µNAS: Constrained Neural Architecture Search for Microcontrollers

Edgar Liberis
el398@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Łukasz Dudziak
Samsung AI Centre Cambridge
Cambridge, United Kingdom

Nicholas D. Lane
University of Cambridge
Samsung AI Centre Cambridge
Cambridge, United Kingdom

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. 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 at least 2×, compared to existing MCU specialist literature and resource-efficient models. µNAS is freely available for download at https://github.com/eliberis/uNAS

CCS Concepts: • **Computer systems organization** → *Embedded software*; • **Computing methodologies** → **Neural networks**.

Keywords: microcontrollers, tinyml, NAS

ACM Reference Format:
Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. 2021. µNAS: Constrained Neural Architecture Search for Microcontrollers. In *The 1st Workshop on Machine Learning and Systems (EuroMLSys '21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3437984.3458836

EuroMLSys '21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8298-4/21/04…$15.00
https://doi.org/10.1145/3437984.3458836

1 Introduction

We would like to use deep learning to add computational intelligence to small personal IoT devices. IoT devices are powered by microcontroller units (MCUs). MCUs are small chips containing a low-frequency processor, a persistent program memory (Flash) and volatile static RAM. They are cheap, but have 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] These severe constraints are an obstacle to running neural network inference, forcing applications to rely on a remote compute server or a smartphone.

Designing neural networks with small computational requirements is tackled by the field of model compression. Methods such as pruning and quantisation compress large networks in a way that minimises the amount of lost classification accuracy [5]. This also reveals a trade-off between a model's accuracy and its resource usage/size, suggesting it is difficult to create small models that generalise well.

However, most compression methods are not suitable for producing MCU-sized networks. Many techniques target platforms which have orders of magnitude of more computational resources than MCUs (e.g. mobile devices), 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 designed resource-efficient models, such as MobileNet and SqueezeNet, *would exceed the assumed resource budget by over 10*×. This necessitates research into specialist methods for deep learning on MCUs.

Here's how resource scarcity affects the execution of a neural network on an MCU, complicating deployment:

- Temporary data generated by the network (activations) 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.

[1] More powerful alternatives are also available at a higher cost, such as ARM Cortex M4/7-based MCUs, however, considering "mid-tier" allows our methodology to be applied more broadly.

To tackle neural network design under hard constraints, we turn to neural architecture search (NAS), which automatically finds well-performing neural networks. When properly conditioned, NAS can produce architectures within certain constraints or targeting several objectives at once.

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, 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 MCUs explicitly. µNAS accurately captures the resource requirements and, combined with model compression, finds fast high-accuracy MCU-sized networks with a low memory footprint (< 64 KB).

In contrast to other NAS systems for MCUs [9, 22], 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 at least 2×, 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;
  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 an ablation study to quantitatively justify the inclusion of network pruning in µNAS.*
- *We conduct extensive experiments* over five microcontroller-friendly image classification tasks to demonstrate the superior performance of µNAS.

2 Related work

Until recently, due to high computational requirements, neural networks have not been widely used on IoT-sized devices. Initially, elements of gradient-based learning were combined with other machine learning algorithms [12, 19] to produce ultra-low memory classifiers (< 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 [26, 37].

Neural architecture search (NAS) is a widely explored topic in deep learning for GPUs. There are approaches based on reinforcement learning [32, 38, 39], evolutionary algorithms [29], Bayesian optimisation [16, 17] and gradient optimisation [24, 25], often employing weight sharing to amortise the cost of training multiple models [11, 28]. Multi-objective NAS has been used to find "mobile"-level networks, *e.g.* by optimising energy [14] and latency [2, 10] in addition to accuracy. Few works consider MCUs as the target.

The closest work to ours is "*SpArSe*" [9], which finds both sparse and dense convolutional neural networks for MCUs. In comparison, we: (1) make search objectives more faithful to how neural networks are executed on an MCU; (2) improve layer connectivity in the search space; and (3) adopt an alternative search procedure and a pruning method—all of the which allows us to produce superior architectures.

