UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science
Cyber-Physical Systems Research Group
MINDS Centre for Doctoral Training

Efficient Deep Learning Inference at the Edge

DOI: 10.5258/SOTON/D3550

by

Sulaiman Sadiq
BE, National University of Sciences and Technology (2011)
MSc, University of Engineering and Technology (2016)
MSc, University of Southampton (2018)
ORCiD: 0000-0003-4959-4623

A thesis for the degree of
Doctor of Philosophy

June 2025

University of Southampton

Abstract

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Doctor of Philosophy

Efficient Deep Learning Inference at the Edge

by Sulaiman Sadiq

Deep Learning has found success in a variety of fields. At the same time, the number of connected Internet of Things (IoT) devices is seeing exponential growth. This has lead to the development of the field of TinyML which aims to perform deep learning inference locally on the resource constrained IoT devices. Realising this objective requires optimisation across the inference stack with design of efficient algorithms and inference systems. In the work carried out in this thesis, we explore techniques for efficient model design and deployment for various complexity constraints that are imposed by the resource constraints of the devices or the application scenarios.

The first contribution of this work is a gradient based approach to derive models of varying complexity using neural architecture search (NAS). This is achieved by combining multi-objective optimisation with NAS where we optimise the complexity of the model in addition to its quality. This method derives models of varying complexity without the need for manual heuristics or expensive hit and trial.

The second contribution of our work studies how inference software can effectively utilise device resources to enable design and deployment of efficient models. Whereas prior works typically focus on performing inference within internal memory constraints, we develop the TinyOps inference framework for MCUs which accelerates inference from external memories. TinyOps significantly lifts the ceiling of accuracy achievable with up to 1.4x-2.5x lower inference latency than previous approaches.

The final contribution of this work is an in-depth analysis of DNN deployment on MCUs to derive heuristics for model design and how inference can be adapted at run-time on MCUs for varying latency constraints. We study limitations of existing approaches and benchmark throughput of low-level operations on MCUs. Using our heuristics, we derive models which achieve state-of-the-art TinyML ImageNet classification when considering accuracy, latency and energy efficiency. The heuristics are also utilised in a super-network based approach to derive multiple models for different latency constraints. We show how an efficient accuracy-latency trade-off can be achieved at run-time with the TinyOps inference framework.

Contents

List of Figures

List of Tables

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. None of this work has been published before submission

Signed:..... Date:.....

To my parents,

Definitions and Abbreviations

API	Application Programming Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DMA	Direct Memory Access
DNN	Deep Neural Network
DSP	Digital Signal Processor
DyNN	Dynamic Neural Network
EEC	Early Exit Classifier
EENN	Early Exiting Neural Network
FC	Fully Connected
FMC	Flexible Memory Controller
GB	Gigabyte
GPU	Graphical Processing Unit
ISR	Interrupt Service Routine
IoT	Internet of Things
KB	Kilobyte
LSTM	Long Short Term Memory
MAC	Multiply and Accumulate
MB	Megabyte
MCU	Microcontroller
NAS	Neural Architecture Search
NN	Neural Network
NPU	Neural Processing Unit
PB	Petabyte
QSPI	Quad Serial Port Interface
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SDRAM	Synchronous Direct Random Access Memory
SIMD	Single Instruction Multiple Data
SRAM	Static Random Access Memory
TFLM	TensorflowLite-Micro
TPU	Tensor Processing Unit

Chapter 1

Introduction

Deep Neural Networks (DNNs) are a class of machine learning models that have recently found success in a variety of applications where there is an abundance of labelled data. The first major example of this can be found with the convolutional neural network (CNN), AlexNet (Krizhevsky et al., 2012), which achieved 10% higher accuracy than the second best entry in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) (Russakovsky et al., 2015). In part, this success was due to the usage of Graphical Processing Units, (GPUs), whose highly parallel architecture was well suited for efficient implementation of the Neural Network (NN) workload, composed of large linear algebraic operations (Chetlur et al., 2014; Jorda et al., 2019). The significant gains in performance achieved in ILSVRC 2012 initiated tremendous research effort into development of DNNs for a number of tasks. Since then, a significant amount of progress has been made and a variety of architectures including CNNs (He et al., 2016), AutoEncoders (Bank et al., 2023), LSTMs (Greff et al., 2016), RNNs (Salehinejad et al., 2017) and Transformers (Lin et al., 2022) have been proposed for a number of different tasks including image classification (Deng et al., 2009), natural language understanding (Hinton et al., 2012), machine translation (Vaswani et al., 2017), and gameplay (Silver et al., 2016; Vinyals et al., 2019) amongst others.

A trend that can be observed over the years, is that to achieve better performance across tasks these models have constantly been increasing in size and computational complexity (Villalobos et al., 2022). To keep up with the computational demands of these workloads, increasingly powerful GPUs have been developed over the past decade which offer an order of magnitude higher core count and memory size (Jeon et al., 2021; Dally et al., 2021), in addition to the development of deep learning accelerators such as Tensor Processing Units (TPUs) and Neural Processing Units (NPUs) (Silvano et al., 2023).

At the same time, with the proliferation of mobile phones, smart wearables and other sensor devices, low power Internet of Things (IoT) devices have been increasingly pervading society with recent forecasts estimating that there will be 41.6 billion connected

IoT devices by 2025 (Insights, 2022). Coupled with the success of DNNs there has been tremendous interest, in application of DNNs in areas such as smart cities (Zanella et al., 2014), health monitoring (Zazzaro et al., 2021; Gibbs et al., 2023) and industrial sensing (Khalil et al., 2021). However, the resource constrained IoT devices (Lin et al., 2020; Liberis et al., 2021; Banbury et al., 2021; Fedorov et al., 2022) have significantly different architectures and characteristics compared to traditional GPUs which DNN workloads are typically designed for. Whereas GPUs have GBs of memory, TBs or PBs of storage and thousands of cores for parallel processing, IoT devices which typically host MCUs have memory and storage in the MB range and a singular core for data processing (Lin et al., 2020). Similarly, these devices are often battery powered and deployed in the wild where they must adapt to the environment whilst adhering to real-time latency and energy constraints (Hong et al., 2020; Gog et al., 2022).

These resource constraints make it challenging to run compute and memory intensive DNN workloads onboard the edge device. Due to this, applications of DNNs in the IoT typically rely on the cloud computing paradigm (Samie et al., 2019) which utilises the low power IoT device itself for data collection and preprocessing with the major number crunching and processing being performed on the cloud providing GPU services. However, this is undesirable for a number of reasons including privacy, security, latency and cost associated with cloud based solutions (Premsankar et al., 2018). These limitations of cloud based solutions have lead to the development of the field of TinyML which investigates how data can be processed locally within the resource constraints of the MCU based edge devices. TinyML encompasses a vast array of specialist research looking to optimise the design and deployment of DNNs across a number of dimensions to optimise the accuracy, latency and energy efficiency within the resource constraints of the deployment platforms.

1.1 Research Justification

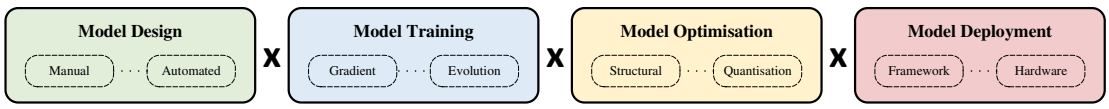


FIGURE 1.1: Inference Components for Deep Neural Networks (DNN)

The research conducted in the field of TinyML or more broadly efficient machine learning can be approximately categorised according to the inference components shown in Figure 1.1. While each of these components can be individually optimised for efficient inference, quite often these components are co-designed with each other. For example, models are often trained with specialised pruning or quantisation-aware training recipes which enables effective model optimisation strategies. In certain cases, such as where there is limited support for low-precision data types or sparse computation,

components can also be removed from the inference stack. We note that the components in inference do not have to be applied in a linear fashion, quite often iterative training methods are used where a feedback loop is established between the optimisation strategy and the training algorithm. Similar loops are often also established between the model deployment and design components. One of the fundamental components in inference is designing the model for a specific task, such as a CNN for an image processing task. Within model design there are two major approaches adopted. The traditional approach adopted for the default GPU platforms was to manually design models by selecting the hyper-parameters of the network such as depth, layer types, kernel sizes, filter sizes, etc to achieve high accuracy while meeting computational or memory constraints (Krizhevsky et al., 2012; Iandola et al., 2017; Sandler et al., 2018; Howard et al., 2017). These works rely on different heuristics for design of models and explore the trade-off between different dimensions of the model such as accuracy, latency, complexity and memory footprint. However, the applicability of these heuristics can vary depending on the underlying platform which often make a model designed for one platform unsuitable for deployment on another (Cai et al., 2020; Dudziak et al., 2020). As edge devices have such different characteristics, there is a need for study of the devices to derive efficient heuristics for the model design process.

A limitation of the manual approach to model design is that it is an involved, time consuming and costly process due to hit and trial involved in developing and trying different heuristics. Additionally, it requires expert human knowledge to apply these heuristics in the design process which is not always available. To alleviate these limitations, Neural Architecture Search (NAS) was proposed to automate the design of DNNs without human intervention in the process. These works (Zoph and Le, 2017; Liu et al., 2018; Real et al., 2019; Tan et al., 2019) utilise reinforcement learning, evolutionary optimisation or gradient based search to derive architectures from a given search space. However, these works focus on deriving the architectures with the highest performing accuracy from the search space. To meet complexity requirements, these models use manual heuristics to simplify a model derived from the search space (Cai et al., 2018; Tan et al., 2019) or simplify the entire search space (Wu et al., 2019). For battery powered edge devices, an accuracy-complexity trade-off is often required as these devices must be able to adapt to changing latency, compute and energy constraints (Xu et al., 2019; Hong et al., 2020; Navardi and Mohsenin, 2023). For such scenarios, it would be desirable to have efficient methods that can explore the accuracy and complexity trade-off in an automated manner. Such approaches would also benefit by using heuristics used in manual model design to inform search space design.

The model deployment scenario is closely related to the model design as the actual performance achieved by any given model depends on the characteristics of target hardware platforms and the ability of the inference framework to leverage the compute capability of the hardware. The field of TinyML typically considers platforms with less

than $1W$ power consumption due to the deployment scenarios where the battery operated devices are energy constrained. DSPs and MCUs are good candidate platforms in the field due to their energy efficiency and instruction sets which support multiply, add or multiply accumulate operations frequently encountered in DNN workloads.

MLOps tools play a crucial role in development and deployment of large models in TinyML. These tools enable efficient management of different stages of the machine learning (ML) life-cycle including training, optimisation and inference in a collaborative, scalable and compliant manner. During training, they handle data versioning, experiment tracking, distributed resource orchestration, and monitoring, allowing teams to efficiently train massive models across multiple GPUs or nodes. For inference, MLOps platforms enable seamless and optimised model deployment with efficient inference frameworks, and monitor production models for drift or performance degradation. By integrating these capabilities into cohesive pipelines, MLOps reduces manual overhead, minimizes errors, and accelerates the end-to-end machine learning lifecycle. In the field of TinyML, the inference framework provides a light-weight run-time environment suitable for resource constrained edge devices that is typically responsible for loading a model, memory management, interpreting the inference graph, and low-level kernels which perform primitive operations (e.g. conv, matmul, add) by interfacing with the hardware through the instruction set. As such, it is important to optimise the performance of the inference framework to enable high throughput inference. Prior work to optimise inference in deployment has focused on generating light-weight code (Lin et al., 2020, 2021, 2022), developing efficient kernels (Lai et al., 2018; Rusci et al., 2018, 2020a) and graph optimisations (Liberis and Lane, 2020, 2022) to lower the memory footprint and latency of deployment. These approaches allow a larger model to be deployed within a smaller memory footprint to achieve higher accuracy and lower latency. While these approaches have been successful to varying degrees, a limitation of these works is that they assume a simplified architecture of the devices which consists of a CPU Core coupled with internal storage and memory. This assumption limits the size of the model design space and the achievable performance. On edge devices, where resources are already limited, it would be beneficial to develop inference frameworks that efficiently utilise the available resources such as external memory interfaces to extract the maximum performance from the devices.

1.2 Research Questions

Following the research justifications, in the work carried out in this research we look at how efficient inference across the accuracy complexity pareto frontier can be performed on edge devices. To study this we look at optimisations across the inference stack to enable efficient model design and deployment. The research addresses four main research questions

RQ1. How can we derive models of varying complexity using automated methods according to varying requirements?
Several approaches have been proposed to derive models in an automated manner using the framework of Neural Architecture Search (NAS). However, prior works focused on deriving models with the highest accuracy and using manual heuristics to modify the models to meet complexity requirements. This research question aims to investigate how we can derive models across the accuracy-complexity pareto frontier directly from the search space without any manual intervention in the process. This question is addressed in Chapter 3 and Chapter 5.

RQ2. How can inference software efficiently utilise external memories to enable design and deployment of efficient models? The majority of prior works have focused on designing models within the constraints of internal storage and memory on the device which limits the achievable performance. In this question we investigate how external memory interfaces available on the platform can be effectively utilised by inference frameworks to expand the design space of efficient DNNs on MCUs and achieve better performance. This question is addressed in Chapter 4.

RQ3. What model design heuristics can we draw from study and analysis of DNN deployment on edge devices at the micro-architecture level to inform efficient model and search space design? The design of models using manual or automated search involves using heuristics to design either the search space for automated methods or the model definition itself for manual methods. While prior works have studied the computational complexity of DNNs at a model level granularity or for different hardware platforms, this research question focuses on performing an in-depth analysis of DNN deployment on MCUs at the micro-architecture level. This is addressed in Chapter 5.

RQ4. How can we adapt inference at run-time to efficiently trade-off between accuracy and latency on edge devices? Edge devices have to adhere to real-time latency constraints in order to meet deadlines imposed by the application or environment. In the research question, we study how an application can efficiently trade-off the accuracy of the model with latency at run-time to meet latency constraints varying in real-time. This research question is addressed in Chapter 5.

1.3 Research Contributions

In addressing the research questions the major contributions made in this thesis are visualised in Figure 1.2. The contributions listed below make optimisations across the

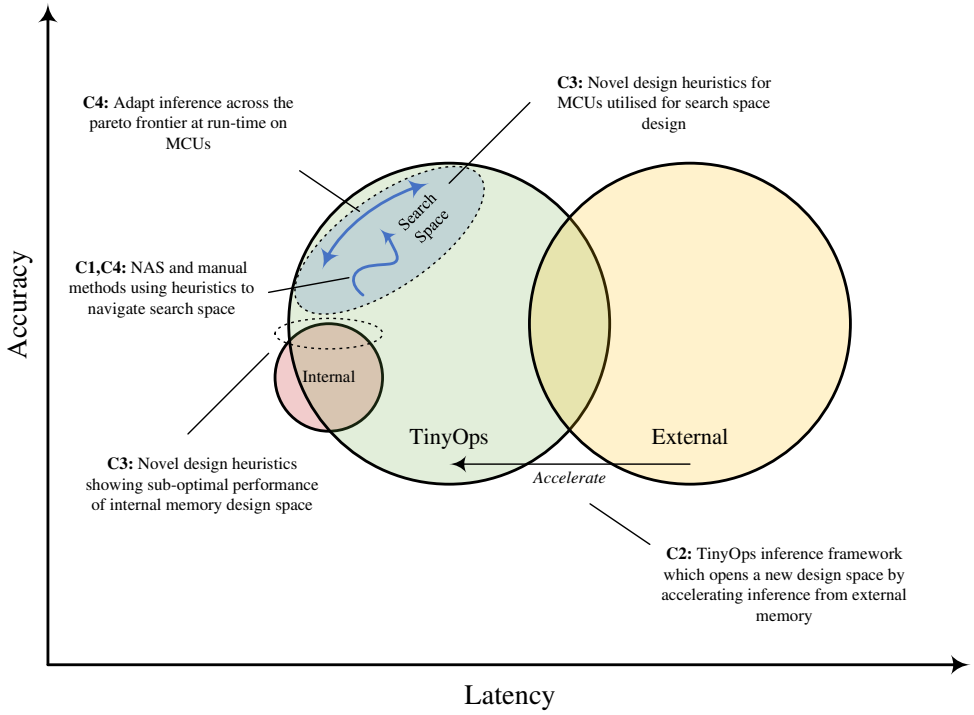


FIGURE 1.2: Overview of Research Contributions

inference stack including design of efficient models and inference software to achieve high performance on the accuracy-latency pareto frontier.

C1. An efficient gradient based search algorithm to automatically derive models of varying complexity.
While prior works utilised automated neural architecture search to derive architectures with high accuracy, we reframe the problem as a multi-objective optimisation problem to trade-off the accuracy and complexity. We formulated a closed form differentiable expression for the computational complexity of candidate models in the search space and successfully navigate the search space via gradient descent to derive models of varying complexity where the trade-off was controlled by a single hyper-parameter. This work directly addresses **RQ1** in Chapter 3 and was published in the Neural Information Processing and Systems (NeurIPS) Machine Learning for Systems workshop (Sadiq et al., 2020).

C2. The TinyOps inference framework that effectively utilises external memories on MCUs to open a new efficient design space for MCUs and speed up inference of large models in external memory by up to 2.5x to set the state of the art in TinyML ImageNet classification.
We study MCU architectures and analysed their capabilities in addition to how

current inference frameworks deploy DNNs onto MCUs. We highlight the limitations of using external memories with current inference frameworks and propose a partitioning and DMA based overlaying framework for MCU devices that combines advantages of external memory (size) and internal memory (speed) to open a new design space for MCUs which achieves state of the art ImageNet classification accuracy in TinyML. This work is reported in Chapter 4 and addresses **RQ2**. Initial findings were published at the Computer Vision and Pattern Recognition (CVPR), Efficient Computer Vision (ECV) workshop (Sadiq et al., 2022), and with incremental improvements in the IEEE Internet of Things Journal (IoTJ) (Sadiq et al., 2023). Additionally, this work was granted a US Patent (Sadiq et al., 2024) filed in collaboration with the industrial partner for this project, ARM.

C3. We derive novel design heuristics which are utilised to design efficient CNNs across the accuracy-latency pareto frontier on MCUs which achieve state-of-the-art ImageNet performance.
We performed an in-depth analysis of DNN deployment on MCUs at the micro-architecture level to derive heuristics and insights into model design for MCUs. We show that the internal memory design space used by prior works yields sub-optimal accuracy and latency. Further, we study the structure of state of the art models and throughput of low-level candidate operations (Conv, DepConv) to motivate model design for MCUs. Using insights from our latency analysis, we use efficient operations to derive models across the accuracy-latency pareto frontier to set the state-of-the-art in ImageNet classification. This work discussed in Chapter 5, addresses **RQ3** with contributions published in IEEE IoTJ (Sadiq et al., 2023).

C4. We propose a supernetwork based neural architecture search method to derive models of varying complexity and an efficient framework to deploy and switch between the models based on latency constraints in real-time with zero switching overhead.
We propose a framework for how an accuracy-latency trade-off can be realised on MCUs for applications with real-time and varying latency constraints. We propose a light-weight super-network based NAS architecture with the search space motivated by our analysis in Chapter 5. We derive multiple models for a number of latency constraints and show that accuracy-latency trade-off can be achieved with zero-switching time at run-time using the TinyOps inference framework developed in Chapter 4. This work addresses **RQ4** and is reported in Chapter 5.

1.4 Research Output

The research described in this thesis appears in the following publications

- Sulaiman Sadiq, Partha Maji, Jonathan Hare, and Geoff Merrett. DEff-Arts: Differentiable Efficient Architecture Search. In *Neural Information Processing Systems Workshop on Machine Learning for Systems, 2020*

- Sulaiman Sadiq, Jonathon Hare, Partha Maji, Simon Craske, and Geoff Merrett. Tinyops: Imagenet scale deep learning on microcontrollers. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshop on Efficient Deep Learning for Computer Vision, 2022*

- Sulaiman Sadiq, Jonathon Hare, Simon Craske, Partha Maji, and Geoff Merrett. Enabling ImageNet-scale deep learning on MCUs for accurate and efficient inference. In *IEEE Internet of Things Journal, 2023*

Part of the research performed in this thesis was also granted a US patent, which was filed in collaboration with the industrial partner for this research, ARM Ltd.

- Sulaiman Sadiq, Jonathon Hare, Geoff Merrett, Partha Maji and Simon Craske. Neural Network Memory Configuration. *US Patent App. 17/813,396, 2024*

1.5 Thesis Outline

An overview of the research carried out in this thesis is shown in Figure 1.3. As shown, we make a number of contributions to the components of the inference stack involving efficient model design and deployment. The diagram also indicates which inference component each of the research questions (RQ1-4) and corresponding contributions are related to. A mapping of the contributions to each of the chapters is also shown. The remainder of this thesis is organised as follows

Chapter 2 begins by providing relevant background on design and deployment of CNNs. This includes model design through manual and automated methods, model optimisation techniques in addition to an overview of the hardware and inference frameworks used in deployment of the models. The limitations of the approaches adopted are also discussed.

Chapter 3 focused on efficient model design using automated methods and explores how we can derive models of varying complexity using gradient based NAS. We desribe DEff-ARTS, which utilises the machinery of DARTS (Liu et al., 2019) discussed in Section 2.2.1.3, to derive architectures of varying complexity. We describe our formulation of the problem as a multi-objective optimisation problem and how we combine the sub-objectives to derive models in an automated manner. This is followed by experimental evaluation of the proposed NAS algorithm.

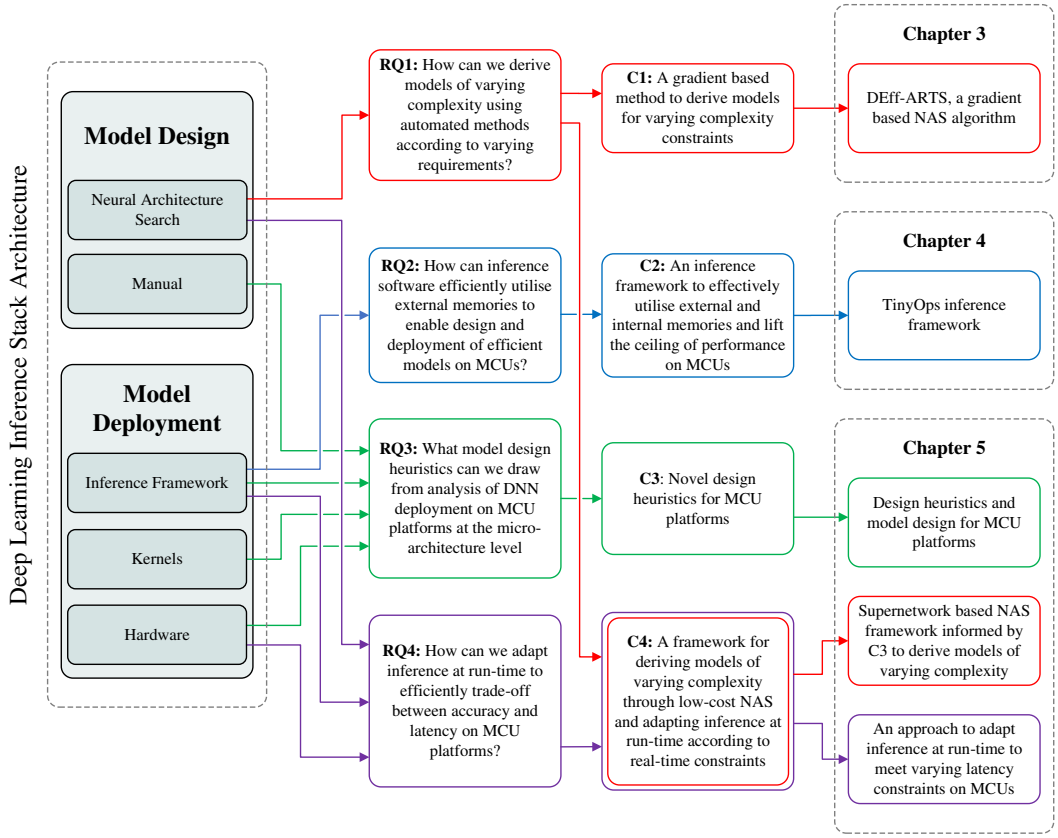


FIGURE 1.3: Overview of Thesis Structure with RQ1-4 representing research questions and C1-4 representing contributions of the research

Chapter 4 presents the TinyOps framework developed to enable efficient model deployment on MCU platforms. We discuss the limitations of prior inference frameworks which do not effectively utilise the memory hierarchy and how this limits the performance achievable on MCU platforms. We describe how TinyOps employs a partitioning and overlaying scheme to combine the advantages of size and speed of external and internal memory. This is followed by experimental results to demonstrate how TinyOps lifts the ceiling of performance achievable on MCU platforms.

Chapter 5 describes our in-depth analysis of DNN deployment on MCU platforms at the micro-architecture level and derives heuristics to design efficient models for varying latency constraints. We show that operation agnostic approaches that focus on designing models within size constraints of fast internal memory suffer from low accuracy and high inference latency. We utilise the derived heuristics to propose a lightweight NAS approach to derive models of varying complexity. We further show how the inference can be adapted at run-time to meet varying real-time latency constraints. Finally, we perform experimental results to demonstrate the performance gained by using the design heuristics to derive models.

Finally, Chapter 6 concludes the thesis by summarising the contributions made by the work carried out in this thesis. Further, we discuss the open research challenges in the field and future avenues of work that can be explored for further performance gains on resource constrained devices.

Chapter 2

Background and Related Work

This chapter provides background and a review of current literature relevant to the work carried out in this thesis. In Section 2.1, we begin by discussing the Convolutional Neural Network (CNN) architecture and its building blocks, used extensively in deep learning for image processing and throughout this thesis. This is followed by prior work into efficient model design including static and dynamic neural networks derived through manual or automated methods in Section 5.1. In this section, we delve into deeper detail of the DARTS (Liu et al., 2018) algorithm, as we utilise this approach in Chapter 3. Next, in Section 2.3, we discuss approaches that are utilised in optimising a given model for deployment including pruning and quantisation approaches. We provide deeper detail for linear quantisation schemes as these are used on MCUs and are utilised in Chapter 4 and Chapter 5. Finally, in Section 2.4, we discuss the characteristics of the hardware considered in this thesis, in addition to the inference frameworks and kernels used for the hardware platforms.

2.1 Convolutional Neural Networks

A number of different DNN architectures have been proposed for a variety of tasks. In this thesis, we focus on CNN architectures which are widely used for image based tasks including image classification, object detection and segmentation. A multi-layered CNN model sequentially performs operations on an input to produce an output representation. These operations are typically parametrised by some weights that can be learned by training the neural network. The operations that are typically used in constructing a neural network include operations such as convolutions, depthwise-separable convolutions, fully connected layers and activation functions. In a multi-layered model, these operations are also referred to as layers of the model.

2.1.1 Convolutional Layers

One of the main building blocks of CNNs is the convolution layer. These layers are applied to an input representation of size $N \times H \times W \times C_{in}$, where N is the batch size, and H, W and C are the height, width and number of channels of the representation. To apply a convolution layer to an intermediate representation, a filter of size $K \times K$ with C_{in} input channels is convolved with the input representation as shown in Figure 2.1 to produce one output channel of the output representation. To produce an output representation with C_{out} channels, C_{out} filters are applied to the input representation. The resulting output representations of convolving C_{out} filters with the input representation are then concatenated together to produce the complete output representation with dimensions $N \times H \times W \times C_{out}$.

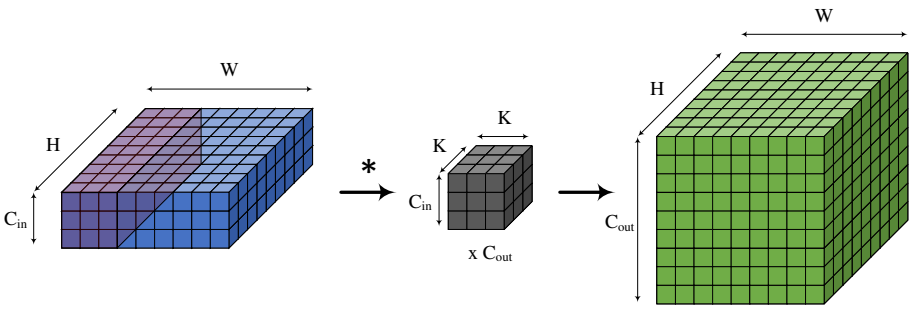


FIGURE 2.1: Convolution Layer

Convolutions are compute intensive operations where the computation can be calculated as below $(K^2) \times C_{in} \times H_{out} \times W_{out} \times C_{out}$. Similarly, the parameters in a convolution layer can be noted as $(K^2) \times C_{in} \times C_{out}$.

2.1.2 Depthwise Separable Convolutions

Depth-wise separable convolutions were proposed as a light-weight alternative to convolution operations. Depth-wise separable convolutions are applied in two stages where the first stage is a grouped convolution and the second part is a pointwise convolution.

Grouped and Depthwise Convolution In a grouped convolution the channels of a filter and input representation are separated into multiple groups where each group is performed as a regular convolution with the outputs being concatenated together in the channel dimension. A common approach employed is to have as many groups as there are channels in the filter or input representation and set the channels in the output equal to the number of groups. In this specific case, referred to as depthwise convolutions, each of the C_{in} channels of the input representation is independently

convolved with a $K \times K$ kernel with the resulting outputs being concatenated together to produce an output with $C_{out} = C_{in}$ channels as shown in Figure 2.2

The computation performed by a depthwise convolution can be computed as $K^2 \times H_{out} \times W_{out} \times C_{in}$ with the parameters equal to $K^2 \times C_{in}$.

Pointwise Convolution A pointwise convolution is then applied to project the representation to the number of output channels, C_{out}. This is performed as a regular convolution with C_{out} filters which have unit kernel size. In this stage the computation and number of parameters can be calculated as $H \times W \times C_{in} \times C_{out}$ and $C_{in} \times C_{out}$ respectively. The application of a depth-wise separable convolution consisting of a grouped convolution followed by a pointwise convolution is shown in Figure 2.2.

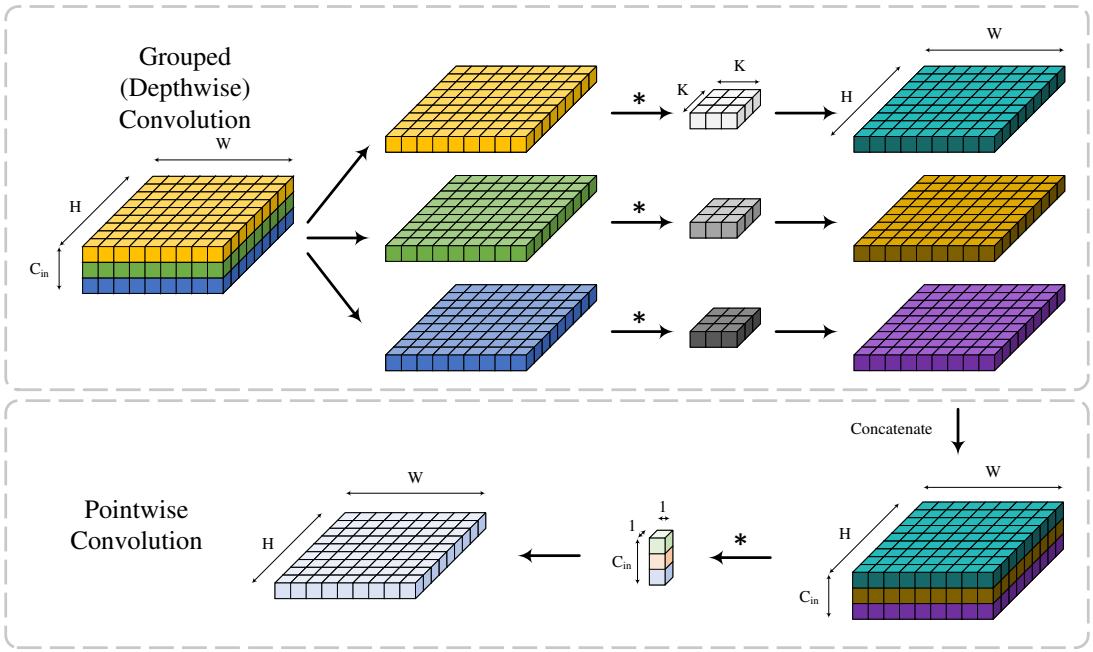


FIGURE 2.2: Depthwise Separable Convolution. Adapted from Pandey (2024).

In total the computation for a depth-wise separable convolution comes out to be $K^2 \times H_{out} \times W_{out} \times C_{in} + H \times W \times C_{in} \times C_{out}$ with the parameters equal to $K^2 \times C_{in} + C_{in} \times C_{out}$. Depthwise separable convolution are more parameter and compute efficient compared to vanilla convolutions.

2.2 Model Design

A number of approaches have been adopted to design efficient CNNs. These approaches aim to optimise the *quality* and *efficiency* of a model. The quality of a model would be indicated by the predictive performance of the model and measured with

metrics such as the accuracy, precision, recall. The efficiency would include metrics such as the parameter count, memory usage, computational complexity or latency. For any application, a practitioner would want the quality to be as high as possible for the best predictions. On the other hand, deployment on edge devices requires models to be designed within certain constraints of the hardware or application such as the memory or latency which requires increasing the efficiency of the model.

In practice, these two objectives to be optimised in CNN design are often conflicting and optimising one usually results in degradation of the other. As such, the objective of model design becomes to find a model that is pareto optimal when considering our quality and efficiency metrics. For applications that need to meet varying complexity constraints, the pareto frontier can be traversed using static or dynamic approaches.

2.2.1 Static Neural Networks

Static Neural Networks have a fixed structure and weights during inference and would represent a single point on the pareto frontier. Meeting multiple latency constraints in this manner requires designing multiple models according to requirements.

2.2.1.1 Manual Design

The traditional approach to developing efficient models relied on using manual heuristics to reduce the size and computational complexity. Szegedy et al. (2015) modified the number of filters, kernel sizes and depth via hit and trial to reduce the latency of inference while retaining accuracy. Iandola et al. (2017) developed CNNs which extensively utilised 1×1 convolutions to reduce the parameter count of CNNs. Similarly, group-wise convolutions and depth-wise separable convolutions have been utilised in the design of CNNs to reduce the parameters and computational complexity (Zhang et al., 2018; Sandler et al., 2018; Howard et al., 2019). Howard et al. (2017), further proposed using two global hyper-parameters, a width and resolution multiplier to reduce the input image size and channels in convolution layers for a reduction in model size and complexity. Tan and Le (2019) added depth as an extra hyper-parameter and proposed a compound scaling rule to uniformly scale width, resolution and depth according to resource budgets. We note that these approaches often develop heuristics in a platform agnostic manner, or for more popular platforms such as mobiles or GPUs which would not necessarily carry over to MCUs. A limitation with manually designing models is the requirement of human-expert knowledge in selecting the various hyper-parameters of the model. Additionally, the hit and trial involved is a tedious process that requires significant compute resources.

2.2.1.2 Neural Architecture Search

Neural Architecture Search (NAS) has emerged as a promising direction to bypass the challenges of manually designing models. NAS approaches remove the human out of the loop and automate the hit and trial process of sampling, evaluating and deriving architectures. A NAS algorithm is typically built from three main components which are the search space, the objective to maximise and the search method. Combining the three, models are derived by navigating the search space with a given method and finding the model that maximises the objective.

Search Space: The search space is defined to contain all the candidate architectures that could be evaluated during the search process. The size of the search space is defined by the degrees of freedom that are available in the model structure being derived. This would include the type of layer, hyper-parameters (e.g. channels, kernel size, stride) and the connectivity between layers.

In an extreme case, the most flexible search space could be represented as an arbitrary number of nodes in a graph which were freely assigned any layer type and connectivity. However, this would yield an extremely large search space that would be difficult to navigate.

As such, the majority of works constrain the search space using heuristics derived from manual design of models. A common method adopted is for the search space to resemble manually derived models (Cai et al., 2018, 2020; Banbury et al., 2021). In these cases, the search space can be seen as mutations to the base model architecture, which can modify the channel numbers, or kernel sizes but not the layer types. We note that the heuristics utilised in designing the search space are typically derived in a platform agnostic manner or for mobile or GPU platforms (Howard et al., 2019; Lin et al., 2020; Banbury et al., 2021).

Another commonly adopted approach is to search for the architecture of smaller blocks or *cells* (Shu et al., 2020; Ying et al., 2019; Mellor et al., 2021). This reduces the size of the search space to make the process tractable. After a cell has been derived, it is then stacked a number of times to produce a final model.

Search Method: The hit and trial approach of manually designing models can be thought of as manually traversing the search space guided by heuristics applied from human knowledge of CNNs. This process is automated in NAS algorithms as an optimisation problem where the architecture is derived from the search space by maximising an objective.

Gradient based optimisation is a commonly used approach (Liu et al., 2018; Cai et al., 2018; Xie et al., 2019; Xu et al., 2020) which employs a differentiable relaxation over the categorical selection of candidate operations to jointly learn the architecture and its weights.

Reinforcement Learning has also been used where a controller is trained to emit an architecture configuration using the objective as a reward signal (Zoph and Le, 2017; Zoph et al., 2018; Pham et al., 2018).

Similarly, evolutionary optimisation has been adopted where the architecture is represented as a genotype in a population (Real et al., 2017, 2019; Sun et al., 2019). During the search, each candidate in the population is trained and evaluated against the fitness criteria after which the next generation of candidates can be produced using a selection strategy such as tournament selection (Blickle, 2000).

Objective: The earlier approaches in NAS typically only optimised the quality metric of accuracy or the differentiable cross-entropy loss (Liu et al., 2018; Zoph et al., 2018; Real et al., 2019) in the search. While these approaches, were able to derive architectures with competitive accuracy, they typically suffered from high computational complexity as the objective did not account for the efficiency of the architecture. Subsequent works proposed to optimise the complexity of the architectures in addition to its predictive quality to derive efficient models. These approaches achieve the same accuracy with lower complexity as the objective is able to differentiate between models in the search space with the same quality but different efficiency.

However, these approaches do not explore an accuracy latency trade-off in an automated manner. After deriving a performant base model, these approaches typically scale the depth, width or input resolution of the model to meet varying efficiency requirements of applications (Tan et al., 2019; Wu et al., 2019; Cai et al., 2018). Later works dealt with this shortcoming by optimising the efficiency to search for models across the pareto frontier (Cai et al., 2020; Banbury et al., 2021).

Specialist works in TinyML quite often include the memory usage and model size in the objective to fit models within internal storage and memory constraints (Fedorov et al., 2019; Lin et al., 2020, 2021; Banbury et al., 2021; Liberis et al., 2021; Fedorov et al., 2022).

Approximations: The quality of the candidate model would ideally be measured as the accuracy of the trained model on the validation data. However, training sampled candidate models to completion in the search process becomes prohibitively expensive with such approaches requiring up to 40,000 GPU hours (Real et al., 2017; Zoph et al.,

2018; Cai et al., 2020). To alleviate this, search algorithms often take approximations to reduce the computational cost.

A common approach adopted is to train the candidate architectures for a limited number of epochs with shared weights to reduce training time (Pham et al., 2018). Similarly, the search is often carried out using lower input resolutions or on a smaller proxy dataset, with the model then being transferred to larger datasets (Zoph et al., 2018; Real et al., 2019; Lin et al., 2020). To avoid the computational cost of evaluating numerous candidate models on the validation dataset which can also be quite large, accuracy predictors are also utilised for evaluating models (Cai et al., 2020; Dudziak et al., 2020).

In a similar manner, the computational complexity is also often estimated using the MACs or latency predictors (Cai et al., 2020; Dudziak et al., 2020) as measuring on-device latency for all target platforms and large search spaces with upto 1.15×10^{152} candidate architectures (Liberis et al., 2021) can become infeasible.

Early works in NAS typically jointly trained and navigated the search space to derive a model (Zoph et al., 2018; Cai et al., 2018; Real et al., 2019; Lin et al., 2020). This approach had the limitation that a single search required training the entire search space and deriving multiple models required training the search space from scratch.

To address this challenge, super-network based approaches have been used where the training of the search space, *i.e. the super-network*, is decoupled from the search (Cai et al., 2020; Wang et al., 2021a,b). In these approaches, a super-network is initialised which contains all the candidate architectures. The super-network is first trained with a specialised training strategy that trains candidate architectures. Once the training is complete, the search can be carried out, according to some objective with a low cost, as it only consists of sampling and evaluating models. Importantly, if the objective changes, the search can be performed on the super-networks without the need for any retraining in the search space. Such approaches often require specialised training recipes as training a large number of candidate architectures with shared weights results in interference between between the models.

