Thesis Dissertation

# PROTEIN SECONDARY STRUCTURE PREDICTION USING CONVOLUTIONAL NEURAL NETWORKS AND HESSIAN FREE OPTIMISATION

**Panayiotis Leontiou**

## UNIVERSITY OF CYPRUS

## COMPUTER SCIENCE DEPARTMENT

May 2020

# UNIVERSITY OF CYPRUS
## COMPUTER SCIENCE DEPARTMENT


**Protein Secondary Structure Prediction Using Convolutional Neural Networks And Hessian Free Optimisation**


**Panayiotis Leontiou**

Supervisor
Dr. Chris Christodoulou


Thesis submitted in partial fulfilment of the requirements for the award of Bachelor degree in Computer Science at University of Cyprus

May 2020

# Acknowledgments

First of all, I would like to express my opinion on academic research. Academic research is of major importance, especially thesis dissertation projects like this one, because students are given the opportunity to attempt something they have not tried before, discover something new. Even if their attempt is not successful it does not really matter, what matters is to learn how to research and analyse information. The research and analytical skills could be very valuable, not only for those who are aiming for an academic career, but also for those who are pursuing an industry career.

So far we had our teachers or professors to guide us through this world full of knowledge. Now, we have to guide ourselves through this maze of information and decide what to learn, in order to improve ourselves. As Albert Einstein once said, "Once you stop learning, you start dying", and one of the best ways to keep learning is through research. Research forces you to look for information from different sources and combine them with your way of thinking to reach some conclusions. Usually some experiments are required, and whether those are for a new evolutionary algorithm or a new pancake recipe is based on our own priorities. The first one could change the entire world, while the second could change the way your friends and family see your cooking skills. What both have in common is the continuous effort for personal improvement.

At this point, I would like to thank my advisor Dr. Chris Christodoulou, not only for his support on my related research, his kindness and motivation, but also for giving me the opportunity to work on a very interesting problem, using machine learning techniques. His guidance played a major role for the completion of this thesis dissertation project. If I could go back in time and choose a different topic or advisor I would choose the same, as they both helped me learn a lot of new things, that I would have not learned otherwise.

I would also like to thank Dr. Michalis Agathokleous and the master student Andreas Dionysiou for providing me with all the necessary data files and additional implementation advice for my project, based on their own experience on this machine learning problem.

Finally, I would like to thank my family for the continuous support, no matter what decisions I take. Even if they could not help me directly with this project, they helped me indirectly with their love and exceptional cooking skills.

# Abstract

This dissertation attempts to solve the protein secondary structure prediction problem, a topic that has been concerning both Computer Science and Biology fields for decades.

Proteins are highly complex substances which are included in all living organisms. Proteins are not only of great nutritional value but are also involved in the chemical processes essential for life. The study of protein structures and functions can contribute to improved food supplements, drugs and antibiotics. In addition, the study of existing proteins could possibly help treat diseases and solve numerous biological problems, like covid-19 which, at the moment of writing, threats human life on earth.

Even though there is a lot of information about the primary structure of millions of proteins, for most of them there is no information about their secondary or tertiary structure. The reason behind that is the extremely high cost, in both money and time, of the current state-of-the-art methods and instruments that are used for protein structure determination. As a result, computational algorithms and techniques, which are cheaper and faster, are essential for predicting the secondary and tertiary structures of proteins.

In the past, there were several attempts to solve the PSSP problem with Convolutional Neural Network (CNNs) and some of them managed to achieve very good results, 81% per residue Q3 accuracy [1]. Furthermore, an attempt with a simple Feed Forward Neural Network (FFNN), trained with the Hessian Free Optimisation (HFO) algorithm, managed to reach 80.4% Q3 accuracy [2]. These results are very close to the best results reported so far for the PSSP problem (84-85%), and the combination of these techniques was the motivation behind this dissertation project.

For the purpose of this dissertation, a CNN was trained with a variation of the HFO algorithm to predict the secondary structure of proteins (PSSP), which has never been attempted before. The original HFO algorithm could not be used, because of the complex structure of CNNs, instead a variation, known as the Subsampled Hessian Newton (SHN) method [3], was used. The results of this combination, for the CB513 dataset, were an overall per residue Q3 accuracy of 78.20% for a single fold and 81.80% for 10-fold cross-validation with ensembles, random forest and external rules filtering, while the SOV score was 75.67 and 78.98, respectively. Moreover, the SHN method did not require much tuning of the hyper parameters, which made the training process much faster compared to other state-of-the-art methods. As regards the PISCES dataset, the Q3 accuracy was 79.88% for a single fold and 83.02% for 5-fold cross-validation with ensembles, random forest and external rules filtering, while the SOV score was 76.67 and 82.64, respectively.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Protein Secondary Structure Prediction problem

Proteins are highly complex substances which are present in all living organisms. There are over 30,000 unique proteins in the human body, which are responsible for performing specific functions that are essential for life. The word protein is derived from the Greek word 'πρώτειος', which means 'of the first quality', 'in the lead' or 'holding first place', and their significance was recognised in the early 19th century by chemists.

These substances consist of smaller units, called amino-acids, which are organic compounds connected to each other, forming long chains. The differences between two proteins are based on their sequence of amino acids, which determines their structure and function. The interactions between the amino acids of a protein are responsible for the fold of the protein into a specific three-dimensional structure, which, under specific conditions, remains the same. This structure determines the function of each protein.

The study of protein structures and functions can contribute to improved food supplements, drugs and antibiotics. In addition, the study of existing proteins can help treat diseases and solve numerous biological problems with the help of modern technology, which is significantly cheaper and more efficient than a few years ago.

A hierarchical approach has been established for analysing the structure of proteins more effectively and observe their different forms. These forms are separated into four distinct categories, the primary, the secondary, the tertiary and the quaternary structure. The primary structure is just a linear sequence of amino acids, that are ordered based on where they appear in the unfolded protein. The secondary structure illustrates how the local parts of a protein are organised in a two-dimensional space. The tertiary structure, which determines the specific function of a protein, has a three-dimensional shape, formed when the amino acid chain is folded. Finally, the quaternary structure is formed when multiple tertiary structures are folded together and also has three-dimensional shape.

Even though the primary structure for millions of proteins is well documented, for most of them the secondary and tertiary structures are unknown, only for a small fraction of these proteins the secondary and tertiary structures are currently available. The research and the experimental determination of the secondary and tertiary structures of a protein are not only time consuming but also an extravagant process. More specifically, in order to determine the tertiary structure of proteins, expensive and tedious methods must be used, such as X-ray crystallography and nuclear magnetic resonance (NMR). The shape of a protein is completely determined by its primary structure, about 70% of the secondary structure is affected by the interactions of the nearby amino acids of the backbone, while the other

30% is affected by more distant interactions [4]. This made prediction techniques and implementations more appealing over the experimental methods, since they have high success rates on the prediction of secondary and tertiary structure of proteins, they cost significantly less and require considerably less time than the experimental methods.

One such prediction method is ab inition prediction, which tries to predict any of the three structures based only on the primary structure and without taking into consideration any patterns. This method is divided into two distinct cases. In the first case, the folding process is simulated or minimisation of the free energy of the polypeptide is attempted, and only the primary structure of the protein is used (no other known structures). On the other hand the second case attempts to predict the structure of a protein using already known and existing protein structures [4]. This thesis is concentrated entirely on the second prediction method, and more specifically on the use of Neural Networks (NN) to predict the secondary structure of proteins. These algorithms are designed based on computational statistics and mathematical optimization techniques. These optimisation techniques help computer systems learn hidden patterns and idiosyncrasies of data, which then gives them the ability to predict and classify new data.

To sum up, because of the extreme costs in both money and time of experimental methods, it is not possible to experimentally determine the structure of all proteins. In this thesis Convolutional Neural Networks (CNN) will be used in combination with the Hessian Free Optimisation (HFO) algorithm in order to predict the secondary structure of proteins.

## 1.2 The Importance of PSSP

The solution of the PSSP problem is very important because the secondary structure is essential in order to determine the tertiary structure, which gives information about the functions of a protein. The experimental methods used for determining the tertiary structure of proteins are extremely expensive in both time and money, which led to the study of just a small portion of known proteins. As a result, the scientific community has information about the functions of just a small subset (a few thousands) of proteins, compared to the millions of proteins that exist.

Furthermore, this means that the PSSP can help identify the tertiary structure of a protein with higher accuracy and less effort. It is very important to note that the functions of a protein are based on the 20 amino acids that compose a protein, which is the main reason why the research in this field is very important. Understanding how these molecules fold around space, assemble and function can help to understand why people are getting older,

why they suffer from dangerous diseases and viruses (such as cancer), how can a cure for a disease be found (like the cure for covid-19), and other 'difficult to answer' questions.

The proteins' functions are related with their structure, which depends on both the physical and chemical parameters of these molecules. Bioinformatics is an interdisciplinary field that develops methods and software tools for understanding biological data. It combines knowledge from biology, computer science, information engineering, mathematics and statistics to analyse and interpret biological data.

## 1.3 Previous Research on PSSP

Researchers from different fields have been working on this problem for more than six decades. A wide variety of machine learning algorithms have been designed specifically for this problem and have achieved accuracy >90% [5], based on the Q3 score (Equation 1.1.). Additional structural templates from databases, which are called sequence-based structural similarity of proteins, were used in order to achieve accuracy higher than 88%. The additional information boosts significantly the learning process as well as the performance of these algorithms. The three-state accuracy for machine learning algorithms, that are not relying on the structural templates, is currently around 82-85%, which is good for such a complex problem. However, considering the theoretical limit of the three-state prediction which is around 88-90%, there is still room for improvement.

Figure 1.1 shows the number of publications per year for the PSSP problem as well as the cumulative number of publications, between 1973 and 2015. According to the graph the cumulative number of publications for the PSSP problem increased significantly between 1973 and 2015. More specifically, between 1973 and 1989 there were less than 5 publications for the PSSP problem per year. In 1990, the PSSP problem started to become more popular and the number of publications increased considerably to 8, while the cumulative number of publications was around 50. During the next two decades, the PSSP problem gained much popularity, probably because in that period there were some major breakthroughs, which helped to increase the three-state accuracy considerably. The popularity of PSSP dropped substantially in 2010 and for the following 5 years the interest for this problem was relatively moderate. A small selection of PSSP publications are mentioned below.

Feedforward Fully Connected Neural Network (FFNN) [7]: A fully connected Neural Network with local input window (usually of 13 amino acids with orthogonal encoding) and just one hidden layer. The output of the network was one of the three categories

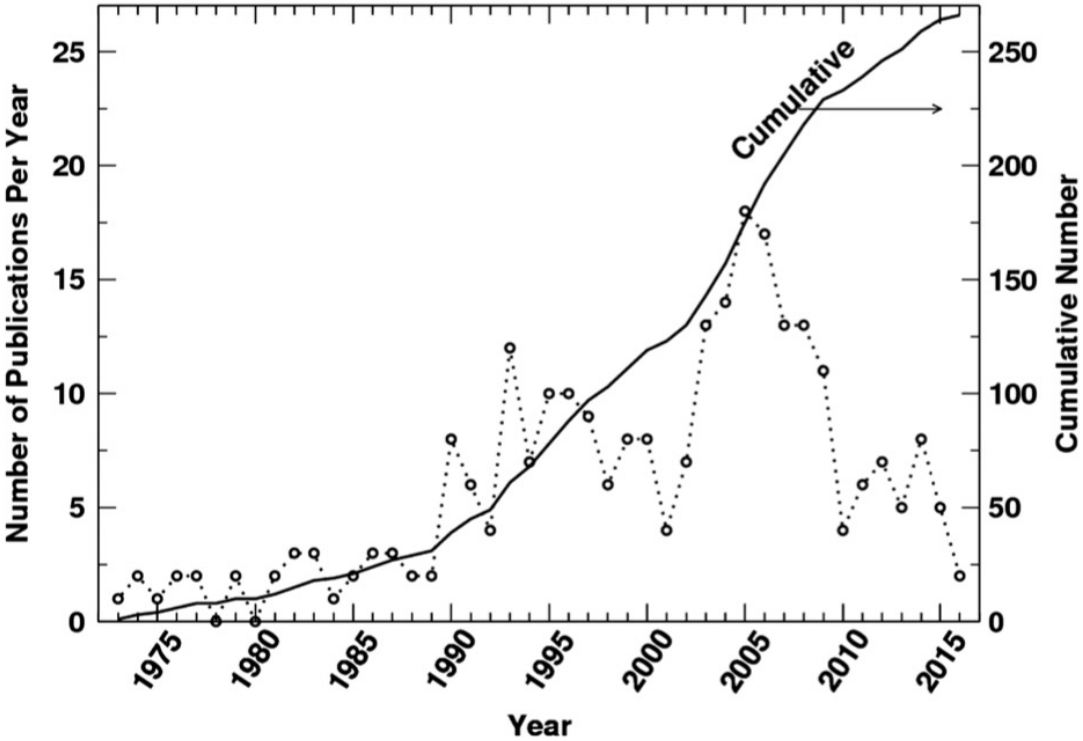



Figure 1.1: Number of publications for PSSP per year [6].

of the secondary structure of proteins (helix, pleated or other) based on the amino acid located in the centre of the input window. A secondary network was also used in this implementation to improve the output of the previous network. This method had issues with overfitting.

PHD: predicting 1D protein structure by profile based neural networks [8]: The structure of the network was the same with the Feed Forward Fully Connected Neural Network of Qian and Sejnowski [7], with the addition of techniques that deal with the overfitting problem. Two methods were used to counter overfitting, early stopping (terminating the training process before it starts to overfit) and ensemble average (training different networks at the same time with different data and learning methods). Furthermore, the multiple alignment technique was used in the input data, to take advantage of evolutionary information.

Gene-finding Programs (NNSSP) [9]: This Neural Network uses the 'nearest neighbour' method to group the sequences of amino acids based on their similarities and compare them with other sequences, that their secondary structure is known. Following that, the network tries to predict the secondary structure of other proteins that their secondary structure is not known.

Discrimination of Secondary structure Class (DSC) [10]: This algorithm groups the output data of the network and by using simple linear static methods attempts to predict the secondary structure of proteins.

PREDATOR [11]: It was implemented in a Neural Network which takes as input a sequence of amino acids and tries to predict the secondary structure based on possible hydrogen bonds that may exist in the output sequence.

Consensus [12]: In this method a Neural Network was used that took as input the multiple alignment with additional information about the protein (rather than just a simple sequence of amino acids). This Network attempts to locate similarities between the input sequence with other amino-acid sequences (similarities in genetic code, evolutionary history and common biological functions) in order to predict the secondary structure.

Bidirectional Recurrent Neural Network (BRNN) - Backpropagation ([13], [14]): This algorithm uses a Neural Network that takes as input a window with a sequence of amino acids and attempts to predict the secondary structure of the amino acid located in the centre of the input window, based on the amino-acids that precede and follow it in the input chain using bidirectional recursion. It it important to note that this algorithm had some of the best results in the PSSP problem at the time it was conceived, with 76% success rate.

Logical Analysis of Data (LAD) [15]: This method, which uses a machine learning algorithm, was implemented to identify properties of amino acids, and therefore, additional information about the homogeneity of proteins, which could help the prediction of the secondary structure of proteins. According to this method, the most important property that affects the helix class is molecular weights, for the pleated class is the mean ambient hydrophobicity, while for the other forms is the polarity.

Multiagent Secondary Structure Predictor with Postprocessing (MASSP3) [16]: This implementation attempts to solve this problem by using two distinct sections. The first section is based on a hybrid structure, which combines genetic and neural techniques, while the second section consists of a Multilayer Perceptron (MLP), which takes as input the output of the first section. The results of this method were fairly good.

Two-Stage method [17]: This approach uses two stages, the first identifies instabilities in how the protein folds into space and attempts to classify the different parts of the protein, while the second splits the proteins into sequences (3 to 7 residues) and tries to predict the secondary structure of these sequences.

Evolutionary method for learning HMM structure [18]: In this research genetic algorithms were used, which can dynamically change the parameters of a Hidden Markov Model (HMM) (since the construction of a HMM is very complicated) and build it dynamically, so that it can predict the secondary structure of the input sequences.

Cascade Bidirectional Recurrent Neural Network (BRNN) [19]: This implementation focused on the long range dependencies between the input data, which plays a major role in the folding of a protein and the correlation between adjacent secondary structures. In this article, the authors refer to the correlation of the secondary structure of an amino acid as regards to secondary structure of the adjacent amino acids. Two BRNNs are used, with the second taking its input from the output of the first BRNN. This method, although, it had relatively good results, could not outperform previous approaches.

Protein Secondary Structure Prediction Using Deep Convolutional Neural Fields [20]: This approach used a Deep Convolutional Neural Fields (DeepCNF), which is an extension of Deep Learning to Conditional Neural Field (CNF) (a combination of Conditional Random Fields (CRF) and shallow neural networks). The DeepCNF is much more powerful than the CNF, since it can model both the complex sequence-structure relationship (from a deep hierarchical architecture) and the interdependence between adjacent secondary structure tags. Based on the experimental results, the DeepCNF can reach prediction accuracy of about 84%, using the protein datasets CASP and CAMEO, surpassing existing methods of predicting the secondary structure of proteins. The DeepCNF networks can also be used to predict other properties of proteins, such as contact number, solvent accessibility and disorder regions.

Protein Secondary Structure Prediction with the use of Convolutional Neural Networks for Image Object Recognition [21]: The purpose of this research was to identify how Convolutional Neural Networks (CNN) can help in solving the PSSP problem. These type of networks take advantage of the spatial structure of the input data, which seems very promising. Furthermore, they manage input data of problems with sequences or problems that use the parameter of space, better, like image processing. This method could only reach an accuracy of about 40%, because there were problems in the representation of input data in the CNN, which prevented the network from learning effectively.

Capturing non-local interactions by long short-term memory bidirectional recurrent neural networks for improving prediction of protein secondary structure, backbone angles, contact numbers and solvent accessibility [22]: Unlike other methods that try to capture short to intermediate interactions between amino acid residues, this approach used Long Short-Term Memory (LSTM) Bidirectional Recurrent Neural Networks (BRNNs) to cap-

ture long range interactions. This method reported some of the best results so far with approximately 84% Q3 accuracy.

Protein Secondary Structure Prediction Using Bidirectional Recurrent Neural Networks (BRNN) and Hessian Free Optimisation (HFO) ([23], [2]): This dissertation was undertaken by a past Computer Science student of University of Cyprus in the context of his diplomatic research. This dissertation showed that simple Feed Forward Neural Networks (FFNNs) can be trained with the powerful second-order learning algorithm, Hessian Free Optimisation (HFO), to predict the secondary structure of proteins. This approach (FFNN with HFO) had very good results as regards the training time of the network and (Q3) accuracy, which was about 80.4% (using the PISCES dataset). The HFO does not require much tuning of the hyper parameters, which makes training much faster than other state of the art methods. The use of HFO seems very promising since it reduces the training time of the network and at the same time offers very good results.

Prediction of Secondary Structure of Proteins using Gabor filters and Support Vector Machines ([24], [1]): This dissertation was conducted by a past Computer Science student of University of Cyprus during his diplomatic research. This thesis project, was focused on the use of Convolutional Neural Network (CNN) with Gabor Filters and Support Vector Machines (SVMs) for filtering. The combination of a CNN with SVMs had very good results with about 81% (Q3) accuracy for the PSSP problem (using the PISCES dataset). A technique was also used to convert the primary structure of proteins from one dimension into two dimensions, since the CNN needs two dimensional input data to be trained.

Sixty-five years of the long march in protein secondary structure prediction: the final stretch? [6]: This paper focused on some of the state-of-the-art methods that are used to predict the secondary structure of proteins and compared them, using the same independent test sets. The reported results ranged from 77.1% to 82.3%. The best results (82.3% Q3 accuracy) were achieved by the DeepCNF [20]. In addition, this paper mentioned alternatives to discrete three-state secondary structure prediction (with eight-state prediction) and noted that the theoretical limit of secondary structure prediction is around 88%. This limit is very close to the best results reported so far (84%), which means that it is a matter of time for the PSSP problem to reach a plateau (where there will be no further improvements in Q3 accuracy).

MUFold-SS [25]: In this research a new deep learning architecture was suggested for the PSSP problem, the Deep inception-inside-inception (Deep3I) network. This network was implemented as a software tool, named MUFOLD-SS, which takes as input a specifically designed array of data, based on the primary structure of the proteins. This array includes

information for each amino acid and general information about the protein. The structure of MUFOLD-SS allows the extraction of information related to local and general interactions, between the amino acids, which made the predictions more accurate. This tool has outperformed other techniques used on the PSSP problem, with an accuracy of approximately 86.49%.

Table 1.1 shows the Q3 accuracy of the aforementioned methods, used on the PSSP problem, in chronological order.

| NO. | METHOD | Q3 ACCURACY (%) |
|---|---|---|
| 1 | Feedforward Fully Connected NN (Qian και Sejnowski, 1988) | 63.30 |
| 2 | PHD (Rost, 2001; Rost και Sander, 1993) | 71.40 |
| 3 | NNSSP (Salamov και Soloveyev, 1997) | 68.41 |
| 4 | DSC (King και Sternberg, 1996) | 71.95 |
| 5 | PREDATOR (Frishman και Argos, 1997) | 68.60 |
| 6 | Consensus (Cuff και Barton, 1999) | 72.70 |
| 7 | BRNN – Backpropagation (Baldi et al., 1999 ) | 76.00 |
| 8 | LAD (Jacek et al., 2005) | 70.60 |
| 9 | MASSP3 (Giuliano et al., 2005) | 76.10 |
| 10 | Evolutionary method for learning HMM structure (Won et al., 2007) | 65.00 |
| 11 | Two-Stage method (Fadime et al., 2007) | 74.10 |
| 12 | Cascade BRNN (Jinmiao και Narendra, 2007) | 74.38 |
| 13 | Deep Convolutional Neural Fields (Wang et al., 2016) | 83.00 |
| 14 | Convolutional Neural Networks (Pavlidis, 2016) | 40.00 |
| 15 | LSTM-BRNN (Heffernan et al., 2017) | 84.00 |
| 16 | MUFold-SS (Fang et al., 2018) | 86.49 |
| 17 | Feed Forward NN with HFO (Charalambous et al., 2020) | 80.40 |
| 18 | Convolutional Neural Network with SVM filtering (Dionysiou et al., 2020) | 81.00 |

Table 1.1: Methods used for PSSP in chronological order.

# Chapter 2

# Background

## 2.1 Biology Background

### 2.1.1 The Biological Role of Proteins

Proteins are large macromolecules or biomolecules, that perform a variety of functions within organisms. Some of these functions are deoxyribonucleic acid (DNA) replication, responding to stimuli, providing structure to cells and organisms, catalysing metabolic reactions, and transporting molecules from one location to another. Proteins consist of hundreds or even thousands of smaller units, called amino acids, which are organic compounds that contain amine (NH2) and carboxyl (COOH) functional groups.

The consumption of food, which contains proteins, is one of the main sources of proteins for the human body. The digestive system breaks down the consumed food into amino acids, which enter the blood stream. In order to perform a variety of functions, the cells of the human body gather amino acids from the blood stream to create all the essential proteins. If there is a shortage of amino acids in the blood stream, probably because of a poor diet with less proteins, the immune system will become weak, causing dizziness, exhaustion or even serious diseases. That happens because in order to create the necessary proteins for the human body, the cells need enough amino acids, otherwise they will not be able to support the needs of the entire human body.

In order to aid in the development of food supplements, drugs and antibiotics, it is mandatory to first understand the base structure and function of each protein. Research or studies on existing proteins could help solve numerous biological problems and treat diseases. This is considerably easier nowadays, with the help of the current technology, which is faster and computationally stronger than ten years ago.

The most important functions of proteins are displayed in table 2.1 and these reveal the significance of proteins, for all living organisms.

### 2.1.2 Amino Acids

Amino acids are organic compounds which contain amine (NH2) and carboxyl (COOH) functional groups. Each amino acid has its specific side chain (R group), which is an atom or group of atoms that replace one or more hydrogen atoms on the parent chain of a hydrocarbon, which turns into a moiety of the resultant new molecule (Figure 2.1). The main elements of an amino acid are carbon (C), hydrogen (H), oxygen (O) and nitrogen (N), however, other elements can also be found in the side chains of some amino acids.

| Type | Function Description | Example |
|---|---|---|
| **Defense** | Defense proteins help organisms fight infection, heal damaged tissue, and evade predators. | *Antibodies* |
| **Enzyme** | Enzymes build and break down molecules. They are critical for growth, digestion, and many other processes in the cell. Without enzymes, chemical reactions would happen too slowly to sustain life. | *Lactase* |
| **Messenger** | Messenger proteins transmit signals to coordinate biological processes between different cells, tissues, and organs. | *Growth Hormone* |
| **Motor** | Motor proteins keep cells moving and changing shape. They also transport components around, inside cells. | *Dynein, Kinesin* |
| **Regulatory** | Regulatory proteins bind DNA to turn genes on and off. | *Androgen, Estrogen* |
| **Sensory** | Sensory proteins help humans learn about their environment. They help them detect light, sound, touch, smell, taste, pain, and heat. | *Opsin* |
| **Signaling** | Signaling proteins allow cells to communicate with each other. | *Insulin* |
| **Storage** | Storage proteins store nutrients and energy-rich molecules for later use. | *Gluten* |
| **Structural** | Structural proteins strengthen cells, tissues, organs, and more. | *Collagen* |
| **Transport** | Transport proteins move molecules and nutrients around the body, in and out of cells. | *Hemoglobin* |

Table 2.1: Types of proteins and their function [26].

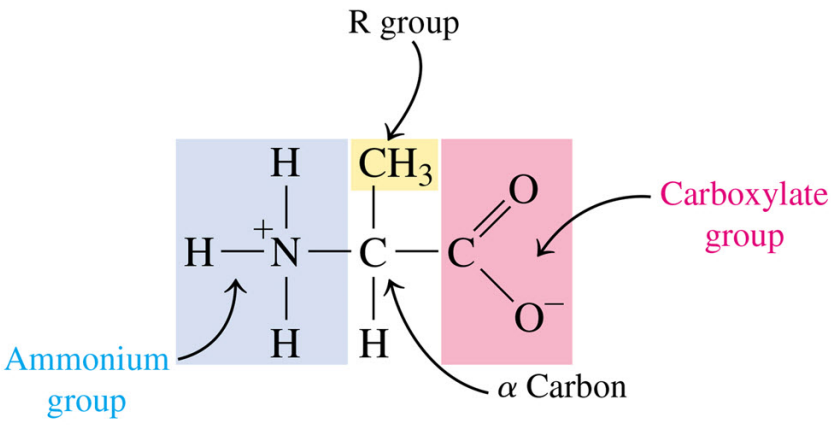



Figure 2.1: The structure of amino acids [27].

Even though there are about 500 known amino acids, only 20 of them appear in genetic code and are considered as the standard amino acids (Figure 2.2). Amino acids can be classified in many different ways, according to the core structural functional groups' locations (alpha ($\alpha$), beta ($\beta$), gamma ($\gamma$), delta ($\delta$)), based on the polarity, pH level or on the side chain group type. Amino acids also participate in a number of other processes, such as neurotransmitter transport and biosynthesis. Short chains of amino acids (30 or less) linked by peptide bonds form peptides, and long, continuous, and unbranched peptide chains form polypeptides. Proteins consist of one or more polypeptides arranged in a biologically functional way.

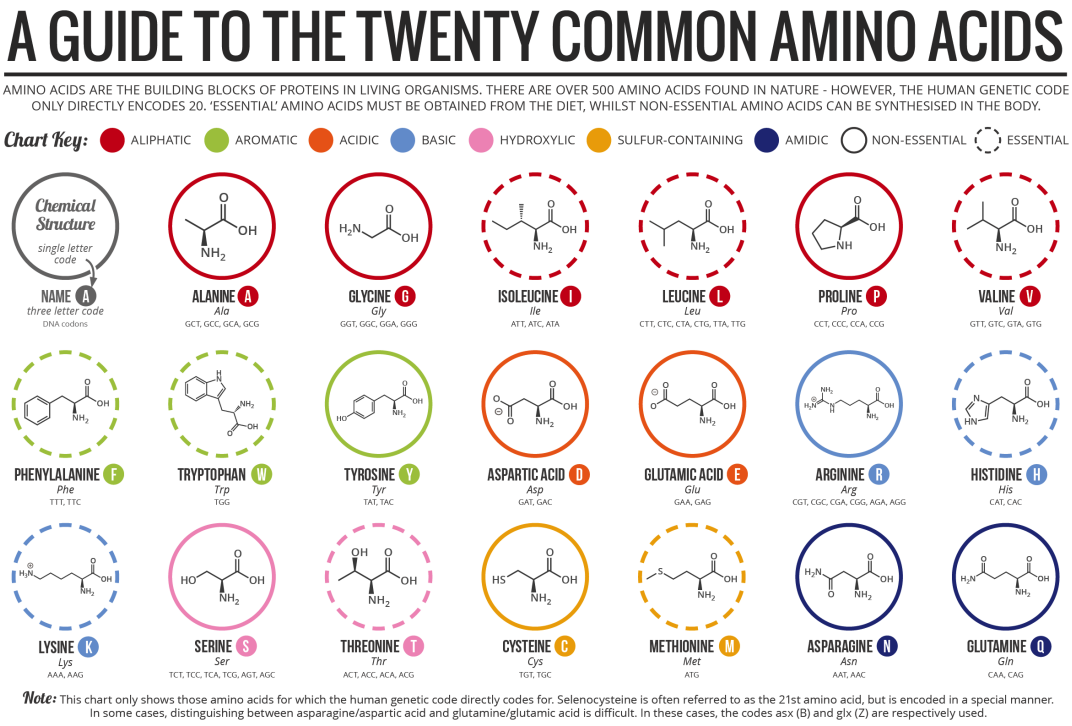



Figure 2.2: The 20 standard amino acids [28].

The process in which chains of amino acids are linked together is called condensation reaction (Figure 2.3). During this reaction, as the amino group of one amino acid joins the carboxyl group of a neighbouring amino acid, a water molecule is extracted, what is left of each amino acid is called amino acid residue.

Each amino acid can be represented by one or three characters from the English alphabet, so it is possible to represent a sequence of amino acids using a sequence of characters. Any change in this sequence, no matter how small it is, can lead to a completely different protein, which will have its own properties and functionalities.

The proteins in an organism are assembled based on its genes, also know as the DNA.

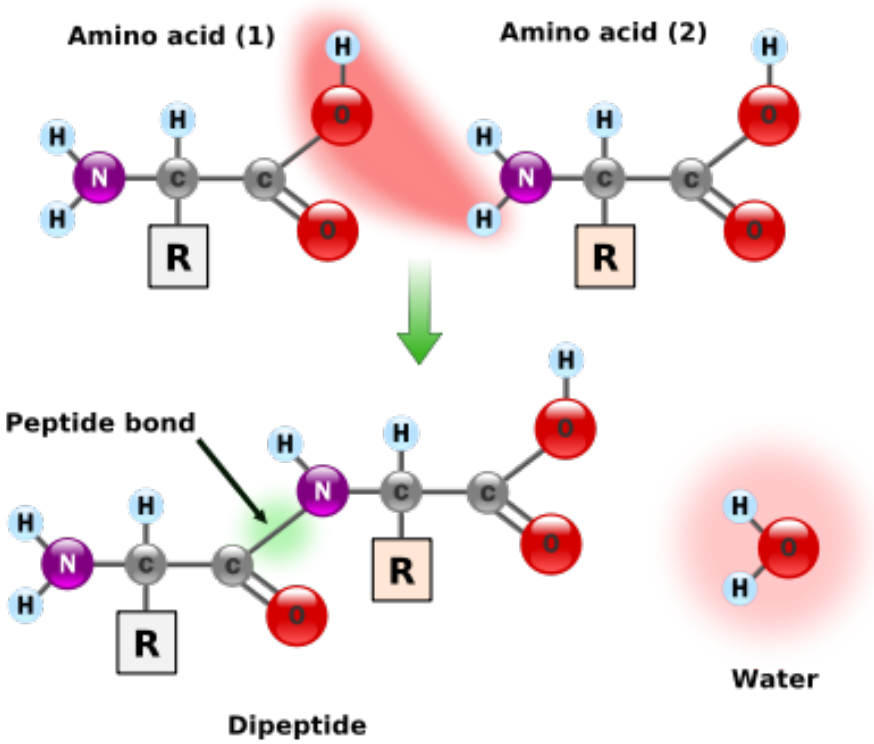



Figure 2.3: An example for condensation reaction [29].

In particular, the nucleotide sequence of a gene, which encodes a protein, specifies the unique amino acid sequence of that protein. For instance, there are around 30,000 genes in the human genome, and each one encodes one unique protein. According to The Central Dogma of Molecular Biology, the 'DNA makes RNA and RNA makes protein' (Figure 2.4). The first stage, 'DNA makes RNA', is called transcription, while the second stage, 'RNA makes protein', is called translation.

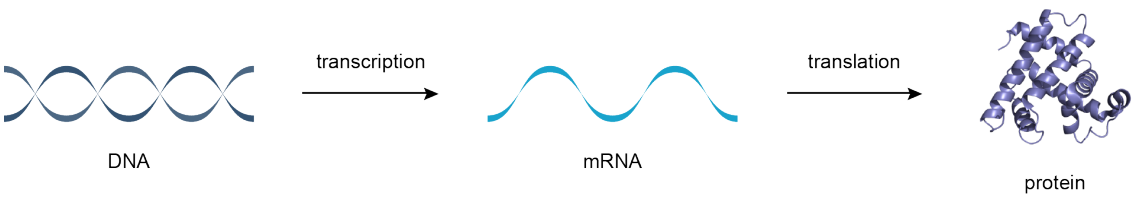



Figure 2.4: The Central Dogma of Molecular Biology: DNA makes RNA makes protein [30].

A sequence of three adjacent nucleotides composing the genetic code is called codon and designates an amino acid. There are four (4) unique nucleotides (adenine - A, uracil - U, guanine - G, and cytosine - C), which means that the maximum number of triplets that can be formed is sixty four ($4^3 = 64$). However, only twenty (20) amino acids can be encoded naturally. This means that some codons do not encode any amino acids or that

some amino acids can be described by multiple codons. Codons that do not encode any amino acids are called stop codons, because they are used as a termination signal in the translation process, signalling the release of the translated polypeptide or protein. Figure 2.5 shows an example of the translation stage, from DNA to protein, while figure 2.6 presents the table of codons, with the amino acid or the stop signal they encode.

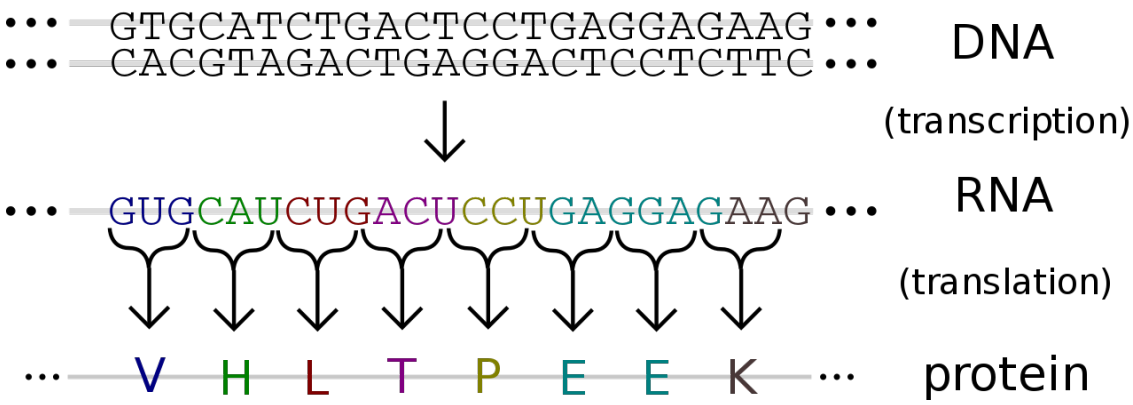



Figure 2.5: Example of the central dogma, which illustrates the first few amino acids for the alpha subunit of hemoglobin [23].

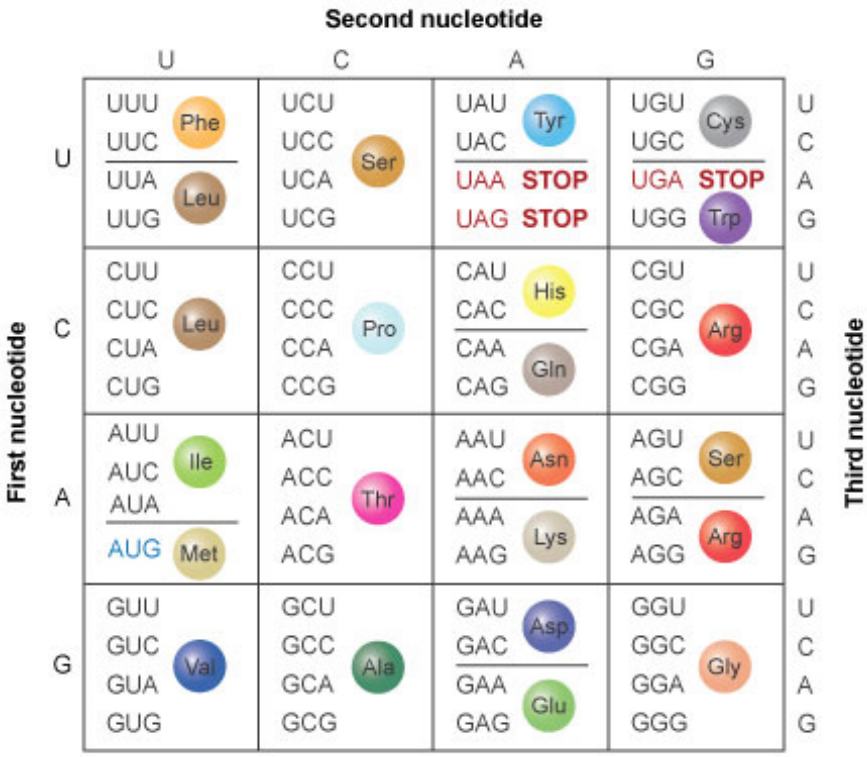



Figure 2.6: The amino acids specified by each codon [31].

### 2.1.3 Protein Structures

Protein structures range in size, from tens to several thousands of amino acids, and are categorised hierarchically into four distinct tiers, the primary, the secondary, the tertiary and the quaternary structure (Figure 2.7). This hierarchical approach was established to facilitate the observations of the various phases of protein formation. The number and type of amino acids of a protein are not enough, since the order and layout of their amino acids plays a major role because they determine the three-dimensional structure and hence the function of the protein.

#### 2.1.3.1 Primary Structure

The primary structure of proteins is the sequence of amino acids in the polypeptide chain. This structure is determined by the gene, which is a sequence of nucleotides in deoxyribonucleic acid (DNA) or ribonucleic acid (RNA), corresponding to the protein. The sequence of a protein defines the structure and function of the protein and is unique to that protein. For example, the pancreatic hormone insulin is composed of 51 amino acids in 2 peptide chains, A chain has 21 amino acids while B chain has 30 amino acids, as shown in figure 2.8. The amino-acid sequences, in both chains, are unique to insulin and have a specific order. In each chain there are three-letter abbreviations, which represent the names of the amino acids. These are displayed in the order that are present and illustrate the primary structure of insulin.

The unique sequence for every protein is determined by the gene encoding of the protein. If the nucleotide sequence of the gene's coding region is changed, a different amino acid might be added to the growing polypeptide chain, which would change the protein structure and function. For instance, in sickle cell anemia (a hereditary disease that affects the red blood cells), a single amino acid substitution (valine in the $\beta$ chain substitutes the amino acid glutamic) in the hemoglobin $\beta$ chain, changes the protein structure and function (Figure 2.9). A hemoglobin molecule is comprised of two alpha and two beta chains, each consisting of about 150 amino acids. Therefore, the molecule has about 600 amino acids. The structural difference between the sickle cell molecule (which dramatically decreases life expectancy) and a normal hemoglobin molecule is just one of the 600 amino acids. As a result of this small change in the chain, hemoglobin molecules form long fibres that distort the biconcave, or disc-shaped, red blood cells and causes them to assume a crescent or 'sickle' shape, which clogs blood vessels and leads to myriad serious health issues such as breathlessness, dizziness, headaches, and abdominal pain, for those affected by this disease.

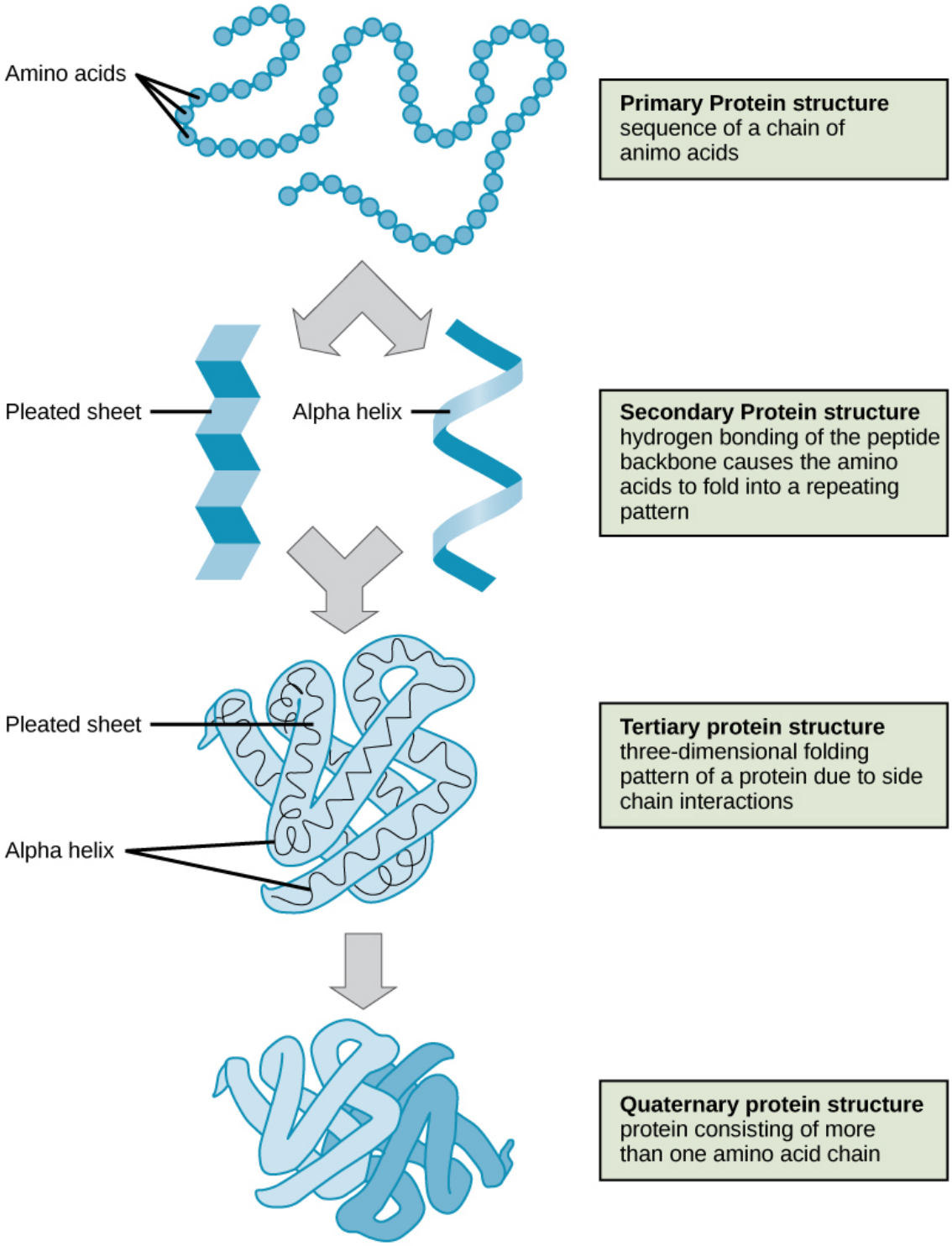



Figure 2.7: All four protein structures.

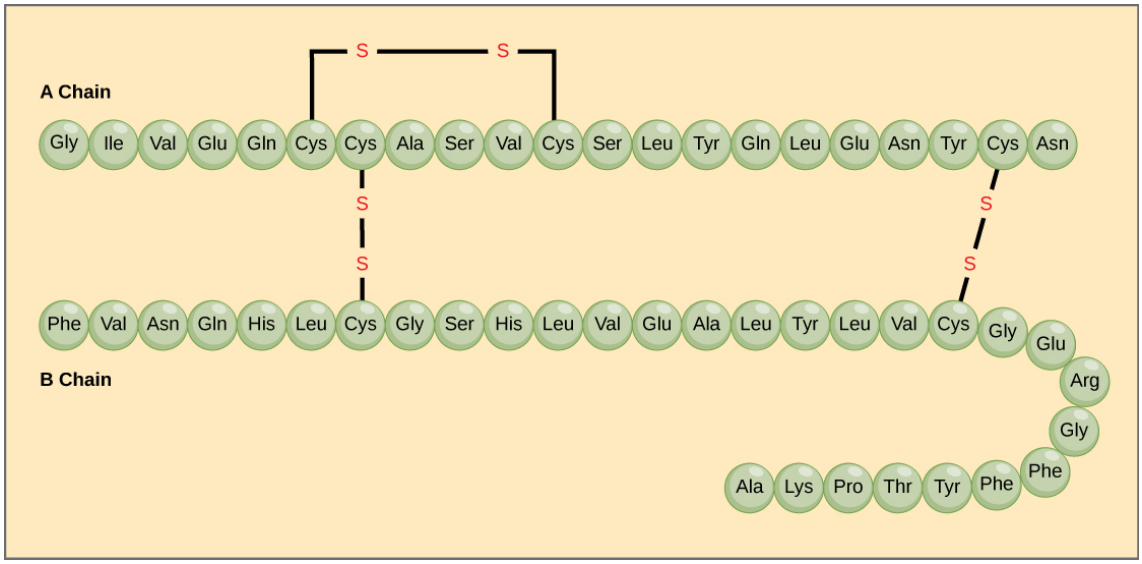



Figure 2.8: The first amino acid of the A chain is glycine (Gly), whereas, the last is asparagine (Asn) [32].