Recently, Lin et al. [22] 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). 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 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: µNAS finds models with a smaller memory footprint.

3 Design of µNAS

3.1 Neural network execution on MCUs

A model's resource usage depends on the runtime, that is on how the software chooses to run the network for inference and manage memory. Here, we follow the execution strategy imposed by the TensorFlow Lite Micro [6] interpreter:

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 memory for its output buffer in SRAM, (b) fully executing the 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`).
5. No operators are executed partially, or more than once.

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 (from the search space) 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. Evaluating $\mathcal{L}(\alpha)$ is expensive since it requires training a model, so we seek to limit the number of queries to $\mathcal{L}$ (evaluations of $\mathcal{L}$).

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

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×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×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).		

Resource constraints can be treated as extra objectives with penalty terms. We include four objectives, three of which are resource constraints: (1) *top-1 validation set accuracy*, (2) *peak memory usage*, (3) *model size* and (4) *latency*.

A solution to a multiobjective problem may no longer be a singular value. It is common to consider a Pareto front of potential solutions: points for which one objective function cannot be improved without making another worse.

3.3 Search space

First design requirement unique to MCU-level NAS is **a highly granular search space**. MCU-sized 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 high granularity is not commonly needed for other platforms and would needlessly enlarge the search space.

The granularity also extends to layer connectivity. GPU-level NASes often constrain the space to 1 or 2 small "cells" (micro-architectures), which are then replicated in a predefined pattern. This is done to curb the number of connectivity options [24] and make the search tractable. Here, as one would not expect to run large models on an MCU, allowing more connectivity does not make the search intractable.

Thus an MCU-tailored search space should consist of small models, with few restrictions on layer connectivity and granular hyperparameters (see Table 1). The search proceeds by generating random architectures and applying changes to them (*morphisms*) to produce derivative (child) networks.

3.4 Resource constraints

Second design requirement is an **accurate resource use computation.** Performant neural networks are large, so 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 makes a precise resource usage computation essential in identifying how each candidate network will use memory, storage and computational resources (based on an assumed execution strategy) to know which networks lie just below the resource limits. This is not often needed for GPU or mobile platforms, as they are not as resource-constrained.

We focus on three key resource constraints of MCU-sized models: peak memory usage, model size and latency.

3.4.1 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. Branches are commonly featured in popular CNNs as residual connections [13], Inception modules [31], or NAS-designed cells [4]. Here's why: upon reaching a branching point, the inference framework has a choice which operator to execute next. Different execution orders change which

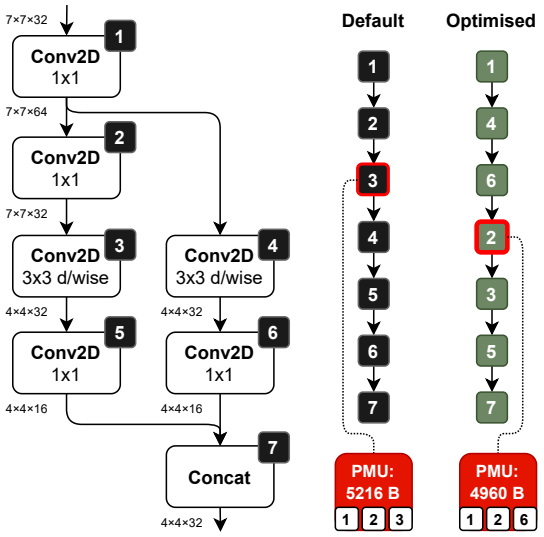

Figure 1. An example network where the default and optimised execution paths yield different peak memory usage.

