UNIVERSITY OF CAMBRIDGE

Efficient deep learning in resource-constrained environments

PhD Proposal

Edgar Liberis

St Catharine's College

First year report submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Contents

Chapter 1

PhD Proposal

1.1 Introduction

Neural networks are a class of machine learning models that have emerged as versatile competitors to complex hand-crafted systems for tasks with an abundance of labelled data. In particular, convolutional and recurrent neural networks (CNNs and RNNs, respectively) have been able to make great strides in computer vision, audio and natural language processing, powering surveillance, translation, knowledge database and medical diagnosis systems, among many others.

The most recent rise in popularity of neural networks was facilitated by the emergence of more powerful computer hardware, namely programmable (general-purpose) graphics processing units (GPUs). The highly parallel design of GPUs makes them a good fit for linear algebra operations, which are at the heart of neural network computation. In 2012, this allowed the breakthrough CNN AlexNet [51] to drastically improve on image classification error rate in the ImageNet competition. The network was powered by two nVidia GTX 580 GPUs and the CUDA programming platform.

Since then, improving prediction accuracy has been the primary objective of neural network development, while continuing to rely on modern graphics cards or newly-developed high-performance accelerators, such as TPUs, for *training* (creation of the model) and *inference* (prediction using the model). The dependency on powerful hardware makes it challenging to use neural networks when such computational power is not available or would be impractical, such as on mobile phones, space satellites or everyday devices powered by microcontroller units (MCUs).

1.2 Tackling resource usage of neural networks

In a real-world deployment, in addition to having an acceptable prediction accuracy, neural networks have to fit within a certain inference latency, power consumption and memory usage (both storage and runtime) constraints. For example, a facial recognition program running on an MCU (*e.g.* a doorbell camera), would be constrained by the amount of on-chip memory and power supply; many event detection systems (*e.g.* pedestrian recognition in self-driving vehicles) would not be constrained by memory or power as much, but would have to respond as quickly as possible.

In neural network research, making large models fit within tight computational resource constraints is achieved by applying model compression techniques [17]. They usually create a smaller model from the larger model by, for example, reducing the number of parameters in the network via pruning. While compression has beneficial effects on the model's storage requirements, memory use and inference speed, it often comes at the cost of reduced accuracy. Hence, model compression methods can be seen as exploring the trade-off between accuracy and runtime properties of a neural network.

In my thesis, I am to investigate ways to reduce resource consumption of neural networks during inference. Specifically, I restrict the focus of the work to two relatively underexplored constraints: *inference latency* and *runtime memory usage*. Resource constraints are interlinked, so, for example, gains in memory usage or latency are likely to also improve storage requirement and power use.

- **Runtime memory usage.** This has received relatively little attention due to most target platforms having a sufficient supply of RAM. However, optimising for (peak) memory usage, can make previously-undeployable models run on hardware with a limited amount of memory, such as microcontrollers. Even on platforms that have abundant RAM, this would allow the model to fit within faster on-chip memories or custom accelerator chips/co-processors, producing further gains in latency and power usage.
- **Latency.** Enabling deep learning models to run faster would create more intelligent rapid-response systems (*e.g.* safety or trading systems). Additionally, finishing execution sooner would allow hardware to go to low power states quicker, thus also conserving energy.

I intend to push the accuracy vs resources trade-off boundary (Pareto front) in model compression by employing a more holistic approach that optimises neural network

architecture and its execution (in software) together, jointly, to make the best use of underlying hardware. At the cost of being less hardware-agnostic, this vertical integration approach would allow optimising with respect to particular execution properties, memory layout and other hardware capabilities of the target platform, producing a tightly coupled system that retains good prediction accuracy while fitting within runtime constraints.

I propose to achieve this by designing automated model design and/or compression methodology, together with a tailored neural network execution runtime (when required), that is capable of producing fast and memory-efficient models which leverage the strengths of the underlying hardware. Using automated algorithms would allow recreating these holistically-optimised systems for many devices (within a particular class of devices that share similar properties), keeping the methodology relevant as hardware continues to develop.

To further explain and justify this holistic optimisation direction, Section 1.3 briefly reviews and identifies opportunities current model compression and efficient inference methodology (full detailed literature survey is available in Chapter 2). Section 1.4 enumerates projects carried out thus far; Section 1.5 proposes concrete future research directions; Section 1.6 proposes a research timeline.

1.3 Resource-efficient deep learning

1.3.1 Model compression: making large networks compact

There are several mostly-independent and orthogonal research directions within model compression that can help to produce compact models.

- **Pruning.** Pruning discards individual parameters or groups of them (fine-grained and structured pruning, respectively) from a neural network. Fine-grained pruning produces sparse parameter matrices, which require costly sparse computation primitives at runtime. Thus many works opt for structured pruning, which drops individual channels or units in convolutional and fully-connected layers. In effect, structured pruning adjusts hyperparameters of an architecture in a more informed way after or during model training. Pruning improves runtime memory usage by shrinking intermediate tensors (activation maps) and latency by reducing the amount of computation required.
- **Weight decomposition.** Weight decomposition approaches break up large param-

eter (weight) matrices into a product of 2 or more smaller matrices. In addition to reducing the size of a model's parameter matrices, computation on smaller matrices uses less memory at runtime.

- **Distillation.** Distillation approaches train a smaller (student) model by requiring it to emulate a larger (teacher) model.
- **Dynamic routing.** A neural network does not have to perform the same computation for all inputs—instead it may choose to route to or skip certain layers based on the perceived difficulty of the input. Early-exit networks [87], in particular, allow reducing inference latency on simpler inputs.
- **Quantisation and binarisation.** Neural networks were shown to perform well when the precision of parameters and activation maps is reduced to 8 or fewer bits per element. In addition to drastically reducing per-parameter storage requirements, certain bit-widths allows using SIMD to speed up the computation on supported hardware. Quantisation can be pushed to the extreme by considering binary weights and/or activations (binarisation), which would allow using efficient bit-wise arithmetic [30].

The aforementioned approaches produce a compact model which is also already trained for the task at hand. However, they do not usually allow precisely targeting a particular runtime constraint, such as imposing an upper bound on running time in seconds or the number of bytes of memory used at inference. In fact, many works only quantify the reduction in model size and the number of floating-point operations (FLOPs) needed for inference, without assuming any properties of the underlying hardware.

1.3.2 Neural architecture search: compact models from scratch

Designing neural networks under hard constraints is an optimisation problem that is well-suited for an automated search algorithm: this is widely explored in the Neural Architecture Search (NAS) research direction. Traditionally, the search aims to produce an architecture that achieves greater accuracy for the task at hand than human-made designs. However, the search goal can be augmented with accurate runtime constraints to automatically discover neural networks with low resource usage and high accuracy.

Common NAS methods create new neural network architectures by assembling them from a vocabulary of primitives (a matrix multiplication, a convolution, a depthwise convolution, *etc.*). Compared to model compression, constrained NAS provides more flexibility during design time, as it creates candidate architectures from scratch rather

than by working from a single large existing architecture. In addition to classification accuracy, constrained NAS systems typically target latency and power constraints for "mobile" hardware [86, 10], others (but very few) consider extremely constrained microcontroller platforms with 0.5–256 KB storage and memory limits [15, 29, 61].

1.3.3 Optimising neural network inference software and hardware

In numerical computation and systems research, satisfying runtime constraints can be approached by optimising inference software or hardware:

- **Software optimisation.** Latency and runtime memory usage can be improved by using algorithmically and computationally efficient implementations of primitive operations, such as fast matrix multiplication and convolution; optimally laying out data in memory and leveraging caching; fusing adjacent operators, changing their order or executing them partially.
- **Hardware optimisation.** A popular way to accelerate a neural network workload is to design custom circuitry, usually based on systolic array architecture. This can be a custom chip/ASIC, an accelerator core in a multicore system or a design for an FPGA board. The designs tend to optimise both execution speed/throughput and power usage.

Works above typically excel at optimising linear algebra workloads for the target software and/or hardware platform but tend to assume very little about the overall neural network architecture that is being executed. They also try to preserve the correctness and precision of the computation, regardless of whether the network could be made to recover from this.

1.3.4 Holistic optimisation opportunities

Previously discussed model compression and NAS methods often assume very little about the underlying inference software and hardware. Similarly, inference software and hardware are often designed to be agnostic about the models they are meant to be executing. While this allows compression methods and hardware/software designs to stay general and applicable in more cases, it also misses out on performance gains that can be achieved if both are optimised jointly.

Of course, one has to be cautious of not over-optimising for a particular platform or chip, resulting in a vertical integration that cannot be generalised to other environments.

However, I believe there's a balance to be struck by developing methodology that can automatically produce specialised solutions for devices within a particular class of hardware platforms, such as microcontrollers or small linear algebra accelerators.

There have been some works that attempt to bridge the gap between neural network architecture and the way it is executed to achieve better efficiency, suggesting it is a developing research direction. For example, neural architecture search for microcontroller platforms [29, 61] or mobile devices [85, 94]; partial evaluation of a residual connection branch in MobileNets [80], or inducing sparsity at training time to leverage an accelerator for sparse linear algebra [14]. However, due to residing between the domains of deep learning and systems, this area has been relatively underexplored.

1.4 Projects carried out so far

So far, I have carried out several projects relating to model compression, neural architecture search and runtime properties of neural network inference.

1.4.1 Exploring network pruning

I have familiarised myself with and implemented pruning of weights in a neural network, *i.e.* sparsifying weight matrices (fine-grained pruning) or removing individual weight groups (structured pruning). This has led me to explore many different ways of applying pruning—how much to prune at a time, whether sparsity should be included as a regularisation term, whether it is possible to apply pruning before training or whether fine-tuning after pruning is necessary, and so on—which showed that applying a model compression method can be non-trivial. The experiments verified that over 95% of parameters could be removed without significantly affecting the classification accuracy, as was reported previously [70], which confirms that neural networks are compressible and adaptable.

1.4.2 Inference software and custom neural networks on MCUs

Microcontroller platforms have a peculiar memory hierarchy: there is typically a limited amount of on-chip memory available (<1 MB of read-only flash and read-write SRAM) and larger but slower memories, such as RAM, are not present (unless external storage is available). I have explored data movement costs of neural network execution, im-

plemented a low-overhead inference engine (a runtime), as well as model training and quantisation pipeline. Using external storage was found to be prohibitively slow, which forces neural network inference to fit within on-chip memory if latency is a concern.

This has taught me the specifics of microcontroller programming and different approaches one could take for network quantisation. Additionally, it has identified the peak memory usage as a big obstacle to deploying models to hardware with limited memory (model size, also an obstacle, has already been more extensively explored in compression research). I co-designed a model with a small memory footprint and leveraged previously-built quantisation pipeline to train it for the Visual Wake Words competition [20] (person detection under 256 KB of RAM and storage usage).

I have also created an interactive practical for deploying keyword recognition models onto an MCU[1], used in the "Principles of Machine Learning Systems (L46)" Part III / MPhil module.

1.4.3 Achieving low memory usage through operator reordering

To execute a neural network on a microcontroller, one may use some of the recent popular runtimes, such as TensorFlow Lite Micro and CMSIS-NN libraries, which implement quantised computation of many network layer types. There, networks are executed in a simple read-execute-write fashion: each operator (layer) is fully executed one at a time in a particular order; no partial evaluation or parallelism is supported.

Measuring peak memory usage algorithmically is not straightforward if the network architecture contains branches (diverging execution paths). Here's why: upon reaching a branching point, the runtime has a choice which operator to execute next. Different execution orders change which tensors will be held in memory at each step, which in turn affects the peak memory usage. To solve this problem, I devised a graph algorithm to find the lowest achievable peak memory usage for any architecture and the order of operator execution that achieves it [59]. By changing the execution order, the memory usage of our chosen model was reduced sufficiently to make it run on our target MCU (see an attached paper in Appendix A for a full description of the methodology).

Later on, I found that integrating this precise peak memory usage computation with NAS was instrumental in discovering models that use almost all available RAM. This algorithmic estimation has eliminated the need to benchmark candidate models on the device during search (device-in-the-loop NAS can be slow) or relying on under-approximations, as it was done in prior works [29].

[1] `https://colab.research.google.com/drive/17I7GL8WTieGzXYKRtQM2FrFi3eLQIrOM`

1.4.4 Machine learning in space (application investigation)

I have explored machine learning in a concrete constrained environment—on-device deep learning for space devices. Space hardware is typically quite underpowered due to it also having to be radiation-hardened and has strict power requirements, to make sure the device can complete its mission. This makes it akin to microcontroller hardware used on Earth and face similar problems with model deployment [8].

1.4.5 Neural architecture search with MCU resource constraints

The usual target for resource-efficient on-device AI is mobile devices; however, in comparison, MCUs have even less computational power. Therefore it is challenging to apply existing NAS methods to find microcontroller-friendly models: they were never designed to handle such extreme resource scarcity. I will briefly touch upon essential components of a constrained NAS system and modifications needed for finding MCU-sized models.

Search space. The search space is a set of all possible candidate architectures, together with a vocabulary of building blocks and a set of allowed changes NAS can make to the architectures during the search. In contrast to mobile or GPU platforms, MCUs are more sensitive to network hyperparameters: for example, choosing between a conv. layer with 172 and 192 channels is unlikely to make a meaningful difference for a GPU-sized model (though the former may have lower accuracy), but on an MCU choosing the larger layer may tip the model over the strict memory budget. This level of hyperparameter granularity (and, by a similar argument, finely-controlled layer connectivity) is not typically required for GPU- or mobile-level NAS due to the relative abundance of RAM.

Constraints. Intuitively, because performant neural networks are large, we would expect the best architectures to make use of all available resources and thus be located at a boundary between models that fit within the resource constraints and ones that do not. This requires NAS to be able to quickly and precisely compute resource requirements to know which networks lie just below the resource constraints.

Search algorithm. Using a highly granular search space and precise resource constraints increase the complexity of the search problem requires a suitable search algorithm. For example, I found that Bayesian optimisation using Gaussian Processes, while known to work in NAS [49, 47], performed worse than evolutionary algorithms, due to a smooth approximation to a network's accuracy (provided by GPs) being inadequate to capture

the subtle differences between networks in the search space.

Model compression. Model compression methodology, such as pruning, can be used in NAS to further shrink the memory footprint of models found during the search. This makes the search and the pruning share the task of determining the model's hyperparameters: the search produces a base network, which is then adjusted by pruning in a more informed way by discarding channels/units that were deemed unimportant during training.

I have incorporated the insights described above in my largest project to date. I built a neural architecture search (NAS) system, called µNAS, to automate the design of resource-efficient and highly-accurate MCU-sized networks. µNAS explicitly targets the three primary aspects of resource scarcity of MCUs: the size of RAM, persistent storage and processor speed, and represents an advance in resource-efficient models, especially for "mid-tier" MCUs with memory requirements ranging from 0.5 KB to 64 KB. On a variety of image classification datasets µNAS is able to (a) improve top-1 classification accuracy by up to 4.8%, or (b) reduce memory footprint by 4–13×, or (c) reduce the number of multiply-accumulate operations by ≈1700×, compared to existing MCU specialist literature and resource-efficient models. The paper is available in Appendix A.

1.5 Research directions

Informed by my current experience, I see the following promising approaches to reduce memory usage and latency at inference by employing holistic hardware-aware model design. The list is tentative, the exact scope and ambition of each project will be refined.

1.5.1 Neural architecture search with compiler-like optimisations

In a system where network architecture and execution software are tightly coupled with the hardware, evaluating the network one operator at a time may not be the most efficient approach. Depending on computation capabilities of the hardware, degree of parallelism, data throughput, amount of on-chip cache memory and which memory access patterns are favourable, execution engine can choose to evaluate multiple operators together, evaluate them partially, change data layout as necessary, *etc.*

To discover which architectures would benefit from optimisation by the execution engine the most, NAS has to be aware of hardware properties at search time, much like

a compiler. Doing so would eliminate the need for hand-crafted evaluation tricks, as those would be discovered automatically.