2.2.1.3 Differentiable Architecture Search (DARTS)

Differentiable Architecture Search utilises a gradient based approach to navigate the search space of architectures. This is done by employing a differentiable relaxation of the categorical selection of candidate architectures in the search space (Liu et al., 2018; Cai et al., 2018; Shin et al., 2018; Xie et al., 2019). This can be achieved through a Softmax or Gumbel-Softmax distribution over some architecture parameters. The target criterion, such as the validation loss can then be minimised with respect to the architecture parameters that parameterise the distribution over the architecture enabling the automatic discovery of architectures.

In the work carried out in this thesis, we build on the approach of DARTS (Liu et al., 2018). One of the limitations of their work was that they derive architectures that achieved the highest accuracy. In our work, we derive models of varying complexity in an automated manner from the search space in Chapter 3 by jointly optimising the predictive performance and complexity of the architectures. As we build on the machinery of DARTS, we go through the details of the original work in this section.

Continuous Relaxation In DARTS, the authors replace the categorical selection of an operation to a softmax distribution over all possible candidate operations. This approach combines the candidate operations to form a *mixed operation*. For a set of candidate operations '$\mathcal{F}$' (e.g depth-wise separable convolutions, max pooling, skip connection), each candidate operation, j would be defined as a function of the form $f_j(s_a; w_j)$, parameterised by weights, w_j and applied to input s_a. The candidate operations in $\mathcal{F}$ were then combined in a mixed operation, m_0 to produce output, $x^{(a,b)}$ as shown below

$$x^{(a,b)} = m_0\left(s_a; w_{0,j}, \alpha_{0,j}\right), \qquad \forall j \in \mathcal{F} \tag{2.1}$$

$$= \sum_{j \in \mathcal{F}} \sigma\left(\alpha_{0,j}\right) \times f_j\left(s_a; w_{0,j}\right) \tag{2.2}$$

$$x^{(a,b)} = \sum_{j \in \mathcal{F}} \frac{\exp\left(\alpha_{0,j}\right)}{\sum_{j' \in \mathcal{F}} \exp\left(\alpha_{0,j'}\right)} \times f_j\left(s_a; w_{0,j}\right) \tag{2.3}$$

where $\alpha_{0,j}$, $j \in \mathcal{F}$ is the architecture weight for the jth candidate operation in the $0th$ mixed operation and $w_{0,j}$ are the weights that parameterise the jth candidate operation in the the $0th$ mixed operation. Note that if the mixed operation, m_0 combined 'q' operations together, the architecture weights between representations s_a and $x^{(a,b)}$ would be a q-dimensional vector, $\vec{\alpha}_0$. The working of a mixed operation that applies the mentioned transformation from s_a to $x^{(a,b)}$ is also shown visually in Figure 2.3. As can be seen from Figure 2.3, $x^{(a,b)}$ would be a weighted average of the outputs of the candidate operations due to the softmax operation on the architecture weights, $\alpha_{0,j}$, $\forall j \in \mathcal{F}$.

It is important to note that every mixed operation on the input s_a would have its own architecture weight vector. For example, a second mixed operation, $m_1(s_a; w_{1,j}, \alpha_{1,j}), \forall j \in \mathcal{F}$ performed on s_a to produce representation $x^{(a,c)}$, would have its own distinct architecture weights, $\alpha_{i,j}$, $\forall j \in \mathcal{F}$. In the DARTS algorithm, the magnitude of the architecture weight is used as a proxy for a candidate operations importance since it would be large for an operation that contributed more to the output representation, $x^{(a,b)}$. In the coming sections we detail how the mixed operations are grouped together into 'cells' which were stacked together to form more complex networks. We also show how the

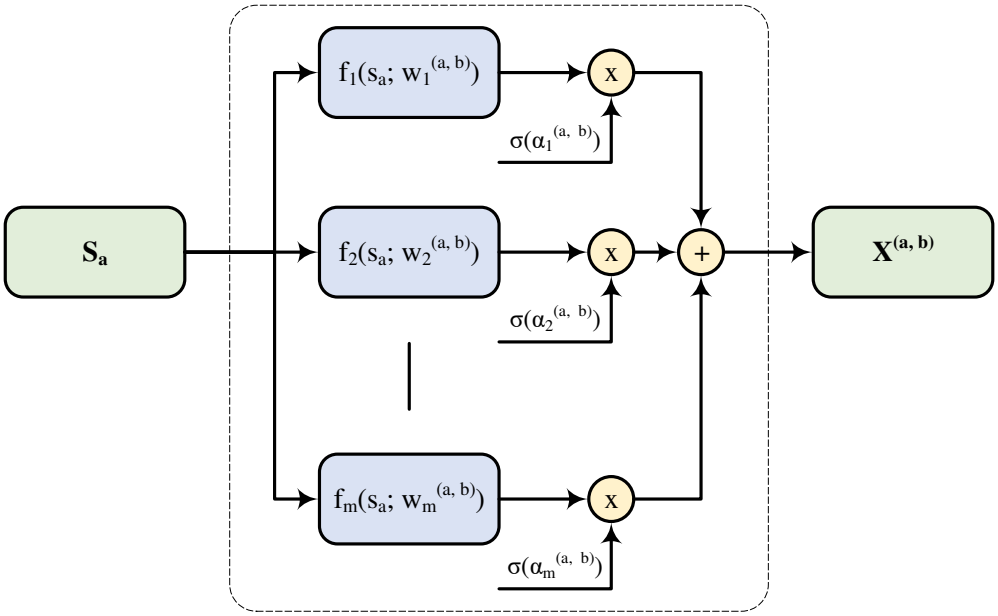


FIGURE 2.3: Structure of a Mixed Operation applied to a representation, s_a to produce an output $x^{(a,b)}$

magnitude of the softmax of the weights is used to select the best candidate operation and derive discrete cells.

Cell Structure The search space for more complex networks is created by combining a number of mixed operations in a 'cell'. In the DARTS algorithm, the architecture search process actually comprises of searching for cells. More complex networks are created by stacking the discovered cells to a user specified depth which would depend on the complexity of the target task for which the network was being derived. This was done to keep the architecture search process tractable since the search space when searching for an entire network would be too large.

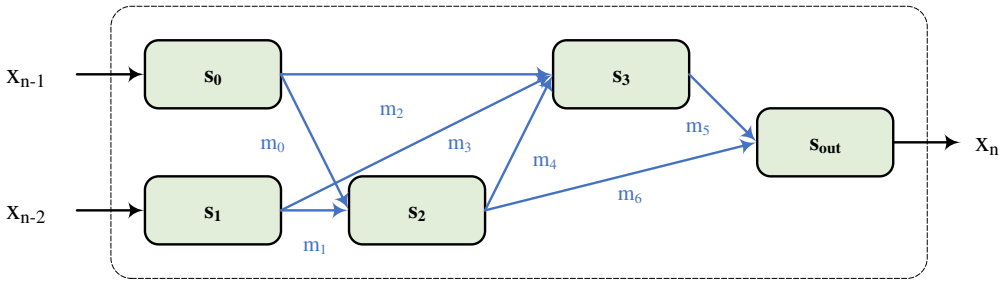


FIGURE 2.4: Structure of a 2-step Cell composed of 5 Mixed Operations

The structure of an N-step cell used in the DARTS approach is shown in Figure 2.4 where cell, C_n is a directed acyclic graph of an ordered sequence of N nodes discounting the output and two input nodes. An N-step cell, C_n would have N nodes where each node, s_k in the nth cell represents an intermediate representation (e.g. a feature map in a convolutional neural network). In the graph each node, s_k is connected to each of its predecessor nodes, s_i through an edge representing a mixed operation. The representations, s_k are then produced by carrying out mixed operations on each of the predecessor representations, s_i which are all concatenated together to produce, s_k. The output of the cell is then computed by concatenating all the intermediate representations in the cell except for the input representations in the cell. The concatenation operations are carried out in the channel dimension of the representations. The intermediate representations or output representation of an N-step cell are computed according to Eq. 2.4 and 2.5 respectively where $\prod$ denotes the concatenation operation and P_k denotes the set of predecessor representations for any representation, s_k.

$$s_k = \prod_{i \in P_k} m_i \left(s_i; w_{i,j}, \alpha_{i,j} \right), \qquad \forall j \in \mathcal{F} \tag{2.4}$$

$$s_{out} = \prod_{k=2}^{N} s_k \tag{2.5}$$

In the DARTS cell structure, the two input representations, s_0 and s_1 are passed in from the output of cells, C_{n-2} and C_{n-1} respectively with some pre-processing performed on them. One thing to note is that the numerous concatenation operations leads to the size of feature maps blowing up in deeper parts of the network. As concatenation is carried out in the channels dimension this results in larger filter sizes and more computational overhead. We discuss how DARTS deals with this issue through pre-processing and size reduction methods in the next section where we discuss how cells are stacked together to form more complex networks.

To control the complexity and representational capacity of a cell, the DARTS algorithm allows two degrees of freedom which are treated as hyper-parameters set according to the complexity of the task for which the architecture was being searched. The first hyper-parameter is the number of candidate operations used to produce a mixed operation. The second is the number of steps in the cell. As mentioned previously, the step size determines the number of intermediate representations within the cell which are connected by mixed operations in order to find the best operation during the architecture search process. For a step size of $N > 0$ the number of mixed operations required to connect each representation node to its predecessors could be simply calculated as shown in Eq. 2.6.

$$p = \sum_{i=2}^{N+1} (i) \tag{2.6}$$

If each of these p mixed operations had $q = \mid \mathcal{F} \mid$ candidate operations the architecture weights for a cell would be contained in a $p \times q$ matrix. Throughout the rest of this report we denote the architecture weight matrix as α. In the notation we use from here on, $\alpha_{i,j}$ can be understood to be the weight for the jth candidate operation in the ith mixed operation of a cell.

In their implementation the authors included the following 7 candidate operations in $\mathcal{F}$: 'Skip Connection', '3×3 Average Pooling', '3×3 Max Pooling', '3×3 and 5×5 Separable Convolutions' and '3×3 and 5×5 Dilated Separable Convolutions'. The authors used the Conv-BatchNorm-ReLU processing order for convolution operations in order to stabilise the training process similar to Zoph and Le (2017) and Real et al. (2017). Similarly, all 3×3 and 5×5 separable convolutions were applied twice and dilated convolutions were always applied with a dilation of 2. To keep the spatial resolution of output representations of candidate operations the same, padding was carried out where necessary.

Network Structure A network is created by stacking together multiple cells to the depth required. This is shown in Figure 2.5 where each cell, C_n has two inputs, which are the output of cells, C_{n-1} and C_{n-2}, except for the first cell where the input is simply fed in twice. The depth of the network is treated as a hyper-parameter dictated by the difficulty of the task to be learned.

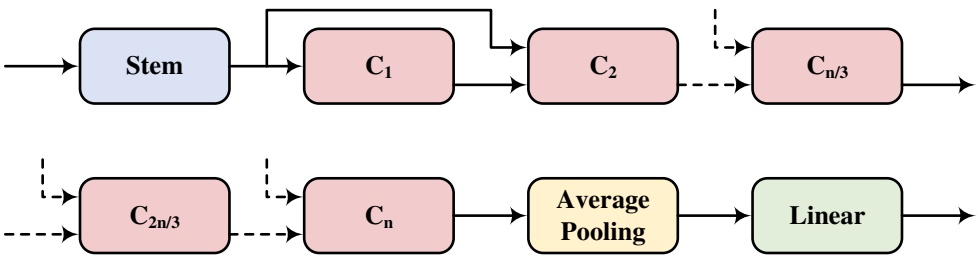


FIGURE 2.5: Structure of Network Composed by Stacking Together Multiple Cells

As mentioned previously, the output representation of a cell was obtained by concatenating all the intermediate representations in the cell (excluding the input feature maps). With cells being consecutively stacked together the representations would become very large due to the successive concatenation operations. To deal with this problem, reduction cells are deployed in the network. In the reduction cells, the operations performed on the input representations are performed with a stride of two. Additionally the pre-processing performed on the input representations of the cell is carried out differently. For normal cells the pre-processing is carried out as a Conv-BatchNorm-ReLU operation with stride 1, while for reduction cells it is applied with a stride of 2.

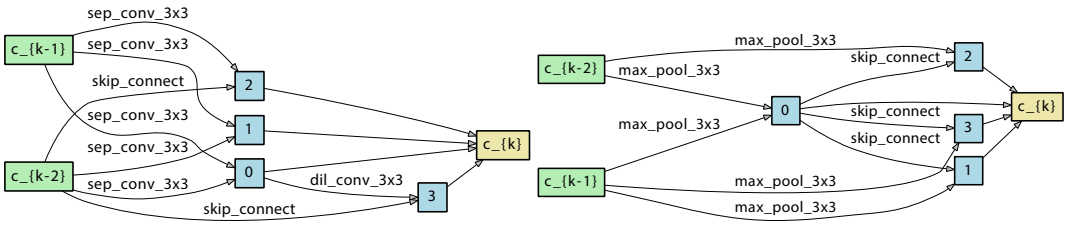
This approach requires searching for two cells, a normal cell and a reduction cell. In DARTS the cells, $C_{n/3}$ and $C_{2n/3}$ at 1/3rd and 2/3rd of the depth of a network with n cells were designated as reduction cells.

When searching for cells, the architecture weights for mixed operations between two representations are shared between cells. However, the weights that parameterised the candidate operations within the mixed operations in each cell are distinct. This is true for both normal and reduction cells. So the process of architecture search requires finding the best architecture weights, α^{normal} and α^{reduce} for normal and reduction cells which was used to create a complex network.

Coming back to the network structure, the authors employ a stem processing block which consists of a Conv-RELU operation which controls the number of channels that are input to the first cell. At the end of the network an adaptive average pooling layer is placed to reduce the size of the output representation of cell, C_n before it is fed to the final linear layer.

Deriving Discrete Architectures By searching for architectures for a number of epochs the architecture weights, α^{normal} and α^{reduce} could be obtained. We defer discussion of the training process for architecture search to the next chapter where we discuss our multi-objective formulation of the problem and show its compatibility with the training scheme of the DARTS work.

In order to discretize the architecture, the best candidate operation in a mixed operation between representations s_a and s_b is obtained by simply selecting the candidate operation with the largest weight. Once all the mixed operations are discretised to the best operations, the top-k strongest operations that produced a representation were retained with the rest being dropped. In the DARTS algorithm k was chosen as 2, similar to prior works. The structure of the cells derived by the DARTS algorithm is shown in Figure 2.6.



(A) Normal Cell (B) Reduction Cell

FIGURE 2.6: Cells Discovered by DARTS

2.2.2 Dynamic Neural Networks

As opposed to static Neural Networks, Dynamic Neural Networks (DyNNs) are able to adapt their structures and parameters during inference. This allows DyNNs to perform inference more efficiently in addition to enabling the models to adaptively trade-off accuracy and efficiency depending on resource constraints.

2.2.2.1 Dynamic Structure

A common approach to realising dynamicism in an NN is to adapt the structure at run-time. According to requirements, a sub-network within the base network can be activated which is used to perform inference. The task of designing the DyNN thus requires designing a network which contains suitable sub-networks which share weights. These approaches leverage the insight that an entire network is not needed to classify all examples. The structure of the sub-networks is typically derived by trimming the base network in three dimensions below

Width: A widely used method to increase the efficiency is to dynamically alter the width of the neural network based on requirements. In the case of CNNs, the width is controlled by the number of channels in a convolutional layer or block. At run-time, the channels of a filter to be convolved with are selected based on the input. This is typically done with a gating module that takes an input tensor and generates a binary mask to select which filters are applied or equivalently which channels of the input to process. The gating module is typically a neural network that is learned when training the base network. A number of gating approaches have been proposed which explore the granularity, the structure and complexity of the gating module along with its training methods (Mullapudi et al., 2018; Gao et al., 2018; Chen et al., 2019; Bejnordi et al., 2019; Herrmann et al., 2020).

These approaches are able to boost the average efficiency of a model over a number of inferences. However, they do not guarantee a given efficiency for a sample apriori. To meet real-time constraints, the width is altered in a non-input dependent manner with hard-coded rules. A common approach is to globally trim the width with the resulting sub-networks sharing weights (Yu et al., 2018; Yu and Huang, 2019). A limitation of these works is that the sampling of selected channels leads to non-contiguous tensors which are incompatible with kernels for MCU based devices (Lai et al., 2018).

Routing: Another approach adopted to achieve dynamicism is to perform dynamic routing in the base-network and adapt the inference graph at run-time. One way this

is achieved is by selectively executing one of several independent candidate modules at each layer based on the input (Liu et al., 2018).

A special case in dynamic routing is, dynamic depth also referred to as *early exiting*. In early-exiting approaches, a number of intermediate classifiers are appended to intermediate layers in the network. In a *budgeted classification* setting, the average efficiency over numerous inferences is optimised with early-exiting based on the confidence of intermediate classifications. This approach executes a subset of layers for easier examples that do not require the full representational power of the model.

The *any-time classification* approach is better suited to meeting real-time latency constraints where an early-exit is made based on the compute constraints. There have been several works which explore the selection of hyper-parameters involved in the design of early-exiting DyNNs such as structure, location and exiting strategy in early-exiting classifiers. These approaches typically utilise an existing model as a base model or manually design the base model such that it is amenable to early-exiting (Teerapittayanon et al., 2016; Huang et al., 2018; Dai et al., 2020).

Recent works have proposed using NAS to derive early exiting DyNNs which have achieved good performance when considering accuracy and efficiency (Gambella and Roveri, 2023; Bouzidi et al., 2023; Li et al., 2023).

2.3 Model Optimisation

For on-device deployment a model is often optimised for efficient inference performance. In this section, we discuss pruning and quantisation which are two popular optimisation techniques relevant to this thesis.

2.3.1 Pruning

Pruning approaches have been proposed to zero out or remove unimportant weights. This can be used to reduce the memory footprint or computational complexity of performing inference. Several different approaches have been adopted for pruning which cater to different performance metrics and hardware as below.

Structured vs Unstructured: LeCun et al. (1990) initially proposed unstructured pruning to remove unimportant weights and reduce the network complexity. Subsequent works have produced impressive compression rates with minimal loss to accuracy (Han et al., 2016; Frankle and Carbin, 2018). The resulting sparse matrices can be stored

using specialised storage formats to reduce the memory footprint of the models (Naumov et al., 2010). However, unstructured pruning approaches require specialised hardware, kernels and very high compression ratios (Wang et al., 2021a; Nunez-Yanez and Hosseinabady, 2021) to achieve latency comparable to dense matrix multiplications.

Structured pruning algorithms impose structure in the process such that the resulting matrix operations can be accelerated with kernels optimised for Basic Linear Algebra Subprograms (BLAS) (Lai et al., 2018; Fatica, 2008). In CNNs, such approaches usually prune the channels or entire filters from the convolutional layers (Li et al., 2017; Roy et al., 2020; Huang et al., 2018). This allows the usage of standard kernels optimised for non-sparse convolutions (Jorda et al., 2019; Chetlur et al., 2014; Lai et al., 2018).

Criterion: An important consideration in pruning approaches is the criteria that is used to determine the importance or *saliency* of a weight or block of weights. A common metric used is the L1/2 norm of the weight or group *(channels in CNNs)*, which utilises the insight that higher magnitude weights are more important than smaller ones (Han et al., 2016; Roy et al., 2020). Older approaches (LeCun et al., 1990; Hassibi and Stork, 1992; Hassibi et al., 1993; Dong et al., 2017) have utilised second order derivatives of the loss with respect to weights to measure their importance, however these are often infeasible for larger networks due to the complexity of computing the hessian matrix. Other approaches inspect the output feature maps to determine the importance of weights producing output elements (Hu et al., 2016).

Regularisation: During training of networks, regularisation can be introduced in the objective function to produce networks amenable to pruning. The regularisation can be performed by applying L_0, L_1 or L_2 norm penalties (Louizos et al., 2018; Wen et al., 2016) or targeted dropout (Gomez et al., 2019) on weights or weight groups.

2.3.2 Quantisation

A common approach in compression adopted to lower the memory and computational complexity of DNNs is quantisation. Quantisation makes use of the insight that the distribution of weights and activation tensors during training and inference have a limited range and can be represented with a coarse granularity compared to the typically used 16/32/64 floating point format. To meet memory or performance requirements, quantisation has been proposed to lower the precision of a model during training and inference. Quite often models will be trained in a high precision data format such as 32 or 64 bit floating point precision. Once a model has been trained, the model can then be quantised to a lower precision during inference. This method of quantisation

after training is formally called *post-training quantisation*. Quantisation approaches may adopt scalar or vector approaches.

2.3.2.1 Vector Quantisation

In vector approaches, also known as *product or block quantisation*, a vector of elements that could be either 1-D or 2-D are clustered into the closest vector that can accurately represent the vectors where the clusters are typically learned through k-means clustering (Han et al., 2016; Fan et al., 2021). This limits the number of unique vectors that are represented in the network which can be stored with some code words to produce a very high compression rate which can massively reduce the size storage requirement for model parameters.

However, such approaches have overheads of having to decode the weights using the codeword and codebook. As such the effect on performance would need to be studied further to discertain these approaches suitability for MCUs.

2.3.2.2 Scalar Quantisation

As mentioned, in the scalar approach to quantisation, each element is represented with lower precision elements. There are typically a number of considerations to take into account when utilising any quantisation scheme.

Precision: A model is usually trained in a high level framework in high precision such as 32/64-bit floating point which occupies 4 or 8 bytes of memory per element. The precision can be lowered to reduce the memory footprint and latency of operations. Various precisions from 8 to 64 bits (Wang et al., 2018) are used with different floating point formats (Kalamkar et al., 2019; Rouhani et al., 2020).

Integer quantisation is also used extensively to avoid expensive floating point arithmetic. With integer quantisation, a number of precisions have been used for inference which are able to preserve accuracy. Courbariaux et al. (2015) showed that the precision using fixed point representations of the weights and activation could be lowered to 10 bits training and inference. In subsequent work, this has been reduced to 8-bits (Jacob et al., 2018), and below (Banner et al., 2019; Zhu et al., 2017; Mellempudi et al., 2017) with binary representations (Zhou et al., 2016; Courbariaux et al., 2015) in the extreme case being used for weights and/or activations.

Lower precisions are able to reduce the latency by lowering costly data fetching operations. The efficiency of low-precision arithmetic depends on the characteristics of the underlying hardware such as the minimum addressable unit (MAU) and precision

of operands in the instruction set (Wang et al., 2019). On devices such as MCUs, 8-bit precision is preferred as this efficiently maps onto SIMD instructions (Lai et al., 2018) as opposed to lower precisions that leads to higher latency (Capotondi et al., 2020; Rusci et al., 2020a).

Granularity: Quantisation can be applied at different levels of granularity. In the extreme, fine grained quantisation performed per element would yield the most accurate model, however, this would have high overheads of determining the range of every element in a tensor. As such, more coarse granularities have been explored. Layer-wise quantisation approaches examine the statistics (e.g. min, max) over all the parameters or elements in a weight or activation tensor using it to determine the quantisation parameters. This often results in sub-optimal accuracy as the distribution of different filters can have different ranges (Krishnamoorthi, 2018).

Groupwise quantisation approaches group together a number of channels or parameters inside a layer to calculate the clipping range used in quantisation (Shen et al., 2020). Channel-wise quantisation can be viewed as a coarse case of groupwise quantisation where one group is the number of channels (Zhang et al., 2018). Further granularity can be achieved through sub-channel quantisation, however this adds considerable overhead.

Channel-wise and per-tensor quantisation for weights and activation tensors respectively has been used extensively with good accuracy and latency performance (Lin et al., 2020, 2021; Banbury et al., 2021).

Symmetric and Asymmetric: Symmetric quantisation schemes have a range that is equally spaced around the origin. Asymmetric quantisation, on the other hand, relaxes this constraint which gives more flexibility to quantise imbalanced distributions, such as the output of ReLU operation which is always non-negative. A benefit of symmetric quantisation is that it requires fewer quantisation parameters which can reduce the computational cost during inference (Wu et al., 2020; Jacob et al., 2018).

Uniform vs Non-Uniform: The process of scalar quantisation maps representations from real values to a representation from set of discrete values *(or quantisation levels)*. In uniform quantisation, the quantisation levels are uniformly spaced (Jacob et al., 2018). This approach can have high levels of quantisation noise as real values may not map well to equally spaced quantisation levels.

Non-uniform scaling approaches have been proposed which allow the quantisation levels to be non-uniformly spaced (Gong et al., 2015; Miyashita et al., 2016). Non-uniform quantisation may achieve higher accuracy than the uniform approach as it

may be able to model the distribution better and reduce quantisation noise. Such approaches can yield higher compression ratios, however, uniform approaches can be more efficiently deployed on commodity hardware (e.g. MCUs, GPUs) which has lead to their widespread adoption (Gholami et al., 2022).

Static vs Dynamic Quantisation: Static or dynamic quantisation dictates at what point the quantisation parameters (e.g. min, max) are determined. As the weights typically do not change during inference, these can be quantised and fixed during inference. On the other hand, activation tensors can vary during inference depending on the input to the model. Static approaches utilise a representative dataset to fix the quantisation parameters during inference for every input (Jacob et al., 2018). Dynamic quantisation, computes the quantisation parameters based on the input or batch of input data (Liu et al., 2022). Typically, the dynamic approach is able to achieve better accuracy, but introduces extra computational overhead.

2.3.2.3 Linear Quantisation on MCUs

In this thesis we utilise the linear quantisation proposed by Jacob et al. (2018) which is widely used in research and has been adopted by popular inference frameworks. In this section we review some details of the quantisation scheme including the integer arithmetic and the quantisation parameters to serve as background knowledge for the research contributions in Chapter 4.

The proposed scheme performs static, uniform quantisation to 8-bit precision to minimise the memory footprint and leverage native instructions of the MCU platforms. The quantisation of real numbers, r is performed as an affine mapping parametrised by a real valued scaling factor, S and integer valued zero point, Z as shown in Figure 2.7 which is mathematically expressed as below

$$r = S(q - Z) \tag{2.7}$$

The proposed scheme uses a static approach to derive the quantisation parameters, S and Z. The parameters are derived per tensor for activation tensors and per channel for weights within the network. To compute the parameters a representative dataset is utilised which typically consists of 500 data points from the training data. The representative dataset is fed to the model to generate a histogram of the elements of the activation tensors. The histogram is then utilised to determine the minimum and maximum value of the real valued elements, r_{min} and r_{max} in each activation tensor which are mapped to the corresponding maximum and minimum integer value, q_{min} and q_{max}

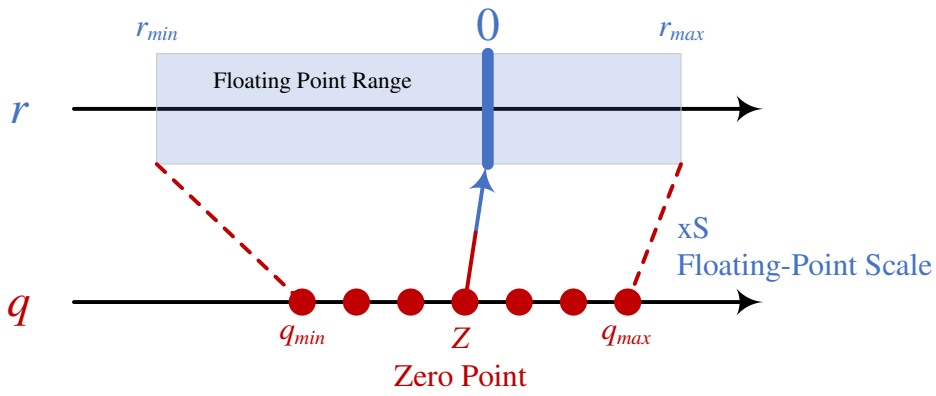


FIGURE 2.7: Linear Quantisation. Adapted from Han (2023).

representable with the quantised representation. For quantisation to 8-bits using an unsigned integer representation, the values of q_{max} and q_{min} would be 255 and 0. The two real values are expressed as below to generate two simultaneous equations.

$$r_{max} = S(q_{max} - Z) \tag{2.8}$$

$$r_{min} = S(q_{min} - Z) \tag{2.9}$$

By subtracting Equation 2.9 from Equation 2.8 we can determine the scale factor, S as below

$$\begin{aligned} r_{max} - r_{min} &= S(q_{max} - Z) - S(q_{min} - Z) \\ r_{max} - r_{min} &= S \cdot q_{max} - S \cdot q_{min} - S \cdot Z + S \cdot Z \\ S &= \frac{r_{max} - r_{min}}{q_{max} - q_{min}} \end{aligned} \tag{2.10}$$

where q_{max} and q_{min} are constants known beforehand. Similarly, r_{max} and r_{min} are known beforehand for weight tensors, while they are determined through the representative dataset for activation tensors. The quantisation scheme performs symmetric quantisation for weight tensors due to which the zero-point of weight tensors is set to 0. On the other hand, once the scaling factor, S is known, the zero-point, Z can be straightforwardly determined from Equation 2.9 as below

$$Z = round\left(q_{min} - \frac{r_{min}}{S}\right) \tag{2.11}$$

As mentioned previously, the quantisation is performed at a per tensor granularity for activation tensors and per channel for weight tensors. Using the quantisation scheme described, there is one pair of quantisation parameters, S and Z for an activation tensor and C pairs for weight tensors, where C is the number of filters in the weight matrix.

Memory Usage The scale parameter, S in the quantisation scheme is a real valued-number. To implement the quantisation scheme using integer-only arithmetic a number of approximations and optimisations are performed which involve pre-computing and storing some quantisation parameters used in performing the arithmetic. This pre-computing results in the scale and zero point being converted into multiplication and shift parameters stored in 32-bit fixed point format which each have a memory requirement of **4C** bytes. The quantisation scheme also combines the bias tensor with the quantisation parameters due to which the bias tensor is stored in 32-bit precision and has a memory requirement of **4C** bytes. Details of how the linear quantisation is performed using only integer arithmetic is provided in Section A.1 in Appendix A.

2.4 Model Deployment

2.4.1 Hardware Characteristics

Hardware platforms suitable for TinyML applications are significantly different to the tradition GPU platforms. In this section we analyse and contrast some relevant characteristics of the tiny platforms with the traditional GPU platforms to highlight the challenges associated with performing DNN inference on the resource constrained devices.

The field of TinyML typically employs hardware with power consumption under 1 W to support battery operated scenarios. As shown in Table 2.1, these platforms, such as DSPs and MCUs, have significantly lower power consumption than GPU platforms. However, the challenge of on-device DNN inference comes from the reduced compute and memory capacity of the devices.

As can be observed, the number of cpu cores and clock speeds are three orders of magnitude lower than GPUs with a comparable difference in the internal and external memory available. Similarly, Tensor Processing Units (TPUs) are deep learning accelerators that provide better energy efficiency than GPUs through specialised matrix multiplication units in the hardware, but the gap in resources remains significant. Other

Device	CPU	Internal		External		Power (W)
		RAM	Storage	RAM	Storage	
GPU: Nvidia A100	6912@1.4GHz	40GB	-	64GB	TBs	300W
TPU: Google TPUv4	8xMXUs@1050MHz	32GB	-	64GB	TBs	170W
Mobile: Adreno 730	8@2.8GHz	8GB	256GB	Shared	GBs	15W
EdgeTPU: Cortex A53+eTPU	4@1.3GHz+1xMXU@500MHz	8MB	-	4GB	GBs	10W
FPGA: Arty A7-100T	240xDSPs@100MHz	101,440 LUTs	-	256MB	16MB	<1W
DSP: TMS320DM6446	1@459MHz	180KB	-	256MB	128MB	<1W
MCU: STM32F469NI	1@180MHz	256KB	1024KB	256MB	256MB	<1W

TABLE 2.1: Characteristics of compute platforms used for DNN workloads

deep learning accelerators have also been developed such as Graphcore Intelligence Processing Units (IPU), Cerebras Wafer Scale Engines (WSE) or Tenstorrent Grayskull devices which have varying memory hierarchies, interconnect technologies and parallelism styles (Jia et al., 2019; Lie, 2022; Doerner, 2024). However, these accelerators also have an order of magnitude more resources than MCUs or DSPs. With the recent interest in performing inference on edge devices, GPU and TPU architectures have been scaled down for lower power consumption in devices such as the Adreno 730 GPU, or the Coral EdgeTPU. These devices retain the parallel processing compute paradigm of cloud and server based GPUs but significantly scale down the compute resources available onboard which in turn reduces the power consumption. These edge GPUs and TPUs operate in a multi-core fashion where they are coupled with a general purpose processor, such as a Cortex A53, which is responsible for driving the GPUs and TPUs. However, as shown in Table 2.1, these devices also have an order of magnitude higher resources and power consumption compared to low-power MCUs or DSPs. Field Programmable Gate Arrays (FPGAs) provide an interesting avenue for deployment of models in TinyML due to their low power consumption and high compute capabilities achieved through specialised hardware. However, these devices suffer from a higher developmental complexity compared to other software-based platforms. One further drawback, is that FPGAs are typically an order of magnitude higher in cost compared to MCUs or DSPs which leads to higher bill of materials. The difference in resources and characteristics of the hardware leads to varying requirements from the inference frameworks for different hardware platforms, and motivates the need for development of specialised models and inference frameworks for MCUs and DSPs.

In the following sections we discuss some additional characteristics of the hardware including the instruction set and available peripherals relevant to prior works and the research carried out in this thesis.

2.4.1.1 Instruction Set

Primitive operations in a deep neural network are mapped to linear algebra operations. Layers such as fully-connected layers are implemented as matrix multiplication

operations. Similarly, a convolution operation is also converted to a matrix multiplication operation through the im2col operation. The workhorse in matrix multiplication operations and therefore a neural network is the multiply-and-accumulate (MAC) operation.

Due to their native instruction set, DSPs and ARM Cortex MCUs supporting DSP instructions are a good fit for algorithms, such as CNNs, that heavily employ the MAC operation. These devices further support SIMD instructions that allow processing multiple operands in a single instruction. An example of such an instruction in the ARM Cortex MCUs is the *SMLAD* instruction that can multiply two sets of operand and accumulate the results into two destinations. DSP and MCU devices typically only support fixed point arithmetic operations which requires usage of fixed point methods discussed previously.

2.4.1.2 Peripherals

In addition to the CPU core which is responsible for executing the instructions, the DSPs and MCUs have a range of peripherals that can be configured and utilised as per application requirements. These include serial ports, timers, data/programme cache, direct memory access (DMA), and external memory interfaces. We provide a brief description of peripherals relevant to this thesis below.

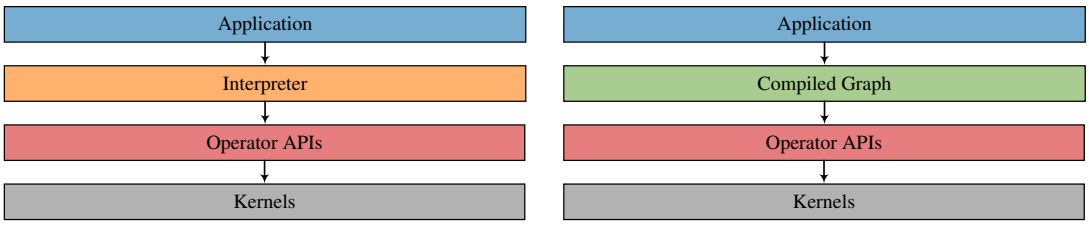
Direct Memory Access (DMA): The DMA is an internal peripheral utilised to perform memory transfers between a source and destination. The source and destination can be configured to be either an address in the memory space or some memory-mapped peripheral e.g. serial port. The CPU is responsible for configuring the DMA to perform data movement. Once configured, the start of the data transaction can be asserted by the CPU or another peripheral. Once the transaction has been initiated, the movement is performed independently by the DMA without any CPU action. After completion of the transaction, the CPU is responsible for maintenance operations (e.g. cache coherence, interrupt servicing).

External Memory Interfaces: The DSP and MCU chips have a limited amount of fast internal memory and storage. This can be supplemented with external memory and storage to extend the address space of the devices. On ARM Cortex MCUs, the Flexible Memory Controller (FMC) and Quad Serial Port Interface (QSPI) are utilised to add additional volatile (DRAM) and non-volatile (NAND Flash) memory for use by the application. Additionally, an SDMMC interface is also available to interface with portable microSD cards.

2.4.2 Inference Frameworks

CPU, GPU and Accelerator based platforms have a number of popular frameworks for model design, training and inference (Paszke et al., 2019; Burrello et al., 2021; Abadi et al., 2016; Vanholder, 2016; Meta, 2024a; Google, 2024a) that are designed according to the underlying hardware. These frameworks typically utilise optimised kernel libraries, e.g. cuDNN (Chetlur et al., 2014), to accelerate the primitive operations *(conv, matmul, add)* found in DNNs.

The memory and compute resource constraints of edge devices compared to traditional CPU or GPU platforms requires the usage of specialist inference frameworks. These are designed to be light-weight with majority of overhead performed statically (e.g. memory management and allocation) at either initialisation or compile-time, with only the most essential operations performed in the run-time. With these objectives and constraints, two main approaches have emerged in the development of deep learning frameworks with the *code-generation* and *interpreter* based approach.



(A) Interpreter Based (B) Compiler Based

FIGURE 2.8: Code-Generator vs Interpreter Runtime Execution Flow. Adapted from David et al. (2021)

Interpreter: The interpreter based approach is designed to be flexible and enables cross-platform interoperability. TensorFlowLite-Micro (TFLM) (David et al., 2021) is a widely used framework that adopts this approach. TFLM uses an interpreter which contains all the run-time information required to perform inference such as the inference graph, data structures for weight/activation tensors, primitive operations etc. As shown in Figure 2.8, the interpreter loads a model that is serialised with a FlatBuffer (Koparkar, 2023) representation. An operator resolver is used to include all relevant primitive operations in the compiled binary. The interpreter uses a block of memory and a memory planner to statically allocate memory for buffers (activation tensors, im2col) needed to perform inference according to their lifetime and size. The interpreter can also be configured to utilise a given set of kernels that are optimised for the underlying hardware. Inference can be invoked by the application on input data through APIs provided by the interpreter. When inference is invoked, the interpreter sequentially runs through operations in the inference graph and computes the operations by invoking primitive operations provided by the kernels through the operator

APIs. The major benefit of the interpreter based approach lies in its flexibility and portability. Additionally, as the framework code is decoupled from the model definition, models can easily be updated in the field.

Code-Generation: The code-generation based approach has been adopted by a number of inference frameworks including uTensor, TinyEngine, MicroTVM, and XCubeAI (uTensor, 2024; Lin et al., 2020; MicroTVM, 2024; Microelectronics, 2019). These approaches are adopted to avoid the overheads of the interpreter based approach, albeit at the cost of flexibility. The code generators parse a DNN definition of a model and generate the minimal amount of code, parameters and primitive operations required to perform inference with a model. Actions such as memory planning, operation resolving are performed at code generation time, with a single invoke API provided for inference. These approaches result in a lower memory footprint for the inference framework (Lin et al., 2020).

2.4.2.1 Compiler Sections

The inference frameworks used for MCUs are written in C/C++. A program written in C/C++ contains components including code and different types of data. When the program is compiled, the C/C++ compiler compiles the components into object files. These object files contain the compiled components in a number of different *compiler sections*. Some examples of sections commonly used by the compiler are given below

- **.text:** used to store executable code
- **.data:** contains all *initialised* data, including global variables and arrays
- **.bss:** used to reserve space for *unitiliased* data, including global variables and arrays

The object files are then passed to a *linker* which combines the respective sections of all the object files and places them in different *memory segments* as shown in Figure 2.9. The memory segments are defined in a *linker command file* according to the memory layout of the underlying hardware. As an example, an ARM Cortex MCU utilises a flat memory model where different types of memory including SRAM, DRAM, Flash, ExtFlash are mapped to different address ranges. The memory segments can be defined in the linker command file with a base address and length which map to the different memory types. The linker can then be configured to place the compiled sections into the declared memory segments. For context, a sample linker command file is given in Section A.2 in Appendix A.

The placement algorithm is typically determined by suitability and requirements. For example, initialised data is often placed in non-volatile memory, whereas frequently accessed data can be placed in faster on-chip memories. We note that the compiler toolchain typically allows the manual creation of sections and placement of variables or functions in these sections through linker or compiler directives.

