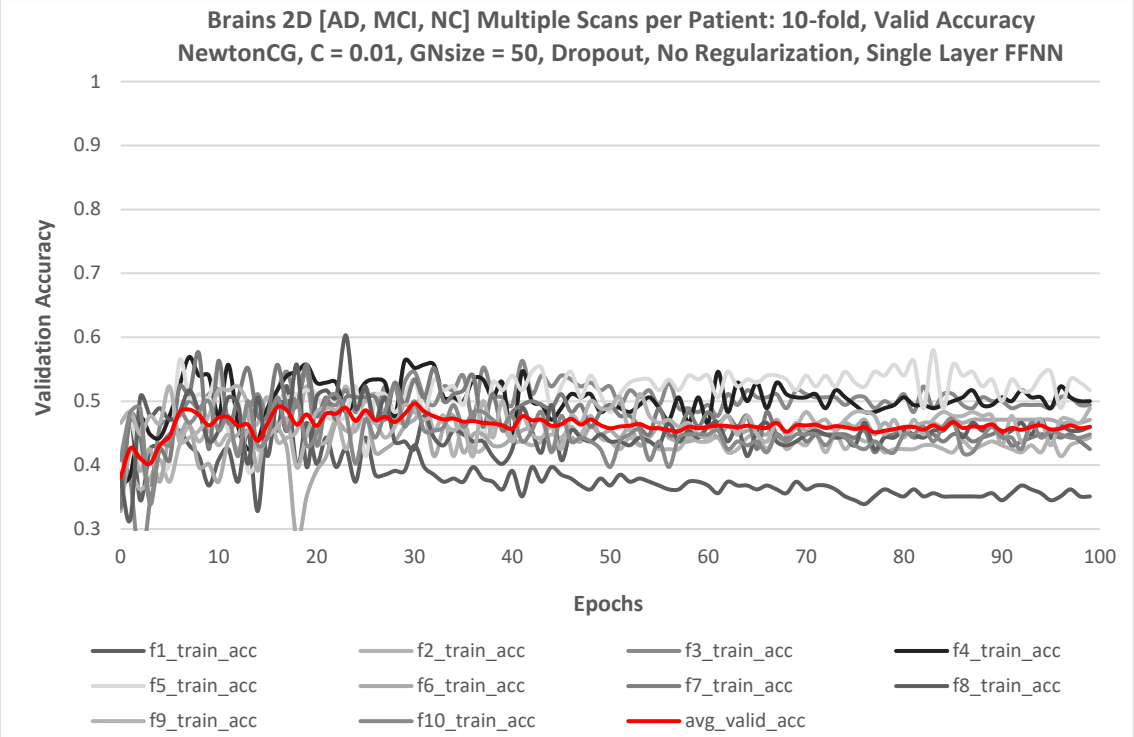


Figure O.53. Caption in the title.

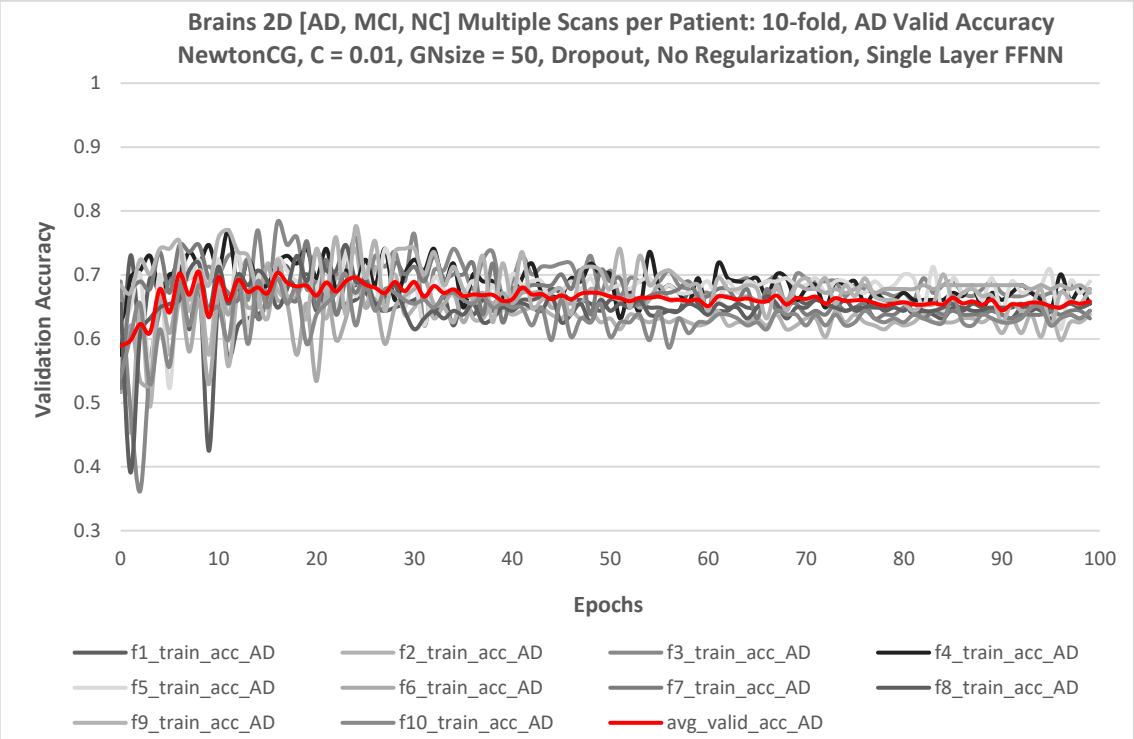


Figure O.54. Caption in the title.