Searching for an optimal way to execute a sequence of linear algebra operations is an active research direction: Jia et al. [46] develop computation graph transformations that improve efficiency and works such as TVM [13] and Tensor Comprehensions [91] offer automated loop tiling and memory management, in particular for novel operators. However, these "model compiler" approaches aim to exactly preserve the computation, which prevents from discovering big optimisations that could have been achieved by a change in network architecture.

Incorporating compiler-like knowledge into NAS would require:

1. including a detailed model of the hardware and its capabilities, like those found in compilers;
2. incorporating knowledge about graph transformations that simplify the architecture without changing the result (akin to an optimisation pass in a compiler);
3. restricting the search space to cope with the combinatorial explosion.

As it is a large project, I intend to split it into **two** works that gradually build towards the goal:

1. Building upon µNAS, which incorporated model resource usage constraints (incl. simple scheduling to determine peak memory usage), I intend to incorporate other operator runtime properties (based on their implementation) and more sophisticated scheduling during search to compute more sophisticated latency constraints.
2. NAS may not be the only driver that suggests architecture changes: computation graph transformations coming from a "model compiler" or model compression (*e.g.* pruning) need to be incorporated in a way that provides feedback for the search and guides it towards more resource-efficient models.

1.5.2 Tackle challenges in a particular end-to-end application

It would be beneficial to apply insights from my prior work and the above research directions to a concrete application. This would introduce more problem-specific challenges and may yield more insights on the construction of resource-efficient models

and inference software. For example, consider *automatic speech recognition (ASR) on microcontrollers.* While a lot of developed microcontroller-specific methodology is applicable, adjustments may have to be made for:

- *Different kinds of models.* 2D-convolutional networks are typically used for images, but recurrent, transformer and 1D-convolutional networks are more common in audio and natural language processing.
- *Memory-efficient vocabulary representation.* In contrast to keyword recognition, which targets a limited vocabulary, ASR (in theory) targets the entire English language. This requires a (likely implicit) memory-efficient process of decoding detected utterances into words.
- *Near real-time inference speed.* If no audio caching is allowed, resulting models and software have to transcribe speech in near real-time (a tighter latency constraint).

1.5.3 Evaluation

Research outcomes can be evaluated by applying the methodology to real-world deployments of neural networks and quantifying the reduction in resource usage, and whether that resulted in additional gains, such as allowing the use of cheaper hardware.

1.6 Timeline

I believe the projects proposed above, in addition to work already carried out, come together to form a PhD thesis that tells a story of achieving hardware-aware neural network optimisation through model compression, tailored runtime design and neural architecture search, and doing so in an automated fashion to allow the methodology to be applied to a variety of tasks and target hardware.

Term	**Work planned**
MT 2020, LT 2021	Explore architectural and runtime changes that yield improvement in inference latency and memory use of neural networks. This will be done for a chosen problem set-up and target platform (*e.g.* audio processing on an MCU). In addition to work carried out by me previously, this will showcase the practical difficulties of building resource-efficient neural network-based applications and may yield further insights.

ET 2022	Take steps towards hardware-aware NAS, first by incorporating runtime properties of each operator and creating constraints based on those properties. This fulfils the first part of the proposed compiler-like NAS project. *Submit the second year report.*
Long vacation 2021	*Break for an internship.*
MT 2021, LT 2022	Continue work on NAS, incorporate computation graph transformations, scheduling information and model pruning. This fulfils the second part of the proposed compiler-like NAS project.
ET 2022	Thesis write-up, finish up outstanding experiments.
LV 2022	Finish write-up, thesis submission, viva, corrections.

Chapter 2

Literature survey

2.1 What are neural networks?

(Artificial) neural networks are machine learning models, given by collections of interconnected artificial neurons, designed to perform a specific computation. The first model of an artificial neuron was introduced by McCulloch and Pitts [68] in 1943, loosely inspired by the neurons in mammalian brains, which fire once they have been stimulated by sufficient input signal. Mathematically, an output neuron of a neuron is a weighted sum of its inputs, gated by an activation function. Rosenblatt [78] in 1958 has successfully applied this idea to develop a linear binary classifier, called the perceptron, whose linear sum coefficients (aka parameters or weights) were learned from a set of training data using an iterative algorithm.

While the potential applications of the perceptron seemed promising at the time [72], it was soon realised that it doesn't have sufficient representation power to learn complex functions. To address this, networks of perceptrons (where neurons are inputs to other neurons) arranged in layers—multilayer perceptrons (MLPs)—have been developed and the backpropagation algorithm [79] was devised to allow gradient-based function optimisation algorithms (*i.e.* gradient descent) to be used for training. Cybenko [24] showed that 2-layer MLPs can represent a wide variety of functions (*universal approximation theorem*), given a sufficient (finite) number of neurons.

Unfortunately, for large inputs, MLPs can get prohibitively big and difficult to train until convergence. To cope with this, neural networks are designed to better match the structure or properties of the data they're processing by imposing weight sharing (some neurons share parameters) and limited connectivity (some neurons only receive a part of the input).

2.2 Convolutional neural networks

While many compression methods can be applied to most neural networks, here I will mostly focus on convolutional neural networks (CNNs). CNNs [31] are the most popular research direction in deep learning; they have practically overtaken the field of computer vision and achieved state-of-the-art results in object detection, recognition, tracking, recommender systems, information extraction from images and others.

Much like MLPs, CNNs have a layered structure, consisting of *convolutional* layers. At the heart of a convolutional layer is a mathematical convolution operation of the input with a learned *kernel*. Tunable parameters of a convolution (hyperparameters) are the kernel size, stride, dilation factor, and the number of output channels.

The availability of high quality labelled image datasets has greatly contributed to the rapid development of new CNN architectures. The commonly used datasets are:

- **MNIST** [56]. The MNIST database consists of preprocessed handwritten digits from 0 to 9, of size 28x28 pixels, with 60000 images for training and 10000 images for testing.
- **CIFAR-10 and CIFAR-100** [50]. The CIFAR-10 and CIFAR-100 datasets contain images from 10 and 100 categories, respectively, such as ”bird”, ”dog”, and ”aeroplane”. Both datasets have 60000 32x32 colour images with 6000 images and 600 images per category, respectively.
- **ImageNet** [25]. The ImageNet dataset consists of 1.28 M and 50000 256x256 (after standard preprocessing) colour images for training and test sets, respectively. The images are labelled into 1000 categories, such as ”panda”, ”cat” and ”platypus”.

In what follows, I describe neural networks from a computational point of view and expose the high resource usage problem. Then I will discuss network compression methodology and neural architecture search in greater detail, that can be used to create more resource-efficient neural networks. Finally, we will touch upon what optimisations can be made in software to make models more resource-efficient.

2.3 Computation of neural networks

Neural network computations directly translate to linear algebra operations: computing the output of a set of independent neurons (a fully-connected layer) can be implemented

with a single matrix multiplication operation, followed by an element-wise application of the activation function; similarly, a convolutional layer can be computed using the mathematical convolution primitive, or matrix multiplication, if the input is laid out in memory in a special way (`im2col` algorithm [11]).

The kinds of computations required, incl. types of operations or the numbers of neurons of each layer, are defined by the architecture of the neural network. Traditionally, neural networks have followed a straightforward iterative (linear) computation path: outputs of one layer are used as inputs only to the succeeding layer to produce a new output. However, modern architectures contain skip-connections [38] and branches [84]. Thus, in software, the neural network can be represented as a computation graph: a directed acyclic graph (DAG), whose nodes correspond to a primitive linear algebra operation (*e.g.* a matrix multiply or an element-wise operation) and edges representing a data dependency between them.

Many software packages have been developed to facilitate the development and training of neural networks (and save practitioners from having to write GPU code!), such as Caffe [45], TensorFlow [1], Keras [18], Chainer [89] and PyTorch [73]. Frameworks offer the following set of features:

- **Static or dynamic computations graphs.** Dynamic computation graphs allow choosing which operations to run based on input data [89, 73], whereas static computation graphs are fixed at compile time [45, 1]. The former allows more flexibility, while the latter gives more scope for ahead-of-time optimisation.
- **Automatic differentiation.** Users don't have to implement backpropagation manually; the computation will be automatically augmented with nodes for computing gradients and parameter updates.

The current renaissance of neural networks is facilitated by the increased computational capabilities of the present hardware. During training, millions of parameters need to be updated [51] based on gradients computed on millions of data points, with the process repeated over multiple training *epochs* (dataset iterations), which requires specially-designed hardware to do reasonably quickly. The AlexNet [51] network popularised the use of graphics processing units (GPUs) and CUDA programming platform, which are designed to support simple parallel computations, making them a natural fit for this task. Original AlexNet implementation took 6 days to train the network until convergence on two nVidia GTX 580 GPUs. Nowadays, more custom hardware is available to accelerate training, such as tensor processing units (TPUs) from Google [48] and Graphcore Intelligence Processing Units (IPUs) [32].

	No. of param-s	Size	FLOPs	Accuracy on ImageNet [25]
AlexNet [51]	60 M	240 MB	1.4 G	61.0%
VGG-19 [82]	144 M	576 MB	39 G	74.5%
ResNet-152 [38]	57 M	228 MB	22.6 G	79.3%
MobileNetV2 [80]	3.4 M	13.6 MB	0.3 G	72.0%
ShuffleNet [94]	2.4 M	9.4 MB	0.3 G	67.6%

Table 2.1: Various CNNs and their resource usage.

Table 2.1 summarises the computational properties of popular convolutional neural networks to show the cost of inference (evaluating the network once to obtain predictions for a particular input). Due to having to fetch a large number of parameters and perform many floating-point operations, running such networks on consumer hardware, especially on mobile devices and microcontroller platforms, can be prohibitively expensive. Addressing resource consumption of neural networks is addressed by model compression.

2.4 Compact models

The following approaches have been explored in model compression literature to create compact models, suitable for inference in resource-constrained environments. Most approaches typically result in some amount of accuracy loss, exhibiting a trade-off between model's accuracy and its resource usage.

2.4.1 Quantisation

Weights of a neural network, as well as any intermediate tensors (activation maps), are typically represented by matrices of 32-bit floating-point numbers, occupying 4 bytes per element. However, it's possible to carry out computations at lower precision, saving the amount of storage and computation required by the model.

In practice, a quantisation implementation has 3 tunable properties:

- **Bit-width of parameters.** Courbariaux et al. [22] show that as little as 10 bits of precision is enough to both train and run the network for inference. Since then, the precision required at inference has been lowered to 8 [42], 4 [6] or 1 bit (binary networks are discussed below). Choosing bit-widths that are divisors of a native

word size of the target platform (*e.g.* 4, 8 or 16 bits for 32-bit platforms) may allow leveraging SIMD instructions to compute multiple outputs at the same time, as done in the CMSIS-NN library [52], and eliminate any decoding overhead.

- **Granularity.** Quantisation granularity dictates how finely the model is quantized. Options include selecting the bit-width per-parameter (fine-grained quantization), per-layer or for the entire model (coarse-grained quantization). Intuitively, finer quantization is more flexible and should approximate the full-precision model better. However, it also requires additional logic and bookkeeping due to the (potential) mismatch of bit-width between layer's operands, weights and activations. As a result, a common compromise is to perform layer-wise quantization [42, 65, 96, 60].

- **Representation formula.** At runtime, quantised values are typically represented by integers. This allows using more efficient integer arithmetic units, instead of floating-point units, as long as the bit-width is carefully chosen to capture the required range and computation is designed to perform saturating arithmetic. In particular, the following ways of representing values have been developed:

  Q-format (fixed-point) quantisation [60]. This a method splits an n bit number into integral and fractional parts, allocating a and b bits to each, respectively. Such number is denoted by $Qa.b$ (with $n = a + b$). For example, for 8-bit numbers, $Q_{0.7}$ (1 bit left for sign) representable values lie in the range $[-1, 1)$ with a step of 2^{-7}. Using Q-format quantisation allows leveraging integer ALUs but has the overhead of requiring to keep track of the decimal position for each operand. The representable range of values is symmetric and centred around 0, which may be a problem for long-tailed or asymmetric value distributions.

  Affine quantisation [42]. This implementation represents the $(i, j)^{th}$ element of a weight matrix $W_{i,j} \in \mathbb{R}$ as the *affine mapping* of an (8-bit) integer $Q_{i,j}$ using parameters S (floating-point scale) and Z (32-bit integral offset), defined as $W_{i,j} = S(Q_{i,j} - Z)$. This allows more precise control over the width and centre of the represented range by controlling shared parameters S and Z. Unless $Z = 0$, the offset term introduces extra terms that need to computed during matrix multiplication.

  Powers-of-two quantisation [33]. In a more extreme case, weights can be represented by powers of two: $W_{i,j} \in \{0, \pm 2^{-n-1}, \ldots, \pm 2^{n}\}$. This allows replacing a multiplication operation with a bit-wise shift, which is faster to compute.

Perhaps the most efficient implementation of neural network inference can be obtained where weights and activations are binary [23, 74, 30]. In this case, on hardware with

native 32-bit word support, 32 multiply-accumulate operations can be performed at once by using XNOR instruction for element-wise multiplication and POPCOUNT to accumulate across the product.

The success of a network quantisation will largely depend on the quantisation approach adopted during training. Methods for quantising the network can be largely grouped into *post-training* quantisation [6], where all matrices are quantised after the network has been trained and no access to training data is needed, and *quantisation during training*. The latter in particular has received a lot of research attention: many methods keep weights at full precision during training as proxies and rounding them during forward (inference) pass to obtain a quantised version [42]. The rounding function (a step) is not differentiable, so a straight-through estimator (STE) [7] is employed to let gradients flow through [95]. Alternatively, quantisation can be approached by learning a categorical distribution for the weights [66, 65] or more carefully designing gradient calculation [44]. The interaction of quantisation with optimiser parameters and batch normalisation layers is studied by Alizadeh et al. [2].

2.4.2 Pruning

Pruning weight matrices has been successfully used to both regularise and reduce the size footprint of the neural network. In practice, pruning methods can be categorised based on the following properties:

- **Criteria/salience metric.** Salience metric tells which weights are deemed unimportant and can be removed from the network. Some works use a weight's magnitude [36]—arguing that, intuitively, a weight is unimportant if it's small—and some use a theoretically-justified Hessian-based measure [37], or a learned importance metric [70]. However, it should be noted that computing an inverse of a Hessian matrix is prohibitively expensive for large networks and some simplifying assumptions should be applied.

- **Regularisation.** Extra specially-designed regularisation terms can be added to the optimisation objective to produce more prunable parameters under the salience criteria. For example, some magnitude-based compression methods add an L_1 or an L_2 loss [3].

- **Granularity.** Pruning can be applied to remove individual weights, resulting in sparse weight matrices (fine-grained pruning) or to remove weights in groups, for example removing channels (filters) in CNNs [88, 58].

- **Mode.** Pruning mode defines the process by which the network's parameters are pruned. If a practitioner is targeting a particular sparsity level, all of the parameters that need to be discarded can be either pruned at once (one-shot pruning) [57, 58] or in multiple instalments (iterative pruning) [36, 62]. The network is usually retrained, or trained for longer, to regain as much lost prediction accuracy as possible.

2.4.3 Cheaper building blocks

Replacing expensive operations, such as large convolutions and matrix multiplications with cheaper alternatives is one of the most popular approaches to model compression.

Low-rank decomposition replaces a weight matrix with a product of two or more lower rank matrices, which allows lowering both the required storage and the computational complexity of a layer. Weight matrices of fully-connected layers can be split using SVD decomposition [9, 53], and convolutional layers have been accelerated by decomposing convolutional kernels using rank-1 tensors [26], and depthwise-separable convolutions [43, 19].

Kernels and weight matrices can also be decomposed as a linear combination of some basis functions, *i.e.* a kernel is represented by a set of coefficients in the new basis. If basis functions are extensively shared among many kernels, this allows compressing the representation of a model. New basis can be either fixed [90] or learned at training time [35, 21].

Finally, a structural constraint can be imposed on the weight matrix, which acts as a form of weight sharing and allows a compressed version of a matrix to be stored in memory. For example, Cheng et al. [16] use a circular projection, which allows the Fast Fourier Transform (FFT) to be used for computation.