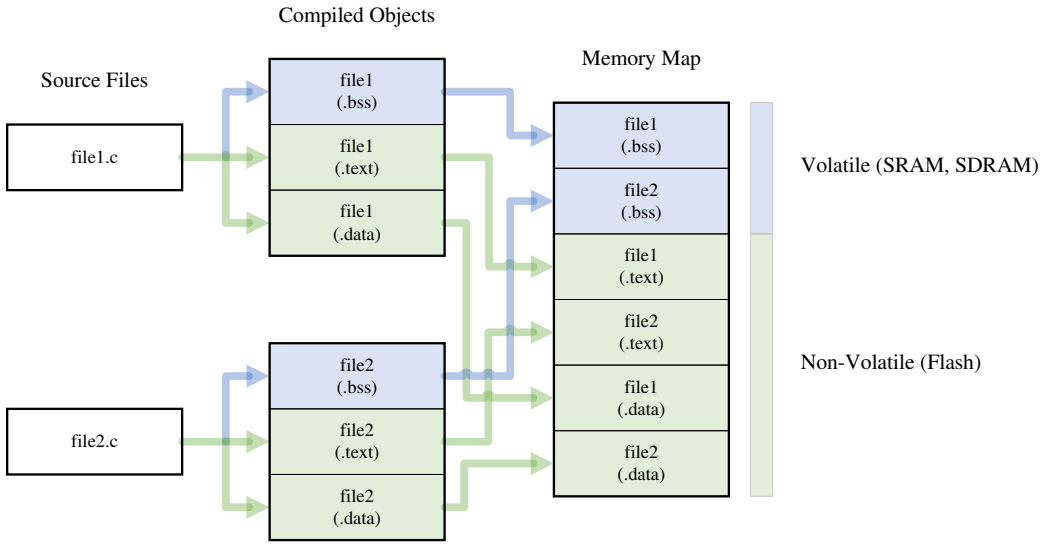


FIGURE 2.9: Compiler Sections to Memory Segments

To compile a model for deployment, the model is serialised into one or more constant arrays and included in the source code which can be accessed by the kernels when performing an operation. These typically contain weights or quantisation parameters that do not change through inference.

The activation tensors or intermediate buffers are allocated by the inference frameworks memory planner and are similarly declared as pointers that are offsets into a statically allocated memory block.

2.4.2.2 Memory Usage

A limitation of current inference frameworks works is that they do not perform any memory management at the program level or utilise the memory hierarchy when compiling the data into sections and memory segments. The majority of inference frameworks utilise the default placement strategy of the C/C++ compiler where the initialised and uninitialized data are placed into their default sections.

With this approach all initialised constant arrays *(weights)* are compiled into the .data section and uninitialized arrays *(activation tensors)* are compiled into the .bss section. As such, this memory usage approach imposes the constraint of having the entirety of

weights and tensors in their respective memory segments. Additionally, during inference, data is only ever read from or written to the memory segment it was placed in during compilation.

Further, we noted that prior works typically only consider internal storage and memory in the memory layout due to energy efficiency and low access latency (Lin et al., 2020, 2021, 2022; Banbury et al., 2021; Rusci et al., 2018, 2020a,b). As mentioned previously, different memory segments can be mapped to memory regions of the address space which are connected to different types memories with unique characteristics. These can be internal memory and storage (SRAM, Flash) which are limited in size but offer low access latency and power consumption. On the other hand, external memories (SDRAM, Flash, SD Cards) can be utilised through external memory interfaces which are larger but have higher latency and power consumption. We note that exploring the memory hierarchy and an efficient memory management system at the program level holds opportunities for performance gains on MCUs.

Alternative Approaches We note and discuss some exceptions to the memory usage approach discussed previously in this section. Wang et al. (2020) proposed a code generation based approach which considers a memory hierarchy and attempts to store weights as close as possible to the CPU. However, the memory hierarchy they consider only consists of internal SRAM and Flash. As we show, in later sections, the amount of internal memory significantly constrained the model size and therefore achievable performance. Further, they considered smaller networks with only fully connected layers which have limited activation tensor sizes. In our work, we focus on CNNs, which can have activation tensors larger than the size of internal memory. As such, this requires a partitioning scheme to reduce the memory requirement that we introduce in Section 4.2.5.

Svoboda et al. (2022) explored a memory hierarchy by utilising external SD Cards on the SDMMC interface which are sufficient to hold the weights and tensors for state-of-the-art CNNs. However, this interface is significantly slower compared to other available external memory interfaces such as the Flexible Memory Controller (FMC) and Quad Serial Port Interface (QSPI). As a result, they achieve an inference latency two orders of magnitude slower (100x↑) than internal memory as compared to using the FMC and QSPI which is also slower (1.5-3.5x↑) but on the same order of magnitude as we demonstrate in Section 4.1. They also lack a method to mitigate the slow access latency of external memory, such as one we propose in Chapter 4.

2.4.3 Kernels

The kernels provide efficient implementations for primitive operations (e.g. conv, matmul) that are used in DNNs. These kernels are hardware specific and are typically developed to deliver high throughput through specific optimisations suitable for the hardware. As such, the implementation of the kernels is usually abstracted away from the inference framework to allow for portability of the inference framework across devices as shown in Figure 2.8. This allows the kernels to be developed independently and integrated with the inference framework to provide efficient inference functionality to an application.

A variety of kernels have been developed for GPUs, CPUs, Mobile including cuDNN, cuBLAS, Math Kernel Library (MKL), Eigen, XNNPack or QNNPack (Chetlur et al., 2014; Nvidia, 2024; Basics, 2005; Guennebaud and Jacob, 2010; Google, 2024b; Meta, 2024b). Examples of kernels developed for MCU platforms include CMix-NN (Capotondi et al., 2020), TinyEngine (Lin et al., 2020) and CMSIS-NN (Lai et al., 2018) which support different quantisation schemes and apply a number of different optimisations. In this thesis, we focus on the widely used and open-source CMSIS-NN kernels.

CMSIS-NN

The CMSIS-NN library (Lai et al., 2018) provides optimised implementations for primitive operations on ARM Cortex MCUs that are quantised to 8-bit integers using the scheme of (Jacob et al., 2018). The library utilises a number of techniques to accelerate inference including SIMD programming to leverage the MCUs instruction set and partial im2col to accelerate convolutions with optimised matmul kernels under internal memory size constraints. We briefly review the matrix multiplication kernel and partial im2col approach to show the memory requirement and provide background knowledge for research contributions in Chapter 4.

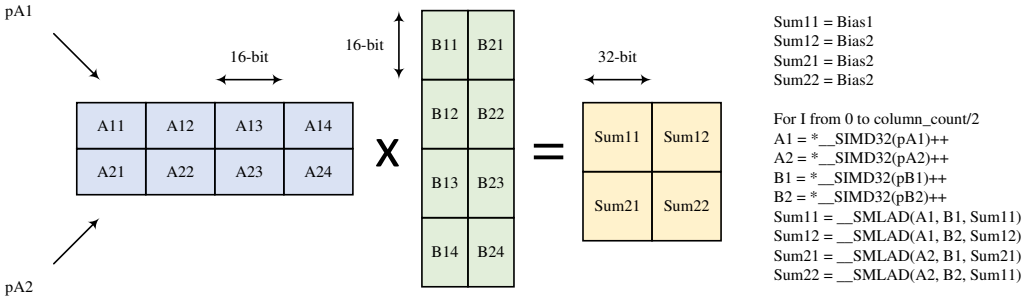


FIGURE 2.10: General Matrix Multiplications. Adapted from Jacob et al. (2018)

Matrix Multiplication The instruction set of the MCU includes DSP and SIMD instructions. These instructions are able to process multiple operands in a single instruction, which reduces memory overheads in addition to instruction cycles. Matrix multiplication kernels extensively utilise the *SMLAD* instruction which is able to multiply and accumulate two sets of 16-bit operands. The CMSIS-NN kernels use the SIMD instructions to perform matrix multiplications using 2×2 kernels. As shown in Figure 2.10, the kernel multiplies two rows and columns at a time. pA_m and pB_m are used to load two 16-bit operands, into A_n and B_n which are 32-bit variables. These are then processed using the SMLAD instruction which multiplies the corresponding upper half words and the lower half words and accumulates the results in Sum_{mn}. By loading and processing two operands at a time, the load, branching and compute overhead is reduced by a factor of two.

We note an extra step omitted from the pseudo-code in Figure 2.10, which is required to perform 8-bit integer arithmetic. This step involves reading 8-bit operands and sign extending them to 16-bits which can be consumed by the SMLAD instruction.

Partial im2col A common approach to performing convolutions uses the im2col algorithm which vectorises the convolution operation. This approach takes receptive fields or patches that a kernel lies over and flattens each of them into vectors. These vectors are then concatenated together to create a matrix. The kernels that the image is to be convolved with are then arranged as a matrix. This allows the convolution to be performed efficiently as a matrix multiplication using general matrix multiply (GEMM) algorithms.

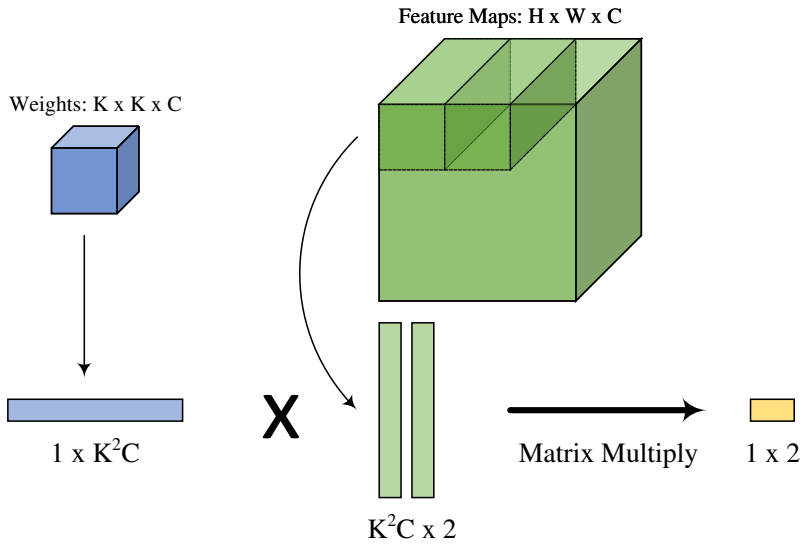


FIGURE 2.11: Convolution with partial im2col

A drawback of the im2col operation is that it has a high memory overhead as every patch that the kernel lies over in the image must be flattened into a vector. For a representation of size $H \times W \times C$, the memory requirement would be K^2CN elements where where K is the kernel size, C is the number of channels, and N is the number of patches the kernel lies over during convolution.

Due to limited memory on MCUs, the CMSIS-NN library implements convolution operations with a partial im2col algorithm. This partial approach vectorises only two receptive fields of the image at a time into two columns of a matrix. This approach performs the convolution in $\frac{C}{2}$ steps where each step consists of creating the partial im2col buffer followed by a matrix multiplication. Using this method the size requirement for the partial im2col buffer in the convolutions is

$$M = 2 \times 2 \times K^2 \times C \tag{2.12}$$

where M is the required memory in bytes, K is the kernel size of the filter, C is the number of input channels. The constant factors of 2 come from vectorising two patches at a time and from the SIMD instructions only supporting half-word (16-bit) operands due to which the operands in the partial im2col buffer have to be cast up from 8-bits to 16-bits. In the CMSIS-NN kernels, the partial im2col algorithm is used in both convolution and grouped convolutions frequently encountered in CNNs.

2.5 Discussion

This chapter summarised the background and literature relevant to the work carried out in efficient deep learning inference on MCUs. We discussed different stages in the deep learning inference pipeline including model design, optimisation, and deployment.

In model design we discussed how the earlier approaches in NAS focused on deriving efficient models with the highest accuracy. However, to meet varying complexity constraints it would be desirable to have automated methods to derive models across the pareto frontier. This motivates **RQ1** which this thesis addresses in Chapter 3 where we explore how gradient based optimisation can be utilised to derive models across the pareto frontier in an automated manner.

We also discussed model deployment approaches for MCU based platforms. We highlighted that the majority of prior works only considered internal storage and memory and designed models within the respective size constraints. This motivates **RQ2**, addressed in Chapter 4 where we explore how unused but available peripherals on the

MCUs including external memories can be utilised to enable efficient design and deployment of models on MCUs.

In our review of efficient model design we discussed how the heuristics used to manually design the models or search space were developed in a platform agnostic manner or for more popular GPU or mobile platforms. This motivates **RQ3** addressed in Chapter 5, where we perform an analysis of DNN deployment on MCU devices to derive design heuristics for efficient model and search space design. Finally, edge based applications must often meet changing latency constraints that are determined by the application or environment. In **RQ4**, studied in Chapter 5, we study how inference can be adapted at run-time across the pareto frontier using dynamic and static NAS based approaches within the constraints of the inference framework.

Chapter 3

DEff-ARTS: Differentiable Efficient ARchiTecture Search

In designing a CNN, a designer typically has multiple degrees of freedom to explore the architecture of the model. These degrees of freedom, or hyper-parameters, include different architectural elements of the model including, but not limited, to the number of layers, the layer type and the configuration of the layer i.e. number of channels, kernel size. When manually designing a model, the hit and trial nature of selecting these operations often leads to a high search cost as evaluating the model requires compute intensive training of the model. Further, the selection of these operations requires expert human-knowledge to guide the manual navigation through the search space.

Neural Architecture Search (NAS) has emerged as a promising approach to automating the tedious process of designing CNN models. These approaches utilise gradient based search, reinforcement learning or evolutionary optimisation to search for the optimal architecture in the search space. However, these works focused on deriving the model of the highest accuracy. On the other hand, edge devices, often need to cater to real-time latency constraints that can vary according to environmental conditions. In such a case, it would be desirable to have methods that are able to trade-off the quality and efficiency of the model. In this chapter we seek to answer the question, **RQ1:** *How can we derive models of varying complexity using automated methods?*.

We present, Differentiable Efficient Architecture Search, or DEff-ARTS (Sadiq et al., 2020), a gradient based algorithm to derive architecture of varying complexity. DEff-ARTS utilises the machinery of DARTS (Liu et al., 2019) discussed in Section 2.2.1.3, as it was shown to have competitive performance with a low search cost. Liu et al. (2019) relax the categorical selection of candidate operations in the inference graph to a softmax distribution which is parameterised by learnable architecture weights. To derive an architecture, they minimise the cross entropy loss and learn the architecture

weights along with the operations weights. After training, an argmax operation is used to derive the architecture that minimises the cross entropy loss.

To derive architectures of varying complexity, DEff-ARTS converts the problem into a multi-objective optimisation problem. In addition to the cross-entropy loss, we optimise the computional complexity of the network with the trade-off between the two sub-objectives being controlled by a single hyper-paramter. We show that this approach can be used with the bi-level gradient based optimisation of DARTS to derive architectures of varying complexity with no extra search cost. The work presented in this chapter, makes the following contributions to answer research question, **RQ1**.

- **Multi-Objective Optimisation** We combine the two sub-objectives of computational cost and cross-entropy loss which have significantly different optimisation landscapes so that the search problem can be treated as a multi-objective optimisation problem where we can trade-off between the sub-objectives
- **Experimental Evaluation** We perform experiments on the CIFAR-10 dataset to navigate the optimisation landscape via gradient descent and derive architectures of varying complexity

3.1 Non-Linear Multi-Objective Bi-Level Optimisation Problem

In this section we explain the formulation of the learning problem as a multi-objective bi-level optimisation problem. By treating the problem as a multi-objective optimisation problem we combined the search for an architecture with low cross-entropy loss, $\mathcal{L}_{ce}$ with searching for the most efficient architecture with a low compute cost, $\mathcal{L}_{com}$. The objectives were combined into a measure we call the performance loss, L_{per} which we explain in the coming sections.

3.1.1 Compute Cost

In order to incorporate efficient architecture search in the search process we minimised the computational compexity of the cell in the continuous search space in addition to minimising the cross entropy loss. We refer to this measure of computational complexity as the *compute cost*. In order to back-propagate gradients from the loss function we required the compute cost function to have a differentiable closed form. Our formulation of the closed form of the compute cost, $\mathcal{L}_{com}$ is shown in Eq. 3.1

$$\mathcal{L}_{com} = \Sigma_{i=1}^{p}\Sigma_{j=1}^{q}\sigma(\alpha_{i,j}^{normal}) \times Cost_j + \Sigma_{i=1}^{p}\Sigma_{j=1}^{q}\sigma(\alpha_{i,j}^{reduce}) \times Cost_j \tag{3.1}$$

where $\alpha_{i,j}^{normal}$ and $\alpha_{i,j}^{reduce}$ are the architecture weights shared between the normal and reduction cells respectively, σ is the softmax function and $Cost_j$ is the amount of CPU cycles required to apply the jth candidate operation. One common way of measuring the computational complexity of networks is to compute the multiply and accumulate (MAC) operations required for one forward pass. However, this assumes that all operations take the same amount of CPU cycles to compute. In our approach, we calculated the number of different CPU operations required by the candidate operations and computed the required cycles according to the cost of the low-level CPU operations.

For simplicity, in our approach we considered the comparison, addition, multiplication and division operations when computing the cost of a candidate operation and considered 32-bit operations. Memory read and write operations from and to external or internal memory were ignored in our computation. We used the Texas Instruments C64x+ DSP architecture (Texas-Instruments, 2019), commonly used in low power embedded systems, as a reference for the number of cycles required for the different operations. These are shown in Table 3.1.

Operations (32 bit)	**Cost (CPU Cycles)**
Comparison (GT, LT)	1
Addition	1
Multiplication	4
Division	4

TABLE 3.1: CPU Operation Costs, C64x+ (CPU Cycles)

We manually computed the number of different operations required by the candidate operations as shown in Eqs. 3.2 - 3.6 where H, W and C_{in} are the height, width and channels of the input representation, whereas k, S and C_{out} are the size of the square kernel, stride size and number of output channels of the operations respectively. Note that the separable convolution cost is multiplied by 2. This was done since in the DARTS implementation depthwise-separable convolutions were always applied twice.

$$\begin{aligned} Cost_{sep} = \Big[&\big[\left(k \times k \times H \times W \times C_{in} \div S^2\right) + \left(C_{in} \times H \times W \times C_{out}\right) \big] \times Cost_{mul} \\ &+ \big[\left(k \times k \times H \times W \times C_{in} \div S^2\right) + \left(C_{in} \times H \times W \times C_{out}\right) \big] \times Cost_{add} \Big] \times 2 \end{aligned} \tag{3.2}$$

$$\begin{aligned} Cost_{dil} = &\big[\left(k \times k \times H \times W \times C_{in} \div S^2\right) + \left(C_{in} \times H \times W \times C_{out}\right) \big] \times Cost_{mul} \\ &+ \big[\left(k \times k \times H \times W \times C_{in} \div S^2\right) + \left(C_{in} \times H \times W \times C_{out}\right) \big] \times Cost_{add} \end{aligned} \tag{3.3}$$

$$Cost_{avg} = (k \times k \times H \times W \times C_{in}) \times Cost_{add} + (H \times W \times C_{in}) \times Cost_{div} \tag{3.4}$$

$$Cost_{max} = (k \times k \times H \times W \times C_{in}) \times Cost_{comp} \tag{3.5}$$

$$Cost_{skip} = 0 \tag{3.6}$$

Candidate Operation	Cost (CPU Cycles)
Max Pool 3x3	27,648
Avg Pool 3x3	30,720
Skip Connect	0
2*(Sep Conv 3x3)	368,340
2*(Sep Conv 5x5)	860,160
Dil Conv 3x3	184,320
Dil Conv 5x5	430,080

TABLE 3.2: Candidate Operation Costs

Looking at Eqs. 3.2 - 3.6 the natural question that arises is what values would be used for the input representation size since the dimensionality of the representation would constantly be changing throughout the depth of the network. For simplicity, we computed the cost according to the input image size where $H = 32$, $W = 32$ $C_{in} = 3$ and $C_{out} = 3$ and assumed a stride of 1. This is a significant assumption to make since it ignores the representation size in deeper parts of the network and the different strides in reduction cells. However it provided a reasonable heuristic to compare the cost of different operations. In Section 3.2, we demonstrate that this implementation of the compute cost discovered efficient architecturesca with a significantly reduced computational complexity. Using Eqs. 3.2 - 3.6, the cost calculated for the candidate operations is given in Table 3.2. We note that this approach would be easily applicable to other target devices as well such as MCUs or other instruction set based architectures. This would only require changing the CPU cycles for the low-level CPU operations in Table 3.1 which would result in the candidate operations cost characterising some other target device.

3.1.2 Performance Loss

3.1.2.1 Linear Combination

The initial approach we considered was to take a simple linear combination of the cross entropy loss, $\mathcal{L}_{ce}$ and the compute cost, $\mathcal{L}_{com}$ as shown in Eq. 3.7.

$$\mathcal{L}_{per}(w, \alpha) = \mathcal{L}_{ce}(w, \alpha) + \Gamma \times \mathcal{L}_{com}(\alpha) \tag{3.7}$$

where Γ was a user defined constant cost weightage parameter. This approach was not pursued too far for two reasons. Firstly, there was a problem of the range and scale of the two sub-objectives being minimised. Secondly, we observed that in its simple form, the compute cost, $\mathcal{L}_{com}$ penalised the computationally expensive operations like separable convolutions too heavily as compared to simpler operations like max-pooling.

To explain the first concern further, we observed that the cross entropy loss during training was in the 10^{-1} to 10^{-2} range, while the compute cost range which can be computed from Eq. 3.1 would range up to 10^{6}. It can also be noted that according to the formulation of the compute cost, it would be monotonically increasing. Using a simple linear combination of the sub-objectives, the problem arises that the landscape of the compute cost would completely dominate the landscape of the cross entropy loss. We explain this problem visually through artificially generated two-dimensional loss landscapes. Figure 3.1 shows an intuitive sketch of the cross entropy landscape with $\sigma(\alpha_1)$ characterising the strength of a more expressive and expensive operation such as a convolution operation and $\sigma(\alpha_2)$ characterising a simpler operation such as a max-pooling. Note that the minima of the landscape would lie in the region where the more expressive convolution operation was stronger.

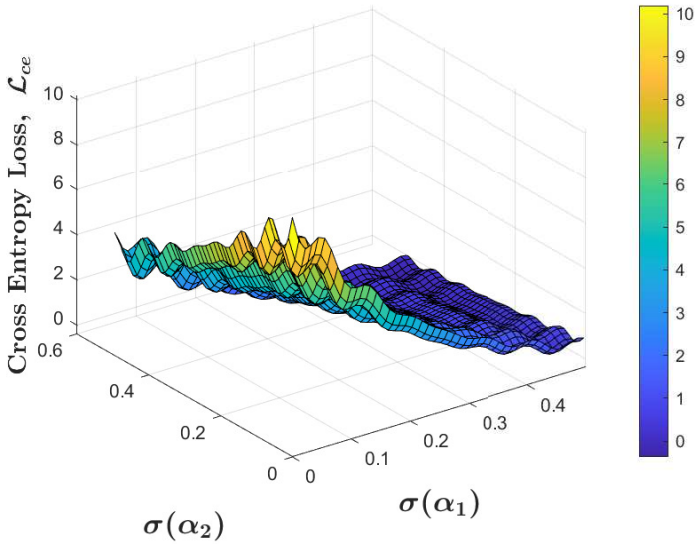
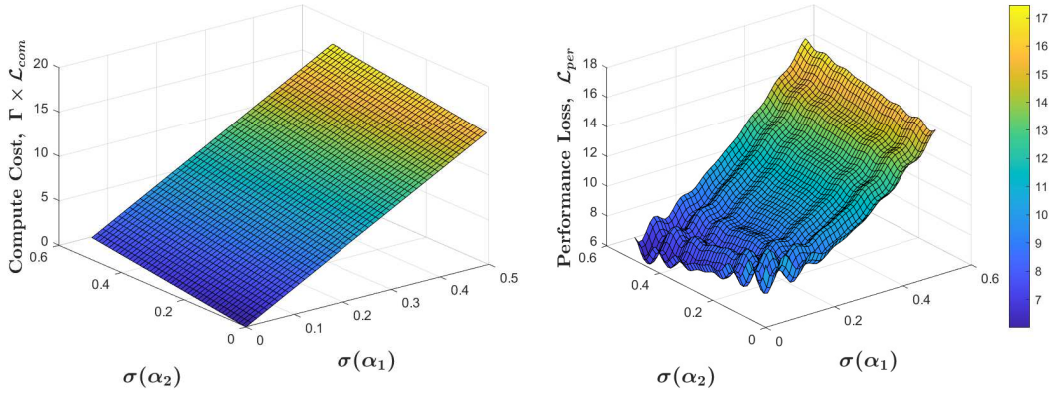


FIGURE 3.1: Sketch of Cross Entropy Loss Landscape, $\mathcal{L}_{ce}$

Similarly, Figure 3.2a shows the landscape for the compute cost with the horizontal axis characterising the same operations. In this case, we can see that when the strength of an expensive operation is high, the compute cost is high and conversely when the strength of a cheap operation is high the compute cost is low. Figure 3.2b shows the performance loss obtained by a linear combination of the sub-objectives. It can be seen that due to the difference of scale between the two sub-objectives the minima of the performance loss is totally determined by the compute cost and the landscape properties of the cross entropy loss are insignificant. In practice, the difference in scale between the two landscapes was observed to be four orders of magnitude larger than the difference shown in our synthetic landscape. Similarly, the difference in gradients in different dimensions of the compute cost was also three orders of magnitude larger than demonstrated in the example.



(A) Compute Cost, $\Gamma \times \mathcal{L}_{com}$, $\Gamma = 1$ (B) Performance Loss, $\mathcal{L}_{per}$

FIGURE 3.2: Compute Cost with Linear Scaling and Resulting Performance Loss

One trivial approach to dealing with this problem would be to scale down the cost weightage parameter, Γ. However, scaling Γ would have an equal effect on all dimensions. If we tried to scale down the landscape in an expensive dimension, the cheaper dimension would become negligible. This would suggest the idea of applying non-linear scaling which we discuss in the next section.

Another problem with taking a linear combination of the sub-objectives is that it assumes that the two can be linearly traded off. However we empirically observed that this was not the case. We trained a number of different architectures with varying cell complexity and recorded their training loss at the end of training. A plot comparing the two and how they varied with each other is shown in Figure 3.3.

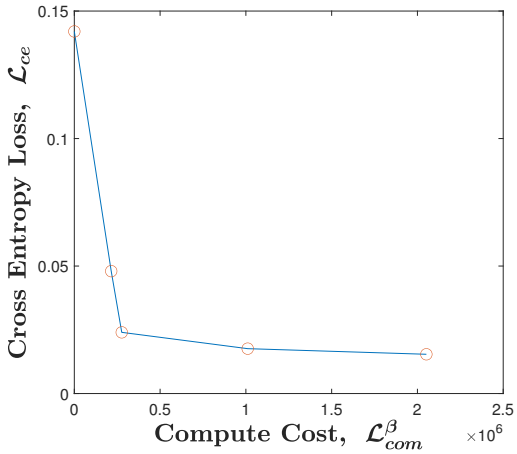


FIGURE 3.3: Compute Cost, $\mathcal{L}_{com}$ vs. Cross Entropy Loss, $\mathcal{L}_{ce}$

It can be clearly seen there is no linear relationship between the compute cost and the cross entropy loss. One way to deal with this issue might be to have a non-linear cost weightage function to replace the constant Γ parameter. The approach we adopted was to apply a non-linear transformation to one of the costs which would enforce a linear

relationship between the sub-objectives. This enabled us to trade-off between the two sub-objectives through the simple cost weightage parameter, Γ.

3.1.2.2 Non-Linear Transformation

In order to avoid the issues with linearly scaling the compute cost discussed in the previous section, we applied a non-linear transformation to the compute cost and took a linear combination of the cross entropy loss, $\mathcal{L}_{ce}$ with the transformed compute cost, $\mathcal{L}_{com}^{\beta}$ to obtain the performance loss, $\mathcal{L}_{per}$. This is shown in Eq. 3.8.

$$\mathcal{L}_{per}(w, \alpha) = \mathcal{L}_{ce}(w, \alpha) + \Gamma \times \mathcal{L}_{com}(\alpha)^{\beta} \tag{3.8}$$

By looking at the above equation the natural question that arises is what the value of the modulation parameter, β should be. One empirical method of estimating this parameter to obtain pareto optimal solutions is to empirically measure how the cross entropy loss changes with the compute cost (Tan et al., 2019). We note that estimating the value of the modulation parameter would be different to a standard curve-fitting problem as the value of the performance loss would be unknown to us in addition to the modulation parameter, β. As such we utilised the empirical approach mentioned previously for which we trained two different architectures with different compute costs and recorded the cross entropy loss after training. The observed values are shown in Table 3.3.

Compute Cost, $\mathcal{L}_{com}$	**Cross-Entropy Loss,** $\mathcal{L}_{ce}$
$322,560$	3.11×10^{-1}
$1,244,160$	9.32×10^{-2}

TABLE 3.3: Observed Change in Cross-Entropy Loss with Change in Compute Cost

These values were substituted into Eq. 3.7, along with a suitable value of $\Gamma = 1$. We utilised a value of $\Gamma = 1$ since we wanted to assess how the sub-objectives changed with respect to each other and a non-unit value would result in scaling of the compute cost.

$$\mathcal{L}_{ce,1} + \mathcal{L}_{com,1}^{\beta} = \mathcal{L}_{ce,2} + \mathcal{L}_{com,2}^{\beta} \tag{3.9}$$

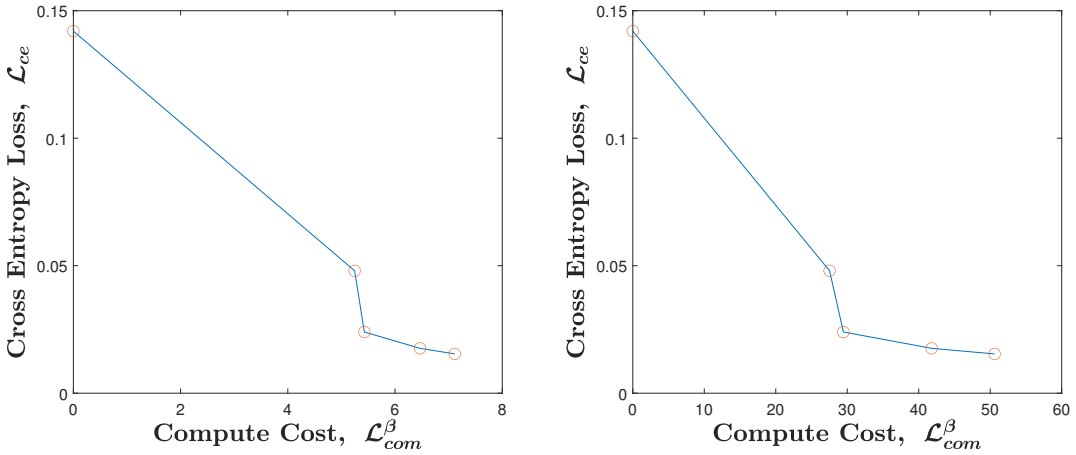
$$9.32 \times 10^{-2} + 1,244,160^{\beta} = 3.11 \times 10^{-1} + 322,560^{\beta} \tag{3.10}$$

However, substituting the values gives us a transcendental equation where we cannot obtain a closed-form solution for β. In that case, we can perform a simple manipulation

to convert the problem into a root finding problem as shown below

$$1,244,160^{\beta} - 322,560^{\beta} - 2.178 \times 10^{-1} = 0 \tag{3.11}$$

Solving for β using Newton's method we obtained the value of $\beta = 6.64 \times 10^{-2}$ for the modulation parameter. However, we were not able to produce consistent results with this value, nevertheless this value proved to be a useful heuristic which guided selection of a usable β value. In practice we used a value of $\beta = 0.27$ for the modulation parameter which was acquired through a logarithmic grid search performed from a starting value of $\beta = 6.64 \times 10^{-2}$. The relationship between the cross-entropy loss after training and the transformed compute cost is shown in Figure 3.4 for different values of the modulation parameter.



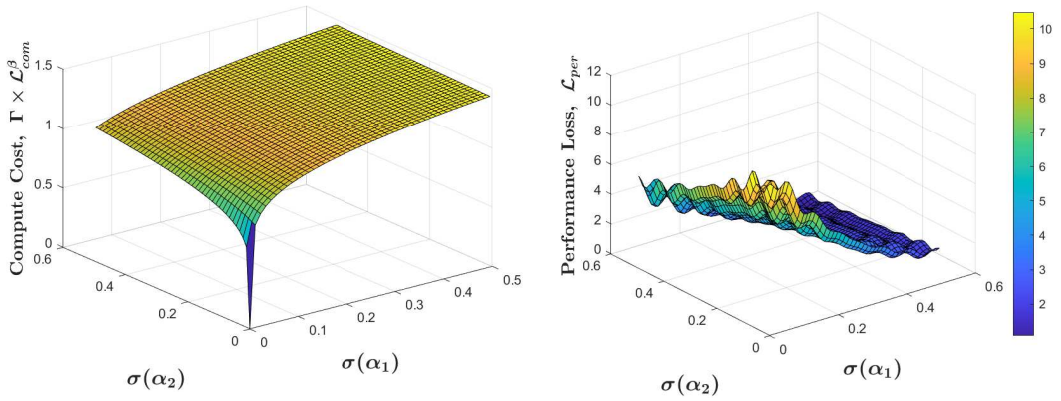
(A) $\beta = 0.067$ (B) $\beta = 0.27$

FIGURE 3.4: Non-Linearly Transformed Compute Cost, $\mathcal{L}^{\beta}_{com}$ vs. Cross Entropy Loss, $\mathcal{L}^{\beta}_{ce}$.

By comparing Figure 3.3 and Figure 3.4 we can see that by modulating the compute cost, the relationship between the two sub-objectives is significantly more linear than it was originally. However, it can be observed that it is still not perfectly linear. It is possible that further improvements could be made by using other non-linear transforms but we leave this as a future work since in practice, we were able to navigate the multi-objective optimisation landscape with exponentiation as we show in Section 3.2.

We further demonstrate how exponentiating the compute cost brought the two sub-objectives to a comparable scale and how the performance loss landscaped changed as a result of the non-linear transform of the sub-objective. The effect of modulating the compute cost on our synthetic landscape with $\beta = 0.135$ is shown in Figure 3.5a with the same description of the axes as in Figure 3.2a. It can be observed that the problem of the scale and gradients varying significantly between the dimensions of the architecture weights has been alleviated. The resulting performance loss landscape is

shown in Figure 3.5b demonstrating that the gradients or minima of the cross entropy landscape are no longer dominated by the compute cost.



(A) Compute Cost, $\Gamma \times \mathcal{L}_{com}^{\beta}$ (B) Performance Loss, $\mathcal{L}_{per}$

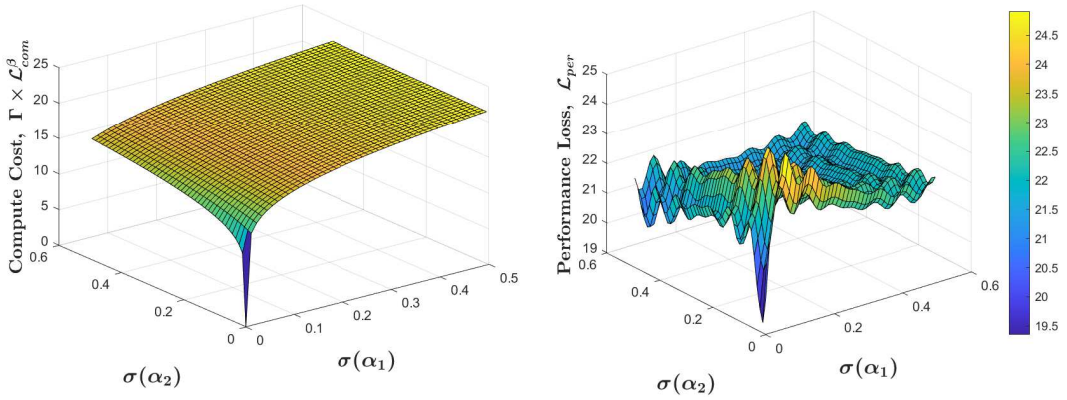
FIGURE 3.5: Compute Cost with Non-Linear Scaling and Resulting Performance Loss, $\Gamma = 1, \beta = 0.135$

Having sufficiently linearised the relationship between the sub-objectives, the trade-off between the two would now be controllable through the cost weightage parameter, Γ. Equation 3.13 analytically shows how the cost weightage parameter controls the strength of gradients back-propagated, and consequently updates to the final architecture weights.

$$\frac{\partial}{\partial \alpha_{i,j}^{normal}} \mathcal{L}_{per}(w, \alpha) = \frac{\partial}{\partial \alpha_{i,j}^{normal}} \left(\mathcal{L}_{per}(w, \alpha) + \Gamma \mathcal{L}_{com}(w, \alpha)^{\beta} \right) \tag{3.12}$$

$$= \frac{\partial}{\partial \alpha_{i,j}^{normal}} \mathcal{L}_{per}(w, \alpha) + \Gamma \times \frac{\partial}{\partial \alpha_{i,j}^{normal}} \mathcal{L}_{com}(w, \alpha)^{\beta} \tag{3.13}$$

As can be seen the cost weightage parameter simply scaled the gradients back-propagated from the compute cost with higher values giving higher weightage to selecting an efficient architecture. For a value of $\Gamma = 15$ and $\beta = 0.135$, the compute cost is shown in Figure 3.6a. The resulting performance loss is shown in Figure 3.6b. It can be noted that the features such as local minima and maxima of the cross-entropy loss landscape are preserved in the performance cost landscape. Further, it can be seen that the global minima in the landscape has moved to a location where the cheaper operation associated with the weight, α_2 is preferred in the architecture which is what we would want the final outcome to be when applying a higher weightage to minimising the compute cost.



(A) Compute Cost, $\Gamma \times \mathcal{L}_{com}^{\beta}$ (B) Performance Loss, $\mathcal{L}_{per}$

FIGURE 3.6: Compute Cost with Non-Linear Scaling and Resulting Performance Loss, $\Gamma = 15$, $\beta = 0.135$

3.1.3 Multi-Objective Bi-Level Optimisation

Having formulated the problem as a multi-objective optimisation problem, in this section we detail the training process used to find the architecture encoded by the weights, α. We used the same scheme proposed by (Liu et al., 2018) which was compatible for our multi-objective loss function as well.

Let $\mathcal{L}_{per,train}(w, \alpha)$ and $\mathcal{L}_{per,val}(w, \alpha)$ denote the performance loss on the training and validation set respectively. The goal of architecture search could then be defined as finding the architecture weights, α^* that minimised $\mathcal{L}_{per,val}(w^*, \alpha^*)$ where w^* are the weights that minimise the training loss such that $w^* = \underset{w}{\text{argmin}} \mathcal{L}_{per,train}(w, \alpha^*)$. The problem then becomes a bi-level optimisation problem with α as the upper-level variable and w as the lower-level variable as shown below

$$\min_{\alpha} \quad \mathcal{L}_{per,val}(w^*(w, \alpha), \alpha) \tag{3.14}$$

$$\text{s.t.} \quad w^*(w, \alpha) = \underset{w}{\text{argmin}}\ \mathcal{L}_{per,train}(w, \alpha) \tag{3.15}$$

In this optimisation problem, the inner level problem requires minimising the performance loss which is a combination of the cross-entropy and compute cost with respect to the network parameters. In this formulation, the compute cost is simply a constant in the inner level optimisation as it is only a function of the architecture parameters, α, as shown in Equation 3.8, whereas the optimisation is with respect to the network parameters, w. On the other hand, the cross-entropy loss landscape is non-convex and difficult to optimise due to the multi-layer structure and complex mapping of neural networks, which leads to local minima in addition to saddle points (Petrulionytė et al., 2024). The difficulty of this problem is exacerbated due to the optimal weights, w^* being a function of the architecture weights, α, which makes it sensitive to the outer

level optimisation. In the outer level optimisation problem, the performance loss is minimised in which the cross-entropy loss is again non-convex. Similarly, the compute cost utilises the softmax operation on the architecture weights followed by a weighted average on the softmax due to which the convexity of the compute cost would not be guaranteed. As such, the non-convex nature of both the inner and outer objective, makes this a difficult optimisation problem.

Another challenge, in the optimisation problem is that the inner level problem is computationally expensive as it requires training the architecture encoded by α till convergence for every update to the architecture weights. As such, computing the architecture gradients would also be prohibitively expensive when using gradient based optimisation. To deal with this issue, the authors of DARTS proposed a simple approximation scheme shown below

$$\nabla_\alpha \mathcal{L}_{per,val}(w^*(w,\alpha),\alpha) \tag{3.16}$$

$$\approx \nabla_\alpha \mathcal{L}_{per,val}(w - \xi \nabla_w \mathcal{L}_{per,train}(w,\alpha),\alpha) \tag{3.17}$$

$$= \nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha), \quad w' = w - \xi \nabla_w \mathcal{L}_{per,train}(w,\alpha) \tag{3.18}$$

where w are the current network weights and ξ is the learning rate for the inner optimisation. The main idea in this approach is to approximate $w^*(w,\alpha)$ by the weights $w'(w,\alpha)$ obtained after taking a single step in the inner optimisation. This reduced the training time significantly, as every step in the architecture space did not require training the candidate architecture till convergence. The authors further used an approximation scheme to derive the architecture gradients which was compatible with our multi-objective metric. This approximation scheme is detailed in Section B.1 in Appendix B.

