ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING - DISI

SECOND CYCLE DEGREE IN ARTIFICIAL INTELLIGENCE

LATENT REPLAY-BASED ON-DEVICE CONTINUAL LEARNING USING TRANSFORMERS ON EDGE, ULTRA-LOW-POWER IoT PLATFORMS

Dissertation in Architectures and Platforms for Artificial Intelligence

Supervisor:

Dr. Francesco Conti

Co-Supervisors:

Alberto Dequino
Davide Nadalini
Luca Bompani

Defended by:

Călin Diaconu

Graduation Session: December 2024

Academic Year 2023/2024

Contents

Abstract

Transformers have recently exploded in popularity thanks to their adoption through tools like ChatGPT. This leads to an ever-growing pressure on computational centers, that are economically and ecologically expensive. Furthermore, sending private data from the end-user to these centers raises major security and privacy issues.

Among the possible solutions to this is continual learning (CL) on embedded devices, which enables lighter and faster retraining procedures, and eases the deployment requirements on low-power platforms.

This work explores this alternative by applying CL methods such as Latent Replay, Copy Weight with Reinit* (CWR*), and Architectural and Regularization 1* (AR1*) on transformer architectures designed for image processing, like the Vision Transformer (ViT).

The thesis opens the way for efficient deployment of transformer architectures on PULP microcontrollers, by implementing a highly flexible ViT golden model test in TrainLib. On the CORe50 dataset, accuracies improve for the evaluated configurations by up to 18% and, by using a ViT setup with less transformer blocks, models become up to 40% lighter, at the expense of less than 6% in accuracy.

1 Introduction

Transformers recently skyrocketed in popularity, thanks to their significantly superior results on text data ([1], [2], [3]) when compared to previous solutions ([4], [5]), their potential of improving results for other problems as well (computer vision [6], [7], [8], [9], audio solutions [10], [11], and even multimodal tasks [12], [13], [14]), and their commercial success with the non-technical public, through Generative Pre-trained Transformer (GPT) [15] based tools, such as virtual personal assistants (OpenAI's ChatGPT, Google's LaMBDA-powered [16] assistant, Meta's AI, based on their Llama configuration [3], and many others), and synthetic image (OpenAI's DALL-E family [17], Google's IMAGEN [18]) or video generation (OpenAI's Sora [19], Google's Lumiere [20], or Meta's Make-A-Video [21]).

Transformers are larger and more complex than previously existent solutions, such as convolutional [22] or recurrent networks [23]. Because of this, they are usually run server-side and the end-device only receives the inference result. The expensive training process and the final forward step happens in large data centers, far from the user. This issue led to a steep increase in loads and costs for computing clusters.

Despite efforts to decrease their memory and computation requirements, transformers are still difficult to deploy on lighter devices. However, this didn't stop the industry and the research community to seek ways to scale-down and push this costly process towards the edge of the computing ecosystem, reaching as far as ultra-low-power platforms, like IoT-dedicated microcontrollers [24].

One such way is continual learning (CL), which enables the adaptation or extension of a model to new scenarios or contexts by learning on newly obtained data, without losing the previous knowledge. This usually aims to specialize a deployed model to a particular environment,

such as understanding the dialect of a certain language, or personalizing a text generator to the style required by a user, or to extend its functionality, by adding more classes of objects to the knowledge base.

The upside of such a method is that the lengthy retraining of an entire model on the full dataset and the specific data, plus their storage requirement, is significantly reduced, especially in the case of heavyweight transformers. Although this looks like a promising solution, it still comes with drawbacks, in particular "catastrophic forgetting" [25], where the model ends up performing very poorly on the initial samples, overfitting on the new examples. Another limitation, especially in the context of costly models, is the computationally-heavy training process, more complex than a simple inference.

This thesis aims to set the foundation for deploying a continual learning pipeline to a microcontroller-level computer, with a focus on self-attention for computer vision.

The purpose was reducing the gap between transformer models and their deployment on edge devices, taking inspiration from previous similar endeavors, such as [24] or [26]. This has been accomplished by the implementation of a highly flexible deployment pipeline of a Vision Transformer (ViT) [6] on the PULP platform [27], through its dedicated TrainLib [28]. This provides a tool to translate one-to-one from a PyTorch [29] implementation of a ViT model, with variable internal configuration, such as input size, hidden dimensions, or number of transformer blocks. A new software utility has been deployed, capable of translating ViT internal variables into TrainLib components, and extended this library with the necessary primitives, such as MHSA, LayerNorm, GELU, and Tanh, to match their PyTorch counterparts.

ViT, a self-attention solution for computer vision, was then combined with 5 continual learning methods: native rehearsal, latent re-

play, CWR*, AR1*, and AR1* free [30]. These were ported from their previous evaluation on the MobileNet [31] convolutional model to the transformer-based one. The accuracies obtained on native rehearsal, combined with any of CWR*, AR1*, and AR1* free, are up to 18% higher than for the MobileNet [31], after pretraining on ImageNet 1k [32] and fine-tuning on CORe50 [33], a visual classification dataset, specially designed for CL problems.

A solution for reducing the memory occupation is also presented, pruning down to 7 transformer blocks, from the default 12 in ViT-Base. This compresses the model down to 60% of its initial size, and the entire CL pipeline to 87.8% of its initial memory requirement, with a loss in accuracy of less than 6%. An analysis of latent replay, the rehearsal method proposed in [30], used on transformers, is also performed, with initial accuracy values between 13.58% and 50.29% showing that it requires more fine tuning and experimentation before it can be successfully applied to this newer architecture.

The next step would be to finalize a deployable training pipeline for the ViT model on the PULP platform, including the CL politics that come with the previously introduced methods.

2 Background

2.1 On-Device Machine Learning

On-Device Machine Learning represents any Machine Learning (ML) task that is undertaken on the end-device, rather than server-side, in a large computational center, especially useful since inference takes about 5 times as many resources as training, according to [34].

The data is processed locally, on the device's hardware, reducing or removing altogether the need of a larger, remote processing unit, and of an Internet connection, implying an increase in the privacy and security for the end-user, and a reduction of the response-time down to real-time processing.

2.1.1 General Overview

Existent solutions optimize on-device machine learning in different ways, ranging from more software-related, to being closer to the hardware.

The critical resources at on-device deployment time are always the processing speed and the memory [35]. Oftentimes though, these 2 problems are improved at the same time, since reducing the model size through methods as simple as using less layers at the expense of performance, or more complex ones, like quantization, helps with both issues.

On the hardware side, there are acceleration techniques used for designing units capable of speeding-up the required parallel computation. The most popular such solutions are graphics processing units (GPUs).

As outlined by [36], hardware accelerators for lighter, edge devices are less common, with just a recently intensified effort to bring them into the main-stream. This is accomplished through developments such as tensor processing units (TPUs) or neural processing units (NPUs),

that can be found in current-day smartphones.

The hardware accelerators for ultra-low-power devices, such as microcontrollers, are even less common and powerful, with solutions covering either assisting hardware, like Digital Signal Processors (DSPs) [37], or computers and architectures designed from the ground-up for improving this type of applications, such as STM32N6, Alif's E1C, GreenWaves Technologies' GAP9 [38], or PULP.

2.1.2 The PULP Computational Platform

The Parallel Ultra Low Power Platform (PULP) [27] is a computational architecture for scalable edge computing, based on the RISC-V computer architecture, focusing on ultra-low-power devices. It includes a microcontroller system and a multi-core cluster. Each core is optimized for low-power operations, which allows the architecture to maintain high energy efficiency even when running complex tasks or handling multiple workloads simultaneously.

Near-threshold computing is a fundamental element of the PULP architecture. By running the processor at voltages just above the switching threshold of transistors, it reduces power consumption while offering enough computational performance for various applications.

The PULP design is used in Greenwave Technologies' GAP9 SoC [39], with the configuration presented in Figure 1. It's divided in a system-on-chip (SoC) region, representing the microcontroller unit (MCU) domain, and a cluster region, with 8+1 RISC-V cores for computation acceleration.

The MCU domain has a single RISC-V core, used for control tasks. It has access to an FPU unit and to up to 2 MB of L2 SRAM, accessible in a single clock cycle.

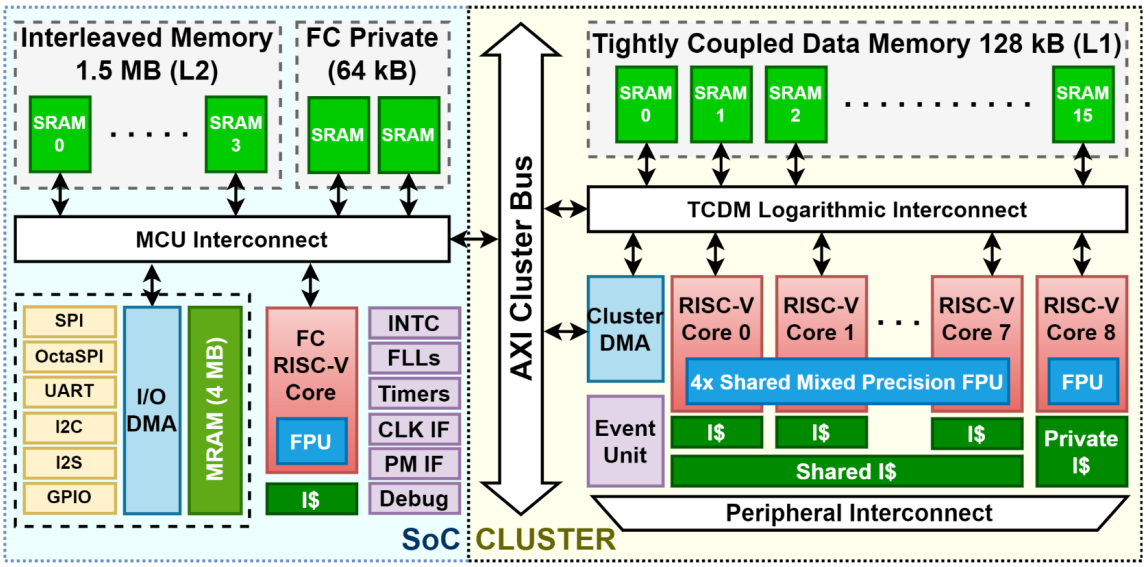

Figure 1: PULP SoC Architecture with 9 RISC-V Cores and 4 shared Mixed Precision FPUs. Source: [39].

The cluster domain benefits from an 8+1 RISC-V core setup, with an RV32IMFC instruction set architecture (ISA). According to [40], the name of the ISA refers to its characteristics: RV32, since it's based on 32-bit RISC-V, I is the base set of instructions, and MFC are 3 sets of included extensions: M - multiply and divide, F - single-precision floating point, and C - compressed instructions.

It also contains a dedicated DMA unit with improvements such as post-incremental load and store instructions, and 2-level hardware loops. The 8+1 CPUs (8 for parallel computation and the 9th one for programming the DMA and dispatching tasks to the others) have access to 4 mixed-precision floating point units (FPUs). Memory-wise, an L1 data-scracthpad of up to 256 kB is shared among the cores and accessible in a single core cycle.

FPUs' can work both on fp32, doing 1 MAC/clock cycle, and on fp16 SIMD instructions, with 2 MACs/cycle. There is also a non-volatile MRAM, with a capacity of up to 4 MB.

PULP benefits from a rich set of peripherals, like I2C, SPI, and GPIO,

as it is designed to be implemented on IoT intelligent sensors. It is included in single-core microcontrollers, such as PULPissimo [41], but also in multi-core IoT processors, like OpenPULP [27]. Ultimately, the goal of PULP is to enable more powerful, yet energy-efficient devices that can function for long periods on small battery power while still supporting demanding tasks.

2.1.3 PULP TrainLib

PULP TrainLib [28] is a software framework for DNN On-Device Learning deployment. It consists of a library of primitives, optimized for parallel execution, capable of executing forward and backward steps on this class of microcontrollers.

It manages to harvest the PULP high-performance and low-power capacities, given by the architectural parallelism and near-threshold execution. The authors remark that the primitives in the library show an almost linear parallel speed-up on a multi-core RISC-V platform. Other features of the library are the support for both fp32 and fp16 operations, and an extensive use of matrix multiplications.

PULP-TrainLib also provides a set of tests for each layer primitive and for more complex components, like the softmax activation, or the recursive neural network layer. They work by comparing the output and gradient values obtained by a reference model, i.e. "golden model (GM)", implemented using PyTorch [29], with the ones obtained by the primitive or by the set of primitives.

The tests work by comparing either an average of the value sets to be compared, or by performing a value-by-value check. This comparison is performed using absolute values, and the test is considered to be passed if the error between any 2 checked values is below a user-defined threshold. Usually, this error tolerance is higher for fp16 primitives since the match between values is looser than fp32, caused by mismatches

in floating point representations between the two platforms and other approximation and casting differences.

2.2 The Transformer Architecture

The authors of [1] describe the transformers as an improvement to traditional sequence transduction methods, which is the machine learning problem that involves converting an input sequence into another output sequence, with the 2 potentially having different lengths. It takes inspiration from attention mechanisms used between encoders and decoders in the recurrent and convolutional networks that were the standard at the time for this type of problems.

It has an encoder-decoder structure, suitable for sequence-to-sequence tasks, which works by transforming the input into a different representation space, and then producing the desired output.

It is based on a self-attention mechanism, also considering previously generated outputs, and extended with point-wise, fully connected layers. The overall Transformer architecture is presented in Figure 2, both the encoder and the decoder being a stack of six layers in the original implementation.

Attention is abstracted by the authors as being a mapping of a query and a set of key-value tuples to an output. This is achieved through a weighted sum of the values, based on a compatibility function between the query and a certain value's corresponding key. The attention function used in this paper is based on the dot-product or multiplicative attention, with an extra scaling factor. This is used as a form of normalization, to compensate for the increase in magnitude of the dot product when the size of the key vector also increases.

The mathematical definition of this attention function can be found in Equation 1, taken from [1], where Q, K, and V are the query, key,

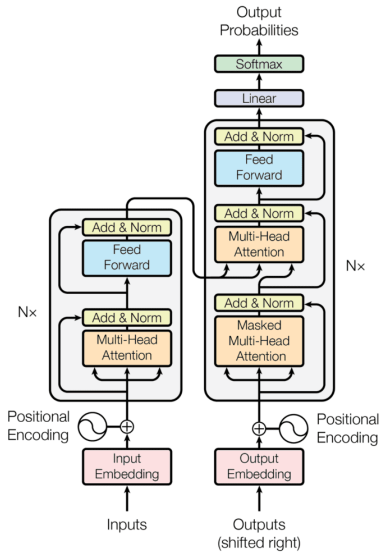

Figure 2: Transformer Architecture. Source: [1].

and value vectors, K^T is the transpose of K, and d_k is the size of K. $softmax$ has its definition from [42], given in Equation 2, where e is Euler's number, and x is a set of N values.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

$$softmax(x) = \frac{e^{x[i]}}{\sum_{j=1}^{N} e^{x[j]}} \tag{2}$$

Another structure particularity is the use of attention in a "multi-headed" fashion. In order to overcome the averaging of attention-weighted positions, that would diminish the ability of the model to treat information from different representation subspace at different positions, the attention mechanism can be split into a number of "heads". Each head will contain an individual set of weights that will be applied to the input sequence, with the output of all heads being concatenated in the end. The resulting matrix is going to be passed through a final linear layer. 8 attention heads are used in the original version, with $d_k = d_v = 64$, and the visual interpretation of a general, h-headed at-

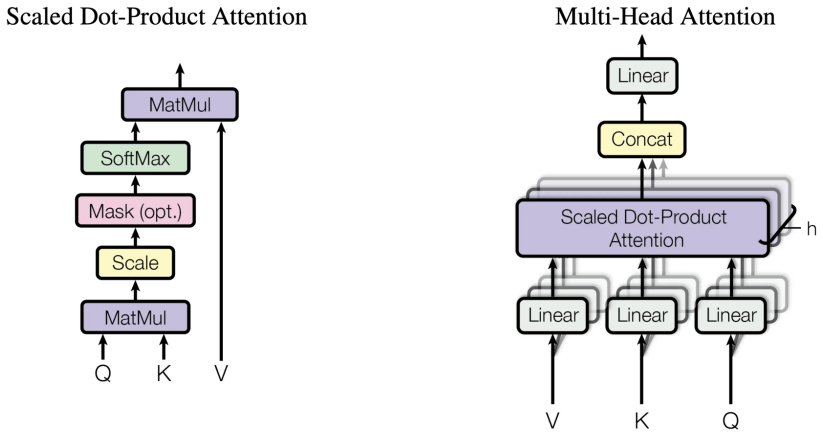

Figure 3: Scaled Dot-Product Attention and its deployment in the Multi-Head Attention. Source: [1].

tention can be found in Figure 3.

