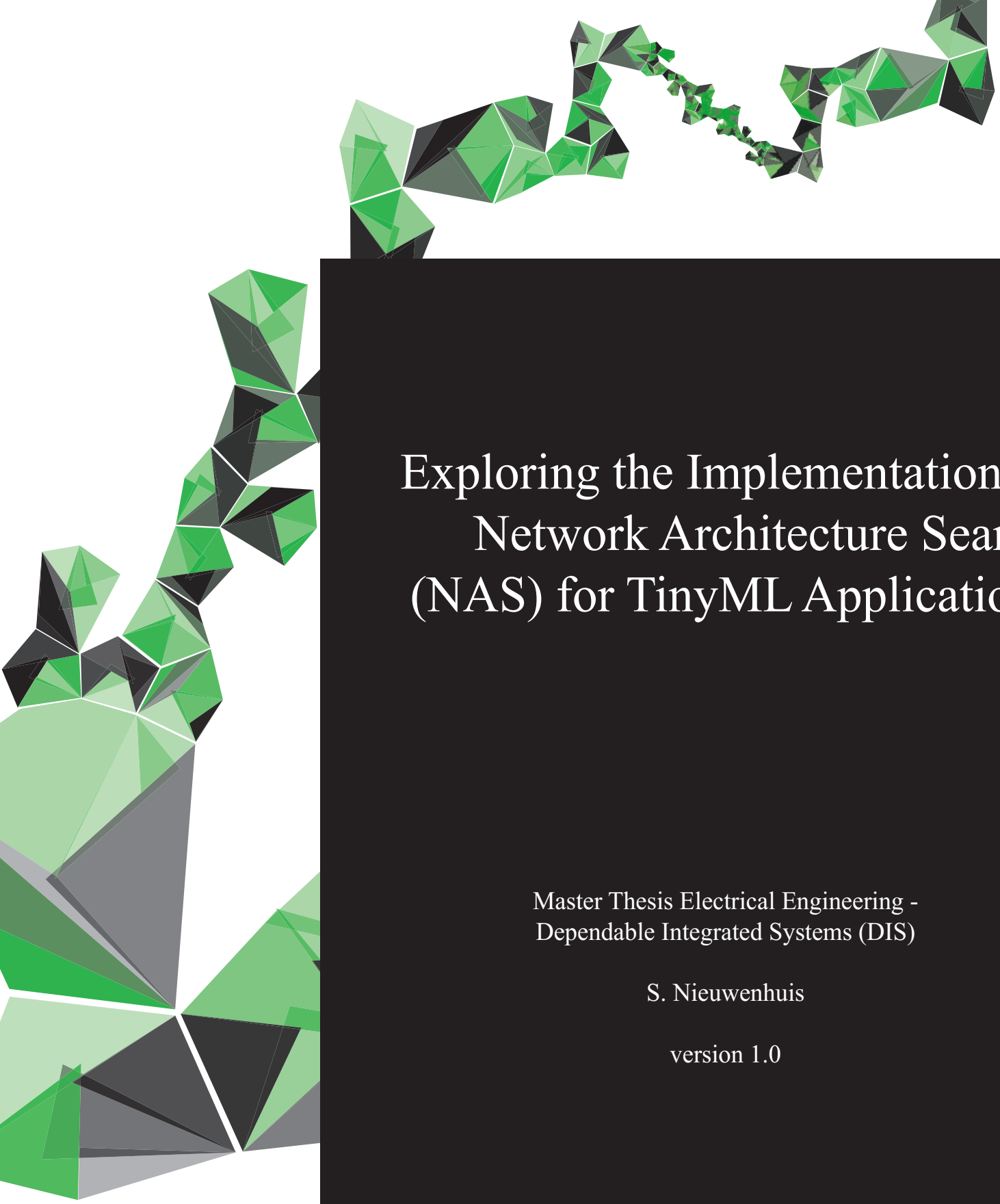


Exploring the Implementation of Network Architecture Search (NAS) for TinyML Applications

Master Thesis Electrical Engineering -
Dependable Integrated Systems (DIS)

S. Nieuwenhuis

version 1.0

Supervisors:
Dr.Ir. S.H. Gerez
Dr.Ir. N. Alachiotis
Dr. C.G. Zeinstra

Enschede, March 2022

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Computer Architecture for Embedded Systems (CAES)
Datamanagement & Biometrics (DMB)

Abstract

The use of *machine learning* (ML) is ever increasing and finding its way in more and more applications, including embedded devices with *microcontrollers* (MCU). The popularity of inference on MCUs results from the low purchase cost and power requirements of microcontrollers. The deployment of neural networks is challenging, since microcontrollers are severely limited by the available memory and storage. This is a problem, especially for the larger and popular *convolutional neural networks* (CNN). In order to make them fit on a microcontroller, neural networks can be reduced in size by means of quantizing the parameters, reducing precision in the process. Additionally, quantization affects the performance of network inference. These aspects of NN quantization are worth studying to gain knowledge about the trade-off between speed and accuracy.

Simultaneously, automated neural network design by means of *neural architecture search* (NAS) has proven to be able to generate high-performance neural networks that are more efficient than man-made networks. When given proper boundary conditions, NAS-generated networks can be made to fit within the constraints of a microcontroller.

In this work, a NAS algorithm targeted specifically to microcontrollers is investigated with quantization in mind. The novelty of this work is the inclusion of quantization levels as a constraint of a search algorithm to construct more complex networks that fit inside the memory requirements of microcontrollers. TinyML implementations of classifiers for the MNIST, FashionMNIST and CIFAR10 datasets have been generated with this search algorithm. Although the performance is not state-of-the-art, several networks, both human- and NAS-designed, are outperformed on the MNIST dataset, where this work achieves 98.91% accuracy at a footprint of 38.7 kB and a peak memory usage of 8.7 kB. On the FashionMNIST dataset, an accuracy of 90.42% is achieved, CIFAR-10 reaches 60.66% accuracy. When combined with a hardware support for datatypes smaller than 8-bit, inference time could halved compared to mainstream microcontrollers today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although the concepts of *artificial intelligence* (AI) and *machine learning* (ML) have been around for a few decades, the advancements made in computing power and availability of data in recent years have been the reason that deep *neural networks* (NN) are adopted for an increasing number of applications. AI is deployed to more and more industrial applications for automation and quality control for fabrication processes, but also to consumer products, such as smartphones with voice assistants for hands-free use or face verification for security. Platforms like these, that process data at the same location as where it is captured, are referred to as *edge platforms*. Edge devices come in many shapes and sizes, with different computational abilities and power requirements. Continuing with the examples above, smartphones and other handheld devices are limited in terms of power by the capacity of a battery and the resulting power-efficient processor. Devices for industrial automation can be powered from the grid and can therefore have more powerful computing hardware. Due to the large differences in available computation power, not all edge platforms have until now been able to execute AI applications. In order to enhance these less powerful devices with AI-driven functionality, it is possible to send captured data to distant *cloud servers* that process the data and send back the result. The differences between edge- and cloud computing is illustrated in Figure 1.1 and a comparison of strong and weak points is provided with Table 1.1.

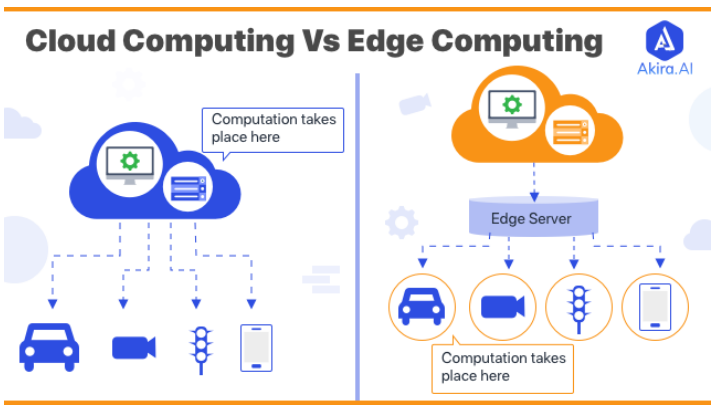


Figure 1.1: Difference between cloud- and edge computing [1]

Although cloud-based ML inference is a fitting solution when compute power is desired, there

Table 1.1: Comparison of cloud and edge platforms for neural network inference

Cloud	**Edge**
Lots of computational power and memory	Limited computational power and memory
Network connection required	No network connection required
High latency, high throughput	Low latency, low throughput

are a few drawbacks. The need for an internet connection whenever the resources are required is a prominent one, especially because the addition of wireless circuit to a product costs both space and power. Another drawback of cloud-based inference is that the data to be computed, such as voice commands or pictures for facial identification have to be uploaded to the cloud server. This is a problem for sensitive data, especially in a time when online privacy and data protection is a topic of relevance.

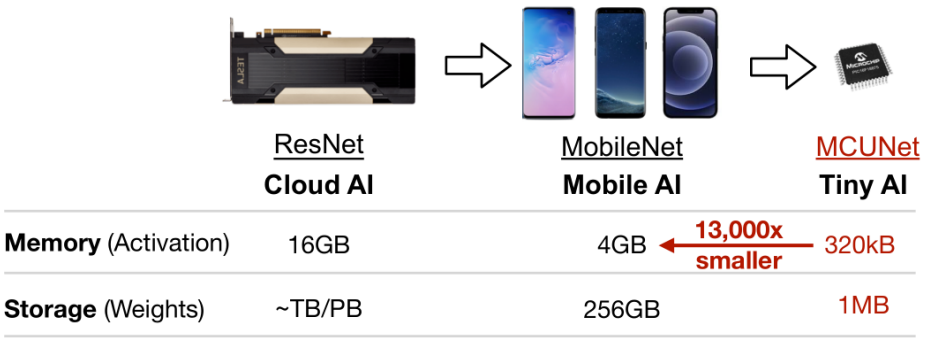


Figure 1.2: Memory requirements for different AI inference platforms and networks [2]

With enhancements on both the computational power and efficiency of processors, and advancements in ML algorithms, smartphones and other power-efficient devices are now able to locally deploy AI applications. Naturally, as research progresses and new challenges are sought after, developments of new ML platforms come to the surface, like *microcontrollers* (MCU). Microcontrollers are small, cheap, and simple processors with a low power draw. These properties make microcontrollers a popular choice for many embedded applications and *internet of things* (IoT) solutions. Research has indicated that it is possible to deploy relevant ML applications on microcontroller platforms, partly because neural networks are becoming more and more efficient. This field of research has been named *tinyML* and has gained a community that is concerned with the development of algorithms, software and hardware solutions related to ML on low power devices. [13, 14, 15]. Figure 1.3 is an example of a recent tinyML application that has been presented by STMicroelectronics in the shape of a credit card with embedded finger print sensor, which can be powered wirelessly [3].

Implementing tinyML applications poses a new challenge, because the available resources on microcontrollers are significantly reduced compared to other platforms. This limitation is illustrated in Figure 1.2 by comparing the memory and storage of the three types of platforms for ML. Whilst the amount of memory and storage between cloud and mobile platforms is a factor 10 or lower, the difference between mobile and tiny platforms is a factor 10^4.

In general, the task of designing and optimizing a neural network already is a time-consuming

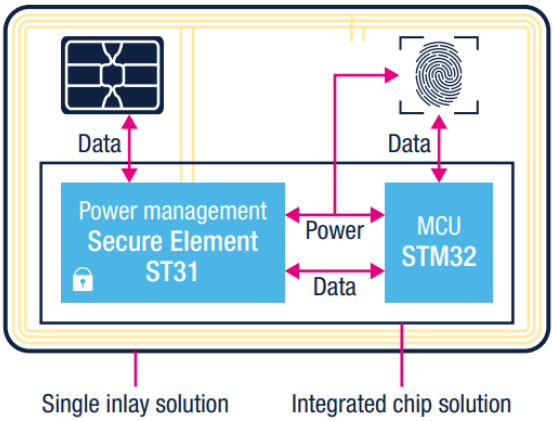


Figure 1.3: Schematic overview of the ST biometric system-on-card [3]

process that heavily relies on the experience of network architects. Add to this the restrictions in available resources for microcontroller devices, and tinyML becomes a major investment to develop a high-performance neural network targeted towards a specific application. As an effort-saving measure or an alternative for lesser experienced developers it is possible to take an existing network and retrain it for the intended use.

Driven by the desire to have networks ready to deploy for non-ML-experts, the topic of automated neural network design, or *AutoML* has gained a lot of momentum in the past years [16]. The research within AutoML is tasked with finding and optimizing design automation strategies, for instance hyperparameter optimization and *neural architecture search* (NAS). NAS automates the design process of a neural network and is therefore interesting for the tightly-constrained tinyML platforms. Given the proper constraints, NAS can result in relatively complex networks for applications like computer vision and natural language processing. By means of adding multiple objectives to NAS, a neural network can be found within a certain memory bound, the inference of which should be able to be performed on microcontrollers.

1.1 Research questions

The objective of this work comes from the difficulties of NN inference on tinyML platforms. With the main difficulty being the restricted availability of memory and storage the motivation behind this work, the objective of this work is to solve this problem by combining two tools: NAS and low-precision quantization.

The problem that has to be researched relates to the possibilities of the combination of compressing a neural network and automated network design for microcontrollers. This goal has been formulated as the main research question:

"To what extent does weight quantization as a NAS constraint have an impact on the accuracy/performance trade-off when performing the inference of a TinyML network on a microcontroller?"

To find an answer to this question, several areas of microcontrollers, NAS, and quantization

will have to be explored and tested, which raises the following sub research questions:

- *"With what precision can the run-time memory and storage requirements for neural network inference be calculated during the execution of NAS?"*

  When adding constraints such as memory and storage to NAS, these parameters will have to be calculated for each architecture. The precision of these calculations are of importance in preventing problems with overflow during deployment on the microcontroller.

- *"What are the consequences of including the quantization of neural network parameters as part of a NAS algorithm to the complexity of the search space?"*

  The search space defines the types of architectures that can be created by the NAS algorithm. Adding more elements to the search space, for example quantization, will expand the number of network architectures that can be generated. Even though a larger search space can lead to the generation of better performing network architectures, a larger search space will take more time to explore. This question will answer on which side of this trade-off the NAS algorithm with quantization will reside.

- *"How are the accuracy and resource utilization of an image classification convolutional neural network that has been implemented in a microcontroller using NAS?"*

  Verification of the NAS algorithm is essential for the answer to the research question. The discovered architectures must adhere to the constraints implemented for memory and storage, while the classification accuracy will be determining the quality of the NAS algorithm.

- *"How does a sub-byte quantized neural network compare to more conventionally quantized network inference in terms of operations and inference time?"*

  The final part of the research is to determine the effect of sub-byte datatypes in NAS. This last question should contribute an insight to the accuracy/performance trade-off by providing an analysis of the possible improvements.

1.2 Contributions

The research questions cannot be answered with the results from a literature research alone. In order to answer some parts of the questions, a NAS architecture will be implemented with sub-byte quantization. A program that estimates the resource utilisation of a neural network during inference is also required, since network architectures are tested on memory and storage footprint to add multiple objectives to the NAS algorithm. With the answer to the research question, this works contributes the following aspects:

- A multi-objective NAS algorithm with low precision quantization layers in the search space to target the constrained resources available on tinyML platforms.

- A software package for NN inference analysis that aims to aid researchers and NAS algorithms with testing the feasibility and performance of MCU inference.

1.3 Structure of this report

In the remainder of this report, the contents are structured as follows. Chapter 2 gives the reader the necessary background information to understand the differences and similarities between conventional NN design and NAS network design. Other works relevant to the topics presented in this work are discussed in Chapter 3. The findings and experiences gained from that chapter are subsequently applied in Chapter 4, where the design decisions and design process are presented. Additionally, the experiments that will be used to test several aspects of the NAS design are introduced. The results of these experiments are presented in Chapter 5, followed by an analysis of these results. Finally, Chapter 6 is reserved for the discussion of the results form the experiments and the literature from Chapters 2 and 3. At the end of Chapter 6, a number of recommendations for future work on this topic is presented along with potential improvements for this work.

Chapter 2

Background

In this chapter, background information is provided for the topics introduced in this work. This information should give the reader the necessary knowledge to understand the concepts and analysis.

2.1 Neural networks

Neural networks are computational filters that have been inspired by the inner workings of a biological brain. A biological brain is built up with neurons: cells that carry information. Neurons in the brain are connected to each other, with connections named axons. This section points out the similarities and differences between the structure of a biological brain and a fully connected neural network, after which convolutional neural networks are discussed.