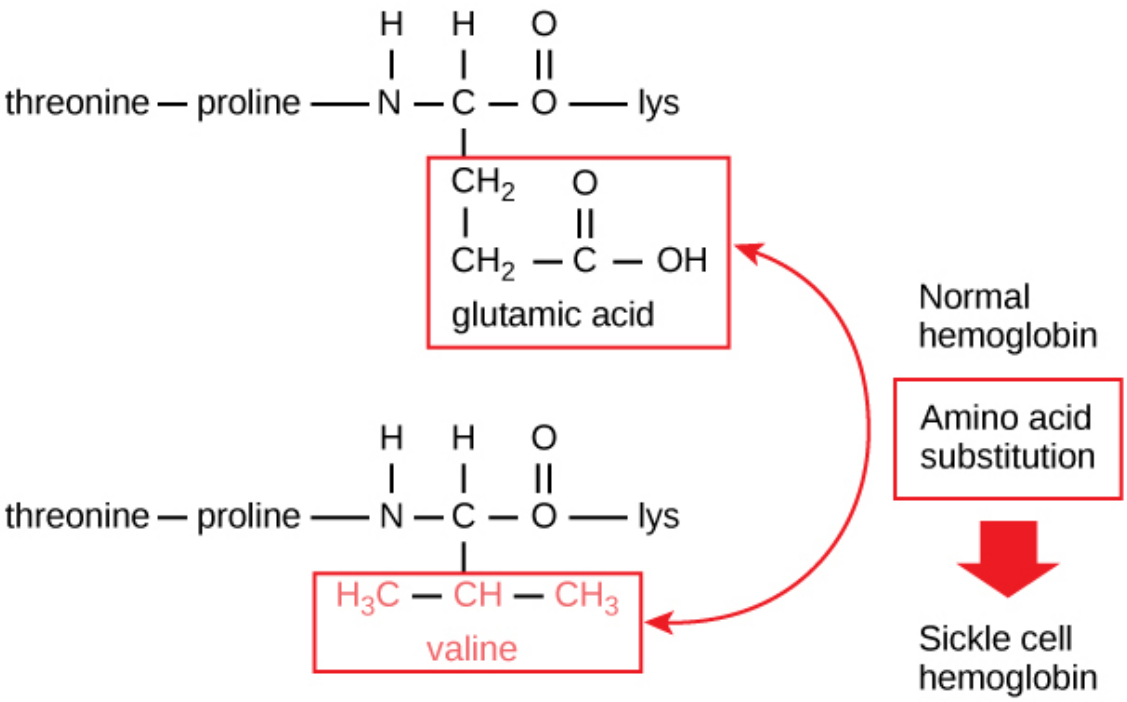



Figure 2.9: The diagram shows the substitution in a small part of the hemoglobin $\beta$ chain, where the amino acid at position seven, glutamate, is replaced by valine, in the sickle cell hemoglobin [32].

#### 2.1.3.2 Secondary Structure

The secondary structure of the protein refers to the local folding of the polypeptide in some regions and are defined by patterns of hydrogen bonds between the main-chain peptide groups. There are two main distinct categories of the secondary structure, the $\alpha$-helix and the $\beta$-strand or $\beta$-sheets. Both of these are held in shape by hydrogen bonds, which form between carbonyl and amino groups in the peptide backbone. Certain amino acids have a propensity to form an $\alpha$-helix, while others have a propensity to form a $\beta$-pleated sheet. The $\alpha$-helix and $\beta$-pleated sheet structures are in most globular and fibrous proteins and play an important structural role.

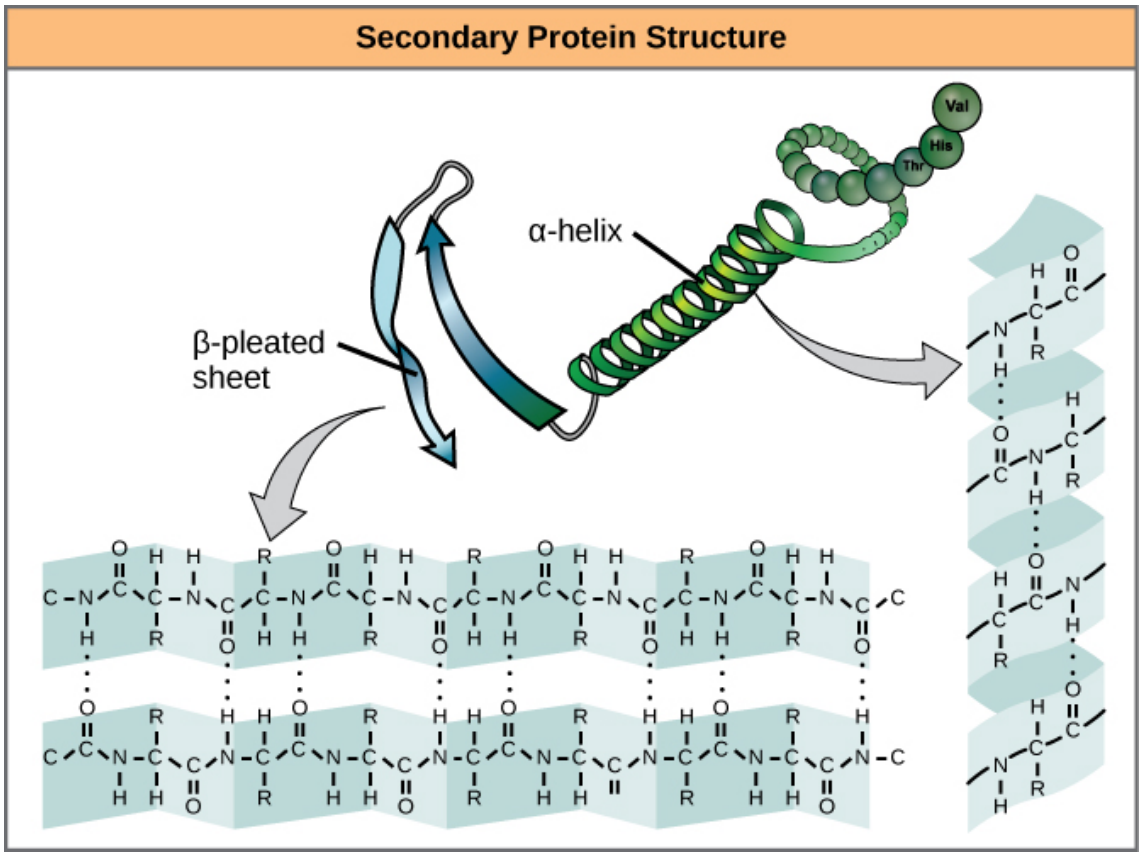



Figure 2.10: The diagram illustrates the shapes of the two main types of the secondary structure of proteins, the $\alpha$-helix and the $\beta$-strand [32].

#### 2.1.3.3 Tertiary Structure

The tertiary structure of proteins refers to a three-dimensional structure of monomeric and multimeric protein molecules. This structure is determined by a variety of chemical interactions on the polypeptide chain, such as ionic bonding, hydrophobic interactions, hydrogen bonding, and disulfide linkages (Figure 2.11). The protein's complex three-dimensional tertiary structure is created by the interactions among R groups. For instance,

R groups with like charges repel each other and those with unlike charges are attracted to each other (ionic bonds). The only covalent bond that forms during protein folding is the disulfide linkages, which are formed by interactions between cysteine side chains, in the presence of oxygen. As regards hydrophobic interactions, during the protein folding stage, the non-polar amino acids' hydrophobic R groups lie in the protein's interior, whereas, the hydrophilic R groups lie on the outside. Once a protein loses its three-dimensional shape, it may no longer be functional. The tertiary structure of a protein highly depends on the characteristics of its secondary structure, which is formed based on the order and layout of the amino acids (primary structure) of the protein.

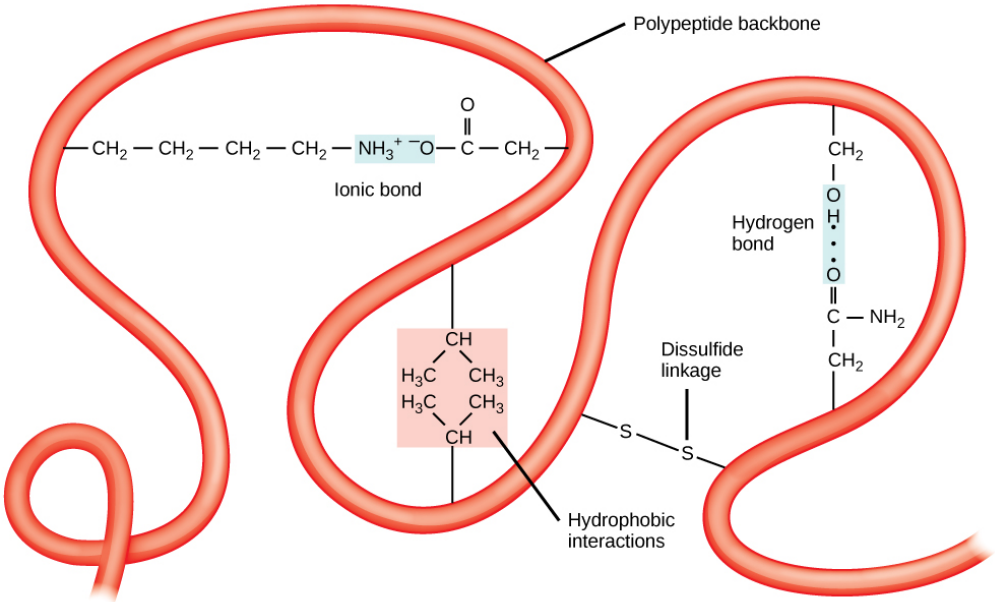



Figure 2.11: The diagram indicates some of the chemical interactions that determine the proteins' tertiary structure [32].

#### 2.1.3.4 Quaternary Structure

The quaternary structure of a protein is the three-dimensional structure consisting of the aggregation of two or more individual polypeptide chains (subunits) that operate as a single functional unit (multimer). For example, insulin (which is a globular protein) has a combination of hydrogen and disulfide bonds, which cause it to mostly clump into a ball shape. Insulin starts out as a single polypeptide and after forming the disulfide linkages that hold the remaining chains together, it loses some internal sequences in the presence of post-translational modification. Silk (which is a fibrous protein), on the other hand, has a $\beta$-pleated sheet structure, which is the result of hydrogen bonding between different chains. A representation of the quaternary structure can be found in figure 2.7.

## 2.2 Artificial Neural Networks Background

### 2.2.1 Origins of Artificial Neural Networks

Artificial neural networks (ANNs) are computing systems which are inspired by the biological neural network that exists in the brains of humans and animals. The term 'neural' comes from the basic functional units of the human nervous system, called 'neurons' or 'nerve cells'. These are located in various parts of the human body, like the brain which contains about $10^{11}$ neurons that are connected to $10^4$ other neurons.

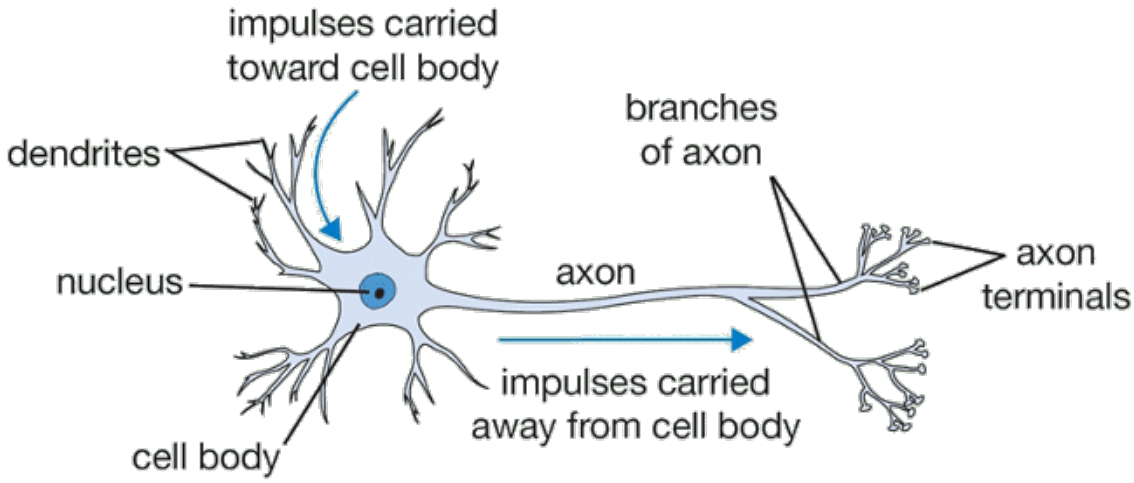



Figure 2.12: Structure of a Biological Neuron [33].

A biological neural network is a collection of neurons that receive, process and transmit information between each other, through electrical and chemical signals via specialized connections called synapses. It consists of three main components, the cell body, the axons and the dendrites. Figure 2.12 shows the direction of the impulses when a signal is carried towards or away from a neuron. The neuron receives signals from other neurons through dendrites. The body of the neuron adds all the incoming signals and calculates the input of the neuron. If the sum exceeds a certain threshold value the neuron triggers and the signal is transmitted through the axon to the other neighbouring neurons. Axon terminals are the connection point between brain neurons. The signal's strength, which is transmitted from one neuron to another, depends on the interconnection force of the neurons. The human nervous system is like an extremely high-connectivity network, which has trillions of neurons and billions of connections between them.

An Artificial Neural Network (ANN) has the same architecture with the biological neural network. An ANN has nodes that represent artificial neurons, a simplified version of biological neurons in terms of functionalities, and connections (edges) instead of synapses.

These connections are responsible for transmitting signals between the connected artificial neurons. ANNs have a similar behaviour with the biological neural network, but as they became more and more popular, the idea of replicating the human brain faded away.

The increasing demand for solving specific tasks, led to the development of various implementations of ANN, and some of them were based on the initial concept of biological neural networks. For instance, an ANN called Recurrent Neural Network (RNN) was based on the concept of short term memory and is used to recognise patterns, where the previous features can help predict the next ones. Another variation of ANNs is the Convolutional Neural Network (CNN) [34], which is used in this dissertation. The CNN is able to recognise patterns in two-dimensional (or three-dimensional) data, like images and videos, and feed the extracted features to a fully connected feed-forward Multi-Layer Perceptron (MLP) to classify the initial input data. There are many other variations of ANNs that were designed for specific tasks like speech translation or recognition, natural language processing, clustering or even playing video games. Some of these variations will be discussed in the following section.

### 2.2.2 Variations of Artificial Neural Networks and Optimizers

#### 2.2.2.1 McCulloch and Pitts (McP)

The first ANN model was suggested by Warren McCulloch and Walter Pitts in 1943 [35] (Figure 2.13). The design of this artificial neuron was very simple and was based on a single biological neuron of the human brain. An input vector performs multiplications with the weight values and provides the signals to the artificial neuron. Then, the artificial neuron sums those signals and transfers the result to a threshold function, also known as step or heaviside function (Figure 2.14), which does not provide enough information about how close or how far the target output is. The output signal of the model was 1, if the value exceeded the a specific threshold value, otherwise the output signal was 0, which means this model can be used only for binary classification.

The inputs are classified based on the weights of the connections and the threshold value (Equation 2.1, where y is the output of the network, x the input vector, w the weight vector, $w \cdot x$ the dot product and s the threshold). For instance, for a two-dimensional input vector, in a simple two-dimensional scenario, the decision line can be calculated with equation 2.2.

$$y = \begin{cases} 1 & \text{if } w \cdot x > s \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

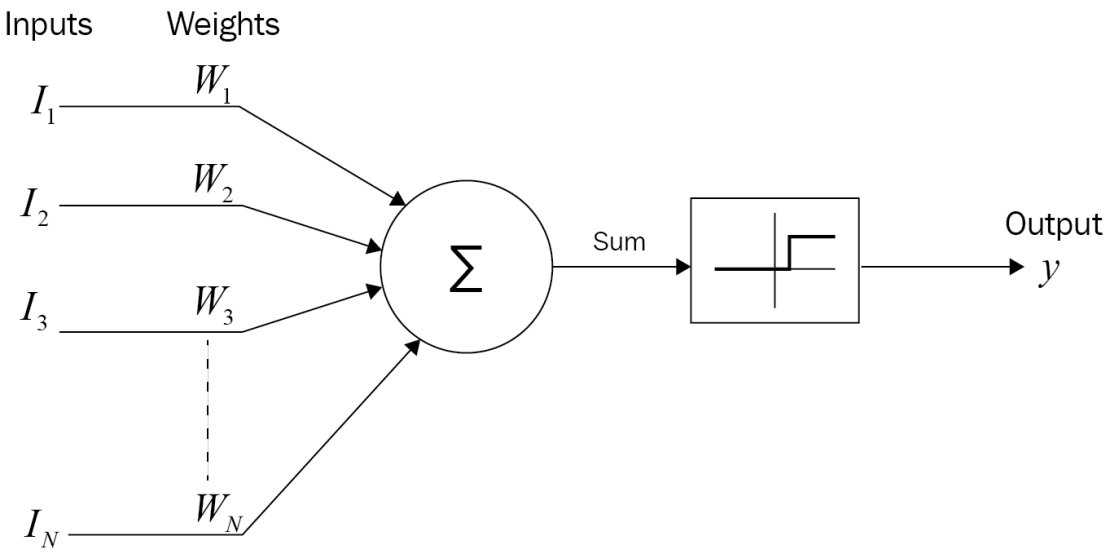



Figure 2.13: McCulloch and Pitts artificial neuron [35].

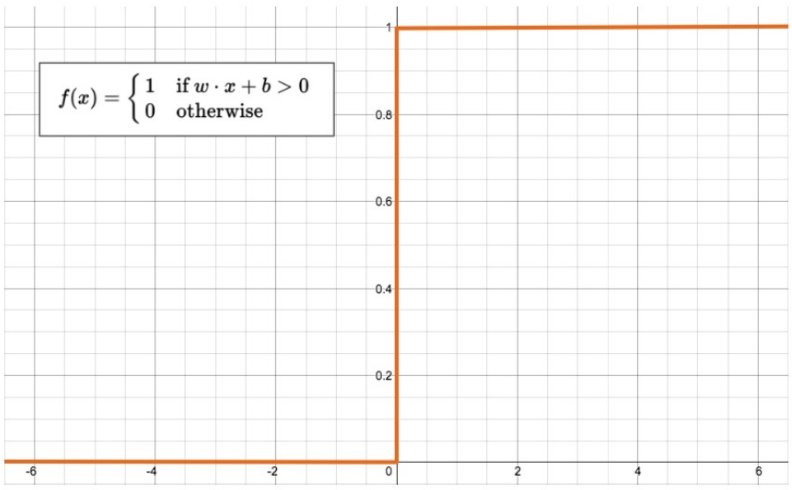



Figure 2.14: The step or heaviside function.

$$x2 = -(\frac{w1}{w2})x1 + (\frac{s}{w2}) \tag{2.2}$$

If the goal is to classify the OR gate (Table 2.2) with a McP model, the model could use infinite different ways to solve the problem. For instance, the model could have weights of W = [2, 2] and a threshold value S = 1. In figure 2.15, (b) illustrates the decision line for the OR gate, where inputs above the line are classified as Class 1 while inputs below the line are classified as Class 0. The equation for this decision line is $x2 = -x1 + 0.5$.

| x1 (input) | x2 (input) | y (output) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Table 2.2: Truth table for OR gate.

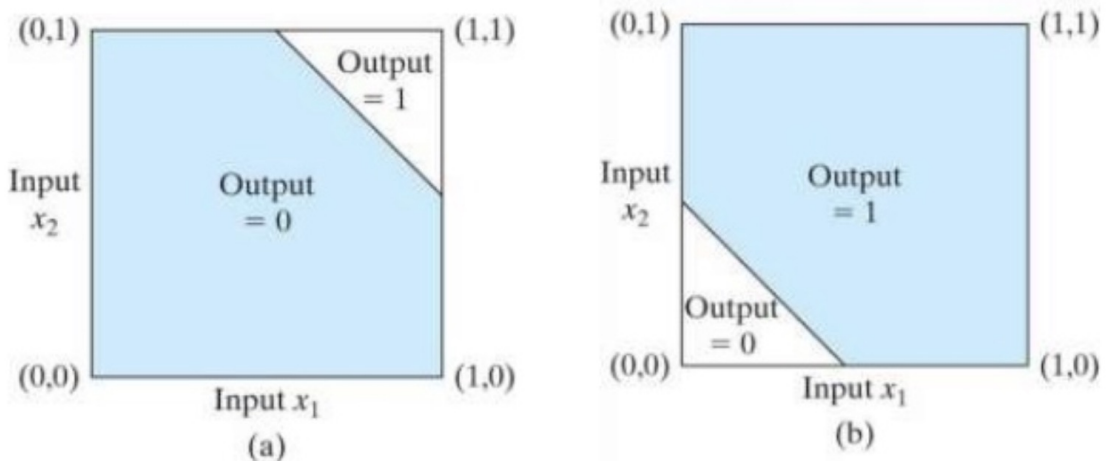



Figure 2.15: Decision lines for AND gate (a) and OR gate (b).

The training phase of McP neurons requires the input and target output to be presented to the network, which calculates the actual output for the given input and adjusts the weights accordingly. For example, if the output is 1 but the target output is 0 the weights are modified, while in the case where both the output and target output are 0 the weights remain the same. This process is also known as the Perceptron Learning algorithm [36] (Algorithm 1).

```
Perceptron Learning Algorithm

   1. Initialize weights and threshold randomly.
   2. Present input and desired output.
   3. Calculate actual output (Equation 2.1).
   4. Adapt weights:

   if output 0, should be 1:  w_i(t+1) = w_i(t) + η · x_i(t)
   if output 1, should be 0:  w_i(t+1) = w_i(t) − η · x_i(t)
   if output is correct    :  w_i(t+1) = w_i(t)

   where 0 ≤ η ≤ 1 the learning rate, controlling the
   adaptation rate.
```

**Algorithm 1:** Perceptron Learning Algorithm.

This algorithm was thought to be very promising, but after a while it was proven that the perceptron algorithm could only solve problems with linearly separable patterns. In these problems, a straight line or hyperplane, which separates the patterns, can be found in space, like the OR gate problem which was mentioned earlier (Table 2.2). However, this algorithm cannot solve problems that require more than one straight lines or hyperplanes to separate the different classes, not even simple ones like the XOR gate problem (Figure 2.16). Except from that, there was no indication on how close to the target output was the predicted output because of the binary (either 1 or 0) output of the heaviside function. This problem was the main motivation for developing more sophisticated networks and algorithms, some of which will be discussed subsequently.

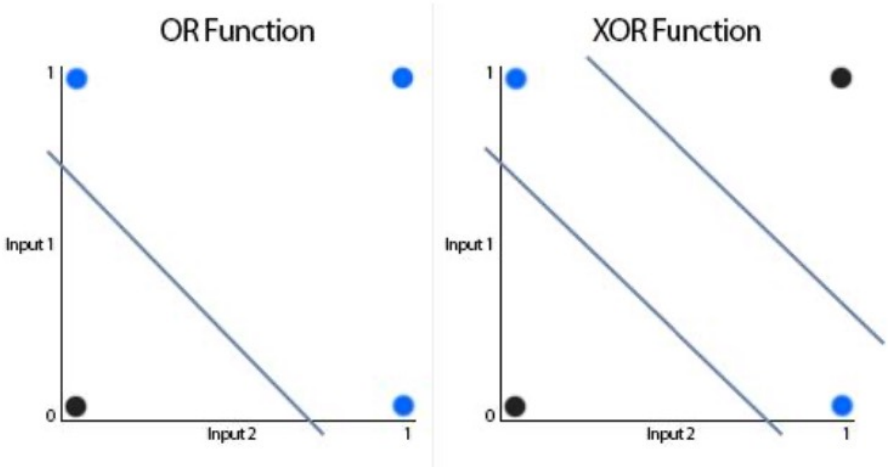



Figure 2.16: The OR gate is linearly separable while the XOR gate is not.

#### 2.2.2.2 Multi-Layer Perceptron (MLP)

Multi-Layer Perceptron (MLP) neural networks are currently the most popular and well-known variants of ANNs. They consist of multiple, slightly modified, McCulloch and Pitts neurons, which form layered feed forward networks (Figure 2.17). McP neurons use a specific threshold activation function (step function) while MLP neurons can use any arbitrary activation function (Table 2.3). This is the reason why McP can only perform binary classification, while MLP can perform regression or classification, depending on the selected activation function. Furthermore, activation functions provide an indication to the network whether the outputs are closer or further of the expected outputs, which helps the network adjust the weights accordingly, to improve predictions.

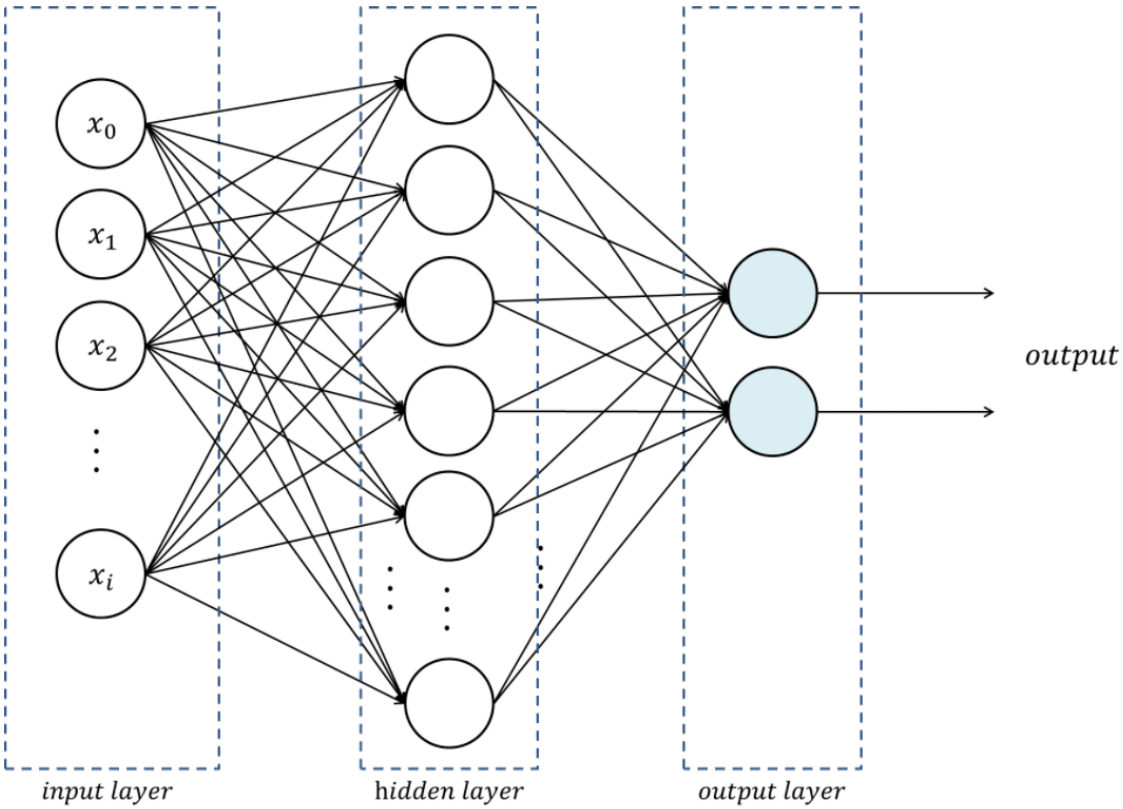



Figure 2.17: Multi-Layer Perceptron Neural Network with one hidden layer.

An MLP neural network consists of an input layer, one or more hidden layers and an output layer. The hidden and output layers are active, while the input layer is not active (only forwards the input data to the network). Each layer has one or more neurons and an independent neuron unit, also known as 'bias', which has a constant input value of one (1). The role of the bias unit is to help the network adapt more effectively to the provided data. The number of hidden layers is very important as it specifies the possibilities of the network and processes the biggest amount of information during the training (learning)

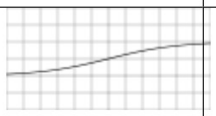
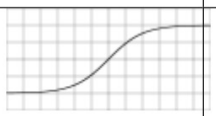
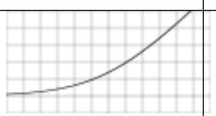
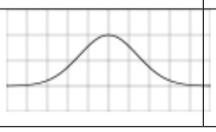
| Name | Plot | Equation | Derivative | Range |
|---|---|---|---|---|
| **Heaviside** |  | $f(x) = \begin{cases} 1 \; if \; x > 0 \\ 0 \; otherwise \end{cases}$ | $f'(x) = \begin{cases} 0 \; if \; x \neq 0 \\ ? \; if \; x = 0 \end{cases}$ | {0,1} |
| **Logistic / Sigmoid** |  | $f(x) = \frac{1}{1+e^{-x}}$ | $f'(x) = f(x)(1 - f(x))$ | (0,1) |
| **TanH** |  | $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ | $f'(x) = \frac{1}{x^2+1}$ | (-1,1) |
| **Rectified linear unit (ReLU)** |  | $f(x) = \begin{cases} 0 \; if \; x < 0 \\ x \; otherwise \end{cases}$ | $f'(x) = \begin{cases} 0 \; if \; x < 0 \\ 1 \; otherwise \end{cases}$ | [0,∞) |
| **SoftPlus** |  | $f(x) = \ln(1 + e^x)$ | $f'(x) = \frac{1}{1+e^{-x}}$ | (0,∞) |
| **Gaussian** |  | $f(x) = e^{-x^2}$ | $f'(x) = -2xe^{-x^2}$ | (0,1) |

Table 2.3: List of the most popular activation functions.

phase (Figure 2.18). The neurons of the first hidden layer determine the number of decision lines that can separate the patterns into classes. The second hidden layer gives the ability to form arbitrary complex decision shapes, which are able to separate any classes, so there is no need for more than two hidden layers in a neural network [37].

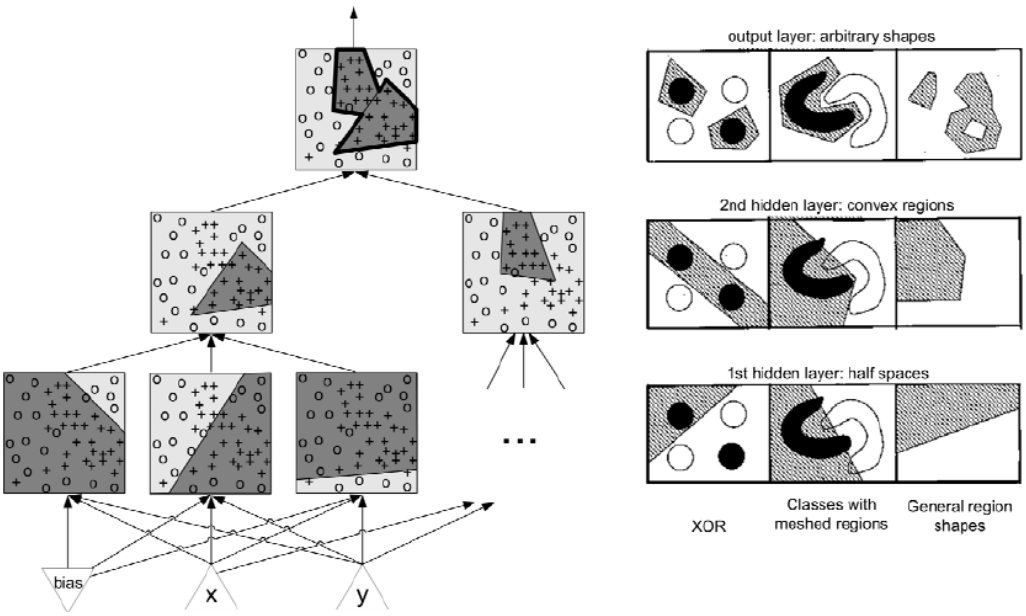



Figure 2.18: Decision regions based on the number of hidden layers.

The calculation process of the networks' output is very similar with the one used in McP. The input layer forwards the input values to the first hidden layer, which calculates the sum of the bias and the dot product of the weights and the input vector, and then passes that value to the activation function (Equation 2.3, where $y$ is the output of a single neuron, $x$ the input vector for that neuron, $w$ the weight vector, $w^T \cdot x$ the dot product, $b$ the threshold and $f$ the arbitrary activation function). The output signals of the activation function are then fed as inputs of the next hidden or output layer, which then repeats this process until there are no more layers to pass the signals.

$$y = f\left(w^T x + b\right) \tag{2.3}$$

**Gradient Descent**

Gradient descent is one of the most popular optimisation algorithms for training ANNs. It is considered a mathematical optimization algorithm that is able to minimize a function by iteratively moving in the direction of steepest descent, which is defined as the negative of the gradient. An error function is used to calculate how successful the network predicting the classes of the input patterns was, like the mean squared error (MSE) function (Equation 2.4, where $t$ is the target output, $o$ the actual output, $p$ denotes the pattern and $j$ the neuron). The objective is to minimise the error value, which is the difference between the target and actual outputs. By adjusting the weight vectors according to the negative of the derivative of the error value, of the current pattern, with respect to each weight (Equation 2.5), where n is the learning rate), at some point the correct classifications will be maximized.

$$E = \frac{1}{2}\sum_j \left(t_{pj} - o_{pj}\right)^2 \tag{2.4}$$

$$\Delta w_{ij} = -n\frac{\partial E_p}{\partial w_{ij}} \tag{2.5}$$

**Backpropagation Algorithm (BP)**

In order to calculate the error and use gradient descent to minimize it, both target and predicted outputs must be known. In the output layer this is fine as both values are available, however, in the hidden layers the target values are unknown, which means that only the weights between the last hidden layer and the output can be adjusted. To solve this issue, the backpropagation algorithm was suggested, which propagates the error from the output layer back to the last hidden layer, which then does the same until all the weights are updated (Algorithm 2, where $\delta_{ij}$ is the error signal, $y_{ij}$ is the actual output and $d_{ij}$ is

the target output of neuron $i$ of layer $j$. The $\delta_{ik}$ is the same as $\delta_{ij}$ but for the previous iteration of the algorithm).

```
Backpropagation

Repeat:
      For each pattern :
            // Forward Pass
            Calculate the output
            // Backward Pass
            For each layer j, starting at the output:
                  For each unit i:
                          // Compute the error
                         If output neuron: δij = yij(1 − yij)(dij − yij)
                         If hidden neuron: δij = yij(1 − yij) Σ δik · Wjk
                         For each weight to this unit:
                             Compute and apply Δw
      Compute total error
      Increment epoch counter
Until small enough error or epoch counter exceeded
```

**Algorithm 2:** The Backpropagation algorithm.

More specifically, to update all the weights two passes are required, a forward pass to calculate the error based on the given input pattern, and a backward pass, where the error is back propagated to the previous layers and all the weights are updated respectively. The entire process is repeated for every pattern, until all patterns have been passed into the network (one epoch), which is also known as the online update mode. There are two alternatives, the batch and mini-batch modes. The first feeds the network with all the patterns at once and gets cumulative updates for the weights, which usually helps the network learn more effectively. However, if the input datasets are too big and cannot fit into memory, this method cannot be used. The second method is a combination of the online and batch mode and can be used for big datasets. This method takes the input patterns and splits them into smaller chunks, called mini-batches, then it feeds the network with one mini-batch at a time. The size of the mini-batch can be adjusted to ensure that there are no 'out of memory' issues, which makes this more flexible compared to the other two methods. The goal is to feed the network all the input patterns several times until the error reaches a specified value or until a number of epochs (when all the patterns have been fed into the network) has passed.

#### 2.2.2.3 Recurrent Neural Network (RNN)

The Recurrent Neural Network (RNN) is a variation of MLP, which instead of feeding the input forward to the next layers, it uses recurrent inputs. Recurrent inputs are the output signals from the hidden layer or the output layer, which are fed into a previous layer or even to the same layer. This technique creates a form of 'memory' for the network, since the output depends on both the current input and the input from the previous iterations. This makes RNNs great for dynamic problems, like timeseries or sequence predictions.

There are two main versions of RNNs, the Jordan RNN [38] and the Elman RNN [39] (Figure 2.19). The main difference between the two versions is that the first transfers its output to a context layer, also known as state units, which then feeds the network along with the new input patterns. The second variation, on the other hand, feeds the hidden layer output to a context layer, also known as context units, which is fed back to the hidden layer.

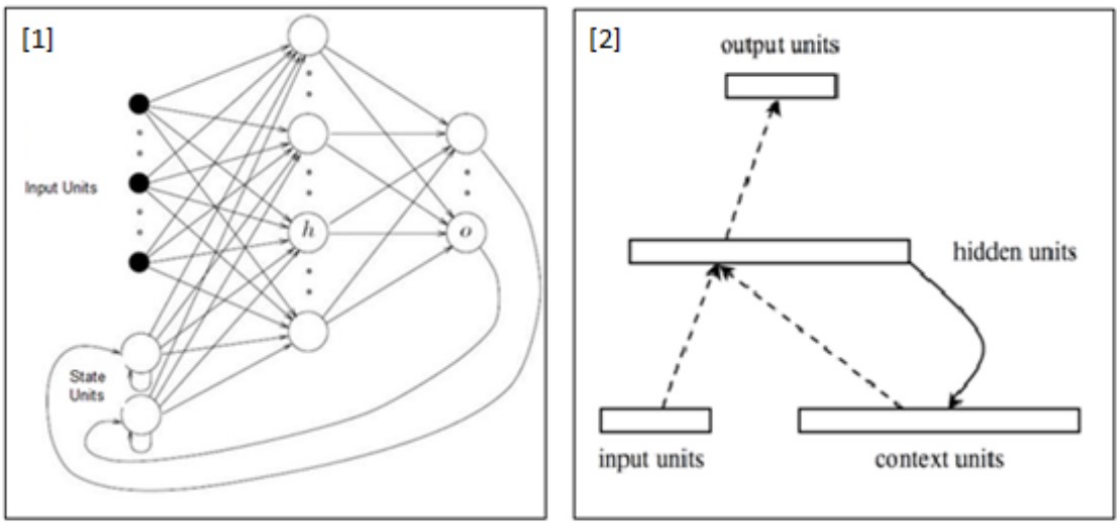



Figure 2.19: RNN variations, Jordan network (left), Elman network (right).

#### 2.2.2.4 Convolutional Neural Network (CNN)

**Description of Convolutional Neural Networks**

A Convolutional Neural Network (CNN) is a class of deep artificial neural networks, which is most commonly applied to analysing visual imagery. Its application ranges from image and video recognition, recommender systems, image classification to medical image analysis, and natural language processing (NLP). CNNs are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. Multilayer perceptrons (MLPs), which are usually fully connected networks (each

neuron in a layer is connected with all the neurons in the next layer), are prone to overfitting data. CNNs on the other hand take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme. The architecture of a CNN is designed so that it can take advantage of the two-dimensional (2D) structure of an input image (or any other 2D input, such as speech signals).

**Architecture of Convolutional Neural Networks**

A Convolutional Neural Network (CNN) consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers. The activation function used is commonly a Rectifier Linear Unit (RELU) layer, and is subsequently followed by additional convolutions, pooling layers and a fully connected layer. The final fully connected layer, which is usually a multilayer perceptron (MLP) network, uses the backpropagation learning algorithm for training. The input of a convolutional layer is an image of size $d \times d \times c$, where $d$ is the height and width of an image and $c$ is the number of channels of the input image (e.g. an RGB image has $c = 3$). A convolutional layer has k filters (or kernels) of size $m \times m \times n$, where $m$ is smaller than the dimensions of the image and $n$ can be either the same as the number of channels $c$ or smaller (may vary for each kernel). Convolutional layers convolve the input, which leads to the creation of k feature maps of size $d - m + 1$, and pass their output to the next layer. Subsequently, each feature map is sampled, typically averaging or maximizing above the same areas in feature maps of size $p \times p$ (where $p$ is between 2, for small images, and usually does not exceed 5, for larger images). A bias and a sigmoid nonlinearity is applied to each feature map, prior or after the pooling layer.

Figure 2.20 illustrates an example of a CNN which is used to classify images of handwritten digits. The diagram shows the different layers of a CNN (convolution, max pooling, multilayer perceptron) and the feature maps that are extracted from each image (small squares). At the end of the CNN (right hand side), there is a fully connected network (MLP) which is used to classify the input image [40].

A pooling layer between the hidden convolutional layers is a common tactic for classic CNN architectures. The pooling layers are mainly used to reduce the dimensions of the output of each layer, the number of parameters and the complexity of the network, which consequently reduces the total computation time of the network. This practice also prevents the network from overfitting (adapting to the training data, making it less effective at predicting new data patterns), which can be determined by observing the training and test error values. The pooling layers are independent from the other layers and they pro-

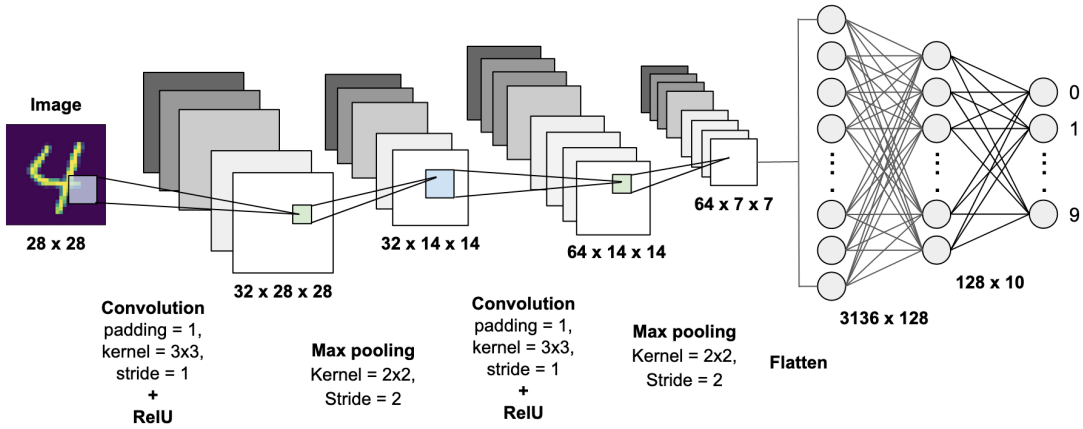



Figure 2.20: A CNN example for digit image classification.

cess the output of each kernel (filter) separately. Even though, there are different types of pooling layers, max pooling, min pooling (which is the opposite of max pooling), average pooling and L2-normalization pooling, the max pooling technique seems to work better than the rest [41]. This technique, as its name suggests, takes the max value from each filtered result and returns it. Figure 2.21 illustrates an example of max and average pooling, where the kernel size (filter size) is $2 \times 2$ and the stride (how many slots to skip) is two (2). The applied filters can be distinguished by their colors.

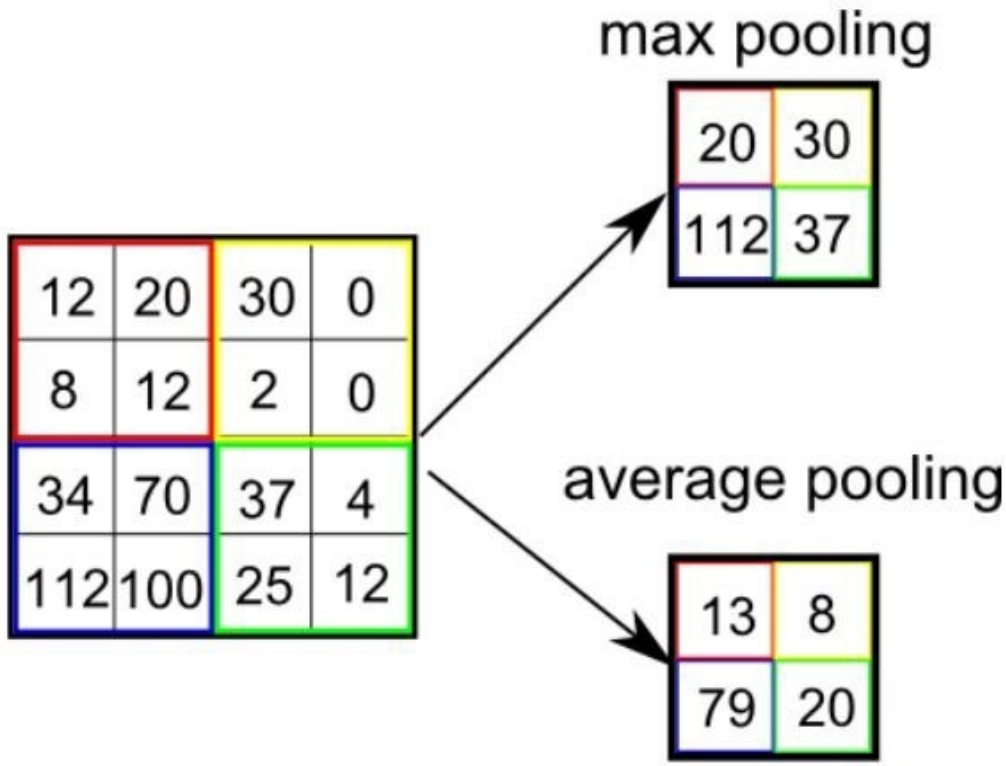



Figure 2.21: Example of max and average pooling.

The input of a CNN is a 3D array, also called 3D tensor. For instance, if the purpose of a CNN is to identify objects in a $50 \times 50$ pixels picture, the input would be a 3D tensor with shape $50 \times 50 \times 3$. That is because each pixel is represented by three values, one for red, one for green and one for blue (RGB). In the PSSP problem, a 2D tensor is enough, as a 2D tensor is the same with a 3D tensor where the third dimensions has size one. The shape of this tensor will be $L \times 20$, where L is the number of lines of the input file and 20 represents the 20 known amino acids. An example with visualizations and more details on how CNNs work can be found here [42].

One of the main advantages of CNNs, is the fact that they can extract features from complex sequences, due to the small number of synaptic weights. For example, if the input array size is $28 \times 28 \times 3$ (RGB) and the kernel size is $5 \times 5$, then each neuron of the convolution layer will be connected with an area of the input array with shape $5 \times 5 \times 3$. This means that each neuron has 76 ($5 \times 5 \times 3 = 75 + 1 = 76$) synaptic weights, which can extract features and adapt to complex input data.

In some cases, it is necessary to add zero padding (append zero values) around the input data (like a frame for the input array). The number of rows and columns of zeroes is variable, which makes it possible to control the dimensions of the output of a hidden convolution layer, using zero padding. Figure 2.22 illustrates an example of zero padding, where two borders of zeros are placed around the $32 \times 32 \times 3$ input.

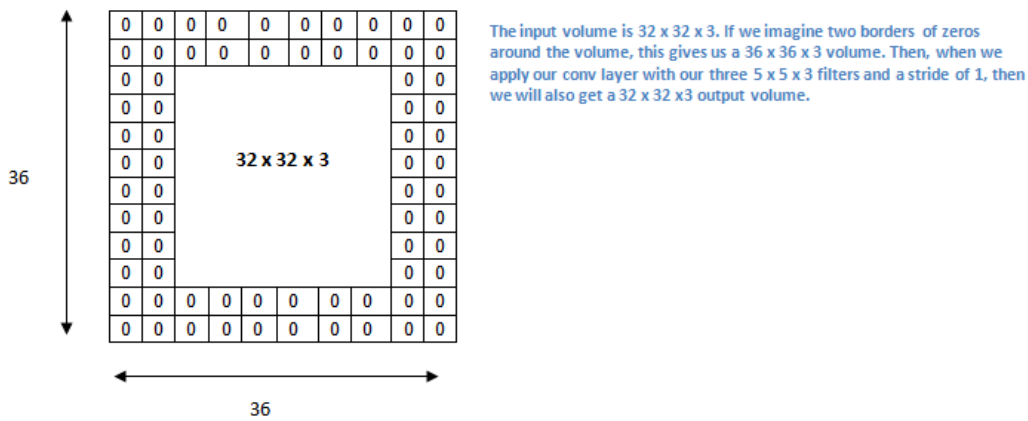



Figure 2.22: Example of zero padding.