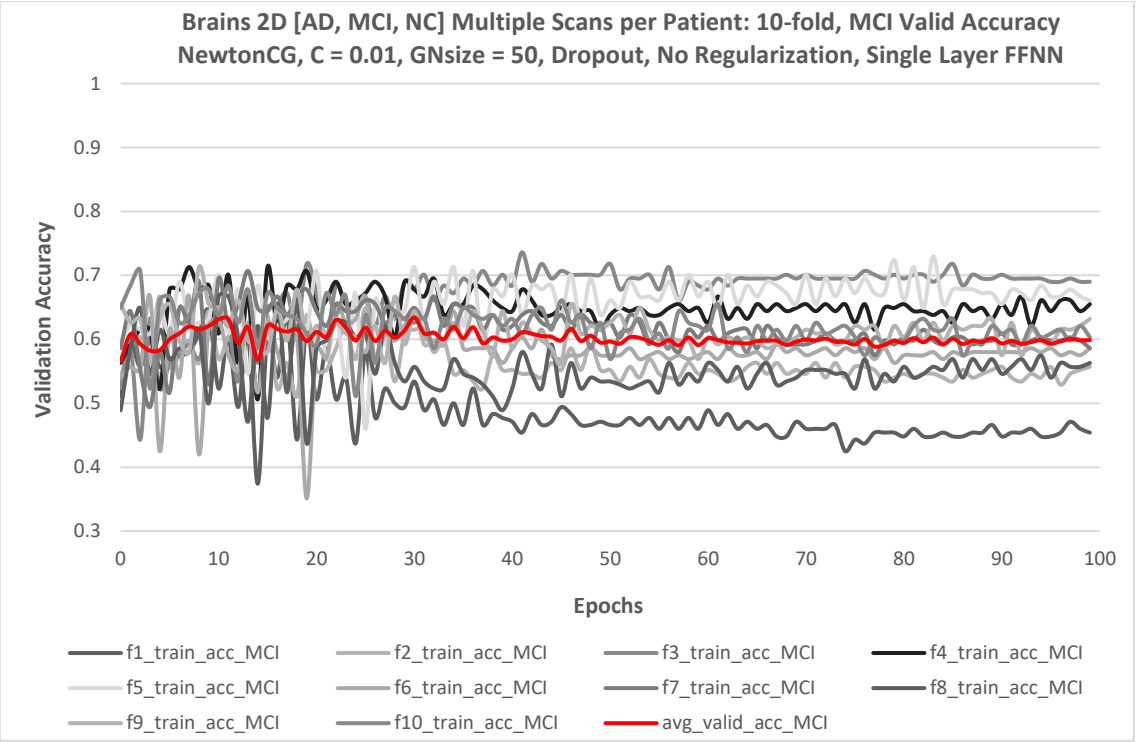


Figure O.55. Caption in the title.

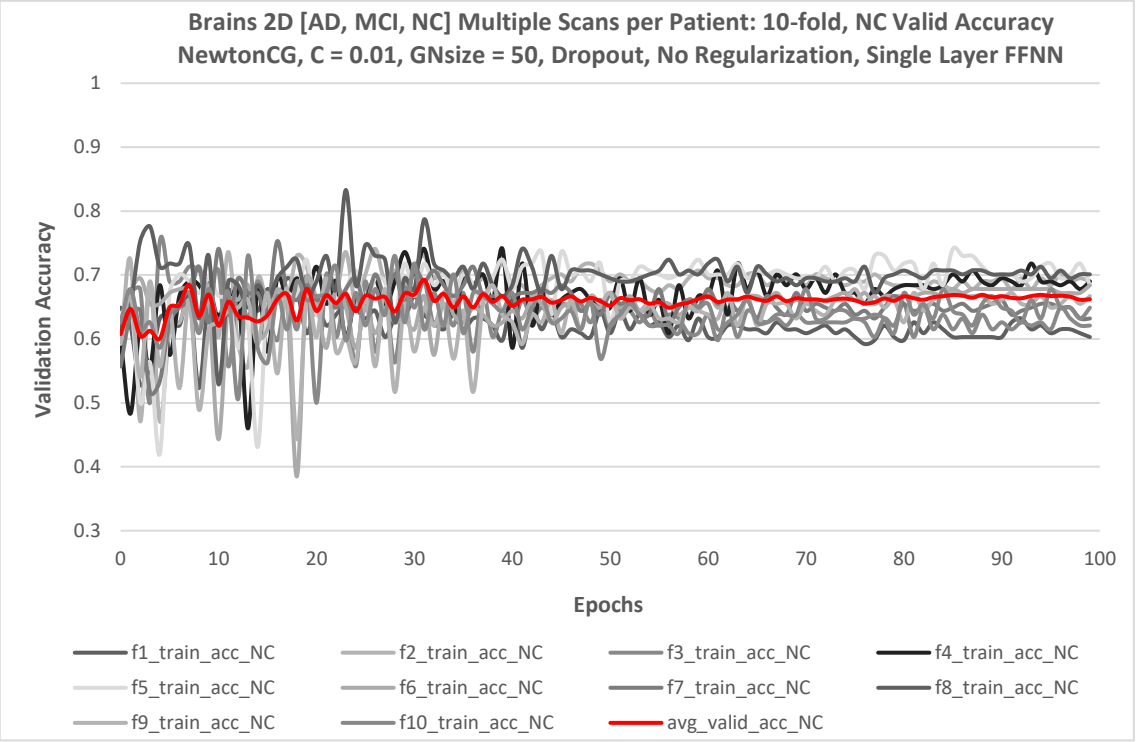


Figure O.56. Caption in the title.

