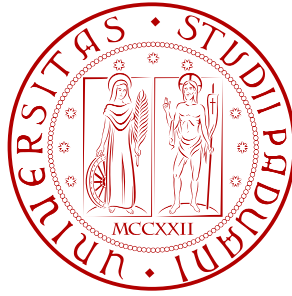


Università degli Studi di Padova

DEPARTMENT OF INFORMATION ENGINEERING

Master Degree in Control Systems Engineering

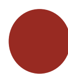
Master Thesis

Latent Replay for Continual Learning on Edge Devices with Efficient Architectures

Candidate:
Matteo Tremonti
2021863

Supervisor:
Prof. Gian Antonio Susto

Co-Supervisor:
Dott. Davide Dalle Pezze



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Academic Year 2022-2023
December 14, 2023

Abstract

Due to the limited computational capabilities, low memory and limited energy budget, training deep neural networks on edge devices is very challenging. On the other hand, privacy and data limitations, lack of network connection, as well as the need for rapid model adaptation, make real-time training on the device crucial.

Standard artificial neural networks suffer from the issue of catastrophic forgetting, making learning difficult. Continual learning shifts this paradigm to networks that can continuously accumulate knowledge on different tasks without the need to retrain from scratch.

In this work, a Continual Learning technique called Latent Replay is employed, in which the activations of intermediate layers are stored and used to integrate training data for each new task. This approach reduces the computation time and memory required, facilitating training on the limited resources of edge devices. In addition, a new efficient architecture, known as PhiNets, was used for the first time in the context of Continual Learning.

An intensive study was conducted to compare PhiNets with efficient architectures already tested in this context, such as MobileNet. Several metrics were considered, such as computation time, inference time, memory used, and accuracy. In addition, the variation of these metrics based on factors such as the layer at which Latent Replay is applied was analysed. Tests were performed on well-known computer vision datasets, evaluating them as a stream of classes.

Acknowledgements

I would like to express my gratitude to my supervisors, Prof. *Gian Antonio Susto* and Dott. *Davide Dalle Pezze*, for their valuable advice and guidance during this work. In addition, I would like to thank Dott. *Francesco Paissan* of the Fondazione Bruno Kessler for his help and collaboration in developing this project.

I am grateful to all the people who have assisted me during these years. First my family, *Giovanna*, *Lorenzo*, *Martina* and *Francesco* who have always been present in times of need; my lifelong friends *Erik*, *Alessandro*, *Sara*, *Omar*, *Enrico*, *Nicola*, *Marco*, *Ludovica* and *Edoardo* for the wonderful moments and support outside the university. A big thanks to my friend and colleague *Antonio* for the many laughs and discussions during these years.

Finally, thanks to all the friends I could not mention in this short acknowledgment section, who have helped and supported me during my university career.

Thank you all for being an important part of this academic journey.

Padua, December 2023 Matteo Tremonti

Contents

Abstract iii

Acknowledgements v

Contents vii

List of Figures ix

List of Tables xi

Acronyms xiii

1 Introduction 1

2 Continual Learning 5
- 2.1 Motivation 5
- 2.2 Continual Learning 6
  - 2.2.1 Formal Definition 6
  - 2.2.2 Data Distribution Changes 8
  - 2.2.3 Continual Learning Scenarios 10
- 2.3 Continual Learning Strategies 13
  - 2.3.1 Upper Bound and Lower Bound 13
  - 2.3.2 Regularization Strategies 15
  - 2.3.3 Architecture Strategies 16
  - 2.3.4 Rehearsal Strategies 17
- 2.4 Latent Replay 18
  - 2.4.1 Motivation 18
  - 2.4.2 Latent Replay Strategies 19
  - 2.4.3 Formal Definition 20

3 Benchmarks and Metrics for CL 23
3.1 Classic Datasets . . . 23
3.2 Continual Learning Benchmark . . . 24
3.2.1 Class-Incremental Learning Benchmark . . . 25
3.2.2 Domain-Incremental Learning Benchmark . . . 26
3.2.3 Dataset for Continual Learning . . . 27
3.3 Metrics . . . 27
3.3.1 Performance metrics . . . 28
3.3.2 Efficiency metrics . . . 30

4 Efficient Architecture for Edge Devices 33
4.1 Introduction . . . 33
4.2 MobileNetV1 . . . 34
4.3 MobileNetV2 . . . 37
4.4 PhiNet . . . 39
4.5 Comparative Analysis . . . 41

5 Results 43
5.1 Experimental Setup . . . 43
5.1.1 Methods . . . 44
5.1.2 Models . . . 45
5.1.3 Benchmarks . . . 47
5.1.4 Evaluation Metrics . . . 48
5.1.5 Learning Details . . . 48
5.2 Experimental Results . . . 50
5.2.1 Data Efficiency . . . 50
5.2.2 Strategies Evaluation . . . 57
5.2.3 Memory Evaluation . . . 65
5.2.4 Memory-Performance Trade-off Evaluation . . . 67

6 Conclusion 75
6.1 Conclusion . . . 75
6.2 Future Research . . . 76

A Kendall Tau Distance 77

B Additional Results 79
B.1 Optimized PhiNet Performances . . . 79
B.2 Similarity Measure . . . 81

Bibliography 83

List of Figures

2.1 Continual Learning schema. 7
2.2 Data distribution shifts. 9
2.3 Data distribution temporal shifts. 10
2.4 Continual Learning scenarios. 11
2.5 Split MNIST according to the three CL scenarios. 12
2.6 Taxonomy of CL approaches. 14
2.7 Experience Replay strategy. 18
2.8 Latent Replay Strategy. 21

3.1 CIFAR-10 Dataset. 24
3.2 Split CIFAR-10 Benchmark. 25
3.3 MNIST Benchmarks. 26
3.4 CIFAR-10 benchmark for DIL scenario. 27
3.5 CORe50 dataset. 28

4.1 Depthwise Separable Convolution. 36
4.2 PhiNet convolutional block. 40

5.1 Inference time comparison between the original and the optimized PhiNet. 46
5.2 Impact of the weight decay in PhiNet A. 48
5.3 Data efficiency on CIFAR-10. 52
5.4 Data efficiency on CORe50. 53
5.5 Percentage change in CIFAR-10 performance across different model layers as the number of memory elements varies. 55
5.6 Percentage change in CORe50 performance across different model layers as the number of memory elements varies. 56
5.7 Accuracy results on CIFAR-10 of Fine-Tuning, Multi-Task, Experience Replay and Latent Replay. 58
5.8 Accuracy results on CORe50 of Fine-Tuning, Multi-Task, Experience Replay and Latent Replay. 59

5.9 Similarity to the ideal case for PhiNet A and MobileNetV2. . . 61
5.10 Comparison of latent activation size and MAC of the selected layer between different models. 66
5.11 Magnified comparison of latent activation size and MAC of the selected layer between different models. 66
5.12 Performance comparison on CIFAR-10 for fixed memory size. . 68
5.13 Performance comparison on CORe50 for fixed memory size. . . 69
5.14 Evaluation of the different architectures on CIFAR-10. 71
5.15 Evaluation of the different architectures on CORe50. 72

B.1 Inference time comparison between the original and the optimized PhiNet changing the α parameters. 79
B.2 Inference time comparison between the original and the optimized PhiNet changing the β parameters. 80
B.3 Inference time comparison between the original and the optimized PhiNet changing the t_0 parameters. 80
B.4 Similarity measures on CIFAR-10 for the different models. . . 81
B.5 Similarity measures on CORe50 for the different models. . . . 82

List of Tables

2.1 Continual Learning scenarios. 10

3.1 Accuracy matrix. 29

4.1 MobileNetV1 Architecture. 35
4.2 MobileNetV2 Architecture. 38
4.3 Inverted Residual and Linear Bottleneck layer. 38
4.4 PhiNet Convolutional Block. 40
4.5 PhiNet Architecture. 41
4.6 Resource usage of different Efficient Architectures. 42

5.1 Size of memory used in the experiments. 45
5.2 Comparison of the original and the optimized PhiNet. 46
5.3 Performance on ImageNet for benchmarked models. 47
5.4 Experiments benchmark for the class-incremental learning scenario. 47
5.5 Number of epochs and mini-batch size for each strategy. . . . 49
5.6 Final accuracy on CIFAR-10 for each strategy. 60
5.7 Final accuracy on CORe50 for each strategy. 60
5.8 Computation, storage, and accuracy trade-off with replay strategies for all the models on CIFAR-10. 63
5.9 Computation, storage, and accuracy trade-off with replay strategies for all the models on CORe50. 64
5.10 Best Performances for each model on CIFAR-10 and CORe50. 73

Acronyms

A

ACC Average Accuracy.
AD Anomaly Detection.
AI Artificial Intelligence.
ANN Artificial Neural Network.

B

BWT Backward Transfer.
BWT$^+$ Positive Backward Transfer.

C

CE Computational Efficiency.
CF Catastrophic Forgetting.
CIL Class-Incremental learning.
CL Continual Learning.
CV Computer Vision.

D

DIL Domain-Incremental learning.
DL Deep Learning.
DNN Deep Neural Network.

E

ER Experience Replay.
EWC Elastic Weight Consolidation.

F

F Forgetting.
FIM Fisher Information Matrix.

FP32 32-bit floating point.
FWT Forward Transfer.

G

GAN Generative Adversarial Network.
GEM Gradient Episodic Memory.

I

i.i.d. independent and identically distributed.
IoT Internet of Things.

L

LR Latent Replay.
LWF Learning Without Forgetting.

M

MAC Multiply-Accumulate.
ML Machine Learning.
MS Model Size Efficiency.

N

NLP Natural Language Processing.
NN Neural Network.

P

PNN Progressive Neural Networks.

R

REM Remembering.

S

SGD Stochastic Gradient Descent.
SI Synaptic Intelligence.
SSS Samples Storage Size Efficiency.

T

TIL Task-Incremental learning.

V

VAE Variational Auto-Encoders.

Chapter 1

Introduction

In recent years, Deep Learning (DL) models have become ubiquitous in various applications, improving the state of the art in several fields such as Computer Vision (CV), Natural Language Processing (NLP), and Audio. Generally, these models require huge computational power to be run and often utilize cloud computing for model inference.

In the Internet of Things (IoT) era, moving these models from the cloud to the edge has become critical, as deployment on edge devices reduces latency, improves privacy, and enhances security. However, bringing Artificial Intelligence (AI) to devices with limited computational power is a challenging task. Recently, several models, known as efficient architectures, have been developed to achieve high performance while minimizing the needed computational resources, making them suitable for edge devices. However, implementing Machine Learning (ML) models on the edge introduces another significant challenge: the need for continuous adaptability of models.

Edge applications are subject to data distribution shifts related to environmental changes, such as lighting conditions. Additionally, the adaptability of applications to new scenarios, such as the incorporation of new classes, is crucial. This introduces another critical challenge for Deep Neural Networks (DNN): the problem of Catastrophic Forgetting (CF). This refers to neural networks' tendency to forget previously acquired knowledge when exposed to newly available data, limiting their ability to continuously learn and adapt to new tasks.

The naive method involves retraining the model from scratch using the entirety of the available data. However, this approach is generally impractical, especially when dealing with edge devices equipped with limited computational resources. On the other hand, if one solely depends on new data to update the model, it would quickly forget all previously acquired knowledge.

Recently a new machine learning paradigm called Continual Learning

(CL) has been introduced. It aims to mitigate the problem of catastrophic forgetting by allowing a model to adapt to new data that arrive in the form of a sequence of tasks. By exploiting Continual Learning techniques, we can not only mitigate the risk of forgetting but also reduce the complexity of the training process. In other words, CL allows models to grow their knowledge over time with minimal computational cost. Several families of CL strategies have been proposed in recent years, such as regularization methods, rehearsal-based methods, and architecture-based methods.

Experience Replay (ER) is an effective rehearsal-based strategy. It stores representative samples of a task and replays them during training of a new task to retain past knowledge. Recently, novel replay techniques have emerged to optimize resource utilization. The Latent Replay (LR) strategy aims to store and replay past data with minimal memory and computational power, moving towards edge applications.

In this thesis, we will use for the first time in the realm of Continual Learning, a novel efficient architecture called PhiNet. It has demonstrated the potential to enhance well-known efficient architectures such as MobileNet and EfficientNet. In particular, we will employ the Latent Replay strategy to compare the performance of PhiNet, MobileNetV1, and MobileNetV2 in a resource-constrained scenario. We will conduct several experiments using two well-known datasets for image classification tasks in the context of Continual Learning: CIFAR-10 and CORe50.

Initially, we aim to examine the impact on the performance of the choice of the layer at which to apply replay and the number of samples stored in the replay memory. We will conduct an empirical analysis to identify factors contributing to model forgetting, demonstrating that networks with greater expressiveness require more data in memory to preserve knowledge of previous tasks than less complex models. Additionally, we will show the effectiveness of Latent Replay compared to the Experience Replay strategy. Finally, we will compare the effectiveness of PhiNet and MobileNet in real-world scenarios by limiting the replay memory size to a few MB. Demonstrating the superiority of PhiNet in situations with limited replay memory size and computational power.

The thesis is structured as follows. Chapter 2 introduces the idea of Continual Learning, providing the necessary theory and presenting related work. In particular, it thoroughly examines the challenges of deep learning and how Continual Learning strategies attempt to solve them. Different CL scenarios will be presented, along with an overview of CL families, introducing some of the most well-known CL approaches for each. Finally, the Latent Replay strategy will be presented. Chapter 3 will present well-known DL datasets and how they can be adapted to be representative of different CL scenarios.

In addition, we will report datasets specifically designed for CL. Moreover, metrics frequently used in the field to evaluate performances and efficiency of CL strategy will be presented. Chapter 4 will introduce the theory and the state-of-the-art of the most recent efficient architectures, motivating the efficacy of these models and emphasizing their differences. In Chapter 5 will discuss the results of the thesis work. By first demonstrating the effectiveness of the Latent Replay strategy. Subsequently, the performance of efficient architectures will be compared. Finally, Chapter 6 provides the thesis conclusions and proposes potential future improvements.

Chapter 2

Continual Learning

This chapter introduces the concept of Continual Learning (CL), which is essential for understanding the terminology and techniques presented in the rest of the thesis.

Section 2.1 discusses the problems associated with classical deep learning (DL) techniques. Section 2.2 provides a formal definition of CL and introduces the problem of data distribution shift and the different Continual Learning scenarios. Subsequently, Section 2.3 will present some of the current state-of-the-art methods in the Continual Learning field. Finally, Section 2.4 formally introduces the Latent Replay (LR) approach, which will be used throughout the rest of the thesis.

2.1 Motivation

Deep Learning (DL) is a researching field that focuses on developing learning algorithms that learn to optimize an objective function on training data. This is achieved through the use of Deep Neural Networks (DNNs), which consist of multiple layers of interconnected neurons. Each layer builds upon the previous one to improve and refine predictions.

Modern deep learning provides a very powerful framework for supervised learning. By adding more layers and more units within a layer, a deep neural network can represent functions of increasing complexity [11]. In recent years, state-of-the-art DNN models have been reported to achieve impressive performance on a wide variety of individual tasks, such as speech recognition, image classification, machine translation, face recognition, gaming, and many other applications. Although these results are impressive, DL requires large amounts of data to train the models and it requires all data to be available at the beginning of the learning process. Furthermore, it makes the strong

assumption that the data are i.i.d. during training. Unfortunately, in many real-world scenarios, it is difficult to maintain this assumption. In addition, it is very likely that a dataset will be integrated with new data over time, making it impossible to have all the data at the beginning [25]. To overcome these problems, the simplest solution would be to fine-tune the models on the new available data. In DL, fine-tuning is an approach that involves further training a pre-trained model on new tasks or data. However, when trained on new data, Artificial Neural Networks (ANNs) forget most of the knowledge from the previous task. This effect is known as Catastrophic Forgetting (CF) [7] and it can be explained by observing the learning process of a neural network. Indeed, during the training phase, the network parameters are updated through an optimization algorithms. The updates are computed with respect to the loss function on the current task, with the aim of minimising the error on the new available data. Thus, the optimization process focuses on the current task and does not take into account the preservation of previously learned information. A simple solution could be to train the neural network from scratch with all the data whenever new data becomes available. Nevertheless, this approach may not be feasible due to the computational power required. Indeed, training large neural networks can take weeks, and retraining every time new data becomes available could be expensive and time-consuming. This is especially true for situations involving data streams. Therefore, being able to improve a pre-trained network with only the new available data would be much more efficient. Furthermore, training with all data may not be possible not only for reasons of efficiency, but also due to legal, security or privacy obligations on previous data.

In recent years, several methods have been proposed to alleviate catastrophic forgetting. One area of research that seeks a solution to the CF problem is known as Continual Learning (CL). CL explores approaches to enable a DL model to learn in an evolving environment and continuously learn to adapt to novel situations and remember previously learned solutions for known situations. CL is directly related to the Plasticity-Stability Dilemma, where plasticity refers to the ability to adapt new knowledge, while stability refers to the retention of previously learned information [7, 8].

2.2 Continual Learning

2.2.1 Formal Definition

A formal definition of Continual Learning can be derived from prior research works [7, 8, 31]. This approach ensures the maintenance of clarity and consis-

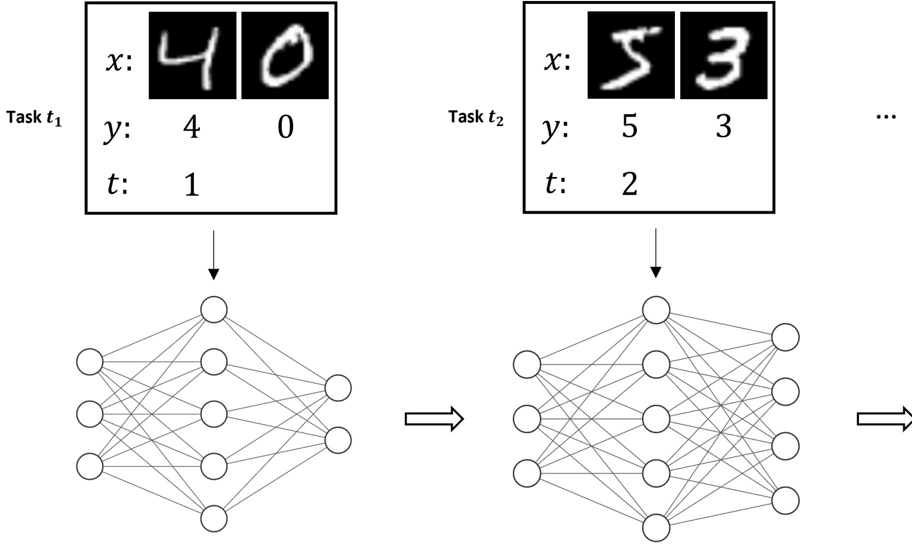


Figure 2.1: Continual Learning schema. In the image is reported a stream of two tasks t_i with the corresponding triplet (x_i, y_i, t_i), with $i = 1, 2$.

tency in the context of CL, allowing for comprehension of metrics as defined in literature.

The starting point is to define a task t_i, which corresponds to a dataset D_{t_i} represented by a triplet (x_i, y_i, t_i), where $x_i \in \mathcal{X}_{t_i}$ denotes the feature vector, $y_i \in \mathcal{Y}_{t_i}$ represents the target vector, and $t_i \in \mathcal{T}$ is the task descriptor. In the simplest case, the collection of tasks descriptor $\mathcal{T} = \{t_1, \ldots, t_T\}$ is a set of integers enumerating the different tasks, $t_i = i \in \mathbb{Z}$. More generally, the task descriptor t_i could be a structured object.

In Continual Learning, we focus on a stream of tasks $t_i \in \mathcal{T}$ of length T

$$(x_1, y_1, t_1), \ldots, (x_i, y_i, t_i), \ldots, (x_T, y_T, t_T) \tag{2.1}$$

where we assume that every triplet is locally i.i.d., i.e., $(x_i, y_i) \overset{\text{iid}}{\sim} P_t(X, Y)$.

The objective is to train a predictor f on the stream of tasks (2.1). It has to accurately predict the corresponding target vector y when presented with input x from past, present, or future tasks. It should be noted that the relationship between task descriptor t and the predictor f may vary. In fact, the task descriptor may or may not be present during training and testing phases. The different relationships will be formally defined in Section 2.2.3, where the scenarios of Continual Learning will be defined. In Figure 2.1 is reported a graphical representation of the Continual Learning settings.

2.2.2 Data Distribution Changes

Classic Machine Learning (ML) models learn the underlying data distribution from the training data, with the aim of leverage this learned distribution to generate accurate predictions for unseen data. Therefore, the data used for testing a model during development should be representative of unseen data, and the performance of the model on the test data should give an idea of how well the model generalize [19]. During the development is generally assumed that the train and test data are drawn i.i.d. from the same stationary probability distribution. However, this assumption does not always hold in the real world. Firstly, the distribution of real-world data is unlikely to be the same as that of the training data. Secondly, the real-world is non-stationary, which could result in data distribution shifts that may cause failures in ML systems.

It is possible to characterised different types of data distribution drift, and it is unlikely that a specific approach will work well for all of them. Therefore, one can characterise CL algorithms based on their capability to learn under certain types of distributional drift [26].

In supervised learning, the training data is a set of samples extracted from a joint probability distribution $P(X, Y)$, where X is the input space and Y is the output space. The purpose of ML is to model the probability $P(Y|X)$, i.e., the conditional probability of an output given an input. The joint probability distribution might be decomposed as:

1. $P(X, Y) = P(Y|X)P(X)$
2. $P(X, Y) = P(X|Y)P(Y)$

where $P(X)$ and $P(Y)$ represent respectively the input and output probability distribution. Three types of data distribution shifts can be defined from this:

1. **Covariate shift** or **Domain drift** : occurs when there is only a shift in the input distribution $P(X)$, i.e. the distribution of the input changes, but the conditional probability of an output given an input remains the same. It could happen because of biases during the data selection process. As an example, if you train a model on images taken in one lighting condition and then deploy it in a different lighting condition, the model's performance may suffer due to the change in the input distribution.

2. **Label shift** or **Virtual Concept drift** : occurs when there is only a shift in the output distribution $P(Y)$, i.e., the distribution of the

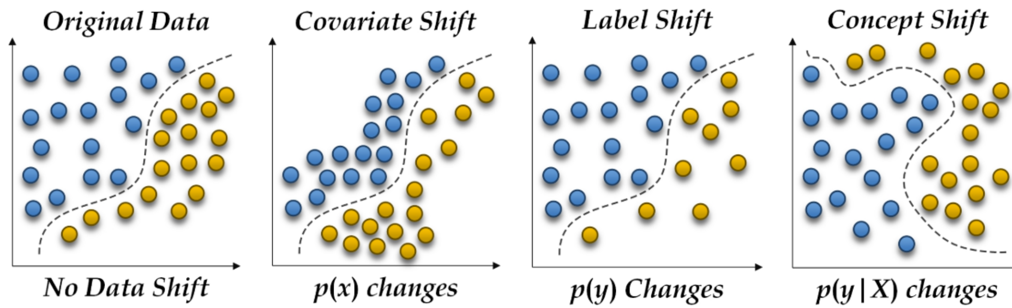


Figure 2.2: Different type of data distribution shifts. From left to right: *Original Data Distribution*, *Covariate Shift*, *Label Shift* and *Concept Shift*. Figure adapted from [1].

output changes, but for a given output, the input distribution remains the same. As an example, suppose you have a model that is trained to identify the species of bird [53]. Now, suppose you want to identify birds in San Francisco and New York. While the probability $P(Y)$ of observing a snowy owl may differ, snowy owls should look similar in New York and San Francisco and thus, the input distribution given the output $P(X|Y)$ will remain unchanged.

3. **Concept drift** : occurs when there is only a shift in the conditional distribution of the output given an input $P(Y|X)$ and the input distribution remains the same, i.e., same input, different output. As an example, consider a house price prediction model where the house's area is an input parameter. Suppose that the house price was \$200 000 before 2020 but fallen to \$150 000 after 2020. Despite the house's features having remained the same, the conditional distribution of the house price, given its features, has changed.