#### 2.2.2.5 Line Search

Line search is one of the basic iterative approaches, used to find a minimum $x^*$ of an objective function. For an ANN, $x$ represents the weights of the network, while the objective function represents the error function. Equation 2.6 illustrates the essential components

to calculate the next iteration of $x$. The step size determines the size of the step of $x$ in that direction. Line search, in each iteration, attempts to find the best step size, which can minimise the objective function in a specific search direction. On the other hand, gradient descent requires a learning rate which determines how small or how big is each step. If the step is too small the learning process will take significantly more time and can lead the network to a local minimum (instead of the global minimum, which is the desired outcome). If the step is too big then it is very likely that the objective function will jump far away from the desired minimum.

$$x_{n+1} = x_n + a_n d_n \tag{2.6}$$

Therefore, applying the optimal step size is very important, as it can prevent the network from moving further away from the minimum. In order to find the step size, a naive approach was to move along a search direction in small steps and after each step calculate the error, if the error starts increasing then stop and change direction [43]. However, this approach is not very efficient, robust or accurate compared to other variations of line search [44].

#### 2.2.2.6 Conjugate Gradient (CG)

The Conjugate gradient algorithm (Algorithm 3), unlike gradient descent, in each iteration changes the direction to prevent the network from becoming counterproductive (reversing the progress). In addition to that, in an N-dimensional problem, the CG algorithm is guaranteed to find a solution in N steps, since in every CG step the network obtains the minimum of that direction. Figure 2.23 compares the CG and the gradient descent algorithm on the same two-dimensional problem. Conjugate gradient managed to converge in just two steps, while gradient descent required several steps.

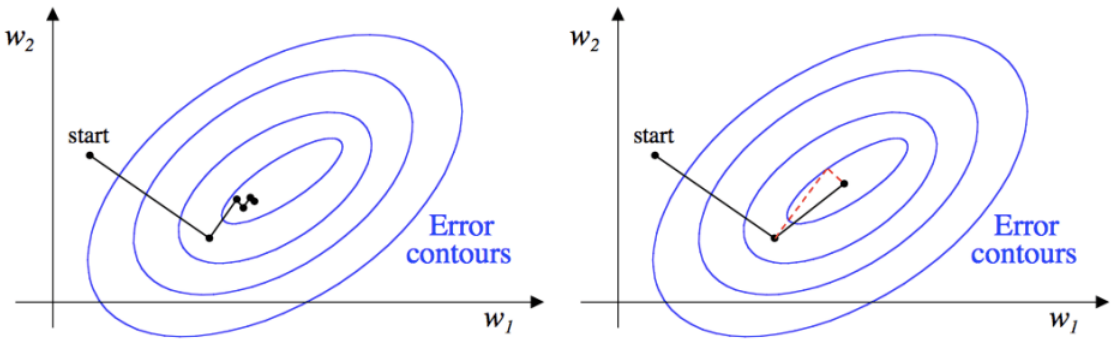



Figure 2.23: Gradient Descent (left) vs Conjugate Gradient (right) on a 2D problem.

```
Conjugate Gradient
  1. Initialize weight vector w0 randomly, set i=0
  2. Evaluate the gradient vector gi, and set the initial
     search direction di =-gi
  3. Use Line Search to find best step size a, which
     minimizes the function f(wi+adi)
  4. Update weights wi+1 = wi + adi
  5. Test stopping conditions
  6. Evaluate new gradient vector gi+1
  7. Evaluate new search direction di+1= -gi+1 + βidi,
     where βi is given by one of:

     βi = (g_{i+1}^T (g_{i+1} − g_i)) / (g_i^T g_i)      (Polar and Ribiere)

     βi = (g_{i+1}^T g_{i+1}) / (g_i^T g_i)              (Fletcher and Reeves)

  8. Set i=i+1 and go to step 3
```

**Algorithm 3:** Conjugate Gradient Algorithm [45].

#### 2.2.2.7 Newton's Method

An iterative method, originally used to find approximations of the roots of real-valued functions, is currently used in optimisation problems to find the maximum or minimum of a function and is known as the Newton's Method. The derivative of a function at a maximum or a minimum point is zero, which makes it possible to find local maxima and minima by using the Newton's Method on the derivative of the optimisation function.

Newton's Method is considered a second-order optimisation algorithm, since it requires information about the second derivative of the optimisation function. Compared to first-order optimisation algorithms (like gradient descent), second-order optimisation methods can achieve faster and more accurate convergence to the minimum of a function.

In a simple first-degree polynomial (Figure 2.24), 1D problem, of a function $f(x)$ and a sub-optimal initial solution $x_0$, Newton's method suggests the following:

1. Set $x_i = x_0$
2. Find the equation of the tangent at $x_i$
3. Find the point $x_{i+1}$ at which the tangent line intersects with the x-axis
4. Find the projection of $x_{i+1}$ on $f(x)$
5. Set $x_i = x_{i+1}$ and repeat from 2 until $f(x_i) <$ threshold

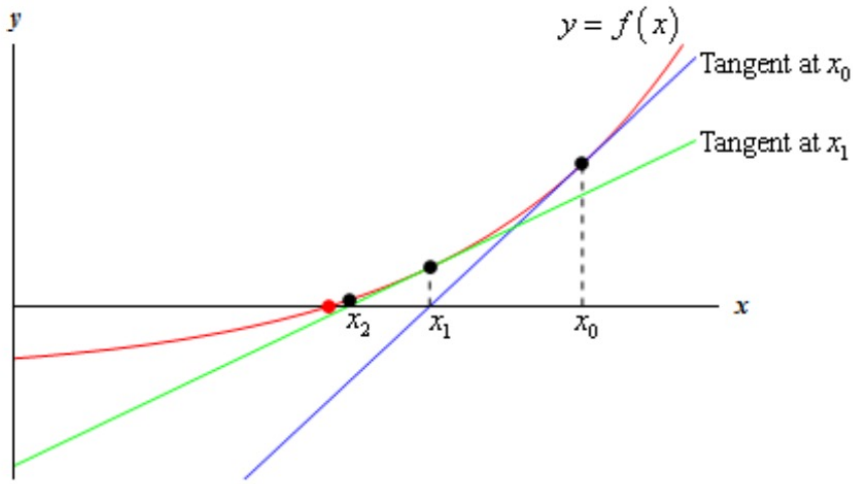



Figure 2.24: Newton's method in a first degree polynomial problem [23].

The equation of a point-slope line is:

$$y - y_1 = m(x - x_1) \tag{2.7}$$

In 2.7 the derivative can be used instead of the slope $m$ and this can be rewritten as:

$$f(x) - f(x_1) = f'(x)(x - x_1) \tag{2.8}$$

Since $x_1$ is the point of interaction on x-axis, $f(x_1) = 0$ which gives the update rule for $x$ for optimizing the function as:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2.9}$$

The previous simple example was used just to provide the intuition behind the method of finding the roots of a function. In optimisation theory, this method actually approximates the function $f(x)$ with a local quadratic function around $x$ and moves towards the minimum of that approximated function with iterative steps. This process is repeated until a specified error threshold is reached or after a certain number of iterations has passed. The quadratic approximations around the weights at each iteration are shown in figure 2.25.

For the approximation of the function $f(x)$, the second-order Taylor expansion (second series Taylor approximation) is being utilized.

$$\mathrm{f}(x_0 + \mathrm{x}) \approx \mathrm{f}(x_0) + \mathrm{f}'(x_0)\,\mathrm{x} + \mathrm{f}''(x_0)\frac{x^2}{2} \tag{2.10}$$

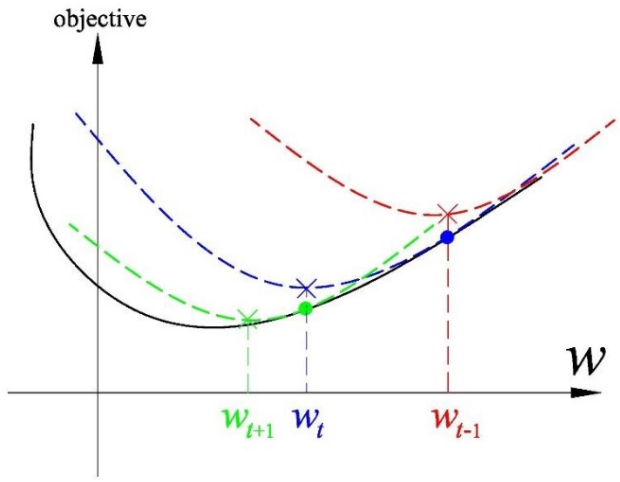



Figure 2.25: Local Quadratic approximations [23].

In order for $f(x_0+x)$ to be a minimum, an optimal $x$ value must be specified. Newton's method takes the derivative of the Taylor series and sets it equal to zero (Equation 2.11).

$$\frac{\mathrm{d}\left(\mathrm{f}(x_0)+\mathrm{f}'(x_0)\,\mathrm{x}+\mathrm{f}''(x_0)\,\frac{x^2}{2}\right)}{\mathrm{dx}} = f'(x_0)+f''(x_0)\,x = 0 \Rightarrow x = -\frac{f'(x_0)}{f''(x_0)} \tag{2.11}$$

This $x$ is just the absolute minimum of the local approximation of $f(x)$ around the initial solution of $x_0$ and not the absolute minimum of $f(x)$. For the minimum of the objective function this process must be repeated multiple times, until it eventually converges to a minimum. The final update rule for optimizing the function $f(x)$ for a 1D problem is given by the equation 2.12 .

$$x_{n+1} = -\frac{f'(x_n)}{f''(x_n)} \tag{2.12}$$

This algorithm, however, can work only for objective functions with a single dimension $(f:\mathbb{R}\rightarrow\mathbb{R})$.

If the objective function, has multiple dimensions $(f:\mathbb{R}^n\rightarrow\mathbb{R})$, the algorithm must be modified by replacing derivatives with gradients and second derivatives with Hessians (the matrix of second partial derivatives, figure 2.26)

$$x_{n+1} = -\frac{\nabla\mathrm{f}(x_n)}{H(f)(x_n)} \tag{2.13}$$

Equation 2.13 is the final update rule, which is the one cited as the Newton's method.

$$H = \nabla^2 f = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \end{pmatrix}$$

Figure 2.26: The Hessian matrix of the error function with respect to the weights.

The Newton's Method seems very efficient computationally because it calculates the quadratic approximation around the solution and immediately finds the minimum of that curvature, instead of fitting a plane to the solution, like the Gradient Descent algorithm. The problem is that it can become computationally impossible to calculate and store the entire hessian matrix of the function, as the parameters increase. Because of that, the standard Newton's method cannot be applied and used in Artificial Neural Networks, which have thousands or even millions of parameters. There are some variations of this algorithm, however, which can be used with ANNs. One such variation is the Hessian Free Optimization algorithm [46] which, instead of calculating and storing the entire Hessian matrix, calculates an approximation that requires less computational resources and does not have to be stored. This algorithm will be discussed in more detail in section 2.3 below.

## 2.3 Hessian Free Optimisation (HFO)

### 2.3.1 Intro to HFO

As mentioned in the previous section, the Newtons's method, as a second order optimization algorithm, can achieve faster and more accurate convergence to the minimum of a function, compared to first order algorithms, like the gradient descent. In high dimensional problems, first order optimization algorithms can be extremely slow or ineffective due to a problematic phenomenon, called Vanishing Gradient. This phenomenon can be described as a state where the updates for the first layers of a network are very close to zero, because of the backpropagation of the error and the decreasing gradient. As a result, the front layers have almost no information to adjust their weights, which means that the training process becomes slower or even ineffective.

On the other hand, second order optimization algorithms, like Newton's method, calculate the curvature of the error surface (Hessian Matrix) which significantly improves each step

of the optimisation process. What makes these algorithms so efficient is the fact that they attempt to find a quadratic curve that tightly fits at each point, which helps them find the minimum of that curvature immediately, unlike first order algorithms which select a fitting plane and then calculate the next step. However, these second order algorithms have some limits. For instance, in case of a big ANN (with thousands to millions of parameters) sometimes it may not be possible to calculate the Hessian Matrix, due to the extremely high memory requirements. Because of that, several variations of Newton's method were suggested, like Newton-CG, CG-Steihaug, Newton-Lanczos [47], and Truncated Newton [48], but their applications on machine learning and neural networks have been either extremely limited or not effective at all [49].

The Hessian Free Optimization (HFO) algorithm [46] is a variation of Newton's method, which uses the local quadratic approximations to generate the suggested updates. Unlike other Newton's variations, HFO managed to lift the memory constraints, which made it an effective optimisation algorithm for ANNs. This algorithm, instead of calculating and storing the entire Hessian Matrix ($H$), calculates the dot product of $H$ with an arbitrary vector $u$ ($Hu$). It takes advantage of mathematical techniques, like finite differences, which computationally costs the same as a single gradient calculation. This means that HFO can calculate the dot products of the Hessian with arbitrary vectors, instead of using the Hessian, and it can optimize the local quadratic objective approximations by using the conjugate gradient (CG) algorithm, to compensate for not having the Hessian Matrix. As mentioned in section 2.2.2.6, the CG method requires $N$ iterations to converge (where $N$ is the number of the network's parameters), but there are various stopping criteria that allow early termination (after significant progress is made), which reduce the total training time.

Even though, the Hessian Matrix is not calculated in HFO, there are no approximations, as the $Hu$ product is computed accurately. In the standard Newton's method the approximated quadratic is fully optimized, while the HFO does not perform complete optimization with the un-converged CG algorithm [46] and this is the only difference between the two approaches. The difference between the accuracy of Newton's method and the HFO with the not fully converged CG is that small that makes it insignificant, where the benefits in terms of efficiency of the HFO (by not calculating the full Hessian Matrix) are obvious.

It is important to note that instead of the $Hu$ product, the $Gu$ product is used, where $G$ is the Gauss-Newton Matrix (an approximation of the Hessian Matrix) [50]. It might look pointless to use an approximation instead of the actual matrix, however, the Gauss-Newton matrix bypasses possible problems that can occur with the use of the Hessian,

which could make it completely ineffective during the training process. Even for the cases where these problems do not appear, the $G$ matrix provides better results, in terms of search directions, which lead to lower memory consumption (about half) and higher running speeds compared to the $H$ matrix.

### 2.3.2 Analysis of HFO

A detailed analysis of how HFO works was described by Charalambous [23] for anyone interested into diving deeper into this variation of Newton's method. As it was mentioned before, when the $H$ matrix is used, some issues can occur. One of the most important problems is the lack of the utilization of the CG algorithm, on a quadratic model with a non-positive definite curvature matrix, since the Hessian matrix in some cases is non-positive definite. To deal with this issue, the Gauss-Newton matrix is used, which is guaranteed to always be positive semi-definite and is an approximation of the Hessian matrix. Except from that, the Gauss-Newton matrix usually outperforms the Hessian matrix in terms of efficiency.

### 2.3.3 Hessian-Vector Multiplication evaluation

As mentioned in previous sections, instead of an explicit evaluation of the Hessian matrix, dot products with the Hessian and arbitrary vectors are performed in HFO, which computationally cost the same with a gradient calculation. If the Hessian is considered as the Jacobian matrix of the gradient, based on the definition of directional derivatives, the $H(w)v$ product is the directional derivative of the gradient $\nabla f(w)$ in the direction $v$ (Equation 2.14).

$$H(w)v = \lim_{\varepsilon \to 0} \frac{\nabla f(w + \varepsilon v) - \nabla f(w)}{\varepsilon} \tag{2.14}$$

In practice, finite-differences suffer from numerical errors, which are troublesome for training ANNs. To counter this issue, a method called 'Forward Differentiation' was proposed [51] and was adapted for ANN training [52]. The main idea was to repeat the chain rule for the value of each node of the gradient, and in order to do that an $R_v(x)$ operator was defined to denote the directional derivative of $x$ in the direction $v$.

$$\mathrm{R}_v X = \lim_{\varepsilon \to 0} \frac{X(\mathrm{w} + \varepsilon v) - X(\theta)}{\varepsilon} = \frac{\partial X}{\partial \mathrm{w}} v \tag{2.15}$$

The R operator is a derivative operator, so it obeys the usual rules of differentiation (2.16):

$$
\begin{aligned}
\mathrm{R}_v(X+Y) &= \mathrm{R}_v X + \mathrm{R}_v Y && \text{linearity} \\
\mathrm{R}_v(XY) &= (\mathrm{R}_v X)Y + X\mathrm{R}_v Y && \text{product rule} \\
\mathrm{R}_v(h(X)) &= (\mathrm{R}_v X)h'(X) && \text{chain rule}
\end{aligned}
\tag{2.16}
$$

If these rules are repeated recursively in the gradient calculation algorithm the $Hv$ product will be computed. The algorithm for a simple gradient evaluation is illustrated in algorithm 4 (where $L(y_l;t_l)$ is one of the loss functions of table 2.4), while the algorithm 5 (where $L(y_l;t_l)$ is one of the loss functions of table 2.4) shows the modified version, where the differentiation rules are used to calculate the $H(w)v$ product. The algorithm 6, illustrates how the $G(w)v$ product is calculated and obviously it is simpler than algorithm 5 [49].

**input:** $y_0$; $\theta$ mapped to $(W_1, \ldots, W_{\ell-1}, b_1, \ldots, b_{\ell-1})$.
**for all** $i$ **from** 0 **to** $\ell - 1$ **do**
  $x_{i+1} \leftarrow W_i y_i + b_i$
  $y_{i+1} \leftarrow s_{i+1}(x_{i+1})$
**end for**
$dy_\ell \leftarrow \partial L(y_\ell; t_\ell)/\partial y_\ell$ ($t_\ell$ is the target)
**for all** $i$ **from** $\ell - 1$ **downto** 0 **do**
  $dx_{i+1} \leftarrow dy_{i+1} s'_{i+1}(x_{i+1})$
  $dW_i \leftarrow dx_{i+1} y_i^\top$
  $db_i \leftarrow dx_{i+1}$
  $dy_i \leftarrow W_i^\top dx_{i+1}$
**end for**
**output:** $\nabla f(\theta)$ as mapped from $(dW_1, \ldots, dW_{\ell-1}, db_1, \ldots, db_{\ell-1})$.

**Algorithm 4:** Algorithm for computing the gradient of a FFNN [49].

**input:** $v$ mapped to $(\mathrm{R}W_1, \ldots, \mathrm{R}W_{\ell-1}, \mathrm{R}b_1, \ldots, \mathrm{R}b_{\ell-1})$
$\mathrm{R}y_0 \leftarrow 0$ (since $y_0$ is not a function of the parameters)
**for all** $i$ **from** 0 **to** $\ell - 1$ **do**
  $\mathrm{R}x_{i+1} \leftarrow \mathrm{R}W_i y_i + W_i \mathrm{R}y_i + \mathrm{R}b_i$ (product rule)
  $\mathrm{R}y_{i+1} \leftarrow \mathrm{R}x_{i+1} s'_{i+1}(x_{i+1})$ (chain rule)
**end for**
$\mathrm{R}dy_\ell \leftarrow \mathrm{R}\left(\frac{\partial L(y_\ell; t_\ell)}{\partial y_\ell}\right) = \frac{\partial\{\partial L(y_\ell; t_\ell)/\partial y_\ell\}}{\partial y_\ell}\mathrm{R}y_\ell = \frac{\partial^2 L(y_\ell; t_\ell)}{\partial y_\ell^2}\mathrm{R}y_\ell$
**for all** $i$ **from** $\ell - 1$ **downto** 0 **do**
  $\mathrm{R}dx_{i+1} \leftarrow \mathrm{R}dy_{i+1} s'_{i+1}(x_{i+1}) + dy_{i+1} R\left\{s'_{i+1}(x_{i+1})\right\}$ (product rule)
  $= dy_{i+1} s''_{i+1}(x_{i+1}) \mathrm{R}x_{i+1}$ (chain rule)
  $\mathrm{R}dW_i \leftarrow \mathrm{R}dx_{i+1} y_i^\top + dx_{i+1} \mathrm{R}y_i^\top$ (product rule)
  $\mathrm{R}db_i \leftarrow \mathrm{R}dy_i$
  $\mathrm{R}dy_i \leftarrow \mathrm{R}W_i^\top dx_{i+1} + W_i^\top \mathrm{R}dx_{i+1}$ (product rule)
**end for**
**output:** $H(\mathrm{w})v$ as mapped from $(\mathrm{R}dW_1, \ldots, \mathrm{R}dW_{\ell-1}, \mathrm{R}db_1, \ldots, Rdb_{\ell-1})$.

**Algorithm 5:** Algorithm for computing the H(w)v product in a FFNN [49].

**input:** $\mathrm{R}W_1, \ldots, \mathrm{R}W_{\ell-1}, \mathrm{R}b_1, \ldots, \mathrm{R}b_{\ell-1}$.
$\mathrm{R}y_0 \leftarrow 0$ ($y_0$ is not a function of the parameters)
**for all** $i$ **from** $1$ **to** $\ell - 1$ **do**
  $\mathrm{R}x_{i+1} \leftarrow \mathrm{R}W_i y_i + W_i \mathrm{R}y_i + \mathrm{R}b_i$ (product rule)
  $\mathrm{R}y_{i+1} \leftarrow \mathrm{R}x_{i+1} s'_{i+1}(x_{i+1})$
**end for**
$\mathrm{R}dy_\ell \leftarrow \dfrac{\partial^2 L(y_\ell; t_\ell)}{\partial y_\ell^2} \mathrm{R}y_\ell$
**for all** $i$ **from** $\ell - 1$ **downto** $1$ **do**
  $\mathrm{R}dx_{i+1} \leftarrow \mathrm{R}dy_{i+1} s'_{i+1}(x_{i+1})$
  $\mathrm{R}dW_i \leftarrow \mathrm{R}dx_{i+1} y_i^\top$
  $\mathrm{R}db_i \leftarrow \mathrm{R}dx_{i+1}$
  $\mathrm{R}dy_i \leftarrow \mathrm{R}W_i^\top dx_{i+1}$
**end for**
**output:** $(\mathrm{R}dW_1, \ldots, \mathrm{R}dW_{\ell-1}, \mathrm{R}b_1, \ldots, \mathrm{R}b_{\ell-1})$.

**Algorithm 6:** Algorithm for computing the G(w)v product in a FFNN [49].

| Name | $L(z;t)$ | $\nabla L(z;t)$ | $L''(z;t)$ |
|---|---|---|---|
| Squared error | $\frac{1}{2}\lVert p - t \rVert^2$ | $-(p-t)$ | $I$ |
| Cross-entropy error | $-t \log p - (1-t)\log(1-p)$ | $-(p-t)$ | $\mathrm{diag}(p(1-p))$ |
| Cross-entropy error (multi-dim) | $-\sum_i [t]_i \log[p]_i$ | $-(p-t)$ | $\mathrm{diag}(p) - pp^\top$ |

Table 2.4: Derivatives and Hessians of typical loss function.

# Chapter 3

# Data Manipulation

## 3.1 PSSP Metrics

The Protein Secondary Structure Prediction (PSSP) problem concentrates on predicting, as accurately as possible, the secondary structure of proteins based on their primary structure. In this thesis project supervised learning methods are utilised, which require both the input data (primary structure) and output data (secondary structure) to train an Artificial Neural Network (ANN) to make predictions. Supervised learning is like learning with a teacher. The input is presented to the ANN which attempts to predict the output and then receives feedback on whether its predictions were correct or not. This way the network can adjust the weights accordingly to improve the prediction results. Both primary and secondary structure data must be encoded in a way that can be fed into the network.

To get an indication of how good are the predictions of the trained models two metrics were used, the per residue Q3 accuracy and the Segment Overlap (SOV), which are commonly used for the PSSP problem. The Q3 accuracy measures the number of correctly classified amino acids divided by the number of total amino acids (Equation 3.1, where $n$ is the number of amino acid residues and $m_i$ takes the value of 1 if the predicted value of the $i^{th}$ amino acid residue is correct and 0 otherwise). The Segment Overlap (SOV) [53] score is used to measure how good are the predicted results for each class and the general structure of the entire protein. More specifically, unlike Q3, SOV considers the size of continuous overlapping segments and assigns extra allowance to longer continuous overlapping segments (instead of just checking the individual positions, like Q3).

$$Q = 100\frac{1}{n}\sum_{i=1}^{n} m_i \tag{3.1}$$

For instance, if the target secondary structure of a protein consists of four (4) helices followed by two (2) coils and then another four (4) helices and the prediction has only ten (10) helices, Q3 and SOV will produce different accuracy values. The Q3 accuracy will be 80%, as eight of the ten amino acids were predicted correctly, while the SOV score would be just 48. Even though the original SOV score was not a percentage, a modified definition of SOV [54] was suggested, which fixed this issue with normalization techniques.

$$SOV_\alpha = \frac{1}{N_\alpha}\sum_{S_\alpha}\frac{\min OV\left(s_1,s_2\right)+\delta\left(s_1,s_2\right)}{\max OV\left(s_1,s_2\right)} \tag{3.2}$$

The SOV score for the $\alpha$-helix can be calculated with equation 3.2, where $s_1$ and $s_2$ are the actual and predicted segments of the secondary structure of the $\alpha$-helices, respectively.

The $s_a$ is the number of segment pairs $(s_1, s_2)$, where $s_1$ and $s_2$ have at least one common residue $\alpha$-helix. The minOV$(s_1, s_2)$ is the length of the overlap between $s_1$ and $s_2$, and the maxOV$(s_1, s_2)$ is the length of the total area for which one of the $s_1$ and $s_2$ has one residue of type $\alpha$-helix. The $N_\alpha$ is the total number of residues of type $\alpha$-helix. The calculation of $\delta(s_1, s_2)$ is based on equation 3.3.

$$\delta(s_1, s_2) = \min \left\{ \begin{array}{c} \text{maxOV}(s_1, s_2) - \min OV(s_1, s_2) \\ \min OV(s_1, s_2) \\ int(0.5 \times \text{len}(s_1)) \\ int(0.5 \times \text{len}(s_2)) \end{array} \right\} \tag{3.3}$$

## 3.2 Protein Databases and DSSP

There are several protein databases, like the iProClass (Protein Information Resource), PDBe (Protein Data Bank in Europe), PDBj (Protein Data Bank in Japan) and RCSB (RCSB Protein Data Bank), which include various information about millions of proteins. This information includes protein names, length, structures (primary, secondary, tertiary and quaternary) and other biological information related to proteins. The protein information included in the datasets of the PSSP problem was extracted from these databases.

| Secondary Structure | 8 class code | 3 class code |
|---|---|---|
| **α-helix** | H | H |
| **3-helix** | G | |
| **π-helix** | I | |
| **β-strand** | E | E |
| **β-bridge** | B | |
| **β-turn** | T | C |
| **bend** | S | |
| **Random coil** | C | |

Table 3.1: Table with the secondary structure abbreviations, grouped in 8 and 3 classes

The Dictionary of Secondary Structure of Proteins (DSSP) [55] defined a standardized format for categorising the secondary structures of proteins. According to this format, there are eight (8) distinct classes of secondary structures, based their shape, which are represented by a capital letter of the English alphabet. These are the $\alpha - helix$ (H), 3-helix

(G), $\pi$-helix (I), $\beta$-strand (E), $\beta$-bridge (B), $\beta$-turn (T), bend (S), and random coil (C) for residues which are not in any of the other conformations (Table 3.1). Usually these eight (8) categories are grouped into three (3) more general categories, which describe the shape of a specific local segment of the protein. For the purpose of this dissertation, the 3-class classification is used, which includes the helix (H) conformations, containing the first three categories (H, G, I), the sheet (E) conformations, containing the next two categories (E, B), and the Coil (C) conformations, containing the rest categories (T, S, C).

## 3.3 Dataset Format

The protein datasets, that were used for training, had records of a 3-line format per protein. An example is shown in figure 3.1, where the first line contains the protein name, the second line the primary structure and the third line the secondary structure of the protein. The protein name can be used to combine the primary and secondary structures with the Multiple Sequence Alignment (MSA) [8] profiles. The primary structure corresponds to the sequence of amino acids of each protein and each letter represents one amino acid. The secondary structure, located in the third line, is the target output which must be predicted by the network, and each letter represents the class of each amino acid (based on table 3.1).

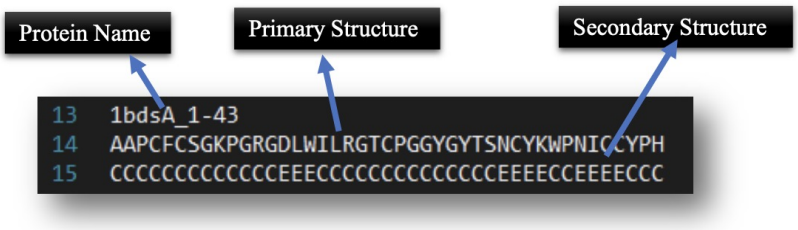



Figure 3.1: Protein representation example for protein 1bdsA_1-43.

To prepare training and validation datasets, a python program was developed, which creates files with comma-separated values (CSV files) based on the input datasets (in the form of figure 3.1) and MSA profiles, which will be discussed in the following sections. This program gives the ability to process multiple datasets at once by including the names of the datasets in the 'datasets' variable, located at the top section of the program. Moreover, if the MSA profiles for some proteins are missing, the program will ignore these proteins and print their names on the screen. This program for CB513 can be found in appendix D and for PISCES can be found in appendix E.

## 3.4 Data Encoding and MSA profiles

It is suggested the input and output data, that is used to train ANNs and most machine learning algorithms, to be normalised before they are used in training. The new encoded data should consist of real values between zero and one (0,1), or between minus one and one (-1,1), according to the selected activation function. The reason behind this suggestion is to speed up the learning process and help the network reach convergence faster.

For the PSSP problem a suggested encoding method is to use the Multiple Sequence Alignment (MSA) profiles along with the protein datasets (that include protein names, primary and secondary structures). These MSA profiles, in general, include information about about DNA and RNA protein sequences, and are very popular in the field of Bioinformatics. In many cases, the proteins that are selected to create the MSA profile have an evolutionary relationship with each other and are descended from a common ancestor. Because of that, these proteins are presumed to have the same secondary and tertiary structure [8]. The amino acids of these proteins are aligned together and are encoded in a way such that each position of their sequence represents the probability of the appearance of each amino acid, to form an MSA profile. An example of this alignment process is illustrated in figure 3.2.


Figure 3.2: Process of MSA profiling

However, the alignment of three or more biological sequences is extremely hard and time consuming when done manually. Because of that, computational algorithms have been developed to analyse and align these amino acid sequences. These algorithms use heuristics to find an approximation of the alignment, as the optimal alignment is computationally expensive. Furthermore, the MSA files that are created for each protein contain $N$ rows (where $N$ is the number of amino acids of the protein) and 20 columns, where each column represents the probability of each amino acid (from the 20 known types) appearing

in that specific position in the protein sequence.

In the example of figure 3.2, the highlighted column would have 89% (8/9) for the V amino acid and 11% (1/9) for the E amino acid, while all other amino acids would have zero (0) values. The values of amino acids for each line must add up to 100 and before they are used for training an ANN they must be normalised in the range (0,1), which can be easily done by dividing them with 100. Therefore, the ANN would be able to adapt to the structure of the data more easily.

## 3.5 CB513 and PISCES Datasets

In general, in order to successfully train a prediction model the datasets, which will be used for the training phase, must be preprocessed. In this phase data selections and data cleaning techniques are performed. There are various datasets for the PSSP problem, that have been created and preprocessed over the years. For the purpose of this dissertation, two widely used datasets were selected, the CB513 dataset [12] and the PISCES dataset [56]. These two datasets were chosen because they have been used for the PSSP problem by many researchers, which makes the comparison of the results possible and gives an indication of how well trained is the neural network. To prevent the network from memorising the order of the input patterns, a good practice is to shuffle the input patterns on each epoch, and therefore get better prediction results.

Initially, the smaller dataset was used, CB513, which has 513 unique proteins, from which eight (8) were excluded (these can be found in Appendix A), due to the fact that their MSA files included only zeros. This dataset required less time to train and helped to identify whether the neural network was able to learn how to predict the secondary structure of proteins or not. In the next phase, the bigger dataset was used, PISCES, which consists of about 8500 sequences, from which 341 were excluded because their MSA files were either corrupted or zeroed and another 16 were excluded due to missing MSA profiles (all of them are shown in Appendix B). The bigger dataset (PISCES) was utilized because in many machine learning problems, by feeding the model with more data, the predictions become more accurate.

Finally, the PISCES dataset was not in the expected form, that was mentioned before (protein name, primary structure, secondary structure) but fortunately a team of University of Cyprus implemented a Java program, which was able to convert the PISCES data into the expected format. These new modified PISCES datasets were provided by Dionysiou ([1], [24]), who also worked on the PSSP problem in the past.

## 3.6 Dataset preprocessing with MSA profiles

Convolutional Neural Networks (CNNs) expect their input as a two-dimension (2D) or three-dimension (3D) array, so in order to train a CNN to predict the secondary structure of proteins, the training and test data must be presented in the form of 2D or 3D arrays. The input representation method, which will be used, is the same with the one used by [24].

Over recent years, the Multiple Sequence Alignment (MSA) profiles approach was used by many researchers. In the Bioinformatics sector, the sequence alignment is a well known approach, that refers to sequences of DNA, RNA or proteins. In general, this method attempts to find similarities between these types of sequences, which can usually define some biological association, leading to a better understanding of the biological mechanisms. An example of an MSA file is illustrated in figure 3.3.

```
0 0 0 100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
29 3 15 0 9 0 0 0 25 3 2 1 0 0 1 7 4 0 1 0
0 0 0 0 0 0 0 0 1 4 1 10 0 2 3 28 3 43 3 2
37 5 58 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
12 4 4 0 8 0 45 0 0 0 0 14 0 0 3 2 0 7 0 0
3 3 31 11 0 0 0 30 0 0 0 15 0 0 7 0 0 0 0 0
2 0 11 7 10 13 47 0 0 0 0 1 0 0 6 2 1 0 0 0
10 1 2 2 0 0 0 1 1 17 23 17 1 0 2 3 5 3 0 11
1 1 2 4 0 0 0 10 1 6 13 34 0 0 1 15 0 9 0 2
5 4 3 1 1 0 1 4 8 6 10 16 0 1 4 3 3 12 6 13
```

Figure 3.3: Example of the encoded form of an MSA file [24].

Since the input data of a CNN must consist of 2D (or 3D) arrays, these MSA files must be visualised in a way so that they can be used to successfully train a CNN. In order to achieve that, all the MSA files were combined in a single file and the desired output label was added at the end of each record.

For instance, the information included in two MSA files, named '1bdoa_77-156' and '1bfga_19-144', before the two files were combined together, is shown in figure 3.4. Figure 3.5 shows the new encoded file (where all spaces were replaced with commas ','), after combining the two MSA files, where the red line separates the data of the first file from the data of the second file. Each line now has twenty one (21) numbers (columns) instead of twenty (20), as the predicted class was added (C:0, E:1, H:2) at the end of each

line, according to the dataset files (with the protein name, primary and secondary structures), described in section 3.3. This means that both PSSP datasets and MSA profiles were utilized to create the new dataset files.

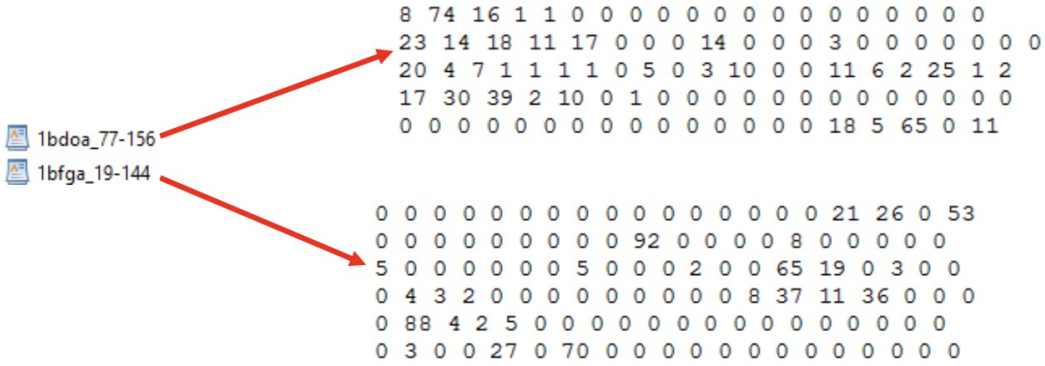



Figure 3.4: The image shows the MSA file (before collapsing into a single file) [24].

By using the above technique, a new file was created, which included all the data from the MSA profiles and the desired labels of the secondary structure for each record (Figure 3.5). The same method was used for both training and test datasets. This new representation with the MSA profiles can be used to successfully train a CNN, since these files can be presented in a 2D table format. The CNN will receive as input one record (one line) of the new dataset at a time and will attempt to predict the secondary structure, using the output class representation mentioned earlier (C:0, E:1, H:2).

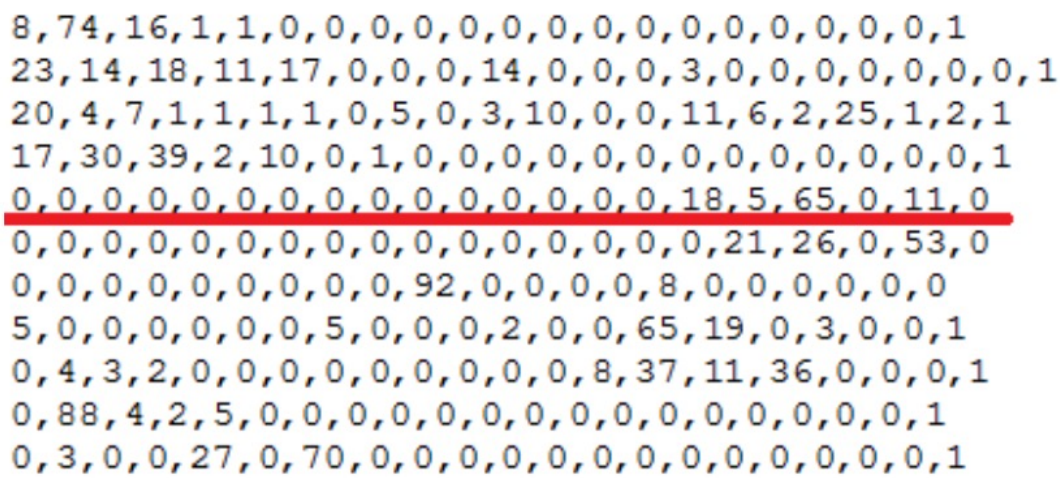



Figure 3.5: The encoding of the new file, after combining the MSA files into a single file [24].

## 3.7 Significant neighboring amino acids

The sequence of amino acids plays a major role, as it determines the interactions that take place and the folds that are formed in the secondary structure of a protein. The secondary structure of an amino acid is significantly affected by the adjacent amino acids (previous and next amino acids), according to the distance between them (short distance usually means bigger impact, while longer distance means less impact) [20].

The training and test datasets were modified to take advantage of this interaction between the neighboring amino acids. For each record, except from the information about each amino acid (from the MSA profiles) and the expected output class, described in the previous section, the information of k-neighboring amino acids was added (where k is an integer variable). For example, if k is equal to one, each record will consist of the MSA records of the left amino acid, the MSA records of the current amino acid, the MSA records of the right amino acid and the target label (class) of the current amino acid (which is located in the middle). If an amino acid is not preceded (first amino acid in a sequence) or followed (last amino acid in a sequence) by another amino acid, zero values are added instead (zero padding) to ensure that all records have the same length. An example where k is one (1) is illustrated in figure 3.6, for a sequence of six (6) amino acids.

| No. | MSA record | Class |
|---|---|---|
| 1 | 0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0,0, | 0 |
| 2 | 0,0,0,0,0,0,0,21,49,0,0,31,0,0,0,0,0,0,0,0, | 1 |
| 3 | 42,0,0,0,28,0,0,0,31,0,0,0,0,0,0,0,0,0,0,0, | 1 |
| 4 | 0,0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0, | 0 |
| 5 | 0,0,0,0,0,0,0,0,0,0,49,51,0,0,0,0,0,0,0,0, | 1 |
| 6 | 100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, | 1 |

Figure 3.6: MSA record for a sequence of 6 amino acids.

The new modified dataset has 61 $(20 \times 3 + 1)$ numbers for each record, as shown in figure 3.7.

| No. | New MSA record | | | Class |
|---|---|---|---|---|
| 1 | 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, | 0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0,0, | 0,0,0,0,0,0,0,21,49,0,0,31,0,0,0,0,0,0,0,0, | 0 |
| 2 | 0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0,0, | 0,0,0,0,0,0,0,21,49,0,0,31,0,0,0,0,0,0,0,0, | 42,0,0,0,28,0,0,0,31,0,0,0,0,0,0,0,0,0,0,0, | 1 |
| 3 | 0,0,0,0,0,0,0,21,49,0,0,31,0,0,0,0,0,0,0,0, | 42,0,0,0,28,0,0,0,31,0,0,0,0,0,0,0,0,0,0,0, | 0,0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0, | 1 |
| 4 | 42,0,0,0,28,0,0,0,31,0,0,0,0,0,0,0,0,0,0,0, | 0,0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0, | 0,0,0,0,0,0,0,0,0,0,49,51,0,0,0,0,0,0,0,0, | 0 |
| 5 | 0,0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0, | 0,0,0,0,0,0,0,0,0,0,49,51,0,0,0,0,0,0,0,0, | 100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, | 1 |
| 6 | 0,0,0,0,0,0,0,0,0,0,49,51,0,0,0,0,0,0,0,0, | 100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, | 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, | 1 |

Figure 3.7: Modified MSA record for a sequence of 6 amino acids.

As soon as this new representation is fed into the CNN it is rearranged into a 2D array, where each line includes the MSA profile vector for each amino acid and the last value will be the target label (class). Figure 3.8 illustrates an example for the data representation method after it was rearranged into a 2D array, for a window size of 15 amino acids (or 'plus7'), where each row represents the vector of the MSA profile for the specific amino acid and the SS label represents the class (H, E, C) of the middle amino acid.

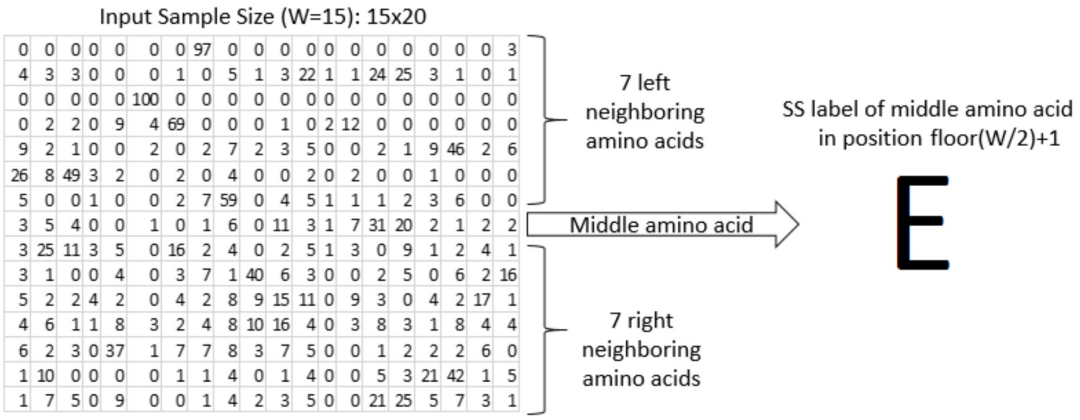



Figure 3.8: An example of input data representation for a window size of 15 (or k = 7) amino acids [57].

In order to create these modified datasets two Python programs were developed, that use the CB513 (Appendix D) and PISCES (Appendix E) datasets along with the MSA profiles to prepare the modified datasets, according to the provided 'plus' value, which can take any positive integer value (>0). In the previous example the plus value was one (1), as the neighboring amino acids were one for each side (one left and one right) of each amino acid. These programs prepare multiple datasets (or folds) at once and print on the screen the names of proteins for which the MSA files were not found.

This technique can improve the accuracy of a CNN [24], as the CNN can identify the neighboring amino acids, which can affect the secondary structure of an amino acid.

## 3.8 Training/ Testing Set and Cross Validation

To train an ANN a specific set of data is required, called training dataset, which is used for training the model so that it can extract features from the input patterns and classify these patterns into a number of classes. However, it is very important to ensure that the model was able to generalize the extracted knowledge so that it can predict patterns that has not 'seen' before. For this reason, another set of data is used, called test dataset,

which is completely different from the training dataset, and its purpose is to measure the effectiveness of the network to classify new data, that has never seen before. In general, a good rule of thumb is to split the entire dataset into 80% for the training dataset and 20% for the test dataset (80-20 rule). However, in different problems, other splitting criteria can be used, which may lead to better results.

Sometimes this method is not enough to test the ability of a network to predict new data, since the accuracy depends on a specific test dataset. A method that can be used to address this issue is to evenly split the data into N folds and train N different models. Each model will have a unique fold selected as the test dataset and the rest N-1 folds will be used as the training dataset. This method is called N-fold cross validation (Figure 3.9) and the cross validation accuracy is equal to the average test accuracy of all models.


Figure 3.9: 10-fold cross validation

## 3.9 Ensembles

Ensemble learning is a method which can be used to improve the performance of a machine learning model. According to this method, instead of training just a single model, multiple models can be trained and then their results can be combined somehow, to improve the final results.

Even though, there are various ensemble methods, ranging from simple to advanced and more sophisticated methods, for the purpose of this dissertation a relatively basic approach was used, also known as averaging ensemble method. This method basically calculates the average of the outputs of its models. In the PSSP problem, for instance, if there are

five different trained models the following steps are applied. First, for each of the input-output pairs, the output of each of the five models is calculated and classified in one of the three classes (H, E, C). Then, the results of each of the five models are compared with the method 'winner takes all' and the class that had the most appearances is chosen as the final class for that input. If there is a tie between some of the classes, an arbitrary class (from those) is selected as the final class.

This ensemble method, even if it is very simple, it can remove random errors from the models, which can lead to improved results. More advanced ensemble methods might have a bigger impact on the predicted results, but at the cost of computation resources, as these are usually more complex.

## 3.10 Filtering

### 3.10.1 External Rules

Post-processing filtering is an additional method that is used to improve the accuracy of a model. The applied filtering method can be problem specific or more generic, with the use of different learning algorithms. Both methods were used in this dissertation, which affected, by a small amount, the final accuracy (Q3 Score) and the quality of the predictions (SOV score).

The first filtering method was based on a set of external rules, that are specific for the PSSP problem. These rules are based on empirical observations and were used to 'fix' the quality of the results (SOV), rather than improving the overall accuracy (Q3).

The external rules applied are (where H, E and C are the three possible classes):

1. Single 'H' or 'E' are replaced with 'C'
2. Sequence 'HEEH' is replaced with 'HHHH'
3. Sequence 'HEH' is replaced with 'HHH'
4. Sequence '!HH!' is replaced with '!CC!'

These simple rules can be applied extremely fast and can increase the SOV score, while sometimes can slightly drop the Q3 accuracy.

### 3.10.2 Support Vector Machines

In 1995, Cortes and Vapnik suggested the use of Support Vector Machines (SVMs), in Machine Learning. Initially SVMs were used for binary classification problems and their purpose was to find hyperplanes that best divide a dataset into classes [58]. If the data cannot be separate linearly, SVMs attempt to map the data into a higher dimension using a non-linear kernel function. These kernel functions are very effective and efficient as they just compute inner products. This transformation to a higher dimension is more likely to make the data linearly separable. Figure 3.10 illustrates four popular SVMs that are currently used, along with their kernels.