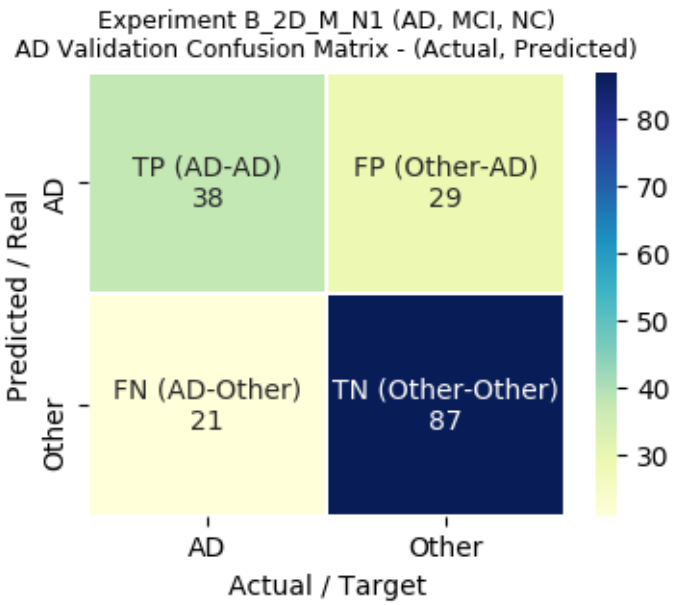


Figure O.57. Caption in the title.

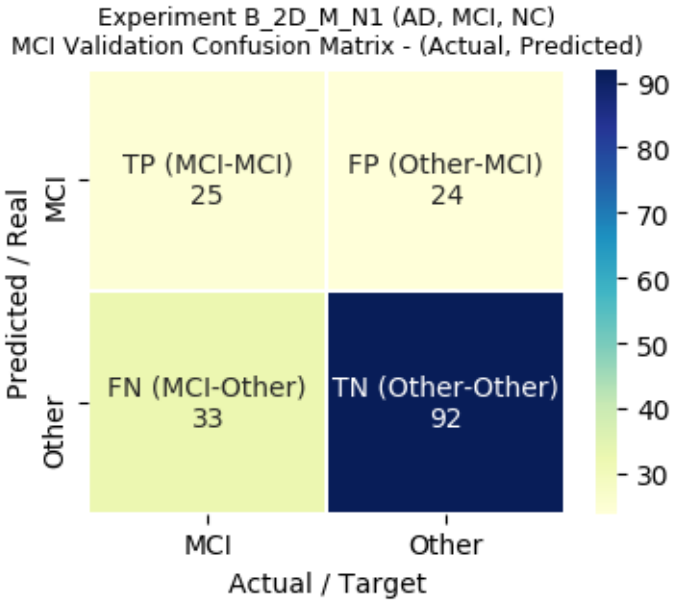


Figure O.58. Caption in the title.

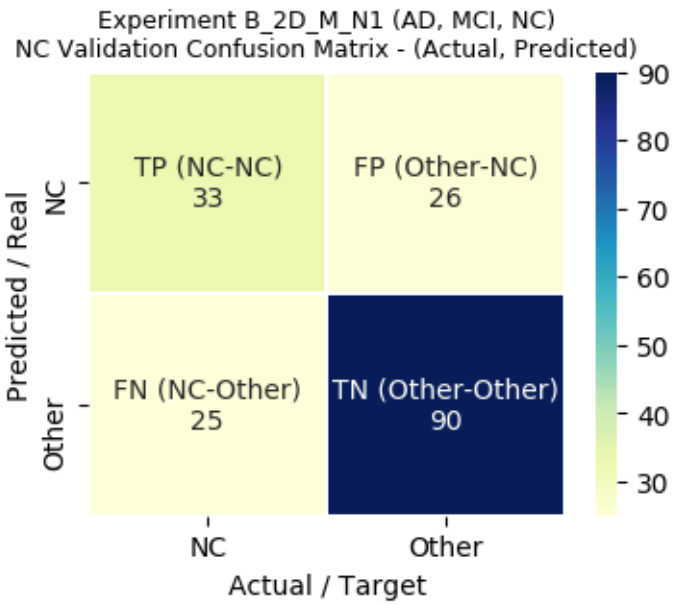


Figure O.59. Caption in the title.