In Figure 2.2 is reported a graphical representation of the different data distribution shift compared to the original data distribution.

Another possible categorization of the data distribution shift is to subdivide the shift as it changes over time: [7, 10]:

1. **Sudden drift**: occurs when there is a sudden change in the data distribution, i.e. the data distribution suddenly changes to a different one.

2. **Incremental drift**: occurs when there is a series of small changes in the data distribution that occur over time, i.e., it consists in many intermediate data distribution over time.

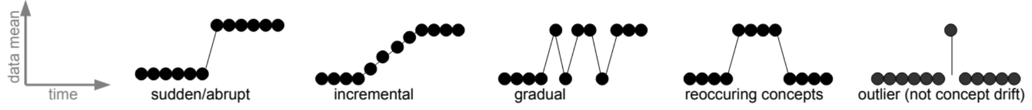


Figure 2.3: Different type of temporal shifts. From left to right: *Sudden Drift*, *Incremental Drift*, *Gradual Drift*, *Recurring Concepts Drift* and *Outlier Drift*. Figure adapted from [10].

Scenario	Description	Mapping to Learn
Task-IL	*Sequentially learn to solve a number of distinct tasks.*	$f : \mathcal{X} \times \mathcal{T} \rightarrow \mathcal{Y}$
Domain-IL	*Learn to solve the same problem in different tasks.*	$f : \mathcal{X} \rightarrow \mathcal{Y}$
Class-IL	*Discriminate between incrementally observed classes.*	$f : \mathcal{X} \rightarrow \mathcal{Y} \times \mathcal{T}$

Table 2.1: Overview of the three Continual Learning scenarios. $\mathcal{X}$ is the input space, $\mathcal{Y}$ is the output space and $\mathcal{T}$ is the task descriptor space. Table adapted from [49].

3. **Gradual drift**: occurs when there is a slow and gradual change in the data distribution over time, the change is smaller and over a long period.

4. **Reoccurring concepts drift**: occurs when there is a periodic change in the data distribution at regular intervals.

5. **Outlier drift**: occurs when there is a once-off random deviation, i.e., a temporary change in the data distribution that returns to its original distribution.

The Figure 2.3 shows a visual representation of the mentioned time distributions shift. Typically, in Continual Learning, it is possible to observe the sudden drift since the tasks are divided very precisely, without ambiguity [7].

2.2.3 Continual Learning Scenarios

In the context of Continual Learning, a well-known framework proposed in [48] categorises the possible scenarios into three categories: (1) Task Incremental Learning (TIL), (2) Domain Incremental Learning (DIL) and

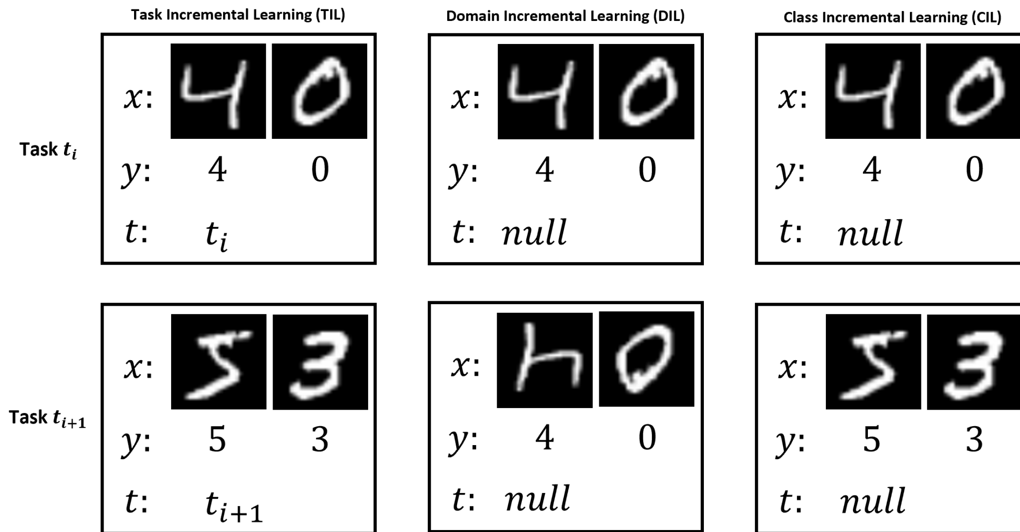


Figure 2.4: Overview of the three Continual Learning scenarios.

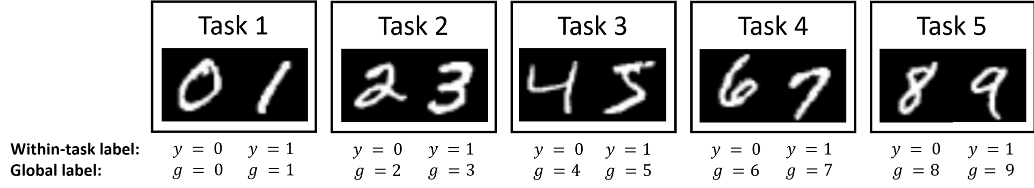
(3) Class Incremental Learning (CIL) (Table 2.1). These scenarios assume that there are clear and well-defined boundaries between the tasks to be learned during training, i.e., a sudden change. If the transition between tasks is gradual or incremental, the aforementioned scenarios are no longer applicable. The main difference between the three scenarios lies in the manner in which task label information is exploited throughout the training and testing phases. Figure 2.4 reports a visual example of the differences between the three Continual Learning scenarios.

Task Incremental Learning (TIL)

Task Incremental Learning (TIL) refers to a scenario where the model gradually learns new tasks over time and during both training and testing, it is always clear which task is being performed. As a result, the model always receives the task label in all the phases. The aim is to train a predictor $f : \mathcal{X} \times \mathcal{T} \rightarrow \mathcal{Y}$ that predict the label y associated with the test pair (x, t).

In this scenario, a common architecture includes a "multi-head" output layer, in which each task has its own output units and the rest of the network is shared between the tasks [48].

Figure 2.5 reports an example with the Split MNIST dataset [52], a modified version of the MNSIT dataset [24], where the images in the dataset are divided into 5 tasks (more details in Chapter 3). In this scenario, given in input the image and task label, the model must predict to which class the input image belongs (i.e., class 0 or 1).



(a) *Split MNIST with 5 tasks example.*

Scenario	Input	Output	Description
Task-IL	(x, t)	y	*Choice between two digits of same task.*
Domain-IL	x	y	*Is the digit odd or even?*
Class-IL	x	g	*Choice between all ten digits.*

(b) *Model expected input/output according to each CL scenario.*

Figure 2.5: Split MNIST datatset with 5 tasks according to the three Continual Learning scenarios. Image adapted from [49].

Domain Incremental Learning (DIL)

Domain Incremental Learning (DIL) refers to a scenario where the model learns and adapts to new data from the same domain over time. The task structure always remains the same, but the input distribution changes. In such cases, the task label is not provided during testing. As a result, the model does not know to which task a sample belongs. However, identifying the task is not necessary, because each task has the same possible outputs. The aim is to train a predictor $f : \mathcal{X} \rightarrow \mathcal{Y}$ that must predict a label y associated with the test input x.

In this scenario, in the example of Figure 2.5, the model has to predict whether the digits belong to the same class (e.g., *odd* or *even*).

Class Incremental Learning (CIL)

Class Incremental Learning (CIL) refers to a scenario where the model must learn and adapt to new data classes gradually. The model needs to solve each task and correctly identify which task a sample is from. Identifying the task is necessary to solve the problem, as it determines which possible classes the current sample belongs to. The aim is to train a predictor $f : \mathcal{X} \rightarrow \mathcal{Y} \times \mathcal{T}$ that must predict a label y and the task t associated with the test input x.

In the CIL scenario in Figure 2.5, the model has to predict to which class the image belongs.

2.3 Continual Learning Strategies

In recent years, many Continual Learning strategies have been proposed to mitigate the problem of catastrophic forgetting. These strategies are usually categorized based on how task information is stored and exploited during the learning phase. Typically, three main groups of approaches are identified in literature [7, 8, 28, 49]:

- **Regularization-based strategies**: involve the introduction of a regularization term in the loss function. This term ensures that the significant weights that retain knowledge from previous tasks are consolidated during training of the new task. These strategies avoid input storage, prioritizing privacy, and reduce memory requirements [8].
- **Architecture-based strategies**: use different architectures or layers to learn new tasks without interfering with old ones. Alternatively, they freeze the weights of previous tasks in the network to prevent forgetting. The main disadvantage of this category is that it typically requires task label information.
- **Rehearsal-based strategies**: involve storing samples of encountered tasks to maintain a memory of the past in the model. Alternatively, generative models are used to generate samples as a memory of encountered tasks. These samples are used then to strengthen the connections of past tasks in the model during training of new tasks.

Figure 2.6 shows a non-comprehensive set of common CL strategies, which will be presented in the following sections.

2.3.1 Upper Bound and Lower Bound

Continual learning strategies are evaluated by comparing performance against upper and lower bounds. This allows to understand the gap between the actual strategy and the ideal and worst case scenarios. It is possible to identify three the upper bounds: the Cumulative, Multi-Task, and Single Model strategies. While for the lower bound, the Fine-Tuning strategy is considered.

Cumulative Strategy

The cumulative strategy assumes a stream of tasks to be learned and no memory or computational limits are taken into account during training. When a new task becomes available, fine tuning is performed using all the data from the previous task and the new task. Another variant of the cumulative

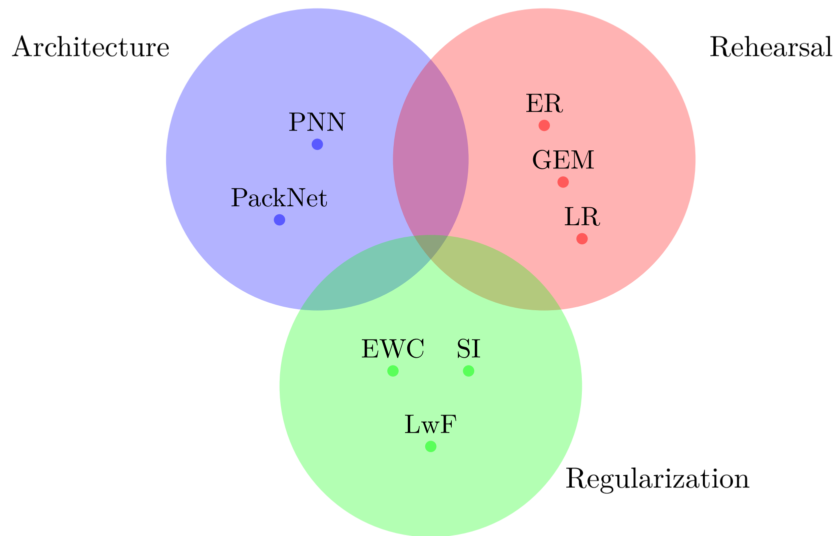


Figure 2.6: Taxonomy of CL approaches.

strategy is to start training from scratch with available data, re-initialising the network weights [28].

Multi-Task Strategy

The multi-task strategy assumes that all the data are available at the beginning of the training phase, and it uses data from all tasks to train the model. As a result, shared information and dependencies are used to improve the performance of all tasks and allow better generalization. The main difference between this strategy and the cumulative strategy is that the multi-task strategy is trained immediately with all the data from all the tasks, whereas the cumulative strategy is trained with all the data only when the last task is reached.

Single Model Strategy

The single-model strategy trains one model for each task it encounters. Each model is specifically optimized for the task, and this strategy represents the ideal case. This approach can lead to superior task-specific performance, although it is less efficient in terms of resource utilization.

Fine-Tuning Strategy

The fine-tuning strategy is the naive method of training the model only with data from the current task, possibly using basic regularisation techniques.

However, significant variation in data distribution across multiple tasks can result in catastrophic forgetting, as previously mentioned.

2.3.2 Regularization Strategies

To prevent catastrophic forgetting, regularization strategies add terms to the loss function that consolidate the weights of the previous task. Two main approaches can be distinguished: one constrains the weights of the neural network by estimating their importance for past tasks, such as Elastic Weight Consolidation (EWC) [22] and Synaptic Intelligence (SI) [52]. Other approaches exploit knowledge distillation from a model trained on the previous tasks to the model trained on the new task, such as Learning Without Forgetting (LwF) [27].

Elastic Weight Consolidation (EWC)

Elastic Weight Consolidation (EWC) reduces the issue of catastrophic forgetting in neural networks by constraining the learning process. The main concept behind this approach is to incorporate a quadratic penalty term into the loss function that assesses the distance between the current weight values and the optimal weights that were obtained during the previous task. In practice, some learned weights from previous tasks may be less important than others. Thus, EWC utilises Fisher Information Matrix (FIM) to estimate the importance of the weights for the previous tasks. This, provides insights into how sensitive the network is to weight changes. Weights with higher importance values significantly influence the performance of previous tasks. Consequently, during the learning of new tasks, updating these weights should be constrained more. When training the network on a new task, the EWC loss function combines the loss for the new task with the penalty term that limits weight updates based on their importance values. Through this approach, EWC reduces interference between tasks, making it easier to balance the learning of new tasks while retaining old ones.

Synaptic Intelligence (SI)

Synaptic Intelligence (SI) extends Elastic Weight Consolidation (EWC) in an online learning fashion. In SI, weight importance is computed online during Stochastic Gradient Descent (SGD), instead of computing the FIM after training, which is too costly. Importance weights tend to be overestimated in SI, and catastrophic forgetting in a pre-trained network becomes inevitable since importance weights cannot be retrieved [8].

Learning Without Forgetting (LwF)

Learning Without Forgetting (LwF) tries to mitigate catastrophic forgetting by using knowledge distillation. Before training the model on the new task, data are fed into the network and the outputs are saved. Subsequently, during training on the new task, the output value are used to distil knowledge from the previous tasks. However, the effectiveness of this method depends on the degree of correlation of the new data with the previous tasks. Furthermore, this method requires an additional forward pass of all new task data and storage of the outputs.

2.3.3 Architecture Strategies

Architecture-based strategies modify the structure of the network for each task to prevent catastrophic forgetting. As a result, the model has specific components for each task. A disadvantage of this methodology is that it generally requires task label during inference to activate the task specific components, which is generally not available in real-world applications. When there are no constraints in the model size, one can expand the model architecture adding task specific parameters or layers, such as Progressive Neural Networks (PNN) [42]. Alternatively, the architecture structure remains fixed and some neurons or layers are specific for a task, such as in PackNet [32].

Progressive Neural Networks (PNN)

Progressive Neural Networks (PNN) add a task-specific column to prevent forgetting. For each new task to be learned, the network is extended with a new column. The idea is to use the columns from previous tasks as knowledge and use lateral connections between them to adapt to learning the new task. Throughout the training process, the previous columns remain frozen while the lateral connections of the new column are being learned. This technique combines parameter freezing and network expansion. Its major disadvantage is the considerable amount of memory it demands, which makes it not scalable.

PackNet

PackNet uses a binary mask to select parameters for each task. The process involves two phases. Firstly, it identifies important weights for previous tasks by pruning the network. Subsequently, the model maintains those parameters while retraining the remaining subsets. One disadvantage is that performance

gradually decreases as more tasks are learned due to the limited number of free parameters in the network.

2.3.4 Rehearsal Strategies

Rehearsal strategies use data from past tasks while training a new task to prevent catastrophic forgetting. There are two main approaches. The first approach stores a subset of samples from the current task in a fixed-size memory, such as in Experience Replay [5, 40] or in Latent Replay [14, 33, 36]. Alternatively, some techniques use generative models to create pseudo-samples of past tasks, such as Generative Replay [46].

Replay strategies have proven to be effective in CL [49]. However, compared to other strategies, these approaches require additional external memory, which is limiting in some scenarios. In some real-world application, data storage can lead to privacy concerns. Furthermore, if the memory size is insufficient, the strategy's performance decreases as the number of tasks increases, causing catastrophic forgetting [7]. Additionally, this strategy requires additional computation, which is not necessary in other approaches.

Experience Replay

In Experience Replay, a small amount of data from previous tasks is used to retain past information during the training phase of a new task. A schematic representation of the strategy is reported in Figure 2.7.

In practice, the model has a replay memory M, and if the total number of tasks T is known, $m = M/T$ samples are allocated for each of the tasks. However, if the total number of tasks T is not known at the beginning, the space m allocated to each task gradually decreases as the number of new tasks increases. In the simplest case, the samples to be stored are randomly selected, but ideally, they should be carefully chosen to be the most representative of the task.

During the training phase on a new task, each batch of new data is mixed with a batch of data from the replay memory. Subsequently, the model is trained on this data union. In this way, the model retains previously learned information and at the same time acquires new knowledge. A trade-off must be found between memory size and model performance for the strategy to be effective. In addition, the used batch size ratio affects the performance. Generally, the two batches can be of equivalent size. Otherwise, in [36, 37] a 1/5 ratio of new samples to samples taken from memory has been shown to be effective in maintaining the memory of past tasks and learning new ones.

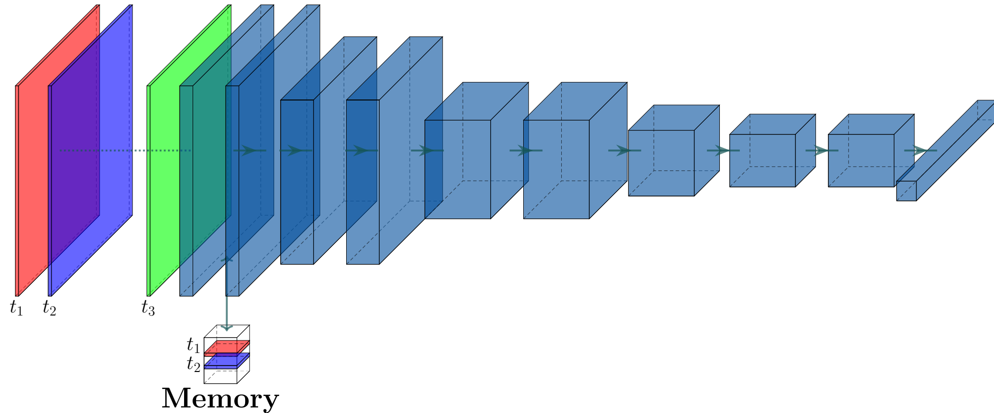


Figure 2.7: Architectural diagram of the Experience Replay. There are three tasks, t_1 and t_2 are past tasks stored in the replay memory and t_3 is the current task.

Generative Replay

Generative Replay approaches avoid storing past training examples by exploiting generative models to replicate the distribution of input data. Typically, Generative Adversarial Networks (GANs) [12] or Variational Auto-Encoders (VAEs) are used as generative models.

In this approach, generative models are trained on the distribution of input data. In this way, memory is preserved by generating samples of past experiences while learning new data, without the need for external memory. In a common Generative Replay configuration, one frozen model generates samples from past experience, while another model learns how to classify the generated past experience and to generate and classify current task samples. At the end of a task, the generative frozen model is replaced with the new learned model, which is also frozen, and a new generative model is initialized for the next task.

2.4 Latent Replay

2.4.1 Motivation

In Section 2.3.4, we introduced the general idea of Experience Replay strategies. These strategies involve merging data from new tasks with some samples of previous ones, and then fine-tuning the neural network with the combined dataset. The idea of replay is inspired by the functioning of the human

brain, where new experiences are initially encoded in the hippocampus. Subsequently, these memories are reactivated alongside with other memories so that the neocortex can assimilate them [14]. The main difference between the replay strategies discussed and the natural learning processes is the manner in which replay is executed. Replay methods involve storing and subsequently replaying raw pixel data, but the representations preserved within the hippocampus for replay purposes are not exact replicas of the original data (such as raw pixels). Instead, they capture higher-level features in the visual processing hierarchy that are located beyond the primary visual cortex or retina [14].

The Latent Replay [14, 33, 36] concept is inspired by this natural behaviour of the human brain, and instead of storing raw images for replay, it stores mid-level features of CNNs. This approach overcomes some of the weaknesses of the Experience Replay approach: (i) the high cost of storing samples belonging to old tasks, (ii) the additional computation required to maintain the memory of old tasks, and (iii) potential for mitigating privacy concerns [33].

2.4.2 Latent Replay Strategies

Latent Replay strategies store activations from an intermediate layer of a neural network and subsequently replay these representations to prevent forgetting. By doing so, they reduce the amount of memory needed to store samples. Furthermore, the strategies assume that the first layers of a CNN produce general low-level features that can be shared across tasks. By leveraging this assumption, the approaches enhance training efficiency when learning new task. For the strategies to be effective, a trade-off between the latent layer selection, the replay memory dimension and the model performance must be found.

In [36], the authors store the latent activations and during the training phase on a new task, these activations are injected into the latent layer. To prevent latent representations shift, they slow down the learning in the layers preceding the latent one, leaving all the other layers free to learn. Another approach proposed in [33], uses a freeze pre-trained models to extract features that are then used to train a classification head model. For each encountered task, a subset of the extracted features are stored and used for replay. Freezing the lower layers reduces the computational cost of Continual Learning by reducing the number of trainable parameters. Furthermore, to store more samples, in [14] the authors propose using Product Quantization (PQ) to compress and efficiently save the features.

2.4.3 Formal Definition

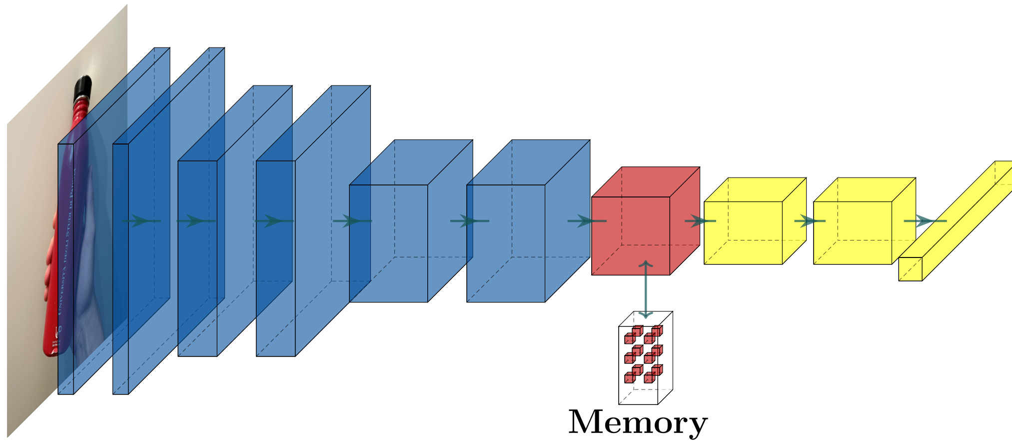
A formal definition of Latent Replay can be derived as in [14], considering the class-incremental scenario discussed in Section 2.2.3.

Consider the Figure 2.8a, the model f can be seen as two nested functions: $g(\cdot)$ (*blue* layers), parametrized by θ_g, corresponds to the layers preceding the latent layer (*red* layer), while $h(\cdot)$ (*yellow* layers), parametrized by θ_h, corresponds to the subsequent layers. The network can be represented as $y = f(x) = h(g(x))$, where $x \in \mathcal{X}$ is the input data and $y \in \mathcal{Y}$ is the output prediction.