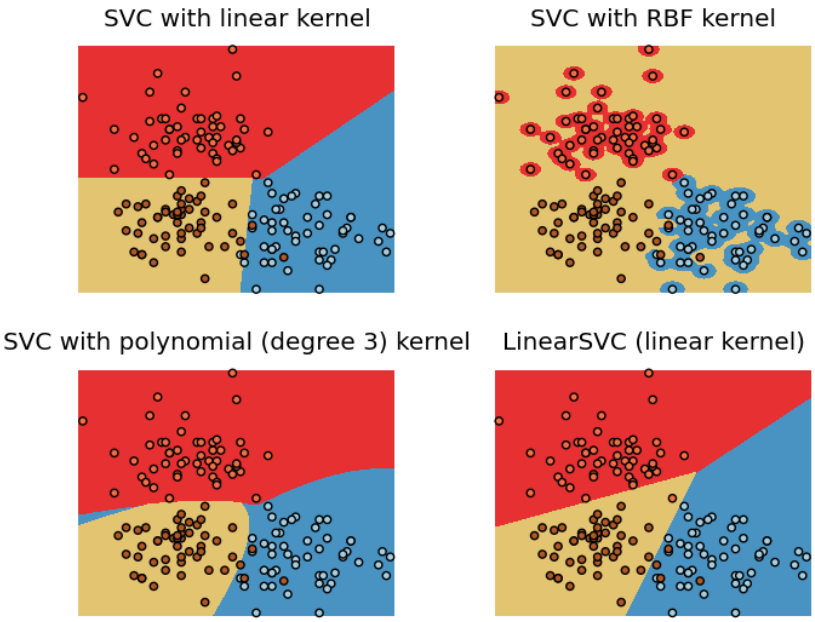



Figure 3.10: Results of different kernels for a 3-class classification problem.

In order to best separate two linearly separable classes by finding the optimal hyperplane, SVMs attempt to maximize the distance between the points, that are closer to the hyperplane, for each different class. The points that are located near the limits of this separation are called support vectors and the points that are located in the area of multiple classes (overlapping classes) are not taken into consideration, in order to create a more generalized model. Figure 3.11 shows three possible separation lines (A, B and C), where the blue star and red circle that are connected with them are considered the support vectors and line C is considered the optimal hyperplane. In figure 3.12, even though the separation of the initial data (left plot) is hard, if they are projected in a higher dimension they

can be easily separated by a hyperplane (right plot).

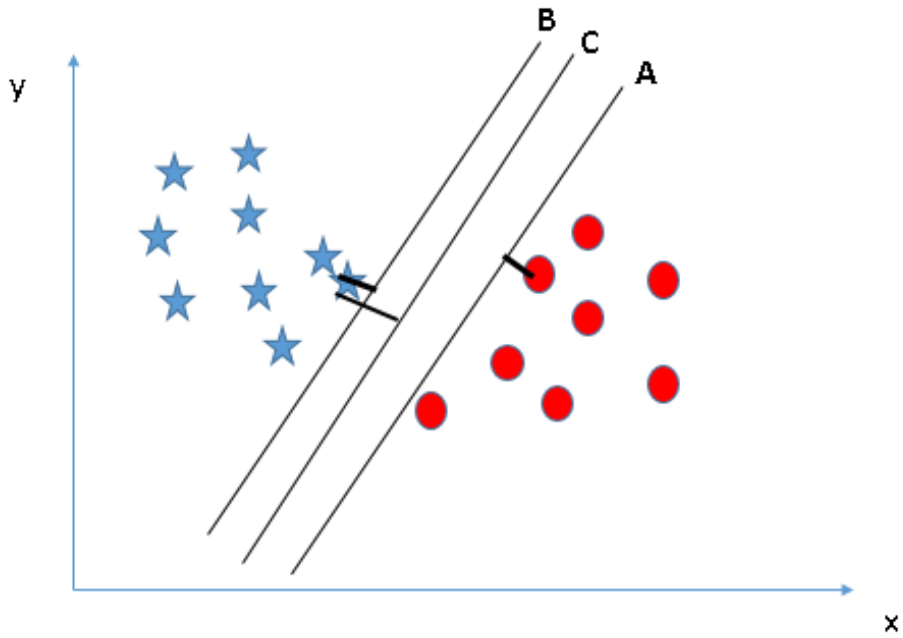



Figure 3.11: SVM example of a linearly separable problem.

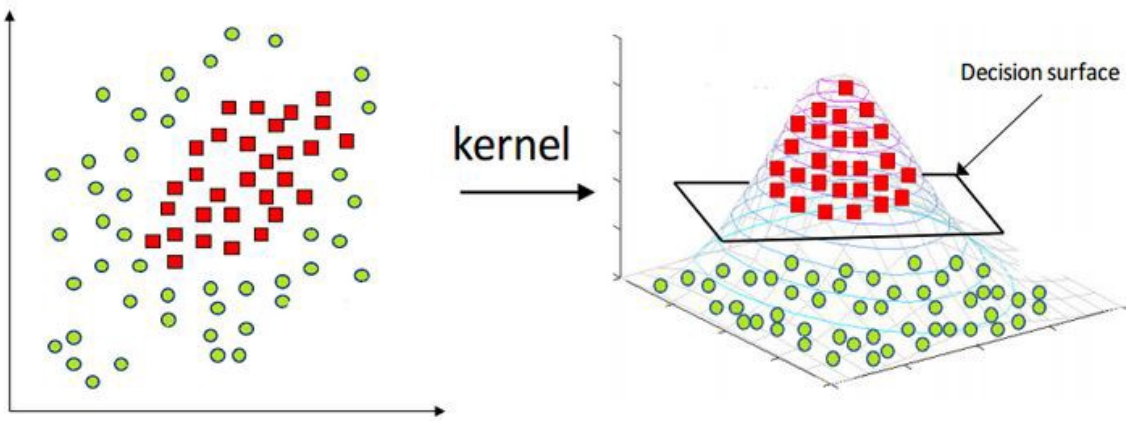



Figure 3.12: SVM projecting a problem in a higher dimension.

In particular, SVMs had very good filtering results for the PSSP problem [59]. More specifically, they were used by Dionysiou [24] and Dionysiou et al. [57] and their good final results makes them very promising.

### 3.10.3 Decision Trees

Decision trees are most commonly know for their use in operations research, and more specifically in decision analysis, but are also a popular tool for machine learning. They can be used to identify a strategy that is most likely to reach a target goal. A decision tree is defined as a support tool, with a tree-like shape, which models decisions and possible consequences, including resource costs, chance events outcomes and utility.

The best way to explain how a decision tree operates is through a simple example. A scenario, where a dataset contains numbers with different features, is illustrate in figure 3.13. There are two 1s and five 0s, which represent the two classes. The goal is to separate the data using their features, which are color (red or blue) and whether the number is underlined or not.

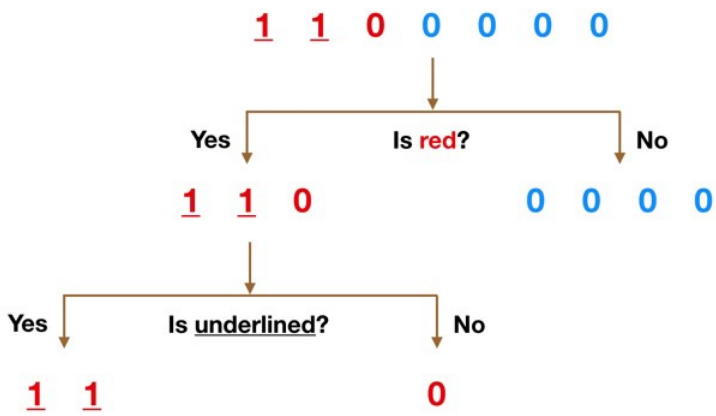



Figure 3.13: Example of simple decision tree [60].

Obviously, the color feature can be used to split the data, as only one of the 0s is red, while the rest are blue. The question 'Is it red?' can be used to split the first node. A node in a tree is like a point where the path splits into two branches, where the data that meet the criteria go under the 'Yes' branch and ones that do not go under the 'No' branch, as shown in figure 3.13. The 'No' branch contains only blue 0s that are not underlined, which means no further splits should be made. On the other hand, the 'Yes' branch contains data that have different features, so the question 'Is it underlined?' can be used to split the red data. The two underlined 1s go under the 'Yes' subbranch, while the not-underlined 0 goes under the 'No' subbranch. At this point no further splits of the data are required.

Even though, in real life examples the data will not be as clean as the one used in this example, the applied logic of a decision tree remains the same. A decision tree will decide at each node which feature can split the observations into two groups in a way that

the differences are maximised, while maximising the similarities between the members of each subgroup.

### 3.10.4 Random Forests

A random forest is a classification algorithm that consists of a large number of individual decision trees that function as an ensemble. Each individual tree, outputs a class prediction and the class with the most votes is selected as the prediction of the random forest (Figure 3.14).

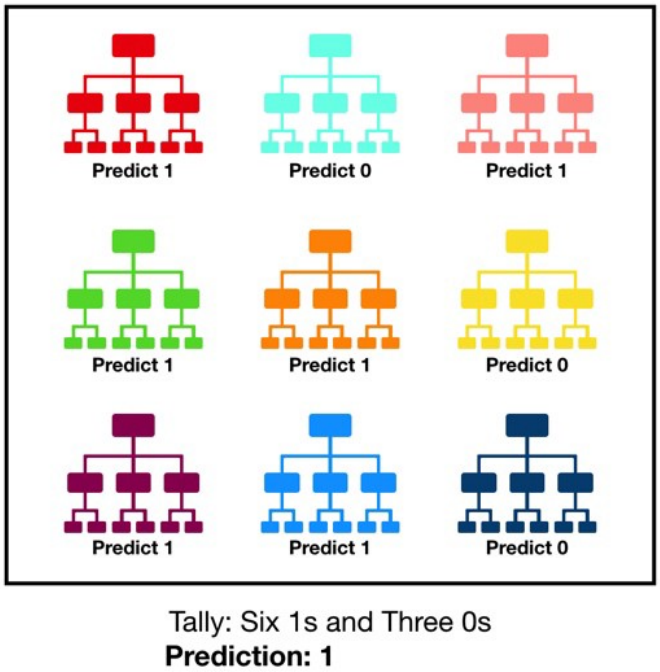



Figure 3.14: Example of random forest prediction [60].

The reason, a random forest model works so well, in data science, is because a large number of unrelated models, that operate as a group, can outperform any of the individual models. One of the most important things in a random forest is the low correlation between the individual models (trees), since the trees 'correct' each others' errors, as long as they do not make the same mistakes in the same direction. In order for a random forest to have good predictive results, there must be an actual signal that helps the models adapt to the features of the data and the correlation, between the predictions of the individual trees, must be as low as possible.

In order to illustrate why uncorrelated predictions are so important, a simple example will be used. In a gambling game a uniformly distributed random generator is used to produce a number between 1 and 100. If the number is above 40 the player wins and earns money

based on the bet amount, which means the player has 60% chance to win. The player has three options, play 100 games betting $1 per game (choice1), play 10 games betting $10 per game (choice2) or play 1 game and bet $100 (choice3). Below are the expected values for all three options:

$$ExpectedValue(choice1) = (0.60 * 1 + 0.40 * (-1)) * 100 = 20$$
$$ExpectedValue(choice2) = (0.60 * 10 + 0.40 * (-10)) * 10 = 20$$
$$ExpectedValue(choice3) = 0.60 * 100 + 0.40 * (-100) = 20$$

It is obvious that all options have the same expected value, which makes it difficult to choose. A visualization of a Monte Carlo simulation could reveal the distributions of the available options. Figure 3.15 illustrates the distribution of the outcome of 10000 simulations for each of the three options. The three options, even though they share the same expected value, they have completely different outcome distributions. With the first option (choice1) there is 97% chance to make money, while for the other two options (choice2 and choice3) the chance to make money is 63% and 60%, respectively. It seems that the more the $100 bet is split up, the higher the chance for the player to make money, as each game does not dependent on the other games.

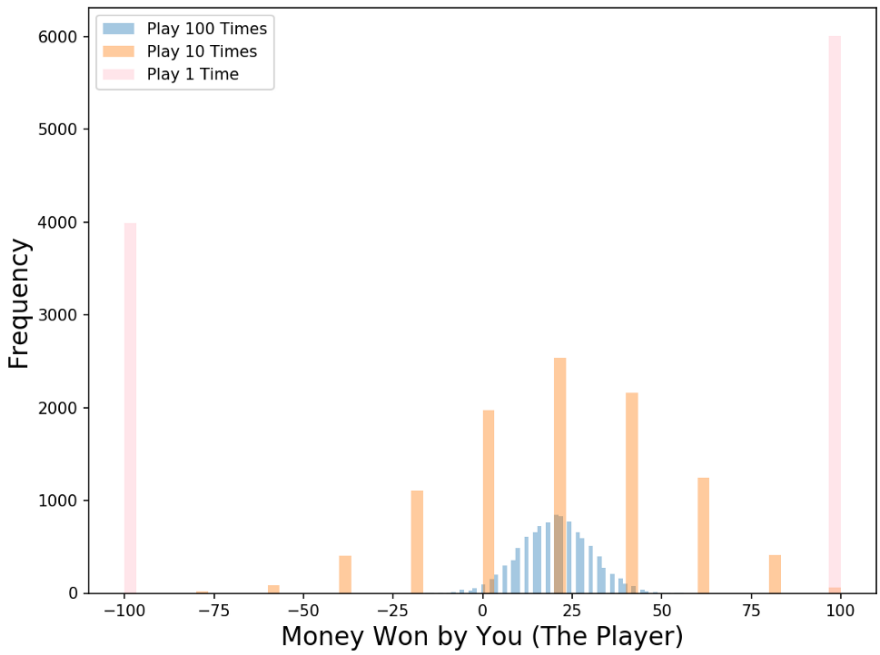



Figure 3.15: Distribution of the outcomes of 10000 simulations for each option [60].

A random forest works in the same way, with the game mentioned above. The higher the number of uncorrelated trees, the higher the chance of making correct predictions.

To ensure that each individual tree is uncorrelated with the other trees, a random forest uses two methods, bagging and feature randomness. The first one (bagging), randomly selects a sample from the dataset for each individual decision tree, instead of using the entire dataset. The second method (feature randomness), restricts the number of features that can be used to split a node in each decision tree, by selecting a random subset of the available features. This increases the variation between the individual trees of the model, which results in lower correlation.

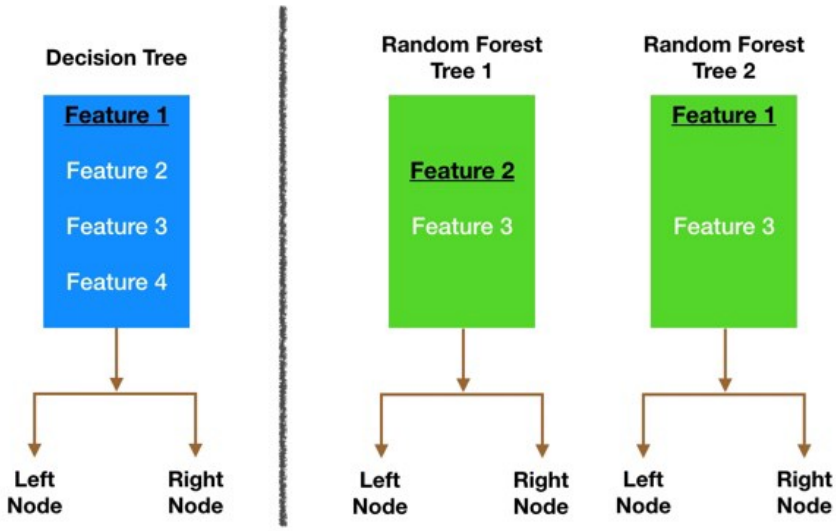



Figure 3.16: Node splitting in a decision tree and a random forest model [60].

In order to make things easier to understand, an example will be illustrated. Figure 3.16 shows a decision tree (blue) and two trees from a random forest (green), where both models can separate the data based on four features. The decision tree chose the Feature 1 to best separate the data into groups. The first tree of the random forest (Tree 1) could only choose between Features 2 and 3, which were selected randomly, to split the data, while the second tree (Tree 2) could only choose between Features 1 and 3. Even though Feature 1 was the best splitting option, only Tree 2 could use it, since it was not included in the available features of Tree 1.

To sum up, bagging helps to create trees that are trained on different sets of data, while feature randomness forces them to use different features to make decisions.

# Chapter 4

# Implementation

## 4.1 A new approach for the PSSP problem

The PSSP problem can be considered a classification problem, which means that an ANN can be used to predict the secondary structure of proteins. For this dissertation, a Convolutional Neural Network (CNN) was selected because, according to previous attempts [1], it managed to produce very good results (>80%), which makes it very promising.

A thesis dissertation should explore new methods or combinations, in order to provide some value to the world of research. Because of that, the use of just a CNN for the PSSP problem would be a poor choice as it has already been used before and it would not help the research community or the researchers that are involved with the PSSP problem. The main idea was to combine a CNN with the Hessian Free Optimisation (HFO) algorithm (a second order optimiser) to predict the secondary structure of proteins, which has never been attempted before, mainly because of the complexity of second order optimisation algorithms. This optimiser, combined with a simple Feed Forward Neural Network (FFNN), managed to achieve great results for the PSSP problem [2], with more than 80% Q3 accuracy.

Usually CNNs, because of their complexity, contain thousands of parameters and that makes the training process very time consuming on a Central Processing Unit (CPU). For small datasets, like CB513, a few hours (around 4-12 hours depending on the selected settings) would be enough. However, for bigger datasets, like PISCES, the training process could take days to complete. To speed up the training process a Graphics Processing Unit (GPU) was utilized, from the Google's Colab cloud service. Colab is a free Jupyter notebook environment that runs entirely in the cloud, does not require any setup and supports many popular machine learning libraries (paid services are also available). A Jupyter Notebook, also known as the IPython Notebook, is an interactive computational environment based on the web (usually ending with the extension '.ipynb') [61]. This allows users to combine code, comments, graphical visualizations and multimedia, in an interactive document, which can be run via a web browser, hosted on a local machine or even a remote server.

The classification model of this dissertation was implemented on a notebook to ensure portability, remove machine constraints (requirements), as this can be run entirely in Colab even with an 'old' machine using just a web browser. In addition to that, a notebook makes it easier to interact with the program and it comes with some of the most popular machine learning libraries and frameworks, like TensorFlow [62], PyTorch [63] and Scikit-learn [64], which are pre-installed and ready to use.

## 4.2 CNN and HFO combination

The purpose of this dissertation was to combine a CNN with HFO and train it to predict the secondary structure of proteins. The PyTorch machine learning framework [63] was initially used, along with the fastai library, to implement a CNN and train it for the PSSP problem. This part was successful and the results were around 70% Q3 accuracy, without tuning the hyper parameters. The next step was to implement the HFO algorithm in PyTorch, since the available implementation was written in pure Python and could not interact with the CNN. An alternative option was to implement the CNN from scratch (in pure Python), which would significantly drop the efficiency as the PyTorch framework (and most machine learning frameworks) has its functions written in C++, which is much more efficient.

The implementation of HFO was probably the most difficult task of this dissertation. The HFO implementation which was already available was very complex, which made this task even harder. After many failed attempts, the HFO algorithm managed to train a Feed Forward Neural Network (FFNN) to predict the XOR gate (a toy problem used check if a network is learning effectively). The next step was to try the new implementation on the PSSP, and fortunately the FFNN with HFO managed to extract some patterns from the proteins which resulted in around 72% Q3 accuracy (without any tuning of the hyper parameters). The final step of the implementation was to combine the two sections, the CNN and the FFNN with the HFO optimiser.

Unfortunately, this was not as simple as it seemed at the beginning of this dissertation. The HFO algorithm was specifically designed for a FFNN and not a CNN, which made the updates totally ineffective. An alternative approach could be to use a different optimisation algorithm, like gradient descent, to train the CNN layers and then train the FFNN at the end with the HFO. However, this practice seems pointless, since the purpose of HFO is to replace a different optimizer, not depend on it. Another approach could be to ignore the CNN layers and train only the FFNN with HFO, which would be a waste of resources, since the effectiveness of the CNN layers would not be utilized.

At this point, an ordinary dissertation project would possibly come to an end, as the main purpose was to attempt to combine the HFO algorithm with a CNN. The conclusion was that this was not possible because the HFO was designed explicitly for a FFNN and not a CNN. However, this is not an ordinary dissertation, so despite the tight margins of available time, an alternative approach was pursued with the help of additional research. This seemed to be a great decision as a recent article [3], published in January 2020,

explained why the HFO algorithm is not compatible with CNNs and suggested a variation of HFO, specifically designed for CNNs. This new method, called Subsampled Hessian Newton (SHN) method [3], is discussed in the following section.

## 4.3 Subsampled Hessian Newton (SHN) Method

There are several studies on Newton methods for training deep ANNs ([65], [66], [67], [46], [68], [69], [70]), but almost all of them used fully connected FFNNs. The Newton methods are very complicated and that is possibly the main reason why CNNs have not been utilized in those studies. Apart from this article [3], there is no evidence, or published documents, that describe how the Newton methods can be applied in deep learning (CNNs) effectively. This made Gradient Descent, and its variations, the most popular optimisation algorithms for CNNs, although the Newton's methods are more robust, more efficient and require less tuning of the hyper parameters (for FFNNs at least).

A new variation of HFO was suggested by Wang et al. (2020) for CNNs which is explained in high detail [3] (mathematical proof included). This new method was labeled as the Subsampled Hessian Newton (SHN) method (Algorithm 7, where (35) is Equation 4.1, (36) is Equation 4.2 and (37) is Equation 4.3). Due to the high complexity of the proof and extensive explanation of this algorithm, it is better to refer to the original paper for a better understanding of the transition from the HFO to the SHN algorithm.

```
Given initial θ. Calculate f(θ);
while ∇f(θ) ≠ 0 do
    Choose a set S ⊂ {1, ..., l};
    Compute ∇f(θ) and the needed information for Gauss Newton matrix-vector products;
    Approximately solve the linear system in (36) by CG to obtain a direction d;
    α = 1;
    while true do
        Compute f(θ + αd);
        if (35) is satisfied then
            break;
        end
        α ← α/2;
    end
    Update λ based on (37);
    θ ← θ + αd;
end
```

**Algorithm 7:** A subsampled Hessian Newton method for CNNs [3].

$$f(\theta + \alpha d) \leq f(\theta) + \eta \alpha \nabla f(\theta)^T d \tag{4.1}$$

$$(G+\lambda I)d = -\nabla f(\theta) \tag{4.2}$$

$$\lambda_{\text{next}} = \begin{cases} \lambda \times \text{ drop} & \rho > \rho_{\text{upper}} \\ \lambda & \rho_{\text{lower}} \leq \rho \leq \rho_{\text{upper}} \\ \lambda \times \text{ boost} & \text{otherwise} \end{cases} \tag{4.3}$$

The memory consumption of the Newton method depends on the size of data, which makes it difficult to handle large datasets. To counter this issue, the SHN method uses a subset $S$ of the training data to obtain the subsampled Gauss-Newton matrix, which is used to approximate the Hessian matrix. This technique not only reduces the execution time per iteration (with a slightly less accurate direction) but also decreases the memory usage considerably.

For instance, at the mth convolutional layer for the Gauss-Newton matrix-vector products only the following matrices must be stored:

$$\frac{\partial z^{L+1,i}}{\partial \operatorname{vec}(S^{m,i})^T}, \forall i \in S \tag{4.4}$$

For the gradient evaluations and the activation function the whole training data is required, so the independent results over all instances for each mini-batch must be summed. If the index set {1, ..., N} of data is split to R equal-sized subsets $S_1, \ldots, S_R$ and the result for each subset is calculated, then to find the final output all the subset results must be accumulated. The utilization of subsets can effectively decrease the memory consumption (Wang et al., 2020, section 3.5 and section 5) [3].

| | Types of Neural Networks |
|---|---|
| LeCun et al. [15] | Fully-connected |
| Martens [20] | Autoencoder |
| Martens and Sutskever [21] | Fully-connected, Recurrent |
| Kiros [9] | Fully-connected, Autoencoder |
| Wang et al. [29, 30] | Fully-connected |
| Botev et al. [1] | Autoencoder |

Table 4.1: Previous studies on Newton methods [3].

Table 4.1 illustrates some of the previous studies on Newton methods, performed on different types of ANNs. Other studies investigated the use of second-order optimization

methods for training CNNs, however, those are different from the Newton method considered in this dissertation.

## 4.4 Network Implementation

For the purpose of this dissertation, the implementation of a Convolutional Neural Network (CNN) with the Subsampled Hessian Newton (SHN) method was used, which was implemented in Python by Wang et al. [3] and can be found here [https://github.com/cjlin1/simpleNN]. Many optimisation tricks were applied to reduce memory consumption and to improve efficiency, which are discussed in that paper. The Python implementation used the Tensorflow [62] machine learning framework and is slightly different from the one used in [3], which was implemented in Matlab.

The initial implementation, which the paper [3] used for the experiments, used Matlab. Consequently, the input datasets used a matlab format (.mat), which was transferred to the Python version. The input files must contain a 'y' variable (of size $N \times 1$), which includes all the labels of the target class, and a 'Z' variable (of size $N \times M$), which includes all the features. Since the datasets were already preprocessed with a specific format which could be easily adapted to the matlab format, a script was implemented which was responsible to convert the text files (.txt) to matlab files (.mat) (Appendix C). This script can be found at [https://gitlab.com/perf.ai/pssp_project/-/blob/master/datasets2mat.sh].

In addition to the above, the implementation was modified so that it could be executed in a Jupyter notebook [61] and the datasets were uploaded to a public Gitlab repository, to be easily accessible. All the necessary scripts, programs, data files and instructions were uploaded in that repository, which can be found here [https://gitlab.com/perf.ai/pssp_project/-/tree/master].

Further modifications were made to the Python implementation to adapt it to the PSSP problem and improve the results, as the initial version was not very effective for this particular problem. For all experiments of this dissertation a free Colab machine was utilized (to use one visit [https://colab.research.google.com/notebooks/welcome.ipynb]). For information according TensorFlow visit [https://www.tensorflow.org/] and for PyTorch visit [https://pytorch.org/]. As regards the fastai library, which can be found here [https://www.fast.ai/], a very informative course is available at [https://course.fast.ai/part2], that describes how to create more advanced neural networks.

# Chapter 5

# Experiments and Results

## 5.1 Experiments for Implementation Evaluation

Many experiments have been performed, in order to find the optimal hyper parameters for the Convolutional Neural Network (CNN) and the Subsampled Hessian Newton (SHM) optimiser. The initial implementation [3] was already tested on some well-known benchmarks, like MNIST and CIFAR10 problems, which proved that the network was able to learn effectively. Because of that, the experiments of this dissertation were focused more on the PSSP problem. Initially, the model was trained with the CB513 dataset, which is relatively small, to identify the best hyper parameters and then additional experiments were performed on the bigger dataset, PISCES.

To ensure that each trained model has the best possible accuracy, during the training process after each iteration the model (all the weights) with the highest test accuracy was saved to an output file. This file can be then loaded to predict the test data and display the Q3 accuracy. This practice ensures that the model does not overfit to the training data and is able to predict new, never seen before data samples.

Figure 5.1 shows the test loss after each iteration for a CNN model trained with fold 5 of CB513. The red line illustrates the test loss after each iteration, while the green line illustrates the test loss of the saved model. The test loss after iteration 5, fluctuated within a narrow margin of about 0.05, while the test loss of the saved model followed a downward trend until it reached a plateau. Figure 5.2 displays the test accuracy for the same model for the first 35 iterations. The red line represents the test accuracy after each iteration, while the green line represents the test accuracy of the saved model. According to the line graph (Figure 5.2), the test accuracy dropped slightly in iteration 20, while the test accuracy of the saved model remained the same. This proves that at any iteration the saved model has the best possible test accuracy, which does not drop throughout the entire training process. In addition to that, the two line graphs (Figure 5.1 and 5.2) confirm that the model is able to train effectively and manages to converge in about twenty iterations.

For each experiment the following steps were performed. First the global parameters for the datasets were set (plus_var: the number of neighboring amino acids added, ds_num: the fold number of the dataset, dataset: 'CB513' or 'PISCES' to choose between the two PSSP datasets). Then, the appropriate dataset was retrieved from the Gitlab repository and all necessary functions were loaded. In the next phase, the hyper parameters were selected to prepare the model for the training phase. As soon as the training process was finished, a new file was created, which had all weights of the model with the best test accuracy. This file was used in the final step, in which the saved model was loaded and

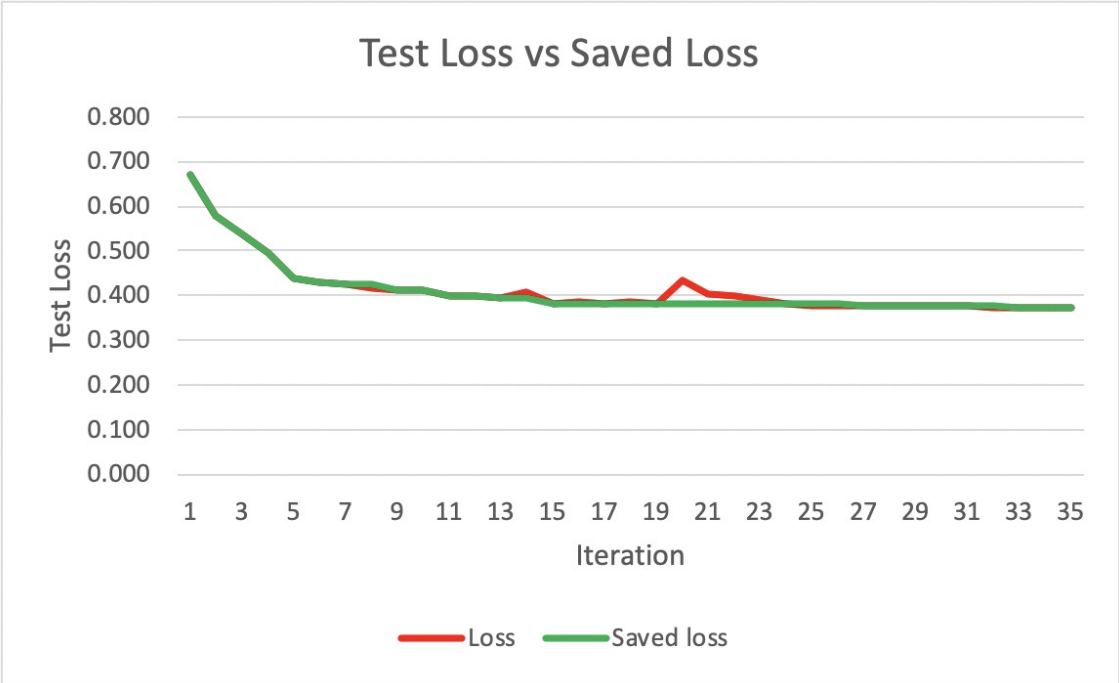



Figure 5.1: The test loss for each iteration compared to the test loss of the saved model.

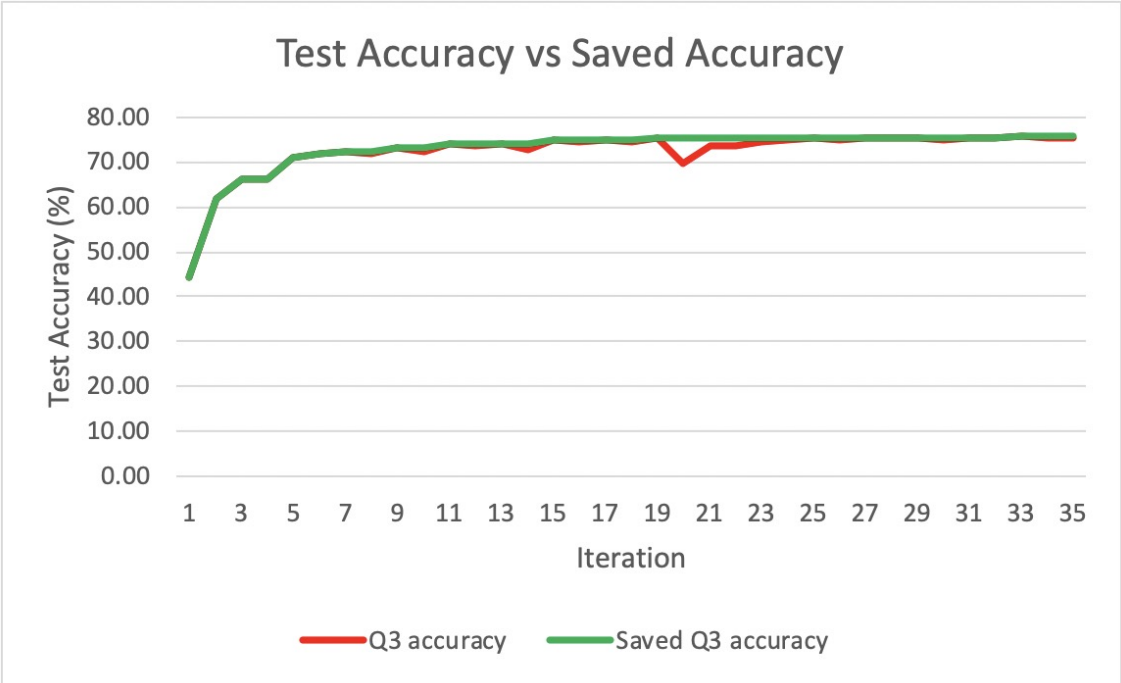



Figure 5.2: The test Q3 accuracy after each iteration compared to the test Q3 accuracy of the saved model.

was used to create two files with the predicted secondary structures of the proteins for the test and train datasets, respectively.

In order to check the efficiency of the Subsampled Hessian Newton (SHN) method, the Gradient Descent algorithm could be used to train CNN models, with the same structure as the one used for the SHN method. This would make it possible to compare the two optimisation algorithms, both in terms of accuracy (for the PSSP problem) and fine tuning of their hyper parameters. The Stochastic Gradient Descent (SGD) algorithm is already implemented and can be selected, as an alternative optimisation method.

## 5.2 Experiments with CB513 dataset

### 5.2.1 Fine Tuning of Hyper Parameters

In order to find the best hyper parameters for the network, experiments must be performed were each time only one hyper parameter is altered (the rest remain the same). Table 5.1 illustrates the hyper parameters used for the CNN, where the last layer has only three (3) neurons (one for each possible class). For each combination of hyper parameters five different models were trained and the average Q3 accuracy was saved in an excel file. For hyper parameter tuning, fold 5 was selected (for all the experiments), because it was observed that models trained with this fold performed very poorly, compared to the other folds. The motivation behind this was to maximize the performance of the hardest-to-learn fold with the hope that this would increase the overall Q3 accuracy and SOV score of the cross-validation.

| | Type | Kernel size | Number of Filters | Activation Function |
|---|---|---|---|---|
| CNN Layer 0 | Convolutional Hidden Layer | 3 x 3 | 64 | ReLU |
| CNN Layer 1 | Convolutional Hidden Layer | 3 x 3 | 64 | ReLU |
| CNN Layer 2 | Convolutional Hidden Layer | 3 x 3 | 64 | ReLU |
| CNN Layer 3 | Fully Connected MLP | - | - | SOFTMAX |

Table 5.1: Hyper parameters for CNN for all experiments.

First of all, the CB513 datasets were prepared with 'plus7' amino acids (seven left and seven right neighbouring amino acids were added, for each amino acid). This selection was based on [24], as this number (7) of neighbouring amino acids seemed very promising. The next step was to choose the number of samples used in the subsampled Gauss-Newton matrix (GNsize). Six different values were tested, as shown in table 5.2, while all

other parameters were selected randomly or based on the default values of the implementation. According to table 5.2, the best value for the GNsize was 2048 with approximately 75.54% Q3 accuracy.

| GNsize | C | CNN layers | bsize | Max Iterations | Dimensions | Q3 Accuracy |
|---|---|---|---|---|---|---|
| 50 | 0.01 | 4 | 8192 | 50 | 15 20 1 | 73.89% |
| 100 | 0.01 | 4 | 8192 | 50 | 15 20 1 | 75.20% |
| 200 | 0.01 | 4 | 8192 | 50 | 15 20 1 | 75.13% |
| 512 | 0.01 | 4 | 8192 | 50 | 15 20 1 | 75.25% |
| 1024 | 0.01 | 4 | 8192 | 50 | 15 20 1 | 75.40% |
| 2048 | 0.01 | 4 | 8192 | 50 | 15 20 1 | 75.54% |

Table 5.2: Q3 accuracy results for GNsize for fold 5 of CB513.

For the following experiments GNsize was selected to be equal to 2048 (GNsize = 2048). After that, the C value had to be determined so the same process was repeated but this time the C values were examined. Table 5.3 illustrates the Q3 accuracy results of the models in relation to the C value. It is obvious that the best option was 0.01 with 75.54% accuracy, so C was set to this value for the following experiments (C = 0.01).

| GNsize | C | CNN layers | bsize | Max Iterations | Dimensions | Q3 Accuracy |
|---|---|---|---|---|---|---|
| 2048 | 0.01 | 4 | 8192 | 50 | 15 20 1 | 75.54% |
| 2048 | 0.05 | 4 | 8192 | 50 | 15 20 1 | 75.36% |
| 2048 | 0.10 | 4 | 8192 | 50 | 15 20 1 | 75.02% |
| 2048 | 0.50 | 4 | 8192 | 50 | 15 20 1 | 75.32% |
| 2048 | 1.00 | 4 | 8192 | 50 | 15 20 1 | 75.29% |

Table 5.3: Tuning the C hyper parameter for fold 5 of CB513.

This process was replicated for the batch size (bsize), which usually is set based on the memory constraints. It is very important to note that if the model cannot begin the training process, it is probably because this value was set too high. In this case, lowering the bsize value can fix the issue. Table 5.4 shows the results for bsize, however, it is not clear which one is the best, as most of them are very close to each other. For the purpose of this dissertation, the value 12288 was selected as the batch size (bsize = 12288) to reduce the training time of the model.

| GNsize | C | CNN layers | bsize | Max Iterations | Dimensions | Q3 Accuracy |
|---|---|---|---|---|---|---|
| 2048 | 0.01 | 4 | 1024 | 50 | 15 20 1 | 75.56% |
| 2048 | 0.01 | 4 | 2048 | 50 | 15 20 1 | 75.39% |
| 2048 | 0.01 | 4 | 4096 | 50 | 15 20 1 | 75.40% |
| 2048 | 0.01 | 4 | 8192 | 50 | 15 20 1 | 75.54% |
| 2048 | 0.01 | 4 | 10240 | 50 | 15 20 1 | 75.57% |
| 2048 | 0.01 | 4 | 12288 | 50 | 15 20 1 | 75.64% |

Table 5.4: Tuning the batch size (bsize) hyper parameter for fold 5 of CB513.

### 5.2.2 10-fold Cross-Validation and Ensembles Results

In order to check whether the results of a model are good just for a specific test dataset or whether the trained network is a good prediction model, additional techniques must be utilized. One such technique is cross-validation, which was described in section 3.8. More specifically, a 10-fold cross-validation was used for the CB513 dataset to validate the model's ability to generalize.

Table 5.5 shows the hyper parameters for all the trained models, which were used for the cross-validation of CB513.

| GNsize | C | CNN layers | bsize | Max Iterations | Dimensions |
|---|---|---|---|---|---|
| 2048 | 0.01 | 4 | 12288 | 50 | 15 20 1 |

Table 5.5: Hyper parameters for trained models.

The cross-validation results for the CB513 dataset are shown in table 5.6. This table displays the overall Q3 accuracy and overall SOV score for the best trained model for each fold. In addition, the Q3 accuracy and SOV scores for each of the three classes (H, E, C) are shown separately, as well as the average results for all folds (cross-validation values).

According to table 5.6, the best trained model achieved 78.20% overall Q3 accuracy and 75.67 overall SOV score, while the cross-validation results were 77.25% and 72.91, respectively. Even though the optimisation for the hyper parameters was based on fold 5, which had the lowest Q3 accuracy as expected, the results for all the other folds were considerably better. It is obvious that most of the models had trouble identifying the class 'E' and that is why the QE accuracy for all folds are substantially lower than the QH and the QC accuracy. Most models were able to predict, to some extend, the class 'C', as the QC accuracy for the 10-fold cross-validation was approximately 82.13%.

| | Q3 | QH | QE | QC | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 78.20% | 80.56% | 68.94% | 81.71% | 75.67 | 81.18 | 72.21 | 72.32 |
| Fold1 | 76.25% | 78.82% | 64.59% | 81.38% | 73.02 | 70.56 | 72.02 | 74.00 |
| Fold2 | 77.85% | 81.19% | 65.77% | 81.15% | 73.26 | 75.80 | 69.85 | 70.43 |
| Fold3 | 77.85% | 80.75% | 66.33% | 81.70% | 74.33 | 73.70 | 67.38 | 72.28 |
| Fold4 | 77.97% | 80.44% | 65.36% | 82.36% | 73.38 | 74.11 | 68.25 | 70.52 |
| Fold5 | 75.77% | 79.45% | 61.01% | 81.05% | 71.60 | 70.57 | 65.69 | 71.05 |
| Fold6 | 77.91% | 75.81% | 65.82% | 85.59% | 74.42 | 74.04 | 70.42 | 74.58 |
| Fold7 | 76.74% | 75.78% | 67.67% | 82.37% | 68.37 | 69.53 | 72.39 | 68.62 |
| Fold8 | 76.82% | 77.10% | 69.57% | 80.82% | 72.61 | 67.49 | 73.48 | 72.57 |
| Fold9 | 77.13% | 79.63% | 62.46% | 83.15% | 72.43 | 80.30 | 66.98 | 72.42 |
| Average | 77.25% | 78.95% | 65.75% | 82.13% | 72.91 | 73.73 | 69.87 | 71.88 |

Table 5.6: Q3 and SOV results for 10-fold cross validation for the CB513 dataset.

Table 5.7 shows the results for the cross validation of the ensembles method, where five (5) CNNs were trained with the SHN method, using the CB513 dataset. For the experiments of this dissertation, multiple models were trained for each fold (about 7-10) and the five (5) models, which formed the best ensembles model, were selected for the final ensembles model of each fold.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 78.46 | 80.86 | 68.99 | 82.07 | 75.19 | 79.61 | 73.17 | 72.87 |
| Fold1 | 76.49 | 79.05 | 64.36 | 81.90 | 72.76 | 71.39 | 72.17 | 74.38 |
| Fold2 | 78.19 | 81.23 | 66.77 | 81.41 | 75.03 | 76.87 | 70.88 | 71.48 |
| Fold3 | 78.15 | 80.87 | 66.63 | 82.16 | 74.70 | 74.46 | 67.75 | 72.54 |
| Fold4 | 78.15 | 80.92 | 65.60 | 82.26 | 74.95 | 77.38 | 66.96 | 71.64 |
| Fold5 | 75.97 | 77.96 | 62.59 | 81.75 | 71.24 | 69.26 | 65.35 | 70.96 |
| Fold6 | 77.91 | 75.86 | 66.77 | 85.08 | 74.25 | 74.23 | 71.83 | 73.94 |
| Fold7 | 76.94 | 76.11 | 68.00 | 82.40 | 69.53 | 69.98 | 73.40 | 68.12 |
| Fold8 | 77.03 | 77.33 | 68.55 | 81.71 | 73.72 | 70.17 | 72.32 | 72.77 |
| Fold9 | 77.29 | 79.63 | 65.24 | 81.98 | 74.12 | 82.34 | 68.70 | 72.90 |
| Average | 77.46 | 78.98 | 66.35 | 82.27 | 73.55 | 74.57 | 70.25 | 72.16 |

Table 5.7: Q3 and SOV results for ensembles (with 5 experiments per fold) cross validation for the CB513 dataset.

Usually, the ensembles method is more effective when there is high variance between the trained models, because each trained model explores a different space of the dataset and it learns how to predict based on different features. That is why, it is better to combine results from different machine learning models. The combination of these models can create a new model with more accurate predictions than any of the separate models, at the cost of time and processing power, but this is not guaranteed.

In this case (Table 5.7), the new ensemble model managed to outperform every single CNN model for all folds (the best CNN model for each fold is shown in table 5.6). The increase in Q3 accuracy was relatively small, probably because all models were trained with the same hyper parameters, which resulted in less variance. The boost in accuracy

could be better if models with different hyper parameters or different types of models were used for creating the ensembles model. For instance, a Convolutional Neural Network, a Recurrent Neural Network, a Bidirectional Recurrent Neural Network, a Feed Forward Neural Network and a Long Short-Term Memory model could be trained and then combined with the ensembles method. This combination could have a greater impact on the accuracy of the new ensembles model.

### 5.2.3 CNN and SVM Combination

As mentioned in section 3.10.2, the final attempt to improve the results was to use Support Vector Machines (SVMs), which managed to improve the results of past PSSP studies [59]. More specifically, an SVM will be used to filter the output data from the CNN, with the ambition that the Q3 accuracy and SOV score could be improved.

In order to train the SVM, a window (of odd size) will be used to extract information from the prediction file created by the CNN, which will be used as the input features of SVM, while the expected output will be the secondary structure of the middle amino acid. A python program (prepare_SVM_files.py) was used to prepare the datasets for the SVM based on the given window size (Appendix K). The SVM was also implemented as a python program (train_SVM.py) (using Scikit-learn machine learning library [64]), which exploits the output datasets from 'prepare_SVM_files.py' to train an SVM and create a new output file with the new filtered predictions (Appendix L). Both programs can be found in this Gitlab repository [https://gitlab.com/perf.ai/pssp_project/-/tree/master].

For the experiments of this dissertation both networks will be utilized and the final results will be compared with the results of a standalone CNN. The CNN will use as input the data described in section 3.6, while the SVM will take as input the output of the CNN, in order to filter the results. The table 5.8 shows the hyper parameters used for all the experiments.

| Kernel | C | Decision Function Shape | Degree | Shrinking | Tol | Gamma |
|---|---|---|---|---|---|---|
| RBF (Radial Basis Function) | 10 | ovr | 3 | TRUE | 0.001 | 0.1 |

Table 5.8: Hyper parameters for SVM filtering.

The same technique could be used with almost any other classification model. In this dissertation, except from SVM filtering, Decision Trees and Random Forests [71] (which are basically ensembles of decision trees) were used, as alternative filtering methods, in combination with external rules, which were explained in section 3.10.1. The parameters

used for the random forest filter are shown in table 5.9. For the decision tree filter, the only non-default parameter used was the maximum depth (max_depth) parameter which was set to twenty (20).

| n_estimators | max_depth | random_state | min_samples_split | min_samples_leaf |
|---|---|---|---|---|
| 100 | 25 | 42 | 2 | 1 |

Table 5.9: Hyper parameters for Random Forest filtering.

### 5.2.4 Filtering Results for CB513

The chosen window size for the filtering methods, for the CB513 experiments, was thirteen (13), because it produced relatively good filtering results without a major impact on the total filtering time.