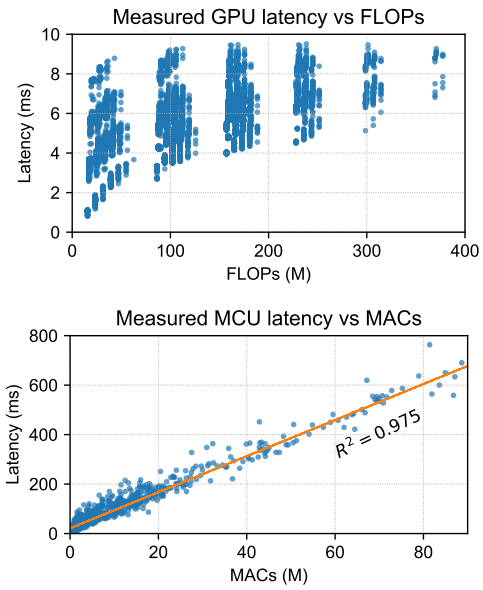

Figure 2. *(Top)* A typical plot of FLOPs vs model latency on a desktop GPU, reproduced from Chau et al. [3]. *(Bottom)* MACs vs measured latency on 1000 models, averaged from ten runs on a NUCLEO-H743ZI2 [30] board. **MACs are a good predictor of model latency on MCUs.**

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 our earlier work [21]: the algorithm which enumerates all topological orders of a network's computational graph to find one that yields the minimal peak memory usage. 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.

3.4.2 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 can be reduced by using quantisation.

We assume an integer-only affine quantisation [15] used in TensorFlow Lite, which quantises each parameter to an 8-bit integer. This is an excellent choice for MCUs, as it is: (a) byte-aligned, (b) does not require floating-point arithmetic units, (c) widely adopted and (d) causes negligible accuracy loss. Thus to compute the storage requirement, NAS counts the number of parameters at 8 bits = 1 byte per parameter.

3.4.3 Latency. 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 [25, 35] or predicting the latency via a surrogate model [2, 3].

Using a predictive model has become the dominant approach for GPU-/mobile-level NAS [3, 32], 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.

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. 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.

3.5 Search algorithm

3.5.1 Handling multiple objectives. We use random scalarisations [27, TS expression] to combine multiple objectives into a single goal (the target function for optimisation).

$$\mathcal{L}^t(\alpha) = \max \begin{cases} \lambda_1^t \ (1.0 - \textsc{ValAccuracy}(\alpha)), \\ \lambda_2^t \ \textsc{PeakMemUsage}(\alpha), \\ \lambda_3^t \ \textsc{ModelSize}(\alpha), \\ \lambda_4^t \ \textsc{MACs}(\alpha) \end{cases} \tag{1}$$

Each objective has an associated scalar term λ_i^t, which specifies its relative importance in the current search goal.

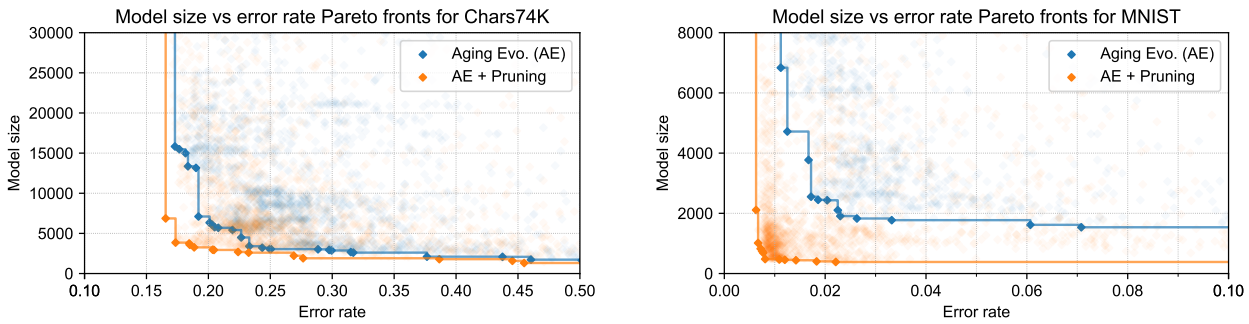

Figure 3. *(Left)* error rate vs size of models discovered by μNAS on the Chars74K dataset; *(Right)* ditto for the MNIST dataset. The further the Pareto front (highlighted) extends into the lower-left corner (low error and low resource usage), the better. **The results show that using pruning yields the furthest advanced Pareto front.**

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 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.