2.1.1 Fully connected networks

The artificial neurons, commonly also referred to as nodes, in a neural network share the same topology as a biological neuron. An illustration of a neural network node is given in Figure 2.1. Every *node* in a network has multiple inputs and an output, which are also referred to as *activations*. This nomenclature comes from the biological brain cells, where different cells are activated by inputs from other cells. The input axon of a brain cell is resembled by the input of the node. The strengths of activations in a biological brain can differ between nodes. To emulate this, neural network nodes contain a variable for each input: the *weights*. Another variable, the *bias*, can shift the output activation of the node up or down, depending on what is a better fit for the data. The output o of a cell is a weighted sum of all inputs x with weight w. This output is offset by a bias b and followed by an activation function a with nonlinear behaviour [17]. Computing the output of a neural network node is a set of multiply accumulate calculations (MACC), see Equation 2.1. The more nodes, the higher the complexity of a networks and thus the higher the number of MACCs.

$$o = a\left(\sum_{i=1}^{I} w_i x_i + b\right) \tag{2.1}$$

Nodes of a neural network can be combined as a vector, which forms a layer of nodes. Several layers can be connected together, with the output and all I inputs of a node in a layer are connected to all inputs and outputs of adjacent layers, as illustrated in Figure 2.2. Such a network design is referred to as a feed forward fully connected network, since all nodes are interconnected to each other and there are no loops in the network. The inputs of feed forward neural networks do not depend on their own outputs.

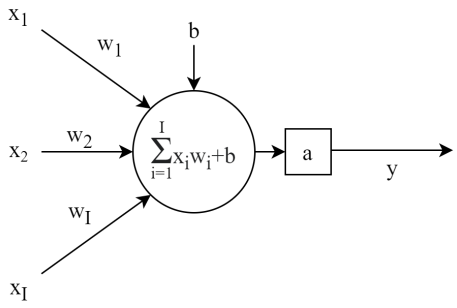


Figure 2.1: A neural network node with I inputs and a bias

Neural networks are large filters with non-linear behaviour, built up by different layers of interconnected nodes or neurons. For an outside observer only the input and output layer are accessible. The other layers, referred to as hidden layers, reside inside of a black-box model. The coefficients of these networks are configured by means of machine learning.

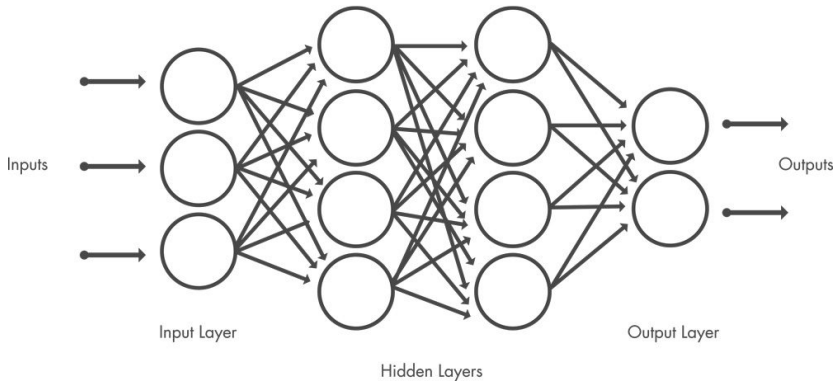


Figure 2.2: Illustration of a feed forward fully connected neural network with input, output and hidden layers [4]

2.1.2 Convolutional layers

Another type of neural network layer is the convolution layer, which is found in convolutional neural networks (CNN) such as the example in Figure 2.3. Convolutional filters, or kernels, are widely used within image processing techniques and are implemented mostly for edge and feature detection. Filtering with kernels is an effective method for feature detection, as similar filters are used in other edge detectors.

In neural networks, the coefficients of a kernel are determined during training. By tuning the kernel filters, a convolutional layer is able to recognize a certain feature. Multiple kernels

in a layer can be used to capture different features in that layer and by stacking multiple convolutional layers, relations between the detected features can be captured. The outputs of the convolutional layers contain a lot of information about the shape of the object given as input. To do something with these shapes, convolutional neural networks often have one or more fully connected layers to link these features to the different object classes at the output.

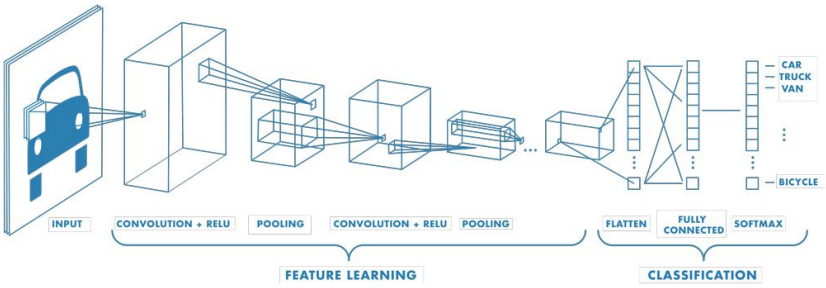


Figure 2.3: A convolutional classification network with two convolutional layers for feature learning and three fully connected layers for classification [4]

A convolutional layer is computed by performing a discrete 3D convolution between K filter kernels F_k and the input tensor I. The *stride* is the number of positions that the kernel will move for every convolution equation. For instance, a stride of two implies that the kernel is placed on every other position of the input. Assuming a stride of 1, this convolution can be computed for every element in the output using Equation 2.2, for which Figure 2.4 provides a reference for the notations of the different dimensions [18]. This results in an output tensor O for every filter, thus in the end the output is K channels deep. Similar to fully connected layers, convolutional layers are composed of a lot of MACCs, since a discrete convolution operation also boils down to a weighted sum of the inputs. The weights are in this case the kernel coefficients.

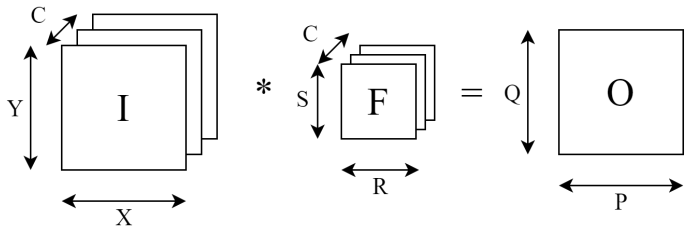


Figure 2.4: A single kernel convolution with notations for input, filter, and output.

$$O(p,q,k) = \sum_{c=0}^{C} \sum_{s=0}^{S} \sum_{r=0}^{R} I(p+r-1, q+s-1, c) \cdot F_k(r,s,c) \tag{2.2}$$

2.1.3 Training and inference

The process of tuning the parameters in a neural network to fit the black-box model to a certain dataset is called training. Neural networks can be trained with different methods, although supervised learning is used most of the time. When the outputs of the network are known during training, supervised learning can achieve accurate results, but if the outputs are

not known, one is forced to move to the less accurate unsupervised learning. With supervised learning, the desired outputs for the training dataset are provided. This dataset is comprised of examples of the inputs that a network is expected to process during inference. An image classification should for instance correctly identify the object in an input image. When all training data of a dataset is passed through the network, an *epoch* is completed. Usually, multiple passes of the training data and thus several epochs are required to train a neural network.

When a network is feed forward, thus no loops exist in the network and inputs do not depend their own outputs, the backpropagation method can be implemented. The first training step is for the neural network to compute the output for the provided input image. With backpropagation, the difference between the desired and actual outputs can be calculated and be used to calculate the *loss* of the network.

Training is a highly computationally expensive task because a dataset usually contains a large number of different input samples that are passed through the network for several epochs. The CIFAR10 dataset does contain 60000 images for training, larger sets such as Imagenet 1.2 million [19, 20]. This implies that during training, the network is evaluated thousands of times. However, since the nodes in a layer are only connected to nodes in other layers and not to each other, the output activations of a single layer can be computed in parallel. Parallel computation can speed up the training process significantly, which is why networks are often trained on GPU compute arrays. The fact that the training process is computationally intensive is not seen as a problem, because it is only supposed to be performed once.

The inference of the deployed application is less computationally demanding than training a network, as only one set of input data is provided for every desired output and the weights do not have to be tuned. However, the power draw for inference is more relevant. In many cases the inference is performed continuously for a long duration and devices on the extreme edge do not have the abundance of resources that the cloud environment has. This has driven researchers to find more optimized platforms to fit different requirements for inference.

2.2 Network Architecture Search

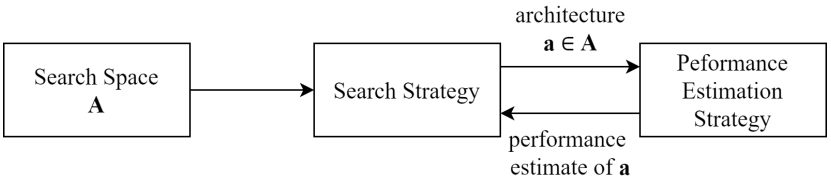


Figure 2.5: The three basic building blocks of NAS algorithms [5]

Neural architecture search (NAS) is one of the automated neural network design paths of AutoML. With NAS, neural networks can be designed with minimal inputs from the network designer. Generally speaking, NAS algorithms are made up of three dimensions which are visualised in Figure 2.5: search space, search strategy and performance estimation strategy [21]. A short description of these dimensions is provided below, before Sections 2.2.1, 2.2.2, and 2.2.3 discuss different implementation approaches.

The *search space* is defined as the set of elements that can be used to construct network architectures during the NAS process. The elements in the search space are selected with the purpose of the desired network in mind. An image classification task, for example, can be expected to have several types of convolutional layers in the search space. With a suiting subset of all possible neural network elements in a search space, the search is performed more efficiently as the scope of possibly created networks is smaller than it would be with a brute force approach. A drawback of this method is that the search space is influenced by the knowledge of the network architect. It it therefore less likely that novel architectures are discovered than it would be with brute force.

The second NAS dimension, *search strategy*, is the method or algorithm that is used to construct different network architectures from the elements available in the search space. The difficulty in finding a good search strategy lies with the inefficiency of evaluating every possibility in the search space.

Finally, *performance estimation strategies* are implemented to verify the performance of each of the generated network architectures. The performance of an individual network can be determined by verifying a network against unseen representative data, which implies that the network needs to be trained. Training every generated network architecture is impractical, given that the population can hold 1000 networks [22]. As a result of all those generated networks, the duration of architecture searches is often measured in days or weeks on a GPU[21, 6], even with optimized performance estimation strategies. It is with this motivation that different strategies have been developed, each with the intention to give a accurate estimation of the performance without fully training the networks.

2.2.1 Search Space

The definition of a NAS algorithm starts with the definition of the types of networks can be constructed. What type of network results, starts at the *search space*, a collection elements suited to generate neural networks. One should think of elements as different types of layers, such as convolutional layers, fully connected layers or pooling layers. These layers can be connected together in different configurations, but also contain parameters that define the composition of a layer, such as the number of neurons or the kernel size, but also the type of activation function that is coupled to a layer. So, besides the number and type of layers, multiple parameters can be chosen for each layer. In an attempt to limit the number of possible combinations, resulting in a faster execution of the search, certain structures can be implemented in the search space. This will limit the number of possible architectures with the benefit of a smarter selection process and a better search efficiency.

A search space is usually set up containing elements that are known to be suitable for the application of the neural network. Setting up the search space with such method does limit the types of architectures that can be created, reducing the complexity of the search algorithm. This reduction in complexity comes with the risk that novel, optimally performing network architectures cannot be created, because the search space is generally set up with common NN knowledge as basis.

The most straightforward search space consists of elements that can be connected sequentially together to form a neural network architecture. Different architectures can be generated by varying the types of layers or the configuration of the layers between models. Such a search

space is called a *chain-structured search space*. In such a search space, there is only one type of connection between different layers, which reduces the possibilities of connections. The number of and types of layers are not necessarily limited, although this is possible to further reduce the size of the space. Figure 2.6 pictures an element of a chain-structured search space.

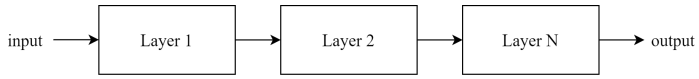


Figure 2.6: A sequential combination of network layers: an element from a chain-structured search space

A more complex neural network can be achieved by defining small neural networks that can be combined and reused. Such a search space is called a *cell-based search space*, with these small networks embedded in cells, such as the one in Figure 2.7. An important difference with the chain-structure is that no layers can be added, only the types of layers can be changed. Layers inside a cell have to be predetermined by the designer and can be any layer that suits the target application. The size of the search space is in this case influenced by the possible types of layers in a cell. The manner in which these cells are connected, however, is commonly predetermined for a specific network. An example of what a network generated form a cell-based search space is ResNet, which is a network that exists of groups of layers with similar architectures, connected to each other.

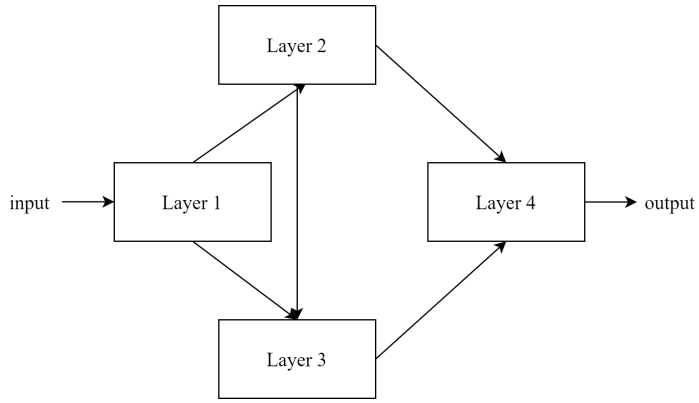


Figure 2.7: A complex combination of network layers: an element out of a block-structured search space

Finally, there is a search space that is one large neural network containing multiple branches between its layers, as illustrated in Figure 2.8. With this specific search space, the high-level architecture of each generated network is the same. The networks differ in the contents of the layers and the contents are selected with the different branches in the *supernetwork*. For instance, a certain position in the network contains a convolutional layer. In the generated architectures, this layer can differ in kernel size or number of filters, but it is always a convolutional layer. This supernetwork contains all the layer configurations from the search space, including connections, and can therefore be used with a one shot performance estimation. This is elaborated upon in Section 2.2.3.

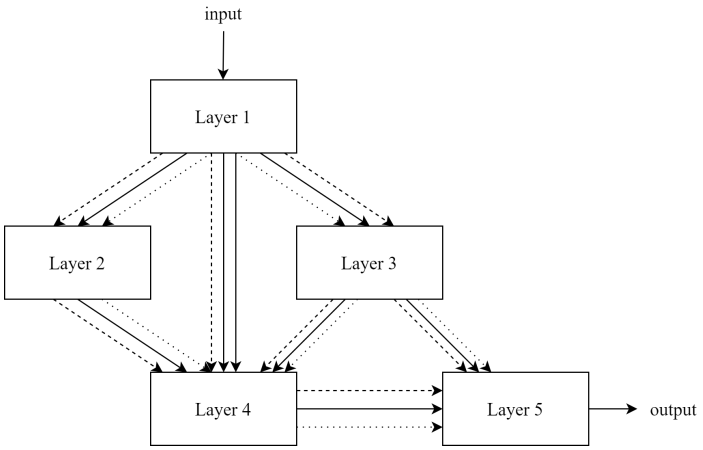


Figure 2.8: A multi-branch supernetwork form which network architectures can be formed. The different connections from one block to another illustrate different options for the layer contents.

2.2.2 Search strategy

The outright optimal network is unknown and difficult to find when not implementing an unbounded search space and brute force search method, looking at every possible network architecture. This is infeasible due to the time required for such a search. The network considered optimal with NAS is therefore located within the search space. The goal of the search strategy is to facilitate a strategic path through the possibilities available in the search space to find this best possible network in the search space.

Grid and random search algorithms are some obvious choices to start with any search problem. The properties of these algorithms, however, make them a good example of the nuance that needs to be found for the strategy in case of NAS. With a grid search algorithm, a grid is mapped on top of the contents of the search space. The left search space in Figure 2.9 illustrates this principle with two parameters; one important and one unimportant. This grid does not cover every solution, but takes probing points in the search space. The difficulty with grid search lies in its efficiency, since every important parameter is tested with multiple unimportant parameters. As a result, the same important parameter is tested multiple times. For instance with the number of points in Figure 2.9, the nine experiments result in three tested important parameters.

The number of inspected important parameters can be refined by increasing the number of parameters in the grid search, though at the cost of more evaluation points. Another option is to randomly explore the search space, which is illustrated in Figure 2.9 on the right-hand side.