Typically, once a cheaper primitive has been built using one of the above methods, it is applied throughout the network and an entire architecture is assembled. Popular such architectures include MobileNet [80], assembled using "inverted residual blocks"; SqueezeNet [40], assembled using blocks that induce an information bottleneck; ShuffleNet [94], that primarily uses pointwise group convolutions and inter-group channel shuffling and LeanResNet [27] impose extra structure on convolutional kernels. Efficient architectures can also be designed automatically [85, 86], which we discuss in Section 2.5.

2.4.4 Distillation

Knowledge transfer, or distillation, compression methods involve teaching compact student models by mimicking larger teacher models. Seminal work by Ba and Caruana [4] discusses differences between shallow and deep networks. As shallow networks are difficult to train from scratch, the authors introduce a stronger training signal by making a shallow network match its logits (outputs of the final layer, softened label probabilities) to those of a deeper network.

In general, student networks are kept deep to ease the training. Further works explore other ways of incorporating hints and output match points from the teacher network:

- Luo et al. [67] attempt to preserve outputs of other hidden layers, which intuitively correspond to high-level features, rather than final label probabilities.
- Chen et al. [12] use function-preserving transformations to quickly transfer knowledge between networks.
- Romero et al. [77] train thinner networks by synchronising more intermediate representations between teacher and student, coping with different representation widths by introducing a linear transformation.
- Zagoruyko and Komodakis [92] train students to mimic attention maps of the teacher network, effectively allowing them to focus on the same areas in the feature maps.

2.5 Neural architecture search

Until recently, advancing state-of-the-art using deep learning has required manually designing and tweaking neural network architectures. Neural architecture search aims to automate this task and has since achieved state-of-the-art in several computer vision problems [76, 85].

NAS designs novel architectures by sampling the search space and evaluating the picked candidate architecture based on a certain performance metric, usually accuracy on the validation set. Different approaches to NAS typically differ in how they: (a) design the search space (space of representable architectures); (b) evaluate candidate models; (c) choose the action space and the search algorithm.

2.5.1 Search space

The search space defines which architectures can be represented (and thus discovered) by NAS. The design of the search space also determines the difficulty of the search.

- **Linear architectures.** Much like early manually-designed neural network architectures, early work in NAS employs linear architectures, where the input is transformed iteratively by each successive layer. Thus the degrees of freedom permitted in the search space are the layer types (convolution, full-connected, depthwise-separable convolution, etc.), their parameters (number of hidden units, stride and kernel size of convolution, etc.) and the maximum number of layers.

- **Branching architectures.** Most current state-of-the-art architectures contain branches [19, 38]. In NAS, if the restriction of linearity above is relaxed, architectures can become arbitrary directed acyclic graphs (DAGs), with the search bounded by the number of nodes in the DAG. In addition to having to choose layer type and their parameters, the search will have to explore many inter-layer connectivity options, which may result in a combinatorial explosion, making the search prohibitively expensive.

- **Block architectures.** To restrict the search space, recent works adopt a combination of the above approaches by considering linear architectures consisting of repeated motifs, called *blocks* or *cells* (which may contain branches). Many existing hand-crafted architectures follow this design pattern [80, 40, 85], which splits the problem into 2 parts: designing the architecture of a block and designing the connectivity between the blocks (the *macro-architecture*). The former can be created using NAS and the macro-architecture can either be chosen manually or borrowed from known networks.

  In CNNs, successive layers typically gradually reduce the spatial dimension of the input, while increasing the feature (filter, channel) dimension. This requires cell designs to be parametrised to accept inputs of several different sizes and reduce their dimensionality for the following cell. Tan and Le [85] restrict the search space by parametrising the output dimensionality of each successive cell, as well as the depth and input resolution of the network via a single compound scale parameter, which allows the authors to obtain models of different sizes by changing the scale.

2.5.2 Evaluating candidate models

During the search, the optimisation algorithm will have to evaluate many candidate architectures. As the aim of the search is typically accuracy, evaluating an architecture involves training it from scratch, which can take days on thousands of GPUs [97, 86]. This calls for ways to speed up the evaluation, typically achieved by taking shortcuts during training or employing cheaper proxy metrics:

- **Extrapolating performance.** The training time of a model can be reduced by simply terminating it earlier, training on a smaller subset of data, lower-resolution images or using a smaller macro-architecture. This would underestimate the performance of a network, but do so consistently for all candidate models.

  Alternatively, learning curves (loss or accuracy plot) of a network can be extrapolated to see which candidate models can be terminated early, based on the initial portion of the curves and architectural hyperparameters. Liu et al. [63] guide the search by training a surrogate model to predict the accuracy of a candidate network based on its architecture. However, using another machine learning model to predict performance can cause the performance to suffer from the lack of training data (we want to make as few "full" evaluations as possible during search) and poor performance on the out-of-distribution inputs (model won't perform well on the kinds of inputs it has not seen before).

- **Transfer learning.** Novel models can be built by gradually expanding candidates: each candidate model's weights can be initialised to those of a previous candidate. This, followed by a few epochs of training to make use of the increased capacity, allows to drastically cut the computation required for evaluation, at the cost of restricting the search procedure.

  Sharing weights between candidate models has been shown to work well where candidate models are sub-graphs of a larger model [10]. Then, NAS can prune certain operations, while the weights of the remaining layers are initialised to those of the parent model. Similarly, while being fast, this restricts the search space.

2.5.3 Search algorithms and action space

Search algorithms and action spaces define how the NAS explores the search space. The following approaches have been explored in NAS literature:

- **Genetic algorithms.** Using genetic (evolutionary) algorithms to design neural networks was pioneered by Miller et al. [69] in 1989. Genetic algorithms evolve a population of architectures, by randomly choosing parent architectures that generate a new architecture (with a small *mutation* probability) in a *crossover* step, and retaining a certain number of the fittest (best-performing) architectures for the next iteration.

  Network architectures are complex objects, so meaningful and robust definitions for the crossover and mutation steps, as well as the individual retention policy, are required. Existing methods employ tournament selection for choosing parents and discard either the oldest or the worst performing architecture at each step [75, 76].

- **Distribution sampling and optimisation.** In a tree-structured search space, Monte Carlo Tree Search can be applied [71]. Alternatively, Bayesian optimisation is a popular technique for hyperparameter search and has recently been applied to jointly optimise for a network's architecture and hyperparameters by efficiently exploring high-dimension spaces [93]. The authors cast NAS an optimisation problem in a categorical distribution and use the BOHB method [28] to explore the distribution.

  Categorical distributions can also be optimised by employing continuous relaxation (approximating a categorical distribution with a continuous one, that gets closer and closer to the categorical distribution over time). This allows using gradient-based methods to perform the optimisation itself and even combining it with neural network training.

- **Reinforcement learning.** A reinforcement learning agent can be used to design the networks. In a stateful setting, an agent modifies existing candidate architectures using a set of allowed actions. In a stateless setting, the agent's action space is the same as the search space, and the agent just performs a single choice. To represent the agent's policy, Q-learning [5] or an RNN controller trained using REINFORCE or proximal policy optimisation [97, 98] were used.

- **Random search.** Random search is a simple, but a surprisingly effective way to search high-dimensional search spaces. Some works found previously listed techniques only marginally better than random search [76] or not at all better [81].

2.5.4 NAS for compact models

Neural architecture search can also be used to produce compact models, that is searching for the best-performing architecture under certain resource constraints, such as model

size or memory usage. Constraint satisfaction is an NP-complete problem, but the search can be turned into an unconstrained optimisation problem by, for example, designing the search space to only include architectures that satisfy the constraints.

Among other works that search for efficient models [83, 64], we note the following ones:

- Fedorov et al. [29] obtain microcontroller-level CNNs (<2 KB) for visual recognition by using multi-objective Bayesian optimisation to jointly optimise for model accuracy, working memory and model size, while also incorporating pruning techniques to more efficiently optimise the sizes of fully-connected and convolutional layers.
- He et al. [39] use reinforcement learning to sample the hyperparameter search space of known models layer by layer, where each layer also has an associated sparsity ratio (obtained by pruning). The actor learns two policies: a resource-constrained one that ensures that the agent is always above a certain sparsity ratio and an accuracy-guaranteed one which optimises both accuracy and resource usage.
- MNasNet [86] directly optimises for accuracy and latency on mobile devices by evaluating models on a grid of Pixel 1 phones (expensive!) instead of opting for proxy metrics, such as the number of FLOPs, and uses a factorised (block) search space to improve search time.

2.6 Deep learning inference software

In software, neural network computation is represented as a computation graph, with nodes corresponding to primitive linear algebra operations. We will discuss both intra-node optimisations, *i.e.* improving memory usage or speed of particular operations, as well as inter-node optimisations, *i.e.* optimising by changing the computation graph. In addition to methods listed below, Lane et al. [53] show that some model compression techniques can be applied at runtime. Intra-node optimisations include:

- **Manually optimised implementations.** Many software packages ship optimised implementations of matrix multiplication and convolution operations. Due to different hardware properties and programming paradigms, efficient implementations would be different for each target platform. For example, cuDNN, which powers most deep learning frameworks on nVidia GPU platforms, contains

manually-tuned tiled matrix multiplication algorithms and several implementations of a convolution (*e.g.* via the im2col algorithm [11] or a Winograd transform [55]), which make best use of the parallel programming and limited amount of shared memory on nVidia hardware. For CPU and mobile phone platforms, Eigen [34] is commonly used, which makes use of caching and vectorisation. For ARM MCU platforms, CMSIS-NN [52] offers quantised operator implementations with SIMD support.

- **Automatically discovered optimised implementations.** Neural network researchers may use an operation that doesn't map to any existing linear algebra primitives, and thus a highly-optimised optimisation would not be available. TVM [13] and Tensor Comprehensions [91] address this gap by allowing to define new operators in a custom DSL. An efficient implementation is then found using an ML-based cost and evolutionary algorithms, respectively. MLIR [54] also introduces an intermediate optimisation to enable the LLVM compiler to perform required optimisations. Optimising an implementation involves performing vectorisation, loop tiling (*e.g.* using polyhedral compilation techniques) and data layout optimisation to make the best use of caching and memory locality. TVM also performs latency hiding and layer fusion optimisations, discussed below.

Inter-node optimisations include:

- **Operator fusion using their mathematical properties.** Properties of some deep learning layers allow combining (fusing) them together to reduce the number nodes in a computation graph. For example, batch normalisation [41] layers perform subtraction of a learned mean and division by standard deviation. Since both are constant at inference, this linear operation can be "folded" into a preceding linear operation (a convolution or a matrix multiplication), by dividing its weights by the standard deviation and appropriately changing the bias.

- **Equivalent graph transformations.** Each time a new node in the computation graph is executed, the runtime faces certain context switch overheads. This can be improved by making nodes do more work, if possible: for example, two matrix multiply operators, that share the same weight matrix, can be combined into one with its inputs appropriate concatenated before and split after the multiplication (see Figure 2.1). Note that concatenation and split nodes don't have to exist at runtime—producer or consumer nodes can simply be instructed to write to or read from certain memory addresses. Jia et al. [46] generalise this idea by discovering equivalent graph transformations that preserve both the mathematical identities

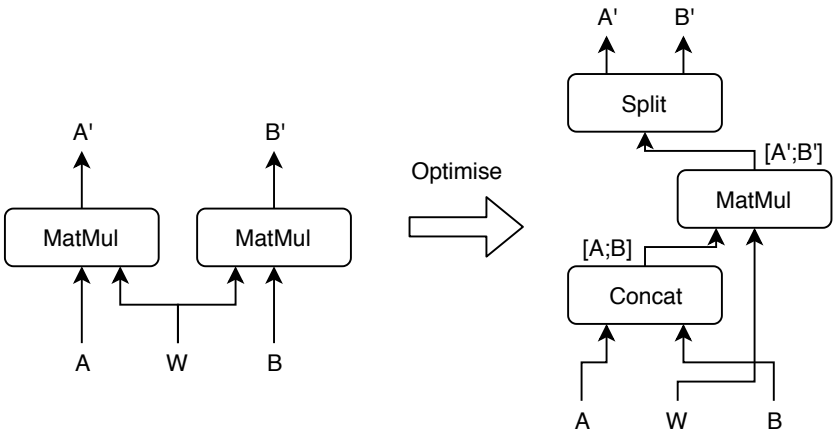


Figure 2.1: An example graph transformation to increase computational efficiency.

(encoded in first-order logic) and computation precision (verified using a set of generated input matrices and outputs compared before and after transformation).

- **Optimisation in generated code.** Computation of a node is typically performed by a tailored piece of code, called a *kernel*, launching which, as discussed previously, has a certain overhead. A deep learning compiler can discover subgraphs that can be merged into a single custom "super-node", powered by a single kernel. Generating code for multiple operators at once allows memory access optimisations: for example, combining element-wise non-linearities (*e.g.* ReLU) with a previous layer, by computing a non-linearity as the last step before the value is written to memory. TensorFlow's [1] XLA compiler performs such optimisations for GPU kernels.

- **Evaluation strategies.** In order to evaluate a node, its inputs and the output buffer should be present in memory. This can become a memory bottleneck when evaluating networks with large layers on platforms with a limited amount of RAM, such as MCUs. This can be circumvented by evaluating a set of nodes partially or as needed by the downstream consumer node. Sandler et al. [80] describe such optimisation for the "bottleneck residual block" in the MobileNetV2 network, which consists of a linear operation, followed by a channel-wise operation, followed by another linear operation. The authors note that the input to the channel-wise operation doesn't need to be materialised in full—only 1 channel at a time is required—which allows them to reduce the memory usage bottleneck. Other approaches explore different ways of scheduling nodes (different topological orderings of the graph) to improve on runtime memory usage [59].

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D Lane, and Yarin Gal. An empirical study of binary neural networks' optimisation. 2018.

[3] Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, pages 2270–2278, 2016.

[4] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[5] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[6] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems*, pages 7948–7956, 2019.

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

[8] P Blacker, CP Bridges, and S Hadfield. Rapid prototyping of deep learning models on radiation hardened cpus. In *2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 25–32. IEEE, 2019.

[9] Chenghao Cai, Dengfeng Ke, Yanyan Xu, and Kaile Su. Fast learning of deep neural networks via singular value decomposition. In *Pacific Rim International Conference on Artificial Intelligence*, pages 820–826. Springer, 2014.

[10] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[11] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 2006.

[12] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[14] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *arXiv preprint arXiv:1807.07928*, 2018.

[15] Hsin-Pai Cheng, Jinwon Lee, Parham Noorzad, and Jamie Lin. QTravelers Visual Wake Words contest submission. `https://github.com/newwhitecheng/vwwc19-submission` (Accessed Sep 2019), 2019.

[16] Yu Cheng, Felix X Yu, Rogerio S Feris, Sanjiv Kumar, Alok Choudhary, and Shi-Fu Chang. An exploration of parameter redundancy in deep networks with circulant projections. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2857–2865, 2015.

[17] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[18] François Chollet et al. Keras. `https://keras.io`, 2015.

[19] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[20] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset. *arXiv preprint arXiv:1906.05721*, 2019.

[21] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999, 2016.

[22] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[23] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[24] George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[26] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[27] Jonathan Ephrath, Lars Ruthotto, Eldad Haber, and Eran Treister. Leanresnet: A low-cost yet effective convolutional residual networks. *arXiv preprint arXiv:1904.06952*, 2019.

[28] Stefan Falkner, Aaron Klein, and Frank Hutter. Practical hyperparameter optimization for deep learning. 2018.

[29] Igor Fedorov, Ryan P. Adams, Matthew Mattina, and Paul N. Whatmough. SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers. pages 1–26, 2019. URL `http://arxiv.org/abs/1905.12107`.

[30] Joshua Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel. Riptide: Fast end-to-end binarized neural networks. *Proceedings of Machine Learning and Systems*, 2, 2020.

[31] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[32] GraphCore. Introducing the GraphCore RackScale IPU-POD (tm). `https://www.graphcore.ai/posts/introducing-the-graphcore-rackscale-ipu-pod` (Accessed Nov 2019), 2018.

[33] Denis A Gudovskiy and Luca Rigazio. ShiftCNN: Generalized low-precision architecture for inference of convolutional neural networks. *arXiv preprint arXiv:1706.02393*, 2017.