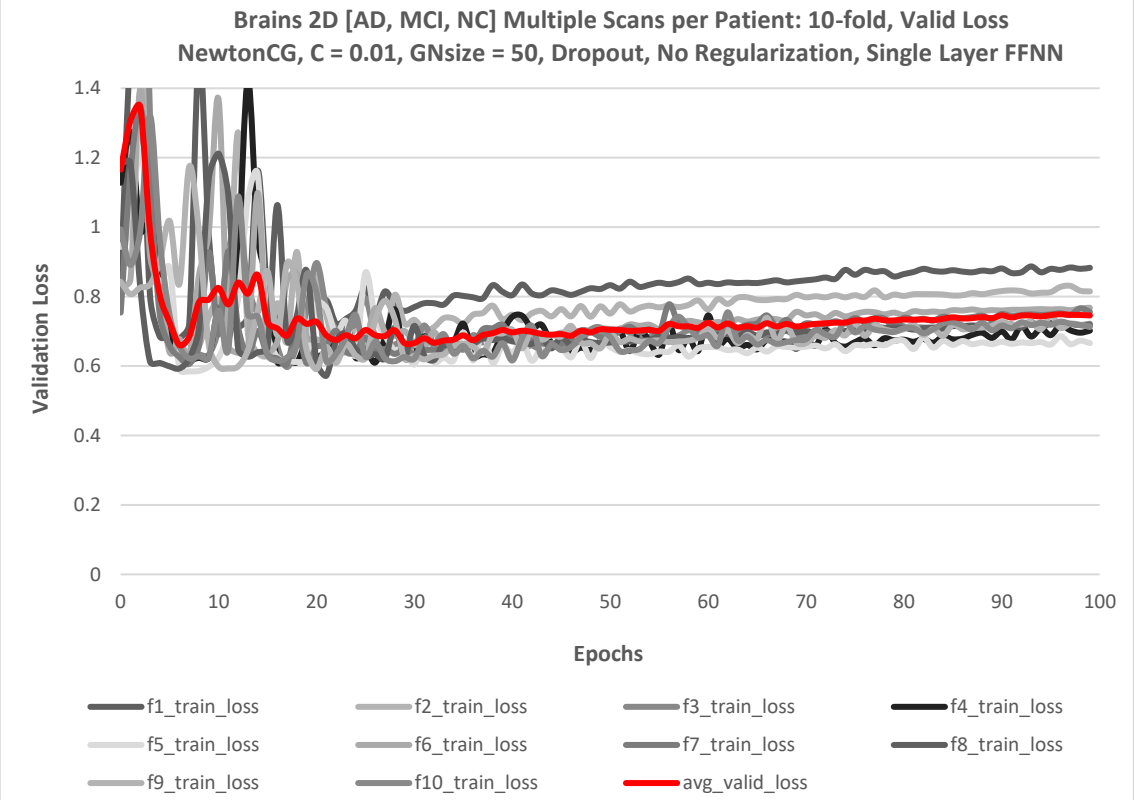


Figure O.60. Caption in the title.

O.2.1.4 Testing

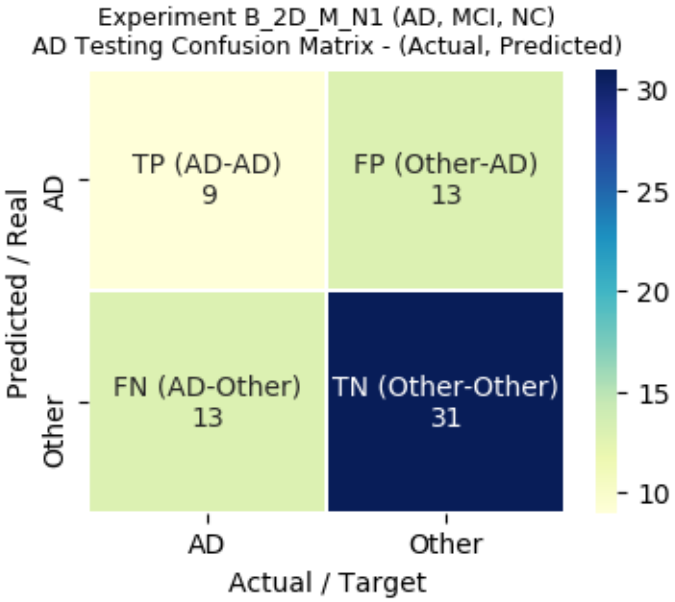


Figure O.61. Caption in the title.

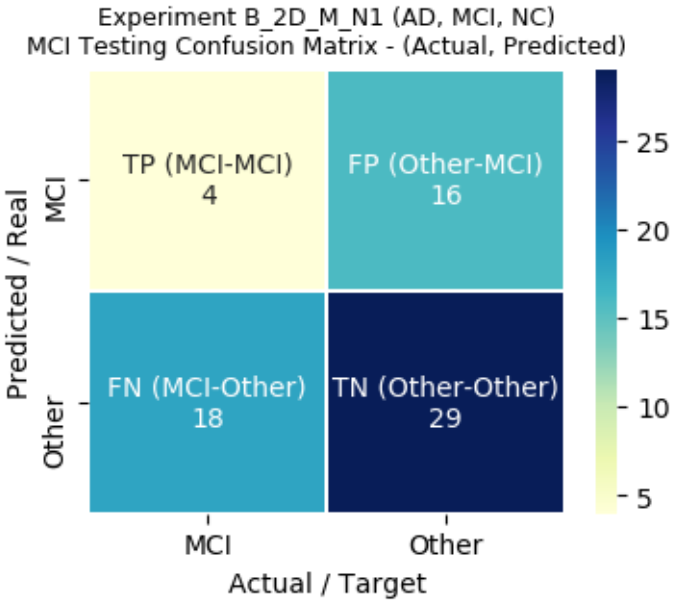


Figure O.62. Caption in the title.