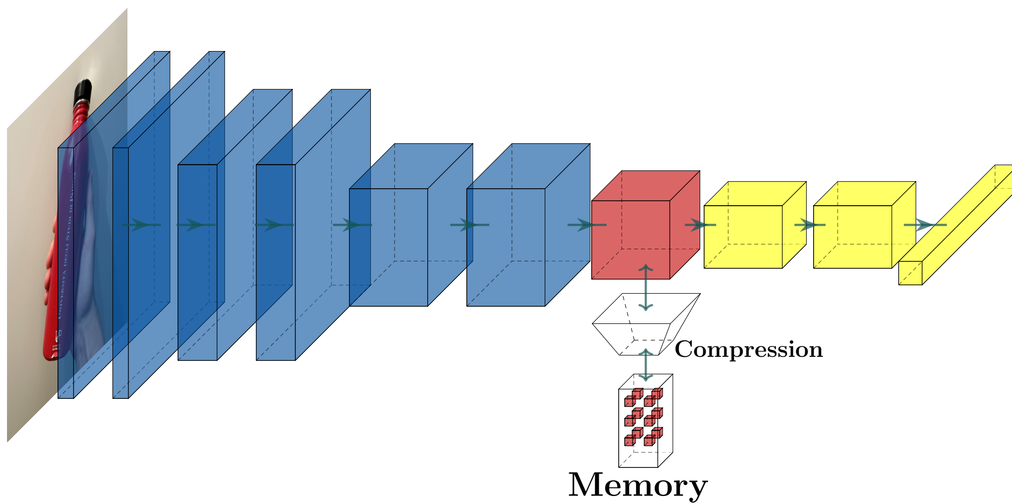
Assuming that the first layers of a CNN are highly transferable, the function $g(\cdot)$ is a pre-trained feature extractor. It could have been supervised pre-training on a portion of the dataset, supervised pre-training on a different dataset, or unsupervised self-taught learning using a convolutional autoencoder [14]. The function $g(\cdot)$ gives in output a tensor $z \in \mathbb{R}^{n \times n \times d}$, where $n \times n$ represent the spatial dimensions of a feature map, while d represents the number of channels.

During the training phase of the model on a new task, a number of latent activations z are stored in the replay memory M, typically by randomly selecting them. As for the Experience Replay, m samples are allocated in the memory for each task, and when it is full, an equal number of samples for each task are randomly removed from the memory. By doing this, the number of samples per task in the replay memory remains balanced.

In order to keep memory of previous learned tasks, r_i latent activation per tasks t_i are randomly selected in the memory and mixed with the current batch of input data in the latent layer. Regardless of the chosen Latent Replay strategy, the θ_h parameters are updated. While, following the approach in [36], the θ_g parameters are trained at a slower rate than θ_h. Instead, following [14, 33], the θ_g parameters are frozen. In addition, in [14] they use Product Quantization (PQ) to compress and store the z tensor reducing the required amount of memory. An architectural scheme of the Latent Replay with quantization is reported in Figure 2.8b.



(a) *Architectural diagram of Latent Replay.*



(b) *Architectural diagram of Latent Replay with compression.*

Figure 2.8: The model takes the image x_i as input, and feeds it to the feature extractor $g(\cdot)$ in *blue*. The output $g(x_i) = z_i$ is stored in the replay memory and, during the training of new tasks, tensors are sampled from the memory and injected into the latent layer in *red*. They are then propagated through the classification head in *yellow*.

Chapter 3

Benchmarks and Metrics for Continual Learning

In this chapter, we discuss some datasets and metrics commonly used in the evaluation of Continual Learning strategies.

An overview of classical computer vision datasets and their limitations for Continual Learning is given in Section 3.1. In Sections 3.2.1 and 3.2.2, some CL benchmarks are reported for the CIL and DIL scenarios. Section 3.2.3 reports on datasets specifically designed for continuous learning purposes.

Finally, Sections 3.3.1 and 3.3.2 present the metrics commonly used in the literature to evaluate the performance and efficiency of CL strategies.

3.1 Classic Datasets

Classic machine learning datasets are not suitable for training models in a Continual Learning scenario, since they are generally based on single tasks. However, with appropriate modifications, Continual Learning datasets can be derived from these datasets. Below is provided a brief description of some of the most common datasets used in classical DL.

- **MNIST** [24]: The MNIST dataset is a widely used dataset in the field of machine learning and computer vision. It consists of 28×28 pixel greyscale images divided into 10 classes of handwritten digits $(0 - 9)$. It includes 60 000 training images and 10 000 test images.
- **CIFAR-10/100** [23]: The CIFAR-10 and CIFAR-100 datasets are used for image classification in computer vision. They consist of 32×32 pixel colour images, which are divided into 10 and 100 classes respectively. Both dataset consists of 50 000 training images, 5000/500 per

Figure 3.1: CIFAR-10 Dataset [23]. From top to bottom, the rows depict images of the following classes: *Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship* and *Truck.*

class, and 10 000 test images, 1000/100 per class. An example of the CIFAR-10 dataset is reported at Figure 3.1.

- **ImageNet-1K** [41]: ImageNet[1] is one of the largest image datasets in the field of computer vision, with more than 14 million images. One of the most widely used subsets is ImageNet-1K, which includes 1000 object classes and contains 1 281 167 training images, 50 000 validation images and 100 000 test images.
- **CUB-200-2011** [16]: The Caltech-UCSD Birds-200-2011 (CUB-200-2011) dataset is used for fine-grained image classification. It contains 11 788 images of birds divided in 200 bird species. It includes 5994 for training and 5794 for testing.

Many other CV and non-CV datasets have been tailored for use in Continual Learning, but a comprehensive explanation exceeds the scope of this thesis.

3.2 Continual Learning Benchmark

Continual learning benchmarks can be generated from deep learning datasets by applying transformations. A common approach is to divide the dataset

[1] `https://www.image-net.org/index.php`

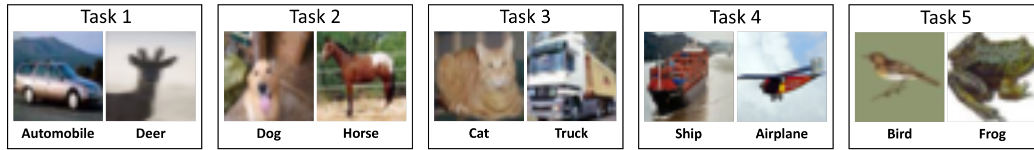


Figure 3.2: Split CIFAR-10 benchmark.

into smaller groups, each containing a different subset of categories, with each subset identified as a task. In addition, techniques such as image rotation or pixel permutation can be used to create several tasks from a single dataset. However, the choice of the transformation to be applied depends on the specific Continual Learning scenario being considered. Note that it is possible to construct a Continual Learning benchmark by combining multiple datasets instead of using a transformation of a single dataset [7].

3.2.1 Class-Incremental Learning Benchmark

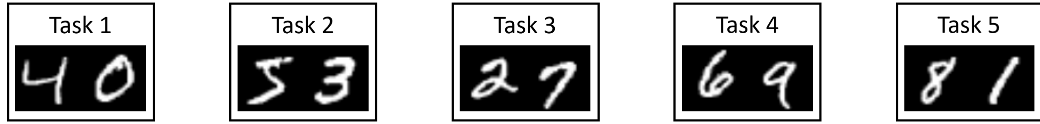
In the case of Class-Incremental Learning, benchmarks can be constructed from a classical dataset that contains an initial set of classes. The classes in the dataset are then split into different tasks. An example is the Split CIFAR-10 [52] shown in Figure 3.2, where the classes of CIFAR-10 have been divided into 5 tasks. Some well-known benchmarks in this scenario are:

- **Split MNIST** [52]: The different digits of the MNIST dataset are divided into 2, 5, 10 tasks, containing 5, 2 and 1 digit(s) per task, respectively. An example with 5 tasks is shown in Figures 3.3a.

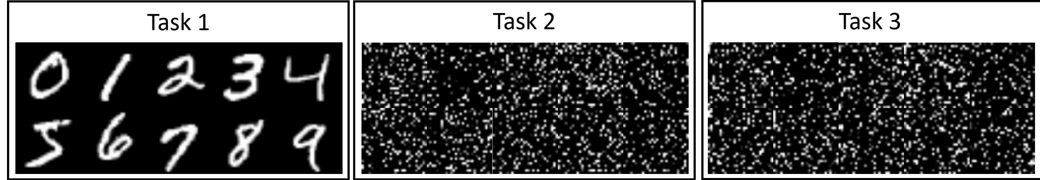
- **Split CIFAR-10/100** [52]: The different classes of CIFAR-10/100 can be divided in various ways. For example, CIFAR-10 can be divided into 2, 5 or 10 tasks. CIFAR-100, among other possibilities, can be divided into 20 disjointed subsets, 5 classes for each task. In [52], the authors use as *Task 1* the entire CIFAR-10 dataset and sequentially 5 additional tasks, each corresponding to 10 classes of the CIFAR-100 dataset. In [39], the 100 classes of CIFAR-100 dataset are divided in tasks of 2, 5, 10, 20 or 50 classes at a time. An example of CIFAR-10 divided in 5 tasks is reported in Figure 3.2.

- **iILSVRC** [39]: A subset of 100 classes of ImageNet dataset are divided in 10 tasks (**iILSVRC-small**) or all the 1000 classes divided in 100 tasks (**iILSVRC-full**).

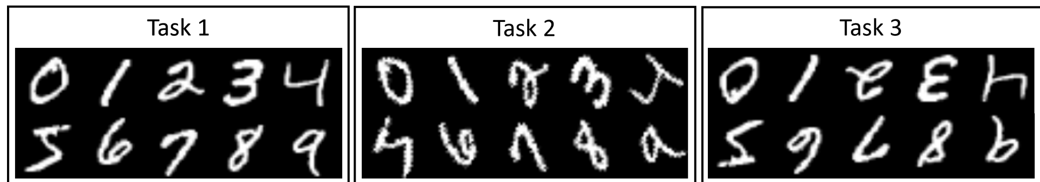
The same benchmarks could be considered in the Task-Incremental Learning scenario if task label information is added to the data set subdivision.



(a) *Split MNIST benchmark with* 5 *tasks.*



(b) *Permuted MNIST Benchmark with* 3 *tasks.*



(c) *Rotated MNIST benchmark with* 3 *tasks:* 0°*,* 60° *and* 180°.

Figure 3.3: MNIST Benchmarks.

3.2.2 Domain-Incremental Learning Benchmark

In the Domain-Incremental Learning, the difference between the tasks is the data input distribution, while the number of classes remains fixed. Some well-known benchmarks in this scenario are:

- **Permuted MNIST** [13]: A random permutation is applied to the original MNIST digits resulting in a new task for each permutation. Although the permuted images are not interpretable for humans, the pixels continue to follow a pattern since similar digits result in similar permuted digits. One benefit of this benchmark for Continual Learning is the ability to define an arbitrarily long sequence of tasks by selecting different permutations for the pixels. Figure 3.3b reports an example with 3 task.

- **Rotated MNIST** [31]: A rotation between 0° and 360° is applied to the original images. Also in this case it is possible to obtain a long sequence of tasks. Figure 3.3c reports an example with 3 task.

Other DIL benchmarks can be the result of a combination of a data set and some artistic transformations, as for the CIFAR-10 in Figure 3.4.

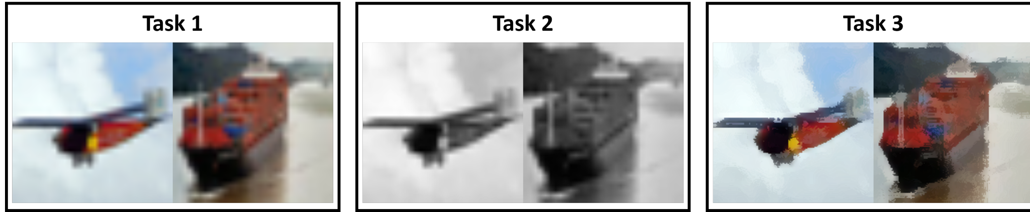


Figure 3.4: CIFAR-10 benchmark for Domain-Incremental Learning scenario. In Task 1, the original images are reported, while in Tasks 2 and 3, a grayscale filter and a filter with a pictorial effect are respectively applied.

3.2.3 Dataset for Continual Learning

In addition to the benchmarks reported in Sections 3.2, there are datasets specifically designed for Continual Learning.

- **CORe50** [29]: It is a collection of 50 domestic objects belonging to 10 categories. It consists of 128×128 pixel colour images with 164 866 samples. The dataset has been collected with different backgrounds and lighting. Classification can be performed at category level (10 classes) or at the object level (50 classes).

- **OpenLORIS-Object** [45]: It is a collection of 69 instances including 19 categories daily necessities objects under 7 scenes. The dataset is collected recording the videos of targeted objects under multiple illuminations, occlusions, camera-object distances/angles, and context information (clutters).

3.3 Metrics

Classical deep learning metrics are not suitable for Continual Learning, which involves training on a stream of tasks over time rather than training on a single task. Hence, specific metrics must be defined to evaluate a CL strategy on a given benchmark. Many metrics employed to evaluate the CL strategy combine classic DL metrics.

Based on the specific task or model, it is possible to use different metrics. For a classification task, an effective CL solution should have high accuracy, low forgetting, low memory consumption, and be computationally efficient [7]. Therefore, a comprehensive assessment must investigate different aspects of the strategy. Including final accuracy in all the encountered tasks, how

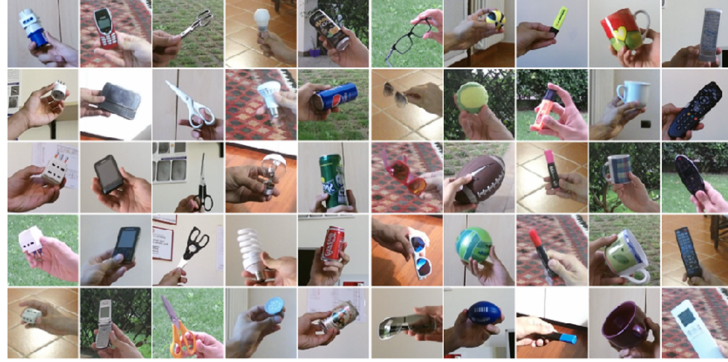


Figure 3.5: CORe50 dataset. From left to right, the columns depict images of the following classes: *plug adapters*, *mobile phones*, *scissors*, *light bulbs*, *cans*, *glasses*, *balls*, *markers*, *cups*, *remote controls*. Image adapted from [29].

fast it learns and forgets, and the algorithm's capacity to transfer knowledge across different tasks. To have a robust evaluation of the CL strategies, it would be also important to determine the amount of computation and memory resources exploited. Otherwise, if we aim to solely evaluate the model's performance, the easiest solution would have been to retrain the model from scratch for each task.

In the following we report a set of the common metrics used in the Continual Learning setting as defined in [9, 31].

For evaluation purposes, we define a training set T_{r_i} and a test set T_{e_i} for each task $t_i \in \mathcal{T}$. This enables the evaluation of the model on all the tasks $t \in \mathcal{T}$ after the training of the task t_i. By doing so, we obtain the accuracy matrix $R \in \mathbb{R}^{T \times T}$ [31], with $T = |\mathcal{T}|$. For $T = 3$, the matrix is reported in Table 3.1, where each entry R_{ij} is the test classification accuracy of the model on task t_j after learning task t_i and $R^* = R_{ii}$ correspond to the classic accuracy on the current task. In the following, the accuracy matrix is used to compute the performance metrics of the model on the task stream.

3.3.1 Performance metrics

Average Accuracy (ACC)

Given the accuracy metrics R, the *Average Accuracy (ACC)* [9] is defined as:

$$ACC = \frac{\sum_{i=1}^{T} \sum_{j=1}^{i} R_{ij}}{\frac{T(T+1)}{2}} \tag{3.1}$$

R	T_{e_1}	T_{e_2}	T_{e_3}
T_{r_1}	R^*	R_{ij}	R_{ij}
T_{r_2}	R_{ij}	R^*	R_{ij}
T_{r_3}	R_{ij}	R_{ij}	R^*

Table 3.1: Accuracy matrix. Elements in R accounted to compute the Accuracy (elements in *white* and *cyan*), BWT (elements in *cyan*), and FWT (elements in *gray*) criteria. $R^* = R_{ii}$, T_{r_i} = training, T_{r_i} = test tasks. Table adapted from [9].

It considers the average accuracy for training set T_{r_i} and test set T_{e_j} by considering the diagonal elements of R, as well as all elements below it (cyan elements). By doing so, it keeps into account the performance of the model at every encountered task and extend the original ACC defined in [31] that asses the performance of the model at the end of the last task, i.e., it considers only the last row of the accuracy matrix.

Forgetting (F)

To quantifies the drop in accuracy of task t_j after the model has been trained on task task t_i, we define the *Forgetting (F)* as in [4, 7]. It is defined as the difference between the maximum knowledge gained about a particular task during the learning process in the past and the knowledge the model currently has about it. The forgetting metric gives an estimate of how much the model has forgotten about the task given its current state.

For a classification problem, we quantify forgetting for the task t_j after the model has been incrementally trained up to task t_k as:

$$f_j^k = \max_{l \in \{1,\dots,k-1\}} R_{l,j} - R_{k,j}, \quad \forall j < k \tag{3.2}$$

It is worth noting that $f_j^k \in [-1, 1]$ is only defined for $j < k$, since we are interested to measure forgetting for previous tasks. Moreover, by normalizing against the number of tasks seen previously, the average forgetting at task t_k is written as:

$$F = \frac{1}{k-1} \sum_{j=1}^{k-1} f_j^k \tag{3.3}$$

A lower value of F implies a less forgetting on previous tasks, thus demonstrating the model's ability to retain past knowledge.

Backward Transfer (BWT)

The *Backward Transfer (BWT)* [9, 31] is the influence that learning a task t has on the performance on a previous tasks. As for the average accuracy, BWT is defined to consider the average of the backward transfer after each task:

$$BWT = \frac{\sum_{i=2}^{T} \sum_{j=1}^{i-1} R_{ij} - R_{jj}}{\frac{T(T+1)}{2}} \tag{3.4}$$

In order to map BWT to also lie on $[0, 1]$ and to distinguish among two semantically different concepts, in [9], the authors define:

$$REM = 1 - |\min(BWT, 0)| \tag{3.5}$$

i.e., *Remembering (REM)* , the originally negative BWT and the improvement over time:

$$BWT^{+} = \max(BWT, 0) \tag{3.6}$$

i.e., *Positive Backward Transfer* , the originally positive BWT.

Forward Transfer (FWT)

The *Forward Transfer (FWT)* [9, 31] is the influence that learning a task t has on the performance on a future tasks. FWT is defined as:

$$FWT = \frac{\sum_{i=1}^{j-1} \sum_{j=1}^{T} R_{i,j}}{\frac{T(T+1)}{2}} \tag{3.7}$$

3.3.2 Efficiency metrics

Model Size Efficiency (MS)

It is important to evaluate the effectiveness of a CL strategy to learn new tasks with a fixed amount of model capacity without deteriorating performance. Ideally, the size of the model at the end of training should be the same as the size of the initial model [7]. For evaluation purposes, we define θ_i as the parameters of the model at the task t_i. The memory size $Mem(\theta_i)$ at each task t_i should not grow too rapidly with respect to the size of the model in the first task t_1, $Mem(\theta_1)$. From this, the *Model Size Efficiency (MS)* [9] is defined as:

$$MS = \min\left(1, \frac{\sum_{i=1}^{T} \frac{Mem(\theta_1)}{Mem(\theta_i)}}{T}\right) \tag{3.8}$$

Samples Storage Size Efficiency (SSS)

As seen in Section 2.3.4, one method to reduce CF is to store training samples from previous tasks and replay them while training a new task. It is important to assess the memory usage of these samples. Indeed, store all the samples form previous task and replay them during the training of a new task could be a solution, but at cost of high memory. Therefore, it is crucial to find a trade-off between the stored data and the model performance.

The memory occupation in bits by the samples storage memory M, $Mem(M)$, should be bounded by the memory occupation of the total number of examples encountered at the end of the last task, $Mem(D)$, where D is the dataset of all the encountered data. Thus, we define *Samples Storage Size (SSS) Efficiency* [9] as:

$$SSS = 1 - \min\left(1, \frac{\sum_{i=1}^{T} \frac{Mem(M_i)}{Mem(D)}}{T}\right) \tag{3.9}$$

Computational Efficiency (CE)

The *Computational Efficiency (CE)* is bounded by the number of multiplication and addition operations for the training set T_{ri}, in[9] the average CE across tasks is defined as:

$$CE = \min\left(1, \frac{\sum_{i=1}^{T} \frac{Ops\uparrow\downarrow(T_{r_i})\cdot\epsilon}{Ops(T_{r_i})}}{T}\right) \tag{3.10}$$

where $Ops(T_{r_i})$ is the number of multiplication and addition operations needed to learn T_{r_i} and $Ops \uparrow\downarrow (T_{r_i})$ is the number of operations required to do one forward and one backward pass on T_{r_i}.

Aggregate Metric ($\mathbf{CL_{SCORE}}$)

In order to summarize into a single final metric the performance of a CL strategy, in [9], the authors propose an *Aggregate Metric (CL_{SCORE})*. For each criterion $c_i \in \mathcal{C}$, with $c_i \in [0, 1]$, is assigned a weight $w_i \in [0, 1]$ where $\sum_i^{\mathcal{C}} w_i = 1$. The final CL_{SCORE} to be maximized is defined as:

$$CL_{SCORE} = \sum_{i=1}^{\#\mathcal{C}} w_i c_i \tag{3.11}$$

where where each criterion c_i that needs to be minimized is transformed to $c_i = 1 - c_i$.

Chapter 4

Efficient Architecture for Edge Devices

This chapter introduces some of the well-known efficient architectures for edge devices, along with the features that make them effective on resource-constrained devices.

Section 4.1 introduces the problem of large deep neural networks and the concept of efficient architectures. Section 4.2 presents MobileNet, an efficient architecture designed for mobile devices, with its main features and design principles. Then, in Section 4.3, MobileNetV2, an improved version of the original MobileNet, is introduced and explored, highlighting its enhancements and optimizations. Section 4.4 introduces PhiNets, a family of scalable backbones for image processing on resource-constrained platforms. Finally, Section 4.5 presents a comparative analysis of the architectures' performance on the ImageNet dataset.

4.1 Introduction

In recent years, advancements in Artificial Intelligence (AI) have enabled technology to achieve state-of-the-art performance in areas such as face detection, Natural Language Processing (NLP), Computer Vision (CV) and Anomaly Detection (AD) [8, 49, 51]. The general trend has been to build deeper and more complex deep learning models to achieve higher accuracy [17]. Nevertheless, the training of these models is computationally demanding, and despite the increasing capability of today's edge devices, they remain insufficient for most of the deep learning models, which have high resource requirements in terms of CPU, GPU, memory, and network [51]. Most of the approaches perform the training offline on multi-GPU servers and deploy

models on edge devices for inference. However, numerous computational tasks must be sent to the cloud, and this poses serious challenges in terms of network capacity and the computing power of cloud computing infrastructure [50]. Furthermore, privacy concerns, the absence of network connections, strict latency requirements, and the need for real-time model adaptation, make real-time training on edge devices crucial.

Efforts to tackle these challenges have resulted in the development of efficient neural network architectures designed to achieve high performance while using fewer computational resources. As an example, MobileNet [17] consists of VGG-style building blocks and, by replacing standard convolutions with depthwise separable convolutions, significantly reducing the number of parameters and computational complexity while maintaining competitive accuracy. Similarly, EfficientNet [47] utilizes a compound scaling technique to optimize model depth, width, and resolution to achieve remarkable performance-efficiency trade-offs. In [34], the authors present PhiNets, a scalable backbone framework for deep neural networks. The architecture is designed to provide image-processing application support for resource-constrained edge devices. It is built on top of inverted residual blocks for decoupling memory, cost, and over-processing [38]. Furthermore, it can be easily tuned using a few hyper-parameters to match the memory and computational resources available on different embedded platforms.

4.2 MobileNetV1

MobileNet [17] is a scalable backbone composed of 28 separate convolutional layers, each followed by a batch normalization layer and a ReLU activation function, the entire architecture is shown in Table 4.1. The baseline model assumes an input size of $224 \times 224 \times 3$ and a final output size of 1000×1.

The main innovation of MobileNet is that it implements depthwise separable convolution, which factorizes a standard convolution into a depthwise convolution and a pointwise convolution. While standard convolution employs kernels on all input channels and combines them in one step, depthwise convolution uses different kernels for each input channel and performs a pointwise convolution to combine the outputs of the depthwise layer. This separation reduces the computational cost and model size of the architecture. In addition, MobileNet introduces the possibility to enhance speed and reduce the model size of the network by acting on the width and resolution multipliers.

