Thesis Dissertation

CONTINUAL LEARNING ON EDGE

USING TENSORFLOW-LITE

Nikolas Stavrou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2023

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Continual Learning on Edge using TensorFlow-Lite

Nikolas Stavrou

Supervisors

Dr. Chris Christodoulou

Dr. Vassilis Vassiliades

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

May 2023

Acknowledgements

I would like to thank my supervisors, professor Dr. Chris Christodoulou and Dr. Vassilis Vassiliades for their continuous support and help.

Finally, I would like to thank my family for all the mental and emotional support throughout my studies.

Abstract

This thesis addresses the challenge of implementing a continual learning model on embedded devices, notably on an Android phone, to address real-world applications. Our focus is on the utilization of a pre-trained model with some fine-tuning to facilitate class incremental learning on common objects through real-time on-device training and inference. We strive to minimize catastrophic forgetting while optimizing performance and limiting computational resource utilization.

The objectives of this thesis are twofold. First, to mitigate the added continual learning capabilities from the old TensorFlow-Lite application to the new one by incorporating a replay buffer. Second, to further experiment offline with the model in order to improve its performance while remaining lightweight and feasible for deployment on resource-limited devices.

The implementation of continual learning in the new TensorFlow-Lite demo app provides several advantages. These include increased customization, simplified model management, and an effortless way to save and restore the weights of the model. Specifically, this approach facilitates more flexible model structure and optimizer selection, streamlines model management by utilizing a single .tflite model for training and inference, and enables more efficient storage and updating of training weights with the use of tf.functions.

In our experimental phase, we did several modifications to our model which consist of a frozen MobileNetV2 as a base and a trainable head learning at full pace. These changes include the integration of replay buffer samples with new samples in the training process, application of a novel replay buffer replacement algorithm, and employment of the latent replay strategy. With these changes, we achieved significant improvements in accuracy on the CORe50 benchmark. We reached a peak accuracy of 69.7% with a replay buffer size (RBS) of 7500 samples and 69.5% with an RBS of 3000 samples which represents a significant leap over the previous model used in [2] that achieved a peak accuracy of 56.6% with an RBS of 7500 and around 62% with an RBS of 30000. Our model, though slightly less accurate than the 72.24% achieved by the AR1* model from [15], maintains a simple and lightweight model architecture that is optimal for edge deployment.

As a result, this study offers valuable insights for the deployment of continual learning models in resource-constrained environments and represents a promising advancements in on-device continual learning capabilities.

Contents

Chapter 1

1 – Introduction

1.1 – Continual Learning Problem

1.2 – Why Continual Learning

1.3 – Importance of Continual Learning on Edge

1.4 – Previous Research on Continual Learning on Edge

1.1 – Continual Learning Problem

Continual Learning is a field in machine learning that deals with the challenge of maintaining the performance of machine learning models as they encounter new data incrementally over time, without suffering from catastrophic forgetting. It essentially refers to the ability of such models to continuously learn and adapt to new information, while retaining previously learned knowledge [14].

Continual Learning has the goal of developing incremental learning models, models that can continually acquire, fine-tune and transfer knowledge and skills in dynamic, real-world environments, just as humans and animal can do. This is a crucial ability for computational systems and autonomous agents that need to process continuous streams of information and learn from experiences in real-world applications [14].

However, applying continual learning to models is a major challenge, making continual learning a complex and active area of research in machine learning. The biggest challenge it tries to tackle is preventing catastrophic forgetting, which occurs when a model overwrites previously learned knowledge with new information, resulting in a drastic decline in performance of old tasks. Furthermore, in real-world applications, continual learning requires models to learn from non-stationary data distributions, in contrast of traditional machine learning models, which are trained on stationary data distributions.

In addition, adding continual learning in the context of embedded devices makes the challenge even harder. Such computational systems have limited resources, such as memory, computation power or energy. Such scenarios require very efficient use of these limited resources [2].

1.2 – Why Continual Learning

One could argue as to why we should use Continual Learning in our models instead of other approaches for addressing the challenges of machine learning such as Federated Learning.

Federated Learning is a machine learning setting in which many individual devices organizations work together to train a shared model. This is done through a central server that coordinates the process without the need for sharing any of the data itself.

The approach has gained significant attention recently because it could lead to reducing privacy risks and costs associated with centralized machine learning [6].

However, it all depends on the environment and the type of data that we are dealing with. When it comes to applications where the data distribution is nonstationary, and where it is necessary to adapt to new data incrementally over time, continue learning is definitely the way forward. We are able to learn from a continuous stream of data and adapt to new tasks without having to retrain the model from scratch.

1.3 – Importance of Continual Learning on Edge

Tackling the problem of continual learning, especially for embedded devices is not an easy task. However, the benefits that we can obtain from applying incremental learning models on edge are numerous, especially since embedded devices are becoming increasingly prevalent in our everyday lives. From smart home devices to medical devices, all embedded devices can benefit from continual learning models since they process and analyse data in real-time on the edge.

What is more, embedded devices often operate in areas with limited or unreliable network connectivity, making it impossible to transfer data and retrain models in the cloud. On such occasions, it is vital to be able to continuously learn and adapt with on-device training.

Continual Learning on Edge can also be beneficial in ensuring our data privacy and preventing the transmission of sensitive information since the need of transferring data to the cloud can be reduced. In addition, with continual learning, models can adapt quickly to new information, without the need of complete retraining from scratch. This leads to faster model adaptation, improving overall performance of the system and also reducing the computational resources required, cutting the costs significantly [2].

1.4 – Previous Research on Continual Learning on Edge

There has been a significant amount of previous research on continual learning on edge in recent years. In this section we are going to summarize some of the most relevant papers and their findings.

“Continual Learning on Edge with TensorFlow Lite”

“Continual Learning on the Edge with TensorFlow Lite” attempted to expand the TensorFlow-Lite library to enable continual learning since at the time Google only had an experimental transfer learning API to their TensorFlow-Lite library which proved to be prone to catastrophic forgetting. In the paper, a continual learning model was tested on the CORe50 benchmark with the use of an Android application, showing that it tackles catastrophic forgetting.

The model used a frozen MobileNetV2 as the backbone which was pretrained on the ImageNet dataset. The head of the network is a simple dense fully connected layer with ReLU activations followed by a classification layer with softmax activations. A simple replay buffer was integrated right after MobileNetV2. The model was then tested with various replay buffer sizes and was compared with one of the current state-of-the-art continual learning model AR1* + Latent Replay (AR1*LR) with an accuracy close to that of AR1*LR.

Models	Last Accuracy
TL	16.9%
CL	56.6%
AR1*LR	59.76%

Figure 1.1 - Last accuracy comparison of the Transfer Learning model, The Continual Learning model of the paper and AR1* on the CORe50 NICv2 – 391 benchmark.
(Source: https://arxiv.org/abs/2105.01946)

With the newer TensorFlow-Lite Demo App that simplifies on-device training, as well as modifying the body and experimenting with numerous approaches for the head of the model we are confident that better results can be achieved [2].

“Continual Learning at the Edge: Real-Time Training on Smartphone Devices”

“Continual Learning at the Edge: Real-Time Training on Smartphone Devices” implemented and deployed a hybrid continual learning strategy (AR1*) on an Android application for real-time on-device personalization without forgetting. The model stands as one of the state-of-the-art continual learning models with great efficiency and effectiveness. The CORe50 dataset was used once again as a benchmark. A latent replay

buffer was used, and the model was built with the Caffe framework for both the inference and training phases.

	AR1 variant:	Pool	Conv5_4	Input
Final accuracy	Initial categories	**97.54%**	97.16%	93.62%
	New categories	65.33%	73.39%	**79.07%**
Training time for a new experience on 100 new patterns, 8 epochs	Feature extraction[3]	8.93	6.70	-
	Forward pass	0.14	125.05	436.33
	Backward pass	0.04	4.82	4.81
	Weights update	0	37.21	58.88
	Overall	0.18	167.08	500.02

Figure 1.2 – Final accuracy and on-device training time in seconds

(Source: https://arxiv.org/abs/2105.13127)

"Latent Replay for Real-Time Continual Learning"

"Latent Replay for Real-Time Continual Learning" introduced a novel technique called "Latent Replay." The authors utilize a replay buffer to preserve past data not from the input space but from the features of the images up to a certain layer of the MobileNetV1. They experiment with moving the replay buffer to different layers to find out which one achieves the highest accuracy. To ensure the consistency of the representations and the validity of the stored activations, they suggest decelerating the learning process in all layers below the one with latent replay, while permitting the upper layers to learn in full pace. They are able to achieve state-of-the-art performance on the CORe50 benchmark which shows the feasibility of nearly real-time continual learning on edge devices, like smartphones.

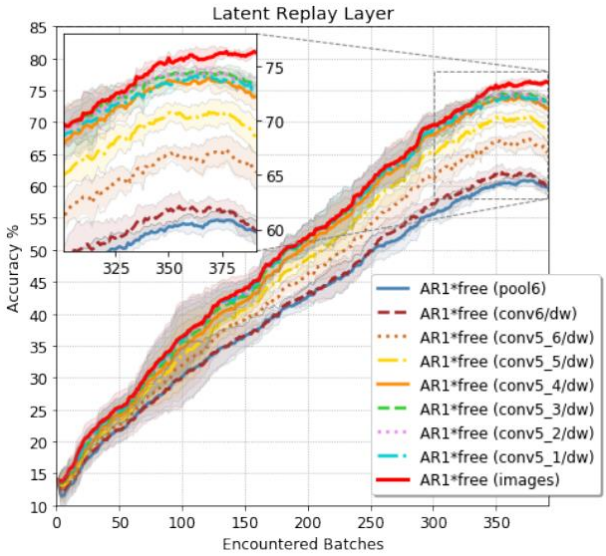


Figure 1.3 – Accuracies of choosing different layers for latent replay in the AR1*free model with a replay memory size of 1500.

(Source: https://arxiv.org/abs/1912.01100)

Chapter 2

2 - Background

2.1 – Artificial Neural Networks Background

2.1.1 – Origin of Artificial Neural Networks

2.1.2 – Variations of Artificial Neural Networks

2.1.2.1 – Multi-Layer Perceptron (MLP)

2.1.2.2 – Convolutional Neural Network (CNN)

2.1.2.2.1 – MobileNet

2.1.2.2.2 - EfficientDet

2.1.3 – Training Artificial Neural Networks

2.1.3.1 - Loss function

2.1.3.2 - Gradient Descent

2.1.3.3 – Backpropagation

2.1.3.4 – Regularization Techniques

2.1.3.5 – Overfitting/Underfitting

2.2 – Continual Learning Scenarios

2.3.1 – Class Incremental

2.3.2 – Task Incremental

2.3.3 - Experiences

2.3 - Evaluation

2.3.1 – Common Continual Learning Benchmarks

2.3.1.1 – Image Benchmarks

2.3.1.1.1 – ImageNet

2.3.1.1.2 – CORe50

2.3.1.2 – Natural Video Benchmarks

2.3.1.2.1 – Stream-51

2.3.1.2.2 - OpenLoris

2.3.2 – Model Evaluation

2.3.2.1 – Baselines

2.3.2.2 – Hyperparameters

2.1 – Artificial Neural Networks Background

2.1.1 – Origin of Artificial Neural Networks

Artificial neural networks (ANN), or neural networks (NN) are computing systems that were inspired by the behavior of biological neural networks. A biological neuron is a type of cell found in the nervous system that receives and processes information from other neurons or sensory organs. It then transmits signals to other neurons or muscle cells. It consists of a cell body, dendrites that receive the input signals, an axon that transmits signals to other neurons and synapses that connect the axon of one neuron to the dendrites of another.

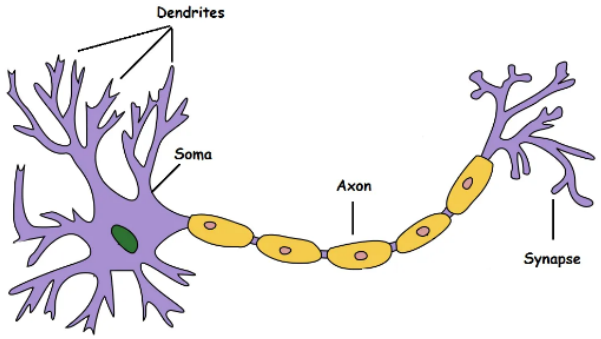


Figure 2.1: A Biological Neuron

(Source: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1)

In a similar fashion, artificial neural networks consist of a series of interconnected units or nodes known as artificial neurons, which are inspired by the neurons found in a biological brain. Each connection, akin to the synapses in a living brain, can transmit signals to other neurons. An artificial neuron takes in these signals, processes them, and can then send signals to the neurons it is connected to. The links between neurons are referred to as edges, and usually, both neurons and edges possess weights that are adjusted during the learning process.

Neurons are grouped into layers, with each layer potentially performing distinct transformations on their inputs. Signals move from the initial layer (the input layer) to the final layer (the output layer), passing through all intermediate layers (hidden layers) along the way.

Nowadays, a plethora of different ANN architectures are being utilized to solve different problems.

2.1.2 – Variations of Artificial Neural Networks

2.1.2.1 – Multi-Layer Perceptron (MLP)

A multilayer perceptron is a fully connected category of feedforward artificial neural network. It is made of a minimum of three layers of nodes: an input layer, a hidden layer, and an output layer. The perceptrons in one layer are connected to the ones in the adjacent layer by a set of trainable weights, which are adjusted during training to minimize the error between the predicted and true output values.

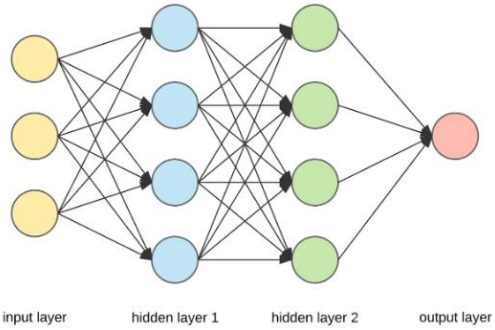


Figure 2.2 – A fully-connected artificial neural network.

(Source: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6)

Excluding the input node, each node is a neuron that uses a nonlinear activation function. The output layer typically produces a classification or regression output, depending on the task.

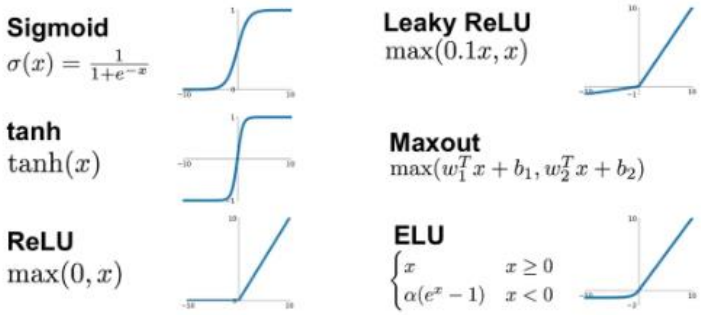


Figure 2.3 – A few of the activation functions.

(Source: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6)

MLPs are widely used for a variety of tasks, including image classification, natural language processing, and speed recognition. They have been shown to be effective in many applications and can be easily adapted to handle a wide range of input data formats.

2.1.2.2 – Convolutional Neural Network (CNN)

Convolutional neural networks are a class of artificial neural networks, most used for image recognition and classification tasks, due to their ability to learn hierarchical representations of image features.

They are made up of multiple layers, typically including convolutional layers, pooling layers, and fully connected layers. Each layer performs a specific type of computation on the input data.

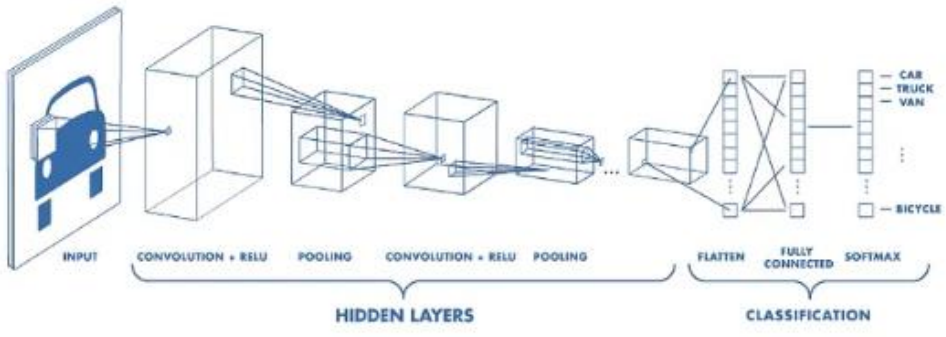


Figure 2.4 – Architecture of a CNN.

(Source: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939)

Convolution Layer

The convolutional layer serves as the primary component of a CNN. It conducts a dot product between two matrices, the filter, and a limited section of the receptive field. As the forward pass progresses, the kernel moves across the image, generating a collection of feature maps. These filters adapt to identify particular in the input, like edges or corners within an image. By arranging several convolutional layers together, a CNN can learn to detect intricate patterns in the input data, such as shapes and objects.

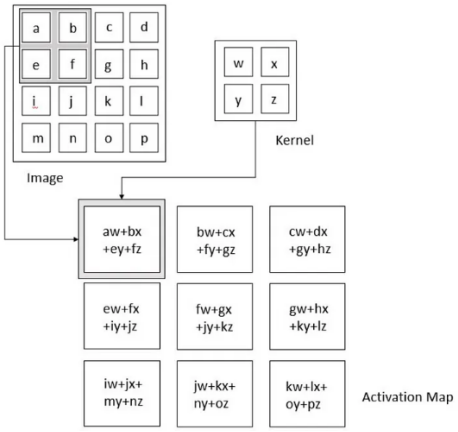


Figure 2.5 – Convolution Operation.

(Source: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939)

Pooling Layer

The pooling layer contributes to the reduction of the spatial size of the representation, which in turn decreases the computational requirements and weights. This process is carried out independently on each slice of the representation. Max pooling is the most prevalent technique, as it records the highest output within the local region.

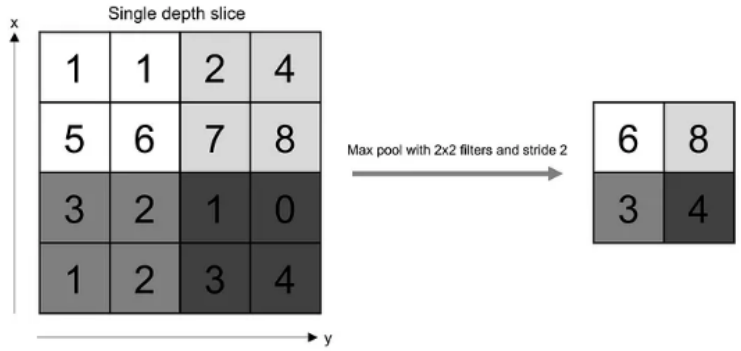


Figure 2.6 – Pooling Operation.

(Source: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939)

Fully connected Layer

Lastly, a fully connected layer takes the output from the previous layer which is a flatten layer that transforms the input into a one-dimensional form. The fully connected layer is used to be able to provide classification on the problem by producing a set of probabilities for each possible class label.

2.1.2.2.1 – MobileNet

The MobileNet is a type of neural network architecture for deep learning that was designed for use in mobile and embedded applications with limited computational resources. It is TensorFlow's first mobile computer vision model.

The architecture uses depthwise separable convolutions, a combination of depthwise convolutions and pointwise convolutions, to reduce the computational cost and model size while maintaining the accuracy of the network.

Depthwise Convolution

A depthwise convolution applies a single filter to each input channel.

Pointwise Convolution

Regarding pointwise convolution, it employs a 1x1 convolution to merge the outputs of the depthwise convolution. Unlike a standard convolution that filters and combines inputs into a new set of outputs in a single step, the depthwise separable convolution divides this process into two layers – one dedicated to filtering and another to combining. Consequently, this factorization reduces computation and model size. [4]

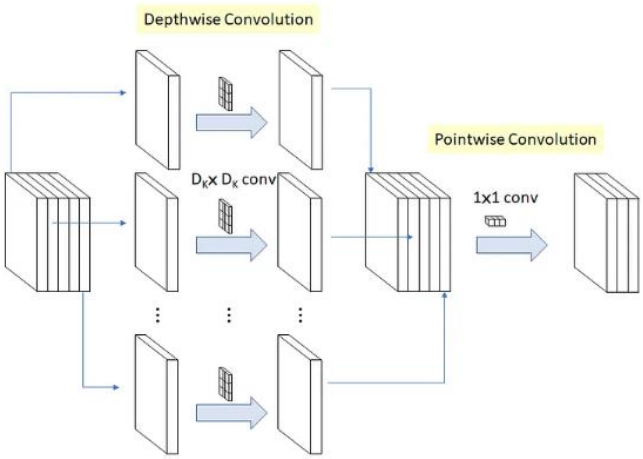


Figure 2.7 – Depthwise Separable Convolution.

(Source: https://medium.com/analytics-vidhya/image-classification-with-mobilenet-cc6fbb2cd470)

2.1.2.2.2 – EfficientDet

EfficientDet is a state-of-the-art object detection model which aims to achieve higher accuracy with fewer parameters and computations.

The EfficientDet architecture consists of a backbone network, a feature network, and a prediction network.

The backbone network is a convolutional neural network called “EfficientNet” which takes an input image and generates feature maps at multiple scales.

The feature network, called “bi-directional feature pyramid network” (BiFPN) is used to combine the feature maps from the backbone network to produce a single feature map.

Lastly, the feature map is then used by the prediction network, called “Box prediction net” to make predictions. The prediction network consists of multiple convolutional

layers and fully connected layers that generate class probabilities and bounding box offsets for each object.

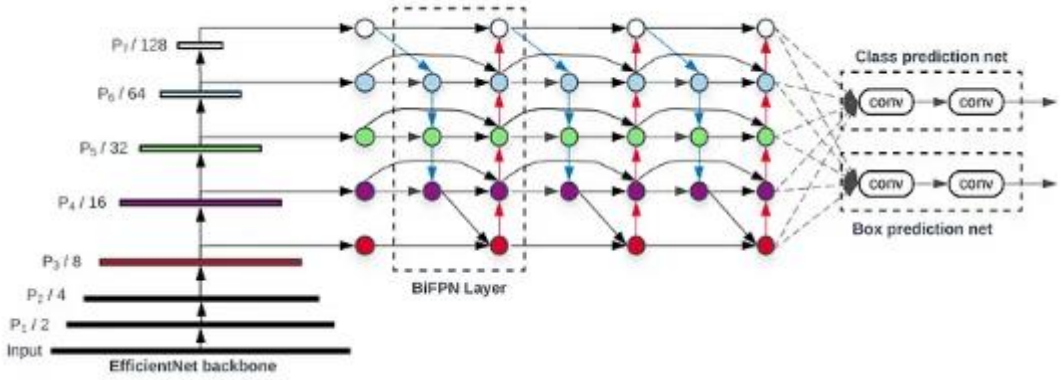


Figure 2.8 – EfficientDet architecture: EfficientNet as the backbone network, BiFPN as the feature network and box shared class/box prediction network.
(Source: https://medium.com/analytics-vidhya/efficientdet-scalable-and-efficient-object-detection-384a5df9011a)

EfficientDet has several benefits over other CNNs, including higher accuracy, smaller model size, and faster inference speed. It achieves state-of-the-art performance on several object detection benchmarks.

What is more, is that there is an EfficientDet-Lite which is the EfficientDet model optimized for TF-Lite, designed for performance on mobile CPU, GPU and EdgeTPU [20].

2.1.3 – Training Artificial Neural Networks

2.1.3.1 - Loss function

Loss functions serve to quantify the discrepancy between the predicted output and the actual output generated by a machine learning model. Gradients, derived from the loss function, are utilized to update the weights.

Several types of loss functions exist, such as:

Mean Squared Error

Mean squared error is used in regression contexts where both expected and predicted outcomes are real-number values. The formula calculates the squared difference between the expected value and the predicted value.

$$L = (y_i - \hat{y}_i)^2$$

Figure 2.9 – Mean Squared Error formula.

(Source: https://programmathically.com/an-introduction-to-neural-network-loss-functions/)

Cross-Entropy

Cross-Entropy based loss functions are typically used in classification situations. They measure the divergence between the predicted probability distribution and the true probability distribution of the target classes, resulting in a value known as the loss.

Binary Cross-Entropy

Binary Cross-Entropy is suitable for binary classification settings, where one of two possible outcomes is expected. The loss is computed using a formula where y represents the expected outcome, and y hat represents the outcome generated by the model.

$$L = -(y_i \; log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

Figure 2.10 – Binary Cross-Entropy formula.

(Source: https://programmathically.com/an-introduction-to-neural-network-loss-functions/)

2.1.3.2 - Gradient Descent

Gradient Descent is an optimisation algorithm used to find a local minimum/maximum of a given function. It is widely used in machine learning and deep learning with the goal of minimizing a cost/loss function.

The model's parameters are initialized with random values, and the algorithm computes the loss function using the training data and current parameter values. The gradient of the loss function concerning each model parameter is then calculated. Finally, the algorithm updates the parameters/weights by subtracting the gradient multiplied by a learning rate, a small positive number.

$$\Delta w_{ij} = -n \frac{dE}{dw_{ij}}$$

Figure 2.11 – Gradient descent change.

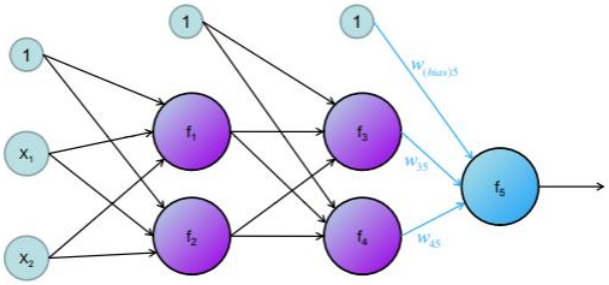
(Source: Laboratory lectures of CS442 course at the University of Cyprus)

The learning rate determines the size of the steps taken in the direction of the negative gradient. Too high or too low learning rates can lead to suboptimal convergence or convergence failure. Choosing an appropriate learning rate is crucial for the convergence of the optimization algorithm.

2.1.3.3 – Backpropagation

The backpropagation algorithm is probably the most fundamental building block in neural networks. It is used to effectively train a neural network through a method called chain rule. It propagates the error backwards starting from the output layer so we will know using the gradient descent how much to change each weight.

It is used after each forward pass through a network. The forward pass calculates the predicted output.



$$y_5 = f_5(w_{(bias)5} + w_{35} \cdot y_3 + w_{45} \cdot y_4)$$

Figure 2.12 – An example of a forward pass for calculating f5 output. Y are the outputs of the neurons.

(Source: Laboratory lectures of CS442 course at the University of Cyprus)

The backward pass adjusts the model's parameters (weights and biases).

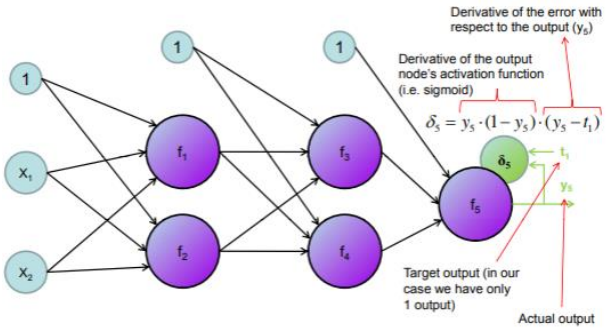


Figure 2.13 – An example of a backward pass. Calculating derivatives and deltas.
(Source: Laboratory lectures of CS442 course at the University of Cyprus)

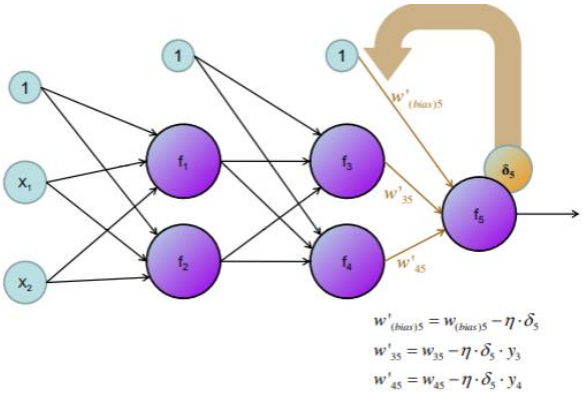


Figure 2.14 – An example of a backward pass. Updating the weights and biases.
(Source: Laboratory lectures of CS442 course at the University of Cyprus)

2.1.3.4 – Overfitting/Underfitting

Overfitting and Underfitting are two primary issues that arise in machine learning, leading to decreased performance in machine learning models. The goal in machine learning is to ensure that models generalize well. Generalization refers to a model's ability to produce appropriate outputs by adapting to a given set of unknown data. Overfitting and underfitting are two factors that must be assessed to determine if a model is generalizing effectively.

Overfitting

Overfitting occurs in a model when it attempts to fit all the data points present in a dataset that it is being trained on. It learns the training data too well, to the point that it starts to capture the noise of random variations in the data, as well as the underlying patterns. As a result, this leads to extremely poor generalization performance on new, unseen data. The overfitted model has low bias and high variance.

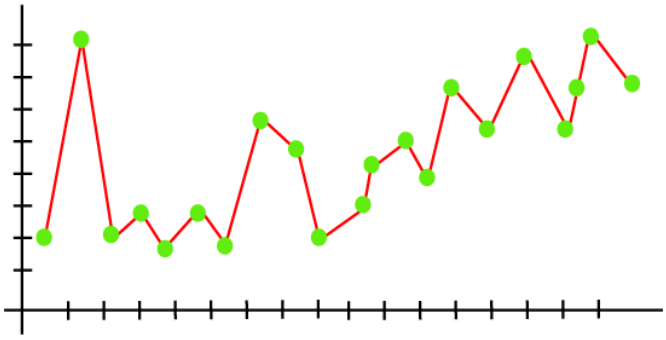


Figure 2.15 – An example of a model overfitting on the training dataset.