In order to reduce the number of duplicate evaluations of a point in the search space, a random search algorithm can be implemented. While grid search evaluates different combinations of, in the case of Figure 2.9, important and unimportant parameters, the same parameter is used for a comparison in multiple instances. With a similar number of evaluation points as grid search, a random search can cover a higher number of different parameter combinations because no parameter is used more than once (for true random instances). This principle

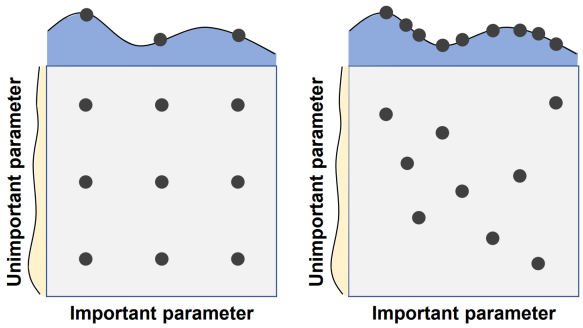


Figure 2.9: Illustration of grid (left) and random search (right) in a search space of two parameters [6]

is illustrated in Figure 2.9 with the number of points in the blue scale for the important parameter. From the same illustration, a trend is visible where low scores for the important parameter have the same chance of being selected as high scores. This is not contributing to the efficiency of the evaluation of the search space, since low scoring models are of no interest for the result of NAS. Not evaluating low scoring network architectures would free up computational resources for evaluating promising networks.

To further optimize the search algorithm, search strategies that can identify promising areas in the search space and bias the search towards those areas have been designed. Examples of such strategies are evolutionary search and reinforcement learning (RL). Evolutionary algorithms have been inspired by evolution in nature. Over several generations an architecture changes by means of mutations and the impact on performance is measured. The selection process is according to survival of the fittest. Reinforcement learning deploys an AI algorithm to create and evaluate network architectures. The performance of the created networks is determined by a score. Based on this score, the AI learns what type of architectures work well and which do not. More on the algorithm of these search strategies is explained in Sections 2.2.2 and 2.2.2 for RL and evolution respectively.

Even though evolution and RL are different algorithms, both can identify which search space combinations are likely to produce well performing networks and which are not. For both implementations it is important that the search starts at a wide variety of points in the search space. If the search starts out too narrow, the outcome could be a network at a local maximum in the space. In order to find the highest local maximum possible, as many of these areas need to be evaluated as possible.

Reinforcement Learning

Reinforcement Learning (RL) is a form of decision making neural networks where a controller, or agent, will evaluate different solutions from a design space. This controller selects a possible solution to the problem to be solved and, in return, receives a score based on the success of the solution. When this process is iterated, the agent will learn which of the solutions have the highest contribution to solving the problem as the score in that case will increase. With less successful decisions, a bad score is returned and will therefore less likely to be used.

When applied for NAS, the problem is defined as the contents of a network architecture that

can achieve for instance the highest possible accuracy on a given dataset. The agent will then assemble a network from the search space for each iteration of the search, before it is given a score based on an estimation or evaluation, see Section 2.2.3. Depending on this score, the agent will assemble similar networks or look in another combination in the search space for the next iteration.

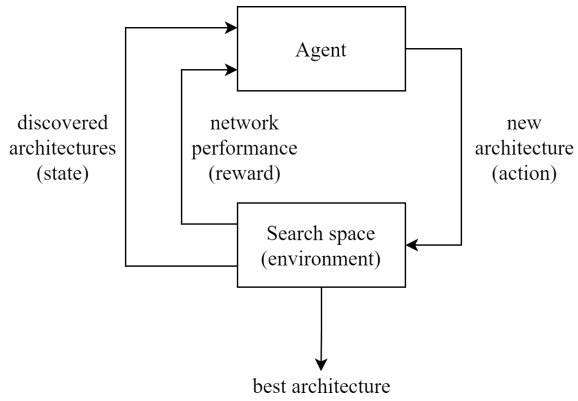


Figure 2.10: Block diagram of a NAS algorithm based on reinforcement learning

Evolutionary Algorithms

The idea behind evolutionary algorithms is rather straightforward. Similar to researchers that evaluate and improve different versions of their hand-crafted networks, evolutionary search is an iterative process that brings modifications to networks in an attempt to find the best performing architecture. Figure 2.11 pictures how the algorithm starts with a population, to which several architectures are added in the first step. These architectures are created from the search space, often at random. Every iteration of the evolution algorithm, some of the networks from the population are promoted to be parent networks. Each parent network receives random mutations to form new networks, so-called child networks. These mutations can be anything from adding or removing layers to or from the network to a change in the size of, for example, the kernel of a specific convolutional layer. As a final step, the child networks are evaluated, after which they are added to the population. To limit the size of the ever expanding population, networks can be removed from the population.

Evolutionary algorithms select the networks that are removed from the population by means of tournament selection. This implies that the network architectures with the lowest accuracy are removed from the population, while the best performing networks are elected to be parent networks and spawn new architectures for the population. A method like tournament selection imposes a certain greediness in the search, as only the best performing networks are explored and mutated. Greediness in the search strategy can land on a local maximum in the search space, where there are no better networks to be created form a certain architecture, but a different architecture performs better.

2.2.3 Performance Estimation Strategy

To determine the best network from the search space, each generated network has to be evaluated. The most straightforward method would be to fully train every network and take the

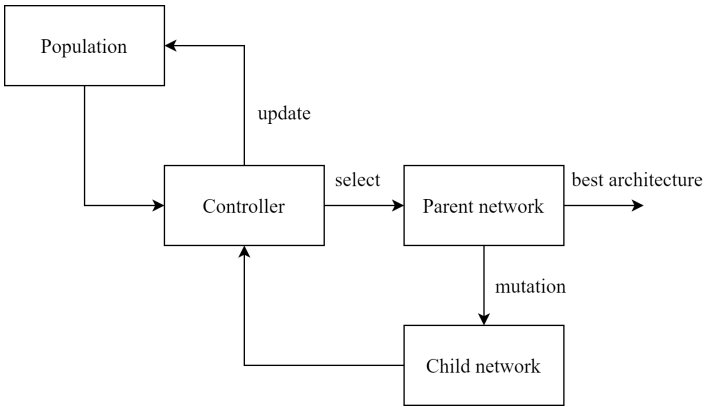


Figure 2.11: Block diagram of a evolutionary NAS algorithm

one with the highest accuracy or any other metric that needs to be evaluated. Training, as discussed in Section 2.1.3, is a computationally expensive task. When a search covers 100 architectures that each need 10 epochs to fully train, a total of 1000 epochs will be computed. The NAS algorithm could complete faster when this evaluation of the networks is optimized, rather than just training networks. To address this problem, performance estimation strategies have been developed. Three of such strategies will be covered in this section.

One of the ways to reduce training time is limit the time necessary to train the networks generated by NAS. This can be achieved by training for instance with fewer epochs, lower input dimensions or by reducing to a subset of the training dataset [5]. All of these optimizations are reducing the data used for training the network and are therefore referred to as coarse estimates. Coarse estimates with fewer epochs can be extended by predicting the learning curve of a network. The coarse estimate and extrapolation methods are best implemented with as little data as possible, leading to the largest reduction in time required for the execution of NAS The rougher the estimate, the less data that is available to base the performance on. It is therefore that other methods are used.

Another method to reduce the training time of NAS is to use already optimized weights across different architectures: weight inheritance. This inheritance gives new architectures a jump start during training, since unmodified parts of the network have already been trained instead of being randomly initialized. All parameters of the network are still trained, though the inherited weights are only slightly optimized compared to training from scratch.

Weight inheritance can also be extended to the entire search space. This third performance estimation strategy is referred to as weight sharing. The search is then started with training one super network, containing the possible structures in the search space. Once this network has been trained, parts of this network can be combined into net architectures, which then do not have to be trained [5]. This gives a better estimate for the eventual performance of an architecture than the low fidelity estimates and requires less retraining of parameters than inheritance. The drawback is that the shared weights are not optimized for the rest of the model.

2.3 Quantization

The different kernels of each convolutional layer, as well as weights and activations for fully connected layers concern variables that need to be stored somewhere in a memory of the deployment platform. There can be millions of these variables in a network By default, machine learning platforms such as TensorFlow and PyTorch use 32-bit floating point parameters to store these variables. This achieves high-precision values that can be tuned precisely during training, but take 4 bytes to store on a chip. Besides the footprint, hardware with a dedicated FPU is required to efficiently work with these datatypes. Not all microcontrollers contain an FPU to save on area and therefore cost and power consumption. Larger networks tend to suffer less from quantization losses, because these networks often are over-parameterized [23, 24]. A single quantization error therefore has a smaller effect than a similar error would in a smaller, lesser parameterized network.

A real-valued parameter r is quantized to q by means of defining the number of levels 2^n, a scaling factor S, and zero-point Z. With these parameters, a value can be quantized as described in Equation 2.3. The scaling factor reduces the range of values that can be represented in r to something that is possible for the number of levels that can be represented by the n-bit integer. In order to catch the largest possible range of approximations, one can shift the range that q represents, by assigning the zero-point Z. The method described in Equation 2.3 is an affine conversion. An affine quantization mapping preserves the capability for the quantized parameters to be implemented with integer arithmetic instead of, for instance, lookup tables [23].

$$q = \text{int[n]}\left(\frac{r}{S}\right) + Z \tag{2.3}$$

In Table 2.1, a comparison of multiple methods to quantize a neural network is presented. The first method is dynamic quantization, a scheme that calculates the quantization range of a set of values dynamically during the inference of a model. The neural network is at this point fully trained and deployed on a device. Dynamic quantization results in the lowest loss of accuracy compared to the non-quantized model, because the range is calculated for each input specifically [25, 26].

Table 2.1: Comparison of different neural network quantization schemes

Type	**Used when?**	**Error gained**
Dynamic quantization	During inference	minimal
Post-training quantization	Between training and inference	low for 8-bit and higher large for sub-byte
Quantization-aware training	During training	low

The range of values that need to be quantized can also be calculated before deployment, this is called static quantization. The advantage of static quantization is the lower computation and memory overhead, since the range is computed before deployment and thus the quantized values can be used. A disadvantage is that the quantized parameters are adapted to a range

of input data rather than a single input as it would for dynamic quantization. This reduces the precision of the network and can therefore result in accuracy losses. Static quantization can be performed after the model has completed training. This is referred to as *post-training quantization* (PTQ). Quantization can also be implemented during training, this is called *quantization-aware training* (QAT). The process of PTQ is displayed in Figure 2.12. A neural network is first fully trained by analyzing the predictions and adjusting the weights. When the training is completed, the network is quantized to result the output network that can be used for inference. Advantages are that any model can be converted and no additional training needs to be performed. The loss of accuracy is also relatively low

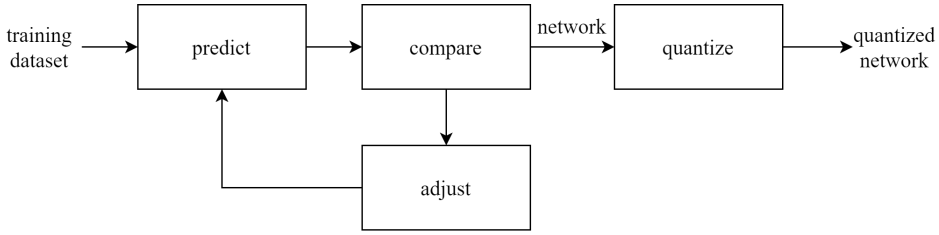


Figure 2.12: Illustration of the post-training quantization process

This loss can be mitigated by quantization-aware training, a static quantization method that computes the quantized values during training and includes them in the backpropagation step of comparing and adjusting predictions. This is illustrated in Figure 2.13. In order to feed the quantized values in the backpropagation algorithm, they are converted back to floating point values during training. While these values are no longer quantized, they do keep their relative quantization errors. The conversion $Float \rightarrow quantInt \rightarrow Float$ does include a rounding error because of the integer conversion in Equation 2.3. This floating point representation of the quantized value is referred to as *simulated* or *fake quantization*.

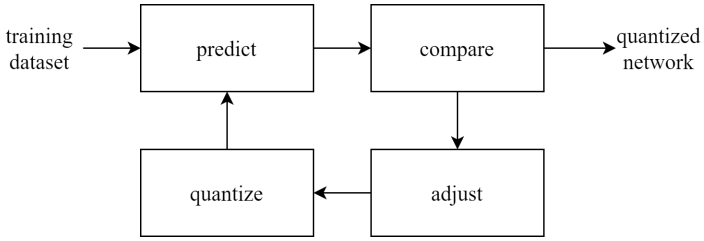


Figure 2.13: Illustration of the quantization-aware training process

2.4 Microcontroller resources

Microcontrollers are simple processors that are deployed in many applications. These processors have been designed to be cheap and efficient, hence the popularity. Compared to mobile and embedded processors, microcontrollers are slower, but more importantly have a lot less resources.

Although several microcontroller architectures exist, the most popular ones are created by ARM. The 32-bit RISC architecture in these microcontrollers is popular for its relative speed. Microcontrollers based on the ARM architectures come in many shapes and sizes, given

that the designs are licensed to other manufacturers, such as STMicroelectronics. These manufacturers can buy a licence to use one of the Cortex-M microcontroller cores and add memories and I/O peripherals. The clock speeds vary between different architectures and range between tens to hundreds of MHz, making microcontrollers are several factors less powerful than embedded processors. Table 2.2 gives an indication of the on-chip resources that microcontrollers pack by comparing different tiers of chips.

Table 2.2: On-chip computational resources of one embedded processor, the BCM2711 [8], and three microcontroller examples, the mid-range RP2040 [9] and STM32F4 [10] and the high-end STM32F7 [11].

Processor	BCM2711	RP2040	STM32F446RE	STM32F746NG
Core	Cortex-A72 (x4)	Cortex-M0+ (2x)	Cortex-M4	Cortex-M7
Frequency	1500 MHz	133 MHz	180 MHz	216 MHz
Memory	2, 4, or 8 GB	264 kB	128 kB	340 kB
Storage	0 kB (SD-card)	0 kB (external)	512 kB	1 MB

Initially, the only ARM 32-bit microcontroller core was the Cortex-M3. Nowadays, there are several skews, each with a specialisation. An overview of these skews and relevant features are presented in Table 2.3. The low-power M0(+) cores support an older version of the architecture, as well as little pipeline stages and extensions to the ISA, all focused on the lowest possible power. The general-purpose M3 and M4 cores are very similar on an architecture and pipeline level, though the M4 core is significantly more powerful for signal processing with the inclusion of an FPU and DSP/SIMD instructions. The Cortex-M7 is the highest power processor core, though all these features come at a cost of power efficiency and purchase cost.

Table 2.3: Comparison of ARM microcontroller cores with parameters relevant for the execution speed of floating point and quantized SIMD neural network inference [12].

Core	Cortex-M0	Cortex-M0+	Cortex-M3	Cortex-M4	Cortex-M7
Architecture	Armv6-M	Armv6-M	Armv7-M	Armv7E-M	Armv7E-M
Pipeline stages	2	3	3	3	6
FPU	no	no	no	yes	yes
DSP/SIMD	no	no	no	yes	yes

A common microcontroller architecture is built up of registers and peripherals, such as an *arithmetic logic unit (ALU)*. Because of the popularity of the ARM architecture for 32-bit microcontrollers, this work will focus on these architectures. The supported datatypes are bytes (8 bits), halfwords (16 bits) and words (32 bits) in memory. Instructions support these as well, though the sub-word datatypes are zero extended in the 32-bit registers. *Single instruction multiple data* (SIMD) instructions are instructions that, as the name implies, can perform actions on more than one piece of data, enabling parallelization for computations and data fetch operations.

2.4.1 Parallelisation

Within microcontrollers, parallelisation can be achieved on an instruction level in the *instruction set architecture (ISA)*. One type of parallelism that is of interest here is *single instruction*

multiple data (SIMD), where one instruction is used to move or compute multiple variables in parallel. The extend of parallelistation is very limited to, for instance, FGPAs as the internal databuses of microcontrollers have a limited width. In those cases, a maximum available parallelisation level would be 4, given that the minimum supported datatype is a byte.

What mainstream microcontrollers lack the support of are the so-called *sub-byte* datatypes, which consists of: nibble (4), crumb (2), or anything in between. Theoretically, using sub-byte datatypes could increase the number of consecutively performed operations, but the hardware is not readily available.

2.4.2 Memory and storage

Given that neural networks are not only complex to calculate but also have many parameters, the most important resources for neural network inference are the memory and storage. Off-chip memory and storage can be added and, although these come in larger capacities, the extra chip takes space on a PCB and the external connection is slower than that of the on-chip resources.

On-chip memories come in two variations: a volatile memory that functions like RAM and the non-volatile memory stores for example the program. The volatile (SRAM) memory will in this work be indicated by the term *memory*, whilst the term *storage* is used for the non-volatile (NOR-Flash) memory. An estimation for the memory and storage usage of a neural network architecture can be made with relative ease and is explained more in-depth in Section 4.2.

