Meta-learning for few-shot on-chip sequence classification

by

Douwe M. J. den Blanken

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday September 28, 2023 at 13:00.

Student number:	4843940	
Project duration:	November 14, 2022 – September 23, 2023	
Thesis committee:	Dr. C. Frenkel,	TU Delft, daily supervisor
	Prof. dr. K.A.A. Makinwa,	TU Delft, department head
	Prof. M. Verhelst.	KU Leuven, external jury

This thesis is confidential and cannot be made public until September 28, 2025.

Disclaimer on open-source contributions

As part of the effort to increase and promote reuse in the hardware design space, the following items are published as open-source:

- hardware design: https://github.com/VOXNIHILI/chameleon,
- meta-learning benchmarking suite: https://github.com/VOXNIHILI/meta-learning-arena,
- all experimental machine learning data: https://wandb.ai/douwe/meta-learning-arena,
- all vector-graphics figures: https://drive.google.com/file/d/1447Nd-lguDoKupiJSHBD_PKoScaEVtCt/view?usp=sharing.

Furthermore, throughout the course of this thesis, contributions were made to the following open-source repositories:

- PyTorch (machine learning framework): https://github.com/pytorch/pytorch,
- QPyTorch (reduced bitwidth simulation): https://github.com/Tiiiger/QPyTorch,
- Harvard Multilingual keyword spotting project: https://github.com/harvard-edge/multilingual_kws,
- AudioLoader (collection of PyTorch speech datasets): https://github.com/KinWaiCheuk/AudioLoader.

"I didn’t have time to write a short letter,
so I wrote a long one instead." – Mark Twain

Acknowledgements

Here, I would like to make use of the opportunity to express my heartfelt gratitude to all those who have contributed to the successful completion of this master's thesis. This journey would not have been possible without their support, guidance, and encouragement.

First and foremost, I want to send out my deepest appreciation to my thesis advisor, Charlotte Frenkel; for her support and mentorship, for her all-in commitment to my work and for the many opportunities I was offered throughout the entire research process. From enabling me to do a tape-out to giving me the chance to visit conferences and training courses. Without you, none of these things would have been possible. Your expertise and 24/7-dedication played a pivotal role in shaping all aspects of my thesis. Thank you for everything.

For the silicon design specifically, I want to thank Nandor, Nicolas and Martin: without you, I would not have been able to navigate so quickly through many of the issues I encountered. Also, I want to express my gratitude to Atef for, even after your retirement, helping us until the very end of the tape-out. Furthermore, I would like to send out a huge thank-you to Yihan, with whom I led our subtape-out group. You have been instrumental in making sure all the hardware came together! I would also like to thank my brother Tim for designing an amazing top mask for the final hardware design.

Moreover, I'm really thankful for all the help and support from my friends and family. First, I want to thank my parents and brother for being there for me throughout my life and especially during this busy time of mine. I also want to thank my girlfriend for consistently offering her love, support, and a sense of calm during this journey.

And last but not least, thanks to Artemis and Charis, Froso and Kyriakos for taking care of everything over the summer, which allowed me to focus on my thesis during the holidays.

Douwe

Abstract

The growing interest in edge computing is driving the demand for more efficient deep learning models that fit into resource-constrained edge devices like Internet-of-Things (IoT) sensors. The challenging limitations of these devices in terms of size and power has given rise to the field of tinyML, focusing on enabling low-cost machine learning on edge devices. Up until recently, the work in this space was primarily focused on static inference scenarios. However, a prominent issue with this is that models cannot adapt post-deployment, leading to robustness issues with shifting data distributions or the introduction of new features in the data. However, at the edge, full on-device retraining, or communicating all new data to a central server, is infeasible: this necessitates the development of data-efficient learning algorithms to adapt locally and autonomously from streaming data. This challenge at the intersection of edge computing and data-efficient learning is currently an open challenge.

In this thesis, we propose to solve this challenge with *meta-learning*. To clarify in which way the application of meta-learning is the most suitable for edge hardware, for the first time, a principled approach for meta-learning at the edge is outlined and investigated in three parts.

The first part of this thesis details the selection of a suitable neural network architecture for few-shot learning over *sequential* data. By not being fixated on one architecture from the start, it is possible to explore different approaches to learning over sequences of temporal data, leading to the identification of the most effective architecture for generalizing from limited temporal examples. The quantitatively evaluated architectures are a recurrent neural network (RNN), a gated recurrent unit (GRU), a long-short-term memory (LSTM) and a temporal convolutional network (TCN). We show that TCNs outperform all architectures, while GRUs and LSTMs have a lower activation memory requirement. However, the latter require a linearly increasing number of multiplications with input sequence length, while it scales logarithmically for TCNs. Our results show that TCNs therefore provide the most favorable trade-off for low-cost temporal feature extraction at the edge.

The second part of the thesis focuses on the algorithmic developments of the few-shot learning setup. Building on recent results from machine learning research, we highlight how meta-learning techniques primarily rely on learning high-quality features that generalize well. Taking into account hardware-driven considerations such as memory and compute overheads and through detailed quantitative analyses, we demonstrate that the best performance-cost trade-off is reached with a simple supervised pre-training scheme, where on-chip learning is performed by comparing the outputs of a TCN-based feature extractor with Manhattan distance. We also analyze the impact of quantization on this trade-off and, accordingly, we select a scheme with 4-bit logarithmic weights and 4-bit unsigned activations.

Building on these results, the third and final part of the thesis covers the design and implementation of an application-specific integrated circuit (ASIC) for few-shot learning from temporal data at the edge, taped out in a TSMC 40-nm technology node with a sub-mm^2 core area, which we codename *Chameleon*. The design follows a typical accelerator-style architecture and supports per-layer variable kernel sizes for up to 16 TCN layers. Processing such networks takes place in a 16×16 processing-element (PE) array performing bit shifts instead of multiplications thanks to the use of logarithmic weights. To benchmark Chameleon, we select two tasks consisting in 5- and 20-class, single-shot classification of never-seen-before handwritten characters from the Omniglot dataset, for which we obtain accuracies of 97.9% and 93.6%, respectively. Furthermore, we show that these on-chip learning capabilities do not degrade the efficiency of regular classification tasks. Indeed, streaming inference for real-time 16-ms-latency keyword spotting can be done at a clock speed of only 4.38 kHz, yielding a 93.5-% classification accuracy on the Google Speech Commands (GSC) dataset.

Our results pave the way for low-cost few-shot learning over temporal data, at the edge. By enabling edge devices to perform this local and data-efficient learning, these devices will be able to adapt autonomously to shifting data distribution or new features altogether, while maintaining user privacy.

Nomenclature

AI	Artifical Intelligence
ANIL	Almost No Inner Loop
ASIC	Application-Specific Integrated Circuit
CL	Continual Learning
DL	Deep Learning
FOMAML	First-Order MAML
GSC	Google Speech Commands
KD	Knowledge Distillation
KWS	KeyWord Spotting
LLMs	Large Language Models
LR	Logistic regression
LSTM	Long Short-Term Memory
MAML	Model-Agnostic Meta-Learning
ML	Machine Learning
MSE	Mean-squared error
MSWC	Multilingual Spoken Words Corpus
NIL	No Inner Loop
PN	Prototypical Networks
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
RNN	Recurrent Neural Network
STE	Straight-through estimator
TCN	Temporal Convolutional Network

Contents

1 Introduction

Along with the increasing interest in edge computing [1], there is a growing need for deploying deep learning (DL) models on edge devices [2], enabling networked smart end-nodes and sensors that form the backbone of the Internet-of-Things (IOT) ecosystem [2]. However, the size and power limitations of these devices contrast with the processing requirements for traditional neural networks [2]. This challenge led to the creation of a new machine learning branch called tiny machine learning (tinyML). [3].

However, most modern tinyML devices are currently focusing on inference [2], exhibiting mostly *static* edge intelligence. This means that the machine learning model has to be trained before deployment [4] and is not able to adapt post-deployment, resulting in a system that cannot accommodate for data distribution shifts over time [2] or new features in data altogether. While the former can cause smart sensors to become unreliable after deployment [5], the latter can cause models to fail in situations that were not predicted before deployment [6].

One possible way to combat this problem is to upload locally sourced data to a centralized server, where it is used to extend the original training dataset [2]. The model then has to be retrained from scratch in order to emulate an independent and identically distributed (*i.i.d.*) dataset, which is necessary to ensure that the new data is taken into account during learning, without forgetting the original information [7]. After this, the new model has to be downloaded to the edge device, which is then able to perform accurate inference on the new data. However, this approach not only costs time, meaning that real-time data changes cannot be dealt with on-the-fly [4], it also requires the device to record, store, upload and download data throughout its life cycle. Furthermore, while the majority of developments linked to edge computing and low-cost machine learning models has historically focused on static image recognition, streaming data (e.g., temporal information such as sound or video) is actually much more representative of real-world edge scenarios, which implies the use of neural network architectures that are able to maintain a state over time.

As conventional retraining is therefore not an option for edge devices, on-device-learning emerges as a key requirement. However, a new challenge arises at the intersection of edge computing and data-efficient learning, which requires a minimal amount of data to acquire knowledge and adapt to new tasks. This challenge necessitates the development of learning algorithms that excel in scenarios where the use of conventional training methods is not feasible. Overall, endowing edge devices with the ability to learn new tasks and features from their environment with little data will directly contribute to:

- data privacy, which is increased by keeping the user data on-chip [1], [8],
- long-term robustness, which is improved as changes in tasks over time can be dealt with locally and in real time,
- electronic waste reduction, thanks to a longer device life cycle,
- maintenance costs reduction, thanks to a reduced reliance on over-the-air updates,
- a reduced dependence on communication network reliability in non-urban areas, as the cloud is removed from the learning process.

These advantages underscore the importance of advancing algorithms that have this ability. *Meta-learning*, a machine learning framework where models learn to learn efficiently, is among the most promising candidates to enable data-efficient learning at the edge. Therefore, the main research question of this thesis is as follows:

"How can meta-learning unlock low-cost adaptation to temporal data for deep neural network accelerator hardware at the edge?"

To address this question, we highlight how meta-learning primarily relies on performing efficient feature extraction, allowing for the use of simple training and evaluation methods. We also show that temporal convolutional networks (TCNs) can be used as excellent feature extractors for temporal data. Building on this, we propose a TCN accelerator that fully removes the need for on-device gradient computation for few-shot learning, keeping the cost of learning good features where resources are abundant, i.e. in servers and cloud environments. The proposed approach also generalizes to continual learning for shifting data distributions, closing the loop back to robust edge intelligence. The main contributions of this work are:

- a first *in-silico* TCN[1] accelerator implementation,
- a first accelerator for few-shot learning, which performs on-chip temporal data classification via meta-learning and continual learning.

Guiding the reader towards these points, the thesis is structured as follows. First, Section 2 introduces relevant background information for understanding the developments at the core of this work. Section 3 then compares multiple architectures for sequence classification tasks, among which one is selected for hardware implementation. Next, Section 4 performs an initially a qualitative, then quantitative, analysis on meta-learning algorithms, from which the meta-learning algorithm and quantization approach for hardware implementation are chosen. The silicon implementation, which is based on these three design decisions, is then covered in Section 5. Finally, Section 6 presents the conclusion of the thesis by looking back and answering the main research question. An outlook for key applications and future work is also provided.

[1] A TCN as defined by Bai *et al.*, using exponentially increasing dilation and residual layers. 1D residual networks, also a type of TCNs, are excluded as they cannot be processed in a streaming fashion.

2 Background

In this section, the required background information for the claims made in this thesis will be presented. First, in Section 2.1, meta-learning and its suitability for edge computing will be discussed, after which in Section 2.2 the neural networks used in this work will be introduced. Finally, Section 2.3 covers the fundamentals of network quantization, which is necessary for deploying deep neural networks on edge devices.

2.1 Meta-learning

Meta-learning forms the core of the developments in this thesis. Therefore, a broad overview of the matter will be provided in this section. First, in Section 2.1.1 the need and applicability of meta-learning will be discussed. After that, in Section 2.1.2 a mathematical formalization of meta-learning is provided to aid the understanding of various meta-learning methods. Following this, Section 2.1.3 presents an overview of the meta-learning landscape, from which a subset of the methods will be covered in Section 2.1.4. This subset will then be qualitatively compared in Section 2.1.5. Three meta-learning benchmarks will then be discussed in Section 2.1.6. Finally, in Section 2.1.7, continual learning and its synergies with meta-learning will be explored.

2.1.1 The need for data-efficient learning

The potency of modern machine learning models, as exemplified by the likes of AlphaGo [9], GPT-4 [10] and Llama 2 [11], emanates from the processing of vast volumes of (simulated) data. For example, the training set for Llama2 consisted of two trillion (10^{12}) tokens [11]. Yet, environments where data quantity and computational resources are constrained are excluded from the successes achieved by such models: a challenge therefore emerges at the intersection of edge computing and data-efficient learning. In the space where conventional training paradigms find themselves tested, this challenge gives rise to the need for data-efficient and low-compute learning algorithms, which have the following advantages:

- data privacy is increased through keeping the user data on-chip [8],
- long-term robustness is improved as changes in tasks over time can be dealt with locally and in real-time,
- electronic waste is reduced due to a longer device life cycle,
- maintenance costs will be lower by reducing reliance on over-the-air updates.

One possible pathway to such ideal algorithms is through the application of *meta-learning*, which is a machine learning approach where models learn how to learn, enabling them to adapt quickly to new tasks with limited data. Using meta-learning avoids the need for large-scale data collection at the edge before a model can be extended with new capabilities: instead of requesting a user to repeat a set of gestures over several epochs before they can be classified, only a few examples are required for each new gesture.

Overall, meta-learning provides a promising pathway toward true edge intelligence, enabling more privacy-friendly systems and allowing for deployment in more unpredictable scenarios.

2.1.2 Formalizing meta-learning

In this section, a formal definition of meta-learning will be established, in order to have a standardized framework for the later evaluation of various meta-learning methods. Note that for the entirety of this section, the mathematical formalization from Hospedales *et al.* [12] is followed.

Starting off, first, conventional machine learning is formally introduced. For this, the following symbols are defined:

- θ: the parameters of the model. θ^* are the optimal (inducing the lowest loss) model parameters,
- f_θ: the predictive model with parameters θ,
- ω: the information about how to perform learning, also referred to as "meta-knowledge". This can be a network architecture, initial network parameters, an optimization algorithm such as gradient descent or parameters that guide the learning, such as the learning rate: meta-knowledge can therefore be seen as a set of hyperparameters,
- $D = \{(x_1, y_1), ..., (x_N, y_N)\}$: an N-sample dataset containing samples (x_i) with corresponding labels (y_i). D^{train} is a subset of D and represents the training dataset,
- L: the loss function, measuring the error between the ground-truth labels (y_i) and predicted ($\hat{y}_i$) outputs.

Using these symbols, supervised learning can be formalized as

$$\hat{y}_i = f_\theta(x_i), \tag{1}$$

$$D^{\text{train}} \subset D, \tag{2}$$

$$\theta^* = \arg\min_\theta L\left(D^{\text{train}}; \theta, \omega\right). \tag{3}$$

Note that Eqs. (1) to (3) can be represented by an optimization loop as shown in Fig. 1, since Eq. (3) cannot always be solved for explicitly and therefore requires an approximation by iteration. There are two important assumptions in this case [12]:

- ω is fixed or chosen by the user before training,
- f_θ is trained from scratch for every new task/dataset.

Consequently, specifying different values for ω can have a drastic impact on the performance of the final learned model. However, when the search space is very large, manually finding the optimal ω can be impractical. For example, when ω represents all possible optimization hyperparameter values, it is not feasible to find the optimal set of values by hand, which is a well-known issue for large machine learning workloads [13].

The key idea of meta-learning is to view conventional machine learning as a process that can itself be optimized. Indeed, meta-learning is often referred to as "learning to learn" (a term first coined by Thrun *et al.* in [14]), indicating that unlike conventional machine learning methods, where a model is learned, it is learned how to optimally learn a model.

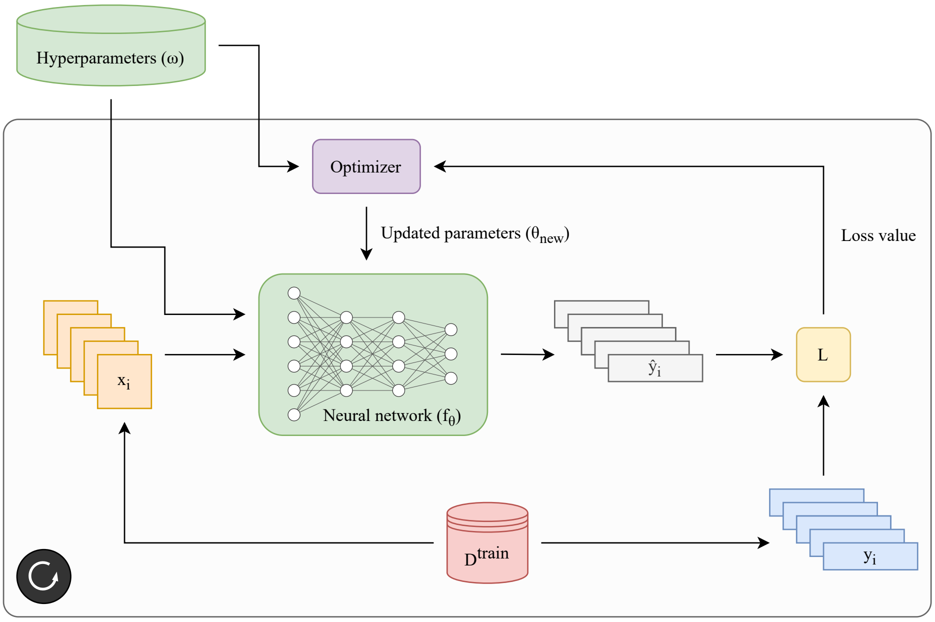


Figure 1: Overview of the conventional supervised learning pipeline. Hyperparameters ω can influence both the optimization and the neural network as per the definition. The left-bottom circular arrow indicates that the steps in this figure, starting from the training dataset D^{train} and ending with the updated parameters, θ, are repeated multiple times until $\theta \approx \theta^*$.

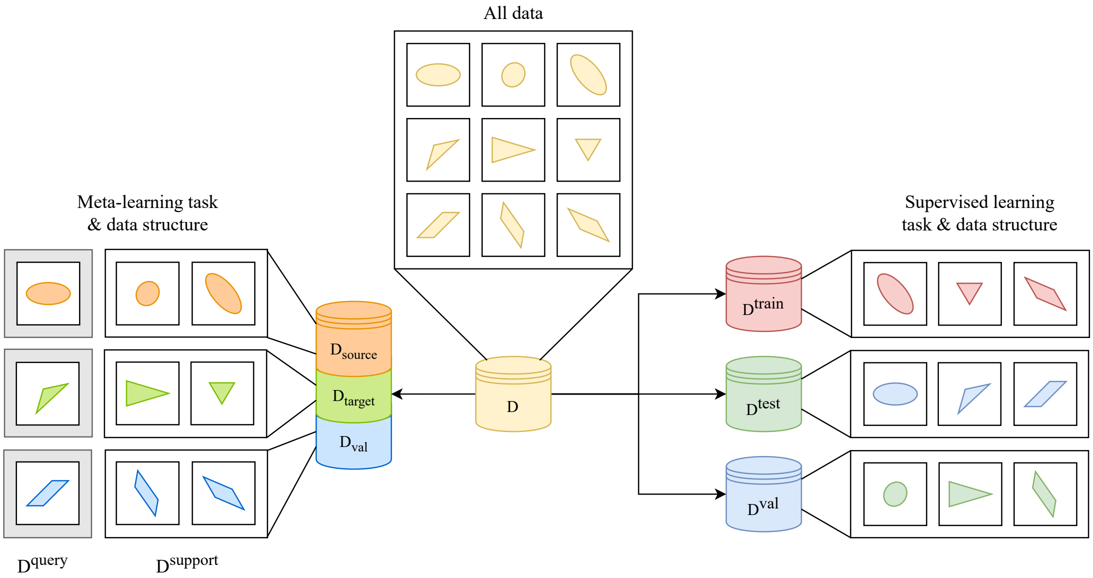


Figure 2: Splitting of dataset D for meta-learning (on the left) into D_{source}, D_{target}, D_{val} and for conventional supervised learning (on the right) into D^{train}, D^{test} and D^{val}. Note how in the supervised setting, every split has the same classes, while in the meta-learning scenario, every split has unique classes.

This formalization indicates that meta-learning can be viewed as a bilevel optimization problem [12]: there is an *inner loop*, where a model is learned (equivalent to supervised learning) and an *outer loop*, where this learning process is itself learned.

Therefore, in contrast to supervised learning, with meta-learning, ω is **learned** and not assumed fixed. This means that an optimal value is found by a learning algorithm, just like this is the case for θ. Furthermore, instead of being trained from scratch for every new task, f_θ is now meta-trained through a set of M tasks, with a *source* dataset D_{source} defined as

$$\begin{aligned} D_{\text{source}} &= \left\{ \left(D_{\text{source}}^{\text{support}}, D_{\text{source}}^{\text{query}} \right)^{(i)} \right\}_{i=1}^{M}, \\ D_{\text{source}} &\subset D. \end{aligned} \tag{4}$$

This source dataset represents all the data that is available for meta-training before deployment. It consists of a *support* set and a *query* set, which can be assimilated to the training (D^{train}) and test (D^{test}) sets in conventional machine learning, respectively. This structure is displayed in Fig. 2, where also the validation sets (D^{val}) are shown for both meta-learning and supervised learning.

Formalizing the extra level of optimization/learning for ω [12] through this set of tasks, which is referred to as "meta-training", gives

$$\omega^{*} = \arg\min_{\omega} \sum_{i=1}^{M} L^{\text{meta}} \left(D_{\text{source}}^{\text{query}(i)}; \theta^{*(i)}(\omega), \omega \right), \tag{5}$$

$$\text{s.t.} \quad \theta^{*(i)}(\omega) = \arg\min_{\theta} L^{\text{task}} \left(D_{\text{source}}^{\text{support}(i)}; \theta, \omega \right), \tag{6}$$

where L^{meta} and L^{task} represent the objectives in the outer and inner loops, respectively and ω^{*} is the set of optimal hyperparameters. They can differ but do not have to: for example, when dealing with a set of regression tasks, both objectives are equal to the Euclidean distance function.

Graphically, the full nested optimization is shown in Fig. 3. Note that the inner loop of this figure is exactly the same as in Fig. 1, where $D^{\text{support}} = D^{\text{train}}$ and $L^{\text{task}} = L$. Putting Eq. (3) and Eq. (6) side by side, it can indeed be seen that regular machine learning reduces to purely the inner loop. Also, similar to the assumptions for conventional machine learning, parameters ω are fixed in the inner loop during meta-training.

After completing the meta-training phase, the learned meta-knowledge ω is used for training the base model for a new target task i, taken from a set of Q tasks in a target dataset D_{target} (also see Fig. 2):

$$\begin{aligned} D_{\text{target}} &= \left\{ \left(D_{\text{target}}^{\text{support}}, D_{\text{target}}^{\text{query}} \right)^{(i)} \right\}_{i=1}^{Q}, \\ D_{\text{target}} &\subset D, \\ D_{\text{source}} \cap D_{\text{target}} &= \emptyset. \end{aligned} \tag{7}$$

The target support set $D_{\text{target}}^{\text{support}}$ represents the example data that the user provides to the model for learning a new task, while the target query set $D_{\text{target}}^{\text{query}}$ is the data used for inference after the examples have been shown. Training with each of the Q target tasks is called "meta-testing":

$$\theta^{*(i)} = \arg\min_{\theta} L^{task} \left(D_{\text{target}}^{\text{support}(i)}; \theta; \omega^{*} \right), \tag{8}$$

$$\hat{y}_i = f_{\theta^{*(i)}} \left(x_{\text{source}}^{\text{query}(i)} \right). \tag{9}$$

Meta-testing is exactly the same as the inner loop of meta-training and consequently the same as supervised learning (see Fig. 1). However, comparing Eq. (8) to Eq. (3), it can be seen

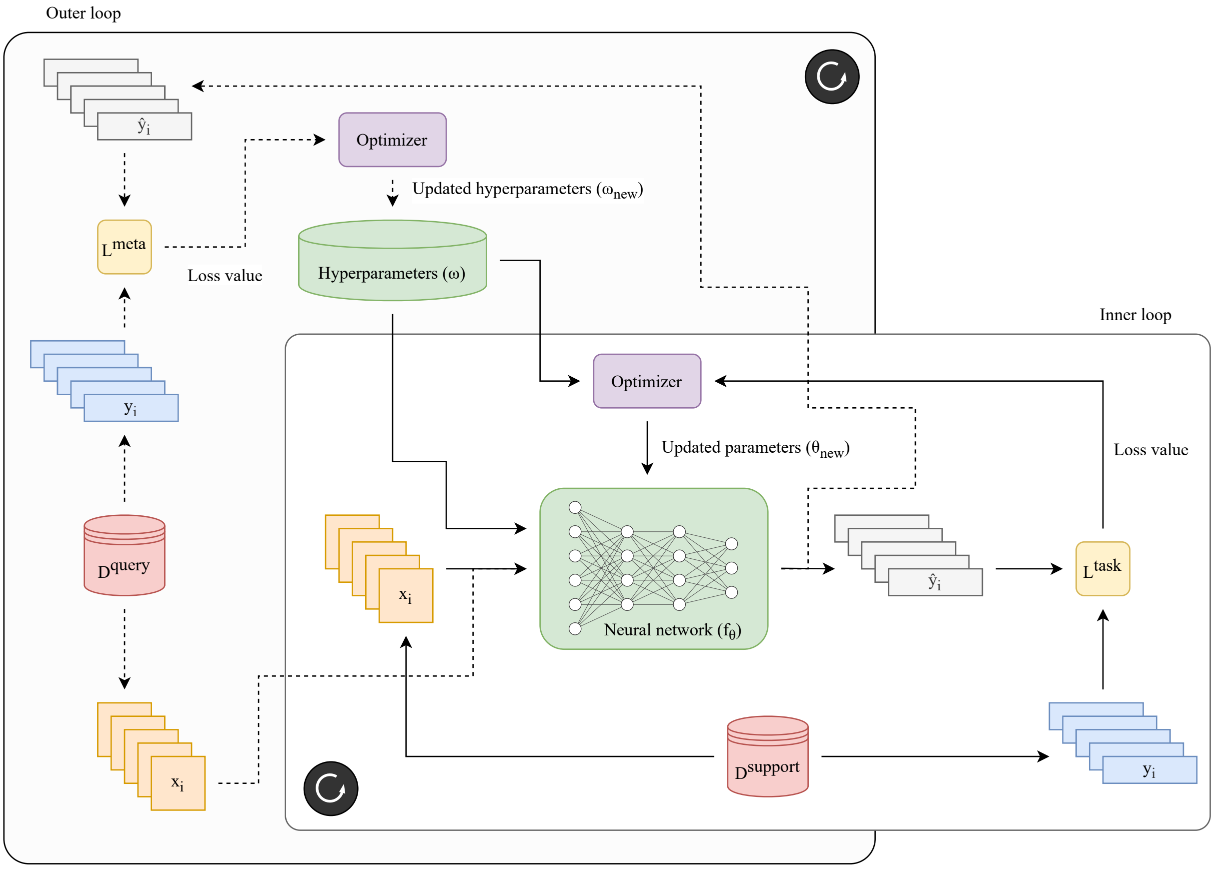


Figure 3: Graphical depiction of the bilevel optimization scheme of meta-learning. Dashed arrows represent information flow in the outer loop, whereas continuous lines represent information flow in the inner loop.

that the learning on the support set now benefits, after meta-learning, from the learned meta-knowledge about the algorithm to use, which is handpicked in the conventional case rather than part of an optimization problem.

Considering the full formalization, it can now be put into the context of learning at the edge. Firstly, the process of meta-training always occurs off-chip, typically carried out in environments such as the cloud or a local GPU cluster. Conversely, the phase of meta-testing is exclusively executed on-chip. This means that the design or complexity of the outer loop (Eq. (5)) does not influence the design or operation of the edge device that should perform learning. Consequently, an edge device that supports a certain inner loop for meta-testing automatically supports the deployment of any meta-learning method using that inner loop.

2.1.3 Taxonomy

With the formalization of meta-learning outlined, the different types of meta-learning methods can now be considered. However, due to the large variety in approaches [12], first, a meta-learning taxonomy is defined, from which a subset is selected for further study in this thesis.

In this section, the taxonomy as introduced by Hospedales *et al.* [12] is used. This taxonomy contains the following three axes (see Fig. 4):

- Meta-Optimizer: defining **how** to learn. This indicates the choice of optimizer/learning algorithm for outer-loop optimization in meta-training (see Eq. (5)),

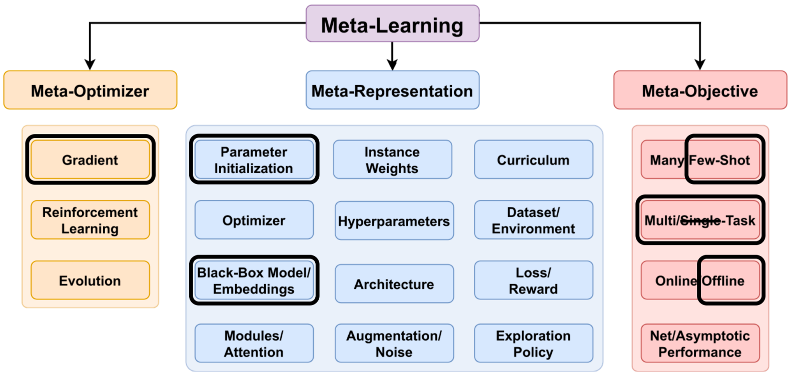


Figure 4: The meta-learning landscape, following the taxonomy proposed by [12]. Highlighted boxes indicate what is investigated in the thesis. Image adapted from [12].

- Meta-Representation: defining **what** to learn. This relates to the contents of ω,

- Meta-Objective: defining **why** to learn. This relates to the goal of the meta-learning setup, which is set by choosing L_{meta} (see Eq. (5)), the task distribution and the interactions between both optimization levels [12].

Considering the meta-training formalization of Eqs. (5) to (6), the mentioned axes can be color-coded as follows:

$$\omega^* = \arg\min_{\omega} \sum_{i=1}^{M} L^{\text{meta}} \left(D_{\text{source}}^{\text{query}(i)}; \theta^{*(i)}(\omega), \omega \right), \tag{10}$$

$$\text{s.t.} \quad \theta^{*(i)}(\omega) = \arg\min_{\theta} L^{\text{task}} \left(D_{\text{source}}^{\text{support}(i)}; \theta, \omega \right). \tag{11}$$

For each axis, highlighted boxes in Fig. 4 indicate the selection of the meta-learning landscape for this thesis. This subset is chosen to fit the use case of this work: few-shot sequential data classification at the edge. *Few-shot* indicates here that only a low number of samples is available for training, which is often the case at the edge. Meta-training is then used to train a model for such few-shot classification tasks. Generally, few-shot learning can be seen as a subset of meta-learning: a model learns to classify previously unseen inputs using k labeled examples (*shots*). The number of classes that the model learns for a new task is referred to as *ways*. A visual overview of this can be found in Fig. 5. Based on this use case, the following explanation outlines the selected subset for each axis and the rationale behind these choices.

- Starting with the meta-optimizer axis, only methods that use *gradient descent* for optimization are considered. Reinforcement learning is often used when f_θ or L_{meta} is non-differentiable [12], while evolutionary algorithms only work well for smaller models [12]. Since, in this thesis, fully differentiable neural networks for classification will be used, containing a minimum of 10k parameters, gradient-based methods are best suited for this situation.

- For the meta-representation axis, only *parameter initialization* and *black-box model/ embeddings* are considered. This choice is made as the other meta-representations do not have a straightforward hardware implementation (see Section 4.1).

- Finally, in this work, there is only a single meta-objective as per the use case. The three marked options together form this objective: few-shot multi-task offline learning. A *few-shot* learning setup is chosen, as it is assumed that a low number of examples/shots (1-20) will be available at the edge. The reason for the objective being *multi-task*, is that after meta-training, the learner is expected to solve any task taken from the D_{target} set instead of simply solving one specific task. More specifically, the model is trained to learn different classes for every task as shown in Fig. 5, where for task i, the classes *horse*, *dog* and *worm* are learned, while in task $i+1$ the classes *piano*, *polar bear* and *lion* are learned. Lastly, *offline* indicates that during meta-testing, only inner-loop updates are performed, while in the *online* case, outer-loop updates can be performed during meta-testing, which does not correspond to the selected edge scenario.

Since only the meta-representation axis is not yet fixed by the research question of this thesis, this axis will be the main focus of the next section.

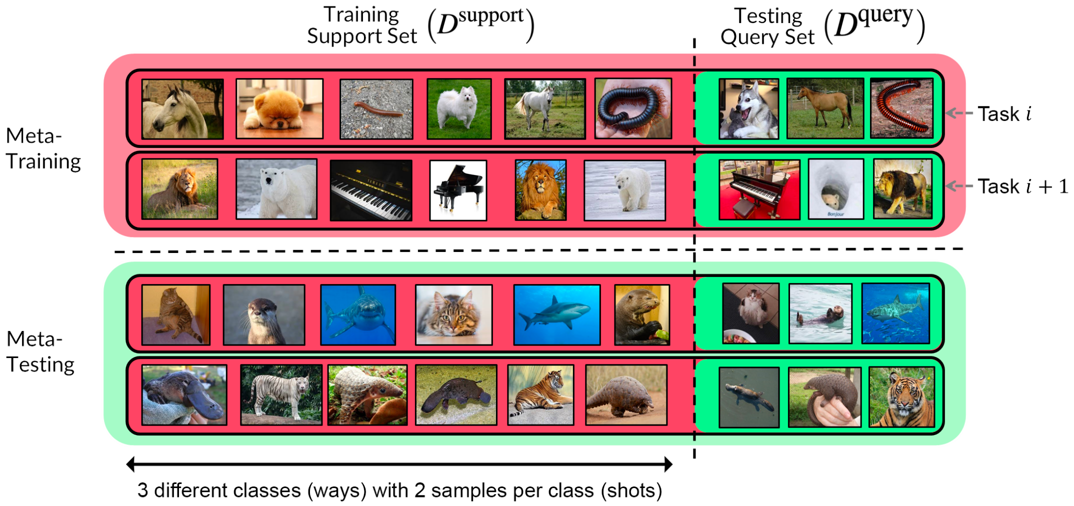

Training Support Set (D^{support})
Testing Query Set (D^{query})
Meta-Training
Meta-Testing
Task i
Task $i+1$
3 different classes (ways) with 2 samples per class (shots)

Figure 5: Example of meta-training and meta-testing with a 3-way 2-shot task.[2]

2.1.4 Meta-learning algorithms

Using the formalization and categorization described, this section will cover different meta-learning algorithms, grouped by meta-representation (ω).

Note that while the black-box model and embeddings are combined to form one category in the taxonomy of Fig. 4, in this work, they are treated separately as they vary significantly at a high level. However, it is possible to formulate all embedding methods as special forms of black-box models [12], hence why they are shown combined.

2.1.4.1 Learning an initialization In the first category, ω represents the initial values of the neural network parameters for the inner optimization step (see Eq. (6) and Eq. (8)). Model-agnostic meta-learning (MAML) [15] is one of the most popular algorithms in this space [12]. The goal of MAML is to learn a good network initialization so that only a few steps of gradient descent are required to update the network for a new task, instead of requiring many iterations.

[2] Image adapted from `https://iclr-blog-track.github.io/2022/03/25/understanding_mtr_meta/`.

Within the context of edge computing, this translates to the inner-loop edge device adapting with only a few iterations. In Algorithm 1, the steps of MAML are displayed. The **foreach** loop represents the inner optimization loop as per Eq. (6), while line 12 represents the outer optimization loop as per Eq. (5). Note that performance can be improved by performing multiple gradient descent steps in the inner loop (i.e. $k > 1$). This inner-outer optimization is repeated until convergence or after a fixed number of repetitions. After convergence, meta-testing is performed to calculate the final performance.

Algorithm 1: Pseudocode for meta-training via model-agnostic meta-learning (MAML) where g indicates the gradient. Based on [16].

```
   Data: step size hyperparameters $\alpha, \beta$; number of inner loop gradient descent steps $k$
1  initialize $\omega$ with any strategy;
2  repeat
3      foreach $D_{source}^{(i)} \in D_{source}$ do
4          initialize $\theta_i$ with $\omega$;
5          take $\{D_{\text{source}}^{\text{support}(i)}, D_{\text{source}}^{\text{query}(i)}\} = D_{\text{source}}^{(i)}$;
6          for $i \leftarrow 0$ to $k$ do
7              evaluate $g = \nabla_{\theta_i} L^{task}(D_{\text{source}}^{\text{support}(i)}, \theta_i)$;
8              update $\theta_i \leftarrow \theta_i + \alpha g$;
9          end
10         evaluate test loss $L_i = L^{task}(D_{\text{source}}^{\text{query}(i)}, \theta_i)$;
11     end
12     update $\omega \leftarrow \omega - \beta \sum_{D_{\text{source}}^{(i)} \in D_{\text{source}}} \nabla_\omega L_i$;
13 until convergence;
```

One of the downsides of MAML is that it requires computing second-order gradients for all parameters in the network. This can be seen by substituting lines 6-10 (assuming $k = 1$) into line 12 in Algorithm 1, after which the ∇ operator appears twice (see Eq. (12)). This makes training, especially for larger networks, rather slow. To resolve this, the authors propose a first-order approximation of MAML (FOMAML) [15], where the red part in Eq. (12) is treated as a constant [17]. This results in almost equivalent performance compared to regular MAML.

$$\omega \leftarrow \omega - \beta \sum_{D_{\text{source}}^{(i)} \in D_{\text{source}}} \nabla_\omega L^{task} \left(D_{\text{source}}^{\text{query}(i)}, \theta_i + \alpha \nabla_{\theta_i} L^{task} \left(D_{\text{source}}^{\text{support}(i)}, \theta_i \right) \right) \tag{12}$$

Due to its flexibility, MAML has inspired many derivative works. MAML++ [18] extends MAML with a set of changes to the meta-training procedure to stabilize training. Reptile [17] builds on top of the first-order MAML approximation by repeatedly sampling tasks and updating ω with a gradient equal to the difference in weights before and after each task (see Algorithm 2).

Sign-MAML [19] also uses the first-order approximation method, but only considers the sign of the gradient. Orthogonally, MetaSGD [20] uses a vector $\boldsymbol{a}$ instead of a scalar for α in Algorithm 1 and makes it learnable by updating it after line 12 via $\boldsymbol{a} \leftarrow \boldsymbol{a} - \beta \sum_{D_{\text{source}}^{(i)} \in D_{\text{source}}} \nabla_{\boldsymbol{a}} L_i$. MetaCurvature [21] is similar to MetaSGD, but uses a matrix M instead of a vector $\boldsymbol{a}$. Overall, while all of these methods improve the baseline in data efficiency or generalization performance, MAML remains a simple yet elegant initialization baseline for meta-learning [18].

Algorithm 2: Pseudocode for Reptile. Based on [17].

```
Data: step size hyperparameter $\alpha$; number of inner loop gradient descent steps $k$
initialize $\omega$ with any strategy;
foreach $D^{(i)}_{source} \in D_{source}$ do
    initialize $\theta_i$ with $\omega$;
    take $\{D^{\text{support}(i)}_{\text{source}}, D^{\text{query}(i)}_{\text{source}}\} = D^{(i)}_{\text{source}}$;
    for $i \leftarrow 0$ to $k$ do
        evaluate $g = \nabla_{\theta_i} L^{task}(D^{\text{support}(i)}_{\text{source}}, \theta_i)$;
        update $\theta_i \leftarrow \theta_i + \alpha g$;
    end
    update $\omega \leftarrow \omega + \alpha(\theta_i - \omega)$;
end
```

2.1.4.2 Feed-forward models In feed-forward or black-box meta-learning approaches, ω represents the parameters of a model that provides a direct feed-forward mapping from the support samples (D^{support}) to parameters required to perform predictions on query samples (D^{query}) [12]. Therefore, in contrast to initialization-based methods, feed-forward meta-learning methods do not require gradient updates to solve Eq. (8).

One well-known feed-forward method is the simple neural attentive meta-learner (SNAIL) proposed by Mishra *et al.* [22]. SNAIL effectively converts meta-learning to a sequence-to-sequence task, as shown in Fig. 6. In written language, a prompt for such a task would be: *"5 times 3 is 15; 6 times 3 is 18; 4 times 3 is 12; 8 times 3 is __"*. The model then, in an autoregressive fashion, predicts after three labeled examples (supports) the outcome of the last (queried) sample.

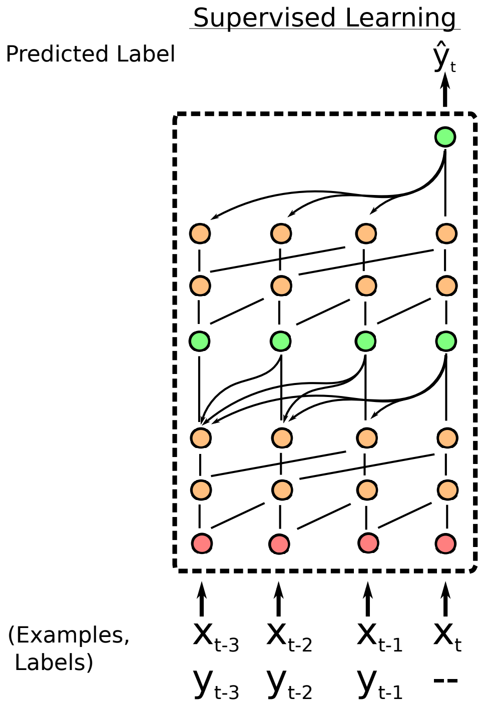

Supervised Learning
Predicted Label
$\hat{y}_t$
(Examples, Labels)
x_{t-3} x_{t-2} x_{t-1} x_t
y_{t-3} y_{t-2} y_{t-1} --

Figure 6: Overview of SNAIL. (x_{t-3}, y_{t-3}) to (x_{t-1}, y_{t-1}) represent the support samples D^{support} while (x_t, y_t) represents a single query sample from D^{query}. Figure taken from [22].

Another idea in feed-forward approaches is a concept called *hypernetworks* [23], where a network generates the parameters of another network. In terms of formalization, a hypernetwork with parameters ω generates parameters θ for a main network based on D^{support}.

The main network operates similarly to a typical neural network, as it is responsible for mapping raw inputs to their intended targets. Conversely, the hypernetwork receives inputs that encapsulate weight structure information and produces the corresponding weights for a given layer.

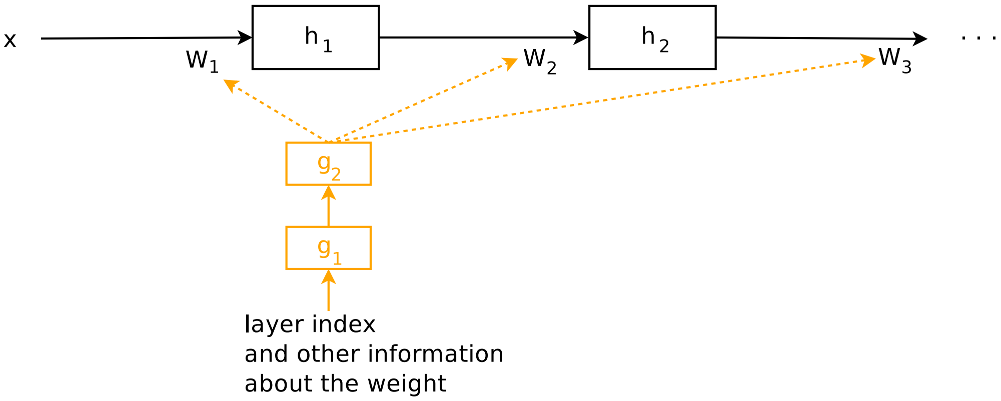


Figure 7: Visualization of a hypernetwork. Black connections and parameters are part of the main network, whereas orange connections and parameters are part of the hypernetwork; g_1, g_2 and h_1, h_2 are the activations from the hidden layers of the hypernetwork and the main network, respectively. The main network weights W_1, W_2, etc. are subsequently generated from the hypernetwork. Taken from [23].

Recently, Zhao *et al.* applied this technique to few-shot learning [24], achieving similar performance to Meta-SGD [20]. A similar approach is proposed by Qiao *et al.* in [25], where the final-layer parameters of the model are predicted from the activations of the support data (D^{support}).