We note that Liu et al. (2018) or subsequent works in differentiable architecture search do not provide any convergence guarantees for the algorithm, however, in practice the authors found that the algorithm was able to find a fixed point with a suitable choice of ξ, the learning rate for a step of the inner optimisation shown in Equation 3.18. While, there are some works which provide convergence guarantees for bi-level optimisation problems, these assume the inner level objective to be convex which is not true in our case where neural network parameters are optimised in the inner level (Liu et al., 2021). In our experiments, we utilised the same value of the inner-optimisation learning rate as proposed by Liu et al. (2018) and found that the algorithm converged with our multi-ojective formulation.

The training algorithm used for architecture search is shown below in Algorithm 1 where the architecture weights, α and the network weights, w are alternately updated till the search has converged to an architecture. After convergence, the network in the

continuous search space was discretised as explained previously to obtain the discrete architecture.

Algorithm 1 Differentiable Efficient Architecture Search Algorithm

Create mixed operations, m_k parameterised by weights $\alpha_{i,j}$ for cells in network
while not converged **do**
1. Update architecture weights, α by descending $\nabla_\alpha \mathcal{L}_{per,val}(w - \xi \nabla_w \mathcal{L}_{per,train}(w, \alpha), \alpha)$
2. Update network weights, w by descending $\nabla_w \mathcal{L}_{per,train}(w, \alpha)$
end while
Derive final architecture based on the learned α

We note that in our analysis and experiments we utilised the performance loss composed of the cross-entropy loss and the compute cost for training the architecture and model weights, α and w respectively. The cross entropy loss was used as one of the sub-objectives as it is the most common loss function utilised for CNNs on classification tasks. The other sub-objective of the compute cost was used to characterise the complexity of candidate architectures. Utilising a different loss function in our formulation would also be possible with the only requirement being that the loss function be differentiable. We note that for any differentiable loss function the analysis performed previously in this section would hold including the multi-objective formulation and the approximate architecture gradients derived in Appendix B. In practice, we would likely find that some of the hyper-parameters would need to be changed for the search to converge such as the modulation parameter, β and the outer and inner optimisation learning rates, λ and ξ. However, the same is true for any general gradient based training recipe, and the hyper-parameters could be set with methods similar to those discussed in the previous sections.

3.2 Results

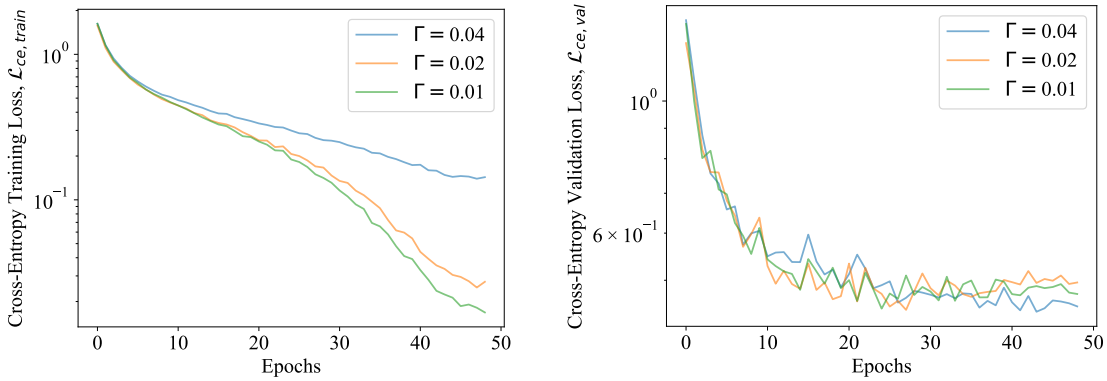
In this section we detail the experimental setup and results obtained when searching for and evaluating efficient CNN architectures on the CIFAR-10 classification task. To make our results comparable to the original DARTS approach we use the same experimental setup for search and evaluation, except for changing of the loss function to the multi-objective performance loss, and the selection of the associated cost weightage and modulation parameters, Γ and β.

3.2.1 Architecture Search

In the search process, the set of candidate operations, $\mathcal{F}$ was kept the same as the original DARTS paper such that $\mathcal{F}$: 3×3 and 5×5 separable and diluted separable

convolutions, 3×3 max-pooling, 3×3 average-pooling and skip-connections. The compute cost of the candidate operations was obtained as detailed in Section 3.1.1. We searched for cells on the CIFAR-10 (Hinton, 2007) dataset with networks created by stacking together 8 4-step cells. In the network, the 2nd and 5th cells were reductions cells with the rest being normal cells. The modulation parameter, β was set to a value of 0.27 by a logarithmic grid search process guided by the reference value worked out as demonstrated in Section 3.1.2.2. The architecture search was conducted for 50 epochs. A complete list of hyper-parameters is shared in Section B.2 in Appendix B.

Figure 3.7 shows the training and validation cross-entropy loss for the architecture search. For all values of the cost weightage, Γ, it can be observed that the training loss continues to decrease as expected. On the other hand the validation loss stops decreasing towards the end. This would indicate that the chosen 50 epochs of training was a reasonable amount of training without overfitting to the sub-objective.

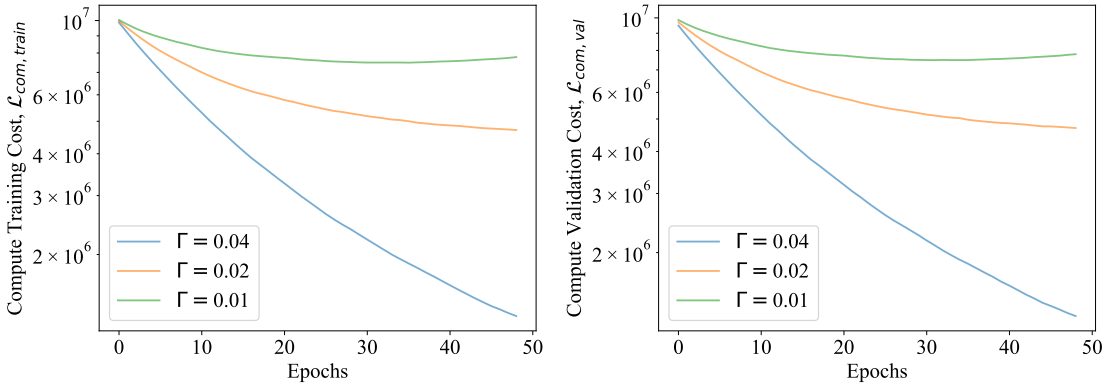


(A) Cross-Entropy Train Loss, $\mathcal{L}_{ce,train}$ (B) Cross-Entropy Validation Loss, $\mathcal{L}_{ce,val}$

FIGURE 3.7: Cross-Entropy Training and Validation Loss

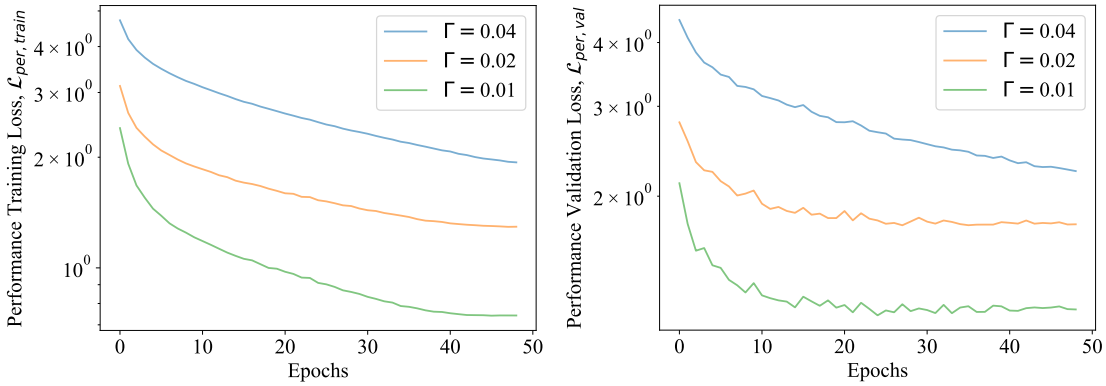
The compute cost through the training is shown in Figure 3.8. Similar to the cross-entropy this can also be seen to be generally decreasing. By comparing Figure 3.7 and Figure 3.8 both the sub-objectives are decreasing at the same time indicating that we were able to strike a reasonable balance between the sub-objectives without any one dominating the other. Further, it can be seen that when the cost weightage parameter, Γ is very low at 0.01, towards the latter part of training, the compute cost actually starts increasing. It should be noted that since by formulation the operation cost is a monotonically increasing function, the only reason this would occur was if the gradients back-propagated from the cross-entropy loss were greater than the gradients from the compute cost which would again indicate a compatibility between the sub-objective landscapes.

Figure 3.9 shows the performance validation and training loss across the search process. Note that due to the different values of the cost weightage parameter used, the landscapes are on a different scale so they are not directly comparable. However, we



(A) Compute Training Cost, $\mathcal{L}_{com,train}$ (B) Compute Validation Cost, $\mathcal{L}_{com,val}$

FIGURE 3.8: Compute Training and Validation Cost

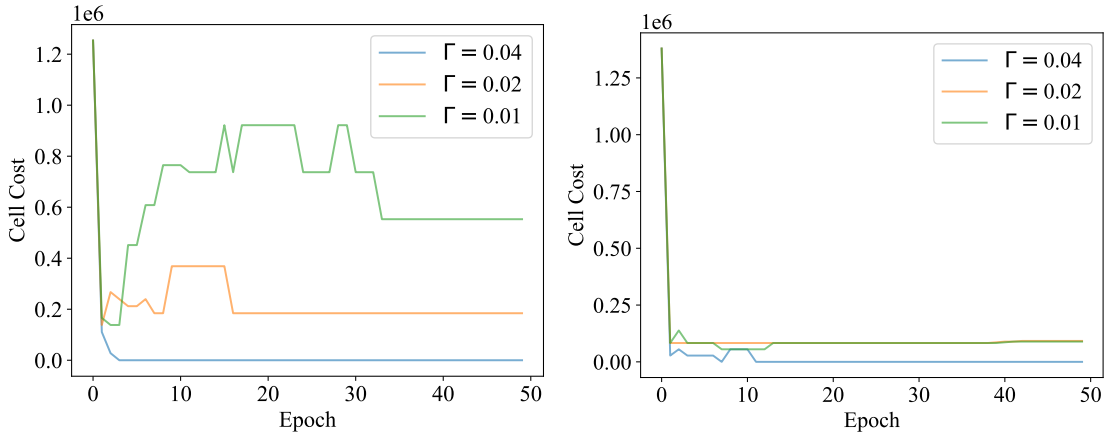


(A) Performance Training Loss, $\mathcal{L}_{per,train}$ (B) Performance Validation Loss, $\mathcal{L}_{per,val}$

FIGURE 3.9: Operation Training and Validation Loss

make some interesting observations by comparing them to their corresponding cross-entropy loss and compute cost plots. For $\Gamma = 0.01$, the training and validation performance loss can be seen to have stabilised towards the end of training. However, if we compare around epoch 40 of the performance loss and the sub-objectives for $\Gamma = 0.01$, the training operation cost can be seen to be increasing while the training cross-entropy loss continues to decrease. This would indicate that the optimisation process was able to locate the pareto frontier and had begun running through all pareto efficient solutions where the performance loss was minimised. In this scenario, pareto optimality would be defined as the situation in which no sub-objective could be minimised further without increasing the other sub-objective. This is precisely the phenomenon observed during the latter part of the search process for $\Gamma = 0.01$. It can be seen for the rest of the values for Γ that all three training losses are decreasing. This would indicate that the training fell short of finding a pareto optimal outcome in these cases. However, it is important to note that these solutions fell short of pareto efficiency in the continuous search space. It would not necessarily hold true that a pareto inefficient outcome in the continuous search space would produce a pareto inefficient outcome in the discretised space and vice versa.

We examine this further by looking at the compute cost of the final cells derived by the discretisation step. This is shown in Figure 3.10 where the compute cost was calculated by simply adding the cost of the selected candidate operations. For $\Gamma = 0.01$, when the pareto frontier is found at around epoch 45 corresponding to the 350th step, it can be seen that the discretised architecture remains the same. This would show that the different continuous architectures had converged to a discrete architecture.



(A) Normal Cells (B) Reduction Cells

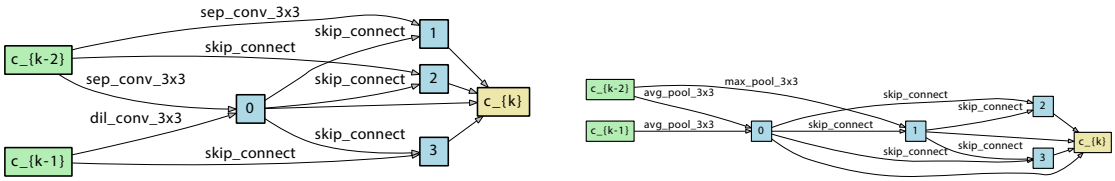
FIGURE 3.10: Discretised Reduction and Normal Cell Costs

Another simple trend observed in Figure 3.10 is that higher values of the cost weightage parameters derived cells with lower computational complexity and vice versa. This is what we would expect and would indicate that the compute cost correctly characterised complexity of the cells such that gradient descent was able to navigate the space and find efficient cells. An interesting observation to make is the difference between the cost of the reduction and normal cells where the latter is always higher. We discuss this further in the next section where we show the derived cell architectures.

3.2.2 Derived Cells

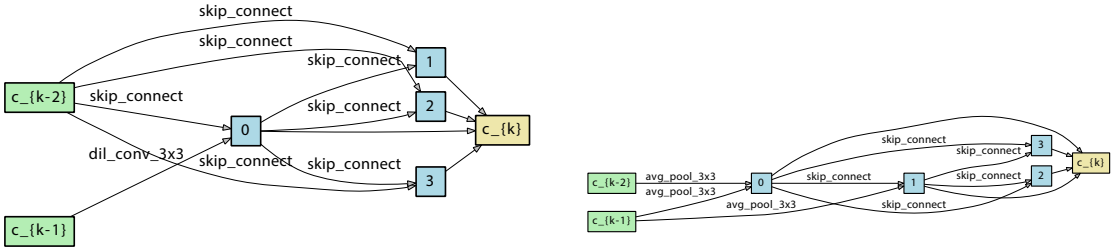
The cells derived for the different values of the cost weightage after 50 epochs of search are shown in Figures 3.13 to 3.11. As we observed in the previous section, the complexity of the cells increases as we decrease the cost weightage parameter. It can be seen that with a higher cost weightage of $\Gamma = 0.04$, the derived cells have only skip connections. On the other hand, with $\Gamma = 0.02$ the cells grew in complexity with a 3×3 dilated convolution (cheapest of the available convolution operations) and pooling operations. Further decreasing the cost weightage to $\Gamma = 0.01$ produced architectures with expensive operations such as 3×3 separable convolutions in addition to average, max pooling operations.

The observation made in the previous section regarding the different operation costs of the reduction cells and normal cells can also be explained by examining the internal



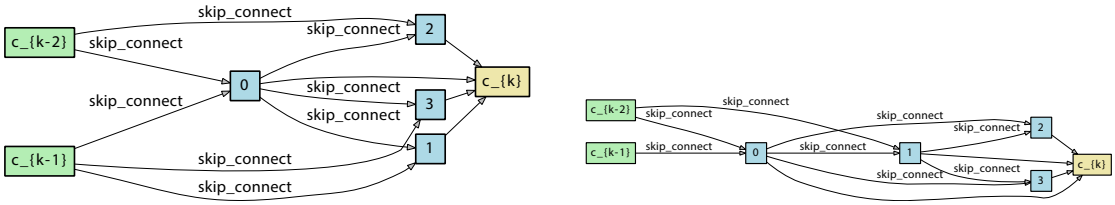
(A) Normal Cell (B) Reduction Cell

FIGURE 3.11: DEff-ARTS Derived Cells, $\Gamma = 0.01$, $\mathcal{L}_{com} = 1,010,688$



(A) Normal Cell (B) Reduction Cell

FIGURE 3.12: DEff-ARTS Derived Cells, $\Gamma = 0.02$, $\mathcal{L}_{com} = 276,480$



(A) Normal Cell (B) Reduction Cell

FIGURE 3.13: DEff-ARTS Derived Cells, $\Gamma = 0.04$, $\mathcal{L}_{com} = 0$

structure of the derived cells. A clear trend can be seen that the cheaper average and max pooling operations only ever appear in reduction cells. We consider explainability of derived architectures to be out of the scope of this work and do not attempt to explain the derived architectures any further due the general problem of interpretability in the field of differential programming and deep learning. However, in the next section we comment on the possibility of the presence of some unintended bias by comparing the derived cells with other approaches.

3.2.3 Architecture Evaluation

The derived architectures were evaluated on the CIFAR-10 dataset. For evaluation, networks were created with a depth of 20 cells, where the 6th and 13th cells were reduction cells and the rest were normal cells. For a network composed of 20 4-step cells, the network would have 120 layers of operations if we included the pre-processing within a cell. This would lead to the vanishing gradient problem. To deal with this issue, auxiliary towers (Ntziachristos, 2010) were used on the output of the 13th cell with a weight of 0.4. Additionally, cut-out (DeVries and Taylor, 2017) was used in the

Architecture	Test Error (%)	Params (M)	Search Cost (GPU days)	#ops	Search Method	GMACs	Compute Cost
NASNet-A + cutout	**2.83**	3.1	2000	13	RL	0.624	5,861,376
AmoebaNet-A + cutout	**3.12**	3.1	3150	19	evolution	0.506	6,100,992
DARTS + cutout	**2.76 ± 0.09**	3.3	**4**	7	gradient-based	0.547	1,244,160
DEff-ARTS + cutout, $\Gamma = 0.01$	**3.24 ± 0.26**	**2.3**	**4**	7	gradient-based	**0.376**	**642,048**
DEff-ARTS + cutout, $\Gamma = 0.02$	**4.42 ± 0.07**	**1.6**	**4**	7	gradient-based	**0.262**	**276,480**
DEff-ARTS + cutout, $\Gamma = 0.04$	16.01 ± 0.41	**1.45**	**4**	7	gradient-based	**0.242**	**0**

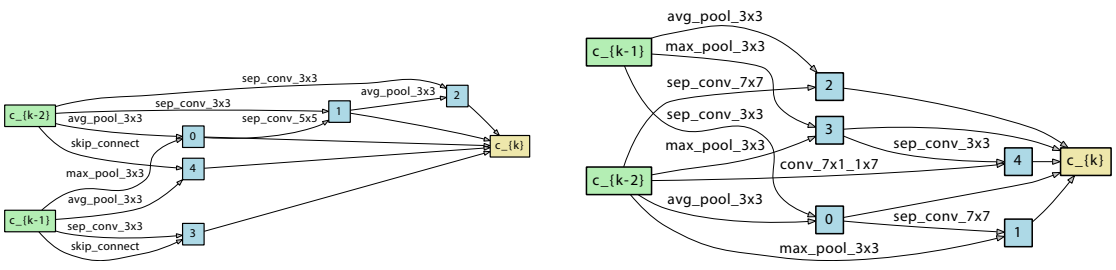
TABLE 3.4: Comparison with state-of-the-art image classifiers on CIFAR-10. All values other than DEff-ARTS are taken from Liu et al. (2019)

search process to produce better generalisation performance following Liu et al. (2018). All networks were trained for 600 epochs. Training time varied between 11 to 21 hours depending on the complexity of the cells composing the network. The complete set of hyper-parameters used for training is provided in Section B.3 in Appendix B.

Table 3.4 compares the results obtained from networks derived by DEff-ARTS with other approaches in the literature. Note that in addition to measuring the computational complexity through the compute cost, we compare the Multiply and Accumulate (MAC) operations between different network architectures as well. This is because the compute cost would only be indicative of the mixed operations in a cell and not the hard-coded pre-processing blocks. While it would be better at characterising low-level operation differences (e.g multiply, addition, comparison) between candidate operations, it would not give a complete picture of the networks performance.

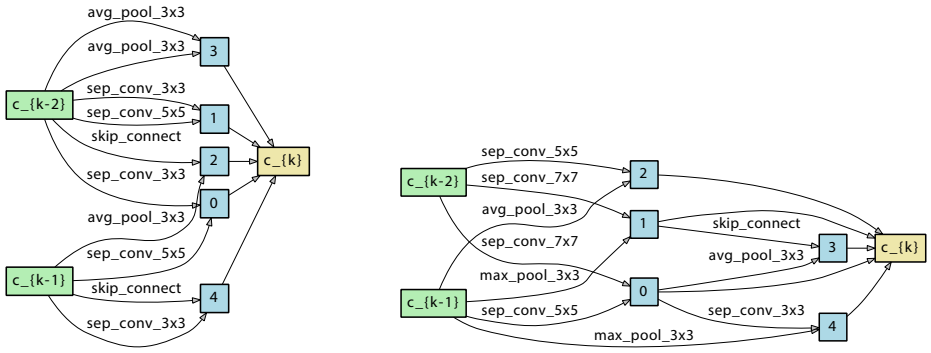
It can be observed in Table 3.4 that for $\Gamma = 0.01$ and 0.02 the networks achieved results comparable to the state-of-the-art architectures with model sizes $30 - 50\%$ smaller and a lower number of required Multiply-Accumulate (MAC) operations. It is also worth pointing out that for $\Gamma = 0.04$, the network achieved an accuracy of 83.99% . This was due to the Conv-BatchNorm-ReLU pre-processing blocks in every cell which would contribute to the networks representational capacity.

Comparing the search cost between the different networks, we can see that the differentiable approach of DARTS and DEff-ARTS was three orders of magnitude faster than the reinforcement learning or evolutionary approaches of NASNet or AmoebaNet.



(A) Normal Cells (B) Reduction Cells

FIGURE 3.14: AmoebaNet Cells



(A) Normal Cells (B) Reduction Cells

FIGURE 3.15: NASNet Cells

The cells derived by AmoebaNet and NASNet are shown in Figure 3.14 and 3.15. Regarding the observation that the DARTS and DEff-ARTS derived cells only had average or max pooling operations in the reduction cells, it can be observed that this was not the case for the AmoebaNet or NASNet cells. For the DEff-ARTS approach, it is possible that the search balanced the accuracy and complexity by reducing the complexity of the two reduction cells in the network and retaining more expressive operations in the normal cells. However, the same behaviour was also observed in the baseline DARTS architecture which had no explicit optimisation for the complexity of the cells. This could point to some unintended bias in the differentiable search method, but we leave further examination of this as a future work.

3.3 Discussion

Neural Architecture Search (NAS) has been shown to be a promising approach to derive models in an automated manner without the need for human intervention in the process. However, prior approaches utilised the framework of NAS to derive architectures with the highest accuracy. To meet varying complexity constraints, the derived model was typically simplified using manual heuristics.

We proposed Differentiable Efficient Architecture Search (DEff-ARTS), to derive architectures of varying complexity in an automated manner. We utilised the machinery of efficient gradient based NAS and converted the problem to a multi-objective optimisation. The multi-objective optimisation metric, or performance loss, included a differentiable metric for the complexity of the model, the compute cost, in addition to the cross-entropy loss measuring the predictive quality of the model. The trade-off between the quality and efficiency of the derived model was controlled through a single hyper-parameter configured by the user. We successfully navigated the optimisation landscape to derive models directly from the search space. The models demonstrated up to 1.4x-2.1x lower complexity and parameters without any addition to the search

cost of deriving the architectures. We believe further increases in performance could be achieved through better design of the search space. This could be carried out through inclusion of further candidate operations and structural modifications to the search space. The approach of multi-objective optimisation we adopted is now widely found in recent NAS based works, although it is often performed with different optimised search spaces. In the following chapters, we utilised insights from these approaches to develop our experimental setup and improve upon the work of this chapter.

Chapter 4

TinyOps: A New Model Design Space for MCUs

The previous chapter explored how we could design models to explore the accuracy-complexity trade-off. However, in deployment the inference framework is of equal importance as it dictates the design space of models that can be deployed and the achievable performance.

A number of inference frameworks have been designed for edge devices which focus on minimising the footprint of the code whilst offering portability and flexibility. However, as we discussed in Section 2.2.1, the majority of prior works in TinyML do not consider a memory hierarchy and consider only internal storage and memory in the deployment scenario. This poses a challenge for model design as state-of-the-art CNNs have a memory and storage requirement larger than the amount available on low-power edge devices which is typically less than 512KB of memory and 2MB of storage as can be observed in Table 2.1. Within the constraints of internal memory, prior works have achieved incremental progress through graph optimisations (Liberis and Lane, 2020), light-weight code-generation (Lin et al., 2020, 2021) or sub-byte quantisation (Rusci et al., 2018, 2020a), but the constraint on model size has still left a sizeable gap between the performance achievable on the low-power platforms. As an example, the state-of-the-art MCUNetV2 model derived for deployment with 256KB of memory and 1MB of storage achieves 64.9% top-1 accuracy (Lin et al., 2021) compared to a ProxylessNAS model derived for mobile devices with 74.6% top-1 accuracy (Cai et al., 2018).

We explore how we can efficiently lift the ceiling of performance achievable on low-power edge devices. While prior works have focused on internal memory, we investigate how we can utilise unused peripherals of the devices to achieve further performance gains. As we discussed previously in Section 2.4.1, these devices have peripherals including DMA and external memory interfaces such as the FMC and QSPI on

MCUs which can be used to supplement the limited internal memory and storage as shown in Figure 4.1. In this chapter, we seek to answer the research question, **RQ2**: *how can inference software efficiently utilise external memories to enable design and deployment of efficient models?*

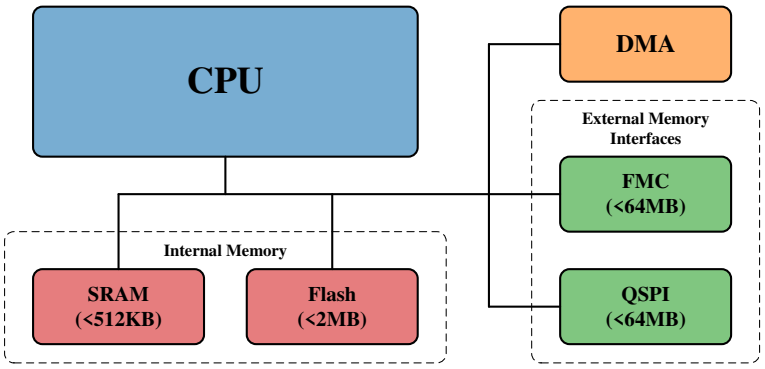


FIGURE 4.1: Microcontroller (MCU) Architecture Block Diagram

We answer this question by analysing how current inference frameworks can be integrated with external memories. For this exploration, we utilised MCU platforms by STMicroElectronics based on the ARM Cortex-M architectures as these devices had a mature and open-source software stack that made investigation easier and also lead to these to be used as a widely used platform for inference at $< 1W$ (Lin et al., 2020; Banbury et al., 2021; Liberis and Lane, 2020; Rusci et al., 2020a; Fedorov et al., 2019). However, we note that the generic model of the target device we adopt in Figure 4.1 would be applicable to other low-power devices as well. In our study of current inference frameworks, which is detailed in Section 2.4.2.1, we showed that current inference frameworks had a limitation where they placed all weight and activation tensors in the same memory segment, which is mapped to either internal or external memory. We show that this low resolution placement strategy is unable to achieve acceptable accuracy or latency as it does not utilise the memory hierarchy. While internal storage and memory is fast, its limited capacity imposes constraints on model size resulting in models having low accuracy. We show that the size constraints can be alleviated, and acceptable accuracy can be achieved by using external alternatives which have enough capacity to hold state-of-the-art networks weights and activation tensors. However, this approach to using external memories suffers from low latency due to the slower access latency of external memories.

We propose the TinyOps inference frameworks to combine the advantages of speed and size of internal and external memories. TinyOps uses external memories as main memory and overlays data for an operation in the inference graph in internal memory when it is to be performed. To meet internal memory constraints we utilise a partitioning

scheme which partitions operations in the inference graph into a number of independent *tiny operations* which process smaller independent blocks of the input tensor. To mitigate the access latency of external memories, the data copying is overlapped with data processing. This is accomplished by offloading the data movement to the DMA and employing a double buffering strategy which allows the DMA to copy data in parallel with the CPU. This approach opens a new design space for deploying models from the larger external memory design space with fast internal memory like inference latency.

The work presented in this chapter addresses the research question to make the following contributions

- **TinyOps Inference Framework:** We propose the TinyOps inference framework that opens a new design space for MCUs. We adopt a holistic view of MCU architectures applicable to a range of MCUs to show how external memories can be effectively used in combination with the DMA to perform inference from external memory with fast internal memory like latency.
- **Experimental Evaluation:** We perform experiments to evaluate the performance of the TinyOps inference framework with state-of-the-art CNN models on the ImageNet dataset (Deng et al., 2009) demonstrating that it is able to reduce inference latency by up to 1.4-2.5x. Experiments are performed across three different MCU platforms to demonstrate the portability of TinyOps.
- **Open-Source Software:** The TinyOps inference framework is ported to three commercial off-the-shelf platforms and integrated with the low-level drivers of the platform with code available at `https://github.com/sulaimansadiq/TinyOps`.

4.1 Internal vs External Memory

To analyse the performance of external memory and how it can be used in DNN inference on MCUs we studied how external memories could be utilised with current inference frameworks. As our baseline, we utilised the TensorflowLite-Micro (TFLM) inference framework and CMSIS-NN kernels due to their open-source nature, flexibility and portability between devices.

As discussed in Section 2.4.2.1, a limitation of inference frameworks on MCUs, including TFLM, was that they did not support the use of a memory hierarchy. Looking at this from an implementation lens, this was due to the usage of default compiler sections and memory placement strategies. This strategy was used to place objects required for inference including the inference framework code, the constant weight tensors and the

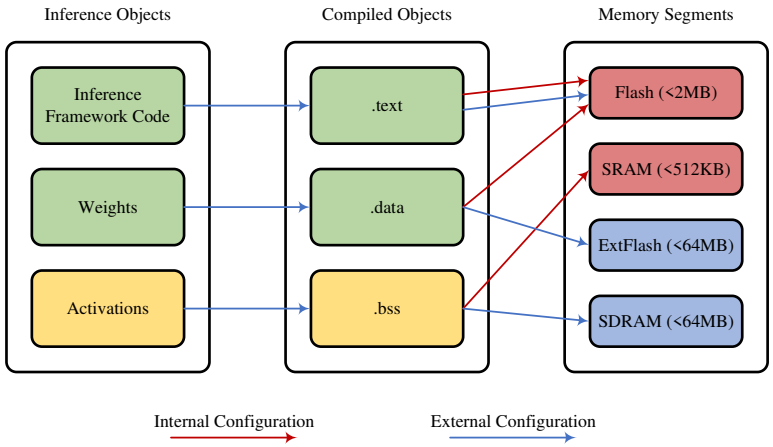


FIGURE 4.2: Inference Frameworks Memory Configurations

activation tensors. By default, these were compiled into the .text, .data and .bss compiler sections respectively.

Internal Memory Configuration With the default memory placement strategy, the compiled sections were placed in volatile and non-volatile memory segments which mapped to internal SRAM and NOR Flash as shown in Figure 4.2. This configuration was utilised by the majority of prior works which we refer to this as the internal memory configuration for deployment.

With this flat memory hierarchy, the operations were executed as shown in Figure 4.3 with the procedure to perform inference shown in Algorithm 2. As can be observed the procedure simply consists of sequentially performing the operations in the inference graph. The operations are performed by simply getting references to the input, output, weight tensors (if applicable), quantisation parameters in addition to the low-level kernel required to perform the operation. In the internal memory configuration, all the weights reside in internal Flash with activation tensors allocated memory in internal SRAM. As can be observed in Figure 4.3, this approach materialises the entire tensor in internal memory for processing.

We studied the storage and memory requirement of state-of-the-art CNN architectures derived for MCU and mobile devices which are shown in Table 4.1. Statistics for the MCUNet family of models are as reported by Lin et al. (2020, 2021), while the statistics for the remaining models derived for mobiles were generated by deploying the models using TFLM. The models were trained on the ImageNet dataset. A complete set of hyper-parameters is given in Section C.3 in Appendix C. As can be observed, the MCUNet family of models derived for internal storage and memory have lower accuracy due to the constraint on the number of weights in the model and the size of the

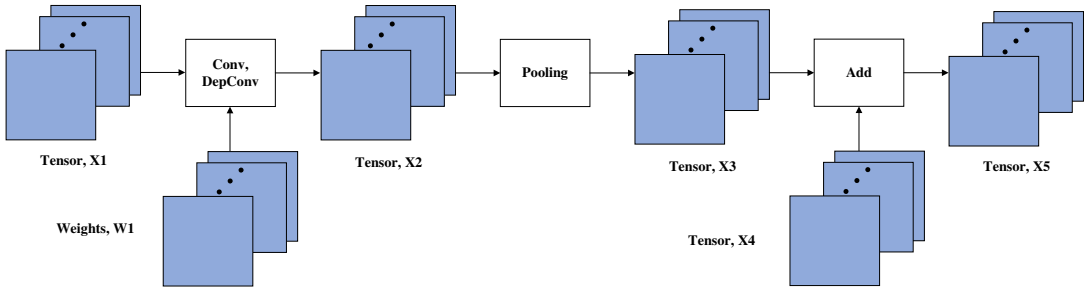


FIGURE 4.3: Traditional Operation Execution

Algorithm 2 Vanilla Inference Pipeline

Inputs: NN_graph, the inference graph. $input$, the tensor input to the NN
Output: $output$, the tensor output from the NN

```
function InvokeVanilla(NN_graph, input)
    for op in NN_Graph do
        inputs = GetInputTensors(op)
        weights = GetWeightTensors(op)
        quantParams = GetQuantisationParameters(op)
        Kernel = GetKernel(inputs, weights, quantParams)
        output = Kernel(op)
    end for
     return output
end function
```

Model	MACs	RAM	Flash	Acc
MCUNetV1-F469	67M	242KB	878KB	60.3%
MCUNetV2-F469	119M	196KB	1010KB	64.9%
MCUNetV1-F746	82M	293KB	897KB	61.8%
MNASNet	314M	1118KB	4.79MB	75.2%
ProxylessNAS	320M	1095KB	4.47MB	74.6%
MobileNetV3	215M	1232KB	4.26MB	72.3%

TABLE 4.1: Memory and storage requirements for state of the art models developed for microcontroller and mobile deployment scenarios

activation tensors. The mobile models on the other hand, have a much higher memory and storage requirement.

Comparing with the amount of internal storage and memory available on MCU platforms, we found that even high-end MCUs, such as the STM32H7 series by STMicroElectronics which host up to 512KB of SRAM and 2048KB of Flash, are unable to meet the requirements of the state-of-the-art mobile models.

TABLE 4.2: Cortex-M based off-the-shelf platforms used with varying specifications and constraints.

Platform	Architecture	Core	Clock (MHz)	Internal			External	
				D-Cache (KB)	SRAM (KB)	Flash (KB)	FMC (KB)	O/QSPI (KB)
L552	ARMv8	M33	110	✗	192	512	2048	8192
F469	ARMv7	M4	180	✗	256	1024	8192	8192
F746	ARMv7	M7	216	4KB	320	1024	8192	8192

External Memory Configuration To alleviate the constraint on model size, we propose using external memories. As shown in Figure 4.1 and discussed previously in Section 2.4.1, the MCUs, feature external memory interfaces, which can be used to extend the limited internal storage and memory address space with sizeable external memory and storage.

We utilised off-the-shelf MCU development kits by STMicroelectronics which supplemented the internal storage and memory as in Table 4.2. On the F469 and F746, the internal memory and storage address space were extended with SDRAM and NOR Flash on the FMC and QSPI interfaces respectively. On the L552 development kit, a slightly different memory configuration was used due to limitations with available off-the-shelf development kits. On this platform, the internal memory address space was extended with SRAM and HyperRAM on the FMC and OSPI interface with storage being supplemented with a 32GB SD Card.

To benchmark the performance of external memories with the existing TFLM framework, we configured the inference framework to perform inference with an external memory configuration as shown in Figure 4.2. This was accomplished by configuring the low-level drivers for the FMC, O/QSPI and SDMMC interfaces on the MCU and modifying the linker command file to place the .data and .bss objects containing the weight and activation tensors into the external storage and memory segments. On the L552, the external SD Card providing non-volatile storage for the model weights was not memory mapped due to which the inference framework could not directly access this space during inference. This issue was dealt with by using the SD Card as the load address and copying the weights from the SD Card to memory mapped external memory on the OSPI interface on device startup.

4.1.1 Performance Evaluation

We evaluated the performance of the external memory configuration by comparing it to the internal memory configuration. To decouple the effect of the model on inference latency, we deployed the same model derived for internal memory using either configuration on the platforms. The internal memory model was derived by using the conventional approach of scaling.

Model	Width	Resolution (pixels)	Flash (KB)	SRAM (KB)
ProxylessNAS	1.00	224	4579	1095
	1.00	208	4579	987
	0.90	224	3980	1072
	0.90	208	3980	964

TABLE 4.3: Effect of Scaling Width and Resolution on Memory Requirement

We note that we moved from models designed from the cell based search space of DARTS (Liu et al., 2018) to models derived from the mobile search space (Tan et al., 2019; Howard et al., 2019; Cai et al., 2020) as these were shown to be superior in performance to DARTS. Further we adopted the approach of searching for the width and resolution of models as this provided better performance than searching for candidate operations in the cell based search space. A performance comparison of the two approaches can be found in Section C.1 in Appendix C.

Scaling for Reduced Memory and Storage The scaling of the width and resolution is frequently used to reduce the storage and memory requirement of deploying a model. The width hyper-parameter is scaled by reducing the number of channels in the backbone model while the resolution is scaled by reducing the spatial pixels of the input resolution. The memory requirement in deployment is a function of the width and resolution, while the storage requirement is a function of the width only. As shown in Table 4.3, reducing the input resolution from 224 to 208 has no effect on the non-volatile memory requirement but reduces the required amount of volatile memory as the spatial dimension of the intermediate representations decreases throughout the network. On the other hand, scaling down the width from 1.00 to 0.90 reduces the required non-volatile memory from 4579KB to 3980KB as the filters in the Conv and DepConv layer have less channels which reduces the number of parameters in the model. Additionally, it also reduces the volatile memory usage as it results in intermediate tensors with a smaller channel dimension and therefore overall smaller tensor sizes.

Through the remainder of this thesis, we refer to a ProxylessNAS/MNASNet/MobileNetV3 base model scaled down to any width or resolution e.g. width of 0.35 and resolution of 192 as ProxylessNAS/MNASNet/MobileNetV3-w0.35-r192.

Latency and Power Using the conventional approach of scaling, we derived scaled variations of a ProxylessNAS model which achieved maximum accuracy according to the diverse internal storage and memory constraints of the platforms in Table 4.2. These models were derived to be Proxyless-w0.10-r192, Proxyless-w0.30-r144 and Proxyless-w0.30-r176 for the L552, F469 and F746 respectively. The inference latency and power

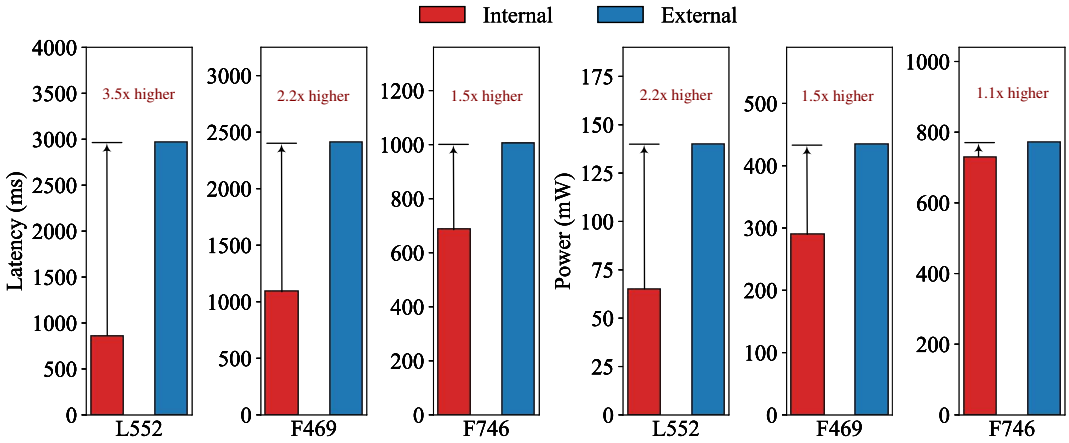


FIGURE 4.4: Performance Comparison of Internal and External Memory

consumption of deploying the models using either internal or external memory configuration on the platforms is shown in Figure 4.4. As can be observed, the inference latency of the external memory configuration was 1.5-3.5x slower than the internal memory configuration whilst the power consumption was 1.1x-2.2x higher.