2.5 Inference libraries

As generated in PyTorch, neural networks are represented as graphs that show the connections between inputs and outputs. When exported, this graph is stored with the weights from a network. The network cannot be deployed on a platform in this shape, as the scheduling of the data needs to be coupled to the weights. There are several solutions for scheduling the neural network graph, for instance TensorFlow Lite for microcontrollers specifically for microcontrollers. Where these libraries schedule the neural network, the execution of the different layers is handled by a different library on a lower abstraction level. For ARM microcontrollers this abstraction layer is the Common *Microcontroller Software Interface Standard (CMSIS)*.

2.5.1 Computing neural network layers

A popular implementation to compute a convolutional layer on processors and microcontrollers is with a combination of the im2col transformation and generic matrix multiplication (GEMM) algorithms [27, 28, 29]. This method is also used in the ARM CMSIS library.

The im2col step flattens the input image into an intermediate matrix. Each column of this matrix holds the pixels that would overlap with the kernel for every position of the output matrix. The intermediate matrix can then be multiplied with the kernel, flattened to a vector, see Figure 2.14. Each convolution window is expanded to a column, which then forms the rows of the matrix with the other columns. Flattening the input image is a bit more complicated, since the dimension of the flattened input should be matching with the flattened kernel in

order to be eligible for matrix multiplication. Combined with single instruction multiple data (SIMD) instructions, matrix multiplications can be performed partly in parallel.

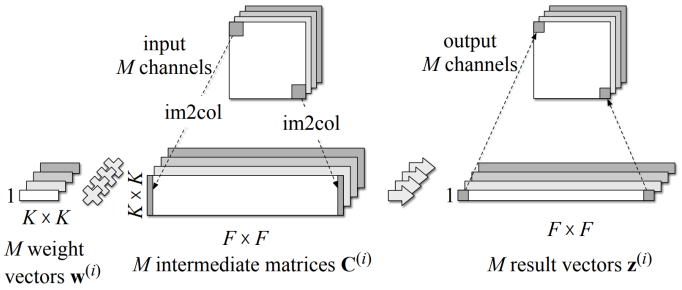


Figure 2.14: With the kernel flattened and the input transformed with im2col, the output can be computed by means of matrix multiplication [7].

The computation of a fully connected layer can be represented by an input vector and a weight matrix. In doing so, the GEMM algorithm can be used in similar fashion to a convolutional layer.

Chapter 3

Previous work

With some of the necessary background theories covered, other works in the area of NAS and quantization for microcontrollers can be discussed. In this chapter, findings and results of other works are presented and discussed to serve as a basis for the design choices in Chapter 4 and a baseline for the results in Chapter 5.

3.1 Neural Architecture Search

The work of Zoph et al. [30] implemented a cell-based search space by defining two types of cells: a normal cell and a reduction cell. This search space has been given the name NASNet. The contents of the cells are to be determined during the execution of the RL-based NAS algorithm and are comprised from convolutional and nonlinear elements. The combination of the final network is determined beforehand, but is a certain combination of reduction and normal cells. This decision was based on the success of repeated patterns in NN algorithms such as Inception [31], VGG [32], and ResNet [33] [30, 5]. A version NASNet designed for CIFAR-10, got an error rate of 2.4% at 27.6M parameters, which was state-of-the-art at that time, outperforming all other works. On ImageNet NASNet achieved 82.7% accuracy with 88.9M parameters. This performance is on par with the state-of-the-art, whilst requiring 56M less parameters than other state-of-the-art works.

The study in [21] researched methods to bound the resource utilisation of NAS-generated networks. This has been achieved by evaluating other constraints besides accuracy, resulting in a multi-objective evolutionary algorithm. The resource utilisation and accuracy of the generated networks are combined to find the pareto optimum networks. Pareto optimum networks lie on the asymptote of the accuracy/resource utilisation trade-off, so a network cannot achieve a higher accuracy without increasing resource utilisation. Other optimizations that have been implemented were weight inheritance between parent and child networks. The resulting LEMONADE network resulted an error between 4.57% and 2.58% on CIFAR-10, with 0.5M and 13.1M parameters respectively.

Unsatisfied with the performance achieved by evolutionary networks implementing tournament selection, [22] modified the algorithm. Instead of promoting the best performing networks and removing the worst performing networks form the population, the contents of the

population have been determined by age. This ageing evolution, named regularized evolution, reduces the greediness of tournament selection and results a wider exploration of the search space. This resulted in an ImageNet accuracy of 82.8% with 86.7M parameters, which outperformed the work of [30]. On CIFAR-10, the achieve test error is 3.34% with 3.2M parameters.

3.2 Quantization

Inspired by ternary weight networks and binary neural networks, [23] proposed an integer quantization scheme to compress and approximate neural networks that otherwise would require floating point arithmetic. They managed to compress a network with a factor 4 by quantizing all weights and activations to 8-bit integers, whilst improving the inference efficiency with optimizations from the ARM NEON library. For training the networks, [23] implemented quantization-aware training to simulate the quantization losses during training.

The work of [34] found out that the large number of parameters that must be represented in a per-layer quantization scheme negatively impacted the accuracy of a neural network. Instead, they proposed a per-channel quantization scheme, where the scale and zero point are computed for every convolution layer channel instead of for every layer. Additionally, a mixed-precision method has been created to reduce the memory requirements of the neural network to be deployed. With this scheme, an integer-only quantized version of MobilenetV1, a CNN targeted towards mobile deployment, could run on a microcontroller with 512 kB of memory and 2 MB storage. The accuracy achieved was 68%, which is an improvement of 8% over 8-bit quantized networks.

The study [24] has written a whitepaper about quantization techniques for neural networks that proposed the support for 4, 8, and 16 bit precision datatypes. The paper includes experiments and recommendations on how to quantize NNs. They state that quantizing networks to 8 bits invokes a 2% accuracy loss, regardless of whether the quantization is implemented per-layer of per-channel. The recommendation is to use per-channel quantization, however, since that yields higher precision of the quantized parameters. Quantization-aware training is said to further improve accuracy losses, down to 2% for 8-bit quantization schemes. Sub-byte quantization is also allowed with QAT, resulting between 2% and 10% for 4-bit networks. Compared to floating point parameters, a speedup of a factor between 2 and 3 is observed, which increased to a factor 10 when fixed point SIMD instructions are supported.

3.3 NAS and edge devices

The first NAS algorithm that implemented multi-objective constraints to target CNN inference on MCUs is proposed in [35]. The sparse architecture search targeted very tight constraints, such as a memory limit of 2 kB. This aggressive limit was met with the development of a NAS algorithm and network pruning. On the MNIST dataset, an accuracy of 98.64% has been achieved, with a storage footprint of 2.77 kB and a memory requirement of 1.96 kB. The NAS algorithm incorporated weight inheritance and a chain-structure search space.

Besides memory and storage, [36] focused on the number of operations to improve inference

time on MCU platforms. The result is a NAS algorithm according to the one shot principle by training a supernetwork. When choosing network algorithms from the supernetwork, options that exceed the memory and storage bounds are discarded. The final networks is quantized to 8 bits.

Also [37] turns to NAS as a tool to design neural networks under tight constraints, targeting medium-sized microcontrollers with a maximum of 64 kB memory or storage. They state that the one shot search space limits the number of networks that can be evaluated, although more complex datasets such as ImageNet can targeted. Compared to [35], more attention has been given to how networks are executed on MCUs. With a granular, or chain-structured, search space, an accuracy of 99.19% is achieved on the MNIST dataset, with a complexity of 28.5k MACCs. On a binary version of CIFAR-10 that is reduced to two output classes, an accuracy of 86.49% is achieved with 384k MACCs.

Another work using the one-shot search space is [2], although the main contribution is a co-design of network and inference runtime code. Interestingly, the search algorithm is split up in two stages. The first of which determines the size of the input and the depth of the model, all options that influence the size of the resulting network. This is different to the choice for pareto-optimum networks from other work [21, 37].

3.4 Inference on microcontrollers

Researches at ARM [27] have developed a performance optimized library for neural network inference on their microcontroller line-up. Software kernels have been implemented to most importantly efficiently compute convolutional and fully connected layers. These kernels make use of SIMD instructions that parallelise the loaded data by a factor two. The im2col kernel cannot compute an entire convolutional layer in one go, limited by the available memory and therefore computes the output in kernels of 2 by 2 pixels. Compared to a baseline using the CMSIS-DSP library, this new CMSIS-NN runtime resulted an increase in throughput of 4.6 times of a small CNN targeting the CIFAR-10 dataset.

Capotondi et al. [38] have taken advantage of the open-source CMSIS-NN library and optimized it to support mixed-precision datatypes of 8, 4, and 2 bits. This allowed them to store more weights in the same footprint, though the convolution layer is computed analogue to the works of [27]. Compared to a 8-bit quantized model, the accuracy achieved on MobilenetV1 is 8% higher in the same storage footprint. As expected, most of the latency originates form the lack of SIMD operations on lower bit-width datatypes.

Chapter 4

Method

The goal of this work is to investigate the inclusion of network quantization in a NAS algorithm and its impact on the accuracy/performance trade-off during tinyML inference. Based on the background knowledge and insights gained from previous works, a method to achieve this goal has been set up. This chapter covers the method used to investigate four aspects of this work, that will be used to answer the four sub research questions. Section 4.3 covers the creation of a NAS algorithm targeted with the creation of a CNN for MCU inference. How this NAS algorithm is aware of the resource constraints is discussed in Section 4.2. The best-suited quantization method and how it is implemented with the NAS algorithm is described in Section 4.4. Before that, however, Section 4.1 gives an introduction to the types of applications, datasets and software used for this work.

4.1 Target application

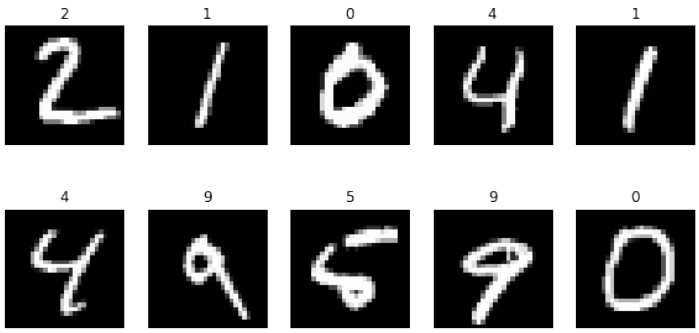
Before starting the design, it is good practice to determine the target application. With knowledge of the application, the datasets and their dimensions can be determined. For the target platform, the constraints for memory and storage can be determined.

4.1.1 Datasets

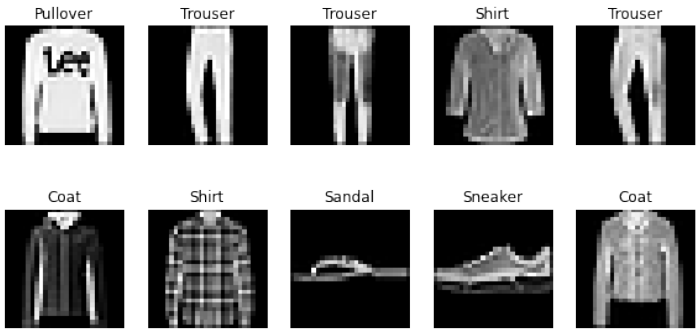
Three datasets will be used to verify the operation and performance of this work, each with a different level of complexity: MNIST [39], FashionMNIST [40], and CIFAR-10 [19]. These datasets are relatively small and simple with 10 output classes compared to ImageNet for instance, which is a dataset with 1000 classes. The required complexity for neural networks is therefore expected to be low. Low complexity networks are better suited for MCU deployment than more complex networks, given that the computational resources are restricted.

The MNIST dataset is a set of hand-written numbers, ranging from 0 to 9, intended for a classification task. The numbers, plotted in Figure 4.1a are black on a white background. This high contrast makes this a very simple dataset that is a good method to verify initial performance and for visual debugging. The input resolution is 28 by 28 pixels and there is only one channel for colours. The output is given in 10 categories for the numbers 0 to 9.

Inspired by MNIST, the FashionMNIST dataset has been developed with the intent to classify



(a) MNIST



(b) FashionMNIST

(c) CIFAR-10

Figure 4.1: Example data for the datasets used in this work

pieces of clothing. Some examples of images from the dataset have been visualised in Figure 4.1b. It is a bit more complex to solve than MNIST, because the images with pieces of clothing are stored in grey scale, resulting in a lower contrast between the features and the background. In- and output dimensions are similar to MNIST.

The CIFAR-10 dataset is a subset of CIFAR-100, a classification dataset with 100 categories of objects. The types of objects are of a larger variety than the numbers of MNIST and clothing of FashionMNIST, which makes the tasks more complex. Additionally, as can be observed in Figure 4.1c, the images are stored in colour, which adds more layers to the input and therefore more combinations of features for the network to explore. Images are stored with a resolution of 32 by 32 pixels and 3 channels, for the RGB values.

4.1.2 Hardware

For the experiments in this work, a mid-range microcontroller form STMicroelectronics has been selected. The STM32F446RE is equipped with an ARM Cortex M4 processor that features the ARM DSP-related SIMD instructions. These instructions are taken advantage of by the CMSIS-NN code library, supporting parallelization of two MACC operations and loading two variables simultaneously from storage. The chip has 128 kB of memory and 512 kB of storage [41].

4.2 Memory and storage prediction

The prediction of the model size of a neural network is relatively straightforward. To start the storage estimation, one first has to compute the total number of parameters in a network. These parameters include the weights and biases, but not the activations. With all parameters known, it is a matter of multiplying the number of parameters with the number of bits used to represent each parameter as in Listing 4.1. Note that this is solely the storage space required for the model weights and biases, not the code of the inference runtime.

```
for every layer in the network:
    count the weights in the layer
    count the biases in the layer
    total parameters = weights + biases
    storage requirement = total parameters * bits to represent parameters
```

Listing 4.1: Algorithm to predict the storage requirements of a neural network.

The weights and biases of a model are read-only and can therefore be loaded from storage into the registers directly. Flash memory (storage) access times are high compared to SRAM memory. Since no write operations are required for the weights, clock cycles are saved compared to reading and writing activations to storage. Activations are, with the opposite motivation, stored in the memory. The interesting metric is the peak memory which, given that the network is computed on a per-layer basis, is the maximum memory usage to compute a layer of the network. This per-layer execution is used by the TensorFlow Lite for microcontrollers runtime. For every layer, the input resolution and channels are added to the output resolution and channels, resulting in the number of parameters in memory per layer, as in Listing 4.2. Multiplying these parameters with the number of representation bits results the memory usage per layer [42].

```
for every layer in the network:
    count the input activations of the layer
    count the output activations of the layer
    total activations = input activations + output activations
    memory usage = total activations * bits to represent activations
    if (memory usage > than peak memory usage):
        peak memory = memory usage
```

Listing 4.2: Algorithm to predict the peak memory usage of a neural network.

Neither of Listing 4.1 and 4.2 take the overhead introduced by any inference runtime environment into account.

4.3 Neural architecture search

The design choices made for the algorithm are discussed in this section, split in the three pillars that define most NAS implementations. The block diagram in Figure 4.2 visualizes how the different parts discussed in this section work together.

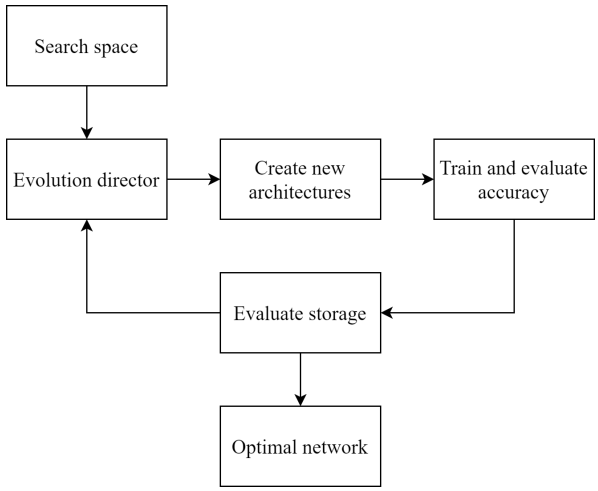


Figure 4.2: The process of network architecture search

4.3.1 Search space

Possible search space implementations have been discussed in Sections 2.2.1 and 3.1, these options are chain-structured search spaces and cell-based search spaces. Cell-based search spaces tend to result in networks that perform well on large datasets, at the cost of larger networks than with chain-structured search spaces. The search space elements of a chain-structure are less complex and more compact, given that they are individual layers instead of micro architectures. To be more flexible with the search space, a chain-structure has been chosen.

The search space consists, as most neural networks, of several convolutional layer options, a pooling layer, and fully connected layer. The input dimensions of all these layers depend on the output dimensions of the previous layer, except for when these layers are the first or

the last layer in the network. In this exception, the first layer in the network has its input dimensions defined by the dimensions of inputs from the dataset. The last layer is always a fully connected layer with as many outputs as there are output classes in the dataset, this is a necessary classification layer.