3.5.2 NAS via local search.

We use aging evolution (AE) as a search algorithm for NAS [29] which optimises the goal function by repeatedly evolving a set of candidate points. AE keeps a population of P architectures and, at each search round, samples S architectures and chooses 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.

3.6 Model compression

μNAS supports using model compression during the search: in particular, we hypothesise that network pruning would help in finding small architectures with high accuracy.

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 uses the L_2 norm of channels/units to discard weight groups until the desired proportion is removed [23]. The network is gradually pruned during training until the target "sparsity" proportion (set by μNAS) is reached.

4 Evaluation and Discussion

The aim of the evaluation is to show that our microcontroller-tailored design allows μNAS to **produce superior models**, as compared to previous work on CNNs for MCUs. We also verify the utility of network pruning during search.

4.1 Datasets

We evaluate μNAS on five image classification problems: MNIST [20], CIFAR-10 [18], Chars74K [7] (English subset: 26 upper- and 26 lowercase letters + 10 digits), Fashion MNIST [34] and Speech Commands [33]. Validation sets are used for evaluating objectives and tuning hyperparameters, and the accuracy is reported on the unseen test sets.

We also consider "*binary*" versions of Chars74K and CIFAR-10, which have images partitioned into two classes. We set constraints 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. Experiment details can be found in Appendix A.

4.2 Determining whether to prune

To determine whether pruning helps find models with low resource usage, we run μNAS on Chars74K and MNIST datasets with and without it. Figure 3 shows the model size *vs* error rate (1.0 − accuracy) of discovered models in both cases.

The results show that pruning does help. Here's why: model compression postulates that it is difficult to train small models from scratch (at least for MCU-sized networks). This is also confirmed by the inferior accuracy of models found without pruning. Searching for larger base networks to subsequently prune avoids training small models from scratch: a substantial number of channels/units are only pruned away after a model has reached a high level of generalisation.

4.3 Discovered architectures

Table 2 shows networks discovered by μNAS. For each task, we present a baseline model at multiple objective trade-off

Table 2. Pareto-optimal architectures discovered by µNAS vs others, together with search times. The last column compares a baseline to a model discovered by µNAS in the group. "*Unknown (unk.)*" denotes unreported data. **The results show that µNAS outperforms most baselines by either improving accuracy for comparable resource usage or vice versa.**