The variation in the difference of the latency and power between the configurations across the devices could be attributed to the presence of data cache on the F746, different models being deployed across devices or the types of memory being utilised in the hierarchy on the devices. In Chapter 5, we study the design of different models and their impact on inference latency. However, we consider a study of the effect of using different types of memory in the memory hierarchy on the inference latency outside the scope of this thesis. In the next section, we discuss the design of TinyOps and how we accelerate inference from external memory.

4.2 Design of TinyOps

In the previous section we compared the performance of performing inference from external or internal memory using the traditional computing paradigm. We highlighted the limitations of the design spaces where the internal memory design space achieved limited accuracy whilst the external memory design space suffered from high inference latency when using the prior inference frameworks. Our analysis of the limitations of both of these approaches leads to the question: *how can we mitigate the high inference latency of CNNs when using external memories?*

4.2.1 Insights

We propose the TinyOps framework that combines the advantage of external memory (size) and internal memory (speed). We make a number of insights considering the MCUs hardware and how the required computation is perfomed on MCUs to develop TinyOps

- Microcontrollers have a range of external memory interfaces and peripherals available across all platforms which are not utilised.
- External memories and storage are plentiful and sufficient to store state-of-the-art architecture's intermediate buffers and weights.
- The microcontrollers have a single CPU that, at any given time is processing only a small part of the data.
- Computation on different parts of a tensor can be divided such that the sub-computations are independent of each other.
- The DMA peripheral is able to operate in parallel with the CPU to enable data transfer to take place in the background.
- Data that is less frequently read or written to, can be off-loaded to external memory with negligible performance loss.

Combining these insights, we propose the TinyOps framework which utilises a partitioning and overlaying engine to accelerate inference from external memory. We declare and initialise all model data in external memory and propose the usage of overlaying techniques to move data into internal memory when it is required for processing. However, intermediate tensors can get quite large and it would not be possible to hold the entirety of the input and output tensor in internal memory. To meet the internal memory constraints, a partitioning strategy is utilised to divide the operations in the inference graph into multiple tiny operations with reduced memory requirement allowing their data to be held in internal memory. To effectively hide the latency of fetching data from external memory, the overlaying scheme offloads the data movement to the DMA and employs a double-buffering strategy to overlap the data movement performed by the DMA with the processing performed by the CPU.

4.2.2 Seamless Integration

TinyOps integrates seamlessly into existing inference frameworks as shown in Figure 4.5. We utilised TensorflowLite-Micro as a baseline on which to build TinyOps due to its open-source nature and widespread use. In the default mode of operation,

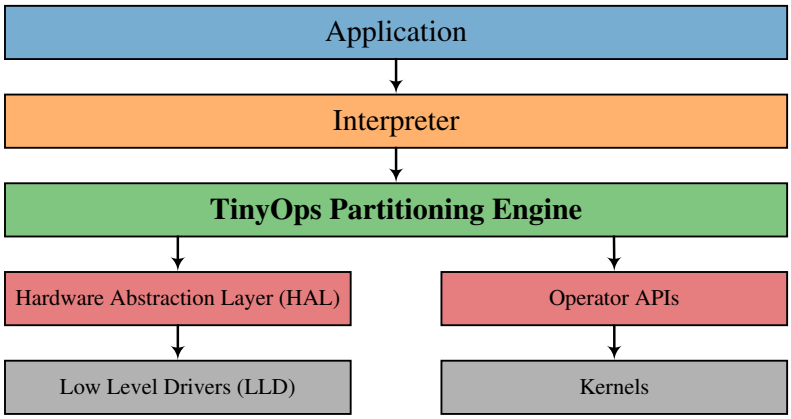


FIGURE 4.5: TinyOps Overview

the interpreter performs inference by sequentially running through operations in the inference graph, and making calls to the low-level kernel associated with the operation through the APIs. TinyOps partitioning and overlaying engine sits between the interpreter and the low-level kernels and drivers as shown in Figure 4.5. Whenever the interpreter makes a call to the low-level kernel, TinyOps catches these calls and executes the operation by interfacing with the low-level kernels and DMA drivers to perform the data movement and processing on tiny tensors in a seamless manner.

4.2.3 Operation Categorisation

To accelerate the inference of an NN from external memory would require accelerating each of the operations that compose the NN. For the purpose of developing the TinyOps partitioning engine we categorise the operations utilised in state-of-the-art CNNs into three categories as below to analyse what data they require in performing an operation and what parts of the network to focus on.

- **One-Input Parameterised Operations** In state-of-the-art CNNs, Pointwise Convolutions, Grouped Convolutions fall into this category where there is one input activation tensor and a set of weight and bias tensors which are inputs to the low-level convolution or matrix-multiply kernels to produce an output tensor. Additionally, the quantisation scheme produces multiplier and shift parameters which are used in the quantisation arithmetic to produce the output tensor. Convolution operations in this category also require a partial im2col buffer. We note this as the most important category as $>$ 80% of operations in state-of-the-art CNNs fall into this category.

- **Two-Input Non-Parameterised Operations** Add operations generated by skip connections in the network fall into this category. These operations have two

input tensors which are fed to the low-level kernels to produce an output tensor. They have scalar multiplier and shift quantisation parameters.

- **One-Input Non-Parameterised Operations** This category includes the Average Pooling operation. In state-of-the-art CNNs, there is only one average pooling operation at the end of the network.

We note that Batch Normalisation and Activation Function (ReLU6) layers extensively utilised in state-of-the-art models also fall in the category of one-input non-parameterised operations, however as these are fused with Convolution or Linear layers they did not need to be treated separately. We note that Linear of Fully Connected layers were implemented and treated as pointwise convolutions with unit spatial resolution.

4.2.4 Overlaying Strategy

The internal memory configuration adopted by inference frameworks held all the data including tensors, quantisation parameters, data structures utilised by the run-time in internal memory. The limitation in this case, however, is that internal memory is not sufficient to hold all the data for an operation. In our approach we reduce the memory requirement by utilising internal memory only for frequently accessed data. This included input, weight and bias tensors, the partial im2col buffer in addition to quantisation parameters.

Frequently Accessed Data We did not overlay output tensors, data structures used to manage tensors or other meta-data in the inference graph. The decision to not overlay output tensors was based on the fact that the output tensors are accessed sparsely compared to the input tensor in convolution operations which compose the bulk of the CNN. We demonstrate this analytically by comparing the number of read and write operations required to produce one output element in pointwise and grouped convolutions.

For pointwise convolutions the number of read operations to produce a single output element would be C or the number of input channels. The produced element would be written to memory with a single write operation. We found that in modern CNNs (EfficientNet, MobileNetV3, MNASNet, ProxylessNAS), the number of channels throughout the network was between 32 and 1152. Similarly, grouped convolutions would require K^2 read operations which would produce and write one output element, where K is the kernel size often with a value of 3, 5, 7. This would imply that the read operations in convolution operations, would be atleast an order of magnitude higher than the write overhead. However, as output tensors are typically similar in size to input tensors, the memory requirement for overlaying would be similar in both cases. As

such, we did not overlay output tensors as this would provide little performance gain with a high memory cost.

This approach alone however, was not sufficient in reducing the memory requirement. Specifically, we noted that input tensors in the network could grow quite large. For example, in a MobileNetV3 model the output of the fifth pointwise convolution was observed to be $112 \times 112 \times 64$. This would require $802,816$ bytes which would be infeasible on the MCU platforms with limited memory. To reduce this memory requirement, we utilised a partitioning scheme discussed in the next section that decomposed the input tensor into *tiny tensors* that could be accommodated in internal memory.

We note that the partitioning scheme was only applied to the input tensors. The memory requirement for bias tensors and quantisation parameters was found to be low at $4C_{max}$ bytes, where C_{max} was the maximum number of channels in the network and the constant factor of 4 accounted for the 32 bit precision used for the parameters. As mentioned previously, we found C_{max} to be 1152 across state-of-the-art CNNs which would lead to a peak memory requirement of $4,608$ bytes to overlay all of the bias and quantisation parameters.

Similarly, we found the size of the filters to range from between 288 bytes to $221,184$ bytes. The smaller filter sizes were found towards the beginning of the network, where the channel dimension of activation tensors is small whilst the spatial resolution is large. The larger filters were found in the latter part of the network where the channel dimension of activation tensors was large and spatial resolution was small. A large spatial resolution would imply a larger number of receptive fields which would require multiple reads of the filter weights, while a lower spatial resolution would require less reads of the weights. Based on this observation we only overlayed filters that could be accommodated in internal memory.

Another approach could be to partition the filters and break down the convolution into multiple convolutions. However, this would require modifying the underlying kernels which we opted to avoid, as kernel specific optimisations would limit portability of the framework. As we demonstrate later in Section 4.3, overlaying a subset of filters for larger models yielded satisfactory performance.

Data Fetching and Processing Overlapping We note that in performing inference in an operation, the input tensors had the largest memory requirement. Sequentially performing the data movement and processing of the input tensors would lead to high data fetching overhead. As such, we double buffered the overlaying of the tiny tensors which allowed overlapping of the data movement and processing using the DMA. The remaining data including the weight and bias tensors in addition to the quantisation parameters was not double buffered. This allowed for a network level pipeline to be established between the DMA and the CPU that we detail in Section 4.2.7.

Memory Requirement In total, our overlaying strategy required seven fast buffers to be realised in internal memory. Four buffers were utilised to buffer the tiny input tensors of two-input add operations, with three being utilised for the bias tensor and multiplier and shift quantisation parameters. To overlay the filters we reused one of the four tiny input tensors which would be otherwise unused in one-input parametric operations. Similarly, the second of the otherwise unused buffers was used for the im2col buffer. To overlay as many filters as possible, any leftover internal memory was assigned to the buffer reused for filter overlaying.

Using this approach the buffer sizes were allocated as in Table 4.4 where H_t, W_t and C_t are the height, width and number of channels of the largest tiny tensor in the network, I_{max} is the size of the largest im2col buffer in the network, C_{max} is the largest number of channels in the network, M is the memory budget and $R = M - (3 \times B0 + B3 + B4 + B5 + B6)$ is the memory left in the budget after all buffers have been assigned. We detail how the sizes of the tiny tensors were determined in the next section.

TABLE 4.4: Sizes of Fast Buffers Used in TinyOps Overlaying Strategy

Buffer #	**Usage**	**Size (bytes)**
B0/B1	Tiny Tensors	$H_t \times W_t \times C_t$
B2	Tiny Tensors, Filters	$H_t \times W_t \times C_t + R$
B3	Tiny Tensors, im2col	$max(H_t \times W_t \times C_t, I_{max})$
B4	Bias Tensor	$4 \times C_{max}$
B5	Quantisation Parameters	$4 \times C_{max}$
B6	Quantisation Parameters	$4 \times C_{max}$

As can be observed, the size of B0 and B1 are determined from the largest tiny tensor in the network. Similarly, B4, B5 and B6 are allocated for the bias and quantisation parameters according to the largest number of channels in the model. As B3 is used for tiny tensors and the im2col buffer, its size is determined by the larger of the two sizes. Finally, as B2 is used for tiny tensors and filter overlaying, its size is selected to accomodate the largest tiny tensor in the worst case. However, in order to overlay the largest number of filters we assigned any memory remaining in the budget to this buffer.

4.2.5 Partitioning Scheme

To lower the memory requirement of operations in the inference graph, we partitioned the operations into *tiny operations* that processed smaller independent blocks or *tiny tensors* in the input tensor. We declared the input tensor in external memory and logically partitioned the input tensor into a number of tiny tensors that could be accommodated in fast internal memory buffers. In this section we detail the implementation of the partitioning scheme.

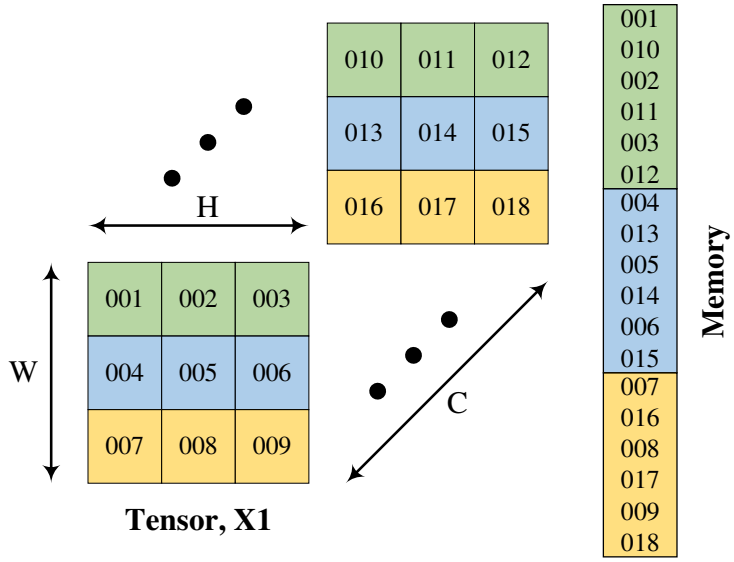


FIGURE 4.6: Partitioning of NHWC tensors along H dimension

4.2.5.1 NHWC Partitioning

We utilised TensorflowLite-Micro as a baseline which stored activation and weight tensors in the NHWC layout. We logically partitioned tensors in the H dimension due to the limitations of the on-board DMA which is only able to copy contiguous chunks of data. Partitioning along the H dimension, corresponding to the rows, resulted in tiny tensors that occupied contiguous and consecutive blocks of memory. This is demonstrated in Figure 4.6, where the first tiny tensor occupies a contiguous block of memory. This allowed tiny tensors to be copied in a single DMA transaction.

4.2.5.2 Partitioning an Operation

We introduced a *partitioning factor*, which controlled how many tiny tensors a tensor was logically partitioned into. When partitioning any operation in the inference graph, the tiny tensors were kept equal in size. In the case of the number of rows, H not being a multiple of the partition factor, P, $H\%P$ rows were equally split between the first $H\%P$ tiny tensors.

Using this approach, the two-input add operations used to perform element-wise addition of tensors were easily partitioned. On the other hand, convolution operations required closer attention with the partitioning details depending on the kernel size and padding. For convolution layers with a kernel size greater than 1, there would be a number of receptive field that fell over two tiny tensors. This case was dealt with by

copying in extra rows being included in a tiny tensor needed to create valid receptive fields.

Another factor to consider in the logical partitioning was the application of padding. Whereas an input tensor, would be padded on the outer edges, this would not always be true for tiny tensors as a tiny tensor in the middle of the input tensor would only have padding to the left and right edges but not the top and bottom. Similarly, tiny tensors at the top or bottom of the input tensor would require padding on only three sides as opposed to the input tensor having padding on all four sides. According to the location of the tiny tensor within the input tensor, the padding details were stored alongside the partitioning strategy.

The details of the partitioning strategy required to process the input tensor including the partition factor or number of tiny tensors, their start and end, dimensions and operation data such as padding were determined statically at initialisation and stored in a data structure alongside the inference graph. For context, the details of the data structure are shared in Listing C.1 in Appendix C.

4.2.5.3 Limitations

We note that a limitation of this 1-D partitioning approach would be that the minimum size would be dictated by the kernel size of the filter in addition to the width or channels of an intermediate representation. For example, the size of each of the two $120 \times 120 \times 32$ tensors input to add operation would be $460,800$ bytes assuming 8-bit quantisation. Using a 1-D partitioning approach in the H dimension these tensors could, at maximum, be partitioned with a partition factor of 120. This would produce 120 tiny tensors of size $3,840$ bytes.

The minimum amount of required memory to overlay these tiny tensors with double buffering would therefore by $4 \times 3,840 = 15,360$ bytes. As such, the 1-D partitioning approach would not be usable on devices with less than $15,360$ bytes of internal memory. Similarly if the $120 \times 120 \times 32$ were to be convolved with a $3 \times 3 \times 32$ kernel, the smallest tiny tensor would be of size $3 \times 120 \times 32 = 11,520$ bytes with 4 tiny tensors requiring a minimum of $46,080$ bytes. In practice however, we found that this partitioning scheme used in conjunction with our overlaying approach was able to deploy large models within internal memory constraints of a range of MCUs as we show in the coming sections.

Another limitation of this approach is that it is not applicable to linear layers. In our work, we implemented linear layers with pointwise convolutions where the input tensor had unit spatial resolution. In this case, we could not partition the input tensor as

there was only one row in the input tensor that would not be divisible further. However, we noted that state-of-the-art CNNs only utilised one linear layer at the final classification layer. Across the CNNs we examined, including EfficientNet, MobileNetV3, MNASNet, ProxylessNAS, the input to the linear layer was observed to be less than 1280 bytes which did not require partitioning anyway.

4.2.6 Partitioning the Inference Graph

Having discussed our approach to how we logically partitioned an operation, we now detail how we derived the parameters of the partitioning strategy of an inference graph and allocate memory for the fast buffers required by our overlaying scheme.

The partitioning scheme used a simple greedy approach to determine the partitioning strategy that is able to implement the overlaying criterion described in Section 4.2.4 under the available memory budget. The algorithm to derive the partitioning strategy is listed in Algorithm 3.

The partitioning scheme first determines how much memory is utilised by the bias tensors, quantisation parameters and im2col buffers as in Line 3 to 5. By sequentially looping through all possible partition factors (Line 7), the algorithm then determines the smallest possible partition factor, p_{max} through which all operations can be partitioned to meet the memory constraint. The partition factor is incremented to reduce the memory requirement till the total memory required for the seven buffers is lower than the memory constraint (Line 14, 16). When a valid partitioning strategy is found, the buffers are allocated according to the sizes calculated by the partitioning scheme. This greedy approach determines the smallest partition factor, p_{max} that is able to partition the operation with the largest memory requirement which dictates the peak memory usage and allows us to allocate buffer sizes according to the worst case scenario.

However, there would be many operations in the inference graph that could utilise a smaller partition factor than p_{max} and still be accommodated within the allocated internal memory buffers. In such a case, having a higher partition factor would result in unnecessary overheads including cache maintenance, DMA interrupt servicing and redundant data copying. To reduce these overheads, we perform a second pass through the inference graph to find the smallest partition factor for all operations that can be accommodated in the allocated buffers as shown in Line 24 onwards.

4.2.7 Network Inference Pipeline

We used external memory as main memory and declared fast buffers in internal memory which were able to accommodate data for *tiny operations* derived through the previously described partitioning scheme. With this overlaying approach, inference is

Algorithm 3 Pseudocode to determine the partitioning strategy for an NN that meets a particular memory constraint. PARTITIONOP is used to partition an operation with a partition factor, p as described in Section 4.2.5.2. The helper functions GETMAXRECEPTIVEFIELDSIZE, GETMAXBIASSIZE, GETMAXTINYTENSORSIZE are used to find the size of the largest receptive field, bias and tiny tensors in the network.

Inputs: NN_graph, the inference graph. $m_{constraint}$, the memory constraint
Output: $strategy\{num\_ops\}$, partitioning strategy for all operations in NN_graph

function PARTITIONNN($NN\_graph, m_{constraint}$)

    m_{im2col} = GETMAXRECEPTIVEFIELDSIZE(nn_Graph)
    m_{bias} = GETMAXBIASSIZE(nn_Graph)
    $m_{quant} = 2 \cdot m_{bias}$

    for $p \leftarrow 2, 224$ **do** ▷ Loop over possible partition factors, p

        for op in NN_graph **do** ▷ Calculate strategy for each op in graph
            $strategy[op]$ = PARTITIONOP(op, p)
        end for

        m_{tiny_tensor} = GETMAXTINYTENSORSIZE($strategy$)
        $m_{total} = 3 \cdot m_{tiny\_tensor} + \text{MAX}(m_{tiny\_tensor}, m_{im2col}) + m_{bias} + m_{quant}$

        if $m_{total} \leq m_{constraint}$ **then** ▷ If memory constraint is met
            ALLOCATEBUFFERS() ▷ Allocate seven fast buffer in internal memory
            $p_{max} = p$ ▷ p_{max} required to partition the largest operation
            break
        end if

    end for

    for op in NN_graph **do**

        for $p \leftarrow 2, p_{max}$ **do** ▷ Find smallest partition factor $< p_{max}$

            $opStrategy$ = PARTITIONOP(op, p)

            if GETMAXTINYTENSORSIZE($opStrategy$) $\leq m_{tiny\_tensor}$ **then**
                $strategy[op] = strategy_{temp}$ ▷ Update ops partitioning strategy
                break
            end if

        end for

    end for
return $strategy$

end function

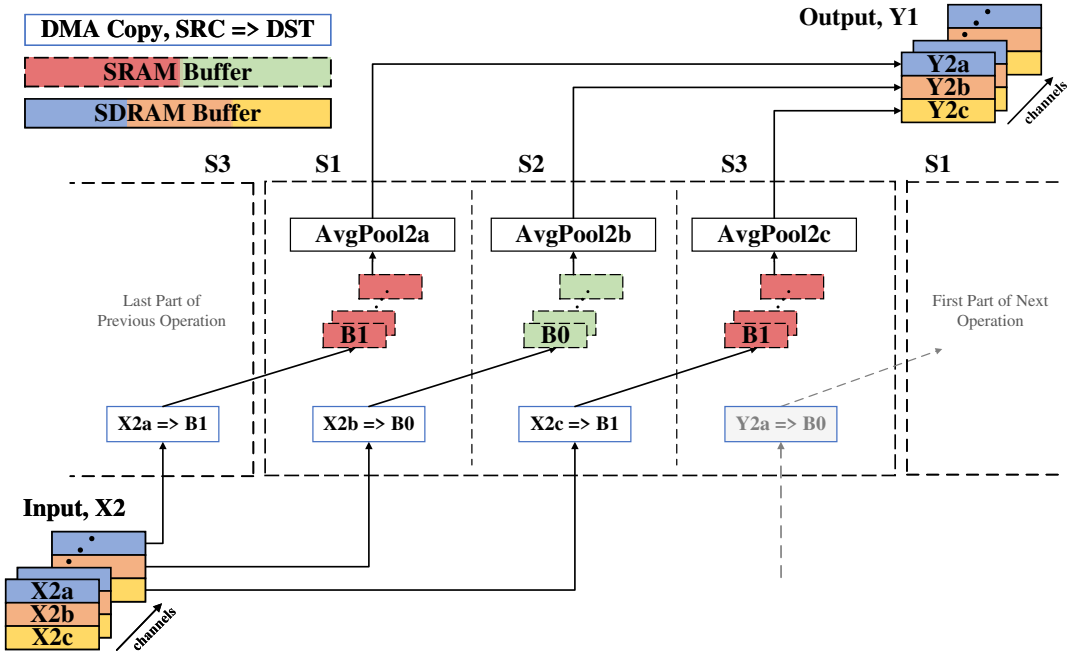


FIGURE 4.7: TinyOps Operation Execution

performed by fetching data from external memory to internal memory on a need to compute basis.

To hide the latency of fetching data from external memory to the internal memory buffers, we offload the data movement from external memory to internal memory. As the DMA is able to operate independently of the MCU core, we are able to establish a pipeline between the MCU core and the DMA. In the pipeline, as the MCU core is processing one tiny tensor of the input, the DMA copies in the next tiny tensor in parallel. This effectively allows us to hide the latency to fetch data from the slower external memory. The pipeline with the logical paritioning of tensors and DMA based overlaying is illustrated in Figure 4.7 for an average pooling operation.

As can be observed the input and output tensors, X2 and Y1 are stored in external SDRAM. These tensors are logically partitioned by a *partition factor* of 3 into 3 equally sized tiny tensors X2a, X2b and X2c. The limited SRAM is used for declaration of fast buffers, B0 and B1, which are allocated memory according to the size of the tiny tensors. The pipeline begins with the DMA copying data for the tiny tensor X2a to the fast buffer B1. After the data has been copied, the MCU core proceeds to apply the average pooling operation, AvgPool2a to the tiny tensor and write the output to the logical tiny tensor, Y2a which is located in SDRAM. While the MCU core is processing the tiny tensor, X2a residing in buffer B1, the DMA is configured to copy the next tiny tensor to the second buffer, B0. By the time the MCU core is done processing X2a in B1, the data has been copied from X2b to B0, and the MCU core seamlessly moves to processing the data in B0. The DMA then begins copying X2c to the newly freed buffer B1. This

process continues until the entire operation has been executed as a sequence of tiny operations. The inference algorithm for an entire NN using this pipelined approach is shown in Algorithm 4.

Network Level Pipeline: To perform inference, TinyOps establishes a network level pipeline between the CPU and DMA that is synchronised at the tiny operation level. The prolog of the pipeline consists of copying in the data for the first tiny operation of the first operation using the meta-data from the inference graph and the partitioning strategy for the operation as shown in Line 3 to 5. The framework then sequentially loops through the operations in the inference graph. For each operation, the framework then sequentially computes the tiny operations that have been produced through the partitioning algorithm (Line 9). Before getting references to the input data, the CPU synchronsises with the DMA to make sure the data has been copied in as in Line 11. Once the data has been copied, the DMA is configured to move the next tiny operations data into internal memory using the previously described double buffering strategy. If the current tiny operation is the last in the current operation, a DMA transaction is initiated by the CPU to copy in data for the next operations first tiny operation (Line 14), otherwise the data for the current operations next tiny operation is copied in (Line 18). After the DMA transactions are initialised the CPU gets references to the data for the current tiny operation and computes the current tiny operation in parallel with the DMA as in Lines 21 to 25. The tiny operations and operations are sequentially performed in this manner to perform inference with the NN.

DMA Configuration & Synchronisation: Communication between the DMA and CPU was maintained via a light-weight circular queueing mechanism. Using this approach, the CPU is able to asynchronously make data movement requests that can be performed by the DMA. The circular queue consisted of a number of buffers used to store data structures holding configuration parameters for a data transaction (source, destination address and copy size) along with a read and write pointer. The queue is initialised with the read and write pointer set to the same value which represents the queue empty condition. To make a request, the CPU writes the configuration to the buffer pointed to by the write pointer followed by an increment of the write pointer. After putting the request into the queue, the DMA state is checked. If the DMA is idle, the DMA is triggered to start the transaction. In case the DMA is busy with a transaction, the CPU simply puts the request in the queue without configuring the DMA as only one transaction can be performed at a time. In this scenario, the DMA is configured in the interrupt service routine (ISR) that is called on completion of a transaction. When the ISR is entered, the read pointer is incremented to signal that the previous data movement request was read and successfully completed. Following this, the read and write pointers are checked. If the pointers are unequal, it implies that there is a

pending transaction in which case the DMA is configured by the CPU in the ISR. The case of equal pointers implies that the queue is empty and that there are no pending transactions, in which case the ISR can be exited.

In the inference pipeline described in Algorithm 4, we maintained synchronisation between the DMA and CPU by polling the read and write pointers. As shown in Line 11, the CPU waited until all data transactions were completed by polling the condition of the read and write pointers being equal.

Algorithm 4 TinyOps Inference Pipeline

Inputs: *NN_graph*, the inference graph. *strategy*, the partitioning strategy. *input*, the tensor input to the NN

Output: *output*, the tensor output from the NN

```
1:  function InvokeTinyOps(NN_graph, input)
2:
3:      CopyInWeightTensor(1)                                  ▷ Initialise pipeline
4:      CopyInQuantisationParamters(1)
5:      CopyInTinyTensors(strategy[1], 1)
6:
7:      for op in NN_Graph do
8:
9:          for i ← 1, num_tinyops_in_op do      ▷ Sequentially compute tiny ops in op
10:
11:             SynchroniseDmaCpu()                    ▷ Wait till data has been copied
12:
13:             if IsLastTinyOp() then          ▷ Copy in data for next ops first tiny op
14:                 CopyInWeightTensor(op+1)
15:                 CopyInQuantisationParamters(op+1)
16:                 CopyInTinyTensors(strategy[op+1], 1)
17:             else                            ▷ Copy in data for current ops next tiny op
18:                 CopyInTinyTensors(strategy[op+1], i+1)
19:             end if
20:
21:             tiny_inputs = GetTinyInputTensors(op, i)
22:             weights = GetWeightTensor(op)
23:             quant_params = GetQuantisationParameters(op)
24:
25:             tiny_output = Kernel(tiny_inputs, weights, quant_params)
26:
27:         end for
28:
29:     end for
30: return GetOutputTensor()
31:
32: end function
```

4.2.8 Performance Analysis

Using the previously explained strategy of operation partitioning and overlaying from external to internal memory, the TinyOps inference framework mitigates the overhead of memory fetches from external to internal memory. This allows inference to be performed with internal memory like latency. To achieve exactly the same latency as performing inference from internal memory would not be possible due to extra overheads in the TinyOps inference scheme such as repetitive cache maintenance operations, DMA configurations and interrupt handling operations. In such a scenario, it might be useful to look at theoretical bounds on performance that could be achieved with such methods. However, it is difficult to make such theoretical estimates accurately. This is due to the complexity of analytically modelling the inference execution on the device which involves control and compute instructions, complex data movement in the memory hierarchy, different data storage layouts and access patterns in addition to constantly varying cache states. Prior works in performance modelling usually take assumptions to simplify the inference execution such as ignoring the cache, limiting the considered instructions or in some cases ignoring compute latencies if the workload is memory bound (Ofenbeck et al., 2014; Yuan et al., 2024) which can lead to inaccurate latency estimates. As such, prior works developing inference frameworks and kernels (Lai et al., 2018; Lin et al., 2020) rely on measuring on-device inference latency for performance analysis. Due to the mentioned limitations, we adopt a similar approach where we measure on-device inference latency and compare the performance of TinyOps with the internal and external memory configuration for performance analysis which we detail in the next section.

4.3 Experiments and Results

In this section, we detail the experiments we carried out to evaluate our design decisions in addition to comparing the performance of the internal and external memory configuration with the TinyOps approach. In our evaluation we considered metrics including the accuracy, latency, power and energy per inference which are vital for high performance deployment on resource constrained edge devices. We used the commercial off the shelf devices by STMicroElectronics specified in Table 4.2.

4.3.1 Overlaying Strategy

We analysed the effect of our strategy of overlaying different data used by operations including the input tensors, filter weights, bias tensors and quantisation parameters. We derived a scaled variation of a ProxylessNAS model for the constraints of the F469 device with internal memory and storage constraints of 256KB and 1MB respectively.

Within these constraints, the proxyless-w0.30-r144 model was deployed which had a memory and storage requirement of 230KB and 908MB. The latency reduced by overlaying each of the parameters was measured on-device and compared with the latency of deploying the model in the internal or external memory configuration. This is shown in Figure 4.8.

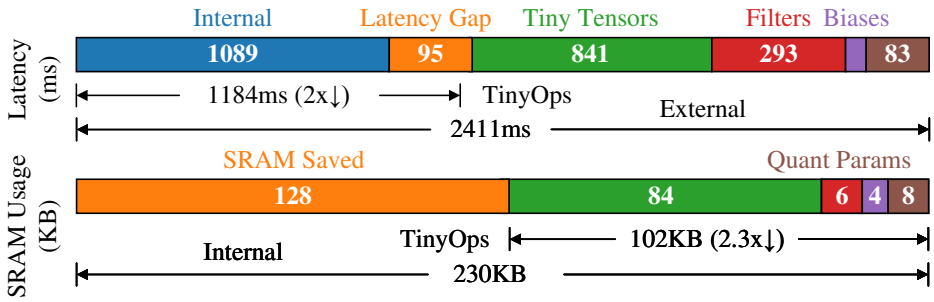


FIGURE 4.8: Overlaying Strategy Effect on Latency

As can be observed, TinyOps was able to reduce the latency of the external memory configuration by 2x↓ using only 102KB of internal SRAM. Compared to the internal memory configuration TinyOps had a slightly higher inference latency (95ms↑) with 2.3x lower internal SRAM usage than the internal memory configuration which required 230KB for all of the models data.

Looking at how each of the parameters contributed to the reduction in latency, we observe that the overlaying of the tiny tensors and the filters had the greatest effect as multiple elements are read from the tensors when convolving the filter with a receptive field. The tiny tensors had the largest memory usage as they required double buffering in our approach. We note that the filters only used a small amount of memory (6KB) as they reused memory allocated for the tiny tensor buffer that would be unused in two-input non-parameterised operations (e.g add). Overlaying the bias tensor and quantisation parameters utilised in the operations were able to further reduce latency at low memory cost.

4.3.2 Energy Consumption

We compared the energy efficiency of the TinyOps approach by measuring the on-device power consumption when performing inference. We measured the power consumption using the Qoitech Otii Arch. The power consumption in a deployment scenario would depend on the read and write patterns of the particular model. For example, one model might make more reads and/or write to external memory than another. To decouple the effect of the model architecture on power consumption and study the effect of the memory configuration on power consumption, we deployed the same model under the three different memory configurations.

TABLE 4.5: Power Consumption of TinyOps compared to Internal and External Memory Deployment

Platform	Power (mW)			Latency (ms)			Energy per Inference (mJ)		
	Int	Ext	TinyOps	Int	Ext	TinyOps	Int	Ext	TinyOps
L552	65	205	140	855	2963	977	56	607	137
F469	290	435	380	1089	2411	1131	316	1049	430
F746	645	770	798	685	1004	772	442	773	616

We derived models for the internal memory constraints of the three platforms described in Table 4.2 and deployed the models with the internal, external and TinyOps configuration. The models were derived from a ProxylessNAS baseline via scaling with proxyless-w0.10-r192, proxyless-w0.30-r144 and proxyless-w0.30-r176 deployed to the L552, F469 and F746 respectively. We measured the average power consumption of an inference and used this metric to derive the energy per inference based on the latency of inference. We did not observe any significant variation between multiple independent inferences in latency or power consumption. For deployment with TinyOps, the memory budget to derive the partitioning strategy was set as the SRAM size of the device. The power consumption, latency and energy per inference for the different scenarios is shown in Table 4.5.

As expected, we observed that TinyOps and the external memory configuration had higher power consumption than the internal memory configuration due to usage of external memory. Comparing TinyOps with the external memory configuration, we observed that on the L552 and F469, TinyOps had lower power consumption than external memory inference even though it used the additional DMA peripheral. This is due to external memory inference requiring repetitive high energy reads of filters and tensors from SDRAM for every stride of convolution in convolution operations in external memory operations. On the other hand, TinyOps overlays data from SDRAM to SRAM, requiring only one read from SDRAM with subsequent reads made from the low power SRAM. The power saved from reduced SDRAM reads outweighed the overhead of the DMA resulting in an overall reduction for TinyOps.

However, the same behaviour was not observed on the F746 due to the presence of cache which reduces SDRAM reads in external memory inference by providing repetitive low energy data access from itself. In this case, the additional DMA used by TinyOps outweighs the reduced SDRAM reads in external memory inference for a higher power consumption. Nevertheless, we observed that TinyOps lower latency outweighed the 3.5% higher power consumption, for lower energy-per-inference. Comparing the latency of TinyOps to the different configurations on each device, we observed that TinyOps accelerated inference by 1.3x-3x compared to the external memory configuration, whilst only being 1.1x slower than the internal memory configuration.

4.3.3 Adaptive Memory Usage

TinyOps derived the partitioning strategy for the NN according to the specified internal memory constraint. As shown in Figure 4.9, for a Proxyless-w0.50-r192 model requiring 4.2MB of storage and 627KB of memory, a memory budget of 256KB or 320KB for the F469 or F746 devices can be met while accelerating inference up to 2.1x.

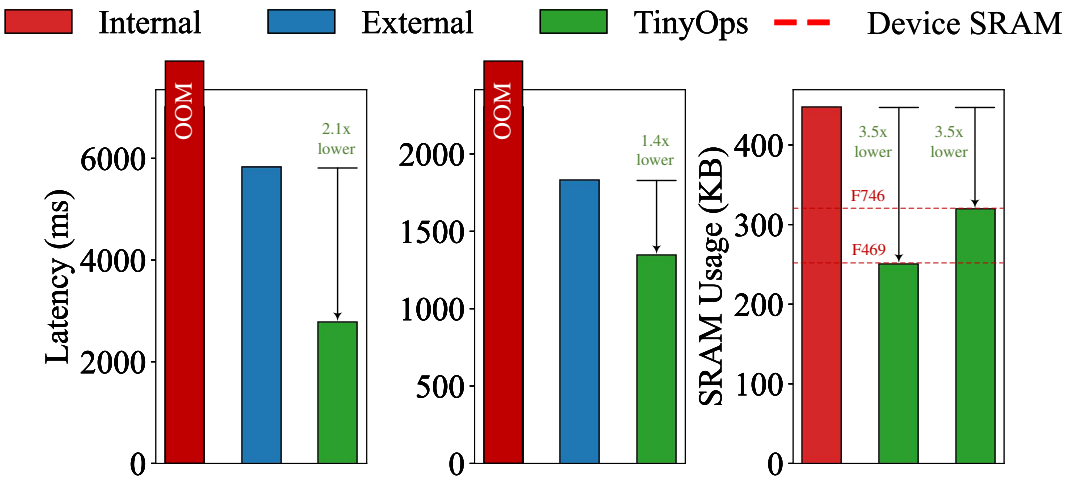


FIGURE 4.9: The adaptive partitioning strategy allows us to meet diverse memory budgets

4.3.4 Design Space Performance Comparison

We deployed large models with the TinyOps inference framework to demonstrate how it is able to combine the advantages of low latency and high accuracy of the internal and external memory configuration. We utilised scaled variations of an MNASNet and ProxylessNAS model for deployment on each of the devices listed in Table 4.2. Where the partitioning scheme was unable to sufficiently lower the internal memory usage to the device constraint, we reduced the size of tensors in the model by reducing the channels or width of the network through a uniform multiplier. The width multiplier was reduced in increments of 0.25 starting from 1.00 down to 0.50 to find the largest model that could be partitioned to meet the specific devices internal memory constraints. The results for the largest model that we were able to deploy with the TinyOps inference framework and a latency comparison with naive approaches utilising external memory is shown in Table 4.6.

As can be observed, TinyOps was able to accelerate the inference latency by up to 1.4x-2.5x across the devices compared to the external memory configuration. We note that our prior observation, of TinyOps being more effective in reducing latency when cache was not present on the device held true for large models as well. On the L552 and

TABLE 4.6: TinyOps lifts the accuracy acheivable on MCUs with low inference latency

Platform	Model	MACs (M)	Params (M)	Acc (%)		Design Space	Latency (ms)
				FP32	INT8		
L552	MNASNet-w0.50-r224	88	1.47	65.25	**63.64**	External **TinyOps**	20058 **8134**
F469	MCUNetV1-F469	67	0.73	60.8	59.47	Internal	2547
	MCUNetV2-F469	119	<1	-	64.9	Internal	-
	MNASNet-w0.75-r224	191	2.75	70.94	70.37	External **TinyOps**	13139 **6220**
	Proxyless-w0.75-r224	193	2.59	71.23	**70.67**	External **TinyOps**	14660 **7008**
F746	MCUNetV1-F746	82	0.75	62.5	61.47	Internal	1131
	MCUNetV1-F746-int4	170	1.4	-	63.5	Internal	-
	MNASNet-w1.00-r224	314	4.38	73	**72.67**	External **TinyOps**	5278 **3890**
	Proxyless-w0.75-r224	193	2.59	71.23	70.67	External **TinyOps**	4353 **3123**

F469 where cache was not present, the inference latency was accelerated by up to 2.1x-2.5x. On the F746, the cache was able to mitigate the slower access latency of external memory due to which the inference latency was reduced by only 1.4x.

We observed that as the amount of internal memory available decreased on the devices, the size of the largest model that could be deployed, and therefore the accuracy achieved in deployment reduced. For example, for the MNASNet model, the F746 with 320KB of SRAM was able to accomodate the entire model, whereas the width had to be reduced to 0.75 and 0.50 to deploy a model on the F469 and L552 with 256KB and 192KB of SRAM respectively. This was due to the limitation of the H dimensional partitioning scheme utilised by TinyOps. As TinyOps only partitioned tensors across the H dimension, the minimum size of a tiny tensor would be $K \cdot W \cdot C$ where K is the kernel size, W is the width of the representation and C is the number of channels. For larger networks, this scheme produced tiny tensors that could not be partitioned to meet the internal memory constraint when either K, W or C were high. To reduce the memory requirement, we reduced C or the number of channels corresponding to the width. While this reduced the memory requirement, it also reduced the accuracy of the model.