Table 5.10 shows the results for the Q3 accuracy and SOV score after applying the external rules to the ensembles model. It seems that the Q3 accuracy increased only by a tiny amount, while the SOV score rose by 1.33, which is relatively good.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 78.83 | 79.47 | 68.27 | 84.41 | 77.35 | 77.96 | 72.07 | 75.74 |
| Fold1 | 76.22 | 77.30 | 62.99 | 83.41 | 72.57 | 70.78 | 71.79 | 72.44 |
| Fold2 | 78.17 | 79.83 | 65.77 | 83.18 | 76.68 | 76.98 | 71.88 | 72.92 |
| Fold3 | 78.26 | 80.06 | 65.79 | 83.57 | 75.34 | 73.92 | 67.42 | 72.66 |
| Fold4 | 78.07 | 80.04 | 63.89 | 83.73 | 74.78 | 77.09 | 65.58 | 70.69 |
| Fold5 | 75.89 | 76.64 | 61.06 | 83.38 | 72.42 | 68.98 | 65.97 | 70.56 |
| Fold6 | 78.07 | 74.93 | 65.52 | 86.74 | 76.35 | 76.02 | 71.33 | 75.46 |
| Fold7 | 76.97 | 75.10 | 66.99 | 83.75 | 70.87 | 70.16 | 72.37 | 68.07 |
| Fold8 | 77.46 | 76.46 | 67.76 | 83.98 | 76.72 | 73.16 | 72.52 | 73.06 |
| Fold9 | 77.30 | 78.88 | 63.79 | 83.47 | 75.69 | 84.14 | 68.71 | 73.19 |
| Average | 77.52 | 77.87 | 65.18 | 83.96 | 74.88 | 74.92 | 69.96 | 72.48 |

Table 5.10: Q3 accuracy and SOV score for ensembles (with 5 executions per fold) and external rules filtering for CB513 dataset.

Table 5.11 illustrates the Q3 accuracy and SOV score for the ensembles model after applying external rules and SVM filtering. The external rules filtering usually offers a significant boost in the overall SOV score and sometimes a slight drop in the Q3 accuracy.

The order in which the filters are applied can produce different results, so the same filters could be applied in various ways (different orders). For this purpose, a bash script was created, which applies the filtering methods in various orders and creates an output file with all the results (Appendix M). The results for the ensembles model with SVM filtering are shown in table 5.12.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 79.24 | 80.52 | 72.51 | 82.13 | 76.80 | 77.09 | 74.23 | 73.38 |
| Fold1 | 76.75 | 79.00 | 73.33 | 77.14 | 72.87 | 72.04 | 75.10 | 69.87 |
| Fold2 | 78.43 | 81.84 | 77.56 | 75.79 | 77.24 | 77.12 | 72.87 | 71.79 |
| Fold3 | 78.56 | 82.40 | 72.19 | 78.78 | 74.17 | 72.84 | 70.41 | 69.30 |
| Fold4 | 78.42 | 82.39 | 76.89 | 75.78 | 74.91 | 77.74 | 69.21 | 68.31 |
| Fold5 | 76.38 | 77.75 | 71.02 | 78.26 | 72.31 | 70.23 | 68.51 | 68.56 |
| Fold6 | 78.05 | 77.63 | 76.11 | 79.34 | 77.53 | 76.56 | 74.23 | 74.05 |
| Fold7 | 77.44 | 77.07 | 78.18 | 77.31 | 69.65 | 69.72 | 73.08 | 67.02 |
| Fold8 | 77.16 | 77.60 | 76.62 | 77.09 | 75.49 | 70.53 | 75.06 | 69.80 |
| Fold9 | 77.42 | 80.71 | 73.23 | 76.91 | 76.07 | 85.19 | 73.08 | 70.92 |
| Average | 77.79 | 79.69 | 74.76 | 77.85 | 74.70 | 74.91 | 72.58 | 70.30 |

Table 5.11: Q3 accuracy and SOV score for ensembles (with 5 executions per fold), external rules and SVM filtering for CB513 dataset.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 80.44 | 84.00 | 74.47 | 81.14 | 75.95 | 82.45 | 76.98 | 71.19 |
| Fold1 | 79.00 | 83.48 | 75.44 | 77.79 | 75.55 | 77.70 | 78.09 | 72.51 |
| Fold2 | 80.03 | 84.21 | 80.86 | 75.79 | 78.29 | 77.47 | 74.29 | 73.83 |
| Fold3 | 79.44 | 83.10 | 74.22 | 79.17 | 75.61 | 74.67 | 73.21 | 71.32 |
| Fold4 | 79.19 | 84.23 | 79.34 | 74.76 | 75.98 | 78.05 | 73.74 | 68.63 |
| Fold5 | 77.84 | 80.86 | 74.82 | 77.25 | 73.15 | 69.51 | 70.32 | 70.94 |
| Fold6 | 79.57 | 79.37 | 77.88 | 80.57 | 77.43 | 75.99 | 76.77 | 73.39 |
| Fold7 | 78.72 | 79.17 | 80.67 | 77.35 | 71.46 | 72.70 | 75.24 | 68.51 |
| Fold8 | 79.42 | 79.11 | 76.23 | 81.54 | 77.19 | 71.45 | 77.70 | 73.76 |
| Fold9 | 79.01 | 82.50 | 76.59 | 77.33 | 77.31 | 85.96 | 75.42 | 72.35 |
| Average | 79.27 | 82.00 | 77.05 | 78.27 | 75.79 | 76.60 | 75.18 | 71.64 |

Table 5.12: Q3 accuracy and SOV score for ensembles and SVM filtering for CB513 dataset.

Table 5.13 illustrates the Q3 accuracy and SOV score for the ensembles model with SVM and external rules filtering. According to tables 5.10 and 5.11, the SVM filtering improved the overall Q3 accuracy by a small amount, but decreased the overall SOV score slightly. The impact of SVM filtering was significant, for both overall Q3 accuracy and overall SOV score (Tables 5.7 - before, and 5.12 - after), with 79.27% and 75.79, respectively. It seems that if SVM filtering is applied before the external rules, the cross validation results are substantially better, with approximately 1.75% increase in overall Q3 accuracy and about 1.65 growth in overall SOV score.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 80.55 | 83.59 | 73.18 | 82.45 | 78.55 | 82.86 | 76.21 | 74.03 |
| Fold1 | 78.89 | 82.56 | 74.36 | 78.90 | 75.88 | 76.96 | 76.14 | 71.84 |
| Fold2 | 80.06 | 83.46 | 80.17 | 76.90 | 78.63 | 77.80 | 73.64 | 73.68 |
| Fold3 | 79.36 | 82.37 | 72.61 | 80.49 | 76.15 | 74.00 | 70.71 | 72.27 |
| Fold4 | 79.51 | 83.93 | 78.66 | 76.13 | 77.79 | 78.19 | 73.33 | 70.94 |
| Fold5 | 77.75 | 80.24 | 73.46 | 78.23 | 74.11 | 71.05 | 69.18 | 70.31 |
| Fold6 | 79.95 | 78.86 | 76.70 | 82.39 | 78.83 | 78.00 | 76.35 | 73.88 |
| Fold7 | 78.60 | 78.33 | 80.10 | 77.99 | 72.13 | 72.23 | 74.79 | 68.43 |
| Fold8 | 79.22 | 78.39 | 75.10 | 82.33 | 77.93 | 70.94 | 75.55 | 73.30 |
| Fold9 | 79.09 | 82.23 | 75.84 | 78.17 | 78.01 | 86.06 | 74.59 | 73.46 |
| Average | 79.30 | 81.40 | 76.02 | 79.40 | 76.80 | 76.81 | 74.05 | 72.21 |

Table 5.13: Q3 accuracy and SOV score for ensembles, SVM and external rules filtering for CB513 dataset.

The results for the ensembles model with the external rules and decision tree filtering are shown in table 5.14. Table 5.15 illustrates the Q3 accuracy and SOV score for the ensembles model with decision tree filtering, while table 5.16 displays the results for the ensembles model with decision tree and external rules filtering. The decision tree filtering improved the results significantly (Tables 5.7 - before, and 5.15 - after), reaching 81.69% overall Q3 accuracy and 75.93 overall SOV score. According to tables 5.14 and 5.16, when the decision tree filtering is applied before the external rules, the prediction results of the model are considerably better. More specifically, there is an increase of about 2.27% in the overall Q3 accuracy and approximately 5.11 in the overall SOV score.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 80.18 | 80.77 | 78.16 | 80.88 | 76.08 | 75.57 | 74.92 | 72.28 |
| Fold1 | 78.50 | 79.88 | 74.59 | 79.83 | 72.27 | 72.92 | 76.51 | 69.75 |
| Fold2 | 78.90 | 81.09 | 82.54 | 75.00 | 75.95 | 74.95 | 75.64 | 71.06 |
| Fold3 | 79.92 | 82.02 | 75.30 | 80.65 | 73.81 | 70.73 | 73.27 | 70.12 |
| Fold4 | 79.46 | 81.65 | 78.36 | 78.13 | 73.09 | 75.81 | 72.17 | 67.85 |
| Fold5 | 78.26 | 78.96 | 75.10 | 79.45 | 73.23 | 69.87 | 72.26 | 69.85 |
| Fold6 | 79.45 | 77.89 | 78.00 | 81.30 | 75.40 | 72.63 | 75.64 | 73.74 |
| Fold7 | 79.14 | 78.00 | 78.69 | 80.23 | 68.38 | 71.51 | 73.84 | 66.06 |
| Fold8 | 79.08 | 77.10 | 78.32 | 81.25 | 74.59 | 68.77 | 74.50 | 71.09 |
| Fold9 | 78.58 | 80.82 | 76.77 | 77.66 | 74.43 | 80.86 | 74.58 | 70.19 |
| Average | 79.15 | 79.82 | 77.58 | 79.44 | 73.72 | 73.36 | 74.33 | 70.20 |

Table 5.14: Q3 accuracy and SOV score for ensembles, external rules and decision tree filtering for CB513 dataset.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 82.47 | 84.72 | 81.56 | 81.26 | 76.30 | 80.35 | 79.15 | 72.91 |
| Fold1 | 82.25 | 84.86 | 81.50 | 80.76 | 76.29 | 80.57 | 83.03 | 72.36 |
| Fold2 | 82.01 | 84.68 | 84.54 | 78.24 | 77.79 | 79.46 | 79.07 | 72.41 |
| Fold3 | 80.98 | 83.79 | 79.43 | 79.44 | 73.83 | 74.74 | 77.94 | 69.90 |
| Fold4 | 81.33 | 84.34 | 83.69 | 77.50 | 76.02 | 76.71 | 78.33 | 70.07 |
| Fold5 | 81.34 | 83.51 | 80.59 | 80.13 | 74.44 | 73.88 | 75.34 | 70.47 |
| Fold6 | 81.39 | 81.61 | 82.91 | 80.45 | 77.52 | 77.07 | 78.86 | 74.10 |
| Fold7 | 81.60 | 81.31 | 83.27 | 80.90 | 73.31 | 78.21 | 79.10 | 70.13 |
| Fold8 | 81.94 | 81.24 | 82.72 | 82.10 | 77.45 | 70.24 | 81.65 | 73.14 |
| Fold9 | 81.55 | 82.79 | 82.50 | 79.95 | 76.33 | 81.65 | 80.33 | 71.13 |
| Average | 81.69 | 83.29 | 82.27 | 80.07 | 75.93 | 77.29 | 79.28 | 71.66 |

Table 5.15: Q3 accuracy and SOV score for ensembles and decision tree filtering for CB513 dataset.

Table 5.17 shows the results for the ensembles model with external rules and random forest filtering. The Q3 accuracy and SOV score for the ensembles model with random forest filtering are displayed in table 5.18, while the results for the ensembles model with random forest and external rules filtering are presented in table 5.19. The boost of random forest filtering was great (Tables 5.7 - before, and 5.18 - after), since it increased the overall Q3 accuracy to 81.75% and the overall SOV score to 76.33. It is obvious that the results are better when the random forest filtering is applied before the external rules filtering (Tables 5.17 and 5.19).

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 82.67 | 84.00 | 79.33 | 83.57 | 80.17 | 81.83 | 78.42 | 76.82 |
| Fold1 | 82.43 | 83.62 | 79.61 | 83.24 | 78.58 | 81.72 | 77.98 | 74.73 |
| Fold2 | 82.18 | 83.42 | 82.29 | 80.99 | 80.58 | 79.29 | 75.54 | 75.87 |
| Fold3 | 81.12 | 82.48 | 77.03 | 82.19 | 77.39 | 76.58 | 74.07 | 73.36 |
| Fold4 | 81.69 | 83.79 | 81.48 | 79.97 | 78.24 | 78.87 | 75.89 | 71.63 |
| Fold5 | 81.44 | 82.35 | 78.78 | 82.21 | 76.76 | 75.13 | 73.45 | 72.30 |
| Fold6 | 81.84 | 80.93 | 81.19 | 82.81 | 80.92 | 79.78 | 79.62 | 76.50 |
| Fold7 | 81.57 | 80.18 | 81.40 | 82.68 | 75.84 | 78.85 | 78.15 | 72.00 |
| Fold8 | 81.92 | 79.68 | 81.31 | 84.21 | 80.85 | 73.83 | 79.49 | 76.04 |
| Fold9 | 81.71 | 81.53 | 81.05 | 82.24 | 79.00 | 82.81 | 79.48 | 74.14 |
| Average | 81.86 | 82.20 | 80.35 | 82.41 | 78.83 | 78.87 | 77.21 | 74.34 |

Table 5.16: Q3 accuracy and SOV score for ensembles, decision tree and external rules filtering for CB513 dataset.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 80.00 | 81.91 | 79.66 | 78.73 | 76.32 | 76.51 | 76.54 | 70.72 |
| Fold1 | 78.29 | 81.31 | 75.39 | 77.79 | 72.81 | 76.14 | 75.82 | 69.33 |
| Fold2 | 78.55 | 83.14 | 83.79 | 71.63 | 74.11 | 75.06 | 74.38 | 67.22 |
| Fold3 | 79.89 | 83.63 | 76.79 | 78.38 | 73.58 | 71.49 | 73.57 | 68.60 |
| Fold4 | 79.35 | 83.60 | 80.32 | 75.17 | 72.80 | 77.21 | 72.32 | 65.82 |
| Fold5 | 78.34 | 81.15 | 75.61 | 77.74 | 72.73 | 69.35 | 71.73 | 69.45 |
| Fold6 | 79.11 | 79.15 | 78.47 | 79.40 | 76.42 | 76.08 | 74.53 | 72.51 |
| Fold7 | 78.91 | 80.89 | 79.48 | 77.16 | 68.24 | 72.67 | 73.65 | 64.55 |
| Fold8 | 79.07 | 79.42 | 78.83 | 78.91 | 74.94 | 71.56 | 75.57 | 69.61 |
| Fold9 | 78.78 | 82.68 | 77.11 | 76.33 | 75.58 | 85.09 | 74.33 | 69.10 |
| Average | 79.03 | 81.69 | 78.55 | 77.12 | 73.75 | 75.12 | 74.24 | 68.69 |

Table 5.17: Q3 accuracy and SOV score for ensembles, external rules and random forest filtering for CB513 dataset.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 82.27 | 85.81 | 83.74 | 78.73 | 76.39 | 82.40 | 79.65 | 71.18 |
| Fold1 | 82.47 | 86.29 | 83.55 | 78.97 | 77.02 | 81.79 | 83.20 | 72.90 |
| Fold2 | 82.18 | 87.41 | 86.47 | 75.16 | 77.99 | 80.07 | 78.40 | 71.31 |
| Fold3 | 81.36 | 85.17 | 80.74 | 78.45 | 74.72 | 77.20 | 76.71 | 69.18 |
| Fold4 | 81.61 | 86.51 | 84.86 | 75.68 | 74.66 | 77.52 | 77.64 | 68.22 |
| Fold5 | 81.07 | 84.67 | 81.55 | 78.14 | 74.04 | 73.18 | 74.84 | 69.70 |
| Fold6 | 81.12 | 82.20 | 83.68 | 79.03 | 77.96 | 78.38 | 78.53 | 73.18 |
| Fold7 | 81.66 | 83.99 | 84.91 | 78.20 | 74.18 | 79.35 | 80.48 | 69.76 |
| Fold8 | 82.17 | 82.68 | 83.85 | 80.75 | 79.30 | 74.91 | 82.88 | 74.00 |
| Fold9 | 81.55 | 84.95 | 84.94 | 76.72 | 77.06 | 85.18 | 80.14 | 70.15 |
| Average | 81.75 | 84.97 | 83.83 | 77.98 | 76.33 | 79.00 | 79.25 | 70.96 |

Table 5.18: Q3 accuracy and SOV score for ensembles and random forest filtering for CB513 dataset.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold0 | 82.29 | 85.01 | 81.73 | 80.53 | 79.98 | 84.08 | 78.58 | 73.99 |
| Fold1 | 82.62 | 85.42 | 81.67 | 81.10 | 78.76 | 82.96 | 78.73 | 74.28 |
| Fold2 | 82.38 | 86.26 | 84.41 | 77.78 | 81.07 | 81.60 | 76.53 | 76.04 |
| Fold3 | 81.42 | 84.02 | 78.29 | 80.91 | 77.83 | 77.43 | 73.99 | 73.54 |
| Fold4 | 81.74 | 85.85 | 82.53 | 77.78 | 78.32 | 80.52 | 75.29 | 71.37 |
| Fold5 | 81.07 | 84.09 | 79.63 | 79.61 | 76.72 | 76.00 | 72.81 | 71.34 |
| Fold6 | 81.35 | 81.90 | 81.79 | 80.73 | 80.63 | 80.30 | 79.57 | 75.56 |
| Fold7 | 81.42 | 82.94 | 83.04 | 79.43 | 75.96 | 79.43 | 78.51 | 71.51 |
| Fold8 | 81.93 | 81.12 | 82.33 | 82.40 | 80.97 | 75.23 | 79.86 | 75.37 |
| Fold9 | 81.73 | 83.95 | 83.14 | 79.01 | 79.56 | 85.64 | 78.75 | 74.14 |
| Average | 81.80 | 84.06 | 81.86 | 79.93 | 78.98 | 80.32 | 77.26 | 73.71 |

Table 5.19: Q3 accuracy and SOV score for ensembles, random forest and external rules filtering for CB513 dataset.

### 5.2.5 Additional experiments with CB513

Since the order of the filtering methods matters, the order of the ensembles method could also lead to different results. A few experiments were performed with fold 0 of CB513, where the ensembles method was applied after the various filtering methods. According to table 5.20, applying the ensembles after the filtering methods, leads to better results. Apart from that, it is possible to combine all the filtering methods into one ensembles model, which was not possible in the scenario where the ensembles method was applied first. The new ensembles model, which combined 15 models (5 models with external rules and SVM filtering, 5 models with external rules and decision tree filtering, and 5 models with external rules and random forest filtering), had the highest SOV score. Further experiments have not been performed because of the shortage of time.

| Method Used (for fold 0 of CB513) | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Ensembles + External Rules + SVM | 79.24 | 80.52 | 72.51 | 82.13 | 76.80 | 77.09 | 74.23 | 73.38 |
| Ensembles + External Rules + Decision Tree | 80.18 | 80.77 | 78.16 | 80.88 | 76.08 | 75.57 | 74.92 | 72.28 |
| Ensembles + External Rules + Random Forest | 80.00 | 81.91 | 79.66 | 78.73 | 76.32 | 76.51 | 76.54 | 70.72 |
| External Rules + SVM + Ensembles | 79.49 | 79.93 | 72.57 | 83.13 | 76.56 | 77.30 | 74.36 | 73.41 |
| External Rules + Decision Tree + Ensembles | **81.55** | 80.48 | 78.49 | 84.12 | 77.05 | 75.55 | 77.35 | 75.07 |
| External Rules + Random Forest + Ensembles | 81.52 | 81.70 | 81.01 | 81.68 | 76.59 | 76.91 | 76.67 | 74.01 |
| Ensembles (SVM + Decision Tree + Random Forest) | 81.21 | 80.98 | 77.26 | 83.64 | **77.16** | 76.53 | 76.81 | 75.17 |

Table 5.20: Results for fold 0 of CB513 with the ensembles method applied before and after the filtering methods.

### 5.2.6 Final results for CB513

After collecting all the results for all filtering methods, the 10-fold cross validation method (average) was used to combine the results for all folds. These results are presented in table 5.21, which makes it easier to compare the different filtering methods. According to table 5.21, the best results for CB513, in terms of overall Q3 accuracy and overall SOV score, came from the ensembles model with the random forest and external rules filtering. This model managed to reach 81.80% Q3 (per residue) accuracy and 78.98 SOV score, which is very close with the current state-of-the-art results (84-85% Q3 accuracy).

| METHOD | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| CROSS-VALIDATION | 77.25 | 78.95 | 65.75 | 82.13 | 72.91 | 73.73 | 69.87 | 71.88 |
| ENSEMBLES (5 EXPERIMENTS / FOLD) | 77.46 | 78.98 | 66.35 | 82.27 | 73.55 | 74.57 | 70.25 | 72.16 |
| ENSEMBLES + EXTERNAL RULES | 77.52 | 77.87 | 65.18 | 83.96 | 74.88 | 74.92 | 69.96 | 72.48 |
| ENSEMBLES + EXTERNAL RULES + SVM | 77.79 | 79.69 | 74.76 | 77.85 | 74.70 | 74.91 | 72.58 | 70.30 |
| ENSEMBLES + SVM | 79.27 | 82.00 | 77.05 | 78.27 | 75.79 | 76.60 | 75.18 | 71.64 |
| ENSEMBLES + SVM + EXTERNAL RULES | 79.30 | 81.40 | 76.02 | 79.40 | 76.80 | 76.81 | 74.05 | 72.21 |
| ENSEMBLES + EXTERNAL RULES + DECISION TREE | 79.15 | 79.82 | 77.58 | 79.44 | 73.72 | 73.36 | 74.33 | 70.20 |
| ENSEMBLES + DECISION TREE | 81.69 | 83.29 | 82.27 | 80.07 | 75.93 | 77.29 | 79.28 | 71.66 |
| ENSEMBLES + DECISION TREE + EXTERNAL RULES | 81.86 | 82.20 | 80.35 | 82.41 | 78.83 | 78.87 | 77.21 | 74.34 |
| ENSEMBLES + EXTERNAL RULES + RANDOM FOREST | 79.03 | 81.69 | 78.55 | 77.12 | 73.75 | 75.12 | 74.24 | 68.69 |
| ENSEMBLES + RANDOM FOREST | 81.75 | 84.97 | 83.83 | 77.98 | 76.33 | 79.00 | 79.25 | 70.96 |
| ENSEMBLES + RANDOM FOREST + EXTERNAL RULES | 81.80 | 84.06 | 81.86 | 79.93 | 78.98 | 80.32 | 77.26 | 73.71 |

Table 5.21: 10-fold Cross validation, Q3 accuracy and SOV score for all methods for CB513 dataset.

## 5.3 Experiments with PISCES dataset

The PISCES dataset is much bigger than the CB513 dataset and the experiments for this dataset required a lot more time. Because of that, most hyper parameters used in PISCES experiments were derived from the CB513 experiments. This will probably have an impact on the prediction capabilities of the final model, but further experiments could not be made due to the shortage of time. The hyper parameters used for the CNN in the PISCES experiments are shown in table 5.1. The max epochs (max iterations) were increased from 50 to 100 because the model required more epochs to learn the bigger dataset. Table 5.22 shows the hyper parameters that were used to train all PISCES models.

| GNsize | C | CNN layers | bsize | Max Iterations | Dimensions |
|---|---|---|---|---|---|
| 2048 | 0.01 | 4 | 12288 | 100 | 15 20 1 |

Table 5.22: Hyper parameters for SHN method, used for all PISCES experiments.

### 5.3.1 5-fold Cross-Validation and Ensembles Results

For the PISCES dataset a 5-fold cross validation was used, where seven (7) models were trained per fold and the best one was selected. The main reason a 5-fold cross validation was chosen, instead of a 10-fold, was to make the results comparable with past studies on PSSP, that used the PISCES dataset. Table 5.23 displays the Q3 accuracy and SOV score of the best model for each fold.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 79.22 | 84.45 | 69.90 | 79.39 | 76.00 | 78.84 | 77.47 | 71.60 |
| Fold2 | 79.41 | 84.19 | 71.02 | 79.59 | 76.47 | 77.80 | 76.85 | 72.44 |
| Fold3 | 79.60 | 84.33 | 70.47 | 79.87 | 76.48 | 78.23 | 76.09 | 72.16 |
| Fold4 | 79.88 | 84.62 | 71.06 | 80.09 | 76.67 | 79.04 | 77.30 | 72.55 |
| Fold5 | 79.41 | 84.38 | 70.04 | 79.96 | 76.59 | 80.37 | 76.65 | 72.26 |
| Average | 79.50 | 84.39 | 70.50 | 79.78 | 76.44 | 78.86 | 76.87 | 72.20 |

Table 5.23: Q3 accuracy and SOV score for 5-fold cross validation for PISCES dataset.

To create the ensembles model five (5) from the seven (7) trained models were selected, so that the Q3 accuracy of the new model was maximized. Table 5.24 presents the results for Q3 accuracy and SOV score of the new ensembles model, for each fold of PISCES.

The comparison between table 5.23 and 5.24 reveals that there is a similar issue with the CB513 dataset. There is not enough variance between the trained models, which results in only a small improvement in overall Q3 accuracy (0.30%) and SOV score (0.63).

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 79.55 | 84.62 | 70.19 | 79.88 | 76.80 | 79.44 | 77.84 | 72.12 |
| Fold2 | 79.74 | 84.24 | 71.05 | 80.36 | 77.01 | 78.08 | 77.09 | 73.05 |
| Fold3 | 79.83 | 84.47 | 70.43 | 80.36 | 76.92 | 78.49 | 76.42 | 72.50 |
| Fold4 | 80.15 | 84.65 | 71.30 | 80.60 | 77.34 | 79.48 | 77.61 | 73.18 |
| Fold5 | 79.72 | 84.46 | 70.28 | 80.52 | 77.29 | 80.75 | 77.16 | 72.95 |
| Average | 79.80 | 84.49 | 70.65 | 80.34 | 77.07 | 79.25 | 77.22 | 72.76 |

Table 5.24: Q3 accuracy and SOV score for ensembles method (with 5 trained models per fold) for PISCES dataset.

### 5.3.2 Filtering Results for PISCES

A bigger dataset, like PISCES, can help the model to learn more effectively the patterns of the data, and that is why the cross validation results (Table 5.23) are better compared to the CB513 results (Table 5.6), but at the same makes it very difficult to use SVM filtering. SVMs are usually very effective for small datasets, however, on big datasets the memory scales quadratically with the number of data points, which makes them very difficult to train and impractical. Several attempts were made to train an SVM with samples from the PISCES dataset, but the results were worse than the results without the SVM filtering. Because of that, the SVM filtering was not applied in any of the PISCES experiments. If a good sample is extracted from the PISCES dataset, it might be possible to train an SVM for filtering the results of models trained with the PISCES dataset. However, for this dissertation such technique could not be found.

Table 5.25 shows the results for each of the five folds after applying the external rules to the ensembles model. It seems there was a slight increase (0.06) to the overall Q3 accuracy and a considerable increase in the overall SOV score (1.18), which was expected, since external rules are used mainly to improve the overall SOV score.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 79.64 | 83.79 | 69.08 | 81.61 | 78.06 | 79.90 | 77.61 | 72.95 |
| Fold2 | 79.79 | 83.40 | 70.07 | 81.93 | 78.04 | 78.36 | 76.89 | 73.57 |
| Fold3 | 79.91 | 83.66 | 69.40 | 81.98 | 78.15 | 78.91 | 76.11 | 73.17 |
| Fold4 | 80.22 | 83.81 | 70.24 | 82.25 | 78.53 | 79.85 | 77.40 | 73.88 |
| Fold5 | 79.75 | 83.54 | 69.23 | 82.14 | 78.47 | 80.75 | 76.93 | 73.63 |
| Average | 79.86 | 83.64 | 69.60 | 81.98 | 78.25 | 79.55 | 76.99 | 73.44 |

Table 5.25: Q3 accuracy and SOV score for ensembles with external rules filtering for PISCES dataset.

The results for the ensembles model with external rules and decision tree filtering are displayed in table 5.26. The decision tree filtering improved the Q3 accuracy by 0.80% and affected slightly the SOV score.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 80.37 | 82.94 | 76.74 | 79.90 | 77.76 | 77.91 | 80.07 | 72.75 |
| Fold2 | 80.78 | 83.01 | 77.19 | 80.68 | 78.39 | 77.28 | 79.61 | 73.97 |
| Fold3 | 80.62 | 82.85 | 77.28 | 80.20 | 78.01 | 77.17 | 78.95 | 72.86 |
| Fold4 | 80.92 | 83.16 | 77.71 | 80.46 | 78.46 | 78.52 | 80.16 | 73.35 |
| Fold5 | 80.62 | 82.93 | 76.95 | 80.47 | 78.47 | 79.25 | 79.25 | 73.97 |
| Average | 80.66 | 82.98 | 77.17 | 80.34 | 78.22 | 78.03 | 79.61 | 73.38 |

Table 5.26: Q3 accuracy and SOV score for ensembles with external rules and decision tree filtering for PISCES dataset.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 81.69 | 84.45 | 81.54 | 78.98 | 78.68 | 78.99 | 83.19 | 73.64 |
| Fold2 | 82.10 | 84.55 | 81.06 | 80.24 | 79.59 | 78.85 | 82.70 | 75.32 |
| Fold3 | 81.79 | 84.39 | 81.15 | 79.46 | 78.95 | 78.93 | 82.11 | 73.97 |
| Fold4 | 82.13 | 84.57 | 81.93 | 79.75 | 79.52 | 79.58 | 82.93 | 74.97 |
| Fold5 | 81.93 | 84.31 | 80.65 | 80.31 | 79.52 | 80.80 | 82.03 | 75.50 |
| Average | 81.93 | 84.45 | 81.27 | 79.75 | 79.25 | 79.43 | 82.59 | 74.68 |

Table 5.27: Q3 accuracy and SOV score for ensembles with decision tree filtering for PISCES dataset.

Table 5.27 shows the results, for each fold of PISCES, for the ensembles model after applying the decision tree filtering, while table 5.28 illustrates the results for the ensembles model with decision tree and external rules filtering. It is obvious that applying the decision tree filtering before the external rules produces significantly better results. Moreover, the decision tree filtering improved the results of the ensembles model considerably (81.93% overall Q3 accuracy and 79.25 overall SOV score).

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 81.75 | 83.12 | 80.08 | 81.32 | 81.28 | 80.67 | 82.87 | 75.91 |
| Fold2 | 82.10 | 83.32 | 79.67 | 82.33 | 81.57 | 79.71 | 82.33 | 76.84 |
| Fold3 | 81.86 | 83.26 | 79.52 | 81.75 | 81.11 | 80.18 | 81.32 | 75.86 |
| Fold4 | 82.16 | 83.34 | 80.47 | 81.91 | 81.48 | 80.77 | 82.41 | 76.54 |
| Fold5 | 81.91 | 83.06 | 79.20 | 82.37 | 81.55 | 82.02 | 81.58 | 76.98 |
| Average | 81.96 | 83.22 | 79.79 | 81.94 | 81.40 | 80.67 | 82.10 | 76.43 |

Table 5.28: Q3 accuracy and SOV score for ensembles with decision tree and external rules filtering for PISCES dataset.

The Q3 accuracy and SOV score for the ensembles model with external rules and random forest filtering are presented in table 5.29. A comparison between table 5.26 and table 5.29 shows that random forest filtering is clearly more effective than the decision tree filtering, for the PISCES dataset.

Table 5.30 illustrates the results for the ensembles model with the random forest filtering, while table 5.31 illustrates the results for the ensembles model with random forest and external rules filtering. According to tables 5.29 and 5.31, when random forest filtering

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 80.62 | 84.11 | 77.92 | 78.66 | 77.99 | 79.28 | 80.40 | 72.05 |
| Fold2 | 80.98 | 84.01 | 78.49 | 79.40 | 78.43 | 78.26 | 80.05 | 72.99 |
| Fold3 | 80.85 | 83.89 | 78.52 | 79.04 | 78.37 | 78.60 | 79.33 | 72.39 |
| Fold4 | 81.15 | 84.19 | 78.99 | 79.26 | 78.83 | 79.74 | 80.48 | 72.93 |
| Fold5 | 80.88 | 83.97 | 77.99 | 79.49 | 78.86 | 80.65 | 79.88 | 73.42 |
| Average | 80.90 | 84.03 | 78.38 | 79.17 | 78.50 | 79.31 | 80.03 | 72.76 |

Table 5.29: Q3 accuracy and SOV score for ensembles with external rules and random forest filtering for PISCES dataset.

is applied before external rules, the final results are much better (2.12% for Q3 accuracy and 4.14 for SOV score). Furthermore, if the external rules are applied after the random forest filtering, the overall Q3 accuracy drops slightly, while the overall SOV score grows by a small amount (Tables 5.30 and 5.31).

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 82.86 | 85.98 | 83.74 | 79.19 | 80.97 | 82.15 | 84.83 | 75.21 |
| Fold2 | 83.23 | 86.12 | 83.81 | 79.99 | 81.71 | 81.80 | 84.59 | 76.44 |
| Fold3 | 83.00 | 85.78 | 83.87 | 79.63 | 81.08 | 81.62 | 84.29 | 75.63 |
| Fold4 | 83.31 | 86.09 | 84.36 | 79.86 | 81.68 | 82.54 | 84.66 | 76.32 |
| Fold5 | 83.19 | 86.02 | 83.51 | 80.19 | 81.89 | 83.91 | 84.39 | 76.83 |
| Average | 83.12 | 86.00 | 83.86 | 79.77 | 81.47 | 82.40 | 84.55 | 76.09 |

Table 5.30: Q3 accuracy and SOV score for ensembles with random forest filtering for PISCES dataset.

| | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| Fold1 | 82.79 | 85.21 | 82.27 | 80.64 | 82.37 | 82.79 | 83.71 | 76.45 |
| Fold2 | 83.13 | 85.36 | 82.42 | 81.31 | 82.74 | 82.09 | 83.57 | 77.41 |
| Fold3 | 82.89 | 85.01 | 82.24 | 81.05 | 82.25 | 82.08 | 82.66 | 76.59 |
| Fold4 | 83.18 | 85.27 | 82.90 | 81.21 | 82.82 | 82.77 | 83.58 | 77.28 |
| Fold5 | 83.09 | 85.21 | 81.96 | 81.64 | 83.00 | 84.35 | 83.08 | 77.79 |
| Average | 83.02 | 85.21 | 82.36 | 81.17 | 82.64 | 82.82 | 83.32 | 77.10 |

Table 5.31: Q3 accuracy and SOV score for ensembles with random forest and external rules filtering for PISCES dataset.

### 5.3.3 Final Results for PISCES

The 5-fold cross validation (average) was applied for all filtering methods and the results are shown in table 5.32. The best results according to this table, came from the ensembles model after applying the random forest and external rules filtering. This method managed to reach 83.02% overall Q3 (per residue) accuracy and 82.64 overall SOV score, which is very good, considering that the state-of-the-art results were around 84-85% Q3 accuracy.

| METHOD | Q3 (%) | QH (%) | QE (%) | QC (%) | SOV | SOVH | SOVE | SOVC |
|---|---|---|---|---|---|---|---|---|
| CROSS-VALIDATION | 79.50 | 84.39 | 70.50 | 79.78 | 76.44 | 78.86 | 76.87 | 72.20 |
| ENSEMBLES (5 EXPERIMENTS / FOLD) | 79.80 | 84.49 | 70.65 | 80.34 | 77.07 | 79.25 | 77.22 | 72.76 |
| ENSEMBLES + EXTERNAL RULES | 79.86 | 83.64 | 69.60 | 81.98 | 78.25 | 79.55 | 76.99 | 73.44 |
| ENSEMBLES + EXTERNAL RULES + DECISION TREE | 80.66 | 82.98 | 77.17 | 80.34 | 78.22 | 78.03 | 79.61 | 73.38 |
| ENSEMBLES + DECISION TREE | 81.93 | 84.45 | 81.27 | 79.75 | 79.25 | 79.43 | 82.59 | 74.68 |
| ENSEMBLES + DECISION TREE + EXTERNAL RULES | 81.96 | 83.22 | 79.79 | 81.94 | 81.40 | 80.67 | 82.10 | 76.43 |
| ENSEMBLES + EXTERNAL RULES + RANDOM FOREST | 80.90 | 84.03 | 78.38 | 79.17 | 78.50 | 79.31 | 80.03 | 72.76 |
| ENSEMBLES + RANDOM FOREST | 83.12 | 86.00 | 83.86 | 79.77 | 81.47 | 82.40 | 84.55 | 76.09 |
| ENSEMBLES + RANDOM FOREST + EXTERNAL RULES | 83.02 | 85.21 | 82.36 | 81.17 | 82.64 | 82.82 | 83.32 | 77.10 |

Table 5.32: 5-fold cross-validation, Q3 accuracy and SOV score for all methods for the PISCES dataset.

## 5.4 Best Results for CB513 and PISCES

In this section the hyper parameters for the best models will be displayed, along with the filtering methods used. For both CB513 and PISCES datasets the hyper parameters used to train the CNN models are shown in table 5.1. Figure 5.3 shows the hyper parameters for the trained models (CNN with SHN), the hyper parameters for the random forest filtering and the order of the applied techniques that produced the best final results for the CB513 dataset. These techniques resulted in an overall 81.80% Q3 (per residue) accuracy and an overall 78.98 SOV score. The confusion matrix for fold 0 of CB513 of a single CNN model trained with SHN method is shown in figure 5.4. The confusion matrix for the same fold for the ensembles model (combination of 5 trained models) with random forest filtering is shown in figure 5.5. It seems that the miss-predictions for classes 'E' and 'H' are less in the ensembles model, while the miss-predictions for class 'C' are slightly more, compared to the single CNN model.

**MODEL HYPER PARAMETERS**

| GNsize | C | CNN layers | bsize | Max Iterations | Plus |
|---|---|---|---|---|---|
| 2048 | 0.01 | 4 | 12288 | 50 | 7 |

**FILTERING HYPER PARAMETERS**

| n_estimators | max_depth | random_state | min_samples_split | min_samples_leaf | Window |
|---|---|---|---|---|---|
| 100 | 25 | 42 | 2 | 1 | 13 |

**METHODS USED (IN ORDER)**

ENSEMBLES (5 EXPERIMENTS / FOLD) -> RANDOM FOREST FILTERING -> EXTERNAL RULES FILTERING
|
V
10-FOLD CROSS-VALIDATION

Figure 5.3: Hyper parameters and methods used that resulted in the best overall Q3 accuracy and best overall SOV score for CB513 dataset.

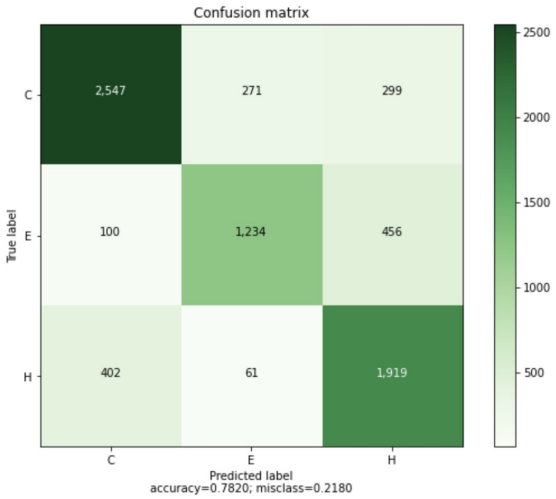



Figure 5.4: CM for CB513 fold 0 of single CNN model.

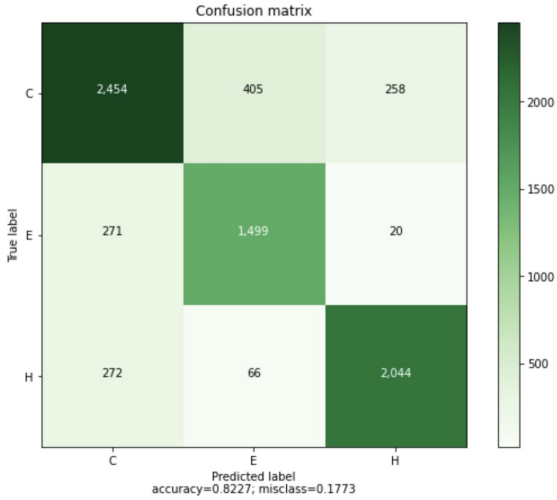



Figure 5.5: CM for CB513 fold 0 of ensembles model with random forest.

Figure 5.6 shows the hyper parameters for the trained models (CNN with SHN), the hyper parameters for the random forest filtering and the order of the applied methods, which produced the best final results for the PISCES dataset. These methods resulted in an overall 83.02% Q3 (per residue) accuracy and an overall 82.64 SOV score, which is very close with the current state-of-the-art results (84-85%).

**MODEL HYPER PARAMETERS**

| GNsize | C | CNN layers | bsize | Max Iterations | Plus |
|---|---|---|---|---|---|
| 2048 | 0.01 | 4 | 12288 | 100 | 7 |

**FILTERING HYPER PARAMETERS**

| n_estimators | max_depth | random_state | min_samples_split | min_samples_leaf | Window |
|---|---|---|---|---|---|
| 100 | 25 | 42 | 2 | 1 | 19 |

**METHODS USED (IN ORDER)**

**ENSEMBLES (5 EXPERIMENTS / FOLD) -> RANDOM FOREST FILTERING -> EXTERNAL RULES FILTERING**
**|**
**V**
**5-FOLD CROSS-VALIDATION**

Figure 5.6: Hyper parameters and methods used that resulted in the best overall Q3 accuracy and best overall SOV score for PISCES dataset.

The confusion matrix for a single CNN model trained with SHN method, using fold 4 of PISCES, is displayed in figure 5.7. Figure 5.8 illustrates the confusion matrix for the ensembles model (combination of 5 CNN models) with random forest filtering, for

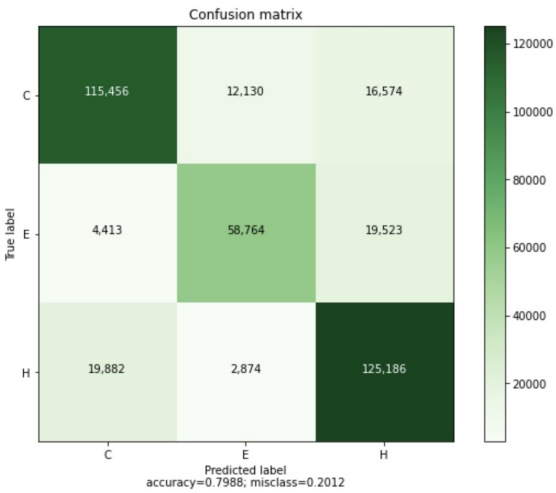



Figure 5.7: CM for PISCES fold 4 of single CNN model.

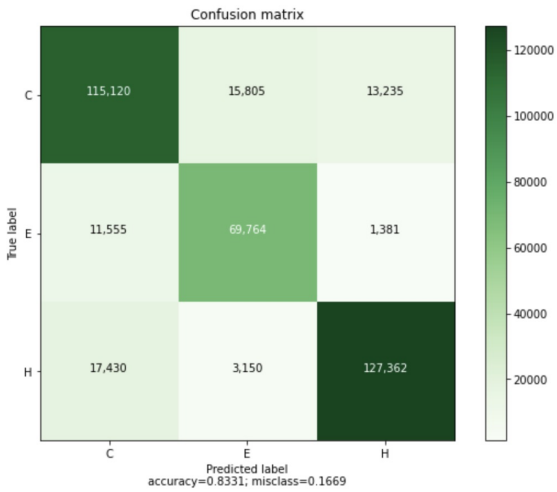



Figure 5.8: CM for PISCES fold 4 of ensembles model with random forest.

the same fold (fold 4). It is obvious that the correct predictions for classes 'E' and 'H' were increased, after applying the ensembles and filtering methods, while the correct predictions for class 'C' decreased by a small amount, compared to the single CNN model.

There are some hyper parameters that could still be modified and potentially improve the accuracy of the single CNN model with SHN. However, due to the shortage of available time, additional experiments could not be performed. In the next chapter, there will be some suggestions for further research regarding Convolutional Neural Networks with the Subsampled Hessian Newton method and the Protein Secondary Structure Prediction (PSSP) problem.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusions

The initial purpose of this dissertation was to combine a Convolutional Neural Network (CNN) with the Hessian Free Optimisation (HFO) algorithm in order to train a model that predicts the Secondary Structure of Proteins (PSSP), given its primary structure, by exploiting the MSA profiles. The attempt to solve the PSSP problem was very important, since the experimental methods that are currently available are extremely expensive in both money and time. The ability to infer (predict) the secondary structure of proteins based on the primary structure could be very beneficial for the manufacture of pharmaceutical drugs, food complements and antibiotics. Except from that, the secondary structure of proteins could be used to determine the tertiary and quaternary structures, which could help scientists define the exact functionality of the studied proteins, and possibly provide an indication on how dangerous diseases, like cancer or covid-19, can be cured.

The attempt to combine the CNN with HFO was not fruitful because of the nature of HFO, which was specifically designed for Feed Forward Neural Networks (FFNNs). However, a new variation of HFO, called Subsampled Hessian Newton (SHN) method, was utilized to train a CNN for the PSSP problem. The results for the CB513 dataset were extremely promising with about 78.20% Q3 accuracy for a single fold (fold 0), 77.25% using 10-fold cross validation and approximately 77.46% using the ensembles method with 5 trained CNN models. The highest overall Q3 accuracy was 81.80% (Table 5.19) and was achieved by combining the ensembles model with random forest filtering and external rules. For the CB513 dataset, the overall SOV score for the 10-fold cross validation was 72.91, while the same figure increased to 73.55, after applying the ensembles method (with 5 trained models). The highest overall SOV score was achieved by combining the ensembles model with random forest and external rules filtering (Table 5.19), which was approximately 78.98.

Even though some studies reported results of 84-85% Q3 accuracy, the datasets used for training, were much larger than CB513, which means that they cannot be compared directly with the results of this dissertation. The results for CB513, however, can be compared with the results of [24] and [23], who also used the CB513 dataset (with the same 10-fold cross validation). The comparison between their results and the best results of this dissertation, for CB513 dataset, shows that the single CNN with SHN (78.20% Q3 accuracy) outperformed both the CNN with SGD (76.47% Q3 accuracy) [24], and the BRNN with HFO (77.01% Q3 accuracy) [23]. The same applies for the 10-fold cross validation, where the CNN with SHN achieved 77.25% (Table 5.6), the CNN with SGD achieved 75.16% and the BRNN with HFO achieved 75.80% overall Q3 accuracy. Moreover, the

ensembles model with random forest and external rules filtering (81.80% Q3 accuracy) managed to outperform the best results reported by [24] (80.40% overall Q3 accuracy) and [23] (78.15% overall Q3 accuracy).

For the PISCES dataset, the best overall Q3 accuracy of a single CNN with SHN was 79.88% (Table 5.23), while the overall Q3 accuracy for the 5-fold cross validation was 79.50%. The highest Q3 accuracy for the 5-fold cross validation was 83.12% and was achieved with the ensembles model with the random forest filtering (Table 5.30). The best overall SOV score for a single model was 76.67 and for the 5-fold cross validation was 76.44. The highest overall SOV score achieved was 82.64, with the combination of the ensembles model (with 5 trained CNNs), random forest and external rules filtering (Table 5.31).