Dataset	Model	Acc. (%)	Model size	RAM usage	MACs	Difference
MNIST	SpArSe [9]	98.64	2770	≥ 1960 B	*unk.*	Size ↑ 5.7×
	SpArSe	96.49	1440	≥ 1330 B	*unk.*	Acc. ↓ 2.7%
	BonsaiOpt [19]	94.38	490	< 2000 B	*unk.*	Acc. ↓ 4.8%
	ProtoNN [12]	95.88	63'900	< 64'000 B	*unk.*	Acc. ↓ 3.3%
	µNAS (1174 steps, 1 GPU-day)	**99.19**	**480**	**488 B**	28.6 K	
CIFAR-10 (binary)	SpArSe	73.84	780	≥ 1280 B	*unk.*	Acc. ↓ 3.7%
	µNAS (1206 steps, 2 GPU-days)	**77.49**	**685**	**909 B**	41.2 K	
Chars74K (binary)	SpArSe	77.78	460	**≥ 720 B**	*unk.*	Acc. ↓ 3.4%
	µNAS (743 steps, 0.5 GPU-day)	**81.20**	**390**	867 B	107 K	
Speech Commands	RENA [38]	94.04	47 K	*unk.*	≈700 M	MACs ↑ 636×
	RENA	**94.82**	67 K	*unk.*	≈3'265 M	MACs ↑ 2968×
	DS-CNN [37]	94.45	**<38.6 K**	*unk.*	≈2.7 M	MACs ↑ 2.2×
	MCUNet [22]	≈91.20	< 1 M	80 KB	*unk.*	Acc. ↓ 4.4%
	MCUNet	**≈95.91**	< 1 M	311 KB	*unk.*	RAM ↑ 14.7×
	µNAS (1960 steps, 39 GPU-days)	**95.58**	**37 K**	**21.1 KB**	**1.1 M**	
	MN-KWS (L) [1]	**95.3**	612 K	208 K	*unk.*	Size ↑ 31.8×
	µNAS (1514 steps, 30 GPU-days)	**95.36**	**19.2 K**	25.7 KB	1.1 M	
Fashion MNIST	reported [36]	92.50	≈100 K	*unk.*	*unk.*	Size ↑ 1.6×
	µNAS (1161 steps, 3 GPU-days)	**93.22**	**63.6 K**	12.6 KB	4.4 M	
CIFAR-10	LEMONADE [8]	**≈91.77**	**10 K**	*unk.*	*unk.*	
	µNAS (4205 steps, 23 GPU-days)	86.49	11.4 K	15.4 KB	384 K	Acc. ↓ 5%
Chars74K	*µNAS (1755 steps, 2 GPU-days)*	82.65	3.85 K	9.75 KB	279 K	
	µNAS (1724 steps, 2 GPU-days)	76.05	13.9 K	1.81 KB	111 K	

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**. We observe: (a) an improved 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 at least ≈2× (see Speech Commands).

Targeting Speech Commands in particular shows the importance of considering MACs during the search: orders of magnitude faster models (in MACs) can be discovered for comparable model size and accuracy. For MNIST, the search found a pruned (yet still dense) model with < 0.5 KB parameters and > 99% accuracy.

For CIFAR-10, µNAS is able to outperform the baseline for the binary dataset; however, it falls short compared to LEMONADE [8] for a full (10-class) dataset. Upon closer investigation, we find that the µNAS does gradually push the Pareto front, but requires many steps to do so.

We predict that if given more search time, µNAS can discover better models for CIFAR-10. It is not unusual for NAS systems that use evolutionary optimisation with no model training shortcuts to take many GPU-days: in fact, original AE experiments are reported to have taken over 3000 GPU-days [29]. If search time is an issue, we envision the above being rectified by: (a) not using the same search space throughout, 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 [3]; or (c) employing weight sharing to amortise the cost of training each candidate network.

5 Conclusions

Neural architecture search is a powerful tool for automating model design, especially when manual design 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 and explicit targeting of the three primary resource bottlenecks, we are able to create a NAS system, µNAS, that discovers resource-efficient models for a variety of image classification tasks.

Acknowledgements

This work was supported by Samsung AI and by the UK's Engineering and Physical Sciences Research Council (EPSRC) with grants EPM50659X1 and EPS0015301 (the MOA project) and the European Research Council via the REDIAL project (Grant Agreement ID: 805194).

References

[1] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas Navarro, Urmish Thakkar, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul N Whatmough. 2020. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. *arXiv preprint arXiv:2010.11267* (2020).

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

[3] Thomas Chau, Łukasz Dudziak, Mohamed S Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D Lane. 2020. BRP-NAS: Prediction-based NAS using GCNs. *arXiv preprint arXiv:2007.08668* (2020).

[4] Hsin-Pai Cheng, Tunhou Zhang, Yukun Yang, Feng Yan, Shiyu Li, Harris Teague, Hai Li, and Yiran Chen. 2019. SwiftNet: Using Graph Propagation as Meta-knowledge to Search Highly Representative Neural Architectures. *arXiv preprint arXiv:1906.08305* (2019).