Orthogonally, Rusu *et al.* propose *latent embedding optimization* (LEO) in [26], which effectively combines MAML with a feed-forward model in the inner loop. In this approach, a latent code z that has lower dimensionality than θ is conditioned on D^{support} through parameters ω. This vector z is then mapped to parameters θ which perform the classification. This process is visualized in Fig. 8. The main advantage of this approach is that MAML now does not optimize in the high-dimensional space of θ, but in the low-dimensional space of z [26].

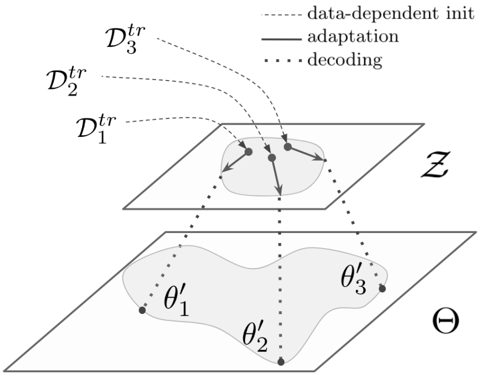


Figure 8: Visualization of the key idea from LEO, where optimization now takes space in the lower dimensional space Z instead of Θ. Taken from [26].

2.1.4.3 Metric learning In embedding methods or metric learning, ω represents the parameters of an embedding network, generating M-dimensional vector representations from input samples (see Fig. 9). Using these embeddings, the classification of unseen samples can be performed without any gradient updates, similar to feed-forward methods. As the exact way classification is performed varies per algorithm, a detailed workout of metric learning following the formalization of meta-learning is provided in Appendix A.

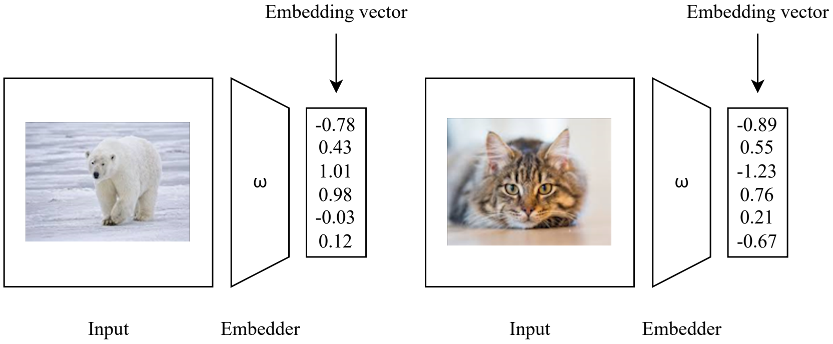


Figure 9: Example of generating an $M = 6$-dimensional embedding from an input image. Notice how the dimensionality of the input is severely reduced, going from a full-color image to only six scalars.

Prototypical networks are one of the most popular metric learning algorithms, proposed by Snell *et al.* in [27]. In this method, the M-dimensional embeddings for each sample x_i of class k in a set of labeled examples S_k are averaged into a "prototype" c_k [27]:

$$c_k = \frac{1}{|S_k|} \sum_{(x_i,y_i)\in S_k} f_\omega(x_i). \tag{13}$$

Illustrated in Fig. 10, is that classification then takes place by comparing the embedding from the query sample with the prototypes of all support classes through a distance function d: $\mathbb{R}^M \times \mathbb{R}^M \to [0,+\infty)$ [27]. The class of the prototype with the lowest distance to the query embedding is assigned to the query. Since any distance metric d works in this setting, two popular distance metrics are compared: squared Euclidean distance and cosine distance. The authors then empirically find that squared Euclidean distance gives better few-shot classification accuracy.

As opposed to prototypical networks, which are trained with a set of tasks using the bi-level meta-learning structure, the *Baseline++* method from Chen *et al.* [28] simply pre-trains an embedder using normal supervised learning on all data in D_{source}. Evaluation is then performed using cosine distance between support and query embeddings, similar to prototypical networks.

Note that metric learning methods do not have to rely on a distance algorithm by definition. For example, *matching networks* [29] use an attention mechanism over the support embeddings combined with the query embeddings to predict the class of the query sample (see Fig. 11) while, in *relation networks* [30], a "relation module" that produces a similarity score between support and query samples is employed (see Fig. 12). However, even with such constructions, metric learning methods suffer from one downside compared to initialization-based and feed-forward-based methods: they only work for classification problems [1]. For example, during classification, the output embedding is used to compare with other embeddings. This is due to the fact that embeddings do not mean anything for the continuous input and output space of regression tasks.

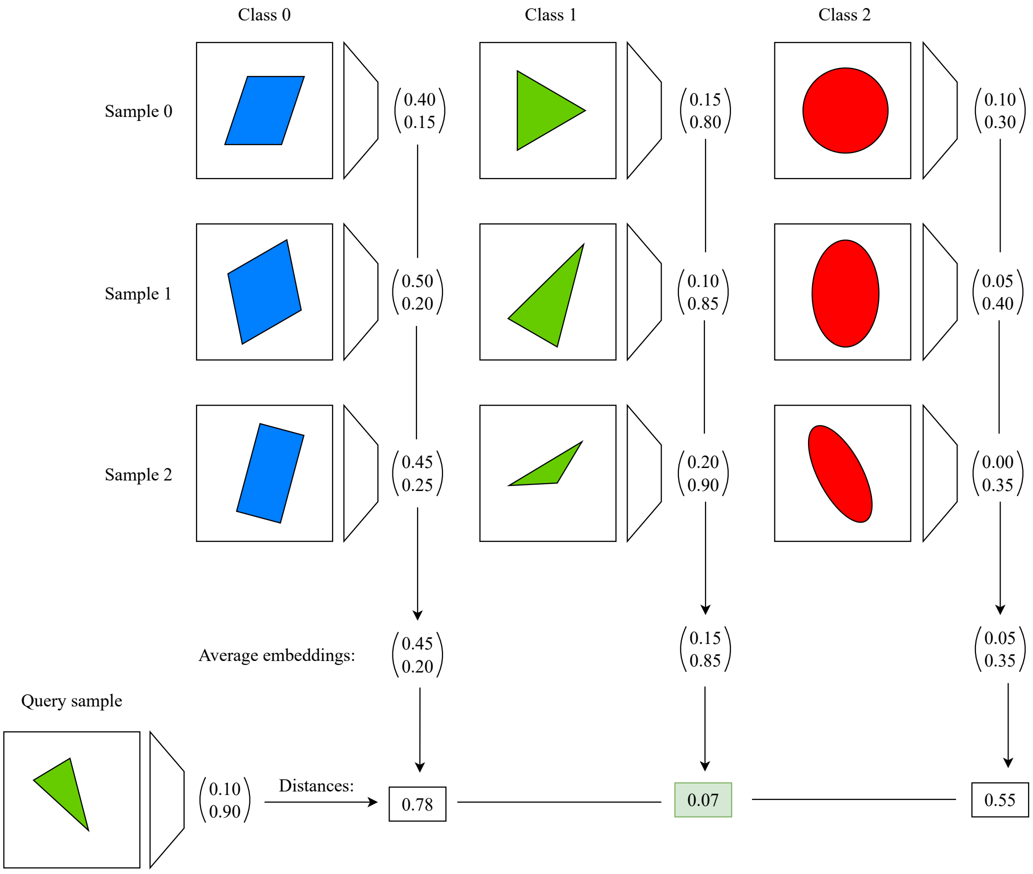


Figure 10: Example evaluation of a prototypical network in a 3-way 3-shot scenario with Euclidean distance as a loss function. The class of the prototype with the lowest distance is assigned to the incoming query sample. The trapezia represent the embedder.

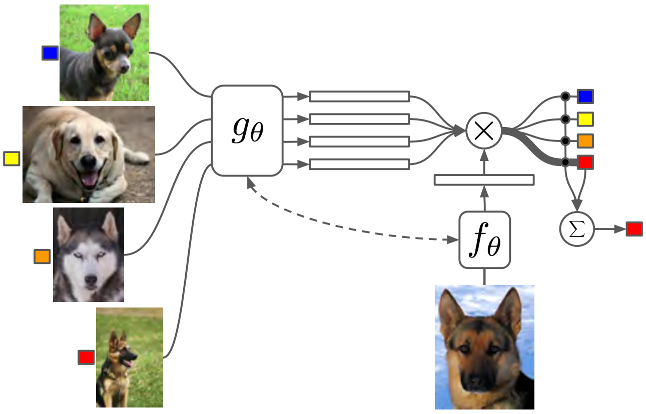


Figure 11: Matching network architecture for a 4-way 1-shot problem with one query sample, where g_θ and f_θ are two (potentially identical) embedders and × is an attention mechanism. Taken from [29].

2.1.5 Qualitative algorithm comparison

Having covered a variety of meta-learning methods, a qualitative comparison is displayed in Table 1. It lists all the discussed methods, ordered by appearance in the text and line-separated by meta-representation. Per method, the key idea(s), the on-chip friendliness, as well as the key advantages and disadvantages are listed.

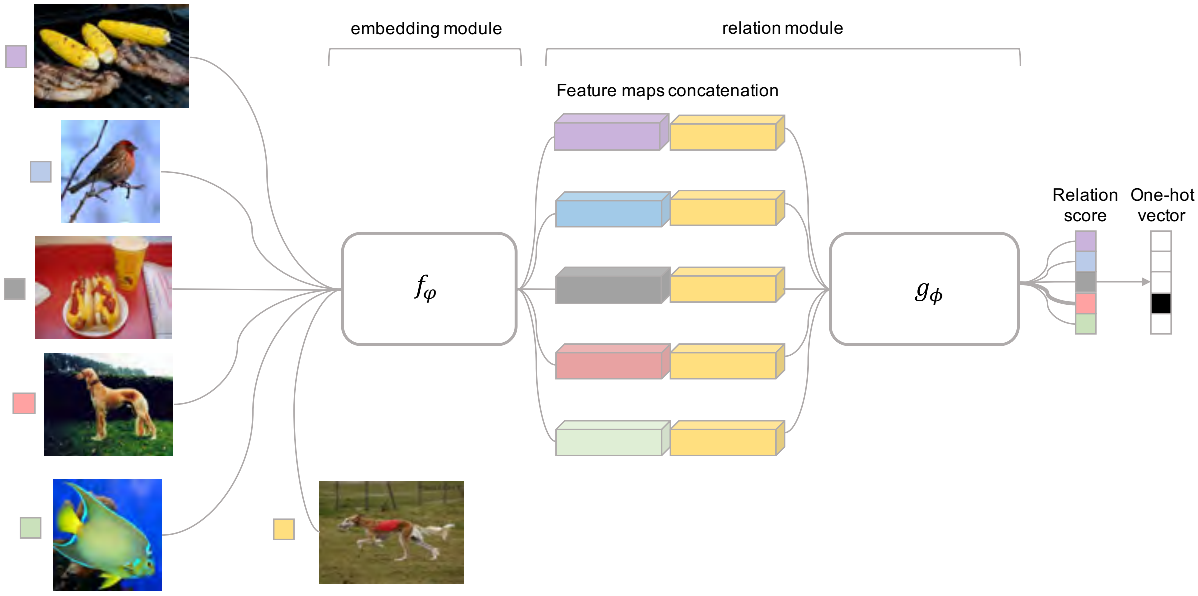


Figure 12: Relation network architecture for a 5-way 1-shot problem with one query sample, where f_ϕ is any embedder and g_ϕ is the relation network. Taken from [30].

For brevity reasons, on-chip friendliness is a visual score based loosely on three criteria: meta-testing complexity (or the control complexity thereof), total required computation steps and memory usage. The first aspect mainly affects design complexity and testability, while the second impacts the real-time constraints as well as power usage. The third aspect not only concerns power usage but also the total silicon area and thus the overall cost.

Among the surveyed methods, the need for gradient descent during meta-testing is considered one of the largest disadvantages as it negatively affects all three aspects: it significantly increases complexity, requires more computation steps than pure inference and needs larger memories to store the intermediate activations for gradient calculation. Next to this, extra modules besides the main network during meta-testing are also considered a disadvantage as they directly imply extra hardware overhead.

The two highest-scoring methods in this on-chip friendliness score are prototypical networks [27] and the Baseline++ method from [28]. Both of them then do not require on-chip gradient descent and only compare embeddings during meta-testing. This means that, on top of the network, extra on-chip storage is only required for the averaged embeddings. However, neither of them is state-of-art in terms of few-shot classification accuracy compared to some of the more complex methods such as MAML++ [18] or LEO [26].

Hence, in order to accelerate the adoption of meta-learning for edge devices, novel yet simple methods (i.e. gradient-free) are required that offer similar or better few-shot classification accuracy than the more complex methods.

2.1.6 Benchmarks

Beyond qualitative comparisons, in order to quantitatively compare the performance of different meta-learning methods, standardized benchmarks are required. This section covers two such benchmarks: Section 2.1.6.1 reviews the Omniglot dataset [31] while Section 2.1.6.2 discusses (few-shot) keyword spotting. Both of these will be used as guiding tasks throughout this work,

Table 1: Qualitative comparison between the introduced meta-learning methods. In the on-chip friendly?-column, ◐ ○ ○ ○ indicates very low, while ● ● ● ◐ indicates very high.

Name	Key idea(s)	On-chip friendly?	Advantages	Disadvantages
MAML [15]	Model-agnostic meta-learning	● ● ○ ○	Simple	Requires gradient descent
MAML++ [18]	= MAML + improved training procedure	● ○ ○ ○	Better performance than baseline MAML	Complex setup for training and inference
Reptile [17]	= gradient descent on each task *approx* first-order MAML	● ● ○ ○	Very simple, fast training	Requires gradient descent
Sign-MAML [19]	= first-order MAML but only using the sign of the gradient	● ● ○ ○	Better performance than first-order MAML	Requires gradient descent
MetaSGD [20]	= MAML but inner loop learning rate is a vector	◐ ○ ○ ○	Better all-round performance than MAML	Parameter count doubles
MetaCurvature [21]	= MAML but inner loop learning rate is a matrix	◐ ○ ○ ○	Better few-shot performance than MetaSGD	Parameter count increases, complex inner loop
SNAIL [22]	Attention over convolved support sequence for few-shot tasks	● ● ● ○	Gradient-free	Contains multiple attention layers
Meta-learning via Hypernetworks [24]	Generating task-specific network weights	◐ ○ ○ ○	Better performance than baseline MAML	More parameters, gradient calculation + hypernetwork forward-pass
Predict parameters by activations [25]	= hypernetwork to generate classification layer	● ● ● ○	Only one matrix multiplication to compute weights	Increased parameter count through hypernetwork
LEO [26]	= MAML + ω is a hypernetwork	◐ ○ ○ ○	Very high few-shot performance	Gradient calculation + decoder forward-pass, more parameters
Prototypical Networks [27]	Computing class prototypes for few-shot tasks	● ● ● ◐	Simple, gradient- free, fast training	Harder to deal with out-of-domain tasks, not state-of-art accuracy
Baseline++ [28]	Supervised embedder pre-training for few-shot tasks	● ● ● ◐	Extremely simple, gradient-free extremely fast training	Difficulty with out-of-domain tasks, more training samples required
Matching Networks [29]	Attention layer over support embeddings for few-shot tasks	● ● ● ○	Gradient-free	Requires attention calculation
Relation Networks [30]	Learning relationships for few-shot learning	● ● ● ○	Gradient-free	Increased parameter count through relation module

not only for the software but also for the hardware design.

This means that the most widely adopted dataset for comparing meta-learning methods, *mini*ImageNet [29] (discussed in Appendix B), is not used in this thesis. The reason for this is that this work focuses on few-shot **sequence** classification: while converting a color image to a sequence is definitely possible, much semantic information is lost when doing so.

2.1.6.1 Omniglot The Omniglot dataset [31] consists of handwritten characters from a diverse range of writing systems and was introduced in 2015 by Lake *et al.* to address the challenge of one-shot learning.

The dataset contains characters from 50 alphabets, including but not limited to, Armenian, Hebrew, Cyrillic, Korean, Japanese and Braille. It comprises a collection of 1,623 different characters, each handwritten by 20 different people, resulting in a total of 32,460 distinct character images. Fig. 13 displays a subset of alphabets and characters contained in the dataset.

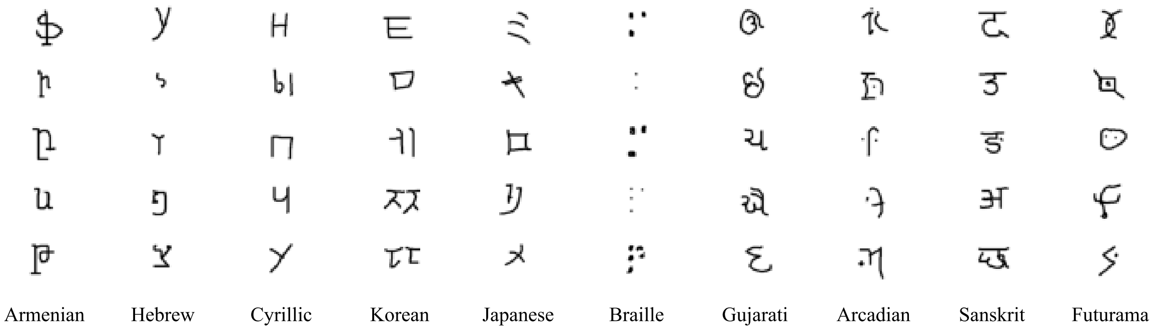


Figure 13: A selection of characters for various alphabets in the training split of the Omniglot dataset.

Each character image is grayscale and 105 by 105 pixels in size. The images can be resized to, e.g. to 28 × 28 pixels as in the MNIST dataset [29] [27], so that smaller neural networks can be used. However, unlike the MNIST dataset of handwritten digits [32], the Omniglot dataset also contains the stroke data for all characters, which was released in 2019 [33].

As this work focuses on streaming applications at the edge, inputting static images of characters into a model is not immediately representative of the desired application. However, due to its widespread adoption and low compute requirements, using Omniglot eases comparison between previous meta-learning approaches, hence it will be used in this work. Nevertheless, to emulate a streaming setting, the pixel rows of a single image can be concatenated from top to bottom to form a sequence (see Fig. 14). This procedure is taken from Le *et al.* [34], where it was used to convert the MNIST digit images [32] to sequences for temporal classification by a recurrent neural network (RNN).

The Omniglot dataset has two splits: a background and an evaluation split, containing 964 and 659 characters, respectively. Originally, the background set was meant for meta-training and the evaluation set for meta-testing [33]. However, Vinyals *et al.* created a new split with 1200 characters for meta-training in 2016 [29], which was subsequently used by key papers in the meta-learning domain [15] [27]. Another extension from the same authors was to create three new classes from each character by rotating it in multiples of 90 degrees, yielding 4800 classes in total.

Increasing the number of classes in the meta-training set makes the task at hand easier since more data is available for training while less data is available to test on. However, to provide a fair comparison to other methods that mainly use this split, the 1200-class rotation-extended split from Vinyals *et al.* is also employed in this work.

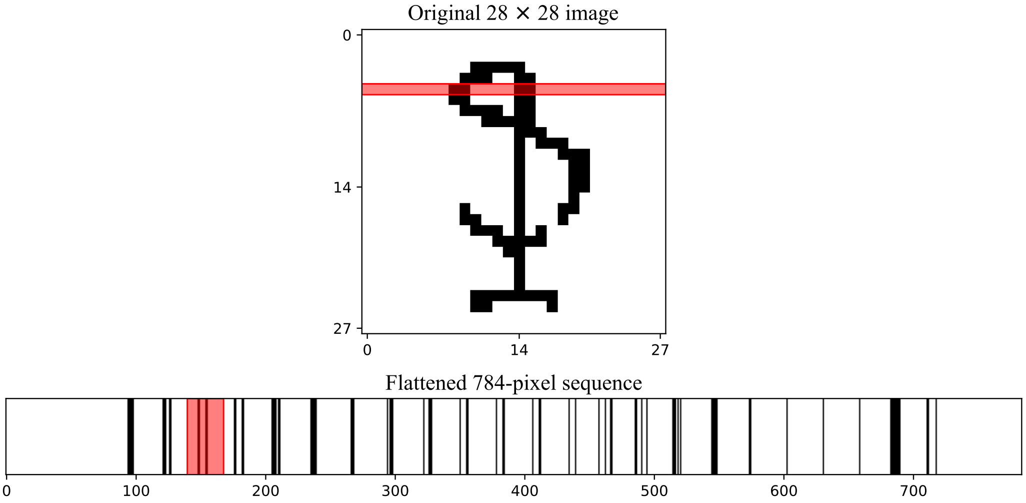


Figure 14: Sample conversion from an Omniglot image to a sequence, where the highlighted row in the 2D image is mapped to the highlighted slice in the flattened 1D sequence.

2.1.6.2 Few-shot keyword spotting While Omniglot and *mini*ImageNet are well-recognized datasets for few-shot learning, both are static image-only datasets at their core. As this work focuses on streaming applications, a keyword-spotting (KWS) benchmark is chosen that deals with real-world sequential data, which is a supervised learning task where a model tries to detect when one or two words are spoken [35]. A real-world application of keyword spotting is to trigger the voice assistant of a smartphone or tablet to launch. For example, at any point, while the assistant is switched off, a user can say *"Hey [name of company]"*, after which the assistant program is launched and starts listening for the next voice input query. This query, usually multiple words or a sentence, can then be processed in a cloud environment with more computing power to perform the voice-to-text operation. The initial keyword spotting task needs to take place in an always-on fashion directly on the device, as it otherwise would require a continuous streaming of device audio to the cloud, which is not scalable to many clients, increases privacy risks and would require a significant energy footprint for communication [35].

As a popular real-world streaming application, there are many software implementations [36] [37] [38] and edge hardware implementations [39] [40] [41] [42] that target keyword spotting. Also, various public datasets are available [35] [43] in this domain.

Feature extraction Keyword spotting is rarely performed on the raw audio of the keywords. Instead, it is common to extract mel-frequency cepstral coefficients (MFCCs) features from the audio and feed these features into a neural network [44]. These features are extracted from windows of the digital audio signal, where each window has a typical length of 25–32 ms with an overlap of 10–16 ms [44]. This means that every 10-16ms, a new feature vector for processing in the neural network becomes available. Each feature vector contains between 10 and 40 MFCCs [45].

Datasets The most frequently used dataset for benchmarking keyword spotters is the Google speech commands (GSC) dataset [35]. It contains 35 classes / unique keywords with a total of 105,289 1-second utterances[3] sampled at 16 kHz (see Fig. 15 for the visualization of a data

[3] This sample count is for version 2 of the dataset; version 1 contained 64,727 samples [35]. Unless specified otherwise, the V2 variant is used by default in this work.

sample). Also included is a set of minute-long clips that contain background noise, such as a recording of someone doing the dishes and a clip of generated white noise.

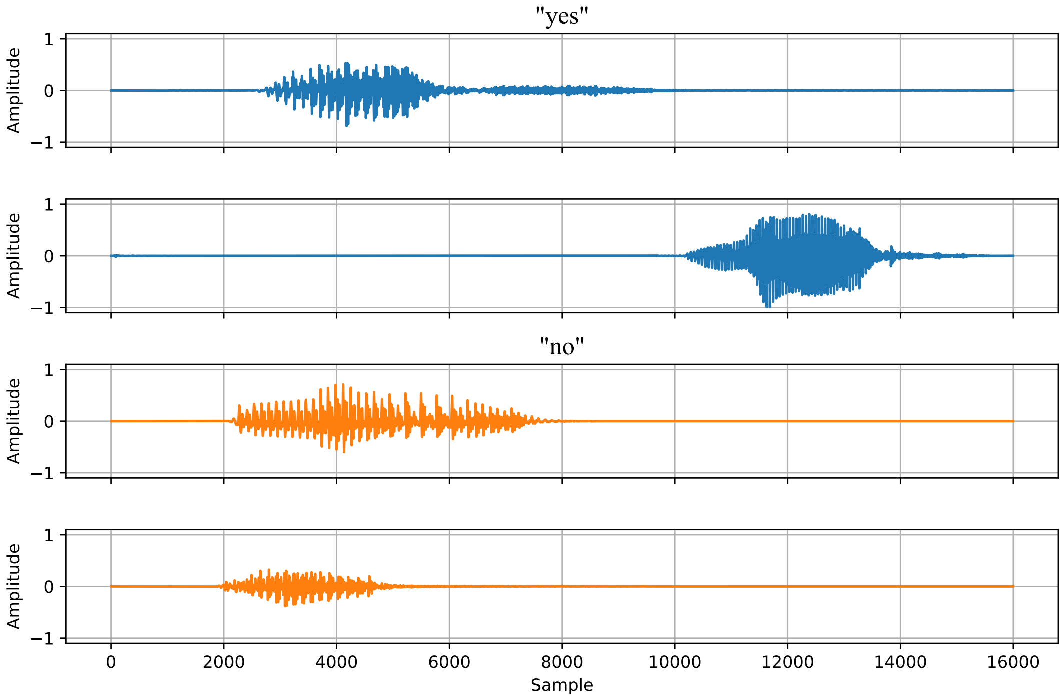


Figure 15: Sample audio waveforms for the "*Yes*" and "*No*" keywords from the V2 Google speech commands dataset [35].

The standard evaluation procedure is to *spot* ten words: *"Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop"* and *"Go"*. Randomly sampled clips from the remaining 25 words form an 11th class "Unknown word", while a 12th class "Silence" contains randomly selected 1-second clips from the background recordings. The goal is to correctly classify the keywords while minimizing false positives for the non-keyword words and background sounds, accurately representing the expected input from a real-world deployment [35].

Another dataset for keyword spotting is the Multilingual Spoken Words Corpus (MSWC) [43]. This dataset contains 340,000 unique keywords from 50 languages with a total of 23.4 million 1-second clips. The benchmark setup for MSWC is a 761-class supervised learning classification task: 760 keywords and one silence class (1.4 million samples, taken from nine languages).

Baseline task definition Before going into the few-shot KWS task definition, a baseline supervised learning task is defined. This is done as there is very little literature in on-chip learning via meta-learning, meaning that it might prove hard to compare this thesis' hardware design with other designs purely based on meta-learning performance.

Therefore, the 12-class GSC variant will be used as a supervised keyword spotting task. This variant is chosen as it is the most popular benchmark for edge hardware and will therefore allow for a fair comparison with many hardware designs.

Few-shot task definition GSC and MSWC do not directly present a true few-shot benchmark scenario. To adapt keyword spotting to a realistic few-shot scenario, it is assumed that the user can train the model at the edge to spot new keywords with only a few user-generated examples.

For example, instead of *"Hey [name of company]"*, a user can enable their virtual assistant with the words *"Wake up [any name]"*. However, compared to few-shot image classification, few-shot keyword spotting is not as popular in the literature. Because of this, there is not yet one commonly agreed-on method to benchmark this task. Therefore, a few possibilities are outlined, after which a single format is selected.

In [46], Chen *et al.*, formulate an N+M way few-shot learning scenario, where N and M are the number of new classes and fixed classes, respectively. The dataset used in this setup is the GSC dataset. Fixed classes are classes that are known a priori, in this case, "Unknown word" and "Silence" ($M = 2$). Two tasks were created to fit this scenario: digit classification using digits zero to nine (see Fig. 16) and command classification using the commands from the standard evaluation procedure of GSC. Next to these ten ($N = 10$) user-defined keywords, five keywords form the "Unknown word" set, while the remaining twenty keywords form the meta-training (D_{source}) set from which tasks can be sampled [46].

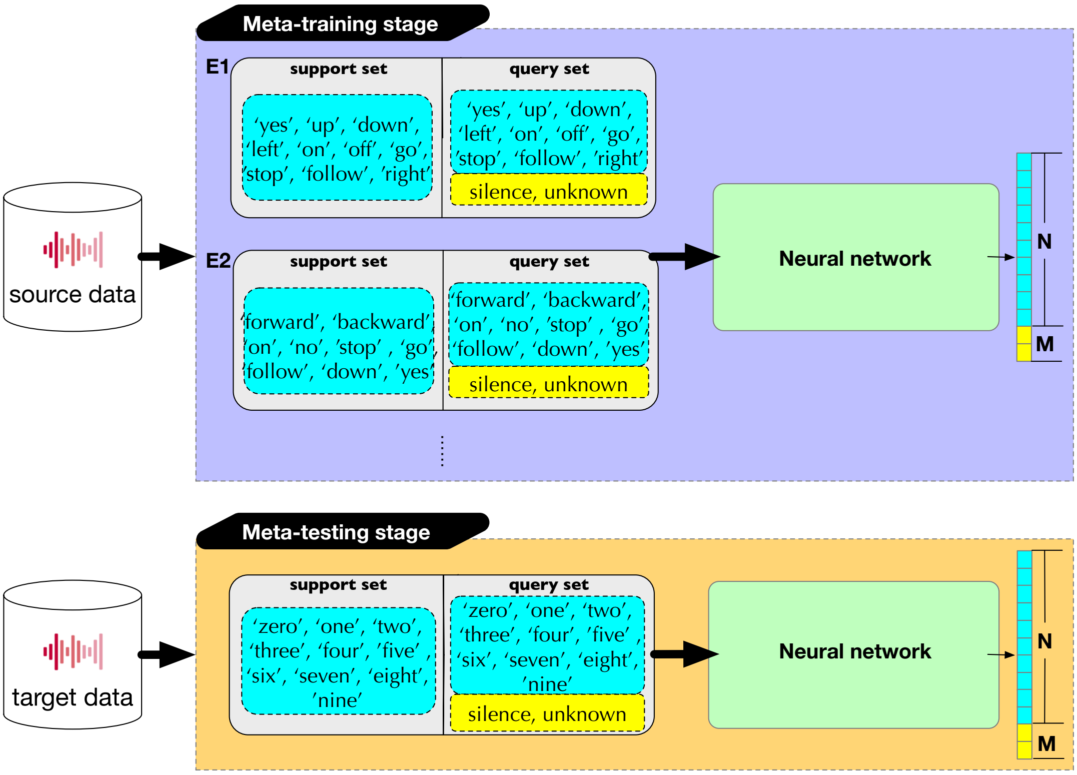

Meta-training stage
E1
support set
query set
'yes', 'up', 'down', 'left', 'on', 'off', 'go', 'stop', 'follow', 'right'
'yes', 'up', 'down', 'left', 'on', 'off', 'go', 'stop', 'follow', 'right'
silence, unknown
source data
E2
support set
query set
'forward', 'backward', 'on', 'no', 'stop' , 'go', 'follow', 'down', 'yes'
'forward', 'backward', 'on', 'no', 'stop' , 'go', 'follow', 'down', 'yes'
silence, unknown
Neural network
N
M
Meta-testing stage
support set
query set
'zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine'
'zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine'
silence, unknown
target data
Neural network
N
M

Figure 16: Digit classification task as presented in [46]. Figure adapted from [46].

For the MSWC dataset, next to the supervised learning setup, Mazumber *et al.* also define a 1+2-way[4] 5-shot learning task in [47]. The 1-way is a new keyword from any of the fifty languages, while the +2-way again represents the "Unknown word" and "Silence" categories. For the "Unknown word" category, 128 samples from a set of 5000 utterances are sampled for each few-shot task.

Alternatively, in [48], Jung *et al.* create a large-scale keyword dataset called LibriSpeech Keywords (LSK), consisting of 1,000 keyword classes extracted from the LibriSpeech corpus [49]. This dataset acts as the pre-training set, after which fine-tuning is performed on GSC. For this, the same ten-digit keywords as in [46] are selected as the user-defined set; 15 keywords then form the "Unknown word" set and the remaining ten command keywords (the same key-

[4] Originally described as 3-way, but using the notation from [46] here.

words as in the standard evaluation procedure of the GSC dataset), form the meta-training set. Overall, this setup corresponds to a 10+1-way task.

While these are not the only approaches for few-shot keyword spotting [50] [51], each of them provides a slightly different setup on a different dataset, as summarized in Table 2. To select a final benchmark in this work, in order to keep development time under control, the primary criterion will be the ease of use. Firstly, the LSK → GSC approach can be discarded, as Jung *et al.* did not open source their dataset. Also, since the benchmark does not contain a silence class, it is not entirely realistic for deployment in an edge device, where most of the input audio will be silent. Between the MSWC few-shot approach and the GSC-based approach, it is clear that the GSC-based approach is favorable to ensure a low iteration time, as training will be 14× faster with 14× fewer data. Therefore, the 10+2-way command classification task from Chen *et al.* [46] is chosen as the few-shot keyword spotting benchmark task for this thesis.

Table 2: Comparison of the three discussed few-shot keyword-spotting benchmarks. **Bold** means best.

Approach name	N+M-way problem [46]	MSWC few-shot [47]	LSK → GSC few-shot [49]
Used dataset(s)	Google Speech-Commands [35]	MSWC [43]	LibriSpeech Corpus [49], GSC [35]
Total sample count	105,289	**1,455,300**	1,000,000[5]
# of meta-training classes	20+2	760+1	**1000+10+1**
# of meta-testing classes	10+2	256+2	10+1
Ways benchmarked	10+2	1+2	9+1
Shots benchmarked	**1, 5, 10**	5	**1, 5, 10**
Public data release	**Yes**	No	**Yes**
Public split release	**Yes**	No[6]	No

2.1.7 Continual learning

Complementary to the concept of meta-learning is *continual learning* (CL), which refers to the fact that neural networks should not only be able to learn from little data, but also to keep learning new tasks sequentially without forgetting previous knowledge, similar to animals. For this reason, CL is sometimes also referred to as *incremental learning* or *lifelong learning* [8].

In continual learning, there are two tracks [2]: learning from new/unseen data belonging to known classes, called *domain-incremental* CL and learning from new/unseen data containing entirely new classes, called *class-incremental* CL. While both tracks relate to meta-learning, class-incremental CL is closest to few-shot classification as new classes are learned and hence links to the contents of this thesis.

While CL does not mandate the use of little data for learning new tasks by definition, this is still a very desirable property at the edge. Such a few-shot continual-learning ability brings the same advantages as those obtained through meta-learning (e.g., privacy-friendly, low maintenance costs), with the ability to keep learning, a key benefit since the outside world evolves continuously [52]. Compared to regular few-shot learning, where the gained knowledge from the previous task is progressively erased when learning on a new task, in CL, knowledge can accumulate. This is visualized in Fig. 17.

[5] This value is estimated from Table 3 in the paper

[6] `https://github.com/harvard-edge/multilingual_kws/issues/41`

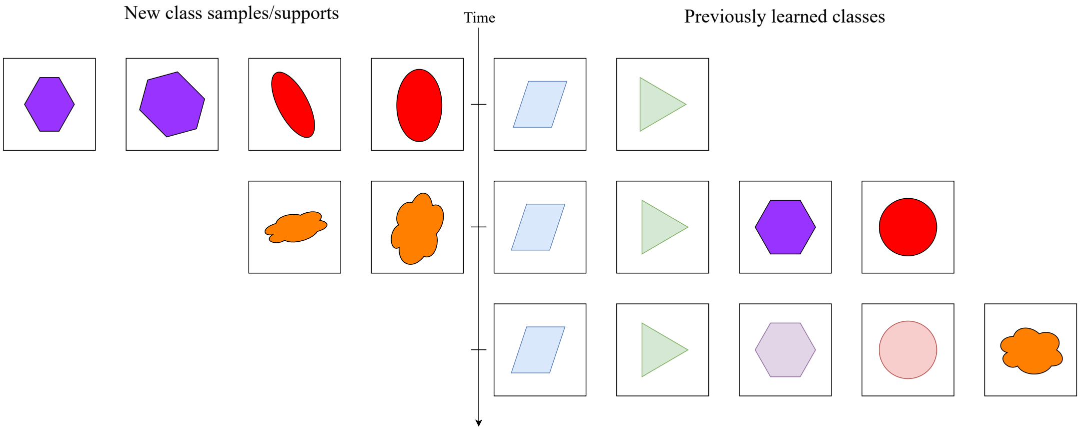


Figure 17: Illustration of class-incremental continual learning versus few-shot learning. The faded classes are learned during continual learning but progressively forgotten in few-shot learning.

The main challenge in continual learning is tackling *catastrophic forgetting/inference* [53], which refers to the phenomenon where, as the neural network gains new knowledge, it starts to forget previously learned knowledge. A few of the key approaches that aim to tackle this problem are discussed next.

In [52], Aljundi *et al.* propose to penalize weight changes of important networks weight while learning a new task, allowing at the same time to overwrite less important weights. Alternatively, Pellegrini *et al.* present a technique called *latent replay* [4]. This builds on top of the rehearsal strategy for continual learning, where some of the input samples from previously learned tasks are stored and added to the training set of a new task [54]. With latent replay, instead of storing the data in the input space, activations at some intermediate layer are stored and used for rehearsal, significantly reducing compute and memory requirements with respect to the naive baseline. Notably, meta-learning can also be used to tackle catastrophic forgetting: the goal is then to learn how to "not forget". MAML [15] is often used as the baseline algorithm for this [55] [56].

Currently however, none of these approaches is completely able to alleviate the *catastrophic forgetting* phenomenon: therefore, an open challenge remains in learning quickly for little data.

2.2 Classifying time-series data

In this section, two neural network architectures for classifying and processing time-series data will be covered. Note that any network designed to perform sequential data classification can also perform few-shot sequential data classification when part of a meta-learning framework (see Section 2.1.4).

The reason for focusing on classifying time-series or sequential data is that at the edge, streaming data is much more common than static data such as images. Examples of such streaming data at the edge are biological signals such as heart rate or blood oxygen over time [57], environmental data such as temperature, humidity or air quality metrics [58], communication data such as human speech [35] or animal behavior [59] and surveillance video for traffic monitoring [60] or anomaly detection [61].

For classifying sequential data, different architectures are available. Excluding hybrid architectures or ensemble approaches, key base architectures are: transformers [62], recurrent neural networks (RNNs), temporal convolutional residual networks (TC-ResNets) [63] and temporal convolutional networks (TCNs) [64].

Transformers demonstrated state-of-art performance on tasks with large amounts of data (4.5 and 36 million sentence pairs [62] for two translation tasks) [62], however, they perform on par or worse compared to TCN and RNN architectures for a variety of tasks that have one or two orders of magnitude less training data (1 million words for a language modeling task) [65]. Furthermore, popular transformer architectures such as BERT [66] only support input sequences of up to 512 tokens, which is limiting even for the Omniglot task having 784 tokens per timestep. Similarly, TC-ResNets [63] have not been demonstrated on sequences beyond 150 timesteps. In addition, traditional transformers have memory requirements that scale quadratically with input sequence length [67].

In contrast, RNNs and TCNs have been demonstrated on tasks with 16,000 timesteps [68] and have a required memory size that is constant (RNNs) or grows logarithmically (TCNs) with sequence length, which is beneficial for hardware design. Therefore, as the datasets used in this thesis are in the order of 100k samples with relatively high sequence lengths, all under the constraints of a custom hardware implementation, only RNNs (Section 2.2.1) and TCNs (Section 2.2.2) are considered for further study.

2.2.1 Recurrent Neural Networks

Recurrent neural networks (RNNs) are ANNs with recurrent connections, capable of learning temporal dependencies from sequential input data. Until the arrival of the transformer architecture [62], these models were the dominant sequence transduction models [62], having effectively become the *de facto* standard for sequence modeling tasks [64].

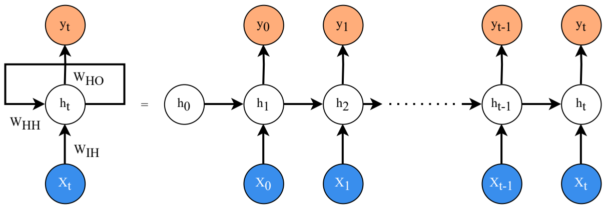


Figure 18: Basic structure of an RNN. Note that the hidden states are dependent on each other, inducing recurrence.

The basic structure of an RNN consists of three layers [69]: an input layer, a **recurrent** hidden layer and an output layer (Fig. 18). The data entering into the input layer is made up of a time-ordered sequence of vectors [69]: $\{\boldsymbol{x}_0, \boldsymbol{x}_1 \ldots, \boldsymbol{x}_{t-1}, \boldsymbol{x}_t\}$.

$$\begin{aligned} \boldsymbol{h}_t &= \mathrm{act}_h\left(\boldsymbol{W}_{IH}\boldsymbol{x}_t + \boldsymbol{W}_{HH}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h\right) \\ \text{s.t.} \quad \boldsymbol{h}_0 &= \mathrm{init}_h() \end{aligned} \tag{14}$$

$$\boldsymbol{y}_t = \mathrm{act}_o\left(\boldsymbol{W}_{HO}\boldsymbol{h}_t + \boldsymbol{b}_o\right) \tag{15}$$

Initially, two fully-connected layers parameterized by $\boldsymbol{W}_{IH}$ and $\boldsymbol{W}_{HH}$ in Eq. (14) map the current input ($\boldsymbol{x}_t$) and the previous hidden state ($\boldsymbol{h}_{t-1}$), together with bias $\boldsymbol{b}_h$ through activation function act_h, into the current hidden state $\boldsymbol{h}_t$ [69]. This hidden state is what forms the recurrent connection between the hidden layers, as shown in Fig. 18. Through training, the hope is then that all necessary information to make the next prediction is embedded/stored in the hidden state. Since for the initial input ($\boldsymbol{x}_0$), the hidden state is not yet computed, an initialization function init_h can be used to define $\boldsymbol{h}_0$.

Finally, the values $\boldsymbol{y}_t$ of the output layer are computed using the current hidden state $\boldsymbol{h}_t$. For this, a linear layer parametrized by weight $\boldsymbol{W}_{HO}$ and bias $\boldsymbol{b}_o$ through activation function act_o is used (see Eq. (15)) [69].

Commonly, both activation functions are nonlinear functions such as sigmoid or tanh (Fig. 19) with the aim of representing increasingly complex features in deeper layers [69]. This is possible since such functions can draw nonlinear boundaries between layers [69], as opposed to stacked linear layers with linear activation functions, which can be represented by a single linear layer [70].

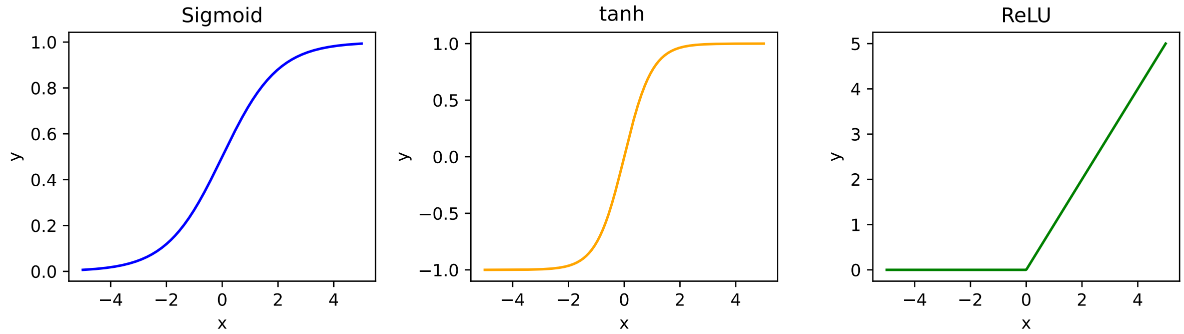


Figure 19: Overview of the sigmoid tanh and rectified linear unit (ReLU) activation functions. Note the difference in y-limits per plot.

Overall, RNNs provide a simple paradigm for modeling time-series data. However, this simplicity has consequences for training such networks. While RNNs can readily be optimized using stochastic gradient descent (SGD), due to the recurrence of the hidden state, gradients can vanish or explode [71]. This effect causes the network to stop learning long-range dependencies and occurs especially when dealing with extended temporal depths.

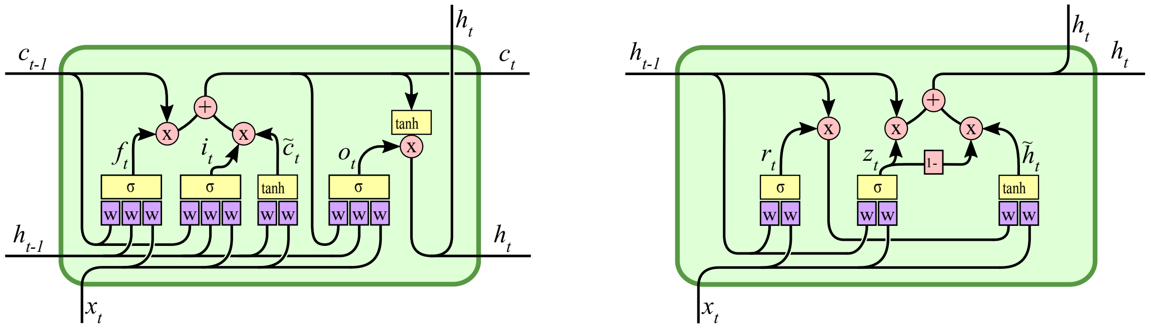
Another issue is the lack of data parallelism within examples during RNN training [62]. This is also an effect of the hidden state: it is not possible to make a prediction for $\boldsymbol{y}_t$ and $\boldsymbol{y}_{t+1}$ in parallel, as in order to compute $\boldsymbol{y}_{t+1}$, $\boldsymbol{h}_t$ first needs to be computed, prohibiting parallelization over the temporal axis within training examples [62]. This problem is most critical for long sequences with high-dimensional $\boldsymbol{x}_t$ vectors, where due to memory limitations, batch processing across multiple examples is restricted [62].