(Source: https://www.javatpoint.com/overfitting-and-underfitting-in-machine-learning)

There are a variety of ways to minimize the likelihood of overfitting in our model such as:

- Cross-Validation
- Training with mode data
- Removing features
- Early stopping the training
- Regularization

Underfitting

Underfitting is a condition where the machine learning model is unable to recognize the underlying trend of the data. This occurs when the fed training data stops at an early stage to prevent overfitting, however it leads to the model learning insufficiently from the data. As a result, the model fails to find the best fit of the dominant trend in the data, thereby leading to less accurate and unreliable predictions.

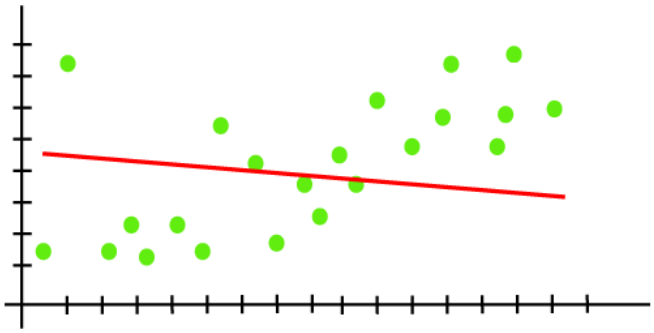


Figure 2.16 – An example of a model underfitting, unable to capture the datapoints in the plot. (Source: https://www.javatpoint.com/overfitting-and-underfitting-in-machine-learning)

In order to prevent underfitting, we can increase the training time of the model as well as increasing the number of features.

2.1.3.5 – Regularization Techniques

Regularization is a technique used to prevent overfitting in a model. It adds new constraints or penalties into the model to encourage it to prioritize simpler solutions that are less likely to overfit.

A few of the most used regularization techniques in machine learning are L1 and L2 regularization, dropout, early stopping and data augmentation.

L1 & L2 Regularization

L1 and L2 regularization are the most prevalent types. They modify the overall cost function by incorporating an additional term known as the regularization term.

Dropout

The dropout method randomly selects some nodes during each iteration and removes them, along with all their incoming and outgoing connections. This forces the model to learn more robust features.

Early Stopping

Early stopping involves reserving a portion of the training set as a validation set. When the performance on the validation set starts to deteriorate, training on the model is halted immediately.

Data Augmentation

Lastly, in data augmentation we increase the size of the training data by creating additional synthetic training data from the existing training data. We apply various operations such as rotation, flipping, zooming, cropping, changing brightness or contrast to the existing training data to create new variations of the same data. They are similar enough to the original data to allow the model to learn from them, but different enough that it helps prevent overfitting.

2.2 – Continual Learning Scenarios

With Continual Learning we strive to achieve training of a machine learning model on a sequence of tasks over time, where the model learns each task while maintaining its performance on previously learned ones.

There are different Continual Learning scenarios that can occur, and they can be categorized in the following types based on how the tasks are presented to the model.

2.2.1 – Class Incremental

In this situation, the model must learn new classes over time without forgetting previously learned classes. It should be capable of solving each task encountered so far and determining which task is being presented. This is a typical real-world problem involving the incremental learning of new object classes and is of great interest to us [21].

An example would be a model that learns to classify animals. It would need to learn new animal species while also remembering the previously learned ones.

2.2.2 – Task Incremental

In a task incremental scenario, the model is always informed about the task it needs to perform. This is the simplest Continual Learning scenario, as the task-ID is always provided. To address this scenario, output units are typically dedicated to each task, while the remainder of the network is shared among tasks.

An example would be a model that recognizes speech commands. It will need to recognize new speech commands without forgetting the previously learned ones. Each command is considered a task and no classes exist [21].

2.2.3 – Domain Incremental

Lastly, we have the domain incremental scenario. It is similar to the task incremental scenario; however, the task identity (task-ID) is not available during test time. The model only needs to solve the task and not required to infer which task is it.

An example would be an agent that needs to adapt to different environments to survive, without requiring explicit identification of the environment [21].

2.3 – Evaluation

2.3.1 – Common Continual Learning Benchmarks

In order to evaluate the performance of continual learning models, it is necessary to have benchmark datasets that allow the comparison of different approaches. There are several datasets that have been proposed for this purpose. They cover a wide range; however, we are most interested in the ones that are mostly used for image recognition and object detection in a continual learning scenario.

2.3.1.1 – Image Benchmarks

Image benchmarks are datasets consisting of static images, mostly used for classification and object detection. The most popular image benchmarks that are used for continual learning are ImageNet and CORe50.

2.3.1.1.1 – ImageNet

ImageNet is a large-scale dataset of images that has been used as a benchmark for image classification since 2010. The dataset contains over 14 million images, each belonging to one of 1000 predefined classes. The images have been collected from the web and manually annotated with class labels. ImageNet has been used to evaluate the

performance of many state-of-the-art image classification algorithms, including deep neural networks and it is also used to evaluate continual learning models [17].

2.3.1.1.2 – CORe50

CORe50 is a recent benchmark dataset that has been proposed specifically for the task of continual learning. Datasets such as ImageNet has been designed with "static" evaluation protocols in mind with the dataset being split in a training set and a test set. On the other hand, CORe50 focuses on being able to more effectively train and test continuous learning approaches. It does so by presenting multiple views of the same objects taking in different environments (lighting, pose, background etc.).

The dataset consists of 50 object categories, each with 10000 images. The dataset is designed to simulate a continual learning scenario, where the model must learn to recognize new classes while also retaining its ability to identify previously learned classes [11].

Figure 2.17 – Example Images of the 50 objects in CORe50. Each column denotes one of the 10 categories.

(Source: https://arxiv.org/abs/1705.03550.pdf)

2.3.1.2 – Natural Video Benchmarks

Natural video benchmarks are datasets that contain videos that capture a dynamic and continuous stream of visual information. They are particularly useful for evaluating the performance of machine learning models that need to process continuous streams of

data and adapt to changing visual contexts over time. Two examples of widely used natural video benchmarks are Stream-51 and OpenLoris.

2.3.1.2.1 – Stream-51

Stream51 is a benchmark dataset for continual learning in videos, which is an extension of the CORe50 dataset. It consists of 51 different complex actions or events such as playing the guitar, eating, and brushing teeth, with each event containing multiple videos. The dataset is designed to evaluate the ability of models to learn and adapt to new tasks and data streams in the video domain [24].

2.3.1.2.2 – OpenLoris

OpenLoris is a lifelong robotic vision dataset collected via RGB-D cameras. The dataset includes difficulties that a robot could encounter in a real-life situation and offers benchmarks to assess the effectiveness of a lifelong object recognition algorithm [19].

2.3.2 – Model Evaluation

Model evaluation is a process in which various metrics are used to assess the performance of a machine learning model, as well as its strengths and weaknesses. Evaluating a model's effectiveness and optimizing it for better performance is crucial.

2.3.2.1 – Baselines

Baseline models assist in the evaluation process. A baseline model is essentially a simple model that serves as a reference in machine learning. It provides context for the results of trained models. Typically, there are upper and lower bound baseline models that function as benchmarks.

In the context of continual learning some of the common baselines are explained below:

Naïve

We have the naïve approach where we just do a continuing backpropagation. It is an approach prone to forgetting and usually used as a lower bound.

JointTraining / Offline

On the other hand, JointTraining/Offline is a pure multi-task learning approach, where it is assumed that access to all data is available, and all tasks are trained simultaneously. This serves as an upper bound.

Greedy Sampler and Dumb Learner (GDumb)

Another intriguing control baseline method that has questioned the progress of recently proposed approaches for continual learning is GDumb. This method greedily stores samples in memory as they arrive, and at test time, it trains a model from scratch using only the samples in memory. There is no knowledge transfer in this strategy. However, despite its simplicity, it has achieved impressive performance that outperforms numerous existing and more complex continual learning strategies. As a result, it is important that any new continual learning approaches should be tested against GDumb. If they cannot outperform it, then something is definitely wrong about the proposed strategy [16].

Cumulative

Lastly, we have the cumulative approach, where for every experience, we accumulate all data and re-train from scratch. This approach is unsustainable over time and is used as an upper bound.

2.3.2.2 – Hyperparameters

A hyperparameter is a parameter with a value set before the learning process starts, and it is used to regulate the learning process. They are used by the learning algorithm, but they are not part of the resulting model. Choosing the most appropriate hyperparameters is incredibly important in obtaining the best results from our model.

Some of the most important hyperparameters that we can choose is our train-test split ratio, our learning rate, the type of activation function that we choose as well as the optimization algorithm.

2.3.3 – Model Metrics

One of the major factors for evaluating continual learning models is the focus on preventing forgetting of the data used in our model. However, there is a more comprehensive set of metrics that appear to have practical implications worth considering, especially when we are trying to deploy a real AI system that learns continually.

2.3.3.1 – Accuracy

Accuracy is a fundamental metric used to evaluate the performance of a model. It measures the proportion of correct predictions made by the model relative to the total number of predictions. A high accuracy indicates that the model is effective at classifying or predicting the correct outcomes. However, it is essential to consider other metrics as well, particularly in cases with imbalanced datasets [3].

2.3.3.2 – Samples storage size efficiency

In many Continual Learning approaches, training samples are stored using a replay strategy to prevent forgetting. Sample Storage Size Efficiency is a metric that measures the efficiency of the model's memory usage in relation to the storage of training samples. A model with high Sample Storage Size Efficiency can maintain good performance while requiring less memory for storing training data, making it more suitable for real-world applications with storage constraints [3].

2.3.3.3 – Computational efficiency

Computational Efficiency is a metric that measures the number of computational resources, such as multiplication and addition operations, required for a model to perform its tasks, including training and inference. A computationally efficient model can achieve good performance with less processing power and time, making it more practical for real-world applications where computational resources may be limited [3].

2.4 – Continual Learning Methodologies

In this section, we provide a brief overview of the most popular continual learning strategies found in the literature. These methodologies generally aim to address the challenges inherent in learning new tasks sequentially while preserving knowledge from previous tasks. Key approaches include architectural modifications, regularization techniques, rehearsal strategies, and hybrid methods that combine elements from various strategies to create more robust and efficient solutions for continual learning.

2.4.1 – Architectural Strategies

Architectural strategies are techniques used in continual learning to modify the architecture of a neural network. By modifying the architecture of the neural network, we are able to create more efficient and effective models that can learn new tasks without forgetting the old ones.

Some of the most used architectural strategies are the following:

Multi-Head Architectures

Multi-head architectures in continual learning are approaches where the model has multiple output heads, each responsible for a different task. Each head is essentially a separate classifier. It allows the model to separate the representation space for different tasks in an attempt to reduce the likelihood of catastrophic forgetting.

An example of a multi-head architecture is the MORE technique which can be seen in [8] that uses the multi-head approach along with a transformer network. A multi-head network is trained as an adapter to a pre-trained network with each head being an out-of-distribution detection model for a task. In addition, hard attention masks are utilized to prevent forgetting.

Progressive Neural Networks (PNNs)

The PNN architecture consists of a series of connected sub-networks, with each sub-network dedicated to learning a particular task. As new tasks are introduced, a new sub-network is added to the architecture, which learns the new task while the previously learned tasks are still preserved in the existing sub-networks.

During training, the output of the previous sub-networks is used as input to the new sub-network, allowing the new sub-network to build on the knowledge learned from the previous sub-networks. This process of building new sub-networks for new tasks while preserving the knowledge of previously learned tasks continues over time, resulting in a progressively expanding network that can handle a wide range of tasks. without forgetting previous knowledge [18].

2.4.2 – Regularization Strategies

Regularization strategies, as mentioned earlier, are techniques that impose constraints on the model's parameters. Besides the common regularization techniques to prevent overfitting, there are certain strategies used to address the issue of catastrophic forgetting with some of the most known explained below:

Learning Without Forgetting (LWF)

The idea behind Learning Without Forgetting is to use the outputs of the previous model to constrain the training of the new one. During the training process, the model is trained on both the old and new data. The old data is replayed, and the new data is trained on the model with the previous weights.

To prevent the model from completely forgetting the old knowledge, the model's output on the old data is compared with the output of the previous model. The difference between the two outputs is minimized by adding a distillation loss to the training objective. This way, the model is forced to maintain the same outputs on the old data as the previous model [10].

Elastic Weights Consolidation (EWC)

Elastic Weights Consolidation works by selectively retraining important weights for solving previously learned tasks while allowing the network to adjust its weights for new tasks. For example, we have two tasks A and B and the EWC method of selective regularization of parameters theta. After we learn A, the regularization method figures out which parameters are important for A and penalizes any change made to the network parameters according to their importance while learning B [1].

The penalty term that is used to constrain the changes in the important weights is calculated using the Fisher information matrix. The matrix is used to compute a diagonal matrix of importance weights which assigns high importance to parameters that are most critical for solving the previously learned tasks.

When we train the network on a new task, EWC adds a regularization term to the loss function that encourages the weights to remain close to their previous values. The amount of regularization is dependant to the importance weights that were computed with the Fisher information matrix [9].

2.4.3 – Rehearsal Strategies

Replay is a very general and effective strategy for Continual Learning.

Things to consider when choosing your implementation option:

- Fixed or “adaptive” external memory?
- Sample selection: random or representative examples?
- Mini-batch sample selection – what examples to choose and use in the current mini-batch?
- Separate buffers per class / tasks / notable distributions?
- Sample based on time – different timescales?
- Sample replacement – which examples to throw away when the memory is full?

There is not really a clear answer to all these questions. It depends on the scenario / problem you are solving, and a lot of experimenting is needed when choosing the correct implementation.

Some approaches that are used are the following:

Random Replay

The main idea of the random replay approach is to sample randomly a subset of the previously observed data and includes them in the current training process along with the current batch of data with the goal of mitigating the forgetting of previously learned information.

One variation of the random replay strategy is to fill a fixed-size memory with examples from previous experiences. When we train a new experience, a random subset of the fixed memory is sampled along with the current data. This allows the model to continue learning from previously seen data while adjusting to the new data as well.

We can also take a different approach and replace examples of the random replay randomly to maintain an approximate equal number of examples for experience, preventing the model to become biased towards a particular experience.

Latent Replay

Replay in the input space is inefficient. What we do is storing a subset of previously learned data in a buffer and replaying it during the training of new data. We do not replay the raw data but the features that were extracted from it. This means that the latent replay buffer is towards the end of our network near our output layer. As a result, it allows the model to retrain the key features of the previous data and prevent catastrophic forgetting while also avoiding the issue of storing and handling big amounts of raw data.

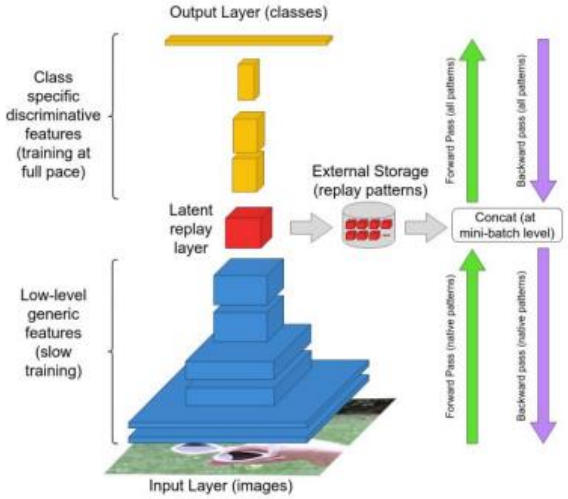


Figure 2.18 – A latent replay buffer replaying the latent activations of the model. (Source: https://arxiv.org/abs/1912.01100.pdf)

Generative replay

In generative replay, instead of a replay memory we generate examples that are then used to augment the current training data, helping the model retain the knowledge from the past tasks. Although in theory it seems like a promising approach, in reality is it still

challenging to scale this approach on high-dimensional data on complex problems and find good accuracy-efficiency trade-offs.

Dynamic Data Removal (DDR)

It is worth mentioning the Dynamic Data Removal strategy, which is a data efficiency strategy that can be used to increase the efficiency of the replay buffer. It is a simple yet effective strategy to reduce training data for further acceleration. It was proposed as one of the methodologies of the SparCL framework proposed on [22] at NeurIPS 2022. DDR measures the importance of each training example by the occurrence of misclassification during continual learning, and gradually removes training data at the end of each stage based on a specific policy.

The policy involves calculating the total number of misclassifications for each training example on each stage and removing a proportion of training samples with the least number of misclassifications. As a result, by removing "easier" samples, the probability of saving "harder" samples to the buffer is increased and the construction of a more informative buffer is archived without heavy computation.

Furthermore, DDR attempts to solve the data imbalance issue of the buffer since it is of much smaller size than the training set. The data removal proportion for each task is set as $\rho \in [0, 1]$, and a cut-off stage is defined to control the trade-off between efficiency and accuracy.

Setting the cut-off stage earlier leads to faster training for the following stages, but we need to be cautious not to lead to underfitting of the removed data.

2.4.4 – Hybrid Strategies

In the pursuit of improving continual learning algorithms, researchers have explored a variety of strategies including architectural, regularization, and rehearsal methods. Each strategy has its own strengths and weaknesses, making them better suited for specific scenarios. Interestingly, biological learning systems seem to utilize multiple approaches for continual learning, suggesting that hybrid approaches may be effective in machine learning as well. Despite their potential, hybrid strategies are still relatively

underexplored in the field and could lead to more optimal effectiveness and efficiency trade-offs.

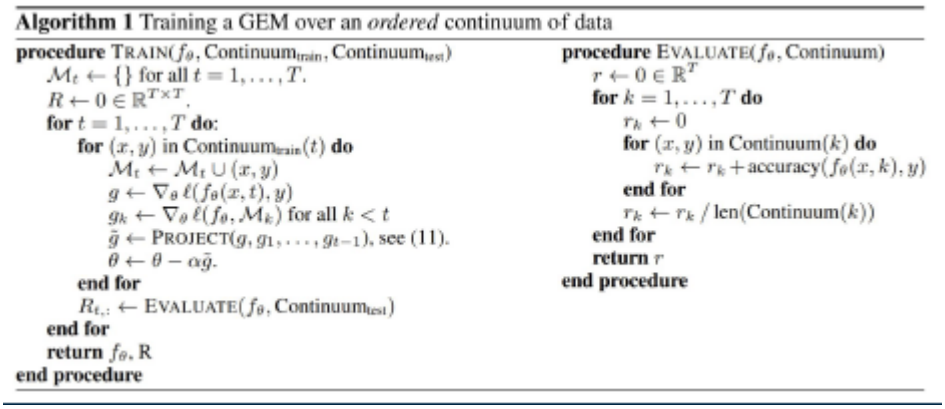
Some of the most popular hybrid approaches in continual learning are described below:

Gradient Episodic Memory (GEM)

Gradient Episodic Memory (GEM) is a hybrid strategy for continual learning that combines the use of a memory buffer with gradient-based regularization. GEM aims to prevent catastrophic forgetting by storing past task information in a memory buffer and using it to constrain the optimization process during the learning of new tasks.

During the training of a new task, GEM modifies the gradient descent step to ensure that the current task's loss does not increase with respect to the stored tasks. The stored task information is updated by storing the current task's gradients and using them to update the memory buffer with a projection operation that ensures that the stored task information is not overwritten.

The hybrid approach of GEM allows for effective use of the memory buffer while minimizing the risk of negative interference between the current and stored tasks [12].



Algorithm 1 Training a GEM over an *ordered* continuum of data

procedure TRAIN(f_θ, Continuum$_{\text{train}}$, Continuum$_{\text{test}}$)
  $\mathcal{M}_t \leftarrow \{\}$ for all $t = 1, \ldots, T$.
  $R \leftarrow 0 \in \mathbb{R}^{T \times T}$.
  for $t = 1, \ldots, T$ **do**:
    for (x, y) in Continuum$_{\text{train}}(t)$ **do**
      $\mathcal{M}_t \leftarrow \mathcal{M}_t \cup (x, y)$
      $g \leftarrow \nabla_\theta \ell(f_\theta(x, t), y)$
      $g_k \leftarrow \nabla_\theta \ell(f_\theta, \mathcal{M}_k)$ for all $k < t$
      $\tilde{g} \leftarrow$ PROJECT$(g, g_1, \ldots, g_{t-1})$, see (11).
      $\theta \leftarrow \theta - \alpha\tilde{g}$.
    end for
    $R_{t,:} \leftarrow$ EVALUATE$(f_\theta,$ Continuum$_{\text{test}})$
  end for
  return f_θ, R
end procedure

procedure EVALUATE(f_θ, Continuum)
  $r \leftarrow 0 \in \mathbb{R}^T$
  for $k = 1, \ldots, T$ **do**
    $r_k \leftarrow 0$
    for (x, y) in Continuum(k) **do**
      $r_k \leftarrow r_k +$ accuracy$(f_\theta(x, k), y)$
    end for
    $r_k \leftarrow r_k$ / len(Continuum(k))
  end for
  return r
end procedure

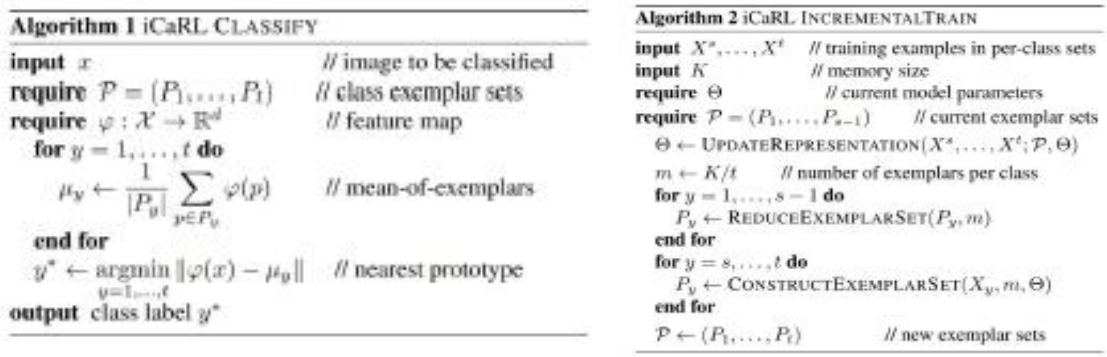
Figure 2.19 – GEM Algorithm

(Source: https://course.continualai.org/)

Incremental Classifier and Representation Learning (iCaRL)

iCaRL is a hybrid strategy for continual learning that combines incremental learning with representation learning. The model uses a memory buffer to store a subset of previously seen examples, and during training, it simultaneously updates the classification model and the representation model.

The classification model is trained using a distillation loss that compares the output probabilities of the current model with the probabilities of the exemplars in the memory buffer. The representation model is trained using a mean-squared loss that encourages the feature vectors of the current model to be similar to the feature vectors of the exemplars in the memory buffer.



Algorithm 1 iCaRL CLASSIFY

input x // image to be classified
require $\mathcal{P} = (P_1, \ldots, P_t)$ // class exemplar sets
require $\varphi : \mathcal{X} \to \mathbb{R}^d$ // feature map
for $y = 1, \ldots, t$ **do**
$\mu_y \leftarrow \frac{1}{|P_y|} \sum_{p \in P_y} \varphi(p)$ // mean-of-exemplars
end for
$y^* \leftarrow \underset{y=1,\ldots,t}{\operatorname{argmin}} \|\varphi(x) - \mu_y\|$ // nearest prototype
output class label y^*

Algorithm 2 iCaRL INCREMENTALTRAIN

input $X^s, \ldots, X^t$ // training examples in per-class sets
input K // memory size
require Θ // current model parameters
require $\mathcal{P} = (P_1, \ldots, P_{s-1})$ // current exemplar sets
$\Theta \leftarrow$ UPDATEREPRESENTATION$(X^s, \ldots, X^t; \mathcal{P}, \Theta)$
$m \leftarrow K/t$ // number of exemplars per class
for $y = 1, \ldots, s-1$ **do**
$P_y \leftarrow$ REDUCEEXEMPLARSET(P_y, m)
end for
for $y = s, \ldots, t$ **do**
$P_y \leftarrow$ CONSTRUCTEXEMPLARSET(X_y, m, Θ)
end for
$\mathcal{P} \leftarrow (P_1, \ldots, P_t)$ // new exemplar sets

Figure 2.20 – iCaRL Algorithm.

(Source: https://course.continualai.org/)

AR1: A Flexible Hybrid Strategy for Continual Learning

AR1 is a hybrid strategy in continual learning that combines architectural, regularization and replay components. Copy-weights with Re-init (CWR) but improved with mean-shift and zero initialization (called CWR*), as the architectural component, online synaptic intelligence as the regularization component, and a latent replay as the replay component. Details on the strategy can be found on [13].

2.5 – Future Directions

In this section, we will look at some exciting directions for future research in continual learning. Three areas that have recently received considerable attention are Continual Meta-Learning, Continual Reinforcement Learning, and Continual Unsupervised Learning.

2.5.1 – Continual Meta-Learning (CML)

Continual Meta-Learning (CML) aims to enable fast adaptation to new tasks by learning reusable and task-agnostic representations from previous tasks. Recent research has shown the potential of CML in reducing forgetting and improving generalization in

continual learning. However, there is still much to be done to fully utilize the benefits of CML [5].

2.5.2 – Continual Reinforcement Learning (CRL)

Continual Reinforcement Learning (CRL) is an important direction for research due to its potential to enable agents to learn from and adapt to a changing environment without forgetting previously learned skills. This area is particularly challenging as it involves balancing the exploration of new skills with the preservation of previously learned ones [7].

2.5.3 – Continual Unsupervised Learning (CUL)

Continual Unsupervised Learning (CUL) is an area of continual learning that aims to learn from a continuous stream of unsupervised data. The potential of CUL lies in its ability to learn from real-world data that is often plentiful but unlabelled. However, it poses many challenges, such as maintaining the quality of the learned representations and avoiding catastrophic forgetting [23].

Overall, these areas offer exciting opportunities for future research in continual learning, with the potential to greatly improve the effectiveness and efficiency of machine learning models in real-world applications.

Chapter 3

3 –Application Implementation

3.1 – Overview of existing continual learning demo app

3.1.1 – TensorFlow and TensorFlow-Lite

3.1.1.1 – TensorFlow

3.1.1.2 – TensorFlow-Lite

3.1.2 – Personalized Machine Learning

3.1.3 – Transfer Learning

3.1.4 – TensorFlow-Lite Object Classifier App

3.1.5 – Transfer Learning Pipeline

3.1.6 – Extending the TensorFlow-Lite Demo App for Continual Learning

3.1.6.1 – Limitations of Transfer Learning

3.1.6.2 – Introducing Continual Learning to TensorFlow-Lite

3.2 – Updated TensorFlow-Lite Model Personalization demo app

3.2.1 – Improvement Over the Previous Approach

3.2.2 – Overview of the new demo app

3.2.3 – Modifications and Implementation Details

3.2.4 – On-Device Training and Continual Learning Implementation

3.2.5 – Showcasing Different Scenarios

3.2.5.1 – Cumulative Scenario – Batch Training

3.2.5.2 – New Instances Scenario – Batch Training

3.2.5.3 – New Classes Scenario – Batch Training

3.1 – Overview of existing continual learning demo app

3.1.1 – TensorFlow and TensorFlow-Lite

3.1.1.1 – TensorFlow

TensorFlow is an open-source machine learning library developed by Google that has gained popularity among researchers and developers in artificial intelligence. It provides a comprehensive and flexible platform for creating, training, and deploying machine learning models across a wide range of applications. TensorFlow lets users create complex neural networks easily, using a high-level API and many pre-built functions. It also offers support for various hardware accelerators, such as GPUs and TPUs, which allows for efficient and parallelized computation. The library's ecosystem includes numerous tools, such as TensorBoard for visualization, TensorFlow Hub that contains pre-trained models and TensorFlow-Lite making it a powerful tool for machine learning practitioners.

3.1.1.2 – TensorFlow-Lite

TensorFlow Lite is a lightweight version of the TensorFlow library designed specifically for deploying machine learning models on devices with limited resources, such as mobile phones and embedded systems. It focuses on optimizing model size, latency, and power consumption, making it ideal for on-device inference. TensorFlow-Lite allows developers to convert TensorFlow models into a smaller, more efficient format that can run easily on mobile devices and other edge hardware. The framework supports a wide range of pre-trained models and enables the implementation of custom models, offering flexibility and adaptability for various use cases. By bringing machine learning capabilities to resource-constrained devices, TensorFlow-Lite empowers developers to create intelligent applications that can run efficiently on the edge, without relying on constant connectivity or powerful servers.

3.1.2 – Personalized Machine Learning

Personalized machine learning aims to tailor models to individual users' needs, offering a more customized and relevant user experience. Traditional machine learning models rely on massive datasets to be as generic and unbiased as possible. However, this

approach may not always cater to specific user requirements. Moreover, sending user data to the cloud for model training raises concerns regarding privacy, data consumption, and power usage. On-device training offers a solution to these issues, enabling personalized machine learning while preserving user privacy, saving bandwidth, and working without an internet connection.

3.1.3 – Transfer Learning

Transfer learning is a machine learning technique that leverages pre-trained models, typically developed for data-rich tasks, to solve related but data-poor tasks. By retraining a portion of the pre-trained model's layers (usually the last ones), transfer learning allows models to be quickly adapted to new tasks without requiring extensive training data or computational resources. This approach is highly beneficial for on-device applications, as it enables personalized models to be trained efficiently and effectively, even with limited resources.