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

[6] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, et al. 2020. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *arXiv preprint arXiv:2010.08678* (2020).

[7] Teófilo Emídio De Campos, Bodla Rakesh Babu, Manik Varma, et al. 2009. Character recognition in natural images. *VISAPP (2)* 7 (2009).

[8] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2018. Efficient multi-objective neural architecture search via Lamarckian evolution. *arXiv preprint arXiv:1804.09081* (2018).

[9] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. 2019. SpArSe: Sparse architecture search for CNNs on resource-constrained microcontrollers. In *Advances in Neural Information Processing Systems*. 4977–4989.

[10] Javier Fernandez-Marques, Paul N Whatmough, Andrew Mundy, and Matthew Mattina. 2020. Searching for Winograd-aware quantized networks. *arXiv preprint arXiv:2002.10711* (2020).

[11] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. 2019. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420* (2019).

[12] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. ProtoNN: Compressed and accurate knn for resource-scarce devices. In *International Conference on Machine Learning*. 1331–1340.

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

[14] Chi-Hung Hsu, Shu-Huan Chang, Jhao-Hong Liang, Hsin-Ping Chou, Chun-Hao Liu, Shih-Chieh Chang, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Da-Cheng Juan. 2018. MONAS: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332* (2018).

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

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

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

[18] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[19] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient machine learning in 2 KB RAM for the internet of things. In *International Conference on Machine Learning*. 1935–1944.

[20] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

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

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

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

[24] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[25] Jieru Mei, Yingwei Li, Xiaochen Lian, Xiaojie Jin, Linjie Yang, Alan Yuille, and Jianchao Yang. 2019. AtomNas: Fine-grained end-to-end neural architecture search. *arXiv preprint arXiv:1912.09640* (2019).

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

[27] Biswajit Paria, Kirthevasan Kandasamy, and Barnabás Póczos. 2020. A flexible framework for multi-objective bayesian optimization using random scalarizations. In *Uncertainty in Artificial Intelligence*. PMLR, 766–776.

[28] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameters Sharing *(Proceedings of Machine Learning Research (PMLR), Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). 4095–4104.

[29] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 4780–4789.

[30] STMicroelectronics. 2020. STM32 Nucleo-144 development board with STM32H743ZI MCU, supports Arduino, ST Zio and morpho connectivity. https://www.st.com/en/evaluation-tools/nucleo-h743zi.html (Accessed Aug 2020).

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

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

[33] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209* (2018).

[34] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[35] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. 2018. SNAS: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926* (2018).

[36] Zalando. 2020. `zalandoresearch/fashion-mnist` Github repository. https://github.com/zalandoresearch/fashion-mnist (Accessed Aug 2020).

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

[38] Yanqi Zhou, Siavash Ebrahimi, Sercan Ö Arık, Haonan Yu, Hairong Liu, and Greg Diamos. 2018. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912* (2018).

[39] Barret Zoph and Quoc V Le. 2016. 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 1) 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.

As far as we are aware, the full Chars74K or Fashion MNIST datasets do not have strong low resource usage baselines. Except for CIFAR-10, µNAS was run for 2000 steps. The search took up to 39 GPU-days for Speech Commands, and up to 3 GPU-days for the remaining tasks.

A.2 Dataset preprocessing and model training schedule

Dataset preprocessing, as well as model training and pruning information, is given in Table 4 (see next page).

A.2.1 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. [22], we observe no loss in accuracy. This is not surprising due to the full-integer quantisation in TensorFlow Lite [15] 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.

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

Table 4. Dataset preprocessing, model training and pruning configurations used in our experiments.

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 ± 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 [0.10; 0.85]; • pruning between epochs 20 and 53
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 [0.10; 0.90]; • pruning between epochs 30 and 60;
Speech Commands	as given here[2], with 30ms window size, 10ms stride, and MFCCs extracted between 20 Hz and 4000 Hz.	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 [0.10; 0.90]; • 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