The type of attention that is used there is called "self-attention" from the fact that all three elements, queries, keys, and values, have the same source. They are obtained by passing the same input through three different linear layers, which means that the attention layers have access to information belonging to all positions in the input sequence.

Transformer blocks also use position-wise feed-forward networks. These are essentially blocks with containing three elements, in this order: a linear layer, a ReLU activation, another linear layer.

Another notable part is the positional encoding applied on the input sequence before being fed to the encoder layers. This is a way to pass information about each token's position inside the sequence by interpreting the position through sine and cosine functions, which can be seen in equations 3 and 4 ([1]), whose results get added to the initial input vector. In the two equations, *pos* refers to the position index of the token inside the input sequence, i is a variable used to differentiate between even and odd sequences, and d_{model} is the dimension of the token embedding.

$$PositionalEncoding(pos, 2i) = \sin \frac{pos}{10000^{\frac{2i}{d_{model}}}} \tag{3}$$

$$PositionalEncoding(pos, 2i+1) = \cos \frac{pos}{10000^{\frac{2i}{d_{model}}}} \tag{4}$$

Although attention and transformers have been initially designed for sequence transduction, it has since been successfully extended to other language problems, such as generative [15] or predictive [2] ones, but also to other categories of tasks such as audio ([43], [10], [11]) or computer vision ([7], [8], [9]). Regarding the latter, one of the most popular solutions is the Vision Transformer (ViT).

2.2.1 The Vision Transformer (ViT)

ViT [6], released by Google Research, aims to be a new adaptation to computer vision tasks of the attention-based architecture [1], with the aim to reduce the reliance on convolutional neural networks (CNNs).

ViT tries to solve issues such as architectures that are too complex to work on current hardware accelerators, or solutions only suitable for small input images. By not using convolutional layers, they also manage to obtain a solution that is rather robust when it comes to locality and translation equivariance, resulting in a reduced image-specific inductive bias.

The computer vision task that the authors focus on is classification, mainly using two datasets for training: the popular, publicly available ImageNet [32], and the private, Google-owned JFT-300M dataset. On ImageNet, they report an increase of 1% in accuracy, while requiring almost 75% less computation, when compared to the ResNet-based [44] BiT-L [45].

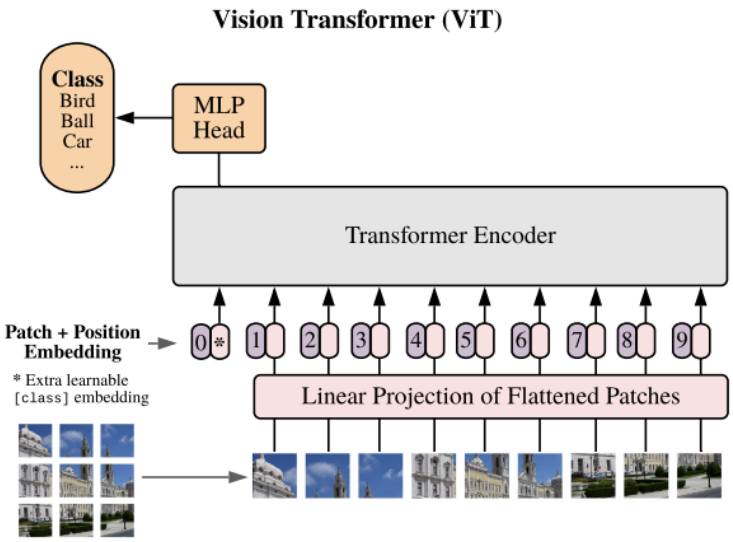

Figure 4: Overview of the ViT architecture. Source: [6].

From the original Transformer Architecture, ViT is using the encoder, but with an increased number of layers, in three variants, Base, Large, and Huge, with 12, 24, and 32 layers. In order to adapt the images to the input format required by transformers, they are reshaped into a sequence of flattened patches. These are then linearly projected into the required dimension, thus obtaining the patch embeddings that are fed to the encoder. Also similarly to the original Transformer, an extra element is added to the encoder input, in the form of a 1D learnable positional embedding.

The embedded image representation, provided by the encoder, is then passed through a multilayer perceptron (MLP), with the purpose of obtaining the final classifications. This head contains a single linear layer for fine-tuning, but also an extra hidden layer during pre-training.

A difference from the encoder presented in [1] consists of the introduction of layer normalizations before every block, inspired by [46], and residual connections after, as seen in [47]. The MLP blocks inside the transformer encoder also contain GELU activations (defined in a later section, in equation 9), to introduce non-linearity.

The authors also propose a hybrid architecture, where the input sequence is obtained by using convolutional layers applied straight to the input image. It's the activation of these layers that is being fed to the patch embedding projector.

ViT is tested on a number of popular computer vision benchmarks and manages to obtain competitive results, and state of the art status, with lower pre-training costs. An overview of the architecture, also found in the original paper, is available in Figure 4.

2.3 Continual Learning

Continual learning (CL) represents a dynamic machine learning technique which uses input data to further train the model even after deployment.

The purpose of this method is to adapt to new data, both in terms of out-of-training distributions (e.g. performing well in new contexts), but also in terms of domain shifts (supporting new categories in a classification problem, for example), without forgetting the previously gained knowledge.

And indeed, one of the major issues of incremental learning is catastrophic forgetting [25]. This is the phenomenon that happens when a model overfits the new data, and its performance drops significantly when evaluated on the initial set. It can be resolved through retraining, implying increased processing efforts or memory usage.

Another difficulty when dealing with this type of processes is the cost of the retraining, which may make it unfeasible to run on-device, especially when working with ultra-low-power computers. Some solutions that try to explore deployable CL pipelines are SIESTA [48], a data-driven updating procedure, and Tiny Episodic Memories [49], that gets

infused into an Experience Replay workflow.

In terms of types of continual learning, [50] splits them into three categories:

- Rehearsal-based, when training samples are stored over time and combined with new data for optimization. Some examples are native rehearsal [51], a basic method that stores the input data in its raw format and feeds it later, but also more complex ones, such as Pseudorehearsal [52] or Expeirence Replay [49];
- Regularization-based, introducing new factors in the optimizer, such that old knowledge degrades slower. Examples include the Adaptive Regularization from [53], or Synaptic Intelligence [54], that makes use of a measure called weight importance, and that it's described in detail in one of the following sections;
- Architectural-based, which bring changes to the structure of the model itself, like DyTox [55], or the dual-memory in [56].

One that may be classified as both architectural and rehearsal is latent replay. It brings modifications both to the structure of the model and to its workflow, depending on the type of the provided data. It works by injecting previously stored activations, if a sample that has already been processed before is requested, and by keeping the regular flow of operations when a new input appears. This process of storing the activations can enable the classification of the method as a rehearsal one, the patterns replacing the images that are usually stored in the case of native rehearsal. Its implementation is described in more detail in one of the following sections.

2.3.1 Latent Replay

The current project takes inspiration from the solutions described in [57], [30], [51], [50], [33] and [54]. All the experiments employed in that paper are based on a convolutional neural network, namely MobileNet

[31], pre-trained on ImageNet-1K [32], and further trained and evaluated using the NICv2-391 configuration of the CORe50 dataset [33], detailed in a later section.

The terminology used below regarding the grouping of data is the following: the term "batch" is going to be used for a batch of data provided by the CORe50 ([33], described in another section of this thesis), with 391 batches used for training, and the term "mini-batch" for the batch of data used during any single one of the optimization steps of the SGD algorithm.

The CWR* method is first introduced in a previous paper of the same authors, [51], where it was used in a native replay fashion. This is an improvement of another algorithm from them, called CWR+, described in [50], which, in turn, develops the initially defined Copy Weight with Reinit (CWR), from [33].

After determining a baseline, obtained through the Cumulative approach, they continue with three continual learning approaches: CWR*, AR1*, and AR1* free. These three approaches are applied in two different manners: native replay, introduced in a previous work, and latent replay.

The Baseline A baseline value, called cumulative upper bound, is obtained by training a full MobileNet on the entire shuffled dataset. This means that the images in the set can occur in any order, rendering inactive any continual learning situation. It is the most memory-intensive scenario, in which all provided images are stored and then used to retrain a model from scratch. This baseline value is reported to be around 85% in the paper.

CWR The initial CWR algorithm was presented together with the CORe50 dataset, with the purpose of being used in the new classes

(NC) situation (described in the Dataset subsection of this thesis), obtaining improvements of about 30%, compared to the naive approach. Experiments have also been performed in the New Instances and Classes (NIC) scenario, with significantly inferior results when compared to the cumulative baseline: the accuracy drops to about 50% of the previous value.

CWR is designed to work on class-specific weights in the "head" of the model, namely in the fully connected (FC) layers in the last part of the model, that are responsible for the final classification. In order to be able to exactly associate these weights to each class, they remove 2 of the pre-existent FC layers, named fc6 and fc7 in the official MobileNet [31]. They end up connecting fc8, which has as output an array with the length equal to the number of possible classes, to the backbone, and more precisely to the pool5 pooling layer.

Two sets of weights are used, called consolidated weights (denoted with cw), and temporary weights (tw). An intuitive description is given in the paper as cw being a longer-term memory, while tw being the working, short-term memory. The tw part is randomly re-initialized before each batch, effectively learning from scratch for exactly one batch. The cw are initialized to 0 at the beginning of the session, before the first batch, and then updated after each batch according to the newly obtained tw.

The procedure, adapted from [50], can be described as in Algorithm 1, where the backbone refers to convolutional layers previous to the "head" layers, responsible of feature extraction, and w_i denotes "batch-specific weights", a multiplicative factor given as hyperparameter and specific to each batch. Since it's working in the NC scenario, in each batch there will be a number of new classes that have never been seen before. Thus the need of the fourth step that shows how tw starts with the number of neurons needed for the classes that show up in the first

batch, and it gets incrementally expanded with the classes specific to each subsequent batch. In the end, the cw are used for inference, since the tw are discarded and refreshed during each batch.

Algorithm 1 Copy Weight with Reinit (CWR). Adapted from [50]

$cw \leftarrow 0$
Initialize backbone weights (randomly or from pre-trained)
for each training batch, B_i, with i from 1 to n, where n is the number of batches, **do**
    Expand output layer for the new classes in B_i
    Randomly re-initialize tw for all neurons
    if $i == 1$ **then**
        Train backbone + tw with SGD on B_i
    else
        Train tw with SGD on B_i (freeze backbone)
    end if
    for each class j in B_i **do**
        $cw[j] \leftarrow w_i * tw[j]$
    end for
    Validate on backbone + cw
end for

CWR+ The CWR+ method brings two modifications to the basic CWR method that regard both the cw and the tw. They are detailed in [50].

The first one is replacing the w_i hyperparameter, which is sensitive to tuning, and which led to poor performances during experiments. The replacement is a mean-shift adjustment, where the multiplication with w_i is replaced by a subtraction with the global average of tw. The authors state that they empirically discovered that the scaling is no longer necessary when applying this new operation.

The second change regards the initialization of the tw, where it is recommended to give an equal value to all the weights inside tw, which helps the training process by equalizing the output of the subsequent softmax layer. This in turn helps to moderate any large values inside the errors, during the backpropagation step, that may lead to unnecessarily large weight updates. The authors motivate this change both

through mathematical demonstrations and improved results in experiments. They use 0 as the initialization value for simplicity reasons.

The changes can be observed in Algorithm 2. The symbols and names have the same meaning as described above for Algorithm 1, with the avg function representing the global average over all values inside tw.

Algorithm 2 CWR+. Adapted from [50]

$cw \leftarrow 0$
Initialize backbone weights (randomly or from pre-trained)
for each training batch, B_i, with i from 1 to n, where n is the number of batches, **do**
    Expand output layer for the new classes in B_i
    $tw \leftarrow 0$
    if $i == 1$ **then**
        Train backbone + tw with SGD on B_i
    else
        Train tw with SGD on B_i (freeze backbone)
    end if
    for each class j in B_i **do**
        $cw[j] \leftarrow tw[j] - avg(tw)$
    end for
    Validate on backbone + cw
end for

CWR* CWR* is introduced in [51], and it is presented as being efficient for both the NC and NIC scenarios.The tw "0-reset" is replaced with a loading of the learned weights from cw, thus regaining information previously learned on classes that have been present in previous batches and that reoccur in the current one.

The cw update step is called the consolidation step. It is adjusted by weighting the updated proportionally to the total number of occurrences of the class of interest in previous batches, and inversely proportional to the number of occurrences in the current one.

The changes are included in Algorithm 3, with the entire CWR* procedure being described. As before, the variables and names keep

their meanings, $current[j]$ representing the number of occurrences of class j in the current batch.

Algorithm 3 CWR*. Adapted from [51]

$cw \leftarrow 0$
$past \leftarrow 0$
Initialize backbone weights (randomly or from pre-trained)
for each training batch, B_i, with i from 1 to n, where n is the number of batches, **do**
    Expand output layer for the new classes in B_i
    $tw \leftarrow 0$
    for each class j in B_i **do**
        $tw[j] \leftarrow cw[j]$
    end for
    if $i == 1$ **then**
        Train backbone + tw with SGD on B_i
    else
        Train tw with SGD on B_i (freeze backbone)
    end if
    for each class j in B_i **do**
        $w\_past[j] \leftarrow \sqrt{\frac{past[j]}{current[j]}}$
        $cw[j] \leftarrow \frac{cw[j] * w\_past[j] + (tw[j] - avg(tw))}{w\_past[j]+1}$
        $past[j] \leftarrow past[j] + current[j]$
    end for
    Validate on backbone + cw
end for

AR1 AR1 is described as an architectural and regularization approach, which extends CWR+ by unfreezing the backbone and applying a regularization factor to its parameters, namely Synaptic Intelligence (SI).

While in all the variations of the CWR algorithm the weights in the backbone are only adjusted for the initial batch, here they are also subjected to training. In order to limit the catastrophic forgetting phenomenon in this part of the network, the regularization factor needs to be introduced. After being compared with Learning Without Forgetting (LWF) from [58], and Elastic Weights Consolidation (EWC), from [59], SI was selected for its stability and ease of tuning when used with CWR+, and for the small overhead required.

All the steps needed for AR1 are described in Algorithm 4, with the symbols having the same meaning as described in previous Algorithms.

Algorithm 4 AR1. Adapted from [50]

$cw \leftarrow 0$
Initialize backbone weights (randomly or from pre-trained)
$\hat{F} \leftarrow 0$
for each training batch, B_i, with i from 1 to n, where n is the number of batches, **do**
    Expand output layer for the new classes in B_i
    $tw \leftarrow 0$
    Train backbone with SGD and SI regularization on B_i
    Train tw with SGD and no regularization on B_i
    for each class j in B_i **do**
        $cw[j] \leftarrow tw[j] - avg(tw)$
    end for
    Update $\hat{F}$ according to the SI requirements
    Validate on backbone + cw
end for

Synaptic Intelligence (SI) is first presented in [54] as a variant of EWC [59] which computes the weight importance described in the latter online, by making use of elements already required by SGD. This importance measure defines each weight, and it's used when updating them, such that changes to important parameters are penalized. A trivial interpretation may present this as a method to avoid overwriting "old memories", which would lead to catastrophic forgetting.