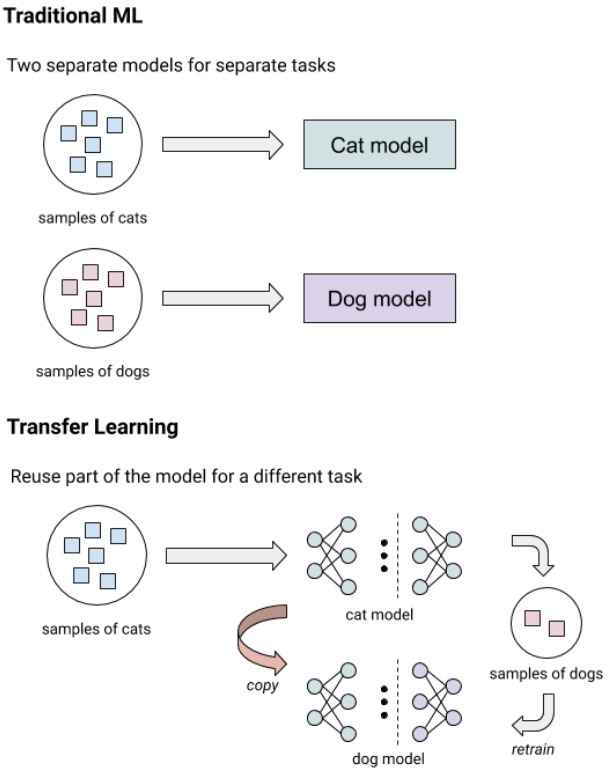


Figure 3.1 – Traditional ML vs Transfer Learning

(Source: https://blog.tensorflow.org/2019/12/example-on-device-model-personalization.html)

3.1.4 – TensorFlow-Lite Object Classifier App

The TensorFlow-Lite demo app as an example project demonstrates on-device transfer learning by classifying images taken from our camera in real-time. The app performs training by capturing sample photos of different classes. By leveraging the power of transfer learning on the MobileNetV2 model pre-trained on ImageNet, the app replaces the last few layers with a trainable softmax classifier. Users can train the model to recognize up to four new classes, with accuracy depending on the complexity of the classes. Even a small number of samples can be sufficient to achieve satisfactory results.

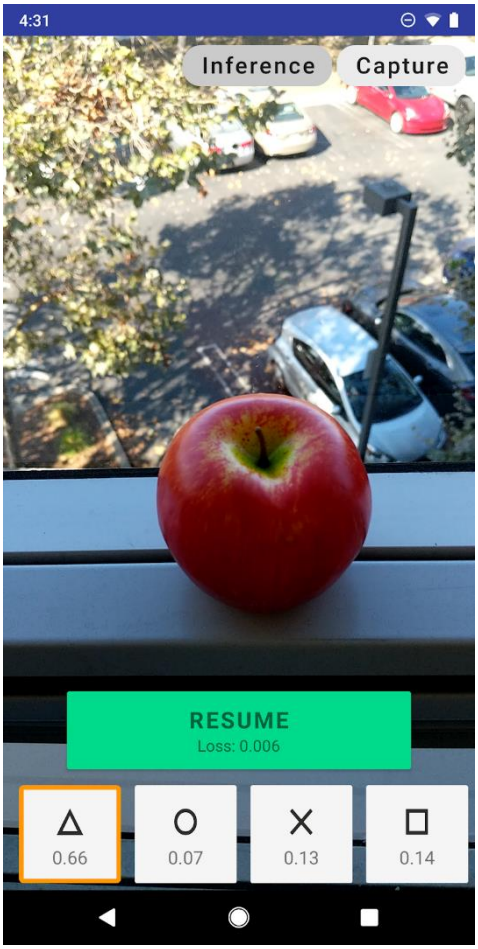


Figure 3.2 – Snapshot of the android application

(Source: https://blog.tensorflow.org/2019/12/example-on-device-model-personalization.html)

3.1.5 – Transfer Learning Pipeline

The transfer learning pipeline includes three main components: the converter, Android library, and TensorFlow-Lite interpreter. The converter generates a transfer learning model by selecting a base model (a deep neural network pre-trained on a generic task)

and a head model (a simpler network that learns from the base model's features to solve the personalized task). The Android library provides an intermediate layer to handle the non-linear lifecycle of the transfer learning model, while the TensorFlow Lite interpreter enables on-device inference. The on-device training is performed on the head of the model, which consists of a dense layer with softmax activations.

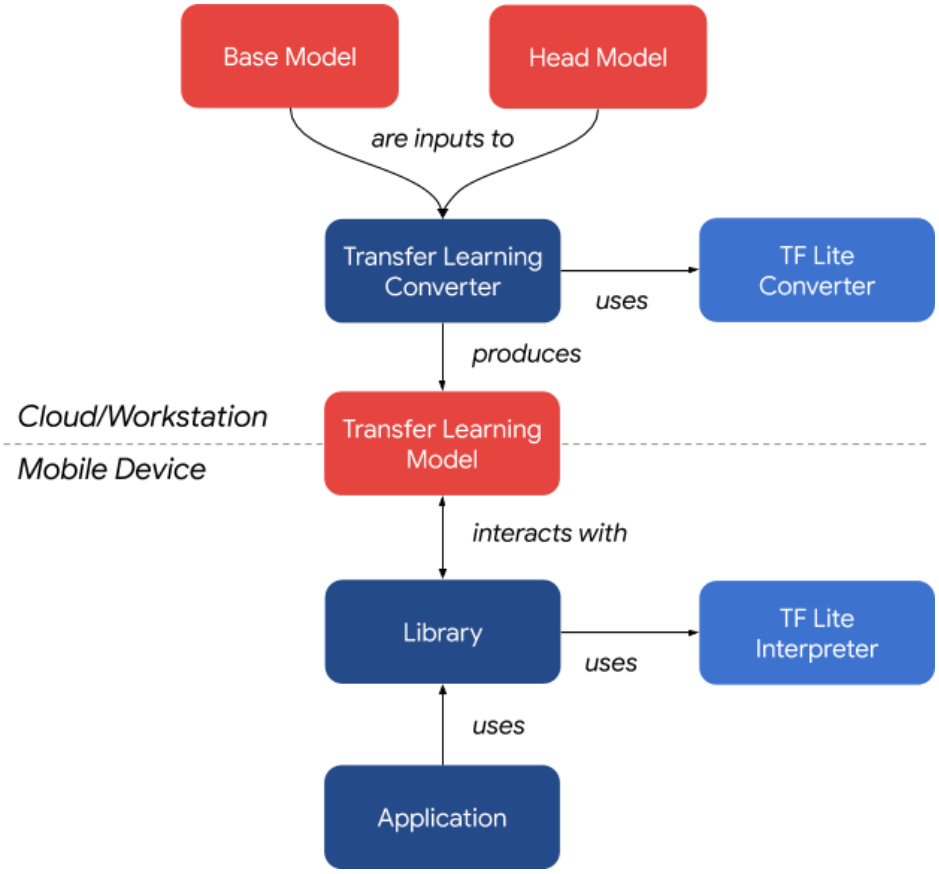


Figure 3.3 – TF-Lite Transfer Learning Pipeline

(Source: https://blog.tensorflow.org/2019/12/example-on-device-model-personalization.html)

3.1.6 – Extending the TensorFlow-Lite Demo App for Continual Learning

3.1.6.1 - Limitations of Transfer Learning

Deploying deep learning models on embedded devices presents numerous challenges, such as privacy concerns, data limitations, network connection issues, and the need for fast model adaptation. Google is currently working on addressing these challenges by embedding an experimental transfer learning API into their TensorFlow-Lite machine learning library. Although transfer learning serves as a good starting point for on-device model training, it has limitations in more realistic scenarios, particularly in terms of catastrophic forgetting. This issue can be seen by the extension that the previous paper

touched upon where it extended the demo app by adding continual learning capabilities and compared the transfer learning and continual learning accuracies of the model on the CoRE50 benchmark [2].

3.1.6.2 - Introducing Continual Learning to TensorFlow-Lite

To overcome the limitations of transfer learning, [2] expanded the TensorFlow-Lite library to include continual learning capabilities by integrating a simple replay buffer approach into the head of the current transfer learning model. The continual learning model was tested on the CORe50 benchmark to show its effectiveness in tackling catastrophic forgetting and its ability to continually learn even under non-ideal conditions, as demonstrated using the developed Android application. The source code of the Android application was open sourced to enable developers to integrate continual learning into their own smartphone applications and facilitate further development of continual learning functionality within the TensorFlow-Lite environment.

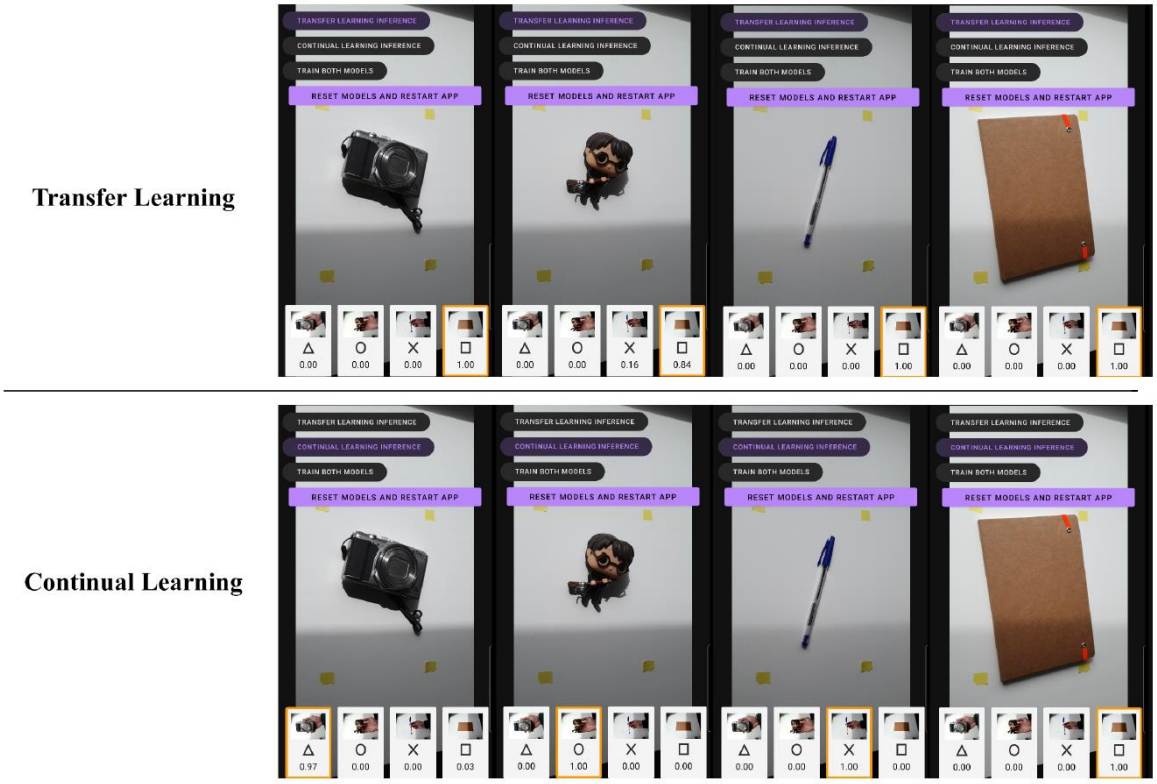


Figure 3.4 – During the New Classes Scenario experiments, the Android application reveals that Continual Learning successfully classifies objects, while Transfer Learning faces significant issues due to catastrophic forgetting caused by incremental training.

(Source: https://arxiv.org/abs/2105.01946.pdf)

3.2 – Updated TensorFlow-Lite Model Personalization demo app

3.2.1 – Improvement Over the Previous Approach

The new demo app addresses several limitations of the previous on-device training approach. These improvements make the new app more efficient and user-friendly for developers and end-users:

Customization

The new approach allows for easier customization of the model structure and optimizers, giving developers more flexibility when building their on-device training applications. We use tf.functions to define the loading of the bottleneck features, the training of the head of the model and for invoking an inference on the given feature which we can alter as we please. The TensorFlow model is then converted to a TensorFlow-Lite model where we can load in our application and with the use of an interpreter, we can call the signatures of our defined functions.

Simplified model management

Unlike the earlier method, which required dealing with multiple physical TensorFlow Lite (.tflite) models, the updated approach uses a single TensorFlow Lite model for both training and inference. This simplification makes model management and deployment more streamlined.

Improved weight storage and updating

The new approach provides an easier way to store and update the training weights. By implementing tf.functions for saving and loading weights, the app can efficiently save the training weights after each epoch and restore them for the next training epoch. This can be used to maintain the trainable weights even when closing and reopening the application.

These improvements offer a more streamlined and efficient on-device training process, making it easier for developers to create personalized applications and end-users to benefit from fine-tuned models that cater to their specific needs.

3.2.2 – Overview of the New Demo App

The updated TensorFlow Lite Model Personalization demo app leverages the latest improvements in TensorFlow Lite to provide an efficient and user-friendly experience for on-device training. The app integrates on-device training into an Android app, allowing users to fine-tune an image classification model based on their specific needs. The demo app follows the same principle as the older demo app which allows us to perform real-time training and classification for up to 4 classes.

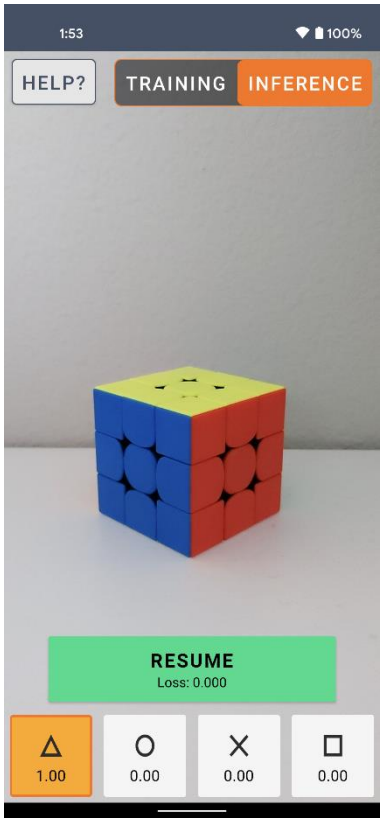


Figure 3.5 – Snapshot of the new TensorFlow-Lite demo application (Source:https://github.com/tensorflow/examples/blob/master/lite/examples/model_personalization/README.md)

3.2.3 – Modifications and Implementation Details

The application follows a series of steps to gather samples, train the model, and perform inference. Here is a well-structured explanation of how the application works:

Gathering samples

The CameraFragment class captures images using the device's camera.

When a user captures an image, it is preprocessed and converted into a TensorImage. The corresponding class of the image is also saved to be used later in the inference process. The image is then passed through the pre-trained feature extractor that uses MobileNetV2, which generates bottleneck features. These features, along with the corresponding class label, are stored as a TrainingSample in the ModelPersonalizationHelper class. All the TrainingSamples are stored in a mutable list to be used later for training the model.

Training

Before we start the training, we make sure to get our .tflite model and load it into the interpreter to be able to use the signatures of the functions that we defined. Once enough samples are collected and we press the train button, the TransferLearningHelper class starts training the model. It first determines the training batch size, which depends on the total number of training samples collected. The training samples are then divided into batches using an iterator.

For each batch of training samples, the TransferLearningHelper class extracts bottleneck features and corresponding labels. These are fed as inputs to the model, and the model is trained using the specified training signature from the .tflite model file. The training process also calculates the loss for each training step, which is provided to the ClassifierListener for monitoring the training progress.

After training, the training samples are cleared by calling resetTrainingSamples() to make room for new samples. This is where a replayBuffer is of great use because we can save a portion of the samples to retain the previously learned knowledge in the next training cycles. We will discuss the replayBuffer in the next subsection.

Performing inference

The application uses the trained model to perform inference on new images. When an image is captured, it is preprocessed and converted into a TensorImage. The image is then passed through the pre-trained feature extractor to generate bottleneck features. These bottleneck features are used as input to the trained model, which outputs the probability of each class. The ClassifierListener receives the results along with the inference time, allowing the application to display the predictions and their corresponding confidence scores.

3.2.4 – On-Device Training and Continual Learning Implementation

In addition to the on-device training capabilities demonstrated in the updated demo app, we further extended the app to enable continual learning by implementing a replay buffer. The replay buffer is a crucial component for continual learning as it helps the model to learn from past experiences while adapting to new data. Here is an overview of the replay buffer implementation and some additional details:

Integration of replay buffer

The replay buffer was integrated into the existing on-device training workflow. It stores a fixed amount of recent training data, which is reused during the training process.

Data management

The replay buffer efficiently manages the stored data by replacing the oldest data with new incoming data when the buffer reaches its maximum capacity. The training samples are replaced randomly as it has shown to achieve higher accuracy than a FIFO (First In First Out) replacement method [2].

Continual learning process

During the on-device training process, a certain percentage of the training data comes from the replay buffer, and the rest is new training samples gathered for the first time. This allows the model to maintain the knowledge it has already acquired while learning from new data, effectively enabling continual learning.

Parameter tuning

The ratio of replay buffer data to new data used during training, the replay buffer's maximum capacity, and other relevant parameters can be fine-tuned to optimize the model's performance and adaptability. Currently, the maximum size of the replay buffer is 100 training samples. In each training cycle we add 25% of the training samples into the replay buffer and replace existing samples randomly if the buffer is full.

Conclusion

By extending the demo app with a replay buffer implementation, we have successfully extended the new TensorFlow-Lite demo app that is used for on-device training to enable continual learning capabilities.

This allows the model to continually improve its performance over time as it encounters more examples from the user's environment. Most importantly, it now enables the model to inference correctly all classes in a class incremental scenario.

3.2.5 – Showcasing Different Scenarios

Figure 3.6 – Snapshot of the continual learning application that performs on-device training

In this section, we present three different scenarios to validate the effectiveness of the continual learning (CL) model in comparison with the traditional learning (TL) model, using the newly extended TensorFlow Lite demo app with a replay buffer. We examined three scenarios that demonstrate the capabilities of the CL model in handling various learning conditions. These scenarios include Cumulative Scenario - Batch Training, New Instances Scenario - Batch Training, and New Classes Scenario - Incremental Training. Our aim is to prove that the CL model can mitigate catastrophic forgetting under different circumstances where the default TensorFlow Lite demo app with the TL model fails to do so.

For the following 3 scenarios, our Class 1 consists of Pen samples, Class 2 consists of Calculator samples, Class 3 consists of Stapler samples and lastly Class 4 consists of Small Plant samples.

Figure 3.7 – The 4 samples used for the scenarios

3.2.5.1 – Cumulative Scenario – Batch Training

For the Cumulative Scenario, we collect 50 samples from each of the four available classes, ensuring that the position and rotation of the objects vary within a limited area of capture. We then trained both the TL and CL models using all 200 samples, with zero samples replayed for the CL model. After training, we evaluate the performance of both models to understand their ability to classify the objects and compare their overall accuracies.

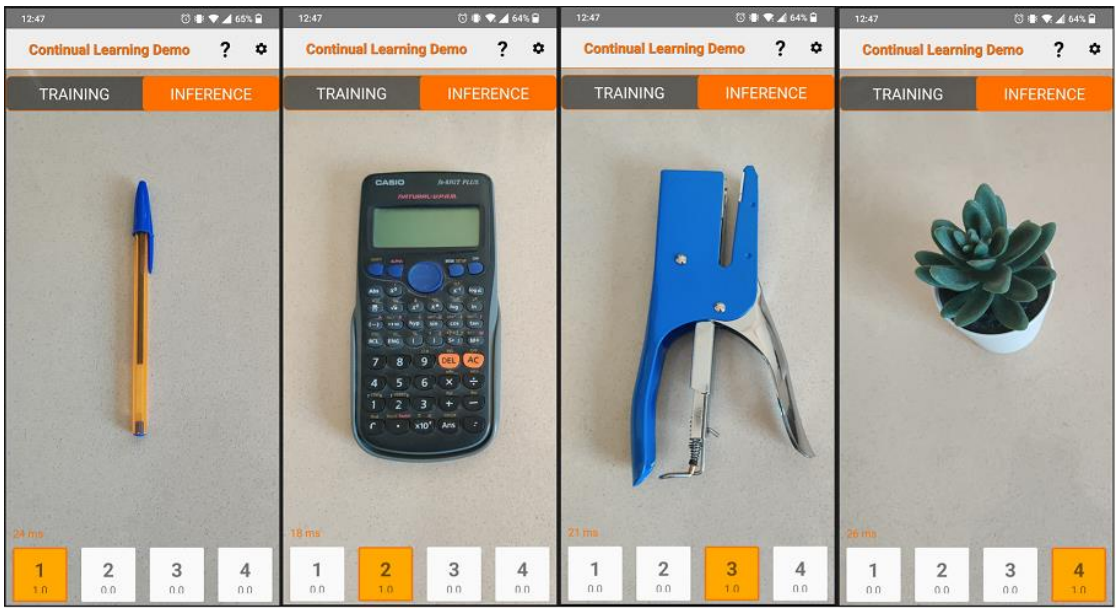


Figure 3.8 – Inference of TL model for the Cumulative Scenario. The inference is correct for all the 4 different classes

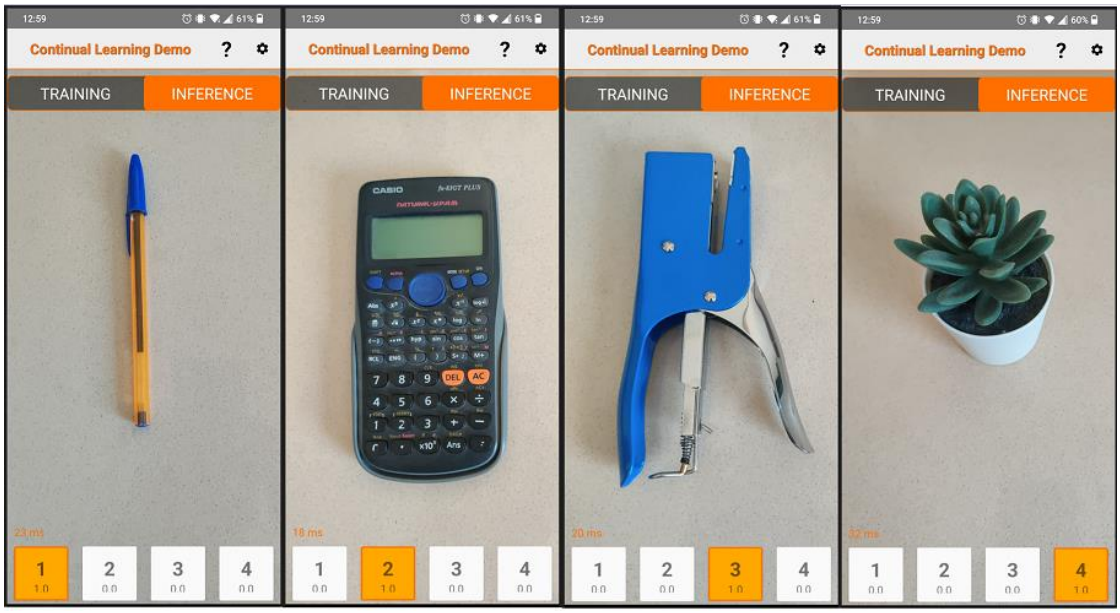


Figure 3.9 – Inference of CL model for the Cumulative Scenario. The inference is correct for all the 4 different classes

In the Cumulative Scenario, we can observe that both the CL model with the replay buffer as well as the original TL model of the TensorFlow-Lite demo app perform correct inference on all 4 classes.

3.2.5.2 – New Instances Scenario – Batch Training

In the New Instances Scenario, we follow a similar procedure as the Cumulative Scenario by first training both models on 200 samples from the First Instance row. The replay buffer is filled with 40 random samples (20%) from the first training cycle. Subsequently, we show new instances of the same four classes, and add 20 new samples per class. We then train both models with these 80 samples as well as replay the 40 samples from the buffer to the CL model. Finally, we compare the TL and CL models on how well they are able to remember the first instances of each class and analyse their performance under these conditions.

Figure 3.10 – The 4 new samples used for the New Instances Scenario

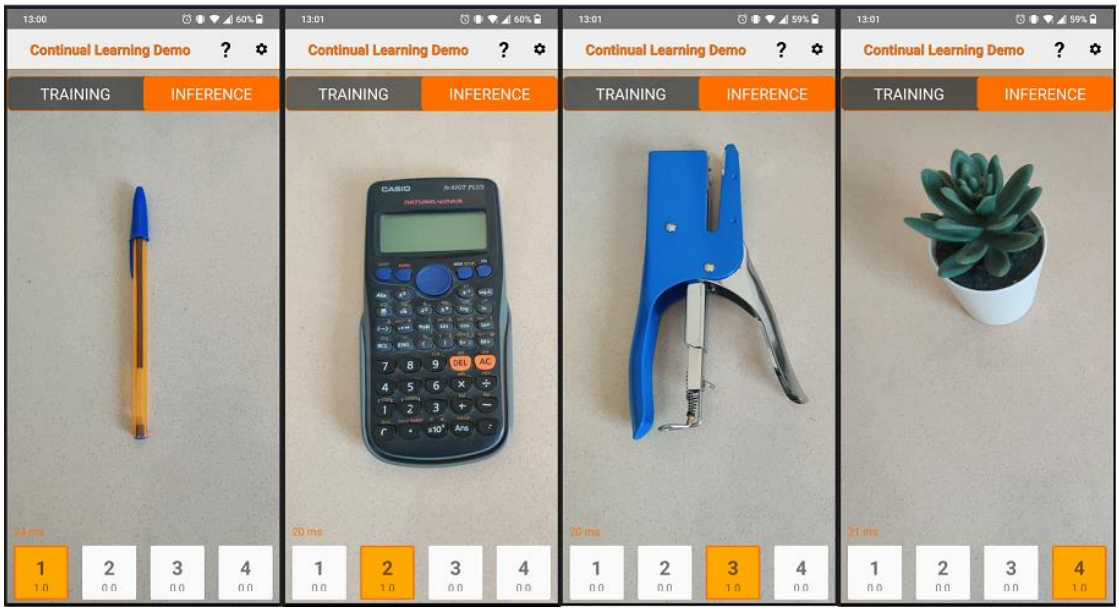


Figure 3.11 – Inference of TL model for the New Instances Scenario. The inference is correct for all the 4 different classes

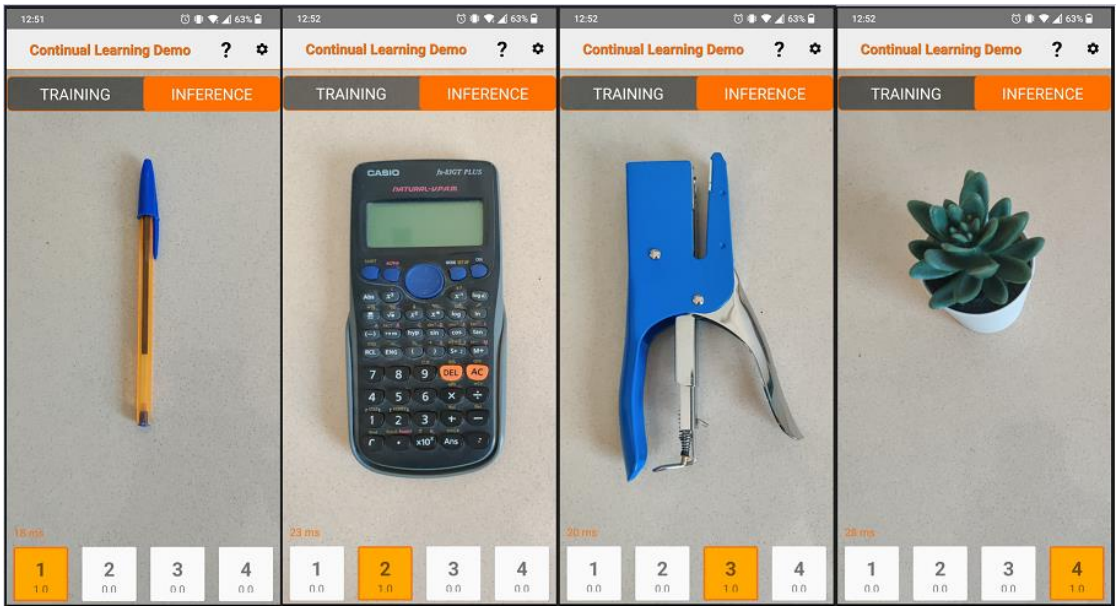


Figure 3.12 – Inference of CL model for the New Instances Scenario. The inference is correct for all the 4 different classes

In the New Instances Scenario, we can again observe that both the CL model as well as the original TL model perform correct inference on all 4 classes. However, it is worth mentioning that there would be rare occurrences of forgetting. The 3rd class (Stapler) would rarely indicate a wrong inference classifying the Stapler in the 4th class as we can see in the figures below. In the end, the inference would stay correctly on the 3rd class.

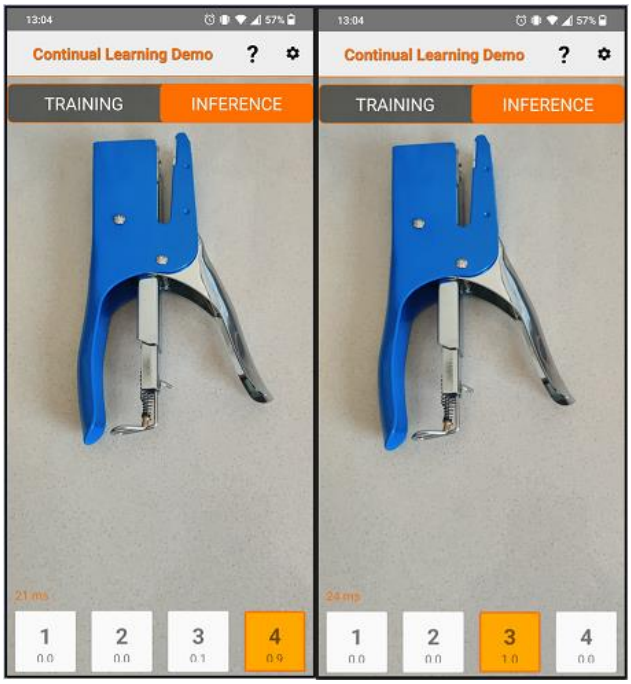


Figure 3.13 – Occasionally the 3rd class would be wrongly inferenced as the 4th class

3.2.5.3 – New Classes Scenario – Batch Training

The New Classes Scenario is designed to simulate a more natural setting for the deployment of the models. In this scenario, we incrementally introduce new classes. We start by adding 50 samples for each class and train both the TL and CL model We then move on to the next class. As each new class is introduced, the old samples that are in the buffer are replayed to the CL model. 10 random samples (20%) from the new class are added to the replay buffer. This process is repeated until both models have been trained on all four classes. Finally, we evaluate the TL and CL models by their ability to classify objects successfully in incremental learning settings.