Input	Type	s	c
$224 \times 224 \times 3$	3×3 conv	2	32
$112 \times 112 \times 32$	3×3 dw conv	1	32
$112 \times 112 \times 32$	1×1 conv	1	64
$112 \times 112 \times 64$	3×3 dw conv	2	64
$56 \times 56 \times 64$	1×1 conv	1	128
$56 \times 56 \times 128$	3×3 dw conv	1	128
$56 \times 56 \times 128$	1×1 conv	1	128
$56 \times 56 \times 128$	3×3 dw conv	2	128
$28 \times 28 \times 128$	1×1 conv	1	256
$28 \times 28 \times 256$	3×3 dw conv	1	256
$28 \times 28 \times 256$	1×1 conv	1	256
$28 \times 28 \times 256$	3×3 dw conv	2	256
$14 \times 14 \times 256$	1×1 conv	1	512
$5\times$ $14 \times 14 \times 512$	3×3 dw conv	1	512
$14 \times 14 \times 512$	1×1 conv	1	512
$14 \times 14 \times 512$	3×3 dw conv	2	512
$7 \times 7 \times 512$	1×1 conv	1	1024
$7 \times 7 \times 1024$	3×3 dw conv	1	1024
$7 \times 7 \times 1024$	1×1 conv	1	1024
$7 \times 7 \times 1024$	7×7 avgpool	1	1024
$1 \times 1 \times 1024$	FC	1	k

Table 4.1: MobileNetV1 architecture. c represents the number of output channels of each layers. s represent the stride.

Depthwise Separable Convolution

MobileNet model is based on depthwise separable convolutions which factorize a standard convolution into a depthwise convolution and a 1×1 convolution known as pointwise convolution.

To understand the benefit of this approach, it is worth comparing the MACs using the standard convolution and the depthwise separable convolution. Assume depthwise separable convolution takes as input a $D_F \times D_F \times M$ feature map $\mathbf{F}$ and produces a $D_G \times D_G \times N$ feature map $\mathbf{G}$, where D_F is the spatial width and height of a square input feature map, M is the number of input channel, D_G is the spatial width and height of a square output feature map and N is the number of output channel. The depthwise convolution applies one filter per input channel. Specifically, the kernel $\mathbf{K}$ is of size $D_K \times D_K \times M$, where the m_{th} filter is applied to the m_{th} channel in the input $\mathbf{F}$ to produce the m_{th} channel of the output $\hat{\mathbf{G}}$. This has a computational cost of [17]:

$$D_K \cdot D_K \cdot M \cdot D_G \cdot D_G \tag{4.1}$$

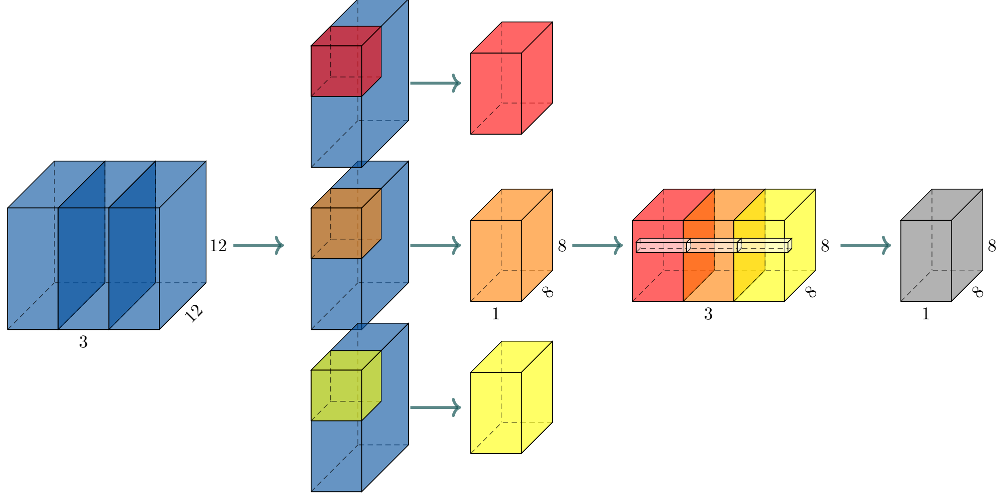


Figure 4.1: Depthwise Separable Convolution. The input has a dimension of $12 \times 12 \times 3$. By using the depthwise convolution with a kernel of dimension $5 \times 5 \times 3$ and stride of 1, results in an output $\hat{\mathbf{G}}$ with dimension $8 \times 8 \times 3$. Finally, applying a pointwise convolution with kernel $1 \times 1 \times 3$ results in an output of dimension $8 \times 8 \times 1$.

A simple $1 \times 1 \times M$ convolution, known as pointwise convolution, is used to build new features through computing a linear combination of the output $\hat{\mathbf{G}}$. Using N of this pointwise convolution, we obtain an output features map $\mathbf{G}$ of dimension $D_G \times D_G \times N$. With a computational cost of [17]:

$$M \cdot N \cdot D_G \cdot D_G \tag{4.2}$$

The combination of the depthwise convolution and the pointwise convolution is called depthwise separable convolution, a graphical representation is reported in Figure 4.1. It has a total computational cost of :

$$D_K \cdot D_K \cdot M \cdot D_G \cdot D_G + M \cdot N \cdot D_G \cdot D_G \tag{4.3}$$

On the other hand, as reported in [17], the standard convolution has a computation cost of:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_G \cdot D_G \tag{4.4}$$

Therefore, the use of depthwise separable convolution results in a reduction of $\frac{1}{N} + \frac{1}{D_K^2}$ of operations.

Width and Resolution Multiplier

MobileNet model is specified by two hyperparameters: width and resolution multiplier. These allow to build models that are smaller and require less computational power [17].

The width multiplier $\alpha \in (0, 1]$ thins the network uniformly at each layer, i.e., controls the number of channels. These are implemented to be always divisible by 8. Instead, the resolution multiplier $\rho \in (0, 1]$, modifies the input image resolution and the internal representation of every layer is subsequently reduced by the same multiplier. In conclusion, the computational cost of the depthwise separable convolution with the width and resolution multiplier is:

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_G \cdot \rho D_G + \alpha M \cdot \alpha N \cdot \rho D_G \cdot \rho D_G \tag{4.5}$$

4.3 MobileNetV2

MobileNetV2 [44] builds on the ideas of MobileNetV1 [17], using depth-separable convolution as an efficient convolutional block. However, it introduces two new concepts: linear bottlenecks and shortcuts between the bottlenecks. The idea is that the bottlenecks encode the model's intermediate inputs and outputs, while the inner layer encapsulates the model's ability to transform from lower-level concepts to higher-level descriptors [43]. Finally, the model uses the ResNet [15] concept of shortcut connections to achieve faster training and increased accuracy. In contrast to the typical approach of connecting non-bottleneck layers, MobileNetV2 inverts this notion and connects the bottlenecks directly. By combining these two concepts, the authors introduce a novel layer known as the inverted residual with linear bottleneck.

The architecture of MobileNetV2 is shown in the Table 4.2 and includes a standard convolutional layer with 32 kernels, followed by 19 inverted residuals and linear bottlenecks. As for version 1, the baseline model assumes an input of $224 \times 224 \times 3$ and a width multiplier of 1. The model could be tuned for different applications by exploiting the resolution and width multiplier hyperparameters.

Inverted Residual and Linear Bottleneck

Inverted Residual with Linear Bottleneck layers are based on the idea that information may be embedded in low-dimensional subspaces, which can be leveraged by reducing the dimensionality of a layer. However, this approach can break down when non-linear activations like ReLU are present in neural

Input	Type	n	s	t	c
$224 \times 224 \times 3$	3×3 conv2d	1	2	-	32
$112 \times 112 \times 32$	bottleneck	1	1	1	16
$112 \times 112 \times 16$	bottleneck	2	2	6	24
$56 \times 56 \times 24$	bottleneck	3	2	6	32
$28 \times 28 \times 32$	bottleneck	4	2	6	64
$14 \times 14 \times 64$	bottleneck	3	1	6	96
$14 \times 14 \times 96$	bottleneck	3	2	6	160
$7 \times 7 \times 160$	bottleneck	1	1	6	320
$7 \times 7 \times 320$	1×1 conv2d	1	1	-	1280
$7 \times 7 \times 1280$	7×7 avgpool	1	-	-	1280
$1 \times 1 \times 1280$	1×1 conv2d	-	-	-	k

Table 4.2: MobileNetV2 architecture. Each layer is repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. t is the expansion factor of the input channels. Table adapted from [44].

Input	Operator	Activation	Output
$h \times w \times k$	1×1, conv2d	ReLU6	$h \times w \times tk$
$h \times w \times tk$	3×3, depthwise conv, stride $= s$	ReLU6	$\frac{h}{s} \times \frac{w}{s} \times tk$
$\frac{h}{s} \times \frac{w}{s} \times tk$	1×1, conv2d	Linear	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 4.3: Bottleneck residual block. Table adapted from [44].

networks. Non-linear activations have the potential to increase representational complexity but may also lead to a loss of information [44]. However, it is possible to preserve information if a lot of channels are present. The Inverted Residual with Linear Bottleneck layer was designed with these two concepts in mind.

The module takes in input a low-dimensional tensor with k channels and performs three different convolutions: (1) a pointwise expansion convolution, (2) a depthwise convolution, and (3) a pointwise projection convolution. The structure of the bottleneck module is reported in Table 4.3.

The first pointwise convolution expands the low-dimensional input to a higher-dimensional space suited to non-linear activation, and ReLu6 is applied. The expansion factor is controlled by t, resulting in tk channels after the first convolution. Subsequently, a depthwise convolution is executed using 3×3 kernels followed by a ReLu6 activation, achieving a channel transformation. Lastly, the feature map is projected back into a low-dimensional

subspace using another pointwise convolution. Based on the idea that non-linear activation results in loss of information, it is important to have a linear activation function in the final layer. This, has been proven empirically by the authors in [44].

The spatial resolution of the channels is controlled in the depthwise convolution through the stride. If the input and output channels are equal and the stride in the depthwise convolution is equal to one, and thus the input and output feature map have the same dimension, a residual connection is added. The residual connection should be added in the bottleneck rather than in the expanded layers, as empirically shown to be more effective [44].

4.4 PhiNet

PhiNet is a scalable backbone for image processing on resource-constrained edge devices [34]. It aims to solve the main drawbacks of current state-of-the-art scalable backbones for image processing at the edge.

PhiNet is based upon the idea of Inverted Residual with Linear Bottleneck presented in MobileNetV2 [44]. In addition, the authors insert a Squeeze-and-Excitation layer [18] before the final pointwise projection convolution. This emphasizes the channel information before compressing it to a lower-dimensional subspace. Furthermore, a hardware-aware scaling paradigm is used to optimize the number of operations, dynamic memory, and parameter memory by tuning various hyperparameters. This allows for superior performance with respect to networks generated using hardware-constrained scaling technique [34].

PhiNet Convolutional Block

The PhiNet Convolutional Block, shown in Figure 4.2, is based on an extension of the Inverted Residual Block. As in MobileNetV2, the low-dimensional input feature map is expanded with a pointwise convolution, this time followed by an h-swish activation [2]. A depthwise convolution followed by an h-swish activation is then performed to transform the channel information. Then, to evaluate the importance of each channel in the output of the depthwise convolution, a squeeze-and-excitation block has been added. Finally, a pointwise projection convolution is used to bring the feature map back to low dimensionality. A residual connection between input and output channels with the same dimension has been added as for MobileNetV2. The structure of the PhiNet convolutional block module is reported in Table 4.4.

In order to meet the requirements of resource-constrained devices, PhiNet

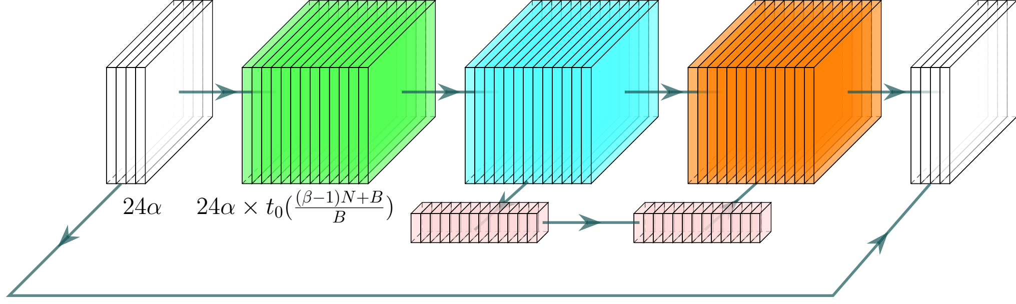


Figure 4.2: PhiNet convolutional block. The input number of channels is increased with a pointwise convolution, followed by a depthwise convolution and a SE block. Finally, a second pointwise convolution project back to the low-dimensional space the number of channel. If the dimension of the input channels and the output channels are equal a residual connection is added. Image adapted from [34].

Input	Operator	Activation	Output
$h \times w \times k$	1×1, conv2d	HardSwish	$h \times w \times tk$
$h \times w \times tk$	3×3, depthwise conv, stride $= s$	HardSwish	$\frac{h}{s} \times \frac{w}{s} \times tk$
$\frac{h}{s} \times \frac{w}{s} \times tk$	SEBlock	HardSwish	$\frac{h}{s} \times \frac{w}{s} \times tk$
$\frac{h}{s} \times \frac{w}{s} \times tk$	1×1, conv2d	Linear	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 4.4: PhiNet Convolutional Block.

utilizes a hardware-aware network scaling pipeline. Specifically, using four hyperparameters enables network optimization in a decoupled manner to achieve optimal use of the available hardware:

- **Width multiplier** α: linearly adjusts the channels count in the convolutions block of the network. The base model has $24 \cdot \alpha$ channels in the first bottleneck layer.

- **Shape parameter** β: defines the channels count of the depth-wise convolution blocks in the networks.

- **Base expansion factor $\mathbf{t_0}$**: affects the channels count in the expansion convolutions of the inner blocks. In particular, the expansion factor linearly depends with the depth of the network:

$$t = t_0\Big(\frac{(\beta - 1)N + B}{B}\Big) \tag{4.6}$$

  where N is the index of the convolutional block and B is the total number of convolutional block.

Input	**Type**	**n**	**s**	**S – E**	**c**
$224 \times 224 \times 3$	3×3 dw conv	1	2	✗	3
$112 \times 112 \times 3$	1×1 conv	1	1	✗	144
$112 \times 112 \times 144$	PhiNetConvBlock	1	1	✗	72
$112 \times 112 \times 72$	PhiNetConvBlock	2	2	✓	72
$56 \times 56 \times 72$	PhiNetConvBlock	2	2	✓	144
$28 \times 28 \times 144$	PhiNetConvBlock	2	2	✓	288
$14 \times 14 \times 288$	PhiNetConvBlock	1	1	✓	576
$7 \times 7 \times 576$	7×7 avgpool	1	-	✗	1280
$1 \times 1 \times 576$	1×1 conv2d	-	-	✗	k

Table 4.5: PhiNet architecture with $\alpha = 3$, $\beta = 0.75$, $t_0 = 6$, and $B = 7$. Each layer is repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. S-E indicate the presence of the Squeeze-and-Excitation layer.

- **Number of layers B**: is the total number of PhiNet convolutional block and thus modify the depth of the network.

The variation of the hyperparameters change the structure of the networks. The optimization of resource usage can be achieved through varying the combinations of these hyper-parameters. It should be noted that different hyper-parameter configurations may result in comparable parameter counts, but different architectures [3]. An example of the structure of the PhiNet model is reported in Table 4.5

4.5 Comparative Analysis

The three efficient architectures described can be specified with different hyperparameters. By combining them differently, it is possible to build smaller models that require less computational power.

Specifically, the width and resolution parameters can be set for both versions of MobileNet. In this comparison, we will keep the resolution parameter fixed, maintaining the input resolution at 224. We will compare the resource utilization by adjusting the width parameter, thus reducing the size of the models. The same approach will be applied to PhiNet, where the input resolution is fixed at 224, and adjustments to the parameters α, β, and t_0 will change the model's structure. It is important to note that while MobileNet utilizes a single parameter to scale the entire model, in PhiNet, the three hyperparameters allow the size of the input/output channels and expansion

Model	α	β	t_0	Nr. of Layer	MACs	Params
PhiNet	0.8	0.75	8	7	195.8M	1.7M
	0.9	0.5	4	7	100.2M	0.5M
	0.9	0.5	4	9	123.3M	2.1M
	1.2	0.5	6	7	248.8M	1.6M
MobileNet V1	1	-	-	19	567.8M	3.3M
	0.75	-	-	19	324.7M	1.9M
	0.5	-	-	19	149.0M	0.9M
	0.25	-	-	19	40.8M	0.3M
MobileNet V2	1	-	-	28	312.4M	2.2M
	0.75	-	-	28	185.4M	1.4M
	0.5	-	-	28	91.0M	0.7M
	0.25	-	-	28	29.2M	0.2M

Table 4.6: Resource usage of different Efficient Architectures assuming an input of $3 \times 224 \times 224$.

channels to be changed independently. This enables the adjustment of the number of parameters and Multiply–Accumulate (MACs) according to the needs of the application.

Table 4.6 shows a comparison between the models in terms of MACs and the number of parameters, varying the respective hyperparameters.

Chapter 5

Results

In order to empirically assess the quality of the Latent Replay strategy applied to the efficient architectures, several experiments were conducted. The aim is to evaluate the performance and efficiency of the models for different selections of the latent layer and different dimensions of the replay memory. The experiments were conducted in a class-incremental scenario to solve image classification tasks.

In Section 5.1, we report the experimental setup, describing the methods, the models, the benchmarks, and the metrics used in the experiments. Following this, in Section 5.2, we present the main findings of this study. Initially, we examine the influence of the number of elements in the replay memory on performance. Subsequently, we conduct a comparative analysis of the performance of Latent Replay against other strategies. Finally, we delve into the performances of the efficient architectures in real-world scenarios, specifically under the constraints of memory for the replay.

5.1 Experimental Setup

In this section, we describe the experimental setup used in our study. We provide a summary of the CL strategy settings used. Next, we present the models considered and the details of their architectures, as well as the hyperparameters used. Lastly, we detail the benchmarks and metrics utilized during the experiments, along with the hardware specifications of the experimental environment.

5.1.1 Methods

In this work, we focus on applying the Latent Replay strategy introduced in Section 2.4 to efficient architectures. We evaluate how the choice of the latent layer for replay affects the performance in terms of accuracy, memory size, and efficiency across different the architectures. To comprehend the effectiveness of the approach on each model, we used three baselines to identify the minimum and maximum performance we can expect.

Baseline

In order to have an upper and lower bound for performance, the following baselines were taken into account:

- **Fine-Tuning**: Each time a new task becomes available, we fine-tune the model trained on the previous task using only the newly available data. This approach can lead to catastrophic forgetting and is considered the minimum required performance.

- **Multi-Task:** We assume that all data are available at the beginning of the experiment, and we train the pre-trained model on all tasks of the benchmark simultaneously. This approach should improve the generalization of the model and it is considered the best achievable performance.

- **Experience Replay**: The basic Experience Replay from the input is used to compare the final average accuracy, the amount of memory used, and the number of operation with respect to the Latent Replay strategy.

Latent Replay

During Latent Replay experiments, the weights of the pre-trained model are completely frozen up to the layer selected for replay. All subsequent layers are free to learn. In the most extreme case, the entire feature extractor is completely frozen, leaving only the classification head free to learn. The layer utilized for Latent Replay depends on the model. For MobileNetV1, we utilized the output layer of the Depthwise Separable Convolution. While, for MobileNetV2 and PhiNet, the output layer of the Inverted Residual with Linear Bottleneck was used.

During the training of new tasks, a subset of data is stored in memory to retain knowledge of the current task during the training of future tasks. The replay memory is implemented to maintain a balanced amount of data

Memory	Amount
Element	500, 1000, 1500, 2000, 3000, 4000, 5000, 6000
MB	0.5, 2, 5, 10, 20, 50, 100

Table 5.1: Size of memory used in the experiments.

for each encountered task. Once the memory is full and a new task becomes available, an equal number of elements are removed for each stored task to free up space for an equal number of samples for the new task. In this way, it is ensured that a fixed number of elements, equally distributed between tasks, are kept in memory. The selection of elements to be retained and removed is done randomly.

Memory has been implemented in two different versions. The first implementation allows storing a fixed number of elements in memory, regardless of the size in bytes of the latent activations. This implementation enables the evaluation of the data needed to retain previous knowledge by varying the layer to which the replay is applied. In contrast, the second implementation takes into account the size of the latent activations, limiting the number of bytes available. In this implementation, a different number of elements is saved based on the replay layer chosen, representing a real-world scenario. In Table 5.1, we report the memory sizes used during the experiments.

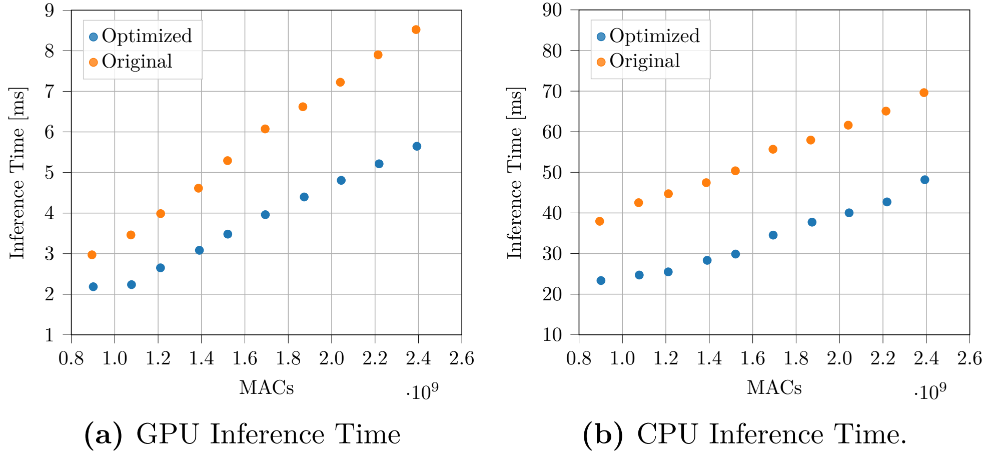
5.1.2 Models

In the experiments, we utilized an optimized version of PhiNet. Specifically, the original PhiNet, based on the hyperparameters configurations, had a number of convolutional channels that were not optimized for use on CPUs and GPUs. Following the approach of MobileNets, we ensured that the channels of the convolutional blocks were divisible by 8 and empirically demonstrated the effectiveness of this modification in improving the inference time. Table 5.2 reports the statistics of a PhiNet in the two versions. Figure 5.1 shows the inference time comparison between the original model and the optimized model as the number of convolutional blocks increases, additional results are reported in Appendix B.1. Interestingly, although the optimized model requires slightly more parameters and operations, it reduces inference time by $\sim 30\%$.

In the following, experiments are conducted with the optimized PhiNet in different configurations, with MobileNetV1 and two versions of MobileNetV2. For PhiNet, we primarily maintained the number of convolutional blocks at

Model	MACs	Params
Original	1,520,210,158	4,394,545
Optimized	1,521,181,866	4,406,997

Table 5.2: Comparison of the original and the optimized PhiNet with $\alpha = 3$, $\beta = 0.75$, $t_0 = 5$, and 7 PhiNet convolutional block.



(a) GPU Inference Time **(b)** CPU Inference Time.

Figure 5.1: Comparison of the inference time between the original model and the optimized model as the number of PhiNet Convolutional Blocks layers increases[1].

the default value of 7. Subsequently, we adjusted the value of α to control the number of input and output channels in the PhiNet convolutional block. A low α value reduces the size of the latent activation stored in memory, while a high value retains more information but decreases the number of storable samples. By changing β and t_0, the number of internal channels in the block, i.e., those obtained after input expansion, varied. A larger value increases the number of transformations applied to the channels, extracting a higher number of features. Additionally, we tested a deeper PhiNet by increasing the number of convolutional blocks to 9.

For MobileNetV1, we only tested the basic model used in previous studies [36, 37]. While, for MobileNetV2, we utilized both the original model and a smaller version with a reduced α, which decreased the size of the input and output channel of the inverted residual block. As a result, it reduces the size of latent activations to be stored.