Tackling the first issue are long short-term memory (LSTM) networks [72] from Hochreiter *et al.* As one of the most popular RNN variants, LSTMs have effective gating mechanisms (dot products weighing the inputs and outputs, see Fig. 20a) built into their architecture to reduce the effects of vanishing and exploding gradients. This leads them to have effectively two hidden states: $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$.

Another popular, more recently proposed RNN architecture is the gated recurrent unit (GRU) [73] (see Fig. 20b). Compared to LSTMs, GRUs have a lower overall memory requirement, as there is only a single hidden state $\boldsymbol{h}_t$. However, they still include the gating mechanisms, which is what made LSTMs so effective.

Note that the LSTM and RNN basically both provide a different equation only for the hidden state calculation of Eq. (14). Then, independent of the RNN architecture, the linear layer of Eq. (15) is usually added on top of $\boldsymbol{h}_t$ to compute the final output. For example, for classification, this linear layer is used to compute the last time step logits $\boldsymbol{y}_t$.

[7] `https://kvitajakub.github.io/2016/04/14/rnn-diagrams/`



(a) LSTM architecture

(b) GRU architecture.

Figure 20: Architecture diagrams of two RNNs. The w-boxes represent matrix multiplication with weights w and $\times$ represents a dot-product. Multiple purple boxes below a yellow box indicate that the inputs in the yellow box are added together before passing through the activation function. Figures are taken from.[7]

2.2.2 Temporal Convolutional Networks

Orthogonal to RNNs are temporal convolutional networks (TCNs) [64]. While both neural network types have been developed for sequence modeling, the former is based on recurrence and therefore has a hidden state (see Eq. (14)) while the latter is fully *stateless*. Furthermore, as the name implies, T***C***Ns are based on the convolution operation.

TCNs were, in large part, inspired by WaveNets [68], proposed by Van den Oord *et al.*. A WaveNet is a deep convolutional neural network designed for generating raw audio waveforms: since raw audio has a very high temporal resolution, at a minimum of 16,000 samples per second, WaveNets were specifically designed to handle such long temporal dependencies.

The TCN [64] architecture then is effectively a simplified WaveNet with the following characteristics [64]:

1. the convolutions in the architecture are *causal*: there can be no information leakage from the future to the past,

2. the architecture can perform arbitrary-length sequence-to-sequence modeling (meaning that both the input and the output are same-length sequences), similar to RNNs,

3. TCNs can look further back into the past to make predictions than RNNs, using a combination of very deep networks, residual layers and dilated convolutions.

The first point is crucial, as for sequence-to-sequence modeling, dependencies on future input cause a chicken-and-egg problem: namely, to compute the next output, first, the next output is needed. In contrast to RNNs (i.e. GRUs or LSTMs), which are already inherently causal as only past and current inputs can be used to compute the current output through the current hidden state (see Fig. 18), convolutions are not causal by construction. This is visualized in Fig. 21: with a normal one-dimensional convolution, it is possible that the output at time t depends on an input in the future, $t+1$. With a causal convolution, as shown in Fig. 22, the output at time t always only depends on current or past inputs.

This causal convolution forms the main building block of a TCN: every layer in the network is a 1D causal convolution, where the length of each hidden layer is kept the same as the input sequence length, to fulfill point 2. One-dimensional instead of the more common two-dimensional convolutions (compare Fig. 54b with Fig. 24) are used, as 2D convolutions

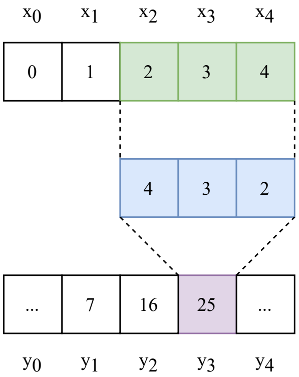


Figure 21: Regular 1D single-channel convolution.

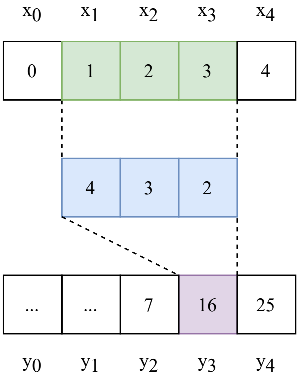


Figure 22: Causal 1D single-channel convolution.

are mainly suited for inputs with two main dimensions, such as images, having a width and a height. Sequential data on the other hand only has one dimension: the width or time dimension. Note that when performing 1D convolutions on *multichannel* sequences, the operation (see Fig. 25) is nearly the same as a single-channel 2D convolution, except that there is now only one (horizontal) sliding axis.

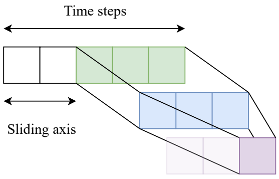


Figure 23: 1D single-channel convolution with kernel size 3 and input sequence length 5.

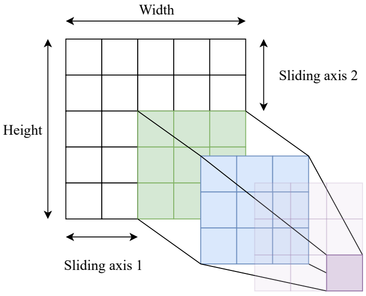


Figure 24: 2D single-channel convolution with a 3 × 3 kernel on a 5 × 5 input image.

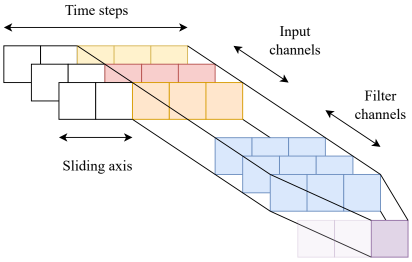


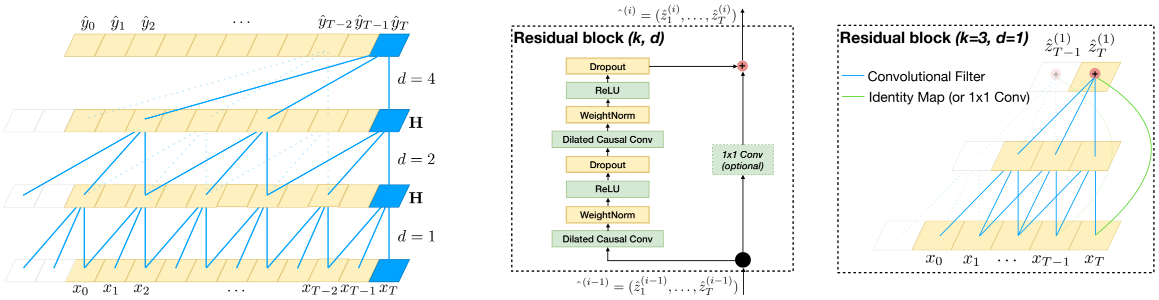
Figure 25: 1D *multichannel* convolution with kernel size 3, input sequence length 5 and 3 channels.

To achieve a large *receptive field* size (i.e. the total input sequence length that the network can receive), as per point 3, using only stacked 1D causal convolutions requires that the network should either be very deep or that it should have large-sized filters [64]. This is because the receptive field size of normal stacked causal convolutional layers only grows linearly with the depth of the network [64].

Therefore, TCNs instead use *dilated* causal convolutions, which is also what allowed WaveNets [68] to handle such large contexts. In a dilated convolution, there are $d-1$ elements of space between the filter elements of the kernel. In TCNs, this dilation doubles every layer, resulting in an exponentially growing receptive field size with network depth [64]. Fig. 26a visualizes this structure. Based on the kernel size k (assuming it is the same in every layer), the number of layers n and the dilation exponent base d_{exp} of the network, the receptive field size r of a TCN can then be calculated using Eq. (16), taken from[8] and [74]:

$$r(n,k,d_{\text{exp}}) = 1 + \sum_{l=1}^{n} 2 \cdot (k-1) \cdot {d_{\text{exp}}}^{l-1}. \tag{16}$$

[8] `https://github.com/locuslab/TCN/issues/44#issuecomment-677949937`


$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$ ... $\hat{y}_{T-2}$ $\hat{y}_{T-1}$ $\hat{y}_T$
$d = 4$
H
$d = 2$
H
$d = 1$
x_0 x_1 x_2 ... x_{T-2} x_{T-1} x_T
$\hat{z}^{(i)} = (\hat{z}_1^{(i)}, \ldots, \hat{z}_T^{(i)})$
Residual block (k, d)
Dropout
ReLU
WeightNorm
Dilated Causal Conv
Dropout
ReLU
WeightNorm
Dilated Causal Conv
1x1 Conv (optional)
$\hat{z}^{(i-1)} = (\hat{z}_1^{(i-1)}, \ldots, \hat{z}_T^{(i-1)})$
Residual block (k=3, d=1)
$\hat{z}_{T-1}^{(1)}$ $\hat{z}_T^{(1)}$
Convolutional Filter
Identity Map (or 1x1 Conv)
x_0 x_1 ... x_{T-1} x_T

(a) Stacked dilated convolutions. **(b)** Residual block definition. **(c)** Residual block usage.

Figure 26: Overview of the TCN architecture elements. Taken from [64].

Furthermore, inspired by the residual block for deep convolutional neural networks [75], every TCN layer (see Fig. 26a) also has this residual-block structure, where a layer can be bypassed, as shown in Fig. 26b. However, unlike the original block, the authors also include spatial dropout [76] for regularization and weight normalization [77]. Finally, a 1×1 convolution (i.e. a matrix-vector multiplication) is optionally applied in case the incoming and outgoing channel counts do not match [64].

Completing the architectural description, in order to perform classification, a classification head is added to the last output step activation (in the same way as Eq. (15)).

The design of the TCN gives the architecture several benefits compared to RNNs [64]. The first one is that TCNs have stable gradients throughout the network while performing back-propagation, as the data flows not through time but through the depth of the network (compare Fig. 18 with Fig. 26a). TCNs therefore do not suffer from vanishing/exploding gradients. TCNs also offer processing parallelism as they are stateless, enabling the efficient utilization of computational resources. Moreover, their low memory requirements during training make them computationally efficient [64]. However, it is important to note that the receptive field of a TCN is fixed (per Eq. (16), unlike in an RNN, where it could theoretically be infinite). Therefore, one should be careful not to reuse a TCN designed for a short context window in a situation where much more context is required, as it might result in a performance drop[64].

2.3 Neural network quantization

In order to deploy modern deep neural networks on low-power edge hardware where memory is limited, quantization is an essential ingredient. Therefore, in this section, background information about this process is provided.

First, in Section 2.3.1, the necessity of quantization is discussed, after which the fundamentals of quantization are outlined in Section 2.3.2. Following this, one way to perform training for quantized neural networks is discussed in Section 2.3.3. Finally, Section 2.3.4 discusses the possible choices for number representations in quantization.

2.3.1 The need for quantization

As mentioned earlier, modern deep neural networks are capable of performing increasingly complex tasks [9] [11]. However, such systems do not only require vast amounts of data for training (as mentioned in Section 2.1.1), they also come with a high computational and memory cost [78]. Therefore, to deploy modern deep networks on edge devices having limited memory and compute resources, it is key to effectively minimize the footprint of neural network pro-

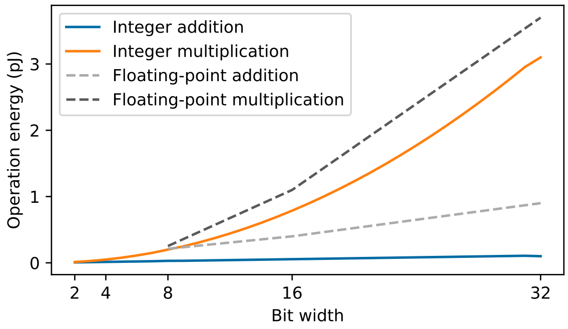


Figure 27: Energy for addition and multiplication with integer and floating-point formats for varying bit-width. Data taken from [79].

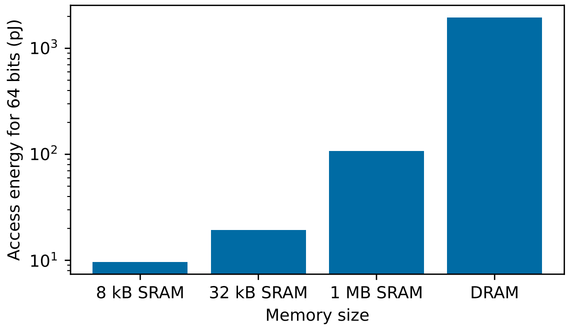


Figure 28: Energy to access 64 bits of data from different memories. Data taken from [79].

cessing. Quantization, i.e. the process of **reducing** the bit width of neural network weights, activations, biases etc., is one of the most effective ways to do this [78]: it namely jointly reduces the network's memory footprint, power and latency.

This originates from the fact that quantization affects the hardware execution of neural networks at the two most power-consuming levels: computation and data transfer. Starting with the former, Fig. 27 demonstrates that the energy per addition operation increases linearly with bit-width while the energy per multiplication scales quadratically with bit-width, as multiplication is effectively a chained addition. This means that switching from 32- to 8-bit numbers saves a factor of 4 in addition energy and a factor of 16 in multiplication energy. More savings can be gained by using simpler number representations: notice how the energy for floating-point operations in Fig. 27 is always higher than its integer counterpart.

Concerning the latter, Fig. 28 indicates that, the smaller the memory, the lower the energy to access a fixed-bitwidth word will be: it also shows how off-chip data access to DRAM incurs a significant energy penalty. Again, going from 32- to 8-bit operands makes the required memory size a factor 4 smaller, leading to both lower energy per access and lower overall access count.

However, neural network quantization does not come without a cost: noise introduced by the loss of resolution in operators can unfortunately significantly decrease the accuracy of the network [78].

2.3.2 Quantization fundamentals

With the need for quantization outlined, this section will dive into the core fundamentals. First, in Section 2.3.2.1, the main operation affected by quantization is introduced. Following this, in Section 2.3.2.2, the basic quantization operation is explained, after which Section 2.3.2.3 covers the conversion back to floating point. Then, Section 2.3.2.4 discusses the difference between symmetric and asymmetric quantization operations. Finally, Section 2.3.2.5 compares per-tensor and per-channel quantization.

2.3.2.1 Multiply-accumulate operation Processing modern ANNs mainly involves performing multiply-accumulate (MAC) operations in the form of matrix-vector multiplications: $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$. This operation forms the basis of matrix-matrix multiplication and convolution [78], commonly found in ANNs.

An example hardware implementation of a block processing this matrix-vector multiplication can be seen in Fig. 29, where the operational equations are:

$$A_n = b_n + \sum_m C_{n,m} \tag{17}$$

$$\text{s.t.} \quad C_{n,m} = W_{n,m} x_m \tag{18}$$

In a non-quantized setting, all of these variables would be 32-bit floating point (FP32) values. However, in the quantized version of this operation, the weights ($W_{n,m}$), biases (b_n), inputs (x_m) and accumulators (A_n) are all integers of a given bit width. Note that this bit width does not have to be the same between these three types of values. Subsequently, Eq. (17) can be computed in fixed-point/integer arithmetic, which is considerably cheaper than the equivalent FP32 computation, even if the bit width remains 32 (as per Fig. 27).

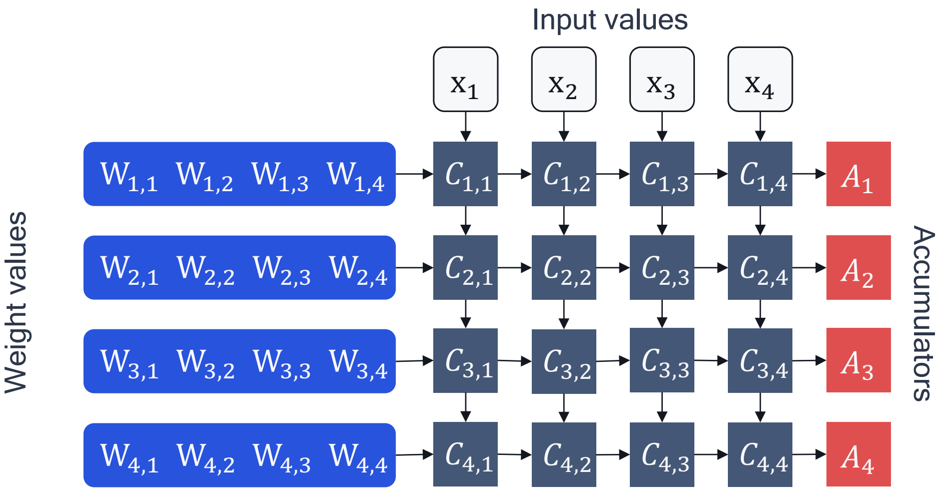

Input values
x_1 x_2 x_3 x_4
Weight values
$W_{1,1}$ $W_{1,2}$ $W_{1,3}$ $W_{1,4}$
$W_{2,1}$ $W_{2,2}$ $W_{2,3}$ $W_{2,4}$
$W_{3,1}$ $W_{3,2}$ $W_{3,3}$ $W_{3,4}$
$W_{4,1}$ $W_{4,2}$ $W_{4,3}$ $W_{4,4}$
$C_{1,1}$ $C_{1,2}$ $C_{1,3}$ $C_{1,4}$
$C_{2,1}$ $C_{2,2}$ $C_{2,3}$ $C_{2,4}$
$C_{3,1}$ $C_{3,2}$ $C_{3,3}$ $C_{3,4}$
$C_{4,1}$ $C_{4,2}$ $C_{4,3}$ $C_{4,4}$
A_1 A_2 A_3 A_4
Accumulators

Figure 29: Overview of a hypothetical matrix-vector multiplier block. Figure taken from [78].

In order to convert floating-point operations to their fixed-point counterparts, a scheme for converting FP32 values to quantized integers is required. For this, Eq. (19) provides a simplified conversion:

$$\hat{\boldsymbol{x}} = s_{\boldsymbol{x}} \cdot \boldsymbol{x}^{\text{int}} \approx \boldsymbol{x}, \tag{19}$$

where $\boldsymbol{x}$ is the floating point value that should be converted, $\hat{\boldsymbol{x}}$ the approximated (after quantization) $\boldsymbol{x}$, $s_{\boldsymbol{x}}$ a floating-point scale value, and $\boldsymbol{x}^{\text{int}}$ the quantized value corresponding to $\boldsymbol{x}$. For example, if $\boldsymbol{x} = 0.0912$ and $s_{\boldsymbol{x}} = 0.001$, then $\boldsymbol{x}^{\text{int}} = 91$ and $\hat{\boldsymbol{x}} = 0.091$. The difference $\hat{\boldsymbol{x}} - \boldsymbol{x}$ is due to the integer approximation and is the first type of quantization error: the *rounding error*.

As not only the inputs ($\boldsymbol{x}$) but also the weights ($\boldsymbol{W}$) need to be quantized to perform Eq. (17) and Eq. (18) in fixed-point, a scale $s_{\boldsymbol{W}}$ is also defined for the weights. The resulting fully quantized equations are then [78]

$$\begin{aligned} \hat{A}_n &= \hat{b}_n + \sum_m C_{n,m} \\ &= \hat{b}_n + \sum_m W_{n,m} x_m \\ &= \hat{b}_n + \sum_m \left(s_{\boldsymbol{W}} W^{\text{int}}_{n,m} \right) \left(s_{\boldsymbol{x}} x^{\text{int}}_m \right) \\ &= \hat{b}_n + s_{\boldsymbol{W}} s_{\boldsymbol{x}} \sum_m W^{\text{int}}_{n,m} x^{\text{int}}_m , \end{aligned} \tag{20}$$

where $\hat{A}_n$, $\hat{b}_n$ are the approximated A_n and b_n and $W^{\text{int}}_{n,m}$, x^{int}_m are the quantized weights and inputs. While $s_{\boldsymbol{x}}$ and $s_{\boldsymbol{W}}$ could be combined into one scale value via multiplication, using separate values makes it easier to, for example, use different quantization schemes for the activations and weights. As these two floating-point scales are moved out of the summation in Eq. (20), it means that all MAC operations can now be performed in fixed-point [78]. Note that the accumulation in Eq. (20) is usually performed in a higher bit width than the operands to avoid overflow.

The bias was left as an approximated value in Eq. (20) as its quantization approach varies from strategy to strategy. Usually, however, the combined weight and activation scale is used for the bias scale s_b [80]:

$$\begin{aligned}
\hat{A}_n &= \hat{b}_n + s_{\boldsymbol{W}} s_{\boldsymbol{x}} \sum_m W^{\text{int}}_{n,m} x^{\text{int}}_m \\
&= s_b b^{\text{int}}_n + s_{\boldsymbol{W}} s_{\boldsymbol{x}} \sum_m W^{\text{int}}_{n,m} x^{\text{int}}_m \\
&= (s_{\boldsymbol{W}} s_{\boldsymbol{x}})\, b^{\text{int}}_n + s_{\boldsymbol{W}} s_{\boldsymbol{x}} \sum_m W^{\text{int}}_{n,m} x^{\text{int}}_m \\
&= s_{\boldsymbol{W}} s_{\boldsymbol{x}} \left(b^{\text{int}}_n + \sum_m W^{\text{int}}_{n,m} x^{\text{int}}_m \right).
\end{aligned} \tag{21}$$

Defining s_b like this allows for only a single scaling operation (i.e. a floating point multiplication), which makes the required hardware simpler. Finally, due to this scaling by $s_{\boldsymbol{W}} s_{\boldsymbol{x}}$, $\hat{A}_n$ also becomes a floating-point value. However, as the current layer's output $\hat{A}_n$ is the next layer's input activation x_m, $\hat{A}_n$ should be requantized before going back into the MAC array. This step is called *requantization* and it is effectively the inverse of Eq. (19).

2.3.2.2 Quantization As briefly touched upon, quantization is the process of mapping floating-point values to the integer grid [78]: this section introduces the mapping scheme used throughout this thesis

The *asymmetrical quantization* or *uniform affine quantization* scheme is the most commonly used scheme [78], as it can be readily implemented using hardware that supports fixed-point arithmetic (see Fig. 29). The scheme builds on top of the simplified quantization approach of Eq. (19) and depends on three parameters: an already-introduced quantization scale s, the quantization zero-point z and the bit width of the quantized number b, which are an FP32 value and two integers, respectively. Furthermore, the following other symbols are defined:

- x: any real number that requires quantization,
- x_{int}: the quantized number, an integer.

A real number x is then mapped to the unsigned integer grid $\{0, ..., 2^b - 1\}$ via [78]:

$$x_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{x}{s} \right\rceil + z; 0, 2^b - 1 \right), \tag{22}$$

where $\lfloor \cdot \rceil$ is the round-to-nearest operation and the clamp function is defined as [78] (plotted in Fig. 30), where a and c are the lower and upper clipping limits respectively.

$$\text{clamp}(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \leq x \leq c, \\ c, & x > c \end{cases} \tag{23}$$

From Eq. (22), it can be seen that $\frac{1}{s}$ controls the step size of the quantization: for example, when s is large, a large change in x is required to change x_{int} while when s is small, only a small change in x results in a large change in x_{int} (see Fig. 31).

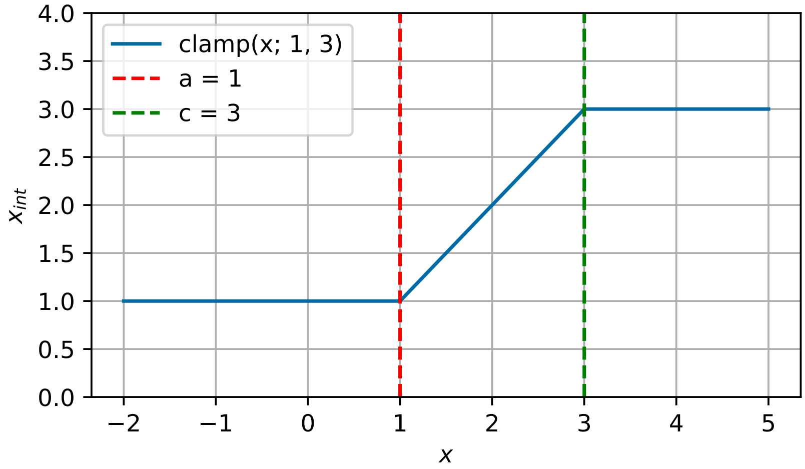


Figure 30: Plot of the clamp function for $a = 1$ and $c = 3$.

The function of the zero-point, z on the other hand, is less evident: z is the scaled-and-rounded value corresponding to the real 0 (the "zero-point") in the x domain, ensuring that the real 0 is always quantized without error [78].

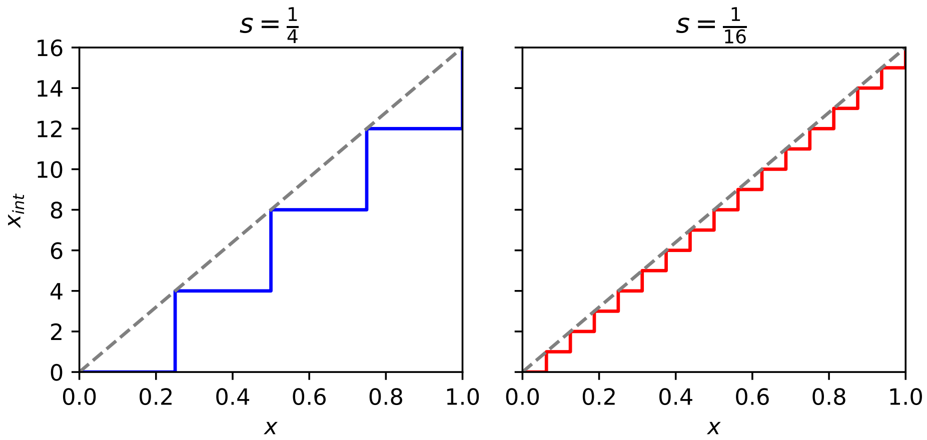


Figure 31: Effect of the quantization scale s on the step size of the quantization.

2.3.2.3 Dequantization After the mapping or quantization step, dequantization is performed to compute the approximated real value from the quantized value (similar to Eq. (19)):

$$x \approx \hat{x} = s\left(x_{\text{int}} - z\right). \tag{24}$$

Substituting Eq. (22) into Eq. (24), the so-called "quantization function" is obtained [78]:

$$\hat{x} = q(x; s, z, b) = s\left(\text{clamp}\left(\left\lfloor\frac{x}{s}\right\rceil + z; 0, 2^b - 1\right) - z\right). \tag{25}$$

Subsequently, the grid limits (q_{min} and q_{max}) of the quantization in Eq. (25), can be computed. Any value of x that lies outside these limits is clipped to the limits, inducing the first type of quantization error, called clipping error [78]:

$$\begin{aligned} q_{min} &= -sz, \\ q_{max} &= s(2^b - 1 - z). \end{aligned} \tag{26}$$

As an effect, a trade-off between the clipping and rounding error is introduced. By choosing a larger s, the clipping error is reduced as the clipping bounds grow (per Eq. (26)), however, the rounding error is increased as it lies in the range $\left[-\frac{1}{2}s, \frac{1}{2}s\right]$ (based on Eq. (22)) [78].

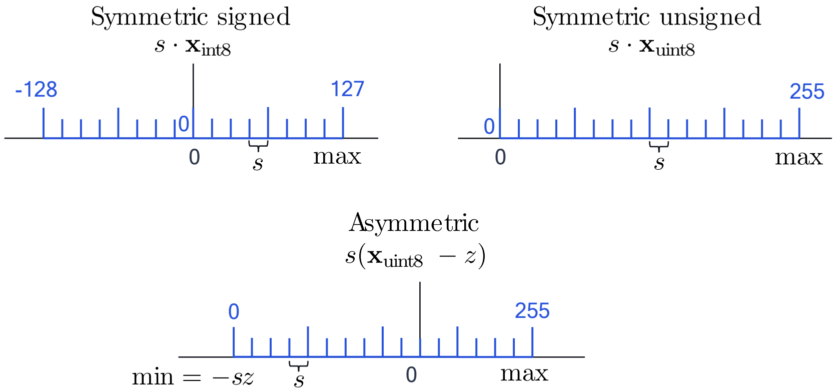
As mentioned before, s is stored in a floating-point format [78]; however, an alternative is to use *power-of-two* scale values [78]. This limits the possible scale values to $2^{s_{\text{exp}}}$, where s_{exp} is any integer [78]. Multiplying by the scale can then be done via a cheap bit shift, instead of via an FP32 multiplication.

2.3.2.4 Symmetric vs. asymmetric quantization In the previous section, the mapping procedure for the asymmetric quantization scheme was covered. However, for the special case of $z = 0$, a symmetric quantization scheme is extracted.

The advantage of a symmetric scheme is that the zero-point offset does not have to be considered while computing Eq. (17). However, due to the lack of zero-point, the only way to represent negative numbers now is by using a signed integer grid:

$$x_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{x}{s} \right\rceil; -2^{b-1}, 2^{b-1} - 1\right). \tag{27}$$

Symmetric signed schemes prove most useful in situations where the to-be-quantized values are symmetric around zero, whereas the symmetric unsigned scheme can be used for ReLU activated values [78]. For a visual overview of these different schemes, see Fig. 32.



Symmetric signed
$s \cdot \mathbf{x}_{\text{int8}}$
-128
127
0
0
s
max
Symmetric unsigned
$s \cdot \mathbf{x}_{\text{uint8}}$
255
0
0
s
max
Asymmetric
$s(\mathbf{x}_{\text{uint8}} - z)$
0
255
$\min = -sz$
s
0
max

Figure 32: An overview of the different quantization grids for $b = 8$. The floating-point grid is in black and the integer quantized grid is in blue. Figure taken from [78].

As separate scales for the weights and activations were defined in Eq. (20), this built-in flexibility now also allows for choosing a symmetric or asymmetric quantization scheme for each of these. Most commonly, however, asymmetric activation quantization and symmetric weight quantization are used [78]. To understand this, consider using asymmetric quantization for the weights ($\boldsymbol{W}$) and activations/inputs ($\boldsymbol{x}$)

$$\begin{aligned} W_{n,m}x &= s_{\boldsymbol{W}}\left(W_{n,m}^{\text{int}} - z_w\right) s_x \left(x_{\text{int}} - z_x\right) \\ &= s_{\boldsymbol{W}} s_x W_{\text{int}} x_{\text{int}} - s_{\boldsymbol{W}} z_w s_x x_{\text{int}} - s_{\boldsymbol{W}} s_x z_x W_{\text{int}} + s_{\boldsymbol{W}} z_w s_x z_x, \end{aligned} \tag{28}$$

where z_w is the zero-point for W and z_x the zero-point for x. While the blue part in Eq. (28) can be pre-computed and added to the bias for the operation, the red part depends on x_{int} and is therefore not known in advance. Therefore, to avoid the computational overhead of this term, often only the activations are quantized asymmetrically.

2.3.2.5 Per-tensor and per-channel quantization

So far, every weight matrix has its own, unique scale and optionally zero-point (see Eq. (28)): this is called per-tensor quantization. Instead, with per-channel quantization, every output channel has its own zero-point and scale [78]. At the cost of extra parameters, this method can increase a quantized model's performance when the distribution of weight values varies significantly between channels.

Per-channel quantization can also be performed for the activations. However, while in per-channel weight quantization, $s_{\mathbf{W}}$ has only to be replaced by $s_{\mathbf{W},n}$ in Eq. (20), keeping the overall same equation structure, for per-channel activation quantization, every MAC entry in the summation term also requires a different scale $s_{\mathbf{x},m}$. This is not preferred, as it introduces many extra scaling operations [78]. Hence, usually, only per-channel weight scaling is considered.

2.3.3 Quantized training

In this section, the way neural networks are prepared and converted to a quantized format for deployment on inference accelerators is discussed.

Starting off, there are two main approaches for quantizing networks: post-training quantization (PTQ) and quantization-aware training (QAT). The easiest of the two approaches is PTQ, as no retraining is required [78]. However, PTQ can have issues reaching desirable accuracies on bit widths of 4 or less [78]. As in this thesis, minimizing the power consumption of the hardware is very important, which can partially be achieved by using low bit width representations (see Fig. 27), it was chosen to focus on QAT directly.

Therefore, first, in Section 2.3.3.1, quantization-aware training is explained. Then, Section 2.3.3.2 covers how residual layers are dealt with in this framework. Finally, Section 2.3.3.3 discusses the integration of the batch normalization [81] operation during training and before deployment.

2.3.3.1 Quantization-Aware Training

In QAT, the quantization function (Eq. (25)) is part of the network's computational graph during the backward and forward pass. However, a direct implementation of this will not work, as the basic quantization operation (see Eq. (22)) contains a round-to-nearest operation. Backpropagating through this operation results in a gradient that is zero or undefined everywhere [78].

To solve this, Bengio *et al.* [82] proposed the straight-through estimator (STE), which approximates the gradient of the round-to-nearest operation to be 1. Formally, this can be written as (refer to [78] for more details on the full gradient definition of Eq. (22)):

$$\frac{\partial \lfloor y \rceil}{\partial y} = 1. \tag{29}$$

Using this equation, also the gradients through the quantization operation are defined. Subsequently, backpropagation can now be applied to train a quantized network. In Fig. 33, the forward graph and backward graph are illustrated for a dummy convolutional layer. In the forward pass, the quantization function (Eq. (25)) is computed in the quantizer blocks as defined: however, in the backward pass, this function is replaced by STE. Therefore, the backward pass in QAT using STE is exactly the same as a normal backward pass: the quantizer block is effectively bypassed. Through QAT, not only are the neural network weights directly optimized for quantized inference, also all the scales and zero-points are now learnable parameters and optimized through gradient descent [78].

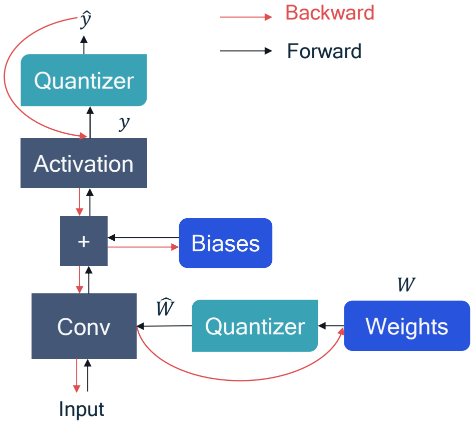


Figure 33: Forward and backward computational graph with quantizer blocks for activations and weights. W and y are the FP32 weights and outputs, while $\hat{W}$ and $\hat{y}$ are the quantized weights and outputs, respectively. Figure taken from [78].

Please note that contrary to what the name might suggest, QAT is rarely performed starting from a randomly initialized model. Commonly, QAT is preceded by FP32 training [78], where the FP32-trained model then forms the starting point for QAT.

2.3.3.2 Residual layer handling When quantizing neural network models that have residual connections (see Fig. 34), such as the TCN architecture discussed in Section 2.2.2, special care to be dedicated to quantization of the weights in the residual path.

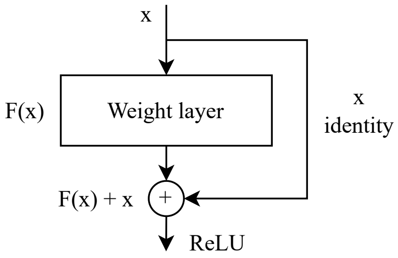


Figure 34: Residual layer block with ReLU activation function, where $F(x)$ is any function that maps x. Figure based on [75].

This is due to the fact that the quantization operation of Eq. (25) is not a linear operation [83]: $q(a+b) \neq q(a)+q(b)$. As such, performing the summation and then quantizing is not the same as summing quantized values. Assuming that the weight layer $F(\boldsymbol{x})$ is $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$, which $\boldsymbol{y} = s_{\boldsymbol{W}} s_{\boldsymbol{x}} \boldsymbol{W}^{\text{int}} \boldsymbol{x}^{\text{int}}$ when quantized and assuming that the output after residual summation $(F(\boldsymbol{x})+\boldsymbol{x})$ is $\boldsymbol{a} = s_{\boldsymbol{a}} \boldsymbol{a}^{\text{int}}$, the final quantized output is:

$$\boldsymbol{a}^{\text{int}} = \frac{s_{\boldsymbol{x}}}{s_{\boldsymbol{a}}} \boldsymbol{x}^{\text{int}} + \frac{s_{\boldsymbol{W}} s_{\boldsymbol{x}}}{s_{\boldsymbol{a}}} \boldsymbol{W}^{\text{int}} \boldsymbol{x}^{\text{int}} \tag{30}$$

It can now be seen that **two** separate scaling operations are required (unlike with the normal quantization function, Eq. (25)), before performing the residual addition. Therefore, dedicated hardware support is required to process neural networks with residual connections, as the operation does not fit in a standard quantized inference engine.

2.3.3.3 Batch normalization folding Batch normalization [81], where the output of a linear layer is normalized before scaling and offsetting it, has become an essential building block of modern convolutional networks. While it was not originally part of the architectural description of the discussed network types (Section 2.2.1 and Section 2.2.2), both LSTMs [84] and TCNs can be extended with this block to improve classification performance possibly.

The batch normalization operation is defined as follows:

$$\text{BatchNorm}(\boldsymbol{x}) = \gamma\left(\frac{\boldsymbol{x}-\mu}{\sqrt{\sigma^2+\varepsilon}}\right)+\beta, \tag{31}$$

where γ and β are learnable parameters (in other words, optimized by gradient descent) and μ and σ are the exponential moving average mean and variance of the incoming $\boldsymbol{x}$ batches during training, all of them defined per-channel. Both μ and σ are fixed during inference, meaning that before deployment on an edge device, the batch normalization operation can be folded [85] [80] into the previous linear layer's weights. This effectively removes the complete batch normalization step from the network, with the operation absorbed by the linear layer computation. For a linear layer with weights $\boldsymbol{W}$ and inputs $\boldsymbol{x}$ passing through a batch normalization layer $\boldsymbol{y} = \text{BatchNorm}(\boldsymbol{W}\boldsymbol{x})$, the folding for the kth channel is defined as:

$$\begin{aligned}
\boldsymbol{y}_k &= \text{BatchNorm}\left(\boldsymbol{W}_{k,:}\boldsymbol{x}\right)\\
&= \boldsymbol{\gamma}_k\left(\frac{\boldsymbol{W}_{k,:}\boldsymbol{x}-\boldsymbol{\mu}_k}{\sqrt{\boldsymbol{\sigma}_k^2+\varepsilon}}\right)+\boldsymbol{\beta}_k\\
&= \frac{\boldsymbol{\gamma}_k\boldsymbol{W}_{k,:}}{\sqrt{\boldsymbol{\sigma}_k^2+\varepsilon}}\boldsymbol{x}+\left(\boldsymbol{\beta}_k-\frac{\boldsymbol{\gamma}_k\boldsymbol{\mu}_k}{\sqrt{\boldsymbol{\sigma}_k^2+\varepsilon}}\right)\\
&= \tilde{\boldsymbol{W}}_{k,:}\boldsymbol{x}+\tilde{\boldsymbol{b}}_k
\end{aligned} \tag{32}$$

where $\tilde{\boldsymbol{W}}_{k,:}$ and $\tilde{\boldsymbol{\beta}}_k$ can subsequently be defined as:

$$\begin{aligned}
\tilde{\boldsymbol{W}}_{k,:} &= \frac{\boldsymbol{\gamma}_k\boldsymbol{W}_{k,:}}{\sqrt{\boldsymbol{\sigma}_k^2+\varepsilon}}\\
\tilde{\boldsymbol{\beta}}_k &= \boldsymbol{\beta}_k\ -\frac{\boldsymbol{\gamma}_k\boldsymbol{\mu}_{k,:}}{\sqrt{\boldsymbol{\sigma}_k^2+\varepsilon}}
\end{aligned} \tag{33}$$

When dealing with batch normalization layers in QAT, a simple but effective strategy is to statically, in other words, before QAT, fold the batch normalization layers into the pre-trained FP32 network's weights [78]. After this, the batch normalization layer can be removed from the network

2.3.4 Number representations

So far, quantization has been discussed in the context of converting to n-bit integer formats. However, the definition of quantization is not limited to using integer number representations

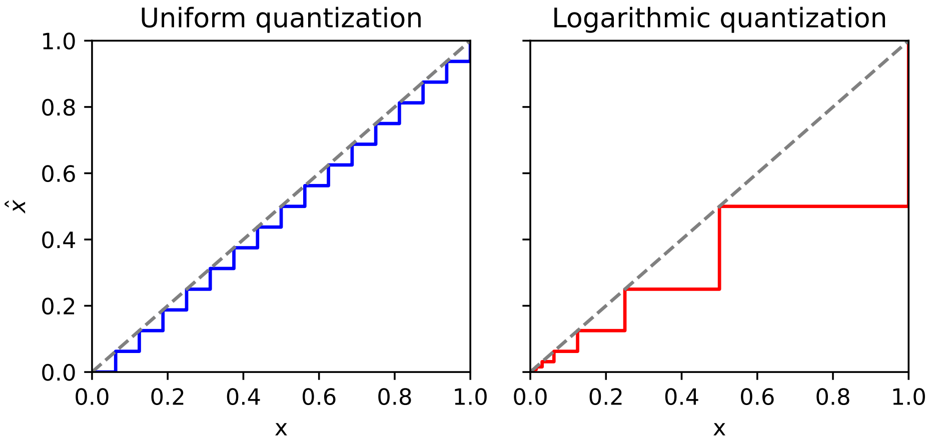


Figure 35: Comparison of weight distribution for a 4-bit (16 levels) uniform and base-2 logarithmic format. Note how the logarithmic weights are much more densely distributed close to zero than further away from zero.

only. Therefore, one other number representation will be discussed in this section: the non-uniform base-2 logarithmic quantization scheme.

In this scheme, first introduced by Miyashita *et al.* [86], all the quantized values are powers of two, for example, $\frac{1}{2}$, 4, 16, 2 and 64. Such a scheme has multiple advantages compared to linear quantization:

1. the distribution of the weights of a neural network is often non-uniform (high concentration around zero) [87]: this is naturally supported by a logarithmic definition (see Fig. 35),
2. integer quantization methods incur a significant drop in accuracy when less than 8 bits are used [86], however for logarithmic weights, even with 3-bits the original floating point accuracy can be nearly maintained [86],
3. there is no need for multiplier hardware, saving area and power, as all computations can be performed via bit shifts [86].

In a similar format to Eq. (25) (the linear quantization function), a quantization function for this logarithmic scheme can be defined, per [87]:

$$\hat{x} = \mathrm{LogQuant}\left(x; b, \mathrm{FSR}\right) = \begin{cases} 0, & x = 0, \\ 2^{\tilde{x}}, & \text{otherwise} \end{cases} \tag{34}$$

where:

$$\tilde{x} = \mathrm{clip}\left(\lfloor \log_2\left(|x|\right) \rceil ; \mathrm{FSR} - 2^b, \mathrm{FSR}\right) \tag{35}$$

$$\mathrm{clip}\left(x; \min, \max\right) = \begin{cases} 0, & x < \min, \\ \max - 1, & x \geq \max, \\ x, & \text{otherwise} \end{cases} \tag{36}$$

where FSR is the full scale range and determines the maximum and minimum quantization exponent: in a QAT setting, this scale is optimized through gradient descent. Furthermore, in the case of signed quantization, the sign of x can be prepended to Eq. (34), requiring one extra bit of storage. Notably, this quantization function prevents zero-valued weights, as all weights near zero are promoted to the minimum quantization level [87].

Based on this quantization function, there are two main scheme variants: the first variant quantizes both the weights and activations using base-2 logarithmic quantization, while the second one only quantizes either the weights or activations logarithmically whereas the other values are quantized linearly. To compare these two, the dot product $y = \boldsymbol{w} \cdot \boldsymbol{x}$ is defined, where $\boldsymbol{x} \in \mathbb{R}^n$ is the input and $\boldsymbol{w} \in \mathbb{R}^n$ is the weight vector. Starting with the former variant:

$$\begin{aligned} \boldsymbol{w}^T \boldsymbol{x} &\approx \sum_{i=1}^{n} 2^{\tilde{w}_i + \tilde{x}_i} \\ &= \sum_{i=1}^{n} \text{bitshift}(1, \tilde{w}_i + \tilde{x}_i), \end{aligned} \tag{37}$$

where the bitshift(a,c) function shifts value a by c bits to the left. Next, defining the latter variant (in this case, only $\boldsymbol{x}$ is logarithmically quantized):

$$\begin{aligned} \boldsymbol{w}^T \boldsymbol{x} &\approx \sum_{i=1}^{n} w_i \cdot 2^{\tilde{x}_i} \\ &= \sum_{i=1}^{n} \text{bitshift}(w_i, \tilde{x}_i), \end{aligned} \tag{38}$$

Note that in Eq. (38), $\boldsymbol{w}$ can also be an FP32 vector, as in floating-point, this operation is simply an addition of each $\tilde{x}_i$ with the exponent part of each w_i.