The PISCES results can be compared with [1] and [2], where the same 5-fold cross validation was used. The overall Q3 accuracy for the 5-fold cross validation was 79.50% in this dissertation (for PISCES), which was slightly lower than [2] (79.57%) and considerably lower than [1] (80.65%). This means that there is still room for improvement for the models trained with the PISCES dataset. However, the final results for PISCES dataset were better (83.02% overall Q3 accuracy and 82.64 overall SOV score) compared to [1] (80.98% overall Q3 accuracy and 77.26 overall SOV score) and [2] (80.37% overall Q3 accuracy and 76.71 overall SOV score).

Based on the results from the various filtering methods, it seems that the order in which the filtering is applied plays a major role on the final outcome. That was expected, as the filtering of each method applies its own 'corrections' to the results, which could 'reveal' or 'hide' the errors for the next filtering method. For instance, in a sequence of seven (7) amino acids, where the predicted secondary structure is 'HHHHEHE', two (hypothetical) filtering methods could be applied. The first filtering method replaces sequences of 'HHEH' with 'HHHH', while the second method replaces 'EHE' with 'EEE'. If the first method is applied, the sequence would become 'HHHHHHE' and after the second method it would remain the same, as there are no corrections that can be made. On a different scenario, the second filtering method could be applied first, which would result in a new sequence 'HHHHEEE', that would remain the same after applying the first method. If the desired output was 'HHHHHHH' the two approaches would result in completely different results. In the first scenario (1st method + 2nd method), the correct results would be 6 out of 7 (85.7%), while in the second scenario (2nd method + 1st method), the correct results would be 4 out of 7 (57.1%). This simple example shows that the order of the filtering methods applied can produce different final results, so various combinations

should be tested.

In addition to the above, the best filtering technique depends on the dataset and the output of the initial machine learning model. Even if two models are trained with the same machine learning architecture (e.g. CNN, FFNN, MLP, etc.) but with different optimisation algorithms (e.g. SGD, HFO, SHN, etc.), there is no guarantee that the boost from any of filtering methods will be the same for both models. The filtering results can vary between datasets and that can be observed from the filtering results of CB513 and PISCES datasets. It seems that there is no clear approach or a 'best filtering' method that will guarantee better results for all the machine learning models. Consequently, different filtering methods should be applied and the ones that produce the best results should be selected.

As mentioned earlier, some of the hyper parameters have not been exploited in this dissertation, not to a higher extend at least. This means there is still room for improvement for the prediction results. Some of these hyper parameters are the 'plus' parameter (window for CNN), the number of convolution layers, the number of filters and the kernel size of the CNN, as well as the number of trained models used in ensembles method. In addition to that, for the PISCES dataset, the hyper parameters used, to train the models, could be tuned in the same way that were tuned for the CB513 dataset.

## 6.2 Suggestions for Future Work on PSSP

Over the past years, multiple machine learning algorithms were utilized to predict the secondary structure of proteins, given their primary structure. However, none of these techniques managed to reach the maximum theoretical limit for the 3-class prediction of the PSSP problem, which is around 88-90% Q3 accuracy. That leaves the question whether a single machine learning model can even manage to reach such high accuracy, for such a complex problem (PSSP).

Maybe it is time to look for other alternative methods, like training multiple models and then combining their predicted results. One such method is called stacking ensemble method. This technique is similar to the ensembles method used in this dissertation, with the only difference that, instead of training the same type of model multiple times, it suggests to train different types of machine learning models, like K-means, Decision Trees, Support Vector Machines, Naive Bayes, Logistic Regression, and variations of Neural Networks. After the selected models are trained, they must be used to predict the test (or validation) dataset, and the predictions must be stored in an output file. Then a logistic regression model could be utilized, to learn how to best combine the predictions from

each of the separate models. This method does not guarantee that the stacked ensemble results will be better than the results of all separate models, however, even in that case, the model with the highest accuracy could be used instead of the stacked ensemble.

As many data scientists claim, 'the answers are in the data'. Given this it is possible that the reason behind the accuracy limitations of the prediction models are related with the input data. Therefore, another suggestion would be to use different datasets to train the models or perform some modifications to the datasets in order to help the network extract the most important features. In addition to that, a separate dataset could be used only for tuning the hyper parameters and another one just for testing the model. According to the final results, it was obvious that the models were able to predict the classes 'H' and 'C' with higher success rate than the 'E' class. This means that the network could not extract all necessary features to be able to predict the 'E' class accurately. This phenomenon was observed because the datasets were not balanced.

It was observed that the accuracy of the predictions for various proteins with different lengths was not the same. A statistical analysis, on the final results, could help identify for which proteins the model had higher or lower accuracy. This could give an indication on which types of proteins the accuracy should be improved, in order to increase the overall accuracy of the model.

Different filtering methods can produce different results for different algorithms used for the PSSP and for different protein datasets, which means that experimentation, with various filtering methods, is essential in order to find the optimal filtering method. In this dissertation, only a few filtering methods were used, which leaves the door open for further experimentation with other filtering methods. The order, in which the filtering methods were applied, can affect the final results. Because of that, it is highly suggested to apply the filtering methods in different orders and choose the one that produces the best results. Moreover, the filtering methods could be applied before the ensembles method, which could lead to better results. Another approach could be to apply the external rules or other filtering methods multiple times (e.g. apply external rules, SVM filtering, ensembles method and then apply external rules for a second time).

In this dissertation, one of the goals was to train a CNN with the stochastic gradient descent (SGD) algorithm and compare the results with the SHN method. However, due to the lack of time these experiments have not been performed. These experiments could illustrate whether the SHN method could outperform the SGD algorithm for the PSSP problem. Theoretically, the CNN with SHN should require less time and iterations to train, compared to the CNN with SGD. The comparison between the CNN with SHN

and the CNN with SGD from [1] could not be made, because the implementations were different, the machines used for the experiments had different specifications and in [1] a CPU was used, instead of a GPU. According to some benchmark results from other problems [3], even though SHN performed almost the same with SGD in terms of accuracy, it was more robust than SGD in terms of hyper parameter tuning. This can still make SHN a better optimisation option, since the trained models that are required, in order to find the best hyper parameters, are significantly less than SGD. In addition, the total training iterations of SHN, for each model, are considerably less which means less training time.

# References


[1] A. Dionysiou, M. Agathocleous, C. Christodoulou, and V. Promponas, *Input representation of sequence to structure prediction problems for deep learning, (in preparation)*, 2020.

[2] K. Charalambous, M. Agathocleous, C. Christodoulou, and V. Promponas, "Solving the protein secondary structure prediction problem with the hessian free optimization algorithm", *IEEE Access*, 2020, Under review.

[3] C.-C. Wang, K. Tan, and C.-J. Lin, "Newton methods for convolutional neural networks", *ACM Transactions on Intelligent Systems and Technology*, vol. 11, pp. 1–30, 2020.

[4] J. M. Berg, J. L. Tymoczko, and L. Stryer, *Biochemistry*, 5th. New York: NY: WH. Freeman, 2002.

[5] C. Magnan and P. Baldi, "Sspro/accpro 5: Almost perfect prediction of protein secondary structure and relative solvent accessibility using profiles, machine learning and structural similarity.", *Bioinformatics*, vol. 30, no. 18, pp. 2592–2597, 2014.

[6] Y. Yang, J. Gao, J. Wang, R. Heffernan, J. Hanson, K. Paliwal, and Y. Zhou, "Sixty-five years of the long march in protein secondary structure prediction: The final stretch?", *Briefings in Bioinformatics*, vol. 19, no. 3, pp. 482–494, 2018.

[7] N. Qian and T. J. Sejnowski, "Predicting the secondary structure of globular proteins using neural network models", *Journal of Molecular Biology*, vol. 202, no. 4, pp. 865–884, 1988.

[8] B. Rost and C. Sander, "Improved prediction of protein secondary structure by use of sequence profiles and neural networks", *Proceedings of the National Academy of Sciences, USA*, vol. 90, pp. 7558–7562, 1993.

[9] A. Salamov and V. Soloveyev, "Ab initio gene finding in drosophila genomic dna", *Genome Research*, vol. 10, no. 4, pp. 516–522, 2000.

[10] R. D. King and M. J. Sternberg, "Identification and application of the concepts important for accurate and reliable protein secondary structure prediction", *Protein Science*, vol. 5, no. 11, pp. 2298–2310, 1996.

[11] D. Frishman and P. Argos, “Knowledge-based protein secondary structure assignment”, *Proteins: Structure, Function, and Bioinformatics*, vol. 23, no. 4, pp. 566–579, 1995.

[12] J. A. Cuff and G. J. Barton, “Evaluation and improvement of multiple sequence methods for protein secondary structure prediction”, *Proteins: Structure, Function, and Bioinformatics*, vol. 34, no. 4, pp. 508–519, 1999.

[13] P. Baldi, S. Brunak, P. Frasconi, G. Soda, and G. Pollastri, “Exploiting the past and the future in protein secondary structure prediction”, *Bioinformatics*, vol. 15, no. 11, pp. 937–946, 1999.

[14] P. Baldi, S. Brunak, P. Frasconi, G. Pollastri, and G. Soda, *Bidirectional Dynamics for Protein Secondary Structure Prediction. In: Sun R., Giles C.L. (eds) Sequence Learning*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2000, vol. 1828, pp. 80–104.

[15] J. Blazewicz, P. Hammer, and P. Lukasiak, “Predicting secondary structures of proteins”, *IEEE engineering in medicine and biology magazine*, vol. 24, no. 3, pp. 88–94, 2005.

[16] G. Armano, A. Orro, and E. Vargiu, “Massp3: A system for predicting protein secondary structure”, *EURASIP Journal on Advances in Signal Processing*, vol. 2006, no. 17195, pp. 1–9, 2006.

[17] F. U. Yüksektepe, O. Yılmaz, and M. Türkay, “Prediction of secondary structures of proteins using a two-stage method”, *Computers & Chemical Engineering*, vol. 32, no. 1–2, pp. 78–88, 2008.

[18] K. J. Won, T. Hamelryck, A. Prugel-Bennett, and A. Krogh, “An evolutionary method for learning hmm structure: Prediction of protein secondary structure”, *BMC bioinformatics*, vol. 8, p. 357, Feb. 2007.

[19] J. Chen and N. S. Chaudhari, “Cascaded bidirectional recurrent neural networks for protein secondary structure prediction”, *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 4, no. 4, pp. 572–582, 2007.

[20] S. Wang, J. Peng, J. Ma, and J. Xu, “Protein secondary structure prediction using deep convolutional neural fields”, *Scientific reports*, vol. 6, 2016.

[21] P. Pavlidis, Πρόβλεψη δευτεροταγούς δομής των πρωτεϊνών με τη χρήση των *convolutional neural networks* για οπτική αναγνώρηση αντικειμένων, University of Cyprus, Computer Science Department, Thesis Project, 2016.

[22] R. Heffernan, Y. Yang, K. Paliwal, and Y. Zhou, "Capturing non-local interactions by long short-term memory bidirectional recurrent neural networks for improving prediction of protein secondary structure, backbone angles, contact numbers and solvent accessibility", *Bioinformatics*, vol. 33, pp. 2842–2849, 2017.

[23] K. Charalambous, *Protein secondary structure prediction using bidirectional recurrent neural networks and hessian free optimisation*, BSc Thesis, Department of Computer Science, University of Cyprus, 2018.

[24] A. Dionysiou, Πρόβλεψη δευτεροταγούς δομής πρωτεϊνών με χρήση συνελικτικών νευρωνικών δικτύων σε συνδιασμό με φίλτρα *gabor* και *support vector machines*, BSc Thesis, Department of Computer Science, University of Cyprus, 2018.

[25] C. Fang, Y. Shang, and D. Xu, "Mufold-ss: New deep inception–inside–inception networks for protein secondary structure prediction", *Proteins: Structure, Function, and Bioinformatics*, 2018.

[26] Learn.Genetics, *Genetic science learning center*, https://learn.genetics.utah.edu/, [Online; accessed April 21, 2020], 2018.

[27] C. Simons, *Food science toolbox*, https://cwsimons.com/wp-content/uploads/2017/12/AA.jpg, [Online; accessed April 23, 2020].

[28] A. Brunning, *A guide to the 20 common amino acids*, http://www.compoundchem.com/wp-content/uploads/2014/09/20-Common-Amino-Acids-v3.png, [Online; accessed April 23, 2020].

[29] BioTopics, *Amino acid condensation*, http://www.biotopics.co.uk/as/aminocon.html, [Online; accessed April 27, 2020].

[30] T. Brown and T. Brown Jr, *Nucleic acids book*, https://www.atdbio.com/, [Online; accessed April 25, 2020].

[31] S. Clancy and W. Brown, "Translation: Dna to mrna to protein", *Nature Education*, vol. 1, no. 1, p. 101, 2008.

[32] M. A. Clark, M. Douglas, and J. Choi, *Biology 2e*, https://openstax.org/books/biology-2e/pages/3-4-proteins, [Online; accessed December 15, 2019], 2018.

[33] A. Byun, *Convolutional neural networks for visual recognition*, https://cs231n.github.io/neural-networks-1/, [Online; accessed April 23, 2020].

[34] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[35] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[36] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain”, *Psychological Review*, pp. 65–386, 1958.

[37] V. Kurková, “Kolmogorov’s theorem and multilayer neural networks”, *Neural Networks*, vol. 5, no. 3, pp. 501–506, 1991.

[38] M. I. Jordan, *Serial order: A parallel distributed processing approach*, San Diego: University of California, Institute for Cognitive Science, Technical Report number AD-A-173989/5/XAB; ICS-8604, 1986.

[39] J. L. Elman, “Finding structure in time”, *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[40] K. Patel, *Mnist handwritten digits classification using a convolutional neural network (cnn)*, [Web; accessed December 16, 2019], 2019.

[41] Y. L. Boureau, J. Ponce, and Y. LeCun, “A theoretical analysis of feature pooling in visual recognition”, in *In Proceedings of the 27th international conference on machine learning*, 2010, pp. 111–118.

[42] T. Ganegedara, *Intuitive guide to cnns*, https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-convolution-neural-networks-e3f054dd5daa, [Online; accessed April 27, 2020], 2018.

[43] D. R. Hush and J. M. Salas, “Improving the learning rate of back propagation with the gradient reuse algorithm”, in *Proceedings of the IEEE 1988 International Conference on Neural Networks*, vol. 1, San Diego, CA, USA, 1988, pp. 441–447.

[44] C. Charalambous, “Conjugate gradient algorithm for efficient training of artificial neural networks”, *Circuits, Devices and Systems, IEE Proceedings G*, vol. 139, no. 3, pp. 301–310, 1992.

[45] C. M. Bishop, *Neural networks for pattern recognition*, Oxford University Press, Oxford, UK., 1995.

[46] J. Martens, “Deep learning via hessian-free optimization”, in *Proceedings of the 27th International Conference on Machine Learning (ICML’10), Bottou, L. and Littman, M., (eds.)*, 2010, pp. 735–742.

[47] S. G. Nash, “Newton-type minimization via the lanczos method”, *SIAM Journal on Numerical Analysis*, vol. 21, no. 4, pp. 770–788, 1984.

[48] ——, “A survey of truncated-newton methods”, *Journal of Computational and Applied Mathematics*, vol. 124, no. 1, pp. 45–59, 2000.

[49] J. Martens and I. Sutskever, *Training Deep and Recurrent Networks with Hessian-Free 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, 2012, vol. 7700.

[50] N. Schraudolph, "Fast curvature matrix-vector products for second-order gradient descent", *Neural Computation*, vol. 14, no. 7, pp. 1723–1738, 2002.

[51] R. E. Wengert, "A simple automatic derivative evaluation program", *Communications of the ACM*, vol. 7, no. 8, pp. 463–464, 1964.

[52] B. A. Pearlmutter, "Fast exact multiplication by the hessian", *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994.

[53] B. Rost, C. Sander, and R. Schneider, "Phd–an automatic mail server for protein secondary structure prediction", *Bioinformatics*, vol. 10, no. 1, pp. 53–60, 1994.

[54] A. Zemla, Č. Venclovas, K. Fidelis, and B. Rost, "A modified definition of sov, a segment–based measure for protein secondary structure prediction assessment", *Proteins: Structure, Function, and Bioinformatics*, vol. 34, no. 2, pp. 220–223, 1999.

[55] W. Kabsch and C. Sander, "Dictionary of protein secondary structure: Pattern recognition of hydrogen–bonded and geometrical features", *Biopolymers*, vol. 22, no. 12, pp. 2577–2637, 1983.

[56] G. Wang and R. L. Dunbrack Jr, "Pisces: A protein sequence culling server", *Bioinformatics*, vol. 19, no. 12, pp. 1589–1591, 2003.

[57] A. Dionysiou, M. Agathocleous, C. Christodoulou, and V. Promponas, "Convolutional neural networks in combination with support vector machines for complex sequential data classification", *Artificial Neural Networks and Machine Learning - ICANN 2018, Lecture Notes in Computer Science, ed. by V. Kurkova, Y. Manolopoulos, B. Hammer, L. Iliadis, I. Maglogiannis, Cham: Springer*, vol. 11140, pp. 444–455, 2018.

[58] C. Cortes and V. Vapnik, "Support-vector network", *Machine Learning*, vol. 20, pp. 1–25, 1995.

[59] P. Kountouris, M. Agathocleous, V. J. Promponas, G. Christodoulou, S. Hadjicostas, V. Vassiliades, and C. Christodoulou, "A comparative study on filtering protein secondary structure prediction", *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, no. 3, pp. 731–739, 2012.

[60] T. Yiu, *Understanding random forest*, Torward Data Science, 2019. [Online]. Available: https://towardsdatascience.com/understanding-random-forest-58381e0602d2.

[61] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks - a publishing format for reproducible computational workflows", in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds., IOS Press, 2016, pp. 87–90.

[62] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[63] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché–Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[64] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[65] A. Botev, H. Ritter, and D. Barber, "Practical gauss-newton optimisation for deep learning", in *In Proceedings of the 34th International Conference on Machine Learning*, 2017, pp. 557–565.

[66] X. He, D. Mudigere, M. Smelyanskiy, and M. Taká, *Large scale distributed hessian free optimization for deep neural network*, 2017.

[67] R. Kiros, *Training neural networks with stochastic hessian-free optimization*, arXiv preprint arXiv:1301.3641, 2013.

[68] O. Vinyals and D. Povey, "Krylov subspace descent for deep learning", in *In Proceedings of Artificial Intelligence and Statistics*, 2012, pp. 1261–1268.

[69] C.-C. Wang, C.-H. Huang, and C.-J. Lin, "Subsampled hessian newton methods for supervised learning", *Neural Computation*, vol. 27, no. 8, pp. 1766–1795, 2015.

[70] C.-C. Wang, K.-L. Tan, C.-T. Chen, Y.-H. Lin, S. S. Keerthi, D. Mahajan, S. Sundararajan, and C.-J. Lin, "Distributed newton methods for deep learning", *Neural Computation*, vol. 30, no. 6, pp. 1673–1724, 2018.

[71] L. Breiman, "Random forests", *Machine Learning*, vol. 45, pp. 5–32, 2001.

# Appendix A

# Excluded proteins from CB513

Table A.1 shows the names of the eight (8) proteins that were excluded from the CB513 dataset, because of zeroed MSA profiles.

| No. | Protein |
|---|---|
| 1 | 1coiA_1-29 |
| 2 | 1mctI_1-28 |
| 3 | 1tiiC_195-230 |
| 4 | 2erlA_1-40 |
| 5 | 1ceoA_202-254 |
| 6 | 1mrtA_31-61 |
| 7 | 1wfbB_1-37 |
| 8 | 6rlxC_-2-20 |

Table A.1: Excluded CB513 proteins due to zeroed MSA profiles.

# Appendix B

# Excluded proteins from PISCES

Table B.1 displays the identities of the sixteen (16) proteins that were excluded from the PISCES dataset, because their MSA profiles were missing.

| No. | Protein | No. | Protein |
|---|---|---|---|
| 1 | 1VPPX | 9 | 4P6LA |
| 2 | 3MLSP | 10 | 3S0RA |
| 3 | 4P2OP | 11 | 1WFBA |
| 4 | 1G0YI | 12 | 1RPQW |
| 5 | 4JQIV | 13 | 4JO6Y |
| 6 | 3UKWC | 14 | 4H8LA |
| 7 | 4H25C | 15 | 1YODA |
| 8 | 3SGRA | 16 | 4KE2A |

Table B.1: Excluded PISCES proteins due to missing MSA profiles.

Tables B.2, B.3 and B.4 illustrate the names of the PISCES proteins that were excluded from the PISCES dataset because their MSA profiles were corrupted or zeroed (according to [24]). In total those proteins were three hundred forty one (341).

| No. | Protein | No. | Protein | No. | Protein | No. | Protein |
|---|---|---|---|---|---|---|---|
| 1 | 3P51A | 31 | 2OU5A | 61 | 3U5WA | 91 | 2R19A |
| 2 | 1V96A | 32 | 3L60A | 62 | 3O6QA | 92 | 3OP6A |
| 3 | 3L7HA | 33 | 3H0DA | 63 | 3UV0A | 93 | 2PW9A |
| 4 | 4DHXA | 34 | 1Q2HA | 64 | 3NS4A | 94 | 4GOFA |
| 5 | 4F2LA | 35 | 4I86A | 65 | 2HQLA | 95 | 3N7XA |
| 6 | 2D7EA | 36 | 2G7SA | 66 | 3TE8A | 96 | 2P9XA |
| 7 | 4MTUA | 37 | 2P63A | 67 | 2O38A | 97 | 2VS0A |
| 8 | 4BSXA | 38 | 3Q18A | 68 | 4PF3A | 98 | 2WG8A |
| 9 | 4F87A | 39 | 2FI1A | 69 | 4H41A | 99 | 4P49A |
| 10 | 4MYVA | 40 | 2Y5PA | 70 | 3H16A | 100 | 3ZC0A |
| 11 | 1QV9A | 41 | 4Q53A | 71 | 2D59A | 101 | 3B0FA |
| 12 | 3FF5A | 42 | 2Q3TA | 72 | 1VR4A | 102 | 2OX7A |
| 13 | 4P2VA | 43 | 1VPRA | 73 | 2O1QA | 103 | 4I16A |
| 14 | 1WWPA | 44 | 3DFUA | 74 | 2HX5A | 104 | 4PSFA |
| 15 | 2D68A | 45 | 3DNXA | 75 | 2NPTA | 105 | 4AP5A |
| 16 | 2R85A | 46 | 4GUCA | 76 | 3CRYA | 106 | 3K8RA |
| 17 | 4MO0A | 47 | 3Q6CA | 77 | 2ERWA | 107 | 3D3OA |
| 18 | 4L3UA | 48 | 4LTBA | 78 | 3C4RA | 108 | 3HLSA |
| 19 | 4J5RA | 49 | 3MD9A | 79 | 2IP6A | 109 | 1WPNA |
| 20 | 2OL5A | 50 | 3ESMA | 80 | 3GO9A | 110 | 2O99A |
| 21 | 4KTWA | 51 | 3H0NA | 81 | 1YPYA | 111 | 3I76A |
| 22 | 3D33A | 52 | 4HHVA | 82 | 3E0RA | 112 | 4LQBA |
| 23 | 3PD7A | 53 | 3M5QA | 83 | 4F27A | 113 | 4JX0A |
| 24 | 3KVPA | 54 | 4N74A | 84 | 3PL0A | 114 | 4R7RA |
| 25 | 3QH6A | 55 | 4KQIA | 85 | 3IBWA | 115 | 4GT9A |
| 26 | 3VS8A | 56 | 4IHQA | 86 | 3TS9A | 116 | 4F4WA |
| 27 | 3CPTA | 57 | 4K92A | 87 | 3D55A | 117 | 2UVPA |
| 28 | 3GP6A | 58 | 4Q7OA | 88 | 3NOQA | 118 | 1Z4RA |
| 29 | 3O65A | 59 | 4J1VA | 89 | 4E6WA | 119 | 2OU1A |
| 30 | 1TU9A | 60 | 4X33A | 90 | 4AQOA | 120 | 3F67A |

Table B.2: Excluded PISCES proteins due to corrupted or zeroed MSA profiles (1-120).

| No. | Protein | No. | Protein | No. | Protein | No. | Protein |
|---|---|---|---|---|---|---|---|
| 121 | 1Q9UA | 151 | 2R5SA | 181 | 3ESLA | 211 | 1U6HB |
| 122 | 1WV3A | 152 | 2FB0A | 182 | 1U9LA | 212 | 1UVQC |
| 123 | 3ONQA | 153 | 3L49A | 183 | 3H35A | 213 | 1ZVZB |
| 124 | 2YF2A | 154 | 2O4AA | 184 | 1BB1A | 214 | 1ZW2B |
| 125 | 3Q0HA | 155 | 2NS0A | 185 | 1BB1B | 215 | 2BPA3 |
| 126 | 3TUOA | 156 | 2ZEXA | 186 | 1BB1C | 216 | 2BPTB |
| 127 | 2E1FA | 157 | 3UV1A | 187 | 1C94A | 217 | 2C5IP |
| 128 | 3C0DA | 158 | 3I4UA | 188 | 1DPJB | 218 | 2C5KP |
| 129 | 3IV4A | 159 | 3RK6A | 189 | 1DTDB | 219 | 2DS2A |
| 130 | 2CVIA | 160 | 4P78A | 190 | 1F8VD | 220 | 2ERLA |
| 131 | 1SQWA | 161 | 1UV7A | 191 | 1GWMA | 221 | 2GWWB |
| 132 | 4BOQA | 162 | 2HL7A | 192 | 1HX6A | 222 | 2HZSI |
| 133 | 3W6SA | 163 | 4U9OA | 193 | 1KD8A | 223 | 2MLTA |
| 134 | 3TVQA | 164 | 4BSVA | 194 | 1KP6A | 224 | 2P06A |
| 135 | 3LQ9A | 165 | 2C0NA | 195 | 1KVEA | 225 | 2PBDV |
| 136 | 3BPQA | 166 | 3I00A | 196 | 1L2WI | 226 | 2PLXB |
| 137 | 2BDRA | 167 | 3FH3A | 197 | 1M3WA | 227 | 2QUOA |
| 138 | 3F43A | 168 | 3NR5A | 198 | 1M45B | 228 | 2W4YA |
| 139 | 3G21A | 169 | 4E6SA | 199 | 1M46B | 229 | 2WBYC |
| 140 | 4J91A | 170 | 4LKUA | 200 | 1MCTI | 230 | 2WFUA |
| 141 | 4K12A | 171 | 3KTOA | 201 | 1MQSB | 231 | 2WFVA |
| 142 | 2Q3SA | 172 | 4QRNA | 202 | 1MW5A | 232 | 2WWXB |
| 143 | 4QSGA | 173 | 2V9PA | 203 | 1O06A | 233 | 2X5CA |
| 144 | 2V73A | 174 | 3F95A | 204 | 1P9IA | 234 | 2X5RA |
| 145 | 2WVQA | 175 | 1M1QA | 205 | 1PJMA | 235 | 2XF7A |
| 146 | 4NUAA | 176 | 3CSXA | 206 | 1PJNA | 236 | 2XZEQ |
| 147 | 4G97A | 177 | 4FX7A | 207 | 1SVFB | 237 | 3AJBB |
| 148 | 3BRVA | 178 | 3JQOA | 208 | 1T01B | 238 | 3C5TB |
| 149 | 3O12A | 179 | 4L2WA | 209 | 1TQEX | 239 | 3DT5A |
| 150 | 3KUPA | 180 | 3U97A | 210 | 1TTWB | 240 | 3E8YX |

Table B.3: Excluded PISCES proteins due to corrupted or zeroed MSA profiles (121-240).

| No. | Protein | No. | Protein | No. | Protein | No. | Protein |
|---|---|---|---|---|---|---|---|
| 241 | 3FBLA | 271 | 3UKXC | 301 | 4H62V | 331 | 4PNBA |
| 242 | 3GP2B | 272 | 3UL1A | 302 | 4H7RA | 332 | 4PNDA |
| 243 | 3H0TC | 273 | 3V62C | 303 | 4H8MA | 333 | 4PW1A |
| 244 | 3HE4B | 274 | 3V86A | 304 | 4H8OA | 334 | 4QMFA |
| 245 | 3HE5A | 275 | 3VU5B | 305 | 4HB1A | 335 | 4R0RA |
| 246 | 3HE5B | 276 | 3VU6B | 306 | 4HBEA | 336 | 4R80A |
| 247 | 3L9AX | 277 | 3VVIA | 307 | 4HLBA | 337 | 4R8TA |
| 248 | 3LCNC | 278 | 3W8VA | 308 | 4HR1A | 338 | 4RIQC |
| 249 | 3LJMA | 279 | 3W92A | 309 | 4I7ZE | 339 | 4TTLA |
| 250 | 3M6ZA | 280 | 3WKNE | 310 | 4IIKA | 340 | 4UEBB |
| 251 | 3NK4C | 281 | 3WOEB | 311 | 4J4AA | 341 | 4W6YA |
| 252 | 3OWTC | 282 | 3WX4A | 312 | 4JGLA | | |
| 253 | 3P06A | 283 | 3WY9C | 313 | 4JHKC | | |
| 254 | 3PLVC | 284 | 3ZTAA | 314 | 4KVTA | | |
| 255 | 3QKSC | 285 | 4A94C | 315 | 4KYTB | | |
| 256 | 3R46A | 286 | 4BFHA | 316 | 4LOOB | | |
| 257 | 3R4AA | 287 | 4BLQA | 317 | 4M1XA | | |
| 258 | 3RA3B | 288 | 4BPLB | 318 | 4M6BC | | |
| 259 | 3RF3C | 289 | 4C1AA | 319 | 4MGPA | | |
| 260 | 3RKLA | 290 | 4CAYC | 320 | 4MI8C | | |
| 261 | 3S1BA | 291 | 4CU4B | 321 | 4N3BB | | |
| 262 | 3S6PE | 292 | 4CXFB | 322 | 4N3CB | | |
| 263 | 3SHPA | 293 | 4DACA | 323 | 4OGQE | | |
| 264 | 3SJHB | 294 | 4EHQG | 324 | 4OQ9A | | |
| 265 | 3TQ2A | 295 | 4F87A | 325 | 4OYDB | | |
| 266 | 3TWEA | 296 | 4FBWC | 326 | 4OZKA | | |
| 267 | 3TZ1B | 297 | 4FTBD | 327 | 4PC0C | | |
| 268 | 3U4VA | 298 | 4FZ0M | 328 | 4PN8A | | |
| 269 | 3U4ZA | 299 | 4G1AA | 329 | 4PN9A | | |
| 270 | 3UC7A | 300 | 4GVBB | 330 | 4PNAA | | |

Table B.4: Excluded PISCES proteins due to corrupted or zeroed MSA profiles (241-341).

# Appendix C

# Convert datasets to Matlab files

The following bash script was created and used to convert the '.txt' datasets (text files) to '.mat' datasets (Matlab files).

```
#!/bin/bash
# This script finds all the testSet and trainSet files in the current directory,
# converts them to matlab datasets and saves them in the folder mat_datasets.
folderName="mat_datasets"
mkdir "$folderName"

runAll=""
datasets=$(ls | grep -e 'testSet' -e 'trainSet' | grep -v '\.mat')

for ds in $datasets
do
  # echo "$ds"
  loaded=$(echo "$ds" | sed "s:.txt::")
  # echo "$loaded"
  outFile=$(echo "./$folderName/$loaded.mat")
  # echo "$outFile"
  runMat="load␣$ds;␣y␣=␣$loaded(1:end,␣end);␣Z␣=␣$loaded(1:end,␣1:end-1);␣save␣$outFile␣y␣Z␣-v7.3;␣clear;"
  # echo "$runMat"
  runAll="$runAll$runMat␣"
done
runAll="$runAll␣exit;"
# echo "$runAll"
/Applications/MATLAB_R2019a.app/bin/matlab -nodisplay -r "$runAll" > "./$folderName/log.txt"
```

# Appendix D

# CB513 dataset pre-processing

This Python program prepares the CB513 datasets for training the Convolutional Neural Network. It was implemented for the purposes of this dissertation.

```
"""
Uses the DATASETS files to create new datasets for CB513 based
on the specified number of amino acids (ADD_AMINO_ACIDS).
"""

import os, sys

ADD_AMINO_ACIDS = 7 # 7 + 1 + 7 = 15 amino acids per row
DATASETS = ['trainSet0.txt', 'testSet0.txt',
        'trainSet1.txt', 'testSet1.txt',
        'trainSet2.txt', 'testSet2.txt',
        'trainSet3.txt', 'testSet3.txt',
        'trainSet4.txt', 'testSet4.txt',
        'trainSet5.txt', 'testSet5.txt',
        'trainSet6.txt', 'testSet6.txt',
        'trainSet7.txt', 'testSet7.txt',
        'trainSet8.txt', 'testSet8.txt',
        'trainSet9.txt', 'testSet9.txt']

FOLDER_NAME = 'plus{0}_CB513'.format(ADD_AMINO_ACIDS)

if not os.path.exists(FOLDER_NAME):
    os.makedirs(FOLDER_NAME)

protein_name = None
hssp_file = None
CATEGORIES = ['C', 'E', 'H']

def enumerate_cat(labels):
    for i, cat in enumerate(CATEGORIES):
        labels = labels.replace(cat, str(i))
    return labels

def get_zero_lines(num_of_lines):
    if (num_of_lines < 1):
        return ""
    zeros = (("0," * 20) + '\n') * num_of_lines
```

```
    return zeros

for dataset_name in DATASETS:
    # dataset_name = DATASETS[0]
    print('Preparing {0}... Missing hssp files:'.format(dataset_name))
    output_file = './{0}/plus{1}_{2}'.format(FOLDER_NAME, ADD_AMINO_ACIDS, dataset_name)

    with open(dataset_name, 'r') as ds_f:
        with open(output_file, 'w') as out_f:
            line_num = 0
            for line in ds_f:
                if line_num == 0:
                    protein_name = line.split()[0]
                    hssp_file = './msaFiles/{0}.hssp'.format(protein_name)
                    # print(hssp_file)
                    line_num += 1
                elif (line_num == 1):
                    # print(line)
                    line_num += 1
                    continue
                else:
                    labels = (line[:-1]).replace('!', '')
                    label_nums = enumerate_cat(labels)
                    label_index = 0
                    # print(labels)
                    try:
                        with open(hssp_file, 'r') as hssp_f:
                            buf = get_zero_lines(ADD_AMINO_ACIDS)
                            buf_len = ADD_AMINO_ACIDS
                            amino_count = 0
                            for msa_line in hssp_f:
                                if (buf_len > 2 * ADD_AMINO_ACIDS):
                                    temp = buf.replace('\n', '') + label_nums[amino_count]
                                    out_f.write(temp)
                                    out_f.write('\n')
                                    buf = buf.split("\n", 1)[-1]
                                    buf_len -= 1
                                    amino_count += 1

                                modif_line = (msa_line[:-1]).replace(' ', ',')
                                buf = '{0}{1}\n'.format(buf, modif_line)
                                buf_len += 1


                            for i in range(0, ADD_AMINO_ACIDS+1):
                                temp = buf.replace('\n', '') + label_nums[amino_count]
                                out_f.write(temp)
                                out_f.write('\n')
                                buf = buf.split("\n", 1)[-1]
                                buf = buf + get_zero_lines(1)
                                amino_count += 1
                            assert amino_count == len(label_nums)
                    except Exception:
                        print(protein_name)
                    line_num = 0
    print('Done with {0} file!'.format(dataset_name))
```

# Appendix E

# PISCES dataset pre-processing

This Python program prepares the PISCES datasets for training the Convolutional Neural Network. It was implemented for the purposes of this dissertation.

```
"""
Uses the DATASETS files to create new datasets for PISCES with
the specified number of neighboring amino acids (ADD_AMINO_ACIDS).
"""

import os, sys

ADD_AMINO_ACIDS = 7 # 7 + 1 + 7 = 15 amino acids per row
DATASETS = ['trainSet1.txt', 'testSet1.txt',
        'trainSet2.txt', 'testSet2.txt',
        'trainSet3.txt', 'testSet3.txt',
        'trainSet4.txt', 'testSet4.txt',
        'trainSet5.txt', 'testSet5.txt']

FOLDER_NAME = 'plus{0}_PISCES'.format(ADD_AMINO_ACIDS)
MSA_FOLDER = 'msaFiles'

if not os.path.exists(FOLDER_NAME):
    os.makedirs(FOLDER_NAME)

protein_name = None
hssp_file = None
CATEGORIES = ['C', 'E', 'H']

def enumerate_cat(labels):
    for i, cat in enumerate(CATEGORIES):
        labels = labels.replace(cat, str(i))
    return labels

def get_zero_lines(num_of_lines):
    zeros = (("0," * 20) + '\n') * num_of_lines
    return zeros

for dataset_name in DATASETS:
    # dataset_name = DATASETS[0]
    print('Preparing ␣{0}... ␣Missing ␣hssp ␣files:'.format(dataset_name))
    output_file = './{0}/plus{1}_{2}'.format(FOLDER_NAME, ADD_AMINO_ACIDS, dataset_name)
```

```python

with open(dataset_name, 'r') as ds_f:
  with open(output_file, 'w') as out_f:
    line_num = 0
    for line in ds_f:
      if line_num == 0:
        protein_name = line.split()[0]
        hssp_file = './{0}/{1}.hssp'.format(MSA_FOLDER, protein_name)
        # print(hssp_file)
        line_num += 1
      elif (line_num == 1):
        # print(line)
        line_num += 1
        continue
      else:
        labels = line[:-1]
        label_nums = enumerate_cat(labels)
        label_index = 0
        # print(labels)
        try:
          with open(hssp_file, 'r') as hssp_f:
            buf = get_zero_lines(ADD_AMINO_ACIDS)
            buf_len = ADD_AMINO_ACIDS
            amino_count = 0
            for msa_line in hssp_f:
              if (buf_len > 2 * ADD_AMINO_ACIDS):
                temp = buf.replace('\n', '') + label_nums[amino_count]
                out_f.write(temp)
                out_f.write('\n')
                buf = buf.split("\n", 1)[-1]
                buf_len -= 1
                amino_count += 1

              modif_line = (msa_line[:-1]).replace(' ', ',')
              buf = '{0}{1}\n'.format(buf, modif_line)
              buf_len += 1


            for i in range(0, ADD_AMINO_ACIDS+1):
              temp = buf.replace('\n', '') + label_nums[amino_count]
              out_f.write(temp)
              out_f.write('\n')
              buf = buf.split("\n", 1)[-1]
              buf = buf + get_zero_lines(1)
              amino_count += 1
            assert amino_count == len(label_nums)
        except Exception:
          print(protein_name)
        line_num = 0
print('Done with {0} file!'.format(dataset_name))
```

# Appendix F

# Python Implementation

The following code includes the implementation of the Convolutional Neural Network with the Subsampled Hessian Newton method. This program was used to perform all the experiments of this dissertation. Note that commands that begin with '!' should be executed as bash commands. It is highly advised to use the notebook version of the implementation which can be found at [https://gitlab.com/perf.ai/pssp_project/-/blob/master/Notebooks/CNN_HFO.ipynb]. This implementation was based on the Python implementation from [3], however, several modifications have been made to improve the results of the CNN for the PSSP problem.

```
# -*- coding: utf-8 -*-
"""shn_cnn_May22.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1KZtk3v3joX5pAUQJIbpGV9I-kmnddRjV
"""

# plus_var=7
# ds_num=1
# dataset="PISCES"
plus_var=7
ds_num=5
dataset="CB513"

"""## Imports ##"""

# Commented out IPython magic to ensure Python compatibility.
# %load_ext autoreload
# %autoreload 2

# %matplotlib inline

!pip install hdf5storage

import pdb
import numpy as np
```

```
import tensorflow as tf
tf.compat.v1.disable_eager_execution()
import time
import math
import argparse
import os
import scipy.io as sio
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
from tensorflow.python.client import device_lib
import pandas as pd
import hdf5storage

"""## Get data ##"""

test_url="https://gitlab.com/perf.ai/pssp_project/-/raw/master/plus{0}_{1}/mat_datasets/plus{2}_testSet{3}.mat".format(str(plus_var), dataset, str(plus_var), str(ds_num))
train_url="https://gitlab.com/perf.ai/pssp_project/-/raw/master/plus{0}_{1}/mat_datasets/plus{2}_trainSet{3}.mat".format(str(plus_var), dataset, str(plus_var), str(ds_num))
TEST_FILE="plus" + str(plus_var) + "_testSet" + str(ds_num) + ".mat"
TRAIN_FILE="plus" + str(plus_var) + "_trainSet" + str(ds_num) + ".mat"

!echo "$test_url"
!echo "$train_url"

![ -f "$TEST_FILE" ] && echo "$TEST_FILE exist" || wget "$test_url"
![ -f "$TRAIN_FILE" ] && echo "$TRAIN_FILE exist" || wget "$train_url"

!ls

NEIGHBOURS = plus_var # number of amino-acids to add left and right
AMINO_ACID_LEN = 20
WINDOW = 2 * NEIGHBOURS + 1
TOTAL_AMINO_ACIDS = WINDOW * AMINO_ACID_LEN
TOTAL_COLS = TOTAL_AMINO_ACIDS + 1 # plus the secondary structure category
CATEGORIES = 3 # number of different classification categories
TOTAL_COLS

"""## VGG ##"""

"""
Codes are modifeid 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 tensorflow.compat.v1 as tf
# tf.disable_v2_behavior()
# import numpy as np
# import pdb
from tensorflow.keras.applications.vgg16 import VGG16 as vgg16
from tensorflow.keras.applications.vgg19 import VGG19 as vgg19

__all__ = ['VGG11', 'VGG13', 'VGG16','VGG19']

def VGG(feature, num_cls):

```

```
    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):
    for v in cfg:
      if v == 'M':
        x = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='valid')(x)
      else:
        x = tf.keras.layers.Conv2D(
        filters=v,
        kernel_size=[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'],
    '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'],
}

def VGG11(x_images, num_cls):
    feature = make_layers(x_images, cfg['A'])
    return VGG(feature, num_cls)

def VGG13(x_images, num_cls):
    feature = make_layers(x_images, cfg['B'])
    return VGG(feature, num_cls)

def VGG16(x_images, num_cls):
    feature = make_layers(x_images, cfg['D'])
    return VGG(feature, num_cls)

def VGG19(x_images, num_cls):
    feature = make_layers(x_images, cfg['E'])
    return VGG(feature, num_cls)

"""## Net ##"""

# import tensorflow.compat.v1 as tf
# tf.disable_v2_behavior()
# import math
# import pdb
# from tensorflow.python.client import device_lib
# import numpy as np
# from net.vgg import *

```

```
def CNN_4layers(x_image, num_cls, 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)

  with tf.variable_scope('conv2', reuse=reuse) as scope:
    conv = tf.keras.layers.Conv2D(
      filters=64,
      kernel_size=[3, 3],
      padding='SAME',
      activation=tf.nn.relu
    )(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
    )(conv)

  with tf.variable_scope('fully_connected', reuse=reuse) as scope:
    dim =np.prod(conv.shape[1:])
    flat = tf.reshape(conv, [−1, dim])
    outputs = tf.keras.layers.Dense(units=_NUM_CLASSES, name=scope.name)(flat)

  return outputs


  # with tf.variable_scope('conv1', reuse=reuse) as scope:
  #  conv = tf.keras.layers.Conv2D(
  #   filters=32,
  #   kernel_size=[5, 5],
  #   padding='SAME',
  #   activation=tf.nn.relu
  #  )(x_image)
  #  pool = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='valid')(conv)
  #  # N x 16 x 16 x 32

  # with tf.variable_scope('conv2', 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=2, padding='valid')(conv)
  #  # N x 8 x 8 x 64

  # 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=2, padding='valid')(conv)
    # # N x 4 x 4 x 64

    # 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_7layers(x_image, num_cls, 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)
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(conv)

    with tf.variable_scope('conv2', reuse=reuse) as scope:
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(conv)
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(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
        )(conv)
        conv = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )(conv)
```

```
    with tf.variable_scope('fully_connected', reuse=reuse) as scope:
      dim = np.prod(conv.shape[1:])
      flat = tf.reshape(conv, [−1, dim])
      outputs = tf.keras.layers.Dense(units=_NUM_CLASSES, name=scope.name)(flat)

    return outputs

    # with tf.variable_scope('conv1', reuse=reuse) as scope:
    #  conv = tf.keras.layers.Conv2D(
    #   filters=32,
    #   kernel_size=[5, 5],
    #   padding='SAME',
    #   activation=tf.nn.relu
    #  )(x_image)
    #  conv = tf.keras.layers.Conv2D(
    #   filters=32,
    #   kernel_size=[3, 3],
    #   padding='SAME',
    #   activation=tf.nn.relu
    #  )(conv)
    #  pool = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='valid')(conv)
    #  # N x 16 x 16 x 32

    # with tf.variable_scope('conv2', reuse=reuse) as scope:
    #  conv = tf.keras.layers.Conv2D(
    #   filters=64,
    #   kernel_size=[3, 3],
    #   padding='SAME',
    #   activation=tf.nn.relu
    #  )(pool)
    #  conv = tf.keras.layers.Conv2D(
    #   filters=64,
    #   kernel_size=[3, 3],
    #   padding='SAME',
    #   activation=tf.nn.relu
    #  )(conv)
    #  pool = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='valid')(conv)
    #  # N x 8 x 8 x 64

    # 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)
    #  conv = tf.keras.layers.Conv2D(
    #   filters=128,
    #   kernel_size=[3, 3],
    #   padding='SAME',
    #   activation=tf.nn.relu
    #  )(conv)
    #  pool = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='valid')(conv)
    #  # pool = tf.layers.dropout(pool, rate=0.25, name=scope.name)
    #  # N x 4 x 4 x 128