[34] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[35] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[36] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[37] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

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

[39] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[40] 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. *arXiv preprint arXiv:1602.07360*, 2016.

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

[42] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[43] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[44] Sambhav R. Jain, Albert Gural, Michael Wu, and Chris H. Dick. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. *arXiv preprint arXiv:1903.08066*, 2019.

[45] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[46] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62. ACM, 2019.

[47] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-Keras: an efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019.

[48] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[49] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in neural information processing systems*, pages 2016–2025, 2018.

[50] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10/100 (canadian institute for advanced research). 2010. URL `http://www.cs.toronto.edu/~kriz/cifar.html`.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[52] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: Efficient neural network kernels for ARM Cortex-M CPUs. *arXiv preprint arXiv:1801.06601*, 2018.

[53] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power

deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*, page 23. IEEE Press, 2016.

[54] Chris Lattner and Jacques Pienaar. Mlir primer: A compiler infrastructure for the end of moore's law, 2019.

[55] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[56] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 1999. URL `http://yann.lecun.com/exdb/mnist/`.

[57] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

[58] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[59] Edgar Liberis and Nicholas D Lane. Neural networks on microcontrollers: saving memory at inference via operator reordering. *arXiv preprint arXiv:1910.05110*, 2019.

[60] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. *arXiv preprint arXiv:1511.06393*, 2015.

[61] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. MCUNet: Tiny deep learning on iot devices. *arXiv preprint arXiv:2007.10319*, 2020.

[62] Tao Lin, Sebastian U Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. Dynamic model pruning with feedback. *arXiv preprint arXiv:2006.07253*, 2020.

[63] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[64] Yu Liu, Xuhui Jia, Mingxing Tan, Raviteja Vemulapalli, Yukun Zhu, Bradley Green, and Xiaogang Wang. Search to Distill: Pearls are Everywhere but not the Eyes. *arXiv preprint arXiv:1911.09074*, 2019.

[65] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298, 2017.

[66] Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. Relaxed quantization for discretized neural networks. *arXiv preprint arXiv:1810.01875*, 2018.

[67] Ping Luo, Zhenyao Zhu, Ziwei Liu, Xiaogang Wang, and Xiaoou Tang. Face model compression by distilling knowledge from neurons. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[68] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[69] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. Designing neural networks using genetic algorithms. In *ICGA*, volume 89, pages 379–384, 1989.

[70] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org, 2017.

[71] Renato Negrinho and Geoff Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.

[72] Mikel Olazaran. A sociological study of the official history of the perceptrons controversy. *Social Studies of Science*, 26(3):611–659, 1996.

[73] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*, 2017.

[74] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[75] 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. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org, 2017.

[76] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Aging evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence*, 2019.

[77] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[78] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[79] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[80] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[81] Christian Sciuto, Kaicheng Yu, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*, 2019.

[82] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[83] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path NAS: Designing hardware-efficient convnets in less than 4 hours. *arXiv preprint arXiv:1904.02877*, 2019.

[84] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[85] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[86] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[87] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[88] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster gaze prediction with dense networks and fisher pruning. *arXiv preprint arXiv:1801.05787*, 2018.

[89] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[90] Vincent WS Tseng, Sourav Bhattachara, Javier Fernández-Marqués, Milad Alizadeh, Catherine Tong, and Nicholas D Lane. Deterministic binary filters for convolutional neural networks. International Joint Conferences on Artificial Intelligence Organization, 2018.

[91] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[92] Sergey Zagoruyko and Nikos Komodakis. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer. *arXiv preprint arXiv:1612.03928*, 2016.

[93] Arber Zela, Aaron Klein, Stefan Falkner, and Frank Hutter. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *arXiv preprint arXiv:1807.06906*, 2018.

[94] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018. ISBN 9781538664209. doi: 10.1109/CVPR.2018.00716.

[95] 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. *arXiv preprint arXiv:1606.06160*, 2016.

[96] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive quantization for deep neural network. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[97] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[98] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

Appendix A

Attached papers

I attach the following 2 papers produced during my PhD so far:

- *"Neural networks on microcontrollers: saving memory at inference via operator reordering"*. Published at the "On-device Intelligence Workshop at MLSys 2020: 3rd Conference on Machine Learning and Systems".
- *"µNAS: constrained neural architecture search for microcontrollers"*. Currently under review at the "MLSys 2021: 4th Conference on Machine Learning and Systems".

Neural networks on microcontrollers: saving memory at inference via operator reordering

Edgar Liberis[1]
edgar.liberis@cs.ox.ac.uk

[1]Department of Computer Science
University of Oxford, Oxford, UK

Nicholas D. Lane[1,2]
nicholas.lane@cs.ox.ac.uk

[2]Samsung AI Center Cambridge
Cambridge, UK

Abstract

Designing deep learning models for highly-constrained hardware would allow imbuing many edge devices with intelligence. Microcontrollers (MCUs) are an attractive platform for building smart devices due to their low cost, wide availability, and modest power usage. However, they lack the computational resources to run neural networks as straightforwardly as mobile or server platforms, which necessitates changes to the network architecture and the inference software. In this work, we discuss the deployment and memory concerns of neural networks on MCUs and present a way of saving memory by changing the execution order of the network's operators, which is orthogonal to other compression methods. We publish a tool for reordering operators of TensorFlow Lite models[1] and demonstrate its utility by sufficiently reducing the memory footprint of a CNN to deploy it on an MCU with 512KB SRAM.

1 Introduction

Deep learning can bring computational intelligence to personal and IoT devices. Using deep learning models directly on the edge devices allows for greater cost-efficiency, scalability and privacy for end-users, compared to relying on a remote server to carry out the processing. However, the development of lightweight neural networks, suitable for such underpowered hardware, is centred around mobile phones as the target platform.

Here, we venture further and explore a more resource-constrained platform—microcontroller units (MCUs). MCUs are cheap, widespread and are geared towards energy-efficient workloads. They offer an alternative to designing a purpose-built custom chip, allowing to save on development cost and time. However, a unit typically consists of a low-frequency processor and only several hundred kilobytes of on-chip memory [1], and thus severely underpowered compared to mobile devices.

Specially designed network architectures and inference software are required to cope with hardware constraints of MCUs. In this work, we: (a) discuss memory limitations of a microcontroller platform and how it affects neural network deployment; (b) devise a way to minimise the peak memory usage of a neural network by making inference software follow a particular execution order of its operations.

We implement our methodology as a tool for reordering operators within TensorFlow Lite models.[1] We successfully apply it to a chosen convolutional neural network, reducing the memory footprint enough to make it fit within the on-chip memory of our microcontroller platform, which would not have been possible using the default provided operator execution order. Operator reordering is carried

[1]Available for download at `https://github.com/oxmlsys/tflite-tools`

A version of this paper appears at the On-device Intelligence Workshop at MLSys 2020: 3rd Conference on Machine Learning and Systems.

out only at inference and does not change the architecture or the output of a neural network, making it fully orthogonal to many other network compression methods.

2 Background

2.1 Neural network execution

A neural network can be thought of as a computation graph which expresses dependencies between individual operations (also called layers or operators). An operator, *e.g.* a 2D convolution or addition, takes one or more input tensors and produces a single output. Modern deep learning frameworks optimise the network's computation graph for inference in advance by *e.g.* fusing adjacent operators and folding batch normalisation layers into preceding linear operations. The execution proceeds by evaluating one operator at a time in a topological order of nodes in the graph.

An operator requires buffers for its inputs and output to be present in memory before its execution can commence. Once the operator has finished executing, memory occupied by its inputs can be reclaimed (if not use elsewhere) and the output buffer will eventually be used as an input to other operators. We define a *working set* as a set of tensors that need to be kept in memory at any given point in execution. This comprises input and output tensors of a pending operator and other tensors that were already produced and need to be held back in memory for subsequent operators.

Classic neural network architectures, such as the original multilayer perceptron, AlexNet [2], VGG [3], consist of a linear sequence of layers, which are iteratively applied to transform the input. However, more recent architectures, such as ResNet [4], Inception [5], NasNet [6], introduce divergent processing paths where the same tensor can be processed by several layers, *i.e.* their computation graph is no longer linear and has branches. This means that the inference software may have multiple operators available for execution at any given step.

2.2 Resource scarcity of microcontroller hardware

A microcontroller unit that would be powerful enough to execute neural networks reasonably quickly (*e.g.* ARM Cortex M series) typically has a low-frequency RISC processing core (up to 400 MHz) and 128–2048KB of on-chip memory [1]. The memory is partitioned into read-write static RAM (SRAM) and read-only NOR-Flash memories, with the latter housing the executable code and static data. In contrast to mobile or desktop hardware, there are no intermediate levels in this memory hierarchy, although the set-up may have some backing storage. A backing storage, *e.g.* an SD-card, typically has a high capacity but is slow [7] and power-costly to access ($\approx$100x more energy required to read a value outside of on-chip memory [8]).

The lack of intermediate memories forces applications to fit within the on-chip memory to remain fast. For neural networks, this makes aiming for a small peak working set size and parameter count (model size) an important goal in model design. Note that it's not necessary to pursue the maximal memory saving—aiming just below the on-chip memory capacity is sufficient.

Memory requirements of a neural network can be directly mapped to the two types of on-chip memory. Parameters (trainable weights, constants) of a network are immutable and can be embedded into the executable code as static data stored in NOR-Flash. Any intermediate tensors that are dependent upon input (so-called activation matrices) are produced at runtime and would have to be stored in SRAM. Thus the model size and peak memory usage are constrained by the capacities of NOR-Flash and SRAM memories, respectively.

In Section 4, we show how choosing operator execution order affects which tensors reside in SRAM (are in the working set). We exploit this to minimise the peak working set size (peak memory usage).

3 Related work

The design of compact models is an active topic of deep learning research, albeit usually under less extreme constraints than those of microcontroller hardware. One can obtain a smaller neural network by using layer decomposition [9, 10], pruning [11, 12], quantisation [13] and binarisation [14, 15], distillation [16] or exploiting sparsity [17]. Popular mobile-friendly CNN architectures include

MobileNet [18] and ShuffleNet [19]. MNasNet [20] and EfficientNet [21] develop architecture search algorithms to design a network within a certain floating-point operation count or memory budget. In particular, Fedorov *et al.* [22] incorporate the maximal working memory size of an operator into their optimisation goal.

A relatively underexplored set of methods include complex evaluation strategies for parts of the network to save memory at runtime. For example, authors of MobileNet [18] note that a building block of their model has a channel-wise operation, whose output is accumulated into another tensor, which allows processing the input tensor in parts. Also, Alwani *et al.* [23] propose not to materialise an output tensor of a large convolution operation in memory at all, and compute its individual output elements as needed by succeeding operators.

The development of low-power machine learning models is fostered by the TinyML Summit [24] and the Visual Wake Words competition [25], which looked for performant MCU-sized CNNs for person detection. Compact deep learning models have been built for use on wearables devices [26] and for keyword spotting [27, 28]. Concerns about the memory usage of neural networks and data movement during execution are also discussed in neural network accelerator chip design literature [29–31].

4 Methods and implementation

Neural networks whose computation graphs contain branches allow some freedom over the order of evaluation of their operators. When execution reaches a branching point, the inference software has to choose which branch to start evaluating next. This choice can affect which tensors need to be kept in memory (working set), so we can construct an execution schedule that minimises the total size of the working set at its peak (memory bottleneck).

To illustrate this, Figure 1 shows an example of a computation graph, adapted from a real-world CNN. Evaluating operators as numbered 1 through to 7 will result in peak memory usage of 5216, coming from operator #3 (input and output buffers + the output of operator #1 that is held back for operator #4). However, fully evaluating the rightmost branch first (execution order 1, 4, 6, 2, 3, 5, 7), would result in peak memory usage of 4960, coming from operator #2 (input and output buffers + output of operator #6 that is held back for operator #7). Appendix A gives a more detailed breakdown of the memory usage during computation, together with plots produced by our tool, for both default and optimised operator schedules.

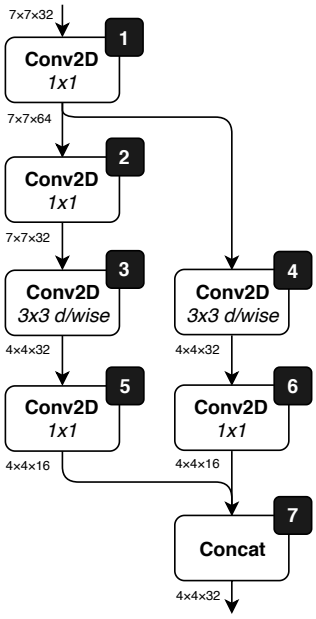


Figure 1: An example computation graph consisting of 1x1 and 3x3 depthwise convolution operators. Annotations on arrows represent tensor sizes.

We approach finding a memory-optimal execution schedule for an arbitrary computation graph by algorithmically enumerating all execution schedules and calculating their peak memory usage. To simplify the problem, we assume that no operator will be executed twice (this assumption is also made in TensorFlow). A computation graph is a directed acyclic graph (DAG) and any topological order of its nodes would produce a valid execution schedule; in general, enumerating all topological orders of a DAG is an explored problem in graph algorithms literature [32].

In Algorithm 1 (procedure MEM), we describe a dynamic programming algorithm that is concerned with the minimal peak memory usage required to produce (and keep in memory) a set of tensors X. It enumerates execution schedules recursively by trying to "undo" operators that produced each of the tensors in X. The algorithm should be invoked on a set of network's output tensors and the optimal execution schedule can be traced by checking which recursive calls contributed to the answer. The complexity of the algorithm is $\mathcal{O}(|V|2^{|V|})$, where $|V|$ is the number of operators of the network.

To simplify the implementation, the algorithm begins by filtering out tensors that don't have an operator that produced them (so-called constants), as those just contribute to memory usage and don't affect the execution schedule. A restriction that no operator is evaluated twice is implemented by

Algorithm 1 Computing the minimal peak memory usage of a neural network. PARTITION function splits a set into two using a predicate; *producer(x)* denotes an operator that produced tensor x.

procedure MEM(X) ▷ Minimum amount of memory needed to compute tensors in X
  ▷ Partition tensors into constants (no producer) and activation matrices
  $cs, as \leftarrow$ PARTITION(X, x : *producer*(x) **is None**)
  if as is empty **then**
    return $\sum_{c \in cs} |c|$ ▷ No operators left to order, report sizes of remaining constants
  end if
  $m \leftarrow \infty$
  for x **in** as **do** ▷ Try to unapply the operator that produced x
    rs $\leftarrow as \setminus x$ ▷ Remaining tensors that need to be kept in memory
    is $\leftarrow$ *producer*(x).*inputs* ▷ Tensors required to produce x
    if any(x is a predecessor of r **for** r **in** rs) **then**
      continue ▷ x is a predecessor to r, so *producer*(x) would have to be evaluated twice
    end if
    ▷ Peak memory usage will be determined by either the producer of x—*i.e.* memory used its input tensors (is), output tensor (x) and other tensors (rs)—or by other operators in the execution path (recursive case MEM($rs \cup is$))
    $m' \leftarrow \max(\text{MEM}(rs \cup is), \sum_{t \in rs \cup is \cup \{x\}} |t|)$
    $m \leftarrow \min(m, m')$ ▷ Pick the execution path that gives minimal memory usage
  end for
  return $\sum_{c \in cs} |c| + m$
end procedure

checking whether an operator is a predecessor to any of the remaining tensors, as this would require it to be executed at some point again in the schedule. Note that MEM(X) may be invoked on the same set of tensors multiple times (from different execution paths), so it should be memoized (*i.e.* the output should be cached) to avoid recomputing the result.

We use a lightweight TensorFlow Light Micro inference engine [33] (henceforth micro-interpreter) to run the neural network on the MCU itself. At the time of writing[2], the software did not support reclaiming memory from tensors that were no longer needed, so we implement our own dynamic memory allocator for tensor buffers. Internally, TensorFlow Lite assumes that tensors reside in contiguous blocks of memory and cannot be fragmented. The memory allocator is only used by the micro-interpreter, which allows us to ensure that C/C++ pointers to memory blocks are not being remembered anywhere in the code. This enables us to move buffers in memory as needed for defragmentation. We adopt a very simple defragmentation strategy of moving all tensor buffers to the start of the memory region as much as possible after the execution of every operator.