Depending on the required memory savings (see representation sizes in Fig. 36), the desired performance (see Table 3) and the functionality offered by the considered hardware, a scheme from the above options should be selected.

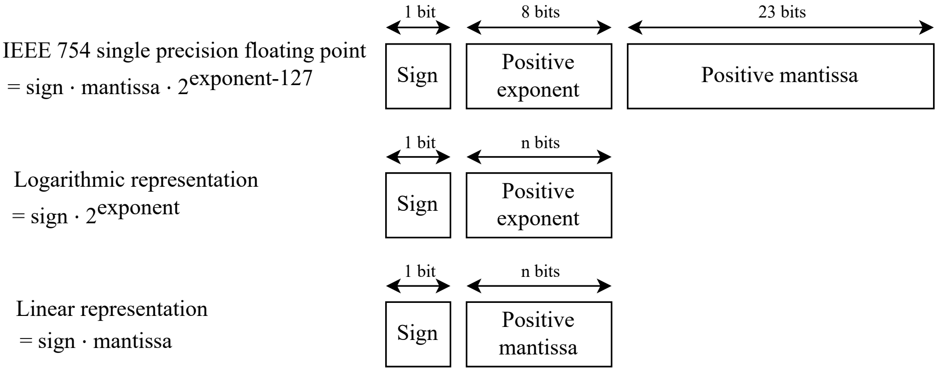


Figure 36: Bitwise overview of three different number representations: FP32, base-2 logarithmic representation and linear/integer representation.

Table 3: Accuracy of various quantized models on ImageNet [88], where "Act" is activations, "Conv" indicates the convolutional weights and "FC" refers to the fully connected weights. "L4" means logarithmic quantization with 4 bits, while U3 stands for uniform quantization with 3 bits. Data from [87] [86].

Quantization			Top-5 accuracy		Top-1 accuracy	
Act	Conv	FC	AlexNet [89]	VGG16 [90]	ResNet-18 [75]	ResNet-50 [75]
FP32	FP32	FP32	78.3	89.2	69.76	76.13
FP32	4L	8U	-	-	69.87	76.38
FP32	4L	4L	-	-	69.53	76.31
L4	FP32	4L	76.8	89.5	-	-
L4	U5	4L	73.6	85.1 $	-	-
L4	L5	4L	70.6	83.4	-	-
L3	FP32	FP32	76.9	89.8	-	-
L4	FP32	FP32	76.9	89.8	-	-
U3	FP32	FP32	77.1	83.0	-	-
U4	FP32	FP32	77.6	89.4	-	-

3 Network design

In this section, neural networks for the three benchmarks of this thesis will be designed. These neural networks will all be based on the same architecture type, for which hardware support will then be developed.

This is done in two parts: first, in Section 3.1, the neural network type will be selected by considering various metrics for RNNs and TCNs, after which Section 3.2 derives the architectural hyperparameters of the selected network type per task.

3.1 Architecture type choice

In this section, the two neural network architectures covered in the background section, RNNs and TCNs, will be compared on various levels, after which an architecture is selected for implementation in hardware.

Initially, Section 3.1.1 compares the performance between the two approaches to sequence modeling, after which Section 3.1.2 investigates the training characteristics of the network types. These two sections use empirical data from Bai *et al.* [64]. Next, Section 3.1.3 covers the hardware considerations per architecture. Then, in Section 3.1.4, based on the information presented in the previous sections, the final network type is chosen.

3.1.1 Performance

This section compares the performance between RNNs (LSTMs, GRUs, pure RNNs) and TCNs on four widely used benchmarks for recurrent neural networks. In Section 3.1.1.1 to 3.1.1.3, first the used datasets are discussed, after which Section 3.1.2 summarizes the performance of the networks on these datasets.

3.1.1.1 Sequential & permuted MNIST As briefly touched upon in Section 2.1.6.1, sequential MNIST is a task where the MNIST handwritten digit images have to be classified, although the images are now presented as a sequence. In permuted MNIST (P-MNIST) [34], the order of the pixels is permuted randomly: this permutation is the same for all samples (see Fig. 37), however, it can be different between training for multiple separate models.

These two tasks are included in the performance comparison as they are classification tasks (although not few-shot) over a relatively long sequence length of 784 timesteps. Furthermore, the data looks visually very similar to the Omniglot data, hence the performance of a model on this task is likely a good proxy metric for performance on Omniglot with the same model.

3.1.1.2 Adding problem The adding problem is a task that was first introduced by Hochreiter *et. al* [72]. In this task, a 2D input is presented at every timestep for a total sequence length T: the first dimension contains zeros everywhere except for two entries that are 1 and the second dimension is a random number in $[0, 1]$. The aim is then to find the sum of the two values that were marked with a 1. Note that in every sequence, the positions of the two one-valued entries change and the random numbers are regenerated. The mean-squared error (MSE) is used as the loss function for this task. Fig. 38 displays an example adding task.

This task is included in the benchmark selection, as it is a test that can be used to understand how long networks can remember. By training a model for each (increasing) value of T and monitoring the loss, it can be determined how far back the network can look. The value for T at which the loss suddenly increases indicates this point.

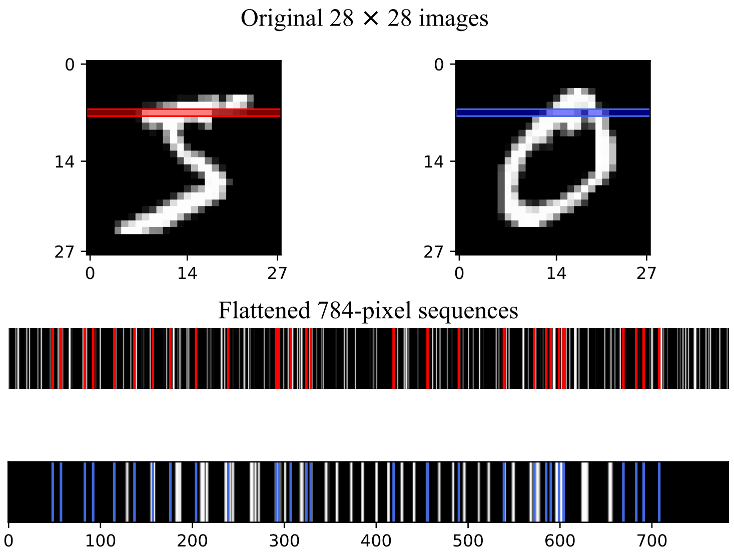


Figure 37: Two P-MNIST samples. Note how the permuted locations are the same for the two samples.

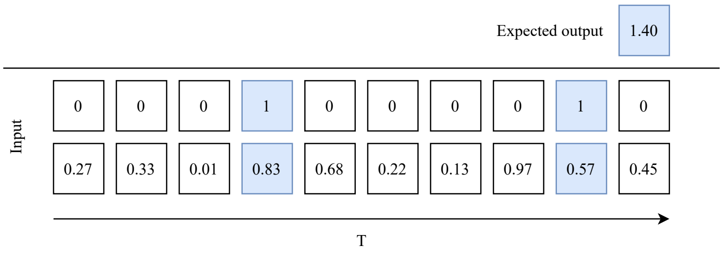


Figure 38: Example task for the adding problem.

3.1.1.3 Copy memory In this task, first introduced by Arjovsky *et. al* [91], every input sequence comprises a length of $T+20$. The initial ten values are drawn at random from the numbers 1 through 8, while the remaining values are set to zero, excluding the final 11 entries, which are occupied by the digit 9 (with the first 9 serving as a separator). The objective is then to generate an output sequence with the same length as the input sequence, where all values are zero except for the final ten values occurring after the separator. In this segment, the model needs to replicate the ten values it encountered at the beginning of the input. Similarly to the adding problem, the copy problem is also an *effective memory* test and also uses MSE as the loss function. An example copy memory task is shown in Fig. 39.

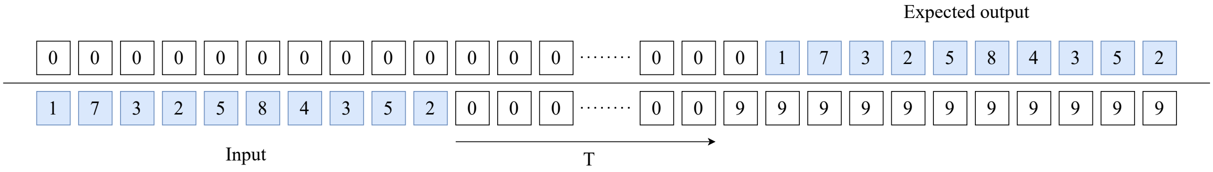


Figure 39: Example copy memory task.

3.1.1.4 Results In Table 4, the performance of the four models (LSTMs, GRUs, pure RNNs and TCNs) on the previously discussed benchmarks are shown. For each benchmark, the parameter count between the models is approximately the same to make sure that none of the models have a performance advantage in that respect. The exact hyperparameters used for each model across these tasks are reported by Bai *et al.* in [64].

Table 4: Performance results from four network architectures on the discussed four sequence modeling tasks. ^{h} indicates that higher metrics are better, ^{l} indicates that lower metrics are better. **Bold** highlights the best performance per task. Data taken from [64].

Task	Model Size (≈)	LSTM	GRU	RNN	TCN
Sequential MNIST (accuracy^h)	70K	87.2	96.2	21.5	**99.0**
Permuted MNIST (accuracy^h)	70K	85.7	87.3	25.3	**97.2**
Adding problem T = 600 (loss^l)	70K	0.164	**5.3e-5**	0.177	**5.8e-5**
Copy memory T = 1000 (loss^l)	16K	0.0204	0.0197	0.0202	**3.5e-5**

It can be clearly seen that the TCN architecture systematically outperforms the recurrence-based architectures. While the GRU is still on par with the TCN in sequential MNIST and in the adding problem, for P-MNIST and the copy memory task, it is surpassed by the TCN. Furthermore, the pure RNN performs severely worse than any of the other architectures. Therefore, the pure RNN architecture is removed from the network selection options: in the following, only the LSTM, GRU and TCN will be considered.

Next to pure performance, the effective receptive field can be compared between the architectures. Recurrent architectures theoretically have an infinite effective receptive field or memory [64], as they do not have any architectural limits to how many inputs a network can maintain in its working memory, unlike TCNs (see Eq. (16)). To compare the effective receptive field size of recurrent networks with that of the TCN, the copy task can be used. Fig. 40 shows the accuracy of the different modes on the copy for increasing values of T. All models trained for this experiment contained 10k parameters.

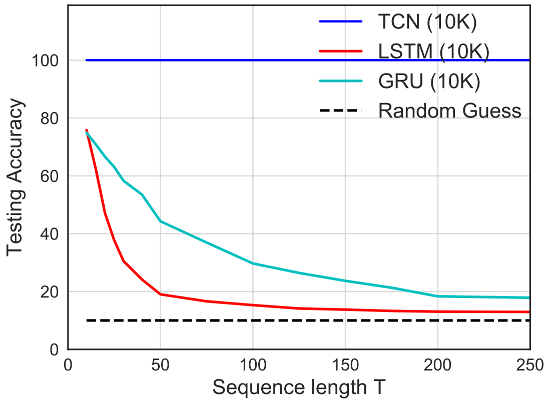


Figure 40: Accuracy on the copy memory task for sequences of different lengths T. Taken from [64].

It can be seen that as T increases, the accuracy of the GRU and LSTM drops asymptomatically to random guessing performance, while the TCN has 100% accuracy throughout all values of T. While this is a purely synthetic task, it demonstrates that TCNs are able to have a longer effective receptive field size than the GRU or the LSTM.

3.1.2 Training properties

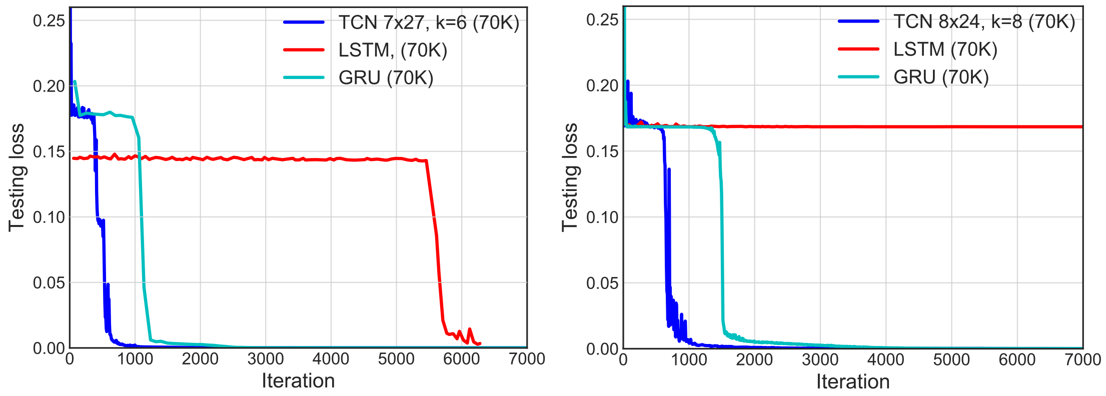
Next, the training speed and convergence speed will be compared between the different architectures. The training speed (or wallclock time per epoch) is relevant, since in meta-learning, often either small batch sizes are used or double derivatives need to be calculated, further slowing down training. The training convergence, i.e. how fast the network reaches a desired performance, is also an appropriate metric, since in few-shot learning, only a few examples are available to update the network. Therefore, an architecture that converges in fewer gradient steps is preferred.

Comparing training speed on tasks with sequences between 100-16,000 elements, it is demonstrated that TCNs are approximately 1.5-3× slower than RNNs (Table 5). Notably, the difference increases for longer sequences: this is likely due to the fact TCNs require more layers to fully cover the increasing input length, requiring more computation.

Table 5: Comparison of training speed per epoch on various-length tasks. All training hyperparameters were kept the same for each task. Own data.

			Seconds per epoch		
Task	**Model size**	**Length**	**LSTM**	**TCN**	**LSTM speed-up**
KWS (MFCC)	17k	115	2.87	3.79	1.32x
Sequential MNIST	70k	784	14.57	26.04	1.79x
KWS (raw data)	30k	16,000	31.62	88.32	2.79x

To compare the convergence speed, the adding problem will be used, as it tests learning over long timescales in a challenging setup. In Fig. 41a, the loss progression during this task with $T = 200$ can be seen, while in Fig. 41b, the loss progression for this task with $T = 600$ is shown. All the networks in both tests have approximately the same parameter count.



(a) $T = 200$.

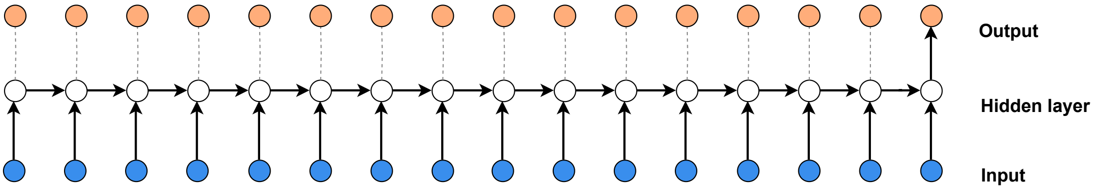
(b) $T = 600$.

Figure 41: Loss progression on two adding problems for three different sequence modeling architectures. The TCN is found to converge at a minimum 12× faster than the LSTM and 2× faster than the GRU. Figure adapted on [64].

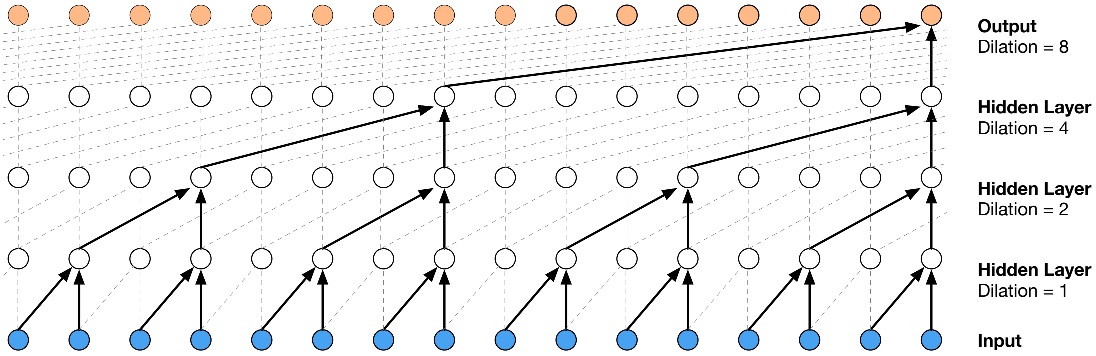
Fig. 41 clearly demonstrates that TCNs learn much quicker and converge rapidly to a near-zero loss solution. The GRU is second-best (approximately 2× slower than the TCN), while the LSTM only converges in the $T = 200$ case, but after 12× as many iterations as the TCN required.

So while the LSTM is faster to train per iteration, it takes significantly more iterations to converge. Offsetting this, the TCN is very roughly speaking 4× faster than the LSTM, while the GRU is approximately 50% faster than the TCN.[9]

[9] Note that the seconds per epoch were not published for the adding task by Bai *et al.*. Therefore, due to time



(a) RNN receptive field.



(b) TCN (with $k = 2$) receptive field. Taken from [68].

Figure 42: Comparison between RNN and TCN architectures showing the receptive field from and information flow through the networks. Thick arrows indicate information flow to the final output, while dashed lines indicate information flow to other outputs.

The reason that TCNs converge faster is likely due to the fact that the gradients are more stable through the network (explored in Section 2.2.2), as they do not have to flow through time but through the depth of the network. This can be seen when comparing the flow of information between Fig. 42a and Fig. 42b. It is clear that the longest path through the network scales linearly with input length in the case of the RNN, while it scales logarithmically for a TCN.

3.1.3 Hardware considerations

Since the selected neural network architecture has to be implemented in hardware, it is important to compare the hardware implications of each of the network types. Therefore, in this section, the required activation memory size (Section 3.1.3.1) and the number of multiplication operations (Section 3.1.3.2) are derived per architecture.

3.1.3.1 Activation memory size The reason for picking the activation memory size as a metric is that it has a direct impact on both the power usage and the on-chip area. Namely, a larger activation memory size means that a physically larger memory is required, which takes up more silicon area and uses more power overall. The required weight memory size on the other hand is purposefully not considered in this thesis. The reason for this is that the weight memory size will be chosen based on a high-level requirement on the maximum number of parameters, in other words, independently from the architecture.

Now, for each network type, the required activation memory size is calculated. This calculation is based on the following assumptions:

- a classification setting is assumed, similar to what is required during few-shot learning,

constraints, it was decided to measure training speeds using already configured benchmarks, which the adding task was not.

- only the last timestep output of the network is used for classification,
- the batch size is 1,
- each input vector is presented sequentially,
- all computations are performed greedily. This means that as soon as a new input comes in, all computations that can be performed, will be performed: for example, when the 4th input is available in Fig. 42b, only the two states above the input will be calculated after which new inputs are awaited.

Under these assumptions, Eq. (39) is found:

$$\begin{aligned} A_{\text{size,LSTM}} &= 6 \cdot n_h + n_i, \\ A_{\text{size,GRU}} &= 4 \cdot n_h + n_i, \\ A_{\text{size,TCN}} &= 2nk \cdot n_h + k \cdot n_i, \end{aligned} \tag{39}$$

where n_h is the number of hidden units in the case of the LSTM and GRU and the number of channels in every layer in case of the TCN. While, by definition, the number of channels does not have to be the same in every TCN layer, for simplicity this is assumed here. n_i then is the dimensionality of the input and n is the number of layers and k is the kernel size in the TCN.

Based on Eq. (39), Fig. 43 displays a plot of the total network parameter count versus the activation value count per network for many combinations of n_h and n_i for all architectures and n and k for the TCN.

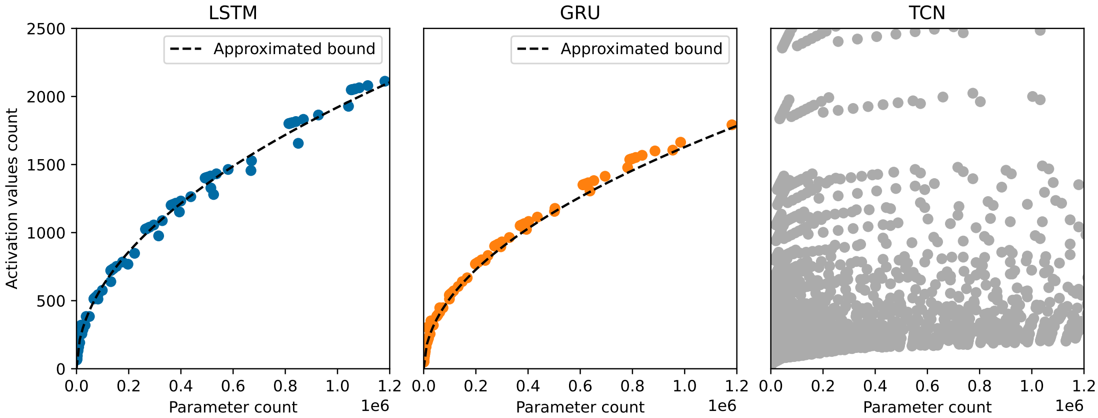


Figure 43: Parameter count versus activation values count for LSTMs, GRUs and TCNs. The approximated bound is a scaled square-root function.

It can be seen that while for LSTMs and GRUs, the number of values to keep in memory is high when the network is large, for TCNs even small networks can have a very high requirement on the number of values to keep in memory.

Filling in the specifications from the network architectures used for sequential MNIST[10], results in the following activation memory size requirements:

[10] The hidden size of the GRU network was estimated from its parameter count, as the authors did not publish this information in [64].

$$\begin{aligned} A_{\text{size,LSTM}} &= 6 \cdot 130 + 1 = 780 \\ A_{\text{size,GRU}} &= 4 \cdot 150 + 1 = 601 \\ A_{\text{size,TCN}} &= 2 \cdot 8 \cdot 7 \cdot 25 + 7 \cdot 1 = 2807 \end{aligned} \tag{40}$$

Clearly, the TCN has the largest activation memory requirement for this task, with the LSTM and GRU being comparable to each other. More generally speaking, on average, TCNs require more values to be stored in the activation memory than its recurrent counterparts.

3.1.3.2 Required multiplications The total required multiplications are compared between the three architectures as it is a proxy metric (assuming that a fixed number of multiplications can be computed per second) for the inference time and energy per inference (a result of the number of multiplications and number of weight accesses).

In this regard, Eq. (41), Eq. (42) and Eq. (43) display the number of multiplications for the LSTM, GRU and TCN respectively under the same assumptions stated in Section 3.1.3.1. Eqs. (41) to (42) are taken from Rezk *et al.* [92]; where T is the input sequence length, whereas Eq. (43) is derived in this thesis, with r referring to Eq. (16) and again assuming that all intermediate layers in the TCN have the same channel count.

$$\mathrm{M}_{\text{count,LSTM}} = T\left(4n_h^2 + 4n_h n_i + 3n_h\right) \tag{41}$$

$$\mathrm{M}_{\text{count,GRU}} = T\left(3n_h^2 + 3n_h n_i + 3n_h\right) \tag{42}$$

$$\begin{aligned} \mathrm{M}_{\text{count,TCN}} = n_h \sum_{l=1}^{n} k\left(r\left(l,k,d_{\text{exp}}\right) - k + 1\right) \left(\begin{cases} n_i, & l = 0, \\ n_h, & \text{otherwise} \end{cases}\right) \\ + \frac{1}{2}\left(k + \left(\begin{cases} 0, & l \neq n, \\ n_i/n_h, & \text{otherwise} \end{cases}\right)\right)\left(r\left(l,k,d_{\text{exp}}\right) - 2k + 3\right) n_h \end{aligned} \tag{43}$$

In Fig. 44 three different scenarios are plotted: in each scenario, every network has the presented number of parameters and input channels. This means that for the LSTM and GRU, the minimum number of hidden units has to match that parameter count has to be calculated knowing n_i. For the TCN, at every receptive field size value, the combinations of k and n are found that have the smallest r still larger than the required receptive field. Knowing k, n and n_i, n_h for the TCN can then be found in the same way as for the LSTM and GRU.

Fig. 44 shows that the TCN always requires a lower number of multiplications than its recurrent counterparts. Fig. 44 also shows that the number of operations increases linearly for the GRU and LSTM (due to the multiplication with T in Eqs. (41) to (42)) while it increases logarithmically for the TCN.

This means that for a given network size, sequence length and sequence dimensionality, the inference time of a TCN network is always lower (assuming the same number of multiplications per second) thus requiring less energy due to reduced runtime.

3.1.4 Final choice

With all of the above in mind, the final network can be chosen. First, the results from the previous sections will be briefly summarized.

Initially, Section 3.1.1 showed that TCNs have superior performance across the board compared to LSTMs, GRUs and RNNs, although GRUs were not too far off. However, in the

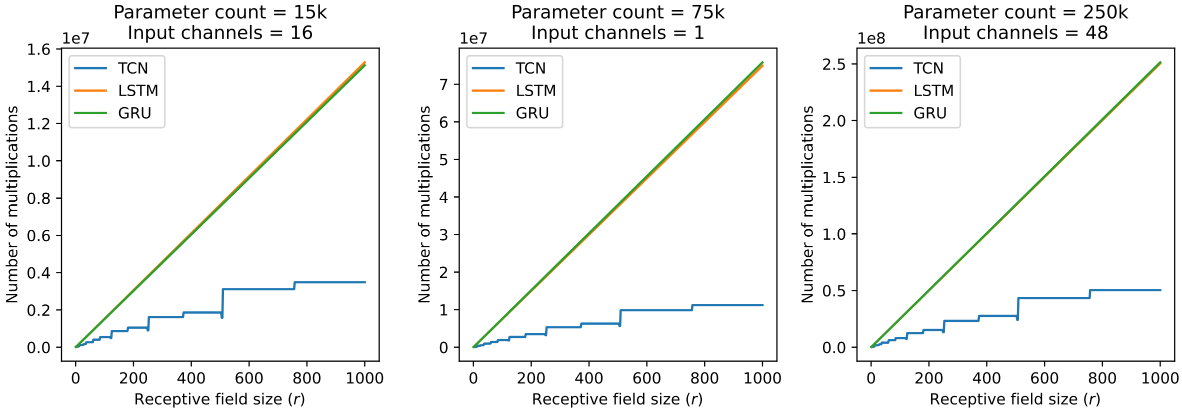


Figure 44: Comparison of the number of multiplications between TCNs, GRUs and LSTMs for various network configurations on tasks with increasing receptive field. Note the difference in scaling factor for the three y-axes.

effective history size experiment, TCNs were a clear winner. Due to very poor performance, the pure RNN was then dropped from the list of options. Then, in Section 3.1.2 their training characteristics of the remaining three architectures were compared: TCNs converge in less iterations but also need more training time per iteration. Putting these metrics together, GRUs are the best option in this respect. Finally, Section 3.1.3 showed that both recurrent architectures generally require fewer activations to be stored, but that TCNs require far fewer multiplications in all settings.

It is then decided to pick the TCN architecture. It exhibits the best performance, has a very good memory and requires the least number of iterations to converge (very important for meta-learning). However, it does come at the expense of training time and on-chip activation memory size, both of which have to be investigated further to see if there are possible improvements for these downsides.

3.2 Network designs per benchmark

With the architecture type chosen, in this section, the structural hyperparameters of one network per benchmark will be chosen. First, Section 3.2.1 briefly discusses how k and n should be chosen per task based on the input sequence length, after which Section 3.2.2 presents the final network architectures per benchmark.

3.2.1 Receptive field size considerations

In a TCN, the structure of the network defines a physical limit to the maximum receptive field size (as per Eq. (16)). Therefore, care has to be taken in selecting k and n. Note that in a TCN, by definition, every layer could have a different kernel size and channel count, in the following it is assumed that the kernel size k is the same throughout the network.

Plotting Eq. (16) as a heatmap ($d_{\text{exp}} = 2$ is assumed) makes it easier to consider which combinations of k and n can be used for a certain task. Ideally, one of the combinations results in a receptive field exactly equal to the data length of the task. However, if that is not the case, one of the combinations with the lowest receptive field values still above the input sequence length should be used. This lower limit should be respected as otherwise, the network will not receive the full input sequence.

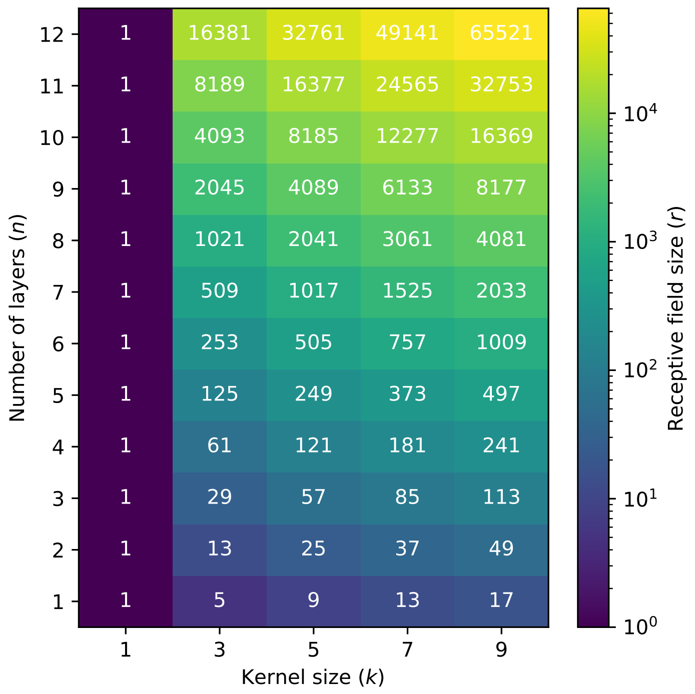


Figure 45: Heatmap plot of Eq. (16). Each box contains the receptive field size for the given kernel size k and layer count n. $d_{\text{exp}} = 2$ is assumed for this plot, the default value for TCNs.

For networks with a higher receptive field size than input length, the input can simply be left-padded with zeroes. This is not a problem in cases where the network has a slightly higher receptive field size than the input sequence length, however, for large differences, the network will have parameters that are never used, thus obsolete.

3.2.2 Final designs

In this section, the final TCN architectures for the three selected tasks will be discussed. The reason for pre-designing these architectures instead of optimizing them is that multiple experiments need to be performed with each of the networks: having a stable baseline therefore aids in comparing the results of these experiments.

First, the network for the 12-class KWS task will be covered in Section 3.2.2.1. After this, the few-shot KWS network architecture is derived in Section 3.2.2.2. Finally, the TCN architecture for sequential Omniglot is derived in Section 3.2.2.3. Serving as an overview for this section, Table 6 displays the final parameters for the three discussed architectures.

Note that while the original TCN architecture [64] shipped with weight normalization [77], in this thesis batch normalization is applied. Furthermore, to avoid excessive hyperparameter optimization, a dropout [76] value of 0.025 was selected for all architectures. Finally, although not shown in Table 6, the Kaiming weight initializer [93] is chosen over random weight initialization for all TCNs, as this initializer is specifically designed for ReLU-activated networks, including the TCN.

3.2.2.1 12-class KWS Starting off, the network architecture for the 12-class KWS task is discussed (leftmost column in Table 6). Note that in this model, the channel count values are

[11] Includes batch normalization parameters (as discussed in Section 2.3.3.3) but excludes linear layer

Table 6: Final TCN network designs for the three benchmark tasks of this thesis.

Dataset/task	12-class KWS	Few-shot KWS	Sequential Omniglot
k	7	7	5
n	3	3	7
Receptive field size	85	85	1017
Channels	$[16,16,32]$	$[48,48,64]$	$[32,32,32,48,48,48,48]$
Input dimensionality	32	32	1
Dropout	0.025	0.025	0.025
Weight norm	No	No	No
Batch norm	Yes	Yes	Yes
Parameter count[11]	21,484	114,784	116,960

all divisible by 16 to make sure that it will be easy to design the core compute block of the hardware design.[12]

For this architecture, it was desired to have a model size of around 20k parameters to allow for a fair comparison with other hardware designs that often use network sizes slightly below and above this value [39] [44] [42] [40] [94].

It was chosen to have more channels in deeper layers instead of the other way around, as deeper layers in convolutional neural networks form more refined representations [95] and therefore benefit from having more channels available. Next to this, large convolutional neural networks [75] [90] increase their channel count with depth, as a testimony to this. In the end, only the last layer in this network uses 32 channels, as using 32 channels in any of the other layers increases the parameter count to approximately 30k, 50% more than the desired network size.

Furthermore, a window size of 32 ms and a stride of 16 ms were used for MFCC feature extraction, yielding a total of 63 steps from each audio sample (16,000 original timesteps). These values were selected as they are often used by other accelerator designs [40] [94] [44], again making comparing the design of this thesis against other designs easier.

While most of these works extract 40 MFCCs per frame, in this work it is chosen to extract 32 MFCCs per frame. This was done as 40 is not exactly divisible by 16, leading to lower-than-100% utilization in the case of a 16×16 multiplier array, while 32 is exactly divisible by 16. While 48 could have also been chosen as it is also divisible by 16, it was decided to use 32 MFCCs since there are no hardware designs that use close to 50 MFCCs while there are worrks that use 30 MFCCs [42]. Furthermore, using 48 coefficients would make comparisons against 40D methods unfair, as the task would be made easier through the higher quality of information stored in this dimensionally larger space.

Finally, the exact values of $k = 7$ and $n = 3$ for this network were found by using the configuration lowest receptive field size still larger than 63 (see Fig. 45), making sure that most of the network parameters are part of the computational graph.

3.2.2.2 Few-shot KWS Next, the network architecture for the few-shot keyword spotting task is explained (second column, Table 6). While for this task, the exact same network architecture as described previously could be used, since the input format is the same, it was decided

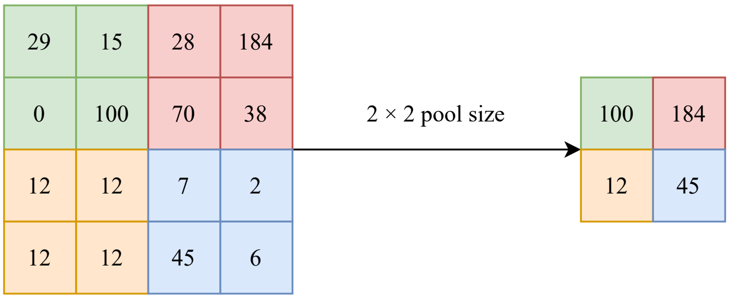
[12] Often, compute array sizes of 2×2, 4×4, or 8×8 are used. By choosing channel values that are a multiple of 16, these compute arrays will always be 100% utilized during processing.

to keep the choice of $k = 7$, $n = 3$ but increase the channel count for each layer until a desired model size.

This model size is based on the the size of the 2D convolutional neural network used in the original work for the 10+2 few-shot keyword spotting task [46]. It has a total of 134,400 parameters: however, for such a network, assuming standard 8-bit weights, more than 134 kB on-chip memory is required. It is presumed here that this thesis' design will not have more storage than 128 kB, as most have other edge hardware designs have a total (for weights, activations and biases) storage capacity ranging from 30 [40] [42] to 75 [41] to 100 [94] or maximum 130 [39] kB.

Therefore, it was chosen to tailor the model so that the parameter count is close to 110k instead, matching the size sequential Omniglot architecture (see Table 6). The channel dimensions per layer are again divisible by 16 and the number of channels is once more increased with increasing network depth, until the maximum number of parameters is reached.

3.2.2.3 Sequential Omniglot Lastly, the network architecture for the sequential Omniglot task is explained (rightmost column, Table 6). This architecture originates mainly from two points: it was designed to have the lowest receptive field size, still larger than 784 while having approximately the same parameter count as the 64-64-64-64 architecture.



29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

100	184
12	45

Figure 46: Single-channel 2×2 max-pooling operation.

This architecture, first described by Vinyals *et al.* [29] is the most commonly used network to benchmark few-shot learning on images. The reason for requiring the TCN architecture to have roughly the same parameter count as this architecture is to make sure that these two models but also approaches based on either model can be compared fairly.

The 64-64-64-64 network has four modules, each performing a 3×3 convolution with 64 output channels followed by batch normalization [81] a ReLU (see Fig. 19) function and 2×2 max-pooling operation (see Fig. 46). It has a total number of parameters equal to 111,936, compared to 116,960 in the TCN.

Finally, also in this network, all channel dimensions are divisible by 16 and the number of channels increases with increasing network depth until the maximum number of parameters is reached.

4 Algorithmic development

This section delves into the software aspects of the contributions made by this thesis. More specifically, after selecting a meta-learning algorithm and a quantization scheme, a pipeline for meta-training and quantization will be developed.

Initially, within Section 4.1, the hardware limitations that must be considered during the meta-learning algorithm development will be discussed. Section 4.2 then contains a discussion of various approaches to meta-learning, after which, in Section 4.3, a quantitative comparison is done between selected methods, thereby allowing for the final meta-learning algorithm to be chosen. Moving on, Section 4.4 outlines the quantization scheme selection. Finally, Section 4.5 lists the final performance results of the quantized models on the three benchmarks of this thesis.

4.1 Hardware-driven considerations

In this section, the main hardware-driven considerations that should be taken into account for the meta-learning algorithm selection will be outlined. These points were already briefly brought up in the description of the *On-chip friendly?* column of Table 1 and will be discussed in more depth here. First, we remind here three points that chiefly impact the relation between a meta-learning algorithm and its corresponding hardware implementation:

- memory overhead or increased activation and parameter storage compared to regular inference during meta-testing,
- processing overhead during meta-testing compared to regular inference,
- complexity of controlling the meta-testing process.

The first point is considered as increasing the storage size requirement for the activations and parameters, on top of what is required for inference, requires physically larger memories, which impacts the power and area footprints. Reducing the memory required for few-shot learning at the edge is therefore a key study point in multiple works [1] [6] [96].

The second point concerns the increased computation originating from a meta-learning method. For example, in initialization-based meta-learning methods, higher bit-precision [96] or different numerical representations [1] are required to avoid substantial accuracy loss during backpropagation, a key step in such algorithms. Next to increased computation at a tensor level through more expensive number representations, the increase can also come from requiring more on-chip iterations [1] [6]. Therefore, requiring re-training with gradient calculation and the use of extra meta-learning-related modules on top of the main network are considered strong drawbacks.

Finally, the last point is concerned with the hardware design complexity of orchestrating the meta-testing phase. Designing a controller for inference only for a specific network architecture is already not simple, however, extending this controller or building more controllers to manage gradient flow in the case of initialization-based methods or attention operations such as in *matching networks* [29] is even more costly.

In this thesis, the goal for the hardware design is to perform the inner loop (meta-testing) of a meta-learning algorithm on-chip. Therefore, none of the points concerns the meta-training phase: as this phase happens completely off-chip on commodity hardware, any overhead there does not affect the hardware design for the inner-loop/meta-testing phase. This does not mean that the complexity of the outer loop does not matter at all, as it can for example increase training time: it is simply independent from the on-chip meta-testing process.

4.2 From model-agnostic meta-learning to supervised pre-training

With the hardware-driven considerations in mind, the algorithmic development can begin. As the title of this section indicates, this section discusses recent findings in the meta-learning realm, indicating that double-derivative-based algorithms such as MAML (see Section 2.1.4.1) can be replaced by much simpler and faster algorithms, while improving performance.

First, in Section 4.2.1, the origin of the effectiveness of MAML for few-shot learning is covered. Continuing, Section 4.2.2 discusses the performance of two very simple baselines to underline the found effectiveness origin, after which, in Section 4.2.3, pre-training an embedder is considered as an alternative. Finally, Section 4.2.4 draws a conclusion from the evaluated work in this section.

4.2.1 Features or adaptation?

The MAML [15] algorithm has grown hugely popular since its release, almost becoming synonymous with meta-learning. However, the reasons for its effectiveness remained unclear [97]: is it due to the network initialization with parameters ω already containing high-quality features, or is ω trained for rapid adaptation?

Raghu *et al.* introduced this question in [97] and investigated it in an ablation study of MAML with two experiments. In the first experiment, successive layers of a CNN are frozen to prevent inner-loop adaptation (Eq. (6)). Empirically, it is shown that even when all the layers of the network (except for the classification head) are frozen, few-shot classification is barely affected [97]. Furthermore, Raghu *et al.* also demonstrate that the parameters in the network only change very little before and after inner-loop adaptation [97]. Both of these findings seem to indicate that the inner loop is not very effective: to investigate this further, the authors therefore introduce the *Almost No Inner Loop* (ANIL) algorithm (Fig. 47).

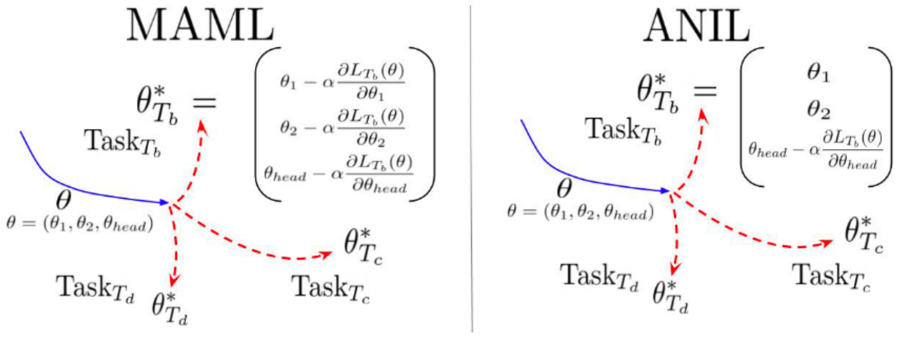

MAML
$\theta^*_{T_b} = \begin{pmatrix} \theta_1 - \alpha\frac{\partial L_{T_b}(\theta)}{\partial\theta_1} \\ \theta_2 - \alpha\frac{\partial L_{T_b}(\theta)}{\partial\theta_2} \\ \theta_{head} - \alpha\frac{\partial L_{T_b}(\theta)}{\partial\theta_{head}} \end{pmatrix}$
Task$_{T_b}$
θ
$\theta = (\theta_1, \theta_2, \theta_{head})$
$\theta^*_{T_c}$
Task$_{T_c}$
Task$_{T_d}$
$\theta^*_{T_d}$
ANIL
$\theta^*_{T_b} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_{head} - \alpha\frac{\partial L_{T_b}(\theta)}{\partial\theta_{head}} \end{pmatrix}$
Task$_{T_b}$
θ
$\theta = (\theta_1, \theta_2, \theta_{head})$
$\theta^*_{T_c}$
Task$_{T_c}$
Task$_{T_d}$
$\theta^*_{T_d}$

Figure 47: Comparison between the MAML and ANIL algorithms, where the blue line represents the outer loop and the dashed red line the inner loop. In ANIL, only the parameters of the classification head (θ_{head}) are updated in the inner loop, while all other network parameters (θ_1, θ_2) stay fixed. Figure taken from [97].

ANIL is effectively the same as MAML, except that in the inner loop, only the classification head is updated: all parameters of the CNN are frozen, where the CNN is now considered as an embedder (Fig. 48). In terms of formalization (Section 2.1.2), ω then represents all initial network parameters (CNN + classification head), while θ corresponds to the parameters of the classification head.

Benchmarking ANIL in few-shot classification and reinforcement learning tasks demonstrates that ANIL matches the performance of MAML [97]. Based on this, it can be concluded

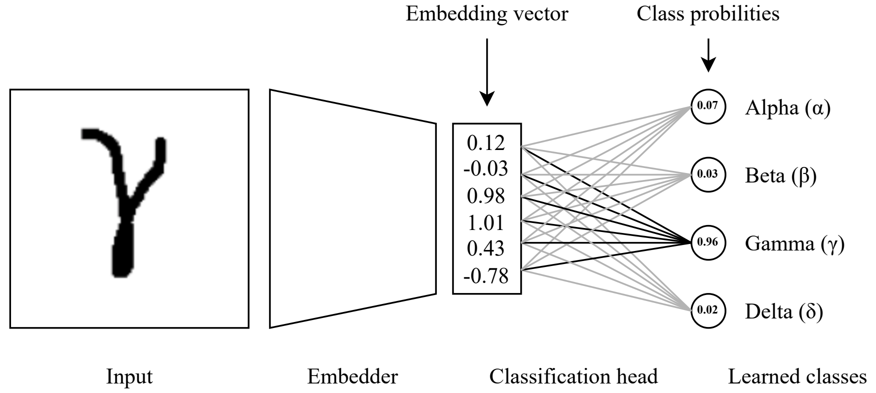


Figure 48: Schematic overview of the embedder and classification head in a neural network. The embedder can be any function or neural network (e.g., a CNN) that produces an embedding vector from an input.

that MAML seems to predominantly rely on feature reuse, with the network body (embedder) already extracting good features before performing the inner-loop update. Besides, ANIL demonstrates that these features do not benefit from inner-loop updates: they rely on outer-loop training. However, the classification head still requires inner-loop adaptation: Raghu *et al.* therefore pose that, if good features have already been learned by the embedder, what exactly is the contribution of the head?