All the used models were pre-trained using the ImageNet dataset. Table

[1]System specifications: NVIDIA Titan Xp, Intel(R) Core(TM) i7-6800K, 32 GB DDR4

Model	α	β	t_0	MACs	Params	Top1 [%]
PhiNet A	0.8	0.75	8	195.8M	1.7M	64.69
PhiNet B	0.9	0.5	4	100.2M	0.5M	53.66
PhiNet B_9	0.9	0.5	4	123.3M	2.1M	61.86
PhiNet C	1.2	0.5	6	248.8M	1.6M	65.21
MobileNet V1	1	-	-	567.8M	3.3M	70.60
MobileNet V2	1	-	-	312.4M	2.2M	73.01
0.75-MobileNet V2	0.75	-	-	185.4M	1.4M	70.01

Table 5.3: Performance on ImageNet for benchmarked models. For PhiNet, the subscript denotes the number of convolutional blocks in the model, in addition to two convolutions added by default. In cases where it is not specified, the default value of 7 has been applied, as in the original implementation.

	Split CIFAR-10	CORe50
Tasks	5	5
Classes/Task	2	2
Train data/Task	10 000	~23 979
Task Selection	Random	Random

Table 5.4: Experiments benchmark for the class-incremental learning scenario.

5.3 presents a summary of the key models statistics and the notation used for the rest of the chapter.

5.1.3 Benchmarks

The experiments are conducted in a class-incremental learning scenario for image classification using two different datasets: CIFAR-10 and CORe50. The datasets are resized to 224×224 and standardized using ImageNet statistics. To generate the Continual Learning benchmarks, the Avalanche library [30] is used, which provides ready-to-use benchmarks and other utilities to create new ones.

The first selected benchmark is Split CIFAR-10. To construct this dataset, two randomly chosen classes are assigned to each task in a sequence of five consecutive tasks. The second benchmark is based on the CORe50 dataset, specifically using the category-level classification version. Similar to the first dataset, five different tasks with two categories each are used. Table 5.4 reports the main details of the two benchmarks.

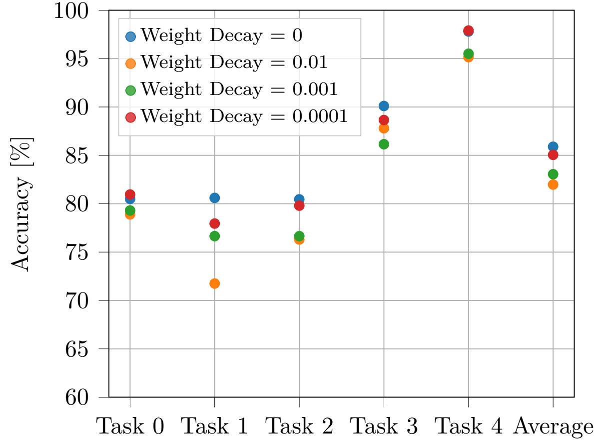


Figure 5.2: Impact of the weight decay on PhiNet A.

5.1.4 Evaluation Metrics

To evaluate the models' performance in our experiments, we focus only on the *Average Accuracy (ACC)*. We measure accuracies on all tasks once the model has completed learning each task. For the sake of simplicity, unless stated otherwise, we report the accuracy at the end of the stream of tasks, i.e. the average of the accuracies achieved on each task after the model has been trained on all tasks. Therefore, considering a stream of T tasks, the ACC introduced in Section 3.3.1 becomes:

$$ACC = \frac{1}{T}\sum_{i=1}^{T} R_{T,i} \tag{5.1}$$

where R is the accuracy matrix. This approach highlights the model's ability to solve all learned tasks at the end of training.

5.1.5 Learning Details

The experiments were conducted using PyTorch 2.0.1 [35] on a single NVIDIA RTX A6000 GPU[2]. We employed a standard Adam optimizer [21] with a constant learning rate. For each model, we performed a manual learning rate search to determine the optimal value.

Following the approach in [17], we empirically found to use very little or no weight decay. Figure 5.2 reports the final accuracies for each task at the

[2]System specifications: AMD Ryzen Threadripper Pro 5995WX (280W), 8x64GB RAM DDR4 3200MHz

Strategy	Epochs	Mini-batch
Fine Tuning	4	128
Multi-Task	20	128
Experience Replay	4	21/107
Latent Replay	4	21/107

Table 5.5: Number of epochs and mini-batch size for each strategy. For rehearsal-based strategies, the mini-batch size is reported as *curr. task*/*past tasks*, where *curr. task* are the elements of the current task and *past task* are the elements of the replay buffer.

end of the learning process with different values of weight decay for PhiNet A in a Latent Replay setting.

We conducted a grid search to determine the ideal number of epochs for the baselines. For the Latent Replay strategy, the training consist of 4 epochs without early stopping to mimic a feasible setting for on-device learning, where storing multiple copies of the weights for retrieval can be unfeasible. Following the approach of [36], for the replay strategies we used a mini-batch size of 128 by concatenating 21 elements from the new task and 107 elements from memory. A summary with the number of training epochs and mini-batch size used for each strategy is reported Table 5.5.

5.2 Experimental Results

In this section, we first show how the number of elements in replay memory affects the performance of Latent Replay. We then present the effectiveness of the strategy employed in retaining knowledge with respect to the considered baselines. Subsequently, we compare the size of latent activations of the considered models as the layer selected for replay varies. Finally, we evaluate the impact of this choice on the performance in a memory-constrained scenario.

In the following, we will refer to the convolutional blocks of each model as $Conv_i$. Depending on the model, it will represent a Depthwise Separable Convolution for MobileNetV1, an Inverted Residual with Linear Bottleneck for MobileNetV2, and a PhiNet Convolutional Block for PhiNet. Additionally, for PhiNet, we will use $StdConv_i$ to denote the first two convolutional blocks that are inserted into the model by default.

5.2.1 Data Efficiency

In this section, we first present the absolute variation in performance as the number of elements in the replay buffer varies for each layer. Subsequently, we show the percentage variation between the accuracy obtained with the maximum value of elements in memory and the values considered in the experiment.

Absolute Performance Variation

In replay-based strategies, the number of elements in the memory has a significant role in retaining knowledge of previous tasks. To assess how the number of elements in the replay buffer impacts performance on the different architectures, we conducted experiments with different memory values: 500, 1000, 1500, 2000, 3000, 4000, 5000, and 6000. Furthermore, we repeated the experiment by changing the layer in which the replay was applied, highlighting how layers closer to the input need more samples in memory to retain the knowledge of past tasks than layers closer to the output.

Figure 5.3 and Figure 5.4 show the average accuracy on all tasks, varying the elements in memory by performing Latent Replay on each layer of the models on CIFAR-10 and CORe50 respectively. As expected, for all the models increasing the number of elements in the buffer results in better accuracy. This effect is particularly evident for layers closer to the input, which, having more free parameters to learn, need to keep more elements in memory to retain previous tasks' knowledge. On the other hand, the

higher expressiveness of the network in these cases allows it to achieve better performance compared to other layers with a sufficient number of elements in the buffer. Even with a large number of elements in memory, the lower layers cannot achieve the same level of accuracy because of their lower expressiveness. As we approach the output layer, we observe a performance saturation as the number of elements increases. However, it should be noted that when the replay buffer contains few samples, the outmost layers exhibit superior performance compared to the inner layers. Therefore, in memory-constrained settings, it appears more convenient to update only the classifier while keeping the feature extractor frozen.

Interestingly, in Figures 5.3 and 5.4, some models exhibit a performance gap between the outmost layer and all the inner layers. This potentially arises because performing replay on the last layer freezes all the weights of the feature extractor, limiting its adaptation to the current task. Therefore, the generality of the extracted features is crucial for achieving high accuracy on new tasks; otherwise, it may result in suboptimal performance. This phenomenon is more evident for CIFAR-10 than for CORe50, possibly due to the greater difficulty of the former. Objects in CORe50 are well distinguished and always represented in the foreground of the image.

Percentage Performance Variation

To better understand the effect of the number of elements in memory on the layers, we calculated the percentage change between the accuracy obtained with the maximum value of elements in memory and the values considered in the experiment in the following way:

$$\Delta_{GAP} = \frac{a_{\ell,6000} - a_{\ell,n}}{a_{\ell,6000}} \cdot 100 \tag{5.2}$$

where $a_{\ell,n}$ is the average accuracy at the layer ℓ with n elements in the replay buffer, and $a_{\ell,6000}$ is the average accuracy at the layer ℓ with 6000 elements in the replay buffer.

Figures 5.5 and 5.6 shows the Δ_{GAP} for each layer of the considered models. We can see that in the first layers, the performance gap between the minimum and maximum value of elements in memory leads to a percentage change of $\sim 20\%$. While, for the last layer, this difference is around $\sim 3\%$, showing that the inner layers are more sensitive to the variation of elements in the memory buffer. Moreover, when considering individual layers, we can see that as the number of elements in memory increases, there is a decrease in the percentage change, eventually reaching a point of performance saturation.

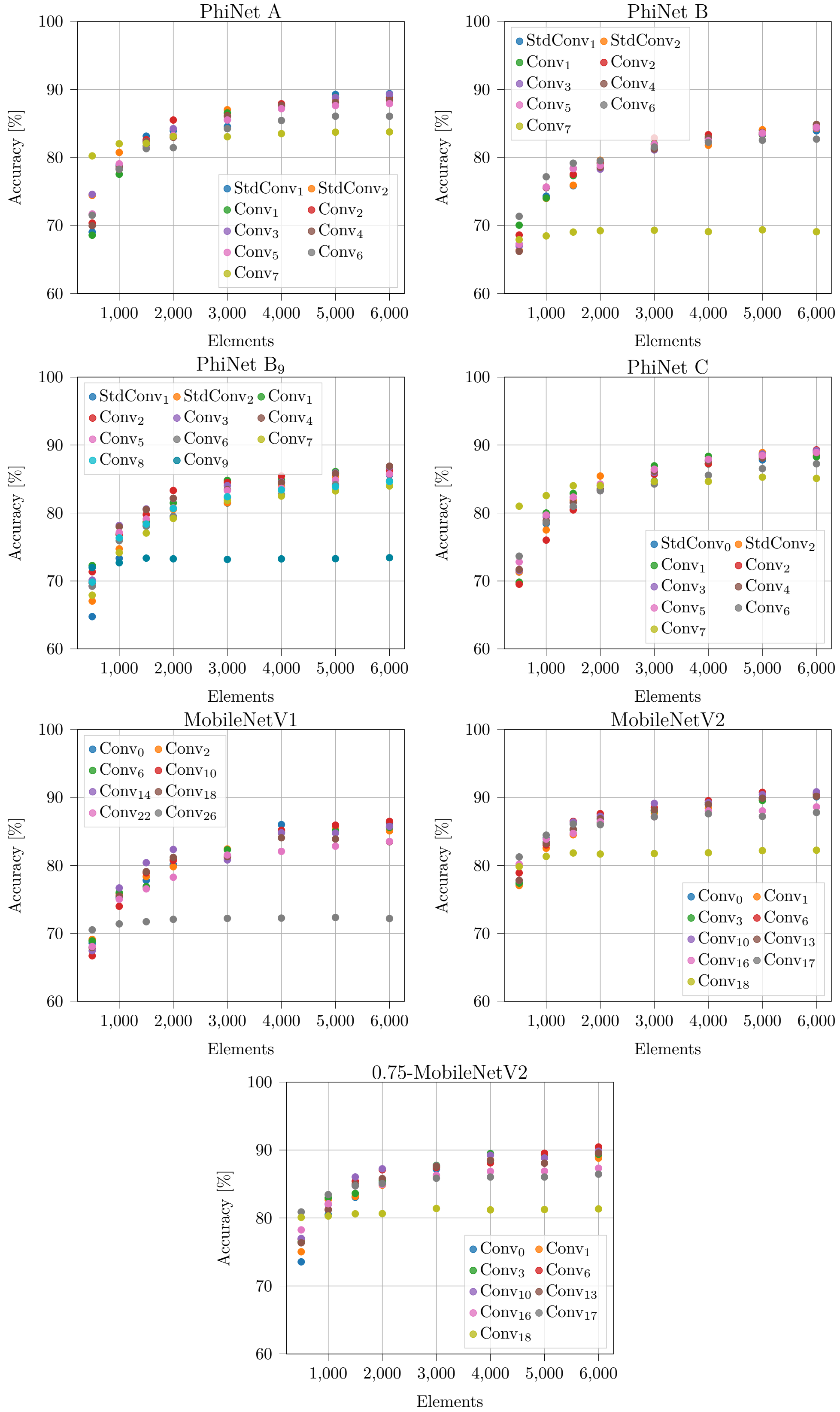


Figure 5.3: Impact of different number of elements in memory on the different architectures using CIFAR-10.

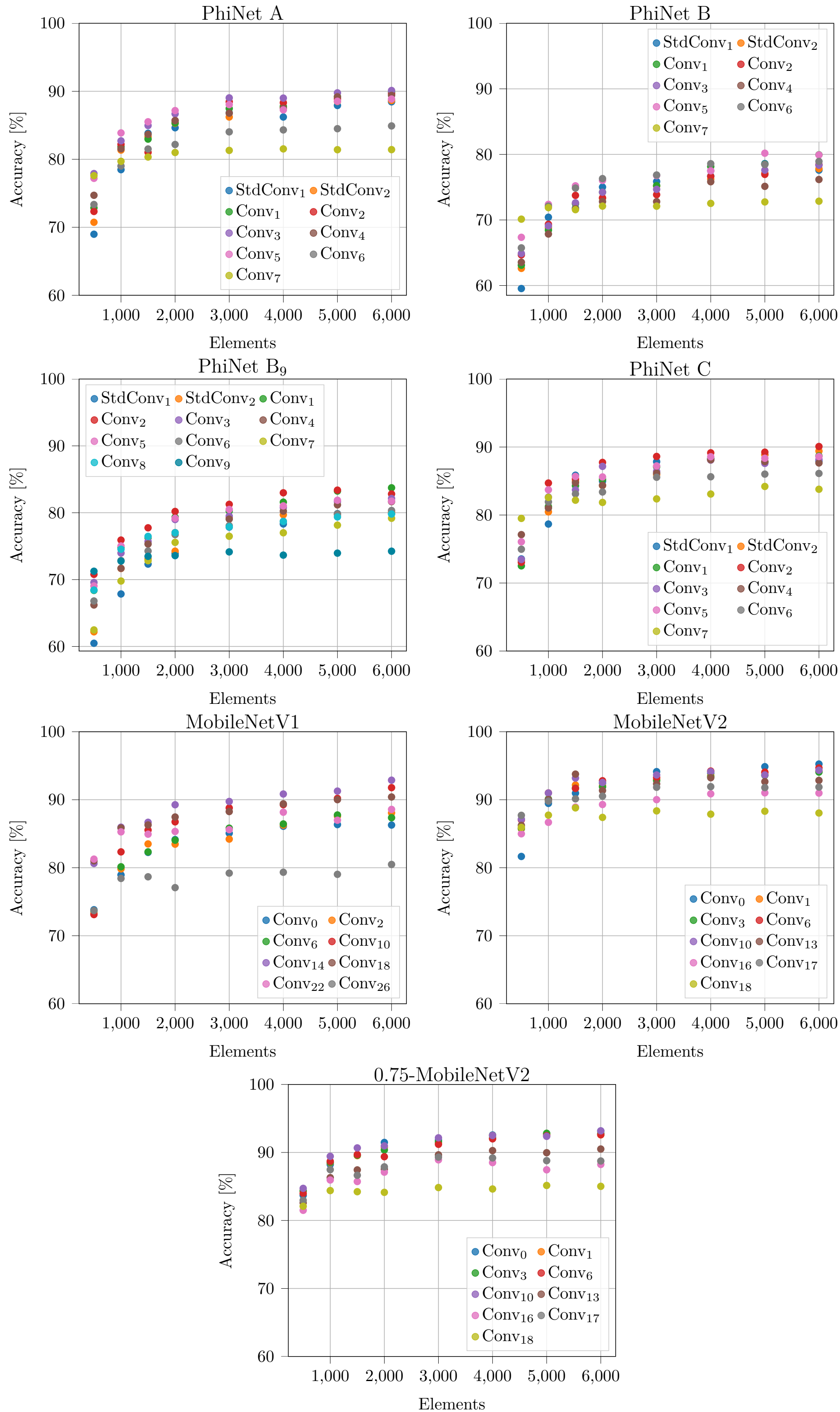


Figure 5.4: Impact of different number of elements in memory on the different architectures using CORe50.

This trend is particularly evident in the final layer, where the percentage variation approaches zero for higher values of elements in memory.

The trend lines confirm this behaviour. In particular, for small memory values, the decreasing trend of the performance gap moving toward the outer layers confirms the need for more elements for the innermost layers. For high memory values, on the other hand, the lines tend to become constant, thus showing that there is a trend toward performance saturation for all layers.

Intuitively, as the number of elements in the memory buffer increases, all layers can be expected to reach a point of performance saturation. Also, we can expect that, with a sufficient number of elements in memory, a network with more expressiveness, and thus fewer frozen layers, will achieve better performance with respect to a simpler model. In the ideal case, with a sufficient number of elements in memory, replaying from input could yield the best performance, and as more layers are frozen, the performance could decrease.

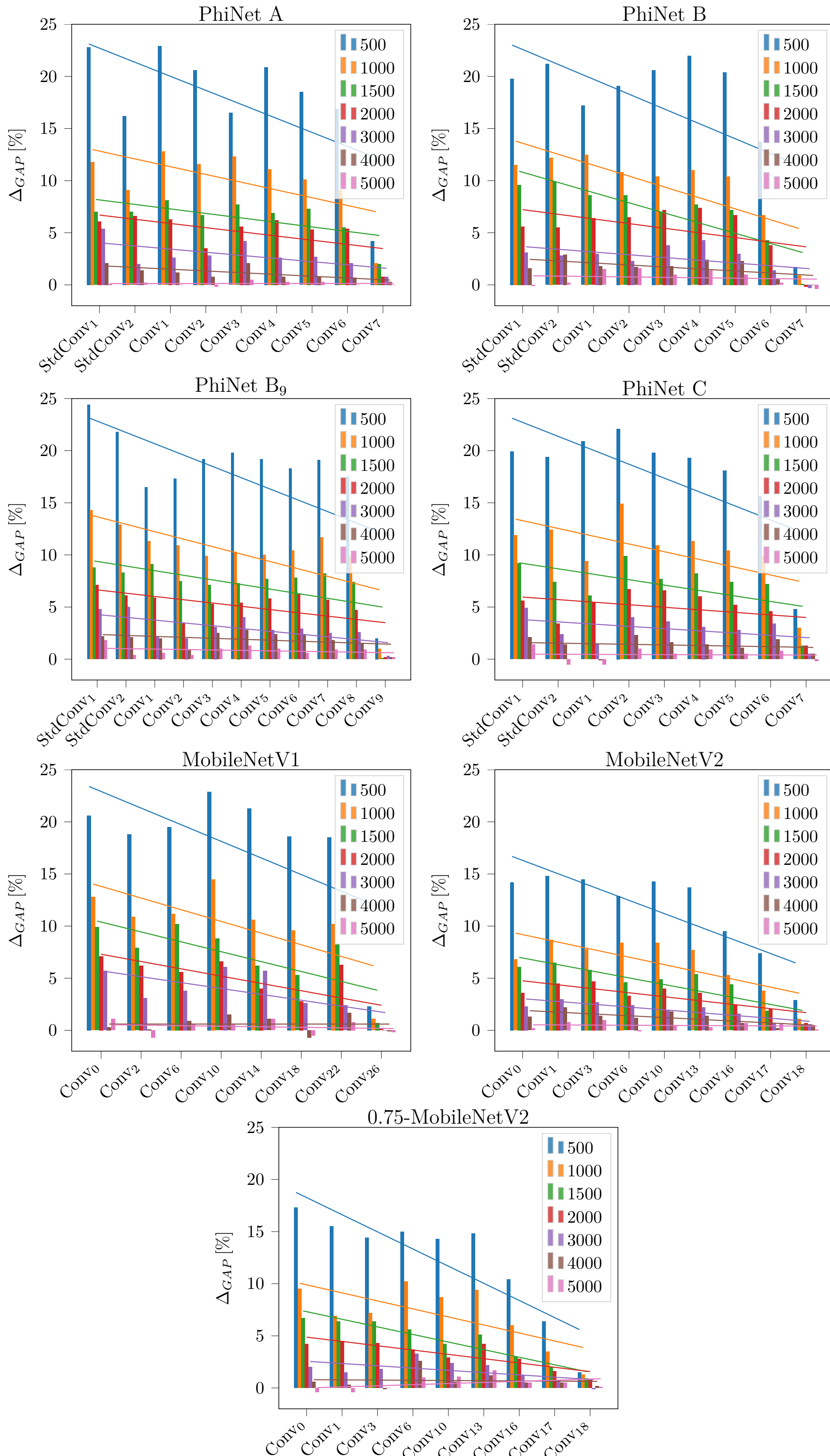


Figure 5.5: Percentage change in CIFAR-10 performance across different model layers as the number of memory elements varies.

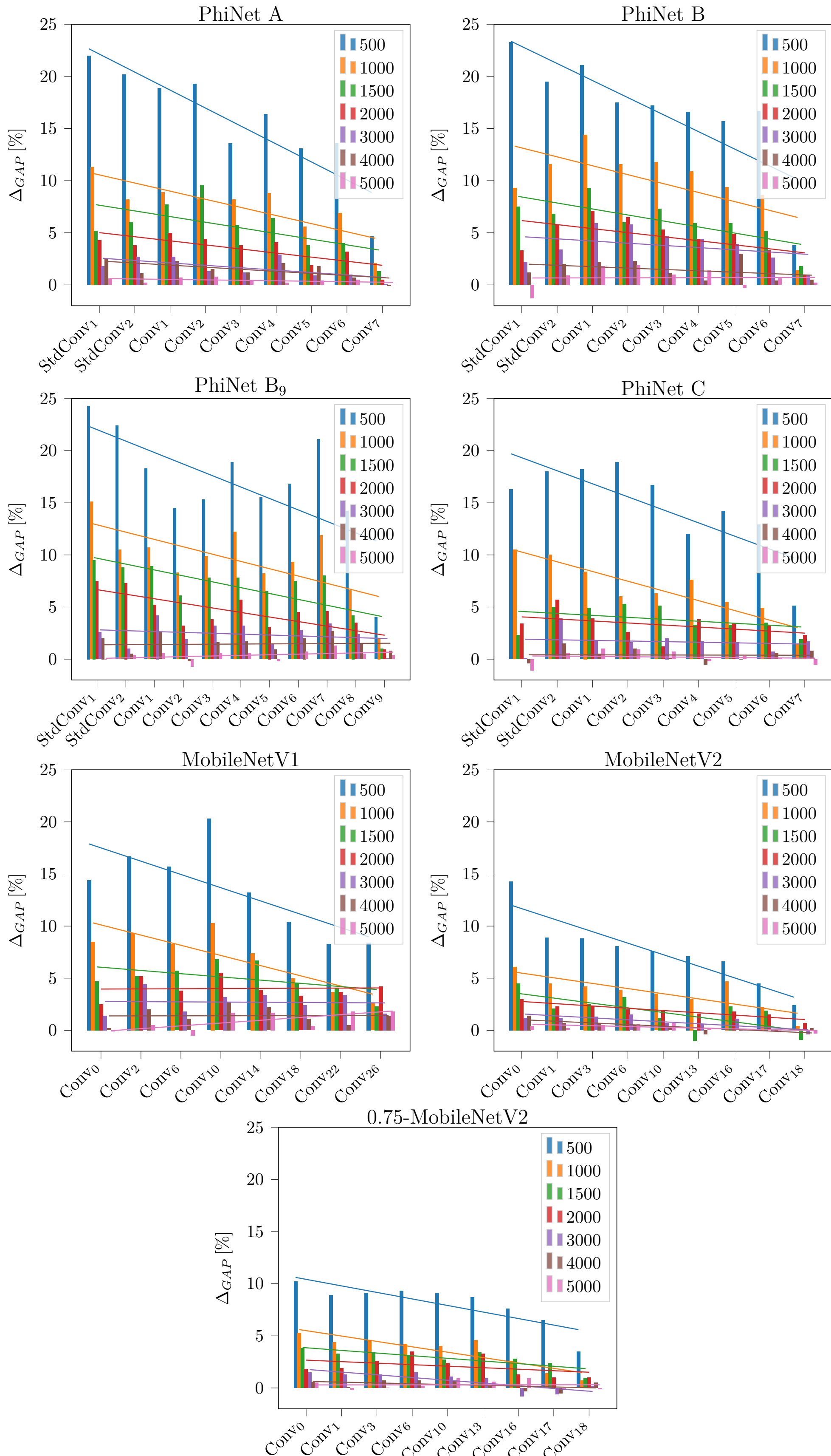


Figure 5.6: Percentage change in CORe50 performance across different model layers as the number of memory elements varies.