SI works by storing an array of values representing this importance, noted with $\hat{F}$, accumulating over all the batches with a factor called batch-specific weight, noted with w_i, tunable, similarly to what is used in the CWR algorithm. The total weight importance is initialized with 0, and updated with a value computed over the current batch that is directly proportional to the total loss change relative to the weight of interest, and this weight's change over the batch. According to [50], it can be mathematically expressed like below (equations 5 and 6), where the following notations are used:

- ΔL_k is the change of the loss relative to a single weight k, and over

a single step of the SGD;

- $\Sigma\Delta L_k$ is the sum of these changes of weight k over all the steps inside a batch;
- $\Delta\theta_k$ is the variation of weight k over a step of the SGD;
- T_k is the total variation of the same weight k, but over an entire batch;
- $\frac{\partial L}{\partial \theta_k}$ is, obviously, the derivative of the loss relative to weight k;
- F_k will end up being the importance of weight k over a batch;
- ξ is a very small constant, set to 1e-7 in experiments, used to avoid division by 0 errors.

$$\Delta L_k = \Delta\theta_k \frac{\partial L}{\partial \theta_k} \tag{5}$$

$$F_k = \frac{\Sigma\Delta L_k}{T_k^2 + \xi} \tag{6}$$

A clipping to the final value is also performed, such that values below 0 are set to 0, and values above a threshold (max_F - a hyperparameter) are set to that value. After the local weight importance is added to the total value, this latter one is used inside the weight update of the SGD by changing the traditional procedure (see equation 7) into a version that uses the weight importance to moderate the scale of the weight update (like in equation 8).

Most of the symbols are the same to the ones associated when describing the SGD update step: θ_{k_n} is weight k used on batch n, λ is the learning rate, and everything else as described above. The element of novelty is the regularization term, $1 - \frac{\hat{F}_{k_n}}{max_f}$, where $\hat{F}_{k_n}$ is the total weight importance of weight k at the moment n, and max_F the cutoff

value, described above.

$$\theta_{k_t} \leftarrow \theta_{k_{t-1}} - \lambda \frac{\partial L}{\partial \theta_{k_{t-1}}} \tag{7}$$

$$\theta_{k_t} \leftarrow \theta_{k_{t-1}} - \lambda (1 - \frac{\hat{F}_{k_{t-1}}}{max_f}) \frac{\partial L}{\partial \theta_{k_{t-1}}} \tag{8}$$

AR1* Similarly to CWR*, AR1* is introduced in [51] as an evolution of AR1 [50], which also includes elements of the previously described CWR+. It simply replaces the use of CWR+ in AR1 with CWR*, and it includes the SI regularization. The updated procedure is described in Algorithm 5, with the same meaning of the symbols as before.

Algorithm 5 AR1*. Adapted from [51]

$cw \leftarrow 0$
$past \leftarrow 0$
Initialize backbone weights (randomly or from pre-trained)
$\hat{F} \leftarrow 0$
for each training batch, B_i, with i from 1 to n, where n is the number of batches, **do**
    Expand output layer for the new classes in B_i
    $tw \leftarrow 0$
    for each class j in B_i **do**
        $tw[j] \leftarrow cw[j]$
    end for
    Train backbone with SGD and SI regularization on B_i
    Train tw with SGD and no regularization on B_i
    for each class j in B_i **do**
        $w\_past[j] \leftarrow \sqrt{\frac{past[j]}{current[j]}}$
        $cw[j] \leftarrow \frac{cw[j] * w\_past[j] + (tw[j] - avg(tw))}{w\_past[j] + 1}$
        $past[j] \leftarrow past[j] + current[j]$
    end for
    Update $\hat{F}$ according to the SI requirements
    Validate on backbone + cw
end for

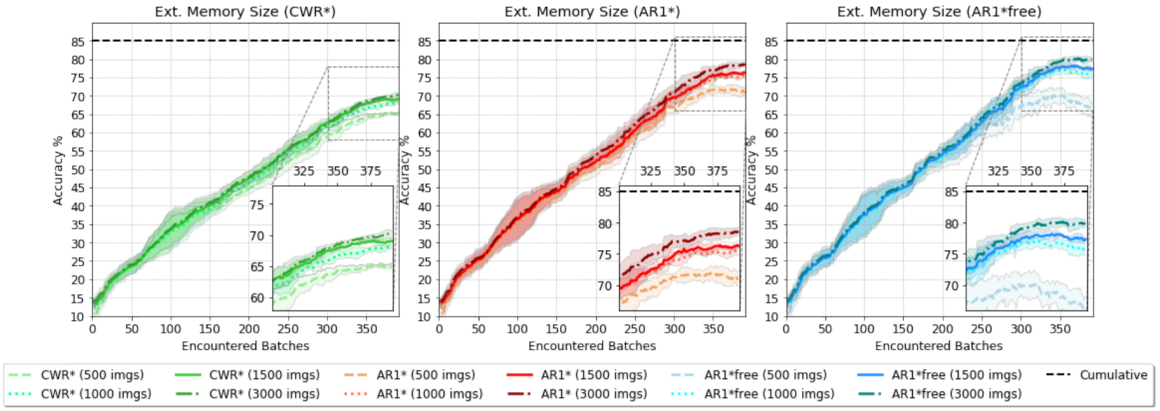

Figure 5: Accuracy evolution, computed on the test set, during the training process of a MobileNetV1, on NICv2-391 of CORe50. The transparent regions over the plot represent the variance in accuracy in each point, since the experiment is repeated 10 times, over all the available runs available for the subset. The solid lines are the averaged values, also used in other reported results. Source: [30].

AR1* Free [51] introduces a variant of the AR1* algorithm, where the SI regularization is removed. This change is motivated by the authors through the fact that measures against forgetting are already taken through rehearsal. This leads to a method that's also analogous to CRW*, with the backbone unfrozen. Due to these close similarities, an explicit algorithm for this procedure is not provided.

Native Rehearsal Native rehearsal as it is named in [51], or basic rehearsal (REHE) in [50], is a method to combat forgetting in continuous learning problems. It involves storing part of the data of previous and current batches to be used in future training steps, together with new data that is going to be original to future batches.

In [50], basic rehearsal is compared to 2 other rehearsal-based approaches: Gradient Episodic Memory (GEM) [60], and Incremental Classifier and Representation Learning (iCaRL) [61]. Both of them have an inferior performance to the equivalent REHE (up to $\tilde{1}2\%$ improvement in accuracy on an NC scenario), meaning when comparing runs with equally sized external memory.

For native rehearsal, this external memory is the name usually given

to the storage place of previous data, and the number of stored patterns is dependent on its size. In these basic cases, the sampling is random, both in terms of data to be saved from the current batch, but also of the data to be removed from the external memory. However, sometimes these choices can also be adjusted to preserve the class ratios that occurred up to a certain iteration in the training loop, and also in terms of the number of elements to be saved from a certain step.

The authors experiment with the three methods mentioned before, CWR*, AR1*, and AR1* free, in the current native rehearsal situation, over multiple rehearsal memory sizes: 500, 1000, 1500, and 3000, by training on the NICv2-391 scenario of CORe50, with results reported in Figure 5. They observe how accuracy increases together with memory size, and they decide that a good trade-off with performance is working with a rehearsal memory size of 1500, on the best performing technique, namely AR1* free.

Latent Replay The main focus of [30] is a novel rehearsing method called latent replay, sometimes shortened as LR, designed for on-the-edge light devices. The core of the technique is storing intermediate layer activations, instead of raw input data, with the purpose of reducing storage space requirements for rehearsal data, and increasing the speed during training time. More details are given in the next section.

3 Methodology

3.1 PULP TrainLib

3.1.1 Multi-Head Self Attention (MHSA) Primitve

One of the goals of this project was to be able to deploy a ViT model on a PULP-based microprocessor through the available PULP TrainLib. For this, primitives for a number of layers had to be finalised or implemented from scratch, the most complex of them being the one for the Multi-Head Self-Attention (MHSA) layer.

The structure for both forward and backward passes existed, but the golden-model-specific tests were not accurate enough, due to a combination of operation and memory allocation issues, which had to be debugged.

Another adjustment for these tests, for the fp16 version, was to switch from absolute to relative value comparison, due to the wide range of values that exist in the activations inside the layer. This was needed to increase the flexibility of the tests, since the error was larger for some elements of the layer that require more approximations, such as softmax, as detailed in the following subsection.

In order to match the publicly available pre-trained ViT model, the MHSA TrainLib implementation has been changed from having one large input projection layer, applied once to the input and outputting a single matrix for all three required elements (key, query, value) to having three smaller ones, one for each element. Like in the original ViT, the projection layers also include biases, and a visual interpretation of the projection changes can be seen in Figure 6. Two more transposition operations had to be added in the forward step, and three vector summations in the backward step.

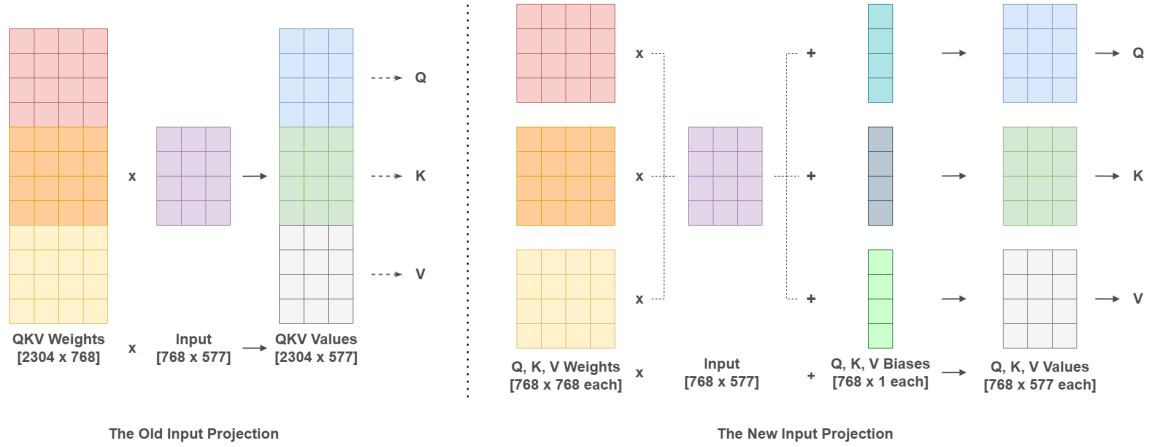

Figure 6: Previous input projection of mhsa, performed using a single, unified linear layer, vs. current input projection, split into 3 linear layers. The number of cells is just for illustrative purposes and is not reflective of the absolute values of the dimensions used in this project's ViT implementation. The "x" symbol represents a matrix multiplication.

3.1.2 Other Primitives

Softmax One of the major challenges of the TrainLib part of the project was identifying and fixing the issue of the softmax activation. Inside of any such activation, there exists an exponential activation, which was replaced here, both inside the golden model and in the primitive, with a different, faster variant, as described in [62], that is an approximation of the initial function. After experimentation, similar results to the ones in the paper have been obtained both in terms of error compared to the regular exponential, and of execution time.

When running a PyTorch [29] module that includes casting of the type required for this method, the automatic gradient computation is lost (casting is not a linear function), with the library replacing the missing gradients with 0. The solution to this was the manual implementation of the backward step as well, consisting of the actual backward step of the original, non-approximated exponential function. Consequently, the back-propagation step does not benefit from the increase in speed that the approximation provides, but it was possible to replace it for the forward step since it does not contain any learnable parameters that might interfere with the publicly available pre-trained weights.

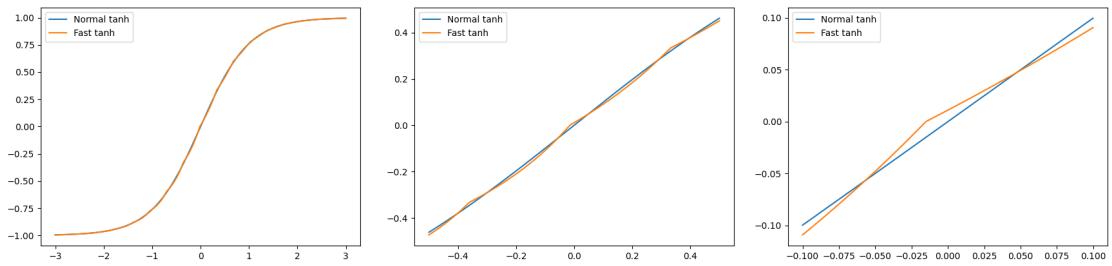

Figure 7: Comparison of the values obtained with the tanh function at 3 different scales. The step used for generating the input was 1e-4.

GELU The Gaussian Error Linear Unit (GELU) is used as an activation function inside the position-wise feed forward layer of ViT. PyTorch has 2 versions implemented, one raw, which abides by the original definition of GELU ($x * \Phi(x)$, where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution), and one that uses a tanh approximation to obtain a similar result.

The tanh approximation version is being used in the primitive due to easier and faster implementation and computation. This is also employed in the GM layer, and, according to the PyTorch documentation, executes the computation in Equation 9.

$$GELU(x) = 0.5 * x * (1 + Tanh(\sqrt{2/\pi} * (x + 0.044715 * x^3))) \quad (9)$$

As for softmax, a faster version is implemented here, by replacing the regular exponential function with its approximation described before. Plots at 3 different scales, comparing the regular tanh function with this one, can be found in Figure 7.

Layer Normalization The layer normalization primitive is parallelized and optimized. It works like in the PyTorch implementation, adapted from the original paper [63], by normalizing across the last n dimensions

of the input array, where n is the number of dimensions of the normalized_shape parameter that must be passed.

The final value in the PyTorch implementation is computed according to Equation 10, x and y are the input and output, $E[x]$ is the average of the values in x, $Var[x]$ is their variance, ϵ is a constant, left to its default 1e-05 value (used to avoid division by 0), and γ and β are the learnable affine transform parameters (the weights and the biases).

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta \tag{10}$$

In the TrainLib primitive, the operations are parallelized over the first dimension, the data being split among the cores in blocks that are as evenly distributed as possible. The computations required are also optimized through a number of mathematical expansions and reductions, with the process being split in 3 steps:

1. Loop once over the elements in the block to compute the necessary sums required for subsequent operations, namely their simple sum, and the sum of their squares.

2. Compute their mean and the square root of the variance, by transforming its formula as described in Equation 11, where $|x|$ is the number of elements in x.

3. Compute the final value of the output, according to Equation 10.

$$
\begin{aligned}
Var[x] &= \sum_i \frac{(x[i] - E[x])^2}{|x|} \\
&= \sum_i \frac{x^2[i] - 2 * x[i] * E[x] + E^2[x]}{|x|} \\
&= \frac{\sum_i x^2[i] - 2 * E[x] * \sum_i x[i]}{|x|} + E^2[x]
\end{aligned}
\tag{11}
$$

3.1.3 ViT Golden Model

An initial step towards deploying a ViT continual training loop on PULP devices has been taken: the creation of a golden model for it. It currently only supports a forward pass, using fp32 data representation, and loads all the data in L2 memory. Thus, to obtain a fully deployable ViT network, that fits on a target board, future steps have to be taken, such as reducing the size of the model by reducing the number of transformer blocks, implement loading and tiling of data into L1 memory, and embedding the continual learning methods.

The GM-based test is robustly and flexibly designed, accepting 2 approaches, one that generates a demonstrative model based on dimensions given inside a configuration file, and another one that generates the golden model and the primitive calls given a PyTorch-specific file, that should contain information about both the configuration of the model and the values contained by its parameters.

All the dimensions related to the demo model can be adjusted, including the input image size, the number of transformer blocks (between 1 and 12), or the number of heads and the hidden dimension of these blocks. All the values required to fill both the mock input and the weights inside the model are randomly generated.

In the case when a pre-trained one is used, these values are automatically obtained and loaded from the given model, as well as given input

data.

To obtain these features, special calls need to be injected in the forward pass of the model, in order to analyze the values and the dimensions inside of it, as well as the input sources of each layer. This ends up passing a list of dictionaries that aim to represent the execution graph of the model. Some user input is also required, in order to match the layer names inside the PyTorch variant of the model with the primitives to be used for each of them.