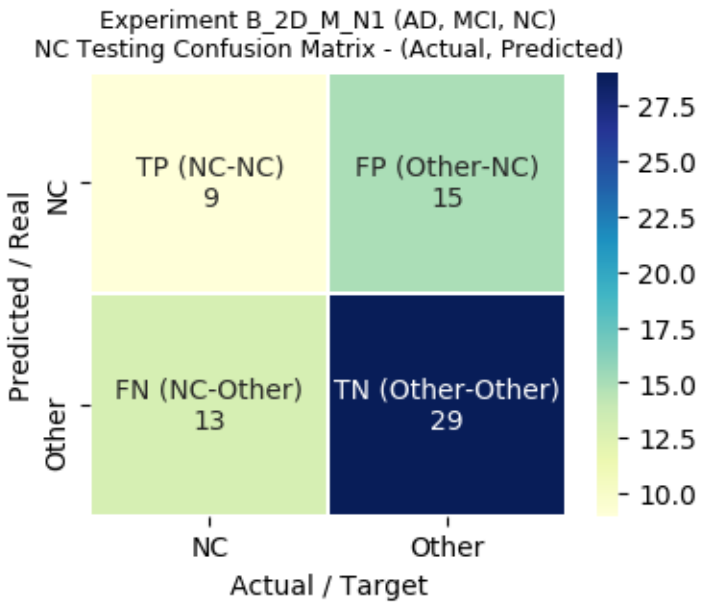


Figure O.63. Caption in the title.

Appendix P

Experiments' Hyperparameters & Performance Metrics

P.1 Experiments' Hyperparameters

Hyperparameters of the Experiments for the AD/NC and AD/MCI/NC problems																	
Experiment Name	Optimizer	Max-Pooling	Batch Normalization	CG iterations	α (alpha)	C	GN size	Max Epochs	Learning Rate	Dropout	Spatial Dropout	L1 Regularization	L2 Regularization	Feature Maps	Filters / Kernels Size	Loss Method	Network Architecture
HF_M_A1	Adam	-	-	-	7	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A2	Adam	-	-	-	7	-	-	500	0.03	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A3	Adam	-	-	-	7	-	-	500	0.003	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A4	Adam	-	-	-	7	-	-	500	0.0003	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A5	Adam	-	-	-	5	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A6	Adam	-	-	-	3	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A7	Adam	-	-	-	1	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A8	Adam	-	-	-	0.1	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A9	Adam	-	-	-	0.01	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A10	Adam	-	-	-	0.001	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_A11	Adam	-	-	-	0.1	-	-	500	0.3	-	-	-	-	-	-	-	[10, 30, 1]
HF_M_A12	Adam	-	-	-	0.1	-	-	500	0.3	-	-	-	-	-	-	-	[10, 100, 1]
HF_M_A13	Adam	-	-	-	0.1	-	-	500	0.3	-	-	-	-	-	-	-	[10, 20, 20, 1]
HF_M_S1	SGD	-	-	-	7	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_S2	SGD	-	-	-	7	-	-	500	0.03	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_S3	SGD	-	-	-	7	-	-	500	0.003	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_S4	SGD	-	-	-	7	-	-	500	0.0003	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_S5	SGD	-	-	-	0.1	-	-	500	0.3	-	-	-	-	-	-	-	[10, 8, 8, 1]
HF_M_H1	HFO	-	-	1	-	-	-	100	-	-	-	-	-	-	-	-	[10, 30, 1]
HF_M_H2	HFO	-	-	2	-	-	-	100	-	-	-	-	-	-	-	-	[10, 30, 1]
HF_M_H3	HFO	-	-	4	-	-	-	100	-	-	-	-	-	-	-	-	[10, 30, 1]
HF_M_H4	HFO	-	-	8	-	-	-	100	-	-	-	-	-	-	-	-	[10, 30, 1]
HF_M_H5	HFO	-	-	16	-	-	-	100	-	-	-	-	-	-	-	-	[10, 30, 1]

HF_M_H6	HFO	-	-	32	-	-	-	100	-	-	-	-	-	-	-	-	[10, 30, 1]
HF_M_H7	HFO	-	-	2	-	-	-	100	-	-	-	-	-	-	-	-	[10, 20, 20, 1]
HF_M_H8	HFO	-	-	2	-	-	-	100	-	-	-	-	-	-	-	-	[10, 100, 1]
HF_M_H9	HFO	-	-	2	-	-	-	100	-	-	-	-	-	-	-	-	[10, 8, 8, 1]
B_2D_M_A1	Adam	T	-	-	-	0.01	-	500	0.1	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_A2	Adam	T	-	-	-	0.01	-	500	0.01	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_A3	Adam	T	-	-	-	0.01	-	500	0.001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N1	NewtonCG	T	-	250	-	0.01	5	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N2	NewtonCG	T	-	250	-	0.1	5	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N3	NewtonCG	T	-	250	-	1	5	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N4	NewtonCG	T	-	250	-	10	5	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N5	NewtonCG	T	-	250	-	1	5	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_5L_1
B_2D_M_N6	NewtonCG	T	-	250	-	1	5	100	-	-	-	-	0.01	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N7	NewtonCG	T	-	250	-	1	5	100	-	0.3	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N8	NewtonCG	T	-	250	-	1	50	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N9	NewtonCG	T	-	250	-	1	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N10	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N11	NewtonCG	T	-	250	-	0.01	200	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N12	NewtonCG	T	-	250	-	0.01	200	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N13	NewtonCG	T	-	250	-	0.01	50	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N14	NewtonCG	-	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_4
B_2D_M_N15	NewtonCG	T	-	250	-	0.01	50	100	-	d1 - 3 = 0.3 d4 - 6 = 0.5	-	-	-	f1 - 3 = 16 f4, 5 = 32, f6 = 64	3 x 3	MSE	2D_CNN_7L_1
B_2D_M_N16	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	Cross Entropy	2D_CNN_4L_1
B_2D_M_N17	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	0.00001	0.0001	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N18	NewtonCG	T	T	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N19	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	5 x 5	MSE	2D_CNN_4L_2
B_2D_M_N20	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	7 x 7	MSE	2D_CNN_4L_3
B_2D_M_N21	NewtonCG	T	-	250	-	0.01	50	100	-	-	sd1 - 3 = 0.5	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N22	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	sd1 - 3 = 0.5	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N23	NewtonCG	T	-	250	-	0.01	50	100	-	-	sd1 - 3= 0.5	0.00001	0.0001	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N24	NewtonCG	T	-	250	-	0.01	50	100	-	-	sd1 – 3 = 0.5	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N25	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	sd1 - 3 = 0.5	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N26	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	sd1 - 3 = 0.5	0.00001	0.0001	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N27	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	sd1 - 3 = 0.5	0.00001	0.0001	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_5L_1
B_2D_M_N28	NewtonCG	T	-	250	-	1	5	500	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_7M_A1	Adam	T	-	250	-	0.01	-	500	0.001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1