5 Experiments

We deploy a neural network onto an MCU with both default and optimised operator schedules to exhibit the difference in memory usage. We use one of the winning submissions of the Visual Wake Words competition [25], called SwiftNet Cell [34, 35], as it has only 250KB of parameters and contains many branches, which enables us to showcase the benefits of reordering. The model is run using the modified micro-interpreter (as described above) on a NUCLEO-F767ZI prototyping board [36]. The board features a Cortex M7 processor, running at 216Mhz, and has 512KB of SRAM.

[2] At the time of publication of this pre-print, a dynamic memory allocator has been implemented by maintainers of TensorFlow Lite Micro, making this change no longer necessary. However, we keep the description of our memory allocation strategy, as well as the power and latency measurements, as an illustrative example of memory management overheads.

	SwiftNet Cell		MobileNet v1	
	Default order	Optimal order	Static alloc.	Dynamic alloc.
Peak memory usage (excl. overheads)	351KB	301KB	241KB	55KB (↓ 186KB)
Execution time	*N/A*	10243 ms	1316 ms	1325 ms (↑ 0.68%)
Energy use	*N/A*	8775 mJ	728 mJ	735 mJ (↑ 0.97%)

Table 1: Peak memory usage, execution time and energy use of chosen models.

Table 1 shows that optimised ordering was able to save 50KB of memory, compared to the order embedded in the model. Including the framework overhead (≈200KB for SwiftNet Cell, proportional to the number of tensors), this made a sufficient difference to make the model's memory footprint fit within SRAM.[3] We also check the overhead introduced by replacing a static memory allocator with a dynamic one by running MobileNet-v1-based [37] person detection model (from the Tensorflow Lite Micro repository [33]). Measurements show negligible (sub-1%) increase in execution time and energy used by the MCU and the memory footprint was decreased by 186KB. In general, latency and power consumption can be reduced with operator implementations that leverage processor capabilities well (SIMD, DSP instructions).

6 Discussion

The results show that employing a different operator execution order for neural network inference can make previously undeployable models fit within the memory constraints of MCU hardware. Reordering operators can be implemented just within the inference software, making it orthogonal to most other network compression approaches, which were likely to have been already used to create an MCU-sized model.

Unlike mobile and server platforms, MCU hardware often doesn't have enough memory to statically pre-allocate all tensor buffers of the network, which requires the inference software to support dynamic memory allocation. We showed that a simple defragmentation strategy is a viable option with little overhead cost. However, when the execution schedule is known in advance, optimal tensor buffer placement in memory may be precomputed.

Having a way of precisely computing peak memory usage for models with complex computation graphs would benefit neural architecture search (NAS) procedures. The algorithm can be extended to support various memory saving tricks: for example, if one of the inputs to the addition operator is not used elsewhere, the result can be accumulated into it, eliminating the need for an output buffer.

7 Conclusion

Microcontrollers are a viable platform for running deep learning applications if the model designer can overcome constraints imposed by limited memory and storage. In this work, we describe how to minimise peak memory usage of a neural network during inference by changing the evaluation order of its operators. By applying our methodology, we were able to achieve sufficient memory savings to deploy the chosen CNN on a microcontroller with 512KB SRAM, which would not have been possible otherwise. Our tool for embedding optimal operator ordering into TensorFlow Lite models is available at `https://github.com/oxmlsys/tflite-tools`.

Acknowledgments

The work was supported by the Engineering and Physical Sciences Research Council UK (EPSRC UK), grant ref. EP/S001530/1, and Samsung AI. The authors would also like to thank Javier Fernández-Marqués for providing help with measuring energy usage.

[3]The authors of the model note that peak memory usage can be lowered, likely by using fused operator implementations. Further savings would come from optimising the memory usage of the micro-interpreter itself.

References

[1] STMicroelectronics, "STM32 High Performance MCUs." `https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html` (Accessed Sep 2019).

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, IEEE, Jun 2016.

[5] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *31st AAAI Conference on Artificial Intelligence, AAAI 2017*, 2017.

[6] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 2018.

[7] SD Association, "Bus Speed (Default Speed/High Speed/UHS/SD Express)." `https://www.sdcard.org/developers/overview/bus_speed/index.html` (Accessed Sep 2019).

[8] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, IEEE, 2014.

[9] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on*, pp. 1–12, IEEE, 2016.

[10] C. Cai, D. Ke, Y. Xu, and K. Su, "Fast learning of deep neural networks via singular value decomposition," in *Pacific Rim International Conference on Artificial Intelligence*, pp. 820–826, Springer, 2014.

[11] L. Theis, I. Korshunova, A. Tejani, and F. Huszár, "Faster gaze prediction with dense networks and Fisher pruning," 2018.

[12] S. Han *et al.*, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding," *CoRR*, vol. abs/1510.00149, 2015.

[13] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[14] M. Alizadeh, J. Fernández-Marqués, N. D. Lane, and Y. Gal, "An empirical study of binary neural networks' optimisation," 2018.

[15] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to $+1$ or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[16] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[17] G. Georgiadis, "Accelerating Convolutional Neural Networks via Activation Map Compression," 2018.

[18] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.

[19] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018.

[20] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "MnasNet: Platform-Aware Neural Architecture Search for Mobile," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018.

[21] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," 2019.

[22] I. Fedorov, R. P. Adams, M. Mattina, and P. N. Whatmough, "SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers," pp. 1–26, 2019.

[23] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 22, IEEE Press, 2016.

[24] "tinyML Summit." `https://www.tinymlsummit.org` (Accessed Sep 2019).

[25] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes, "Visual Wake Words Dataset," 2019.

[26] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pp. 176–189, ACM, 2016.

[27] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.

[28] J. Fernández-Marqués, V. W.-S. Tseng, S. Bhattachara, and N. D. Lane, "On-the-fly deterministic binary filters for memory efficient keyword spotting applications on embedded devices," in *Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning*, pp. 13–18, ACM, 2018.

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

[30] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," 2017.

[31] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, pp. 127–138, Jan 2017.

[32] D. E. Knuth and J. L. Szwarcfiter, "A structured program to generate all topological sorting arrangements," *Information Processing Letters*, vol. 2, no. 6, pp. 153–157, 1974.

[33] TensorFlow Contributors, "TensorFlow Lite Micro." `https://github.com/tensorflow/tensorflow/commits/master/tensorflow/lite/experimental/micro` (Accessed Sep 2019), 2019.

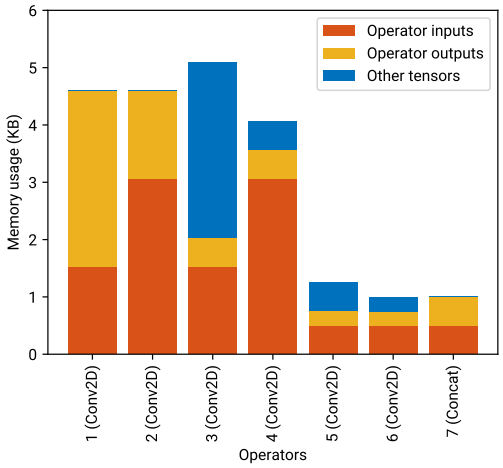
[34] H.-P. Cheng, J. Lee, P. Noorzad, and J. Lin, "QTravelers Visual Wake Words contest submission." `https://github.com/newwhitecheng/vwwc19-submission` (Accessed Sep 2019), 2019.

[35] H.-P. Cheng, T. Zhang, Y. Yang, F. Yan, S. Li, H. Teague, H. Li, and Y. Chen, "SwiftNet: Using Graph Propagation as Meta-knowledge to Search Highly Representative Neural Architectures," Jun 2019.

[36] STMicroelectronics, "NUCLEO-F767ZI development board." `https://www.st.com/en/evaluation-tools/nucleo-f767zi.html` (Accessed Sep 2019), 2019.

[37] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," apr 2017.

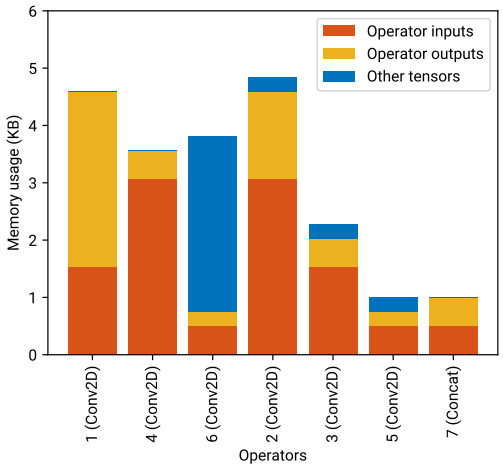
Appendix A



Operator	Tensors in RAM (output of op. #)	Usage (B)
1 (`Conv2D`)	$\{0, 1\}$	4,704
2 (`Conv2D`)	$\{1, 2\}$	4,704
3 (`Conv2D`)	$\{1, 2, 3\}$	5,216
4 (`Conv2D`)	$\{1, 3, 4\}$	4,160
5 (`Conv2D`)	$\{3, 4, 5\}$	1,280
6 (`Conv2D`)	$\{4, 5, 6\}$	1,024
7 (`Concat`)	$\{5, 6, 7\}$	1,024
	Peak:	5,216

Memory usage w/ default operator order

Figure 2: Memory usage of the sample computation graph with default operator ordering.



Operator	Tensors in RAM (output of op. #)	Usage (B)
1 (`Conv2D`)	$\{0, 1\}$	4,704
4 (`Conv2D`)	$\{1, 4\}$	3,648
6 (`Conv2D`)	$\{1, 4, 6\}$	3,904
2 (`Conv2D`)	$\{1, 2, 6\}$	4,960
3 (`Conv2D`)	$\{2, 3, 6\}$	2,336
5 (`Conv2D`)	$\{3, 5, 6\}$	1,024
7 (`Concat`)	$\{5, 6, 7\}$	1,024
	Peak:	4,960

Memory usage w/ optimised operator order

Figure 3: Memory usage of the sample computation graph with optimised operator ordering.

µNAS: CONSTRAINED NEURAL ARCHITECTURE SEARCH FOR MICROCONTROLLERS

Edgar Liberis [1] **Łukasz Dudziak** [2] **Nicholas D. Lane** [1 2]

ABSTRACT

IoT devices are powered by microcontroller units (MCUs) which are extremely resource-scarce: a typical MCU may have an underpowered processor and around 64 KB of memory and persistent storage, which is orders of magnitude fewer computational resources than is typically required for deep learning. Designing neural networks for such a platform requires an intricate balance between keeping high predictive performance (accuracy) while achieving low memory and storage usage and inference latency. This is extremely challenging to achieve manually, so in this work, we build a neural architecture search (NAS) system, called µNAS, to automate the design of such small-yet-powerful MCU-level networks. µNAS explicitly targets the three primary aspects of resource scarcity of MCUs: the size of RAM, persistent storage and processor speed. µNAS represents a significant advance in resource-efficient models, especially for "mid-tier" MCUs with memory requirements ranging from 0.5 KB to 64 KB. We show that on a variety of image classification datasets µNAS is able to (a) improve top-1 classification accuracy by up to 4.8%, or (b) reduce memory footprint by 4–13×, or (c) reduce the number of multiply-accumulate operations by ≈1700×, compared to existing MCU specialist literature and resource-efficient models. µNAS is freely available for download at *(In progress. URL to be provided after review.)*

1 INTRODUCTION

We would like to use deep learning to add computational intelligence to small personal IoT devices. Running neural networks on-device would allow a higher degree of privacy and autonomy, due to the computation happening locally and the user's data never leaving the device. However, deep learning typically requires much more computational power than is available on these devices, forcing them to rely on a remote compute server or a companion smartphone.

IoT devices are powered by microcontroller units (MCUs). MCUs are ultra-small computers with a low-frequency processor, a persistent program memory (Flash) and volatile static RAM—all contained within a single chip. They are orders of magnitude cheaper and more power-efficient compared to mobile phones and desktops, which contributes to their widespread usage: more than 30B units were estimated to have shipped in 2019 (Grand View Research, 2020). However, these benefits come at the cost of drastically reduced computational power. Here, we consider "mid-tier" IoT-sized MCUs with up to 64KB of SRAM and 64KB of persistent storage available[1] for performing neural network inference. These severe resource constraints present a challenge to running resource-intense computations.

Designing neural networks with small computational requirements is tackled by the field of model compression. Methods such as pruning, distillation, quantisation and cheaper layers, have been developed to obtain compressed models from large networks in a way that tries to minimise the amount of lost accuracy (Cheng et al., 2017). This also reveals a trade-off between the accuracy of a neural network and its resource usage, or size, suggesting it is difficult to have small models that generalise well.

However, most model compression methods are not suitable for producing MCU-sized neural networks. Many techniques target mobile devices (or similar platforms), which have orders of magnitude of more computational resources than MCUs, or focus on reducing the number of parameters (model size) or floating-point operations (FLOPs) within the network's layers while keeping the overall architecture (layer connectivity) intact, which does not fully capture all types of resource scarcity of an MCU. Even manually de-

[1]Department of Computer Science and Technology, University of Cambridge, United Kingdom [2]Samsung AI Centre Cambridge, Cambridge, United Kingdom. Correspondence to: Edgar Liberis <el398 *at* cam.ac.uk>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021.

[1]More powerful "top-tier" alternatives are also available at a higher cost, such as ARM Cortex M4- or M7-based MCUs (up to ≈400 MHz clock frequency, ≤ 512KB SRAM, ≤ 1MB Flash). Considering "mid-tier" MCUs allows our methodology to be applied more broadly both now and also in the future, once the current "mid-tier" becomes the new "low-" and the most widespread tier.

signed resource-efficient models, such as MobileNet and SqueezeNet, *would exceed the assumed resource budget by over 10×*. This necessitates further research into specialist methods for deep learning on MCUs.

To illustrate the deployment challenges, let us briefly go through design considerations of executing a neural network on an MCU and how the resource scarcity affects it:

- Temporary data generated by the neural network (activation matrices) need to fit within the SRAM of the MCU ($<$ 64 KB).
- Any static data, such as the neural network's parameters and program code, need to fit within the ROM / Flash memory of the MCU ($<$ 64 KB).
- A network needs to execute quickly to keep the inference process within reasonable power and time constraints while running on an underpowered processor.

To tackle neural network design under hard constraints, we turn to neural architecture search (NAS). NAS automatically designs neural networks and is related to model selection and hyperparameter tuning but refers specifically to selecting an architecture from a large predefined space of all neural networks, called *the search space*. During the search, many common NAS algorithms consider numerous candidate models, each assembled from a predefined set of building blocks, to find the one with the highest accuracy (Elsken et al., 2019). When properly conditioned, NAS can produce architectures within certain constraints or targeting several objectives at once, though this also increases the difficulty of the search problem.

While NAS is a promising direction to consider when designing models for MCUs, most current NAS systems target much larger platforms, such as GPUs or mobile devices. We would expect them to fail to produce MCU-compatible architectures because they were not designed to handle extreme resource scarcity of MCUs, owing at least to either an inability to represent MCU-sized architectures in the search space at all or doing so too coarsely, resulting in too few candidates to choose from during search.

Our work, µNAS, is one of the first few neural architecture search systems that target microcontrollers explicitly. µNAS accurately captures the resource requirements and, combined with model compression, successfully finds fast high-accuracy microcontroller-sized neural networks with a low memory footprint ($<$ 64 KB).

In contrast to other NAS systems for MCUs (Fedorov et al., 2019; Lin et al., 2020a), µNAS uses a standard neural network execution runtime, a fine-grained search space and accurate objective functions, which allows it to improve upon other methods on five image classification datasets using comparable MCU resource requirements. We found that µNAS can either (a) improve top-1 classification accuracy by up to 4.8%, or (b) reduce memory usage by 4–13×, or (c) reduce the number of multiply-accumulate operations by ≈1700×, depending on the task.