3.2 ViT and pretrained weights adaptations

3.2.1 Available Implementations and Pre-Trained Models

The staple model that has been followed for the ViT architecture has been made available here [64]. Most details had to remain unchanged, with some exceptions mentioned throughout this thesis, in order to be able to also load the weights that the author provided, and which are strictly connected to the dimensions of the model layers. These weights had been pretrained on the 1k subset of ImageNet [32], with its main architecture being shown in Figure 8, in a linearly unrolled fashion.

This condition also established the required input dimension to 384x384. However, in the CORe50 dataset, the available image sizes are 350x350 and 128x128. For all the experiments reported in this thesis, the 350x350 size has been chosen, and instead of a traditional resizing to 384x384, a bordering one is applied, with the purpose of not introducing any artifacts or other scaling issues. The bordering happens with a neutral gray value (128 when representing it with unsigned 8-bit integers, or 0.5 on a normalized scale), with equally sized borders for left-right and top-bottom.

For translating the latent replay procedure to the ViT model, inspiration has been taken from [57], a project based on Conda Notebooks.

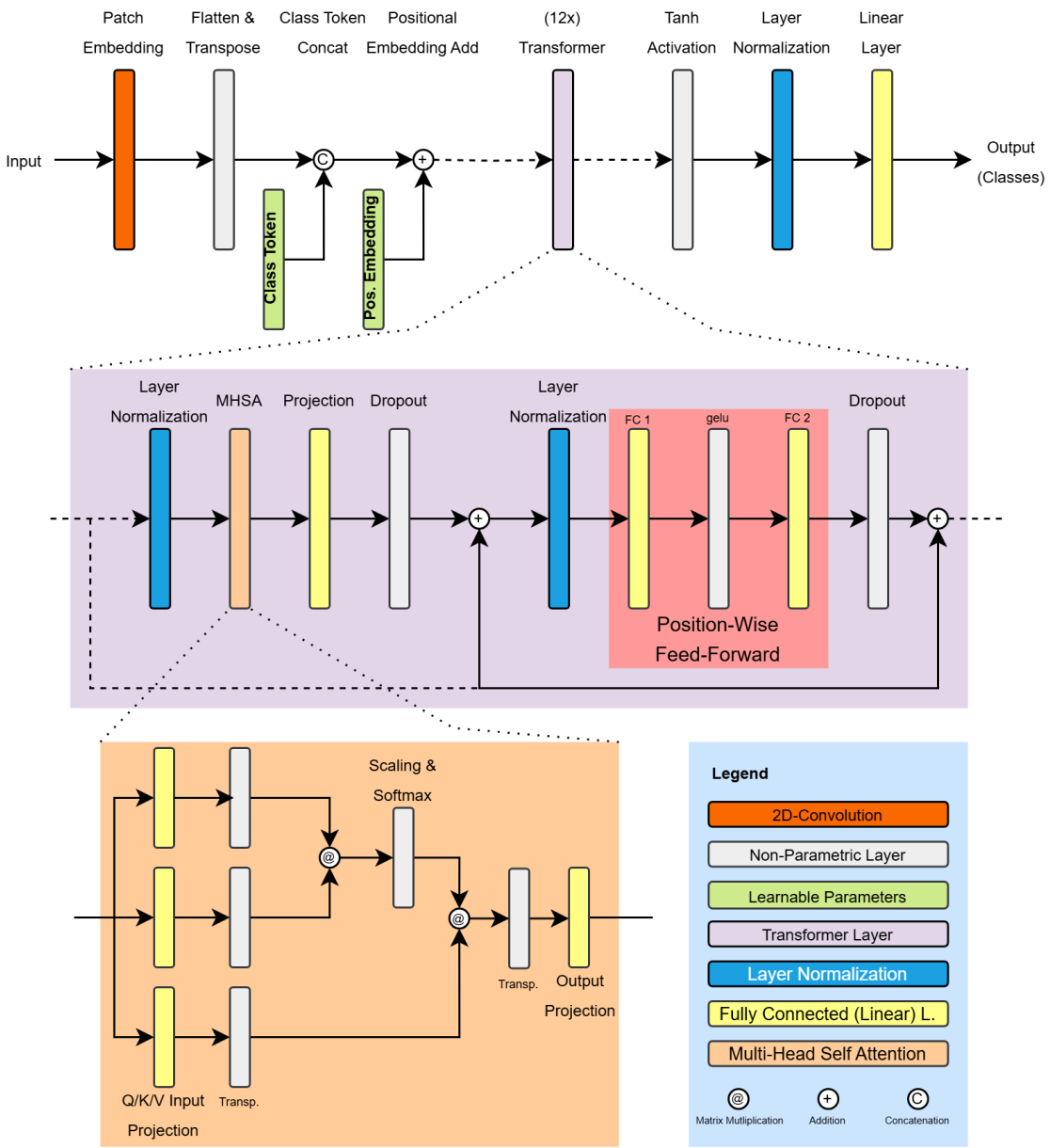

Figure 8: A linearly unrolled visualization of the ViT model implemented in this thesis.

This has been adapted with a more object-oriented command-line-based application, written in pure Python.

3.2.2 Data Loader Adaptations

The data loader is from the implementation provided by the authors of the CORe50 set [33]. However, a number of adaptations and improvements had to be made, since the original one would only allow for using simple batches, loading an entire one in memory, in one of the 3 scenarios (NI, NC, and NIC), which was process intensive.

The first step was to enable loading of custom-sized mini-batches, down to a size of 1, in order to make it compatible with lighter GPUs, since a lot of prototyping has been happening on a laptop-grade GPU.

Another element that had to be introduced was support for the two types of rehearsal used in the current project, native and latent-replay-based. With this purpose, a boolean flag is now passed, together with the model input data, that marks for the functions that need to use it, whether a full input or an activation is involved.

The data loader ends up behaving in 2 different manners, one when no rehearsal is required, so it only loads an original input, with its corresponding label, and a second one for rehearsal scenarios, when it is also responsible for storing the rehearsal memory (RM). This latter one can also be split into 2, one for native rehearsal situations, when the array representing the RM is populated by raw inputs, and one for the latent replay situation, when it is populated by activations received from the model, together with the associated label.

One more feature that was required for certain scenarios was to maintain the ratio of original and rehearsal data in the mini-batch as similar as possible to the batch-level one. The loader is also ready to work with both label splits, either class-wise or category-wise.

3.2.3 Model Adaptions

The main differences in the model, from the default ViT, occur in the handling of the latent replay method. Through the previously mentioned flag attached to input data, the input layer is selected to be at the beginning of the latent replay block, if it has been observed that an activation has been passed, instead of raw data. Furthermore, based on the data that was used in the current forward step, it decides whether it should extract and return the activation from the latent replay layer.

3.2.4 The Training Process

The changes made to the training loop concern especially the optimizer and the mini-batch handling, which are detailed in the following paragraphs. Other elements that are not mentioned, such as the used hyperparameters, or the 4-epoch iteration over each batch, are left to the default values found in [30].

The Optimizer The optimizer used for this project was Stochastic Gradient Descent (SGD), which had to be customized for certain exigencies of the employed continual learning methods. One of them is the different treatment of weights belonging to layers that are before the latent replay one, namely applying a different learning rate or freezing them altogether. This lead to the need of an extra flag stored for each flag that would help the model identify its type, with the pre-latent-replay part being also called backbone in the implementation.

Implementing the Synaptic Intelligence (SI) regularization involved the storage of more variables, such as the weight importance, and its application to the weight update step. As it can be seen in Figure 9, the weight importance for convolutional and transformer-based models are similar, both in absolute value and in distribution.

The "Virtual" Batch Size A "virtual" batch size was used for the training process, method also known as "gradient accumulation" [65]. Through

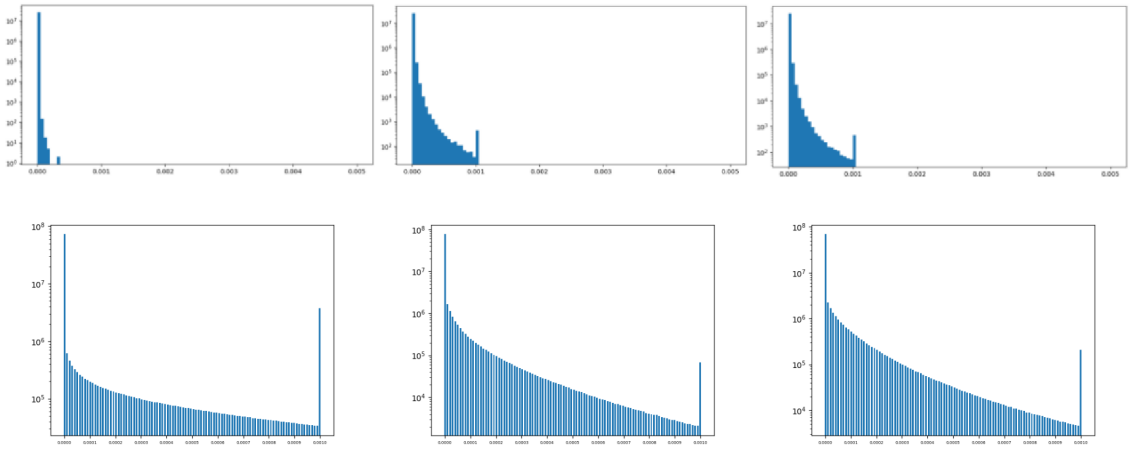

Figure 9: Histogram of the value distribution of the weight importance. The first row is taken from [50], and it represents the distribution obtained with CaffeNet on CORe5 SIT. The second row is the histogram for the implementation described in this thesis. The 3 columns represent the values obtained after training on each of the first 3 batches (represented in their original order). The x-axis represents the value defined by each bin, and the logarithmic y-axis represents the number of values found for each bin.

experimentation, it has been discovered that using the current setup for the ViT model (cross entropy loss, described below, and stochastic gradient descent), it needs a rather large mini-batch in order to be able to converge. Different sizes have been tested, starting from 1, with power of 2 increments. A successful overfit model has only been obtained from a mini-batch size of 32, and the currently used value is 128, like in other publicly available ViT implementations.

In the regular implementation of mini-batch learning, a full set of images is loaded into memory, and the forward and backward passes happen in parallel, for all the samples inside. Thus, separate storage, firstly for all the input data, and secondly for all the gradients computed for all the samples, is required, over all the model parameters. These gradients are accumulated over and used for weight updating, and thus released from the memory, only at the end of a step.

However, this already posed issues when trying to train on the laptop setup, as even the GPU memory there was not enough for loading the model and one mini-batch. This would become even more problematic

on lightweight microprocessors, such as PULP's.

The solution to this issue was the implementation of a so-called "virtual" mini-batch. Thanks to the lack of batch-wise operations, more precisely batch normalization, the images inside the mini-batch can be loaded one-by-one, and the gradients accumulated separately, after each sample is processed by a forward and a backward pass. The accumulation happens simply through summation in a separate array, together with a counter of encountered gradients for each weight, such that latent replay strategies, where mini-batches contain both raw inputs and activations, can be accommodated. In the end, the stored gradients are averaged over the number of encounters, and this value is used inside the optimizer step to perform the weight update.

Thus, only a fraction of the previously required memory is needed, equivalent to that used for a mini-batch of size 1, which enables the training process to take place even on the lighter, laptop-grade GPU, from the setup described in a subsequent section of this thesis, and it prepares it for the training implementation for the microprocessor platform.

Memory and processing time analysis has been undertaken using the TensorBoard tool, developed by the authors of [66]. The experiments happened on the server setup described later in this thesis, with the results available in Figure 10.

Regarding the memory, a number of observations can be drawn, with the first one would about the accumulation process. For the regular mini-batch handling (mini-figure "a" of Figure 10), the allocated memory steadily increases until a peak is reached, ramp which corresponds to the forward and gradient computation pass, and which is reflective of the PyTorch [29] way of storing the gradients. This is followed by a decreasing slope that represents the weight update moment, when these

gradients are gradually released from memory, thus the total allocated memory going down.

In comparison, in the mini-figure "b" of Figure 10, 32 spikes, similar to the single one in "a", can be seen. They correspond to the 32 samples in the mini-batch, with each individual spike following a similar pattern to the single, larger one, in a step of "a". The reason behind this behavior is the same as before since, just like in "a", gradients get accumulated, but this time they get accumulated together instead of being stored individually. Another similar part pops up at the very end of a step, when the update of the weights, and the permanent release of all the gradients, happens.

Section "c" of Figure 10 is used as a memory reference, since it represents the memory requirement for a mini-batch of size 1, handled with the "virtual" mini-batch procedure. Here, there is the equivalent of a single spike in "b", dilated such that it fills the same width inside the plot as a step for a mini-batch of size 32, but on a different time-scale.

In terms of memory usage, a mini-batch peaks at a little above 2,030 MB, regardless of its size, while the classic approach requires more than 9 times that, at almost 19,000 MB, for the same mini-batch size of 32, with this memory requirement scaling alongside mini-batch size.

However, as a drawback, looking at the horizontal axis that measures the execution time in ms, a slight increase can be observed: a step needs around 10% more time for an equally-sized mini-batch. A closer analysis of the report generated by TensorBoard [66] shows that the operations that increase the most from the regular implementation, both in number and execution time, are the ones associated to the handling of the gradients: detaches (so de-allocations of memory) and additions. The conclusion that can be drawn from here is that the parallelization probably doesn't suffer from this, even if the samples of the mini-batch

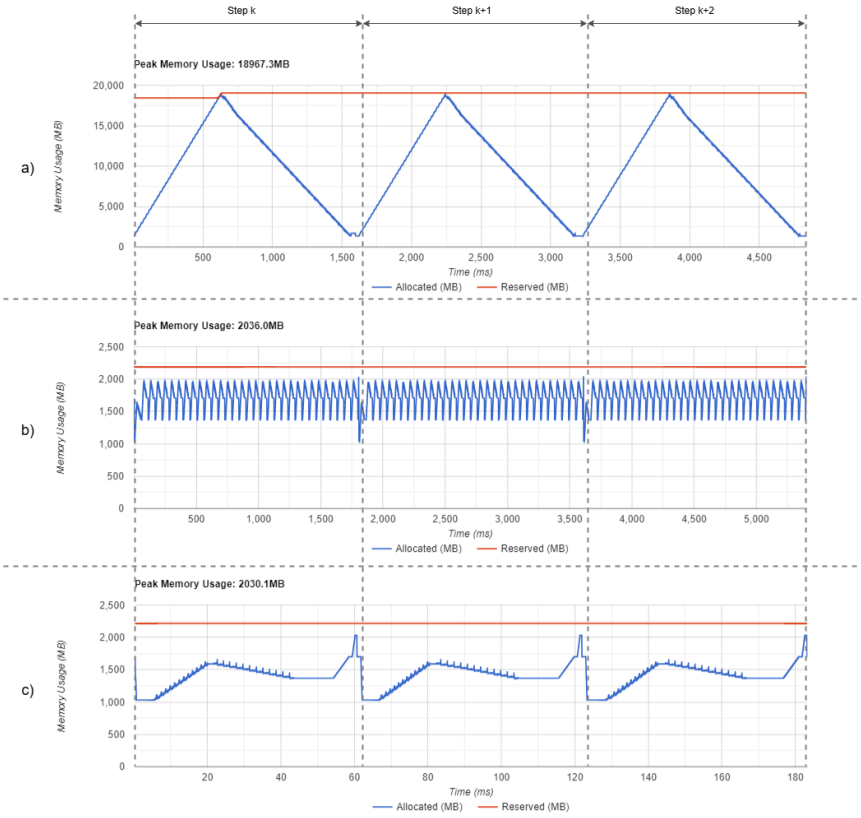

Figure 10: Time and memory plots for different mini-batch configurations, over three consecutive optimizer steps, generically named k, k+1, and k+2. The following setups have been used:
a) Default mini-batch handling with a mini-batch size of 32;
b) "Virtual" mini-batch with size 32;
c) "Virtual" mini-batch with size 1.

are now processed sequentially, but the waiting time to fetch data (albeit the same amount) from the disk into the RAM is the main culprit.