To investigate this, they propose the *No Inner Loop* (NIL) algorithm. It works as follows: first, standard meta-training using MAML or ANIL is performed, after which prototypical-like evaluation is performed during meta-testing.[13] In two few-shot classification tasks, NIL performs similar or better compared to MAML and ANIL. This further demonstrates that the embedder learned during meta-training conditions the performance of MAML-based few-shot learning.

While the ANIL and NIL experiments focused on MAML's effectiveness, which is an initialization-based meta-learning approach (see Section 2.1.4.1), Raghu *et al.* also pose the same question for feed-forward and metric-learning methods. These methods often *jointly encode* the support set before making any predictions on the query set: consider for example relation networks in Fig. 12, matching networks in Fig. 11 and SNAIL in Fig. 6. This way, the model can learn a new task rapidly by adapting to the support samples.

Alternatively, the samples can be encoded independently, like in Prototypical Networks, shown in Fig. 10, after which a distance metric can be used to compare the encodings and perform classification. This is purely a feature reuse setting, as there is no task-specific information used [97].

Putting the performance of these two types of methods side-by-side, Raghu *et al.* demonstrate that the difference in classification accuracy on *mini*ImageNet is small (below 5%). Therefore, it is concluded that also for feed-forward and metric learning methods, feature reuse is the key ingredient for few-shot learning.

[13] Raghu *et al.* chose ANIL for meta-training and used cosine distance as a distance metric during meta-testing, not averaging support embeddings per class.

4.2.2 Simple baselines

In a seminal work by Chen *et al.* [28], comparable results were found for simple pre-training meta-learning schemes. Two methods called *Baseline* and *Baseline++* are proposed: in the former, training is performed in the same way as in [98] while in the latter an embedding is learned for each class during supervised pre-training. In both cases, during meta-testing, the embedder parameters are kept fixed. Few-shot classification is then performed by fine-tuning a linear classifier on the support embeddings (Baseline), or an embedding is learned for each class from the support embeddings (Baseline++).

Comparing the performance of the Baseline model with re-implemented versions of other meta-learning algorithms, it is shown that the Baseline method with data augmentation performs comparably to the other methods (Table 7). Previous Baseline implementations did not include data augmentation, leading to overfitting due to the low amount of samples per class (as per [99], 600 in the case of *mini*ImageNet) and were therefore never competitive.

Table 7: 5-way classification performance on *mini*ImageNet. All non-baseline methods were re-implemented by Chen *et al.* [28]. "Baseline*" indicates the results without data-augmented training. Reported numbers include 95% confidence intervals. All data taken from [28].

Method	**1-shot**	**5-shot**
Baseline [28]	42.11 ± 0.71	62.53 ± 0.69
Baseline* [28]	36.35 ± 0.64	54.50 ± 0.66
Baseline++ [28]	48.24 ± 0.75	66.43 ± 0.63
MatchingNet [29]	48.14 ± 0.78	63.48 ± 0.66
ProtoNet [27]	47.74 ± 0.84	66.68 ± 0.68
MAML [15]	46.47 ± 0.82	62.71 ± 0.71
RelationNet [30]	49.31 ± 0.85	66.60 ± 0.69

In Table 7, it is also demonstrated that the Baseline++ method performs significantly better than the Baseline method and even performs comparably to other meta-learning methods, further solidifying the hypothesis that meta-learning performance primarily relies on feature extraction. As the Baseline++ method performs well via reducing the intra-class variation during training (by forcing samples of the same class to have the same embedding), Chen *et al.* then explore the use of deeper embedders, referred to as *backbones*, which can inherently reduce intra-class variation [28].

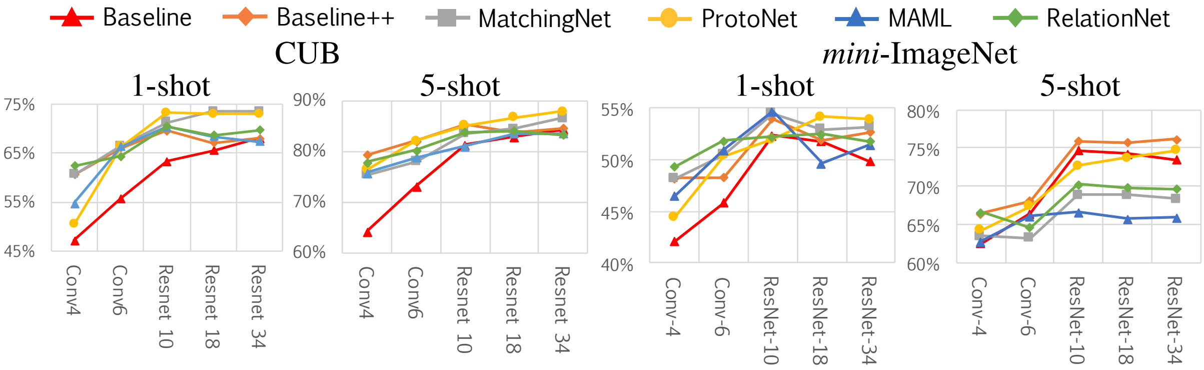


Figure 49: Few-shot classification accuracy with increasing backbone depth. Figure taken from [46].

Results on the CUB dataset[14] in Fig. 49 demonstrate that as a backbone increases its number of layers, the performance difference between different methods quickly reduces, meaning that the performance difference between different meta-learning methods would be significantly smaller if the intra-class variations were all reduced by a deeper embedder. However, for the *mini*ImageNet dataset, the results are not as clear-cut. This is likely due to the fact that there is a larger domain difference between the classes in meta-training and in meta-testing compared to CUB [28]. When meta-training on *mini*ImageNet and performing meta-testing on CUB, it is shown in Fig. 50 that the Baseline method outperforms other meta-learning methods listed with a large margin. This demonstrates that only as the domain difference grows larger, the adaptation based on a few novel class instances becomes more important with respect to solely reducing intra-class variation [28].

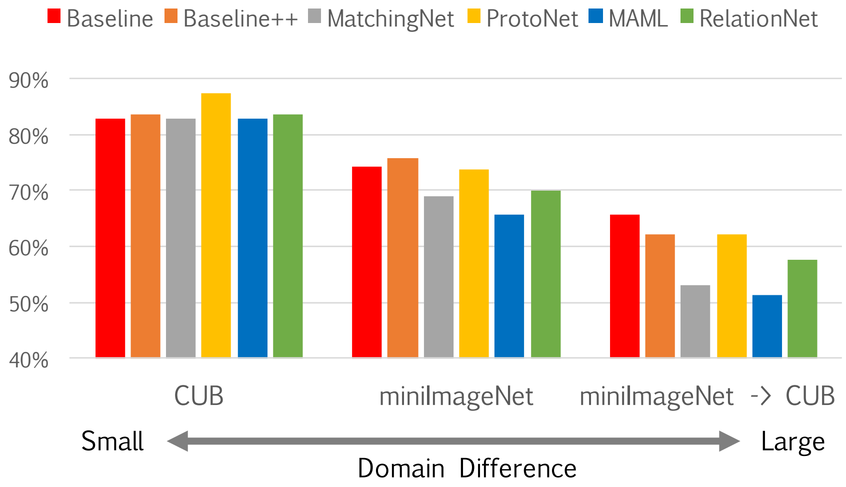


Figure 50: 5-way 5-shot classification with a ResNet-18 as embedder. Figure taken from [46].

4.2.3 Pre-training an embedder

Building on the line of work in feature-reuse and simple baselines, Yonglong *et al.* [98] propose another very simple method. First, supervised training on all classes in the meta-training set is performed (as in Baseline and Baseline++). Then, during meta-testing (see Fig. 51), a logistic regressor (LR) is learned on top of this representation from the support samples, after which query samples can be classified. This means that, similar to ANIL and NIL, the embedder is kept fixed during meta-testing.

This elegant baseline outperforms the current state of the art on four few-shot benchmarks, including *mini*ImageNet. Furthermore, since a supervised pre-training scheme is used, it is easy to apply many of the techniques used in supervised learning to improve classification performance. Therefore, Yonglong *et al.* investigate the performance of self-distillation, a form of knowledge distillation (KD) [101] and unsupervised learning.

In KD, a student model is trained not with the true class labels, but with the distribution of class probabilities output by the last layer of a pre-trained teacher model (Fig. 52). In self-distillation, both models have the same architecture and are trained for the same task. The self-distillation strategy used by Yonglong *et al.* is the *Born-Again* [102] method: at each step, the embedding model of kth generation is trained with knowledge transferred from the embedding model of $(k-1)$th generation.

[14] The CUB-200-2011 dataset (short: CUB) by Wah *et al.* [100] contains 200 classes from 11,788 images of

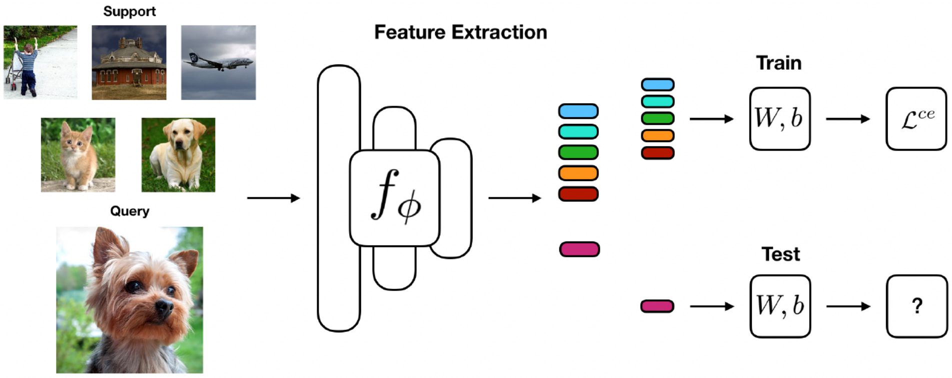


Figure 51: Meta-testing procedure for a 5-way 1-shot task. All samples are embedded in feature vectors. A logistic regressor (parameters W, b) is trained on the support embeddings, after which the query embedded is classified using the trained logistic regressor. Figure taken from [98].

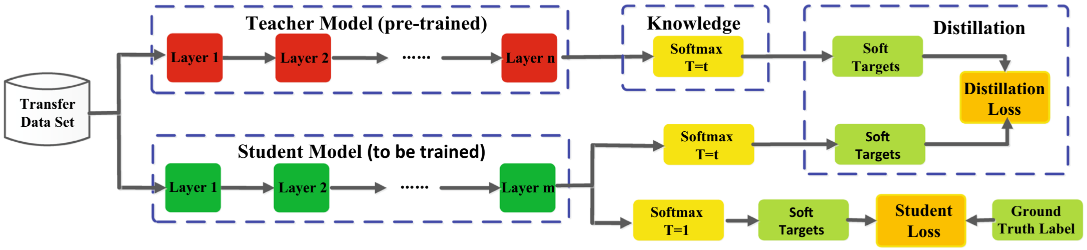


Figure 52: Schematic of knowledge distillation as proposed by [101]. Figure taken from [103].

A quantitative comparison of these pre-training methods against other meta-learning methods is shown in Table 8: the dataset used for this comparison is *mini*ImageNet (see Appendix B). The network architecture is the same for all methods in the table and is a 64-64-64-64 topology, as discussed in Section 3.2.2.3.

It can be seen that the two pre-training methods in the last two rows of Table 8 outperform most methods, except for pre-training without distillation in the 1-shot case, where SNAIL [22] performs slightly better. This further strengthens the claim from Raghu *et al.* [97] that feature-reuse is the dominant factor in the effectiveness of meta-learning methods.

Overall, self-distillation only increases few-shot *mini*ImageNet classification performance across benchmarks by 0.4% for the 64-64-64-64 architecture, but it allows for a 2-3% accuracy gain for a ResNet-12 model [98] (12.42M parameters[15]), demonstrating its effectiveness. Overall, with and without self-distillation, state-of-the-art few-shot learning performance is achieved through this simple pre-training scheme compared to other methods using the same architecture.

Performing nearest-neighbor classification with cosine distance yields equivalent performance in the 1-shot setting, both with and without distillation. Similar results using only the distance between embeddings after pre-training were obtained by Chen *et al.* [105]. Overall, these results strongly underline the strength of simple approaches based on feature extraction.

birds.

[15] Estimated by performing a parameter count with PyTorch [104] on the ResNet-12 model in this repository: `https://github.com/kjunelee/MetaOptNet/tree/master`

Table 8: Accuracy comparison between different meta-learning methods using a 64-64-64-64 architecture on *mini*ImageNet. **Bold** indicates the best performance in a column. Data taken from the respective works.

	***mini*ImageNet 5-way**	
Method	**1-shot**	**5-shot**
Matching networks [29]	46.6? ± ?.??	60.0? ± ?.??
FO-MAM [15]	48.07 ± 1.75	63.15 ± 0.91
MAML [15]	48.70 ± 1.84	63.11 ± 0.92
Prototypical networks (Eucl. distance) [27]	49.42 ± 0.78	68.20 ± 0.66
Reptile [17]	47.07 ± 0.26	62.74 ± 0.37
MAML++ [18]	52.15 ± 0.26	68.32 ± 0.44
Meta-SGD [20]	50.47 ± 1.87	64.03 ± 0.94
Meta-Curvature (1-step, MC1) [21]	53.37 ± 0.88	68.47 ± 0.69
SNAIL [22]	55.71 ± 0.99	68.88 ± 0.92
ANIL [97]	46.7? ± 0.4?	61.5? ± 0.5?
NIL [97]	48.0? ± 0.7?	62.2? ± 0.5?
LR on embedder [98]	55.25 ± 0.58	71.56 ± 0.52
Distilled LR on embedder [98]	**55.88 ± 0.59**	**71.65 ± 0.51**

4.2.4 Key takeaways

In this section, the key takeaways and findings from the discussed work are highlighted and related back to the hardware-driven considerations from Section 4.1:

- popular initialization-based meta-learning methods such as MAML [15] rely on feature-reuse for their performance [97] and the same is true for methods that jointly encode samples,
- reusing a fixed embedder from simple baselines that are trained not episodically, but instead in a supervised manner on the complete meta-training set, results in state-of-the-art performance,
- data augmentation is required for increased performance in these baselines, due to the low sample count per class [28] in the commonly used meta-learning benchmarking datasets,
- reducing intra-class variation is key in currently-defined few-shot learning problems [28],
- supervised pre-training schemes allow for simple extensions to other non-meta-learning techniques, such as distillation [98], unsupervised [98] or self-supervised learning, allowing for further expansion of performance or applications in fields with little or no labeled data.

When considering the hardware-driven considerations, it can be concluded that these simple methods using fixed embedders are not only beneficial in terms of performance compared to initialization and joint-embedding-based methods, but also in terms of the corresponding hardware design, especially in the case of distance-based evaluation. In this case, the only memory overhead comes from storing embeddings: no on-chip gradient calculations are required and classification can simply be done via a single distance computation, introducing little extra control complexity. For LR, more memory is required as the gradients of the regressor parameters have to be stored and processing overhead is induced as multiple gradient steps are required to fine-tune it.

Therefore, due to their hardware friendliness and high performance, in the following quantitative meta-learning method comparison, only the performance of distance-based classification methods will be evaluated on the two few-shot learning tasks of this thesis. This includes ProtoNets [27], NIL (trained via MAML) [97] and supervised pre-training with classification from distances between embeddings [98]. While distillation and unsupervised learning for the pre-training phase could have also been in the mix, they are left for future work as they mostly provide an additional orthogonal direction that can be investigated in combination with any of the aforementioned techniques.

4.3 Meta-learning algorithm choice

In this section, the four mentioned distance-based meta-learning methods will be compared quantitatively. First, in Section 4.3.1, the four remaining meta-learning algorithms will be compared quantitatively, after which Section 4.3.2 explores the possibilities for distance metrics that can be used for embedding comparison. Finally, in Section 4.3.3, based on information presented in this section, a meta-learning algorithm is chosen for implementation in hardware.

4.3.1 Quantitative meta-learning method comparison

This section covers the quantitative meta-learning method comparison between the remaining distance-based meta-learning methods. Section 4.3.1.1 discusses the (sequential) Omniglot performance, while Section 4.3.1.2 discusses the few-shot keyword spotting performance.

4.3.1.1 Sequential Omniglot We report in Table 9 the performance on Omniglot (sequential Omniglot in the case of the TCN). The TCN architecture follows Section 3.2.2.3, while the performance of the 64-64-64-64 architecture is included for reference. For clarity, pre-training an embedder is abbreviated as PTE in Table 9.

Table 9: Comparison of meta-learning approaches on Omniglot, using both the TCN and 64-64-64-64 architecture. Both prototypical networks were trained in 60-way, 5-query shot setup. The TCN with NIL was trained in a 20-way setup. All NIL training was done using MAML. Accuracy with 95% confidence intervals is shown.

		5-way		**20-way**	
Network	**Approach**	**1-shot**	**5-shot**	**1-shot**	**5-shot**
TCN	NIL	92.60 ± 0.31	96.98 ± 0.20	79.61 ± 0.24	89.80 ± 0.18
TCN	ProtoNet	98.10 ± 0.52	99.60 ± 0.39	92.90 ± 0.48	98.02 ± 0.40
TCN	PTE (Eucl. w/ aug.)	97.94 ± 0.15	99.57 ± 0.07	93.62 ± 0.13	98.68 ± 0.06
TCN	PTE (cos.) w/ aug.	98.21 ± 0.16	99.66 ± 0.06	94.46 ± 0.12	98.75 ± 0.06
TCN	PTE (Eucl.)	96.09 ± 0.19	-	-	-
64-64-64-64	NIL [97]	-	-	96.7 ± 0.3	98.0 ± 0.04
64-64-64-64	ProtoNet [27]	98.8	99.7	96.0	98.9
64-64-64-64	PTE (cos.) w/ aug.	97.56 ± 0.16	99.48 ± 0.07	92.51 ± 0.15	98.07 ± 0.07
64-64-64-64	PTE (Eucl.)	95.38 ± 0.23	-	-	-
64-64-64-64	MAML [15]	98.7 ± 0.4	99.9 ± 0.1	95.8 ± 0.3	98.9 ± 0.2

To combat overfitting on the meta-training set with PTE (as described in Section 4.2.2), data augmentation was performed on the meta-training data. For this, the `RandomAffine`[16] transformation from PyTorch [104] is used: rotations are disabled but images are scaled with a factor between 0.8 - 1.2 and are translated by a maximum of 15% in both the horizontal and vertical directions. All PTE training was performed with the same hyperparameters to avoid excessive over-optimization.[17]

From Table 9, it can be seen that NIL performs significantly worse than all other methods, which might however originate from a suboptimal inner-loop learning rate selection as only one NIL experiment was performed for this specific model-dataset configuration. Furthermore, it can be seen that augmentations lead to a non-negligible increase in accuracy of 2% for the 5-way 1-shot case. For TCNs, ProtoNet and PTE using cosine similarity with data augmentation perform similarly for the 5-way cases, while for the 20-way case, PTE outperforms the ProtoNet approach.

4.3.1.2 Few-shot keyword spotting For the few-shot keyword spotting task, the **command** classification task[18] defined by Chen *et al.* [46] is used. While also a digit classification task is defined in this work (see Section 2.1.6.2), the command classification task was chosen as it was empirically found to be harder [46].

Before meta-training, the dataset was balanced as its classes have a large variety in the number of samples, as shown in Fig. 65. Note that, due to the low number of overall classes (32), it is not possible to define a separate meta-validation set. Therefore, the validation split from classes in the meta-train set is used as the meta-validation set. To combat overfitting for PTE on the meta-train dataset, the following augmentations[19] are applied:

- time shifts in the range $[-100, 100]$ ms (as done in [37] [45] [108]),
- background noise with a probability of occurrence of 0.15 with a uniformly sampled scale in $[0, 1]$.

After these augmentations, the audio data is converted to MFCC[20] samples with a window size of 32 ms and a stride of 16 ms (see Section 3.2.2.1). All PTE training was performed with the same hyperparameters to avoid excessive over-optimization.[21] Table 10 then shows the few-shot command classification performance, with the last three rows from the original work by Chen *et al.* included to provide a baseline performance across the shot range.

Table 10 demonstrates that the PTE approach using the TCN architecture and data augmentation outperforms the original approach (fixed-position MAML [46]) by a large margin. Comparing the TCN against the 64-64-64-64 architecture, with both using the Euclidean distance

[16] `https://pytorch.org/vision/main/generated/torchvision.transforms.RandomAffine.html`

[17] The Adam [106] optimizer was used with a learning rate of 0.0005 for a total of 70 epochs, while after 50 epochs the learning rate is multiplied by 0.1

[18] Data splits taken from `https://github.com/Codelegant92/STC-MAML-PyTorch/blob/master/data/`

[19] While the SpecAugment transformations from Park *et al.* [107] are sometimes also applied [37] [108], they were not used in this work as the PyTorch [104] modules related to SpecAugment do not support multiple masks as used in these works.

[20] The last stage of the MFCC algorithm, a discrete cosine transform (DCT), is skipped as it was empirically found (see Appendix C) that after QAT, the performance of quantized models trained on GSC with DCT in the MFCC pipeline is significantly lower than the ones without DCT (-6%).

[21] The Adam [106] optimizer was used with a learning rate of 0.0005 for a total of 70 epochs, while after 50 epochs the learning rate is multiplied by 0.1

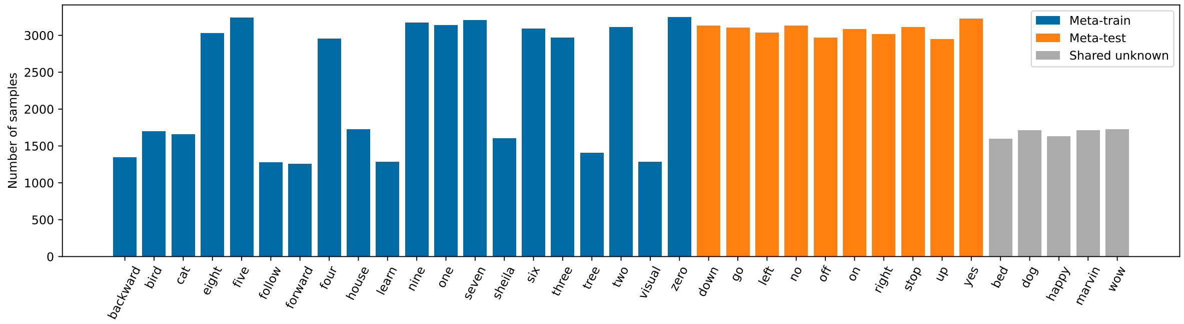


Figure 53: Class wise sample distribution for the command classification few-shot learning task as defined by Chen *et al.* [46]. The silence class is not shown. Shared unknown indicates the classes combined into an unknown class, which are the same classes for meta-training and meta-testing.

Table 10: Comparison of meta-learning approaches on 10+2-way few-shot command classification, using both the TCN and 64-64-64-64 architecture. The prototypical network was trained in a 22-way, 5-query shot setup with Adam and a learning rate of 0.001. The TCN with NIL was trained in a 20-way setup. All NIL training was done using MAML. Accuracy with 95% confidence intervals is shown.

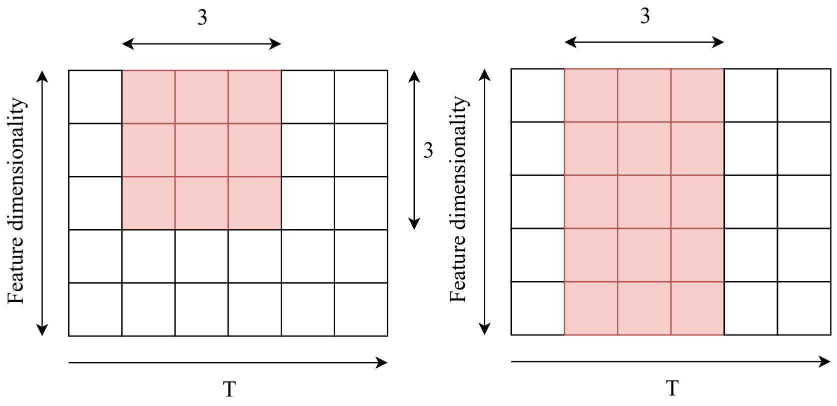
Network	Approach	1-shot	5-shot	10-shot
TCN	NIL w/aug	35.82 ± 0.27	49.67 ± 0.33	53.68 ± 0.30
TCN	ProtoNet w/ aug	10.92 ± 0.53	8.50 ± 0.16	8.33 ± 0.00
TCN	PTE (Eucl.) w/ aug	44.38 ± 0.31	66.44 ± 0.31	70.98 ± 0.31
TCN	PTE (cos.) w/ aug	49.08 ± 0.32	67.90 ± 0.33	72.34 ± 0.29
TCN	PTE (Eucl.)	41.54 ± 0.29	63.34 ± 0.34	68.80 ± 0.31
64-64-64-64	PTE (Eucl.) w/ aug	27.24 ± 1.15	37.27 ± 1.44	43.36 ± 1.52
64-64-64-64	Training on support set [46]	17.03 ± 0.48	22.42 ± 0.33	25.6 ± 0.26
64-64-64-64	MAML [46]	33.35 ± 0.80	50.31 ± 0.50	57.34 ± 0.41
64-64-64-64	Fixed-position MAML [46]	39.54 ± 0.62	52.20 ± 0.51	59.36 ± 0.39

metric and data augmentation, it can be seen that the TCN architecture offers a significant performance benefit compared to the baseline 2D convolutional architecture: this can be attributed to the fact that a 2D convolution on a sequence only convolves across feature-wise sections (Fig. 54a), while a 1D convolution convolves across the entire feature-space (Fig. 54b).

For this benchmark, ProtoNets perform significantly worse than any of the other approaches, reducing to random-guessing performance in the 10-shot case. Furthermore, NIL with a TCN can be seen performing close to baseline MAML performance. Finally, similarly to Section 4.3.1.1, cosine distance outperforms Euclidean distance across all shots, while the application of data augmentation also measurably increases performance.

4.3.2 Embedding distance metric comparison

As the chosen set of meta-learning setups relies on distance calculation between embeddings, a careful evaluation of the different distance metrics available should be performed. While normally, in the context of few-shot learning, only Euclidean (Eq. (46)) and cosine distance (Eq. (47)) are considered, for completeness, Manhattan distance (Eq. (44)) and the dot-product (Eq. (45)) are also included.



(a) 2D convolution with filter size 3×3. **(b)** 1D convolution with filter size 3.

Figure 54: Comparison of data aggregation between 2D and 1D convolutions over a sequence with length T.

$$0 \leq \text{Manhattan distance}\left(\mathbf{a},\mathbf{b}\right) = \|\mathbf{a}-\mathbf{b}\|_1 = \sum_{i=1}^{n} |a_i - b_i| < \infty \tag{44}$$

$$\text{dot product}\left(\mathbf{a},\mathbf{b}\right) = \mathbf{a}\cdot\mathbf{b} = \sum_{i=1}^{n} a_i b_i \tag{45}$$

$$0 \leq \text{Euclidean distance}\left(\mathbf{a},\mathbf{b}\right) = \|\mathbf{a}-\mathbf{b}\|_2 = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2} < \infty \tag{46}$$

$$0 \leq \text{cosine distance}\left(\mathbf{a},\mathbf{b}\right) = 1 - \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|_2\|\mathbf{b}\|_2} = 1 - \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2 \cdot \sum_{i=1}^{n} b_i^2}} \leq 2 \tag{47}$$

$$\text{s.t.} \quad \dim\left(\mathbf{a}\right) = \dim\left(\mathbf{b}\right) = n \tag{48}$$

In Eq. (47), the teal-highlighted part is the definition of cosine **similarity**. The linear transformation to obtain the cosine **distance** is required to transform cosine similarity into a true distance metric, since by definition, distance is always larger than or equal to zero (therefore, the dot product is also not defined as a distance metric in Eq. (45)). A visualization of the Manhattan, Euclidean and cosine distance metrics in 2D-space ($n = 2$) is shown in Fig. 55.

First, the three metrics are compared on a computational level, considering hardware implementation. It can immediately be seen that both Eq. (46) and Eq. (47) contain a square root in their definition, an operation that is not trivial to implement in hardware. Luckily, in the case of the Euclidean distance, its definition can be squared without changing the meaning of the distance metric. However, squaring Eq. (47) would not remove the $\sqrt{}$ operator. Therefore, instead of considering the $\arg\min$ of the cosine distance for hardware implementation, the $\arg\max$ of the squared cosine similarity will be used. This will yield the same classification results, even though it is not a valid distance metric anymore. Taking the above into account, Table 11 displays the number of primitive operations per (revised) distance metric.

Computationally speaking, Manhattan distance is the lightest, while cosine similarity is the heaviest. Furthermore, cosine distance is the only distance metric containing a division due to normalization (Eq. (47)). The dot-product is the lightest among the metrics that require multiplication, as for example the cosine distance essentially uses three dot products in its definition.

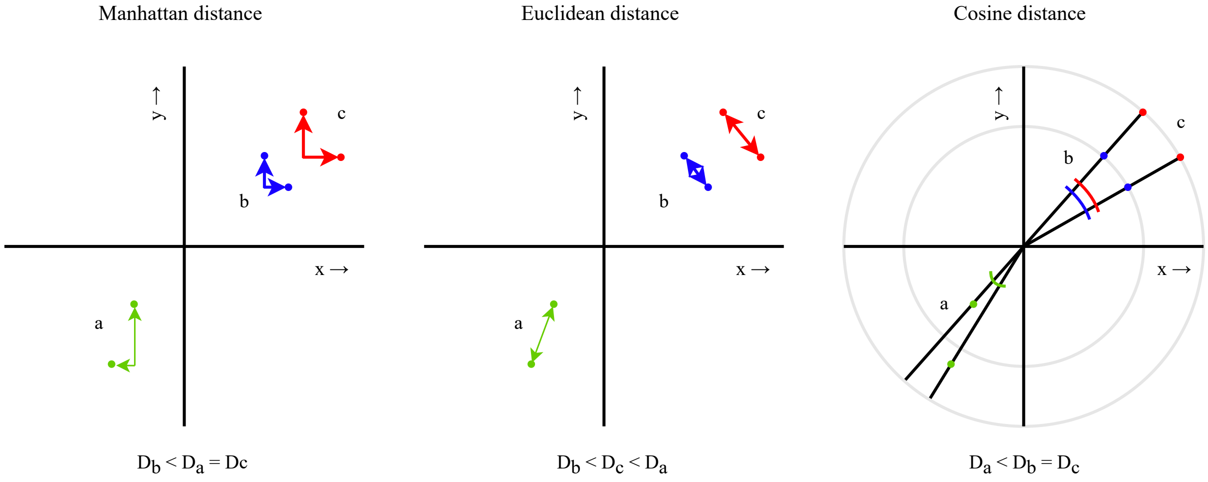


Figure 55: Comparison of Manhattan, Euclidean and cosine distance. Each colored set of points represents one 2D support and one 2D query embedding. D_x indicates the distance between these two points.

Table 11: Number of primitive operations per (revised) distance metric. **Bold** indicates the best value per row.

	Manhattan distance	**Dot product**	**(Euclidean distance)^2**	**(Cosine similarity)^2**
# of additions	$2n-1$	$\mathbf{n-1}$	$2n-1$	$3n-3$
# of multiplications	**0**	n	n	$1+3n+1$
# of divisions	**0**	**0**	**0**	1

Next, Table 12 contains the few-shot learning performance on sequential Omniglot trained using PTE, where for multi-shot evaluation, the embeddings were averaged. Cosine distance is the clear winner here, providing the best performance in all cases. Manhattan distance performs the second best, which is impressive considering that it is by far the simplest metric.

Table 12: Comparison of distance metrics on Omniglot, using the Omniglot TCN architecture meta-trained via PTE. For a k-shot setup where $k > 1$, the support embeddings are averaged. Accuracy with 95% confidence intervals is shown. **Bold** indicates the best value per column.

	5-way		**20-way**	
Metric	**1-shot**	**5-shot**	**1-shot**	**5-shot**
Manhattan distance	97.85 ± 0.16	**99.67 ± 0.06**	93.64 ± 0.13	**98.74 ± 0.06**
(Euclidean distance)^2	97.94 ± 0.15	99.57 ± 0.07	93.62 ± 0.13	98.68 ± 0.06
(Cosine similarity)^2	**98.21 ± 0.16**	**99.66 ± 0.06**	**94.46 ± 0.12**	**98.75 ± 0.06**
Dot product	97.74 ± 0.16	99.53 ± 0.07	93.58 ± 0.13	98.57 ± 0.07

Notably, Snell *et al.* [27] found that squared Euclidean distance actually outperformed cosine similarity (Manhattan distance was not considered) on 5-way *mini*ImageNet by an average 2-% accuracy advantage in the 1-shot case and by up to 16.7% in the 5-shot case, which led to the selection of the Euclidean distance in the ProtoNet implementation. It is assumed that this is due to the fact that cosine distance is not a Bregman divergence (same for Manhattan distance), while the squared Euclidean distance (Mahalanobis distance) is [27]. For Bregman divergences [109], it has been demonstrated that the cluster representative resulting in the smallest distance to its assigned support points is the cluster mean: averaging the support embeddings thus yields optimal cluster representatives when a Bregman divergence is used [27].

Finally, it is noted that there is an equivalent view with a linear layer for two of the metrics. Namely, the dot product can be reinterpreted as a linear layer without bias, where the weights of the layer are the averaged support embedding. Also, the Euclidean distance can be reinterpreted as a linear layer [110], as also noted by the ProtoNet [27] authors. The distance between an averaged embedding or prototype $\boldsymbol{s}^j/k$ in a k-shot setup, where $\boldsymbol{s}^j$ is the sum of all the k-shot embeddings for class j and an unseen incoming embedding $\boldsymbol{x}$ is

$$\begin{aligned}
\text{Euclidean distance}^2\left(\frac{\boldsymbol{s}^j}{k},\mathbf{x}\right) = D_j^2 &= \sum_{i=1}^{n}\left(\left(\frac{s_i^j}{k}\right) - x_i\right)^2 \\
k^2 D_j^2 &= \sum_{i=1}^{n}\left(s_i^j - kx_i\right)^2 \\
D_j^2 &\propto \sum_{i=1}^{n}\left(s_i^j - kx_i\right)^2 \\
D_j^2 &\propto \sum_{i=1}^{n}\left(s_i^{j2} + k^2x_i^2 - 2ks_i^jx_i\right).
\end{aligned} \tag{49}$$

However, as the term $\sum_{i=1}^{n} k^2x_i^2$ is the same for all other distances between $\boldsymbol{x}$ and prototypes, as it only depends on $\boldsymbol{x}$, it can be removed as it does not change the definition with respect to $\propto$:

$$\begin{aligned}
D_j^2 &\propto \sum_{i=1}^{n}\left(s_i^{j2} - 2ks_i^jx_i\right) \\
D_j^2 &\propto \sum_{i=1}^{n} s_i^{j2} - 2k\sum_{i=1}^{n} s_i^jx_i.
\end{aligned} \tag{50}$$

Then, equivalently:

$$\begin{aligned}
&D_j^2 \propto b_j + \boldsymbol{W}_j \cdot \boldsymbol{x} \\
\text{s.t.}\quad &b_j = \sum_{i=1}^{n} s_i^{j2}, \quad \boldsymbol{W}_j = -2k\boldsymbol{s}^j
\end{aligned} \tag{51}$$

From Eq. (51), it is clear that with the bias b_j and weight $\boldsymbol{W}_j$, a linear layer for output neuron j is formed.

4.3.3 Final choice

To summarize, for sequential Omniglot (Section 4.3.1.1), it was found that PTE with cosine distance and data augmentation performs best for all cases, closely trailed by ProtoNets for the 5-way scenarios. NIL lagged significantly behind the other approaches, while it was shown that, for PTE, the use of data augmentation increases performance in all scenarios.

Considering the few-shot keyword spotting task (Section 4.3.1.2), with a smaller meta-training class count than sequential Omniglot but with more samples per class, it was again demonstrated that using data augmentation with PTE increases classification accuracy for all shots and that combining this with cosine distance gives the best overall performance. ProtoNets were non-performant for this benchmark, while NIL with data augmentation was able to match the performance of MAML.

Reviewing the distance metric evaluation, it was shown that the Manhattan distance is, computationally speaking, the simplest metric. It overall achieves a balanced performance that is on par with Euclidean distance and is only slightly outperformed in the 1-shot case by cosine similarity, which is the most computationally-heavy distance metric.

Based on these results, PTE with Manhattan distance under data augmentation is selected following a three-fold rationale. First, PTE demonstrates high and predictable performance across the two benchmarks. Second, it also allows, through its simple definition, for future extension with other methods such as distillation. Finally, the use of Manhattan distance allows for low-cost on-chip learning with no tangible accuracy penalty.

4.4 Quantitative quantization scheme comparison

In this section, various quantization schemes will be compared, after which one will selected for hardware implementation. Before that, all common points between the tested schemes are outlined as follows.

- All biases are quantized using the combined weight and activation scale, as it only requires a single scaling operation (Section 2.3.2.1), using a 16-bit symmetric signed quantizer.
- It was opted to use asymmetric activation quantization and symmetric weight quantization, as when both quantizers are asymmetric, computational overhead is introduced via an extra term (Section 2.3.2.4).
- Unsigned activation quantization is used, as the TCN architecture applies ReLU (see Fig. 19) after every convolution operation, yielding activations that are strictly positive in the range $[0,\infty)$.
- Signed weight quantization is used to accommodate negative and positive weights.
- Only *power-of-two* scale values are used to reduce hardware complexity (Section 2.3.2.3).
- QAT is performed for all experiments on all schemes to ensure maximum performance, also for low-bit-width quantization schemes (Section 2.3.3).
- The residual layers in the TCN are handled as per Section 2.3.3.2.
- All batch-normalization operations are folded prior to training (Section 2.3.3.3).

The considered number representations for these experiments are the uniform representation for both weights and activations and the logarithmic representation for weights (see Section 2.3.4 for more details for each format). Notably, logarithmic activation quantization was not considered as a linear-to-logarithmic conversion is required after accumulation in the case of uniform weights, while for logarithmic weights a logarithmic-to-linear conversion is required before accumulation [86] or accumulation has to be performed in the log-domain [86] via *look-up tables* [111] [112] [113], increasing hardware design and software simulation complexity.

Further, orthogonal to the original proposition of logarithmic quantization [86] but similar to the work of Przewlocka-Rus *et al.* [114], the implemented logarithmic quantizer has a learnable full-scale range (FSR), which removes the hyperparameter tuning for the scale per layer as done in the original work. All quantization-related code was implemented using Brevitas [115].

In Section 4.4.1 the quantization experiments for sequential Omniglot are discussed, after which Section 4.4.2 discusses the results from the quantization experiments for 12-class KWS. Finally, in Section 4.4.3, a conclusion is drawn from this data and a quantization scheme for hardware implementation is selected.

4.4.1 Sequential Omniglot

For the first set of quantization experiments, it was chosen to use the sequential Omniglot task. There are two key reasons for choosing this benchmark:

- since the main focus of this work is few-shot learning, the performance impact of quantization on few-shot learning is critical,
- from the two few-shot learning benchmarks in this thesis, Omniglot is the only one that has published work regarding quantization [1] [6], making comparisons easier.

With this in mind, Table 13 shows the performance on 5-way 1-shot sequential Omniglot using the architecture proposed in Section 3.2.2.3 for various combinations of logarithmic and uniform quantization. QAT for all networks in this table was performed with the same hyperparameters to avoid excessive over-optimization.[22]

Table 13: Comparison of test accuracy (with 95% confidence intervals) on 5-way 1-shot sequential Omniglot for various combinations of quantizers. U*x* means unsigned uniform quantization with *x* bits, while S*x* and L*x* indicate the same for a signed uniform quantizer and logarithmic quantizer respectively. C stands for per-channel scaling while T stands for per-tensor scaling. Some experiments were not performed due to unsatisfactory performance by experiments with larger bit widths in the same row or column, as indicated by arrows and ×.

Activation	Weight quantization								
	Linear				Logarithmic				
quantization	S8	S6	S4	S3	L8	L6	L5	L4	L3
U8	T: 97.99 ± 0.16 C: 97.68 ± 0.18	97.53 ± 0.17	96.15 ± 0.21	87.55 ± 0.33	T: 97.39 ± 0.17 C: 95.40 ± 0.22	97.13 ± 0.18	97.02 ± 0.19	97.21 ± 0.17	20.00 ± 0.00
U4	97.46 ± 0.18	63.85 ± 0.51	T: 92.10 ± 0.27 C: 91.96 ± 0.29	20.00 ± 0.00	96.33 ± 0.19	96.79 ± 0.18	97.12 ± 0.19	T: 96.57 ± 0.21 C: 96.72 ± 0.19	↓
U3	95.81 ± 0.23	91.29 ± 0.29	89.16 ± 0.31	×	20.00 ± 0.00	→	→	→	×

Table 13 shows that for all linear weight quantization schemes, the accuracy steadily decreases with lower activation quantization. Also, for iso activation bit widths, the accuracy decreases expectedly with lower weight bit width. Overall, the best combination is S8 / U8. It can also be seen that the combination S6-U4 has an unusually low accuracy: for two different learning rates in this configuration, the accuracy fell from $\approx 97\%$ to a 20-% (random guessing) validation accuracy after, which the performance increased again.

For the logarithmic weight quantizers in Table 13, the accuracy at iso activation bit widths is very similar: this could indicate that the logarithmic quantizer does not use the extra levels of quantization that are available for higher weight bit widths. Indeed, the average percentage of available quantization levels used for 8- and 4-bit logarithmic weights (Fig. 56 versus Fig. 57) under 8-bit activations is 19.4% and 91.4% respectively. Furthermore, while at 8-bit weights, linear weight quantization outperforms logarithmic quantization, at 4-bit weights, the situation is reversed: in the case where 4-bit activations are used, logarithmic quantization performs 4% better.

[22] The Adam [106] optimizer was used with a learning rate of 0.001 and a weight decay of $7e-4$ for a total of 200 epochs. Training started from a pre-trained FP32 model on this task, wherein all dropout layers were disabled prior to QAT. The Omniglot data setup is the same as in Section 4.3.1.

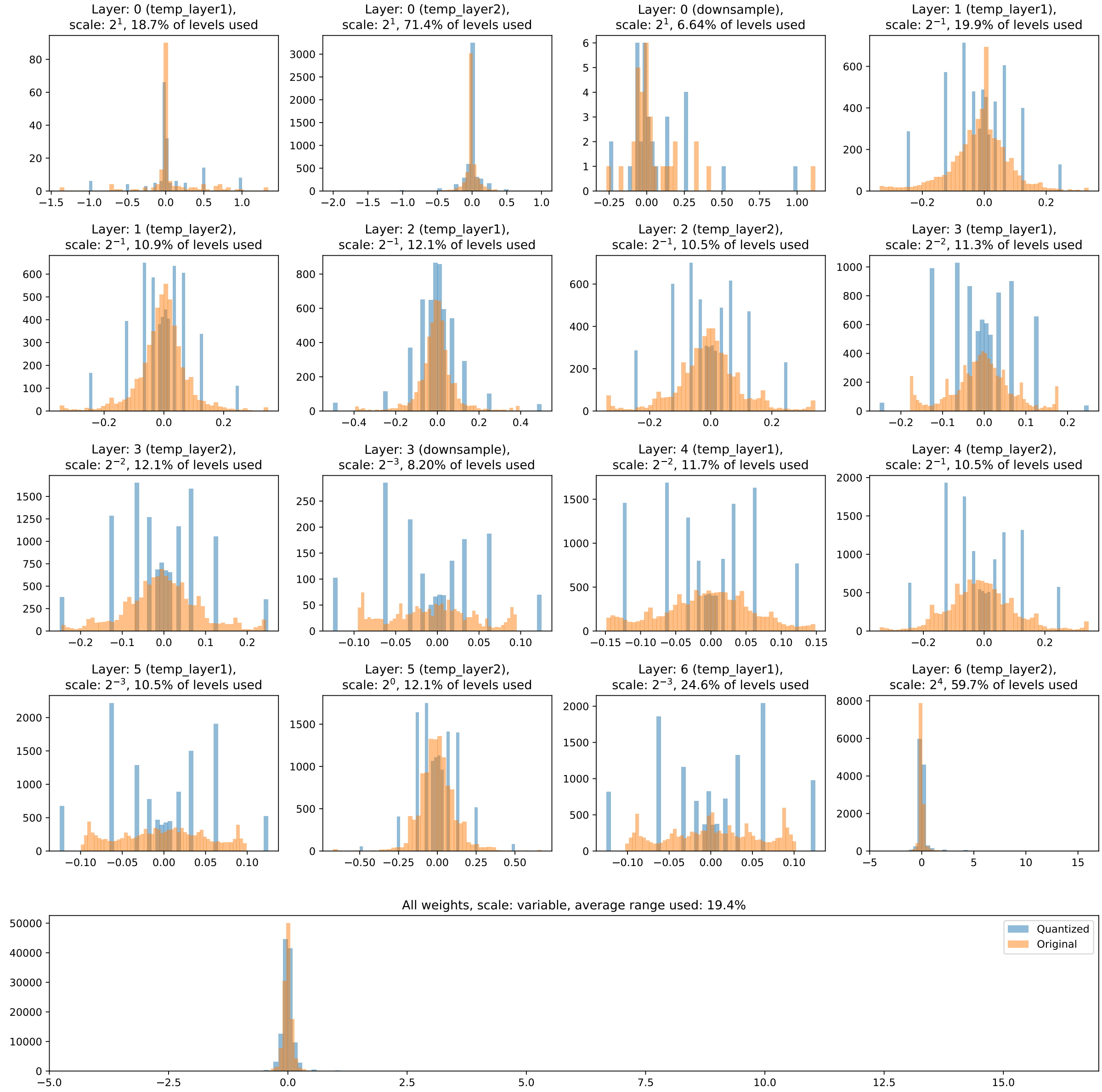


Figure 56: Histogram of quantized and original weights for each tensor in the sequential Omniglot TCN for 8-bit logarithmic weights and 8-bit unsigned activation quantization.