5.2.2 Strategies Evaluation

In this section, we initially compare the performance between Latent Replay and the baseline methods under consideration. We also report a metric to measure the similarity between ideal performance and empirically obtained performance. Subsequently, we conduct a detailed examination of the performance of Latent Replay and Experience Replay by considering not only accuracy but also the trade-off between computation, memory, and accuracy.

Performance Comparison

Figure 5.7 and Figure 5.8, respectively, on CIFAR-10 and CORe50, present a comparison showing the performance of the Latent Replay strategy against the three baselines discussed in Section 5.1.1. The comparison focuses on the evolution of the average accuracy on all tasks once the models have completed learning each task. Following the approach of [36], we use a replay buffer with a fixed amount of 1500 elements for the rehearsal methods.

As a reference, it is important to highlight the upper bound obtained when training offline on all the tasks with the Multi-Task strategies. As expected, the approach improves the generalization, and for all the considered models, it leads to the best performances. On the other hand, the Fine-Tuning baseline suffers from catastrophic forgetting, reaching generally an average accuracy of 20%.

In terms of the replay strategies, we considered the replay from input with the Experience Replay and two layers for Latent Replay. Specifically, we reported the layer that achieved the best performance in these settings and the last layer, i.e., leaving only the classification head free to learn. Using these strategies leads to a performance difference with respect to the upper bound that depends on the model and the layer used for replay, but it reaches a minimum of $\sim 7\%$ in the best cases and a maximum of $\sim 20\%$ in the worst cases. In Tables 5.6 and 5.7, we report the average accuracy values for each model in each of the strategies used.

It is worth noting that the Experience Replay strategy does not always outperform the Latent Replay strategy. As discussed in Section 5.2.1, in the ideal case with a sufficient number of samples in memory, optimal performance could be achieved by performing the replay from the input, and then decreasing as the layers freeze moving toward the output. Figures 5.7 and 5.8 confirm this behaviour for PhiNet. However, MobileNet achieves optimal performance by performing replay in inner layer. This can be explained by assessing the distance from the ideal case as the number of elements in the memory varies for the different models.

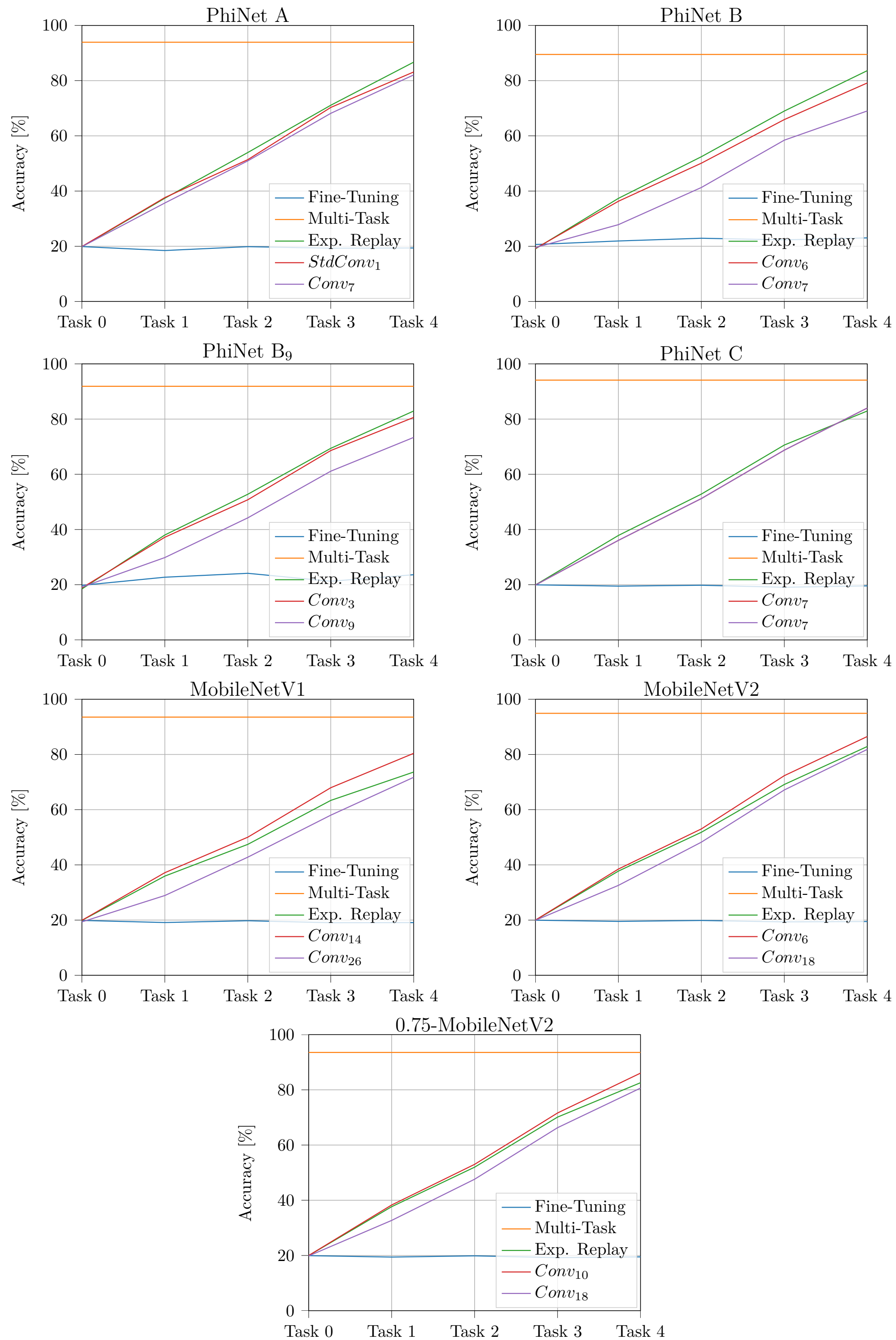


Figure 5.7: Accuracy results on CIFAR-10 of Fine-Tuning, Multi-Task, Experience Replay and Latent Replay.

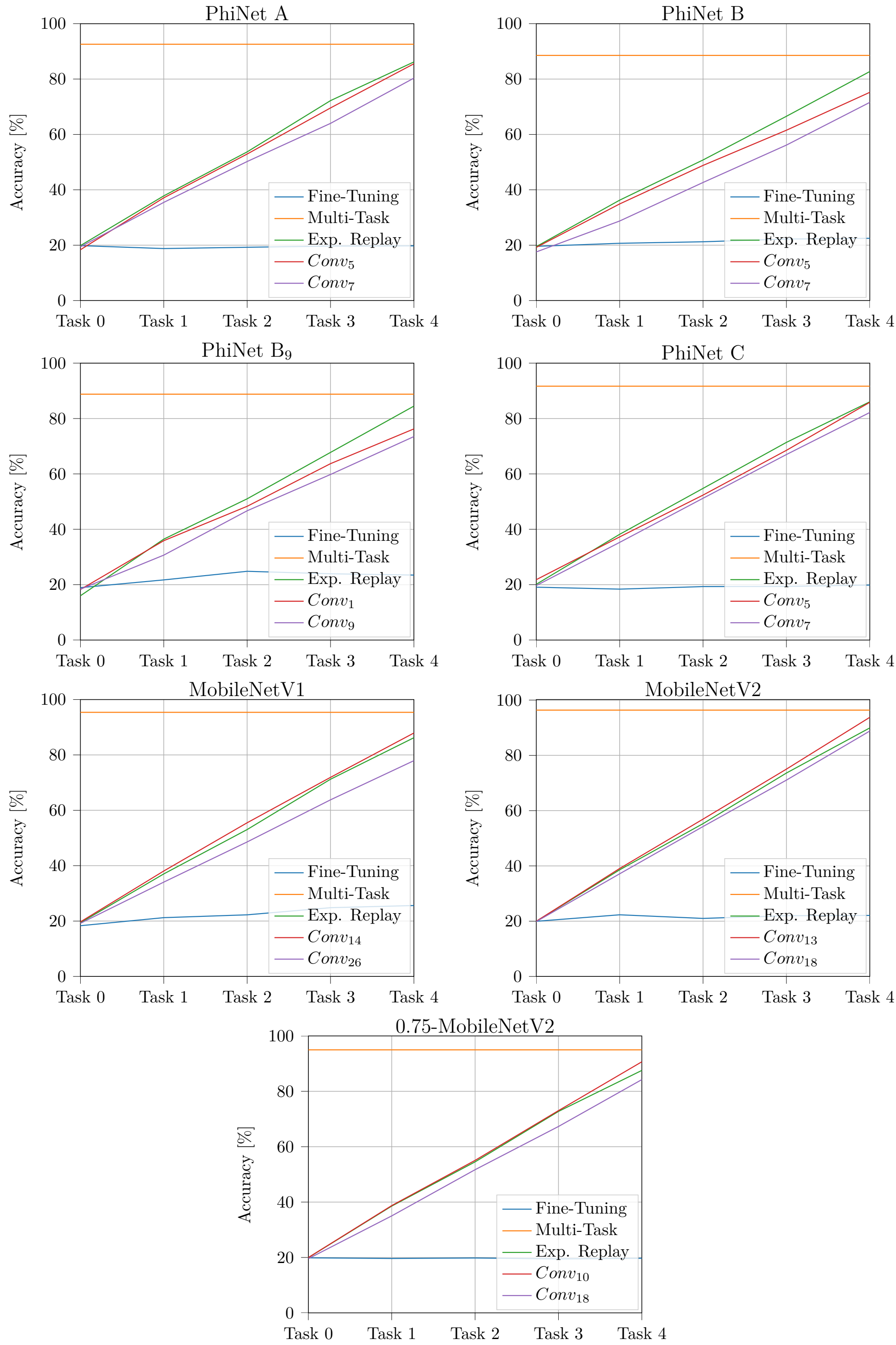


Figure 5.8: Accuracy results on CORe50 of Fine-Tuning, Multi-Task, Experience Replay and Latent Replay.

Model	Average Accuracy [%]				
	Fine-Tuning	**Multi-Task**	**Exp. Replay**	**Best Layer**	**Last Layer**
PhiNet A	19.38	93.91	86.70	83.14	82.08
PhiNet B	23.04	89.49	83.62	79.16	69.02
PhiNet B_9	23.62	91.88	82.93	80.60	73.36
PhiNet C	19.58	94.11	82.93	84.01	84.01
MobileNetV1	19.08	93.53	73.63	80.41	71.73
MobileNetV2	19.48	94.90	82.88	86.50	81.84
0.75-MobileNetV2	19.49	93.55	82.57	86.05	80.62

Table 5.6: Final accuracy on CIFAR-10 for each strategy.

Model	Average Accuracy [%]				
	Fine-Tuning	**Multi-Task**	**Exp. Replay**	**Best Layer**	**Last Layer**
PhiNet A	19.76	92.56	86.18	85.52	80.34
PhiNet B	22.49	88.54	82.71	75.21	71.57
PhiNet B_9	23.46	88.79	84.48	76.25	73.48
PhiNet C	19.86	91.98	86.04	85.85	82.18
MobileNetV1	25.61	95.38	86.21	87.92	77.92
MobileNetV2	22.08	96.37	89.94	93.77	88.80
0.75-MobileNetV2	19.70	94.98	87.55	90.67	84.23

Table 5.7: Final accuracy on CORe50 for each strategy.

Similarity Measure

To understand how far we deviate from the ideal case, we introduce the similarity measure S_n based on the Kendall Tau Rank Distance K_n [6, 20]. This metric calculates the number of pairs that are in different order in two lists. The greater the distance, the more dissimilar the two lists are, more details are reported in Appendix A.

We can define the ideal case with the following list:

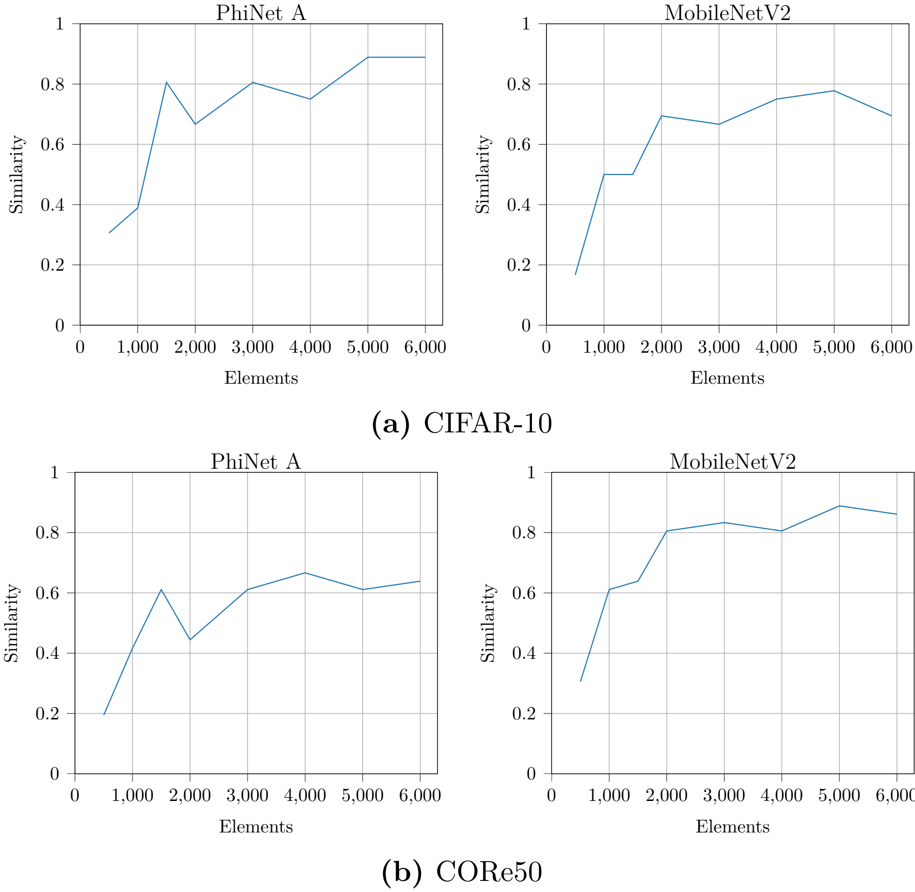
$$[Input, Conv_1, \ldots, Conv_i, \ldots, Conv_n] \tag{5.3}$$

where, with a sufficient number of samples, $Input$ is the layer on which we expect to have the best performance and $Conv_n$ the one with the worst performance.

For each memory size used, we can construct a list sorted by layer performance to assess the distance to the ideal case for each model. In particular, we assess the similarity to the ideal case in the following way:

$$S_n = 1 - K_n \tag{5.4}$$

a high value indicates that the number of samples in the replay buffer is enough to have a performance close to the ideal case. Conversely, a low



(a) CIFAR-10

(b) CORe50

Figure 5.9: Similarity to the ideal case for PhiNet A and MobileNetV2 computed with the Kendall tau distance.

value indicates an insufficient number of samples to reach the ideal case, potentially leading to better performance in outer layers compared to inner layers. It is important to highlight that a low value does not necessarily imply inferior performance. Instead, it means that optimal performance is attained by using internal layers.

Figure 5.9 shows the similarity values for PhiNet A and MobileNetV2. At a memory size of 1500, PhiNet exhibits a value of 0.8, indicating we are close to the ideal case and the Experience Replay could outperform the inner layer. Conversely, MobileNetV2 displays a similarity value around 0.5, suggesting that, in this case, the performance of the inner layer could surpass that of the replay from input, confirming previous findings. A complete assessment of similarity measures is reported in the Appendix B.2.

Computation, Storage and Accuracy Trade-off

To conduct a thorough evaluation of Latent Replay in comparison to Experience Replay, Table 5.8 and 5.9 present relevant dimensions, including: (1) Computation refer to the percentage cost of MAC operation to be performed after the layer relative to a full forward step from the input, (2) Latent Size is the dimension of the sample to be stored in the memory, and (3) the Δ Accuracy represent the difference with respect to the Experience Replay.

Remarkably, when restricting the learning to only the classifier, all models exhibit good performance. In such cases, the computational cost becomes negligible compared to replay, while still maintaining high performance levels. An exception is PhiNet B, experiencing in the worst case a loss of $\sim 14\%$, probably attributed to the suboptimal quality of the extracted features. This model achieved an accuracy of around $\sim 53\%$ on ImageNet.

From Tables 5.8 and 5.9, it is evident that replay from the input generally does not seem to be advantageous. The increase in accuracy is not substantial, and, conversely, it significantly increases computational costs. While the performance of Experience Replay could be enhanced by augmenting the number of samples in memory, this approach becomes impractical for edge applications due to the prohibitive cost associated with storing additional elements.

Model	Layer	Computation [%]	Latent Size [MB]	Avg. Acc.[%]	Δ Acc. [%]
PhiNet A	Input	100.00	215.33	86.70	0.00
	2	99.059	2871.09	83.14	-3.56
	10	0.002	27.83	82.08	-4.62
PhiNet B	Input	100.00	215.33	83.62	0.00
	9	5.056	49.35	79.16	-4.46
	10	0.003	31.49	69.02	-14.60
PhiNet B_9	Input	100.00	215.33	82.93	0.00
	6	36.064	179.44	80.60	-2.33
	12	0.003	31.49	73.36	-9.57
PhiNet C	Input	100.00	215.33	82.93	0.00
	10	0.002	42.48	84.01	1.08
	10	0.002	42.48	84.01	1.08
MobileNetV1	Input	100.00	215.33	73.63	0.00
	14	50.538	574.22	80.41	6.78
	26	0.009	287.11	71.73	-1.90
MobileNetV2	Input	100.00	215.33	82.88	0.00
	6	59.631	143.55	86.50	3.62
	18	0.004	358.89	81.84	-1.04
0.75-MobileNetV2	Input	100.00	215.33	82.57	0.00
	10	47.296	53.83	86.05	3.48
	18	0.007	358.89	80.62	-1.95

Table 5.8: Computation, storage, and accuracy trade-off with replay strategies for all the models on CIFAR-10. For each model 3 layers are shown, from input, the inner layer with best performance and the final layer close to the output.

Model	Layer	Computation [%]	Latent Size [MB]	Avg. Acc.[%]	Δ Acc. [%]
PhiNet A	Input	100.00	215.33	86.18	0.00
	8	13.560	42.62	85.52	-0.66
	10	0.002	27.83	80.34	-5.84
PhiNet B	Input	100.00	215.33	82.71	0.00
	8	12.167	49.35	75.21	-7.51
	10	0.003	31.49	71.57	-11.14
PhiNet B_9	Input	100.00	215.33	84.48	0.00
	4	56.202	430.66	76.25	-8.23
	12	0.003	31.49	73.48	-11.00
PhiNet C	Input	100.00	215.33	86.04	0.00
	2	99.017	4019.53	85.85	-0.19
	10	0.002	42.48	82.18	-3.86
MobileNetV1	Input	100.00	215.33	86.21	0.00
	14	50.538	574.22	87.92	1.71
	26	0.009	287.11	77.92	-8.29
MobileNetV2	Input	100.00	215.33	89.94	0.00
	13	28.703	107.67	93.77	3.83
	18	0.004	358.89	88.80	-1.13
0.75-MobileNetV2	Input	100.00	215.33	87.55	0.00
	10	47.296	53.83	90.67	3.12
	18	0.007	358.89	84.23	-3.32

Table 5.9: Computation, storage, and accuracy trade-off with replay strategies for all the models on CORe50. For each model 3 layers are shown, from input, the inner layer with best performance and the final layer close to the output.

5.2.3 Memory Evaluation

In order to compare the size of latent activations of PhiNet and MobileNet as the layer selected for replay varies. We measure the size of the latent activation for each layer by considering a standard input image of size $3 \times 224 \times 224$, and the tensors were stored in FP32 format. To assess and compare the layers of different architectures, we used the MACs from the replay layer, representing the operations to be performed after it, as this also indicates the latency of the backward step.

From Figure 5.10, distinct trends between the architectures emerge. PhiNet, with fewer layers, exhibit a rapid increase in latent activation size. This is due to the faster increase in latent activation resolution size as one approaches the input. In contrast, MobileNetV2 generally maintains a lower resolution, resulting in a smaller latent activation size for the same number of MACs, with an increase only in the last few layers. However, the focus on keeping only the last layers free is fundamental for edge devices applications since they require much less computation.

As shown in Figure 5.11, a crucial advantage of PhiNet is achieved by selecting the last convolutional block as the latent layer. This results in a significant reduction in the number of operations required during training and minimizes the size of latent activations. This optimization allows for efficient use of computational resources but also results in significant memory savings in the replay buffer, useful when memory is limited. In contrast, MobileNetV2 uses a fixed number of channels in the last convolutional layer. Thus, selecting the last layer increases the size of latent activations, considerably reducing the number of storable samples and compromising the update times for the network. This limits the performance of that layer for replay. However, it should be noted that the layers before the last have a small memory footprint but increase the number of MACs significantly. This allows to improve the performance of MobileNetV2 but at the cost of many operations.

As for MobileNetV1, it is not apparent which is the most convenient layer to extract the Latent Replay. The latent activations of the convolutional blocks turn out to be larger than the size of the single input image. Therefore, it is more convenient to freeze the feature extractor to the desired layer and perform a direct replay from the input. This procedure is more costly than performing Latent Replay using PhiNet or MobileNetV2.

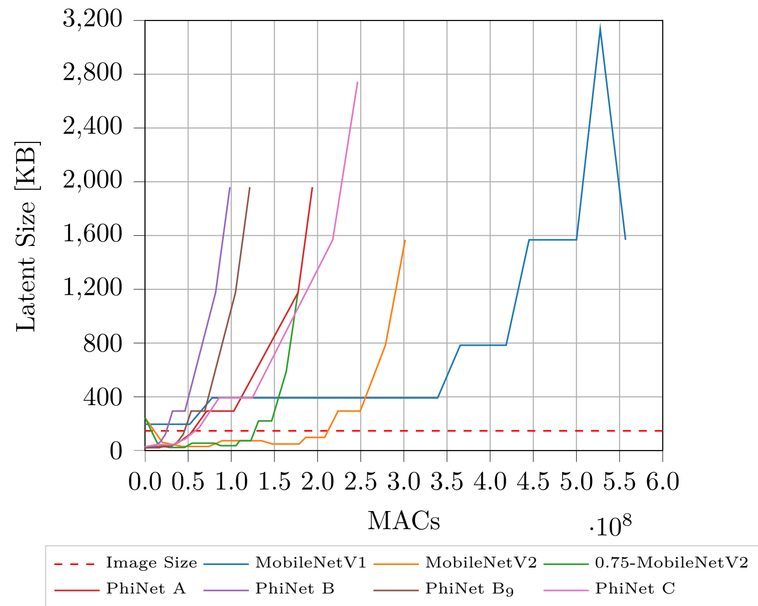


Figure 5.10: Comparison of the latent activation size and MAC of the selected layer between different models. The dimensions refer to a single sample.