Keeping in mind that activation layers are used to introduce nonlinear behaviour between layers, a layer is automatically followed up by an activation layer. This activation layer always is a *rectified linear unit* (ReLU) layer, since it is an easy to compute and effective layer that is used in most NNs. The ReLU operation returns a 0 for negative numbers and passes through the input for other numbers, see Equation 4.1.

$$ReLU(x) = \begin{cases} x, & x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

The final layer is once again an exception, with its activation layer being a SoftMax layer. This layer is common for classification algorithms and normalizes the outputs. Lesser bound are the convolution layer kernel depths and fully connected layer output features; these are assigned a value randomly. For convolutional layers, the depth is constrained depending on the position in the network. Deeper position layers feature more channels than convolutional layers earlier in the network.

Unique to this work, the quantization levels are also part of the search space. The motivation behind this implementation is that it will result a balance between number of parameters and precision of those parameters. When architectures created by NAS fit within the resources of the target platform, the parameters of that network can be quantized to 8-bits, as is conventional for most microcontrollers. Networks that would exceed the memory or storage constraints would need a more aggressive quantization scheme. As found in [24], low precision weights negatively affect the accuracy of a model because of a larger quantization error. Keeping the quantization levels as high as possible should result in good performing neural networks that fit on the target device.

Summarized, the elements in the search space are:

- 2D convolution layer with kernel sizes of 3×3, 5×5, or 7×7 pixels and a randomly defined number of channels
- 2D max pooling layer with a size of 2×2 pixels
- Fully connected layer with a randomly defined number of features
- Quantization of convolution layer between 2 and 8 bits
- Quantization of fully connected layer between 2 and 8 bits

4.3.2 Search strategy

In the NAS algorithm of this work, the search strategy is an evolutionary algorithm. This implies that, while the generation and comparison of networks in the search are determined by means of evolution, the network parameters themselves are trained as one would train a regular NN [43].

Evolution is a slower process than reinforcement learning, but it has the advantage of being faster to set up. Configuring the controller of a RL network is a precise and time-consuming task. The choice has been made to base the search algorithm on evolution, since getting the search up and running is preferred to the faster execution of RL. Time not spent on perfecting a controller can be used to explore extensions of the search algorithm and the integration of quantization-aware training.

With evolution, the choice is between tournament selection and regularized evolution [22]. From the works in Section 3.1, tournament selection more quickly reaches a maximum in the search space. Therefore, tournament selection is to be used in this work.

```
while (current generation < number of generations):

    while (current mutation < number of mutations):
        select a parent network from the population
        copy the parent network
        randomly select an element from the search space
        place the element in the new child network
        place the child network in the population

    evaluate the child networks from the current generation

    remove the lowest scoring models from the population
```

Listing 4.3: Pseudo code of the tournament selection evolution algorithm

4.3.3 Performance estimation

The best performing network architecture is determined by choosing the model with the highest scoring accuracy out of the population. However, this is not the only model parameter that is evaluated during the performance estimation and evaluation step.

There are two constraints implemented with the NAS algorithm, these constraints are the available memory and storage space. These are important constraints, as memory overflow during inference must be avoided. Additionally, the size of the model should be within the limits of the available storage. As larger networks tend to achieve better accuracy and the goal of this work is to find a network that will fit within the memory and storage of a MCU, these boundaries must be included. For these boundaries, storage and memory limits have been set up, making the search algorithm a multi-objective NAS. During the network generation phase, networks that exceed the boundaries are removed from the population with the intent to satisfy the boundaries.

As a method to limit the time required to complete the NAS algorithm, low fidelity estimates have been made of each network by training a network for a few epochs. This coarse estimate for the network performance is further optimized by weight inheritance between parent and child networks. By keeping the weights for all layers that have not been influenced by the mutation. The child network then has some tuned parameters before the first epoch of training, which can kick-start the training of this new architecture. Only the parameters of the modified layers are defined randomly.

4.4 Quantization strategy

As discussed in Section 4.3.2, the NAS algorithm of this work defines a search space that requires networks to be trained after being generated. If a one shot search space would have been chosen, the supermodel could be quantized after training had finished.

Because of the desire to include the quantization process with the NAS algorithm at a point where the network would not be fully trained, the quantization strategy has to be quantization-aware training. Network architectures can with QAT be trained for a few epochs to get a performance validation of the quantization levels. An added benefit of using QAT, as stated by the work of [24], is that it tends to yield lower accuracy losses than post training quantization and handles low quantization levels better.

Brevitas is a framework for quantization-aware training *PyTorch* with support for any number of bits to represent integer variables, this in contrast with the *PyTorch* and *TensorFlow Lite* quantizers that support only 8-bit integers [44]. Although initially targeted towards FPGA development, networks developed with Brevitas can also be exported to *PyTorch* and *ONNX* formats that are suited for deployment on embedded devices and microcontrollers.

When creating a quantized neural network layer with Brevitas, information about the quantization scheme is required in addition to the parameters needed for regular *PyTorch* layers. In Listing 4.4, for example, the creation of a convolutional layer with 6 input channels, 16 output channels, and a kernel of 5 by 5. Setting quantization parameters is done by assigning a type of quantization with `weight_quant` and a bit width with `weight_bit_width`. The imported class `Int8WeightPerTensorFloat` is a structure of parameters that define how a parameter is quantized. In this case an 8-bit integer weight that is computed for every channel in a layer, scaled by a floating point value.

```
import torch.nn as nn
import brevitas.nn as qnn
from brevitas.quant import Int8WeightPerTensorFloat as WeightQuant

layerPyTorch = nn.Conv2D(6, 16, 5)

layerBrevitas = qnn.QuantConv2D(6, 16, 5,
                                weight_bit_width=8,
                                weight_quant=WeightQuant,
                                return_quant_tensor=True)
```

Listing 4.4: Definitions of 2D convolution layers in PyTorch and Brevitas. For the latter, additional parameters configuring QAT are required

4.5 Parametric network design

To streamline the process of modifying neural networks a parametric network class has been created in PyTorch. With this class, it is both possible to quickly generate networks with different architectures and conveniently modify layers. This goal can be achieved by creating layers from scratch, however this network makes use of basic PyTorch layers for compatibility and development time reasons. The layers are ordered in a modular fashion, in combinations that are popular in other works.

To achieve the modular structure, the networks is built up with dictionary objects in a 2-level hierarchy. The block level dictionary contains a regularly occurring combination of layers. A convolution module, for instance, always contains a 2D convolution layer with optionally activation and/or pooling layers. The top level dictionary contains the blocks of layers and provides a structure to forward data through the network. As illustration, Figure 4.3 shows the basic LeNet network in this configuration.

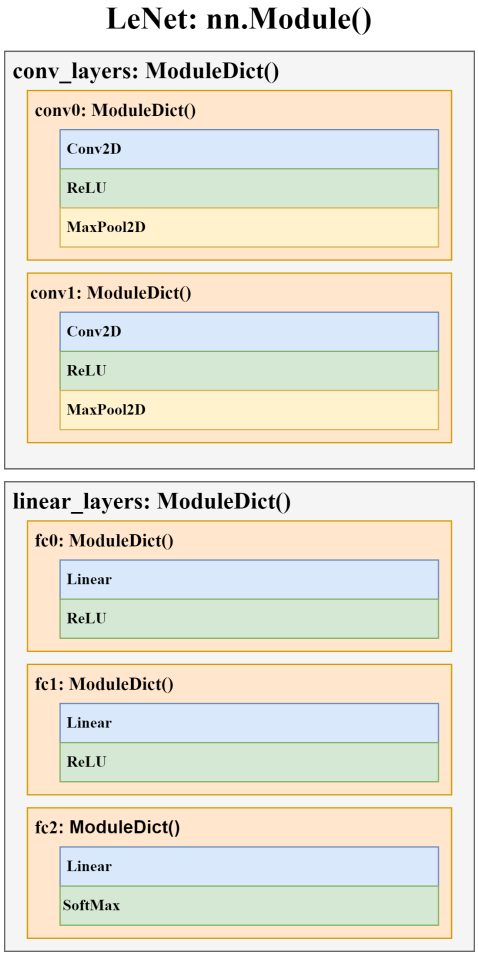


Figure 4.3: LeNet architecture as an example for a neural network created with parametric network design

4.5.1 Weight inheritance

With the evolution algorithm new network architectures are created by means of mutations; modifications to architectures existing in the network pool. Such a mutation could be implemented by using the parametric network design function, copying the architecture of a network bar the parameters that are to be mutated. In principle, there is nothing wrong with this method. In network architecture search however, this would require every model generated through mutations to be trained from scratch, which extends the time needed for the algorithm to complete significantly. Mentioned in Section 4.3.3, weight inheritance between

different architectures has a positive impact on the duration of network architecture search, which is why the choice has been made to include weight inheritance between parent and child networks.

The parametric network supports weight inheritance by means of network modification functions. Examples of these functions are the `insert_layer()` and `remove_layer` methods, the function of which can be derived from the names. By copying the complete model, so it's architecture and parameters, the trained parameters are retained and only the modified layers need to be trained from scratch.

4.6 Experimental setup

Each of the simulations concerned architecture searches that have been performed with the Python packages and versions from Table 4.1. These searches will be performed in duplex on three different datasets, each of increasing complexity: MNIST [39], FashionMNIST [40], and CIFAR10 [40].

Table 4.1: Software versions used

Software	**Version**
Python	3.8.7
PyTorch	1.10.1
Brevitas	0.7.1

Because of the number of experiments, the choice has been made implement a narrow search algorithm, which spans 10 generations. Every generation, 5 new networks are generated and added to the population. In return, the 5 oldest networks are omitted at the end of every generation. The performance evaluation spans 3 epochs in which the networks will be trained, this should be enough to get an idea of the performance of the population.

Table 4.2: Parameter settings for the search algorithm

Parameter	**Value**
Storage limit (kB)	512
Memory limit (kB)	128
Number of iterations (generations)	10
Network pool start size	10
Number of networks to be mutated each generation	5
Number of course training epochs	3
Learning rate (Adam optimizer)	0.005

4.6.1 Memory and storage analysis

The first of the experiments is created as a verification for the precision of the memory and storage predictions performed as discussed in Section 4.2. It serves as a basis for the first sub research question: *"With what precision can the run-time memory and storage requirements for neural network inference be calculated during the execution of NAS?"*.

This comparison will be made in the STM32CubeMX program, which is a configuration tool for ST microcontrollers. This program comes with the X-CUBE-AI extension pack that supports inference for TFLite, ONNX, and Keras models. TFLite models can be executed in the TFLite for Microcontrollers runtime environment or with the STM32Cube.AI inference engine. ONNX only supports the latter. The analysis tool provides the user with information about the memory and storage footprints, as well as MACC operations and a graph of memory usage for different layers. The results of this analysis tool will be compared to the predictions made in the NAS algorihtm.

4.6.2 Quantization as part of a NAS search space

From the works discussed in Section 3.1 has been observed that the contents of the NAS search space are of influence on the level of the result and the time required to reach those results. The results should be sufficient to answer the second sub question: *"What are the consequences of including the quantization of neural network parameters as part of a NAS algorithm to the complexity of the search space?"*.

To verify whether the addition of quantization within the NAS algorithm does not result in worse results, a comparative test will be performed on three cases:

- Floating-point networks that have been quantized to 8-bit integer when training was finished. The bit width of 8 has been chosen since it is supported by the mainstream methods to quantize in TFLite and PyTorch.
- Networks that have been trained quantization-aware with 8-bit integer parameters. This provides a reference for QAT to the PTQ algorithm with a comparable level of quantization.
- N-bit networks that have been quantized using QAT.

The accuracy, storage, and memory of these three implementations will be compared with the goal to find out whether there are differences between the implementations.

4.6.3 NAS design of a CNN for tinyML

The third research question is about the performance of the NAS-generated neural networks: *"How are the accuracy and resource utilization of an image classification convolutional neural network that has been implemented in a microcontroller using NAS?"*. Three experiments, each containing four runs, will be performed. One for each of the datasets mentioned in Section 4.1. Because the NAS is multi-objective and therefore has the constraints for memory and storage, these experiments will provide the balance between the achievable accuracy and the resource constraints.

4.6.4 Sub-byte SIMD prediction

The final subquestion is targeted towards finding the difference in inference time between conventional (8-bit) quantization strategies and sub-byte quantization. This question will be answered with both literature and a MCU simulation script, given that no mainstream hardware that supports sub-byte data on an ISA-level. The ARM CMSIS-NN library will be

used as a basis, though extended with SIMD support for not only 2 8-bit MACCS, but also 4 4-bit and 8 2-bit MACCS. Similar for load/store operations.

Chapter 5

Results

In this work, four experiments have been set up to answer the sub research questions stated in Section 1.1. The experiments have been introduced in Chapter 4. In this chapter, the results for each of these experiments will be presented, accompanied with a discussion to analyse the outcomes. This chapter presents the results to the experiments introduced in Chapter 4 with in sections with the following sturcture:

- Section 5.1: *"With what precision can the run-time memory and storage requirements for neural network inference be calculated during the execution of NAS?"*
- Section 5.2: *"What are the consequences of including the quantization of neural network parameters as part of a NAS algorithm to the complexity of the search space?"*
- Section 5.3: *"How are the accuracy and resource utilization of an image classification convolutional neural network that has been implemented in a microcontroller using NAS?"*
- Section 5.4: *"How does a sub-byte quantized neural network compare to more conventionally quantized network inference in terms of operations and inference time?"*

5.1 Memory and storage analysis

The memory and storage requirements as predicted by the NAS algorithm are computed with the runtime algorithm of *TensorFlow Lite for Microcontrollers* (TFLM) as example. The exact overhead of this runtime is unknown and partially depends on the dimensions of the neural network. Additionally, the STM32Cube.AI inference engine is used for comparison with networks exported to the ONNX format. This runtime is optimized for STMicroelectronics devices and implements a different method of executing the inference.

From Figure 5.1, this overhead is visible as the difference between the non-optimized PyTorch (dark blue) and TFLite micro (light blue) bars. It appears as that the overhead introduced by the TFLite micro runtime engine is in the order of 1 kB, something that is confirmed by Table 5.1. The difference between the non-optimized PyTorch prediction and TFLite micro is 2.9 kB for the memory and 2.3 kB for the storage for inference of the model targeted towards

MNIST. The figures for the CIFAR-10 network are similar; 2.6 kB and 2.3 kB for the memory and storage respectively.

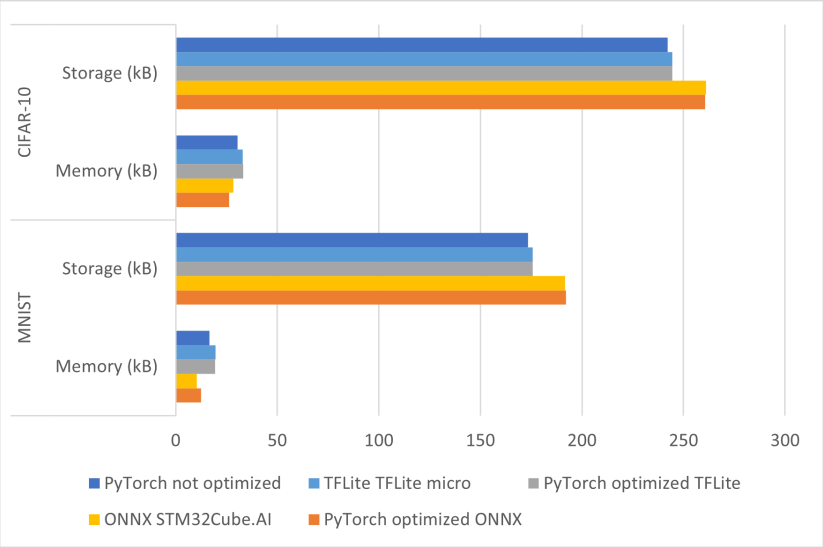


Figure 5.1: Memory and storage predictions from the NAS algorithm (blue), plotted with the STM32Cube.AI analysis results (grey, yellow) and overhead-optimized predictions (light blue, orange)

Table 5.1: Different implementations for memory and storage prediction for a LeNet-5 image classifier network on the MNIST and CIFAR-10 datasets.