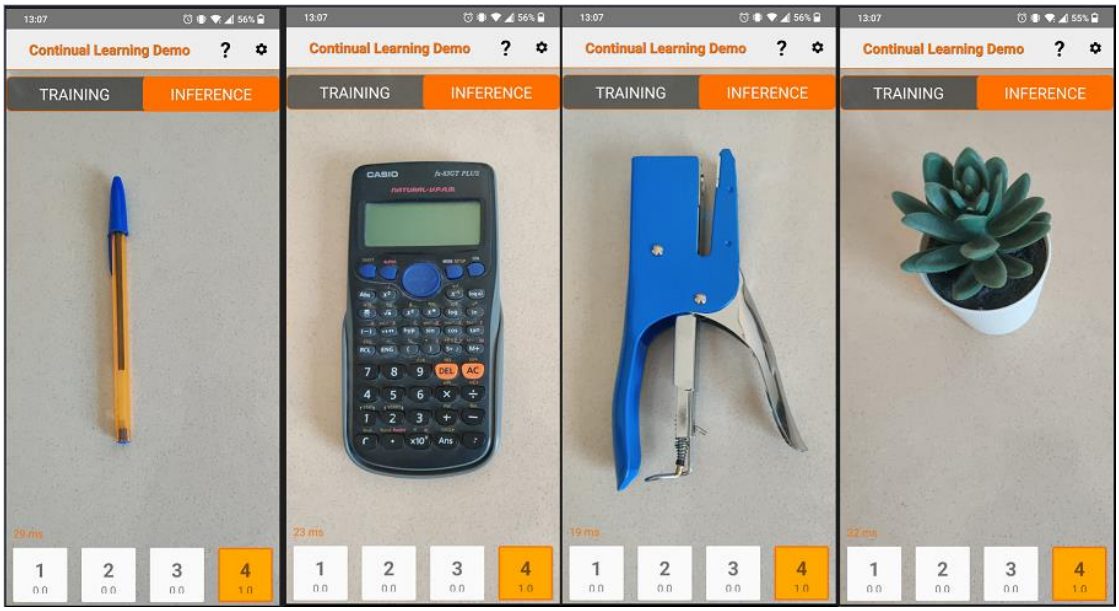


Figure 3.14 – Inference of TL model for the New Instances Scenario. The inference is only correct on the last added class while we have wrong classifications for all the other classes.

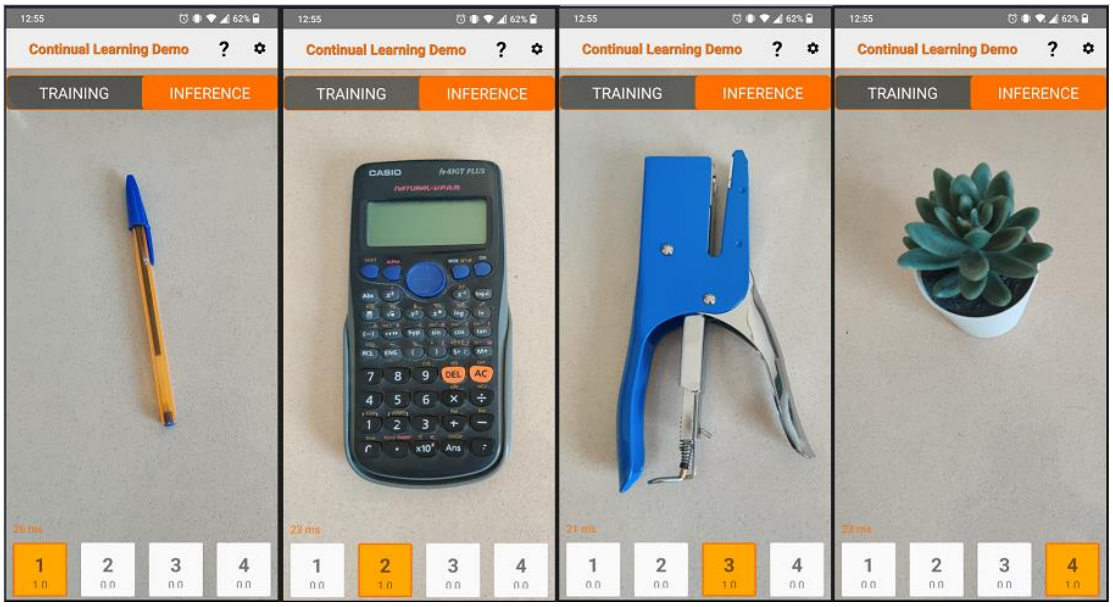


Figure 3.15 – Inference of CL model for the New Instances Scenario. The inference is correct on all 4 classes.

In the New Classes Scenario, we notice a significant difference in performance between the original TL model of the TensorFlow Lite demo app and the CL model. The TL model exhibits a considerable degree of forgetting, struggling to accurately infer the first three classes it was trained in. On the other hand, the CL model with the use of the replay buffer, it can retain knowledge of all four classes, successfully mitigating catastrophic forgetting. This outcome highlights the superiority of the CL model in handling incremental learning situations, where new classes are introduced sequentially.

Chapter 4

4 – Experiments

4.1 – Introduction to Experiments

In this chapter, we outline our experimental setup. The goal of the experiments is to build a lightweight model which will be able to perform well in real life object detection scenarios. If successful, this shows that the model can be used in applications like our TensorFlow-Lite Continual Learning demo app. The previous study [2] conducted offline experiments by maintaining the same model architecture used in the older demo app and altered the replay buffer sizes to examine their influence on the model's accuracy when tested against a benchmark dataset that introduces both new instances and new classes throughout the evaluation.

In the following sections, we offer a comprehensive description of the experimental setup, parameters, and the benchmark employed. We also discuss the objectives and outcomes of the previous offline experiments. Finally, we introduce our newly proposed experiments, which involve modifying the model architecture based on [15] as well as altering the way we add our samples to the replay buffer with the goal to enhance the accuracy of our benchmark results while keeping the size of the buffer relatively small.

4.2 – Experiment Setup

In this section, we provide a detailed description of the experiment setup. The setup is organized into three main components: the controller class, the experiment class, and the model class. The purpose of each component is to streamline the process of building, training, and evaluating the models using the CORe50 NICv2 - 391 benchmark. The replay buffer sizes are varied during the experiments to investigate their impact on the model's accuracy.

4.2.1 – Controller Class

The controller class serves as the conductor of the experiments. It is responsible for defining the experiments to be run, setting up the parameters for each experiment, and managing the interactions between the experiment functions and the model class. This class provides a high-level interface to configure, execute, and plot the experiments, ensuring that the entire process is efficient and easy to manage.

4.2.2 – Experiment Class

The experiment class contains a collection of functions that handle various aspects of the experimentation process, such as loading the CORe50 dataset, building the model, training the model, and calling the replay buffer function. These functions work together to carry out the following tasks:

Load the CoRE50 dataset

The process of loading the CORe50 dataset involves several steps, including instantiation of the dataset using the CORe50 class with the NICv2_391 scenario. The test set is acquired with the help of a method from the data loader helper class of the CORe50 dataset. The images are preprocessed with the help of a utility function which scales the pixel values to the range [0,1]. After these steps, the CORe50 dataset is suitable for training and evaluation.

It is worth mentioning that the NICv2_391 scenario is a specific configuration of the CORe50 dataset designed for evaluating incremental learning algorithms in a New Instances and Classes (NIC) setting. The dataset is organized into 391 incremental batches, with each batch containing both new instances of previously seen classes and entirely new classes. The first batch contains around 3000 samples of different classes of objects while all the other batches contain around 300 samples each. New classes are introduced incrementally throughout the training process. Around the last 40 batches, no more new classes are introduced but rather older classes are shown to the model again. The test set evaluates the model on different instances of all 50 classes. The objective is to evaluate how well a model can learn to recognize and classify new instances and classes as they are introduced incrementally [11].

Build the model

We initialize the model class, set up the base, head, and complete model using the provided parameters. Then, we compile the model for training, specifying the optimizer, loss function, and evaluation metrics.

Training

The model is trained on the CORe50 dataset using the specified training parameters and updates the weights of the model's head accordingly. We use Stochastic Gradient Descent (SGD) as the optimizer and train the head for all the training incremental batches.

Replay Buffer

The samples stored in the replay buffer are fitted in the model's head along with the new samples. Two different methods are used to store the new samples into the replay buffer which will be discussed in the model class. If the replay buffer is full, we randomly replace old samples with new ones. This ensures that the replay memory is updated and used effectively during the training process, which includes shuffling the training data, storing representations, and evaluating the model on the test set.

4.2.3 – Model Class

The model class contains the functions necessary to build the base, head, and complete model, as well as the functions for managing the replay memory. The main functions in the model class include:

BuildBaseHidden

This function constructs the base model using the MobileNetV2 architecture with pre-trained weights from the ImageNet dataset. The base model is designed to extract features from the input images. The base model is truncated at a specified layer given as a parameter, and the remaining layers are transferred to the head model. All of the base models' layers are frozen, so the weights remain fixed and are not updated. This configuration aligns with the context of our study.

In a real-world scenario, our model would be deployed on a resource-constrained device, such as a smartphone. These devices often have limited computational capabilities and cannot afford to retrain the entire network every time new data comes in. This constraint is particularly relevant for continual learning scenarios, where we want the model to learn from new experiences without forgetting the old ones.

Freezing the MobileNetV2 layers simulates this situation because it mirrors the operation of a smartphone that leverages pre-trained models without updating all their parameters. This way, we can focus on training only the head of our model, making the model more lightweight and feasible for deployment on edge devices.

By adopting this approach, we ensure that our experimental setup realistically simulates the conditions under which the model will operate in a real-world context, enabling us to derive results that are directly applicable to practical implementation.

BuildHeadHidden

The BuildHeadHidden function creates the head model by incorporating a number of hidden layers from the MobileNetV2 architecture which are given as a parameter. These layers are added to the head model sequentially and are un-frozen so they can be trained. The model is then compiled with a stochastic gradient descent optimizer and a sparse categorical cross entropy loss function. This approach enables the exploration of different head model complexities and their impact on the overall model performance.

BuildCompleteModel

The BuildCompleteModel function connects the base and head models to form the complete model for training and evaluation. It first defines an input layer with the appropriate shape for the images in the dataset. This input layer is then passed through the base model, which is responsible for feature extraction. The output of the base model is fed into the head model, which handles classification. Finally, the complete model is constructed by defining the inputs and outputs of the model and compiling it. This results in a unified model ready for training and evaluation on the CORe50 dataset.

StoreRepresentations and StoreRepresentationsNativeRehearsal

The StoreRepresentations function is responsible for managing the replay memory by storing feature representations obtained from the feature extractor. It ensures that the replay memory stays within its defined capacity by randomly removing the oldest samples, when necessary, with a 1.5% sample replacement rate. Once the feature representations and corresponding labels are added to the replay memory, the function provides an update on the current size of the replay memory for both feature representations and labels.

The StoreRepresentationsNativeRehearsal follows the same approach with removing samples. However, we use a different algorithm for adding samples to our replay buffer as proposed in [15]. For every training batch, we save x patterns to our replay buffer which are calculated as our replay buffer size divided with the number of the current training batch. If the replacement batch size is larger than the current training batch samples, we just save all of the current training batch samples. This has as a result the number of patterns added to progressively decrease. This ensures an overall balanced contribution from the different training batches, although no specific measures are implemented to achieve equal representation across classes.

Replay

The Replay function is designed to update the head model using the samples stored in the replay memory. By calling this function, the head model is trained with the feature representations and corresponding labels from the replay memory, effectively incorporating the knowledge from past training samples into the current model. This process is performed **to** enhanc**e** the model's ability to generalize across different data distributions. This function will end up being redundant since further in our experiments we will incorporate mixing our replay samples with the new samples, shuffling them, and fitting them to the head once rather in 2 different steps.

4.3 – Previous Experiments

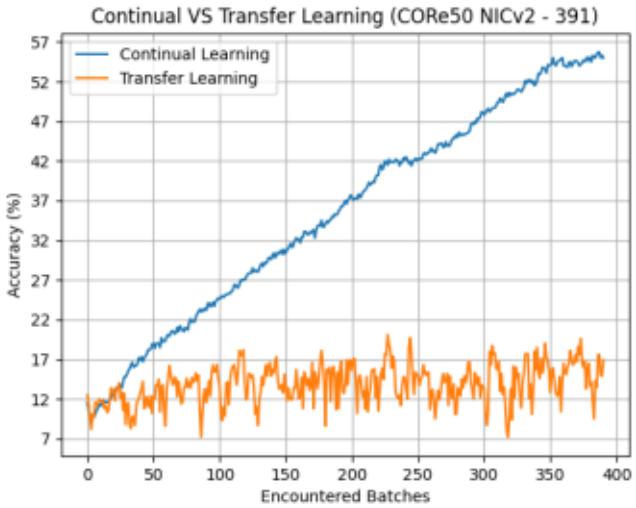
The previous paper conducted experiments to evaluate the original model's performance under various conditions. These experiments can be broadly categorized into three main areas: comparison of transfer learning and continual learning, analysis of different replay buffer replacement strategies, and evaluation of different replay buffer sizes.

Transfer Learning vs Continual Learning

The first experiment compared TensorFlow’s transfer learning model with the continual learning model which has a replay buffer to demonstrate the benefits of continual learning in terms of model adaptability and accuracy. The study highlighted that continual learning models were more capable of adapting to new data distributions and maintaining their performance, whereas transfer learning models tended to struggle with these challenges. Both models were exactly the same with a MobileNetV2 for the body

and a dense layer with ReLU activations along with a softmax layer for the head. The only difference was the replay buffer that was used in the continual learning model.

Figure 4.1 – A comparison of the accuracy of Continual Learning model with the replay buffer



(7500 pattern storage) vs the Transfer Learning model on the CORe50 NICv2 – 391 benchmark (Source: https://arxiv.org/abs/2105.01946.pdf)

Replay Buffer Replacement Strategies – FIFO vs Random

The second experiment analyzed the impact of different replay buffer replacement strategies, namely First-In-First-Out (FIFO) and random replacement. The study aimed to determine which strategy was more effective in maintaining the model's accuracy while adapting to new data distributions. The results indicated that the random replacement strategy provided better performance than the FIFO method, as it allowed the model to retain a more diverse set of samples from the past resulting in a higher accuracy.

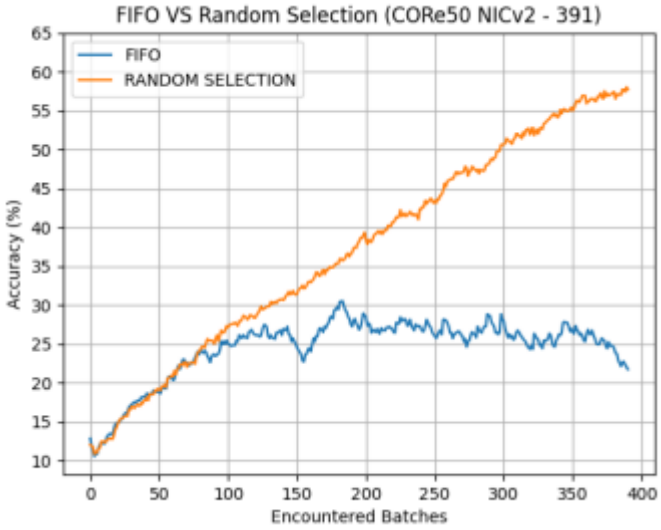


Figure 4.2 – A comparison of the accuracy the Continual Learning model using a FIFO replacement strategy on the replay buffer vs using a random replacement strategy on the CORe50 NICv2 – 391 benchmark

(Source: https://arxiv.org/abs/2105.01946.pdf)

Impact of Different Replay Buffer Sizes

The third experiment investigated the effects of varying the replay buffer size on the model's performance. The study considered several buffer sizes, ranging from 3,000 to 30,000, to understand how the size of the buffer influenced the model's adaptability and accuracy. The results demonstrated that larger buffer sizes led to better performance, as they enabled the model to store more samples from previous tasks and enhance its generalization capabilities. However, it is crucial to balance the buffer size with computational resources and memory constraints. The best balance of accuracy loss and least storage usage had the replay buffer of 7500 feature patterns. It requires 75% less storage compared to the largest buffer size with a minimal accuracy loss of 3.5%.

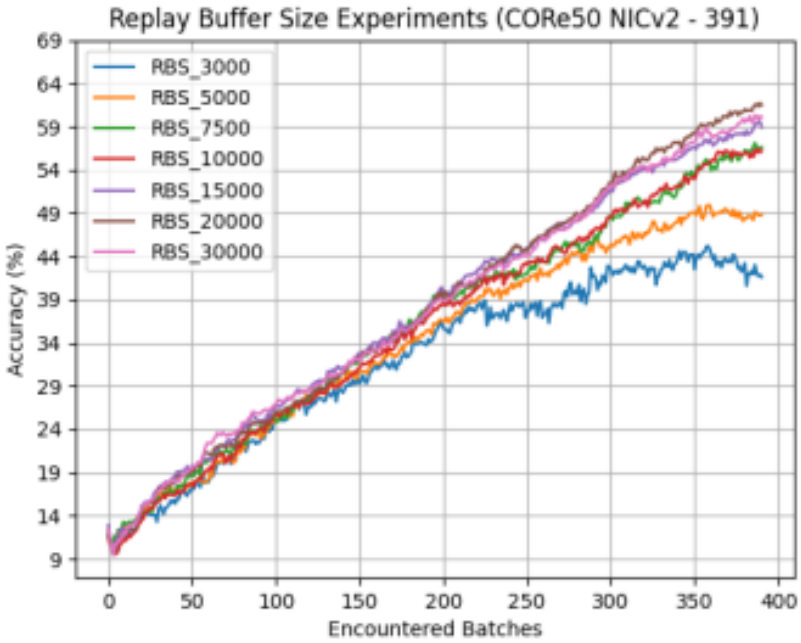


Figure 4.3 – A comparison of the accuracy the Continual Learning model using different replay buffer sizes on the CORe50 NICv2 – 391 benchmark
(Source: https://arxiv.org/abs/2105.01946.pdf)

4.4 – Proposed Experiments

For our experiments, we propose a couple of changes to the way the model head is trained, the way we add a portion of the new samples to our replay buffer and a change in the models' architecture. Lastly, we run the model with all the new changes proposed with various replay buffer sizes. It is worth mentioning that we will focus on maintaining a relatively small replay buffer size since our goal is to prove the effectiveness of a model that can be later added to an embedded device such as a smartphone application.

We propose the following three changes as we can see below.

4.4.1 – Sequential vs. Simultaneous Training: New Samples and Replay Buffer Samples

In this experiment, we aim to compare two different training approaches for the head of our model. The first approach, and existing one, involves training the head on the new samples of our training batch for 3 epochs followed by training on the replay buffer

samples for just one epoch. The second approach involves concatenating the new samples with the replay buffer samples and shuffling them before fitting them to the head for training of 4 epochs.

4.4.2 – Storing New Samples Representations in the Replay Buffer

In the experiment setup subsection, we have already discussed two different approaches for incorporating new samples into the replay buffer. The first approach, which is the existing method, involves adding a fixed percentage of samples (specifically 1.5% of the replay buffer size) from the current training batch to the replay buffer. Once the buffer reaches its capacity, random replacement is performed to accommodate the new samples.

The second and new approach, proposed in [15], utilizes a different algorithm for adding samples to the replay buffer. For each training batch, a certain number of patterns are saved to the replay buffer. This value is calculated by dividing the replay buffer size by the number of the current training batch. If the replacement batch size exceeds the number of current training batch samples, all of them are added to the replay buffer. As a result, the number of patterns added progressively decreases over time.

The objective of this new approach is to ensure a balanced contribution from different training batches with the goal of increasing accuracy. Measures to achieve equal representation across classes are not implemented.

4.4.3 – Latent Replay

We propose moving the last few hidden layers from the body (MobileNetV2) to the head of the model and making them trainable. This approach was inspired by the AR1*LR paper [15], which demonstrated improved accuracy when applied to MobileNetV1. The experiments will be conducted by testing different numbers of moved hidden layers to investigate the potential benefits of this modification. As a result of moving the hidden layers, the replay buffer replayed the samples a few layers earlier, potentially enabling better knowledge retention.

To comprehensively evaluate the impact of this modification, we will implement this change on top of the best performing changes of the other two experiments. Furthermore, we will test different learning rates to see which one performs best. This will provide a deeper understanding of how the combination of moved hidden layers and varying learning rates influences the model's performance, adaptability, and knowledge retention. By analyzing the results, we aim to identify a relatively optimal configuration for the hidden layer movement, which can maximize the model's effectiveness while considering computational and memory constraints by maintaining a small replay buffer size.

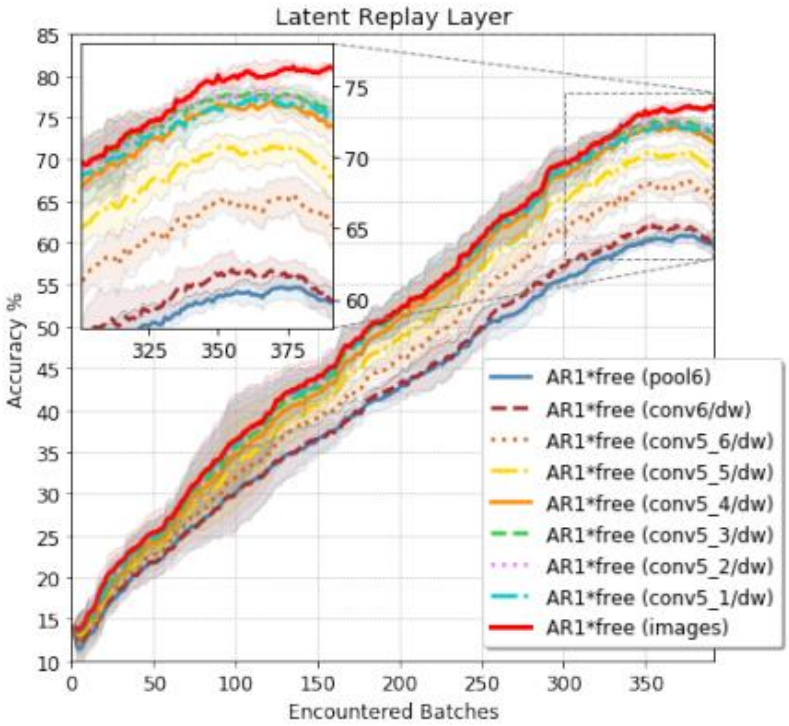


Figure 4.4 – AR1 with latent replay on different hidden layers of the MobileNetV1 showing an increase in accuracy when moving hidden layers of the MobileNetv1 to the head of the model thus making them trainable. This means that the replay occurs a few layers earlier, at the end of the body of the model.

(Source: https://arxiv.org/abs/1912.01100.pdf)

4.4.4 – Different Replay Buffer Sizes

In this last experiment, we explored the impact of varying replay buffer sizes on the performance of our continual learning model after all the proposed changes which had a positive impact in our model's prediction capabilities. The goal here is to provide a showcase of our models' accuracy across different replay buffer sizes in comparison to the old models' experiment results.

Chapter 5

5 – Experimental Results and Discussion

5.1 – Sequential vs. Simultaneous Training: New Samples and Replay Buffer Samples

The previous approach to model training involves a two-step sequential process. This process commences with training the model's head on fresh samples of the current training batch for three epochs. Subsequently, the model's head is trained on the replay buffer samples, over a period of one epoch. Hence, the model's head is fitted twice, with each instance employing different sample sets.

However, we have now adopted a revised methodology where we perform simultaneous training. In this approach, the samples from the new training batch and those contained in the replay buffer are merged and shuffled together. This union creates a comprehensive and diverse dataset that includes both the most recent data as well as the historical data from the replay buffer. Following this, we conduct training on this unified dataset over four epochs, in a single, uninterrupted training process.

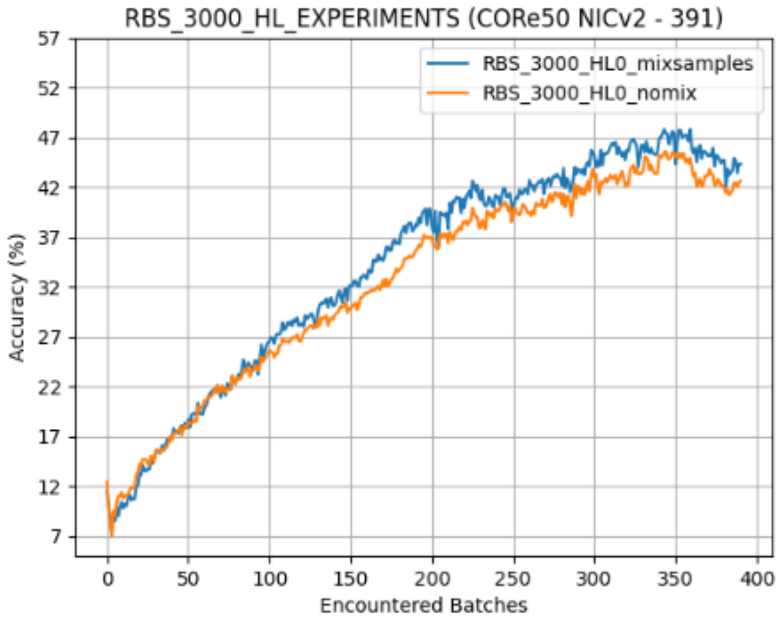


Figure 5.1 – Comparison of training the head simultaneously on a mixture of both replay representations and new samples VS training it sequentially first on the new samples and then on the replay representations.

The graph provided above represents the comparison between the previous sequential training method and our newly implemented simultaneous training approach.

The peak accuracies observed in the graph demonstrate a noticeable difference between the two methodologies. The conventional sequential training process yields a peak accuracy of 45.5%, which, although respectable, is outperformed by the simultaneous

training method. The revised simultaneous training approach exhibits a peak accuracy of 47.8%, indicating an improvement in model performance.

We can also observe that from around the 100th batch onwards, the simultaneous training method consistently maintains a higher level of accuracy compared to the sequential approach, and this trend continues throughout all subsequent training batches until the end of the training process.

This indicates that the simultaneous training method not only increases the peak accuracy but also improves the model's performance stability over time.

5.2 – Storing New Samples Representations in the Replay Buffer

In Section 5.2, we explore the algorithm implemented for adding new samples to our replay buffer. This process is important as it helps ensure a balanced and diverse mix of training samples for our model.

Current algorithm for replay buffer samples

In the current model, the replace algorithm simply replaces a fixed percentage of samples in the replay buffer randomly once the buffer is full. Specifically on the experiments we ran, it would replace 1.5% of the replay buffer size with new samples from the current training batch.

However, this deteriorated the model's performance towards the last 200 training batches. This could be due to starting to lose knowledge in previous classes.

New algorithm for replay buffer samples

As a result, inspired by the algorithm used in [15], we introduced the following algorithm which adds the replay samples in a different manner to mitigate the loss.

Firstly, the algorithm determines the replacement batch size, which will progressively decrease based on dividing the replay buffer size with the current batch number to ensure a balanced contribution from different training batches. This is a crucial part of the algorithm as it helps maintain the variety of the samples in the replay buffer.

If the replacement batch size is larger than the number of samples in a batch, the algorithm adjusts the replacement batch size to match the batch size. This situation may occur during the initial training batches or depending on the size of the replay buffer.

Next, if the replay buffer is at risk of exceeding its capacity due to the addition of the new batch (i.e., the combined size of existing replay buffer and the new batch is greater than the replay buffer limit), the algorithm takes corrective action. It first selects a random number of samples equal to the replacement batch size from the new batch. The selected samples then go through the feature extraction process. Simultaneously, the algorithm randomly identifies old samples for deletion from the replay buffer to make room for the new ones.

If there is no risk of overflow, the algorithm simply selects samples from the new batch, equal to the replacement batch size, and processes them through the feature extraction step.

Finally, regardless of the scenario, the processed samples are added to the replay buffer, replacing the old samples if necessary.

This algorithm was designed keeping in mind the need for continual learning and ensuring a diverse and balanced replay buffer for model training. Following this, we present a graph highlighting the results achieved using this algorithm.

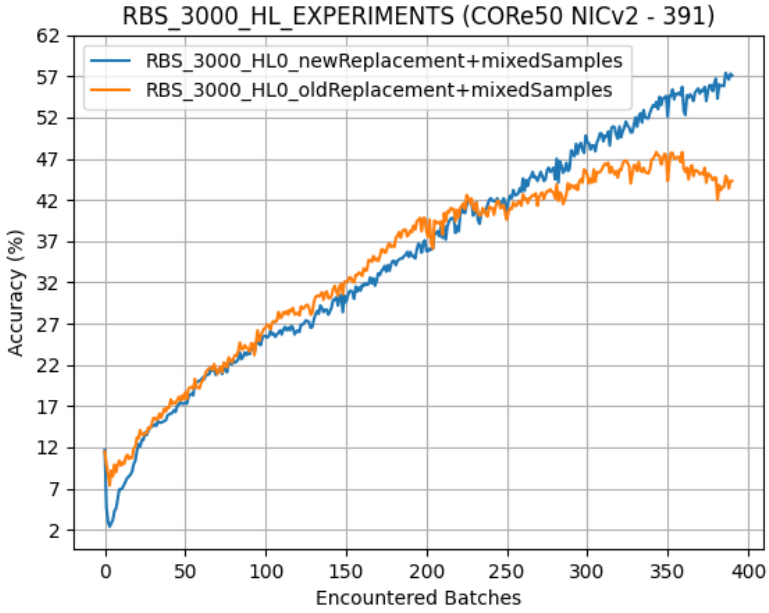


Figure 5.2 – Comparison of the two different algorithms for adding new samples to the replay buffer. The mixedSamples approach was used as well.