B_2D_7M_N1	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_7M_N2	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2 = 0.5	-	-	-	f1 = 64, f2 = 128	3 x 3	MSE	2D_CNN_3L_1
B_2D_7M_N3	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2 = 0.5	-	-	-	f1 = 16 f2 = 32	3 x 3	MSE	2D_CNN_3L_2
B_2D_S_N1	NewtonCG	T	-	250	-	1	5	100	-	0.3	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_S_N2	NewtonCG	T	-	250	-	1	5	100	-	0.5	-	-	0.01	f1 = 8, f2 = 16 f3 = 32	3 x 3	MSE	2D_CNN_4L_5
B_2D_S_N3	NewtonCG	T	-	250	-	1	5	100	-	0.5	-	-	-	f1 = 32, f2 = 64 f3 = 128	3 x 3	MSE	2D_CNN_4L_6
B_2D_5S_N1	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
LH_3D_S_A1	Adam	T	-	250	-	0.01	-	500	0.001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_1
LH_3D_S_A2	Adam	T	-	250	-	0.01	-	500	0.0001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_1
LH_3D_S_A3	Adam	T	-	250	-	0.01	-	500	0.00001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_1
LH_3D_S_A4	Adam	-	-	250	-	0.01	-	500	0.0001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_2
LH_3D_S_N1	NewtonCG	T	-	250	-	0.01	500	100	-	-	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_1
LH_3D_S_N2	NewtonCG	-	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_2
B_3D_S_A1	Adam	T	-	250	-	0.01	-	500	0.0001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_1
B_3D_S_N1	NewtonCG	T	-	250	-	0.01	200	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_1
CB_3D_S_A1	Adam	T	-	250	-	0.01	-	500	0.0001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_1
CB_3D_S_N1	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3 x 3	MSE	3D_CNN_4L_1
CB_3D_S_N2	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2 - 4 = 0.5	-	-	-	f1, f2 = 32 f3, f4 = 64	3 x 3 x 3	MSE	3D_CNN_5L_1
B_2D_M_A1 (AD, MCI, NC)	Adam	T	-	250	-	0.01	-	500	0.001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_A2 (AD, MCI, NC)	Adam	T	-	250	-	0.01	-	500	0.0001	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1
B_2D_M_N1 (AD, MCI, NC)	NewtonCG	T	-	250	-	0.01	50	100	-	d1 = 0.3 d2, d3 = 0.5	-	-	-	f1, f2 = 32 f3 = 64	3 x 3	MSE	2D_CNN_4L_1

Table P.1. Hyperparameters and Network Configurations for all the experiments in this thesis for the AD/NC and AD/MCI/NC problem.

P.2 Experiments' Performance Metrics

The Sensitivities in Table P.2 represent for all the experiments the proportion of correctly predicted ADs, over the total number of AD; and the Specificities, the proportion of correctly predicted NCs, over the total number of NC.