The main contributions of this work are:

- *We propose and motivate a multiobjective constrained NAS algorithm suitable for finding MCU-level architectures, called µNAS.* It is assembled out of:
  1. a granular search space;
  2. a set of constraints that accurately capture resource scarcity of microcontroller platforms: peak memory usage, model size and latency;
  3. a search algorithm capable of optimising for multiple objectives in the said search space;
  4. network pruning, to obtain small accurate models.
- *We perform ablation studies to quantitatively justify design decisions made in µNAS*, namely:
  1. whether using network pruning helps find smaller models than otherwise;
  2. which search algorithm should be used;
  3. how including or excluding individual objectives influences the properties of the models found.
- *We conduct extensive experiments* over five microcontroller-friendly image classification tasks and existing literature for resource-efficient neural network to demonstrate the superior performance of µNAS.

2 Related work

Until recently, due to high computational and memory requirements, neural networks have not been widely used in machine learning for IoT-sized devices. Initially, elements of gradient-based learning were combined with other machine learning algorithms to produce a resource-efficient solution. ProtoNN (Gupta et al., 2017) learn a sparse projection matrix, and use the distance between examples and learned class prototypes for classification. Bonsai (Kumar et al., 2017) learn a sparse, shallow decision tree with feature extraction occurring at a decision path. Both methods have a low memory footprint of under 2KB. More recently, manually designed neural networks with quantisation and binarisation have been used for image classification on MCUs, too, though with a larger memory footprint (Zhang et al., 2017; Mocerino & Calimera, 2019).

Neural architecture search (NAS) is a widely explored topic in deep learning for desktop and server GPUs. There are approaches based on reinforcement learning (Zoph & Le, 2016; Zhou et al., 2018; Tan et al., 2019), evolutionary algorithms (Real et al., 2019), Bayesian optimisation (Kandasamy et al., 2018b; Jin et al., 2019) and gradient optimisation (Liu et al., 2018; Mei et al., 2019), that sometimes employ weight sharing to amortise the cost of training across

multiple candidate models (Pham et al., 2018; Guo et al., 2019). Constrained multiobjective NAS has been traditionally used to find "mobile"-level networks, such as by optimising energy (Hsu et al., 2018) and latency (Fernandez-Marques et al., 2020; Cai et al., 2018) in addition to accuracy. However, few works consider MCUs as the target platform.

The closest work to ours is "*SpArSe*" (Fedorov et al., 2019), which finds both sparse and dense convolutional neural networks suitable for MCUs. In many ways, this work has served us as a promising direction to take: we share similarities with multiobjective optimisation and the use of network pruning. In comparison, we: (1) improve the key aspects of search objectives, making them more faithful to how neural networks are executed on an MCU; (2) improve on the search space, allowing for more layer connectivity; and (3) we adopt an alternative search procedure and a pruning method–all of the which allows us to produce architectures with higher accuracy and lesser resource usage.

Recently, Lin et al. (2020a) developed a neural network execution runtime for MCUs together with a NAS tailored to it (for comparison, we use an off-the-shelf runtime described in Section 3.1). To the best of our understanding, the runtime uses partial operator evaluation to store only one or more columns[2] of the output matrix of each operator. The NAS targets larger MCUs, of $>$ 256 KB SRAM and Flash, and performs the search by selecting a subnetwork from a larger model. This limits the connectivity of resulting models (*i.e.* fewer architectures are considered during search) but allows for faster convergence and targeting a more difficult task of ImageNet classification. We compare discovered models for the Speech Commands dataset: we find that µNAS finds models with a smaller memory footprint (in our range of interest) without using a custom runtime.

3 Design of µNAS

We believe the following two design requirements are essential for an MCU-level NAS and set it apart from mobile- or GPU-level NAS systems:

1. **A highly granular search space.** To discover accurate networks that fit within the strict resource requirements, NAS needs to control all aspects of a network: its layers, their size (such as output size or convolutional kernel size) and connectivity. In contrast, NAS for GPUs typically uses search spaces with a larger resolution, *e.g.* manipulating groups of layers at a time or replicating layers in predefined patterns at predetermined input resolutions. (The granularity of the search space is further discussed in Sections 3.3 and 3.5.)
2. **Accurate resource use computation.** To know which models fit within resource constraints, NAS needs to efficiently and accurately calculate or simulate how each candidate network will use memory, storage and computational resources (based on an assumed execution strategy). A precise computation is not often needed for models running on GPUs or mobiles, as these platforms are not as resource-constrained as MCUs.

In the remainder of this section, we discuss the motivation behind the requirements above and how they are implemented in µNAS. We start by describing how a neural network is executed on an MCU and, after briefly formalising the problem of multiobjective NAS, we will talk about major components in µNAS, namely the search space, objective functions, search algorithms and compression, that are all crucial for finding performant MCU-sized architectures.

3.1 Neural network execution on MCUs

A model's resource usage depends on the runtime, that is on how the underlying software chooses to run the network for inference and manage its memory usage. Knowledge of this has to be incorporated into the NAS to correctly estimate a candidate model's resource usage during the search. Here, we follow the execution strategy used by the TensorFlow Lite Micro[3] interpreter and CMSIS-NN libraries:

1. Operators (neural network layers) are executed in a predefined order, one at a time (no parallelism).
2. An operator is executed by (a) allocating/reserving memory for its output buffer in SRAM, (b) fully executing the body of an operator and (c) deallocating its input buffers (if not used elsewhere later on). Hence, the activation matrices of a neural network are only stored within SRAM.
3. Any static data, such as neural network weights, are read from the executable binary, stored in Flash.
4. The network's weights and all computation are quantised to an 8-bit fixed precision data type (`int8`), *e.g.* using the affine quantisation (Jacob et al., 2018).
5. No operators are executed partially, or more than once, and no data is written to external writeable storage (an SD-card), as this would be prohibitively slow.

It is also possible to adopt a custom execution framework that, for example, leverages partial execution, or operator fusion, and adapt µNAS for that instead. However, by using TensorFlow Lite Micro, which is widely adopted and supported by multiple chip vendors, µNAS can deliver state-of-the-art results that can be readily used, without having to maintain a custom runtime.

[2] Convolution can be implemented by an `im2col` transposition followed by matrix multiplication. Thus here, a "column" is a window of the input covered by a convolutional kernel.

[3] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/micro

3.2 What is Neural Architecture Search (NAS)?

At its core, NAS is a constrained zeroth-order optimisation problem, where we seek to find a neural network α^* (belonging to the search space $\mathcal{A}$) that maximises some objective function (or goal), called $\mathcal{L}$, such as the accuracy on the target dataset with respect to some resource usage constraints. We assume that the validation set accuracy for a particular architecture α is maximised by some weights θ^*, obtained via gradient descent. Evaluating $\mathcal{L}(\alpha)$ is expensive since it requires training a model, so we seek to find the optimum α^* in a limited number of queries to $\mathcal{L}$ (evaluations of $\mathcal{L}$).

Resource constraints can be treated as soft constraints, *i.e.* reformulated as extra objectives with penalty terms. This turns NAS into a multiobjective optimisation problem (rewritten as a minimisation problem below). We include four objectives, three of which are resource constraints (discussed in Section 3.4): (1) top-1 validation set accuracy, (2) peak memory usage, (3) model size and (4) latency.

$$\begin{aligned} \alpha^* &= \text{argmin}_{\alpha \in \mathcal{A}} \quad \mathcal{L}(\alpha) \\ &= \text{argmin}_{\alpha \in \mathcal{A}} \quad \{1.0 - \textsc{ValAccuracy}(\alpha), \\ &\qquad\qquad \textsc{ModelSize}(\alpha), \\ &\qquad\qquad \textsc{PeakMemUsage}(\alpha), \\ &\qquad\qquad \textsc{Latency}(\alpha)\} \end{aligned} \tag{1}$$

A solution to a multiobjective problem is no longer a singular value. It is common to consider a Pareto front as a set of potential solutions: a set of points on which one objective function cannot be improved without making another objective worse. In Section 3.5, we discuss a *scalarisation* approach that turns the multiobjective goal into a single objective that still encourages exploring the Pareto front.

3.3 Search space

Here, neural networks are directed acyclic graphs (DAGs), with nodes and edges representing operators (layers) and their connectivity. For convolutional neural networks (CNNs), operator options include convolution (at different kernel sizes, numbers of channels or strides), pooling, addition and matrix multiplication (fully-connected layers).

State-of-the-art GPU-level CNNs can contain a large number of operators, *e.g.* $>$ 100 layers (He et al., 2016). If architecture search were to be used to design a comparably large model while having full freedom over layer connectivity, the problem would become intractable due to the size of the search space. Instead, some NASes constrain the space to 1 or 2 small "cells" (micro-architectures) to curb the number of connectivity options (Liu et al., 2018). These cells are then replicated in a predefined pattern to produce the final architecture (macro-architecture). Alternatively, the search space can contain pruned versions of some predefined large super-architecture (Guo et al., 2019; Lin et al., 2020a).

However, NAS is known to find good performing architectures that are wired in unexpected ways (Cheng et al., 2019). So to give µNAS more freedom when searching under already tight resource requirements, we avoid imposing big structural constraints. As one would not expect to run large models on an MCU (they would violate the resource constraints), allowing more options in layer connectivity here does not make the search problem intractable.

MCU-level architectures are very sensitive to layer hyperparameters, such as the numbers of channels or units in convolutional and fully-connected layers. For example, choosing between a conv. layer with 172 and 192 channels is unlikely to make a meaningful difference for a GPU-sized model (though the former may have lower accuracy), but on an MCU choosing the larger layer may tip the model over the strict memory budget. This requires considering hyperparameters at a high granularity, which is not commonly needed for GPU-level NAS and has the negative effect of enlarging the search space.

Thus we conclude that a sensible MCU search space would consist of small models, with few restrictions on layer connectivity and highly granular hyperparameter options. The search algorithms (described in Section 3.5) navigate the search space by generating random architectures and applying changes to them (*morphisms*) to produce derivative (child) networks. Morphisms here only affect the architecture: we do not attempt to preserve and transform learned weights between the parent and the child networks.

Our search space, together with all degrees of freedom, parameter limits and morphisms, is given in Table 1. This highly granular search space comprises 1.15×10^{152} models.

3.4 Resource constraints

We focus on three constraints to be used in the search, each representing a key aspect of resource scarcity on MCUs: peak memory usage, model size and latency.

Peak memory usage. Under the execution strategy described earlier, an operator's input and output buffers have to be present in memory during its execution. Additionally, depending on the architecture of the network, there can be other buffers in memory that will be required by subsequent operators and thus cannot be deallocated yet. Overall, let us call the set of tensors that need to be present in memory at each step the *working set*.

As execution proceeds, tensors are allocated and deallocated at different times, changing the working set. To execute the model on an MCU, *the memory occupied by the working set at its peak must be lower than the amount of SRAM*.

Perhaps surprisingly, it is not straightforward to compute the peak memory usage of a network when it has branches.

Degree of freedom	Options	Morphisms
An architecture is N "convolutional" blocks, where each:	N in [1; 10]	append or remove random
• connects either in series or in parallel to the previous one	{*parallel, serial*}	change one to other
• has M convolution layers, where each layer:	M in [1; 3]	insert or remove random
• optionally, has a preceding 2×2 max-pooling operation;	{*yes, no*}	change one to other
• is either a full or a depthwise convolution with stride S, C channels $K \times K$ kernel size ($S = 1$ for 1×1 convolution and C is not configured for depthwise convolution)	{*full, d/wise*}	change types
	K in {1, 3, 5, 7}	change K by ± 2
	C in [1; 128]	change C by ± 1, 3, 5
	S in {1, 2}	change S by ± 1
• is optionally followed by batch norm.;	{*yes, no*}	change one to other
• is optionally followed by ReLU;	{*yes, no*}	change one to other
followed by a $P \times P$ average or maximum pooling,	{*avg, max*}	change one to other
	P in {2, 4, 6}	change P by ± 2
followed by F fully-connected layers, where each:	F in [1; 3]	insert or remove random
• has U units (output dimension), followed by a ReLU	U in [10; 256]	change U by ± 1, 3, 5
followed by a final fully-connected layer (U = number of classes).		

Table 1. A template for candidate models, with free variables, their morphisms and bounds, which define the search space of µNAS.

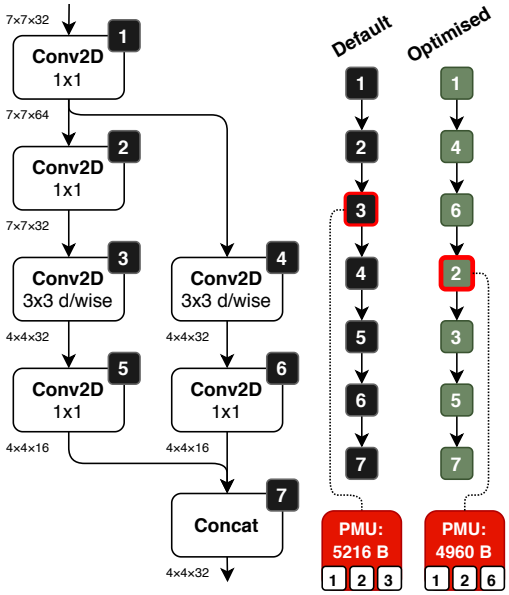


Figure 1. An example computation graph where the default and optimised execution paths yield different peak memory usage (PMU)

Branches are commonly featured in popular CNNs as residual connections (He et al., 2016), Inception modules (Szegedy et al., 2015), or NAS-designed cells (Cheng et al., 2019). Here's why: upon reaching a branching point, the inference framework has a choice which operator to execute next. Different execution orders change which tensors constitute the working set, which in turn affects the peak memory usage (see example in Figure 1). Thus, to most accurately capture the minimum amount of memory required to run a network, a NAS has to compute an execution order that gives the smallest peak working set size.

To achieve this, we build upon the algorithm by Liberis & Lane (2019), which enumerates all topological orders of a network's computational graph to find one that yields the minimal peak memory usage. We extend it to model other features, such as reusing input buffers of an 'Add' operator.

To the best of our knowledge, this is the first work to compute memory usage accurately during the search, without relying on on-device benchmarking (that is often slow) or using under-approximations. Later (Section 3.5), we will discuss that it is reasonable to assume that the most performant networks will have high resource usage. This makes a precise memory usage computation essential in identifying which candidate networks lie just below the resource limits.

Model size. All static data, such as code and parameters (weights) of a neural network, are stored in the persistent (Flash) memory of an MCU. Traditionally, each parameter is represented by a 32-bit floating-point number (a `float`), which occupies 4 bytes. However, we can reduce the per-parameter storage requirement by using quantisation.

We assume an integer-only quantisation technique used in TensorFlow Lite, which quantises each parameter from 32-bit floats to 8-bit integers after training (by calibrating quantisation parameters on the validation set). This is an excellent choice for microcontrollers, as it is: (a) byte-aligned (thus a value can be loaded directly with no decoding), (b) does not require any floating-point arithmetic units to be present on the chip, (c) is also widely adopted and (d) does not make networks suffer from accuracy loss during quantisation (confirmed by our and MCUNet (Lin et al., 2020a) experiments). If needed, µNAS can be easily adapted for other quantisation techniques.

Thus to compute the storage requirement, NAS simply counts the number of parameters of a neural network at 8 bits = 1 byte per parameter.

Latency. In order to discover models that run sufficiently quickly on an MCU, NAS has to have a notion of how long it takes to perform a single inference: *model latency*. Some NAS works have estimated latency by either using a proxy metric, such as the number of floating-point operations (FLOPs) required to complete a forward pass (Mei et al., 2019; Xie et al., 2018) or predicting the latency via a surrogate model. The latter can be either a sum of latency predictions of each layer or a prediction for a whole model which takes into account inter-layer interactions (Chau et al., 2020; Cai et al., 2018).

Using a predictive model has become the dominant approach for GPU-/mobile-level NAS (Tan et al., 2019; Chau et al., 2020), as proxy metrics, such as FLOPs, fail to account for scheduling, caching, parallelism and other properties of the inference software or hardware. However, MCUs typically lack these performance-enhancing features: the software runs on a single-core processor at a fixed frequency with no data caching, which makes modelling the runtime simpler.