"RBS_3000_HL0_newReplacement+mixedSamples" shows a steady increase in accuracy over time. Moreover, towards the last half of our training batches we can see that the model is able to retain both the knowledge of the old classes as well as the new ones since we have a constant accuracy increase which reaches a peak of 57.4%. This suggests that your new algorithm for adding samples to the replay buffer is effective in maintaining a diverse and balanced mix of training samples, thereby enhancing the model's performance as training progresses.

"RBS_3000_HL0_oldReplacement+mixedSamples" also shows a similar pattern of increasing accuracy over time, though there is a slightly higher accuracy around the middle of our training batches. However, the model seems unable to retain knowledge for all the classes and as a result shows a loss in performance towards the end of our training thus being able to reach a peak of only 47.8%.

However, a limitation of the new replacement method is that it might struggle to adequately represent new classes introduced very late in the training process if we have a much higher number of training batches. Since the sample replacement batch size decreases as the batch number increases, fewer samples from these new classes will be able to enter the replay buffer. This lack of diversity in the replay buffer could potentially lead to underperformance on these new classes, as the model might not have enough exposure to their examples.

One way to address this issue could be to periodically reset the sample replacement batch size. This could ensure more samples from later-introduced classes make it into the replay buffer.

Alternatively, another strategy could be to incorporate a prioritization mechanism in the replacement method. This would favor the selection of samples from less-represented classes, thus ensuring diversity in the replay buffer throughout the training process.

5.3 – Latent Replay

With the latent replay methodology, we attempt to boost the performance of our model. This technique involves the movement of layers from the body (in this case, MobileNetV2) to the head of the model, and making them trainable.

Traditionally, in transfer learning applications, the body of the model (pre-trained on a large and diverse dataset like ImageNet) is frozen during the training process to preserve the learned features. The head of the model, which in our case is a dense fully connected layer followed with a softmax layer is then trained to adapt to the specific task at hand.

However, in the latent replay methodology, selected latent layers from the body are made trainable and are moved to the head of the model. This approach would hopefully allow the model to learn more refined features from the data.

It is important to note that this shift in structure required us to adjust the learning rate. With the introduction of more trainable layers, it was necessary to increase the learning rate. The rationale for this is that since the body of our model is frozen, the head should be able to learn at a full pace.

We can see below on our first graph, that even with only the dense layer in our head and without any hidden layers moved, we noticed a significant increase in the accuracy of our model when compared to the learning rate that was used in [2]. With the new learning rate, a peak accuracy of 63.3% was reached while with the old learning rate and the model adjusted with our previous 2 changes, the model reached an accuracy of only 57.4%.

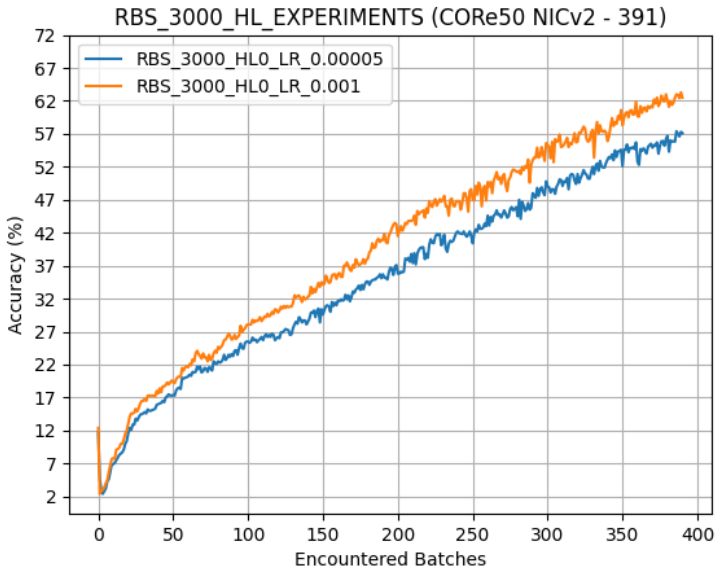


Figure 5.3 – Comparing a low learning rate vs a higher learning rate for the head of our model. For both experiments, no latent replay was used, the head only consisted of the dense layer and the softmax layer.

The learning rate value of 0.001 seems to perform well when compared with lower learning rate values. More experimentation could be conducted with higher learning rate values, however higher values appeared to result in the model oscillating in certain training batches as we can see from our experiment with a learning rate of 0.01. It is worth noting that the oscillation occurred mostly when we moved a certain number of hidden layers with the latent replay approach rather than when using only the dense layer in our head of the model. Different optimizers could also be used. For the experiments we used the SGD optimizer.

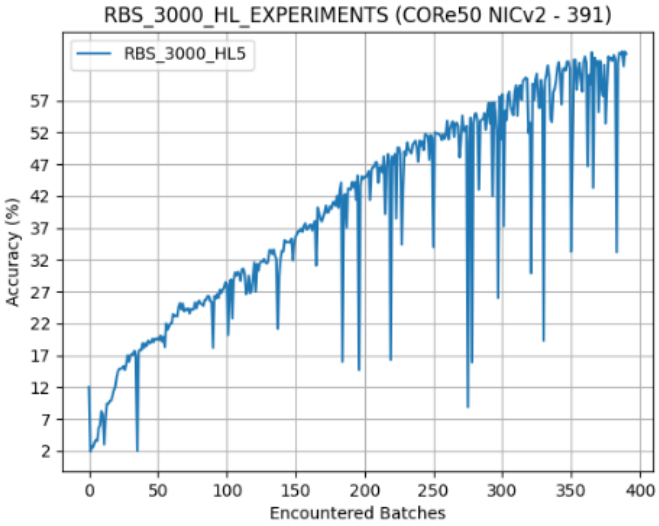


Figure 5.4 – A higher learning rate (0.01) along with latent replay of 5 hidden layers results in a huge oscillation of our model on certain training batches.

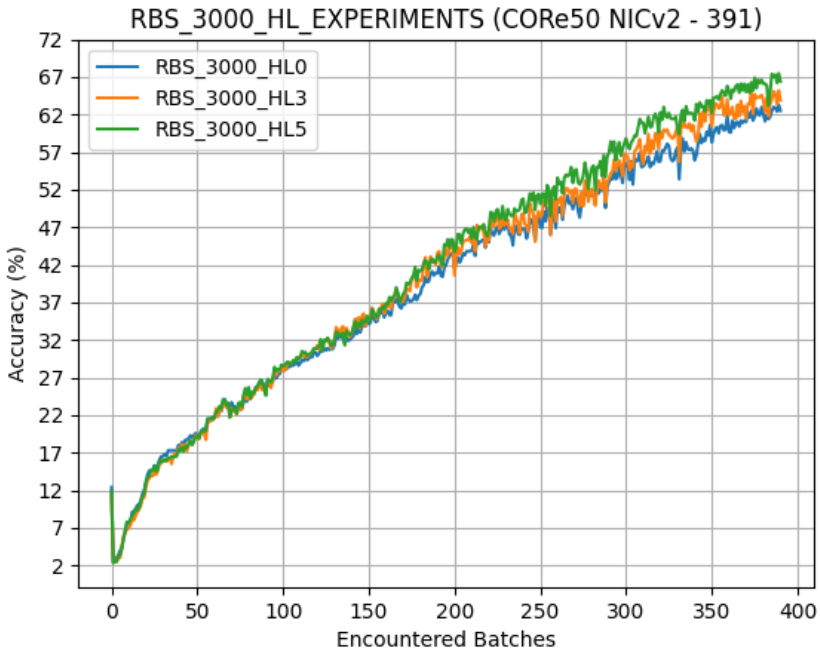


Figure 5.5 – Using the latent replay method we can see the graph for moving 0, 3 and 5 hidden layers to the head of our model

From the graph, the latent replay method significantly impacts the model's accuracy, and this impact is directly correlated with the number of hidden layers moved. We observe an upward trend in accuracy as more hidden layers are involved in the process.

Starting from zero hidden layers moved, the accuracy is at 63.3%. As we move to three hidden layers, we see a marked increase in accuracy, demonstrating the effectiveness of incorporating more complexity in the head of our model through these hidden layers. When we move up to five hidden layers, the trend continues, with the model reaching its peak accuracy of 67.3%. This suggests that the added depth on the head provided by more hidden layers can significantly enhance the power of the model.

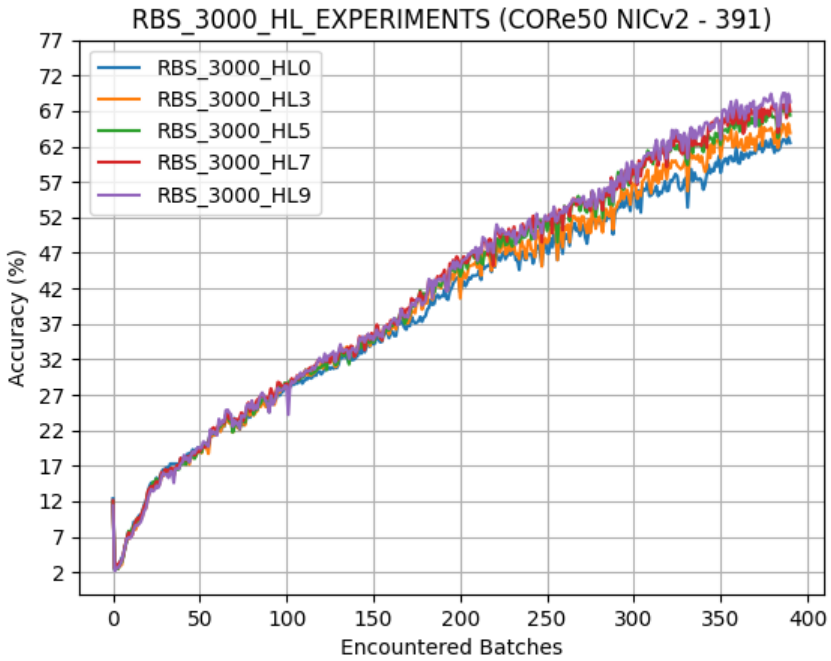


Figure 5.6 – Experimenting with more hidden layers moved we keep seeing increases in the accuracy

Continuing with moving 7 and 9 hidden layers from the body of our model to the head we can see a continued, albeit slight, improvement in accuracy with each increase. The peak accuracy reaches 69.5% when 9 layers are moved, showing that more complex models do, to an extent, continue to benefit from the added depth.

However, these benefits need to be balanced against potential drawbacks. As more layers are moved, the complexity of the model's head increases. This complexity can pose a challenge, particularly when it comes to on-device training. Devices often have computational and power constraints, making it important to optimize models for efficiency and accuracy.

5.4 – Different Replay Buffer Sizes

As we continue our exploration in section 5.4, we investigate the impact of different replay buffer sizes. Building on the previous modifications that have been proven to increase our model's accuracy, we aim to see if increasing the size of our replay buffer could potentially offer additional improvements. Specifically, we test a replay buffer size of 7500 compared to the replay buffer size we have used so far of 3000 samples with performing latent replay on the 5^{th} and 9^{th} hidden layer.

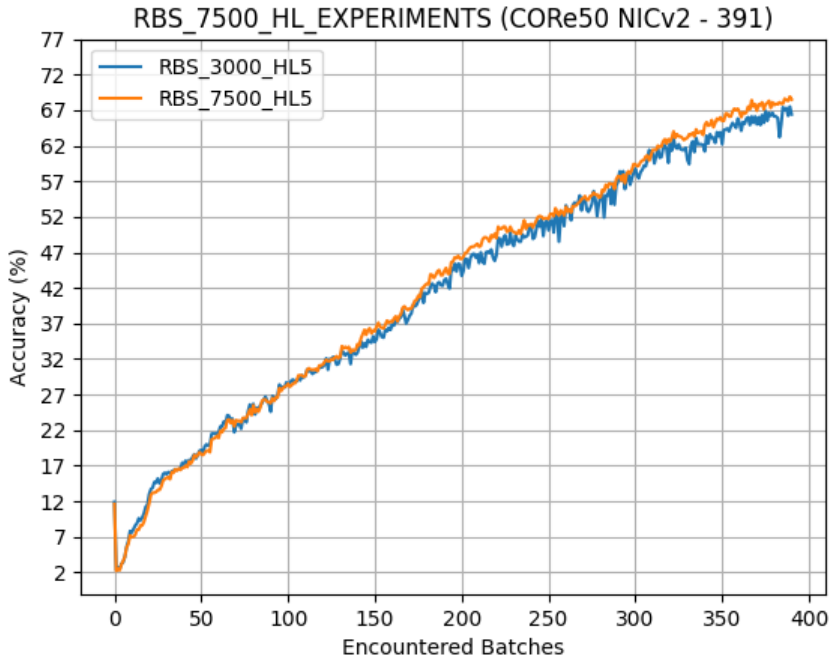


Figure 5.7 – RBS_3000 VS RBS_7500 with moving 5 hidden layers to the head

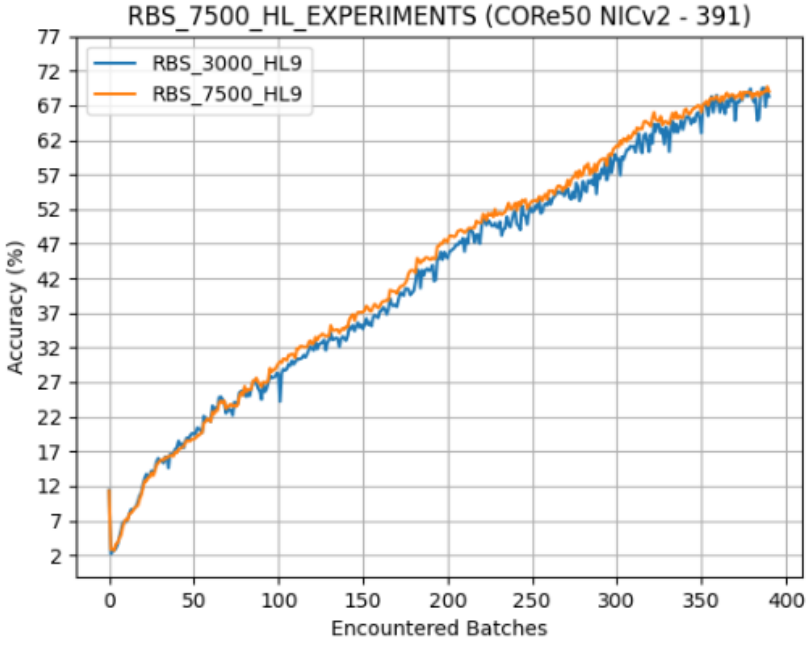


Figure 5.8 – RBS_3000 VS RBS_7500 with moving 9 hidden layers to the head

Upon examining the two graphs, the performance improvement when the buffer size is increased from 3000 to 7500 is minor with almost exact peak accuracy. This is consistent across both scenarios, namely when we move 5 and 9 hidden layers to the head of our model accordingly.

The higher replay buffer size manages to reach a peak accuracy of 69.5% on the first graph and 69.7% on the second graph.

An important observation is the enhanced stability across all training batches when employing the larger buffer size. Unlike with the smaller buffer size, where loss values demonstrated some fluctuations, the larger buffer size provided a more consistent performance. This resulted in a steadier and less variable loss trajectory, further improving the robustness of our training process.

However, it is crucial to put these findings in perspective. While a steadier model is indeed observed, we must remember that the buffer size has been more than doubled in these tests. We should question whether this justifies our increase in resource consumption with a larger buffer size, particularly in the context of our primary aim: enabling efficient on-device training.

Our emphasis should remain on achieving a balance between performance and efficiency, and as such, we should continue to prioritize smaller buffer sizes.

Chapter 6

6 – Conclusion

6.1 – Conclusion

6.2 – Future work

6.1 – Conclusion

This thesis was guided by the goal of integrating a continual learning model into embedded devices, particularly within the realm of smartphone applications. This type of learning model holds immense potential, not only for its adaptability in learning new concepts but also for its capacity to function within the constraints of these devices. Key to this functionality is the model's lightweight structure, efficiency, and optimized use of computational resources.

The objectives of this thesis were anchored in two fundamental aspects. Firstly, we aimed to transition the continual learning capabilities from the previous TensorFlow-Lite application to the updated version. We achieved this by incorporating a replay buffer, which proved instrumental in managing the storage and retrieval of learning patterns. Secondly, we aimed to further experiment offline with the model to enhance its performance while maintaining a lightweight structure suitable for resource-constrained devices.

In terms of practical implementation, we successfully integrated the original model and our replay buffer into the new TensorFlow-Lite application. This integration yielded several advantages, including increased customization potential, streamlined model management, and more efficient storage and updating of training weights. Notably, this approach facilitated a more flexible model structure and optimizer selection, providing a single model solution for both training and inference.

Our experimental phase allowed us to make significant changes in improving the accuracy of our model. We accomplished this by incorporating a mix of replay buffer samples with new samples in the training process, applying a new replay buffer replacement algorithm, and utilizing the latent replay method. As a result of these strategic adjustments, we were able to achieve a peak accuracy of 69.7%.

This improved performance represents a significant advancement from the previous work used in [2] which used the same model architecture with a frozen MobileNetV2 as the body and a trainable head. The previous work managed to achieve a peak accuracy of 56.6% with a replay buffer size (RBS) of 3000 and around 62% with an RBS 30000, a size 10 times bigger than the one we used for our own model. As regards to [15], our peak accuracy of 69.7% falls slightly lower than the latent replay strategy on AR1* with

the latent replay occurring on the conv5_4/dw hidden layer (the 9th hidden layer of MobileNetV1 starting from the end) which achieves an accuracy of 72.24%. However, the AR1* model is a hybrid model which incorporates additional continual learning strategies such as regularization besides rehearsal methods (random and latent replay). This makes the AR1* model much more complex than our model which only utilizes rehearsal continual learning strategies. Despite a slightly lower accuracy, we manage to maintain a simple and lightweight model architecture suitable for edge deployment.

In conclusion, the work presented in this thesis offers valuable insights for the deployment of continual learning models in resource-limited environments, specifically on embedded devices such as smartphones. Our research underscores the potential for continual learning models to address real-world applications effectively and efficiently. As we look ahead, the prospect of incorporating our improved model into future TensorFlow applications holds exciting potential for the advancement of on-device learning capabilities. This thesis thus represents a step forward in the continual learning field and a foundation for future research and development in this area.

6.2 – Future work

Despite the significant strides made in this thesis, there remains considerable potential for future work, particularly within both the application and offline experimental facets of our study.

Application Related:

A promising area of future work lies in expanding the number of classes within the application. Given the reasonable accuracy achieved on the CORe50 dataset under the NICv2 - 391 scenario, which covers 50 distinct classes, it suggests the potential for increasing our application classes well beyond the current count of four.

Additionally, the improved performance of our new model presents an opportunity to implement it within the application. Future studies could consider on-device experimentation to assess metrics beyond accuracy, potentially offering a richer understanding of model performance. Given the simplified processes facilitated by the new application, one could proceed with benchmarking on Android Studio.

The application's enhanced capabilities in saving and loading model weights open the door for future updates which could consider mechanisms to preserve the model state when closing and reopening the application, ensuring continuity in the learning process.

Offline Experiments Related:

One potential approach for future work involves further fine-tuning of model hyperparameters, such as the learning rate. Exploration of different optimizers, particularly those with adaptable learning rates, may help to further boost accuracy.

Another aspect worth investigating is the replay buffer replacement method. Developing an approach to replace samples in a manner that achieves class-balancing could potentially enhance the model's ability to handle diverse data, albeit with an expected increase in complexity. Furthermore, more experimentation should be conducted on the new replay buffer method where the further we are on our training batches the less new samples we replace, specifically in scenarios where new classes are introduced after a lot of training batches. A strategy can be created where a soft reset occurs after a while to increase the number of new samples that we replace.

Furthermore, research could focus on optimizing the size of the replay buffer. Studies could explore the trade-off between accuracy and buffer size to find an optimal balance, potentially following the methodology used in the latent replay paper. This could also aid in integrating the exact model used in offline experiments into the application.

Lastly, several alternative strategies could be investigated to further enhance the model's performance and adaptability. For instance, we could explore the utilization of coresets, which can provide a summary of the dataset while preserving the essential characteristics needed for learning. This approach could lead to an even more efficient use of storage and computational resources enabling us to reduce the needed rehearsal space.

Additionally, dataset distillation, a technique that creates a smaller but equally representative dataset, could be used to reduce the amount of data required for training, further optimizing the model for resource-limited devices. Such techniques could make our model even more efficient and applicable to a wider range of devices and use cases.

Another possibly promising direction is the application of continual learning to regression problems. This would be a significant step in broadening the applicability of continual learning models, as they are currently largely limited to classification tasks. Investigating this possibility could open up new realms of practical applications for these models.

The exploration of these avenues could lead to significant advances in the deployment of continual learning models on edge devices, further pushing the boundaries of what is achievable in this domain.

References

[1] A. Aich, "Elastic Weight Consolidation (EWC): Nuts and Bolts," 2021. DOI: https://doi.org/10.48550/arXiv.2105.04093.

[2] G. Demosthenous, V. Vassiliades, "Continual Learning on the Edge with TensorFlow Lite," 2021. DOI: https://doi.org/10.48550/arXiv.2105.01946.

[3] N. Díaz-Rodríguez, V. Lomonaco, D. Filliat, D. Maltoni, "Don’t forget, there is more than forgetting: new metrics for Continual Learning," 2018. DOI: https://doi.org/10.48550/arXiv.1810.13166.

[4] A. G. Howard, et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," 2017. DOI: https://doi.org/10.48550/arXiv.1704.04861.

[5] K. Javed, M. White, "Meta-Learning Representations for Continual Learning," 2019, In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett (eds), Advances in Neural Information Processing Systems (NeurIPS 2019), Article No.: 163, 1820-1830.

[6] P. Kairouz, et al., "Advances and Open Problems in Federated Learning," 2021. DOI: https://doi.org/10.48550/arXiv.1912.04977.

[7] K. Khetarpal, M. Riemer, I. Rish, D. Precup, "Towards Continual Reinforcement Learning: A Review and Perspectives," 2022. DOI: https://doi.org/10.48550/arXiv.2012.13490.

[8] G. Kim, Z. Ke, B. Liu, "A Multi-Head Model for Continual Learning via Out-of-Distribution Replay," 2022. DOI: https://doi.org/10.48550/arXiv.2208.09734.

[9] J. Kirkpatrick, et al., "Overcoming catastrophic forgetting in neural networks," 2017. Proc. Natl. Acad. Sci. 114, 13, 3521–3526. DOI: https://doi.org/10.1073/pnas.1611835114.

[10] Z. Li, D. Hoiem, "Learning without Forgetting," 2017. DOI: https://doi.org/10.48550/arXiv.1606.09282.

[11] V. Lomonaco, D. Maltoni, "CORe50: a New Dataset and Benchmark for Continuous Object Recognition," 2017. In Proceedings of the 1st Annual Conference on Robot Learning. PMLR, 17–26.

[12] D. Lopez-Paz, M. Ranzato, "Gradient Episodic Memory for Continual Learning," 2022. DOI: https://doi.org/10.48550/arXiv.1706.08840.

[13] D. Maltoni, V. Lomonaco, "Continuous Learning in Single-Incremental-Task Scenarios," 2019. DOI: https://doi.org/10.48550/arXiv.1806.08568.

[14] G. I. Parisi, et al., "Continual Lifelong Learning with Neural Networks: A Review," 2019. Neural Netw. 113, 54–71. DOI: https://doi.org/10.1016/j.neunet.2019.01.012.

[15] L. Pellegrini, G. Graffieti, V. Lomonaco, D. Maltoni, "Latent Replay for Real-Time Continual Learning," 2020. DOI: https://doi.org/10.48550/arXiv.1912.01100.

[16] A. Prabhu, P.H. Torr, P. K. Dokania, "GDumb: A Simple Approach that Questions Our Progress in Continual Learning," 2020. In: Vedaldi, A., Bischof, H., Brox, T., Frahm, JM. (eds), Proc. Eur. Conf. Comput. Vis (ECCV), Lecture Notes in Computer Science, vol 12347. Springer, Cham, 524-540.

[17] O. Russakovsky, et al., "ImageNet Large Scale Visual Recognition Challenge," 2015. DOI: https://doi.org/10.48550/arXiv.1409.0575.

[18] A. A. Rusu, et al., "Progressive Neural Networks," 2022. DOI: https://doi.org/10.48550/arXiv.1606.04671.

[19] Q. She, et al., "OpenLORIS-Object: A Robotic Vision Dataset and Benchmark for Lifelong Deep Learning," 2020. DOI: https://doi.org/10.48550/arXiv.1911.06487.

[20] M. Tan, R. Pang, Q. V. Le, "EfficientDet: Scalable and Efficient Object Detection," 2020. DOI: https://doi.org/10.48550/arXiv.1911.09070.

[21] G. M. van de Ven, A. S. Tolias, "Three scenarios for continual learning," 2019. DOI: https://doi.org/10.48550/arXiv.1904.07734.

[22] Z. Wang, et al., "SparCL: Sparse Continual Learning on the Edge," 2022. DOI: https://doi.org/10.48550/arXiv.2209.09476.

[23] S. Zhang, G. Shen, J. Huang, Z. Deng, "Self-Supervised Learning Aided Class-Incremental Lifelong Learning," 2020. DOI: https://doi.org/10.48550/arXiv.2006.05882.

[24] Stream51. Retrieved February 22, 2023, from https://tyler-hayes.github.io/stream51.