3.3 Latent Replay

Most of the design in the original latent replay paper, both regarding the architectural solutions (CWR*, AR1*, and AR1* free), and the usage of the latent replay itself, has been adapted without changes in functionality to the ViT model. However, a number of elements had to be analyzed and, if needed, adapted. Below, the difference regarding the latent replay layer selection is discussed, and then a case is made about the lack of need to adapt the batch renormalization layer.

3.3.1 Latent Replay (LR) Rehearsal

For the batches following the first one, a mix of new input data and activations is to be used. The activations are extracted from a pre-selected layer, called the LR layer, that doesn't change during the training process. Also, the training of the layers previous to this one is slowed down by using a reduced learning rate, set to 10% of the original one in the experiments conducted by the authors. The reasoning for this design is that early layers are usually responsible for low-level feature extraction, and that these usually have generally representative and stable values, especially in cases of networks pre-trained on large datasets.

The way the training process works is by doing a forward and backward pass through the layers previous to the LR one only for the mini-batch steps that process full input data, and by doing these passes through the layers after the LR point for all types of data: new inputs and also stored activations. The gradients are accumulated over a mini-batch, with pre-LR-layer weights accumulating fewer gradients than post-ones, and at the end of it, the weights are updated accordingly, all in a similar manner, but with different learning rates, as described before.

This rehearsal architecture-based technique is then combined with the during-training technique that performed best with native rehearsal (AR1* free with a rehearsal memory size of 1500), which is tested on the MobileNetV1 [31] convolutional network. The results from Table 1 of [30] are reported in the plots in Figure 11, where accuracy is compared against the computation and storage requirement relative to a non-LR situation, based on the selected LR layer. The names of these layers are the ones from the default MobileNet implementation and can be found in Table 6 of the same paper.

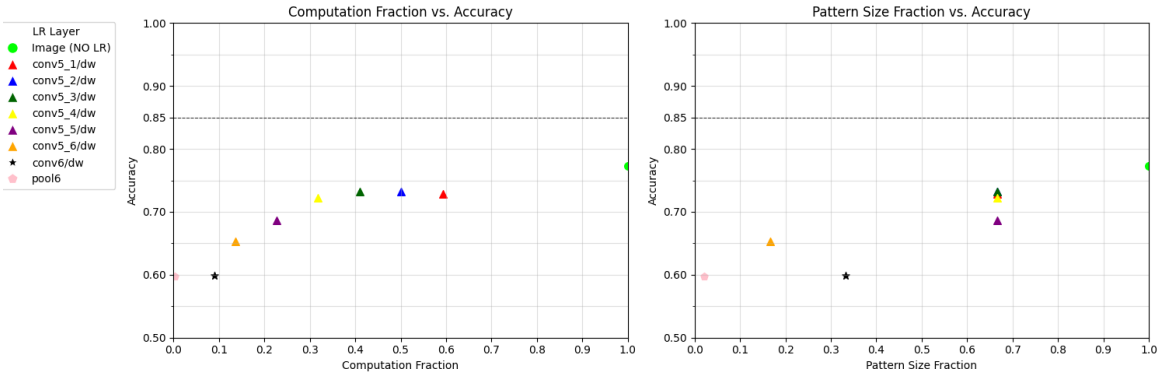

Figure 11: Computation-storage-accuracy trade-off for Latent Replay. Results obtained on MobileNetV1, trained on the NICv2-391 subset of CORe50, with a rehearsal memory size of 1500, and using AR1* free. The values on the x-axis are relative to the non-LR situation (so the fraction of computation or storage required when compared to the situation when full input data is passed), and it's given for a single mini-batch step. The 85% accuracy step is marked as being the best possible results obtained by the authors with the chosen model (trained cumulatively, on the entire set). The accuracy values are obtained as an average over the 10 runs available in the subset. Data source: Table 1 of [30].

3.3.2 Latent Replay Layer Choice

A decision had to be taken regarding the choice of the latent replay layer itself. In the initial implementation [30], that was using a convolution-based MobileNet [31], the latent replay layer was one of the convolutional layers, situated later in the model. This type of layer can be seen as an elementary one, while a transformer is rather a block of layers, containing layer normalizations, dropout layers, fully connected ones, and activations. However, conceptually they represent a single unit, so they are treated as a single layer. Consequently, the user is only given the option to choose one of the 12 transformer blocks as a possible "latent replay layer".

3.3.3 Batch Renormalization to Layer "Renormalization"

A crucial method, according to the authors, that should be included when training with latent replay is **batch renormalization (BRN)**. It is simply a replacement of the traditional batch normalization (BN - [67]), proposed in [68] by the same authors of BN, with the aim to improve performance when working with small and non-i.i.d. mini-batches. It is different from the original BN because of the introduction of a

rolling average, computed during training, that leads to a more stable normalization.

However, this method would not find a place in the ViT model, since it lacks any regular batch normalization layers. However, since BRN is just a modification of BN, a similar change could be translated for the layer normalization layer.

After an analysis of its implementation in PyTorch [29], which can be observed in Equation 10, the affine transform parameters γ and β proved to be proper replacements for the similarly named elements that differentiate BRN from BN. Since they are also learnable, thus making them more robust than the user-defined parameters, that are mentioned as sensitive to tuning in [51], there was no need for the introduction of new parameters.

4 Experiments

4.1 The Dataset

All the experiments are based on the CORe50 dataset, initially described in [33]. It is specialized in continuous learning for image classification, with each image being assigned to a single class.

There are sequences recorded in 11 different scenarios, 8 indoors and 3 outdoors, 8 of all the sequences used for training, and 3 for testing, more precisely sessions marked with ids 3, 7, and 10. Each sequence contains images representing the frames of a 15-second video at 20 fps, with multiple sizes available. Only images of size 350x350 pixels are used in current experiments.

In the set, 10 categories of hand-held objects ("Adapters", "Phones", "Scissors", "Light Bulbs", "Cans", "Sun-glasses", "Balls", "Highlighters", "Cups", "Remote controls") are included, with 5 different objects in each category (identified inside a category by numerical ids), thus resulting 50 classes. Samples of these can be seen in Figure 12.

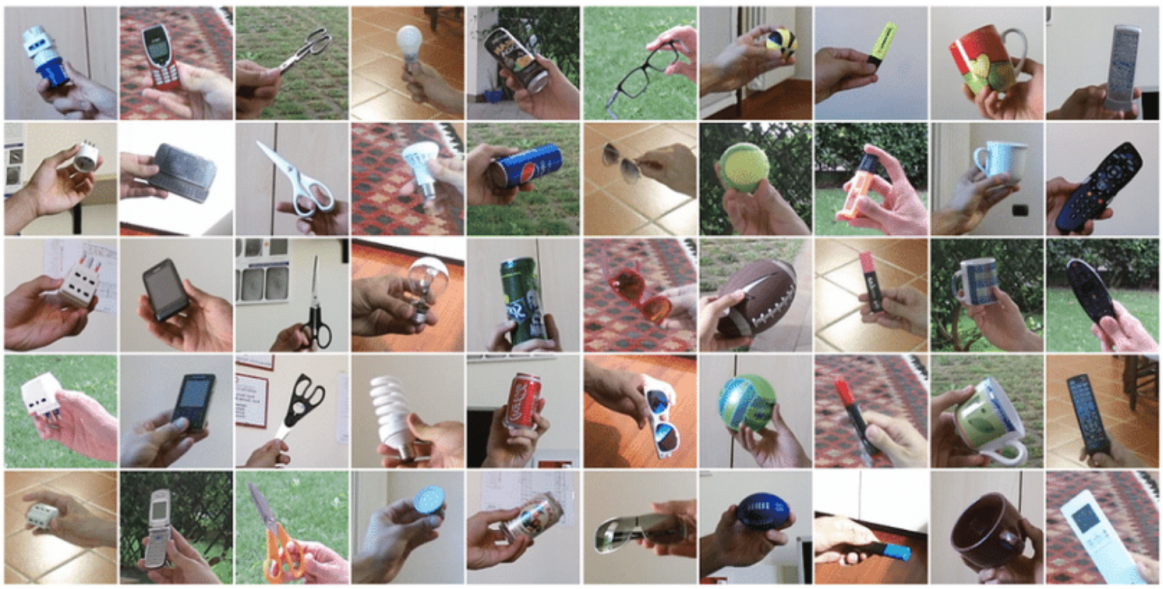

Figure 12: Samples of each class from the CORe50 dataset. Different categories are represented in each column, and different category instances in each row. Source: [33].

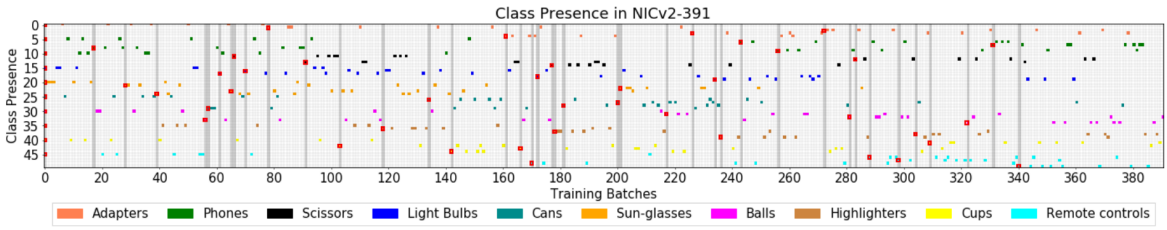

Figure 13: The classes included in each training batch. The horizontal axis represents the batch id, from 0 to 390, and the vertical one, the class id, from 0 to 49. The object classes are color-coded by category, as described in the attached legend. The columns marked with a darker gray represent the batches in which any class is seen for the first time, with that class being specifically marked with a red rectangle around the corresponding cell. Source: [30].

The images are split such that the dataset covers three different scenarios of this task: new instances (NI) - where new distributions, such as different environments or poses in which the object of interest appears are made iteratively available; new classes (NC) - where images with never before seen classes show up in subsequent batches; and new instances and classes (NIC) - the scenario closest to reality, which is a combination of the first 2.

Each scenario includes 10 different runs, a "run" representing a different shuffle of the data in terms of the classes and images included in each batch. In [30], the reported results are an average of trainings and evaluations run on these 10 runs. However, in the current project only the first run is used for computing the metrics (having the id "0" in the original dataset).

The only scenario used for the experiments described in this section is NIC, and more precisely the NICv2-391 protocol, introduced in [51], which contains 391 training batches. The first batch contains images from 10 classes, so a total of approximately 3,000 images, while the remaining 390 incremental batches contain only one class each, so around 300 images per batch. There is also a 392nd batch, used for testing, containing images of all classes, from scenarios not included in the training batches, totaling 44,972 samples. The classes appearing in each batch are reported in Figure 13, plot which will be useful in finding some cor-

relations between this and the confusion matrices reported further down below.

4.2 Setup and Metrics

4.2.1 Hardware Setup

Two main computers have been used for the development of this project, both for training and for evaluation models, with an extra discussion being needed for the PULP simulator that the ViT golden model is being run on.

Prototyping Laptop For developing and prototyping, a simple Lenovo laptop has been used. The hardware specifications that are of interest for running the models consists of an AMD Ryzen 5 5600X CPU (with 3.30 GHz, 6 cores, 12 threads, and a L1/L2/L3 cache split of 384 KB/3 MB/16 MB), 16 GB of RAM, and an NVIDIA GeForce RTX 3060 laptop-grade GPU (6 GB of dedicated memory, and 7.7 GB extra shared memory).

Training Computer For longer, full training sessions, a desktop computer has been used, one that works as a server, and which has been accessed through an SSH connection. It benefits from an AMD Ryzen PRO 5965WX CPU (with 3.8 GHz, 24 cores, 48 threads, and L1/L2/L3 cache split equal to 1.5/12/128 MB), 256 GB of RAM, and two NVIDIA RTX A5000 GPUs, with 24 GB of memory each. However, due to other users needing this computer, only one GPU has been used at a given time.

PULP Simulator For PULP deployment experiments, mainly for implementing the TrainLib primitives and its ViT golden model, a simulator of the PULP platform has been used. This is run through the PULP SDK [69], which uses a tool named GVSoC to simulate PULP chips.

The first obstacle that this project needs to overcome is the limited amount of memory. Inside the simulator, the memory size has been set to 128 KB for L1 memory, and to 8 MB for L2, which is predominantly used at the moment. L3 memory is not used at all, and one of the remaining goals is to move as much as possible of the execution to the faster L1 memory. Also, increases in speed through improved parallelization and other model changes are a subject of possible future improvements.

4.2.2 Metrics

In order to be able to compare with the results in [30], only a simple accuracy metric has been used. Its classical formula is employed: $\frac{\text{correct classifications}}{\text{all classifications}}$, where a classification is correct if the predicted label is identical with the ground truth one (when referring to all 50 classes, not category-wise), and the number of all classifications always is the number of elements in the test set (44,972, as described in a previous section).

For all experiments, confusion matrices can also be provided. They are 2D matrix-shaped plots which give information about the inter-classification of the categories. The value in cell $[i, j]$ of such a matrix has the absolute value equal to the number of times an object belonging to class i has been predicted to be of class j. As the name of the plot tells, they can be used to observe which classes are more often confused with others.

4.2.3 Evaluation Procedure

In [51] and [30], the results are generally reported as an accuracy evolution over the training process. It is measured on the previously discussed evaluation subset, and reported after the network has trained on each batch.

The first paper mentions that, with the purpose of speeding up the evaluation process, because the images are actually frames of video sources, taken at 20 fps, so consecutive ones are very similar, they decide to sub-sample the test set. The sampling frequency they choose is 1 Hz, so only a 20th of the evaluation subset images are used.

However, for the current project, only the loss is monitored during the training loop, and the accuracy of the models is only evaluated at the end of a full training process. This evaluation happens on the same test set, but without the sub-sampling, so on all 44,972 images, belonging to the 3 test sequences, with about 300 frames for each of the 50 classes.

4.3 Results

The reported results can be split into three categories: a performance comparison between transformer and convolution models when using native rehearsal, an analysis of the increase in memory requirements when replacing native rehearsal with latent replay, together with a possible solution to it, and lastly two studies of how the rehearsal memory size and LR layer choice influences accuracy in ViT-based latent replay.

4.3.1 Transformers vs. Convolutions, in the Context of Native Rehearsal

The most extensive performance analysis on the three continual learning methods, CWR*, AR1*, and AR1* free, has been done on the native rehearsal scenario. Consequently, an equivalent one has been performed for the current thesis as well, with a different type of report: instead of showing the evolution of the accuracy during training, the values obtained with the final ViT model will be compared with the ones from [30], which are convolution-based. It is worth noting the difference in size between the models: 4.2 million for MobileNet vs. 93.2 million for ViT

In Figure 14, all 24 configurations are compared based on their ac-

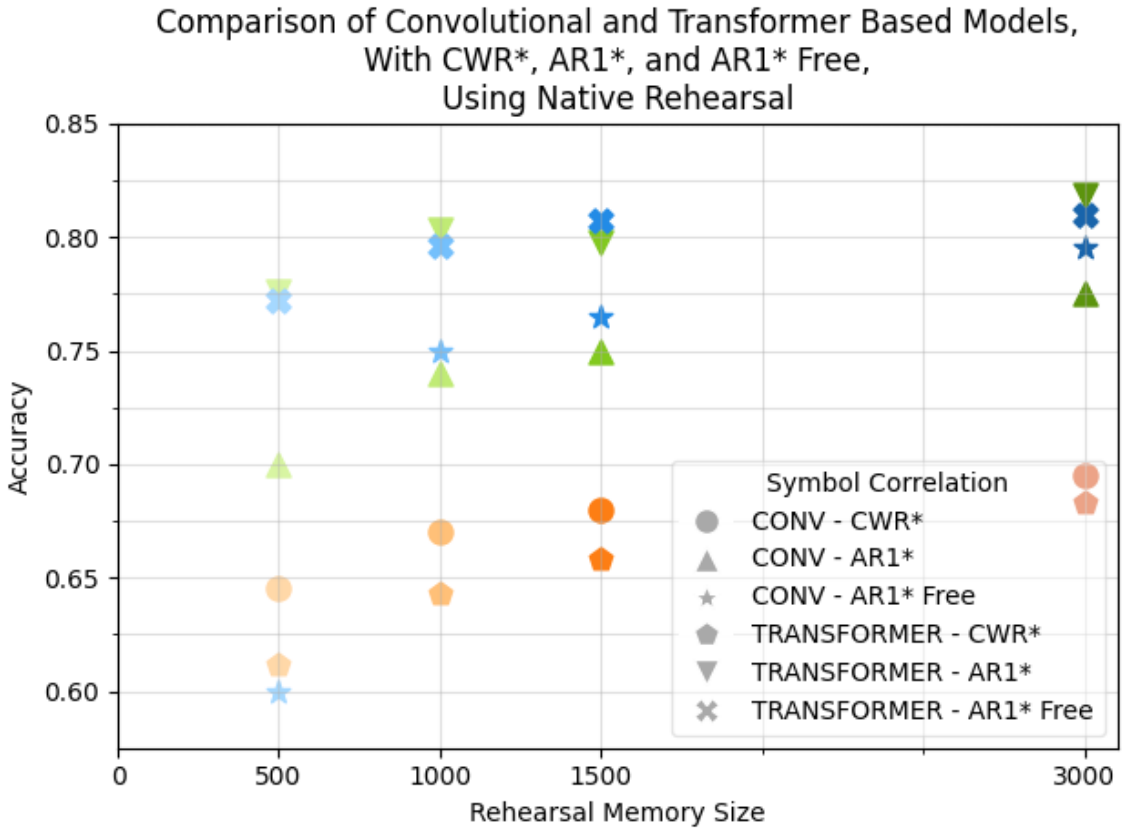

Figure 14: A plot of accuracy evaluated against the rehearsal memory size, for continual learning methods, used on convolutional (MobileNet V1) and transformer (ViT) models, with native rehearsal. The values for the convolutional-based models had to be approximated from [30], since there they are only reported through plots. The colors of the symbols are not relevant in the figure's legend. They are used for correlating the 3 methods (same color used for each of CWR*, AR1*, AR1* free, regardless if it's a convolution or a transformer model).

curacy, with the absolute values for the transformer-based models given in Table 1.