Comparing per-channel (C) and per-tensor (T) quantization in Table 13 for both types of weight quantization shows that per-tensor (T) quantization always performs better, except in the L4 / U4 scheme, where the difference is not significant. Furthermore, 3-bit activation quantization does not work with logarithmic quantizers (see L8 / U3 experiment, where accuracy is reduced to random guessing), while it does maintain accuracy under linear weight quantization. A similar situation occurs for 3-bit-weight quantization, where the linear quantization scheme maintains adequate accuracy but the logarithmic scheme reduces to random guessing performance.

Comparing with other works that performed quantized Omniglot (Table 14), it can be seen that, regardless of the quantization approach, the PTE strategy performs similarly for 5-way 1-shot compared with the AIQ[23] strategy [6], which requires on-chip gradient descent.

[23] Acronym stands for adaptation at inference quantization, where the MAML [15] inner-loop is executed in

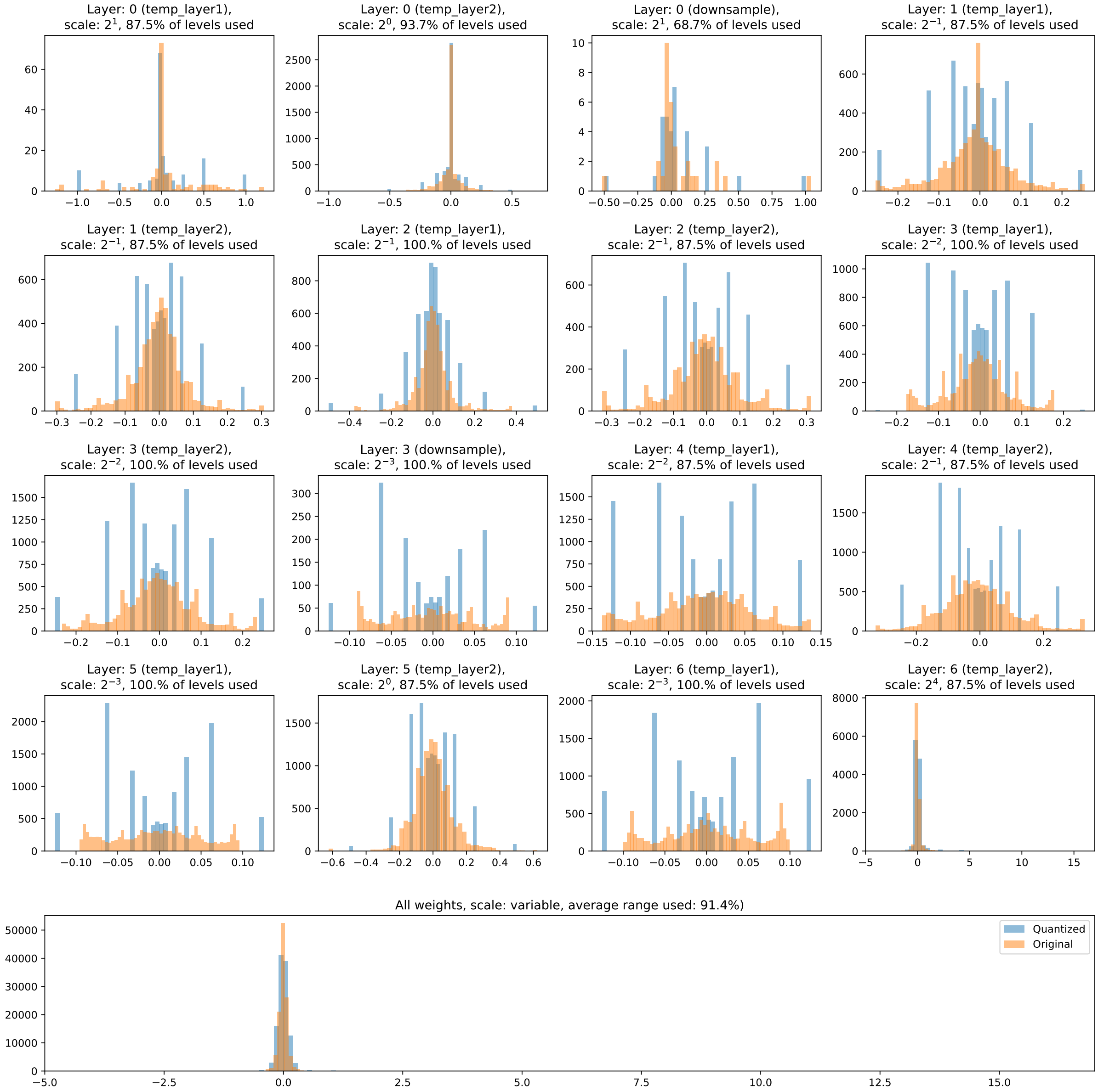


Figure 57: Histogram of quantized and original weights for each tensor in the sequential Omniglot TCN for 4-bit logarithmic weights and 8-bit unsigned activation quantization.

Table 14: Accuracy (%) comparison of quantized meta-learning approaches on Omniglot, both using the 64-64-64-64 architecture. The quantization scheme is noted as the format name, followed by *weight bit width – activation bit width – gradient bit width – error bit width*. Data taken from the respective works.

			5-way		20-way	
Network	**Method**	**Quantization**	**1-shot**	**5-shot**	**1-shot**	**5-shot**
64-64-64-64	AIQ [6]	S4-4-8-4	97	99.2	-	95.5
64-64-64-64	qL2L [1]	BFP[24]4-4-4-4	75.18	91.15	46.38	69.62

hardware at inference-level bit widths [6].

[24] Block floating-point [116].

4.4.2 12-class KWS

For the second set of quantization experiments, the 12-class KWS task was chosen as the selected network is less deep and has fewer parameters than the sequential Omniglot model and thus should be more sensitive to quantization noise. For training the FP32 model, the training data is augmented using the same approach as in Section 4.3.1.2, except during QAT. Also, the training dataset was class-wise balanced, as the original dataset has a large class imbalance (Fig. 58). Finally, the same MFCC approach as in Section 4.3.1.2 is used for all data splits.

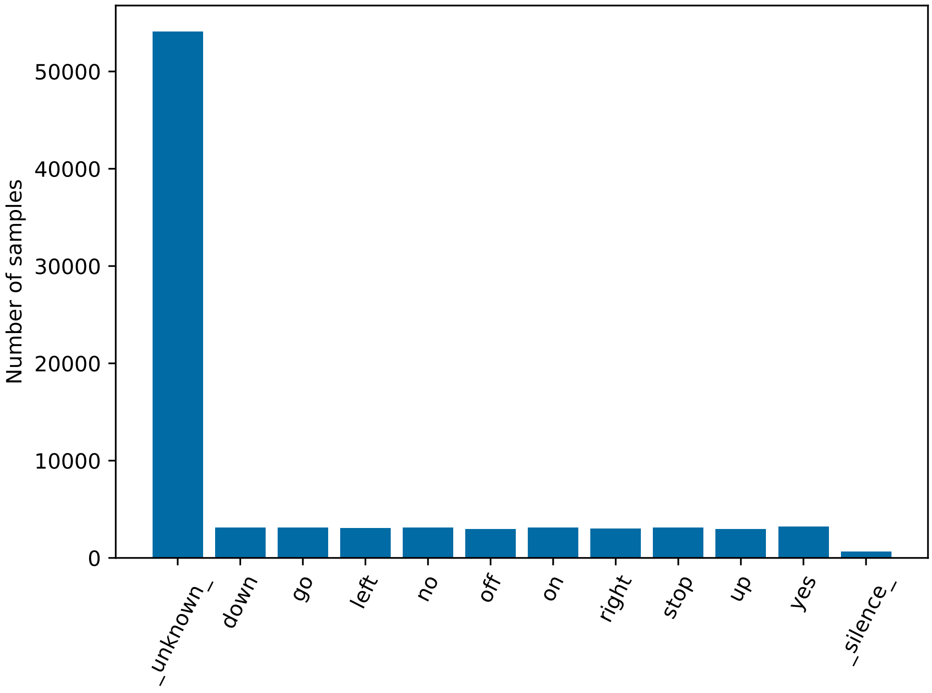


Figure 58: Class sample distribution for the 12-class KWS task.

Similar to Section 4.4.1, QAT for all networks in this table was performed with the same hyperparameters to avoid excessive over-optimization.[25] Furthermore, due to the unsatisfactory performance of 3-bit activations and weights and of per-channel quantization, these quantization combinations will be skipped for the 12-class KWS task. The results using the remaining combinations can be found in Table 15. Note that the final accuracy of the FP32 model is 94.76%.

Table 15: Comparison of test accuracy on 10+2-way keyword spotting using the GSC dataset for various combinations of quantizers. Ux means unsigned uniform quantization using x bits while Sx and Lx indicate the same for a signed uniform quantizer and logarithmic quantizer, respectively.

	Weight quantization						
Activation	**Linear**			**Logarithmic**			
quantization	**S8**	**S6**	**S4**	**L8**	**L6**	**L5**	**L4**
U8	90.86	92.35	92.73	92.33	92.51	92.76	92.61
U4	92.85	92.78	92.01	92.80	93.09	93.44	93.45

[25] The Adam [106] optimizer was used with a learning rate of 0.002 and a weight decay of $5e-4$ for a total of 200 epochs. Training started from a pre-trained FP32 model on this task, wherein all dropout layers were disabled prior to QAT.

Table 15 shows much less of a clear trend compared to Table 13. Across the board, the combinations with 4-bit activations perform better than the ones with 8-bit activations. Furthermore, for the same weight and activation bit width, the logarithmic quantizers perform generally better: decreasing the number of bits available for the weights actually increases performance for the logarithmic weight quantizers. This is not the case for the linear quantizers. The best-performing combination is then the L4 / U4 combination, while the worst-performing combination is U8 / S8.

4.4.3 Quantization scheme choice

Summarizing, for sequential Omniglot (Section 4.4.1), it was found that S8 / U8 quantization resulted in the highest accuracy, while 3-bit weight/activation quantization severely degraded the performance. Furthermore, per-channel quantization was shown not to be effective. It was also demonstrated that, for the same activation bit width, the bit width of logarithmic weights can be reduced from 8 to 4 without incurring a (significant) accuracy penalty. Finally, under low-bit-width weights and activations, logarithmic weights performed better than linear weights.

Considering the 12-class KWS task with a smaller neural network (Section 4.4.2), it was shown that logarithmic quantization performed better than linear quantization, while the use of a lower activation bit width improved performance.

Based on these results, a logarithmic weight quantization and unsigned activation quantization of both 4-bits is selected. This scheme allows for (i) using both smaller weight and activation memories at the same network size, (ii) removing the need for multiplier hardware and (iii) only incurring a small (1.4-%) accuracy penalty for the few-shot learning scenario, while performing the best in the 12-class KWS task.

4.5 Final performance

In this section, the final (quantized) performance on the three benchmarks of this thesis is reported. For the 12-class KWS task, the FP32 accuracy is 94.76%, while the quantized performance is 93.45% (-1.31%). For few-shot keyword spotting, the accuracies are reported in Table 16, while Table 17 shows the results for sequential Omniglot.

Table 16 shows that for the few-shot KWS task, the quantized model loses about 3% accuracy for each of the number of shots, while in Table 17, the effect of quantization is really only visible in the 1-shot cases. Overall, these results demonstrate the possibility of using for low bit-width weights and activations for meta-learning at the edge.

Table 16: Final quantized performance on 10+2-way few-shot command classification. Accuracy with 95% confidence intervals is shown.

Quantization	1-shot	5-shot	10-shot
FP32 / FP32 / FP32	44.38 ± 0.31	66.44 ± 0.31	70.98 ± 0.31
L4 / U4 / S16	41.46 ± 0.31	62.48 ± 0.32	67.31 ± 0.31

Table 17: Final quantized performance on Omniglot. Accuracy with 95% confidence intervals is shown.

	5-way		20-way	
Quantization	1-shot	5-shot	1-shot	5-shot
FP32 / FP32 / FP32	97.94 ± 0.15	99.57 ± 0.07	93.62 ± 0.13	98.68 ± 0.06
L4 / U4 / S16	96.57 ± 0.21	99.34 ± 0.09	90.54 ± 0.17	97.94 ± 0.08

5 Hardware implementation

With the network and meta-learning algorithm selection complete, in this section, the chip designed for few-shot learning over sequential data is discussed. It is codenamed *Chameleon*.

This name is chosen as a chameleon (see Fig. 59) is able to make small changes to its colors to adjust to its environment[26] (for example, turning to a darker shade of their color), similar to how this chip only makes small changes in the network when it is adapting to new classes. Interestingly, only in 2015, the same year that the Omniglot dataset [31] was released and ResNets were introduced [75], the exact way chameleons did this was uncovered. Teyssier *et al.* showed in [117] that chameleons shift color through actively restructuring a lattice of nanocrystals, which, depending on how densely they are packed, reflects different color wavelengths of light.

Figure 59: Photograph of a panther chameleon.[27]

Chameleon is a fully digital design, implemented using (System)Verilog. All code is written in a parameterized fashion, meaning that it is trivial to change parameters such as the activation bit widths or compute array size. The final design has been taped out in a 40-nm TSMC technology node in its low-power (LP) MS/RF plus 1.1V/2.5V flavor. As at the time of writing this thesis, the hardware has not yet been produced, all performance metrics in this section that relate to the hardware, such as power consumption, are based on post-layout simulations.[28]

This section is organized as follows. Starting off, Section 5.1 outlines the key requirements for the design. Then, in Section 5.2, related hardware designs for processing sequential data are presented. Next, Section 5.3 presents the high-level architecture of the design. After the architectural overview, the various blocks of the architecture are explored in more detail. First, Section 5.4 presents the connectivity of the design, after which the design of the network controller is discussed in Section 5.5. Then, Section 5.6 covers the design of the PE array. After this, Section 5.7 presents the memory structure, Next, Section 5.8 covers the few-shot learning implementation, while finally, in Section 5.9, the proposed design of this thesis is compared to the state of the art.

[26] Contrary to popular belief, chameleons are not able to blend into any background.

[27] Image taken from `https://www.wallpaperflare.com/chameleon-on-branch-of-tree-animals-reptile-schuppenkriechtier-wallpaper-wqdvq` (Creative Commons license)

[28] Unless otherwise specified.

5.1 Requirements

In this section, the requirements for the hardware design are presented. The requirements are defined in a SMART way, where SMART is an acronym that stands for *Specific, Measurable, Attainable, Relevant and Timely*. The full list is presented below. Note that **CHA** stands for "Chameleon". The driving requirements, which influenced the design the most, have been highlighted.

- **CHA-LIM-01**: the maximum silicon area of the design shall be 4 mm^2, including the padring.
- **CHA-LIM-02**: the chip shall store all network parameters required for processing on-chip.
- **CHA-CORE-01**: the chip shall be able to perform classification of input sequences.
  - **CHA-CORE-01.01**: the chip shall support linear layers connected to TCN networks.
  - **CHA-CORE-01.02**: the chip shall be able to compute the argmax of the output of a linear layer.
  - **CHA-CORE-01.03**: the chip shall support continuous classification, meaning that classification is performed at every timestep.
- **CHA-CORE-02**: the chip shall be able to perform continuous regression.
- **CHA-CORE-02**: the chip shall support TCN processing.
  - **CHA-CORE-01.01**: the chip shall support a streaming dataflow / sequential processing approach.
  - **CHA-CORE-01.02**: the chip shall support variable-channel 1D convolutions.
    - **CHA-CORE-01.02.01**: the chip shall support variable kernel size from 1 to 9 for different networks.
    - **CHA-CORE-01.02.02**: the chip shall support increasing dilation with a dilation factor (d_f) of 2.
    - **CHA-CORE-01.02.03**: the chip shall support 20-way, 20-shot few-shot testing.
  - **CHA-CORE-01.03**: the chip shall support the processing of the network with 1-24 layers.
  - **CHA-CORE-01.04**: the chip shall support variable input dimensionality size and variably-dimensioned input.
  - **CHA-CORE-01.05**: the chip shall support ReLU activation computation.
  - **CHA-CORE-01.06**: The chip shall support residual layers.
- **CHA-CORE-03**: few-shot learning shall completely take place on-chip, requiring only neural network inputs without any other communication.
  - **CHA-CORE-03.01**: the chip shall be able to find the support embedding with the lowest distance to the input embedding.

  - **CHA-CORE-03.02**: the chip shall be able to average input embeddings.

- **CHA-NUMERIC-01**: the chip shall support computation with weights in the 4-bit logarithmic format.

- **CHA-NUMERIC-02**: the chip shall support computation with 4-bit unsigned activations.

- **CHA-NUMERIC-03**: the chip shall support computation with 16-bit biases.

- **CHA-NUMERIC-04**: the chip shall support computation with 4-bit unsigned scales.

- **CHA-COMM-01**: the chip shall use an SPI communication protocol for configuring the parameters, as well as for reading and writing to all SRAMs.

- **CHA-COMM-02**: the chip shall have an input port that receives the sequential inputs for processing, separate from the SPI.

- **CHA-COMM-03**: the chip shall have an output port that outputs the regression values or classification results, separate from the SPI.

- **CHA-USAB-01**: the chip shall have enough SRAM memory space to store 120k parameters.

- **CHA-USAB-02**: the chip shall have an internal clock generator with a minimum speed of 50 MHz and a maximum speed of 250 MHz.

- **CHA-USAB-03**: the chip shall allow for monitoring the internal clock frequency.

5.2 Related work

With the requirements covered, in this section, recently developed digital hardware designs that share an axis of similarity with this design are introduced. Microcontroller-only implementations are not discussed as their approaches, limitations and possibilities, vary significantly from the field programmable gate array (FPGA) and silicon designs relevant to this thesis. The same goes for analog, spiking, memristor- and computing-in-memory-based implementations.

In Section 5.2.1, FPGA and application-specific integrated circuit (ASIC) designs for TCN acceleration are covered, after which Section 5.2.2 covers non-TCN based accelerators for the keyword spotting task.

5.2.1 TCN acceleration

In [44], Giraldo *et al.* propose a batch and real-time processing scheme for TCN inference as part of a 65nm concept ASIC for keyword spotting. It has an advanced set of features to maximize power savings, namely sprinting, dynamic voltage and frequency scaling (DVFS) and power gating (PG). Furthermore, a cascaded classifier is introduced which only enables the TCN classifier when voice is detected, thereby allowing for further power savings. During real-time inference, the design consumes 25 µW.

Similarly, Bernardo *et al.* [41] propose a TC-ResNet [63] accelerator for keyword spotting in the form of a 22nm concept ASIC. Instead of a cascaded classifier, a conditional classifier is developed, where if a confidence metric is not high enough in a layer, the processing of that sample is stopped to save power. The design also supports PG and uses 8.2 µW during real-time inference.

In [42], He *et al.* design an FPGA implementation for a TENet [118], a derivative architecture of the TC-ResNet. Furthermore, a simplified MFCC algorithm is employed to save power on the feature extraction. It can operate at a clock speed of 90 kHz for real-time inference, during which 209 mW is consumed.

Orthogonally, Jain *et al.* [39] present a 22 nm in-silico full system-on-chip (SoC) consisting of a RISC-V CPU and a custom ML accelerator for inference of CNNs, RNNs, SVMs, autoencoders and TCNs. During real-time keyword spotting with a TCN, the total power consumption is 193 µW.

Carreras *et al.* present a quantitative evaluation for TCN inference using FPGAs in [74]. This exploration is performed with an FPGA-based CNN inference accelerator, which is extended to run TCNs in a streaming and batched fashion, similar to [44]. The final design was tested using three different benchmarks, with the system using approximately 3.3 W.

Also focused on optimizing TCN inference, Ibrahim *et al.* [119] propose a data-flow transformation to convert a dilated convolution to a non-dilated convolution. This approach is part of a 40-nm concept design for a gesture recognition case study using an ultrasound sensor. Unfortunately, no power consumption metrics are reported.

5.2.2 Keyword-spotting acceleration

In [40], Giraldo *et al.* propose a 65nm in-silico LSTM accelerator for keyword spotting. One of the unique features of this design is that it uses approximate computing techniques via look-up tables (LUTs). During inference, it consumes 5 µW.

In another work from Giraldo *et al.* [94], a speech-triggered wake-up mixed-signal system-on-chip is presented that can directly interface with an analog microphone, perform MFCC feature extraction as well as KWS and speaker verification (SV). It is taped out in a 65nm CMOS process and uses 10.6 µW in typical real-time scenarios. The supported network architecture is again an LSTM with a fully connected layer.

Another LSTM-based KWS solution is proposed by Chong in [120] in a 40nm CMOS process. Similar to [94], the MFCC extraction also runs on-chip in this design. Different from [94] is that the accelerated LSTM model is pruned and compressed to reduce the parameter and operation count. During real-time inference, 2.51 µW is used at an accuracy of 90.6% on the 10-way Google Speech Commands task, which is however extracted from simulation results and not validated in silicon.

In [121], Shan *et al.* propose a 510 nW wake-up KWS chip. Similar to [94], MFCC feature extraction is performed on-chip using a serial FFT-based approach, instead of the regular parallel approach. The network used is a binarized depthwise separable CNN, taped out in 28-nm CMOS. However, only the accuracy results for 1- and 2-way keyword classification are clearly presented in the paper, with the accuracy for the more common 10-way classification task discussed as "less than 90%".

Also using a CNN is [122], where Lu *et al.* present a depthwise separable convolution accelerator simulated in 28 nm CMOS. Two special features are the use of an approximate MAC unit and a streaming convolution reuse approach. Unfortunately, Lu *et al.* are not very transparent on their power usage for the complete design.

Combining the convolutional and recurrent paradigm, Liu *et al.* [123] propose a one-dimensional convolutional recurrent neural network (1D-CRNN) accelerator with on-chip MFCC feature extraction in a 22-nm process. Similar to [122], approximate computing is used. The power consumption of the design ranges from 1.4 µW (for 1 keyword + unknown category) to 2.1 µW (for 5 keywords + unknown category) during real-time inference, also resulting from simulation results that are not validated in silicon.

5.3 High-level architecture

In this section, the high-level architecture of the hardware design will be introduced. It was decided to follow a typical accelerator-style architecture, with memories, controllers and compute modules clearly separated, similar to the hardware designs covered in Section 5.2. Fig. 60 shows a visual overview of the architecture for Chameleon.

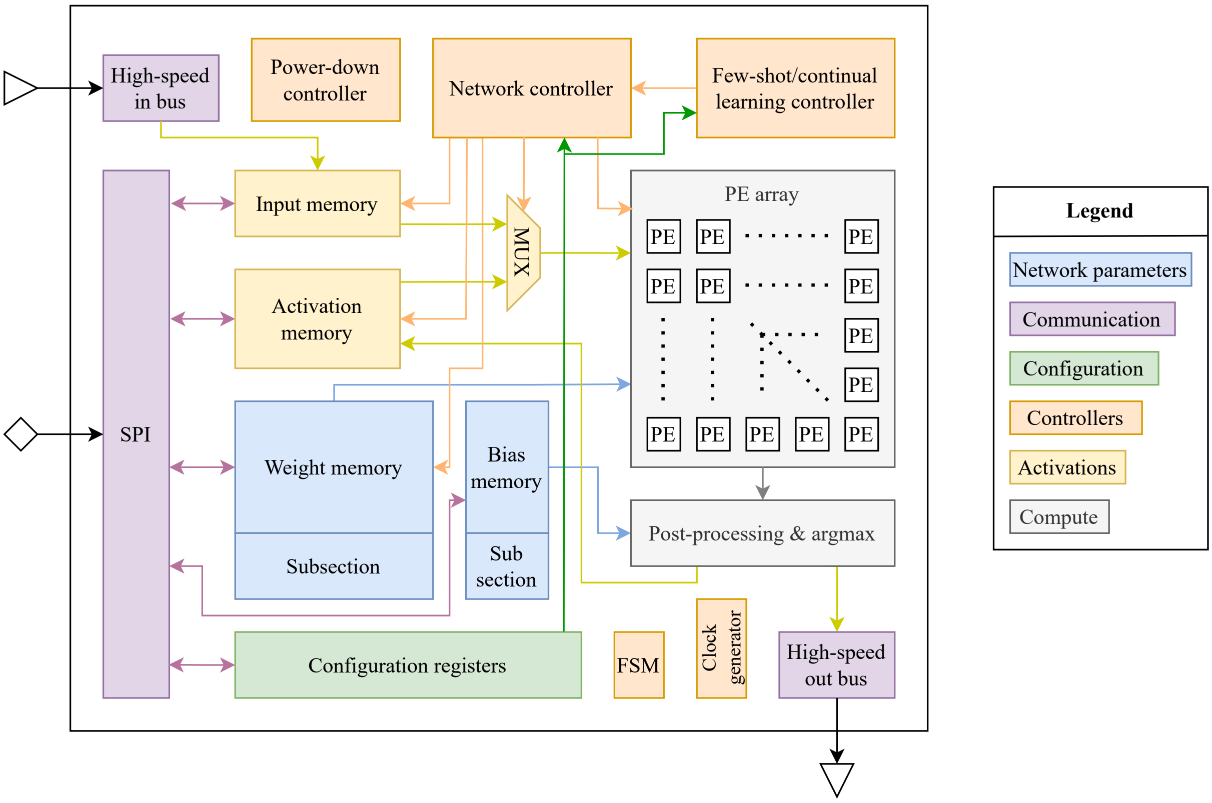


Figure 60: High-level architectural overview for Chameleon. A color-coded legend is displayed on the right.

The chip has three separate interfaces. The SPI interface is used for configuring the network weights, structure and (few-shot) processing options, where the latter two are stored in the configuration registers. The SPI interface can also be used to read out the contents of all the memories and key internal registers. The high-speed input bus is then used to receive the per-timestep feature vectors, while the high-speed output bus sends the processing results.

The processing side of the design consists of a processing element (PE) array connected to a post-processing block in a pipelined fashion. The PE array effectively is a highly parallel matrix-vector compute-unit while the post-processing block handles various tasks, such as rescaling the PE array outputs, performing the activation computation and finding the argmax of the output vector.

Which weights and activations go into the PE array is orchestrated by the network controller: this block generates all the control signals to process a TCN according to its structure and makes sure that all data is loaded and written back properly.

The few-shot and continual-learning controller then operates completely independently from the network controller and overwrites signals coming out of the network controller when few-shot learning is performed. The chip also has a power-down controller, handling when the power can be removed from some of the weight and bias memories. A small finite-state machine (FSM) is used to keep track of the global system state.

The memories that Chameleon has can be divided into four categories: weight storage, bias storage, activation storage and input storage. Both the weight and the bias memory have a subsection that is always on, while the rest of these memories can be turned off to save power when executing small networks. All memories consist of TSMC SRAM IPs, except for the input memory, which is a custom block that uses registers to store data. If the first layer is being processed, data from the input memories flow into the PE array, otherwise, data from the activation memory does so.

Finally, the design includes an embedded clock generator for operation at clock frequencies larger than 50 MHz. Below this frequency, the clock can be provided externally.

The different parts of the architecture, that were briefly touched upon here, will now be explored in more detail in the following sections.

5.4 Connectivity

After having considered the complete high-level architecture, this section details the way the hardware connects to the outside world. First, Section 5.4.1 will discuss the SPI bus (**CHA-COMM-01**), after which Section 5.4.2 will discuss the high-speed in and out buses (**CHA-COMM-02**, **CHA-COMM-03**).

5.4.1 SPI bus

In this design, the SPI bus is the main communication port. It can be used for the following tasks:

- configuring the chip,
- programming the weights and biases,
- reading back the weights and biases,
- reading out the (input) activations,
- inspecting on-chip registers.

The SPI protocol was chosen over other protocols such as I2C or UART for simplicity and the availability of recent silicon-proven open-source implementations in Verilog [124].

Compared to a standard SPI implementation, the SPI bus on Chameleon only has three pins (see Table 18), where the chip-select (CS) pin has been omitted as no multi-chip setup is foreseen.

As the SPI protocol does not define the structure of the data stream, this needs to be defined *a priori* so that the client and server can communicate correctly. Fig. 61 visualizes the message structure for interfacing with Chameleon, where the size of each message is 32 bits.

Before reading data from or writing any data to Chameleon, an instruction message is sent from the server to the client (see Fig. 61). This message consists of four parts: the read-write bit,

Table 18: Pins of Chameleon's SPI bus. Client refers to Chameleon, server refers to an external device.

Pin	Direction	Width	Description
SCK	Input	1-bit	SPI clock generated by SPI server
MOSI	Input	1-bit	Master (server) output, slave (client) input
MISO	Output	1-bit	Master (server) input, slave (client) output.

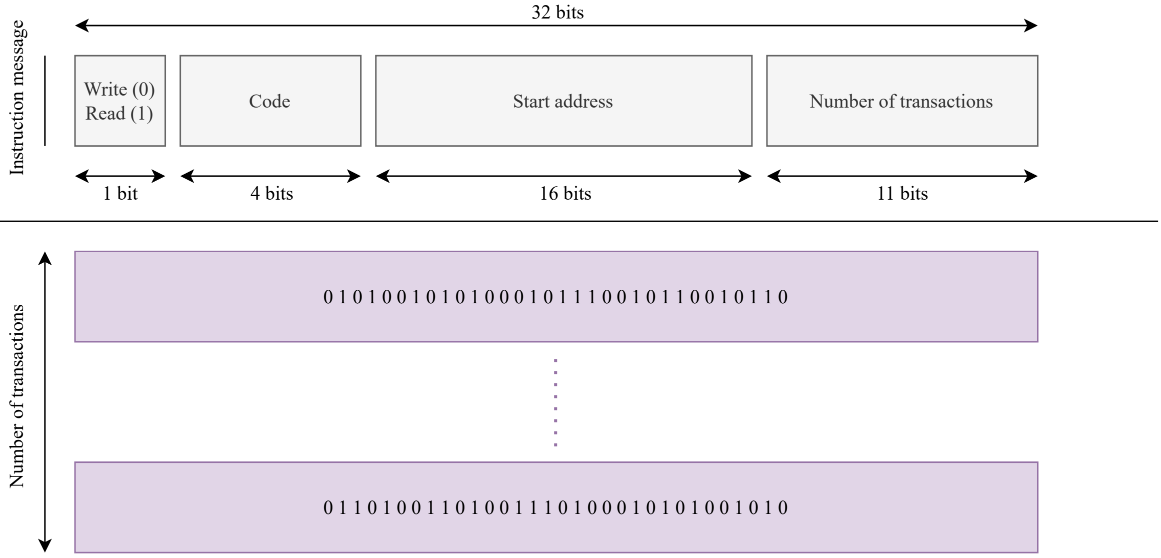


Figure 61: Structure of an SPI transaction, including packet formatting in the implemented SPI protocol.

a 4-bit code, the 16-bit start address of the transaction and the number of transactions encoded over the remaining 11 bits.

The read-write-bit indicates whether data will be written to (0) or read from (1) the chip after the instruction message. The code contains which part of the chip should be read from or written to. The following mapping from code values to storage elements is defined:

1. configuration registers (write only), internal registers referred to as "pointers" (read-only),
2. weight memory (read and write access),
3. bias memory (read and write access),
4. activation memory (read and write access),
5. input memory (read and write access).

Note that, finally, the 4th bit is not used anymore as there are only five locations. While this bit could have been repurposed for the number of transactions, this was not pursued due to time constraints.

Next, the start address is 16 bits wide and indicates the address from which reading or writing should start. Finally, the 11-bit number-of-transactions field contains how many 32-bit messages (so not addresses) should be received or transferred.

Based on this instruction message format, the following steps during SPI communication are defined as follows (see Fig. 62):

1. the server sends an instruction message (a[31:0] in Fig. 62)) to the client, requesting a read or write from multiple addresses (see instruction message format in Fig. 61),
2. in the case of a write request, the server starts sending bits sequentially, while in the case of a read request, the client starts sending out bits sequentially. While bits are transferred sequentially, both from the server and client side, they are virtually grouped in packets of 32 bits,
3. after the number of 32-bit packets (d[31:0] in Fig. 62)) transferred equals the specified number of transactions from the instruction message, Chameleon is awaiting a new instruction message.

The SPI bus only operates when the SPI clock (SCK) is active. To disable SPI communication, SCK can simply be pulled low, effectively acting as a chip-select.

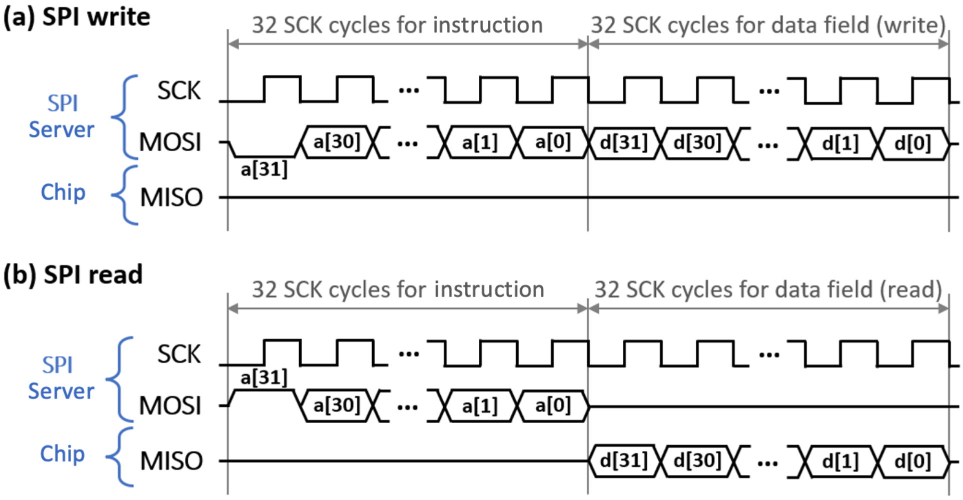


Figure 62: SPI timing diagram. Based on a figure from [124].

5.4.2 High-speed in and out buses

In order to have high-speed communication between Chameleon and external devices, high-speed in and out buses were created, which stem from requirements **CHA-COMM-02** and **CHA-COMM-03**. They were introduced to create a more realistic hardware design, where data goes in and out of the chip over a fast, parallel bus instead of over SPI, which is serial.

These buses use a four-phase handshake protocol for asynchronous communication. See Fig. 63 for a timing diagram of communication via this protocol: the four phases are visualized with arrows.

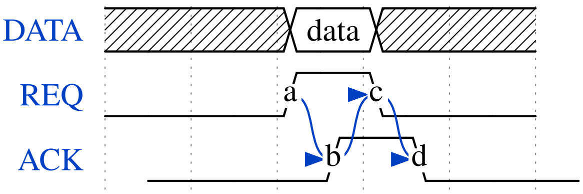


Figure 63: Timing diagram of the four-phase handshake protocol.

When data is coming from an external device, the REQ line will be pulled high (a in Fig. 63) as soon as new data is available on the DATA line. Then, as soon as the ACK line goes high (b), indicating that the data has been received correctly, the REQ line goes low ($b \to c$) and Chameleon will not care about the state of the DATA line anymore. When REQ is back to a low state, ACK will go low again as well ($c \to d$).

When outputting data, the above-mentioned steps are unchanged, except that the DATA and REQ line are controlled by Chameleon and the ACK line is an external input.

Originally, the input bus was intended to have 32 wires, but to avoid a pad-limited design, it was opted to go for 16 wires instead. The DATA line from the output bus then has 8 wires.

5.5 Network controller

In this section, the design of the network controller is discussed. This module is responsible for generating all the control signals to perform streaming inference with a TCN-type network (fulfilling **CHA-CORE-02**).

As opposed to more standard architectures optimized for batched inference, Chameleon focuses on streaming (i.e. batch-size-1) inference (Fig. 64), where after every time t, a new input feature from a single sample is processed and an output is generated, is that it is the most realistic deployment scenario for few-shot learning edge hardware. For example, an edge device for keyword spotting will only receive the outputs of one microphone. Compared to batched inference, streaming inference results in less parallelism and thus fewer data reuse opportunities.

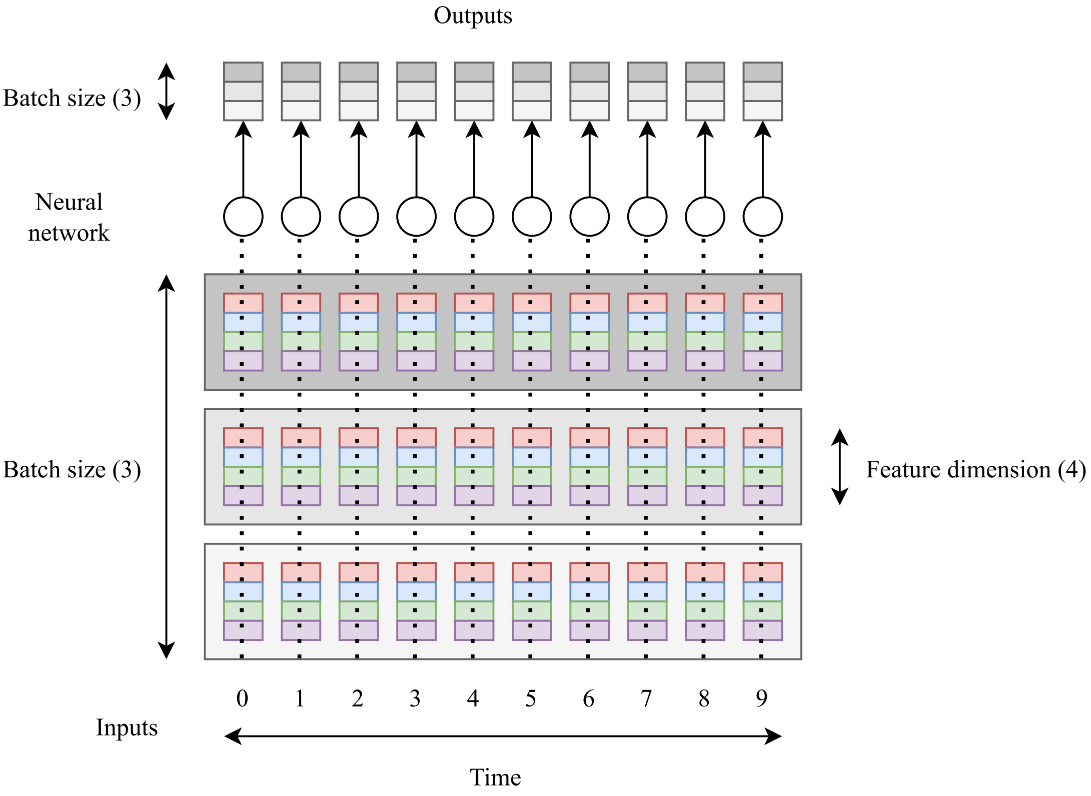


Figure 64: Batched inference with batch size 3 on a sequence of length 10 where each feature is 4D. In streaming inference, the batch size is 1.

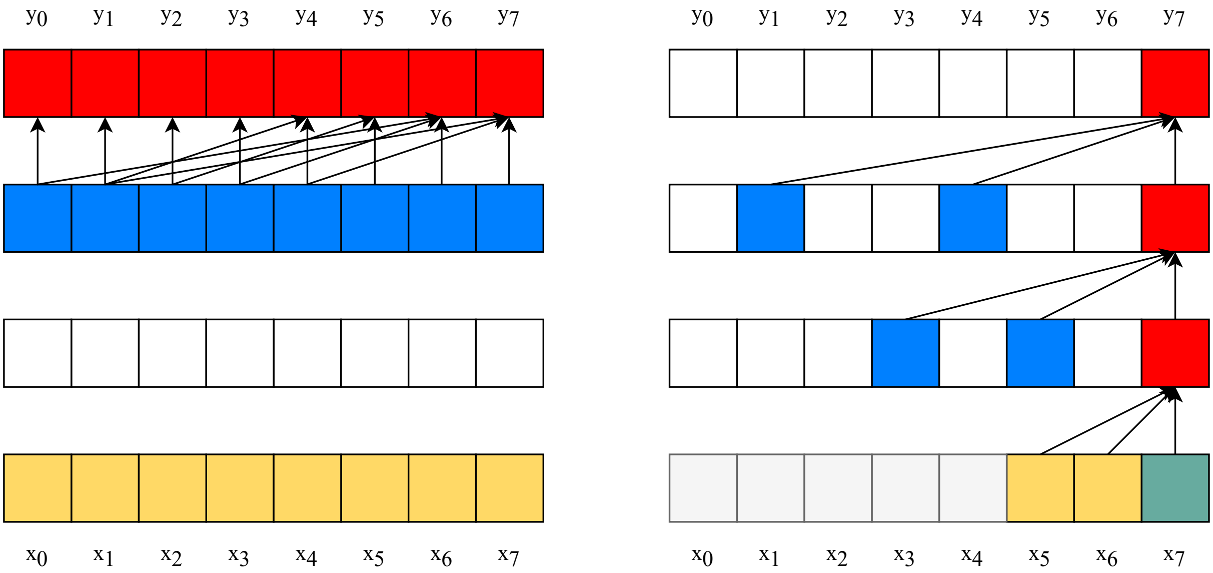
Considering this, first, in Section 5.5.1, various controller alternatives from prior work in this space are reviewed, after which one base approach is chosen in Section 5.5.2. Section 5.5.3

then presents the controller implemented based on this approach.

5.5.1 Survey of process and control alternatives

This section discusses various approaches to controlling TCN acceleration. For each approach, the used dataflow, activation memory required (assuming fixed-size weight memory, as per Section 3.1.3.1), the required implementation and supported TCN features (i.e. dilation and residual layers) are considered.

In [42], He *et al.*, use a control unit that reads a 64-bit instruction per layer, which contains all details required to process that layer. The memory addresses and enable signals are then generated by an address-generation unit based on this configuration. A weight-stationary dataflow [125] is employed: this means that the neural network weights are loaded once from the memory and kept in a compute block, after which they are reused across multiple cycles before new weights are loaded. Using this dataflow, the network is processed layer by layer: this implies a batched-inference process and not a streaming one, as to compute the first layer, all timesteps need to have been received first, see Fig. 65a). For this, three activations memories are used: two of these are used as ping-pong buffers for reading and writing the activations in parallel, while the third one is used to store residual path values. Using this approach, the design has an activation memory to weight memory size ratio of 32.5% at an activation memory size of 6.5 kB. The control unit does not support dilation.



(a) Layer-by-layer TCN inference.

(b) TCN streaming inference as performed by [44] and [74]. Image inspired by [44].

Figure 65: Schematic overview of layer-by-layer and streaming processing for a TCN. Values in red are computed in the current inference pass (after receiving x_9) while values in blue were computed in previous passes. Values in yellow are previously received inputs while the value in teal is the current input.