```

```
  # 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
  _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS = dim

  with tf.name_scope('main_params'):
    x = tf.placeholder(tf.float32, shape=[None, _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS], name='input_of_net')
    y = tf.placeholder(tf.float32, shape=[None, _NUM_CLASSES], name='labels')

  # call CNN structure according to string net
  outputs = globals()[net](x, _NUM_CLASSES)
  outputs = tf.identity(outputs, name='output_of_net')

  return (x, y, outputs)

"""## Utilities ##"""

# import numpy as np
# import math
# import scipy.io as sio
# import os
# import math
# import pdb

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, ... number_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)
    mat_contents = hdf5storage.loadmat(filename)
    images, labels = mat_contents['Z'], mat_contents['y']

    labels = labels.reshape(-1)
    images = images.reshape(images.shape[0], -1)

    _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:])))
    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):
    _IMAGE_HEIGHT, _IMAGE_WIDTH, _IMAGE_CHANNELS = dim
    images_shape = [images.shape[0], _IMAGE_CHANNELS, _IMAGE_HEIGHT, _IMAGE_WIDTH]

    # 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('Normalize 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!')

    images = images - mean_tr

    images = images.reshape(images_shape)
    # Tensorflow accepts data shape: B x H x W x C
    images = np.transpose(images, (0, 2, 3, 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}
            )

        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

"""## Newton - CG ##"""

# import pdb
# import tensorflow as tf
# import time
# import numpy as np
# import os
# import math
# from utilities import predict

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, full_batch, val_batch, saver, network, test_network=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

    tf.compat.v1.summary.scalar('loss', self.f)
    merged = tf.compat.v1.summary.merge_all()
    train_writer = tf.compat.v1.summary.FileWriter('./summary/train', 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)

    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)

    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} prered: {:.5f} time: {:.3f}'.\
    format(k, f, gnorm, alpha, actred/prered, self.config._lambda, CGiter, actred, prered, end-start)
print(output_str)
if not self.config.screen_log_only:
  print(output_str, file=log_file)

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)

  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)
        log_file.close()

"""##Set Train Arguments##"""

# Arguments for HFO - PSSP dataset
train_args = ("--optim NewtonCG --GNsize 2048 --C 0.01 --net CNN_4layers --bsize 12288 --iter_max 50 " +
                "--train_set ./" + TRAIN_FILE + " --val_set ./" + TEST_FILE + " --dim " +
                str(WINDOW) + " " + str(AMINO_ACID_LEN) + " 1").split()

# Arguments for SGD - PSSP dataset
# train_args = ("--optim SGD --lr 0.01 --C 0.01 --net CNN_4layers --bsize 256 " +
# "--train_set ./" + TRAIN_FILE + " --val_set ./" + TEST_FILE + " --dim " +
# str(WINDOW) + " " + str(AMINO_ACID_LEN) + " 1").split()

"""##Declare Train Function##"""

# import pdb
# import numpy as np
# import tensorflow as tf
# tf.compat.v1.disable_eager_execution()
# import time
# import math
# import argparse

# from net.net import CNN
# from newton_cg import newton_cg
# from utilities import read_data, predict, ConfigClass, normalize_and_reshape

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='the␣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=None, 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(args=train_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(config, sess, network, full_batch, val_batch, saver, 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)
sess.run(tf.compat.v1.global_variables_initializer())


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)

    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)

        # 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 separat 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)

        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))

    print(output_str)
    if not config.screen_log_only:
        print(output_str, file=log_file)

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.close()

def newton_trainer(config, sess, network, full_batch, val_batch, saver, 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()
    sess.run(init_model(param))
    newton_solver.newton(full_batch, val_batch, saver, network, test_network)


def train_model():
    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()
    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])

    if config.optim in ('SGD', 'Adam'):
      gradient_trainer(
        config, sess, network, full_batch, val_batch, saver, test_network)
    elif config.optim == 'NewtonCG':
      newton_trainer(
        config, sess, network, full_batch, val_batch, saver, test_network=test_network)

"""## Train ##"""

train_model()

"""## Predict ##"""

# Arguments for prediction PSSP dataset
pred_args = ("--bsize 1024 --test_set ./" + TEST_FILE + " --train_set ./" + TRAIN_FILE +
             " --model ./saved_model/model.ckpt --dim " +
             str(WINDOW) + " " + str(AMINO_ACID_LEN) + " 1").split()

test_origin = "https://gitlab.com/perf.ai/pssp_project/-/raw/master/originalData_" + dataset + "/testSet" + str(ds_num) + ".txt"
train_origin = "https://gitlab.com/perf.ai/pssp_project/-/raw/master/originalData_" + dataset + "/trainSet" + str(ds_num) + ".txt"
excluded_proteins = "https://gitlab.com/perf.ai/pssp_project/-/raw/master/originalData_" + dataset + "/excluded_" + dataset + ".
    ↪ txt"
train_origin, test_origin, excluded_proteins

import requests
test_f = requests.get(test_origin)
test_f = test_f.text.split('\n')[0:-1]
train_f = requests.get(train_origin)
train_f = train_f.text.split('\n')[0:-1]
excluded_f = requests.get(excluded_proteins)
excluded_f = excluded_f.text.split('\n')[0:-1]

excluded_f

TEST_PRED_FILE="pred_testSet{0}.txt".format(ds_num)
TRAIN_PRED_FILE="pred_trainSet{0}.txt".format(ds_num)
TEST_PRED_FILE

"""##Declare Predict Methods##"""

def create_output_pred(pred_test, pred_train):
    pred = pred_test.astype(int)
    labels = ['C', 'E', 'H']
    counter = 0
    outFileName = TEST_PRED_FILE
    with open(outFileName, 'w') as out_file:
        for line in range(0, len(test_f)//3):
            protein_name = test_f[line*3]
            if (protein_name in excluded_f):
```

```
                # print(protein_name)
                continue
            primary_structure = test_f[line*3+1].replace('!', '')
            secondary_structure = test_f[line*3+2].replace('!', '')
            prediction = ""
            for c in secondary_structure:
                if (c != '!'):
                    prediction = prediction + labels[pred[counter]]
                    counter += 1
                # else:
                # prediction = prediction + c
            # print("Protein name: " + protein_name)
            # print("Actual: " + secondary_structure)
            # print("Prediction: " + prediction)
            out_file.write(protein_name + "\n")
            out_file.write(primary_structure + "\n")
            out_file.write(secondary_structure + "\n")
            out_file.write(prediction + "\n")
    pred = pred_train.astype(int)
    counter = 0
    outFileName = TRAIN_PRED_FILE
    with open(outFileName, 'w') as out_file:
        for line in range(0, len(train_f)//3):
            protein_name = train_f[line*3]
            if (protein_name in excluded_f):
                # print(protein_name)
                continue
            primary_structure = train_f[line*3+1].replace('!', '')
            secondary_structure = train_f[line*3+2].replace('!', '')
            prediction = ""
            for c in secondary_structure:
                if (c != '!'):
                    prediction = prediction + labels[pred[counter]]
                    counter += 1
                # else:
                # prediction = prediction + c
            # print("Protein name: " + protein_name)
            # print("Actual: " + secondary_structure)
            # print("Prediction: " + prediction)
            out_file.write(protein_name + "\n")
            out_file.write(primary_structure + "\n")
            out_file.write(secondary_structure + "\n")
            out_file.write(prediction + "\n")

# import tensorflow as tf
# tf.compat.v1.disable_eager_execution()
# from utilities import predict, read_data, normalize_and_reshape
# from net.net import CNN
# import numpy as np
# import argparse
# import pdb

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=None, type=str)
```

```
    parser.add_argument('−−train_set', dest='train_set',
                help='provide␣the␣directory␣of␣.mat␣file␣for␣training',
                default=None, 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(args=pred_args)
    return args

def predict_model():
    args = parse_args()

    sess_config = tf.compat.v1.ConfigProto()
    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)

```

```
    train_batch, num_cls, _ = read_data(args.train_set, dim=args.dim, label_enum=label_enum)
    train_batch[0], _ = normalize_and_reshape(train_batch[0], dim=args.dim, mean_tr=mean_tr)

    avg_loss_train, avg_acc_train, results_train = predict(sess, network, train_batch, args.bsize)
    # convert results back to the original labels
    inverse_map = dict(zip(np.arange(num_cls), label_enum))
    results_train = np.expand_dims(results_train, axis=1)
    results_train = np.apply_along_axis(lambda x: inverse_map[x[0]], axis=1, arr=results_train)

    create_output_pred(results, results_train)

  print('In test phase, average loss: {:.3f} | average accuracy: {:.3f}%'.\
    format(avg_loss, avg_acc*100))

  print('In train phase, average loss: {:.3f} | average accuracy: {:.3f}%'.\
    format(avg_loss_train, avg_acc_train*100))

"""##Run Predict and Display output##"""

predict_model()

# !cat "$TEST_PRED_FILE"

# !cat "$TRAIN_PRED_FILE"
```

# Appendix G

# Ensembles Program

This Python program was used to combine the results from multiple trained models using the ensembles method. It was provided by Dionysiou [24].

```
from numpy import *
import string as string
import sys


class Ensembles:
    def run(filenames, windowSize, ensemble, outPred, outSOV, outWeka):
        f = open(outPred, "w")
        files = open(filenames, "r").readlines()
        files = [w.replace('\n', '') for w in files]
        files = [open(i, "r") for i in files]
        i = 0
        LABELS = ['C', 'E', 'H', '!']
        if ensemble == 1:
            for rows in zip(*files):
                if i == 3:
                    for j in range(0, len(rows[0].translate(str.maketrans('', '', string.whitespace))), 1):
                        count = [0, 0, 0, 0]
                        for k in range(0, len(rows), 1):
                            if rows[k][j] == 'C':
                                count[0] += 1
                            elif rows[k][j] == 'E':
                                count[1] += 1
                            elif rows[k][j] == 'H':
                                count[2] += 1
                            else:
                                count[3] += 1
                        f.write(LABELS[argmax(count)])
                    f.write('\n')
                    i = 0
                else:
                    f.write(rows[0])
                    i += 1
            f.close()
        else:
            print('ERROR!!! Invalid ensemble option.')

```

```
# count accuracy
f = open(outPred, "r")
lines = f.readlines()
f.close()
count = 0
countall = 0
for i in range(0, len(lines), 4):
    for j in range(0, len(lines[i + 2].translate(str.maketrans('', '', string.whitespace))), 1):
        if lines[i + 2][j] == lines[i + 3][j]:
            count += 1
        countall += 1

print('Accuracy: ' + str(float(count) / float(countall) * 100) + '%')

# Confusion Matrix
countHH = 0
countHE = 0
countHC = 0
countEH = 0
countEE = 0
countEC = 0
countCH = 0
countCE = 0
countCC = 0
countH = 0
countE = 0
countC = 0
countHp = 0
countEp = 0
countCp = 0
for i in range(0, len(lines), 4):
    for j in range(0, len(lines[i + 2].translate(str.maketrans('', '', string.whitespace))), 1):
        if lines[i + 2][j] == 'H' and lines[i + 3][j] == 'H':
            countHH += 1
        elif lines[i + 2][j] == 'H' and lines[i + 3][j] == 'E':
            countHE += 1
        elif lines[i + 2][j] == 'H' and lines[i + 3][j] == 'C':
            countHC += 1
        elif lines[i + 2][j] == 'E' and lines[i + 3][j] == 'H':
            countEH += 1
        elif lines[i + 2][j] == 'E' and lines[i + 3][j] == 'E':
            countEE += 1
        elif lines[i + 2][j] == 'E' and lines[i + 3][j] == 'C':
            countEC += 1
        elif lines[i + 2][j] == 'C' and lines[i + 3][j] == 'H':
            countCH += 1
        elif lines[i + 2][j] == 'C' and lines[i + 3][j] == 'E':
            countCE += 1
        elif lines[i + 2][j] == 'C' and lines[i + 3][j] == 'C':
            countCC += 1

        '''if lines[i + 2][j] == 'H':
            countH += 1
        elif lines[i + 2][j] == 'E':
            countE += 1
        elif lines[i + 2][j] == 'C':
            countC += 1
```

```
            if lines[i + 3][j] == 'H':
                countHp += 1
            elif lines[i + 3][j] == 'E':
                countEp += 1
            elif lines[i + 3][j] == 'C':
                countCp += 1'''

    print('\n\t\tCONFUSION␣MATRIX\n')
    print('{0:10}{1:10}{2:10}{3:10}'.format('␣', 'H', 'E', 'C'))
    print('{0:1}{1:10d}{2:10d}{3:10d}'.format('H', countHH, countHE, countHC))
    print('{0:1}{1:10d}{2:10d}{3:10d}'.format('E', countEH, countEE, countEC))
    print('{0:1}{1:10d}{2:10d}{3:10d}'.format('C', countCH, countCE, countCC))

    # SOV input file
    # f = open(outPred, "r")
    f1 = open(outSOV, "w")
    # lines = f.readlines()
    # f.close()

    for i in range(0, len(lines), 4):
        f1.write('>OSEQ\n')
        f1.write(lines[i + 2])
        f1.write('>PSEQ\n')
        f1.write(lines[i + 3])
        f1.write('>AA\n')
        f1.write(lines[i + 1])
    f1.close()

    # weka input file
    f1 = open(outWeka, "w")
    f1.write('@RELATION␣secondary_structure\n\n')
    for i in range(0, windowSize * 2 - 1, 1):
        f1.write('@ATTRIBUTE␣aminoacid' + str(i) + '␣{C,E,H,0.0}\n')
    f1.write('@ATTRIBUTE␣output␣{C,E,H}\n')
    f1.write('\n@DATA\n')

    leadingzeros = zeros((1, (windowSize - 1)))
    for i in range(3, len(lines), 4):
        line = leadingzeros
        line = append(line, list(lines[i].rstrip()))
        line = append(line, leadingzeros)
        for j in range(0, len(lines[i].rstrip()), 1):
            for k in range(0, windowSize * 2 - 1, 1):
                f1.write(str(line[j + k]) + ',')
            f1.write(lines[i - 1].rstrip()[j] + '\n')

    f1.close()

files = sys.argv[1].replace(',', '')
run(files, int(sys.argv[2].replace(',', '')), int(sys.argv[3].replace(',', '')), sys.argv[4].replace(',', ''),
    sys.argv[5].replace(',', ''), sys.argv[6])
# print('\nEnd of ensembles script\n')
```

# Appendix H

# External Rules Program

This Python program was used to apply the external rules filtering. It was provided by Dionysiou [24].

```
import sys

class externalRules:
    def applyRules(filename, outSOV, outPred):
        f = open(filename, "r")
        lines = f.readlines()
        f.close()
        f = open(outSOV, "w")
        f1 = open(outPred, "w")

        for i in range(0, len(lines), 4):
            f1.write(lines[i])
            f1.write(lines[i + 1])
            f1.write(lines[i + 2])
            f.write(">OSEQ\n")
            f.write(lines[i + 2])
            f.write(">PSEQ\n")
            j = 0
            lines[i + 3] = list(lines[i + 3].translate({ord(c):'' for c in ' ␣\n\t\r'}))
            # print(len(lines[i + 3]))
            while j < len(lines[i + 3]):
                if len(lines[i + 3]) − j >= 4:
                    if lines[i + 3][j] == 'H' and lines[i + 3][j + 1] == 'E' and lines[i + 3][j + 2] == 'E' and \
                                lines[i + 3][j + 3] == 'H':
                        lines[i + 3][j] = 'H'
                        lines[i + 3][j + 1] = 'H'
                        lines[i + 3][j + 2] = 'H'
                        lines[i + 3][j + 3] = 'H'
                        j += 4
                        continue
                    if lines[i + 3][j] != 'H' and lines[i + 3][j + 1] == 'H' and lines[i + 3][j + 2] == 'H' and \
                                lines[i + 3][j + 3] != 'H':
                        lines[i + 3][j + 1] = 'C'
                        lines[i + 3][j + 2] = 'C'
                        j += 4
                        continue
                if len(lines[i + 3]) − j >= 3:
```

```
                    if lines[i + 3][j] == 'H' and lines[i + 3][j + 1] == 'E' and lines[i + 3][j + 2] == 'H':
                        lines[i + 3][j + 1] = 'H'
                        j += 3
                        continue
                j += 1

            if lines[i + 3][0] == 'E' and lines[i + 3][1] != 'E':
                f.write("C")
                f1.write("C")
            elif lines[i + 3][0] == 'H' and lines[i + 3][1] != 'H':
                f.write("C")
                f1.write("C")
            else:
                f.write(lines[i + 3][0])
                f1.write(lines[i + 3][0])

            for j in range(1, len(lines[i + 3]) - 1):
                if lines[i + 3][j - 1] != 'E' and lines[i + 3][j] == 'E' and lines[i + 3][j + 1] != 'E':
                    f.write("C")
                    f1.write("C")
                    continue
                elif lines[i + 3][j - 1] != 'H' and lines[i + 3][j] == 'H' and lines[i + 3][j + 1] != 'H':
                    f.write("C")
                    f1.write("C")
                    continue
                f.write(lines[i + 3][j])
                f1.write(lines[i + 3][j])

            if lines[i + 3][len(lines[i + 3]) - 1] == 'E' and lines[i + 3][len(lines[i + 3]) - 2] != 'E':
                f.write("C")
                f1.write("C")
            elif lines[i + 3][len(lines[i + 3]) - 1] == 'H' and lines[i + 3][len(lines[i + 3]) - 2] != 'H':
                f.write("C")
                f1.write("C")
            else:
                f.write(lines[i + 3][len(lines[i + 3]) - 1])
                f1.write(lines[i + 3][len(lines[i + 3]) - 1])

            f.write('\n')
            f1.write('\n')
            f.write(">AA\n")
            f.write(lines[i + 1])

    applyRules(sys.argv[1].replace(',', ''), sys.argv[2].replace(',', ''), sys.argv[3])
    # print('End of external rules script\n')
```

# Appendix I

# SOV calculation

To calculate the SOV score the two following C programs were used. Both were provided by Dionysiou [24].

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[]){
  FILE *fp=fopen(argv[1], "r");
  FILE *out;
  char *line = NULL;
  size_t len = 0;
  ssize_t read;
  fclose(fopen("resultSOV.txt","w"));

  if (fp == NULL)
    exit(0);
  system("cc ./q3_sov_scripts/sov.c -o ./q3_sov_scripts/sov -lm");
  while ((read = getline(&line, &len, fp)) != -1) {
    out=fopen("SOVinput.txt", "w");
    if (out == NULL)
      exit(0);
    fprintf(out,"%s", line);
    getline(&line, &len, fp);
    fprintf(out,"%s", line);
    getline(&line, &len, fp);
    fprintf(out,"%s", line);
    getline(&line, &len, fp);
    fprintf(out,"%s", line);
    getline(&line, &len, fp);
    fprintf(out,"%s", line);
    getline(&line, &len, fp);
    fprintf(out,"%s", line);
    fclose(out);

    system("./q3_sov_scripts/sov SOVinput.txt >>resultSOV.txt");
  }

  free(line);
  fclose(fp);
  return 0;
```

```
}
```

```
/*-----------------------------------------------------------
/
/ Program: sov.c
/
/ Secondary structure prediction accuracy evaluation
/
/ SOV (Segment OVerlap) measure
/
/ Copyright by Adam Zemla (11/16/1996)
/ Email: adamz@llnl.gov
/
/-----------------------------------------------------------
/
/ Compile: cc sov.c -o sov -lm
/
/-----------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define MAXRES 5000

typedef struct {
   int input;
   int order;
   int q3_what;
   int sov_what;
   int sov_method;
   float sov_delta;
   float sov_delta_s;
   int sov_out;
   char fname[100];
} parameters;

char *letter_AA="ARNDCQEGHILKMFPSTWYV-?X"; /* 23 chars */

void default_parameters(parameters *);
int read_aa_osec_psec(char[MAXRES], char[MAXRES], char[MAXRES],
                      parameters *, char*);
float sov(int, char[MAXRES], char[MAXRES], parameters *);
float q3(int, char[MAXRES], char[MAXRES], parameters *);
int check_aa(char, char*, int);

int main(int argc, char *argv[])
{
   int i, n_aa, sov_method;
   char c, aa[MAXRES], osec[MAXRES], psec[MAXRES];
   parameters pdata;
   float out0, out1, out2, out3;

   if(argc<2){
      printf(" Usage: sov <input_data>\n");
      printf(" HELP:  sov -h\n");
      exit(0);
   }
   if(!strncmp(argv[1],"-h\0",2) ||
```

```
   !strncmp(argv[1],"help\0",5) ||
   !strncmp(argv[1],"−help\0",6)) {
  system("more␣./README.sov");
  printf("\n");
  exit(0);
}

default_parameters(&pdata);

strcpy(pdata.fname,argv[1]);

n_aa=read_aa_osec_psec(aa,osec,psec,&pdata,letter_AA);

if(pdata.input==1) {
  n_aa=read_aa_osec_psec(aa,osec,psec,&pdata,letter_AA);
}

if(pdata.order==1) {
  for(i=0;i<n_aa;i++) {
    c=osec[i];
    osec[i]=psec[i];
    psec[i]=c;
  }
}

if(n_aa<=0) {
  printf("\n␣ERROR!␣There␣is␣no␣’AA␣OSEC␣PSEC’␣data␣in␣submited␣prediction.");
  printf("\n␣␣␣␣␣␣␣␣The␣submission␣should␣contain␣an␣observed␣and␣predicted");
  printf("\n␣␣␣␣␣␣␣␣secondary␣structure␣in␣COLUMN␣format.\n");
  exit(0);
}

printf("\n\n␣SECONDARY␣STRUCTURE␣PREDICTION");
printf("\n␣NUMBER␣OF␣RESIDUES␣PREDICTED:␣LENGTH␣=␣%d",n_aa);
printf("\n␣AA␣␣OSEC␣␣PSEC␣␣␣␣␣NUM");
for(i=0;i<n_aa;i++) {
  printf("\n␣␣%1c␣␣␣%1c␣␣␣␣␣%1c␣␣␣␣␣␣%4d",aa[i],osec[i],psec[i],i+1);
}
printf("\n␣−−−−−−−−−−−−−−−−−−−−−−−−\n");
printf("\n␣SECONDARY␣STRUCTURE␣PREDICTION␣ACCURACY␣EVALUATION.␣␣N_AA␣=␣%4d\n",n_aa);
if(pdata.sov_out>=1) {
  printf("\n␣SOV␣parameters:␣␣␣DELTA␣=␣%5.2f␣␣␣DELTA−S␣=␣%5.2f\n",
            pdata.sov_delta,
            pdata.sov_delta_s);
}

printf("\n␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALL␣␣␣␣HELIX␣␣␣STRAND␣␣␣␣␣COIL\n");

pdata.q3_what=0;
out0=q3(n_aa,osec,psec,&pdata);
pdata.q3_what=1;
out1=q3(n_aa,osec,psec,&pdata);
pdata.q3_what=2;
out2=q3(n_aa,osec,psec,&pdata);
pdata.q3_what=3;
out3=q3(n_aa,osec,psec,&pdata);
printf("\n␣Q3␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣␣␣%6.1f␣␣␣%6.1f␣␣␣%6.1f␣␣␣%6.1f",
```

```
                                out0*100.0,out1*100.0,out2*100.0,out3*100.0);
    printf("\n");

    sov_method=pdata.sov_method;

    if(sov_method!=0) pdata.sov_method=1;

    if(pdata.sov_method==1) {
      pdata.sov_what=0;
      out0=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=1;
      out1=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=2;
      out2=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=3;
      out3=sov(n_aa,osec,psec,&pdata);
      printf("\n␣SOV␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣␣␣%6.1f␣␣␣%6.1f␣␣␣%6.1f␣␣␣%6.1f",
                                out0*100.0,out1*100.0,out2*100.0,out3*100.0);
      printf("\n");
    }

    if(sov_method!=1) pdata.sov_method=0;

    if(pdata.sov_method==0) {
      pdata.sov_delta=1.0;

      pdata.sov_what=0;
      out0=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=1;
      out1=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=2;
      out2=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=3;
      out3=sov(n_aa,osec,psec,&pdata);
      printf("\n␣SOV␣(1994␣JMB.␣[delta=50])␣:␣␣␣%6.1f␣␣␣%6.1f␣␣␣%6.1f␣␣␣%6.1f",
                                out0*100.0,out1*100.0,out2*100.0,out3*100.0);

      pdata.sov_delta=0.0;

      pdata.sov_what=0;
      out0=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=1;
      out1=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=2;
      out2=sov(n_aa,osec,psec,&pdata);
      pdata.sov_what=3;
      out3=sov(n_aa,osec,psec,&pdata);
      printf("\n␣SOV␣(1994␣JMB.␣[delta=0])␣␣:␣␣␣%6.1f␣␣␣%6.1f␣␣␣%6.1f␣␣␣%6.1f",
                                out0*100.0,out1*100.0,out2*100.0,out3*100.0);

      printf("\n");
    }

    printf("\n␣-----------------------\n");

    exit(0);
  }
```

```
/*-----------------------------------------------------------
/
/ check_aa - checks an amino acid
/
/-----------------------------------------------------------*/
int check_aa(char token, char* letter, int n)
{
  int i;

  for(i=0;i<n;i++) {
    if(letter[i]==token)
      return i;
  }
  return n;
}

/*-----------------------------------------------------------
/
/ read_aa_osec_psec - read secondary structure segments file
/
/-----------------------------------------------------------*/
int read_aa_osec_psec(char aa[MAXRES], char sss1[MAXRES],
                      char sss2[MAXRES], parameters *pdata, char* letter)
{
  int i, j, n_aa, n_aa_1, n_aa_2, n_aa_3, f_seq, alt_c, alt_e, alt_h;
  float x;
  char line[MAXRES],keyword[MAXRES],first[MAXRES],second[MAXRES],third[MAXRES],junk[MAXRES];
  FILE *fp;

  alt_c=0;
  alt_e=0;
  alt_h=0;

  if((fp = fopen(pdata->fname,"r"))==NULL) {
    printf("\n#␣error␣opening␣file␣%s␣for␣read\n\n",pdata->fname);
    exit(0);
  }

  f_seq=0;
  pdata->input=0;
  n_aa=0;
  n_aa_1=0;
  n_aa_2=0;
  n_aa_3=0;

  while (fgets(line, MAXRES, fp) != NULL) {
    strcpy(keyword,"␣␣␣");
    strcpy(first,"␣␣␣");
    strcpy(second,"␣␣␣");
    strcpy(third,"␣␣␣");
    strcpy(junk,"␣␣␣");
    i=0;
    j=0;
    while(line[i] == '␣' && line[i] != '\n' && line[i] != '\0' && i<MAXRES) i++;
    if(i<MAXRES) {
      j=i;
```

```
229         while(line[i] != '␣' && line[i] != '\n' && line[i] != '\0' && i<MAXRES) i++;
230       }
231       j=i−j;
232       if(j<MAXRES && j>0) {
233         sscanf(line,"%s",keyword);
234       }
235       if(!strncmp(keyword,"#",1)) {}
236       else if(!strncmp(keyword,"−−−−−",5)) {}
237       else if(!strncmp(keyword,"NUMBER\0",7)) {}
238       else if(!strncmp(keyword,"SECONDARY\0",10)) {}
239       else if(!strncmp(keyword,"END\0",4) && f_seq==0) {
240         fclose(fp);
241         return n_aa;
242       }
243       else if(!strncmp(keyword,"AA−OSEC−PSEC\0",13)) {
244         printf("%s", line);
245         sscanf(line,"%s␣%s",keyword,first);
246         strcpy(pdata−>fname,first);
247         pdata−>input=1;
248       }
249       else if(line[0] == '\n' || !strncmp(keyword,"␣␣␣\0",4)) {}
250       else if(!strncmp(keyword,"AA\0",3) && f_seq==0) {
251         sscanf(line,"%s␣%s␣%s",keyword,first,second);
252         if(!strncmp(keyword,"AA\0",3) &&
253             !strncmp(first,"PSEC\0",5) && !strncmp(second,"OSEC\0",5)) {
254           pdata−>order=1;
255         }
256       }
257       else if(!strncmp(keyword,"SOV−DELTA\0",10)) {
258         printf("%s", line);
259         sscanf(line,"%s␣%f",keyword,&x);
260         pdata−>sov_delta=x;
261       }
262       else if(!strncmp(keyword,"SOV−DELTA−S\0",12)) {
263         printf("%s", line);
264         sscanf(line,"%s␣%f",keyword,&x);
265         pdata−>sov_delta_s=x;
266       }
267       else if(!strncmp(keyword,"SOV−METHOD\0",9)) {
268         printf("%s", line);
269         sscanf(line,"%s␣%d",keyword,&i);
270         pdata−>sov_method=i;
271       }
272       else if(!strncmp(keyword,"SOV−OUTPUT\0",9)) {
273         printf("%s", line);
274         sscanf(line,"%s␣%d",keyword,&i);
275         pdata−>sov_out=i;
276       }
277       else if(line[0]=='>') {
278         printf("%s", line);
279         if(f_seq<2) n_aa=0;
280         f_seq++;
281       }
282       else if(f_seq==0) {
283         if(j>1) {
284           if(!strncmp(keyword,"SSP\0",4)) {
285             sscanf(line,"%s␣%s␣%s␣%s",keyword,junk,first,second,third);
```

```
286            }
287            else {
288              printf("\n␣ERROR!␣(line:␣%d)␣Check␣COLUMN␣format␣of␣your␣prediction!\n",n_aa+1);
289              fclose(fp);
290              exit(0);
291            }
292          }
293          else {
294            sscanf(line,"%s␣%s␣%s",first,second,third);
295          }
296          aa[n_aa]=first[0];
297          sss1[n_aa]=second[0];
298          sss2[n_aa]=third[0];
299          if(check_aa(aa[n_aa],letter,23)==23) {
300            printf("\n#␣ERROR!\n%s",line);
301            printf("\n#␣ERROR!␣(line:␣%d)␣Check␣amino␣acid␣code␣␣%c\n",n_aa+1,aa[n_aa]);
302            fclose(fp);
303            exit(0);
304          }
305          if(sss1[n_aa]=='␣' || sss2[n_aa]=='␣') {
306            printf("\n#␣ERROR!\n%s",line);
307            printf("\n#␣ERROR!␣(line:␣%d)␣Check␣secondary␣structure␣code\n",n_aa+1);
308            fclose(fp);
309            exit(0);
310          }
311          if(sss1[n_aa]=='L' || sss1[n_aa]=='T' || sss1[n_aa]=='S') {
312            if(alt_c==0) {
313              printf("#␣WARNING!␣(line:␣%d)␣The␣'%c'␣characters␣are␣interpreted␣as␣'C'␣(coil)\n",n_aa+1,sss1[n_aa]);
314              alt_c=1;
315            }
316            sss1[n_aa]='C';
317          }
318          if(sss1[n_aa]=='B') {
319            if(alt_e==0) {
320              printf("#␣WARNING!␣(line:␣%d)␣The␣'%c'␣characters␣are␣interpreted␣as␣'E'␣(strand)\n",n_aa+1,sss1[n_aa]);
321              alt_e=1;
322            }
323            sss1[n_aa]='E';
324          }
325          if(sss1[n_aa]=='G' || sss1[n_aa]=='I') {
326            if(alt_h==0) {
327              printf("#␣WARNING!␣(line:␣%d)␣The␣'%c'␣characters␣are␣interpreted␣as␣'H'␣(helix)\n",n_aa+1,sss1[n_aa]);
328              alt_h=1;
329            }
330            sss1[n_aa]='H';
331          }
332          if(sss2[n_aa]=='L' || sss2[n_aa]=='T' || sss2[n_aa]=='S') {
333            if(alt_c==0) {
334              printf("#␣WARNING!␣(line:␣%d)␣The␣'%c'␣characters␣are␣interpreted␣as␣'C'␣(coil)\n",n_aa+1,sss2[n_aa]);
335              alt_c=1;
336            }
337            sss2[n_aa]='C';
338          }
339          if(sss2[n_aa]=='B') {
340            if(alt_e==0) {
341              printf("#␣WARNING!␣(line:␣%d)␣The␣'%c'␣characters␣are␣interpreted␣as␣'E'␣(strand)\n",n_aa+1,sss2[n_aa]);
342              alt_e=1;
```

```
343            }
344            sss2[n_aa]='E';
345          }
346          if(sss2[n_aa]=='G' || sss2[n_aa]=='I') {
347            if(alt_h==0) {
348              printf("#␣WARNING!␣(line:␣%d)␣The␣'%c'␣characters␣are␣interpreted␣as␣'H'␣(helix)\n",n_aa+1,sss2[n_aa]);
349              alt_h=1;
350            }
351            sss2[n_aa]='H';
352          }
353          if(sss1[n_aa]!='C' && sss1[n_aa]!='E' && sss1[n_aa]!='H') {
354            printf("\n#␣ERROR!\n%s",line);
355            printf("\n#␣ERROR!␣(line:␣%d)␣Check␣secondary␣structure␣code␣␣%c\n",n_aa+1,sss1[n_aa]);
356            fclose(fp);
357            exit(0);
358          }
359          if(sss2[n_aa]!='C' && sss2[n_aa]!='E' && sss2[n_aa]!='H') {
360            printf("\n#␣ERROR!\n%s",line);
361            printf("\n#␣ERROR!␣(line:␣%d)␣Check␣secondary␣structure␣code␣␣%c\n",n_aa+1,sss2[n_aa]);
362            fclose(fp);
363            exit(0);
364          }
365          n_aa++;
366          if(n_aa>=MAXRES) {
367            printf("\n#␣ERROR!␣Check␣number␣of␣amino␣acid␣lines.␣(MAX␣=␣%d)\n\n",MAXRES);
368            fclose(fp);
369            exit(0);
370          }
371        }
372        else if(f_seq==1) {
373          i=0;
374          while(line[i] != '\n') {
375            if(line[i] != '␣' && line[i] != '\t' && line[i] != '\0' &&
376                line[i] != '\a' && line[i] != '\b' && line[i] != '\f' &&
377                line[i] != '\r' && line[i] != '\v' && i<MAXRES) {
378              aa[n_aa]='X';
379              sss1[n_aa]=line[i];
380              if(sss1[n_aa]=='L' || sss1[n_aa]=='T' || sss1[n_aa]=='S') {
381                if(alt_c==0) {
382                  printf("#␣WARNING!␣The␣'%c'␣characters␣are␣interpreted␣as␣'C'␣(coil)\n",sss1[n_aa]);
383                  alt_c=1;
384                }
385                sss1[n_aa]='C';
386              }
387              if(sss1[n_aa]=='B') {
388                if(alt_e==0) {
389                  printf("#␣WARNING!␣The␣'%c'␣characters␣are␣interpreted␣as␣'E'␣(strand)\n",sss1[n_aa]);
390                  alt_e=1;
391                }
392                sss1[n_aa]='E';
393              }
394              if(sss1[n_aa]=='G' || sss1[n_aa]=='I') {
395                if(alt_h==0) {
396                  printf("#␣WARNING!␣The␣'%c'␣characters␣are␣interpreted␣as␣'H'␣(helix)\n",sss1[n_aa]);
397                  alt_h=1;
398                }
399                sss1[n_aa]='H';
```

```
400             }
401             if(sss1[n_aa]!='C' && sss1[n_aa]!='E' && sss1[n_aa]!='H') {
402               printf("\n#␣ERROR!\n%s",line);
403               printf("\n#␣ERROR!␣Check␣secondary␣structure␣code:␣%c\n",sss1[n_aa]);
404               fclose(fp);
405               exit(0);
406             }
407             n_aa++;
408             if(n_aa>=MAXRES) {
409               printf("\n#␣ERROR!␣Check␣number␣of␣residues.␣(MAX␣=␣%d)\n\n",MAXRES);
410               fclose(fp);
411               exit(0);
412             }
413           }
414           i++;
415         }
416         n_aa_1=n_aa;
417       }
418       else if(f_seq==2) {
419         i=0;
420         while(line[i] != '\n') {
421           if(line[i] != '␣' && line[i] != '\t' && line[i] != '\0' &&
422              line[i] != '\a' && line[i] != '\b' && line[i] != '\f' &&
423              line[i] != '\r' && line[i] != '\v' && i<MAXRES) {
424             aa[n_aa]='X';
425             sss2[n_aa]=line[i];
426             if(sss2[n_aa]=='L' || sss2[n_aa]=='T' || sss2[n_aa]=='S') {
427               if(alt_c==0) {
428                 printf("#␣WARNING!␣The␣'%c'␣characters␣are␣interpreted␣as␣'C'␣(coil)\n",sss2[n_aa]);
429                 alt_c=1;
430               }
431               sss2[n_aa]='C';
432             }
433             if(sss2[n_aa]=='B') {
434               if(alt_e==0) {
435                 printf("#␣WARNING!␣The␣'%c'␣characters␣are␣interpreted␣as␣'E'␣(strand)\n",sss2[n_aa]);
436                 alt_e=1;
437               }
438               sss2[n_aa]='E';
439             }
440             if(sss2[n_aa]=='G' || sss2[n_aa]=='I') {
441               if(alt_h==0) {
442                 printf("#␣WARNING!␣The␣'%c'␣characters␣are␣interpreted␣as␣'H'␣(helix)\n",sss2[n_aa]);
443                 alt_h=1;
444               }
445               sss2[n_aa]='H';
446             }
447             if(sss2[n_aa]!='C' && sss2[n_aa]!='E' && sss2[n_aa]!='H') {
448               printf("\n#␣ERROR!\n%s",line);
449               printf("\n#␣ERROR!␣Check␣secondary␣structure␣code:␣%c\n",sss2[n_aa]);
450               fclose(fp);
451               exit(0);
452             }
453             n_aa++;
454             if(n_aa>=MAXRES) {
455               printf("\n#␣ERROR!␣Check␣number␣of␣residues.␣(MAX␣=␣%d)\n\n",MAXRES);
456               fclose(fp);
```

```
457                 exit(0);
458               }
459             }
460             i++;
461           }
462           n_aa_2=n_aa;
463         }
464         else if(f_seq==3) {
465           i=0;
466           while(line[i] != '\n') {
467             if(line[i] != '␣' && line[i] != '\t' && line[i] != '\0' &&
468                 line[i] != '\a' && line[i] != '\b' && line[i] != '\f' &&
469                 line[i] != '\r' && line[i] != '\v' && i<MAXRES) {
470               aa[n_aa_3]=line[i];
471               if(check_aa(aa[n_aa_3],letter,23)==23) {
472                 printf("\n#␣ERROR!\n%s",line);
473                 printf("\n#␣ERROR!␣(N_res:␣%d)␣Check␣amino␣acid␣code␣␣%c\n",n_aa_3+1,aa[n_aa_3]);
474                 fclose(fp);
475                 exit(0);
476               }
477               n_aa_3++;
478               if(n_aa_3>=MAXRES) {
479                 printf("\n#␣ERROR!␣Check␣number␣of␣residues.␣(MAX␣=␣%d)\n\n",MAXRES);
480                 fclose(fp);
481                 exit(0);
482               }
483             }
484             i++;
485           }
486         }
487       }
488       if(n_aa_1!=n_aa_2) {
489         printf("\n#␣ERROR!␣Check␣format␣of␣your␣submission.");
490         printf("\n#␣␣␣␣␣␣␣Different␣length␣of␣observed␣and␣predicted␣structures.\n");
491         fclose(fp);
492         exit(0);
493       }
494       return n_aa;
495     }
496
497     /*-------------------------------------------------------------
498     /
499     / default_parameters - default parameters for SOV program
500     /
501     /-------------------------------------------------------------*/
502     void default_parameters(parameters *pdata)
503     {
504       pdata->input=0;
505       pdata->order=0;
506       pdata->sov_method=1; // 0 - SOV definition (1994 JMB.) , 1 - SOV definition (1999 Proteins)
507       pdata->sov_delta=1.0;
508       pdata->sov_delta_s=0.5;
509       pdata->sov_out=0;
510
511       return;
512     }
513
```

```
/*-------------------------------------------------------------
/
/ sov - evaluate SSp by the Segment OVerlap quantity (SOV)
/ Input: secondary structure segments
/
/-------------------------------------------------------------*/
float sov(int n_aa, char sss1[MAXRES], char sss2[MAXRES], parameters *pdata)
{
  int i, k, length1, length2, beg_s1, end_s1, beg_s2, end_s2;
  int j1, j2, k1, k2, minov, maxov, d, d1, d2, n, multiple;
  char s1, s2, sse[3];
  float out;
  double s, x;

  sse[0]='#';
  sse[1]='#';
  sse[2]='#';

  if(pdata->sov_what==0) {
    sse[0]='H';
    sse[1]='E';
    sse[2]='C';
  }
  if(pdata->sov_what==1) {
    sse[0]='H';
    sse[1]='H';
    sse[2]='H';
  }
  if(pdata->sov_what==2) {
    sse[0]='E';
    sse[1]='E';
    sse[2]='E';
  }
  if(pdata->sov_what==3) {
    sse[0]='C';
    sse[1]='C';
    sse[2]='C';
  }
  n=0;
  for(i=0;i<n_aa;i++) {
    s1=sss1[i];
    if(s1==sse[0] || s1==sse[1] || s1==sse[2]) {
      n++;
    }
  }
  out=0.0;
  s=0.0;
  length1=0;
  length2=0;
  i=0;
  while(i<n_aa) {
    beg_s1=i;
    s1=sss1[i];
    while(sss1[i]==s1 && i<n_aa) {
      i++;
    }
    end_s1=i-1;
```

```
571         length1=end_s1−beg_s1+1;
572         multiple=0;
573         k=0;
574         while(k<n_aa) {
575           beg_s2=k;
576           s2=sss2[k];
577           while(sss2[k]==s2 && k<n_aa) {
578             k++;
579           }
580           end_s2=k−1;
581           length2=end_s2−beg_s2+1;
582           if(s1==sse[0] || s1==sse[1] || s1==sse[2]) {
583             if(s1==s2 && end_s2>=beg_s1 && beg_s2<=end_s1) {
584               if(multiple>0 && pdata−>sov_method==1) {
585                 n=n+length1;
586               }
587               multiple++;
588               if(beg_s1>beg_s2) {
589                 j1=beg_s1;
590                 j2=beg_s2;
591               }
592               else {
593                 j1=beg_s2;
594                 j2=beg_s1;
595               }
596               if(end_s1<end_s2) {
597                 k1=end_s1;
598                 k2=end_s2;
599               }
600               else {
601                 k1=end_s2;
602                 k2=end_s1;
603               }
604               minov=k1−j1+1;
605               maxov=k2−j2+1;
606               d1=floor(length1∗pdata−>sov_delta_s);
607               d2=floor(length2∗pdata−>sov_delta_s);
608               if(d1>d2) d=d2;
609               if(d1<=d2 || pdata−>sov_method==0) d=d1;
610               if(d>minov) {
611                 d=minov;
612               }
613               if(d>maxov−minov) {
614                 d=maxov−minov;
615               }
616               x=pdata−>sov_delta∗d;
617               x=(minov+x)∗length1;
618               if(maxov>0) {
619                 s=s+x/maxov;
620               }
621               else {
622                 printf("\n␣ERROR!␣minov␣=␣%−4d␣maxov␣=␣%−4d␣length␣=␣%−4d␣d␣=␣%−4d␣␣␣%4d␣%4d␣␣%4d␣%4
                      ↪ d",
623                       minov,maxov,length1,d,beg_s1+1,end_s1+1,beg_s2+1,end_s2+1);
624               }
625               if(pdata−>sov_out==2) {
626                 printf("\n␣TEST:␣minov␣=␣%−4d␣maxov␣=␣%−4d␣length␣=␣%−4d␣d␣=␣%−4d␣␣␣%4d␣%4d␣␣%4d␣%4d"
```

```
                    ↪ ,
                        minov,maxov,length1,d,beg_s1+1,end_s1+1,beg_s2+1,end_s2+1);
            }
          }
        }
      }
    }
    if(pdata->sov_out==2) {
      printf("\n␣TEST:␣Number␣of␣considered␣residues␣=␣%d",n);
    }
    if(n>0) {
      out=s/n;
    }
    else {
      out=1.0;
    }
    return out;
  }

  /*-----------------------------------------------------------
  /
  / Q3 - evaluate SSp by the residues predicted correctly (Q3)
  / Input: secondary structure segments
  /
  /-----------------------------------------------------------*/
  float q3(int n_aa, char sss1[MAXRES], char sss2[MAXRES], parameters *pdata)
  {
    int i, n;
    float out;
    char s, sse[3];

    sse[0]='#';
    sse[1]='#';
    sse[2]='#';

    if(pdata->q3_what==0) {
      sse[0]='H';
      sse[1]='E';
      sse[2]='C';
    }
    if(pdata->q3_what==1) {
      sse[0]='H';
      sse[1]='H';
      sse[2]='H';
    }
    if(pdata->q3_what==2) {
      sse[0]='E';
      sse[1]='E';
      sse[2]='E';
    }
    if(pdata->q3_what==3) {
      sse[0]='C';
      sse[1]='C';
      sse[2]='C';
    }

    n=0;
```

```
683     out=0.0;
684     for(i=0;i<n_aa;i++) {
685        s=sss1[i];
686        if(s==sse[0] || s==sse[1] || s==sse[2]) {
687           n++;
688           if(sss1[i]==sss2[i]) {
689              out=out + 1.0;
690           }
691        }
692     }
693     if(n>0) {
694        out=out/n;
695     }
696     else {
697        out=1.0;
698     }
699
700     return out;
701  }
```

# Appendix J

# Calculation of Q3 accuracy

The following Python program was implemented to calculate the Q3 accuracy for each class and the overall Q3 accuracy.

```
import sys
# Execute: python calc_Q3.py <pred_file>
import string
lines = None
labels = ['H', 'E', 'C']
with open(sys.argv[1]) as file:
    lines = file.readlines()
if lines is None: exit(0)
countCor = [0, 0, 0]
countAll = [0, 0, 0]
for l in range(0, len(lines)//4):
    protein_name = lines[4*l]
    primary = lines[4*l + 1]
    secondary = lines[4*l + 2]
    prediction = lines[4*l + 3]
    for s, p in zip(secondary, prediction):
        if s == '\n': continue
        if s == p:
            countCor[labels.index(s)] += 1
        countAll[labels.index(s)] += 1
total = countAll[0] + countAll[1] + countAll[2]
correct = countCor[0] + countCor[1] + countCor[2]
headers = ['Q3_All', 'Q3_C', 'Q3_E', 'Q3_H']
q3 = [(100*correct/total),
      (100*countCor[0]/countAll[0]),
      (100*countCor[1]/countAll[1]),
      (100*countCor[2]/countAll[2])]
print("\n␣␣␣{0:11}{1:11}{2:11}{3:11}".format('␣Q3_ALL', '␣Q3_H', '␣Q3_E', '␣Q3_C'))
print('{0:11.4f}{1:11.4f}{2:11.4f}{3:11.4f}\n'.format(q3[0], q3[1], q3[2], q3[3]))
```

# Appendix K

# Data pre-processing for filtering

This python program was used to prepare the datasets for the SVM filtering method. The same datasets were used to train the decision trees and random forests. It was provided by Dionysiou [24].