When looking at the differences between identical configurations of separate model types, convolutions perform better in all cases for the CWR* method, but transformers take the lead for the 2 variants of AR1*. The former are, on average, better with 2.34%, while the latter with 5.79%, and 6.89% respectively. This may be explained by the fact that CWR* essentially freezes the backbone, so in the case of ViT, the transformer blocks don't update. Thus, the superiority of the transformers over convolutions is only enabled when the blocks are also adjusted during training.

Rehearsal Size	500	1000	1500	3000
CWR*	61.15%	64.28%	65.85%	68.35%
AR1*	77.64%	80.35%	79.81%	81.84%
AR1* Free	77.20%	79.59%	80.76%	81.00%

Table 1: Accuracy values for the transformer-based model trained with native rehearsal and the 3 continual learning methods, at different rehersal memory size points.

The accuracy of all configurations grows together with the rehearsal memory size, as expected, with an anomaly for the transformer-based AR1*. A particular steep increase happens when going from 500 stored patterns to 1000 for the convolutional-based AR1* free. In general, for convolution models, for a given rehearsal memory size, the increasing order of accuracy for the 3 methods is CWR*, AR1*, and AR1* free. The transformer ones generally show better values for AR1* than AR1* free, meaning that the more complex, weight-importance-based, training method may be better suited for them.

The best overall model is the transformer-based one, trained with AR1* (as opposed to AR1* free for best with convolutions) and a rehearsal size of 3000, reaching 81.84% accuracy. By comparison, the accuracy obtained with the cumulative approach was approximately 85% for the convolution model, and approximately 86% for ViT. This value is considered to be the maximum one for a model because it trains by going through all the data at once, effectively having a rehearsal memory size equal to the entire set.

It is worth mentioning that the values reported in this subsection for the ViT model are for a single run only, while the convolutional-based ones are averaged over 10 runs. Thus, if the variance remains similar with the one in [30], a change of up to 4% in accuracy may be found for the transformers, after averaging over the 10 different runs.

When looking at the confusion matrices in Figures 15 and 16, all methods obtain good results even for the smallest rehearsal memory size. Some categories, such as "mobile_phone" or "glasses", prove to

have instances that are more difficult to differentiate among themselves. However, all gain obvious advantages from an increased rehearsal memory size, with the non-CWR* methods managing almost perfect results. The only category where all models struggles is "adapter" and, indeed, by looking at some examples, in the first column of Figure 12, it can be observed that they have almost identical colors, and also similar shapes and particularities, such as the plug holes.

4.3.2 Latent Replay vs. Native Rehearsal

Since the latent replay method is now applied on a different type of model than in [30], a new memory and speed analysis was needed. As it is explained in the first paragraph, the memory occupancy of latent replay, when comparing to native rehearsal, increases considerably, so the previous advantage becomes a drawback. This leads to an exploration, in the second paragraph, of a way to compensate this decrease in memory efficiency by analyzing ViT based models with smaller numbers of transformer blocks.

Memory and Speed Results In terms of **memory**, the comparison happens between the size of the input images that get stored in the case of native rehearsal and that of the activations that are output by the transformer layers. The former comes in a shape of 384x384x3 (3 channels, since the model processes color images), giving a total 442,368 values to be stored, while the latter has a shape of 577x768, with 443,136 values.

While the absolute number of values is very similar, the activations need to be stored in floating point format, represented in 32 bits in the current implementation. In the meantime, the images are traditionally stored as unsigned integers, with an 8-bit representation, thus giving the latent replay an increase in memory requirement for the rehearsal storage of about 4 times, for an identical number of rehearsal items.

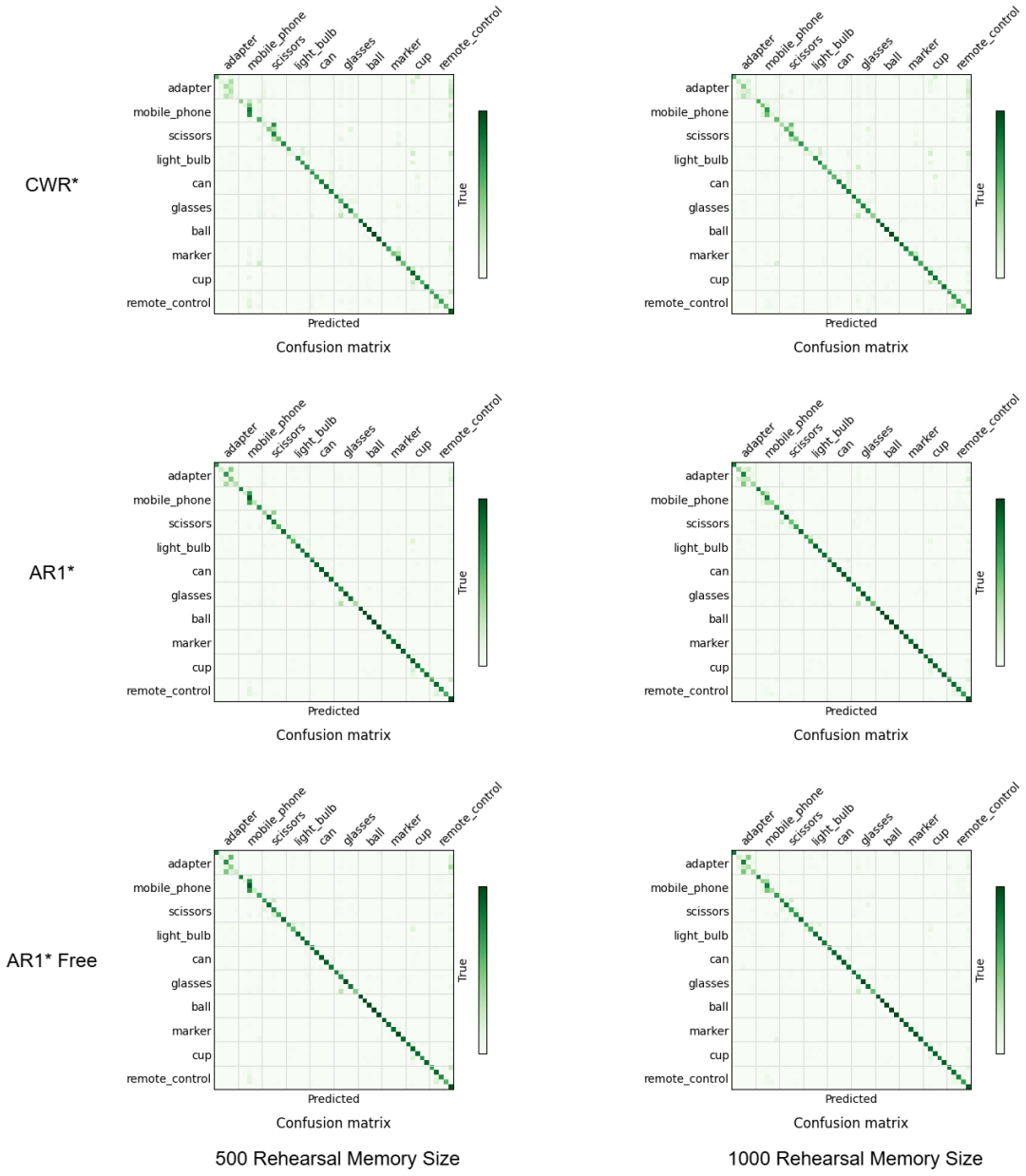

Figure 15: The confusion matrices obtained with ViT models, trained with native rehearsal. Each column of matrices represents the given rehearsal memory size, either 500 or 1000, while each row represents one of the 3 continuous learning methods (CWR*, AR1*, AR1* free, in this order).

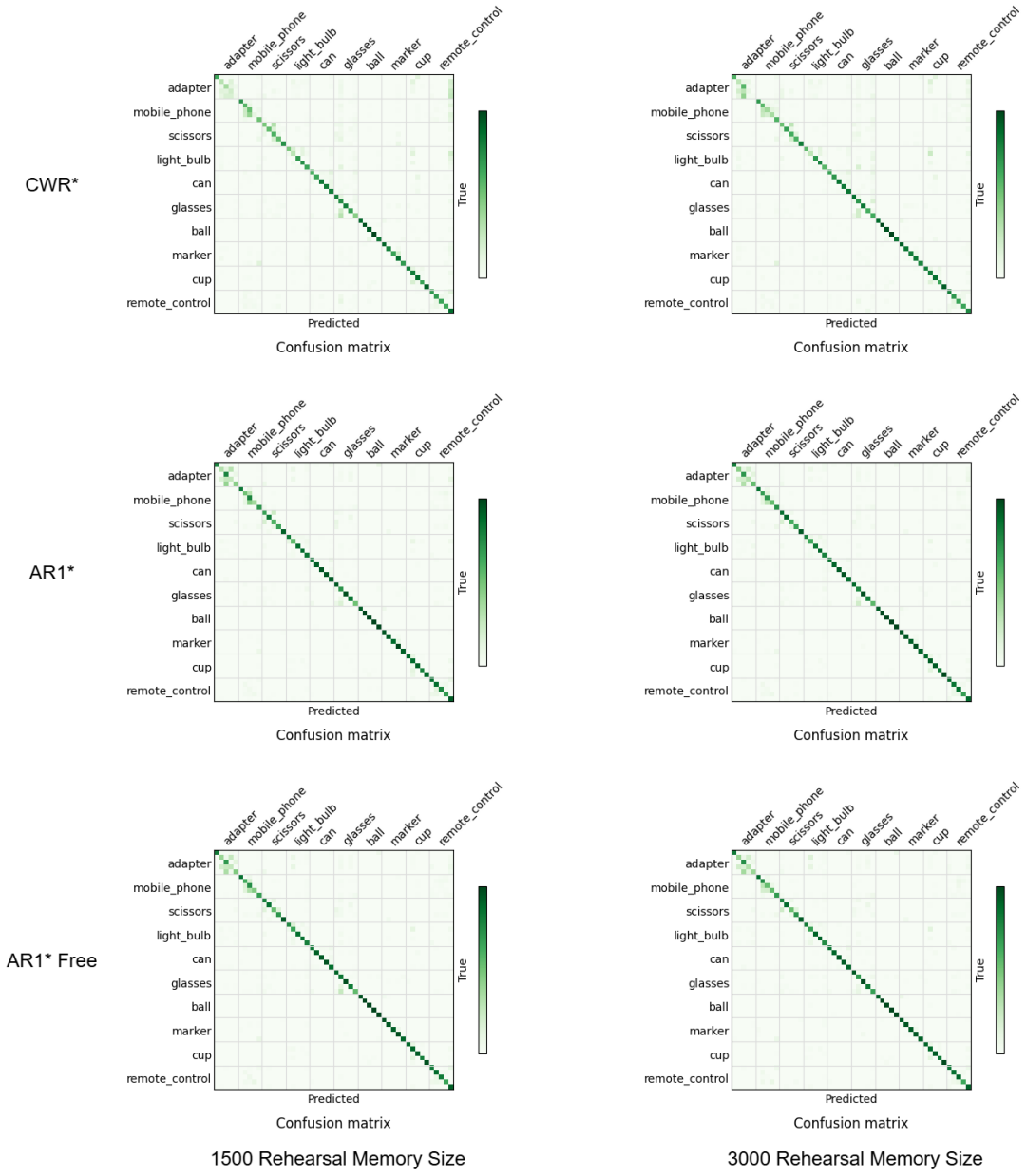

Figure 16: The confusion matrices obtained with ViT models, trained with native rehearsal. Each column of matrices represents the given rehearsal memory size, either 1500 or 3000, while each row represents one of the 3 continuous learning methods (CWR*, AR1*, AR1* free, in this order).

The convolutional-based method described in [30] behaves differently from the transformer one, since choosing a later layer leads to a smaller activation map to be stored, while ViT blocks have an output of constant dimension. However, as seen in Table 1 of the same paper, a moderate decrease in accuracy, of less than 9%, can only be obtained with memory requirements about 2.6 times larger. The memory advantage only appears when choosing conv5_6 or later as the latent replay layer, reducing to about 67% of the native rehearsal size, but with a decrease of more than 12% in accuracy. These values are obtained by considering the activations represented also as fp32.

The **speed** advantage still holds, albeit with lower improvement, since a later latent replay layer leads to less forward and backward propagations to be computed. In the case of [30], this means a computation requirement of 59.26% to 22.62% of the original for a moderate decrease in accuracy, while for ViT, as reported in [57], about 91.72% to 41.73%.

Smaller Model Analysis One solution to decrease the memory needs described in the previous paragraph would be storing the data by approximating it into lower-bit formats usually achieved from higher-bit representations through a process named quantization. There is an entire field studying this type of solutions, sometimes focusing on transformers in general ([70], [71]), on vision transformers in particular ([72]), or specifically on latent replay layers ([73]).

However, this paragraph focuses on adjusting the model size while maintaining the same data representation. A ViT model with a variable number of blocks has been trained using the cumulative scenario, so by passing once over the entire CORe50 dataset, in a randomized order, starting from the same weights pretrained on ImageNet 1k [32], as previously described. In the case of a model with n active transformer blocks, the weights of the first n blocks have been kept from this pretrained initial setup.

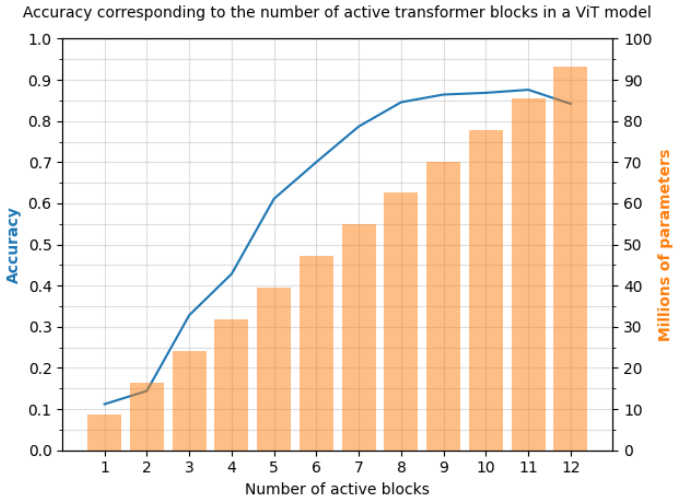

Figure 17: Accuracy results (the blue line) when running a ViT model on the cumulative scenario, having a variable number of transformer blocks included in the model, with the number of parameters represented by the oragne bars. The model with 12 blocks corresponds to ViT-base in [6].