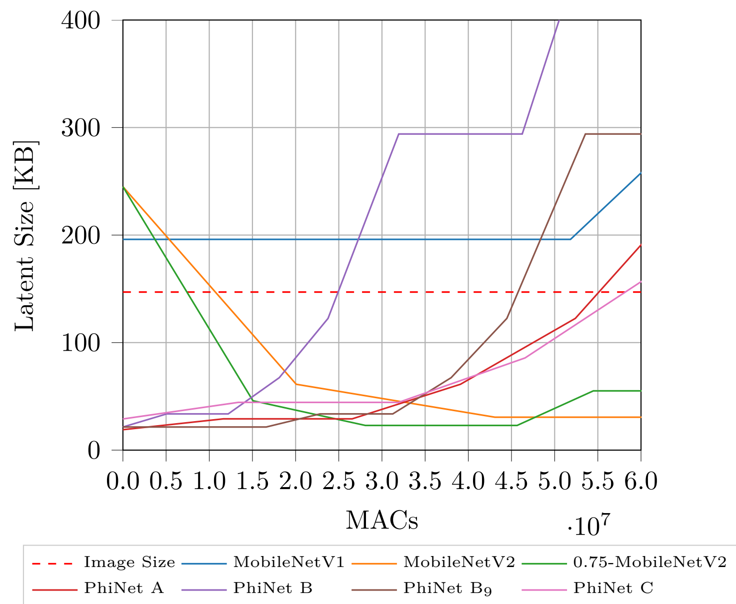


Figure 5.11: Magnified comparison of latent activation size and MAC of the selected layer between different models. The dimensions refer to a single sample.

5.2.4 Memory-Performance Trade-off Evaluation

In Section 5.2.1, we conducted experiments by maintaining a constant number of storable elements in the replay buffer. This approach allowed us to understand how the behaviour of the models varies as the number of stored elements changes. However, it is important to note that this approach does not represent a real-world scenario where memory constraints would be in terms of available bytes, rather than storable elements. Therefore, in this section, we assess how a fixed dimension memory impacts the performances of the different architectures. We conducted experiments with different memory sizes: 0.5MB, 2MB, 10MB, 20MB, 50MB, and 100MB.

Figure 5.12 and Figure 5.13 show the results of experiments conducted with fixed Latent Replay memory in terms of bytes. As expected, for small memory values, PhiNet outperforms MobileNet by choosing the last layer of the feature extractor for replay. The smaller size of the latent activations for PhiNet enables storing more elements in the replay memory, thus significantly improving performance.

Moving towards the innermost layers, we see how the performance of PhiNet degrades, while those of MobileNetV2 grow to a maximum and then decrease. However, the performance of MobileNetV2 does not achieve the best performance of PhiNet, even in the inner layers and with a much higher number of operations. The differences in performance trends can still be attributed to the sample size in memory. As we approach the input, the size of PhiNet's latent activations grows faster than those of MobileNetV2, as discussed in Section 5.2.3 (Figure 5.10). Hence, the number of samples that can be stored in memory are insufficient to guarantee good performance. On the other hand, MobileNetV2 keeps its activations small until the first few layers of the model. Initially, the performance improves, but moving toward the input, despite the small size of the latent activations, the stored samples are insufficient to maintain knowledge of previous tasks. These findings confirms that the inner layers need more memory samples than the last layers to retain memory. As for MobileNetV1, it can be seen that the performance settles around 20%, demonstrating the inefficiency of this model in situations of limited memory.

For larger values of memory, and thus for a larger number of available samples, PhiNets' performance for layers closer to the output are comparable to that of MobileNetV2. Whereas, for the innermost layers MobileNetV2 outperforms PhiNets, which as mentioned can store less samples due to the latent activation size.

Figure 5.14 and Figure 5.15 show the performance of different layers of each model as the memory size changes. The application of Latent Replay

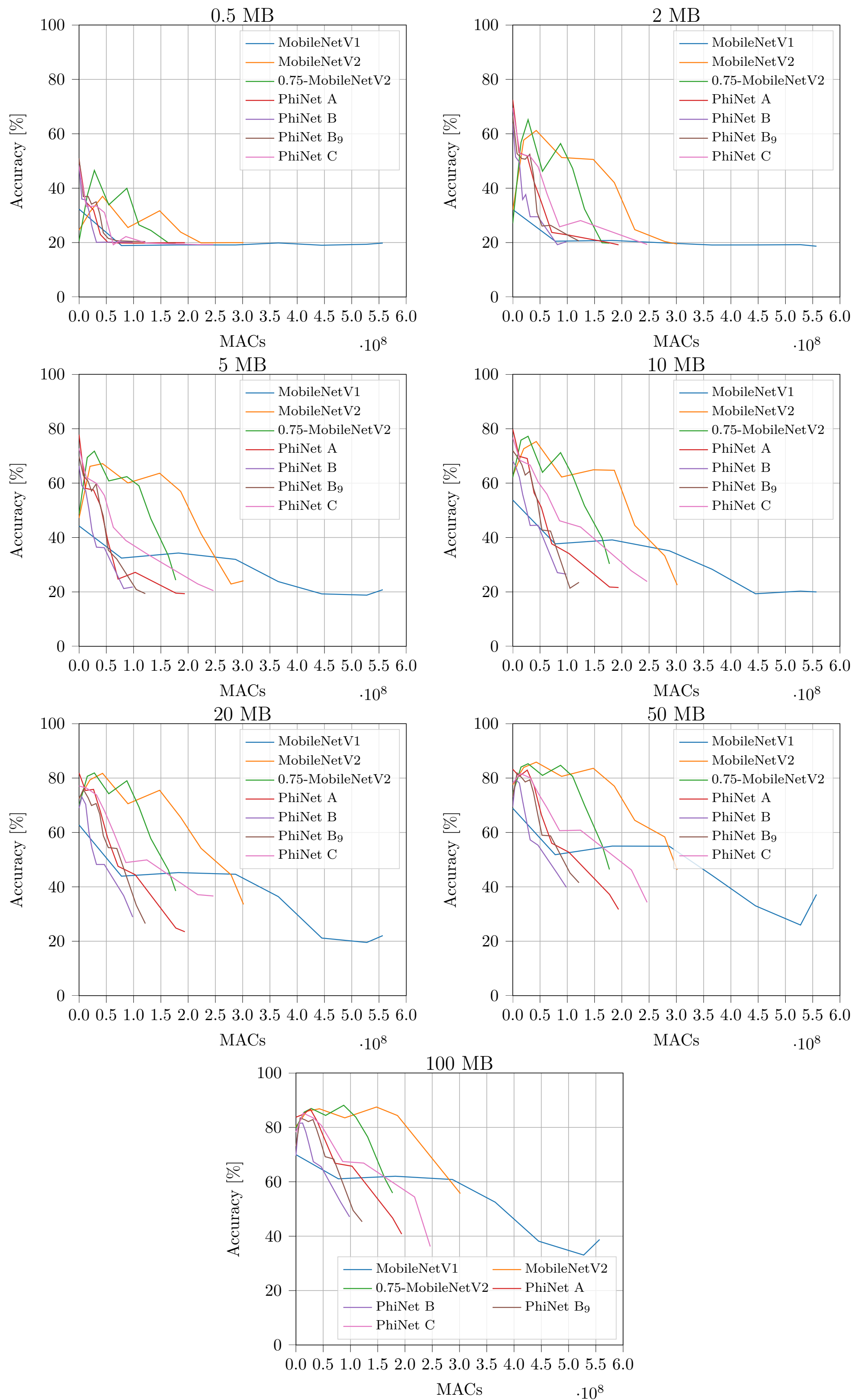


Figure 5.12: Performance comparison on CIFAR-10 dataset using different models and fixed maximum value for the replay buffer.

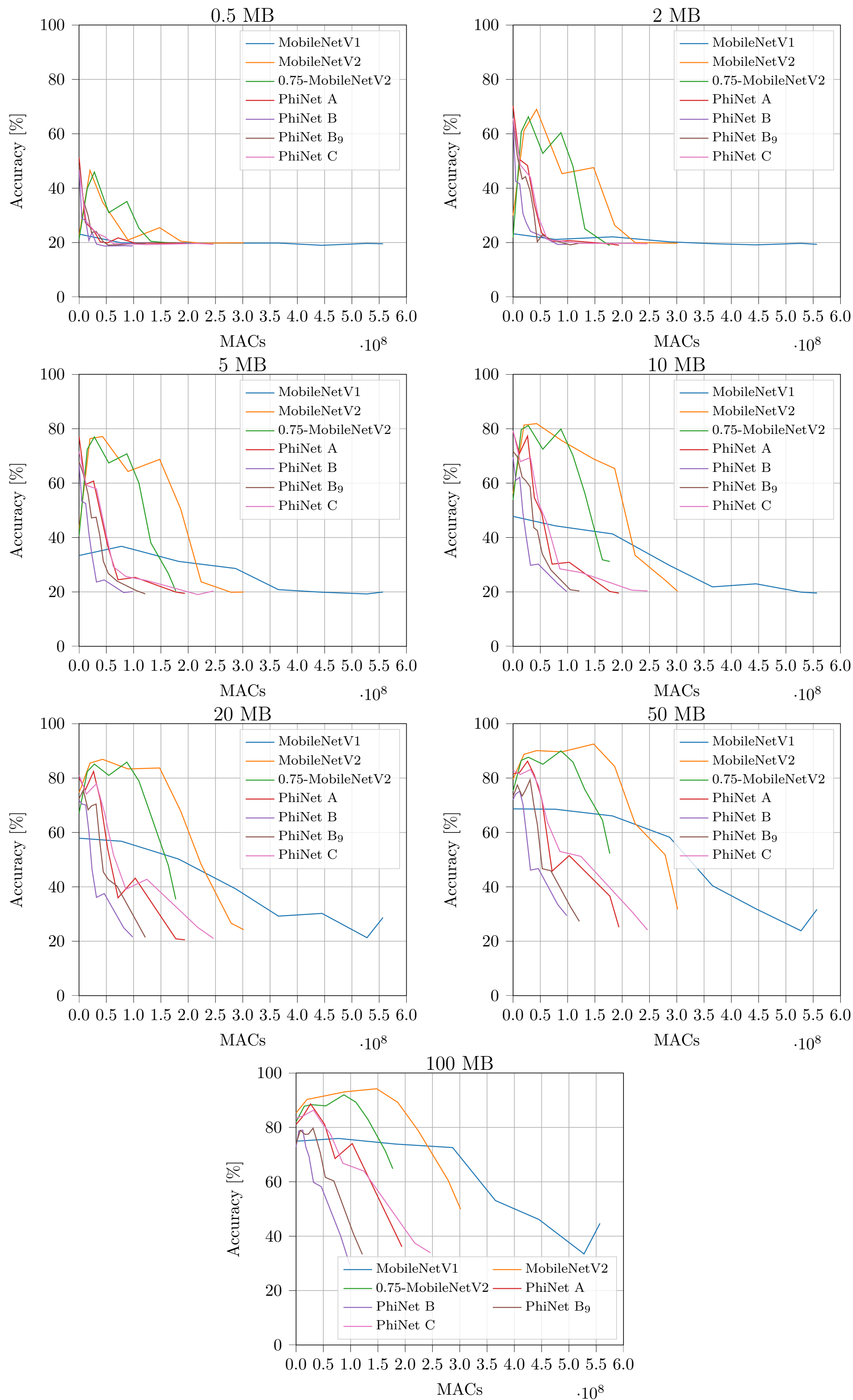


Figure 5.13: Performance comparison on CORe50 dataset using different models and fixed maximum value for the replay buffer.

on internal layers leads to catastrophic forgetting under limited memory conditions for all the evaluated models, making it an unsuitable choice in these applications. This phenomenon arises from the increasing size of activations, consequently, fewer elements are stored in the buffer. As detailed in Section 5.2.1, internal layers with a small number of elements in memory fail to retain knowledge of past tasks. In the extreme case of 0.5 MB, few or no elements are stored in memory in these layers, reducing the Latent Replay strategy to the base case of fine-tuning the current task. As a result, an average accuracy of $\sim$ 20% is achieved, demonstrating complete forgetting of previous tasks. For larger memory sizes, performance improves slightly since more elements can be stored in memory. However, even with a 100MB memory capacity, layers closer to the input cannot store enough elements to maintain the memory of past tasks.

Table 5.10 presents the optimal results achieved by each model as the replay memory size varies. As previously highlighted, for limited memory values, PhiNet outperforms MobileNet. Specifically, with memory limited to 0.5MB, PhiNet A improves the performance of 0.75-MobileNetV2 by 3.72% on CIFAR-10 (5.49% on CORe50), requiring only 0.011% of the operations performed by MobileNet on both datasets. The balance between performance and efficiency makes PhiNet superior up to 20MB of replay buffer.

However, for higher memory values, the last layer of PhiNet seems to be no longer the optimal choice, possibly due to the performance saturation of that layer, visible also in Figures 5.14 and Figure 5.15, where for high memory values the performances in the first layer are approximately equal. Nevertheless, moving toward the inner layers, PhiNet fails to achieve the performance of MobileNet, which shows 0.8% higher accuracy on CIFAR-10 (5.59% on CORe50), while requiring 82% more operations on both datasets.

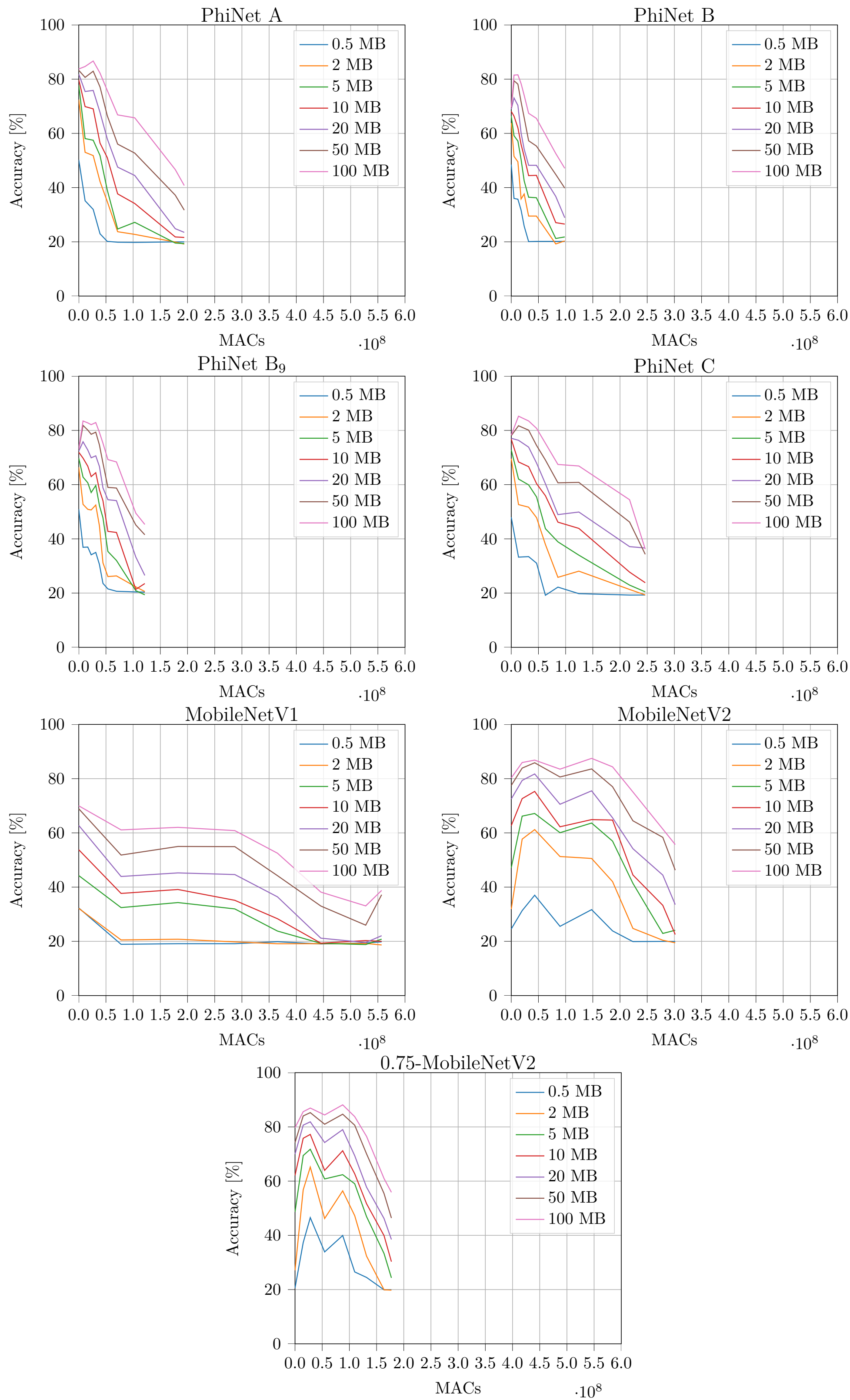


Figure 5.14: Evaluation of the different architectures on CIFAR-10 using different maximum values for the replay buffer.

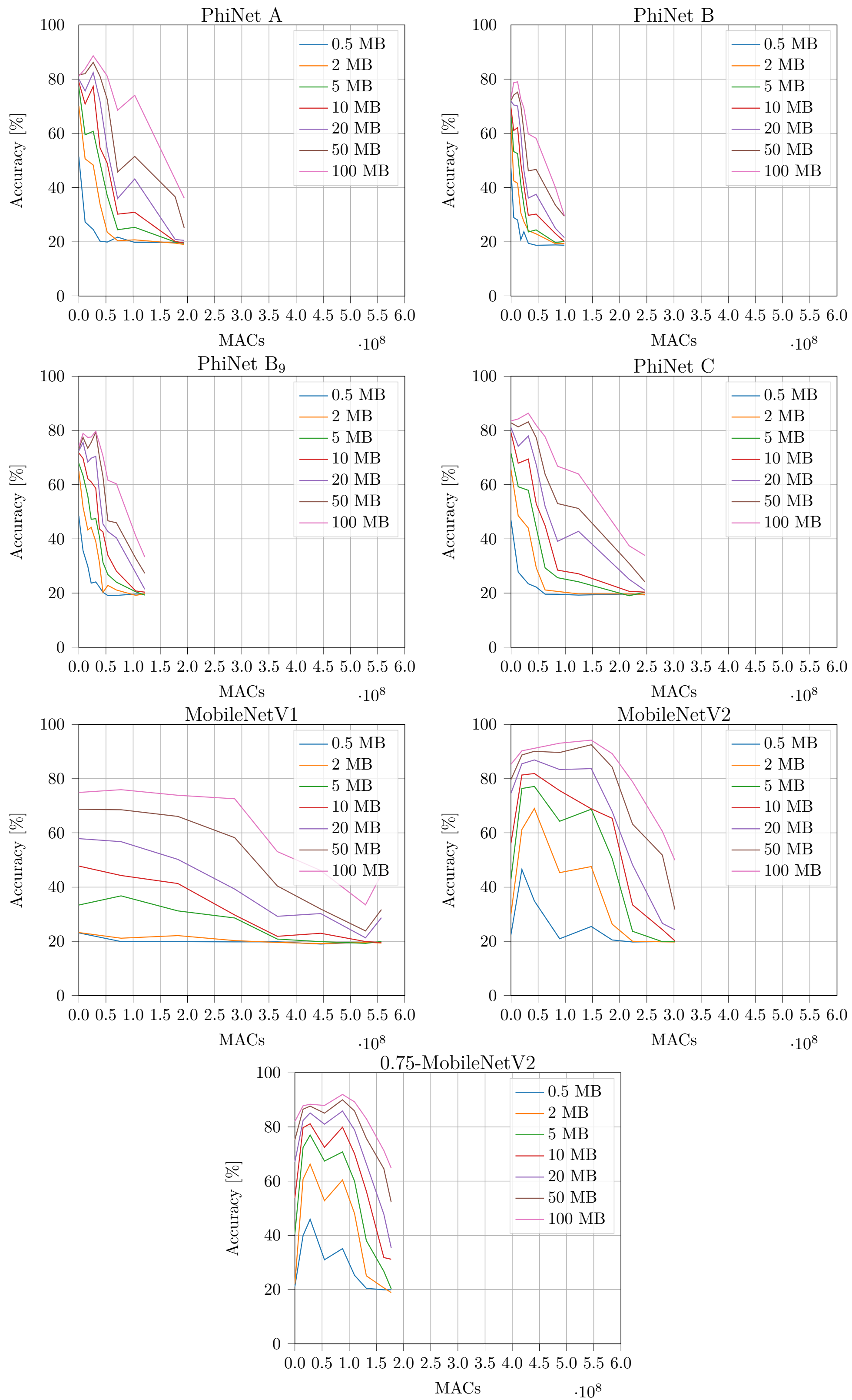


Figure 5.15: Evaluation of the different architectures on CORe50 using different maximum values for the replay buffer.

Model	Memory [MB]	CIFAR10 Avg. Acc. [%]	CIFAR10 MACs	CORe50 Avg. Acc. [%]	CORe50 MACs
PhiNet A	0.5	50.25	3,050	51.42	3,050
PhiNet B		48.48	3,450	47.19	3,450
PhiNet B_9		51.00	3,450	48.51	3,450
PhiNet C		47.96	4,650	46.95	4,650
MobileNetV1		32.29	51,200	23.11	51,200
MobileNetV2		37.01	43,092,800	46.51	20,085,760
0.75 MobileNetV2		46.53	28,089,840	45.93	28,089,840
PhiNet A	2	72.49	3,050	70.20	3,050
PhiNet B		63.83	3,450	63.73	3,450
PhiNet B_9		66.45	3,450	65.10	3,450
PhiNet C		69.58	4,650	65.65	4,650
MobileNetV1		32.15	51,200	23.24	51,200
MobileNetV2		61.21	43,092,800	68.99	43,092,800
0.75 MobileNetV2		65.18	28,089,840	66.26	28,089,840
PhiNet A	5	77.82	3,050	77.23	3,050
PhiNet B		66.34	3,450	67.28	3,450
PhiNet B_9		69.80	3,450	67.92	3,450
PhiNet C		72.78	4,650	71.68	4,650
MobileNetV1		44.24	51,200	36.77	77,806,080
MobileNetV2		67.16	43,092,800	77.14	43,092,800
0.75 MobileNetV2		71.76	28,089,840	77.00	28,089,840
PhiNet A	10	79.83	3,050	79.04	3,050
PhiNet B		68.11	3,450	69.58	3,450
PhiNet B_9		71.95	3,450	71.69	3,450
PhiNet C		76.77	4,650	79.26	4,650
MobileNetV1		53.77	51,200	47.76	51,200
MobileNetV2		75.30	43,092,800	81.89	43,092,800
0.75 MobileNetV2		77.23	28,089,840	81.15	28,089,840
PhiNet A	20	81.88	3,050	82.41	26,554,002
PhiNet B		73.06	5,068,714	71.90	3,450
PhiNet B_9		75.82	7,833,578	75.56	7,833,578
PhiNet C		77.10	4,650	81.10	4,650
MobileNetV1		62.74	51,200	57.88	51,200
MobileNetV2		81.74	43,092,800	86.90	43,092,800
0.75 MobileNetV2		81.88	28,089,840	85.85	87,691,008
PhiNet A	50	83.27	3,050	86.19	26,554,002
PhiNet B		79.41	5,068,714	75.19	12,197,466
PhiNet B_9		81.86	7,833,578	79.37	31,255,778
PhiNet C		81.72	13,354,858	83.16	31,930,098
MobileNetV1		68.88	51,200	68.68	51,200
MobileNetV2		85.86	43,092,800	92.51	147,774,464
0.75 MobileNetV2		85.27	28,089,840	89.99	87,691,008
PhiNet A	100	86.69	26,554,002	88.62	26,554,002
PhiNet B		81.60	12,197,466	79.01	12,197,466
PhiNet B_9		83.48	7,833,578	79.74	31,255,778
PhiNet C		85.25	13,354,858	86.35	31,930,098
MobileNetV1		69.98	51,200	75.93	77,806,080
MobileNetV2		87.49	147,774,464	94.21	147,774,464
0.75 MobileNetV2		88.14	87,691,008	91.99	87,691,008

Table 5.10: Best performance on CIFAR-10 and CORe50 for different models with a fixed replay buffer.

Chapter 6

Conclusion

The goal of this master thesis is to investigate the Latent Replay strategy when applied to edge architectures. In particular, we have employed the efficient architecture PhiNet for the first time in the context of Continual Learning.