Performance Metrics of the Experiments for the AD/NC and AD/MCI/NC problems

	Training				Validation				Testing					
Experiment Name	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. Accuracy	Avg. Standard Deviation of Accuracy	Avg. Sensitivity	Avg. Specificity	Avg. epoch of best Valid Acc. per fold	Best Validation Accuracy
HF_M_A1	0.83	0.04	0.72	0.87	0.85	0.04	0.76	0.94	-	-	-	-	25	0.91
HF_M_A2	0.82	0.04	0.60	0.91	0.82	0.03	0.67	0.97	-	-	-	-	22	0.86
HF_M_A3	0.80	0.09	0.69	0.85	0.82	0.05	0.71	0.92	-	-	-	-	126	0.86
HF_M_A4	0.78	0.06	0.75	0.80	0.82	0.05	0.76	0.88	-	-	-	-	271	0.91
HF_M_A5	0.82	0.04	0.74	0.85	0.86	0.05	0.78	0.93	-	-	-	-	12	0.93
HF_M_A6	0.83	0.03	0.76	0.86	0.86	0.05	0.79	0.94	-	-	-	-	107	0.91
HF_M_A7	0.88	0.05	0.83	0.90	0.89	0.05	0.80	0.97	-	-	-	-	156	0.93
HF_M_A8	0.89	0.05	0.90	0.88	0.90	0.04	0.85	0.95	-	-	-	-	191	0.96
HF_M_A9	0.90	0.04	0.93	0.88	0.89	0.05	0.82	0.96	-	-	-	-	192	0.93
HF_M_A10	0.88	0.07	0.93	0.86	0.89	0.05	0.87	0.91	-	-	-	-	180	0.93
HF_M_A11	0.88	0.07	0.87	0.89	0.90	0.05	0.83	0.96	-	-	-	-	155	0.93
HF_M_A12	0.89	0.05	0.85	0.91	0.88	0.05	0.81	0.95	-	-	-	-	241	0.93
HF_M_A13	0.90	0.06	0.91	0.90	0.89	0.04	0.83	0.96	-	-	-	-	200	0.96
HF_M_S1	0.83	0.05	0.64	0.90	0.82	0.05	0.69	0.95	-	-	-	-	14	0.89
HF_M_S2	0.83	0.06	0.62	0.92	0.82	0.06	0.65	0.98	-	-	-	-	146	0.89
HF_M_S3	0.74	0.11	0.66	0.78	0.79	0.04	0.69	0.88	-	-	-	-	163	0.84
HF_M_S4	0.74	0.09	0.62	0.78	0.74	0.05	0.63	0.85	-	-	-	-	339	0.82
HF_M_S5	0.94	0.03	0.92	0.95	0.88	0.04	0.82	0.95	-	-	-	-	272	0.93
HF_M_H1	0.91	0.04	0.80	0.95	0.86	0.05	0.75	0.98	-	-	-	-	34	0.91
HF_M_H2	0.93	0.04	0.82	0.97	0.87	0.05	0.76	0.98	-	-	-	-	23	0.93
HF_M_H3	0.87	0.07	0.77	0.91	0.86	0.05	0.75	0.97	-	-	-	-	12	0.91
HF_M_H4	0.92	0.09	0.83	0.95	0.86	0.05	0.75	0.96	-	-	-	-	28	0.91

HF_M_H5	0.93	0.06	0.83	0.97	0.86	0.05	0.75	0.97	-	-	-	-	26	0.91
HF_M_H6	0.90	0.10	0.84	0.92	0.86	0.05	0.77	0.94	-	-	-	-	16	0.91
HF_M_H7	0.93	0.04	0.82	0.97	0.86	0.05	0.75	0.97	-	-	-	-	24	0.91
HF_M_H8	0.92	0.05	0.80	0.96	0.86	0.04	0.74	0.98	-	-	-	-	24	0.89
HF_M_H9	0.94	0.04	0.85	0.97	0.86	0.05	0.74	0.98	-	-	-	-	31	0.91
B_2D_M_A1	0.53	0.07	0.47	0.59	0.56	0.07	0.50	0.61	0.51	0.05	0.44	0.58	13	0.74
B_2D_M_A2	0.87	0.06	0.86	0.87	0.79	0.07	0.80	0.78	0.60	0.06	0.58	0.63	246	0.90
B_2D_M_A3	0.91	0.07	0.92	0.90	0.79	0.08	0.77	0.82	0.64	0.05	0.53	0.75	154	0.91
B_2D_M_N1	0.88	0.04	0.87	0.89	0.78	0.08	0.75	0.80	0.61	0.03	0.52	0.71	48	0.90
B_2D_M_N2	0.87	0.04	0.87	0.87	0.76	0.08	0.74	0.77	0.62	0.06	0.51	0.72	38	0.90
B_2D_M_N3	0.91	0.04	0.91	0.90	0.77	0.07	0.77	0.78	0.62	0.03	0.55	0.69	51	0.86
B_2D_M_N4	0.89	0.05	0.87	0.91	0.75	0.10	0.72	0.79	0.63	0.04	0.53	0.72	48	0.90
B_2D_M_N5	0.89	0.04	0.88	0.90	0.75	0.08	0.74	0.77	0.61	0.04	0.52	0.69	47	0.87
B_2D_M_N6	0.88	0.05	0.88	0.88	0.76	0.09	0.78	0.75	0.60	0.04	0.51	0.69	38	0.90
B_2D_M_N7	0.90	0.05	0.90	0.91	0.76	0.08	0.74	0.77	0.62	0.05	0.53	0.71	56	0.86
B_2D_M_N8	0.90	0.09	0.91	0.90	0.78	0.09	0.77	0.79	0.60	0.04	0.52	0.69	9	0.91
B_2D_M_N9	0.93	0.05	0.92	0.94	0.76	0.08	0.74	0.78	0.62	0.05	0.53	0.71	14	0.89
B_2D_M_N10	0.84	0.07	0.83	0.84	0.80	0.05	0.82	0.79	0.61	0.06	0.54	0.67	20	0.91
B_2D_M_N11	0.89	0.06	0.91	0.87	0.78	0.07	0.82	0.74	0.60	0.06	0.55	0.64	12	0.89
B_2D_M_N12	0.90	0.04	0.90	0.89	0.78	0.07	0.78	0.77	0.64	0.05	0.52	0.75	10	0.90
B_2D_M_N13	0.84	0.06	0.82	0.86	0.78	0.08	0.77	0.79	0.60	0.04	0.53	0.67	13	0.89
B_2D_M_N14	0.91	0.12	0.93	0.89	0.71	0.10	0.66	0.76	0.62	0.06	0.56	0.67	37	0.85
B_2D_M_N15	0.77	0.04	0.76	0.78	0.79	0.08	0.78	0.79	0.63	0.07	0.62	0.66	12	0.91
B_2D_M_N16	0.79	0.03	0.79	0.80	0.77	0.08	0.78	0.78	0.64	0.05	0.63	0.67	9	0.90
B_2D_M_N17	0.85	0.04	0.86	0.85	0.78	0.08	0.80	0.78	0.61	0.03	0.61	0.63	23	0.90
B_2D_M_N18	0.82	0.05	0.79	0.84	0.79	0.08	0.78	0.80	-	-	-	-	17	0.91
B_2D_M_N19	0.82	0.04	0.80	0.83	0.81	0.06	0.81	0.80	0.63	0.03	0.52	0.73	17	0.90
B_2D_M_N20	0.83	0.10	0.84	0.81	0.79	0.06	0.81	0.76	0.62	0.04	0.54	0.70	23	0.86
B_2D_M_N21	0.84	0.04	0.87	0.83	0.79	0.08	0.82	0.76	0.60	0.05	0.59	0.62	14	0.91
B_2D_M_N22	0.85	0.03	0.84	0.87	0.78	0.08	0.78	0.81	0.63	0.04	0.61	0.67	17	0.89
B_2D_M_N23	0.79	0.09	0.79	0.81	0.78	0.09	0.79	0.79	0.60	0.07	0.58	0.62	19	0.91
B_2D_M_N24	0.81	0.04	0.79	0.84	0.76	0.07	0.74	0.80	0.62	0.06	0.61	0.65	13	0.86
B_2D_M_N25	0.78	0.04	0.77	0.81	0.75	0.08	0.73	0.78	0.63	0.05	0.62	0.65	12	0.88
B_2D_M_N26	0.79	0.06	0.80	0.80	0.77	0.07	0.77	0.78	0.61	0.05	0.60	0.64	12	0.85
B_2D_M_N27	0.80	0.07	0.81	0.80	0.75	0.06	0.77	0.75	0.59	0.06	0.59	0.60	17	0.83
B_2D_M_N28	0.93	0.06	0.93	0.93	0.77	0.09	0.76	0.78	0.60	0.05	0.50	0.69	115	0.90