# of tr. blocks	1	2	3	4	5	6	7	8	9	10	11	12
M of train params.	8.75	16.43	24.11	31.79	39.46	47.14	54.82	62.55	70.17	77.85	85.53	93.21
Accuracy	11.20%	14.42%	32.81%	42.85%	61.11%	70.04%	78.69%	84.58%	86.43%	86.86%	87.59%	84.17%

Table 2: Accuracy results when running a ViT model on the cumulative scenario, having a variable number of transformer blocks included in the model.

In Figure 17, the accuracy is reported against the number of active transformer blocks. It can be noticed that the accuracy already approaches the 80% value with only 7 blocks, with it being at 3% from the saturation point by activating one more. Since, as observed in Table 2, the number of trainable parameters in the model increases almost linearly with the number of active blocks, so does the rehearsal memory benefit.

For a full ViT model, the parameter memory would represent between 29.61% and 6.55% of the total needed, when varying its rehearsal size from 500 to 3000 items in the rehearsal memory, values used in other experiments in this section. This means a reduction in the total required memory down to 87.8% of the original, when varying the number of transformer blocks, keeping a reduction in accuracy of less than 6%.

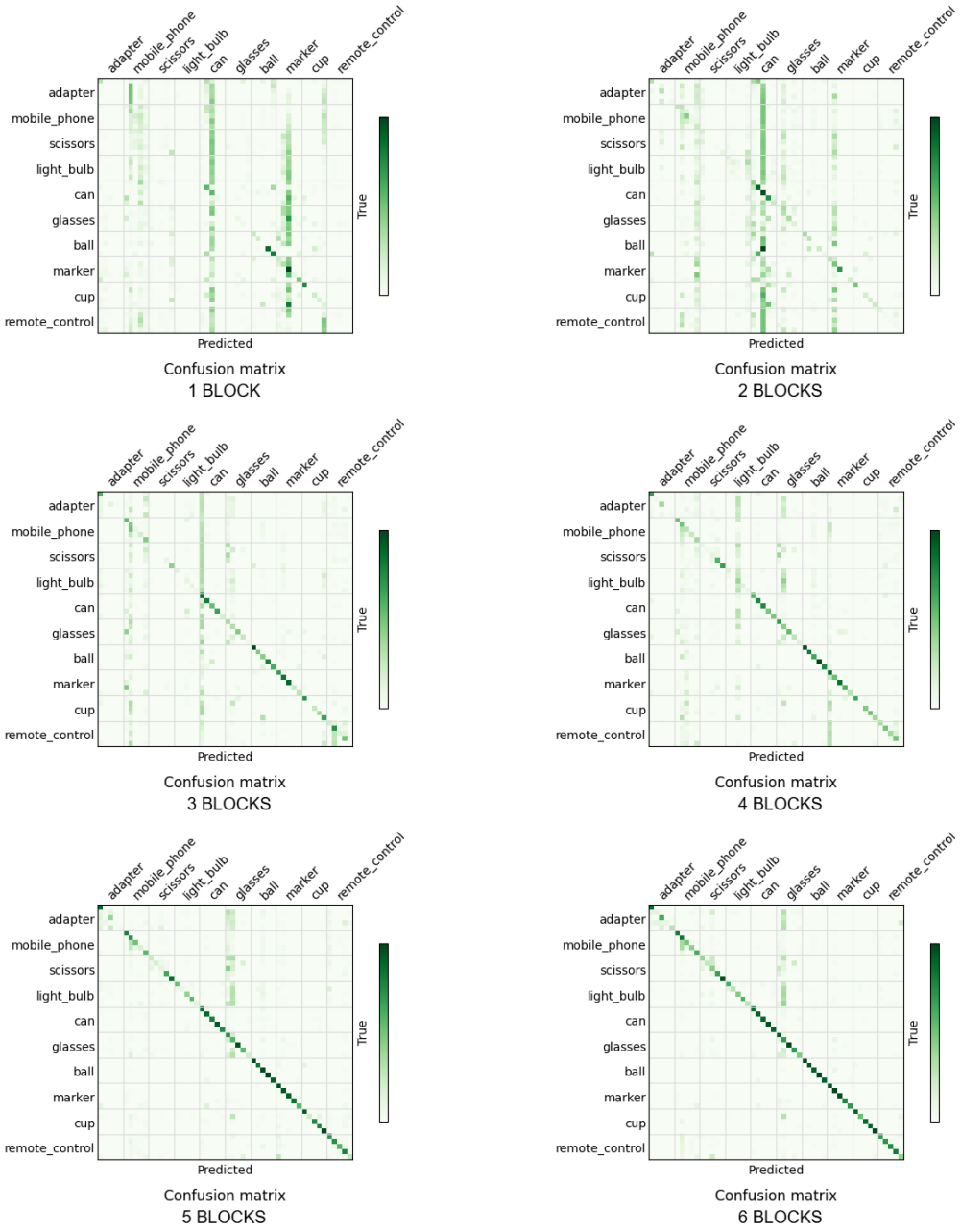

Figure 18: Confusion matrices when running a ViT model on the cumulative scenario, having a variable number of transformer blocks included in the model, between 1 and 6. The label below each subplot represents the number of blocks used for each model.

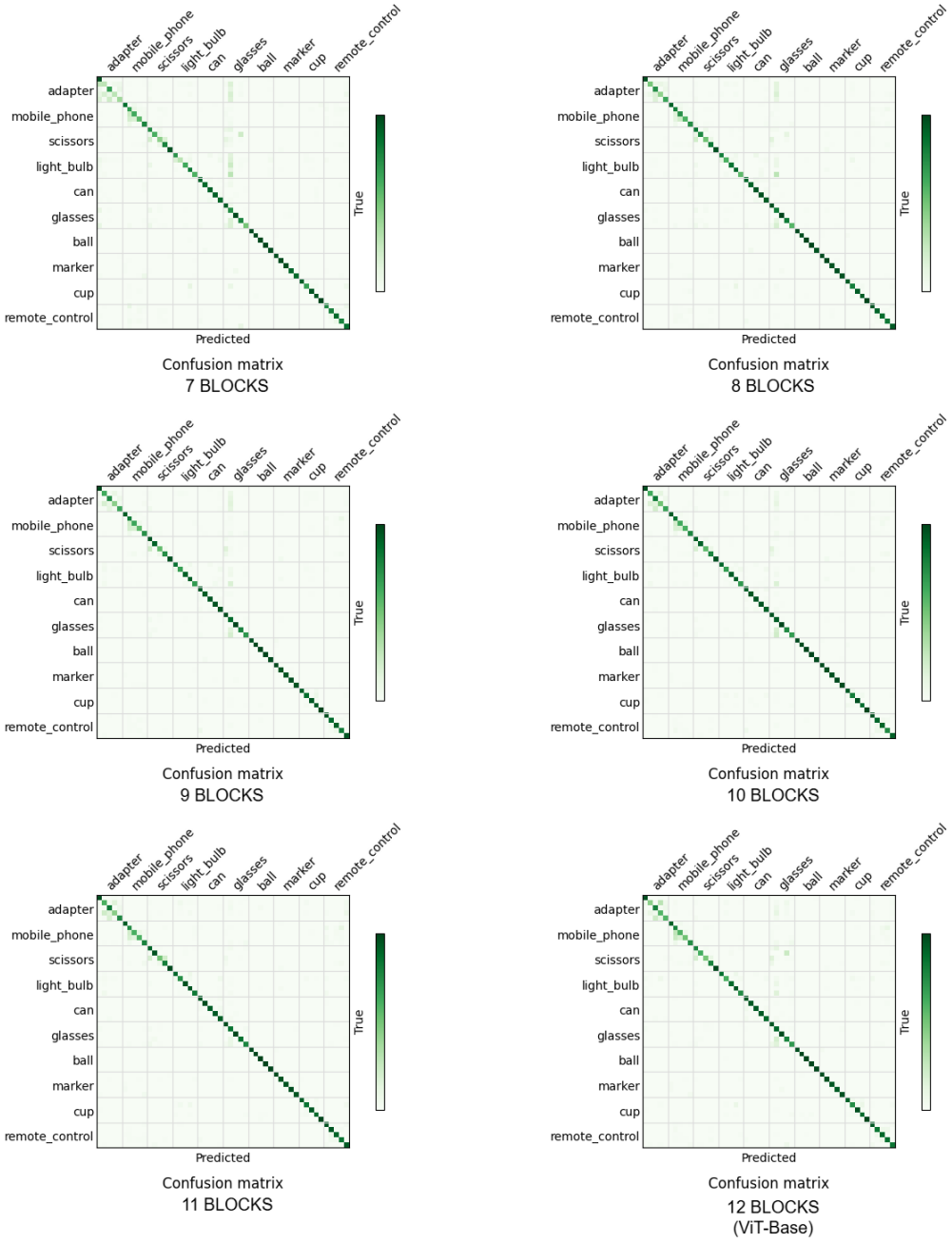

Figure 19: Confusion matrices when running a ViT model on the cumulative scenario, having a variable number of transformer blocks included in the model, between 7 and 12. The label below each subplot represents the number of blocks used for each model.

An analysis has also been done on the classification performance between categories, with difficulties visible in Figure 18 for a low number of available blocks. However, in Figure 19, they fade away by the 7-block mark, and disappear almost altogether from 9 active blocks, even for the most common wrong classes.

4.3.3 Latent Replay on Transformers

For the latent replay implementation on transformers, two types of experiments have been performed: one that looks at the effect of varying the rehearsal memory size, and another one that tries to examine the dependency of the accuracy on the latent replay choice.

LR with CWR* The first test is conducted on a CWR*-infused ViT model, that consecutively receives 500, 1000, 1500, and 3000 rehearsal patterns per batch. It manages to obtain accuracies of 29.25%, 36.73%, 28.37%, and 13.58% respectively. The high confusion, seen in the matrices of Figure 23, corresponds to these values, since they are lower than what has been reported in a similar native rehearsal setup. Possible causes include the frozen backbone and the unsuitable learning rate, if one looks at the spiky training loss evolution in Figure 20.

LR with AR1* Free Regarding the AR1* free experiment, the rehearsal memory size has been set to 1500, and the latent replay layer has been chosen among the first three blocks of the ViT-Base. The losses in Figure 21 exhibit a similar training pattern to the previous setup, with accuracies of 44.859%, 40.401%, 50.291%. This is a counterintuitive result, since the accuracies should decrease when choosing a later LR block. Again, looking at the confusion matrices, found in Figure 24, the accuracies match the ability of the model to differentiate the classes. The main suspected cause of these inferior performance is again improper optimization due to poor learning rate choice.

Losses Sanity Check When comparing the training losses of these two CL methods with a set from an equivalent native rehearsal procedure (Figure 22), one can notice that, while the LR ones are more stable, they have absolute values that are significantly smaller, both a symptom of overfitting given that the model frequently encounters new classes of objects. Thus, these preliminary results show that more fine-tuning of the hyperparameters may be required before these methods can be declared useful in the case of ViT models.

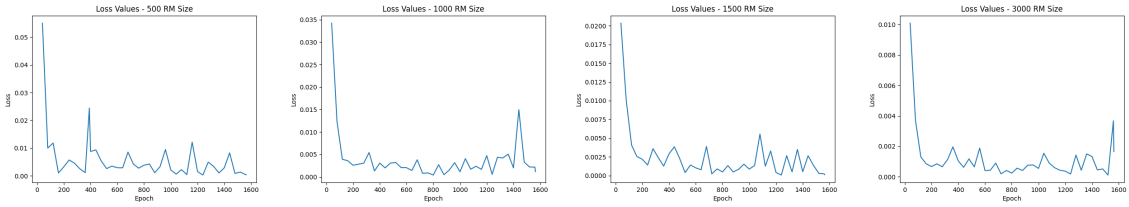

Figure 20: The loss evolution over the epochs when training ViT with latent replay and CWR*, at different rehearsal memory size points.

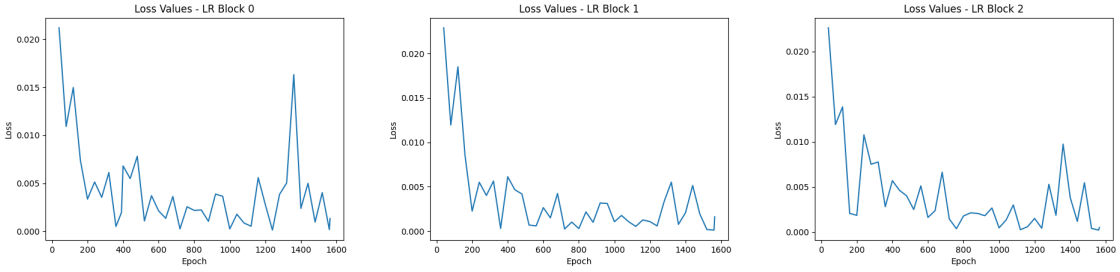

Figure 21: The loss evolution over the epochs when training ViT with latent replay and AR1* free, with different choices for the latent replay layer.

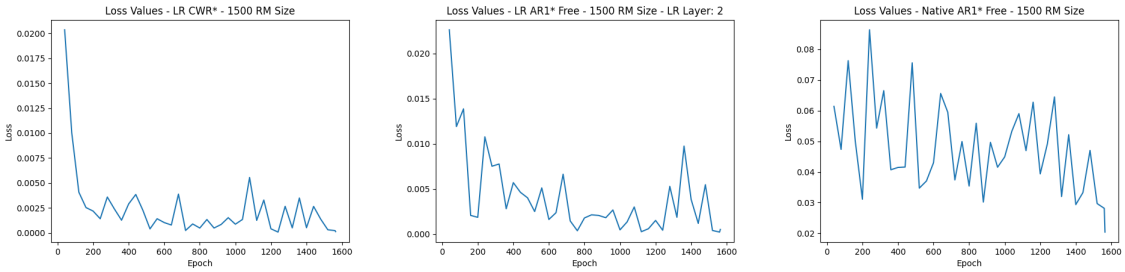

Figure 22: The loss evolution over the epochs when training ViT with 1500 rehearsal size, using native rehearsal AR1* free, LR-CWR*, and LR-AR1* free.

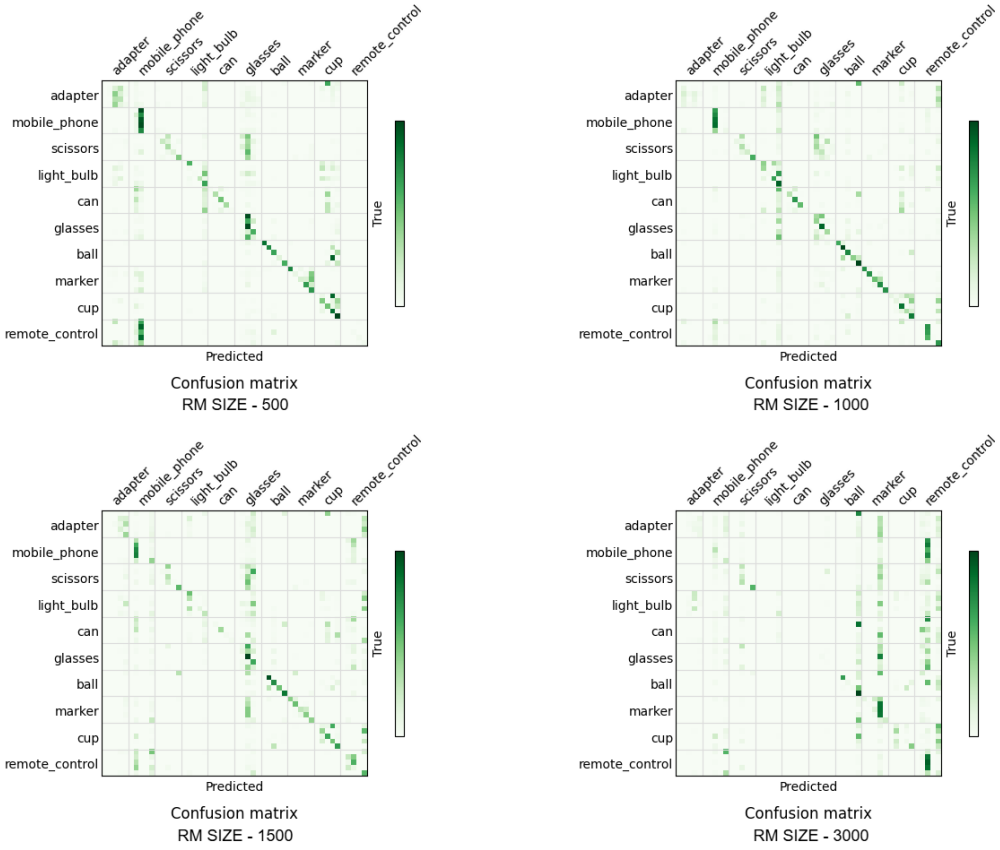

Figure 23: The confusion matrices obtained with ViT, when training with latent replay and CWR*, at different rehearsal memory size points.