Appendix

1 - Application

1.1 – Android Studio Code

1.1.1 - CameraFragment

```
/*
 * Copyright 2022 The TensorFlow Authors. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *             http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.tensorflow.lite.examples.modelpersonalization.fragments

import android.annotation.SuppressLint
import android.content.res.Configuration
import android.graphics.Bitmap
import android.graphics.Canvas
import android.graphics.Color
import android.os.Bundle
import android.os.Handler
import android.os.Looper
import android.os.SystemClock
import android.util.Log
import android.view.LayoutInflater
import android.view.MotionEvent
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.content.res.AppCompatResources
import androidx.camera.core.*
import androidx.camera.lifecycle.ProcessCameraProvider
import androidx.core.content.ContextCompat
import androidx.fragment.app.Fragment
import androidx.fragment.app.activityViewModels
import androidx.navigation.Navigation
import org.tensorflow.lite.examples.modelpersonalization.MainViewModel
import org.tensorflow.lite.examples.modelpersonalization.R
import
org.tensorflow.lite.examples.modelpersonalization.TransferLearningHelp
er
import
org.tensorflow.lite.examples.modelpersonalization.TransferLearningHelp
er.Companion.CLASS_FOUR
```

```
import
org.tensorflow.lite.examples.modelpersonalization.TransferLearningHelp
er.Companion.CLASS_ONE
import
org.tensorflow.lite.examples.modelpersonalization.TransferLearningHelp
er.Companion.CLASS_THREE
import
org.tensorflow.lite.examples.modelpersonalization.TransferLearningHelp
er.Companion.CLASS_TWO
import
org.tensorflow.lite.examples.modelpersonalization.databinding.Fragment
CameraBinding
import org.tensorflow.lite.support.label.Category
import java.util.*
import java.util.concurrent.ConcurrentLinkedQueue
import java.util.concurrent.ExecutorService
import java.util.concurrent.Executors

class CameraFragment : Fragment(),
    TransferLearningHelper.ClassifierListener {

    companion object {
        private const val TAG = "Model Personalization"
        private const val LONG_PRESS_DURATION = 500
        private const val SAMPLE_COLLECTION_DELAY = 300
    }

    private var _fragmentCameraBinding: FragmentCameraBinding? = null
    private val fragmentCameraBinding
        get() = _fragmentCameraBinding!!

    private lateinit var transferLearningHelper:
TransferLearningHelper
    private lateinit var bitmapBuffer: Bitmap

    private val viewModel: MainViewModel by activityViewModels()
    private var preview: Preview? = null
    private var imageAnalyzer: ImageAnalysis? = null
    private var camera: Camera? = null
    private var cameraProvider: ProcessCameraProvider? = null
    private var previousClass: String? = null

    /** Blocking camera operations are performed using this executor
*/
    private lateinit var cameraExecutor: ExecutorService

    // Used for the first time to train with white images to solve the
class incremental issue
    private var _isFirstTime: Boolean = true

    // When the user presses the "add sample" button for some class,
    // that class will be added to this queue. It is later extracted
by
    // InferenceThread and processed.
    private val addSampleRequests = ConcurrentLinkedQueue<String>()

    private var sampleCollectionButtonPressedTime: Long = 0
    private var isCollectingSamples = false
    private val sampleCollectionHandler =
Handler(Looper.getMainLooper())
```

```
    private val onAddSampleTouchListener =
        View.OnTouchListener { view, motionEvent ->
            when (motionEvent.action) {
                MotionEvent.ACTION_DOWN -> {
                    isCollectingSamples = true
                    sampleCollectionButtonPressedTime =
                        SystemClock.uptimeMillis()
                    sampleCollectionHandler.post(object : Runnable {
                        override fun run() {
                            val timePressed =
                                SystemClock.uptimeMillis() -
sampleCollectionButtonPressedTime

view.findViewById<View>(view.id).performClick()
                            if (timePressed < LONG_PRESS_DURATION) {
                                sampleCollectionHandler.postDelayed(
                                    this,
                                    LONG_PRESS_DURATION.toLong()
                                )
                            } else if (isCollectingSamples) {
                                val className: String =

getClassNameFromResourceId(view.id)
                                addSampleRequests.add(className)
                                sampleCollectionHandler.postDelayed(
                                    this,
                                    SAMPLE_COLLECTION_DELAY.toLong()
                                )
                            }
                        }
                    })
                }
                MotionEvent.ACTION_UP -> {

sampleCollectionHandler.removeCallbacksAndMessages(null)
                    isCollectingSamples = false
                }
            }
            true
        }

    // Permission related, doesn't bother us
    override fun onResume() {
        super.onResume()

        if (!PermissionsFragment.hasPermissions(requireContext())) {
            Navigation.findNavController(
                requireActivity(),
                R.id.fragment_container

).navigate(CameraFragmentDirections.actionCameraToPermissions())
        }
    }

    override fun onDestroyView() {
        _fragmentCameraBinding = null
        super.onDestroyView()

        // Shut down our background executor
        cameraExecutor.shutdown()
```

```
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _fragmentCameraBinding =
            FragmentCameraBinding.inflate(inflater, container, false)

        return fragmentCameraBinding.root
    }

    // Initializes different UI components as well as listeners
    @SuppressLint("MissingPermission", "ClickableViewAccessibility")
    override fun onViewCreated(view: View, savedInstanceState:
Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        transferLearningHelper = TransferLearningHelper(
            context = requireContext(),
            classifierListener = this
        )

        cameraExecutor = Executors.newSingleThreadExecutor()

        viewModel.numThreads.observe(viewLifecycleOwner) {
            transferLearningHelper.numThreads = it
            transferLearningHelper.close()
            if (viewModel.getTrainingState() !=
MainViewModel.TrainingState.PREPARE) {
                // If the model is training, continue training with
old image
                // sets.

viewModel.setTrainingState(MainViewModel.TrainingState.TRAINING)
                transferLearningHelper.startTraining()
            }
        }

        // If training state button changes, update the UI.
        viewModel.trainingState.observe(viewLifecycleOwner) {
            updateTrainingButtonState()
        }

        // If capture mode changes, update the UI.
        viewModel.captureMode.observe(viewLifecycleOwner) {
isCaptureMode ->
            if (isCaptureMode) {
                viewModel.getNumberOfSample()?.let {
                    updateNumberOfSample(it)
                }
                // Unhighlight all class buttons
                highlightResult(null)
            }

            // Update the UI after switch to training mode.
            updateTrainingButtonState()
        }
```

```
        viewModel.numberOfSamples.observe(viewLifecycleOwner) {
            // Update the number of samples
            updateNumberOfSample(it)
            updateTrainingButtonState()
        }

        with(fragmentCameraBinding) {
            if (viewModel.getCaptureMode()!!) {
                btnTrainingMode.isChecked = true
            } else {
                btnInferenceMode.isChecked = true
            }
            llClassOne.setOnClickListener {
                addSampleRequests.add(CLASS_ONE)
            }
            llClassTwo.setOnClickListener {
                addSampleRequests.add(CLASS_TWO)
            }
            llClassThree.setOnClickListener {
                addSampleRequests.add(CLASS_THREE)
            }
            llClassFour.setOnClickListener {
                addSampleRequests.add(CLASS_FOUR)
            }
            llClassOne.setOnTouchListener(onAddSampleTouchListener)
            llClassTwo.setOnTouchListener(onAddSampleTouchListener)
            llClassThree.setOnTouchListener(onAddSampleTouchListener)
            llClassFour.setOnTouchListener(onAddSampleTouchListener)
            // listener for pause, resume and start of training
            btnPauseTrain.setOnClickListener {

viewModel.setTrainingState(MainViewModel.TrainingState.PAUSE)
                // We call the replay buffer update in this method
                transferLearningHelper.pauseTraining()
            }
            btnResumeTrain.setOnClickListener {

viewModel.setTrainingState(MainViewModel.TrainingState.TRAINING)
                transferLearningHelper.startTraining()
            }
            btnStartTrain.setOnClickListener {
                // Start training process

viewModel.setTrainingState(MainViewModel.TrainingState.TRAINING)
                transferLearningHelper.startTraining()
            }
            radioButton.setOnCheckedChangeListener { _, checkedId ->
                if (checkedId == R.id.btnTrainingMode) {
                    // Switch to training mode.
                    viewModel.setCaptureMode(true)
                } else {
                    if (viewModel.getTrainingState() ==
MainViewModel.TrainingState.PREPARE) {

fragmentCameraBinding.btnTrainingMode.isChecked = true

fragmentCameraBinding.btnInferenceMode.isChecked = false

                        Toast.makeText(
                            requireContext(), "Inference can only " +
```

```
                                    "start after training is done.",
Toast
                                .LENGTH_LONG
                            ).show()
                        } else {
                            // Pause the training process and switch to
inference mode.
                            transferLearningHelper.pauseTraining()

viewModel.setTrainingState(MainViewModel.TrainingState.PAUSE)
                            viewModel.setCaptureMode(false)
                        }
                    }
                }

                viewFinder.post {
                    // Set up the camera and its use cases
                    setUpCamera()
                }
            }
        }

        // Initialize CameraX, and prepare to bind the camera use cases
        private fun setUpCamera() {
            val cameraProviderFuture =
                ProcessCameraProvider.getInstance(requireContext())
            cameraProviderFuture.addListener(
                {
                    // CameraProvider
                    cameraProvider = cameraProviderFuture.get()

                    // Build and bind the camera use cases
                    bindCameraUseCases()
                },
                ContextCompat.getMainExecutor(requireContext())
            )
        }

        // For configuration change, doesn't care us probably
        override fun onConfigurationChanged(newConfig: Configuration) {
            super.onConfigurationChanged(newConfig)
            imageAnalyzer?.targetRotation =
                fragmentCameraBinding.viewFinder.display.rotation
        }

        // Declare and bind preview, capture and analysis use cases
        @SuppressLint("UnsafeOptInUsageError")
        private fun bindCameraUseCases() {

            // CameraProvider
            val cameraProvider =
                cameraProvider
                    ?: throw IllegalStateException("Camera initialization
failed.")

            // CameraSelector - makes assumption that we're only using the
back camera
            val cameraSelector =
                CameraSelector.Builder()
```

```
.requireLensFacing(CameraSelector.LENS_FACING_BACK).build()

        // Preview. Only using the 4:3 ratio because this is the
closest to our models
        preview =
            Preview.Builder()
                .setTargetAspectRatio(AspectRatio.RATIO_4_3)

.setTargetRotation(fragmentCameraBinding.viewFinder.display.rotation)
                .build()

        // ImageAnalysis. Using RGBA 8888 to match how our models work
        imageAnalyzer =
            ImageAnalysis.Builder()
                .setTargetAspectRatio(AspectRatio.RATIO_4_3)

.setTargetRotation(fragmentCameraBinding.viewFinder.display.rotation)

.setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)

.setOutputImageFormat(ImageAnalysis.OUTPUT_IMAGE_FORMAT_RGBA_8888)
                .build()
                // The analyzer can then be assigned to the instance
                .also {
                    it.setAnalyzer(cameraExecutor) { image ->
                        if (!::bitmapBuffer.isInitialized) {
                            // The image rotation and RGB image buffer
are initialized only once
                            // the analyzer has started running
                            bitmapBuffer = Bitmap.createBitmap(
                                image.width,
                                image.height,
                                Bitmap.Config.ARGB_8888
                            )
                        }

                        // poll is basically pop in queue
                        val sampleClass = addSampleRequests.poll()
                        if (sampleClass != null) {
                            addSample(image, sampleClass)

viewModel.increaseNumberOfSample(sampleClass)
                        } else {
                            // Invokes inference from TLHelper
                            if (viewModel.getCaptureMode() == false) {
                                classifyImage(image)
                            }
                        }
                        image.close()
                    }
                }

        // Must unbind the use-cases before rebinding them
        cameraProvider.unbindAll()

        try {
            // A variable number of use-cases can be passed here -
            // camera provides access to CameraControl & CameraInfo
            camera = cameraProvider.bindToLifecycle(
                this,
```

```
                cameraSelector,
                preview,
                imageAnalyzer
            )

            // Attach the viewfinder's surface provider to preview use
case

preview?.setSurfaceProvider(fragmentCameraBinding.viewFinder.surfacePr
ovider)
        } catch (exc: Exception) {
            Log.e(TAG, "Use case binding failed", exc)
        }
    }

    // Classify image using TLHelper
    private fun classifyImage(image: ImageProxy) {
        // Copy out RGB bits to the shared bitmap buffer
        image.use {
bitmapBuffer.copyPixelsFromBuffer(image.planes[0].buffer) }

        val imageRotation = image.imageInfo.rotationDegrees
        // Pass Bitmap and rotation to the transfer learning helper
for
        // processing and classification.
        transferLearningHelper.classify(bitmapBuffer, imageRotation)
    }

    // addSample to the training data
    private fun addSample(image: ImageProxy, className: String) {
        // Copy out RGB bits to the shared bitmap buffer
        image.use {
bitmapBuffer.copyPixelsFromBuffer(image.planes[0].buffer) }

        // Overview of all the training samples before we start
training
        // Print all of the samples class labels in log.d
        Log.d("addSamples-ClassesLabels", "Sample Class: $className")

        val imageRotation = image.imageInfo.rotationDegrees
        // Pass Bitmap and rotation to the transfer learning helper
for
        // processing and prepare training data.
        transferLearningHelper.addSample(bitmapBuffer, className,
imageRotation)
    }

    private fun createWhiteBitmap(width: Int, height: Int): Bitmap {
        val whiteBitmap = Bitmap.createBitmap(width, height,
Bitmap.Config.ARGB_8888)
        val canvas = Canvas(whiteBitmap)
        canvas.drawColor(Color.WHITE)
        return whiteBitmap
    }

    // Used only at the launch of the app
    // NOT USED ANYMORE - NEWCODE
    private fun addWhiteImageToAllClassesAndTrain() {
        val classNames = arrayOf(CLASS_ONE, CLASS_TWO, CLASS_THREE,
CLASS_FOUR)
```

```
        val whiteBitmap = createWhiteBitmap(bitmapBuffer.width,
bitmapBuffer.height)

        // Add white image to all classes
        for (className in classNames) {
            transferLearningHelper.addSample(whiteBitmap, className,
0)
            viewModel.increaseNumberOfSample(className)
        }

        // Train the model
        Log.d("WhiteImage", "White image start training")
        transferLearningHelper.startTraining()

        // Wait 2 seconds
        Thread.sleep(2000)
        Log.d("WhiteImage", "White image pause training")
        transferLearningHelper.pauseTraining()

        // Clear the samples
        //transferLearningHelper.resetTrainingSamples()
        //transferLearningHelper.clearReplayBuffer()
        //Log.d("WhiteImage", "White image clear training samples and
replay buffer")

        _isFirstTime = false
    }


    @SuppressLint("NotifyDataSetChanged")
    override fun onError(error: String) {
        activity?.runOnUiThread {
            Toast.makeText(requireContext(), error,
Toast.LENGTH_SHORT).show()
        }
    }

    @SuppressLint("NotifyDataSetChanged")
    override fun onResults(
        results: List<Category>?,
        inferenceTime: Long
    ) {
        activity?.runOnUiThread {
            // Update the result in inference mode.
            if (viewModel.getCaptureMode() == false) {
                // Show result
                results?.let { list ->
                    // Highlight the class which is highest score.
                    list.maxByOrNull { it.score }?.let {
                        highlightResult(it.label)
                    }
                    updateScoreClasses(list)
                }

                fragmentCameraBinding.tvInferenceTime.text =
                    String.format("%d ms", inferenceTime)
            }
        }
    }
```

```
    // Show the loss number after each training.
    override fun onLossResults(lossNumber: Float) {
        String.format(
            Locale.US,
            "Loss: %.3f", lossNumber
        ).let {
            fragmentCameraBinding.tvLossConsumerPause.text = it
            fragmentCameraBinding.tvLossConsumerResume.text = it
        }
    }

    // Show the accurate score of each class.
    private fun updateScoreClasses(categories: List<Category>) {
        categories.forEach {
            val view = getClassButtonScore(it.label)
            (view as? TextView)?.text = String.format(
                Locale.US, "%.1f", it.score
            )
        }
    }

    // Get the class button which represent for the label
    private fun getClassButton(label: String): View? {
        return when (label) {
            CLASS_ONE -> fragmentCameraBinding.llClassOne
            CLASS_TWO -> fragmentCameraBinding.llClassTwo
            CLASS_THREE -> fragmentCameraBinding.llClassThree
            CLASS_FOUR -> fragmentCameraBinding.llClassFour
            else -> null
        }
    }

    // Get the class button score which represent for the label
    private fun getClassButtonScore(label: String): View? {
        return when (label) {
            CLASS_ONE -> fragmentCameraBinding.tvNumberClassOne
            CLASS_TWO -> fragmentCameraBinding.tvNumberClassTwo
            CLASS_THREE -> fragmentCameraBinding.tvNumberClassThree
            CLASS_FOUR -> fragmentCameraBinding.tvNumberClassFour
            else -> null
        }
    }

    // Get the class name from resource id
    private fun getClassNameFromResourceId(id: Int): String {
        return when (id) {
            fragmentCameraBinding.llClassOne.id -> CLASS_ONE
            fragmentCameraBinding.llClassTwo.id -> CLASS_TWO
            fragmentCameraBinding.llClassThree.id -> CLASS_THREE
            fragmentCameraBinding.llClassFour.id -> CLASS_FOUR
            else -> {
                ""
            }
        }
    }

    // Highlight the current label and unhighlight the previous label
    private fun highlightResult(label: String?) {
        // skip the previous position if it is no position.
        previousClass?.let {
```

```
            setClassButtonHighlight(getClassButton(it), false)
        }
        if (label != null) {
            setClassButtonHighlight(getClassButton(label), true)
        }
        previousClass = label
    }

    private fun setClassButtonHighlight(view: View?, isHighlight:
Boolean) {
        view?.run {
            background = AppCompatResources.getDrawable(
                context,
                if (isHighlight) R.drawable.btn_default_highlight else
R.drawable.btn_default
            )
        }
    }

    // Update the number of samples. If there are no label in the
samples,
    // set it 0.
    private fun updateNumberOfSample(numberOfSamples: Map<String,
Int>) {
        fragmentCameraBinding.tvNumberClassOne.text = if
(numberOfSamples
                .containsKey(CLASS_ONE)
        ) numberOfSamples.getValue(CLASS_ONE)
            .toString()
        else "0"
        fragmentCameraBinding.tvNumberClassTwo.text = if
(numberOfSamples
                .containsKey(CLASS_TWO)
        ) numberOfSamples.getValue(CLASS_TWO)
            .toString()
        else "0"
        fragmentCameraBinding.tvNumberClassThree.text = if
(numberOfSamples
                .containsKey(CLASS_THREE)
        ) numberOfSamples.getValue(CLASS_THREE)
            .toString()
        else "0"
        fragmentCameraBinding.tvNumberClassFour.text = if
(numberOfSamples
                .containsKey(CLASS_FOUR)
        ) numberOfSamples.getValue(CLASS_FOUR)
            .toString()
        else "0"
    }

    // Update training button states UI
    private fun updateTrainingButtonState() {
        with(fragmentCameraBinding) {
            tvInferenceTime.visibility = if (viewModel
                    .getCaptureMode() == true
            ) View.GONE else View.VISIBLE

            btnCollectSample.visibility = if (
                viewModel.getTrainingState() ==
MainViewModel.TrainingState.PREPARE &&
```

```
                (viewModel.getNumberOfSample()?.size ?: 0) == 0 &&
viewModel
                        .getCaptureMode() == true
            ) View.VISIBLE else View.GONE

            btnStartTrain.visibility = if (
                viewModel.getTrainingState() ==
MainViewModel.TrainingState.PREPARE &&
                (viewModel.getNumberOfSample()?.size ?: 0) > 0 &&
viewModel
                        .getCaptureMode() == true
            ) View.VISIBLE else View.GONE

            btnPauseTrain.visibility =
                if (viewModel.getTrainingState() == MainViewModel
                        .TrainingState.TRAINING && viewModel
                        .getCaptureMode() == true
                ) View.VISIBLE else View.GONE

            btnResumeTrain.visibility =
                if (viewModel.getTrainingState() == MainViewModel
                        .TrainingState.PAUSE && viewModel
                        .getCaptureMode() == true
                ) View.VISIBLE else View.GONE

            // Disable adding button when it is training or in
inference mode.
            (viewModel.getCaptureMode() == true &&
viewModel.getTrainingState() !=
                    MainViewModel.TrainingState.TRAINING).let { enable
->
                llClassOne.isClickable = enable
                llClassTwo.isClickable = enable
                llClassThree.isClickable = enable
                llClassFour.isClickable = enable
            }
        }
    }
}
```

1.1.2 - HelperDialog

```
/*
 * Copyright 2022 The TensorFlow Authors. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *             http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.tensorflow.lite.examples.modelpersonalization.fragments

import android.app.Dialog
import android.os.Bundle
import androidx.appcompat.app.AlertDialog
import androidx.appcompat.app.AppCompatDialogFragment
import org.tensorflow.lite.examples.modelpersonalization.R

// HelperDialog solely used for when  we click on the help button
class HelperDialog : AppCompatDialogFragment() {
    companion object {
        const val TAG = "Helper Dialog"
    }

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        return AlertDialog.Builder(requireActivity()).apply {

setTitle(requireActivity().getString(R.string.helper_dialog_title))
            setMessage(
                requireActivity().getString(
                    R.string
                        .helper_dialog_content
                )
            )
            setPositiveButton("ok") { _, _ ->
                // no-op
            }
        }.create()
    }
}
```

1.1.3 - PermissionsFragment

```
/*
 * Copyright 2022 The TensorFlow Authors. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *             http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.tensorflow.lite.examples.modelpersonalization.fragments

import android.Manifest
import android.content.Context
import android.content.pm.PackageManager
import android.os.Bundle
import android.widget.Toast
import androidx.activity.result.contract.ActivityResultContracts
import androidx.core.content.ContextCompat
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.Navigation
import org.tensorflow.lite.examples.modelpersonalization.R

private val PERMISSIONS_REQUIRED = arrayOf(Manifest.permission.CAMERA)

// A fragment used to request camera permissions, and then navigate to
the camera fragment.

class PermissionsFragment : Fragment() {

    private val requestPermissionLauncher =
        registerForActivityResult(
            ActivityResultContracts.RequestPermission()
        ) { isGranted: Boolean ->
            if (isGranted) {
                Toast.makeText(context, "Permission request granted",
Toast.LENGTH_LONG).show()
                navigateToCamera()
            } else {
                Toast.makeText(context, "Permission request denied",
Toast.LENGTH_LONG).show()
            }
        }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        when {
            ContextCompat.checkSelfPermission(
                requireContext(),
                Manifest.permission.CAMERA
```

```
            ) == PackageManager.PERMISSION_GRANTED -> {
                navigateToCamera()
            }
            else -> {
                requestPermissionLauncher.launch(
                    Manifest.permission.CAMERA
                )
            }
        }
    }

    private fun navigateToCamera() {
        lifecycleScope.launchWhenStarted {
            Navigation.findNavController(requireActivity(),
R.id.fragment_container).navigate(

PermissionsFragmentDirections.actionPermissionsToCamera()
            )
        }
    }

    companion object {

        /** Convenience method used to check if all permissions
required by this app are granted */
        fun hasPermissions(context: Context) =
PERMISSIONS_REQUIRED.all {
            ContextCompat.checkSelfPermission(context, it) ==
PackageManager.PERMISSION_GRANTED
        }
    }
}
```

1.1.4 - SettingFragment

```
/*
 * Copyright 2022 The TensorFlow Authors. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *             http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.tensorflow.lite.examples.modelpersonalization.fragments

import android.app.Dialog
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.DialogFragment
import androidx.fragment.app.activityViewModels
import org.tensorflow.lite.examples.modelpersonalization.MainViewModel
import
org.tensorflow.lite.examples.modelpersonalization.databinding.Fragment
SettingBinding

// SettingFragment is used to set/change the number of threads used
for classification

class SettingFragment : DialogFragment() {
    companion object {
        const val TAG = "SettingDialogFragment"
    }

    private var _fragmentSettingBinding: FragmentSettingBinding? =
null
    private val fragmentSettingBinding
        get() = _fragmentSettingBinding!!

    private var numThreads = 2
    private val viewModel: MainViewModel by activityViewModels()

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        return super.onCreateDialog(savedInstanceState).apply {
            setCancelable(false)
            setCanceledOnTouchOutside(false)
        }
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
```

```
    ): View {

        _fragmentSettingBinding = FragmentSettingBinding.inflate(
            inflater,
            container,
            false
        )
        return fragmentSettingBinding.root
    }

    override fun onViewCreated(view: View, savedInstanceState:
Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewModel.getNumThreads()?.let {
            numThreads = it
            updateDialogUi()
        }

        initDialogControls()

        fragmentSettingBinding.btnConfirm.setOnClickListener {
            viewModel.configModel(numThreads)
            dismiss()
        }
        fragmentSettingBinding.btnCancel.setOnClickListener {
            dismiss()
        }
    }

    private fun initDialogControls() {
        // When clicked, decrease the number of threads used for
classification
        fragmentSettingBinding.threadsMinus.setOnClickListener {
            if (numThreads > 1) {
                numThreads--
                updateDialogUi()
            }
        }

        // When clicked, increase the number of threads used for
classification
        fragmentSettingBinding.threadsPlus.setOnClickListener {
            if (numThreads < 4) {
                numThreads++
                updateDialogUi()
            }
        }
    }

    //  Update the values displayed in the dialog.
    private fun updateDialogUi() {
        fragmentSettingBinding.threadsValue.text =
            numThreads.toString()
    }
}
```

1.1.5 - MainActivity

```
/*
 * Copyright 2022 The TensorFlow Authors. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *             http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.tensorflow.lite.examples.modelpersonalization

import android.content.Context
import android.os.Build
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Toast
import androidx.activity.viewModels
import
org.tensorflow.lite.examples.modelpersonalization.databinding.Activity
MainBinding
import
org.tensorflow.lite.examples.modelpersonalization.fragments.HelperDial
og
import
org.tensorflow.lite.examples.modelpersonalization.fragments.SettingFra
gment
import java.io.File

class MainActivity : AppCompatActivity() {
    private lateinit var activityMainBinding: ActivityMainBinding
    private val viewModel: MainViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        activityMainBinding =
ActivityMainBinding.inflate(layoutInflater)
        setContentView(activityMainBinding.root)

        activityMainBinding.imgSetting.setOnClickListener {
            // Making sure that you can increase threads in settings
only when in training mode
            // and not inference mode
            if (viewModel.getCaptureMode() == true) {
                SettingFragment().show(
                    supportFragmentManager,
                    SettingFragment.TAG
                )
            } else {
                Toast.makeText(
                    this, "Change the setting only available in " +
```

```
                            "training mode", Toast.LENGTH_LONG
                    ).show()
                }
            }
            // Show HelperDialog when the user clicks on the helper icon
            activityMainBinding.tvHelper.setOnClickListener {
                HelperDialog().show(supportFragmentManager,
HelperDialog.TAG)
            }
            // Call refresh function when user clicks on the refresh icon
            // Can implement it in future
//            activityMainBinding.refreshButton.setOnClickListener {
//                restartAppAndModel()
//            }
        }

        // We can implement this as a reset button in future
        private fun restartAppAndModel() {
        }

        // Might be fixed but unsure
        override fun onBackPressed() {
            if (Build.VERSION.SDK_INT == Build.VERSION_CODES.Q) {
                // Workaround for Android Q memory leak issue in
IRequestFinishCallback$Stub.
                // (https://issuetracker.google.com/issues/139738913)
                finishAfterTransition()
            } else {
                super.onBackPressed()
            }
        }

        companion object {
             fun getCheckpointPath(context: Context): String {
                val checkpointDir = context.getDir("checkpoints",
Context.MODE_PRIVATE)
                if (!checkpointDir.exists()) {
                    checkpointDir.mkdirs()
                }
                return File(checkpointDir, "checkpoint").absolutePath
            }
        }
}
```

1.1.6 – MainViewModel

```
/*
 * Copyright 2022 The TensorFlow Authors. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *             http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.tensorflow.lite.examples.modelpersonalization

import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import java.util.*


class MainViewModel : ViewModel() {
    private val _numThread = MutableLiveData<Int>()
    val numThreads get() = _numThread

    private val _trainingState =
        MutableLiveData(TrainingState.PREPARE)
    val trainingState get() = _trainingState

    private val _captureMode = MutableLiveData(true)
    val captureMode get() = _captureMode

    private val _numberOfSamples = MutableLiveData(TreeMap<String,
Int>())
    val numberOfSamples get() = _numberOfSamples

    fun configModel(numThreads: Int) {
        _numThread.value = numThreads
    }

    fun getNumThreads() = numThreads.value

    fun setTrainingState(state: TrainingState) {
        _trainingState.value = state
    }

    fun getTrainingState() = trainingState.value

    fun setCaptureMode(isCapture: Boolean) {
        _captureMode.value = isCapture
    }

    fun getCaptureMode() = captureMode.value

    fun increaseNumberOfSample(className: String) {
```

```
        val map: TreeMap<String, Int> = _numberOfSamples.value!!
        val currentNumber: Int = if (map.containsKey(className)) {
            map[className]!!
        } else {
            0
        }
        map[className] = currentNumber + 1
        _numberOfSamples.postValue(map)
    }

    fun getNumberOfSample() = numberOfSamples.value

    enum class TrainingState {
        PREPARE, TRAINING, PAUSE
    }
}
```