We settle on using a number of multiply-accumulate operations (MACs) as a proxy for model latency. To verify that this estimator approximates actual model latency well, in Figure 2, we plot the measured runtime of a 1000 random models from our search space and versus the number of MAC operations. For reference, we reproduce the figure from BRP-NAS (Chau et al., 2020) to show how the number of FLOPs compares to latency on a desktop GPU. We observe that MACs are *not* systematically under- or over-approximating latency on an MCU, and the result has an $R^2 = 0.975$ goodness-of-fit.

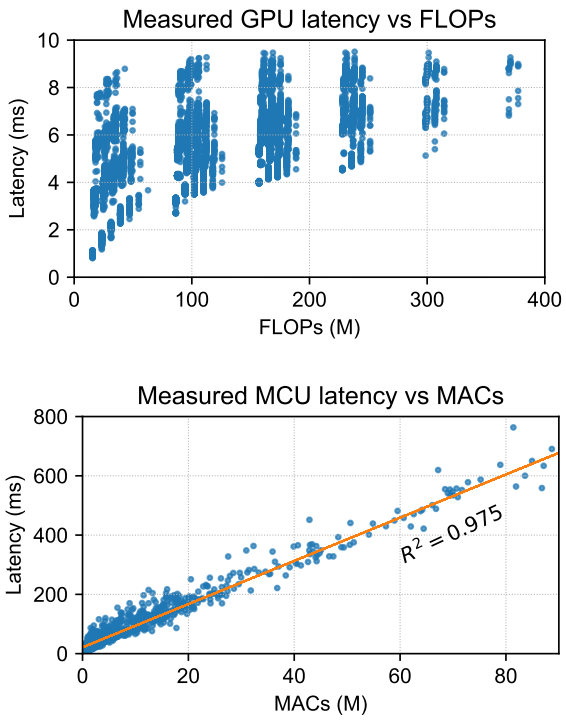


Figure 2. (Top) A typical plot of FLOPs vs model latency on a desktop GPU, reproduced with permission from Chau et al. (2020). *(Bottom)* Our measured MACs vs model latency on a sample of 1000 models from our search space. The models ran on a NUCLEO-H743ZI2[5] MCU board using the TensorFlow Lite Micro runtime. The values were averaged over ten runs. **The data shows that MACs are a good predictor of model latency on MCUs.**

3.5 Search algorithms

Intuitively, because performant neural networks are large, we would expect the best architectures to make use of all available resources and thus be located at a boundary between models that fit within the resource constraints and ones that do not. By design of the search space (both ours and others), a change in a single axis (degree of freedom) can significantly change layer properties and interconnectivity, thus potentially drastically altering the model's resource requirements and accuracy. This makes the boundary jagged, rendering multiobjective NAS a challenging task for black-box optimisation methods.

Now, we establish a way to handle multiple objectives in a single goal and discuss two optimisation algorithms implemented in µNAS: aging evolution (AE) and Bayesian optimisation (BO). Despite being inherently sequential, both AE and BO allow for parallel point evaluation (Real et al., 2019; Kandasamy et al., 2018a), allowing us to scale the search over multiple GPUs. We compare the two in our experiments to discover the best performing approach.

Handling multiple objectives. Akin to *SpArSe*, we use random scalarisations (Paria et al., 2020, TS expression) to combine multiple objectives into a single goal (the optimisation target function).

$$\mathcal{L}^t(\alpha) = \max \begin{cases} \lambda_1^t \, (1.0 - \textsc{ValAccuracy}(\alpha)), \\ \lambda_2^t \, \textsc{PeakMemUsage}(\alpha), \\ \lambda_3^t \, \textsc{ModelSize}(\alpha), \\ \lambda_4^t \, \textsc{MACs}(\alpha) \end{cases} \tag{2}$$

Each objective has an associated scalar term λ_i^t, which specifies its relative importance in the current search goal. To encourage the exploration of the Pareto front, the goal changes at every search round (indexed by t) by resampling coefficients $\boldsymbol{\lambda}^t$. The trade-off preferences between multiple

[5] https://www.st.com/en/evaluation-tools/nucleo-h743zi.html

objectives can be encoded in the distribution for $\boldsymbol{\lambda}$: we use $1/\lambda_i \sim \text{Uniform}[0; b]$, where b is a user-specified soft upper bound for the i^{th} objective.

NAS via local search. Local search optimises the goal function by repeatedly evolving a set of candidate points. We use aging evolution (AE) as a local search algorithm for NAS (Real et al., 2019). AE operates by keeping a population of P architectures and, at each search round, subsampling the population to get S architectures to choose the one that gives the smallest value of $\mathcal{L}^t(\alpha)$. A random morphism is then applied to this winning architecture to produce an offspring, which is then evaluated and added to the population, replacing the oldest architecture. Aging evolution has proved itself a competitive search algorithm for NAS, beating many baselines and random search.

NAS via Bayesian optimisation. Bayesian optimisation (BO) uses surrogate models to approximate the target function and guide the search towards promising unexplored regions of its domain. BO is successfully used in hyperparameter optimisation (Falkner et al., 2018) and NAS (Jin et al., 2019), often using Gaussian Processes (GPs) as a surrogate model for the accuracy of a neural network.

In µNAS, at each step of the search, BO decides which architecture to evaluate next by optimising the target function ($\mathcal{L}^t$), with the only difference that $\text{VALACCURACY}(\theta)$ is computed by sampling from the GP that approximates it (other objectives are cheap to compute and thus do not require a surrogate model, unlike in *SpArSe*). GPs require a kernel that captures the notion of similarity between any two neural networks. For this, we build upon NASBOT (Kandasamy et al., 2018b), which uses optimal transport to define a distance function between two computational graphs of neural networks; we update the kernel to include free variables of our search space.

3.6 Model compression

Model compression methods, such as pruning, quantisation, cheaper building blocks and distillation attempt to create or train a smaller model using a bigger more performant model, while retaining as much of the predictive performance as possible. Parameter-efficient convolutional blocks, such as inverted residual blocks of MobileNet V2 (Sandler et al., 2018), and fire modules of SqueezeNet (Iandola et al., 2016), are already representable in the search space, allowing µNAS to recover them if necessary.

µNAS supports using model compression during the search: in particular, we hypothesise that network pruning would help in finding small architectures with high accuracy—we will empirically determine this in our experiments.

Pruning. Pruning removes individual parameters from a neural network that do not significantly affect generalisation. We use structured pruning, which eliminates entire groups of parameters: channels (in conv. layers) or units/neurons (in fully-connected layers), resulting in smaller dense models, as opposed to sparsifying the weight matrices. This makes the search and the pruning share the task of determining the model's hyperparameters: the search produces a base network, which is then adjusted by pruning in a more informed way by discarding channels/units that were deemed unimportant during training.

µNAS employs DPF pruning (Lin et al., 2020b) which uses the L_2 norm of channels/units to discard weight groups until the desired proportion of groups is removed. µNAS sets this target "sparsity" proportion and the network is gradually pruned during training until the target proportion is reached.

3.7 Summary of the µNAS search procedure

As an example, let us walk through the search procedure when µNAS is used with aging evolution (AE) and pruning:

1. **AE initialisation.** A random initial population of networks is generated and trained, together with their target "sparsity" values (random in the allowed range).
2. **Preamble.** A new search step t begins by generating a new goal function. This is done by resampling coefficients λ in Equation 2.
3. **Search algorithm body.** The search proceeds by the rules of AE: the next architecture to be trained is determined by applying a random morphism (Table 1) to the chosen parent. The target "sparsity" level for the new model is inherited and randomly perturbed.
4. **Write back.** Once the training and pruning are complete, the values of all four objectives of the pruned network (the final accuracy, peak memory usage, storage usage and the number of MACs) are recorded, and the network is added to the population. A new round starts from step 2.

4 EVALUATION

The aim of the evaluation is twofold:

1. **To show that all components of µNAS are required for obtaining small, performant models.** The design of µNAS is empirically validated through ablation studies. Namely, we address: (a) whether aging evolution or Bayesian optimisation perform better at discovering the Pareto front; (b) whether including pruning yields more performant small models; (c) we check the utility of including both model size and peak memory usage objectives by searching without either.
2. **To show that µNAS produces superior models**, as compared to previous work on CNNs for MCUs.

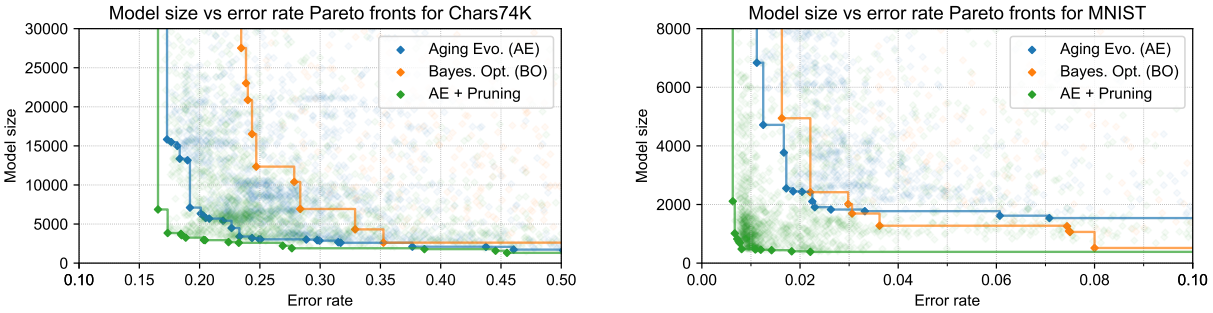


Figure 3. (Left) error rate vs size of models discovered by µNAS on the Chars74K dataset; *(Right)* ditto for the MNIST dataset. **The results show that µNAS configured with aging evolution and pruning finds the furthest advanced Pareto front.**

4.1 Datasets

We evaluate µNAS on five image classification problems: MNIST (LeCun & Cortes, 2010), CIFAR-10 (Krizhevsky, 2009), Chars74K (De Campos et al., 2009) (English subset: 26 uppercase and 26 lowercase letters + 10 digits), Fashion MNIST (Xiao et al., 2017) and Speech Commands (Warden, 2018). Datasets are split into training, validation and test sets, with the validation set used for evaluating objectives and tuning hyperparameters. Accuracy results are reported on the unseen test set.

To compare with related work, "*binary*" versions of Chars74K and CIFAR-10 are included, which have images partitioned into two subclasses. All details on experiment configuration, dataset preprocessing and optimiser parameters are given in Appendix A. As far as we aware, the full Chars74K or Fashion MNIST datasets do not have strong low resource usage baselines to compare against. Except for CIFAR-10, µNAS was run for 2000 steps, taking 2-10 GPU days depending on the dataset. We set resource requirements to either less than 64 KB of peak memory usage and storage, corresponding to our target MCUs, or less than 2KB, when required for comparison. We also perform experiments on searching for sparse models (Appendix B).

4.2 Determining the best µNAS configuration

µNAS can use both AE and BO as a search algorithm and optionally use pruning during the search. We would like to determine which implemented configuration yields the best results for finding models with low resource usage and should therefore be used for further experiments. To do so, we run µNAS in three configurations on Chars74K and MNIST datasets: (1) plain aging evolution (AE); (2) plain Bayesian optimisation (BO); (3) AE with pruning. Figure 3 shows the model size *vs* error rate ($1.0 -$ accuracy) of discovered models for each configuration. The lower-left corner of the plot contains models with low error and low resource usage; the further the Pareto front (highlighted in the plots) extends into that corner area, the better size vs accuracy trade-off µNAS is able to discover.

Search algorithm comparison shows that AE discovers a further advanced Pareto front than BO. We hypothesise that BO performs worse due to a smooth approximation to a network's accuracy (GPs) being inadequate to capture the subtle differences between networks in the search space, especially when few points are available to build the surrogate in the initial stages of the search.

We also observed that BO completes fewer search rounds compared to AE in the same allotted time. This is due to an added computational burden of (a) computing resource usage of many models when optimising an acquisition function, (b) updating the posterior model and (c) computing the kernel function after each new model has been trained.

The data shows that pruning is essential for finding models with low resource usage and AE with pruning outperforms other configurations implemented in µNAS. Therefore, AE with pruning (the search procedure summarised earlier in Section 3.7) is the strongest and default design choice and will be used in further experiments.

4.3 The utility of resource objectives

Having developed precise resource usage computations, we would like to check whether including them actually provides meaningful input to the search by guiding it towards low resource usage models.

Here, we will observe how model size (MS) and peak memory usage (PMU) objectives influence the search. We run µNAS on MNIST in three modes: all constraints present, no model size (MS) constraint, and no peak memory usage (PMU) constraint. We expect these three regimes to exhibit a difference in the kinds of models found by µNAS: the PMU objective should drive the search towards narrow models (a lower memory bottleneck), MS—towards shallow models (fewer layers); when both are present, a trade-off between

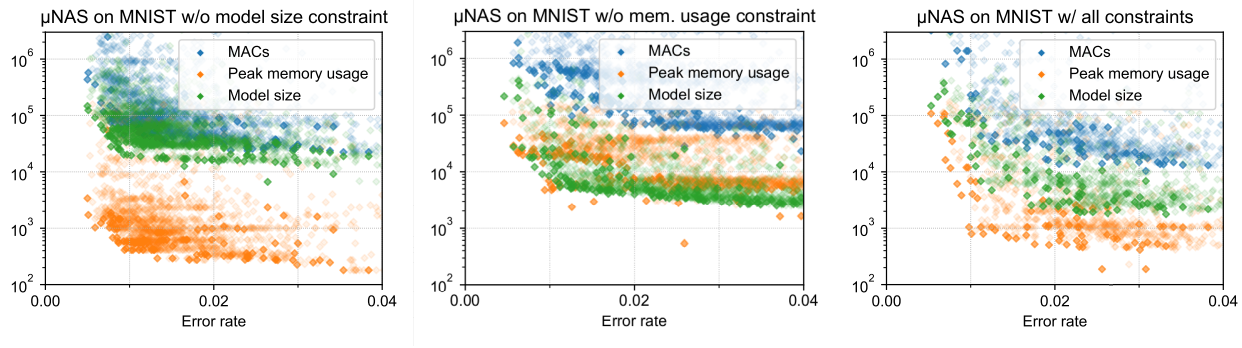


Figure 4. Error rate vs resource usage of MNIST models when *(left)* model size was unconstrained, *(middle)* peak memory usage was unconstrained, *(right)* all constraints were present. Each architecture produces 3 data points: one for each metric (colour). **The plots show that both peak memory usage and model size objectives are influential and informative for the search.**

low resource usage and accuracy should be discovered.

Found models are presented in Figure 4. We do not use pruning and set a generous latency constraint to avoid introducing any additional bias to the results. The data supports the hypothesis above, showing that both objectives are needed for the search: (a) when MS is unconstrained, the search is dominated by the PMU constraint and driven to models with low memory usage; this also implies high accuracy, since it is possible to generate deep and narrow networks with high model size but a low memory bottleneck; (b) when PMU is unconstrained, the search is driven to models with low MS and average PMU; (c) when both objectives are present, µNAS discovers a variety of trade-off points between accuracy and resource usage, with models that have low to moderate PMU and MS.

4.4 Discovered architectures

Table 2 shows networks discovered by µNAS. For each task, we present a baseline model at multiple objective trade-off points, together with a model discovered by µNAS that either improves upon or closely match on all (known) metrics.

The data shows that µNAS can discover truly tiny models, advancing the state-of-the-art for MCU deep learning for 6 out of 7 tasks considered. The models are trainable from scratch with pruning, and are within our assumed constraints of "mid-tier" MCUs with a 64 KB memory and storage limit.

The results show: (a) an improved top-1 classification accuracy by up to 4.8% for the same memory and/or model size footprint (see MNIST; CIFAR-10 (bin.), Chars74K (bin.); Speech Commands), or (b) a reduced memory footprint by 4–13× while preserving accuracy (see MNIST, Speech Commands, Fashion MNIST), or (c) a reduced number of MACs by ≈1700× (see Speech Commands).

Targeting Speech Commands in particular shows the importance of considering MACs during the search: even when model size and accuracy are comparable with the baselines, there is scope for discovering models that are orders of magnitude faster (in MACs). For MNIST, the search found a pruned (yet still dense) model with < 0.5 KB parameters and $> 99\%$ accuracy. Earlier, Figure 3 showed that µNAS discovered many models in this area of the accuracy-size trade-off for MNIST, confirming that this is not an outlier.