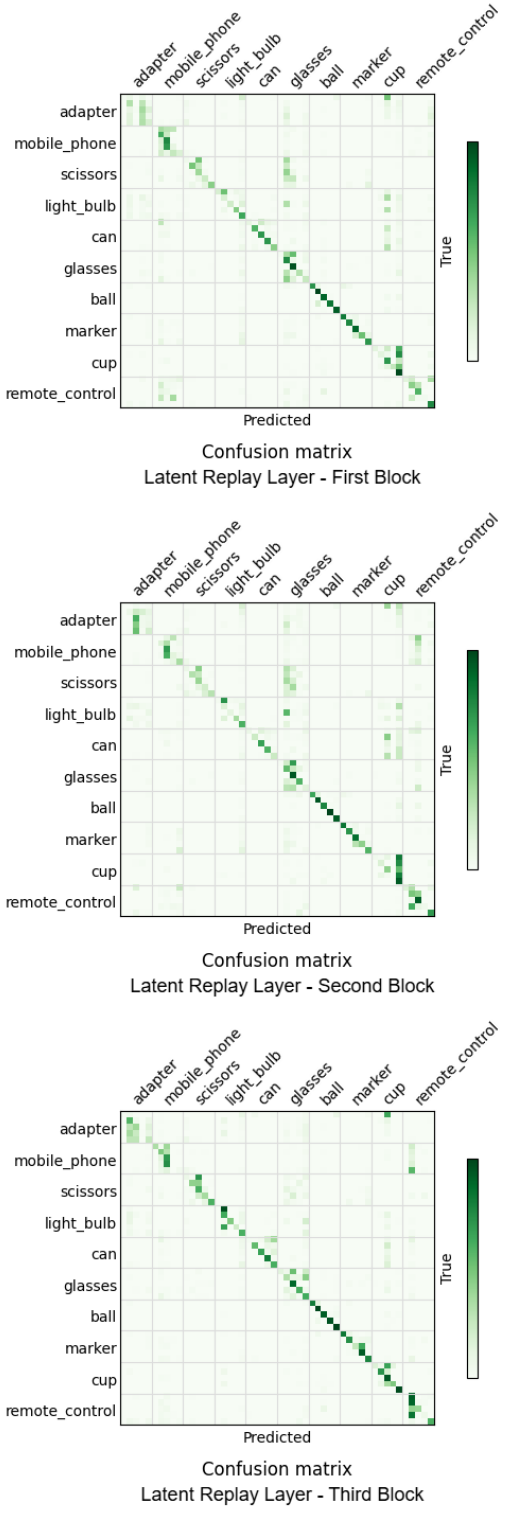

Figure 24: The confusion matrices obtained with ViT, when training with latent replay and AR1* free, with different choices for the latent replay layer.

5 Conclusions and Further Improvements

This work explored a possible solution that would bring together three branches of machine learning: on-device learning, continual learning, and transformers applied on computer vision problems.

Sitting at the intersection of the former two, the thesis has ported a set of continual learning methods from CNNs to the newer, more powerful, self-attention based ViT. After evaluating on a popular set for CL, named CORe50, previous results were surpassed by up to 18%.

The memory increase brought by the newer architecture has been addressed by looking at the block-number/accuracy trade-off in the context of ViT-Base, trained with native rehearsal, and found that cutting out as much as 40% of the initial model leads to a drop in accuracy of at most 6%.

Moreover, the project sets the foundation for a continual learning pipeline deployment on PULP microcontrollers by expanding the TrainLib functionalities with the primitives required for the transformer model, and by creating a highly flexible and customizable golden model test for ViT.

As for possible future directions for this project, the most complex would be finishing the training framework of ViT, inside TrainLib, and extend its functionality to support some of the analyzed continual learning procedures. Another useful tool in this context would also be a general compiler and deployer for ViT and other models, starting from standardized representations, such as ONNX.

Regarding latent replay, variations on this method may include a dynamic choosing of the LR layer, such that newer batches are stored from later points in the model. This may prove to be beneficial, since once all classes have been already seen by the model, for example, the early layers should already have a robust representation space.

Finetuning of hyperparameters could also be desired, such that accuracies when using latent replay with ViT are improved. And also, more stable if the evaluation results would be averaged over the 10 data

shuffles offered by the CORe50 dataset.

Finally, the ViT model may be replaced with smaller variants, such as the TinyViT family of models [74], obtained through distillation from the basic version, and quantization of the latent replay layer [73] may be tested.

References

[1] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[3] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[4] Jie Zhou, Ying Cao, Xuguang Wang, Peng Li, and Wei Xu. Deep recurrent models with fast-forward connections for neural machine translation. *Transactions of the Association for Computational Linguistics*, 4:371–383, 2016.

[5] Yonghui Wu. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[6] Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[7] Pavan Kumar Anasosalu Vasu, James Gabriel, Jeff Zhu, Oncel Tuzel, and Anurag Ranjan. Fastvit: A fast hybrid vision transformer using structural reparameterization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023.

[8] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. Beit: Bert pre-training of image transformers. *arXiv preprint arXiv:2106.08254*, 2021.

[9] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end

object detection with transformers. In *European conference on computer vision*, pages 213–229. Springer, 2020.

[10] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International conference on machine learning*, pages 28492–28518. PMLR, 2023.

[11] Sanyuan Chen, Chengyi Wang, Zhengyang Chen, Yu Wu, Shujie Liu, Zhuo Chen, Jinyu Li, Naoyuki Kanda, Takuya Yoshioka, Xiong Xiao, et al. Wavlm: Large-scale self-supervised pre-training for full stack speech processing. *IEEE Journal of Selected Topics in Signal Processing*, 16(6):1505–1518, 2022.

[12] Yiyang Ma, Xingchao Liu, Xiaokang Chen, Wen Liu, Chengyue Wu, Zhiyu Wu, Zizheng Pan, Zhenda Xie, Haowei Zhang, Liang Zhao, et al. Janusflow: Harmonizing autoregression and rectified flow for unified multimodal understanding and generation. *arXiv preprint arXiv:2411.07975*, 2024.

[13] Zhifei Xie and Changqiao Wu. Mini-omni2: Towards open-source gpt-4o with vision, speech and duplex capabilities. *arXiv preprint arXiv:2410.11190*, 2024.

[14] Fangyu Liu, Julian Martin Eisenschlos, Francesco Piccinno, Syrine Krichene, Chenxi Pang, Kenton Lee, Mandar Joshi, Wenhu Chen, Nigel Collier, and Yasemin Altun. Deplot: One-shot visual language reasoning by plot-to-table translation, 2022.

[15] Alec Radford. Improving language understanding by generative pre-training. -, 2018.

[16] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[17] Giannis Daras and Alexandros G Dimakis. Discovering the hidden vocabulary of dalle-2. *arXiv preprint arXiv:2206.00169*, 2022.

[18] Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models. *Advances in neural information processing systems*, 34:21696–21707, 2021.

[19] Yixin Liu, Kai Zhang, Yuan Li, Zhiling Yan, Chujie Gao, Ruoxi Chen, Zhengqing Yuan, Yue Huang, Hanchi Sun, Jianfeng Gao, et al. Sora: A review on background, technology, limitations, and opportunities of large vision models. *arXiv preprint arXiv:2402.17177*, 2024.

[20] Omer Bar-Tal, Hila Chefer, Omer Tov, Charles Herrmann, Roni Paiss, Shiran Zada, Ariel Ephrat, Junhwa Hur, Yuanzhen Li, Tomer Michaeli, et al. Lumiere: A space-time diffusion model for video generation. *arXiv preprint arXiv:2401.12945*, 2024.

[21] Uriel Singer, Adam Polyak, Thomas Hayes, Xi Yin, Jie An, Songyang Zhang, Qiyuan Hu, Harry Yang, Oron Ashual, Oran Gafni, et al. Make-a-video: Text-to-video generation without text-video data. *arXiv preprint arXiv:2209.14792*, 2022.

[22] José Maurício, Inês Domingues, and Jorge Bernardino. Comparing vision transformers and convolutional neural networks for image classification: A literature review. *Applied Sciences*, 13(9), 2023.

[23] Shigeki Karita, Nanxin Chen, Tomoki Hayashi, Takaaki Hori, Hirofumi Inaguma, Ziyan Jiang, Masao Someki, Nelson Enrique Yalta Soplin, Ryuichi Yamamoto, Xiaofei Wang, et al. A comparative study on transformer vs rnn in speech applications. In *2019 IEEE automatic speech recognition and understanding workshop (ASRU)*, pages 449–456. IEEE, 2019.

[24] Alessio Burrello, Moritz Scherer, Marcello Zanghieri, Francesco Conti, and Luca Benini. A microcontroller is all you need: Enabling transformer execution on low-power iot endnodes. In *2021*

IEEE International Conference on Omni-Layer Intelligent Systems (COINS), pages 1–6, 2021.

[25] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[26] Brendan C Reidy, Mohammadreza Mohammadi, Mohammed E Elbtity, and Ramtin Zand. Efficient deployment of transformer models on edge tpu accelerators: A real system evaluation. In *Architecture and System Support for Transformer Models (ASSYST@ ISCA 2023)*, 2023.

[27] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing. *IEEE Journal of Solid-State Circuits*, 54(7):1970–1981, 2019.

[28] Davide Nadalini, Manuele Rusci, Giuseppe Tagliavini, Leonardo Ravaglia, Luca Benini, and Francesco Conti. Pulp-trainlib: Enabling on-device training for risc-v multi-core mcus through performance-driven autotuning. In Alex Orailoglu, Marc Reichenbach, and Matthias Jung, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 200–216, Cham, 2022. Springer International Publishing.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[30] Lorenzo Pellegrini, Gabriele Graffieti, Vincenzo Lomonaco, and Davide Maltoni. Latent replay for real-time continual learning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10203–10209. IEEE, 2020.

[31] Andrew G Howard. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

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

[34] Gaurav Batra, Zach Jacobson, Siddarth Madhav, Andrea Queirolo, and Nick Santhanam. Artificial-intelligence hardware: New opportunities for semiconductor companies. *McKinsey and Company*, 2, 2019.

[35] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. A survey of on-device machine learning: An algorithms and learning theory perspective. *ACM Transactions on Internet of Things*, 2(3):1–49, 2021.

[36] Shuai Zhu, Thiemo Voigt, JeongGil Ko, and Fatemeh Rahimian. On-device training: A first overview on existing systems. *arXiv preprint arXiv:2212.00824*, 2022.

[37] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. Mandheling: Mixed-precision on-device dnn training with dsp offloading. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 214–227, 2022.

[38] Luka Macan, Alessio Burrello, Luca Benini, and Francesco Conti. Wip: Automatic dnn deployment on heterogeneous platforms: the gap9 case study. In *Proceedings of the International Conference*

on Compilers, Architecture, and Synthesis for Embedded Systems, pages 9–10, 2023.

[39] Davide Nadalini, Manuele Rusci, Luca Benini, and Francesco Conti. Reduced precision floating-point optimization for deep neural network on-device learning on microcontrollers. *Future Generation Computer Systems*, 149:212–226, 2023.

[40] Tony Chen and David A Patterson. Risc-v geneology. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-6*, 2016.

[41] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. Quentin: an ultra-low-power pulpissimo soc in 22nm fdx. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3, 2018.

[42] Ludwig Boltzmann. Studien uber das gleichgewicht der lebenden kraft. *Wissenschafiliche Abhandlungen*, 1:49–96, 1868.

[43] Jade Copet, Felix Kreuk, Itai Gat, Tal Remez, David Kant, Gabriel Synnaeve, Yossi Adi, and Alexandre Défossez. Simple and controllable music generation, 2023.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[45] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10687–10698, 2020.

[46] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. *arXiv preprint arXiv:1906.01787*, 2019.

[47] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*, 2018.

[48] Md Yousuf Harun, Jhair Gallardo, Tyler L Hayes, Ronald Kemker, and Christopher Kanan. Siesta: Efficient online continual learning with sleep. *arXiv preprint arXiv:2303.10725*, 2023.

[49] Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet K Dokania, Philip HS Torr, and Marc'Aurelio Ranzato. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*, 2019.

[50] Davide Maltoni and Vincenzo Lomonaco. Continuous learning in single-incremental-task scenarios. *Neural Networks*, 116:56–73, 2019.

[51] Vincenzo Lomonaco, Davide Maltoni, Lorenzo Pellegrini, et al. Rehearsal-free continual learning over small non-iid batches. In *CVPR Workshops*, volume 1, page 3, 2020.

[52] Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

[53] Hongjoon Ahn, Sungmin Cha, Donggyu Lee, and Taesup Moon. Uncertainty-based continual learning with adaptive regularization. *Advances in neural information processing systems*, 32, 2019.

[54] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International conference on machine learning*, pages 3987–3995. PMLR, 2017.

[55] Arthur Douillard, Alexandre Ramé, Guillaume Couairon, and Matthieu Cord. Dytox: Transformers for continual learning with dynamic token expansion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9285–9295, 2022.

[56] German I Parisi, Jun Tani, Cornelius Weber, and Stefan Wermter. Lifelong learning of spatiotemporal representations with dual-memory recurrent self-organization. *Frontiers in neurorobotics*, 12:78, 2018.

[57] Alberto Dequino, Francesco Conti, and Luca Benini. Vit-lr: Pushing the envelope for transformer-based on-device embedded continual learning. In *2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC)*, pages 1–6, 2022.

[58] Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

[59] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[60] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. *Advances in neural information processing systems*, 30, 2017.

[61] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2001–2010, 2017.

[62] Nicol N Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862, 1999.

[63] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[64] L. Melas-Kyriazi. Vit pytorch. https://github.com/lukemelas/PyTorch-Pretrained-ViT, 2020.

[65] Charles M Stein, Dinei A Rockenbach, Dalvan Griebler, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Luiz G Fernandes. Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. *Concurrency and Computation: Practice and Experience*, 33(11):e5786, 2021.

[66] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[67] Sergey Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[68] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems*, 30, 2017.

[69] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 409–416, 2021.

[70] Chaofei Fan. Quantized transformer. *Tech. Rep.*, 2019.

[71] Harshavardhan Adepu, Zhanpeng Zeng, Li Zhang, and Vikas Singh. Framequant: Flexible low-bit quantization for transformers. *arXiv preprint arXiv:2403.06082*, 2024.

[72] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. Post-training quantization for vision transformer. *Advances in Neural Information Processing Systems*, 34:28092–28103, 2021.

[73] Leonardo Ravaglia, Manuele Rusci, Davide Nadalini, Alessandro Capotondi, Francesco Conti, and Luca Benini. A tinyml platform for on-device continual learning with quantized latent replays. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(4):789–802, 2021.

[74] Kan Wu, Jinnian Zhang, Houwen Peng, Mengchen Liu, Bin Xiao, Jianlong Fu, and Lu Yuan. Tinyvit: Fast pretraining distillation for small vision transformers. In *European conference on computer vision*, pages 68–85. Springer, 2022.