Format	**Runtime**	**MNIST**		**CIFAR-10**	
		Memory (kB)	*Storage (kB)*	*Memory (kB)*	*Storage (kB)*
PyTorch	not optimized	16.6	173.5	30.4	242.2
	optimized ONNX	12.5	192.0	26.3	260.7
	optimized TFLite	19.3	175.8	33.2	244.5
ONNX	STM32Cube.AI	10.4	191.7	28.4	261.1
TFLite	TFLite micro	19.5	175.8	33.0	244.5

For the STM32Cube.AI inference engine, the results deviate more from the non-optimized PyTorch prediction. The memory usage tends to be lower than the prediction by observing the results in Figure 5.1. With 6.2 kB, this difference is about three times larger for MNIST than for CIFAR-10 with 2.0 kB. Whether this difference between the datasets has anything to do with the three colour channels in the CIFAR-10 dataset, as opposed to one with MNIST, is something that is definitely interesting to look in to. The storage overhead of the STM32Cube.AI runtime is for both the MNIST and the CIFAR-10 networks about 18 kB.

With the runtime overheads found from these experiments, a more accurate prediction can be made for the memory utilisation and storage requirements for a TFLite runtime. These predictions are plotted in Figure 5.1 in grey for TFLite and orange for ONNX and STM32Cube.AI. With this optimization, TFLite runtime memory utilisation can be predicted with 0.2 kB deviation. The storage requirements have no notable deviation from the TFLite runtime on

both datasets.

The prediction accuracy for the STM32Cube.AI runtime is less accurate, with clear differences between the optimized prediction in orange and the yellow STM32Cube.AI generated numbers. The inference engine of STMicroelectronics uses a scheduling of the network that differs too much from the TFLite implementation that has been used to predict the PyTorch generated numbers.

5.2 Quantization as part of a NAS search space

To compare the different quantization algorithms, the following three experiments have been performed:

- Floating-point networks that have been quantized to 8-bit integer when training was finished. The bit width of 8 has been chosen since it is supported by the mainstream methods to quantize in TFLite and PyTorch.

- Networks that have been trained quantization-aware with 8-bit integer parameters. This provides a reference for QAT to the PTQ algorithm with a comparable level of quantization.

- N-bit networks that have been quantized using QAT.

In general, three trends for the three quantization methods in this experiment are emerging in Figure 5.2. First of all, the memory requirements are very stable. Most of the datapoints in Figure 5.2a are positioned on the same position in the x axis, though with improving accuracy for networks form different generations. Overall, most of the quantization-aware trained networks result in a lower memory usage than the post-training quantized networks. With exception of the two outliers at 4 kB, it is appears that the 8-bit QAT networks are significantly more memory-efficient with a 22% reduction, while only sacrificing 0.2% accuracy.

The PTQ-generated model scores the highest accuracy, although it achieves this by just 0.1% and requires both more memory and storage to achieve this. It is reasonable to assume that this result is a luck of the draw, with the other searches missing out on this specific network architecture. In terms of storage, the plot in Figure 5.2b shows several datapoints for the quantization-aware trained networks in the top-left corner, indicating high accuracy and low network size. The accuracy/footprint trade-off of the sub-byte quantized network (grey) is promising, scoring on par with the 8-bit QAT network (blue) at lower memory and on average about equal storage requirements.

Finally, the plot in Figure 5.2c confirms the memory efficiency of the sub-byte quantized network. Most of the datapoints in the lower-left area of the plot belong to this network, indicating a low memory usage combined with low model footprint. Overall, the networks with the different quantization levels and techniques show differences in spacial efficiency and performance, though where one implementation is scoring better, others are more efficient in resource utilisation.

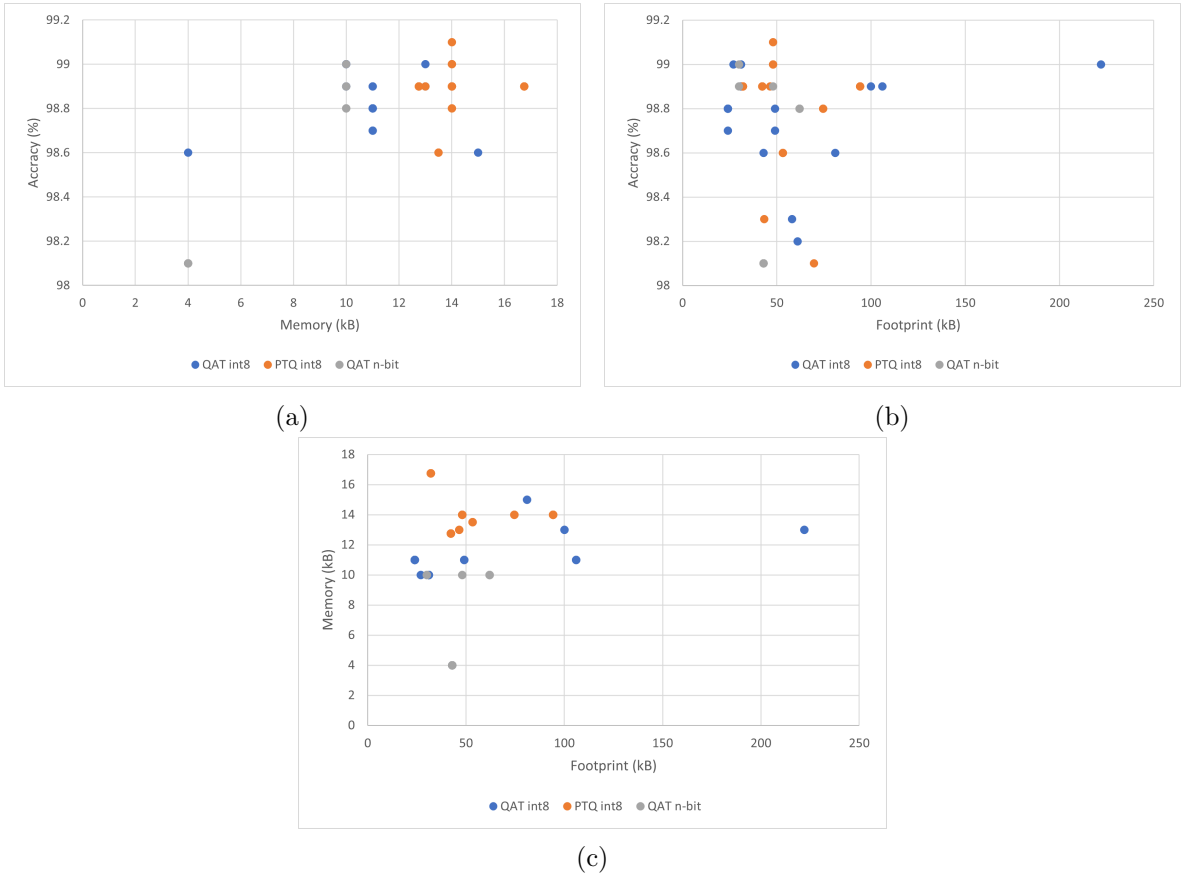


Figure 5.2: Memory and storage requirements for post-training quantization (orange), quantization- aware training (blue), and sub-byte QAT (grey). In (a), the accuracy/memory trade-off on a MNIST classification is plotted. The same is plotted in (b), though with storage over accuracy. The last plot, (c), visualizes the connection between memory usage and model footprint for each of the quantization techniques.

5.3 NAS algorithm for tinyML

The third sub research question is about the capabilities of the NAS algorithm with respect to the accuracy and resource utilisation in the context of a CNN suited for microcontroller deployment. Three experiments have been performed, evaluating the NAS performance for the three dataset introduced in Section 4.1. For each of these experiments, a search has been performed with a duration of 10 generations with 5 newly created architectures per generation. In combination with a start population of 10 networks, this brings the total number of explored networks to 60. As a measure to reduce the execution time of the NAS algorithm, these searches have been performed with a reduced training dataset and a limited number of 5 epochs. The MNIST-derived datasets were evaluated with 10000 out of the 60000 training images, the CIFAR-10 dataset with 25000 out of the 50000, to accommodate the added complexity of this dataset. Note that because of this smaller training set, the results presented in this section might show a lower performance than other works. A better benchmark is provided in Section 5.3.3.

Each of the experiments covers four independent runs of the NAS algorithm, as a verification of the repeatability of the search algorithm. The data presented for every experiment is indicated by the name of the target dataset followed by an index indicating which run of the experiment is represented.

5.3.1 Accuracy and network size

The relation between the accuracy and storage requirements, or network size, is of interest because it can provide insights in the efficiency of a neural network. The point clouds in Figure 5.3 plot the accuracies against the storage requirements of all networks in the search experiments. In this case, the efficiency of the network architectures depends on the accuracy storage trade-off. Higher performing networks with a smaller footprint are desired, to minimize the resource utilisation of the MCU without sacrificing the accuracy of a network. From this figure, the effects of tournament selection and the formation of a pareto-optimal region can be deduced.

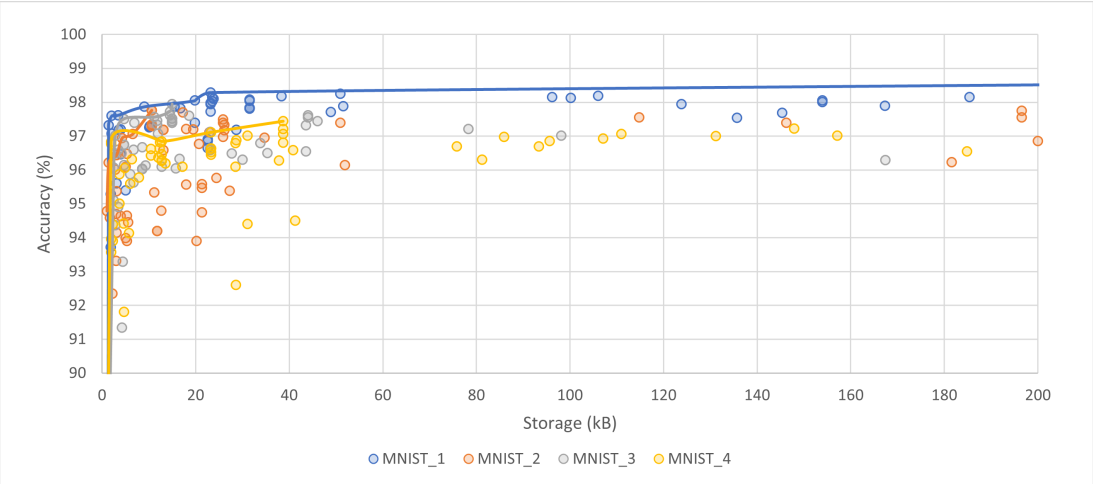
The trade-off between accuracy and the memory requirements of networks generated in this work is less insightful and therefore not discussed in this section.

The effect of tournament selection can be identified by distribution of architectures towards the top-left region of the plot areas, indicating that architectures that have a promising accuracy performance trade-off are frequently selected for mutations. The worse performing architectures are rarely used as a basis for mutations, resulting in a low density of networks outside of the pareto-optimal region. This region is situated at the datapoints where, in this specific case, the highest possible accuracy is achieved for a given storage requirement. In other words, there is no network with a better accuracy for the model size of a pareto-optimal network. For the MNIST and FashionMNIST networks in Figures 5.3a and 5.3b respectively, there are well-defined pareto regions visible by the lines that cover the full range of the storage axis. The positioning of the optimal regions for the different runs tend to differ, this is further discussed in Section 5.3.2.

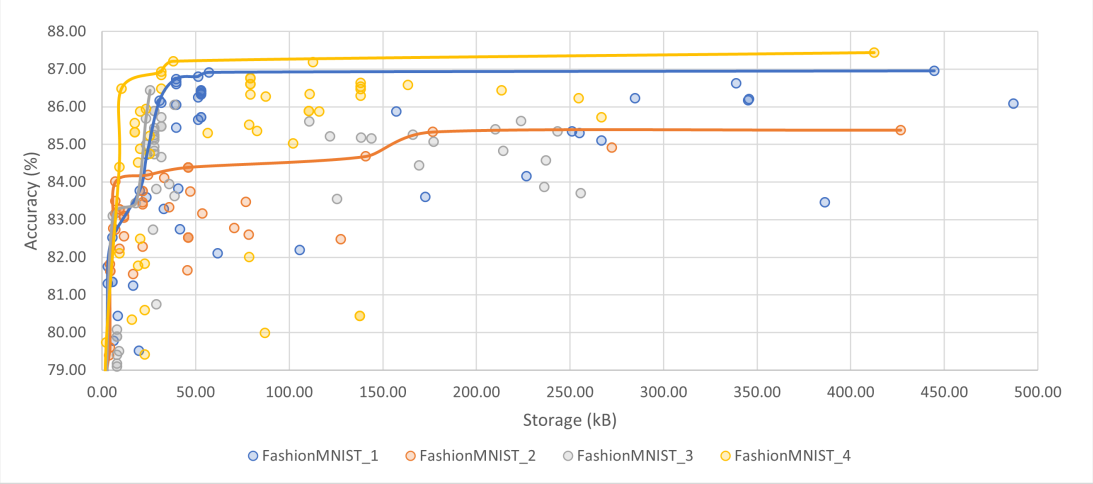
Figure 5.3c illustrates a problem with the experiments, since the pareto front is in most runs defined for networks with a footprint up to 50 kB despite the abundance of network datapoints with larger footprints. This could indicate that those networks are too large to properly train within the search, as the many parameters take more iterations of the training data before being tuned. Architectures with less parameters are trained more quickly than the large networks, making them survive the tournament selection process.

5.3.2 Improvement per generation

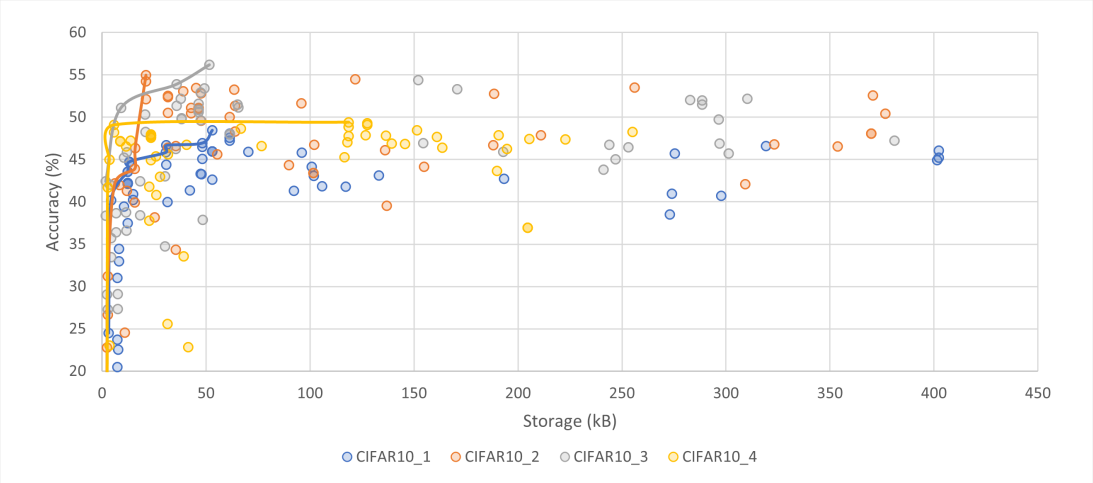
The improvement in network performance from one generation to the next can indicate when the highest scoring architectures are created during the search, and give insights in the developments of the network size and memory consumption. These developments have been plotted in Figures 5.4, 5.5, and 5.6 for the MNIST, FashionMNIST, and CIFAR-10 datasets respectively. Additionally, the memory and storage developments have been plotted, providing a basis to investigate the relation between those parameters and the accuracy during the search. For every generation, the architectures with the highest accuracy have been plotted and generation 0 indicates the initial population of randomly generated networks.



(a) MNIST



(b) FashionMNIST



(c) CIFAR-10

Figure 5.3: Accuracy-storage trade-off for the networks targeted towards the three datasets. The pareto fronts for the different runs have also been included.

In Figure 5.4, one can see that MNIST_1 achieves a high accuracy at the start and then not really improves much until later in the search. The other experiments are closer together and show a growth of about 1% over the generations, with the used storage remaining relatively constant until generation 9. MNIST_1 shows an earlier ans more steady growth, indicating that at some point, larger network architectures could start to outperform the smaller ones. Memory usage is relatively constant for measurements 1 and 3, with MNIST_2 and MNIST_4 fluctuating. Overall, the memory seems to be hardly influenced by the generation in which the network is generated.

Figure 5.5a presents the generational improvements for the accuracy and storage of networks for the FashionMNIST dataset. FashionMNIST_1 and FashionMNIST_4 score relatively good within the first two generations and stay at a relatively constant level. At the end of the search they improve in accuracy, especially in the fourth run of the experiment. That improvement at the end comes from an increase in network size, which has been relatively low for the entire search for all networks, except FashionMNIST_2. This experiment starts with a low accuracy, similar to FashionMNIST_3, but shows the best generational improvements. The model size increases for almost every network on the FashionMNIST dataset around generation 6. The memory usage is again relatively constant throughout the experiments, as is visible in Figure 5.5b.