For CIFAR-10, µNAS is able to outperform the baseline for the binary dataset; however, it falls short compared to the models discovered by LEMONADE (Elsken et al., 2018) for a full (10-class) dataset. Upon closer investigation, we find that the µNAS search does gradually push the Pareto front, but requires many steps to do so (the search took 6000 steps, ≈30 GPU-days). We discuss this effect in the following section and predict that if µNAS is given more search time, it can discover better models for CIFAR-10. After all, it is not uncommon to run aging evolution for tens of thousands of steps (Real et al., 2019).

5 Discussion

We have both qualitatively and quantitatively justified the design decisions made in µNAS, showing that it is able to find accurate yet small models, suitable for deployment on microcontrollers. We now discuss overarching points relating to µNAS and how it can be modified in future work.

5.1 Convergence of aging evolution

CIFAR-10 experiments showed that the search is slow at advancing the Pareto front. We found that this is due to the search space being too granular: while considering hyperparameters and morphisms at a high granularity is instrumental for finding tiny models (see, for example, MNIST and other results), it also makes the search less efficient by requiring multiple steps to change any candidate model in the popula-

Dataset	Model	Accuracy (%)	Model size	RAM usage	MACs	Difference
MNIST	SpArSe (Fedorov et al., 2019)	98.64	2770	$\geq$ 1960 B	*unk.*	Size $\uparrow$ 5.7×
	SpArSe	96.49	1440	$\geq$ 1330 B	*unk.*	Acc. $\downarrow$ 2.7%
	BonsaiOpt (Kumar et al., 2017)	94.38	490	$<$ 2000 B	*unk.*	Acc. $\downarrow$ 4.8%
	ProtoNN (Gupta et al., 2017)	95.88	63'900	$<$ 64'000 B	*unk.*	Acc. $\downarrow$ 3.3%
	µNAS (ours)	**99.19**	**480**	**488 B**	28.6 K	
CIFAR-10 (binary)	SpArSe	73.84	780	$\geq$ 1280 B	*unk.*	Acc. $\downarrow$ 3.7%
	µNAS (ours)	**77.49**	**685**	**909 B**	41.2 K	
Chars74K	SpArSe	77.78	460	$\geq$ **720 B**	*unk.*	Acc. $\downarrow$ 3.4%
	µNAS (ours)	**81.20**	**390**	867 B	107 K	
Speech Commands	RENA (Zhou et al., 2018)	**94.04**	47 K	*unk.*	$\approx$700 M	Size. $\uparrow$ 1.2×
	µNAS (ours)	**94.11**	**41 K**	21.6 KB	**1.5 M**	
	DS-CNN (Zhang et al., 2017)	**93.39**	23 K	*unk.*	$\approx$3'035 M	MACs $\uparrow$ 1785×
	µNAS (ours)	92.31	**21 K**	52.8 KB	**1.7 M**	
	RENA	**94.82**	67 K	*unk.*	$\approx$3'265 M	Size. $\uparrow$ 1.2×
	µNAS (ours)	**95.26**	**56 K**	11.5 KB	**3.6 M**	
	MCUNet (Lin et al., 2020a)	$\approx$91.20	$<$ 1 M	80 KB	*unk.*	Acc. $\downarrow$ 4.4%
	MCUNet	$\approx$**95.91**	$<$ 1 M	311 KB	*unk.*	RAM $\uparrow$ 13×
	µNAS (ours)	**95.60**	**95 K**	**23.8 KB**	6.8 M	
Fashion MNIST	reported[6]	92.50	$\approx$100 K	*unk.*	*unk.*	Size. $\uparrow$ 1.6×
	µNAS (ours)	**93.22**	**63.6 K**	12.6 KB	4.4 M	
CIFAR-10	LEMONADE (Elsken et al., 2018)	$\approx$**91.77**	**10 K**	*unk.*	*unk.*	
	µNAS (ours)	86.79	17.4 K	15.8 KB	401 K	Acc. $\downarrow$ 5%
Chars74K	*µNAS (ours)*	82.65	3.85 K	9.75 KB	279 K	
	µNAS (ours)	76.05	13.9 K	1.81 KB	111 K	

Table 2. Pareto-optimal architectures discovered by µNAS vs others. The difference column compares a baseline to a model discovered by µNAS in the group (the last line of each group). "*Unknown (unk.)*" denotes data not reported by the authors. **The results show that µNAS outperforms most baselines by either improving accuracy for comparable resource usage or vice versa.**

tion significantly. Slow convergence is also likely to occur for large ImageNet-based image classification tasks, such as Visual Wake Words (Chowdhery et al., 2019), exacerbated by the fact that candidate models would also take longer to train (up to an hour) than on CIFAR-10. Thus, while using an evolutionary local search algorithm (like AE) on this search space yields small and performant models, the search can be time-consuming.

If search time is an issue, we envision the above being rectified by: (a) not using the same search space throughout the entire search process, for example, by using a parametrised space where granularity can vary throughout the search; or (b) guiding the search, such as by using ranking models (Chau et al., 2020); or (c) employing weight sharing to amortise the cost of training each candidate network.

5.2 Reusability of µNAS components

We demonstrated that satisfying two previously identified design requirements (Section 3)—having a granular search space and an accurate resource usage computation—leads to a NAS that can discover performant MCU-sized networks.

Overall, µNAS is modular: the search algorithm, objective scalarisation, the search space, the computation of resource usage and model compression (pruning) are all independent of each other and can be reused elsewhere or swapped out, for example, if a different network execution strategy is assumed or should a different search algorithm be needed. We make the source code of µNAS available publicly.[7]

6 Conclusions

Neural architecture search is a powerful tool for automating model design, especially when designing networks manually is challenging due to the need to balance high accuracy and fitting within extremely tight resource constraints. We showed that through the suitable design of the search space, search algorithm and explicit targeting of the three primary resource bottlenecks, we are able to create a NAS system, µNAS, that discovers resource-efficient microcontroller-friendly models for a variety of image classification tasks.

[6] https://github.com/zalandoresearch/fashion-mnist

[7] *(In progress. URL to be provided after review.)*

REFERENCES

Cai, H., Zhu, L., and Han, S. ProxylessNAS: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

Chau, T., Dudziak, Ł., Abdelfattah, M. S., Lee, R., Kim, H., and Lane, N. D. BRP-NAS: Prediction-based NAS using GCNs. *arXiv preprint arXiv:2007.08668*, 2020.

Cheng, H.-P., Zhang, T., Yang, Y., Yan, F., Li, S., Teague, H., Li, H., and Chen, Y. SwiftNet: Using graph propagation as meta-knowledge to search highly representative neural architectures. *arXiv preprint arXiv:1906.08305*, 2019.

Cheng, Y., Wang, D., Zhou, P., and Zhang, T. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

Chowdhery, A., Warden, P., Shlens, J., Howard, A., and Rhodes, R. Visual wake words dataset. *arXiv preprint arXiv:1906.05721*, 2019.

De Campos, T. E., Babu, B. R., Varma, M., et al. Character recognition in natural images. *VISAPP (2)*, 7, 2009.

Elsken, T., Metzen, J. H., and Hutter, F. Efficient multi-objective neural architecture search via Lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

Falkner, S., Klein, A., and Hutter, F. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

Fedorov, I., Adams, R. P., Mattina, M., and Whatmough, P. SpArSe: Sparse architecture search for CNNs on resource-constrained microcontrollers. In *Advances in Neural Information Processing Systems*, pp. 4977–4989, 2019.

Fernandez-Marques, J., Whatmough, P. N., Mundy, A., and Mattina, M. Searching for Winograd-aware quantized networks. *arXiv preprint arXiv:2002.10711*, 2020.

Grand View Research. Microcontroller Market Size, Share & Trends Analysis Report By Product (8-bit, 16-bit, 32-bit), By Application (Automotive, Consumer Electronics, Industrial, Medical Devices, Military & Defense), And Segment Forecasts, 2020–2027. https://www.grandviewresearch.com/industry-analysis/microcontroller-market (Accessed Aug 2020), 2020.

Guo, Z., Zhang, X., Mu, H., Heng, W., Liu, Z., Wei, Y., and Sun, J. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*, 2019.

Gupta, C., Suggala, A. S., Goyal, A., Simhadri, H. V., Paranjape, B., Kumar, A., Goyal, S., Udupa, R., Varma, M., and Jain, P. ProtoNN: Compressed and accurate knn for resource-scarce devices. In *International Conference on Machine Learning*, pp. 1331–1340, 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hsu, C.-H., Chang, S.-H., Liang, J.-H., Chou, H.-P., Liu, C.-H., Chang, S.-C., Pan, J.-Y., Chen, Y.-T., Wei, W., and Juan, D.-C. MONAS: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332*, 2018.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$ 0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

Jin, H., Song, Q., and Hu, X. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1946–1956, 2019.

Kandasamy, K., Krishnamurthy, A., Schneider, J., and Póczos, B. Parallelised Bayesian optimisation via Thompson sampling. In *International Conference on Artificial Intelligence and Statistics*, pp. 133–142, 2018a.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. P. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in neural information processing systems*, pp. 2016–2025, 2018b.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, 2009.

Kumar, A., Goyal, S., and Varma, M. Resource-efficient machine learning in 2 kb ram for the internet of things. In *International Conference on Machine Learning*, pp. 1935–1944, 2017.

LeCun, Y. and Cortes, C. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

Liberis, E. and Lane, N. D. Neural networks on microcontrollers: saving memory at inference via operator reordering. *arXiv preprint arXiv:1910.05110*, 2019.

Lin, J., Chen, W.-M., Lin, Y., Cohn, J., Gan, C., and Han, S. Mcunet: Tiny deep learning on iot devices. *arXiv preprint arXiv:2007.10319*, 2020a.

Lin, T., Stich, S. U., Barba, L., Dmitriev, D., and Jaggi, M. Dynamic model pruning with feedback. *arXiv preprint arXiv:2006.07253*, 2020b.

Liu, H., Simonyan, K., and Yang, Y. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

Mei, J., Li, Y., Lian, X., Jin, X., Yang, L., Yuille, A., and Yang, J. AtomNas: Fine-grained end-to-end neural architecture search. *arXiv preprint arXiv:1912.09640*, 2019.

Mocerino, L. and Calimera, A. CoopNet: Cooperative convolutional neural network for low-power MCUs. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 414–417. IEEE, 2019.

Paria, B., Kandasamy, K., and Póczos, B. A flexible framework for multi-objective bayesian optimization using random scalarizations. In *Uncertainty in Artificial Intelligence*, pp. 766–776. PMLR, 2020.

Pham, H., Guan, M., Zoph, B., Le, Q., and Dean, J. Efficient neural architecture search via parameters sharing. volume 80 of *Proceedings of Machine Learning Research (PMLR)*, pp. 4095–4104, 2018.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pp. 4780–4789, 2019.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. MobileNet V2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. MNasNet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

Warden, P. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Xie, S., Zheng, H., Liu, C., and Lin, L. SNAS: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

Zhang, Y., Suda, N., Lai, L., and Chandra, V. Hello Edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*, 2017.

Zhou, Y., Ebrahimi, S., Arık, S. Ö., Yu, H., Liu, H., and Diamos, G. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912*, 2018.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

A EXPERIMENT CONFIGURATION

A.1 Resource bound configurations

In µNAS, we control which area of the Pareto front gets explored by specifying a preferred trade-off between objectives. The objective scalarisation (Equation 2) together with coefficient sampling force the search to highly penalise any objectives that exceed their specified bound, and not preferentially penalise any objective that is within its bounds. We give our requested bounds in Table 3.

Requested objective bounds			
Error	**Peak memory**	**Model size**	**MACs**
MNIST			
< 0.0350	< 2.5 KB	< 4.5 K	< 30 M
Chars74K			
< 0.3000	< 10 KB	< 20 K	< 1 M
CIFAR-10			
< 0.1800	< 75 KB	< 75 K	< 30 M
Speech Commands			
< 0.0850	< 60 KB	< 40 K	< 20 M
Fashion MNIST			
< 0.1000	< 64 KB	< 64 K	< 30 M

Table 3. Resource bounds requested from µNAS for each dataset.

A.2 Dataset preprocessing and model training schedule

Dataset preprocessing, as well as model training and pruning information, is given in Table 5 (see next page).

Additional notes: Models CIFAR-10 and Fashion MNIST also have a Dropout layer w/ rate = 0.15 inserted before every fully-connected layer, except the last one. The models have been quantised for comparison against baselines in Table 2 after training and, like Lin et al. (2020a), we observe no loss in accuracy. This is not surprising due to the full-integer quantisation in TensorFlow Lite (Jacob et al., 2018) being very expressive: at a cost of extra operations, it allows to represent any finite range in 256 steps (for 8-bit quantisation), and weight decay used during encourages model weights to stay within reasonable bounds.

B SPARSE MODELS

Sparse neural networks allow for an even more significant reduction of model size without sacrificing accuracy. To compare our models with the *SpArSe* NAS (Fedorov et al., 2019), we also allow µNAS to perform unstructured pruning using the same pruning method described in Section 3.6, which results in models with sparse weight matrices. As opposed to dense operations which had been assumed so far, sparse models are executed using sparse matrix multiplications. Thus previously considered latency and peak memory usage constraints would no longer be accurate and are not included.

Dataset	**Model**	**Accuracy (%)**	**Size**
Chars74K	SpArSe	67.82	1.7 K
	µNAS (ours)	**72.40**	1.62 K
CIFAR-10 (binary)	SpArSe	81.77	3.2 K
	µNAS (ours)	**82.69**	**2.23 K**
	SpArSe	76.66	1.4 K
	µNAS (ours)	76.72	**0.94 K**
MNIST	SpArSe	99.16	1 K
	µNAS (ours)	99.19 (dense)	480

Table 4. Sparse models found by µNAS vs others. **Results show an increased accuracy delivered by µNAS for a comparable model size, or a smaller model size for comparable accuracy.**

µNAS search results for sparse models are presented in Table 4. Results show an increased accuracy delivered by µNAS for a comparable model size (up to 4.6%), or a smaller model size (up to two-fold) for comparable accuracy, depending on the task.

[7] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/speech_commands/input_data.py

Dataset	**Data preprocessing and augmentation**	**Training and optimiser configuration**	**Pruning configuration**
MNIST	• random rotate by +/- 0.2 rad with p = 0.3; • random shift (2, 2); • random flip L/R	SGDW: • learning rate = 0.005, • momentum = 0.9, • weight decay = 4e-5; • epochs = 30; • batch size = 128.	• target sparsity in [0.05; 0.80]; • pruning between epochs 3 and 18;
Chars74K	• image size 48x48 (32x32 for binary) • random split into 5000, 705, 2000 • images for train, val. & test sets; • random shift by $\pm$ 10% of H/W	SGDW: • learning rate = 0.01, 0.005 from epoch 35, • momentum = 0.9, • weight decay = 1e-4, • epochs = 60, • batch size = 80.	• target sparsity in [0.1; 0.85]; • pruning between epochs 20 and 5
CIFAR-10	• normalisation; • random flip L/R; • random shift (4, 4);	SGDW: • learning rate = 0.01, 0.005 from epoch 35, 0.001 from epoch 65, • momentum = 0.9, • weight decay = 1e-5, • batch size = 128, • epochs = 80.	• target sparsity in [0.1; 0.9]; • pruning between epochs 30 and 60;
Speech Commands	as given here[8]	AdamW: • learning rate = 0.0005, 0.0001 from epoch 20, 2e-5 from epoch 40; • weight decay = 1e-5; • batch size = 50; • epochs = 45.	• target sparsity in [0.1; 0.85]; • pruning between epochs 20 and 40
Fashion MNIST	• random rotate by +/- 0.2 rad with p = 0.3; • random shift (2, 2); • random flip L/R	SGDW: • learning rate = 0.01, 0.005 from epoch 20, 0.001 from epoch 35; • momentum = 0.9, • weight decay = 1e-5, • epochs = 45; • batch size = 128.	• target sparsity in [0.05; 0.90]; • pruning between epochs 3 and 38

Table 5. Dataset preprocessing, model training and pruning configurations used in our experiments.