The ProxylessNAS model also demonstrates a case of a model that was not amenable to deployment with TinyOps. As can be observed, the ProxylessNAS model had to be scaled down to a width of 0.75 for deployment of the F746, whereas on the L552, a partitioning scheme could not be found even with a width multiplier of 0.50. This was due to the ProxylessNAS model having intermediate representations that could not be overlayed with the TinyOps approach. In this case, the memory requirement for fast

internal memory buffers utilised for overlaying the tiny tensors and the im2col buffer had to be reduced by lowering the width.

Nevertheless, the MNASNet models deployed with TinyOps achieved up 6% and 8% higher accuracy than the state-of-the-art MCUNet models derived within internal memory constraints for the F469 and F746 devices with up to 2.5x lower inference latency compared to the external memory configuration.

4.4 Discussion

In deploying a model to a microcontroller platform, an efficient inference framework is equally important to the design of an efficient model. In this chapter, we showed how current inference frameworks limited the performance achievable on MCU platforms. Prior inference frameworks did not effectively utilise the memory hierarchy, due to which the weights and activations resided in either internal or external memory. We showed that this limited the achievable accuracy or inference latency due to the size constraints of internal memory or access latency of slower external memory respectively.

We proposed the TinyOps inference framework to combine the advantages of size and speed of external and internal memory. TinyOps utilised external memory as main memory and copied data into internal memory as required for processing via the DMA peripheral. To meet internal memory constraints we proposed a partitioning strategy that seamlessly integrated with the underlying kernels. We further overlapped the data fetching with the processing performed by the DMA and CPU respectively to effectively hide the latency of accessing slower external memory.

We adopted a generic model of the MCU platforms that allowed us to port the TinyOps inference framework across the range of off-the-shelf MCU platforms. Compared to prior approaches utilising external memory, we were able to reduce inference latency by up to 1.4x-2.5x to lift the ceiling of performance achievable on MCU platforms. In the next chapter, we study the performance of the different design spaces and how efficient models can be derived across the pareto frontier.

Chapter 5

Efficient Model Design for MCUs

Edge devices often have to cater to various resource constraints and deadlines posed by varying environmental factors. This requires being able to trade-off accuracy and latency in real-time to meet any application requirements. In the previous chapter, we showed how the design space for model deployment was limited in either accuracy or latency by current inference frameworks which lead to a sub-optimal accuracy-latency pareto frontier. To alleviate these limitations, we proposed, the TinyOps inference framework which opened a new design space that lifted the accuracy achievable with low latency. In this chapter, we study the question: *how can we achieve the optimal accuracy-latency pareto frontier and trade-off the accuracy and latency at run-time on MCUs?*.

Deriving the optimal accuracy-latency pareto frontier requires us to look at the design of models from two aspects. We note that the accuracy-latency pareto frontier might be composed of models from different design spaces as the internal memory design space was limited in accuracy and the TinyOps design space had slower inference latency compared to internal memory. As such, exploring the accuracy-latency trade-off requires us to compare the internal memory design space with the TinyOps design space to determine the boundary at which the internal memory design space becomes sub-optimal. For this purpose, we analysed state-of-the-art models derived for internal memory constraints and compared them with models derived from the TinyOps design space. Our analysis of state-of-the-art models designed for internal memory revealed them to be sub-optimal in accuracy, and suprisingly, latency and energy efficiency as well.

The second aspect we look at is how to derive the most performant model from a design space. Prior works typically derive models from search spaces that are designed for efficiency on mobile platforms. In this chapter, we perform a latency analysis of DNN deployment on MCUs at the micro-architecture level to derive heuristics for efficient search space and model design on MCUs. Using these heuristics, we derive static and

dynamic DNNs for deployment on MCUs which enable us to achieve state-of-the-art trade-off between accuracy and latency at run-time.

The work carried out in this chapter addresses the research question with the following contributions

- **Evaluation of Internal Memory Design Space:** We perform an in-depth analysis of state-of-the-art CNNs derived for MCU platforms through NAS and manual approaches. We show that the constraint on model size imposed by internal storage results in sub-optimal accuracy and somewhat counter-intuitively, latency and energy efficiency as well.
- **Model Design Heuristics for MCU platforms Derived from a Novel Latency Analysis at the Micro-Architecture Level:** We benchmark the throughput of low-level operations that compose state-of-the-art CNNs in addition to studying the effect of hyper-parameters used in conventional scaling on inference latency to derive novel model design heuristics for MCU platforms.
- **Accuracy-Latency Tradeoff:** We study static and dynamic approaches to achieve run-time accuracy-latency trade-off on MCU platforms. Using novel model design heuristics, we derive models from an efficient search space using a lightweight supernetwork based NAS approach that is able to meet a number of latency constraints. The derived models outperform state-of-the-art internal memory models with up to 6.7% higher accuracy and 1.4x lower latency.

5.1 Analysis of Model Design on MCUs

5.1.1 The Mobile Search Space

We looked at the mobile search space and how this could be refined for microcontrollers. The building block of the mobile search space is the MobileInvertedConv module shown in Figure 5.1. The MobileInvertedConv module consists of a 1x1 pointwise convolution to expand to a higher number of channels, followed by a $K \times K$ depthwise convolution followed by a last 1x1 pointwise convolution operation which projects the channels to the output number of channels of a MobileInvertedConv block. The search space is constructed by cascading together a number of these MobileInvertedConv blocks. The hyper-parameters that are searched for in the search space are as below

- **Kernel Size, K** In the MobileInvertedConv block, this is the kernel size of the depthwise convolution in the block. The kernel size chosen in the search space includes, 3, 5, and 7.

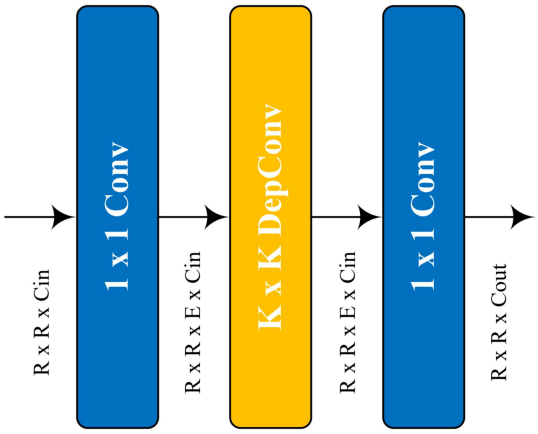


FIGURE 5.1: Mobile Inverted Conv Block Structure

- **Channels, C_{in} and C_{out}** The input and output channels of the MobileInvertedConv module. The hyper-parameter is included in the search space in a varying capacity. A typical approach is to use the channel configuration of a baseline model.
- **Expansion Ratio, E** The number of channels to which a 1x1 convolution projects up, which is the input number of channels to the depthwise convolution. The expansion ratio is chosen between 2 and 6.
- **Depth** The depth of the neural network, which is controlled as the number of MobileInvertedConv modules cascaded together to form the network.
- **Resolution, R** The resolution of the image input to the CNN. For example ImageNet under the mobile setting is resized to 224×224. For MCUs lower resolutions are often used e.g. 96, 128, 176.
- **Width** The starting point for the search space is often taken as an existing mobile model, on which the preceding hyper-parameters can be mutated. The width hyper-parameter applies a global multiplier to the input and output channels of all operations in the base network.

An example ProxylessNAS architecture derived from the mobile search space is shown in Figure 5.2. As can be observed, the derived architecture has a number of MobileInvertedConv blocks cascaded together with varying kernel sizes and channels. MB6 3x3 refers to a MobileInvertedConv block with an expansion ratio of 6 with kernel size 3.

5.1.2 Studying the Pareto Frontier

We studied the pareto frontier by training scaled variations of a ProxylessNAS model. Due to the high cost of training models on the ImageNet dataset, we approximated the

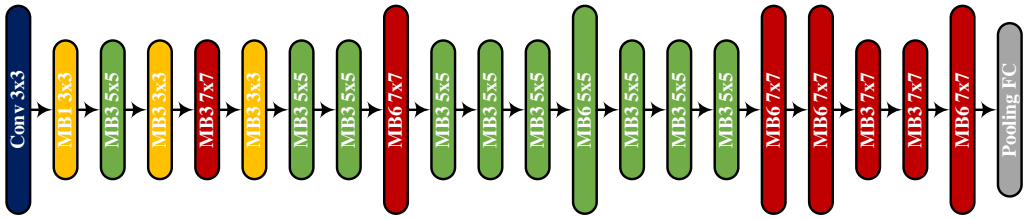
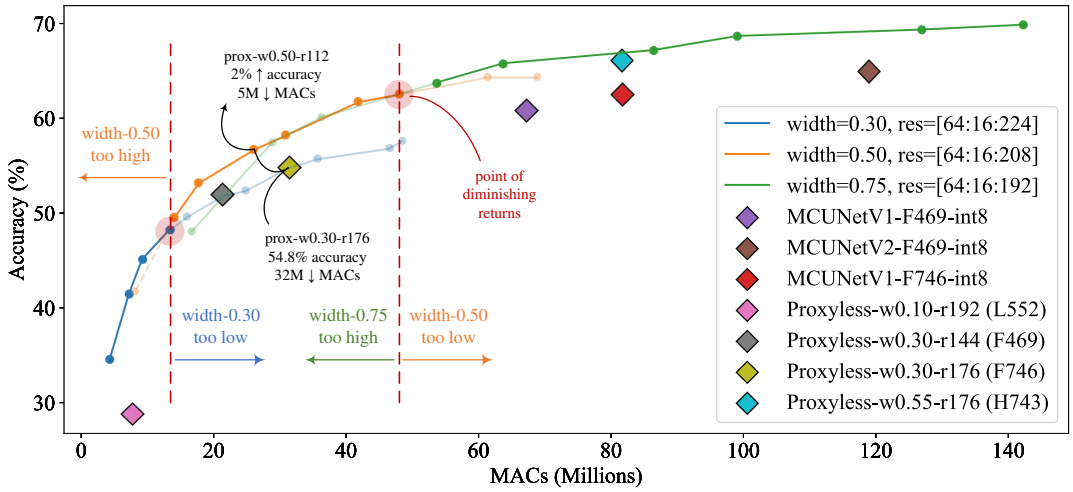


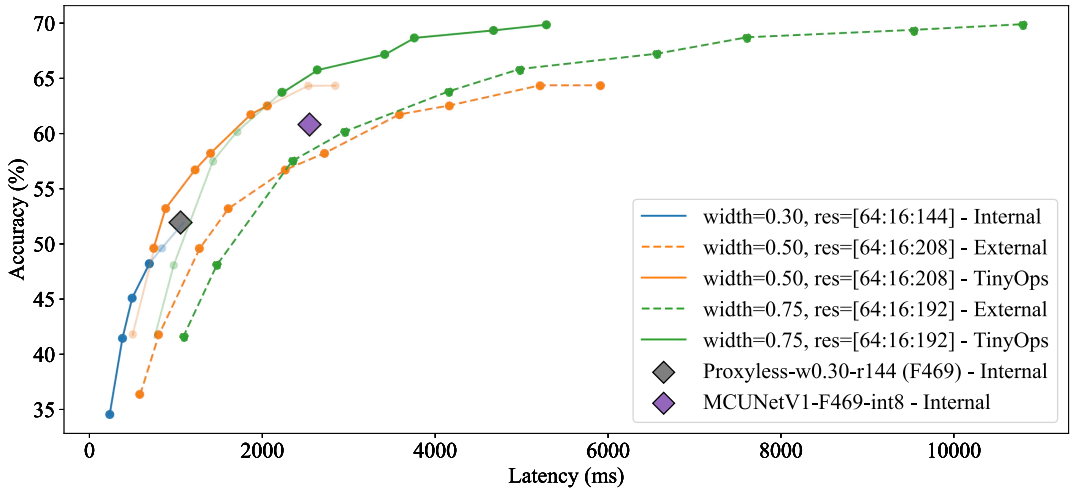
FIGURE 5.2: ProxylessNAS model derived from Mobile Search Space

pareto frontier by limiting the width multipliers used and the input resolutions. To approximate the pareto frontier, we trained models with width multipliers of 0.30, 0.50, and 0.75 while increasing the resolution from 64 in increments of 16. We note that using scaled variations of an existing mobile model would not necessarily deliver pareto optimal architectures. Indeed, it might be possible to perform manual handcrafting of models to derive the pareto optimal architecture for any latency or MAC constraint. However, we show that through our study of an approximate pareto frontier we were able to derive design heuristics that enabled us to significantly advance performance on MCUs. The accuracy-macs and accuracy-latency pareto frontier for an F469 and F746 MCU are shown in Figure 5.3.

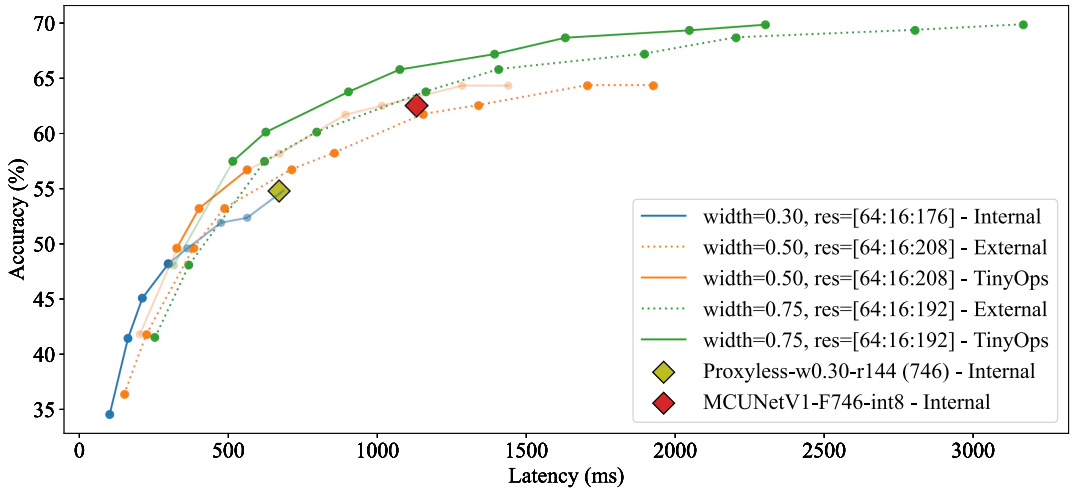
As mentioned previously, for any width multiplier, the accuracy-macs or accuracy-latency curve is drawn by incrementing the input resolution for the given model from 64 in increments of 16. As observed, as the resolution increases, the accuracy increases for all models using any particular width multiplier. This would be expected as the model would have more raw input features to extract information from. Additionally, a higher dimensional space might also increase the chance of learning representations with higher class separability (Howard et al., 2017). However, we interestingly note an intersection point between the models with different width multipliers which we term the *point of diminishing returns*. We note that for any width multiplier, increasing the resolution past this point is sub-optimal as this returns a minor increase in accuracy at the cost of significant computation or inference latency. For a width of 0.30, this point is observed in Figure 5.3a, when the resolution is increased to 112 corresponding to a computational complexity of 15M MACs. As shown, after the point of diminishing returns, using an increased width multiplier of 0.50 would be beneficial, for example, proxyless-w0.30-r176 is outperformed by the more efficient proxyless-w0.50-r112 model with 2% higher accuracy and 5M lower MACs. The same observation can be made for a width multiplier of 0.50 where past the point of diminishing returns, which is observed around the 50M MACs mark, it is beneficial to utilise a higher width multiplier of 0.75 We hypothesise this behaviour occurs due to the representational capacity being limited by the number of parameters or width multiplier of the model.



(A) Accuracy vs. Macs



(B) Accuracy vs. Latency on the F469



(C) Accuracy vs. Latency on the F746

FIGURE 5.3: The Pareto Frontiers

We observed the opposite phenomenon as well where lower resolutions for high width multipliers yielded minor decreases in latency for a significant drop in accuracy. Such a behaviour could be observed if we were simplifying the complexity of a base model to achieve some complexity constraints e.g. MACs, latency. This likely occurs due to the input features not being informative enough and not containing enough raw information to learn representations with good class separability even though the model might have enough free parameters and representational capacity to learn such representations.

Overall we noted that the pareto frontier was composed of models with varying widths and resolutions from which we drew the design insight as below:

> *When simplifying a model for some complexity constraint e.g. MACs, latency achieving the highest accuracy requires balancing the width and input resolution when deriving models through the conventional approach of scaling*

5.1.2.1 Design Limitations of Internal Memory

In our study we analysed models derived for internal storage and memory design constraints. Prior works, focused on the internal memory design space due to its speed and low power consumption. As we discussed in the previous chapter, the internal memory design space limited the accuracy achievable by a model deployed within the size constraints of internal memory. However, somewhat counter-intuitively, we found the models to be sub-optimal for deployment latency as well.

Conventional Scaling To achieve the highest latency within internal memory constraints, prior works utilise a backbone model with the highest width and input resolution that can be accommodated within internal memory constraints. We derived models for the internal storage and memory constraints of a number of devices including the L552 (512KB Flash/192KB SRAM), F469 (1MB/256KB), F746 (1MB/320KB) and the H743 (2MB/512KB) by STMicroelectronics. The performance of these models (proxyless-w0.10-r192, proxyless-w0.30-r144, proxyless-w0.30-r175, proxyless-w0.55-r176) in relation to the accuracy-macs or accuracy-latency pareto frontier can be observed in Figure 5.3.

As observed, the models lie well below the accuracy-macs pareto frontier with the scaled variations lying well past the *point of diminishing returns* for the largest width multiplier that can be accomodated in internal memory. In contrast, there exist more balanced scalings on the pareto frontier with higher accuracy and lower complexity, however, these cannot be deployed within internal storage and memory due to size constraints.

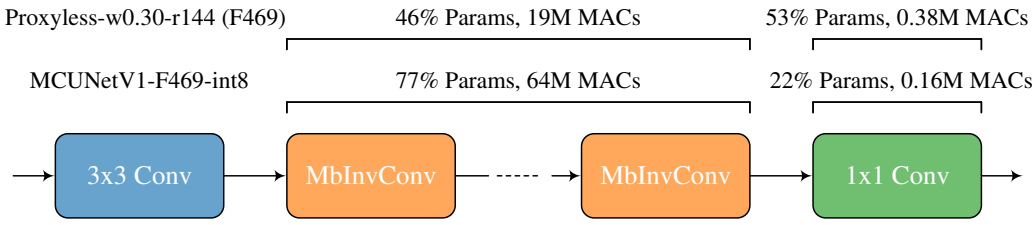


FIGURE 5.4: MCUNet vs ProxylessNAS Parameter and MACs distribution

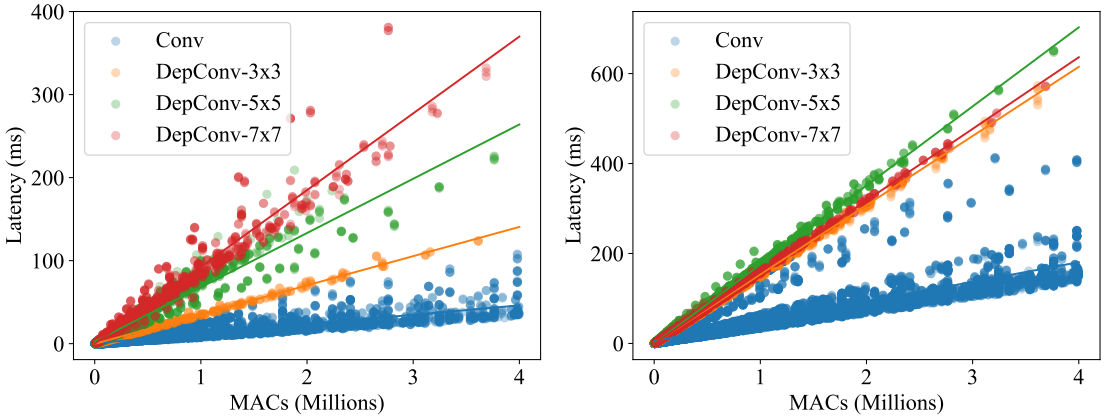
Neural Architecture Search We also analysed the state of the art MCUNetV1/V2 family of models derived within internal memory contraints via neural architecture search. In the search algorithm, the model is derived by maximising the MACs which is used as a proxy for accuracy as the former is computationally inexpensive to measure.

We analysed the architectures derived through this method to find that the MACs are maximised by reducing the parameters in the storage intensive final classification layer and reusing the saved parameters in the compute intensive MobileInvertedConv blocks. This is shown in Figure 5.4 where we compared the distribution of the parameters and MACs of the NAS derived MCUNet models with a ProxylessNAS model scaled for the internal storage and memory constraints of an STM32F469 MCU.

As can be observed, the ProxylessNAS model contains 53% of its parameters in the final dense layer with the rest utilised in the MobileInvertedConv layers. With 53% of the parameters in the final dense layer, this model has 0.38M MACs in the final layer with 19M MACs in the MobileInvertedConv layers. In contrast, the MCUNetV1 model derived for the F469, has only 22% of the total parameters in the final dense layer and 77% of the parameters in the MobileInvertedConv layers which leads to 0.16M MACs in the final layer and 64M MACs in the MobileInvertedConv blocks. As shown in Figure 5.3, this approach has 9% higher accuracy than the conventional scaling approach, however, this is achieved at the cost of significant extra computation (3.2x↑) due to which they fell below the pareto frontier.

5.1.3 Micro-Architecture Latency Analysis

We took a latency driven approach to model design for MCU platforms. While prior works, look at end-to-end latency of models derived from the mobile search space, we performed the latency analysis at the micro-architecture level to derive heuristics for model design. We benchmarked the throughput of operations used in state-of-the-art CNNs to select an efficient set of candidate operations. Similarly, prior works utilise the conventional approach of scaling to lower the memory requirement or computational



(A) With Cache (B) Without Cache

FIGURE 5.5: Benchmarking of Throughput of Operations utilised in the Mobile Search Space

complexity of models. We studied the effect of the conventional approach of scaling on the inference latency when models are deployed on MCUs.

5.1.3.1 Operation Throughput Benchmarking

To derive heuristics for model design, we benchmarked the throughput of the low-level operations that compose state-of-the-art CNNs. As mentioned in Section 5.1.1, the mobile search space extensively utilises 1×1 convolutions and depthwise convolutions with varying hyper-parameters such as kernel, channel configurations and input sizes. In order to benchmark the operation benchmarking, the question arises of what the hyper-parameters should be. The sample of operations we benchmark would need to be representative of operations that are used to compose performant models.

To generate a sample of operations to analysed, we sampled the operations used in models derived via width and resolution scaling of state-of-the-art ProxylessNAS, MobileNetV3 and MNASNet models. We chose these architectures as they have been shown to achieve state of the art performance and have been used by prior works as well. We used width multipliers ranging from 0.10 to 1.00 with increments of 0.05 and input resolutions ranging from 48 to 224 with increments of 16 which produced 228 base models. This process produced a total of 46,056 operations for benchmarking.

To perform the benchmarking we built an automated test bench which read the models from SD Card and logged inference statistics including the layer configuration including operation type, kernel size, input/output channels, input/output resolution, MACs and inference latency. The test bench was built on the F746 MCU hosting an ARM Cortxex M7 MCU. The benchmarking was performed in two scenarios in which cache was enabled and disabled respectively. The cache disabled scenario was used to gain insights of latency performance of MCUs where cache is not available such as the

ARM Cortex M4 based F469. Ideally, the latency statistics would have been gathered by benchmarking on the device itself. However, as we show in our experimental results section, the insights were transferable and accurately applicable to actual on-device inference which we perform in Section 5.2.2. The results of benchmarking the operations is showed in Figure 5.5a and Figure 5.5b.

We note that when separating out the operations by the operation type and kernel size of the operations, a clear trend emerged in both cases from which we drew the following insights

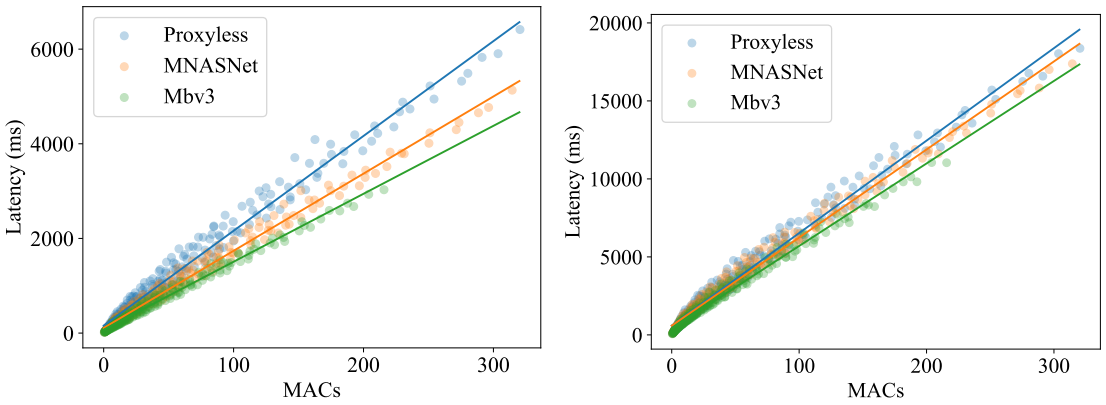
> *Depthwise convolutions are faster than 1x1 convolutions with the same computational complexity*
>
> *3x3 depthwise convolutions are faster than 5x5 and 7x7 depthwise convolutions with the same computational complexity*
>
> *When cache is not present, 3x3 depthwise convolutions are only slightly faster, but not slower, than 5x5 or 7x7 depthwise convolutions*

From these observations we made the design insight that a model derived from the mobile search space for any latency constraints would have the lowest latency when constructed out of 1x1 convolutions and 3x3 depthwise convolutions.

As mentioned in our insights, we observed that 3x3 depthwise convolutions were only slightly faster than 5x5 depthwise convolution when cache was not available. We hypothesise that this is due to the data layout and the kind of data reuse it enables. The NHWC data layout utilised by TinyOps and TfLite-Micro encourages weight reuse in convolution filters. For the case where 4KB of data cache was available, the filters for 3x3 depthwise convolutions could be efficiently cached and reused.

Picking a Backbone A natural question that arises is how we could construct such a model? Prior approaches use the conventional approach of manually scaling models or automate this by incorporating the width and resolution as hyper-parameters into the search space. In our work, we explored different backbone models including ProxylessNAS, MNASNet and MobileNetV3. We observed a key difference between these models that the ProxylessNAS models utilised depthwise convolutions with kernel size of 3, 5 and 7, MNASNet utilised kernel size of 3 and 5, whereas MobileNetV3 only used a kernel sizes of 3. To test whether our insights carried over to actual models, we compared the latency of scaled variations of all three models. Using the same number of multipliers and input resolutions we derived a range of models by varying the width between 0.10 to 1.00 in increments of 0.05 and resolution from 48 to 224 in increments of 16.

The latency of the derived models from the various backbones is shown in Figure 5.6a when cache is enabled. As we can observe for any MAC budget, the MobileNetV3 model yielded the lowest latency due to having only efficient 3x3 depthwise convolutions and 1x1 convolutions. The next best architecture was the MNASNet architecture which contained only 3x3 and 5x5 depthwise convolutions followed by the Proxyless-NAS model which was the least efficient and used all three kernel sizes.



(A) With Cache (B) Without Cache

FIGURE 5.6: Throughput Comparison of Different Backbone Networks

Figure 5.6b shows the latency when cache was disabled. In this case, we still observed the MobileNetV3 derived models to have the lowest latency although the reduction was not as significant. This falls in line with our insights gained by benchmarking the latency of low-level operations where we found that the difference between the depthwise convolutions of varying kernel size was present but not as pronounced.

5.1.3.2 Effect of Hyper-parameter Scaling on Throughput

The hyper-parameters of width and resolution scaling are frequently used to reduce the memory footprint and computational complexity of a model. Through our analysis of an approximate pareto frontier, we showed that achieving a good accuracy-latency trade-off required balancing the width and resolution. We further analysed the effect of width and resolution scaling on the inference latency.

As greater than 98% of the computation is concentrated in the MobileInvertedConv blocks, we studied how the conventional approach of uniformly scaling the width or resolution affected the latency of the MobileInvertedConv block. We studied this by analytically looking at how the computation was scaled in the MobileInvertedConv block as shown in Figure 5.1 under a uniform scaling approach. We looked at how computation is scaled in operations within these blocks when the model is scaled down to meet any constraint. For some width and resolution multipliers, $\alpha, \beta < 1$ the computation

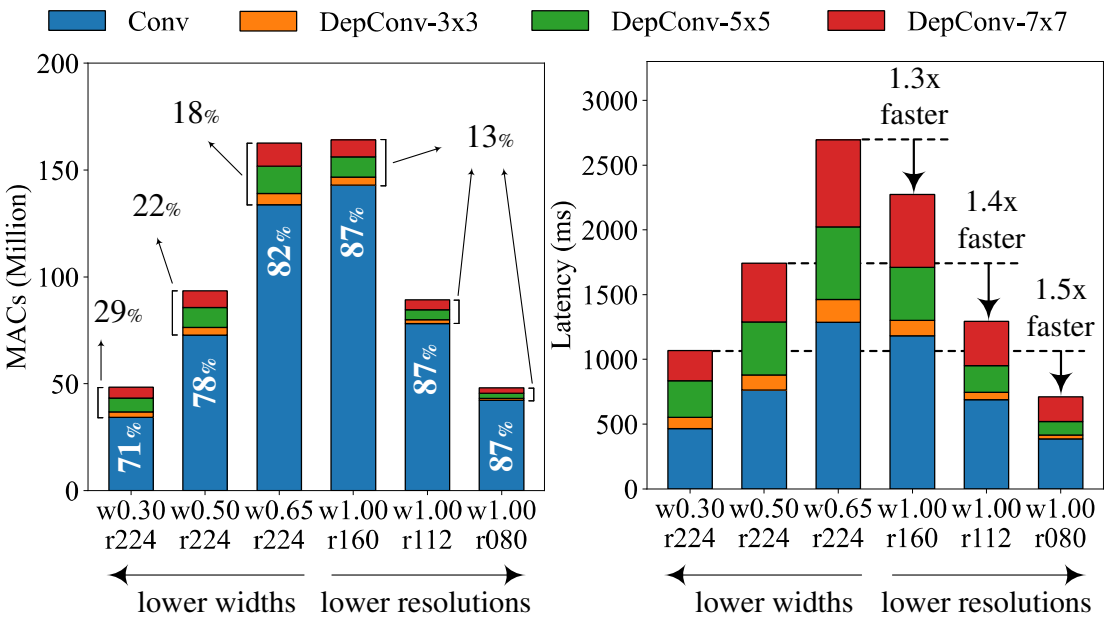


FIGURE 5.7: Computation (MACs) and Latency Distribution of operations in models of varying complexity derived by scaling down width or resolution.

can be calculated as below

$$MACs_{conv1} = \boldsymbol{\alpha^2} \times \boldsymbol{\beta^2} \times R^2 E C_{in}^2 \tag{5.1}$$

$$MACs_{depconv} = \boldsymbol{\alpha} \times \boldsymbol{\beta^2} \times R^2 E C_{in} K^2 \tag{5.2}$$

$$MACs_{conv2} = \boldsymbol{\alpha^2} \times \boldsymbol{\beta^2} \times R^2 E C_{in} C_{out} \tag{5.3}$$

where R is the input resolution, E is the expansion ratio and C_{in} and C_{out} are the input and output channels respectively of the block.

Width It can be observed that lowering the width through a uniform width multiplier, α, quadratically decreases computation by α^2 in Conv operations, while decreasing only linearly by α in DepConv operations. This skews the distribution of computation with a higher percentage of computation performed in DepConvs as shown in Figure 5.7 where progressively reducing the width of a ProxylessNAS model from 1.00 to 0.30 increases computation in DepConvs from 18% to 29%.

Resolution On the other hand, we observe in Equations 5.1-5.3 that meeting any particular MAC budget by lowering resolution by β results in a quadratic reduction in computation by β^2 in either Conv or DepConv operations. This preserves the distribution of computation across operations as shown in Figure 5.7 with a constant 13% in DepConvs with the remaining 87% carried out in Conv operations.

As discussed previously, we found that DepConv operations are less efficient than Conv operations, with 3x3 DepConvs being the most efficient within DepConv operations of different filter sizes. Combined with the insight that reducing width focuses computation in DepConvs, we found that lower width scalings have high inference latency compared to higher width scalings under the same MAC budget, as shown in Figure 5.7 (Right). As observed the proxyless-w1.00-r160 and proxyless-w0.65-r224 models have approximately the same amount of MACs at 160M. However, the lower width proxyless-w0.65-r224 model has 18% computation focused in depthwise convolutions compared to its counterpart which has 13% computation in depthwise convolutions.

From this we derived the insight as below

> *For any number of architectures derived for a particular MAC budget through the conventional approach of scaling, the scaling with the highest width would have the lowest latency*

5.1.4 Internal Memory vs TinyOps

Utilising insights gained from our analysis of operation throughput on MCUs and the pareto frontier, we manually derived balanced scalings of the MobileNetV3 model from the TinyOps design space to compare with models designed for internal memory.

TABLE 5.1: Accuracy, Latency and Energy per Inference comparison of models from TinyOps and Internal Memory Design Space

Platform	Model	Design Space	MACs (M)	Params (M)	Acc (%)		Latency (ms)	Power (mW)	Energy (mJ)
					FP32	INT8			
L552	Proxyless-w0.10-r192	Internal	8	0.19	28.84	27.01	855	**65**	55.6
	MobileNetV3-w0.25-r080	**TinyOps**	3	0.50	31.96	**30.00**	**230**	145	**33.4**
F469	Proxyless-w0.30-r144	Internal	21	0.72	51.91	50.67	1089	**290**	316
	MobileNetV3-w0.50-r112	**TinyOps**	16	1.33	54.91	**52.37**	**674**	410	**276**
	MCUNetV1-F469	Internal	67	0.73	60.8	59.47	2547	**335**	853
	MNASNet-w1.00-r080	TinyOps	48	4.38	61.21	60.83	2146	435	933
	MobileNetV3-w0.75-r128	**TinyOps**	44	2.49	63.06	**62.58**	**1442**	395	**570**
	MCUNetV1-F469-int4	Internal	135	1.4	-	62.00	-	-	-
	MCUNetV2-F469	Internal	119	<1	-	64.90	**-**	-	-
	MobileNetV3-w1.00-r160	**TinyOps**	**111**	3.96	68.84	**68.19**	3472	405	1406
F746	Proxyless-w0.30-r176	Internal	32	0.72	54.79	53.68	686	**645**	442
	MobileNetV3-w0.55-r128	**TinyOps**	28	1.55	58.94	**58.29**	**460**	805	**370**
	MCUNetV1-F746	Internal	82	0.74	62.5	61.47	1131	**690**	780
	MCUNetV1-F746-int4	Internal	170	1.4	-	63.50	-	-	-
	MobileNetV3-w1.00-r112	**TinyOps**	59	3.96	64.57	**64.02**	**732**	805	**589**
	MNASNet-w1.00-r128	TinyOps	104	4.38	68.09	68.01	1367	775	1059
	MobileNetV3-w1.00-r160	TinyOps	**111**	3.96	68.84	**68.19**	**1307**	795	**1038**

As shown in Table 5.1, the MobileNetV3 scalings outperformed optimal scalings of a ProxylessNAS model derived for internal memory constraints of the L552, F469 and

F746 in addition to the MCUNetV1/2 models derived via NAS on the F469 and F746. On the F746, we outperformed MCUNetV1-F746-int8 with 2.5% higher accuracy and 1.5x lower latency with the MobileNetV3-w1.00-r112 model. On the F469, we observed that the MobileNetV3-w0.75-r128 architecture outperformed MCUNetV1-F469-int8 with 3% higher accuracy and 23M lower MACs to achieve 1.8x lower inference latency.

As latency metrics for MCUNetV2 and 4-bit MCUNetV1 models were not published, we compared with accuracy and MACs for these models. As can be observed, on the F469 and F746, the MobileNetV3-w1.00-r160 scaling achieved 4.7% and 6.2% higher accuracy with 24M and 49M lower MACs than MCUNetV1-F469-int4 and MCUNetV1-F746-int4 respectively. Similarly, compared to MCUNetV2-F469-int8, the MobileNetV3-w1.00-r160 yielded 4% higher accuracy with 8M lower MACs. We note that as the approach of MCUNetV2 uses a NAS algorithm similar to MCUNetV1 to derive models, which maximises computation in an operation agnostic manner, our study suggests MCUNetV2-F469-int8 would have significantly higher latency than the MobileNetV3-w1.00-r160 model even under the same MAC budget due to the usage of inefficient 5x5 and 7x7 DepConv operations.

We note that the MCUNetV1 models were deployed with the TinyEngine (Lin et al., 2020) inference framework which utilises optimised kernels up to 1.5x faster than the CMSIS-NN kernels we utilised. It is likely that using the same set of kernels the models deployed with TinyOps would have even faster inference latency in comparison, however, we leave this integration and experimentation as a future work.

5.1.5 Discussion

Through our analysis of model deployment at the micro-architecture level, we were able to derive model design heuristics for MCU platforms. We additionally showed the limitations of prior works that attempt to achieve the highest accuracy within the constraints of the internal memory design space. We utilised the design heuristics to manually derive models from the TinyOps design space which outperformed internal memory models in accuracy, latency and energy efficiency.

Although, we were able to demonstrate the strength of the TinyOps design space, a limitation of this manual approach was the high search cost due to the trial and error involved. On average, 3 trials were required to derive each model where each trial consisted of training and evaluating the candidate model. This approach resulted in a high search cost of 120 GPU hours to derive each model. In the next section, we explore the low-cost NAS to derive static and dynamic DNNs with a low search cost to meet varying latency constraints.

5.2 Accuracy-Latency Trade-Off through Dynamic and Static Models

We demonstrated how models on the accuracy-latency pareto frontier could be derived from the TinyOps design space with better performance compared to the internal or external memory design space. However, these models were derived for fixed latency constraints which yielded better energy efficiency than the internal memory models. Additionally, the manual approach used to derive the models had a high search cost. In this section, we explore how we can efficiently derive models across the pareto frontier and trade-off the accuracy and latency at run-time on MCU platforms. To achieve this we explored dynamic and static approaches to adapt the inference at run-time.

A static approach to meeting multiple latency constraints would involve deriving multiple independent models of varying complexity from the TinyOps design space. The inference latency at run-time could then be adapted by selecting the model to perform inference with. The overhead of this approach would be the storage required for the independent models and interpreters in addition to the cost of training the models independently. We show that this static approach to meet varying latency constraints can be implemented by using external memories, which are large enough to store a number of independent models. However, the cost of training the models independently remains high with this approach.

On the other hand, a dynamic DNN is a single model that is able to adapt its complexity at run-time. As such, a dynamic approach generally has a reduced training time compared to training multiple individual models. As such, we explored an approach to realise dynamic DNNs on MCU platforms.

5.2.1 Dynamic DNNs via Early-Exiting

Dynamicism in DNNs is achieved through either dynamically adapting the parameters or structure of the model at run-time as we discussed in Section 2.2.2. The applicability of dynamic parameter approaches is limited in the case of MCUs as the quantisation schemes and kernels assume that the weights are fixed during inference. A common approach is to dynamically select the number of channels or width. However, this leads to filters occupying non-contiguous space in memory which is incompatible with current kernels. Similarly, such approaches use multiple batch normalisation parameters for different width configurations which would be not be applicable with compiler optimisations such as operator fusion (Conv+BN+ReLU).

In our work, we explore how depth can be altered at run-time to meet varying latency constraints using early-exiting neural networks (EENNs). EENNs append intermediate classifiers at different points in the network which can be used to make an early

classification by running a subset of the layers of the network. In our work, we study EENNs for *anytime classification*, where an exit is performed on the basis of a compute or latency constraint.