Section 6.1 provides a summary of the main results obtained, and we conclude in Section 6.2 presenting an outlook for future works.

6.1 Conclusion

In this thesis work, we investigated the application of the Latent Replay strategy in efficient architectures. In particular, the behaviour in the context of Continual Learning of the efficient architecture PhiNet was analyzed for the first time. This validation process is carried out using established datasets like CIFAR-10 and CORe50.

We first collected empirical evidence of the influence of the number of samples in the replay memory on the performance of the models. Demonstrating that, with a sufficient number of samples, the innermost layers can achieve superior performance. However, in scenarios with limited memory, we observed that the outermost layers become the optimal choice in the accuracy-memory trade-off.

Furthermore, we have compared Latent Replay with other Continual Learning techniques such as Experience Replay. We found that in limited memory scenarios, Latent Replay emerges as the preferred choice, demonstrating its ability to efficiently use resources while maintaining high levels of performance.

In conclusion, we conducted a comparative assessment of PhiNet against well-established architectures, such as MobileNets, utilizing the Latent Re-

play strategy in memory-constrained scenarios. Our findings demonstrate PhiNet's superiority in layers close to the output, where there is a reduced computation, making it a suitable choice for deployment on resource-constrained devices. In terms of overall performance, PhiNet models show an advantage over MobileNet architectures. Specifically, with the memory limited to 0.5MB, the PhiNet models with the best performance show, for the CIFAR-10 dataset, an improvement of 4.47% (or 4.91% for CORe50) compared to their MobileNet counterparts. It is important to note that this high performance was achieved with a significant computational resource efficiency. For the CIFAR-10 dataset, the PhiNet models require only 0.012% (or 0.015% for CORe50) of the computation required by MobileNet. This means not only superior performance but also an optimization of computational efficiency, further strengthening the practical effectiveness of PhiNet models.

6.2 Future Research

This work presents numerous possibilities for future development. Specifically, one could extend the application of PhiNet in Continual Learning to tasks other than image classification by exploring Latent Replay in domains like Object Detection. PhiNet has demonstrated excellent results compared to other edge architectures, such as MobileNetV2, making it a promising candidate for diverse applications.

Furthermore, a crucial future direction to improve the efficacy of Latent Replay involves exploring novel compression techniques. These techniques can reduce the memory footprint, thereby enhancing the efficiency of the Continual Learning process at the edge.

Appendix A

Kendall Tau Distance

The Kendall tau rank distance is a distance function calculates the number of pairs that are in different order in two lists. The larger the distance, the more dissimilar the two lists are.

Following the same definition in [6], we define the set of discord pairs of two lists τ_1 and τ_2 as:

$$D(\tau_1, \tau_2) = \{(i, j) : i < j, [\tau_1(i) < \tau_1(j) \wedge \tau_2(i) > \tau_2(j)] \vee [\tau_1(i) > \tau_1(j) \wedge \tau_2(i) < \tau_2(j)]\} \tag{A.1}$$

The Kendall tao distance is defined in two froms:

$$K_d = |D|, \qquad K_n = \frac{2|D|}{n(n-1)} \tag{A.2}$$

where in the latter case, the distance is normalized to lie in the interval $[0, 1]$, and in the former case the distance lies in the interval $[0, n(n-1)/2]$. We have $K = 0$ only when $\tau_1 = \tau_2$. And the maximum occurs when τ_1 is the reverse of τ_2.

As an example, suppose to have two lists:

$$\tau_1 = [1, 2, 3, 4]$$
$$\tau_2 = [3, 4, 1, 2]$$

the discordant pairs in the two lists are:

$$D(\tau_1, \tau_2) = \{(1, 3), (1, 4), (2, 3), (2, 4)\}$$

Thus, the Kendall Tau distance is:

$$K_d = 4$$
$$K_n = 0.\bar{6}$$

Appendix B

Additional Results

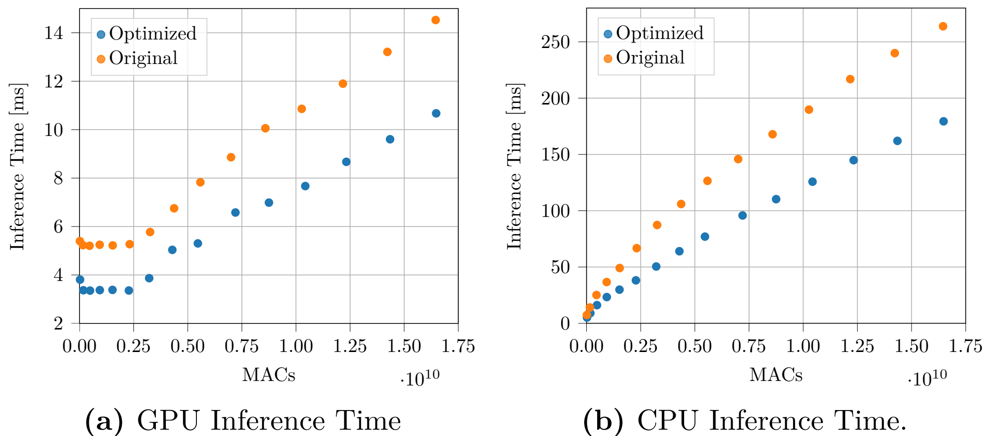
B.1 Optimized PhiNet Performances

Optimized PhiNet evaluation was carried out with a range of values for the parameters α, β and t_0, where each parameter was varied independently while keeping the others constant with the following values:

$$\alpha = 3 \qquad \beta = 0.75 \qquad t_0 = 6$$

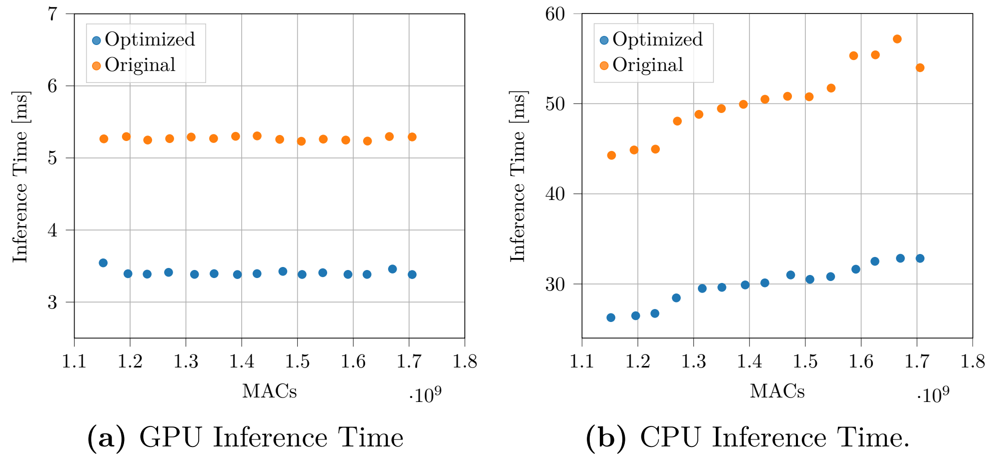
In the tests each of the parameters was tested with the following range of values:

$$\alpha \in [0.25, 10] \qquad \beta \in [0.25, 1] \qquad t_0 \in [1, 8]$$



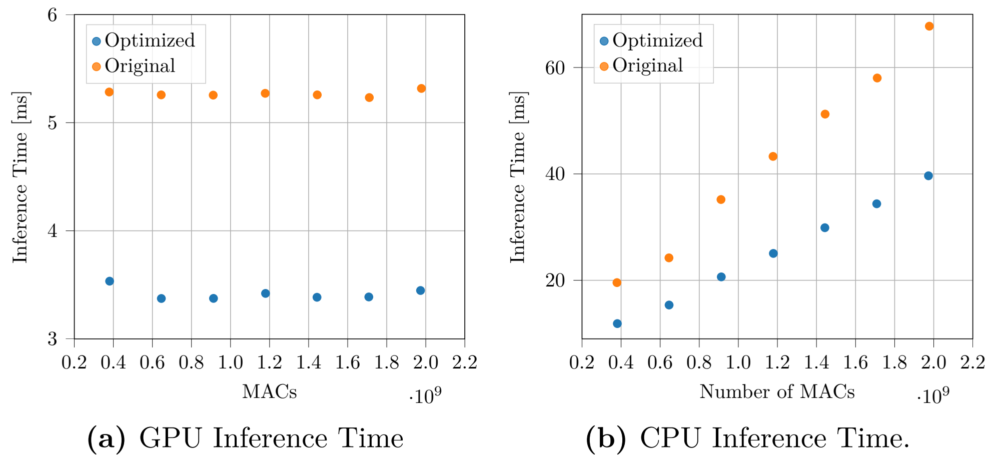
(a) GPU Inference Time **(b)** CPU Inference Time.

Figure B.1: Comparison of the inference time between the original model and the optimized model changing the α parameters.



(a) GPU Inference Time **(b)** CPU Inference Time.

Figure B.2: Comparison of the inference time between the original model and the optimized model changing the β parameters.



(a) GPU Inference Time **(b)** CPU Inference Time.

Figure B.3: Comparison of the inference time between the original model and the optimized model changing the t_0 parameters.

B.2 Similarity Measure

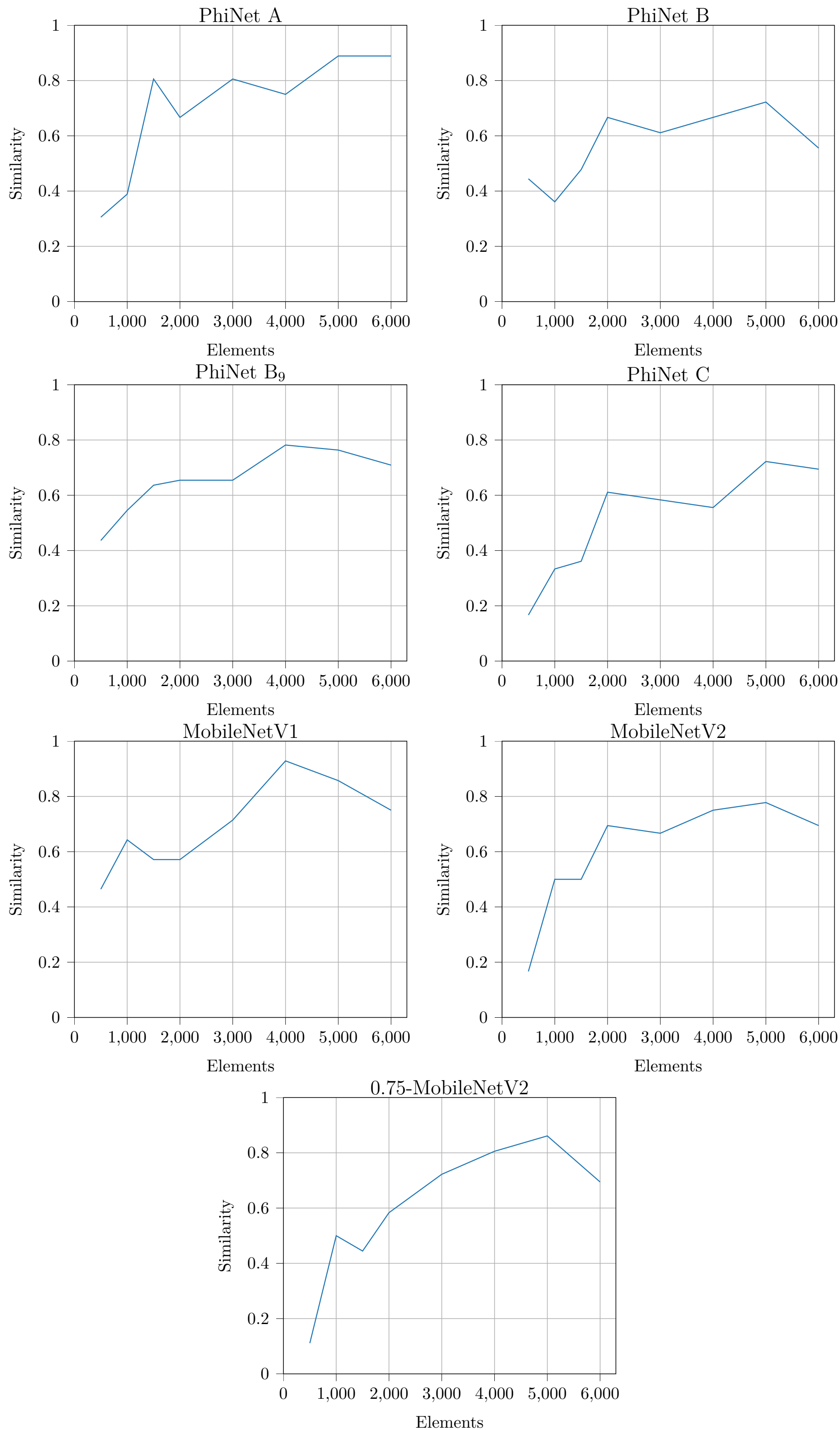


Figure B.4: Similarity measures on CIFAR-10 for the different models.

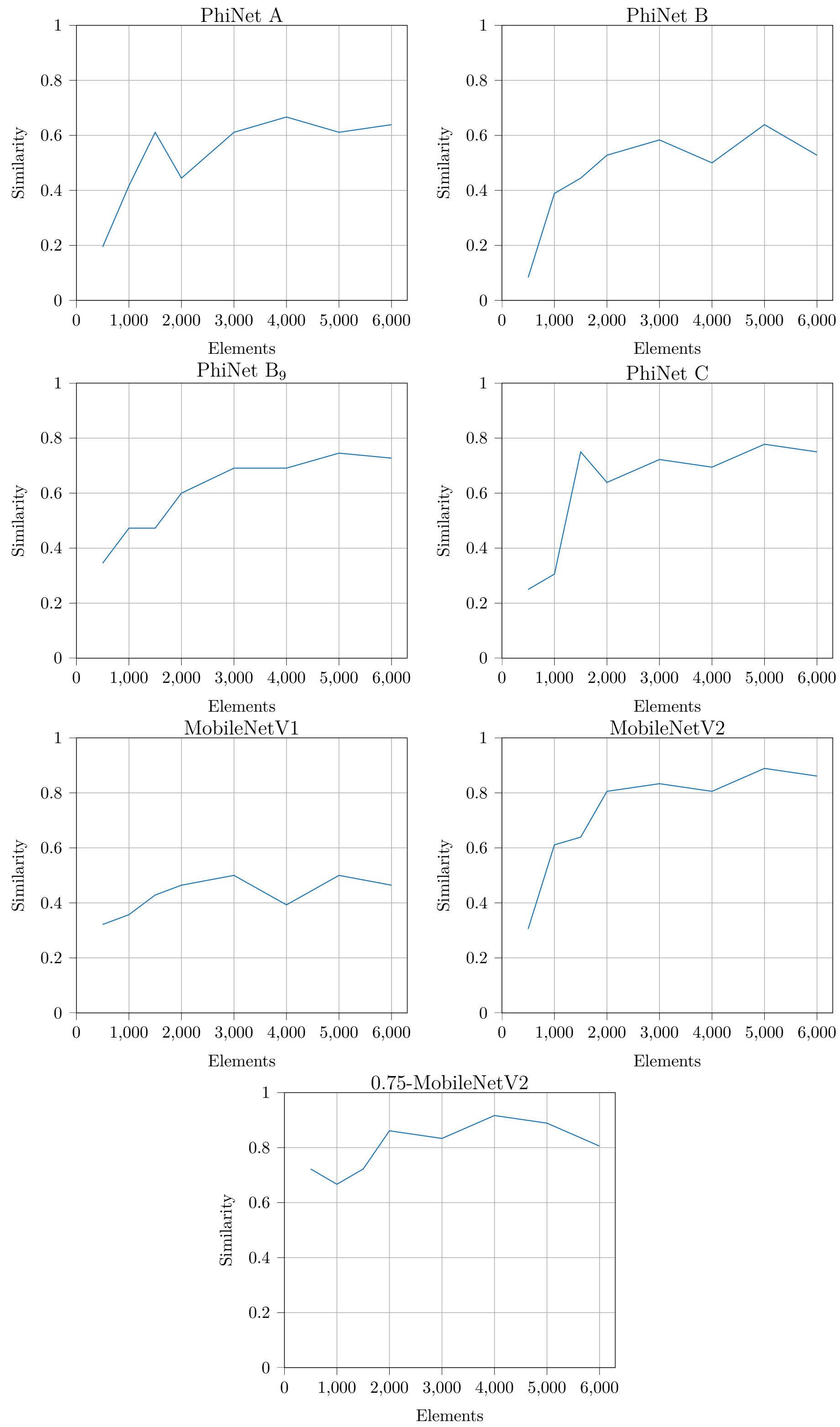


Figure B.5: Similarity measures on CORe50 for the different models.

Bibliography

[1] Mehdi Ataei et al. "Understanding Dataset Shift and Potential Remedies". In: *Vector Institute Tech Report* (2021).

[2] R Avenash and P Viswanath. "Semantic Segmentation of Satellite Images using a Modified CNN with Hard-Swish Activation Function." In: *VISIGRAPP (4: VISAPP)*. 2019, pp. 413–420.

[3] Alessio Brutti et al. "Optimizing PhiNet architectures for the detection of urban sounds on low-end devices". In: *2022 30th European Signal Processing Conference (EUSIPCO)*. IEEE. 2022, pp. 1121–1125.

[4] Arslan Chaudhry et al. "Riemannian walk for incremental learning: Understanding forgetting and intransigence". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 532–547.

[5] Arslan Chaudhry et al. "On tiny episodic memories in continual learning". In: *arXiv preprint arXiv:1902.10486* (2019).

[6] Vincent A Cicirello. "Kendall tau sequence distance: Extending Kendall tau from ranks to sequences". In: *arXiv preprint arXiv:1905.02752* (2019).

[7] Davide Dalle Pezze. "Methodological Advancements in Continual Learning and Industry 4.0 Applications". PhD thesis. Università degli studi di Padova, 2023.

[8] Matthias De Lange et al. "A continual learning survey: Defying forgetting in classification tasks". In: *IEEE transactions on pattern analysis and machine intelligence* 44.7 (2021), pp. 3366–3385.

[9] Natalia Díaz-Rodríguez et al. "Don't forget, there is more than forgetting: new metrics for Continual Learning". In: *arXiv preprint arXiv:1810.13166* (2018).

[10] João Gama et al. "A survey on concept drift adaptation". In: *ACM computing surveys (CSUR)* 46.4 (2014), pp. 1–37.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[12] Ian Goodfellow et al. "Generative adversarial networks". In: *Communications of the ACM* 63.11 (2020), pp. 139–144.

[13] Ian J Goodfellow et al. "An empirical investigation of catastrophic forgetting in gradient-based neural networks". In: *arXiv preprint arXiv:1312.6211* (2013).

[14] Tyler L Hayes et al. "Remind your neural network to prevent catastrophic forgetting". In: *European Conference on Computer Vision.* Springer. 2020, pp. 466–483.

[15] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[16] Xiangteng He and Yuxin Peng. "Fine-grained visual-textual representation learning". In: *IEEE Transactions on Circuits and Systems for Video Technology* 30.2 (2019), pp. 520–531.

[17] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[18] Jie Hu, Li Shen, and Gang Sun. "Squeeze-and-excitation networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2018, pp. 7132–7141.

[19] Chip Huyen. *Data Distribution Shifts and Monitoring.* Feb. 2022. URL: https://huyenchip.com/2022/02/07/data-distribution-shifts-and-monitoring.html.

[20] Maurice G Kendall. "A new measure of rank correlation". In: *Biometrika* 30.1/2 (1938), pp. 81–93.

[21] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[22] James Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the national academy of sciences* 114.13 (2017), pp. 3521–3526.

[23] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[24] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online].* 2 (2010). URL: http://yann.lecun.com/exdb/mnist.

[25] Timothée Lesort. "Continual learning: Tackling catastrophic forgetting in deep neural networks with replay processes". In: *arXiv preprint arXiv:2007.00487* (2020).

[26] Timothée Lesort, Massimo Caccia, and Irina Rish. "Understanding continual learning settings with data distribution drift analysis". In: *arXiv preprint arXiv:2104.01678* (2021).

[27] Zhizhong Li and Derek Hoiem. "Learning without forgetting". In: *IEEE transactions on pattern analysis and machine intelligence* 40.12 (2017), pp. 2935–2947.

[28] Vincenzo Lomonaco. "Continual learning with deep architectures". PhD thesis. Alma Mater Studiorum Università di Bologna, 2019.

[29] Vincenzo Lomonaco and Davide Maltoni. "Core50: a new dataset and benchmark for continuous object recognition". In: *Conference on robot learning*. PMLR. 2017, pp. 17–26.

[30] Vincenzo Lomonaco et al. "Avalanche: an end-to-end library for continual learning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 3600–3610.

[31] David Lopez-Paz and Marc'Aurelio Ranzato. "Gradient episodic memory for continual learning". In: *Advances in neural information processing systems* 30 (2017).

[32] Arun Mallya and Svetlana Lazebnik. "Packnet: Adding multiple tasks to a single network by iterative pruning". In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2018, pp. 7765–7773.

[33] Oleksiy Ostapenko et al. "Continual learning with foundation models: An empirical study of latent replay". In: *Conference on Lifelong Learning Agents*. PMLR. 2022, pp. 60–91.

[34] Francesco Paissan, Alberto Ancilotto, and Elisabetta Farella. "PhiNets: a scalable backbone for low-power AI at the edge". In: *ACM Transactions on Embedded Computing Systems* 21.5 (2022), pp. 1–18.

[35] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[36] Lorenzo Pellegrini et al. "Latent replay for real-time continual learning". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 10203–10209.

[37] Leonardo Ravaglia et al. “Memory-latency-accuracy trade-offs for continual learning on a risc-v extreme-edge node”. In: *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2020, pp. 1–6.

[38] Partha Pratim Ray. “A review on TinyML: State-of-the-art and prospects”. In: *Journal of King Saud University-Computer and Information Sciences* 34.4 (2022), pp. 1595–1623.

[39] Sylvestre-Alvise Rebuffi et al. “icarl: Incremental classifier and representation learning”. In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2017, pp. 2001–2010.

[40] David Rolnick et al. “Experience replay for continual learning”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[41] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[42] Andrei A Rusu et al. “Progressive neural networks”. In: *arXiv preprint arXiv:1606.04671* (2016).

[43] Mark Sandler and Andrew Howard. *MobileNetV2: The Next Generation of On-Device Computer Vision Networks*. Apr. 2018. URL: `https://blog.research.google/2018/04/mobilenetv2-next-generation-of-on.html?m=1`.

[44] Mark Sandler et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[45] Qi She et al. “OpenLORIS-Object: A Robotic Vision Dataset and Benchmark for Lifelong Deep Learning”. In: *2020 International Conference on Robotics and Automation (ICRA)*. 2020, pp. 4767–4773.

[46] Hanul Shin et al. “Continual learning with deep generative replay”. In: *Advances in neural information processing systems* 30 (2017).

[47] Mingxing Tan and Quoc Le. “Efficientnet: Rethinking model scaling for convolutional neural networks”. In: *International conference on machine learning*. PMLR. 2019, pp. 6105–6114.

[48] Gido M Van de Ven and Andreas S Tolias. “Three scenarios for continual learning”. In: *arXiv preprint arXiv:1904.07734* (2019).

[49] Gido M van de Ven, Tinne Tuytelaars, and Andreas S Tolias. “Three types of incremental learning”. In: *Nature Machine Intelligence* 4.12 (2022), pp. 1185–1197.

[50] Xiaofei Wang et al. “Convergence of edge computing and deep learning: A comprehensive survey”. In: *IEEE Communications Surveys & Tutorials* 22.2 (2020), pp. 869–904.

[51] Dianlei Xu et al. “Edge intelligence: Architectures, challenges, and applications”. In: *arXiv preprint arXiv:2003.12172* (2020).

[52] Friedemann Zenke, Ben Poole, and Surya Ganguli. “Continual learning through synaptic intelligence”. In: *International conference on machine learning*. PMLR. 2017, pp. 3987–3995.

[53] Eric Zhao et al. “Active learning under label shift”. In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2021, pp. 3412–3420.