The fourth experiment on the CIFAR-10 dataset shows on average a higher model size than the others, as can be observed in Figure 5.6. Others show the trend of the MNIST-derived datasets, where the larger networks are created further in the search. The accuracy of experiments 2 and 3 show a similar increasing trend with every generation. CIFAR10_1 never manages to create a network better than the randomly generated start network, while the fourth experiment has a low increase of accuracy over the generations. This could indicate that, although the architectures are larger than other searches, larger networks need more training data to achieve similar performance to smaller networks.

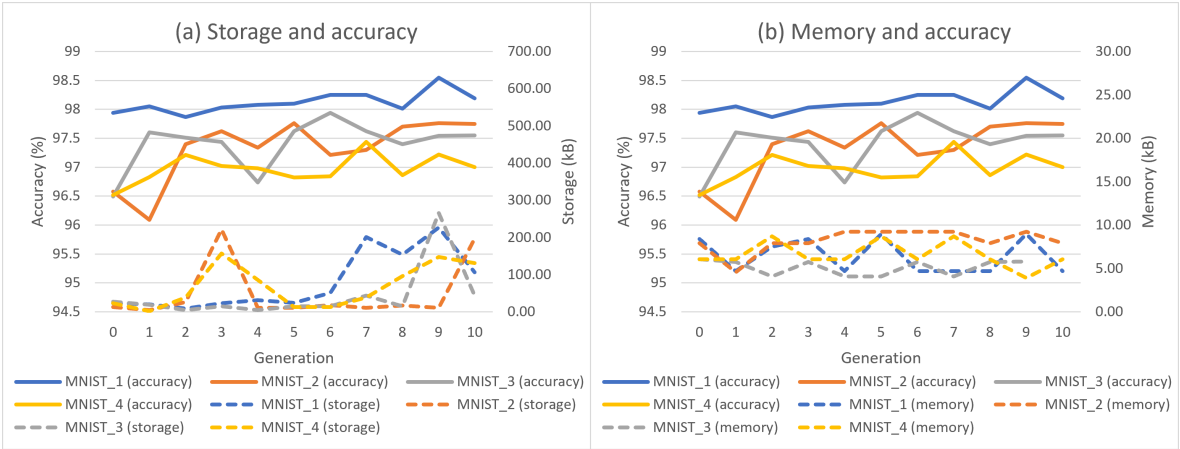


Figure 5.4: Generational behaviour of accuracy, memory and storage for the MNIST dataset models. Measured for the best network at the end of each generation.

Based on the same data as Figures 5.4, 5.5, and 5.6, the mean and standard deviation have been calculated for the different experiments. The four runs in each experiment do not yield the same results, which can be contributed to the size of the search space in comparison to the

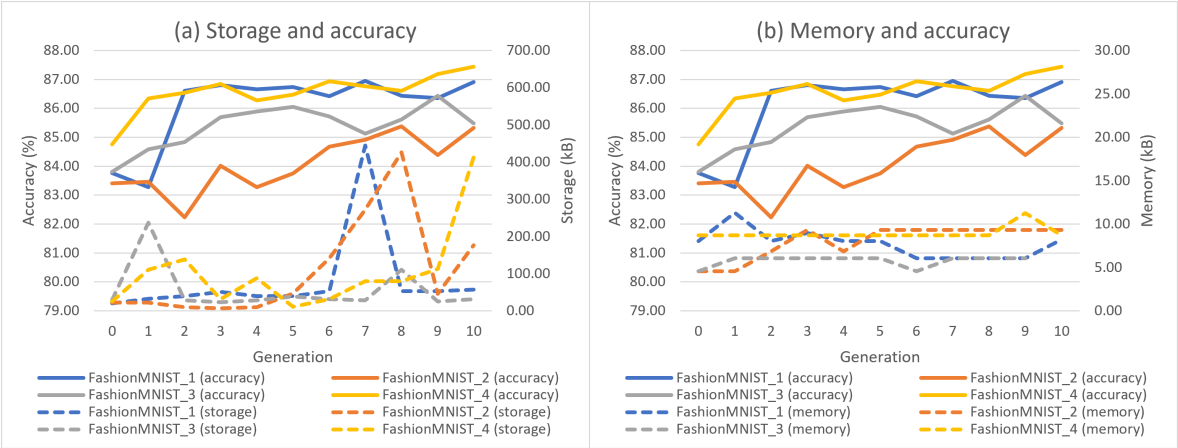


Figure 5.5: Generational behaviour of accuracy, memory and storage for the FashionMNIST dataset models. Measured for the best network at the end of each generation.

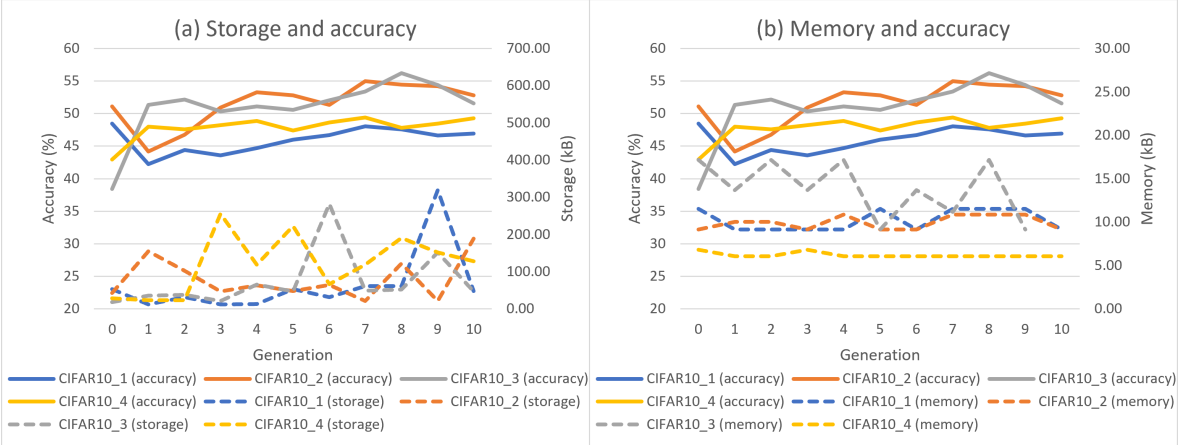


Figure 5.6: Generational behaviour of accuracy, memory and storage for the CIFAR-10 dataset models. Measured from the best network at the end of each generation.

number of networks evaluated. Figure 5.8 presents the mean and standard deviation for the accuracy in (a), memory in (b) and storage in plot (c) derived from the four runs in Figure 5.4. In an analogue manner, Figures 5.5 and 5.6 plot the same data, though for the MNIST and CIFAR-10 datasets respectively.

At the start of every search, just the initial population has been evaluated. As these are a few possibilities in a large space, the deviation between the networks is expected to be relatively large compared to later in the search. Through the generations, the different runs are expected to converge, since the tournament selection algorithm should direct the search towards better performing architectures. This phenomena occurs for all datasets in some shape or form, but is most prominent in Figure 5.5a. Until generation 9, the different runs seem to converge, but then diverge a bit as evident by the larger deviation in the last two generations. The CIFAR-10 experiment shows the least convergence between the different runs as indicated by the constant deviation from the mean in Figure 5.6a.

Over all three datasets, the memory seems the most constant parameter over the generations. Although the mean in Figure 5.4b varies between about 6 and 8 kB, the deviation is constantly following the mean.

For the CIFAR-10 experiment, the mean storage is slowly increasing for every generation. The deviation of the runs in Figure 5.6c differs greatly, indicating that none of the runs converges in terms of storage. The other experiments show a better trend of networks increasing in size for a short while near the start of the search and a large increase near the end, for instance in Figure 5.4c for the MNIST dataset.

Figure 5.7: Mean and standard deviation for accuracy (a), memory (b), and storage (c) of the runs performed for the experiment on the MNIST dataset

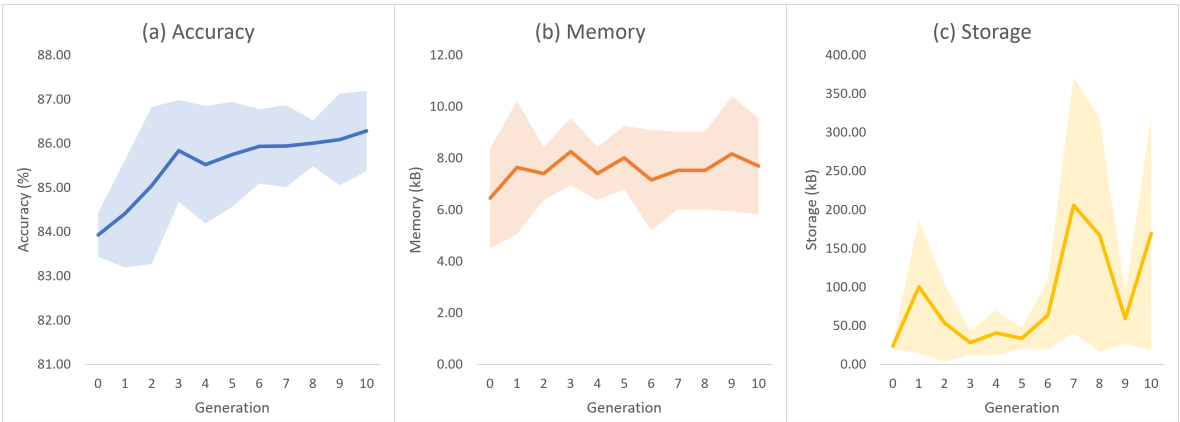


Figure 5.8: Mean and standard deviation for accuracy (a), memory (b), and storage (c) of the runs performed for the experiment on the FashionMNIST dataset

5.3.3 Full training dataset

The results provided so far were not performing very well in terms of accuracy, as the architectures have been trained on a reduced dataset in order to save time during the search. To provide a benchmark to other works, the best network of the runs in the experiments have been trained for an additional 5 epochs, now at the full set of train images. The results can be observed in Figure 5.10 and seem to provide around 3.5% improvement for FashionMNIST,

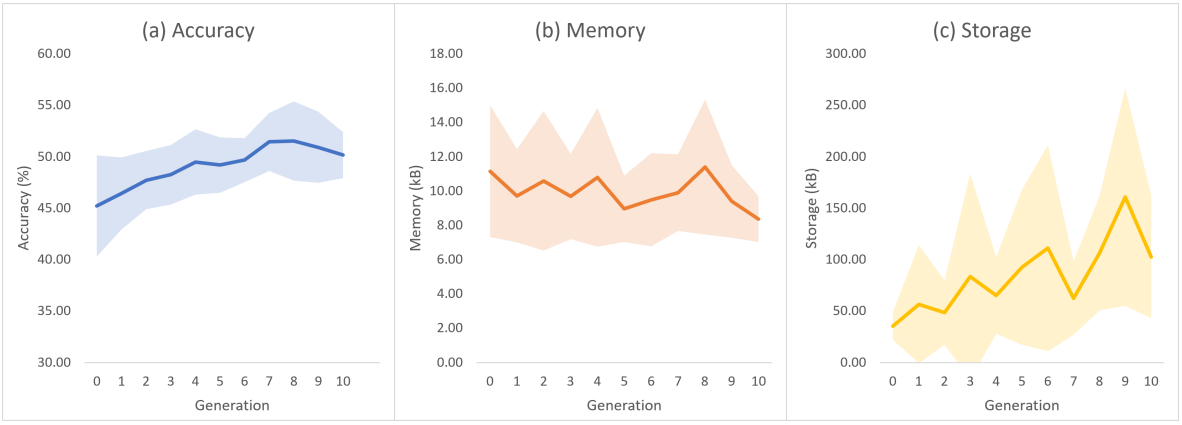


Figure 5.9: Mean and standard deviation for accuracy (a), memory (b), and storage (c) of the runs performed for the experiment on the CIFAR-10 dataset

a very low 0.5% for MNIST. The improvement fluctuates between runs, but never with more than a percent. The CIFAR-10 dataset shows a large improvement compared to the others, with a maximum of 18%.

Figure 5.10: Improvements in accuracy for the best network of each NAS run when trained on the full dataset

5.4 Inference

With the inference runtime simulation algorithm described in Section 4.6.4, an estimation is given for the effects of sub-byte quantized data and higher parallelism implementation of SIMD instructions. The required processor cycles are predicted based on the ARM CMSIS-NN library, with the assumption that he library also supports sub-byte datatypes. The results of the experiments with four different datatypes are provided in Table 5.2. Because of the higher level of SIMD parallelism, the number of cycles requires is significantly lower for every smaller datatype, Figure 5.11 shows a scale of the improvements in clock cycles.

When quantizing a neural network from 32-bit floating point to 8-bit fixed point integers, memory and storage footprints will be about a factor 4 lower. It is not exactly 4 because of the overhead that the runtime engine requires, as has been discussed in Section 5.1. This scaling can be observed in Figure 5.11 and is about a factor 3 for storage with memory just shy of a factor 3. This shows a linear relation between the number of bits used to represent a

Table 5.2: Memory, storage, complexity and required processor cycles to compute a LeNet network with floating point and several fixed-point datatypes.

Datatype	float32	int8	int4	int2
Memory (kB)	20.50	7.38	5.19	4.09
Storage (kB)	184.81	61.20	40.60	30.30
Complexity (MACC)	281876	281876	281876	281876
Cycles	3905088	1161810	704297	475991

parameter and the factor with which the memory or storage requirements are reduced. From the same figure it is possible to spot that the reduction in cycles required to compute the inference is then not linear, rather exponential.

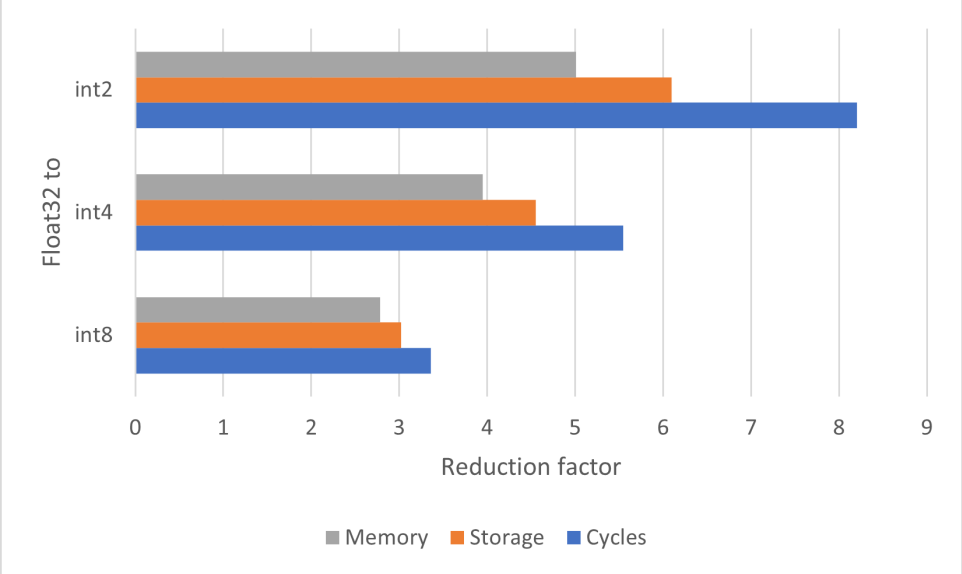


Figure 5.11: Reduction in memory, storage and processor cycles as a result of quantizing float32 to fixed point integers of 2, 4, and 8 bits.

5.5 Discussion

To conclude the presentation of the results for this work, an overall discussion will be provided in this section.

Starting with the first research question, Table 5.1 provided the insight that the computed estimations for the memory and storage requirements of a LeNet network come close to the computations made by the STM32Cube.AI network analyzer, but that the memory and storage required for the runtime program is missing. This is no surprise, though it is difficult to determine how large this overhead is exactly. With an analysis of the differences, an optimized estimation can be made for the runtimes, resulting an estimation that has a deviation between 0.6% and 1% for the memory and no error for storage depending on the network for the TFLite for Microcontrollers runtime. For the STM32Cube.AI runtime, the results are more network dependent. This averages a 8% to 16% error for memory, with the storage estimation accurate

from 0.1% to 0.2%, indicating that this method does not work for all cases. For larger neural networks this error could be mitigated by the size of the network, but smaller networks would benefit from a more sophisticated analysis of the runtime overhead.

Figure 5.2 provided the reader with a multitude of comparisons between differently quantized versions of a NAS run with the aim to investigate the differences between them. Especially of interest is the difference between post-training quantization and quantization-aware training, given that the latter would be able to be included in the search space of a NAS algorithm. An 8-bit QAT network proved to hand in a little accuracy (0.2%) for an average 22% decrease in memory usage. Extending QAT to sub-bytes has no difference in these measurements compared to the 8-bit quantized model. There is a deviation between these models, as they are products of a NAS algorithm and therefore not all identical, although the plots in Figure 5.2 paint a clear difference between the different clusters of networks.