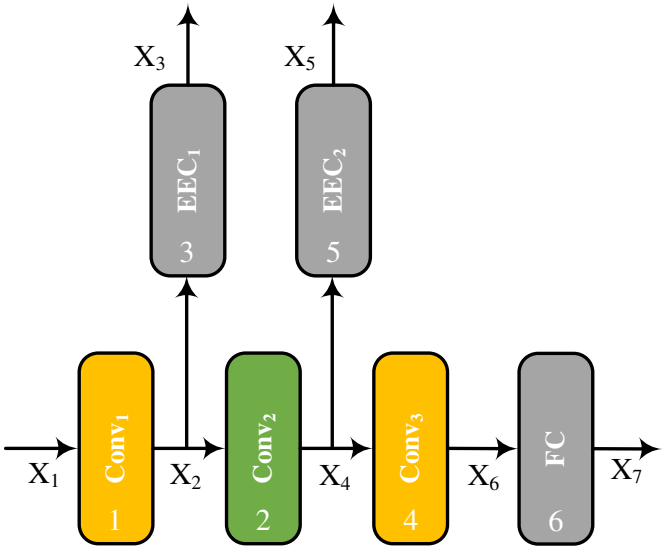


FIGURE 5.8: Structure of an Early-Exiting Neural Network (EENN)

An EENN that is able to meet three latency constraints is shown in Figure 5.8. As can be observed, the backbone of the EENN contains three convolution operations. The three classifiers in the model create three subnetworks in the EENN that can be executed to meet three latency constraints. The highest latency is incurred when the entire backbone is run and classification is performed with the final classifier, *FC*. In this scenario, the early-exit classifiers (EECs), EEC_1 and EEC_2 are ignored. To meet more stringent latency constraints, a fraction of the stem can be computed and inference can be completed at an earlier stage by exiting at one of the EECs, EEC_1 or EEC_2.

5.2.1.1 Deployment of EENNs

The EENNs were compatible with existing kernels and the partitioning strategy of TinyOps as they did not alter the data layout for any operations in the inference graph. However, to realise the early-exiting required a reordering and partial execution of the inference graph. The inference graph contains an ordering for the operations which is utilised by the interpreter to perform the operations in a valid sequence. Additionally, the memory planner of the inference framework uses the ordering to statically allocate memory for intermediate activation tensors according to their lifetime.

The ordering for a simplified three layer CNN with two intermediate classifiers can be observed in Figure 5.8. Trivially, the first operation in the graph is the $Conv_1$ operation which consumes the tensor, X_1 to produce output, X_2. However, after $Conv_1$, as there is an early-exit classifier (EEC) placed after the first operation, there is a branch in the inference path with two operations in either branch that consume the tensor X_2. The memory planner adopts a greedy approach to operation scheduling to reduce the memory usage, where it performs all operations that consume a tensor so that the allocated memory can be freed as early as possible. As such, the $Conv_2$ and EEC_1 operations are ordered as the second and third operations in the inference graph. This allows the memory for the intermediate tensor, X_2 to be freed as soon as $Conv_2$ and EEC_1 are performed. A similar ordering is observed for the second EEC in the network as well.

Partial Execution of Inference Graph In a normal operating scenario where all nodes in the inference graph are executed, the interpreter sequentially performs the operations in the graph according to their ordering as we showed previously in Algorithm 2. However, ending the inference early by exiting at one of the classifiers requires executing a subset of the operations based on which exit we are taking which is dictated by the latency constraint.

This partial execution was performed by extracting the node ids for all the subnetworks and storing them in an ordered list alongside the inference graph. For the smallest subnetwork in Figure 5.8, this ordered list would contain the elements, [1, 2, 4, 5]. Similar lists, were extracted from the EENN for the remaining subnetworks from the inference graph. To perform inference with only a sub-network, the interpreter was modified to only execute operations in the inference graph from the ordered list for a sub-network as shown in Algorithm 5.

As can be observed, the invoke function takes in a list of ordered lists, *op_lists* that define the operations in the subnetworks and the latency constraint, L as extra inputs. The helper function, GetSubNN, is a function that simply returns the list of operations corresponding to the largest subnetwork that meets the latency constraint. The inference with the subnetwork can then be simply performed by sequentially performing the operations in the *active* ordered list.

We note that the GetSubNN function simply consists of returning a reference to the subnetwork corresponding to a latency constraint, which would have negligible overhead. As such, there would be zero overhead to switch between the subnetworks when adapting inference to varying latency constraints.

Algorithm 5 Partial Graph Inference Pipeline

Inputs: NN_graph, the inference graph. $input$, the tensor input to the NN. L, the latency constraint. $op\_lists[num\_subnets]$, a list of ordered lists containing operations in each subnetwork.

Output: $output$, the tensor output from the NN

```
1: function InvokePartialInference(NN_graph, input, op_lists[num_subnets], L)
2:     SubNN_Graph = GetSubNN(op_lists, L)
3:     for op in SubNN_Graph do
4:         inputs = GetInputTensors(op)
5:         weights = GetWeightTensors(op)
6:         quantParams = GetQuantisationParameters(op)
7:         Kernel = GetKernel(op)
8:         output = Kernel(inputs, weights, quantParams)
9:     end for
10:    return output
11: end function
```

5.2.1.2 EENN Design

We demonstrated how a partial execution of the inference graph could be used to achieve dynamicism on MCU platforms with EENNs. Designing an efficient EENN required selecting an efficient backbone model, the number of early exits and determining their placement which we describe in this section. The design of the backbone model would dictate the accuracy achieved at the final classifier and the early-exit classifiers. As we were looking to maximise the performance across the accuracy-latency pareto frontier, we utilised a NAS based approach to derive the backbone model that yielded the highest accuracy across all the classifiers in the model.

Backbone Model The backbone model in an EENN used for *anytime classification* would correspond to the worst case latency constraint. The number of early-exits and their placement would determine how many and what different constraints could be met.

We utilised a supernetwork based NAS approach to select the backbone model. We utilised the heuristics derived in Section 5.1 to inform the design of the search space. The search space was created using the MobileNetV3 model with hyper-parameters of width multipliers and input resolution used to create the candidate models. The candidate width multipliers ranged from 0.30 to 1.00 with steps of 0.05. The candidate input resolutions ranged from 64 to 224 with increments of 16 in between. This produced a search space of 165 candidate models.

As we were looking to maximise the accuracy-latency frontier, we appended the early-exit classifiers into the supernetwork. These were added after every MobileInvertedConv block. The structure of the early-exit classifier was adapted from the final block of operations used after the last MobileInvertedConv block in the MobileNetV3 model shown in Figure 5.9.

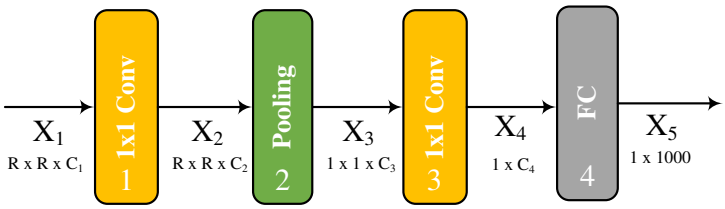


FIGURE 5.9: Structure of Early-Exit Classifier

A limitation for using the structure for the EEC shown in Figure 5.9, was that it had a very high computational complexity when placed after MobileInvertedConv blocks early in the network. This was due to the first 1×1 convolution operation applied in the EEC. In models derived from the mobile search space, including the MobileNetV3 model, the input resolution is high in the early layers. This is eventually reduced through the usage of strided convolutions. For EECs appended to earlier MobileInvertedConv blocks in the MobileNetV3 model, we found that that the complexity of the EEC was too high to yield a reduction in the latency. The MobileNetV3 model had a total of 15 MobileInvertedConv blocks. We found that the inference latency could be reduced by using the EEC structure from the $7th$ MobileInvertedConv block onwards.

Training the Supernetwork The supernetwork was trained by building on the approach of Yang et al. (2020) which explore dynamic widths and resolutions in a DNN. However, these are explored in a mobile setting and consider a limited number of candidate width and resolution hyper-parameters which produce only 16 configurations. Further they did not consider any early exiting in the search.

We build on their work to train the supernetwork in our NAS approach with backpropagation of gradients from the EECs appended to a MobileNetV3 backbone model. In the training, we appended EECs after every MobileInvertedConv block. Further, we utilised a larger number of width and resolution hyper-parameters which yield a total of 165 configurations as described previously. The recipe for training is described in Algorithm 6.

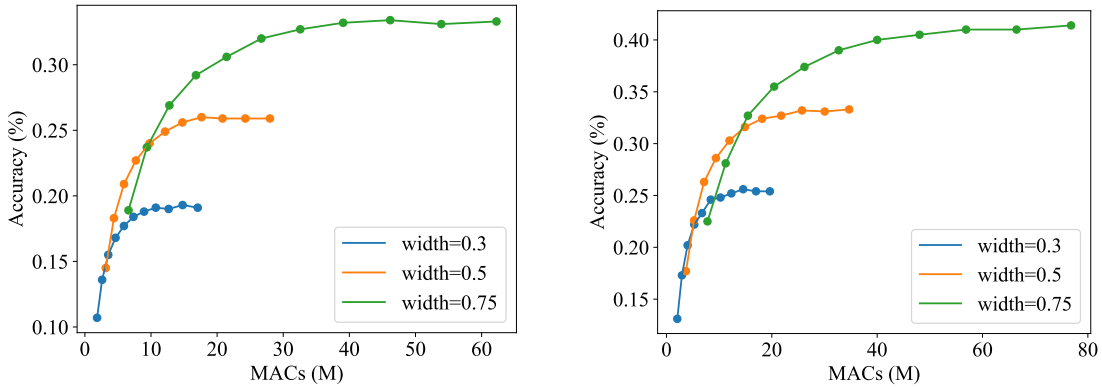
The training algorithm firstly performs a forward pass from the full baseline model, after which it backpropagates gradients from all the N classifiers which are accumulated in Line 10. The next step involves sampling a candidate resolution and two candidate

Algorithm 6 Training algorithm for EENN Supernetwork

1: **for** $t \leftarrow 1, T_{iters}$ **do**
2:
3: $grad = 0$
4: x, y = GETDATAMINIBATCH()
5:
6: M' = SETACTIVESUBNET(M, 1.0, 224)
7: $\mathbf{Y}' = M'(\mathrm{x})$ ▷ $\mathbf{Y}$, Matrix containing logits of all EECs
8:
9: $loss = \sum_{i=1}^{N}$ KL_DIVERGENCE(y'_i, y) ▷ Sum losses of all N classifiers
10: $grad$ += BACKPROPGRAD($loss$) ▷ Backprop gradients from all classifiers
11:
12: $width\_list$ = [MIN($width\_candidates$)]
13: $width\_list$ += UNIFORMSAMPLE($width\_candidate$, 2) ▷ Add two more widths
14: res = UNIFORMSAMPLE($res\_candidate$, 1) ▷ Sample a candidate resolution
15:
16: **for** $width$ in $width\_list$ **do**
17:
18: M' = SETACTIVESUBNET(M, $width$, res)
19: $\mathbf{Z}' = M'(\mathrm{x})$ ▷ $\mathbf{Y}$, Matrix containing logits of all EECs
20:
21: $loss = \sum_{i=1}^{N}$ KL_DIVERGENCE(z'_i, y'_i) ▷ In-Place Distillation
22: $grad$ += BACKPROPGRAD($loss$)
23:
24: **end for**
25:
26: UPDATEWEIGHTS(M, $grad$) ▷ Update weights from accumulated gradients
27:
28: **end for**

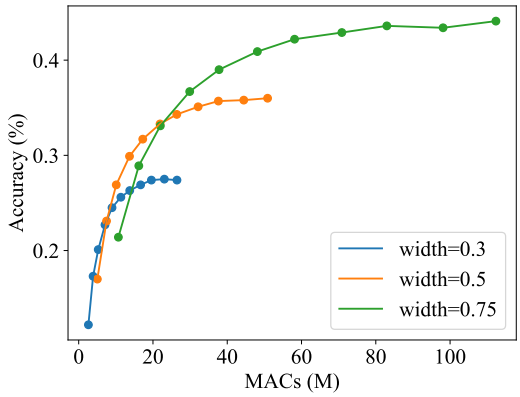
widths in addition to the smallest width to sample three subnetworks in the supernetwork. Yang et al. (2020) built on the approach of Yu and Huang (2019) who proposed always sampling the minimum width to boost the lower bound of performance of the supernetwork. The forward pass for each subnetwork is performed with the the weights inherited from the supernetwork. The gradients are then backpropagated from all classifiers in the EENN supernetwork as shown in Line 22. The gradients accumulated thus far are then finally used to update the weights as shown in Line 26 to complete one step of training. In order to keep the training cost low, we trained the supernetwork for 20 epochs to approximate the final network weights. A complete list of hyper-parameters used to train the super-network is provided in Section D.1 in Appendix D

We evaluated the training of the supernetwork with approximate weights to determine whether it demonstrated behaviour similar to our observations made on the pareto frontier. We evaluated the accuracies of the candidate architectures in the supernetwork and plotted the pareto frontier for different EECs derived by varying the width

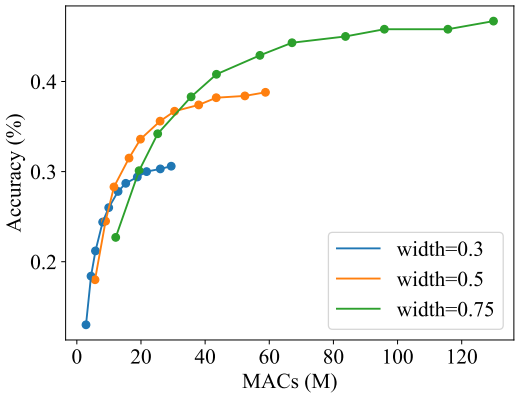


(A) Subnetwork 7

(B) Subnetwork 10



(C) Subnetwork 13



(D) Subnetwork 15

FIGURE 5.10: Accuracy of EECs in the MobileNetV3 search space

and resolution.

We observed in Figure 5.10 that the pareto frontier across the different EECs was composed of different width and resolution configurations. We also observed that the *point of diminishing returns*, at which a particular width multiplier became sub-optimal varied across the EECs. This would imply that the optimal backbone model derived by the width and resolution would be different for each EEC.

5.2.1.3 Model Search and Training

We utilised a simple linear search algorithm to derive architectures for varying complexity constraints. As the number of candidate models in the search space was small, we were able to build a look-up table of the on-device inference latency and validation accuracy of the candidate models in the search space. As we were looking to maximise the accuracy-latency pareto frontier, we defined the quality objective to be the sum of the accuracies across the EECs defined as below, where N is the number of classifiers in the network. As mentioned previously, this was

$$Acc = \sum_{i=1}^{N_{EECs}} Acc_i \tag{5.4}$$

For a given latency constraint, the search algorithm performed a linear search that returned the backbone model with the highest sum of accuracies across the EECs. In our initial evaluation, we measured the latency statistics using the backbone model and the early-exiting subnetworks using external memory. Once a model was derived it was trained from scratch with EECs placed from the $7th$ MobileInvertedConv block onwards. This produced an EENN with 9 usable latency constraints as the MobileNetV3 model had a total of 15 MobileInvertedConv blocks. The EENN was trained in a joint manner as described below

$$Loss = \sum_{i=1}^{N} L\left(\mathbf{y_i}, \mathbf{y}'\right) \tag{5.5}$$

where N is the number of classifiers, $\mathbf{y_i}$ is the output of the ith classifier in the EENN, $\mathbf{y}$ is the target label and the cross-entropy loss was used as the loss function, L.

5.2.1.4 Evaluation

We ran the search for user defined latency constraints using the linear search algorithm. For the initial evaluation, we compared with the manually derived models and scaled variations of the ProxylessNAS model when deployed with an external memory configuration instead of with TinyOps. We note that the optimal comparison would be to deploy the EENNs with the TinyOps inference framework. However, this would require modifications to the inference pipeline of the TinyOps inference framework. The models were deployed with the external memory configuration as an intermediate step to compare the accuracy and latency of the early-exits in the EENN to the manually derived scaled variations of the MobileNetV3 models with the assumption that the same observations would hold when the models were deployed with TinyOps.

In this experimental setup, we derived the EENNs for a worst case latency constraint of 1100ms and 1600ms on the F746. The derived EENNs were then trained from scratch using the joint training recipe discussed previously. The performance of the derived EENNs in comparison to the scaled variations of the ProxylessNAS and MobileNetV3 models is shown in 5.11.

As can be observed the EENNs derived the MobileNetV3 backbone had better performance across a range of latency constraints compared to the ProxylessNAS pareto

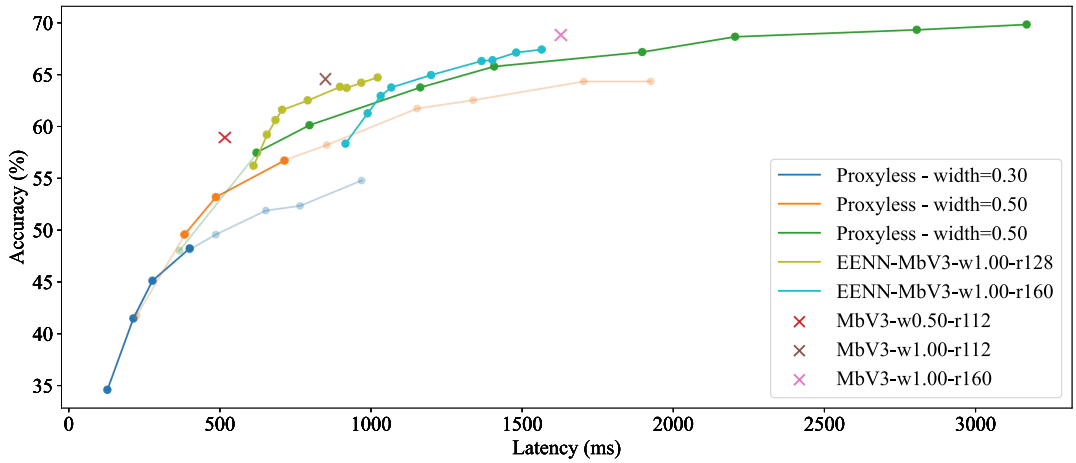


FIGURE 5.11: Performance of EENNs

frontier. However, they fell below the manually derived MobileNetV3 models in performance. In part, this was due to the joint training scheme used for the EENNs producing a reduction in accuracy. As can be observed the individually trained MbV3-w1.00-r160 model had a validation accuracy of 68.19%. However, when this model was converted to an EENN by appending the EECs in the backbone model, the validation accuracy at the final classifier dropped to 67.4%. Towards, the earlier classifiers in the backbone, we note that the performance fell off quite sharply, even dropping below the ProxylessNAS pareto frontier. We believe this was due to the search space not being diverse enough to contain architectures that yielded high accuracy at the earlier EECs. In future work, we believe the search space could be diversified by including more hyper-parameters in the search space such as the number of channels and expansion ratio. This might enable us to derive models that yield better accuracy at the EECs whilst retaining the same accuracy at the final classifier.

While, the EENN approach did have the benefit that the joint training allowed training of only a single model, the limitations of performance lead us to explore a static approach to achieve an accuracy-latency trade-off at run-time.

5.2.2 Static Models for Accuracy-Latency Trade-Off

In addition to the dynamic approach, we explored a static model based approach to trading-off the accuracy and latency at run-time. This approach would be realised by deriving multiple models for a number of latency constraints that would be trained and deployed independently. The usage of external memories would allow the storage of multiple independent models. These could be configured at initialisation with multiple interpreters and paritioning schemes to enable fast inference with TinyOps.

The MCU platforms we utilised, had an address space supporting up to 256MB of additional SDRAM and Flash storage. We observed that the state-of-the-art CNN models required up to $5MB$ of storage and $1.6MB$ of memory. Assuming that each model derived for a latency constraint had a memory footprint of $5 + 1.6 = 6.6MB$, this would allow the support for $256/6.6 = 38$ latency constraints. To switch between the models at run-time would require multiple interpreters to be declared and configured with the corresponding models at run-time to perform inference. Similarly, the interpreters would have independent partitioning strategies for the corresponding models which would be determined at initialisation.

5.2.2.1 Search and Evaluation

To search for static models, we reused the EENN supernetwork trained for 20 epochs used to search for EENNs. We attempted to train a supernetwork without the EECs, however, we found that this trained slowly. On the other hand, a benefit of the EECs in the EENN supernetwork was they also acted as auxiliary heads which injected gradients into the network and sped up training. We opted to use this approach as the weights after 20 epochs of training would yield a better approximation of the fully trained network weights.

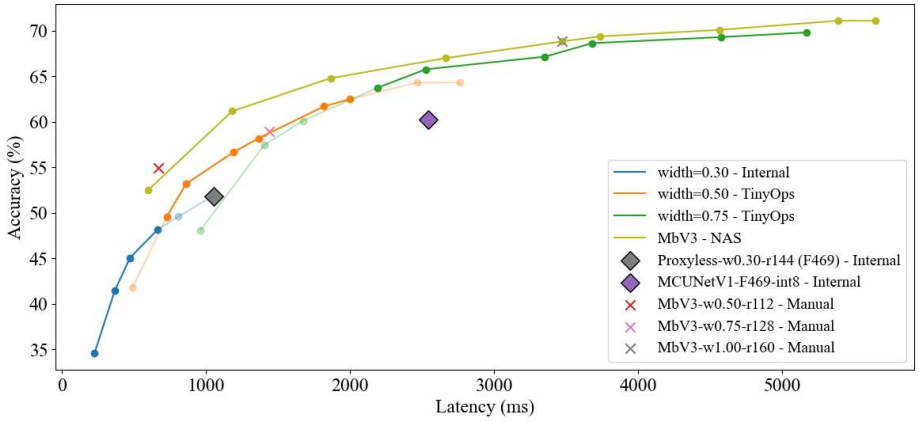
As we were searching for independent models the quality objective was changed to simply measure the accuracy at the final classifier. The search algorithm simply consisted of a linear search that took a latency constraint as input and returned the configuration for the subnetwork in the supernetwork that had the highest validation accuracy.

Using this approach, we derived models for a number of latency constraints ranging from 250ms to 2500ms. The models were derived for the F746 platform by gathering the latency statistics of the models deployed with TinyOps. Performance of the models derived for the latency constraints are shown in Table 5.2. We also compared the performance of the models with the previously manually derived MobileNetV3 and ProxylessNAS scalings on the F469 and F746 device which is shown in Figure 5.12.

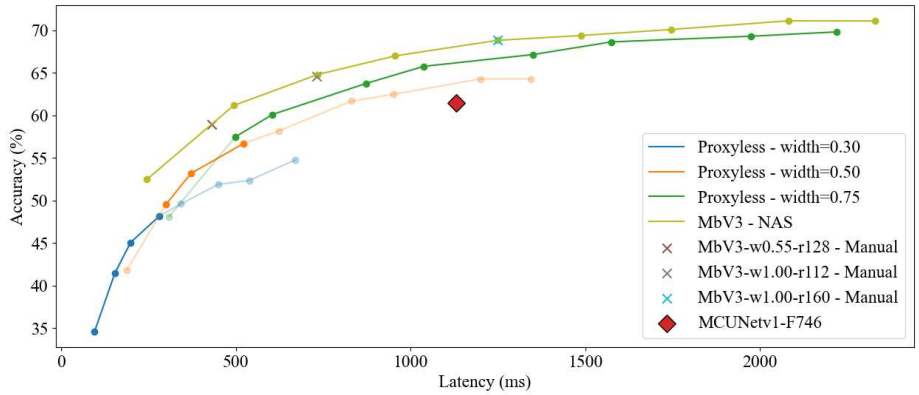
As can be observed, the NAS approach to deriving the models outperformed the ProxylessNAS models on either device. The performance difference between the ProxylessNAS models was higher on the F469 and the F746. We note that although the models were derived according to on-device inference latency of the F746, they transferred well to the F469 device as well. However, it is likely that further gains could be made by deriving models for the inference latency of the F469. Compared to the manually derived models, the NAS derived models had comparable performance. This would indicate that the heuristics were utilised effectively to manually derive the balanced scalings of the MobileNetV3 model.

TABLE 5.2: Static Models Derived for a Range of Latency Constraints throught Light-Weight NAS using MobileNetV3 search space. Models were derived for the inference latency of the F746.

Width	Res	Params (M)	MACs (M)	Accuracy (%)		F746		F469	
				FP32	INT	Latency (ms)	Energy (mJ)	Latency (ms)	Energy (mJ)
0.45	112	1.15	15	52.55	51.66	261	210	623	249
0.65	128	1.99	35	61.29	60.58	520	421	1223	477
0.75	144	2.49	57	64.81	63.74	762	617	1924	798
0.90	144	3.35	82	67.01	66.43	993	799	2736	1163
1.00	160	3.96	111	68.84	68.19	1306	1038	3472	1406
0.95	176	3.65	132	69.51	68.74	1561	1272	3847	1539
1.00	192	3.96	159	70.19	69.35	1830	1482	4691	1853
1.00	208	3.96	192	71.32	70.67	2182	1767	5545	2218
1.00	224	3.96	216	71.16	70.37	2448	1983	5820	2270



(A) F469



(B) F746

FIGURE 5.12: Performance of Static Models Derived via Light-Weight NAS for Varying Latency Constraints

Search Method	Search Cost		Training Cost	Total Cost
	Training	Search		
Manual	$40 \cdot 3 \cdot DN$	0	0	$120 \cdot DN$
EENN-NAS	50	1	$40 \cdot D$	$51 + 40 \cdot D$
Static-NAS	50	1	$40 \cdot DN$	$51 + 40 \cdot DN$

TABLE 5.3: Compute Cost of Different NAS Approaches

5.2.3 Deployment Cost

The major benefit of utilising the light-weight supernetwork based NAS approach, comes from the reduced computional cost of deriving the models. This cost can be broken down into the time for training the search space, searching for architectures in the space, and training of the models for deployment onto the MCU devices. We compare these costs for a number of different approaches in Table 5.3 where **D** is the number of devices for which the models are to be derived, and **N** is the number of latency constraints that are to be met.

As can be observed in Table 5.3, the cost of training the search space was the highest for the manual approach. On average, the hit and trial nature of the manual approach to deriving models for a particular device and latency constraints required training and evaluating 3 models for a particular latency constraint, where the training required 40 GPU hours. As models were trained to convergence the search process yielded a fully trained model, due to which there was no separate training cost incurred once the model was derived. As such, the total cost for the manual approach to derive models for D devices and N latency constraints was $120DN$.

To derive EENNs, we adopted a supernetwork based approach where the training and searching of the supernetwork was decoupled from the training of the derived model. As mentioned previously in Section 5.2.1.2, the supernetwork approximated the fully and independently trained weights of candidate architectures in the search space by training weights shared between the candidate architectures for a limited number of epochs to reduce the computational cost. Using this approach, the supernetwork only needed to be trained once for any number of devices and latency constraints which required 50 GPU hours. After the supernetwork was trained, the accuracy of all candidate architectures in the search space was evaluated. This required a total of 1 GPU hour. The training and evaluation of the the supernetwork therefore had a one-time total compute cost of 51 GPU hours. To derive a model, the search as described in Section 5.2.1.3, consisted of a simple greedy search with a negligible search cost ($<$ 10 CPU seconds). Once the model was derived it was then trained from scratch requiring 40 GPU hours. We note that the derived EENN was able to meet multiple latency constraints by design. As such, multiple models did not need to be derived for multiple latency constraints. However, the optimal EENN would vary for each device which

would require multiple models to be derived and trained for each device. For this deployment scenario the total training cost would be $40D$, for D devices with a constant 51 GPU hours to train and evaluate the supernetwork.

For the static approach which derived multiple independent models to meet multiple latency constraints, the cost of training and search associated with the supernetwork was the same as for the approach deriving EENNs. However, this approach derived multiple models which had to be trained independently. As such this had a training cost of $40DN$ for D devices and N latency constraints, as training one model required 40 GPU hours.

We note that the cost of training the derived models was higher for the static NAS approach compared to the EENN approach as it required training multiple independent models. We showed that this approach yielded zero-switching time and high performance and could be realised under the external memory size constraints of current devices. However, to reduce the training time, it would still be desirable to train and deploy a single model that could adapt its performance on the fly. The EENNs we derived had this desirable quality, but did not achieve high predictive performance. We believe the search space could be made more diverse to derive architectures that might yield a better pareto frontier. This would likely have a higher cost of training the supernetwork, but could be amortised through multiple deployment scenarios.

5.3 Discussion

In this chapter, we explored how we could achieve an efficient accuracy-latency trade-off at run-time on MCUs. We studied the pareto frontier to show that prior approaches that derive models for fast internal memory suffer not only from low accuracy, but surprisingly, high inference latency as well. We showed that this was due to size constraints of internal memory and operation agnostic model design adopted by prior approaches. We performed a latency analysis of CNN deployment to MCUs at the micro-architecture level to derive heuristics for model design.

We proposed a framework for deriving models across the accuracy-latency pareto frontier and also perform the trade-off at run-time with zero switching time. The framework utilised a supernetwork based approach which utilised our model design heuristics to design the search space with a low search cost. We derived multiple models across the accuracy-latency pareto frontier for varying latency constraints which outperformed models designed for internal memory with up to 3% higher accuracy and 1.8x lower latency.

A limitation of our static approach to adapting inference was that it had a high training cost for the multiple independent models that were deployed. We explored a dynamic approach utilising EENNs, with a lower training cost, however, we found that these were lacking in performance at the earlier exits. We believe that performance gains could be achieved by utilising a larger search space which would include hyperparameters such as the number of channels and expansion ratio in the MobileInvertedConv blocks.

Chapter 6

Conclusions

The proliferation of IoT devices in society has opened significant opportunities for applications involving smart cities, healthcare and agriculture amongst others. In this area, the field of TinyML is a cost and energy efficient approach to address the limitations of security, privacy, latency and connectivity of the cloud computing paradigm by providing intelligence on the device with inference performed locally. However, DNN inference workloads have traditionally been deployed on mobile or GPU platforms. As such, the significantly different and constrained hardware characteristics of MCU based platforms requires the design of a highly optimised inference system which is able to extract the maximum performance from the devices.

6.1 Summary of Contributions

In this thesis, we studied how we could enable deep learning inference on the accuracy-complexity pareto frontier on low power and resource constrained edge devices. A number of approaches have been proposed at different layers of the inference stack to optimise the deployment of DNNs. In this thesis, we performed optimisations across the inference stack as shown in Figure 6.1.

Our work studied the design of efficient models and inference frameworks while considering the interplay between the different components of the inference stack including model design and optimisation, the inference framework and the target hardware. We achieved high performance across the accuracy-complexity frontier with the contributions as below

- **DEff-ARTS:** We proposed a gradient based method to trade-off the accuracy and complexity in an automated manner from a cell-based search space

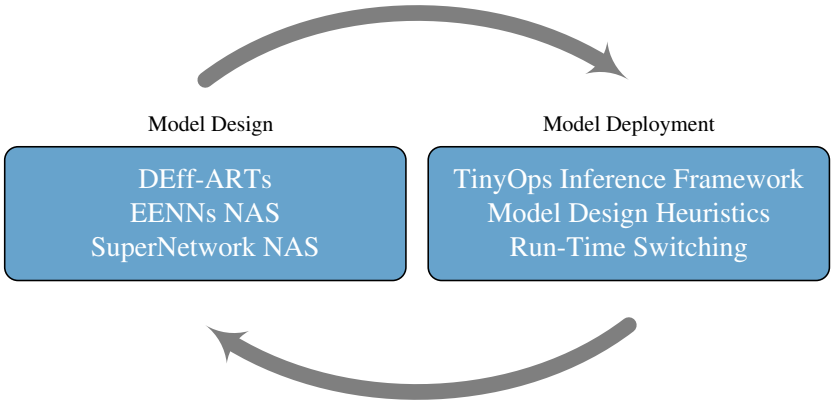


FIGURE 6.1: Summary of Contributions in Model Design and Deployment to achieve High Performance Across the Pareto Frontier

- **TinyOps Inference Framework:** We developed the TinyOps inference framework to combine the advantages of internal memory (low inference latency) and external memory (high accuracy). We deployed models derived from the mobile search space with TinyOps to acheive record TinyML ImageNet performance
- **MCU Deployment Analysis:** We performed an in-depth study of model deployment on MCUs at the micro-architecture level to derive novel model design heuristics and demonstrated that the internal memory design space is sub-optimal for accuracy, latency and energy efficiency.
- **SuperNetwork and EENN NAS:** We utilised the derived heuristics to inform search space design for a super-network based NAS approach to derive dynamic and static models to enable run-time trade-off between accuracy and complexity. Using this approach we acheived state-of-the-art TinyML ImageNet classification performance across the accuracy-complexity pareto frontier.

Our work achieved multiple milestones in performing efficient deep learning inference on edge devices. The TinyOps inference framework introduced a novel approach to inference on MCUs that significantly lifted the ceiling of performance on MCUs by achieving 1.4x-2.5x lower inference latency than existing approaches. Similarly, TinyOps opened a new design space on MCUs from which we derived models using our design heuritics that outperformed prior works with up to 3% higher accuracy and 1.8x lower inference latency.

6.2 Future Work

Moving forward, unlocking further performance on edge devices requires continuing efforts across the inference stack. These are informed by current work that has been performed in the field of TinyML as well as larger scale cloud based deep learning from which design principles and insights can be transferred to edge devices.

6.2.1 Search Space for EENNs

We derived model design heuristics to inform the search space design which was used to search for EENNs and static CNNs. In our search, we utilised two degrees of freedom to reduce the size of the search space. When deriving EENNs, we found that the derived models, whilst being competitive in performance, ultimately fell short in performance. We believe that including further hyper-parameters in the search space could lead to better performance. Similarly, we showed that the static CNNs derived from the search space achieved record performance. We believe expanding the size of the search space could lead to better performance. However, the subsequent gains in performance would likely come at an extra search cost incurred in training and navigating the search space.

6.2.2 Moving Away From CNNs

The recent success of transformer based architectures has sparked a tremendous amount of research effort in the area. Typically, these models have a very large number of parameters which make them unsuitable for deployment on MCUs with the conventional approaches utilising internal memory (Li et al., 2022). We believe the TinyOps approach to inference could be utilised to realise efficient transformer deployments on MCUs.

In the context of CNNs, we adopted a partitioning strategy along the H dimension of the tensor to reduce the tensor sizes. This also allowed us to gather the output tensors of the tiny output tensors with an in-place concatenation operation with minimal overhead. The same methods can also be applied to linear layers which dominate recent transformer based models. For matrix multiplication operations of the form $X \times W$, the activation and weight tensors can be partitioned across the rows (H) and columns (W) dimension respectively into tiny tensors. Due to the structure of matrix multiplication operations, this would produce tiny output tensors which could be efficiently reduced through concatenation to produce the final output and perform matrix multiplication operations with lower memory requirement. Such solutions would further expand the design space for efficient inference on MCUs.

6.2.3 Improving MLOps Tools

A significant part of this research involved studying how inference frameworks could effectively utilise external memories and other peripherals to enable efficient inference. We showed that the limitation of current inference frameworks which utilised internal or external memory exclusively for the weights and activations lead to design and deployment of models with sub-optimal accuracy and latency as well. To address this limitation, we developed TinyOps which enabled deployment from an efficient design space to achieve state-of-the-art inference performance. In industry, the current inference MLOps tools are structured such that the inference frameworks responsible for interpreting the model and memory management, such as TfLite-Micro, STMXCubeAI and TinyEngine, are decoupled from the kernels responsible for running low-level operations on the hardware. While the inference framework is designed to be hardware agnostic to enable portablility, the kernels are closely coupled with the hardware and typically provided by the hardware vendors. To support variations in the characteristics of a hardware family supplied by a vendor, the kernels are often written in a generic manner and are reconfigurable in software through compiler directives.

We believe our work proposing the TinyOps inference framework could have a significant impact in improving current MLOps tools and realising efficient DNN deployment in TinyML. In our work, we developed TinyOps as an intermediary layer in the inference stack that interfaced with the kernels and the interpreter in the inference framework, as shown in Figure 4.5. To improve current workflows, the TinyOps inference framework, available at `https://github.com/sulaimansadiq/TinyOps` could be used as a drop-in replacement for any pipelines utilising TfLite-Micro which is one of the most widely used frameworks in TinyML. This would be possible since we built TinyOps into the baseline TfLite-Micro inference framework and retained all of the original features by design. In our work, this design decision also allowed us to utilise industry standard tools for model training and optimisation such as PyTorch and TensorflowLite.

To support different inference frameworks used in MLOps, the TinyOps partitioning engine could be integrated directly into other inference frameworks using an approach similar to the one adopted in this research. Alternatively, the partitioning engine could be integrated with the underlying kernels that are provided by hardware vendors. This would require minor changes to the memory allocation procedures in the inference frameworks and kernels. Currently, the inference frameworks are typically responsible for allocating memory requested by kernels for memory buffers such as the im2col buffer or input and output tensors. To support the TinyOps approach, the kernels would need to request additional memory for the overlaying buffers. In this case, the partitioning strategy would also be managed by the kernels. These overlaying capabilities could be kept configurable through compiler directives so that they could be

modified according to varying user requirements and platform characteristics. This approach would be beneficial as it would allow a variety of inference frameworks to take advantage of the overlaying scheme integrated in the kernels with minimal changes.

6.2.4 Next-Generation of Hardware

The design of higher layers in the inference stack including the model design, optimisation and training are informed by the capabilities of the underlying hardware. Similarly, the design of the hardware is often informed by the requirements of the particular algorithm that is implemented on it. In this thesis, we focused on current generation hardware and its constraints related to memory, instruction set and peripherals. We believe the work carried out in this thesis and other parallel research provides insight into useful avenues of the development of the next-generation of hardware.

Enhanced On-Chip DMA In Section 4.3.4, we discussed a limitation of the TinyOps inference framework which was imposed by the limitations of the on-chip DMA. As the DMA only supported transfer of contiguous blocks of memory between different memory regions, the TinyOps framework partitioned tensors along the H dimension which corresponded to the tiny tensors occupying contiguous blocks of memory that could be moved by the DMA. With this approach, the minimum size of the tensor would be limited by $K \cdot W \cdot C$ where K is the kernel size, W is the width of the representation and C is the number of channels of the tensor. For larger networks, this scheme produced tiny tensors that could not be partitioned to meet the internal memory constraint when W or C were high. This resulted in a smaller model being deployed with reduced accuracy as shown in Table 4.6. The size of the tiny tensor could further be reduced by partitioning in an additional dimension, such as the width (W) of the tensor, however this would require interleaved copying which was not supported by the on-chip DMA. While an attempt could be made to perform such data movement via software based methods, this would have overheads such as multiple calls to the DMA, in addition to extra cache maintenance and interrupt servicing so this was avoided. Natively supporting interleaved copying in next-generation MCUs at the hardware-level would avoid overheads in software and allow for finer-grain partitioning of the tensors. This would allow further reduction in the internal memory requirement for inference with the TinyOps inference framework and enable inference with larger models to achieve better performance on devices with smaller internal memories.

Reduced Internal Storage In the conventional approach to memory management, the inference frameworks used the entirety of internal storage to store the model weights and inference code which lead to a high internal storage requirement. On the other hand, TinyOps utilised external storage to store the model weights and only used the

internal storage to store the inference framework program code which accounted for approximately $< 10\%$ of the storage footprint. This reduction in internal storage would allow the usage of MCU variants with less internal flash which would lower the bill of materials as internal storage can be expensive to manufacture. As an example, we compared the bill of materials for the internal memory and TinyOps approach for an ARM® Cortex® M4 deployment scenario to find that at the time of writing, the cost of using an STM32F469NE by STMicroelectronics with 320KB of internal flash, supplemented with 8MB of external SDRAM and NOR Flash was $18.87, and actually $0.52 lower than using an STM32F469NI with 2048KB of internal flash. The internal storage could be lowered even further from 320KB for an additional reduction in the bill of materials. We could expect similar cost benefits on different platforms in the MCU family.

Native Support for Low-Precision Arithmetic In Section 2.4.3 we discussed details of the implementation of the kernels that are responsible for carrying out the low-level convolution operations by interfacing with the hardware through the instruction set. As discussed previously, 8-bit precision is the industry standard adopted for quantisation of CNNs as this provides low accuracy drop whilst accelerating inference latency. However, as the hardware only supported multiplication and accumulation with 16-bit operands, an extra step was performed in the kernels where all operands were sign extended from 8-bit to 16-bit. This step involved overheads of extra memory accesses to get data into the correct format for processing. Adding support for native 8-bit multiplication and accumulation would reduce compute and memory overheads associated with formatting the data in the kernels whilst also giving a compute benefit as lower precision operations can typically be performed at higher throughput. Further, while the current standard for deployment on MCUs in 8-bit quantisation, recent research into sub-byte quantisation (Rouhani et al., 2023) could also be utilised to further reduce the data access latency and yield better performance. This could be added into hardware by supporting efficient sub-byte load and arithmetic operations such as multiply and accumulate.

6.2.5 Applications in Other Architectures

In this thesis, we focused on low-power devices such as DSPs or MCUs which have single cores and limited internal storage and memory. However, we believe many of the insights from this research would be applicable to other architectures as well such as GPU or multi-core architectures. This would be possible since the model of the hardware we assume in Figure 4.1 is transferable to other devices without much difficulty. One major difference in the hardware model assumed in this work with other architectures is that MCUs have single cores, while other architectures may have multiple cores

for parallelisation of compute. However, as we discuss below, this difference does not preclude application of this research to other hardware platforms.