1.1.7 - TransferLearningHelper

```
/*
 * Copyright 2022 The TensorFlow Authors. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *             http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

// ANYTHING THAT IS RELATED TO THE REPLAY BUFFER IS NEW CODE

package org.tensorflow.lite.examples.modelpersonalization

import android.content.Context
import android.graphics.Bitmap
import android.os.Handler
import android.os.Looper
import android.os.SystemClock
import android.util.Log
import org.tensorflow.lite.DataType
import org.tensorflow.lite.Interpreter
import org.tensorflow.lite.support.common.FileUtil
import org.tensorflow.lite.support.common.ops.NormalizeOp
import org.tensorflow.lite.support.image.ImageProcessor
import org.tensorflow.lite.support.image.TensorImage
import org.tensorflow.lite.support.image.ops.ResizeOp
import org.tensorflow.lite.support.image.ops.ResizeWithCropOrPadOp
import org.tensorflow.lite.support.image.ops.Rot90Op
import org.tensorflow.lite.support.label.Category
import org.tensorflow.lite.support.label.TensorLabel
import org.tensorflow.lite.support.tensorbuffer.TensorBuffer
import java.io.IOException
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.nio.FloatBuffer
import java.util.*
import java.util.concurrent.ExecutorService
import java.util.concurrent.Executors
import kotlin.math.max
import kotlin.math.min
import kotlin.random.Random

// This class is responsible for the training process and inference
process.

class TransferLearningHelper(
    var numThreads: Int = 2,
    val context: Context,
    // A listener to receive updates on the models performance during
training.
```

```
    val classifierListener: ClassifierListener?
) {

    // A tf.lite interpreter used for inference and training.
(contains the signatures)
    private var interpreter: Interpreter? = null
    // A list of training samples objects which contain the bottleneck
    // and label data needed for model training (I THINK).
    // Changed from private to public
    val trainingSamples: MutableList<TrainingSample> = mutableListOf()
    // ExecutorService running the model training process
    private var executor: ExecutorService? = null

    //This lock guarantees that only one thread is performing training
and
    //inference at any point in time.
    // Other threads could be adding samples to the trainingSamples
list
    // or updating the UI and handling user input (I THINK)
    private val lock = Any()
    // Input image dimensions for the MobileNet model.(224, 224)
    private var targetWidth: Int = 0
    private var targetHeight: Int = 0
    // Handler running on the main thread (UI thread).
    private val handler = Handler(Looper.getMainLooper())

    // Our replayBuffer - NEW
    private val replayBuffer: MutableList<TrainingSample> =
mutableListOf()

    // Used as a flag because pauseTraining is called both when we
pause the training
    // and when the inference button is called. We only want to update
the replayBuffer once
    private var replayBufferUpdated = false
    private var firstTrainingFlag = true

    init {
        if (setupModelPersonalization()) {
            targetWidth = interpreter!!.getInputTensor(0).shape()[2]
            targetHeight = interpreter!!.getInputTensor(0).shape()[1]
        } else {
            classifierListener?.onError("TFLite failed to init.")
        }
    }

    // Close the interpreter and executor.
    fun close() {
        executor?.shutdownNow()
        executor = null
        interpreter = null
    }

    // KNOWN BUG
    // Keep an eye that if we press pause and train again, every time
we do that, the reply buffer
    // updates again. So make sure to train and then pause and add new
samples if you want
    // to train again.
```

```
// NEW
    fun pauseTraining() {
        // Update replayBuffer with samples from this training cycle

        // Apparently, pauseTraining is called when we pause the
training but
        // also when we click the inference button. This is the fix to
it
        if(!replayBufferUpdated){
            Log.d("PauseTraining", "Updating replay buffer")
            // Disabling these for now
            updateReplayBuffer()
            resetTrainingSamples()
            replayBufferUpdated = true
        }

        executor?.shutdownNow()
    }

    // We get the model.tflite and load it into the interpreter.
    private fun setupModelPersonalization(): Boolean {
        val options = Interpreter.Options()
        options.numThreads = numThreads
        return try {
            val modelFile = FileUtil.loadMappedFile(context,
"modelnew.tflite")
            interpreter = Interpreter(modelFile, options)
            true
        } catch (e: IOException) {
            classifierListener?.onError(
                "Model personalization failed to " +
                        "initialize. See error logs for details"
            )
            Log.e(TAG, "TFLite failed to load model with error: " +
e.message)
            false
        }
    }

    // Process input image and add the output into list samples which
are
    // ready for training.

    fun addSample(image: Bitmap, className: String, rotation: Int) {
        synchronized(lock) {
            if (interpreter == null) {
                setupModelPersonalization()
            }
            processInputImage(image, rotation)?.let { tensorImage ->
                val bottleneck = loadBottleneck(tensorImage)
                val newSample = TrainingSample(
                        bottleneck,
                        encoding(classes.getValue(className))
                    )
                trainingSamples.add(newSample)
            }
        }
    }

    // Start training process
```

```
// FUNCTION IS UPDATED WITH THE NEW CODE
    fun startTraining() {

        if (interpreter == null || firstTrainingFlag) {
            setupModelPersonalization()
            firstTrainingFlag = false

        }
        else
        {
            // Saving and loading of weights can be used in future if
we have a lot of
            // layers in the head of the model

            // Save weights
//              val checkpointPath =
MainActivity.getCheckpointPath(context)
//              val saveInputs: MutableMap<String, Any> = HashMap()
//              saveInputs[SAVE_INPUT_KEY] = checkpointPath
//              val saveOutputs: MutableMap<String, Any> = HashMap()
//              saveOutputs[SAVE_OUTPUT_KEY] = checkpointPath
//              interpreter?.runSignature(saveInputs, saveOutputs,
SAVE_KEY)

            setupModelPersonalization()

            // Load weights
//              val restoreInputs: MutableMap<String, Any> = HashMap()
//              restoreInputs[RESTORE_INPUT_KEY] = checkpointPath
//              val restoreOutputs: MutableMap<String, Any> = HashMap()
//              val restoredTensors = HashMap<String, FloatArray>()
//              restoreOutputs[RESTORE_OUTPUT_KEY] = restoredTensors
//              interpreter?.runSignature(restoreInputs, restoreOutputs,
RESTORE_KEY)
        }

        // Reset the replayBufferUpdated flag
        replayBufferUpdated = false

        // Create new thread for training process.
        executor = Executors.newSingleThreadExecutor()
        val trainBatchSize = getTrainBatchSize()

        // The fix of this exception I think is not in the
getTrainBatchSize() function
        // but rather adding both training samples and replay buffer
samples in the if statement
        if (trainingSamples.size + replayBuffer.size < trainBatchSize)
{
            throw RuntimeException(
                String.format(
                    "Too few samples to start training: need %d, got
%d",
                    trainBatchSize, trainingSamples.size
                )
            )
        }

        Log.d("ReplayBuffer", "Replay buffer size:
${replayBuffer.size}")
```

```
        Log.d("ReplayBuffer", "Training samples size:
${trainingSamples.size}")

        // Combine the training samples and the replay buffer samples
        val combinedSamples = (trainingSamples +
replayBuffer).toMutableList()
        // combinedSamples.shuffle()

        Log.d("ReplayBuffer","Combined samples size:
${combinedSamples.size}")

        executor?.execute {
            synchronized(lock) {
                var avgLoss: Float

                // Keep training until the helper pause or close.
                while (executor?.isShutdown == false) {
                    var totalLoss = 0f
                    var numBatchesProcessed = 0

                    // Shuffle training samples to reduce overfitting
and
                    // variance.
                    combinedSamples.shuffle()

                    // Now trainingBatches will be called with both
the training samples and the replay buffer samples
                    // The function implementation will change to
adapt to this
                    trainingBatches(trainBatchSize, combinedSamples)
                        .forEach { combinedSamplesCurrentBatch ->
                            val trainingBatchBottlenecks =
                                MutableList(trainBatchSize) {
                                    FloatArray(
                                        BOTTLENECK_SIZE
                                    )
                                }

                            val trainingBatchLabels =
                                MutableList(trainBatchSize) {
                                    FloatArray(
                                        classes.size
                                    )
                                }

                            // Copy a training sample list into two
different
                            // input training lists.
                            combinedSamplesCurrentBatch.forEachIndexed
{ index, trainingSample ->
                                trainingBatchBottlenecks[index] =
                                    trainingSample.bottleneck
                                trainingBatchLabels[index] =
                                    trainingSample.label
                            }

                            val loss = training(
                                trainingBatchBottlenecks,
                                trainingBatchLabels
                            )
```

```
                            totalLoss += loss
                            numBatchesProcessed++
                        }

                    // Calculate the average loss after training all
batches.
                    avgLoss = totalLoss / numBatchesProcessed
                    handler.post {
                        classifierListener?.onLossResults(avgLoss)
                    }

                }
            }
        }
    }

    // Used for debugging to check the weights after training
// NEW
    private fun getModelWeightsHead(bottlenecks:
MutableList<FloatArray>): FloatArray {
        val wsSize = BOTTLENECK_SIZE * NUM_CLASSES * 4 // 4 bytes per
float
        val wsBuffer =
ByteBuffer.allocateDirect(wsSize).order(ByteOrder.nativeOrder())

        val inputs: MutableMap<String, Any> = HashMap()
        // Used just because we need an input. Has no use
        inputs[TRAINING_INPUT_BOTTLENECK_KEY] =
bottlenecks.toTypedArray()

        val outputs: MutableMap<String, Any> = HashMap()
        outputs["ws"] = wsBuffer

        // Get weights and biases as defined in our signature in the
model.
        interpreter?.runSignature(inputs, outputs, "initialize")

        val wsArray = FloatArray(BOTTLENECK_SIZE * NUM_CLASSES)
        wsBuffer.rewind()
        wsBuffer.asFloatBuffer().get(wsArray)

        return wsArray
    }

    // Runs one training step with the given bottleneck batches and
labels
    // and return the loss number.
    private fun training(
        bottlenecks: MutableList<FloatArray>,
        labels: MutableList<FloatArray>
    ): Float {

        val inputs: MutableMap<String, Any> = HashMap()
        inputs[TRAINING_INPUT_BOTTLENECK_KEY] =
bottlenecks.toTypedArray()
        inputs[TRAINING_INPUT_LABELS_KEY] = labels.toTypedArray()

        val outputs: MutableMap<String, Any> = HashMap()
        val loss = FloatBuffer.allocate(1)
        outputs[TRAINING_OUTPUT_KEY] = loss
```

```
        // Training as defined in our signature in the model.
        interpreter?.runSignature(inputs, outputs, TRAINING_KEY)

        Log.d("Loss", "Loss is ${loss.get(0)}")

        // Used for debugging

        // Get weights and biases after training
        // val weights = getModelWeightsHead(bottlenecks)
        // Use or print the weights and biases as needed
        // Log.d("WeightsAndBiases", "Weights:
${weights.contentToString()}")

        return loss.get(0)
    }

    // Invokes inference on the given image batches.
    fun classify(bitmap: Bitmap, rotation: Int) {
        processInputImage(bitmap, rotation)?.let { image ->
            synchronized(lock) {
                if (interpreter == null) {
                    setupModelPersonalization()
                }

                // Inference time is the difference between the system
time at the start and finish of the
                // process
                var inferenceTime = SystemClock.uptimeMillis()

                val inputs: MutableMap<String, Any> = HashMap()
                inputs[INFERENCE_INPUT_KEY] = image.buffer

                val outputs: MutableMap<String, Any> = HashMap()
                val output = TensorBuffer.createFixedSize(
                    intArrayOf(1, 4),
                    DataType.FLOAT32
                )
                outputs[INFERENCE_OUTPUT_KEY] = output.buffer

                interpreter?.runSignature(inputs, outputs,
INFERENCE_KEY)
                val tensorLabel = TensorLabel(classes.keys.toList(),
output)
                val result = tensorLabel.categoryList

                inferenceTime = SystemClock.uptimeMillis() -
inferenceTime

                classifierListener?.onResults(result, inferenceTime)
            }
        }
    }

    // Loads the bottleneck feature from the given image array.
    private fun loadBottleneck(image: TensorImage): FloatArray {
        val inputs: MutableMap<String, Any> = HashMap()
        inputs[LOAD_BOTTLENECK_INPUT_KEY] = image.buffer
        val outputs: MutableMap<String, Any> = HashMap()
        val bottleneck = Array(1) { FloatArray(BOTTLENECK_SIZE) }
```

```
        outputs[LOAD_BOTTLENECK_OUTPUT_KEY] = bottleneck
        interpreter?.runSignature(inputs, outputs,
LOAD_BOTTLENECK_KEY)
        return bottleneck[0]
    }

    // Preprocess the image and convert it into a TensorImage for
classification.
    private fun processInputImage(
        image: Bitmap,
        imageRotation: Int
    ): TensorImage? {
        val height = image.height
        val width = image.width
        val cropSize = min(height, width)
        val imageProcessor = ImageProcessor.Builder()
            .add(Rot90Op(-imageRotation / 90))
            .add(ResizeWithCropOrPadOp(cropSize, cropSize))
            .add(
                ResizeOp(
                    targetHeight,
                    targetWidth,
                    ResizeOp.ResizeMethod.BILINEAR
                )
            )
            .add(NormalizeOp(0f, 255f))
            .build()
        val tensorImage = TensorImage(DataType.FLOAT32)
        tensorImage.load(image)
        return imageProcessor.process(tensorImage)
    }

    // encode the classes name to float array
    private fun encoding(id: Int): FloatArray {
        val classEncoded = FloatArray(4) { 0f }
        classEncoded[id] = 1f
        return classEncoded
    }

    // Training model expected batch size.
    // We can ask as many samples as we want
    private fun getTrainBatchSize(): Int {

        Log.d("TrainBatch", "Training samples:
${trainingSamples.size}")
        Log.d("TrainBatch", "Replay buffer: ${replayBuffer.size}")

        // Added replayBuffer sizes here too
        return min(
            max( /* at least one sample needed */1,
trainingSamples.size + replayBuffer.size),
            EXPECTED_BATCH_SIZE
        )
    }

    // Constructs an iterator that iterates over training sample
batches.
    // Altered the function to include replayBuffer samples as well
    private fun trainingBatches(trainBatchSize: Int, samples:
List<TrainingSample>): Iterator<List<TrainingSample>> {
```

```
        return object : Iterator<List<TrainingSample>> {
            private var nextIndex = 0

            override fun hasNext(): Boolean {
                return nextIndex < samples.size
            }

            override fun next(): List<TrainingSample> {
                val fromIndex = nextIndex
                val toIndex: Int = nextIndex + trainBatchSize
                nextIndex = toIndex
                return if (toIndex >= samples.size) {
                    // To keep batch size consistent, last batch may
include some elements from the
                    // next-to-last batch.
                    samples.subList(
                        samples.size - trainBatchSize,
                        samples.size
                    )
                } else {
                    samples.subList(fromIndex, toIndex)
                }
            }
        }
    }

// NEW FUNCTION
    private fun updateReplayBuffer(){
        // Portion of trainingSamples to add to replayBuffer
        // I could zero this if I want to disable replayBuffer
        val portion = 0.25

        val samplesToAdd = (trainingSamples.size * portion).toInt()

        // Might not be necessary
        trainingSamples.shuffle()

        Log.d("ReplayBuffer","Number of trainingSamples before
updating replayBuffer are: ${trainingSamples.size}")

        val samplesToAddToReplayBuffer = trainingSamples.subList(0,
samplesToAdd)

        // If samplesToAddToReplayBuffer are more than
REPLAY_BUFFER_SIZE we will remove some
        if (samplesToAddToReplayBuffer.size > REPLAY_BUFFER_SIZE){
            samplesToAddToReplayBuffer.subList(0,
samplesToAddToReplayBuffer.size - REPLAY_BUFFER_SIZE)
        }

        // Add them to replayBuffer
        replayBuffer.addAll(samplesToAddToReplayBuffer)

        Log.d("ReplayBuffer", "Adding
${samplesToAddToReplayBuffer.size} samples to replay buffer")
        Log.d("ReplayBuffer", "Replay buffer size before removing
extra samples is now: ${replayBuffer.size}")

        // We randomly remove samples from the replay buffer
```

```
        // CAUTION HERE
        // We basically add the new samples to the replay buffer
        // and then remove from the buffer until it reaches it's
        // maximum size. This means that probably some new samples
        // will be removed instantly.
        if(replayBuffer.size > REPLAY_BUFFER_SIZE){

            val samplesToRemove = replayBuffer.size -
REPLAY_BUFFER_SIZE

            Log.d("ReplayBuffer", "We will remove $samplesToRemove
samples from replay buffer")

            // We remove the excess samples
            for(i in 0 until samplesToRemove){
                val randomIndex = Random.nextInt(replayBuffer.size)
                val removedSample = replayBuffer.removeAt(randomIndex)
                Log.d("ReplayBuffer", "Removing samples at index
$randomIndex from replay buffer")
            }
        }

        Log.d("ReplayBuffer", "Replay buffer size after removing extra
samples is now: ${replayBuffer.size}")
    }

    // We want to remove the samples from the list after
    // training because the replayBuffer will retain the previous
knowledge
    public fun resetTrainingSamples(){
        trainingSamples.clear()
    }

    public fun clearReplayBuffer(){
        replayBuffer.clear()
    }

    interface ClassifierListener {
        fun onError(error: String)
        fun onResults(results: List<Category>?, inferenceTime: Long)
        fun onLossResults(lossNumber: Float)
    }

    companion object {
        const val CLASS_ONE = "1"
        const val CLASS_TWO = "2"
        const val CLASS_THREE = "3"
        const val CLASS_FOUR = "4"
        private val classes = mapOf(
            CLASS_ONE to 0,
            CLASS_TWO to 1,
            CLASS_THREE to 2,
            CLASS_FOUR to 3
        )
        private const val LOAD_BOTTLENECK_INPUT_KEY = "feature"
        private const val LOAD_BOTTLENECK_OUTPUT_KEY = "bottleneck"
        private const val LOAD_BOTTLENECK_KEY = "load"

        private const val TRAINING_INPUT_BOTTLENECK_KEY = "bottleneck"
        private const val TRAINING_INPUT_LABELS_KEY = "label"
```

```
        private const val TRAINING_OUTPUT_KEY = "loss"
        private const val TRAINING_KEY = "train"

        private const val INFERENCE_INPUT_KEY = "feature"
        private const val INFERENCE_OUTPUT_KEY = "output"
        private const val INFERENCE_KEY = "infer"

        private const val SAVE_INPUT_KEY = "checkpoint_path"
        private const val SAVE_OUTPUT_KEY = "checkpoint_path"
        private const val SAVE_KEY = "save"

        private const val RESTORE_INPUT_KEY = "checkpoint_path"
        private const val RESTORE_OUTPUT_KEY = "restored_tensors"
        private const val RESTORE_KEY = "restore"

        private const val NUM_CLASSES = 4
        private const val BOTTLENECK_SIZE = 1 * 7 * 7 * 1280
        private const val EXPECTED_BATCH_SIZE = 20
        private const val TAG = "ModelPersonalizationHelper"

        // Replay buffer size
        private const val REPLAY_BUFFER_SIZE = 100
    }

    data class TrainingSample(val bottleneck: FloatArray, val label:
FloatArray)
}
```

1.2 - TensorFlow-Lite Model Creation Code

```
# Copyright 2021 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""CLI wrapper for tflite_transfer_converter.

Converts a TF model to a TFLite transfer learning model.
"""

import os

import numpy as np
import tensorflow as tf

IMG_SIZE = 224
NUM_FEATURES = 7 * 7 * 1280
NUM_CLASSES = 4


class TransferLearningModel(tf.Module):
  """TF Transfer Learning model class."""

  def __init__(self, learning_rate=0.001):
    """Initializes a transfer learning model instance.

    Args:
      learning_rate: A learning rate for the optimzer.
    """
    self.num_features = NUM_FEATURES
    self.num_classes = NUM_CLASSES

    # trainable weights and bias for softmax
    self.ws = tf.Variable(
        tf.zeros((self.num_features, self.num_classes)),
        name='ws',
        trainable=True)
    self.bs = tf.Variable(
```

```
        tf.zeros((1, self.num_classes)), name='bs', trainable=True)

    # base model
    self.base = tf.keras.applications.MobileNetV2(
        input_shape=(IMG_SIZE, IMG_SIZE, 3),
        alpha=1.0,
        include_top=False,
        weights='imagenet')
    # loss function and optimizer
    self.loss_fn = tf.keras.losses.CategoricalCrossentropy()
    self.optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

  @tf.function(input_signature=[
      tf.TensorSpec([None, IMG_SIZE, IMG_SIZE, 3], tf.float32),
  ])
  def load(self, feature):
    """Generates and loads bottleneck features from the given image batch.

    Args:
      feature: A tensor of image feature batch to generate the bottleneck
from.

    Returns:
      Map of the bottleneck.
    """
    x = tf.keras.applications.mobilenet_v2.preprocess_input(
        tf.multiply(feature, 255))
    bottleneck = tf.reshape(
        self.base(x, training=False), (-1, self.num_features))
    return {'bottleneck': bottleneck}

  @tf.function(input_signature=[
      tf.TensorSpec([None, NUM_FEATURES], tf.float32),
      tf.TensorSpec([None, NUM_CLASSES], tf.float32),
  ])
  def train(self, bottleneck, label):
    """Runs one training step with the given bottleneck features and labels.

    Args:
      bottleneck: A tensor of bottleneck features generated from the base
model.
      label: A tensor of class labels for the given batch.

    Returns:
      Map of the training loss.
    """
    with tf.GradientTape() as tape:
      logits = tf.matmul(bottleneck, self.ws) + self.bs
```

```
      prediction = tf.nn.softmax(logits)
      loss = self.loss_fn(prediction, label)
    gradients = tape.gradient(loss, [self.ws, self.bs])
    self.optimizer.apply_gradients(zip(gradients, [self.ws, self.bs]))
    result = {'loss': loss}
    for grad in gradients:
      result[grad.name] = grad
    return result

  @tf.function(input_signature=[
      tf.TensorSpec([None, IMG_SIZE, IMG_SIZE, 3], tf.float32)
  ])
  def infer(self, feature):
    """Invokes an inference on the given feature.

    Args:
      feature: A tensor of image feature batch to invoke an inference on.

    Returns:
      Map of the softmax output.
    """
    x = tf.keras.applications.mobilenet_v2.preprocess_input(
        tf.multiply(feature, 255))
    bottleneck = tf.reshape(
        self.base(x, training=False), (-1, self.num_features))
    logits = tf.matmul(bottleneck, self.ws) + self.bs
    return {'output': tf.nn.softmax(logits)}

  @tf.function(input_signature=[tf.TensorSpec(shape=[], dtype=tf.string)])
  def save(self, checkpoint_path):
    """Saves the trainable weights to the given checkpoint file.

    Args:
      checkpoint_path: A file path to save the model.

    Returns:
      Map of the checkpoint file path.
    """
    tensor_names = [self.ws.name, self.bs.name]
    tensors_to_save = [self.ws.read_value(), self.bs.read_value()]
    tf.raw_ops.Save(
        filename=checkpoint_path,
        tensor_names=tensor_names,
        data=tensors_to_save,
        name='save')
    return {'checkpoint_path': checkpoint_path}

  @tf.function(input_signature=[tf.TensorSpec(shape=[], dtype=tf.string)])
```

```
  def restore(self, checkpoint_path):
    """Restores the serialized trainable weights from the given checkpoint
file.

    Args:
      checkpoint_path: A path to a saved checkpoint file.

    Returns:
      Map of restored weight and bias.
    """
    restored_tensors = {}
    restored = tf.raw_ops.Restore(
        file_pattern=checkpoint_path,
        tensor_name=self.ws.name,
        dt=np.float32,
        name='restore')
    self.ws.assign(restored)
    restored_tensors['ws'] = restored
    restored = tf.raw_ops.Restore(
        file_pattern=checkpoint_path,
        tensor_name=self.bs.name,
        dt=np.float32,
        name='restore')
    self.bs.assign(restored)
    restored_tensors['bs'] = restored
    return restored_tensors

  # Added this extra function
  # @tf.function(input_signature=[])
  # def get_weights_and_biases(self):
  #   """Returns the weights and biases of the head model.

  #   Returns:
  #     Map of weight and bias.
  #   """
  #   return {'ws': self.ws, 'bs': self.bs}

  # This is basically get weights and biases. For some reason if i use a
different name the input tensor gets zeroed
  # bottleneck not used, we just need an input for some reason it cant be
empty
  @tf.function(input_signature=[
    tf.TensorSpec([None, NUM_FEATURES], tf.float32)
  ])
  def initialize_weights(self, bottleneck):
    """Initializes the weights and bias of the head model.

    Returns:
```

```
      Map of initialized weight and bias.
    """
    #self.ws.assign(tf.random.uniform((self.num_features, self.num_classes)))
    #self.bs.assign(tf.random.uniform((1, self.num_classes)))
    return {'ws': self.ws}


def convert_and_save(saved_model_dir='saved_model'):
  """Converts and saves the TFLite Transfer Learning model.

  Args:
    saved_model_dir: A directory path to save a converted model.
  """
  model = TransferLearningModel()

  tf.saved_model.save(
      model,
      saved_model_dir,
      signatures={
          'load': model.load.get_concrete_function(),
          'train': model.train.get_concrete_function(),
          'infer': model.infer.get_concrete_function(),
          'save': model.save.get_concrete_function(),
          'restore': model.restore.get_concrete_function(),
          #'get_weights_and_biases':
model.get_weights_and_biases.get_concrete_function(),
          'initialize': model.initialize_weights.get_concrete_function(),
      })

  # Convert the model
  converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
  converter.target_spec.supported_ops = [
      tf.lite.OpsSet.TFLITE_BUILTINS,  # enable TensorFlow Lite ops.
      tf.lite.OpsSet.SELECT_TF_OPS  # enable TensorFlow ops.
  ]
  converter.experimental_enable_resource_variables = True
  tflite_model = converter.convert()

  model_file_path = os.path.join('modelnew.tflite')
  with open(model_file_path, 'wb') as model_file:
    model_file.write(tflite_model)


if __name__ == '__main__':
  convert_and_save()
```

2 - Offline Experiments

2.1 - Controller class

```
from experiments import Experiments
import argparse
import numpy as np
import tensorflow as tf

# Set seeds for reproducibility
# https://keras.io/getting_started/faq/#how-can-i-obtain-reproducible-
results-using-keras-during-development
# Due to the fact that we are running the experiments on GPU, there is
some non-determinism involved.
SEED = 1
np.random.seed(SEED)
tf.random.set_seed(SEED)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--exp_RBS_3000", action="store_true")
    parser.add_argument("--exp_RBS_5000", action="store_true")
    parser.add_argument("--exp_RBS_7500", action="store_true")
    parser.add_argument("--exp_RBS_10000", action="store_true")
    parser.add_argument("--exp_RBS_15000", action="store_true")
    parser.add_argument("--exp_RBS_20000", action="store_true")
    parser.add_argument("--exp_RBS_30000", action="store_true")
    args = parser.parse_args()
    experiments = Experiments()

    # Experimenting with different replay buffer sizes.
    # For each experiment we move the replay layer 1 hidden layer
backwards (We move hidden layers from MobileNetV2 to the head of the
model and make them trainable)

    if args.exp_RBS_3000:
        print("> Experiment: RBS_3000")

        experiments.runHiddenLayersExperiment(experiment_name="RBS_3000_H
L_EXPERIMENTS",
                                              usecase="RBS_3000_HL0
",
                                              replay_size=3000,
num_hidden_layers=0)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_3000_H
L_EXPERIMENTS",
                                              usecase="RBS_3000_HL1
",
```

```
                                                    replay_size=3000,
num_hidden_layers=1)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_3000_H
L_EXPERIMENTS",
                                                    usecase="RBS_3000_HL2
",
                                                    replay_size=3000,
num_hidden_layers=2)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_3000_H
L_EXPERIMENTS",
                                                    usecase="RBS_3000_HL3
",
                                                    replay_size=3000,
num_hidden_layers=3)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_3000_H
L_EXPERIMENTS",
                                                    usecase="RBS_3000_HL4
",
                                                    replay_size=3000,
num_hidden_layers=4)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_3000_H
L_EXPERIMENTS",
                                                    usecase="RBS_3000_HL5
",
                                                    replay_size=3000,
num_hidden_layers=5)

        experiments.plotExperiment(experiment_name="RBS_3000_HL_EXPERIMEN
TS",
                                   title="RBS_3000_HL_EXPERIMENTS (CORe50
NICv2 - 391)")

    elif args.exp_RBS_5000:
        print("> Experiment: RBS_5000")

        experiments.runHiddenLayersExperiment(experiment_name="RBS_5000_H
L_EXPERIMENTS",
                                                    usecase="RBS_5000_HL0
",
                                                    replay_size=5000,
num_hidden_layers=0)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_5000_H
L_EXPERIMENTS",
```

```
                                                    usecase="RBS_5000_HL1
",
                                                    replay_size=5000,
num_hidden_layers=1)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_5000_H
L_EXPERIMENTS",
                                                    usecase="RBS_5000_HL2
",
                                                    replay_size=5000,
num_hidden_layers=2)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_5000_H
L_EXPERIMENTS",
                                                    usecase="RBS_5000_HL3
",
                                                    replay_size=5000,
num_hidden_layers=3)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_5000_H
L_EXPERIMENTS",
                                                    usecase="RBS_5000_HL4
",
                                                    replay_size=5000,
num_hidden_layers=4)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_5000_H
L_EXPERIMENTS",
                                                    usecase="RBS_5000_HL5
",
                                                    replay_size=5000,
num_hidden_layers=5)

        experiments.plotExperiment(experiment_name="RBS_5000_HL_EXPERIMEN
TS",
                                   title="RBS_5000_HL_EXPERIMENTS (CORe50
NICv2 - 391)")

    elif args.exp_RBS_7500:
        print("> Experiment: RBS_7500")

        experiments.runHiddenLayersExperiment(experiment_name="RBS_7500_H
L_EXPERIMENTS",
                                                    usecase="RBS_7500_HL0
",
                                                    replay_size=7500,
num_hidden_layers=0)
```

```
        experiments.runHiddenLayersExperiment(experiment_name="RBS_7500_H
L_EXPERIMENTS",
                                              usecase="RBS_7500_HL1
",
                                              replay_size=7500,
num_hidden_layers=1)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_7500_H
L_EXPERIMENTS",
                                              usecase="RBS_7500_HL2
",
                                              replay_size=7500,
num_hidden_layers=2)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_7500_H
L_EXPERIMENTS",
                                              usecase="RBS_7500_HL3
",
                                              replay_size=7500,
num_hidden_layers=3)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_7500_H
L_EXPERIMENTS",
                                              usecase="RBS_7500_HL4
",
                                              replay_size=7500,
num_hidden_layers=4)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_7500_H
L_EXPERIMENTS",
                                              usecase="RBS_7500_HL5
",
                                              replay_size=7500,
num_hidden_layers=5)

        experiments.plotExperiment(experiment_name="RBS_7500_HL_EXPERIMEN
TS",
                                   title="RBS_7500_HL_EXPERIMENTS (CORe50
NICv2 - 391)")

    elif args.exp_RBS_10000:
        print("> Experiment: RBS_10000")

        experiments.runHiddenLayersExperiment(experiment_name="RBS_10000_
HL_EXPERIMENTS",
                                              usecase="RBS_10000_HL
0",
```

```
                                                    replay_size=10000,
num_hidden_layers=0)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_10000_
HL_EXPERIMENTS",
                                                    usecase="RBS_10000_HL
1",
                                                    replay_size=10000,
num_hidden_layers=1)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_10000_
HL_EXPERIMENTS",
                                                    usecase="RBS_10000_HL
2",
                                                    replay_size=10000,
num_hidden_layers=2)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_10000_
HL_EXPERIMENTS",
                                                    usecase="RBS_10000_HL
3",
                                                    replay_size=10000,
num_hidden_layers=3)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_10000_
HL_EXPERIMENTS",
                                                    usecase="RBS_10000_HL
4",
                                                    replay_size=10000,
num_hidden_layers=4)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_10000_
HL_EXPERIMENTS",
                                                    usecase="RBS_10000_HL
5",
                                                    replay_size=10000,
num_hidden_layers=5)

        experiments.plotExperiment(experiment_name="RBS_10000_HL_EXPERIME
NTS",
                                   title="RBS_10000_HL_EXPERIMENTS
(CORe50 NICv2 - 391)")

    elif args.exp_RBS_15000:
        print("> Experiment: RBS_15000")

        experiments.runHiddenLayersExperiment(experiment_name="RBS_15000_
HL_EXPERIMENTS",
```

```
                                                    usecase="RBS_15000_HL
0",
                                                    replay_size=15000,
num_hidden_layers=0)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_15000_
HL_EXPERIMENTS",
                                                    usecase="RBS_15000_HL
1",
                                                    replay_size=15000,
num_hidden_layers=1)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_15000_
HL_EXPERIMENTS",
                                                    usecase="RBS_15000_HL
2",
                                                    replay_size=15000,
num_hidden_layers=2)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_15000_
HL_EXPERIMENTS",
                                                    usecase="RBS_15000_HL
3",
                                                    replay_size=15000,
num_hidden_layers=3)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_15000_
HL_EXPERIMENTS",
                                                    usecase="RBS_15000_HL
4",
                                                    replay_size=15000,
num_hidden_layers=4)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_15000_
HL_EXPERIMENTS",
                                                    usecase="RBS_15000_HL
5",
                                                    replay_size=15000,
num_hidden_layers=5)

        experiments.plotExperiment(experiment_name="RBS_15000_HL_EXPERIME
NTS",
                                   title="RBS_15000_HL_EXPERIMENTS
(CORe50 NICv2 - 391)")

    elif args.exp_RBS_20000:
        print("> Experiment: RBS_20000")
```

```
        experiments.runHiddenLayersExperiment(experiment_name="RBS_20000_
HL_EXPERIMENTS",
                                              usecase="RBS_20000_HL
0",
                                              replay_size=20000,
num_hidden_layers=0)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_20000_
HL_EXPERIMENTS",
                                              usecase="RBS_20000_HL
1",
                                              replay_size=20000,
num_hidden_layers=1)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_20000_
HL_EXPERIMENTS",
                                              usecase="RBS_20000_HL
2",
                                              replay_size=20000,
num_hidden_layers=2)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_20000_
HL_EXPERIMENTS",
                                              usecase="RBS_20000_HL
3",
                                              replay_size=20000,
num_hidden_layers=3)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_20000_
HL_EXPERIMENTS",
                                              usecase="RBS_20000_HL
4",
                                              replay_size=20000,
num_hidden_layers=4)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_20000_
HL_EXPERIMENTS",
                                              usecase="RBS_20000_HL
5",
                                              replay_size=20000,
num_hidden_layers=5)

        experiments.plotExperiment(experiment_name="RBS_20000_HL_EXPERIME
NTS",
                                   title="RBS_20000_HL_EXPERIMENTS
(CORe50 NICv2 - 391)")

    elif args.exp_RBS_30000:
```

```
        print("> Experiment: RBS_30000")

        experiments.runHiddenLayersExperiment(experiment_name="RBS_30000_
HL_EXPERIMENTS",
                                              usecase="RBS_30000_HL
0",
                                              replay_size=30000,
num_hidden_layers=0)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_30000_
HL_EXPERIMENTS",
                                              usecase="RBS_30000_HL
1",
                                              replay_size=30000,
num_hidden_layers=1)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_30000_
HL_EXPERIMENTS",
                                              usecase="RBS_30000_HL
2",
                                              replay_size=30000,
num_hidden_layers=2)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_30000_
HL_EXPERIMENTS",
                                              usecase="RBS_30000_HL
3",
                                              replay_size=30000,
num_hidden_layers=3)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_30000_
HL_EXPERIMENTS",
                                              usecase="RBS_30000_HL
4",
                                              replay_size=30000,
num_hidden_layers=4)

        experiments.runHiddenLayersExperiment(experiment_name="RBS_30000_
HL_EXPERIMENTS",
                                              usecase="RBS_30000_HL
5",
                                              replay_size=30000,
num_hidden_layers=5)

        experiments.plotExperiment(experiment_name="RBS_30000_HL_EXPERIME
NTS",
                                   title="RBS_30000_HL_EXPERIMENTS
(CORe50 NICv2 - 391)")
```

```
    else:
        print("> No valid experiment option provided")
```