In [44], Giraldo *et al.* follow a similar approach: the architecture includes an instruction memory that contains, for each layer, the hyperparameters and configuration variables required for its execution. A separate module then controls the PEs and memory accesses. In contrast to [42], the processing in this work focused on dilated convolutions, for which two processing modes are proposed: batched inference and streaming inference. In the streaming mode, the computational graph of the TCN is traversed greedily (Fig. 65b): as soon as a new input is available, all intermediate values and outputs that can be computed, will be computed. However, the handling of residual layers is not discussed. The convolution operation in the network is computed like a vector-matrix multiplication, which is the same operation used in an FC layer: a partial output-stationary dataflow is therefore used for processing the TCN [125] where, for every cycle, new weights are loaded but the output of the computation is not immediately written back to the memory. Two activation memories are used in a ping-pong configuration with a balanced activation memory to weight memory size ratio of 100%, with a large activation memory of 64 kB. However, note that this large activation memory is likely the result of also supporting inference with a batch size of 64.

Carreras *et al.* [74] also propose a streaming and batched inference approach to execute TCNs on an existing CNN accelerator. Their streaming processing of the TCN is nearly the same as in [44], however residual layers, as well as variable dilation values, are now considered. Unlike [44] however, a RISC-V CPU instead of a custom hardware module is the inference controller. The memory implications for streaming inference are not discussed.

Finally, in [119], Ibrahim *et al.* propose a data-flow transformation converting a dilated convolution to a non-dilated convolution. This removes the need to scale the delays between input samples going into MAC units. The dataflow processes a TCN layer by layer, similar to [42]. Unfortunately, Ibrahim *et al.* do not consider residual layers in their approach and require a relatively large 20-kB activation memory (weight memory size not provided).

5.5.2 Controller choice

Table 19: Comparison between different TCN inference control methods. **Bold** indicates best performance in row while - indicates that the work does not explicity mention what is used.

	He *et al.* [42]	**Giraldo *et al.* [44]**	**Carreras *et al.* [74]**	**Ibrahim *et al.* [119]**
Dataflow	Weight stationary	Partial output stationary	-	-
Processing style	Layer-by-layer	**Streaming**	**Streaming**	Layer-by-layer
Activation memory overhead	**32.5%**	100%	-	-
Supports dilation?	No	**Yes**	**Yes**	Yes
Support residual path?	**Yes**	No	**Yes**	No
Implementation	**Instruction + address-generator**	**Instruction + address-generator**	RISC-V controller	**Configurable address-generator**

To make a decision on which method to use for controlling the inference, the described approaches are compared side-by-side in Table 19, which shows that only the approach from Carreras *et al.* [74] supports both dilation and residual layers. Furthermore, as in this thesis it is desired to maximally reduce the size of the activation memory, a streaming approach is preferred as fewer values have to be kept in memory at the same time (compare the blue blocks in Fig. 65a with Fig. 65b). Therefore, only the approach from Carreras *et al.* remains. However, their processing controller was implemented in a separate CPU, which is not desirable for a small accelerator like Chameleon.

Therefore, as Giraldo *et al.* [44] have demonstrated that a similar approach can be implemented as a configurable hardware module, it is decided to (i) follow the greedy streaming inference from both works, (ii) to implement it in a dedicated hardware controller as in [44] and (iii) to expand it to support residual layers.

5.5.3 Implementation

The final implemented controller supports:

- p TCN layers followed by q FC layers, where $2p+q \leq 32$ (**CHA-CORE-01.03**),
- up to $32n$ channels per layer, where n is the size of the PE array,
- per-layer configurable kernel size with any kernel size from 1 to 16 (**CHA-CORE-01.02.01**),
- residual layers (**CHA-CORE-01.06**): optionally, an identity operation can be scheduled in case the incoming and outgoing channel count is the same, otherwise a 1×1 convolution is scheduled,
- a global flag to enable or disable dilation in all TCN layers,
- streaming and finite-length sequence classification (**CHA-CORE-01**), as well as streaming and finite-length sequence regression (**CHA-CORE-02**).

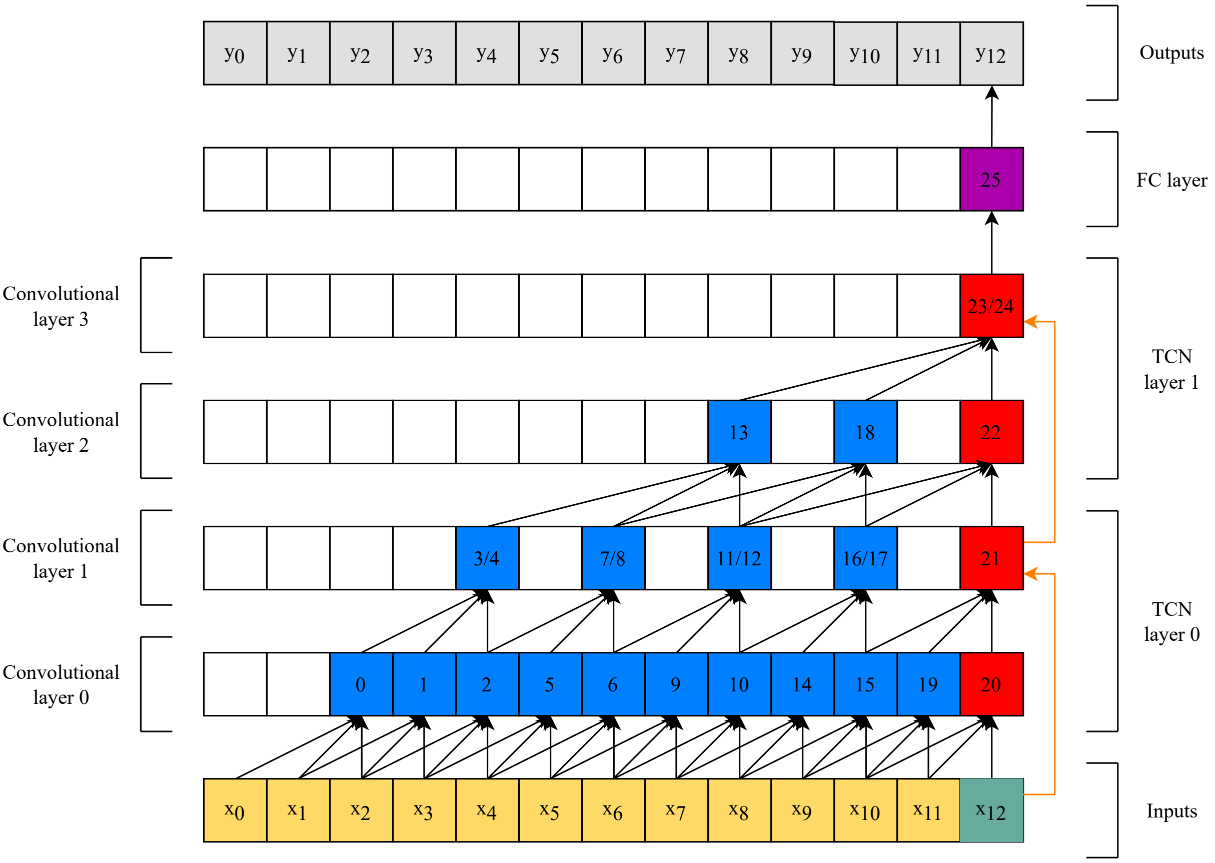


Figure 66: Computational graph for a 2-layer TCN with kernel size 3 that is connected to a classification layer. The processing order is indicated by the numbers in the boxes: every step increase represents one convolution operation. Since every second convolutional layer needs to include a residual path, some boxes contain two numbers: one for performing the convolution and one for the residual step.

The controller generates, every cycle, a weight, bias, input activation and output activation address and provides the required signals to enable reading and writing from the respective memories. As discussed, the TCN structure is traversed greedily: Fig. 66 illustrates the computation order for a 2-layer TCN. This means that there is no weight or activation reuse, as in every step different weights and activations are required.

To support residual layers, every time an activation has been computed that allows for another convolution to be performed in the next odd-numbered layer (e.g., in Fig. 66, after step 20 is computed, step 21 in convolutional layer 1 can be computed), the value along the residual path is computed, the convolution is performed and its result is added.

As the controller operates in a streaming fashion, it can be the case the processing has to be halted to wait for input data. For example, if after step 4 in Fig. 66, feature vector x_5 has not yet been sent to Chameleon, processing cannot continue. The controller detects such situations and stops processing until a new input is ready, after which processing resumes.

The controller is fully PE-array-size-agnostic, as it operates in terms of blocks instead of channels, where each block then is an c-dimensional feature vector that is fed to the PE array. It is implemented as a low-footprint FSM that requires < 0.1 kB of registers.

5.6 Processing element array

In this section, the processing element (PE) array of Chameleon is discussed, as per the structure shown in Fig. 60. This array, a matrix-vector multiplier unit, performs all computations required to do few-shot learning and to perform inference on TCNs (**CHA-CORE-02**).

First, Section 5.6.1 discusses the design of individual PEs, then Section 5.6.2 covers their integration into an array as well as the related design choices. Finally, Section 5.6.3 covers the post-processing and argmax block.

5.6.1 Processing element design

The PE is a core building block of an accelerator: it is often replicated many times and organized in a grid to form the main compute module of a design, e.g. a MAC array (see Section 2.3.2.1). The design of the PE is linked to requirements **CHA-NUMERIC-01** and **CHA-NUMERIC-02**.

When uniform weights are used, the PE is simply a $n_a \times n_w$ multiplier, where n_a and n_w are the numbers of bits used for the activations and weights respectively, possibly accounting for signed operands. However, as Chameleon uses logarithmic weights, the PE is now a signed shift unit, a parameterized Verilog implementation of which is shown in Listing 1.

partial-output stationary dataflow is used (K/C), similar to an FC layer deployment.

Listing 1: Parameterized description of the PE.

```
module pe
    #(parameter WEIGHT_BIT_WIDTH = 4, parameter INPUT_BIT_WIDTH = 4,
      localparam OUTPUT_BIT_WIDTH = INPUT_BIT_WIDTH + (2**
      WEIGHT_BIT_WIDTH)/2)
    (
        input [INPUT_BIT_WIDTH-1:0] in,
        input [WEIGHT_BIT_WIDTH-1:0] weight,
        output signed [OUTPUT_BIT_WIDTH-1:0] out
    );

    wire weight_sign;
    wire [WEIGHT_BIT_WIDTH-2:0];

    assign {weight_sign, weight_abs} = weight;

```

```
14      wire signed [OUTPUT_BIT_WIDTH-1:0] out_abs = in << weight_abs;
15
16      assign out = weight_sign == 1'b1 ? -out_abs : out_abs;
17
18  endmodule
```

As visible in Listing 1, first the sign and weight value are separated, after which the input is shifted to the left by weight value bits. Then, based on the sign, where 1 indicates that the weight is negative while 0 indicates that it is positive, the value is negated or not.

To make this more concrete, Eqs. (52) to (53) show an example PE operation with weight w and input activation x:

$$w = -64_{10} = -\left(2^{\mathbf{6}}\right) \xrightarrow{\text{stored as}} 1110_2, \quad x = 13_{10} = 1101_2 \tag{52}$$

$$w \cdot x = -(1101_2 << \mathbf{6}) = -1101000000_2 \xrightarrow{\text{stored as}} 111111001100_2 = -832_{10} \tag{53}$$

where $_{10}$ indicates that a number is decimal and $_2$ indicates that a number is binary. Note that that the signed output of $w \cdot x$ is stored in 2s complement format.

The equation for calculating the bit width of this output is shown in Listing 1 on line 2, namely $n_a + (2^{n_w - 1})$. This can derived as follows: it is possible to shift the input (n_a bits) a maximum of $2^{n_w-1} - 1$ bits. Adding the maximum shift value (the maximum amount of zero bits that could be appended to the input) and bit width of the activation yields $n_a + 2^{n_w-1} - 1$. Then, to account for the sign of the output, one extra bit is required, canceling out the -1 and resulting in $n_a + (2^{n_w-1})$.

Putting this together, Figure 67 contains a schematic of the PE outlined in Listing 1 where the final bit widths for Chameleon are filled in.

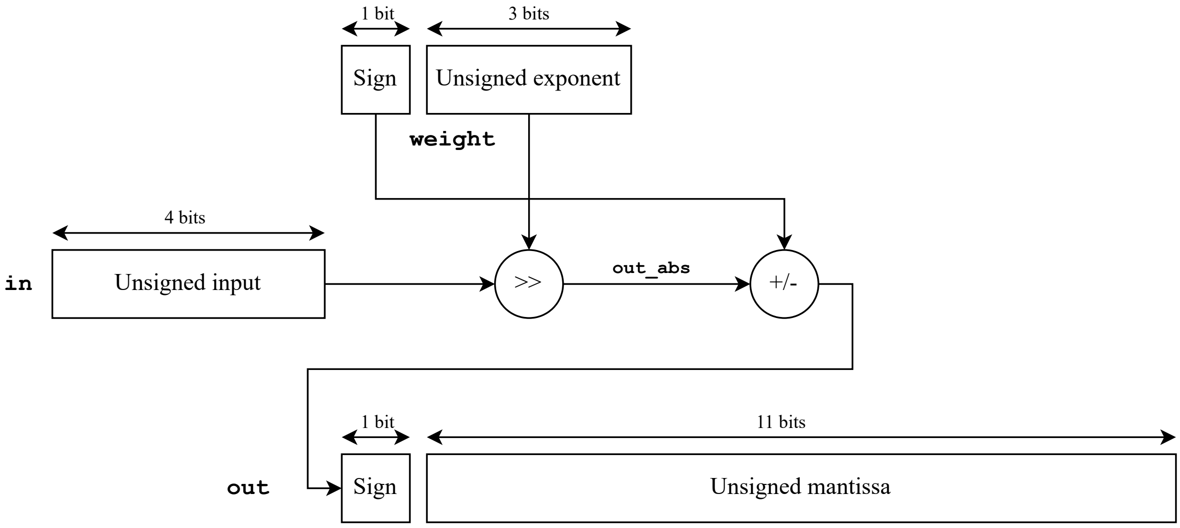


Figure 67: Schematic of a processing element for operation with a 4-bit logarithmic weight and 4-bit uniform input.

5.6.2 Array design

In this section, the integration of the PE into an array is discussed. For Chameleon, the PEs are organized in a grid with dimensions $n \times n$: these dimensions are kept parameterized throughout this section, as the final values are optimized in conjunction with the available TSMC SRAMs later. The grid is designed specifically for an *output-stationary* (OS) [125] dataflow. This dataflow keeps the partial sums of convolutional and fully-connected layers stationary (i.e. they stay in local registers), aiming to minimize the energy required for reading and writing the partial sums [125].

In Fig. 68, the $n \times n$ grid integration is shown: the general structure is the same as the conceptual MAC array of Fig. 29. Following the standard definition of the OS dataflow [125], every input activation (in[0]–in[$n-1$]) is horizontally streamed across the PE array, while each PE receives a unique weight value. Per column, the outputs of the PEs are then added together using an adder tree, resulting in n partial sum values.

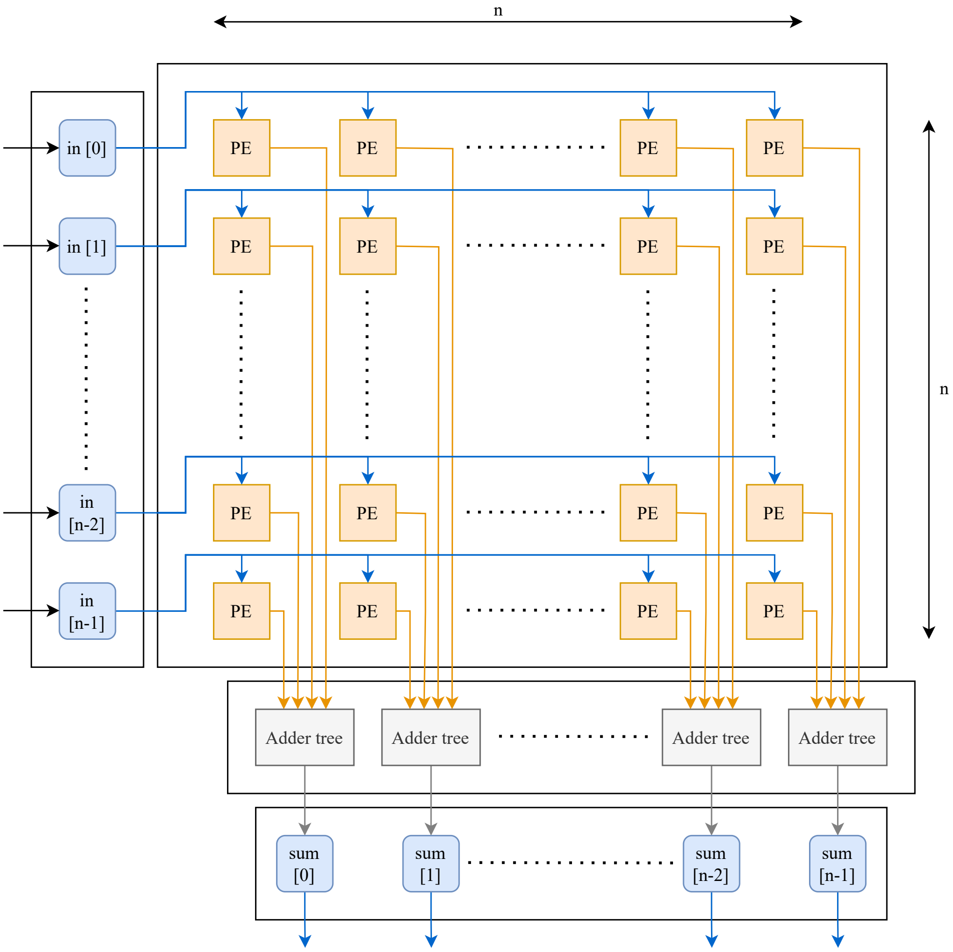


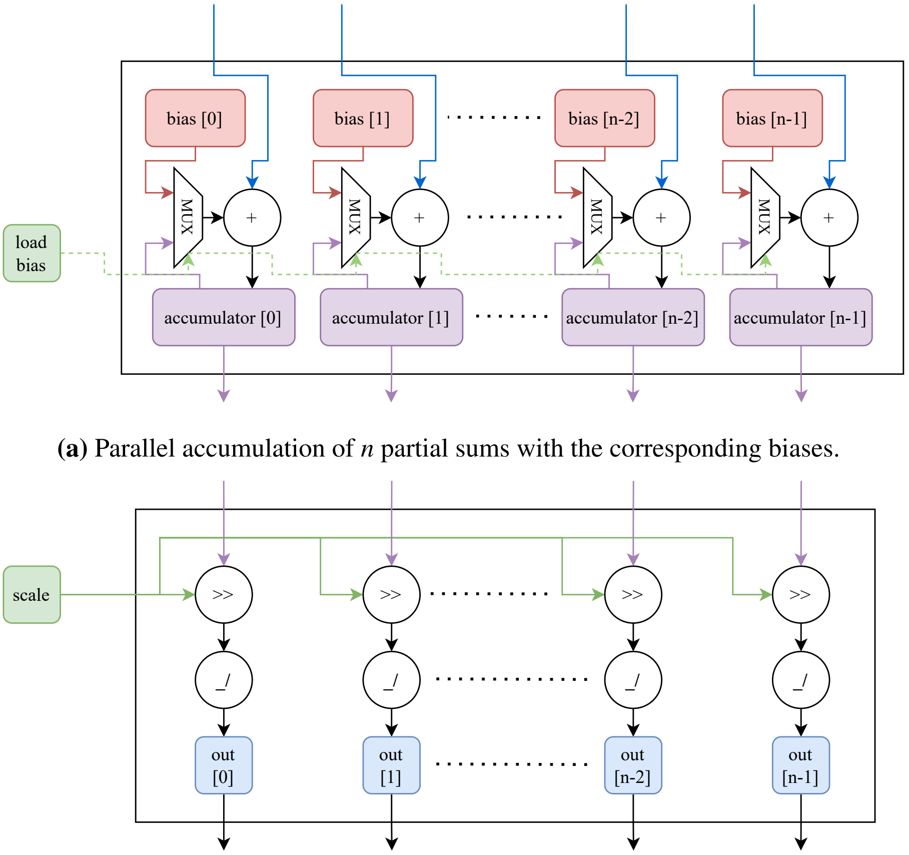
Figure 68: Schematic overview of an $n \times n$ PE grid as used in Chameleon.

In this grid, each PE performs one MAC, which is executed as a shift-accumulate (SAC) operation, per cycle. Therefore, to feed the entire grid of PEs, an input vector of $4n$ bits and a weight matrix of $4n^2$ bits has to be retrieved from the memories every cycle, to enable it to produce a partial-sum vector of $12n$ bits every cycle.

5.6.3 Post-processing

From the PE grid, the n partial sums flow directly into the post-processing module (as shown in Fig. 60). The post-processing module then contains two blocks: an aggregator (Fig. 69a) and an activator (Fig. 69b).

In the aggregator block, the partial sums from subsequent cycles are added to compute a complete sum. The MUX in every column (Fig. 69a) switches between adding the bias or the value in the accumulator register to the incoming partial sum, based on the `load bias` signal. In the first cycle of an operation that requires multiple partial sums to be added, the `load bias` is high: the partial sum + bias is now stored in the per-column accumulator register. After this cycle, the signal can go low as each subsequent partial sum has to be added to the bias and previously computed partial sums, stored in the accumulator.



(a) Parallel accumulation of n partial sums with the corresponding biases.

(b) Scaling and activating n accumulators to compute n final outputs.

Figure 69: Schematic overview of the two blocks in the post-processing module of the PE array.

After completing the computation of all the partial sums, the n accumulator registers contain the final, complete sums. Following the quantization function, these sums have to be scaled Eq. (25): since per-tensor *power-of-two* scale values are used, this operation can be performed by bit shifting the accumulators in all columns by `scale` bits (Fig. 69b). After this, the ReLU activation function is applied to the scaled accumulators (**CHA-CORE-01.05**). Finally, the 4 LSBs per column are taken and written back to the memories.

5.7 Storage

In this section, the selection of memory elements to store the neural network weights, biases and activations will be discussed. As the dimensions of the PE array were kept variable through n, the optimal value of n will first be determined in Section 5.7.2. Then, Section 5.7.3 covers the neural network weights storage, followed by Section 5.7.4 where the storage of the biases is discussed. Section 5.7.5 discusses the activation storage configuration. Finally, in **??**, the way the incoming inputs are stored is discussed.

5.7.1 Generic storage setup

In this section, the storage configuration of Chameleon will be discussed. We had access to the following five memory types in the selected 40nm node:

- a dual-port SRAM, allowing two reads, two writes, or one read and one write at the same time,
- two single-port SRAMs,
- a high-performance single-port SRAM,
- a one-port register file,
- a two-port register file, allowing one read and one write at the same time.

Before being able to select the optimal memory types, the required storage sizes for processing each of the three developed networks (see Section 3.2.2) need to be determined. These values are displayed in Table 20: the weight and bias counts are calculated using PyTorch [104], while the number of activations required comes from simulating the behavior of the TCN controller (see Section 5.5).

Table 20: Number of weights, activations, and biases to be stored for the three benchmarks of this thesis.

	# of weights	# of activations	# of biases
12-class KWS	21,296	704	188
Few-shot KWS	114,352	1,856	432
Sequential Omniglot	116,304	2,688	656

For each of the columns, the maximum is taken as the minimum storage requirement per type: in this case, the design of the network for sequential Omniglot dictates all the maxima. From these maxima and the bit widths for each of the three value types, the closest power-of-two products[29] to the total required bits are derived: the first value in this product indicates the SRAM word size, while the second value indicates the number of SRAM rows. These are 1024×512 (64 kB), 256×64 (2 kB), 64×64 (0.5 kB) for the weights, activations and biases, respectively. This means that taking the weight memory as an example, the shapes 2048×256 and 128×4096 are also added to the search space.

Knowing the required memory sizes, it is possible to loop back to the PE array requirements. As mentioned in Section 5.6.2, for the weight (resp. activation) memory, a word size of $4n^2$ (resp. $4n$) is required to make sure that the PE array can produce new outputs at every cycle. Subsequently, this means that the bias needs a word size of $16n$ to load all the biases for the PE array in a single cycle. However, while the interaction between the PE array and the weight and bias memories is read-only[30], the interaction between the PE array and the activation memory is both read and write.

[29] Power-of-two products are used as most of the available memories have a width and depth that are powers of two.

[30] These values do not change during inference.

Therefore, the weight storage configuration chosen is a single-port SRAM that fulfills the dimensions $4n^2 \times 2^{17}n^{-2}$, while the bias configuration chosen is a single-port SRAM that fulfills the dimensions $16n \times 2^8 n^{-1}$. However, as parallel read and write capabilities are required for the activation memory, a single-port memory will not suffice.

Considering other hardware designs, usually two single-port SRAMs are used in a ping-pong configuration [44] [39] to allow for reading and writing activations to and from the PE array in the same cycle; for processing networks with residual connections, often a third single-port SRAM is added [42] [41]. However, for this design, there is also the possibility of using dual-port memories.

Comparing the two dual-port memories available, i.e. an SRAM and a register file, the former has a 25-% higher leakage current, a 3.0× (resp. 1.5×) higher read (resp. write) current and a 50-% larger area than the latter for the same storage size. When putting the register file option side-by-side with a triple single-port SRAM configuration, the register file solution demonstrates 50% lower area and 5.0× lower leakage. Therefore, the activation storage configuration for Chameleon consists of a single two-port register file with dimensions $n \times 2^{12}n^{-1}$. The final, parameterized storage configuration, including all memories, is then shown in Fig. 70.

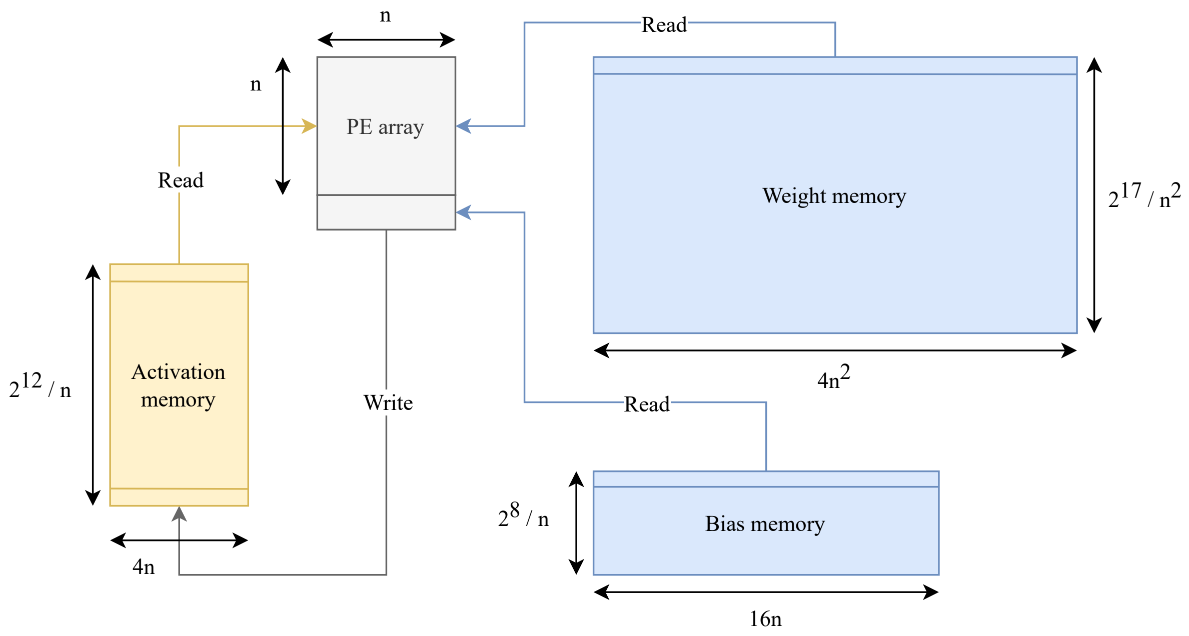


Figure 70: Final parametrized storage configuration of Chameleon, with a single-port weight SRAM and bias SRAM and a two-port activation register file.

5.7.2 PE array size determination

With the parameterized PE array and storage configuration outlined, in this section, the optimal value of n will be determined by considering the maximum TOPS/W and minimum real-time power. To be able to do this, the following assumptions are made:

- the leakage of the PE array is assumed to scale with n^2, where the leakage value from $n = 16$ is used as a baseline value,
- only the leakage power from the memories and PE array is taken into account, as it is assumed that the remaining blocks have negligible leakage,

- the dynamic power of the PE array is assumed negligible during real-time operation and is assumed to be absorbed by the power required for reading the memories during maximum frequency operation,
- the minimum clock speed required to maintain real-time processing is used to calculate the lowest power while the maximum internal clock speed (250 MHz) is used to evaluate maximum TOPS/W,
- an operating voltage of 1.1V is used,
- the activation memory is read from every cycle, but written to only f_{act} % of the cycles,
- the weight memory is read from every cycle, but the bias memory is only read from f_{bias} % of the cycles,
- f_{act} and f_{bias} are task-dependent and computed by simulating the TCN controller,
- the read and write current per cycle from all memories are taken from TSMC documentation,
- the maximum word size and depth for all memories are 128 and 8096 respectively, meaning that in case a larger word size or depth than these values is required, multiple memories are used,
- for each value of n, the lowest power memories possible are selected that fulfill the dimensions and total storage capacity of Section 5.7.1, not considering area.

Fig. 71 shows the results of this approach, where for five values of n, the maximum TOPS/W and minimum power for real-time[31] sequential Omniglot was found. It can be seen that a PE array size of 2 gives the lowest real-time power, with $n = 4$ trailing it closely, but this comes at the cost of peak TOPS/W. Going from $n = 4$ to $n = 8, 16, 32$, at every step, both a 50% increase in minimum power and in maximum TOPS/W is taking place while going from $n = 16$ to $n = 32$ more than doubles the minimum power at less than 20% increase in TOPS/W.

It it then decided to choose both $n = 4$ and $n = 16$, effectively creating a 4×4 subsection in the $n = 16$ PE array. This is made possible by the fact that a separate power rail can be used for some of the memories: all memories that are required for $n = 4$ operation are always-on, while the memories that are only required for $n = 16$ can be turned on and off, depending on the configured effective PE array size. Furthermore, the memories required for $n = 4$ can be reused during $n = 16$ operation. However, the leakage of the PE array for this new $n = 4$ configuration remains the same as for $n = 16$, as the full PE array is always on.

As the total weight and bias storage capacity in the $n = 4$ case is lower than the $n = 16$ case, this mode is specially developed for KWS scenarios with small networks. Fig. 72 then displays the maximum TOPS/W and minimum power for KWS, where it is clear that the use of a subsection mode not only allows for maintaining a high TOPS/W since $n = 16$, it also enables a very low minimum power, even lower than the $n = 2$ case due to the reduced memory size. Furthermore, during $n = 4$ operation, now achieves higher TOPS/W than $n = 8$.

Therefore, $n = 16$ is chosen for Chameleon. In the remainder of this section, it is then covered, per memory type, how the values for both regular and subsection operations are stored.

[31] 2.6 images per second (about 2000 time steps per second), which was empirically found by writing 100 characters.

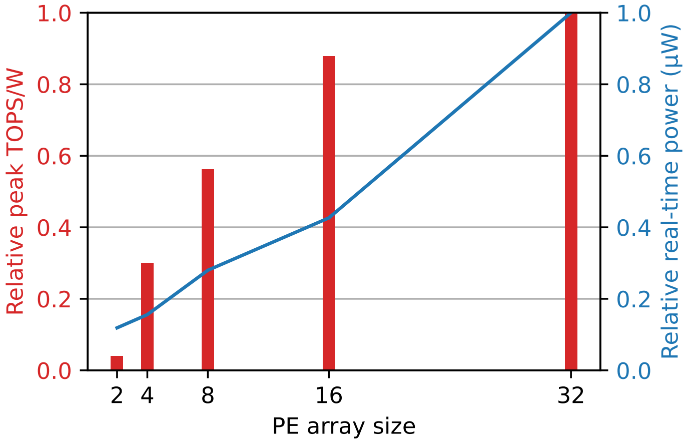


Figure 71: Maximum TOPS/W and minimum real-time power for $n = 2,4,8,16,32$ while performing sequential Omniglot. Both y-axes are made relative to not display any technology-dependent information.

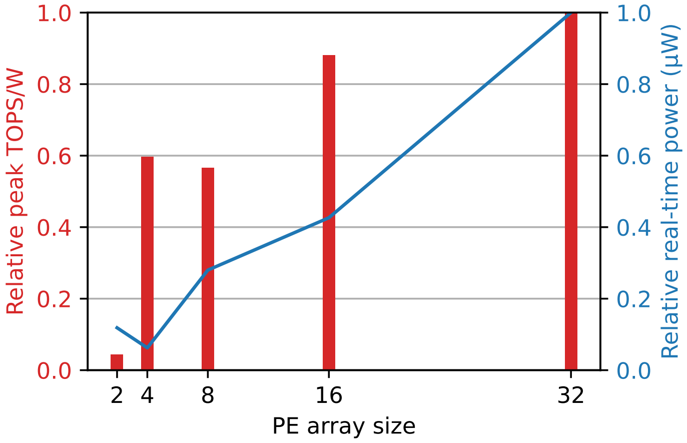


Figure 72: Maximum TOPS/W and minimum real-time power for $n = 2,4,8,16,32$ while performing KWS. Both y-axes are made relative to not display any technology-dependent information.

5.7.3 Weight storage

In this section, the neural network weights storage is discussed. As explained in Section 4.4.3, the neural network weights that can be used in Chameleon are 4 bits wide and since the final matrix-vector multiplier can perform a multiplication of a 16D vector with a 16×16 matrix every cycle, the total word size is 1024. This is achieved by horizontally (word-wise) stacking 8 memories with word size 128 and 512 rows.

Fig. 73 shows how the weights from a single memory word of 256 elements of 4 bits are inserted into the 16×16 PE array. In situations where the kernel size is a multiple of 16×16, multiple words are used as shown in Fig. 74, where the words are stored in a row-kernel-column order to maintain output stationary when using subsequent weights.

Fig. 75 shows how, in order to support 4×4 operation, one of the 128×512 memories is replaced by two 64×512 memories that are stacked depthwise during 4×4 operation, with all the other weight memories switched off. This means that in 4×4-mode, a network with a maximum size weight count of 16,384 can be deployed. The current KWS network has

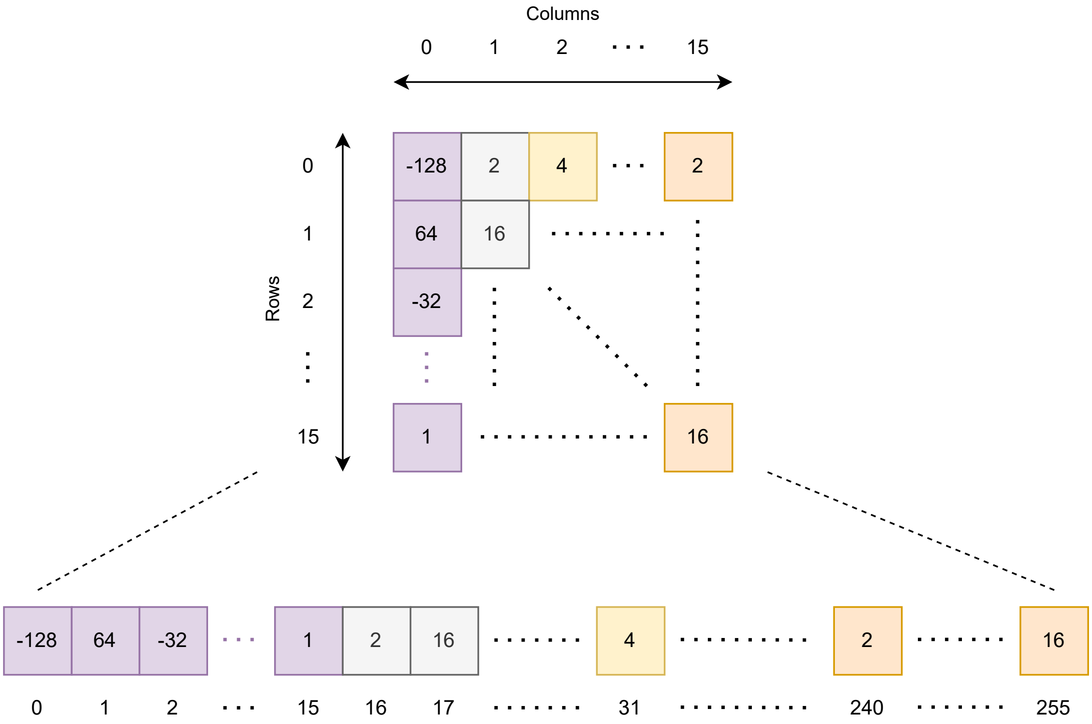


Figure 73: Overview of insertion of 16 × 16 word into the PE array.

21,296 parameters and therefore future work will investigate smaller TCNs that still achieve good classification accuracy, to make full use of the subsection mode.

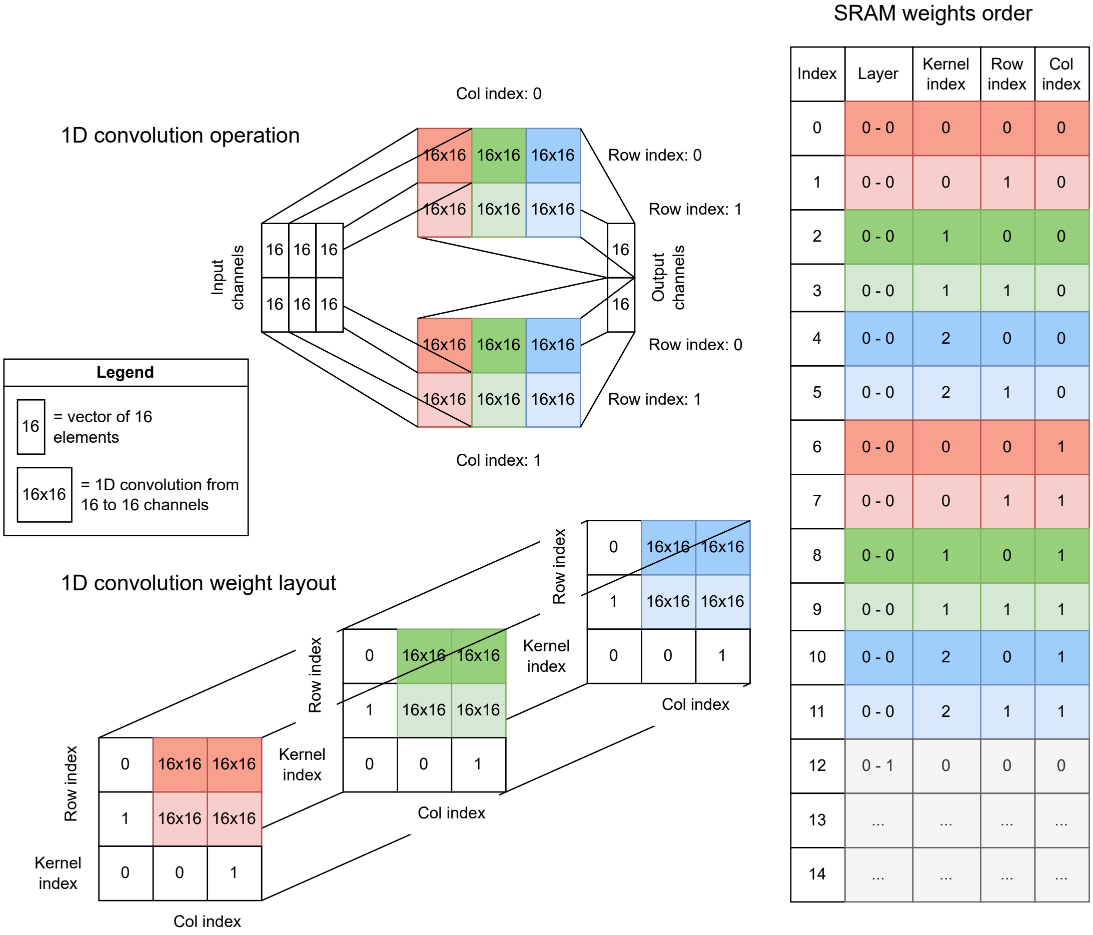
5.7.4 Bias storage

In this section, the neural network bias storage is discussed. As explained in Section 4.4.3, the neural network biases that can be used in Chameleon are 16 bits wide and since the final matrix-vector multiplier can perform an addition with a 16D vector with every cycle, the total word size is 256. This is achieved by horizontally (word-wise) stacking two memories with a word size of 128 and 128 rows. This is larger than the required memory size for the biases as for future work[32], support is required for very deep networks with a large (> 1000) number of biases.

Thus far the storage of the scale values for quantized inference has not been touched upon. While they could also be stored in an array of registers, it was decided to store them together with the biases as each bias is loaded at the same time a scale value is needed, saving energy and area. However, since 4-bit scale values (**CHA-NUMERIC-0**) are required and since the entire word size is already used by the biases, it was decided to instead use a bias bit-width of 15 to allow for storing the 4-bit scale.

Furthermore, as shown in Fig. 76, to support both 16 × 16 and 4 × 4 operations, similar to the weight memories, one of the 128 × 128 memories is split into two 64 × 128 (splitting them has less than 5% overhead in terms of leakage and read power) and the effective layout can be reconfigured at run-time.

[32] Raw audio keyword spotting at 16 kHZ



Index	Layer	Kernel index	Row index	Col index
0	0 - 0	0	0	0
1	0 - 0	0	1	0
2	0 - 0	1	0	0
3	0 - 0	1	1	0
4	0 - 0	2	0	0
5	0 - 0	2	1	0
6	0 - 0	0	0	1
7	0 - 0	0	1	1
8	0 - 0	1	0	1
9	0 - 0	1	1	1
10	0 - 0	2	0	1
11	0 - 0	2	1	1
12	0 - 1	0	0	0
13	...	...	...	...
14	...	...	...	...

Figure 74: Weight memory layout for 16×16 operation.

5.7.5 Activation storage

In this section, the neural network activation storage is discussed. As explained in Section 5.7.1, a two-port register file is used for this. With $n = 16$, this means that the register file has a word size of 64 and a depth of 256. However, while studying the access pattern induced by the TCN controller (Fig. 77), it was shown that the feature-vectors inputs for inference are accessed significantly (twice as often) more than the most accessed intermediate activation address: in total, for this network, the inputs are read from 1200 times while the intermediate activations are only read 700 times. Therefore, to save energy on these input reads, it was decided to, next to the two-port register file, also use a separate register-based input memory.

Furthermore, due to the real-time processing support of Chameleon, situations can occur where both the input memory and the activation memory need to be written to in parallel, the former to store a new input and the latter to store an output from the PE array, causing write-contentions. However, this streaming nature also allows for a small input memory, as only $k \times$ input channel count values have to be kept in the input memory at all times, instead of the entire input sequence, which is the case for 1D ResNets [63] as they cannot be processed stream-wise. The number of rows in this input memory was determined to be 32 (0.25 kB): this means that networks with a $k \times$ input channel count$/16 < 32$ are supported, for example, a network with $k = 7$ and 64 input channels fits into this memory.

It was also considered splitting the two-port register file into multiple pieces to more efficiently support the 4×4 subsection mode, as the input memory is already able to store a

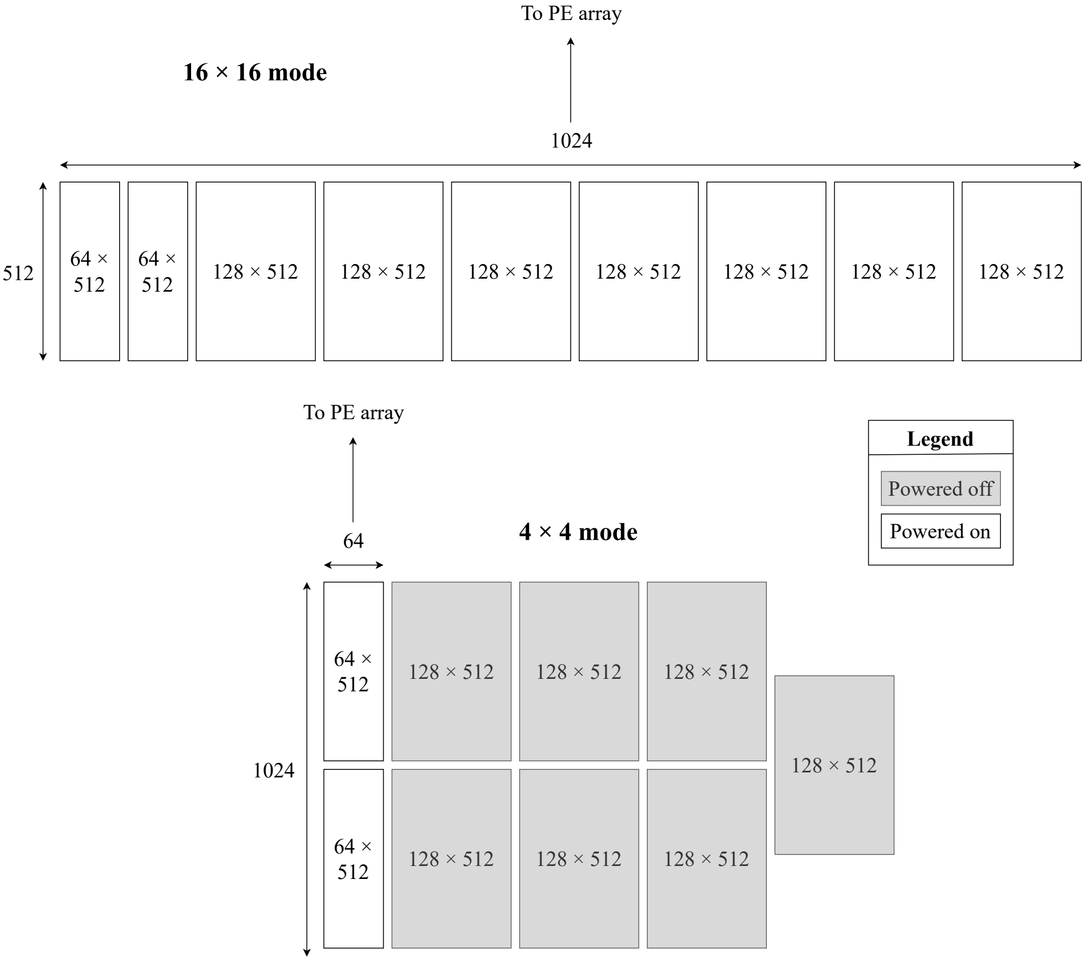


Figure 75: Weight memory layout reconfiguration for 16 × 16 and 4 × 4 operation.