There are instances of follow-up work which employs some of the principles used in this research. A recent work on multi-core mobile devices accelerated inference by moving in data on demand from external to internal memory to bypass internal memory size constraints (Alizadeh et al., 2024). However, this work does not employ data overlapping or partitioning of the tensors. As the mobile devices have a similar hierarchy and also have DMA peripherals on-board, the method of partitioning and overlaying proposed in this thesis would be directly applicable to such devices to reduce internal memory usage and accelerate inference. The overlapping between compute and memory access would serve as another level of parallelism performed in tandem with compute parallelism achieved through multi-core processing. The multi-core compute parallelism could be viewed as being similar to the compute parallelism achieved on MCUs through SIMD operations where two operations are computed in parallel on the same core as discussed in Section 2.4.1.1. We could expect differences in the extent to which memory and compute could be overlapped as multi-core processors would likely achieve higher compute parallelism. This could lead to a different amount of inference acceleration achieved when utilising the partitioning approaches on these platforms. For example, if the compute time is reduced enough through higher parallelisation, we might move to a memory bound regime, whereas in this research inference was in the compute bound regime. Nevertheless, the application of the partitioning approaches to multi-core architectures would still be able to achieve inference acceleration which makes this an interesting avenue of future work.

Partitioning methods similar to those used in this research have also been adopted in follow-up work based on GPUs to minimise memory requirement of matrix multiplication operations encountered in attention blocks in transformer models (Dao et al., 2022). However, these approaches were adopted to allow training of large transformer models. For GPU based deployments, other factors would come into play when accelerating inference with such partitioning approaches such as the model serving requirements and inference stage. Whereas in TinyML we consider a unit batch size and look at inference latency, GPU based model serving has different characteristics and requirements. For example, model serving on GPUs often utilises larger batch sizes to maximise weight reuse in internal memory and optimises metrics such as throughput per user in addition to inference latency. Similarly, inference in transformer models is often divided into prefill and decode stages where each stage of inference has different tensor sizes and performance bottlenecks. In such cases, partitioning approaches could be applied by changing the partitioning dimension to other dimensions, such as the batch dimension, in order to optimise for throughput or latency. Similarly, partitioning strategies might vary over the prefill and decode stages to optimise for performance metrics utilised in auto-regressive inference such as time-between-tokens (TBT)

and time-to-first-token (TTFT). An interesting exploration in this space would be to study the usage of multiple partitioning strategies in model deployment to cater to the different inference requirements.

6.3 Research Impact

In this section we assess the impact of the research carried out in this report. The assessment was done with reference to the AREA 4P Orbit framework and details considerations taken into account to produce desirable and impactful outcomes of the research.

The research was focused on enabling high performance deep learning on low power IoT devices. There was a focus on achieving this using automated methods that did not require human-expert knowledge. While currently only large organisations with access to human experts have been able to deploy state-of-the-art DNN solutions, the research area that this work focuses on significantly simplifies the process of efficient DNN design by obviating the need for human experts. With such a democratisation of AI, smaller organisations would be enabled to develop and deploy smart solutions in new domain areas such as security, crisis response, infrastructure and others. Similarly, the high performance enabled by the work carried out in this thesis and other research works will also attract further applications in the IoT space. With the IoT market expected to grow to 1.6 billion U.S. dollars by 2025, this would yield tremendous economic impact.

However, it is also necessary that with such a proliferation of smart AI systems, we anticipate the potential negative impacts that might be perceived by the public and different stakeholders in different applications so that they can be addressed and social benefit can be maximised.

One of the potential issues with current AI solutions based on DNNs is of their interpretability and explainability. As tasks continue to be automated, it is possible that the public may not trust systems whose behavior cannot be causally explained. AI arguably has a bit of a problem when it comes to its perception by the public. This could possibly be attributed to its often bleak portrayal in popular media. It is important to understand the concerns of the public so that these can be assuaged. Other concerns might be raised in relation to the collection of data which is required for such smart systems. As an example, security surveillance applications which aim to achieve socially desirable outcomes such as better public safety, might be considered to be a breach of privacy. It would be important to engage with and educate the public on the applications of the technology and its implications on society which could alleviate such concerns. This could be done through outreach activities such as demonstrations

at science fairs or communication through creation of digital content circulated on online media platforms. Surveys and questionnaires could also be used on events and different platforms to better understand the public concern.

As new applications emerge, it would also be imperative for government to define rigorous standards for robustness in addition to creating legislation on how the systems could be used and how they would interact with existing infrastructure. For example, the National Surveillance Camera Strategy defines regulations for data collection, storage and curation in addition to defining protocols for surveillance footage usage in legal proceedings. With reference to these issues, the EPSRC has recently launched a research project on Trustworthy Autonomous Systems (TAS) to explore ways to reliably and safely deliver secure autonomous systems while considering ethical and legal constraints.

Another concern is of job displacement produced by increasing automation. An example would be the interest in IoT solutions for industrial predictive maintenance. In many settings industrial machines are regularly monitored by maintenance staff. Predictive maintenance solutions may significantly reduce the amount of human monitoring, reducing the role to only maintenance. However, while certain jobs may cease to exist in the future, new roles will be created. In this case, it is vital to carefully monitor how the employment landscape develops and appropriately offer retraining opportunities to the public. In this space, the government has already launched the National retraining scheme which was recently alloted £100m to help the public explore alternative occupations and training opportunities to develop new skills. A policy recommendation would be for the TAS hub to be added to the partner organisations of the National Retraining scheme. This would enable knowledge transfer between the two organisations with the TAS hubs knowledge of emerging AI applications guiding the curriculum for the retraining programmes.

As one of the key stakeholders in the project, the interests of the industrial partner, ARM, were considered from the start of the project. This was done by conducting an initial study of their work in the area of Tiny Machine Learning through digital resources available on their website and other online platforms. Following this, a meeting with the supervisory team was held before the start of the project to ensure alignment between the research project and the industrial sponsors. The research proposal was also initially presented at the annual ARM-ECS review meeting. Following this, regular meetings were carried out with the industrial supervisors who provided insight into the industrial landscape and relevant research questions. This information exchange guided the research in a direction that allowed for innovation that could be embedded into innovative and marketable products.

Appendix A

A.1 Linear Quantisation with Integer-Only Arithmetic

In Section 2.3.2.3, we described how the real-valued scaling factor, S and integer-valued zero-point, Z can be determined in the quantisation scheme. In this section, we review how the integer only arithmetic is performed with the given quantisation parameters. Through our discussion of the arithmetic, we derive the memory requirement of using the proposed quantisation scheme which is utilised in the research contributions in Chapter 4.

We consider the case of matrix multiplication of two quantised matrices as this represents the main workload of CNN architectures where dense fully connected layers and convolutional layers directly map to matrix multiplications through the partial im2col algorithm discussed in Section 2.4.3.

The multiplication of two quantised matrices, $\mathbf{W}$ and $\mathbf{X}$, represented with the affine mapping $r = S(q - Z)$ can be performed using the quantisation scheme as below

$$\begin{aligned} \mathbf{Y} &= \mathbf{W}\mathbf{X} + \mathbf{b} \\ S_Y(\mathbf{q_Y} - Z_Y) &= S_W(\mathbf{q_W} - Z_W) \cdot S_X(\mathbf{q_X} - Z_X) + S_b(\mathbf{q_b} - Z_b) \end{aligned} \tag{A.1}$$

As the quantisation is performed symmetrically for weight tensors, Z_w and Z_b is set to zero. Another assumption the scheme takes is to assume $S_b = S_W S_X$ which simplifies the above to

$$\begin{aligned} S_Y(\mathbf{q_Y} - Z_Y) &= S_W\mathbf{q_W} \cdot S_X(\mathbf{q_X} - Z_X) + S_W S_X\mathbf{q_b} \\ \mathbf{q_Y} &= \frac{S_W S_X}{S_Y}(\mathbf{q_W}\mathbf{q_X} - \mathbf{q_W}Z_X + \mathbf{q_b}) + Z_Y \end{aligned} \tag{A.2}$$

In the above $\mathbf{q_W q_X}$ is a matrix multiplication of 8-bit integers that can be efficiently performed on the MCU. The term $\mathbf{q_W} Z_X$ consists of constants known beforehand so this is precomputed. Similarly, the quantised representation of the bias, $\mathbf{q_b}$ is also known apriori so this can be combined with the previous term to precompute $\mathbf{q_{bias}} = \mathbf{q_W} Z_X + \mathbf{q_b}$ which effectively fuses the bias addition with the matrix multiplication. The $\mathbf{q_{bias}}$ is a 32-bit number stored in memory to deal with overflow when multiplying and adding the lower precision 8-bit operands.

In Equation A.2, $M = \frac{S_W S_X}{S_Y}$ is a non-integer term as the scaling factors are real-valued numbers. To limit the arithmetic to only integer mathematics, the term is approximated as $M = 2^{-n} M_o \approx \frac{S_W S_X}{S_Y}$, where M_o is an integer value. This allows the multiplication by the real-valued M to be approximated by an integer multiplication with M_o followed by a shift by n. This also generates the multiplier and shift parameters for operations in the inference graph. With the assumptions Equation A.2 simplifies to

$$\mathbf{q_Y} = 2^{-n} M_o (\mathbf{q_W q_X} - \mathbf{q_{bias}}) + Z_Y \tag{A.3}$$

The above assumes quantisation performed at per tensor granularity for the weight matrix. For convolution layers however, the quantisation is performed at a per channel granularity which is equivalent to performing matrix multiplication with a weight matrix having a number of scaling factors, $\mathbf{S_W}$. With per channel quantisation, there are as many scaling factors as there are bias elements. Subsequently, the derivation above produces an array of multiplier parameters, $\mathbf{M_o}$ and shift parameters, $\mathbf{n}$.

Memory Usage The quantisation scheme requires precomputing and storing $\mathbf{q_{bias}}$ in 32-bit format which has a memory requirement of $\mathbf{4C_{out}}$ bytes. Similarly, the multiplier and shift parameters are stored in 32-bit which have a memory footprint of $\mathbf{4C_{out}}$ bytes. In the case of a dense layer, the single weight matrix has only one scaling factor due to which M_o and n are scalar values each requiring 4 bytes.

A.2 Sample Linker Command File

Listing A.1 shows an example of a simplified linker command file. Two memory segments are defined for internal flash (FLASH) and ram (RAM), with two segments defining external ram (SDRAM) and flash (QSPI). After, the memory segments are defined, the linker command file directs the compiler to place the .text, .data and .bss section in internal flash and ram. As the .data section contains initialised data, it must be stored in non-volatile flash storage. For faster access latency, it is typically accessed from RAM.

The >RAM AT> FLASH, syntax is used to place the initialised data in non-volatile flash and then load it to volatile ram at initialisation.

The linker command file also shows two sections, .sdram_data and .sdram_bss, that were defined by the user and placed in external SDRAM. Variables in C/C++ code can be placed in these section using compiler directives.

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM);    /* end of "RAM" Ram type memory */

_Min_Heap_Size = 0x600 ;         /* required amount of heap  */
_Min_Stack_Size = 0xB00 ;        /* required amount of stack */

MEMORY
{
        FLASH    (rx)      : ORIGIN = 0x08000000,   LENGTH = 1024K
        RAM      (xrw)     : ORIGIN = 0x20000000,   LENGTH = 320K
        SDRAM    (xrw)   : ORIGIN = 0x60000000,   LENGTH = 8M
        QSPI     (xrw)     : ORIGIN = 0x90000000,   LENGTH = 16M
}

/* Sections */
SECTIONS
{

        /* The program code and other data into "FLASH" Rom type memory */
        .text :
        {
                . = ALIGN(4);
                *(.text)            /* .text sections (code) */
                *(.text*)           /* .text* sections (code) */
                *(.glue_7)          /* glue arm to thumb code */
                *(.glue_7t)         /* glue thumb to arm code */
                *(.eh_frame)

                KEEP (*(.init))
                KEEP (*(.fini))

                . = ALIGN(4);
                _etext = .;         /* define a global symbols at end of code */
        } >FLASH

        /* Used by the startup to initialize data */
        _sidata = LOADADDR(.data);

        /* Initialized data sections into "RAM" Ram type memory */
        .data :
        {
                . = ALIGN(4);
                _sdata = .;              /* create a global symbol at data start */
                *(.data)                 /* .data sections */
                *(.data*)                /* .data* sections */
                *(.RamFunc)              /* .RamFunc sections */
                *(.RamFunc*)     /* .RamFunc* sections */
```

```
        . = ALIGN(4);
        _edata = .;         /* define a global symbol at data end */

    } >RAM AT> FLASH

    /* Used by the startup to initialize data */
    _sisdram_data = LOADADDR(.sdram_data);

    /* Initialized data sections into "RAM" Ram type memory */
    .sdram_data :
    {
        . = ALIGN(4);
        _ssdram_data = .;        /* global symbol at data start */
        *(.sdram_data)       /* .data sections */
        *(.sdram_data*)      /* .data* sections */
        *(.sdram_RamFunc)    /* .RamFunc sections */
        *(.sdram_RamFunc*)   /* .RamFunc* sections */

        . = ALIGN(4);
        _esdram_data = .;    /* define a global symbol at data end */

    } >SDRAM AT> FLASH

    /* Uninitialized data section into "RAM" Ram type memory */
    . = ALIGN(4);
    .bss :
    {
        /* Used by startup in order to initialize the .bss section */
        _sbss = .;          /* define a global symbol at bss start */
        __bss_start__ = _sbss;
        *(.bss)
        *(.bss*)
        *(COMMON)

        . = ALIGN(4);
        _ebss = .;          /* define a global symbol at bss end */
        __bss_end__ = _ebss;
    } >RAM

    /* Uninitialized sdram data section into "SDRAM" Ram type memory */
    . = ALIGN(4);
    .sdram_bss :
    {
        _ssdram_bss = .;
        __bss_start__ = _ssdram_bss;
        *(.sdram_bss)
        *(.sdram_bss*)

        . = ALIGN(4);
        _esdram_bss = .;
        __bss_end__ = _esdram_bss;
    } >SDRAM

}
```

LISTING A.1: Sample Linker Command File

Appendix B

B.1 Derivation of Approximate Architecture Gradients

In this section we demonstrate that the approximation scheme utilised by DARTS (Liu et al., 2019) was compatible with our multi-objective architecture gradients. To derive the architecture gradients using the approximation scheme, the multivariate chain rule could be applied as shown below,

$$\begin{aligned}\nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) &= \nabla_\alpha w'(w,\alpha) \times \nabla_{w'}\mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \\ &\quad + \nabla_\alpha \alpha \times \nabla_\alpha \mathcal{L}_{per}, val(w'(w,\alpha),\alpha) \qquad \text{(B.1)}\\ &= \nabla_\alpha (w - \xi\nabla\mathcal{L}_{per,train}(w,\alpha) \times \nabla_{w'}\mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \\ &\quad + \nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \qquad \text{(B.2)}\\ &= \left[\nabla_\alpha w - \xi\nabla^2_{\alpha,w}\mathcal{L}_{per,train}(w,\alpha)\right] \times \nabla_{w'}\mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \\ &\quad + \nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \qquad \text{(B.3)}\\ &= -\xi\nabla^2_{\alpha,w}\mathcal{L}_{per,train}(w,\alpha)\nabla_{w'}\mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \\ &\quad + \nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \qquad \text{(B.4)}\end{aligned}$$

In the first term in Eq. B.4, the first derivative term, $\nabla_{w'}\mathcal{L}_{per,val}(w'(w,\alpha),\alpha)$ is a large vector of derivatives with a number of dimensions equal to the free weight parameters, w in the network. Similarly, the term involving the second derivative, $\nabla^2_{\alpha,w}\mathcal{L}_{per,train}(w,\alpha)$ is a large matrix of second derivatives with a dimensionality of the number of free weight parameters by the number of architecture weights. Computing the vector-matrix product of the two would be prohibitively expensive. To bypass this complex operation, the central finite difference approximation was employed by Liu et al. (2019) with respect to w to substantially reduce the complexity as shown below where $w^{\pm} = w \pm \epsilon\nabla_{w'}\mathcal{L}_{per,val}(w',\alpha)$.

$$\nabla^2_{\alpha,w}\mathcal{L}_{per,train}(w,\alpha) = \frac{\nabla_\alpha \mathcal{L}_{per,train}(w^+,\alpha) - \nabla_\alpha \mathcal{L}_{per,train}(w^-,\alpha)}{2\epsilon\nabla_{w'}\mathcal{L}_{per,val}(w',\alpha)} \qquad \text{(B.5)}$$

After substituting the simplified expression in Eq. B.5 into Eq. B.4 we can obtain the final required architecture gradient as shown below in Eq. B.8.

$$\begin{aligned}\nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) &= \nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \\ &\quad - \xi \nabla^2_{\alpha,w} \mathcal{L}_{per,train}(w,\alpha) \nabla_{w'} \mathcal{L}_{per,val}(w'(w,\alpha),\alpha)\end{aligned} \tag{B.6}$$

$$\begin{aligned}&= \nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \\ &\quad - \xi \frac{\nabla_\alpha \mathcal{L}_{per,train}(w^+,\alpha) - \nabla_\alpha \mathcal{L}_{per,train}(w^-,\alpha)}{2\epsilon \nabla_{w'} \mathcal{L}_{per,val}(w',\alpha)} \\ &\quad \times \nabla_{w'} \mathcal{L}_{per,val}(w'(w,\alpha),\alpha)\end{aligned} \tag{B.7}$$

$$\begin{aligned}\nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) &= \nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha) \\ &\quad - \xi \frac{\nabla_\alpha \mathcal{L}_{per,train}(w^+,\alpha) - \nabla_\alpha \mathcal{L}_{per,train}(w^-,\alpha)}{2\epsilon}\end{aligned} \tag{B.8}$$

Computing the second term in Eq. B.8 would require only two forward and two backward passes to obtain the gradients with respect to the the network weights, $w^\pm$ and the architecture weights, α. Further, the complexity was reduced from $\mathcal{O}(|\alpha||w|)$ to $\mathcal{O}(|\alpha| + |w|)$.

Note that in Eq. B.8, setting ξ to zero would result in $\nabla_\alpha \mathcal{L}_{per,val}(w'(w,\alpha),\alpha)$ being equal to $\nabla_\alpha \mathcal{L}_{per,val}(w,\alpha)$. This case, referred to as the first-order approximation would correspond to the simple heuristic of assuming the current set of weights, w as the optimal weights, w^* and would speed-up the search process. In practice, the Liu et al. (2018) noted that the first-order approximation lead to empirically worse performance so in our work we used the case of $\xi > 0$, referred to as the second-order approximation.

B.2 DEff-ARTS Architecture Search Hyper-Parameters

- Number of Cells: 8
- Cell Step Size: 4
- Epochs: 50
- Batch Size: 64
- Initial Stem Block Output Channels: 16
- Networks Weights Optimiser: SGD with momentum
  - Learning Rate: 0.025 annealed through cosine schedule
  - Momentum: 0.9
  - Weight Decay: 3×10^{-4}

  - Gradient Norm Clipping: 5.0

- Architecture Weights Optimiser: Adam
  - Learning Rate: 3×10^{-4}
  - Momentum β: $(0.5, 0.999)$
  - Weight Decay: 10^{-3}
- Train/Validation Split: 50 : 50
- Modulation Parameter, $\beta = 0.27$
- Cost Weightage Parameter, $\Gamma = 0.01, 0.02, 0.04$
- $Cost_{zero} = 10^6 - 1$, to apply selection pressure towards non-zero candidate operations considered in discretisation

B.3 DEff-ARTS Architecture Training Hyper-Parameters

Results reported were an average of 3 runs.

- Number of Cells: 20
- Epochs: 600
- Batch Size: 96
- Initial stem Block Output Channels: 36
- Optimiser: SGD with momentum
  - Learning Rate: 0.025 annealed through cosine schedule
  - Momentum: 0.9
  - Weight Decay: 3×10^{-4}
  - Gradient Norm Clipping: 5.0
- Auxiliary Tower Weight: 0.4
- Cutout Length: 16
- Path Dropout Probability: 0.2

Appendix C

C.1 Cell Based Search Space vs Manual Width Scaling

We compared the performance of the DEff-ARTS models derived from the cell based search space with the popular approach of scaling the width of models. To reduce the compute time required for evaluation, the networks were constructed by stacking 5 cells. A comparison of the two approaches is shown in Table C.1. It can be observed that the manual approach of scaling outperformed the cells derived via the automated approach of DEff-ARTS.

TABLE C.1: Comparison of DEff-ARTS Architectures with Manual Width Scaling

Architecture	Test Error (%)	Params (K)	Search Cost (GPU days)	#ops	Search Method	MACs (M)
DARTS, $w = 0.8x$	92.38	88.4	-	-	Manual	19.9
DEff-ARTS + cutout, $\Gamma = 0.01$	90.83	90.1	4	7	Gradient-Based	21.3
DARTS, $w = 0.7x$	91.5	65.3	-	-	Manual	15.0
DEff-ARTS + cutout, $\Gamma = 0.02$	84.56	63.6	4	7	Gradient-Based	16.7

C.2 Data Structure for Partitioning Strategy

The data structure used to store the partitioning strategy for the operations listed in Section 4.2.3 is shown in Section C.1. The PARTITION_FACTOR macro was defined to be 224 for the worst case in CNNs designed for ImageNet classification in the MCU setting where the maximum resolution or H dimension is 224. We note that this could be arbitrarily changed to any other constant.

```
typedef struct PartitionInfo {
        TfLiteIntArray *        dims;

        // pointer to input tensors
        const TfLiteEvalTensor * input1;
        const TfLiteEvalTensor * input2;

        // pointer to filter and bias tensors
```

```
        const TfLiteEvalTensor * filter;
        // pointer to bias tensor
        const TfLiteEvalTensor * bias;

        // partition factor for the particular operation
        uint8_t numParts;

        // number of rows in the tiny tensor
        uint8_t partSizes[PARTITION_FACTOR];
        // starting row of the partitions
        uint8_t partitionStart[PARTITION_FACTOR];
        // ending rows of the partitions
        uint8_t partitionEnd[PARTITION_FACTOR];

        // dimension of partitioned input tiny tensors
        uint16_t inp_partitioned_dim[PARTITION_FACTOR];
        uint16_t out_partitioned_dim[PARTITION_FACTOR];

        // tiny tensor operation info, e.g. padding
        uint8_t op_part_data[PARTITION_FACTOR][PARTITION_OP_DATA_SIZE];


    } PartitionInfo;
```

LISTING C.1: Operation Partitioning Strategy Data Structure

C.3 Architecture Training Hyper-Parameters

- Epochs: 150
- Batch Size: 256
- Networks Weights Optimiser: SGD with momentum
  - Learning Rate: 0.025 annealed through cosine schedule
  - Momentum: 0.9
  - Weight Decay: 4×10^{-5}

Appendix D

D.1 SuperNetwork Training Hyper-Parameters

- Epochs: 20
- Batch Size: 128
- Networks Weights Optimiser: SGD with momentum
  - Learning Rate: 0.1 annealed through cosine schedule
  - Momentum: 0.9
  - Weight Decay: 1×10^{-4}
- Candidate Widths: 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, 0.50, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85, 0.90, 0.95, 1.00
- Candidate Resolutions: 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, and Michael Isard. Tensorflow: A system for large-scale machine learning. pages 265–283, 2016. ISBN 1931971331.

Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.

Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Nat Jeffries, Csaba Kiraly, Jeremy Holleman, Pietro Montino, David Kanter, Pete Warden, and Danilo Pau. Mlperf tiny benchmark. 2021.

Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *Machine learning for data science handbook: data mining and knowledge discovery handbook*, pages 353–374, 2023. survey papers on autoencoders.

Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Advances in Neural Information Processing Systems*, 32, 2019.

Linear Solvers Basics. Intel® math kernel library. 2005.

Babak Ehteshami Bejnordi, Tijmen Blankevoort, and Max Welling. Batch-shaping for learning conditional channel gated networks. 2019.

Tobias Blickle. Tournament selection. *Evolutionary computation*, 1:181–186, 2000.

Halima Bouzidi, Mohanad Odema, Hamza Ouarnoughi, Mohammad Abdullah Al Faruque, and Smail Niar. Hadas: Hardware-aware dynamic neural architecture search for edge performance scaling. pages 1–6. IEEE, 2023. ISBN 9798350396249.

Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 70:1253–1268, 2021.

Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *International Conference on Learning Representations*, 2018.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *8th International Conference on Learning Representations (ICLR)*, pages 1–15, 2020.

Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67:871–875, 2020.

Zhourong Chen, Yang Li, Samy Bengio, and Si Si. You look twice: Gaternet for dynamic filter selection in cnns. pages 9172–9180, 2019.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. original paper proposing cudnn.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *International Conference on Learning Representations*, 2015.

Xin Dai, Xiangnan Kong, and Tian Guo. Epnet: Learning to exit with flexible multi-branch network. pages 235–244, 2020.

William J Dally, Stephen W Keckler, and David B Kirk. Evolution of the graphics processing unit (gpu). *IEEE Micro*, 41:42–51, 2021. ISSN 0272-1732. good survey. info on compute clusters and how they have scaled.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3, 2021.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. pages 248–255. Ieee, 2009. ISBN 1424439922.

Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout, 2017.

Matthew Stephen Doerner. An analysis of machine learning hardware from tenstorrent. 2024.

Xin Dong, Shangyu Chen, and Sinno Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *Advances in neural information processing systems*, 30, 2017.

Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. Brp-nas: Prediction-based nas using gcns. *Advances in Neural Information Processing Systems*, 33:10480–10490, 2020.

Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Rémi Gribonval, Herve Jegou, and Armand Joulin. Training with quantization noise for extreme model compression. *International Conference on Learning Representations*, 2021.

Massimiliano Fatica. Cuda toolkit and libraries. pages 1–22. IEEE, 2008. ISBN 1467388718.

Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems*, 32, 2019.

Igor Fedorov, Ramon Matas, Hokchhay Tann, Chuteng Zhou, Matthew Mattina, and Paul Whatmough. Udc: Unified dnas for compressible tinyml models for neural processing units. *Advances in Neural Information Processing Systems*, 35:18456–18471, 2022.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. 2018.

Matteo Gambella and Manuel Roveri. Edanas: Adaptive neural architecture search for early exit neural networks. pages 1–8. IEEE, 2023. ISBN 1665488670.

Xitong Gao, Yiren Zhao, Łukasz Dudziak, Robert Mullins, and Cheng zhong Xu. Dynamic channel pruning: Feature boosting and suppression. 2018.

Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. *A survey of quantization methods for efficient neural network inference*, pages 291–326. Chapman and Hall/CRC, 2022.

Michael Gibbs, Kieran Woodward, and Eiman Kanjo. Combining multiple tinyml models for multimodal context-aware stress recognition on constrained microcontrollers. *IEEE Micro*, 2023. ISSN 0272-1732.

Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E Gonzalez, and Ion Stoica. D3: a dynamic deadline-driven approach for building autonomous vehicles. *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 453–471, 2022.

Aidan N Gomez, Ivan Zhang, Siddhartha Rao Kamalakara, Divyam Madaan, Kevin Swersky, Yarin Gal, and Geoffrey E Hinton. Learning sparse networks using targeted dropout. *CoRR*, abs/1905.13678, 2019.

Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *International Conference for Learning Representations*, 2015.

Google. Tensorflow lite, 2024a. URL https://www.tensorflow.org/lite.

Google. Xnnpack, 2024b. URL https://github.com/google/XNNPACK.

Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28:2222–2232, 2016. ISSN 2162-237X. lstms survey.

Gaël Guennebaud and Benoit Jacob. Eigen. *URl: http://eigen. tuxfamily. org*, 3, 2010.

Song Han. Tinyml and efficient deep learning computing, lec 6 quantisation part 1, 2023.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations*, 2016.

Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, 5, 1992.

Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. pages 293–299. IEEE, 1993. ISBN 0780309995.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 2016.

Charles Herrmann, Richard Strong Bowen, and Ramin Zabih. Channel selection using gumbel softmax. pages 241–257. Springer, 2020.

Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

Geoffrey E Hinton. Learning multiple layers of representation. *Trends in cognitive sciences*, 11:428–434, 2007.

Shenda Hong, Yanbo Xu, Alind Khare, Satria Priambada, Kevin Maher, Alaa Aljiffry, Jimeng Sun, and Alexey Tumanov. Holmes: health online model ensemble serving for deep learning models in intensive care units. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1614–1624, 2020.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. pages 1314–1324, 2019.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *CoRR*, abs/1607.03250, 2016.

Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Weinberger. Multi-scale dense networks for resource efficient image classification. 2018.

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *International Conference on Learning Representations*, 2017.

Transforma Insights. Iot connected devices worldwide 2019-2030. *Statista*, 2022. URL https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

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

Won Jeon, Gun Ko, Jiwon Lee, Hyunwuk Lee, Dongho Ha, and Won Woo Ro. *Deep learning with GPUs*, volume 122, pages 167–215. Elsevier, 2021. ISBN 0065-2458. this has great table comparing gpu characteristics.

Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

Marc Jorda, Pedro Valero-Lara, and Antonio J Pena. Performance evaluation of cudnn convolution algorithms on nvidia volta gpus. *IEEE Access*, 7:70461–70473, 2019. ISSN 2169-3536. shows that input size and channels are useful when deciding which cudnn kernel to use.

Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, and Hector Yuen. A study of bfloat16 for deep learning training. *CoRR*, abs/1905.12322, 2019.

Ruhul Amin Khalil, Nasir Saeed, Mudassir Masood, Yasaman Moradi Fard, Mohamed-Slim Alouini, and Tareq Y Al-Naffouri. Deep learning in the industrial internet of things: Potentials, challenges, and emerging applications. *IEEE Internet of Things Journal*, 8:11016–11040, 2021.

Chaitanya Koparkar. Efficient data representation using flatbuffers. *XRDS: Crossroads, The ACM Magazine for Students*, 29:50–51, 2023. ISSN 1528-4972.

Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *CoRR*, abs/1801.06601, 2018.

Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. pages 598–605, 1990.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *International Conference on Learning Representations*, 2017.

Xiangjie Li, Chenfei Lou, Yuchi Chen, Zhengping Zhu, Yingtao Shen, Yehan Ma, and An Zou. Predictive exit: Prediction of fine-grained early exits for computation-and energy-efficient inference. volume 37, pages 8657–8665, 2023. ISBN 2374-3468.

Yanyu Li, Geng Yuan, Yang Wen, Ju Hu, Georgios Evangelidis, Sergey Tulyakov, Yanzhi Wang, and Jian Ren. Efficientformer: Vision transformers at mobilenet speed. *Advances in Neural Information Processing Systems*, 35:12934–12949, 2022.

Edgar Liberis and Nicholas D Lane. Neural networks on microcontrollers: saving memory at inference via operator reordering. *Machine Learning for Systems, On-Device Intelligence Workshop*, 2020.

Edgar Liberis and Nicholas D Lane. Pex: Memory-efficient microcontroller deep learning through partial execution. *CoRR*, abs/2211.17246, 2022.

Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. unas: Constrained neural architecture search for microcontrollers. pages 70–79, 2021.

Sean Lie. Cerebras architecture deep dive: First look inside the hw/sw co-design for deep learning: Cerebras systems. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–34. IEEE Computer Society, 2022.

Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.

Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Memory-efficient patch-based inference for tiny deep learning. *Advances in Neural Information Processing Systems*, 34:2346–2358, 2021.

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI Open*, 2022. ISSN 2666-6510. Transformers survey.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. 2018.

Risheng Liu, Jiaxin Gao, Jin Zhang, Deyu Meng, and Zhouchen Lin. Investigating bi-level optimization for learning and vision from a unified perspective: A survey and beyond. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(12):10045–10067, 2021.

Zhenhua Liu, Yunhe Wang, Kai Han, Siwei Ma, and Wen Gao. Instance-aware dynamic neural network quantization. pages 12434–12443, 2022.

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *7th International Conference on Learning Representations, ICLR 2019*, pages 1–21, 2019.

Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. 2018.

Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. Ternary neural networks with fine-grained quantization. *CoRR*, abs/1705.01462, 2017.

Joseph Mellor, Jack Turner, Amos Storkey, and Elliot J. Crowley. Neural architecture search without training. *Proceedings of the 38th International Conference on Machine Learning (ICML)*, 2021.

Meta. Pytorch mobile, 2024a. URL https://pytorch.org/mobile/home/.

Meta. Qnnpack, 2024b. URL https://github.com/pytorch/QNNPACK.

S T Microelectronics. Stm32x-cube-ai. 2019. URL https://www.st.com/en/embedded-software/x-cube-ai.html.

MicroTVM. Microtvm design, 2024. URL https://tvm.apache.org/docs/arch/microtvm_design.html.

Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025, 2016.

Ravi Teja Mullapudi, William R Mark, Noam Shazeer, and Kayvon Fatahalian. Hydranets: Specialized dynamic architectures for efficient inference. pages 8080–8089, 2018.

Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. 2010.

Mozhgan Navardi and Tinoosh Mohsenin. Mlae2: Metareasoning for latency-aware energy-efficient autonomous nano-drones. *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2023.

V Ntziachristos. Going deeper than microscopy: the optical imaging frontier in biology. *Nature methods*, 7:603–614, 2010.

Jose Nunez-Yanez and Mohammad Hosseinabady. Sparse and dense matrix multiplication hardware for heterogeneous multi-precision neural networks. *Array*, 12:100101, 2021. ISSN 2590-0056.

Nvidia. cublas, 2024. URL https://docs.nvidia.com/cuda/cublas/.

Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85. IEEE, 2014.

Atul Pandey. Depth-wise convolution and depth-wise separable convolution, 2024.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, and Luca Antiga. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Ieva Petrulionytė, Julien Mairal, and Michael Arbel. Functional bilevel optimization for machine learning. *Advances in Neural Information Processing Systems*, 37:14016–14065, 2024.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *35th International Conference on Machine Learning, ICML 2018*, 9:6522–6531, 2018.

Gopika Premsankar, Mario Di Francesco, and Tarik Taleb. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal*, 5:1275–1284, 2018.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. pages 2902–2911. PMLR, 2017. ISBN 2640-3498.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019 and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019*, pages 4780–4789, 2019. ISSN 2159-5399. doi: 10.1609/aaai.v33i01.33014780.

Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, and Ray Bittner. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in neural information processing systems*, 33:10271–10281, 2020.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, and Kristof Denolf. Microscaling data formats for deep learning. *CoRR*, abs/2310.10537, 2023.

Sourjya Roy, Priyadarshini Panda, Gopalakrishnan Srinivasan, and Anand Raghunathan. Pruning filters while training for efficiently optimizing deep learning networks. *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2020.

Manuele Rusci, Alessandro Capotondi, Francesco Conti, and Luca Benini. Quantized nns as the definitive solution for inference on low-power arm mcus? work-in-progress. pages 1–2, 2018.

Manuele Rusci, Alessandro Capotondi, and Luca Benini. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. *Proceedings of Machine Learning and Systems*, 2:326–335, 2020a.

Manuele Rusci, Marco Fariselli, Alessandro Capotondi, and Luca Benini. Leveraging automated mixed-low-precision quantization for tiny edge microcontrollers. *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning: Second International Workshop, IoT Streams 2020*, 8 2020b. URL http://arxiv.org/abs/2008.05124.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, and Michael Bernstein. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115: 211–252, 2015. ISSN 0920-5691.

Sulaiman Sadiq, Partha Maji, Jonathan Hare, and Geoff Merrett. Deff-arts: Differentiable efficient architecture search. *Advances in Neural Information Processing Systems, ML for Systems Workshop*, 2020.

Sulaiman Sadiq, Jonathon Hare, Partha Maji, Simon Craske, and Geoff V Merrett. Tinyops: Imagenet scale deep learning on microcontrollers. pages 2702–2706, 2022.

Sulaiman Sadiq, Jonathon Hare, Simon Craske, Partha Maji, and Geoff Merrett. Enabling imagenet-scale deep learning on mcus for accurate and efficient inference. *IEEE Internet of Things Journal*, 2023. ISSN 2327-4662.

Sulaiman Sadiq, Jonathon Hare, Geoffrey Merrett, Partha Prasun Maji, and Simon John Craske. Neural network memory configuration, 1 2024.

Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. *CoRR*, abs/1801.01078, 2017. RNN survey.

Farzad Samie, Lars Bauer, and Jörg Henkel. From cloud down to things: An overview of machine learning in internet of things. *IEEE Internet of Things Journal*, 6:4921–4934, 2019.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. pages 4510–4520, 2018.

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. volume 34, pages 8815–8821, 2020. ISBN 2374-3468.

Richard Shin, Charles Packer, and Dawn Song. Differentiable neural network architecture search. *6th International Conference on Learning Representations, ICLR 2018 - Workshop Track Proceedings*, pages 1–4, 2018.

Yao Shu, Wei Wang, and Shaofeng Cai. Understanding architectures learnt by cell-based neural architecture search. *8th International Conference on Learning Representations, (ICLR)*, 2020.

Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, and Enrico Calore. A survey on deep learning hardware accelerators for heterogeneous hpc platforms. *CoRR*, abs/2306.15552, 2023. this survey includes other hardware such as tpus, npus, fpgas.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484, 2016.

Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G Yen. Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 24:394–407, 2019. ISSN 1089-778X.

Filip Svoboda, Javier Fernandez-Marques, Edgar Liberis, and Nicholas D Lane. Deep learning on microcontrollers: A study on deployment costs and challenges. pages 54–63, 2022.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. pages 1–9, 2015.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. pages 6105–6114, 2019.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2815–2823, 2019.

Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. pages 2464–2469. IEEE, 2016. ISBN 1509048472.

Texas-Instruments. Tms320c64x/c64x+ dsp cpu and instruction set. *Technical Documentation*, 2019.

uTensor. utensor, 2024. URL https://utensor.github.io/website/.

Han Vanholder. Efficient inference with tensorrt. volume 1, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap. *CoRR*, abs/2207.02852, 2022.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, and Petko Georgiev. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575:350–354, 2019. ISSN 1476-4687.

Dilin Wang, Chengyue Gong, Meng Li, Qiang Liu, and Vikas Chandra. Alphanet: Improved training of supernets with alpha-divergence. pages 10760–10771. PMLR, 2021a. ISBN 2640-3498.

Dilin Wang, Meng Li, Chengyue Gong, and Vikas Chandra. Attentivenas: Improving neural architecture search via attentive sampling. pages 6418–6427, 2021b.

Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8612–8620, 2019.

Xiaying Wang, Michele Magno, Lukas Cavigelli, Mufti Mahmud, Claudia Cecchetto, Stefano Vassanelli, and Luca Benini. Embedded classification of local field potentials recorded from rat barrel cortex with implanted multi-electrode array. pages 1–4, 2018.

Xiaying Wang, Michele Magno, Lukas Cavigelli, and Luca Benini. Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things. *IEEE Internet of Things Journal*, 7:4403–4417, 2020.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019-June:10726–10734, 2019. ISSN 10636919. doi: 10.1109/CVPR.2019.01099.

Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *CoRR*, abs/2004.09602, 2020.

Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: Stochastic neural architecture search. *7th International Conference on Learning Representations, ICLR 2019*, pages 1–17, 2019.

Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo jun Qi, Qi Tian, and Hongkai Xiong. Pc-darts: Partial channel connections for memory-efficient architecture search. *Iclr*, 2:1–12, 2020.

Zirui Xu, Fuxun Yu, Chenchen Liu, and Xiang Chen. Reform: Static and dynamic resource-aware dnn reconfiguration framework for mobile device. pages 1–6, 2019. to cite that mobile devices run multiple workloads.

Taojiannan Yang, Sijie Zhu, Chen Chen, Shen Yan, Mi Zhang, and Andrew Willis. Mutualnet: Adaptive convnet via mutual learning from network width and resolution. pages 299–315. Springer, 2020. ISBN 3030584518.

Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. pages 7105–7114. PMLR, 2019. ISBN 2640-3498.

Jiahui Yu and Thomas S Huang. Universally slimmable networks and improved training techniques. pages 1803–1811, 2019.

Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. 2018.

Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1:22–32, 2014.

Gaetano Zazzaro, Salvatore Cuomo, Angelo Martone, R Valentino Montaquila, Gerardo Toraldo, and Luigi Pavone. Eeg signal analysis for epileptic seizures detection by applying data mining techniques. *IEEE Internet of Things Journal*, 14:100048, 2021.

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. pages 6848–6856, 2018.

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.

Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *International Conference on Learning Representations*, 2017.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, pages 1–16, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. pages 8697–8710, 2018.