For the MNIST dataset used for the quantization experiment, the deviation between different networks is relatively small compared to the other datasets, as proven by the third set of experiments. All experiments show a weakly converging behaviour between the different runs, but at some point a strong divergence enters. This is illustrated in plot (a) of Figures 5.7, 5.8, and 5.9. Interestingly, the more complex CIFAR-10 experiment indicates that the search is not really progressing towards a region of well performing networks in the search space. This is confirmed by the pareto-optimum regions in Figure 5.3c which only cover a small part of the networks. The MNIST-based experiments performed better, with high accuracies and a low storage footprint. The pareto fronts of those experiments differ from run to run, but cover a large part of Figures 5.3a and 5.3b. Finally, an interesting observation is that larger networks tend to perform better near the end of each search, though the accuracy does not necessarily beat smaller networks.

Although the results of the individual experiments do not show promising accuracies because of the subset of training data used for evaluation, the networks can make significant improvements when trained on the full dataset after the search. As illustrated in Figure 5.10, an increase of 18% accuracy can be achieved depending on the dataset. It must me mentioned, though, that the higher the initial accuracy is, the lower the performance gained by this training step.

For the final research question, an analysis of the performance gains by sub-byte SIMD inference has been examined with a predictive calculation. Where the memory and storage tend to decrease by a linear fashion, the clock cycles required to infer a neural network reduces exponentially.

Chapter 6

Conclusion and recommendations

6.1 Conclusion

At the start of this report, neural architecture search and quantization have been presented as tools that promise to make the inference of convolutional neural networks possible on tinyML devices, such as microcontrollers. The goal of this work is to verify that this statement is true, and evaluate the added benefit of including the quantization of network parameters to sub-byte datatypes. The main research question as introduced in Section 1.1, that will be answered in this conclusion is: *"To what extent does weight quantization as a NAS constraint have an impact on the accuracy/performance trade-off when performing the inference of a TinyML network on a microcontroller?"*

The answer to this main research question will be given in four sub research questions. These will be answered in Sections 6.1.1 until 6.1.4.

6.1.1 Memory and storage requirements

The answer to the first of the questions, *"With what precision can the run-time memory and storage requirements for neural network inference be calculated during the execution of NAS?"*, is best given by an evaluation of the two different criteria. As defined in Section 2.4, the volatile memory or SRAM memory is referred to as *memory*. For the non-volatile or flash memory, the term *storage* is used.

The proposed method of predicting the storage requirements of a network architecture by counting the number of parameters and multiplying it with the number of bits that each parameter is stored in, holds with one exception. This exception comes from the required storage for the inference runtime engine. By compensating for this runtime overhead, the TFLM engine storage requirement can in both cases in Section 5.1 be predicted with a precision of 0.1 kB. For the STM32Cube.AI runtime, the storage overhead scales with the size of the model and is therefore more difficult to predict exactly.

A similar conclusion can be drawn for the peak memory utilisation. Although, it must be taken into account that the memory utilisation is more dependent on how data is scheduled by the inference engine. The scheduling of TFLM, which loads each layer in one go in memory to compute the output, is used as a basis for the prediction algorithm. Unsurprisingly, this

results in another close prediction when the runtime overhead is compensated for. For both the networks in Figure 5.1, the peak memory utilisation can be predicted with 0.2 kB error, resulting in a prediction error of less than 1%. Although the data scheduling used for the STM32Cube.AI runtime is on a per-layer basis just like TFLM, the prediction is less accurate as the memory usage scales with the size of the input data.

On a microcontroller with 128 kB memory and 512 kB storage, these predictions should be sufficiently accurate, though caution should be taken when the peak memory utilisation is close to the 128 kB boundary.

6.1.2 Quantization as part of NAS

Embedding the quantization within a NAS algorithm expands the contents of the search space, To find out whether that has any effect on the resulting networks, the second sub question is *"What are the consequences of including the quantization of neural network parameters as part of a NAS algorithm to the complexity of the search space?"* The answer to this question is given with a literature discussion, supported by a small experiment.

The different quantization schemes introduced in Section 2.3 are dynamic quantization, static post-training quantization, and static quantization-aware training. Dynamic quantization is not preferred for MCU inference because of its high computational overhead, even though the loss of accuracy because of quantization losses is the lowest with this method. That leaves both static quantization methods as options for this work.

Post-training quantization is a possible extension to the search space when a supernetwork is trained according to the one shot performance estimation strategy. The supernetwork search space has not been selected, in favour of a chain-structured search space. With this search space and the low-fidelity performance estimation, newly generated networks require to be trained. That makes it impossible to include a PTQ scheme into the NAS algorithm of this work. Quantization-aware training can be exploited within this architecture search, since it allows for the model to be trained after the evaluation. From the works in Section 3.2, QAT additionally appears to yield lower quantization losses for sub-byte integer datatypes.

The experiment results in Figure 5.2 confirm the findings from the literature. The sub-byte integer quantized network scores 0.1% lower on accuracy than the 8-bit quantized networks, but yields the lowest memory and storage requirements as a result. Between the 8-bit QAT and PTQ networks, the difference seems to be more on an architectural level than being caused by the quantization schemes. The PTQ network is from a different architecture search run than the QAT network. It is difficult to draw a proper conclusion from this, as more data is required for a proper comparison.

6.1.3 Usage of NAS for tinyML

"How are the accuracy and resource utilization of an image classification convolutional neural network that has been implemented in a microcontroller using NAS?" That is the question that will be answered in this section, with the results of the three experiments in Section 5.3.

Considering these results standalone, the NAS algorithm discovers networks with increasingly better accuracy over the generations for all of the tested datasets. Between different runs, there is a lot of deviation between the results initially, but the different runs tend to converge

over time. This is as expected, since the different runs have different start points and work towards achieving the best possible performance. At some point, an increase in storage requirement seems occur at the point where the accuracies of different runs start to diverge again. There seems to be a point where large networks start to perform well, though the exact position of this point and the eventual gains from the larger networks is unknown. Within the resource limits of the specific microcontroller used in this work, the STM32F446RE, it is possible to deploy a convolutional neural network performing at the level of other work.

Compared to NAS-generated networks from the other works in Section 3.3, it can be concluded that the achieved accuracy on the MNIST dataset is in line with or exceeding other NAS algorithms targeting microcontroller devices. For the CIFAR-10 and Fashion-MNIST networks, no comparable data can be found in the other works discussed in Section 3.3. For the more complex CIFAR-10 dataset, the NAS architecture in this work starts to meet its limits, with an accuracy of about 60%. Despite that, the network is more of a challenge to run on MCUs in general, with [37] performing an experiment on a binary version of CIFAR-10. They achieved a significantly higher accuracy of 86.39%, though it is unknown is how this would compare to regular CIFAR-10.

6.1.4 Sub-byte quantization performance

"How does a sub-byte quantized neural network compare to more conventionally quantized network inference in terms of operations and inference time?"

Although quantizing to datatypes smaller than the trivial 8-bits can cut the model size and peak memory usage in half or even lower, the lack of hardware support limits the performance gains. Since there is no support for these datatypes in common microcontroller instruction sets, data will have to be unpacked to single bytes, which introduces an overhead that has a significant impact on throughput.

Assuming that a STM32-based MCU would support sub-byte datatypes, the results look promising. The number of MACC operations between conventionally (8-bit) quantized networks and sub-byte quantzied networks should be the same, because they share the same network architecture. The reductions in storage and memory requirements are similar to software sub-byte execution, but the number of clock cycles required to compute the inference decreases exponentially. As a result, a 2-bit quantized network would require about half of the processor cycles that a 8-bit quantized network would with a quarter of the memory and storage footprint. The limiting factor here is the low speed of the Flash storage, which takes in the STM32F446RE 6 times as long as a single MACC operation. When the hardware will support it, aggressive quantization for microcontrollers could increase the speed at which CNNs are computed, consuming less power while doing so.

6.2 Recommendations

6.2.1 Microcontroller instruction sets

Like most processors, the instruction set architecture (ISA) of most microcontrollers is 32-bit. Usually, there is support for smaller datatypes, like 16 bits (halfwords) and bytes, but not smaller than that. Boolean values are for example are packed in a whole byte.

There are many instructions that accelerate digital signal processing (DSP) algorithms on board of modern microcontrollers. As neural networks are black-box filters on the input data, DSP instructions are applicable to network inference as well. Current ISA libraries for neural network inference, such as CMSIS-NN [27], use an optimization to calculate convolution with the im2col and GEMM algorithms.

A future research direction is to evaluate the performance benefits of microcontroller ISA targeted towards sub-byte quantized neural network inference. Work has been done on the creation of a mixed-precision version of the CMSIS-NN libraries [38].

The evaluation of neural network inference on a instruction set that supports SIMD instructions for sub-byte datatypes can open up some performance as the parallelism of computations could theoretically be doubled or even quadrupled. In this case, it would be interesting to see the performance improvements brought by the fact that data can be loaded and stored in a higher level of parallelism as well.

6.2.2 NAS performance

As mentioned in Section 6.1.3, the performance of the CIFAR-10 network generated by the NAS algorithm can be better. Besides that, the experiments in this work have been targeting networks with a low input resolution, which leads to memory and storage requirements that are too low to fully utilise the capabilities of higher-end microcontrollers.

A step beyond classification on CIFAR-10 would be to target more complex datasets with higher resolution inputs and more output categories, such as ImageNet. Neural networks computing classification algorithms on the ImageNet dataset have been transferred to tinyML platforms in [2] with post-training quantization to 8 bits. Since there are not many other works that have achieved this, it would be interesting to see whether more aggressive quantization within the NAS algorithm, as implemented in this work, does have an impact.

6.2.3 Pruning

Quantization of the network parameters is not the only method to reduce the resource utilisation of neural networks. The removal of nodes that have weights that are close to 0, a process which is called *pruning*, is another option. Pruning has even been implemented for some of the works in Section 3.3, such as [35, 37]. For the target MCU and datasets used in the experiments of this work, the usefulness of pruning is difficult to quantify, because the resource utilisation did not reach critical levels. It would interesting to see whether the effect of pruning is more noticeable on larger datasets, perhaps in combination with the previous recommendation given.

6.2.4 Memory management

As indicated by MCUNetV2 [42], the memory distribution of neural network inference is very imbalanced, with a high peak usage in the first layers and a lower usage in later networks. A better memory distribution over the inference process would not only make more efficient use of the resources of the target device, it could also allow for layers or inputs that are larger than the memory allows to be processed. This direction for future work does have more to

do with the inference runtime environment than with the NAS algorithm itself, although the method in which the memory usage is calculated would have to be revisited.

Bibliography

[1] J. K. Gill, "What's the difference between cloud, edge, and fog computing?." https://www.akira.ai/blog/difference-between-cloud-edge-and-fog-computing/, Jan. 2021. Accessed: 2021-07-14. v, 1

[2] J. Lin, W.-M. Chen, J. Cohn, C. Gan, and S. Han, "Mcunet: Tiny deep learning on iot devices," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020. v, 2, 23, 50

[3] STMicroelectronics, "Biometric system-on-card." https://www.st.com/content/ccc/resource/sales_and_marketing/promotional_material/flyer/group0/57/d2/14/df/73/9f/4e/80/Flyer-Biometric-System-on-Card/files/flbiometricsoc1220_mr.pdf/jcr:content/translations/en.flbiometricsoc1220_mr.pdf, Dec. 2020. Accessed: 2021-03-26. v, 2, 3

[4] MathWorks, "What is a convolutional neural network?." https://nl.mathworks.com/discovery/convolutional-neural-network-matlab.html, 2022. Accessed: 2022-01-21. v, 7, 8

[5] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019. v, 9, 15, 21

[6] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106622, 2021. v, 10, 13

[7] Z. Qin, Z. Zhang, D. Li, Y. Zhang, and Y. Peng, "Diagonalwise refactorization: An efficient training method for depthwise convolutions," in *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2018. v, 20

[8] Raspberry Pi Foundation, "Raspberry pi 4 tech specs." https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/, 2022. Accessed: 09-03-2022. vii, 18

[9] Raspberry Pi, *RP2040 Datasheet*. Raspberry Pi Trading Ltd, https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf, Nov. 2021. vii, 18

[10] STMicroelectronics, "X-cube-ai - ai expansion pack for stm32cubemx." https://www.st.com/en/embedded-software/x-cube-ai.html, 2021. Accessed: 2021-06-17. vii, 18

[11] STMicroelectronics, *STM32F745xx STM32F746xx*. STMicroelectronics, https://www.st.com/resource/en/datasheet/STM32F746NG.pdf, Feb. 2016. vii, 18

[12] STMicroelectronics, "Arm 32-bit microcontrollers." https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers.html, 2022. Accessed: 11-03-2022. vii, 18

[13] tinyML Foundation, "About tinyml foundation." https://www.tinyml.org/about/, 2021. Accessed: 2021-07-14. 2

[14] S. Higginbotham, "Privacy and new functions will make tinyml big." https://mailchi.mp/iotpodcast/stacey-on-iot-tinyml-gets-big, 2021. Accessed: 2021-07-14. 2

[15] G. Anadiotis, "Machine learning at the edge: Tinyml is getting big." https://www.zdnet.com/article/machine-learning-at-the-edge-tinyml-is-getting-big/, July 2021. Accessed: 2021-07-14. 2

[16] AutoML, "Automl." https://www.automl.org/automl, 2021. Accessed: 2021-05-05. 3

[17] S. Walczak and N. Cerpa, "Artificial neural networks," pp. 631–645, 2003. 6

[18] K. Siu, D. M. Stuart, M. Mahmoud, and A. Moshovos, "Memory requirements for convolutional neural network hardware accelerators," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 111–121, 2018. 8

[19] A. Krizhevsky, "The cifar-10 dataset." https://www.cs.toronto.edu/~kriz/cifar.html, 2009. 9, 24

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009. 9

[21] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via Lamarckian evolution," in *International Conference on Learning Representations*, 2019. 9, 10, 21, 23

[22] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4780–4789, Jul. 2019. 10, 21, 29

[23] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018. 16, 22

[24] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *CoRR*, vol. abs/1806.08342, 2018. 16, 22, 28, 30

[25] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *CoRR*, vol. abs/2103.13630, 2021. 16

[26] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L. Wang, Q. Huang, Y. Wang, M. Mahoney, and K. Keutzer, "Hawq-v3: Dyadic neural network quantization," in *Proceedings of the 38th International Conference on Machine Learning* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 11875–11886, PMLR, 18–24 Jul 2021. 16

[27] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: efficient neural network kernels for arm cortex-m cpus," *CoRR*, vol. abs/1801.06601, 2018. 19, 23, 50

[28] H. Wang and C. Ma, "An optimization of im2col, an important method of CNNs, based on continuous address access," in *2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE)*, pp. 314–320, 2021. 19

[29] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 19–24, 2017. 19

[30] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 2018. 21, 22

[31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015. 21

[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations*, 2015. 21

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. 21

[34] M. Rusci, A. Capotondi, and L. Benini, "Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers," in *Proceedings of Machine Learning and Systems* (I. Dhillon, D. Papailiopoulos, and V. Sze, eds.), vol. 2, pp. 326–335, 2020. 22

[35] I. Fedorov, R. P. Adams, M. Mattina, and P. Whatmough, "Sparse: Sparse architecture search for CNNs on resource-constrained microcontrollers," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019. 22, 23, 50

[36] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough, "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," in *Proceedings of Machine Learning and Systems* (A. Smola, A. Dimakis, and I. Stoica, eds.), vol. 3, pp. 517–532, 2021. 22

[37] E. Liberis, L. Dudziak, and N. D. Lane, *NAS: Constrained Neural Architecture Search for Microcontrollers*, p. 70–79. New York, NY, USA: Association for Computing Machinery, 2021. 23, 49, 50

[38] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "Cmix-nn: Mixed low-precision CNN library for memory-constrained edge devices," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 871–875, 2020. 23, 50

[39] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist databaseof handwritten digits." `http://yann.lecun.com/exdb/mnist/`, 1998. 24, 32

[40] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms." https://github.com/zalandoresearch/fashion-mnist, 2017. 24, 32

[41] STMicroelectronics, *STM32F446xC/E*. STMicroelectronics, https://www.st.com/resource/en/datasheet/stm32f446re.pdf, Jan. 2021. 26

[42] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Mcunetv2: Memory-efficient patch-based inference for tiny deep learning," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021. 26, 50

[43] P. Ren, Y. Xiao, X. Chang, P. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *CoRR*, vol. abs/2006.02903, 2020. 28

[44] A. Pappalardo, "Xilinx/brevitas." https://github.com/Xilinx/brevitas. 30