```
# Execute: python prepare_SVM_files.py <test_filename> <train_filename> <WINDOW> <out_test> <out_train>
import sys
#open TEST file to read data
with open(sys.argv[1],"r") as testfile:
    lines_test = testfile.readlines()
#open TRAIN file to read dat
with open(sys.argv[2],"r") as trainfile:
    lines_train = trainfile.readlines()
linenum = 1
window = int(sys.argv[3])
leftwindow = int(window/2)
#create train file
with open(sys.argv[5], "w") as svmtrain:
    for line in lines_train:
        if linenum == 5: linenum = 1
        if linenum == 3:
            target_out = line
            # if linenum == 4:
            for i in range(leftwindow):
                zeros = leftwindow - i
                for zer in range(zeros):
                    svmtrain.write("0,")
                for rem in range(i):
                    if line[rem] == "C": svmtrain.write("0,")
                    if line[rem] == "E": svmtrain.write("1,")
                    if line[rem] == "H": svmtrain.write("2,")
                #place right aminos
                for j in range(leftwindow+1):
                    if line[i+j] == "C": svmtrain.write("0,")
                    if line[i+j] == "E": svmtrain.write("1,")
                    if line[i+j] == "H": svmtrain.write("2,")
                #place label at the end
                if target_out[i] == "C": svmtrain.write("0")
                if target_out[i] == "E": svmtrain.write("1")
                if target_out[i] == "H": svmtrain.write("2")
```

```
36                svmtrain.write("\n")
37            #place aminos with no boundary constraints
38            for amino in range(leftwindow,len(line)−leftwindow−1):
39                for curr in range(−leftwindow,leftwindow+1):
40                    if line[amino+curr] == "C": svmtrain.write("0,")
41                    if line[amino+curr] == "E": svmtrain.write("1,")
42                    if line[amino+curr] == "H": svmtrain.write("2,")
43                #place label
44                if target_out[amino] == "C": svmtrain.write("0")
45                if target_out[amino] == "E": svmtrain.write("1")
46                if target_out[amino] == "H": svmtrain.write("2")
47                svmtrain.write("\n")
48            #place last aminos with padding
49            for i in range(len(line)−leftwindow−1,len(line)−1):
50                printed=0
51                for left in range(i−leftwindow−1,i):
52                    if line[left] == "C": svmtrain.write("0,")
53                    if line[left] == "E": svmtrain.write("1,")
54                    if line[left] == "H": svmtrain.write("2,")
55                for j in range(i,len(line)−1):
56                    if line[j] == "C": svmtrain.write("0,")
57                    if line[j] == "E": svmtrain.write("1,")
58                    if line[j] == "H": svmtrain.write("2,")
59                    printed=printed+1
60                zeros = leftwindow−printed
61                for z in range(zeros):
62                    svmtrain.write("0,")
63                # place label
64                if target_out[i] == "C": svmtrain.write("0")
65                if target_out[i] == "E": svmtrain.write("1")
66                if target_out[i] == "H": svmtrain.write("2")
67                svmtrain.write("\n")
68        linenum += 1
69    svmtrain.flush()
70 linenum=1
71 #create TEST file
72 with open(sys.argv[4], "w") as svmtest:
73    for line in lines_test:
74        if linenum == 5: linenum = 1
75        if linenum == 3: target_out = line
76        if linenum == 4:
77            for i in range(leftwindow):
78                zeros = leftwindow − i
79                for zer in range(zeros):
80                    svmtest.write("0,")
81                for rem in range(i):
82                    if line[rem] == "C": svmtest.write("0,")
83                    if line[rem] == "E": svmtest.write("1,")
84                    if line[rem] == "H": svmtest.write("2,")
85                #place right aminos
86                for j in range(leftwindow+1):
87                    if line[i+j] == "C": svmtest.write("0,")
88                    if line[i+j] == "E": svmtest.write("1,")
89                    if line[i+j] == "H": svmtest.write("2,")
90                #place label at the end
91                if target_out[i] == "C": svmtest.write("0")
92                if target_out[i] == "E": svmtest.write("1")
```

```
                if target_out[i] == "H": svmtest.write("2")
                svmtest.write("\n")
            #place aminos with no boundary constraints
            for amino in range(leftwindow,len(line)−leftwindow−1):
                for curr in range(−leftwindow,leftwindow+1):
                    if line[amino+curr] == "C": svmtest.write("0,")
                    if line[amino+curr] == "E": svmtest.write("1,")
                    if line[amino+curr] == "H": svmtest.write("2,")
                #place label
                if target_out[amino] == "C": svmtest.write("0")
                if target_out[amino] == "E": svmtest.write("1")
                if target_out[amino] == "H": svmtest.write("2")
                svmtest.write("\n")
            #place last aminos with padding
            for i in range(len(line)−leftwindow−1,len(line)−1):
                printed=0
                for left in range(i−leftwindow−1,i):
                    if line[left] == "C": svmtest.write("0,")
                    if line[left] == "E": svmtest.write("1,")
                    if line[left] == "H": svmtest.write("2,")
                for j in range(i,len(line)−1):
                    if line[j] == "C": svmtest.write("0,")
                    if line[j] == "E": svmtest.write("1,")
                    if line[j] == "H": svmtest.write("2,")
                    printed+=1
                zeros = leftwindow−printed
                for z in range(zeros):
                    svmtest.write("0,")
                # place label
                if target_out[i] == "C": svmtest.write("0")
                if target_out[i] == "E": svmtest.write("1")
                if target_out[i] == "H": svmtest.write("2")
                svmtest.write("\n")
        linenum += 1
    svmtest.flush()
```

# Appendix L

# Training Filtering Methods

The following program was implemented to train the filtering models and apply the filtering techniques on the output data of the Convolutional Neural Network.

```
# Execute: python train_SVM.py <test_filename> <train_filename> <WINDOW> <pred_file> <out_prediction> <out_sov> <filter_opt>
from __future__ import print_function
import sys
import string
import numpy as np
import numpy as np
from sklearn.metrics import classification_report
from sklearn.svm import SVC
from sklearn import svm, pipeline
from sklearn import linear_model
from sklearn.metrics import confusion_matrix
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

def get_balanced_data(X_train, y_train):
    classH = []
    classE = []
    classC = []
    for i,label in enumerate(y_train):
        if label == 0:
            classH.append(i)
        elif label == 1:
            classE.append(i)
        else:
            classC.append(i)
    rows = min(len(classH), len(classE), len(classC))

    # Create a balanced data set
    X_balanced = np.concatenate((X_train[classH][0:rows], X_train[classE][0:rows], X_train[classC][0:rows]), axis=0)

    y_balanced = np.concatenate((y_train[classH][0:rows], y_train[classE][0:rows], y_train[classC][0:rows]), axis=0)

    balanced = np.zeros((X_balanced.shape[0], X_balanced.shape[1]+1), dtype=int)

    balanced[:,-1] = y_balanced
    balanced[:,:-1] = X_balanced
```

```
    np.random.shuffle(balanced)
    return balanced[:,:−1], balanced[:,−1]


def create_output_pred(pred, input_f, out_f, outSOV):
    with open(input_f, "r") as pred_file:
        pred_lines = pred_file.readlines()
    pred = pred.astype(int)
    labels = ['C', 'E', 'H']
    counter = 0
    with open(out_f, 'w') as out_file:
        for line in range(0, len(pred_lines)//4):
            protein_name = pred_lines[line∗4][0:−1]
            primary_structure = pred_lines[line∗4+1][0:−1]
            secondary_structure = pred_lines[line∗4+2][0:−1]
            prediction = ""
            for c in secondary_structure:
                prediction = prediction + labels[pred[counter]]
                counter += 1
            out_file.write(protein_name + "\n")
            out_file.write(primary_structure + "\n")
            out_file.write(secondary_structure + "\n")
            out_file.write(prediction + "\n")

    with open(out_f, "r") as out_file:
        lines = out_file.readlines()
    with open(outSOV, "w") as f1:
        for i in range(0, len(lines), 4):
            f1.write('>OSEQ\n')
            f1.write(lines[i + 2])
            f1.write('>PSEQ\n')
            f1.write(lines[i + 3])
            f1.write('>AA\n')
            f1.write(lines[i + 1])

train_dataset = np.loadtxt(sys.argv[2], delimiter=",")
win=int(sys.argv[3])
X_train = train_dataset[:, 0:win]
y_train = train_dataset[:, [win]]
test_dataset = np.loadtxt(sys.argv[1], delimiter=",")
X_test = test_dataset[:, 0:win]
y_test = test_dataset[:, [win]]
y_train = np.reshape(y_train,len(y_train))
y_test = np.reshape(y_test,len(y_test))
X_train, y_train = get_balanced_data(X_train, y_train)

print("Training␣...")

if (sys.argv[7] == '1'):
    clf = SVC(C=10, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
        decision_function_shape='ovr', degree=3, gamma=0.1, kernel='rbf',
        max_iter=−1, probability=False, random_state=None, shrinking=True,
        tol=0.001, verbose=False)
elif (sys.argv[7] == '2'):
    clf = DecisionTreeClassifier(max_depth=20)
elif (sys.argv[7] == '3'):
    clf = RandomForestClassifier(max_depth=25, random_state=42)
```

```
elif (sys.argv[7] == '0'):
  kernels = ['Polynomial', 'RBF', 'Sigmoid','Linear']
  #A function which returns the corresponding SVC model
  def getClassifier(ktype):
      if ktype == 0:
          # Polynomial kernal
          return SVC(kernel='poly', degree=8, gamma="auto")
      elif ktype == 1:
          # Radial Basis Function kernel
          return SVC(kernel='rbf', gamma="auto")
      elif ktype == 2:
          # Sigmoid kernel
          return SVC(kernel='sigmoid', gamma="auto")
      elif ktype == 3:
          # Linear kernel
          return SVC(kernel='linear', gamma="auto")

  for i in range(1, 4):
    # Train a SVC model using different kernels
      svclassifier = getClassifier(i)
      svclassifier.fit(X_train, y_train)
    # Make prediction
      y_pred = svclassifier.predict(X_test)
    # Evaluate model
      print("Evaluation:", kernels[i], "kernel")
      print(classification_report(y_test, y_pred))

  from sklearn.model_selection import GridSearchCV
  param_grid = {'C': [0.1, 1, 10], 'gamma': [1, 0.1, 0.01, 0.001],'kernel': ['rbf']}
  grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=2)
  grid.fit(X_train, y_train)
  print(grid.best_estimator_)

  y_pred = grid.predict(X_test)
  print(confusion_matrix(y_test, y_pred))
  print(classification_report(y_test, y_pred))
  exit(0)
else:
  print('Error! train_SVM.py currently has no such filtering option.')
  print('Please try again (availiable options: 0-3)')
  exit(0)

# Predict the response for test dataset
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print("THE SCORE: ", clf.score(X_test, y_test))
print("")

# creating a confusion matrix
cm = confusion_matrix(y_test, y_pred)
print('Confusion Matrix')
print(cm)
print("")

create_output_pred(y_pred, sys.argv[4], sys.argv[5], sys.argv[6])
```

# Appendix M

# All filtering methods on CB513

This bash script was implemented and used to apply the ensembles and the filtering methods in various orders and display the results for each fold of the CB513 dataset.

```bash
#!/bin/bash
# Author : Panayiotis Leontiou
# Since : April 2020
# Version: 1.0
# Bugs : No known bugs

TEST_FOLDER="./CB513_test_pred"
TRAIN_FOLDER="./CB513_train_pred"
WINDOW="15"
SVM_WIN="13"
filterOpt=( "1" "2" "3" )

echo "


PPPPPPPPPPPPPPPPP␣␣␣␣␣SSSSSSSSSSSSSSS␣␣␣␣SSSSSSSSSSSSSSS␣PPPPPPPPPPPPPPPPP
P:::::::::::::::P␣␣␣SS::::::::::::::S␣SS::::::::::::::SP::::::::::::::::P
P::::::PPPPPP:::::P␣S:::::SSSSSS::::::SS:::::SSSSSS::::::SP::::::PPPPPP:::::P
PP:::::P␣␣␣␣␣P:::::PS:::::S␣␣␣␣␣SSSSSSSS:::::S␣␣␣␣␣SSSSSSSPP:::::P␣␣␣␣␣P:::::P
␣␣P::::P␣␣␣␣␣P:::::PS:::::S␣␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣␣␣␣␣␣␣␣␣␣␣␣P::::P␣␣␣␣␣P:::::P
␣␣P::::P␣␣␣␣␣P:::::PS:::::S␣␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣␣␣␣␣␣␣␣␣␣␣␣P::::P␣␣␣␣␣P:::::P
␣␣P::::PPPPPP:::::P␣␣S::::SSSS␣␣␣␣␣␣␣␣␣␣S::::SSSS␣␣␣␣␣␣␣␣␣␣␣P::::PPPPPP:::::P
␣␣P::::::::::::::PP␣␣␣␣SS::::::SSSSS␣␣␣␣␣␣SS::::::SSSSS␣␣␣␣␣␣P::::::::::::::PP
␣␣P::::PPPPPPPPP␣␣␣␣␣␣␣␣SSS::::::::SS␣␣␣␣␣␣SSS::::::::SS␣␣␣␣P::::PPPPPPPPP
␣␣P::::P␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣SSSSSS::::S␣␣␣␣␣␣␣␣SSSSSS::::S␣␣␣P::::P
␣␣P::::P␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣P::::P
␣␣P::::P␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣P::::P
PP::::::PP␣␣␣␣␣␣␣␣␣␣SSSSSSS␣␣␣␣␣S:::::SSSSSSSS␣␣␣␣␣S:::::SPP::::::PP
P::::::::P␣␣␣␣␣␣␣␣␣␣S::::::SSSSSS:::::SS::::::SSSSSS:::::SP::::::::P
P::::::::P␣␣␣␣␣␣␣␣␣␣S::::::::::::::SS␣S::::::::::::::SS␣P::::::::P
PPPPPPPPPP␣␣␣␣␣␣␣␣␣␣SSSSSSSSSSSSSSS␣␣␣␣SSSSSSSSSSSSSSS␣␣␣PPPPPPPPPP


"
print_fold () {
  case $1 in
    fold0)
```

```
        cat << 'EOF'
                  ___ ___ _ ___ __
      o O O | __| / _ \ | | | \ / \
     o | _| | (_) | | |__ | |) | ___ | () |
   TS__[O] _|_|_ \___/ |____| |___/ |___| _\__/
  {======| _| """ ␣|␣_|"""""| _|"""""|␣_|"""""| _|"""""|␣_|"""""|
./o--000'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'
EOF
␣␣␣␣␣;;
␣␣␣␣fold1)
␣␣␣␣␣␣cat␣<<␣"EOF"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣___␣␣␣␣␣___␣␣␣␣␣␣_␣␣␣␣␣␣␣␣___␣␣␣␣␣␣␣␣␣␣␣␣␣␣_
␣␣␣␣o␣O␣O␣␣␣␣|␣__|␣␣␣/␣_␣\␣␣␣␣|␣|␣␣␣␣␣␣|␣␣␣\␣␣␣␣␣␣␣␣␣␣␣␣/␣|
␣␣␣o␣␣␣␣␣␣␣␣␣|␣|␣␣␣|␣(_)␣|␣␣␣|␣|__␣␣␣␣|␣|)␣|␣␣␣␣___␣␣␣␣␣␣|␣|
␣␣TS__[O]␣␣␣␣|_|_␣␣␣␣␣\___/␣␣␣␣|____|␣␣␣|___/␣␣␣␣|___|␣␣␣␣|_|_
␣{======|␣_|␣"""␣|␣_|"""""|␣_|"""""|␣_|"""""|␣_|"""""|
./o--000' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-'
EOF
          ;;
       fold2)
          cat << "EOF"
                  ___ ___ _ ___ ___
      o O O | __| / _ \ | | | \ |_ )
     o | _| | (_) | | |__ | |) | ___ / /
   TS__[O] _|_|_ \___/ |____| |___/ |___| /___|
  {======| _| """ ␣|␣_|"""""| _|"""""|␣_|"""""| _|"""""|␣_|"""""|
./o--000'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'
EOF
␣␣␣␣␣;;
␣␣␣␣fold3)
␣␣␣␣␣␣cat␣<<␣"EOF"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣___␣␣␣␣␣___␣␣␣␣␣␣_␣␣␣␣␣␣␣␣___␣␣␣␣␣␣␣␣␣␣␣␣␣␣____
␣␣␣␣o␣O␣O␣␣␣␣|␣__|␣␣␣/␣_␣\␣␣␣␣|␣|␣␣␣␣␣␣|␣␣␣\␣␣␣␣␣␣␣␣␣␣␣|__/
␣␣␣o␣␣␣␣␣␣␣␣␣|␣|␣␣␣|␣(_)␣|␣␣␣|␣|__␣␣␣␣|␣|)␣|␣␣␣␣___␣␣␣␣␣␣|_␣\
␣␣TS__[O]␣␣␣␣|_|_␣␣␣␣␣\___/␣␣␣␣|____|␣␣␣|___/␣␣␣␣|___|␣␣␣␣|___/
␣{======|␣_|␣"""␣|␣_|"""""|␣_|"""""|␣_|"""""|␣_|"""""|
./o--000' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-'
EOF
          ;;
       fold4)
          cat << "EOF"
                  ___ ___ _ ___ _ _
      o O O | __| / _ \ | | | \ | | |
     o | _| | (_) | | |__ | |) | ___ |_ _|
   TS__[O] _|_|_ \___/ |____| |___/ |___| _|_|_
  {======| _| """ ␣|␣_|"""""| _|"""""|␣_|"""""| _|"""""|␣_|"""""|
./o--000'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'
EOF
␣␣␣␣␣;;
␣␣␣␣fold5)
␣␣␣␣␣␣cat␣<<␣"EOF"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣___␣␣␣␣␣___␣␣␣␣␣␣_␣␣␣␣␣␣␣␣___␣␣␣␣␣␣␣␣␣␣␣␣␣␣___
␣␣␣␣o␣O␣O␣␣␣␣|␣__|␣␣␣/␣_␣\␣␣␣␣|␣|␣␣␣␣␣␣|␣␣␣\␣␣␣␣␣␣␣␣␣␣␣␣|␣__|
␣␣␣o␣␣␣␣␣␣␣␣␣|␣|␣␣␣|␣(_)␣|␣␣␣|␣|__␣␣␣␣|␣|)␣|␣␣␣␣___␣␣␣␣␣|__␣\
␣␣TS__[O]␣␣␣␣|_|_␣␣␣␣␣\___/␣␣␣␣|____|␣␣␣|___/␣␣␣␣|___|␣␣␣␣|___/
␣{======|␣_|␣"""␣|␣_|"""""|␣_|"""""|␣_|"""""|␣_|"""""|
./o--000' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-'
```

```
EOF
        ;;
      fold6)
        cat << "EOF"
                  ___ ___ _ ___ __
      o O O | __| / _ \ | | | \ / /
     o | _| | (_) | | |__ | |) | ___ / _ \
   TS__[O] _|_|_ \___/ |____| |___/ |___| \___/
  {======| _| """ _|_|"""""| _|"""""|_|"""""| _|"""""|_|"""""|
./o--000'_"`-0-0-' "`-0-0-'_"`-0-0-'_"`-0-0-' "`-0-0-'_"`-0-0-'
EOF
      ;;
    fold7)
      cat << "EOF"
                  ___ ___ _ ___ __
      o O O | __| / _ \ | | | \ / /
     o | _| | (_) | | |__ | |) | ___ / _ \
   TS__[O] _|_|_ \___/ |____| |___/ |___| \___/
  {======| _| """ _|_|"""""| _|"""""|_|"""""| _|"""""|_|"""""|
./o--000' "`-0-0-'_"`-0-0-'_"`-0-0-' "`-0-0-'_"`-0-0-'_"`-0-0-'
EOF
        ;;
      fold8)
        cat << "EOF"
                  ___ ___ _ ___ ___
      o O O | __| / _ \ | | | \ ( _ )
     o | _| | (_) | | |__ | |) | ___ / _ \
   TS__[O] _|_|_ \___/ |____| |___/ |___| \___/
  {======| _| """ _|_|"""""| _|"""""|_|"""""| _|"""""|_|"""""|
./o--000'_"`-0-0-' "`-0-0-'_"`-0-0-'_"`-0-0-' "`-0-0-'_"`-0-0-'
EOF
      ;;
    fold9)
      cat << "EOF"
                  ___ ___ _ ___ ___
      o O O | __| / _ \ | | | \ ( _ )  /_ _\
     o | _| | (_) | | |__ | |) | ___ \_,_/
   TS__[O] _|_|_ \___/ |____| |___/ |___| \___/
  {======| _| """ _|_|"""""| _|"""""|_|"""""| _|"""""|_|"""""|
./o--000' "`-0-0-'_"`-0-0-'_"`-0-0-' "`-0-0-'_"`-0-0-'_"`-0-0-'
EOF
      ;;
    *)
      ;;
  esac

}

print_SOV_score(){
  cat ./resultSOV.txt | grep -e 'SOV' | awk -F' ' '{sovAll += $3; sovH += $4; sovE += $5; sovC += $6} END {
      printf "\n    SOV_ALL    SOV_H      SOV_E      SOV_C\n    %.4f    %.4f    %.4f    %.4
      f\n", sovAll/NR, sovH/NR, sovE/NR, sovC/NR}'
}

get_filter_name(){
  case $1 in
    "1")
```

```
        filter_name="SVM"
        ;;
      "2")
        filter_name="Decision␣Tree"
        ;;
      "3")
        filter_name="Random␣Forest"
        ;;
      *)
        filter_name="Unknown␣Filter"
        ;;
    esac
}

get_filter_abr(){
    case $1 in
      "1")
        filter_abr="svm"
        ;;
      "2")
        filter_abr="dtree"
        ;;
      "3")
        filter_abr="rforest"
        ;;
      *)
        filter_abr="unknown"
        ;;
    esac
}

SCRIPTS="./q3_sov_scripts"
TEMP_FOLDER="./temp_runAll_CB513"
RUN_ALL_FOLDER="./CB513_runAll_out_files"
CROSS_VAL_FOLDER="./CB513_cross_validation"
[ -d "$TEMP_FOLDER" ] || mkdir "$TEMP_FOLDER"
[ -d "$RUN_ALL_FOLDER" ] || mkdir "$RUN_ALL_FOLDER"

echo "=================================================================="
echo "␣>Cross␣Validation␣Results"
echo "-----------------------------------------------------------------------------------------------------------------------------------------"
for i in `ls "$CROSS_VAL_FOLDER"`
do
    echo "$i"
    new_folder="$RUN_ALL_FOLDER/cross_val_res"
    [ -d "$new_folder" ] || mkdir "$new_folder"
    out_file=("$TEMP_FOLDER/$i""_cross_val.txt")
    for j in `ls "$CROSS_VAL_FOLDER/$i"`
    do
        echo "$CROSS_VAL_FOLDER/$i/$j"
    done > "$out_file"
    python "$SCRIPTS/ensembles.py" "$out_file" "$WINDOW" 1 "$new_folder/ens_pred.txt" "$new_folder/ens_sov.txt" "$new_folder/ens_weka.txt"
    "$SCRIPTS/runSOV" "$new_folder/ens_sov.txt"
    print_SOV_score
```

```
  python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_pred.txt"
  echo "
      ↪ ----------------------------------------------------------------------------------------------------
      ↪ "
done
echo "==================================================================="
echo ""
for i in `ls "$TEST_FOLDER"`
do
  print_fold $i
  new_folder="$RUN_ALL_FOLDER/$i""_results"
  [ -d "$new_folder" ] || mkdir "$new_folder"
  out_file=("$TEMP_FOLDER/$i""_files.txt")

  for j in `ls "$TEST_FOLDER/$i"`
  do
    echo "$TEST_FOLDER/$i/$j"
  done > "$out_file"
  echo "==================================================================="
  echo "␣>Ensembles␣Results"
  echo "
      ↪ ----------------------------------------------------------------------------------------------------
      ↪ "
  python "$SCRIPTS/ensembles.py" "$out_file" "$WINDOW" 1 "$new_folder/ensembles_pred.txt" "$new_folder/ensembles_sov.
      ↪ txt" "$new_folder/ensembles_weka.txt" > "$new_folder/ensembles_out.txt"
  "$SCRIPTS/runSOV" "$new_folder/ensembles_sov.txt"
  print_SOV_score
  python "$SCRIPTS/calc_Q3.py" "$new_folder/ensembles_pred.txt"
  echo "==================================================================="
  echo "␣>Ensembles␣+␣External␣Rules␣Results"
  echo "
      ↪ ----------------------------------------------------------------------------------------------------
      ↪ "
  python "$SCRIPTS/externalRules.py" "$new_folder/ensembles_pred.txt" "$new_folder/ens_rules_sov.txt" "$new_folder/
      ↪ ens_rules_pred.txt"
  "$SCRIPTS/runSOV" "$new_folder/ens_rules_sov.txt"
  print_SOV_score
  python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_rules_pred.txt"

  for filter in "${filterOpt[@]}"
  do
    get_filter_name $filter
    get_filter_abr $filter
    echo "==================================================================="
    echo "␣>Ensembles␣+␣External␣Rules␣+␣$filter_name␣Results"
    echo "
        ↪ ----------------------------------------------------------------------------------------------------
        ↪ "
    python "$SCRIPTS/prepare_SVM_files.py" "$new_folder/ens_rules_pred.txt" "$TRAIN_FOLDER/$i""_train_pred.txt" "
        ↪ $SVM_WIN" "$new_folder/temp_svm_test.txt" "$new_folder/temp_svm_train.txt"
    python "$SCRIPTS/train_SVM.py" "$new_folder/temp_svm_test.txt" "$new_folder/temp_svm_train.txt" "$SVM_WIN" "
        ↪ $new_folder/ens_rules_pred.txt" "$new_folder/ens_rules_$filter_abr""_pred.txt" "$new_folder/ens_rules_$filter_abr
        ↪ ""_sov.txt" "$filter" > "$new_folder/ens_rules_$filter_abr""_out.txt"
    "$SCRIPTS/runSOV" "$new_folder/ens_rules_$filter_abr""_sov.txt"
    print_SOV_score
    python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_rules_$filter_abr""_pred.txt"
    echo "==================================================================="
```

```
    echo "␣>Ensembles␣+␣$filter_name␣Results"
    echo "----------------------------------------------------------------------------------------------------"
    python "$SCRIPTS/prepare_SVM_files.py" "$new_folder/ensembles_pred.txt" "$TRAIN_FOLDER/$i""_train_pred.txt" "$SVM_WIN" "$new_folder/temp_svm_test.txt" "$new_folder/temp_svm_train.txt"
    python "$SCRIPTS/train_SVM.py" "$new_folder/temp_svm_test.txt" "$new_folder/temp_svm_train.txt" "$SVM_WIN" "$new_folder/ensembles_pred.txt" "$new_folder/ens_$filter_abr""_pred.txt" "$new_folder/ens_$filter_abr""_sov.txt" "$filter" > "$new_folder/ens_$filter_abr""_out.txt"
    "$SCRIPTS/runSOV" "$new_folder/ens_$filter_abr""_sov.txt"
    print_SOV_score
    python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_$filter_abr""_pred.txt"
    echo "=================================================================="
    echo "␣>Ensembles␣+␣$filter_name␣+␣External␣Rules␣Results"
    echo "----------------------------------------------------------------------------------------------------"
    python "$SCRIPTS/externalRules.py" "$new_folder/ens_$filter_abr""_pred.txt" "$new_folder/ens_$filter_abr""_rules_sov.txt" "$new_folder/ens_$filter_abr""_rules_pred.txt"
    "$SCRIPTS/runSOV" "$new_folder/ens_$filter_abr""_rules_sov.txt"
    print_SOV_score
    python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_$filter_abr""_rules_pred.txt"
  done
  echo "=================================================================="
  echo ""
  # exit 0
done

# Remove temp files
rm -rf "$TEMP_FOLDER"
rm resultSOV.txt
rm SOVinput.txt
```

# Appendix N

# View filtering results of CB513

The following bash script was implemented and used to view all the ensembles and filtering results in a table format, for the CB513 dataset.

```bash
#!/bin/bash

file="./final_results_CB513.txt"

echo "Ensembles Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + External Rules Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + External Rules Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + External Rules + SVM Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + External Rules + SVM Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + SVM Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + SVM Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

```

```
echo "Ensembles + SVM + External Rules Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + SVM + External Rules Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + External Rules + Decision Tree Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + External Rules + Decision Tree Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + Decision Tree Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + Decision Tree Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + Decision Tree + External Rules Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + Decision Tree + External Rules Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + External Rules + Random Forest Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + External Rules + Random Forest Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + Random Forest Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + Random Forest Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + Random Forest + External Rules Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + Random Forest + External Rules Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=
```

```
↪ $4;␣switch=2;}␣else␣{printf␣"%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n",␣$1,␣$2,␣$3,␣$4,␣v1,␣v2,␣v3,␣v4;
↪ ␣switch=1}}'
```

# Appendix O

# All filtering methods on PISCES

This bash script was implemented and used to apply the ensembles and the filtering methods in various orders and display the results for each fold of the PISCES dataset.

```
#!/bin/bash
# Author : Panayiotis Leontiou
# Since : May 2020
# Version: 1.0
# Bugs : No known bugs

TEST_FOLDER="./PISCES_test_pred"
TRAIN_FOLDER="./PISCES_train_pred"
WINDOW="15"
SVM_WIN="19"
filterOpt=( "2" "3" )

echo "



PPPPPPPPPPPPPPPPP␣␣␣␣␣SSSSSSSSSSSSSSS␣␣␣␣SSSSSSSSSSSSSSS␣PPPPPPPPPPPPPPPPP
P:::::::::::::::P␣␣␣SS::::::::::::::S␣SS::::::::::::::SP::::::::::::::::P
P::::::PPPPPP:::::P␣S:::::SSSSSS::::::SS:::::SSSSSS::::::SP::::::PPPPPP:::::P
PP:::::P␣␣␣␣␣P:::::PS:::::S␣␣␣␣␣SSSSSSSS:::::S␣␣␣␣␣SSSSSSSPP:::::P␣␣␣␣␣P:::::P
␣␣P::::P␣␣␣␣␣P:::::PS:::::S␣␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣␣␣␣␣␣␣␣␣␣␣␣P::::P␣␣␣␣␣P::::P
␣␣P::::P␣␣␣␣␣P:::::PS:::::S␣␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣␣␣␣␣␣␣␣␣␣␣␣P::::P␣␣␣␣␣P::::P
␣␣P::::PPPPPP:::::P␣␣S::::SSSS␣␣␣␣␣␣␣␣␣␣S::::SSSS␣␣␣␣␣␣␣␣␣␣␣P::::PPPPPP:::::P
␣␣P::::::::::::PP␣␣␣␣SS::::::SSSSSS␣␣␣␣␣␣SS::::::SSSSSS␣␣␣␣␣␣P::::::::::::PP
␣␣P::::PPPPPPPPP␣␣␣␣␣␣␣␣SSS::::::::SS␣␣␣␣␣␣SSS::::::::SS␣␣␣␣P::::PPPPPPPPP
␣␣P::::P␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣SSSSSS::::S␣␣␣␣␣␣␣␣SSSSSS::::S␣␣␣P::::P
␣␣P::::P␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣P::::P
␣␣P::::P␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣␣␣␣␣␣␣␣␣␣S:::::S␣␣P::::P
PP::::::PP␣␣␣␣␣␣␣␣␣␣SSSSSSS␣␣␣␣␣S:::::SSSSSSSS␣␣␣␣␣S:::::SPP::::::PP
P::::::::P␣␣␣␣␣␣␣␣␣␣S:::::SSSSSS:::::SS::::::SSSSSS:::::SP::::::::P
P::::::::P␣␣␣␣␣␣␣␣␣␣S::::::::::::::SS␣S::::::::::::::SS␣P::::::::P
PPPPPPPPPP␣␣␣␣␣␣␣␣␣␣SSSSSSSSSSSSSSS␣␣␣␣SSSSSSSSSSSSSSS␣␣␣PPPPPPPPPP



"
print_fold () {
```

```
  case $1 in
    fold0)
      cat << 'EOF'
                 ___ ___ _ ___ __
    o O O | __| / _ \ | | | \ / \
   o | _| | (_) | | |__ | |) | ___ | () |
  TS__[O] _|_|_ \___/ |____| |___/ |___| _\__/
 {======| _| """ ␣|␣_|"""""| _|"""""|␣_|"""""| _|"""""|␣_|"""""|
./o--000'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'
EOF
␣␣␣␣␣␣;;
␣␣␣␣fold1)
␣␣␣␣␣␣cat␣<<␣"EOF"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣___␣␣␣␣␣___␣␣␣␣␣␣_␣␣␣␣␣␣␣␣___␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣_
␣␣␣␣o␣O␣O␣␣␣␣|␣__|␣␣␣/␣_␣\␣␣␣␣|␣|␣␣␣␣␣␣|␣␣␣\␣␣␣␣␣␣␣␣␣␣␣␣␣/␣|
␣␣␣o␣␣␣␣␣␣␣␣|␣|␣␣␣|␣(_)␣|␣␣␣|␣|__␣␣␣␣|␣|)␣|␣␣␣␣___␣␣␣␣␣␣|␣|
␣␣TS__[O]␣␣␣␣|_|_␣␣␣␣\___/␣␣␣␣|____|␣␣␣|___/␣␣␣␣|___|␣␣␣␣|_|_
␣{======|␣__|␣"""␣|␣_|"""""|␣_|"""""|␣_|"""""|␣_|"""""|
./o--000' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-'
EOF
      ;;
    fold2)
      cat << "EOF"
                 ___ ___ _ ___ ___
    o O O | __| / _ \ | | | \ |_ )
   o | _| | (_) | | |__ | |) | ___ / /
  TS__[O] _|_|_ \___/ |____| |___/ |___| /___|
 {======| _| """ ␣|␣_|"""""| _|"""""|␣_|"""""| _|"""""|␣_|"""""|
./o--000'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'
EOF
␣␣␣␣␣␣;;
␣␣␣␣fold3)
␣␣␣␣␣␣cat␣<<␣"EOF"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣___␣␣␣␣␣___␣␣␣␣␣␣_␣␣␣␣␣␣␣␣___␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣____
␣␣␣␣o␣O␣O␣␣␣␣|␣__|␣␣␣/␣_␣\␣␣␣␣|␣|␣␣␣␣␣␣|␣␣␣\␣␣␣␣␣␣␣␣␣␣␣␣␣|__␣/
␣␣␣o␣␣␣␣␣␣␣␣|␣|␣␣␣|␣(_)␣|␣␣␣|␣|__␣␣␣␣|␣|)␣|␣␣␣␣___␣␣␣␣␣␣|_␣\
␣␣TS__[O]␣␣␣␣|_|_␣␣␣␣\___/␣␣␣␣|____|␣␣␣|___/␣␣␣␣|___|␣␣␣␣|___/
␣{======|␣__|␣"""␣|␣_|"""""|␣_|"""""|␣_|"""""|␣_|"""""|
./o--000' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-'
EOF
      ;;
    fold4)
      cat << "EOF"
                 ___ ___ _ ___ _ _
    o O O | __| / _ \ | | | \ | | |
   o | _| | (_) | | |__ | |) | ___ |_ _|
  TS__[O] _|_|_ \___/ |____| |___/ |___| _|_|_
 {======| _| """ ␣|␣_|"""""| _|"""""|␣_|"""""| _|"""""|␣_|"""""|
./o--000'␣"`-0-0-' "`-0-0-'␣"`-0-0-'␣"`-0-0-' "`-0-0-'␣"`-0-0-'
EOF
␣␣␣␣␣␣;;
␣␣␣␣fold5)
␣␣␣␣␣␣cat␣<<␣"EOF"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣___␣␣␣␣␣___␣␣␣␣␣␣_␣␣␣␣␣␣␣␣___␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣___
␣␣␣␣o␣O␣O␣␣␣␣|␣__|␣␣␣/␣_␣\␣␣␣␣|␣|␣␣␣␣␣␣|␣␣␣\␣␣␣␣␣␣␣␣␣␣␣␣␣|␣__|
␣␣␣o␣␣␣␣␣␣␣␣|␣|␣␣␣|␣(_)␣|␣␣␣|␣|__␣␣␣␣|␣|)␣|␣␣␣␣___␣␣␣␣␣␣|__␣\
␣␣TS__[O]␣␣␣␣|_|_␣␣␣␣\___/␣␣␣␣|____|␣␣␣|___/␣␣␣␣|___|␣␣␣␣|___/
```

```
␣{======|␣_|␣"""␣|␣_|"""""""|␣_|"""""""|␣_|"""""""|␣_|"""""""|␣_|"""""""|
./o−−000’ "‘−0−0−’␣"‘−0−0−’␣"‘−0−0−’ "‘−0−0−’␣"‘−0−0−’␣"‘−0−0−’
EOF
        ;;
    fold6)
        cat << "EOF"
                ___ ___ _ ___ __
    o O O | __| / _ \ | | | \ / /
   o | _| | (_) | | |__ | |) | ___ / _ \
  TS__[O] _|_|_ \___/ |____| |___/ |___| \___/
 {======| _| """␣|␣_|"""""""| _|"""""""|␣_|"""""""| _|"""""""|␣_|"""""""|
./o−−000’␣"‘−0−0−’ "‘−0−0−’␣"‘−0−0−’␣"‘−0−0−’ "‘−0−0−’␣"‘−0−0−’
EOF
␣␣␣␣␣␣;;
␣␣␣␣fold7)
␣␣␣␣␣␣cat␣<<␣"EOF"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣___␣␣␣␣␣___␣␣␣␣␣␣_␣␣␣␣␣␣␣␣___␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣____
␣␣␣␣o␣O␣O␣␣␣␣|␣__|␣␣␣/␣_␣\␣␣␣␣|␣|␣␣␣␣␣␣|␣␣␣\␣␣␣␣␣␣␣␣␣␣␣␣␣|__␣␣|
␣␣␣o␣␣␣␣␣␣␣␣␣|␣_|␣␣␣|␣(_)␣|␣␣␣|␣|__␣␣␣␣|␣|)␣|␣␣␣␣___␣␣␣␣␣␣␣/␣/
␣␣TS__[O]␣␣␣_|_|_␣␣␣␣␣\___/␣␣␣␣|____|␣␣␣|___/␣␣␣␣|___|␣␣␣␣_/_/_
␣{======|␣_|␣"""␣|␣_|"""""""|␣_|"""""""|␣_|"""""""|␣_|"""""""|␣_|"""""""|
./o−−000’ "‘−0−0−’␣"‘−0−0−’␣"‘−0−0−’ "‘−0−0−’␣"‘−0−0−’␣"‘−0−0−’
EOF
        ;;
    fold8)
        cat << "EOF"
                ___ ___ _ ___ ___
    o O O | __| / _ \ | | | \ ( _ )
   o | _| | (_) | | |__ | |) | ___ / _ \
  TS__[O] _|_|_ \___/ |____| |___/ |___| \___/
 {======| _| """␣|␣_|"""""""| _|"""""""|␣_|"""""""| _|"""""""|␣_|"""""""|
./o−−000’␣"‘−0−0−’ "‘−0−0−’␣"‘−0−0−’␣"‘−0−0−’ "‘−0−0−’␣"‘−0−0−’
EOF
␣␣␣␣␣␣;;
␣␣␣␣fold9)
␣␣␣␣␣␣cat␣<<␣"EOF"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣___␣␣␣␣␣___␣␣␣␣␣␣_␣␣␣␣␣␣␣␣___␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣___
␣␣␣␣o␣O␣O␣␣␣␣|␣__|␣␣␣/␣_␣\␣␣␣␣|␣|␣␣␣␣␣␣|␣␣␣\␣␣␣␣␣␣␣␣␣␣␣␣␣/␣_␣\
␣␣␣o␣␣␣␣␣␣␣␣␣|␣_|␣␣␣|␣(_)␣|␣␣␣|␣|__␣␣␣␣|␣|)␣|␣␣␣␣___␣␣␣␣␣\_,␣/
␣␣TS__[O]␣␣␣_|_|_␣␣␣␣␣\___/␣␣␣␣|____|␣␣␣|___/␣␣␣␣|___|␣␣␣␣_/_/_
␣{======|␣_|␣"""␣|␣_|"""""""|␣_|"""""""|␣_|"""""""|␣_|"""""""|␣_|"""""""|
./o−−000’ "‘−0−0−’␣"‘−0−0−’␣"‘−0−0−’ "‘−0−0−’␣"‘−0−0−’␣"‘−0−0−’
EOF
        ;;
    ∗)
        ;;
  esac

}

print_SOV_score(){
  cat ./resultSOV.txt | grep −e ’SOV’ | awk −F’␣’ ’{sovAll␣+=␣$3;␣sovH␣+=␣$4;␣sovE␣+=␣$5;␣sovC␣+=␣$6}␣END␣{
        ↪ printf␣"\n␣␣␣␣SOV_ALL␣␣␣␣SOV_H␣␣␣␣␣␣SOV_E␣␣␣␣␣␣SOV_C\n␣␣␣␣%.4f␣␣␣␣%.4f␣␣␣␣%.4f␣␣␣␣%.4
        ↪ f\n",␣sovAll/NR,␣sovH/NR,␣sovE/NR,␣sovC/NR}’
}

get_filter_name(){
```

```
  case $1 in
    "1")
      filter_name="SVM"
      ;;
    "2")
      filter_name="Decision␣Tree"
      ;;
    "3")
      filter_name="Random␣Forest"
      ;;
    *)
      filter_name="Unknown␣Filter"
      ;;
  esac
}

get_filter_abr(){
  case $1 in
    "1")
      filter_abr="svm"
      ;;
    "2")
      filter_abr="dtree"
      ;;
    "3")
      filter_abr="rforest"
      ;;
    *)
      filter_abr="unknown"
      ;;
  esac
}

SCRIPTS="./q3_sov_scripts"
TEMP_FOLDER="./temp_runAll_PISCES"
RUN_ALL_FOLDER="./PISCES_runAll_out_files"
CROSS_VAL_FOLDER="./PISCES_cross_validation"
[ -d "$TEMP_FOLDER" ] || mkdir "$TEMP_FOLDER"
[ -d "$RUN_ALL_FOLDER" ] || mkdir "$RUN_ALL_FOLDER"

echo "==================================================================="
echo "␣>Cross␣Validation␣Results"
echo "
    ↪ ---------------------------------------------------------------------------
    ↪ "
for i in `ls "$CROSS_VAL_FOLDER"`
do
  echo "$i"
  new_folder="$RUN_ALL_FOLDER/cross_val_res"
  [ -d "$new_folder" ] || mkdir "$new_folder"
  out_file=("$TEMP_FOLDER/$i""_cross_val.txt")
  for j in `ls "$CROSS_VAL_FOLDER/$i"`
  do
    echo "$CROSS_VAL_FOLDER/$i/$j"
  done > "$out_file"
  python "$SCRIPTS/ensembles.py" "$out_file" "$WINDOW" 1 "$new_folder/ens_pred.txt" "$new_folder/ens_sov.txt" "
      ↪ $new_folder/ens_weka.txt"
```

```
204     "$SCRIPTS/runSOV" "$new_folder/ens_sov.txt"
205     print_SOV_score
206     python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_pred.txt"
207     echo "
            ↪ ------------------------------------------------------------------------------
            ↪ "
208   done
209   echo "==================================================================="
210   echo ""
211   for i in `ls "$TEST_FOLDER"`
212   do
213     print_fold $i
214     new_folder="$RUN_ALL_FOLDER/$i""_results"
215     [ -d "$new_folder" ] || mkdir "$new_folder"
216     out_file=("$TEMP_FOLDER/$i""_files.txt")
217
218     for j in `ls "$TEST_FOLDER/$i"`
219     do
220       echo "$TEST_FOLDER/$i/$j"
221     done > "$out_file"
222     echo "==================================================================="
223     echo "␣>Ensembles␣Results"
224     echo "
            ↪ ------------------------------------------------------------------------------
            ↪ "
225     python "$SCRIPTS/ensembles.py" "$out_file" "$WINDOW" 1 "$new_folder/ensembles_pred.txt" "$new_folder/ensembles_sov.
            ↪ txt" "$new_folder/ensembles_weka.txt" > "$new_folder/ensembles_out.txt"
226     "$SCRIPTS/runSOV" "$new_folder/ensembles_sov.txt"
227     print_SOV_score
228     python "$SCRIPTS/calc_Q3.py" "$new_folder/ensembles_pred.txt"
229     echo "==================================================================="
230     echo "␣>Ensembles␣+␣External␣Rules␣Results"
231     echo "
            ↪ ------------------------------------------------------------------------------
            ↪ "
232     python "$SCRIPTS/externalRules.py" "$new_folder/ensembles_pred.txt" "$new_folder/ens_rules_sov.txt" "$new_folder/
            ↪ ens_rules_pred.txt"
233     "$SCRIPTS/runSOV" "$new_folder/ens_rules_sov.txt"
234     print_SOV_score
235     python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_rules_pred.txt"
236     for filter in "${filterOpt[@]}"
237     do
238       get_filter_name $filter
239       get_filter_abr $filter
240       # echo "$filter $filter_name"
241       echo "==================================================================="
242       echo "␣>Ensembles␣+␣External␣Rules␣+␣$filter_name␣Results"
243       echo "
              ↪ ------------------------------------------------------------------------------
              ↪ "
244       python "$SCRIPTS/prepare_SVM_files.py" "$new_folder/ens_rules_pred.txt" "$TRAIN_FOLDER/$i""_train_pred.txt" "
              ↪ $SVM_WIN" "$new_folder/temp_svm_test.txt" "$new_folder/temp_svm_train.txt"
245       python "$SCRIPTS/train_SVM.py" "$new_folder/temp_svm_test.txt" "$new_folder/temp_svm_train.txt" "$SVM_WIN" "
              ↪ $new_folder/ens_rules_pred.txt" "$new_folder/ens_rules_$filter_abr""_pred.txt" "$new_folder/ens_rules_$filter_abr
              ↪ ""_sov.txt" "$filter" > "$new_folder/ens_rules_$filter_abr""_out.txt"
246       "$SCRIPTS/runSOV" "$new_folder/ens_rules_$filter_abr""_sov.txt"
247       print_SOV_score
```

```
    python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_rules_$filter_abr""_pred.txt"
    echo "=================================================================="
    echo "␣>Ensembles␣+␣$filter_name␣Results"
    echo "
        ↪ ------------------------------------------------------------------------------
        ↪ "
    python "$SCRIPTS/prepare_SVM_files.py" "$new_folder/ensembles_pred.txt" "$TRAIN_FOLDER/$i""_train_pred.txt" "
        ↪ $SVM_WIN" "$new_folder/temp_svm_test.txt" "$new_folder/temp_svm_train.txt"
    python "$SCRIPTS/train_SVM.py" "$new_folder/temp_svm_test.txt" "$new_folder/temp_svm_train.txt" "$SVM_WIN" "
        ↪ $new_folder/ensembles_pred.txt" "$new_folder/ens_$filter_abr""_pred.txt" "$new_folder/ens_$filter_abr""_sov.txt"
        ↪  "$filter" > "$new_folder/ens_$filter_abr""_out.txt"
    "$SCRIPTS/runSOV" "$new_folder/ens_$filter_abr""_sov.txt"
    print_SOV_score
    python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_$filter_abr""_pred.txt"
    echo "=================================================================="
    echo "␣>Ensembles␣+␣$filter_name␣+␣External␣Rules␣Results"
    echo "
        ↪ ------------------------------------------------------------------------------
        ↪ "
    python "$SCRIPTS/externalRules.py" "$new_folder/ens_$filter_abr""_pred.txt" "$new_folder/ens_$filter_abr""_rules_sov.txt
        ↪ " "$new_folder/ens_$filter_abr""_rules_pred.txt"
    "$SCRIPTS/runSOV" "$new_folder/ens_$filter_abr""_rules_sov.txt"
    print_SOV_score
    python "$SCRIPTS/calc_Q3.py" "$new_folder/ens_$filter_abr""_rules_pred.txt"
  done
  echo "=================================================================="
  echo ""
  # exit 0
done

# Remove temp files
rm -rf "$TEMP_FOLDER"
rm resultSOV.txt
rm SOVinput.txt
```

# Appendix P

# View filtering results of PISCES

The following bash script was implemented and used to view all the ensembles and filtering results in a table format, for the PISCES dataset.

```bash
#!/bin/bash

file="./final_results_PISCES.txt"

echo "Ensembles Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + External Rules Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + External Rules Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + External Rules + Decision Tree Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + External Rules + Decision Tree Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + Decision Tree Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + Decision Tree Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

```

```
echo "Ensembles + Decision Tree + External Rules Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + Decision Tree + External Rules Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + External Rules + Random Forest Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + External Rules + Random Forest Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + Random Forest Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + Random Forest Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
echo ""

echo "Ensembles + Random Forest + External Rules Results"
echo -e "Q3_ALL\tQ3_H\tQ3_E\tQ3_C\tSOV_ALL\tSOV_H\tSOV_E\tSOV_C"
echo "-----------------------------------------------------------------------"
sed -n '/Ensembles + Random Forest + External Rules Results/,/=====/p' "$file" | grep -E '[0-9]+'| grep -v '[a-zA-Z]' | tr -s " " | sed -e 's/^[ \t]*//' | awk -F' ' 'BEGIN{switch=1}{if (switch == 1) {v1=$1; v2=$2; v3=$3; v4=$4; switch=2;} else {printf "%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", $1, $2, $3, $4, v1, v2, v3, v4; switch=1}}'
```