2.2 - Experiments class

```
from models import ContinualLearningModel
from data_loader import CORE50
from utils import *
import os
os.environ['TF_GPU_ALLOCATOR'] = 'cuda_malloc_async'
import tensorflow as tf
from matplotlib import pyplot as plt
import json

DATASET_ROOT = 'C:/Users/nikol/Desktop/University/Year-
4/ADE/ThesisCodeExperiments/CORe50-Dataset/core50_128x128/'

class Experiments:

    def __init__(self):
        print("> Experiments Initialized")

    def plotExperiment(self,experiment_name,title):
        min_val = 100
        max_val = 50
        with open('experiments/' + experiment_name + '.json', ) as
json_file:
            usecases = json.load(json_file)
            for usecase in usecases:
                for key, value in usecase.items():
                    plt.plot(value['acc'], label=key)
                    cur_min = min(value['acc'])
                    cur_max = max(value['acc'])
                    if cur_min < min_val:
                        min_val = cur_min
                    if cur_max > max_val:
                        max_val = cur_max

        plt.title(title)
        plt.ylabel("Accuracy (%)")
        plt.xlabel("Encountered Batches")
        plt.yticks(np.arange(round(min_val), round(max_val)+10, 5))
        plt.grid()
        plt.legend(loc='best')
        #plt.show()
        plt.savefig(experiment_name)

    # Function used to store our experiment results in a json file
    def storeExperimentOutputNew(self, experiment_name, usecase_name,
accuracies, losses):
        data = []
```

```
        # Load previously recorded usescases
        with open('experiments/' + experiment_name + '.json', ) as
json_file:
            data = json.load(json_file)

        # Store new usecase
        exp = dict()
        exp[usecase_name] = dict()
        exp[usecase_name]["acc"] = accuracies
        exp[usecase_name]["loss"] = losses
        data.append(exp)

        # Write the updated data back to the file
        with open('experiments/' + experiment_name + '.json', 'w') as
outfile:
            json.dump(data, outfile)

    def print_trainable_status(self, model):
        for layer in model.layers:
            print(f"{layer.name}: {layer.trainable}")

    # New implementation with hidden layers
    def runHiddenLayersExperiment(self, experiment_name, usecase,
replay_size, num_hidden_layers):
        print("> Running Hidden Layers experiment")

        dataset = CORE50(root=DATASET_ROOT, scenario="nicv2_391",
preload=False)
        test_x, test_y = dataset.get_test_set()
        test_x = preprocess(test_x)

        # Building main model
        cl_model = ContinualLearningModel(image_size=128,
name=usecase,replay_buffer=replay_size)
        cl_model.buildBaseHidden(hidden_layers=num_hidden_layers)
        cl_model.buildHeadHidden(sl_units=128,
hidden_layers=num_hidden_layers)
        cl_model.buildCompleteModel()

        #### Used for debugging ####

        # # After building the complete model
        # print("Base model trainable status:")
        # self.print_trainable_status(cl_model.base)

        # print("\nHead model trainable status:")
        # self.print_trainable_status(cl_model.head)
```

```
        # print("\nComplete model trainable status:")
        # self.print_trainable_status(cl_model.model)

        # print("\nComplete model summary:")
        # cl_model.model.summary()

        # # Stop the program
        # exit()

        #### End of debugging ####

        accuracies = []
        losses = []

        # Training, loop over the training incremental batches
        for i, train_batch in enumerate(dataset):
            train_x, train_y = train_batch
            train_x = preprocess(train_x)

            print("----------- batch {0} -------------".format(i))
            print("train_x shape: {}, train_y shape: {}"
                  .format(train_x.shape, train_y.shape))

            if i == 1:
                # Previous values on both: 0.00005
                # A higher learning rate on the head such as 0.001 works
way better especially with the latent replay buffer
                # Increasing the learning rate even more to e.g. 0.01
makes the model unstable since on some batches we have huge loss
                # but the overall accuracy remains more or less the same
                cl_model.model.compile(optimizer=tf.keras.optimizers.SGD(
learning_rate=0.00005),
                                       loss='sparse_categorical_crossentr
opy', metrics=['accuracy'])
                cl_model.head.compile(optimizer=tf.keras.optimizers.SGD(l
earning_rate=0.001),
                                      loss='sparse_categorical_crossentro
py', metrics=['accuracy'])

            # Padding of the first batch. Unsure about this
            if i == 0:
                (train_x, train_y), it_x_ep = pad_data([train_x,
train_y], 128)

            shuffle_in_unison([train_x, train_y], in_place=True)

            print("---------------------------------")
```

```
            features = cl_model.feature_extractor.predict(train_x)

            # Combining the new samples and the replay buffer samples
before training
            print("> Combining new samples and replay buffer samples
before training")
            if i >= 1:
                # Get replay samples
                replay_x = np.array(cl_model.replay_representations_x)
                replay_y = np.array(cl_model.replay_representations_y)

                # Combine new samples with replay samples
                combined_x = np.concatenate((features, replay_x), axis=0)
                combined_y = np.concatenate((train_y, replay_y), axis=0)
            else:
                combined_x = features
                combined_y = train_y

            # Shuffle the combined samples
            shuffle_in_unison([combined_x, combined_y], in_place=True)

            print("combined-x shape: {}, combined-y shape:
{}".format(combined_x.shape, combined_y.shape))

            # Fit the head on the combined samples
            cl_model.head.fit(combined_x, combined_y, epochs=4,
verbose=0)

            #### Used if we want to fit the head on the new samples and
the replay buffer samples separately ####

            # cl_model.head.fit(features, train_y, epochs=4, verbose=0)
            # if i >= 1:
            #     cl_model.replay()
            # cl_model.storeRepresentations(train_x, train_y)

            #### End of the above ####

            # Store the representations of the new samples in the replay
buffer
            cl_model.storeRepresentationsNativeRehearsal(train_x,
train_y, i+1)

            # Evaluate the model on the test set
            loss, acc = cl_model.model.evaluate(test_x, test_y)
            accuracies.append(round(acc*100,1))
            losses.append(loss)
```

```
            print("> ", cl_model.name, " Accuracy: ", acc, " Loss: ",
loss)
            print("---------------------------------")

        # Store results in json file
        self.storeExperimentOutputNew(experiment_name=experiment_name,
                                      usecase_name=usecase,
                                      accuracies=accuracies,
                                      losses=losses)
```

2.3 – Models class

```
import tensorflow as tf
from keras import layers
from keras.regularizers import l2
import numpy as np
import random
import gc

class ContinualLearningModel:

    def __init__(self,image_size=224,name="None",replay_buffer=3000):
        print("> Continual Learning Model Initiated")
        # base = bases.MobileNetV2Base(image_size=224)
        self.image_size = image_size
        self.name = name
        self.replay_representations_x = []
        self.replay_representations_y = []
        self.replay_buffer = replay_buffer # The number of patterns
stored

    # Base of our model. We remove the last N layers of MobileNetV2
    def buildBaseHidden(self,hidden_layers=0):
        baseModel =
tf.keras.applications.MobileNetV2(input_shape=(self.image_size,
self.image_size, 3),
                                                    alpha=1.0,
                                                    include_top=False,
                                                    weights='imagenet')

        # Batch normalization layers replaced with bactch renormalization
layers - Better for CL
        for l in baseModel.layers:
            if ('_BN' in l.name):
                l.renorm = True

        baseModel.trainable = False

        base_model_truncated = tf.keras.Model(inputs=baseModel.input,
outputs=baseModel.layers[-hidden_layers-1].output)
        self.base = base_model_truncated

        inputs = tf.keras.Input(shape=(self.image_size, self.image_size,
3))
        f = inputs
        f_out = self.base(f)
        self.feature_extractor = tf.keras.Model(f, f_out)
```

```
        self.feature_extractor.compile(optimizer=tf.keras.optimizers.SGD(
learning_rate=0.001), loss='categorical_crossentropy',
                                       metrics=['accuracy'])

    # Head of our model. We add N layers to the end of MobileNetV2 + a
dense layer + softmax layer
    def buildHeadHidden(self, sl_units=32, hidden_layers=0):

        baseModel =
tf.keras.applications.MobileNetV2(input_shape=(self.image_size,
self.image_size, 3),
                                                    alpha=1.0,
                                                    include_top=False,
                                                    weights='imagenet')

        self.sl_units = sl_units

        # Create a new head model
        self.head = tf.keras.Sequential()

        # Add the last N layers of MobileNetV2 to the head model
        for i in range(-hidden_layers, 0):
            layer = baseModel.layers[i]
            layer.trainable = True
            self.head.add(layer)

        self.head.add(layers.Flatten(input_shape=(4, 4, 1280)))
        # Removed the dense layer since we add Hidden Layers from
MobileNetV2 now
        self.head.add(layers.Dense(
        units=sl_units,
        activation='relu',
        kernel_regularizer=l2(0.01),
        bias_regularizer=l2(0.01)
        ))

        # Softmax layer (Last layer)
        self.head.add(layers.Dense(
            units=50,  # Number of classes
            activation='softmax',
            kernel_regularizer=l2(0.01),
            bias_regularizer=l2(0.01)),
        )

        # It's worth noting that the compiling of the head and the model
here is probably redundant
        # since we compile the head and model again on the experiments
function with a different learning rate
```

```
        self.head.compile(optimizer=tf.keras.optimizers.SGD(learning_rate
=0.001), loss='sparse_categorical_crossentropy',
                          metrics=['accuracy'])

    def buildCompleteModel(self):
        inputs = tf.keras.Input(shape=(self.image_size, self.image_size,
3))
        x = inputs
        x = self.base(x)
        outputs = self.head(x)
        self.model = tf.keras.Model(inputs, outputs)
        self.model.compile(optimizer=tf.keras.optimizers.SGD(learning_rat
e=0.001), loss='sparse_categorical_crossentropy',
                           metrics=['accuracy'])

    # Refreshing replay memory with new samples and removing old ones if
necessary
    # Fixed bug with patterns being a little less than they should be
    # This bug I think that it is caused because the replacement batch
size is calculated as a percentage of the replay buffer
    # Which is greater than the actual number of patterns stored in the
replay buffer
    #### Old function ####
    def storeRepresentations(self, train_x, train_y):
        # We need to add replay representations and replacement batch
size not train_x
        replacement_batch_size = int(self.replay_buffer * 0.015) # 1.5%
sample replacement
        replay_memory_size = len(self.replay_representations_x) +
replacement_batch_size

        # It's good to know that the first batch has around 3000+ samples
        # The rest are around 300ish
        # The last few batches don't introduce new classes
        # If we want to look in depth each training batch contents we can
check batches filelists in CORe50
        # https://vlomonaco.github.io/core50/index.html#download

        # For testing purposes
        # print("Replay Memory Size: ",replay_memory_size)
        # print("replay representation x:
",len(self.replay_representations_x))
        # print("train x: ",len(train_x))

        # A check to see if the replacement batch size is greater than
the number of samples in a batch
        # This could happen for e.g buffer sizes of 30000 since the batch
size is around 300 samples and 1.5% of 30000 is 450
```

```
        if replacement_batch_size > len(train_x):
            replacement_batch_size = train_x

        # If the replay buffer will overfill we need to remove some old
samples
        if replay_memory_size >= self.replay_buffer:

            x_sample, y_sample = zip(*random.choices(list(zip(train_x,
train_y)), k=replacement_batch_size))
            x_sample = self.feature_extractor.predict(np.array(x_sample))

            # Removing old samples
            patterns_to_delete =
random.sample(range(len(self.replay_representations_x)),
replacement_batch_size)
            for pat in sorted(patterns_to_delete, reverse=True):
                del self.replay_representations_x[pat]
                del self.replay_representations_y[pat]

        # If it will not overfill we just add the new samples
        else:
            x_sample, y_sample = zip(*random.choices(list(zip(train_x,
train_y)), k=replacement_batch_size))
            x_sample = self.feature_extractor.predict(np.array(x_sample))

        # Adding new ones
        for i in range(len(x_sample)):
            self.replay_representations_x.append(x_sample[i])
            self.replay_representations_y.append(y_sample[i])

        gc.collect()
        print("Replay X: ",len(self.replay_representations_x)," Replay Y:
",len(self.replay_representations_y))

    # Different approach where we follow the algorithm as shown in Latent
Replay for Real-Time Continual Learning
    # Performs much better than the previous approach
    # We essentially reduce the replacement batch size the further we go
into our trianing batches
    # https://arxiv.org/abs/1912.01100.pdf
    def storeRepresentationsNativeRehearsal(self, train_x, train_y,
batch_num):

        # The replacement batch size will progressively decrease to keep
a balanced
        # contribution from the different training batches. We don't have
class balancing
        replacement_batch_size = int(self.replay_buffer / batch_num)
```

```
        # print("Replacement Batch Size: ",replacement_batch_size)
        # new_replay_memory_size = len(self.replay_representations_x) +
replacement_batch_size
        # print("Current Replay Memory Size:
",len(self.replay_representations_x))

        # A check to see if the replacement batch size is greater than
the number of samples in a batch
        # This will happen in the first few training batches and
depending on how big the replay buffer is
        if replacement_batch_size > len(train_x):
            replacement_batch_size = len(train_x)

        # If the replay buffer will overfill we need to remove some old
samples
        # Bare in mind with current implementation, the buffer might end
up saving a little bit more than the shown replay buffer size (it's not
an issue)
        if batch_num != 1 and (len(self.replay_representations_x) +
replacement_batch_size) >= self.replay_buffer:

            x_sample, y_sample = zip(*random.choices(list(zip(train_x,
train_y)), k=replacement_batch_size))
            x_sample = self.feature_extractor.predict(np.array(x_sample))

            # Removing old samples
            patterns_to_delete =
random.sample(range(len(self.replay_representations_x)),
replacement_batch_size)
            for pat in sorted(patterns_to_delete, reverse=True):
                del self.replay_representations_x[pat]
                del self.replay_representations_y[pat]

        # If replay buffer will not overfill we just add the new samples
        else:
            x_sample, y_sample = zip(*random.choices(list(zip(train_x,
train_y)), k=replacement_batch_size))
            x_sample = self.feature_extractor.predict(np.array(x_sample))

        # Adding new samples
        for i in range(len(x_sample)):
            self.replay_representations_x.append(x_sample[i])
            self.replay_representations_y.append(y_sample[i])

        # Essential to call gc otherwise we get out of memory errors
        gc.collect()
```

```
        print("Replay X: ",len(self.replay_representations_x)," Replay Y:
",len(self.replay_representations_y))

    # Reduntant function since we mix the replay samples with the new
samples and train them together in experiments function
    def replay(self):

        replay_x = np.array(self.replay_representations_x)
        replay_y = np.array(self.replay_representations_y)

        print("> REPLAYING")
        # Fitting for just 1 epoch
        self.head.fit(replay_x,replay_y,epochs=1,verbose=0)
```

2.4 – Data_Loader class

```
# IMPORTANT NOTE - THIS CODE WAS NOT WRITTEN BY THE AUTHOR OF THE PAPER
SUBJECT TO DOUBLE-BLIND REVIEW
# CODE USED FROM PAPER: https://arxiv.org/abs/1912.01100
#!/usr/bin/env python
# -*- coding: utf-8 -*-

################################################################################
#######
# Copyright (c) 2020. Vincenzo Lomonaco, Gabriele Graffieti,
Lorenzo           #
# Pellegrini, Davide Maltoni. All rights
reserved.                              #
# See the accompanying LICENSE file for
terms.                                  #
#
     #
# Date: 01-04-
2020                                                                  #
# Authors: Vincenzo Lomonaco, Gabriele Graffieti, Lorenzo Pellegrini,
Davide   #
#
Maltoni.
    #
# E-mail:
vincenzo.lomonaco@unibo.it                                             #
# Website:
vincenzolomonaco.com                                                  #
################################################################################
#######

""" Data Loader for the CORe50 Dataset """

# Python 2-3 compatible
from __future__ import print_function
from __future__ import division
from __future__ import absolute_import

# other imports
import numpy as np
import pickle as pkl
import os
import logging
from hashlib import md5
from PIL import Image
```

```
class CORE50(object):
    """ CORe50 Data Loader calss

    Args:
        root (string): Root directory of the dataset where
``core50_128x128``,
            ``paths.pkl``, ``LUP.pkl``, ``labels.pkl``,
``core50_imgs.npz``
            live. For example ``~/data/core50``.
        preload (string, optional): If True data is pre-loaded with look-
up
            tables. RAM usage may be high.
        scenario (string, optional): One of the three scenarios of the
CORe50
            benchmark ``ni``, ``nc``, ``nic``, `nicv2_79`,``nicv2_196``
and
             ``nicv2_391``.
        train (bool, optional): If True, creates the dataset from the
training
            set, otherwise creates from test set.
        cumul (bool, optional): If True the cumulative scenario is
assumed, the
            incremental scenario otherwise. Practically speaking
``cumul=True``
            means that for batch=i also batch=0,...i-1 will be added to
the
            available training data.
        run (int, optional): One of the 10 runs (from 0 to 9) in which
the
            training batch order is changed as in the official benchmark.
        start_batch (int, optional): One of the training incremental
batches
            from 0 to max-batch - 1. Remember that for the ``ni``, ``nc``
and
            ``nic`` we have respectively 8, 9 and 79 incremental batches.
If
            ``train=False`` this parameter will be ignored.
    """

    nbatch = {
        'ni': 8,
        'nc': 9,
        'nic': 79,
        'nicv2_79': 79,
        'nicv2_196': 196,
        'nicv2_391': 391
    }
```

```
    def __init__(self, root='', preload=False, scenario='ni',
cumul=False,
                 run=0, start_batch=0):
        """ Initialize Object """

        self.root = os.path.expanduser(root)
        self.preload = preload
        self.scenario = scenario
        self.cumul = cumul
        self.run = run
        self.batch = start_batch

        if preload:
            print("Loading data...")
            bin_path = os.path.join(root, 'core50_imgs.bin')
            if os.path.exists(bin_path):
                with open(bin_path, 'rb') as f:
                    self.x = np.fromfile(f, dtype=np.uint8) \
                        .reshape(164866, 128, 128, 3)

            else:
                with open(os.path.join(root, 'core50_imgs.npz'), 'rb') as
f:
                    npzfile = np.load(f)
                    self.x = npzfile['x']
                    print("Writing bin for fast reloading...")
                    self.x.tofile(bin_path)

        print("Loading paths...")
        with open(os.path.join(root, 'paths.pkl'), 'rb') as f:
            self.paths = pkl.load(f)

        print("Loading LUP...")
        with open(os.path.join(root, 'LUP.pkl'), 'rb') as f:
            self.LUP = pkl.load(f)

        print("Loading labels...")
        with open(os.path.join(root, 'labels.pkl'), 'rb') as f:
            self.labels = pkl.load(f)

    def __iter__(self):
        return self

    def __next__(self):
        """ Next batch based on the object parameter which can be also
changed
            from the previous iteration. """
```

```
        scen = self.scenario
        run = self.run
        batch = self.batch

        if self.batch == self.nbatch[scen]:
            raise StopIteration

        # Getting the right indexis
        if self.cumul:
            train_idx_list = []
            for i in range(self.batch + 1):
                train_idx_list += self.LUP[scen][run][i]
        else:
            train_idx_list = self.LUP[scen][run][batch]

        # loading data
        if self.preload:
            train_x = np.take(self.x, train_idx_list, axis=0)\
                        .astype(np.float32)
        else:
            print("Loading data...")
            # Getting the actual paths
            train_paths = []
            for idx in train_idx_list:
                train_paths.append(os.path.join(self.root,
self.paths[idx]))
            # loading imgs
            train_x =
self.get_batch_from_paths(train_paths).astype(np.float32)

        # In either case we have already loaded the y
        if self.cumul:
            train_y = []
            for i in range(self.batch + 1):
                train_y += self.labels[scen][run][i]
        else:
            train_y = self.labels[scen][run][batch]

        train_y = np.asarray(train_y, dtype=np.float32)

        # Update state for next iter
        self.batch += 1

        return (train_x, train_y)

    def get_test_set(self, reduced=True):
        """ Return the test set (the same for each inc. batch). """
```

```
        scen = self.scenario
        run = self.run

        test_idx_list = self.LUP[scen][run][-1]

        if self.preload:
            test_x = np.take(self.x, test_idx_list,
axis=0).astype(np.float32)
        else:
            # test paths
            test_paths = []
            for idx in test_idx_list:
                test_paths.append(os.path.join(self.root,
self.paths[idx]))

            # test imgs
            test_x =
self.get_batch_from_paths(test_paths).astype(np.float32)

        test_y = self.labels[scen][run][-1]
        test_y = np.asarray(test_y, dtype=np.float32)

        if reduced:
            # reduce test set 20 substampling
            idx = range(0, test_y.shape[0], 20)
            test_x = np.take(test_x, idx, axis=0)
            test_y = np.take(test_y, idx, axis=0)

        return test_x, test_y

    next = __next__  # python2.x compatibility.

    @staticmethod
    def get_batch_from_paths(paths, compress=False, snap_dir='',
                             on_the_fly=True, verbose=False):
        """ Given a number of abs. paths it returns the numpy array
        of all the images. """

        # Getting root logger
        log = logging.getLogger('mylogger')

        # If we do not process data on the fly we check if the same train
        # filelist has been already processed and saved. If so, we load
it
        # directly. In either case we end up returning x and y, as the
full
        # training set and respective labels.
        num_imgs = len(paths)
```

```
        hexdigest = md5(''.join(paths).encode('utf-8')).hexdigest()
        log.debug("Paths Hex: " + str(hexdigest))
        loaded = False
        x = None
        file_path = None

        if compress:
            file_path = snap_dir + hexdigest + ".npz"
            if os.path.exists(file_path) and not on_the_fly:
                loaded = True
                with open(file_path, 'rb') as f:
                    npzfile = np.load(f)
                    x, y = npzfile['x']
        else:
            x_file_path = snap_dir + hexdigest + "_x.bin"
            if os.path.exists(x_file_path) and not on_the_fly:
                loaded = True
                with open(x_file_path, 'rb') as f:
                    x = np.fromfile(f, dtype=np.uint8) \
                        .reshape(num_imgs, 128, 128, 3)

        # Here we actually load the images.
        if not loaded:
            # Pre-allocate numpy arrays
            x = np.zeros((num_imgs, 128, 128, 3), dtype=np.uint8)

            for i, path in enumerate(paths):
                if verbose:
                    print("\r" + path + " processed: " + str(i + 1),
end='')
                x[i] = np.array(Image.open(path))

            if verbose:
                print()

            if not on_the_fly:
                # Then we save x
                if compress:
                    with open(file_path, 'wb') as g:
                        np.savez_compressed(g, x=x)
                else:
                    x.tofile(snap_dir + hexdigest + "_x.bin")

        assert (x is not None), 'Problems loading data. x is None!'

        return x
```

```
if __name__ == "__main__":

    # Create the dataset object for example with the "NIC_v2 - 79
benchmark"
    # and assuming the core50 location in ~/core50/128x128/
    dataset = CORE50(root='/home/admin/Ior50N/128', scenario="nicv2_79")

    # Get the fixed test set
    test_x, test_y = dataset.get_test_set()

    # loop over the training incremental batches
    for i, train_batch in enumerate(dataset):
        # WARNING train_batch is NOT a mini-batch, but one incremental
batch!
        # You can later train with SGD indexing train_x and train_y
properly.
        train_x, train_y = train_batch

        print("----------- batch {0} -------------".format(i))
        print("train_x shape: {}, train_y shape: {}"
              .format(train_x.shape, train_y.shape))

        # use the data
        pass
```

2.5 – Utils class

```
import numpy as np

def preprocess(images, scale=True, norm=False):
    if scale:
        # convert to float in [0, 1]
        images = images / 255

    if norm:
        # normalize
        images[:, :, :, 0] = ((images[:, :, :, 0] - 0.485) / 0.229)
        images[:, :, :, 1] = ((images[:, :, :, 1] - 0.456) / 0.224)
        images[:, :, :, 2] = ((images[:, :, :, 2] - 0.406) / 0.225)

    return images

def shuffle_in_unison(dataset, seed=None, in_place=False):
    """
    Shuffle two (or more) list in unison. It's important to shuffle the
images
    and the labels maintaining their correspondence.

        Args:
            dataset (dict): list of shuffle with the same order.
            seed (int): set of fixed Cifar parameters.
            in_place (bool): if we want to shuffle the same data or we
want
                             to return a new shuffled dataset.
        Returns:
            list: train and test sets composed of images and labels, if
in_place
                  is set to False.
    """

    if seed:
        np.random.seed(seed)
    rng_state = np.random.get_state()
    new_dataset = []
    for x in dataset:
        if in_place:
            np.random.shuffle(x)
        else:
            new_dataset.append(np.random.permutation(x))
        np.random.set_state(rng_state)

    if not in_place:
        return new_dataset
```

```
def pad_data(dataset, mb_size):
    """
    Padding all the matrices contained in dataset to suit the mini-batch
    size. We assume they have the same shape.

        Args:
            dataset (str): sets to pad to reach a multile of mb_size.
            mb_size (int): mini-batch size.
        Returns:
            list: padded data sets
            int: number of iterations needed to cover the entire training
set
                 with mb_size mini-batches.
    """

    num_set = len(dataset)
    x = dataset[0]
    # computing test_iters
    n_missing = x.shape[0] % mb_size
    if n_missing > 0:
        surplus = 1
    else:
        surplus = 0
    it = x.shape[0] // mb_size + surplus

    # padding data to fix batch dimentions
    if n_missing > 0:
        n_to_add = mb_size - n_missing
        for i, data in enumerate(dataset):
            dataset[i] = np.concatenate((data[:n_to_add], data))
    if num_set == 1:
        dataset = dataset[0]

    return dataset, it
```

Image Sources

Figure 2.1: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1

Figure 2.2: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

Figure 2.3: https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092

Figure 2.4: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

Figure 2.5: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

Figure 2.6: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

Figure 2.7: https://medium.com/analytics-vidhya/image-classification-with-mobilenet-cc6fbb2cd470

Figure 2.8: https://medium.com/analytics-vidhya/efficientdet-scalable-and-efficient-object-detection-384a5df9011a

Figure 2.9: https://programmathically.com/an-introduction-to-neural-network-loss-functions/

Figure 2.10: https://programmathically.com/an-introduction-to-neural-network-loss-functions/

Figure 2.11: Taken from laboratory lectures of CS442 course at University of Cyprus

Figure 2.12: Taken from laboratory lectures of CS442 course at University of Cyprus

Figure 2.13: Taken from laboratory lectures of CS442 course at University of Cyprus

Figure 2.14: Taken from laboratory lectures of CS442 course at University of Cyprus

Figure 2.15 – 2.16: Overfitting and Underfitting in Machine Learning - Javatpoint

Figure 2.17 – 2.20: Continual Learning: On Machines that can Learn Continually - Continual Learning Course (continualai.org)

Figure 3.1 – 3.3: Example on-device model personalization with TensorFlow Lite — The TensorFlow Blog

Figure 3.4: https://arxiv.org/abs/2105.01946.pdf

Figure 3.5: examples/README.md at master · tensorflow/examples · GitHub

Figure 3.6 – 3.16: Snapshots from the application that was made as part of this thesis

Figure 4.1 – 4.4: [2105.01946] Continual Learning on the Edge with TensorFlow Lite (arxiv.org)