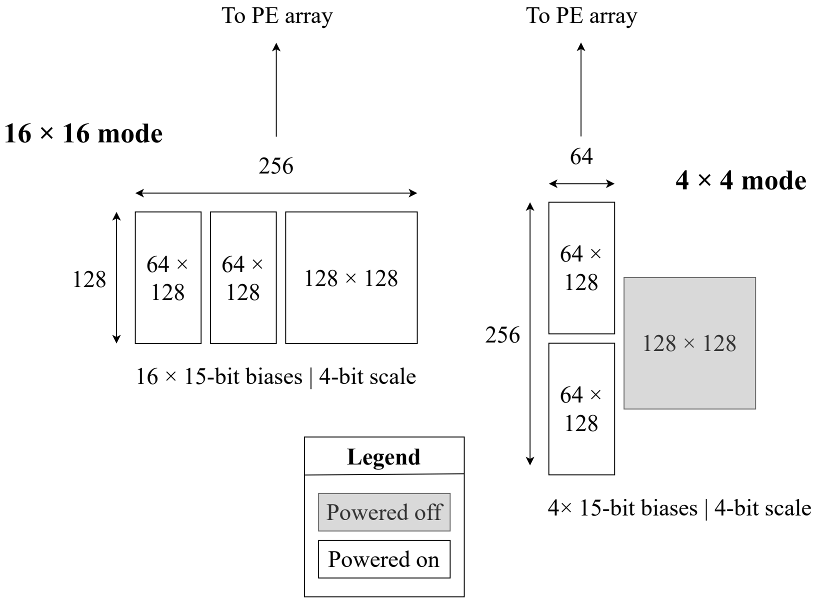


Figure 76: Bias memory layout reconfiguration for 16 × 16 and 4 × 4 operation.

significant amount of the total used values and since the two most suitable split leads to 20% higher leakage and 20% read current, it was decided to stick with one activation memory.

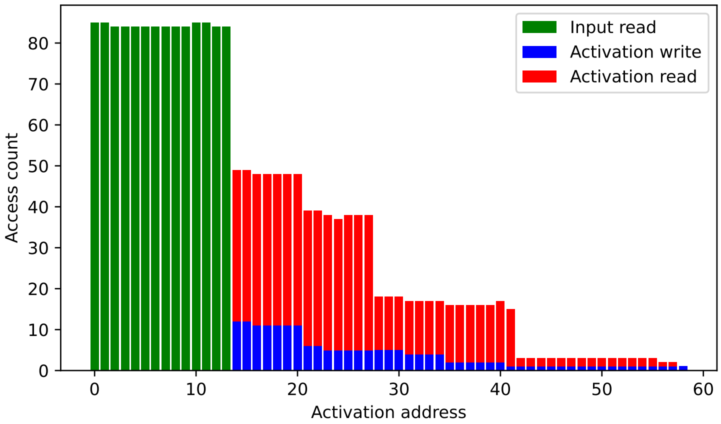


Figure 77: Access count per activation address during 16×16 operation, split between inputs and intermediate activations.

5.8 Few-shot learning

In this section, the hardware implementation for on-chip few-shot learning is discussed (**CHA-CORE-03**). As there exists no accelerator-style hardware design yet that performs few-shot learning by comparing embeddings with Euclidean distance, an approach from scratch had to be developed.

Ideally, few-shot learning can be integrated in such a way that it does not introduce any extra inference/meta-test cycles compared to regular inference while requiring little extra hardware. Since Chameleon already supports inference of TCNs with linear layers and since computing the Euclidean distance[33] between an embedding and a prototype can be performed as a linear layer (see Equation 51), it was chosen to pursue an implementation based on this.

A key advantage of this linear layer mapping is that the computation can simply be controlled by the TCN controller and performed in the PE array. This approach is also suitable for regular inference accelerators, without requiring extra hardware and complex control. A major downside for Chameleon however, is that extra multipliers are required in the PE array, on top of the shifters, to perform the Euclidean distance computation, as all activations are positive integers (i.e. not in logarithmic format). However, as Chameleon is an experimental design, the trade-off was made to include multipliers in the PE array to demonstrate this mapping, accepting the leakage and area overhead.

Before this linear layer computation can be done, first the bias and weight vector per class have to be computed from the support embeddings. This is handled by the few-shot learning controller, as it is not part of regular TCN inference. The implemented few-shot learning controller is effectively an address-generator module, similar to the TCN controller (Section 5.5). After waiting until all the embeddings from the specified number of support shots have been computed and stored in the activation memory (always outside the address range used for TCN inference), it initiates the prototype computation. In this operational phase, all control signals from the TCN controller are overridden by those of the few-shot learning controller.

To then compute the prototype/average support embedding, every support embedding is fed into the PE array, multiplied by $1/k$ and accumulated in the accumulators. The division by k is done via a LUT that controls the PE array: for example, a division by 5 is performed

[33] As the full results of the experiments in Section 4.3.2 were not available by the tapeout deadline, the Euclidean distance was implemented in hardware as it is a standard choice [27] [126] that is hardware-friendly.

by multiplying all support embeddings, prior to accumulation, with 13, after which the final accumulated value is scaled by 2^{-6}: this results in an effective division by 5, since $13 \cdot 2^{-6} \approx 0.20 = \frac{1}{5}$.

The separate entries of the embedding vector are the squared and summed in one cycle and written to the bias memory, while at the same time, the averaged embedding is written to the weight memory. While performing inference on unseen sample belonging to one of the learned classes, the embedding computation is followed by a linear layer computation using the learned weights and biases, from which the class can be determined.

Chameleon is designed to be ways-agnostic: before performing few-shot learning, the number of ways that will be learned does not have to be specified. Therefore, Chameleon also has a natural continual learning extension: next to a pre-trained classification layer, any number of extra classes can be learned. In practice, the maximum number of ways is limited to the maximum number of ways regular classification supports, which is 256.

5.9 Comparison with the state of the art

In this section, the final silicon implementation of Chameleon is compared to state-of-the-art keyword spotting accelerators and few-shot learning hardware. Fig. 78 shows the layout of the Chameleon chip with the top mask image overlaid.

First, in Table 21, Chameleon is put side-by-side with other KWS accelerators. It can be seen that Chameleon's 0.954 mm^2 post-shrink core area is relatively similar to other designs, while Laika [40] from Giraldo *et al.* stands out with a 3× lower core area than Chameleon, normalized to a 40-nm node. Furthermore, the total on-chip memory for Chameleon comparable to that of the other designs.

Looking at the peak efficiency in GOPS/W of the accelerators in Table 21, it can be seen that Chameleon comes out on top, being about 40% more efficient than TinyVers [39]. Regarding real-time power usage, the LSTM accelerator from Giraldo *et al.* demonstrates the lowest value for a KWS accelerator, although it can only classify four keywords. From the accelerators that have been demonstrated on GSC, Vocell [94], also from Giraldo *et al.* demonstrates the lowest power metric. For Chameleon, two values are reported, these are the values for assumed[34] 4×4 and performed 16×16 operation. Comparing the clock speeds required for real-time operation, it can be seen that Chameleon can operate at a relatively low frequency with respect to the other hardware accelerators, even in 4×4-mode.

Furthermore, Chameleon demonstrates the best classification accuracy from all silicon designs, only losing out to the TENet implementation [42] on an FPGA. Most importantly, it is the only design that is not just able to perform regular KWS, but also few-shot KWS, at an accuracy of 41.5% (better than the previous FP32 state-of-the-art [46] of 39.54 ± 0.62 %) while only receiving one example for 10 keywords. This comparison demonstrates that Chameleon can perform few-shot learning at the edge while maintaining high efficiency for low-cost KWS inference compared to state-of-the-art inference-only designs.

Comparing Chameleon with other few-shot learning hardware in Table 22, it can be seen that Chameleon is the only fabricated silicon implementation. Looking at the supported number of ways and shots, Chameleon is the most flexible, as it supports a maximum number of ways and shots of 256 and 32 respectively. Furthermore, considering the accuracy of Chameleon, it can be seen that the only approach that significantly outperforms it, is deployed on an NVIDIA Jetson Nano, which is by no means edge-friendly in terms of power usage.

[34] Assumed, as the current KWS network cannot fit in the memories associated to the 4×4-mode.

Overall, Chameleon demonstrates high accuracy on few-shot learning tasks, while operating at a minimum power budget without negatively impacting regular inference performance.

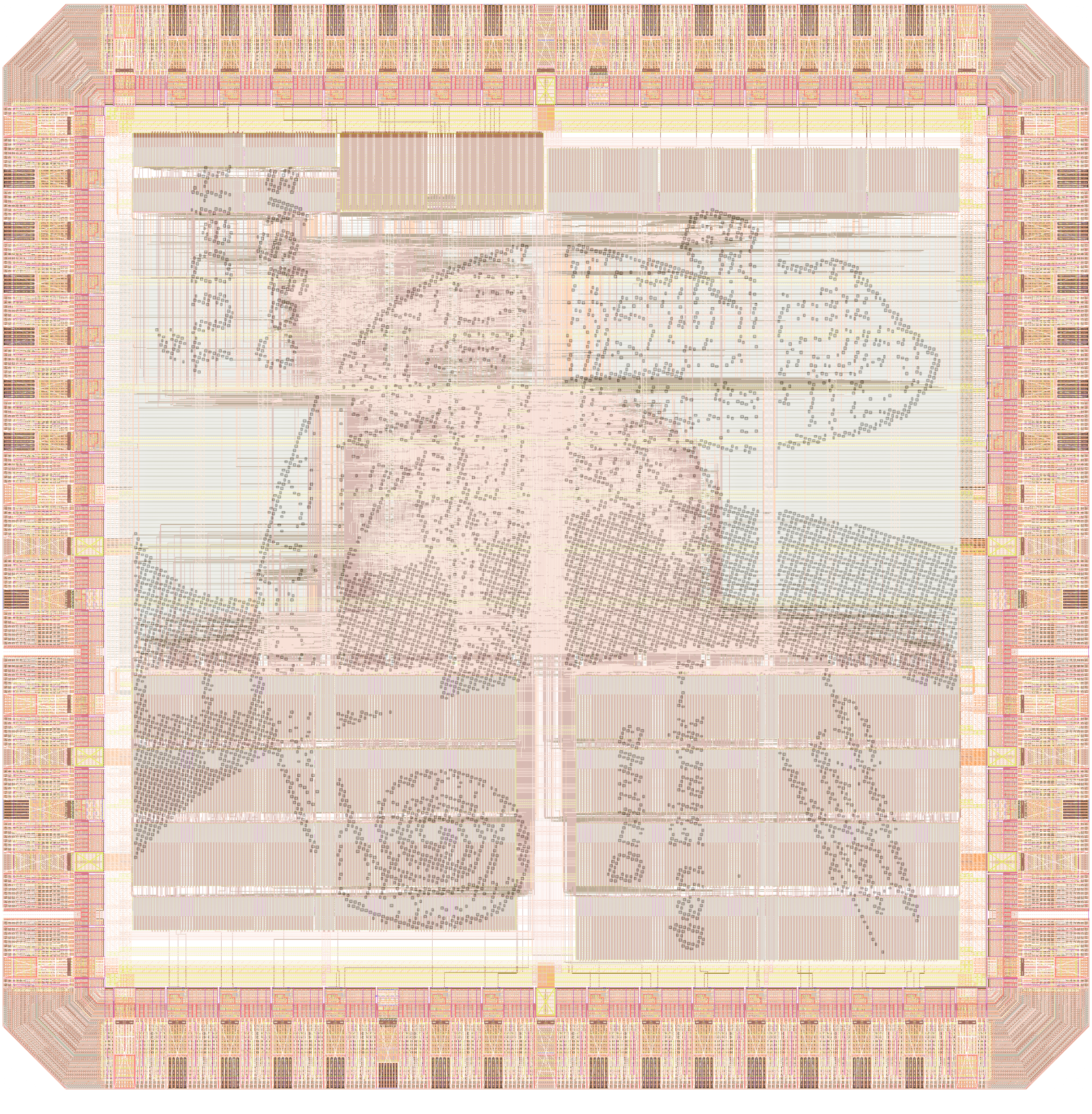


Figure 78: Layout of Chameleon captured inside Virtuoso, where the top-layer metal mask has been overlaid for visibility.

Table 21: Comparison of hardware architectures for keyword spotting. * indicates an estimated value, while '-' indicates missing information.

	Laika [40]	**Vocell [94]**	**UltraTrail [41]**	**TENet on FPGA [42]**	**TinyVers [39]**	**Chameleon (this work)**
Publication	ESSCIRC'2018	JSSC'2020	TCAD'2020	MDPI Electronics'2022	JSSC'2023	TBD
Silicon results	**Yes**	**Yes**	No (simulation only)	No (FPGA)	**Yes**	No (not yet)
Technology	65 nm	65 nm	GlobalFoundries 22 nm	Xilinx ZYNQ 7Z020	GlobalFoundries 22nm	TSMC 40 nm
Core area (mm^2)	**0.76***	2.56	0.2	N.A. (FPGA)	6.25	0.954
Core area normalized to 40 nm (mm^2)	**0.288***	0.969	0.661	N.A. (FPGA)	20.66	0.954
Voltage (V)	0.575	0.6	0.8	N.A. (FPGA)	0.4-0.9	1.1
Peak efficiency (Gops/W)	-	-	-	5.36	1,050	**1,396**
On-chip memory (kB)	**32**	97	75.904	29.44	132	69.632
Real-time power (μW)	**5**	10.6	8.2	209,000	193	17.2 / 46.5
Real-time clock speed (kHz)	250	250	250	90	-	**4.38 / 17.50**
Runtime per window (cycles)	> 5,376*	> 4,976*	22,481	7,266	-	**70 / 280**
Latency (ms)	**16**	**16**	100	100	**11**	**16**
On-chip MFCC?	**Yes**	**Yes**	No	No	No	No
MFCC (dim., stride, window)	39D/16ms/32ms	13D/16ms/32ms	40D/10ms/30ms	30D/16ms/16ms	40D/32ms/16ms*	32D/16ms/32ms
Network	LSTM, $\dim(h) = 56$	LSTM, $\dim(h) = 64$	TC-ResNet8	TENet	TCN	TCN
# of parameters	21,504*	19,968	66,000	18,000	23,000*	21,484
Quantization scheme	N4 / S8* / ?	N4 / S8* / ?	S6 / U8 / S8	S8 / U*8 / S8*	S8* / U8* / ?	L4 / U4 / S14
# of keywords	4	**10+2**	**10+2**	**10+2**	**10+2**	**10+2**
Classification accuracy (%)	90.00 (TIMIT)	90.87 (GSC)	93.09 (GSC)	**95.98 (GSC)**	93.3 (GSC)	93.5 (GSC)
1-shot KWS accuracy (%)	N.A.	N.A.	N.A.	N.A.	N.A.	**41.46 ± 0.31**
# cycles for few-shot learning	N.A.	N.A.	N.A.	N.A.	N.A.	**3·ways·shots**

Table 22: Comparison of hardware-tested approaches for few-shot learning. * indicates an estimated value, while '-' indicates missing information.

	Lungu *et al.* [127]	**Stewart *et al.* [128]**	**AIQ [6]**	**Reis *et al.* [129]**	**Chameleon (this work)**
Publication	JETCAS'2020	AICAS'2020	DATE'2021	DATE'2020	TBD
Silicon results	No	No	No	No (simulation only)	**No (not yet)**
Technology	NVIDIA Jetson Nano	Intel Loihi	TSMC 45nm	14nm FinFET	TSMC 40nm
Real-time power (µW)	-	-	17,720	-	46.50
Few-shot learning approach	Siamese networks	Surrogate gradient descent	MAML [15]	ProtoNets [27]	PTE
Maximum ways/shots	∞ / ∞ *	11 / 20 *	20 / 5 *	5 / 5 *	256 / 32
Network	8-layer ConvNet	Spiking ConvNet	64-64-64-64	64-64-64-64	TCN
# of parameters	$4.75 \cdot 10^6$	-	111,936	111,936	116,960
Omniglot accuracy (%)					
5-way, 1-shot	**99.1 ± 1.3**	N.A	97.0	98.35	96.57 ± 0.21
5-way, 5-shot	-	N.A	99.2	-	**99.34 ± 0.09**
20-way, 1-shot	-	N.A	-	-	**90.54 ± 0.17**
20-way, 5-shot	-	N.A	95.5	-	**97.94 ± 0.08**

6 Conclusion and outlook

In this section, first, a brief summary of the work presented in this thesis will be given, after which a conclusion is drawn and the latest trends (Section 6.1) and possible directions for future work (Section 6.2) are provided.

Coming back to the original research question ("How can meta-learning unlock low-cost adaptation to temporal data for deep neural network accelerator hardware at the edge?"), the following three key claims are extracted:

1. it was demonstrated that TCNs, compared to RNNs, provide a low-cost neural network architecture for few-shot learning over *sequential* data at the edge. TCNs not only outperform all recurrent architectures on the four discussed benchmarks, they also have a more favorable scaling of operations with input sequence length, compared to recurrent architectures.

2. Building on recent results from machine learning research, it was then shown how meta-learning techniques chiefly rely on learning high-quality features. Considering hardware-related factors like memory and computing resources, it was concluded that a supervised pre-training approach, involving on-chip learning by comparing network outputs via Manhattan distance, achieves the best performance-cost trade-off. Additionally, it was demonstrated that a resolution as low as 4 bits for the weights and activations can be used thanks to logarithmic weight quantization.

3. The hardware implementation then brings the network architecture and meta-learning approach together, in a 40-nm sub-mm^2 chip code-named *Chameleon*. Chameleon is both the first in-silico TCN accelerator and the first accelerator to perform on-chip temporal data classification via meta-learning and continual learning.

Demonstrating a low-cost and data-efficient on-chip learning scheme for temporal data classification has the potential to positively impact edge devices in multiple avenues, by increasing data-privacy, reducing electronic waste and maintenance costs and allowing for adaptive operation even in areas with limited connectivity.

6.1 Latest trends

Below, two of the latest trends for some of the aspects in this work are provided:

- equiangular basis vectors (EBV) [130] for few-shot distance-based classification: EBV is a drop-in replacement for a linear layer using fixed equi-distance embeddings between classes,

- pre-training the embedder on a larger spoken keywords dataset, such as MSWC [43]. This has recently been investigated by Rusci *et al.* [131].

6.2 Future work

In the following list, an overview of possible directions for future work is provided:

- distilled quantization [132], where quantization and distillation are performed in parallel to increase the quantized model's performance,

- using a projector layer (a second linear layer) as is often done in self-supervised learning [133],
- an investigation into the optimal embedding dimensionality for Euclidean distance comparison,
- fine-tuning the embedder after PTE using ProtoNets [27], as done in [126] by Hu *et al.*,
- keyword spotting on raw audio data,
- pre-training the embedder via unsupervised learning, as done in [98],
- experiment with on-chip linear-to-logarithmic conversion to allow for removal of multipliers in the design.

References

[1] Nitish Satya Murthy, Peter Vrancx, Nathan Laubeuf, Peter Debacker, Francky Catthoor, and Marian Verhelst. Learn to Learn on Chip: Hardware-aware Meta-learning for Quantized Few-shot Learning at the Edge. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, pages 14–25, December 2022.

[2] 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, December 2021. Conference Name: IEEE Journal on Emerging and Selected Topics in Circuits and Systems.

[3] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[4] 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.

[5] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

[6] Mengyuan Li and Xiaobo Sharon Hu. A Quantization Framework for Neural Network Adaption at the Edge. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 402–407, February 2021. ISSN: 1558-1101.

[7] Miguel de Prado, Manuele Rusci, Alessandro Capotondi, Romain Donze, Luca Benini, and Nuria Pazos. Robustifying the deployment of tinyml models for autonomous mini-vehicles. *Sensors*, 21(4):1339, 2021.

[8] Vincenzo Lomonaco, Lorenzo Pellegrini, Pau Rodriguez, Massimo Caccia, Qi She, Yu Chen, Quentin Jodelet, Ruiping Wang, Zheda Mai, David Vazquez, German I. Parisi, Nikhil Churamani, Marc Pickett, Issam Laradji, and Davide Maltoni. CVPR 2020 Continual Learning in Computer Vision Competition: Approaches, Results, Current Challenges and Future Directions, September 2020. arXiv:2009.09929 [cs, stat].

[9] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[10] OpenAI. Gpt-4 technical report, 2023.

[11] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David

Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models, July 2023. arXiv:2307.09288 [cs].

[12] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-Learning in Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5149–5169, September 2022. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[13] Tong Yu and Hong Zhu. Hyper-Parameter Optimization: A Review of Algorithms and Applications, March 2020. arXiv:2003.05689 [cs, stat].

[14] Sebastian Thrun and Lorien Pratt. *Learning to Learn: Introduction and Overview*, pages 3–17. Springer US, Boston, MA, 1998.

[15] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks, July 2017. arXiv:1703.03400 [cs].

[16] Marc Ruswurm, Sherrie Wang, Marco Korner, and David Lobell. Meta-Learning for Few-Shot Land Cover Classification. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 788–796, Seattle, WA, USA, June 2020. IEEE.

[17] Alex Nichol, Joshua Achiam, and John Schulman. On First-Order Meta-Learning Algorithms, October 2018. arXiv:1803.02999 [cs].

[18] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your MAML, March 2019. arXiv:1810.09502 [cs, stat].

[19] Chen Fan, Parikshit Ram, and Sijia Liu. Sign-MAML: Efficient Model-Agnostic Meta-Learning by SignSGD.

[20] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-SGD: Learning to Learn Quickly for Few-Shot Learning, September 2017. arXiv:1707.09835 [cs].

[21] Eunbyung Park and Junier B. Oliva. Meta-Curvature, January 2020. arXiv:1902.03356 [cs, stat].

[22] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A Simple Neural Attentive Meta-Learner, February 2018. arXiv:1707.03141 [cs, stat].

[23] David Ha, Andrew Dai, and Quoc V. Le. HyperNetworks, December 2016. arXiv:1609.09106 [cs].

[24] Dominic Zhao, Seijin Kobayashi, and João Sacramento. Meta-Learning via Hypernetworks.

[25] Siyuan Qiao, Chenxi Liu, Wei Shen, and Alan Yuille. Few-Shot Image Recognition by Predicting Parameters from Activations, November 2017. arXiv:1706.03466 [cs].

[26] Andrei A. Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. Meta-Learning with Latent Embedding Optimization, March 2019. arXiv:1807.05960 [cs, stat].

[27] Jake Snell, Kevin Swersky, and Richard S. Zemel. Prototypical Networks for Few-shot Learning, June 2017. arXiv:1703.05175 [cs, stat].

[28] Wei-Yu Chen, Yen-Cheng Liu, Zsolt Kira, Yu-Chiang Frank Wang, and Jia-Bin Huang. A Closer Look at Few-shot Classification. Technical Report arXiv:1904.04232, arXiv, January 2020. arXiv:1904.04232 [cs] type: article.

[29] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching Networks for One Shot Learning. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[30] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip H. S. Torr, and Timothy M. Hospedales. Learning to Compare: Relation Network for Few-Shot Learning, March 2018. arXiv:1711.06025 [cs].

[31] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, December 2015.

[32] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. Conference Name: Proceedings of the IEEE.

[33] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. The Omniglot challenge: a 3-year progress report, May 2019. arXiv:1902.03477 [cs].

[34] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units, April 2015. arXiv:1504.00941 [cs].

[35] Pete Warden. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition, April 2018. arXiv:1804.03209 [cs].

[36] Yuan Gong, Yu-An Chung, and James Glass. AST: Audio Spectrogram Transformer, July 2021. arXiv:2104.01778 [cs].

[37] Axel Berg, Mark O'Connor, and Miguel Tairum Cruz. Keyword Transformer: A Self-Attention Model for Keyword Spotting. In *Interspeech 2021*, pages 4249–4253. ISCA, August 2021.

[38] Byeonggeun Kim, Simyung Chang, Jinkyu Lee, and Dooyong Sung. Broadcasted Residual Learning for Efficient Keyword Spotting, October 2022. arXiv:2106.04140 [cs, eess] version: 3.

[39] Vikram Jain, Sebastian Giraldo, Jaro De Roose, Linyan Mei, Bert Boons, and Marian Verhelst. TinyVers: A Tiny Versatile System-on-chip with State-Retentive eMRAM for ML Inference at the Extreme Edge. *IEEE Journal of Solid-State Circuits*, pages 1–12, 2023. arXiv:2301.03537 [cs].

[40] J.S.P Giraldo and Marian Verhelst. Laika: A 5uW Programmable LSTM Accelerator for Always-on Keyword Spotting in 65nm CMOS. In *ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC)*, pages 166–169, September 2018. ISSN: 1930-8833.

[41] Paul Palomero Bernardo, Christoph Gerum, Adrian Frischknecht, Konstantin Lübeck, and Oliver Bringmann. UltraTrail: A Configurable Ultralow-Power TC-ResNet AI Accelerator for Efficient Keyword Spotting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4240–4251, November 2020. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[42] Keyan He, Dihu Chen, and Tao Su. A Configurable Accelerator for Keyword Spotting Based on Small-Footprint Temporal Efficient Neural Network. *Electronics*, 11(16):2571, January 2022. Number: 16 Publisher: Multidisciplinary Digital Publishing Institute.

[43] Mark Mazumder, Sharad Chitlangia, Colby Banbury, Yiping Kang, Juan Manuel Ciro, Keith Achorn, Daniel Galvez, Mark Sabini, Peter Mattson, David Kanter, Greg Diamos, Pete Warden, Josh Meyer, and Vijay Janapa Reddi. Multilingual spoken words corpus. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[44] J. S. P. Giraldo, Vikram Jain, and Marian Verhelst. Efficient Execution of Temporal Convolutional Networks for Embedded Keyword Spotting. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(12):2220–2228, December 2021.

[45] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*, 2017.

[46] Yangbin Chen, Tom Ko, Lifeng Shang, Xiao Chen, Xin Jiang, and Qing Li. An Investigation of Few-Shot Learning in Spoken Term Classification, September 2020. arXiv:1812.10233 [cs] version: 3.

[47] Mark Mazumder, Colby Banbury, Josh Meyer, Pete Warden, and Vijay Janapa Reddi. Few-Shot Keyword Spotting in Any Language. In *Interspeech 2021*, pages 4214–4218, August 2021. arXiv:2104.01454 [cs, eess].

[48] Jaemin Jung, Youkyum Kim, Jihwan Park, Youshin Lim, Byeong-Yeol Kim, Youngjoon Jang, and Joon Son Chung. Metric Learning for User-defined Keyword Spotting, November 2022. arXiv:2211.00439 [cs, eess].

[49] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015.

[50] Dongjune Lee, Minchan Kim, Sung Hwan Mun, Min Hyun Han, and Nam Soo Kim. Fully Unsupervised Training of Few-shot Keyword Spotting, October 2022. arXiv:2210.02732 [eess].

[51] Abhijeet Awasthi, Kevin Kilgour, and Hassan Rom. Teaching keyword spotters to spot new keywords with limited examples, June 2021. arXiv:2106.02443 [cs, eess].

[52] Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory Aware Synapses: Learning What (not) to Forget. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, volume 11207, pages 144–161. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.

[53] 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.

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

[55] Khurram Javed and Martha White. Meta-Learning Representations for Continual Learning, October 2019. arXiv:1905.12588 [cs, stat].

[56] Zhixiang Chi, Li Gu, Huan Liu, Yang Wang, Yuanhao Yu, and Jin Tang. MetaFSCIL: A Meta-Learning Approach for Few-Shot Class Incremental Learning. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14146–14155, New Orleans, LA, USA, June 2022. IEEE.

[57] Rodriguez Garcia Arturo Ramirez Lopez Leonardo Juan, Puerta Aponte Gabriel. Internet of things applied in healthcare based on open hardware with low-energy consumption. *hir*, 25(3):230–235, 2019.

[58] Zeba Idrees, Zhuo Zou, and Lirong Zheng. Edge Computing Based IoT Architecture for Low Cost Air Pollution Monitoring Systems: A Comprehensive System Analysis, Design Considerations & Development. *Sensors*, 18(9):3021, September 2018. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

[59] Juan P. Dominguez-Morales, Lourdes Duran-Lopez, Daniel Gutierrez-Galan, Antonio Rios-Navarro, Alejandro Linares-Barranco, and Angel Jimenez-Fernandez. Wildlife Monitoring on the Edge: A Performance Evaluation of Embedded Neural Networks on Microcontrollers for Animal Behavior Classification. *Sensors*, 21(9):2975, January 2021. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

[60] Ruimin Ke, Chenxi Liu, Hao Yang, Wei Sun, and Yinhai Wang. Real-Time Traffic and Road Surveillance With Parallel Edge Intelligence. *IEEE Journal of Radio Frequency Identification*, 6:693–696, 2022. Conference Name: IEEE Journal of Radio Frequency Identification.

[61] Devashree R. Patrikar and Mayur Rajaram Parate. Anomaly detection using edge computing in video surveillance system: review. *International Journal of Multimedia Information Retrieval*, 11(2):85–110, June 2022.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. June 2017.

[63] Seungwoo Choi, Seokjun Seo, Beomjun Shin, Hyeongmin Byun, Martin Kersner, Beomsu Kim, Dongyoung Kim, and Sungjoo Ha. Temporal Convolution for Real-time Keyword Spotting on Mobile Devices, November 2019. arXiv:1904.03814 [cs, eess].

[64] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling, April 2018. arXiv:1803.01271 [cs].

[65] Zhiwei Wang, Yao Ma, Zitao Liu, and Jiliang Tang. R-Transformer: Recurrent Neural Network Enhanced Transformer, July 2019. arXiv:1907.05572 [cs, eess].

[66] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. arXiv:1810.04805 [cs].

[67] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. *arXiv preprint arXiv:1905.07799*, 2019.

[68] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A Generative Model for Raw Audio, September 2016. Number: arXiv:1609.03499 arXiv:1609.03499 [cs].

[69] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent Advances in Recurrent Neural Networks, February 2018. arXiv:1801.01078 [cs].

[70] Yoshua Bengio and Yann Lecun. *Scaling Learning Algorithms towards AI*. MIT Press, 2007.

[71] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[72] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[73] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches, October 2014. arXiv:1409.1259 [cs, stat].

[74] Marco Carreras, Gianfranco Deriu, Luigi Raffo, Luca Benini, and Paolo Meloni. Optimizing Temporal Convolutional Network Inference on FPGA-Based Accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(3):348–361, September 2020.

[75] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015.

[76] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[77] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks, 2016.

[78] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A White Paper on Neural Network Quantization, June 2021. arXiv:2106.08295 [cs].

[79] Mark Horowitz. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, February 2014. ISSN: 2376-8606.

[80] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[81] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[82] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[83] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael W. Mahoney, and Kurt Keutzer. HAWQV3: Dyadic Neural Network Quantization, June 2021. arXiv:2011.10680 [cs].

[84] Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*, 2016.

[85] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[86] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional Neural Networks using Logarithmic Data Representation. *arXiv preprint arXiv:1603.01025*, March 2016. arXiv:1603.01025 [cs].

[87] Dominika Przewlocka-Rus, Syed Shakib Sarwar, H. Ekin Sumbul, Yuecheng Li, and Barbara De Salvo. Power-of-Two Quantization for Low Bitwidth and Hardware Compliant Neural Networks, March 2022. arXiv:2203.05025 [cs].

[88] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[89] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[90] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[91] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International conference on machine learning*, pages 1120–1128. PMLR, 2016.

[92] Nesma M. Rezk, Madhura Purnaprajna, Tomas Nordström, and Zain Ul-Abdin. Recurrent Neural Networks: An Embedded Computing Perspective. *IEEE Access*, 8:57967–57996, 2020. Conference Name: IEEE Access.

[93] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[94] Juan Sebastian P. Giraldo, Steven Lauwereins, Komail Badami, and Marian Verhelst. Vocell: A 65-nm Speech-Triggered Wake-Up SoC for 10- W Keyword Spotting and Speaker Verification. *IEEE Journal of Solid-State Circuits*, 55(4):868–878, April 2020. Conference Name: IEEE Journal of Solid-State Circuits.

[95] Klaus Greff, Rupesh K. Srivastava, and Jürgen Schmidhuber. Highway and Residual Networks learn Unrolled Iterative Estimation, March 2017. arXiv:1612.07771 [cs].

[96] Meiqi Wang, Ruixin Xue, Jun Lin, and Zhongfeng Wang. Exploring Quantization in Few-Shot Learning. In *2020 18th IEEE International New Circuits and Systems Conference (NEWCAS)*, pages 279–282, June 2020.

[97] Aniruddh Raghu, Maithra Raghu, Samy Bengio, and Oriol Vinyals. Rapid Learning or Feature Reuse? Towards Understanding the Effectiveness of MAML, February 2020. arXiv:1909.09157 [cs, stat].

[98] Yonglong Tian, Yue Wang, Dilip Krishnan, Joshua B. Tenenbaum, and Phillip Isola. Rethinking Few-Shot Image Classification: A Good Embedding is All You Need? In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, volume 12359, pages 266–282. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

[99] Sachin Ravi and H. Larochelle. Optimization as a model for few-shot learning. In *International Conference on Learning Representations*, 2016.

[100] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie. The caltech-ucsd birds-200-2011 dataset. Technical Report CNS-TR-2011-001, California Institute of Technology, 2011.

[101] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[102] Tommaso Furlanello, Zachary Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born again neural networks. In *International Conference on Machine Learning*, pages 1607–1616. PMLR, 2018.

[103] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129:1789–1819, 2021.

[104] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[105] Yinbo Chen, Zhuang Liu, Huijuan Xu, Trevor Darrell, and Xiaolong Wang. Meta-baseline: Exploring simple meta-learning for few-shot learning. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9062–9071, 2021.

[106] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[107] Daniel S. Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D. Cubuk, and Quoc V. Le. SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition. In *Interspeech 2019*, pages 2613–2617, September 2019. arXiv:1904.08779 [cs, eess, stat].

[108] Oleg Rybakov, Natasha Kononenko, Niranjan Subrahmanya, Mirko Visontai, and Stella Laurenzo. Streaming keyword spotting on mobile devices. In *Interspeech 2020*, pages 2277–2281, October 2020. arXiv:2005.06720 [cs, eess].

[109] Arindam Banerjee, Srujana Merugu, Inderjit S. Dhillon, and Joydeep Ghosh. Clustering with bregman divergences. *J. Mach. Learn. Res.*, 6:1705–1749, dec 2005.

[110] Thomas Mensink, Jakob Verbeek, Florent Perronnin, and Gabriela Csurka. Distance-based image classification: Generalizing to new classes at near-zero cost. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(11):2624–2637, 2013.

[111] Mark Arnold, Thomas Bailey, J Cowles, and Jerry Cupal. Implementing back propagation neural nets with logarithmic arithmetic. In *Proceedings International. AMSE Conference on Neural Networks*, volume 1, pages 75–86, 1991.

[112] M.G. Arnold, T.A. Bailey, J.J. Cupal, and M.D. Winkel. On the cost effectiveness of logarithmic arithmetic for backpropagation training on simd processors. In *Proceedings of International Conference on Neural Networks (ICNN'97)*, volume 2, pages 933–936 vol.2, 1997.

[113] Arnab Sanyal, Peter A. Beerel, and Keith M. Chugg. Neural Network Training with Approximate Logarithmic Computations. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3122–3126, Barcelona, Spain, May 2020. IEEE.

[114] Dominika Przewlocka-Rus, Syed Shakib Sarwar, H. Ekin Sumbul, Yuecheng Li, and Barbara De Salvo. Power-of-Two Quantization for Low Bitwidth and Hardware Compliant Neural Networks, March 2022. arXiv:2203.05025 [cs].

[115] Alessandro Pappalardo. Xilinx/brevitas, 2023.

[116] Zhourui Song, Zhenyu Liu, and Dongsheng Wang. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[117] Jérémie Teyssier, Suzanne V. Saenko, Dirk Van Der Marel, and Michel C. Milinkovitch. Photonic crystals cause active colour change in chameleons. *Nature Communications*, 6(1):6368, March 2015.

[118] Ximin Li, Xiaodong Wei, and Xiaowei Qin. Small-Footprint Keyword Spotting with Multi-Scale Temporal Convolution, October 2020. arXiv:2010.09960 [cs, eess].

[119] Emad A. Ibrahim, Bart van den Dool, Sayandip De, Manil Dev Gomony, Jos Huisken, and Marc Geilen. Dilate-Invariant Temporal Convolutional Network for Real-Time Edge Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(3):1210–1220, March 2022.

[120] Y. S. Chong, W. L. Goh, V. P. Nambiar, and A. T. Do. A 2.5 W KWS Engine With Pruned LSTM and Embedded MFCC for IoT Applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(3):1662–1666, March 2022. Conference Name: IEEE Transactions on Circuits and Systems II: Express Briefs.

[121] Weiwei Shan, Minhao Yang, Tao Wang, Yicheng Lu, Hao Cai, Lixuan Zhu, Jiaming Xu, Chengjun Wu, Longxing Shi, and Jun Yang. A 510-nW Wake-Up Keyword-Spotting Chip Using Serial-FFT-Based MFCC and Binarized Depthwise Separable CNN in 28-nm CMOS. *IEEE Journal of Solid-State Circuits*, 56(1):151–164, January 2021. Conference Name: IEEE Journal of Solid-State Circuits.

[122] Yicheng Lu, Weiwei Shan, and Jiaming Xu. A Depthwise Separable Convolution Neural Network for Small-footprint Keyword Spotting Using Approximate MAC Unit and Streaming Convolution Reuse. In *2019 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 309–312, November 2019.

[123] Bo Liu, Hao Cai, Zilong Zhang, Xiaoling Ding, Ziyu Wang, Yu Gong, Weiqiang Liu, Jinjiang Yang, Zhen Wang, and Jun Yang. More is Less: Domain-Specific Speech Recognition Microprocessor Using One-Dimensional Convolutional Recurrent Neural Network. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(4):1571–1582, April 2022. Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers.

[124] Charlotte Frenkel and Giacomo Indiveri. ReckOn: A 28nm Sub-mm2 Task-Agnostic Spiking Recurrent Neural Network Processor Enabling On-Chip Learning over Second-Long Timescales. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pages 1–3, February 2022. ISSN: 2376-8606.

[125] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey, August 2017. arXiv:1703.09039 [cs].

[126] Shell Xu Hu, Da Li, Jan Stuhmer, Minyoung Kim, and Timothy M. Hospedales. Pushing the Limits of Simple Pipelines for Few-Shot Learning: External Data and Fine-Tuning Make a Difference. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9058–9067, New Orleans, LA, USA, June 2022. IEEE.

[127] Iulia Alexandra Lungu, Alessandro Aimar, Yuhuang Hu, Tobi Delbruck, and Shih-Chii Liu. Siamese Networks for Few-Shot Learning on Edge Embedded Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(4):488–497, December 2020. Conference Name: IEEE Journal on Emerging and Selected Topics in Circuits and Systems.

[128] Kenneth Stewart, Garrick Orchard, Sumit Bam Shrestha, and Emre Neftci. On-chip Few-shot Learning with Surrogate Gradient Descent on a Neuromorphic Processor. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 223–227, August 2020.

[129] Dayane Reis, Ann Franchesca Laguna, Michael Niemier, and Xiaobo Sharon Hu. A Fast and Energy Efficient Computing-in-Memory Architecture for Few-Shot Learning Applications. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 127–132, March 2020. ISSN: 1558-1101.

[130] Yang Shen, Xuhao Sun, and Xiu-Shen Wei. Equiangular Basis Vectors, March 2023. arXiv:2303.11637 [cs].

[131] Manuele Rusci and Tinne Tuytelaars. Few-Shot Open-Set Learning for On-Device Customization of KeyWord Spotting Systems, June 2023. arXiv:2306.02161 [cs].

[132] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. TernaryBERT: Distillation-aware Ultra-low Bit BERT, October 2020. arXiv:2009.12812 [cs, eess].

[133] Jure Zbontar, Li Jing, Ishan Misra, Yann LeCun, and Stephane Deny. Barlow twins: Self-supervised learning via redundancy reduction. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 12310–12320. PMLR, 18–24 Jul 2021.

[134] Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Utku Evci, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. Meta-Dataset: A Dataset of Datasets for Learning to Learn from Few Examples, April 2020. arXiv:1903.03096 [cs, stat].

A Metric learning formalization

Considering the formalization in Section 2.1.2, θ in the inner loop (see Eq. (6)) is then the set of embeddings generated from the support examples (D^{support}). Using the formalization notation L^{task} is:

$$\theta^{*(i)}(\omega) = \arg\min_{\theta} L^{task}\left(D_{\text{source}}^{\text{support}(i)};\theta,\omega\right) = f_{\omega}\left(X_{\text{source}}^{\text{support}(i)}\right) \tag{54}$$

$$\text{s.t.} \quad L^{task}\left(D_{\text{source}}^{\text{support}(i)};\theta,\omega\right) = \left(f_{\omega}\left(X_{\text{source}}^{\text{support}(i)}\right) - \theta\right)^2 \tag{55}$$

Where f_{ω} is the neural network with parameters ω that generates embeddings.

B *mini*ImageNet

The *mini*ImageNet dataset was created by Vinyals *et al.* [29] as a more lightweight alternative to the original ImageNet dataset [88]. The *mini*ImageNet subset is curated for few-shot learning, containing 100 classes with 600 samples per class of size 84 × 84 pixels (see Fig. 79 for a set of sample images from the *mini*ImageNet dataset). In the original version, 80 classes were used for training and 20 for testing [29]. However, since the exact splits were not made public [99], Ravi *et al.* [99] proposed a 64-16-20 training-validation-testing split with 100 randomly selected 600-sample classes, which has been widely adopted by the few-shot learning community [98] [30] [19].

Another dataset that uses a subset of the original ImageNet dataset for few-shot learning is "Meta-Dataset" [134]. The Meta-Dataset is effectively a dataset of 10 datasets, where generalization performance can be measured across all datasets, after training only on the ImageNet subset or after training on all datasets.

C MFCC with and without discrete cosine transform comparison

This appendix contains Table 23

[35] `https://www.kaggle.com/datasets/arjunashok33/miniimagenet`

Figure 79: Sample images from the *mini*ImageNet [29] dataset. Image taken from [35].

Table 23: Performance comparison between full MFCC and MFCC without discrete cosine transform (DTC) with and without quantization (via QAT) for three different TCN architectures. W / A / B indicates the weight, activation and bias quantization respectively. * indicates that the run was stopped early due to low validation accuracy, which is the accuracy displayed for those runs.

Architecture	# of parameters	With DCT?	FP32 / FP32 / FP32	L4 / U4 / S16	S8 / U8 / S16 S16
7 x [16,16,32]	21,484	Yes	94.50	87.46	86.76
3 x [16,16,32,32,32]	22,764	Yes	94.56	71.23*	-
3 x [32,16,16,32,32]	22,764	Yes	94.38	68.90*	-
7 x [16,16,32]	21,484	No	94.76	93.79	91.63
3 x [16,16,32,32,32]	22,764	No	95.69	93.13	92.92
3 x [32,16,32,32,32]	22,764	No	95.60	92.94	90.00