B_2D_7M_A1	0.92	0.06	0.91	0.93	0.75	0.02	0.74	0.76	0.64	0.03	0.60	0.67	49	0.78
B_2D_7M_N1	0.94	0.04	0.94	0.94	0.76	0.02	0.75	0.77	0.64	0.02	0.58	0.69	32	0.78
B_2D_7M_N2	0.94	0.04	0.93	0.95	0.73	0.01	0.72	0.74	0.66	0.04	0.64	0.68	34	0.75
B_2D_7M_N3	0.96	0.02	0.96	0.97	0.73	0.02	0.71	0.75	0.66	0.02	0.63	0.69	40	0.76
B_2D_S_N1	0.85	0.12	0.84	0.86	0.75	0.07	0.74	0.77	0.68	0.12	0.68	0.68	22	0.87
B_2D_S_N2	0.90	0.10	0.90	0.89	0.73	0.06	0.73	0.74	0.72	0.09	0.71	0.72	41	0.84
B_2D_S_N3	0.85	0.09	0.86	0.85	0.77	0.05	0.78	0.75	0.70	0.14	0.78	0.62	23	0.84
B_2D_5S_N1	0.92	0.04	0.95	0.90	0.75	0.03	0.77	0.73	0.69	0.05	0.74	0.65	31	0.78
LH_3D_S_A1	0.66	0.04	0.54	0.78	0.72	0.03	0.62	0.81	0.65	0.04	0.46	0.84	37	0.75
LH_3D_S_A2	0.73	0.04	0.63	0.83	0.76	0.04	0.68	0.84	0.69	0.03	0.51	0.87	117	0.81
LH_3D_S_A3	0.73	0.09	0.76	0.69	0.72	0.03	0.75	0.68	0.62	0.07	0.56	0.68	208	0.75
LH_3D_S_A4	1.00	0.00	1.00	1.00	0.77	0.04	0.67	0.86	0.67	0.04	0.60	0.73	209	0.83
LH_3D_S_N1	0.73	0.11	0.68	0.79	0.70	0.03	0.64	0.77	0.67	0.09	0.53	0.80	4	0.73
LH_3D_S_N2	0.86	0.11	0.84	0.87	0.73	0.06	0.68	0.78	0.69	0.05	0.63	0.74	21	0.83
B_3D_S_A1	0.83	0.05	0.86	0.80	0.75	0.03	0.82	0.69	0.70	0.02	0.73	0.67	131	0.81
B_3D_S_N1	0.72	0.12	0.64	0.80	0.64	0.04	0.50	0.79	0.64	0.07	0.47	0.80	6	0.68
CB_3D_S_A1	0.98	0.03	0.96	1.00	0.73	0.02	0.69	0.77	0.65	0.06	0.60	0.71	71	0.76
CB_3D_S_N1	0.76	0.08	0.75	0.77	0.71	0.04	0.71	0.72	0.67	0.04	0.65	0.68	13	0.76
CB_3D_S_N2	0.68	0.02	0.71	0.65	0.69	0.06	0.73	0.64	0.69	0.04	0.67	0.72	9	0.74
B_2D_M_A1 (AD, MCI, NC)	0.83	-	-	-	0.54	-	-	-	0.35	-	-	-	26	-
B_2D_M_A2 (AD, MCI, NC)	0.87	-	-	-	0.53	-	-	-	0.32	-	-	-	45	-
B_2D_M_N1 (AD, MCI, NC)	0.78	-	-	-	0.55	-	-	-	0.33	-	-	-	23	-

Table P.2. Hyperparameters and Network Configurations for all the experiments in this thesis for the AD/NC and AD/MCI/NC problem.

The End.