Deep Learning on Microcontrollers: A Study on Deployment Costs and Challenges

Filip Svoboda
University of Cambridge

Javier Fernandez-Marques
Samsung AI

Edgar Liberis
University of Cambridge

Nicholas D. Lane
University of Cambridge and Samsung AI

Abstract

Microcontrollers are an attractive deployment target due to their low cost, modest power usage and abundance in the wild. However, deploying models to such hardware is non-trivial due to a small amount of on-chip RAM (often < 512KB) and limited compute capabilities. In this work, we delve into the requirements and challenges of fast DNN inference on MCUs: we describe how the memory hierarchy influences the architecture of the model, expose often under-reported costs of compression and quantization techniques, and highlight issues that become critical when deploying to MCUs compared to mobiles. Our findings and experiences are also distilled into a set of guidelines that should ease the future deployment of DNN-based applications on microcontrollers.

CCS Concepts: • **Computer systems organization** → **Embedded and cyber-physical systems**; • **Computing methodologies** → *Modeling and simulation.*

Keywords: microcontrollers, neural networks, compression, quantization

ACM Reference Format:
Filip Svoboda, Javier Fernandez-Marques, Edgar Liberis, and Nicholas D. Lane. 2022. Deep Learning on Microcontrollers: A Study on Deployment Costs and Challenges. In *2nd European Workshop on Machine Learning and Systems (EuroMLSys'22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3517207.3526978

1 Introduction

Modern smartphones can run a 22-layer DNN for image classification in under 100ms [13], reaching competitive accuracy levels on the ImageNet [8] dataset. In this work, we study the challenges associated with the deployment of DNNs on significantly more constrained platforms: microcontrollers (MCUs). We focus on the challenges which do not generally arise in other mobile systems (e.g., smartphones), but are central to the deployment and execution of DNNs on MCUs.

Despite the limited memory and compute resources, MCUs can be an attractive target platform for some DNN-powered applications. Battery-powered devices that perform scheduled or occasional event-triggered processing [9, 15] would benefit from near-zero power use in sleep mode. Using an off-the-shelf MCU would also significantly reduce both development and per-unit costs. Finally, running necessary computations on-device would also eliminate the need for server-side processing and communication overheads.

Indeed, deploying DNNs on MCUs has become an area of active study in the recent years [3, 23, 24, 26, 36], and frameworks such as TensorFlow Lite Micro [7] make the deployment relatively straightforward in certain settings. However, the deployment of models that do not trivially fit the limits of MCUs faces very significant difficulty in closing the resource gap and remains an unsolved problem. In many respects, one could say that the deployment of DNNs on MCUs is where it used to be for smartphones back in 2015.

In this work, we study the difficulties introduced primarily by the lack of on-chip memory along with other residual factors. Notably, we elect to not use any of the existing MCU supporting DNN frameworks at inference time. This bare-metal approach allows us to make observations without interference from the design decisions of any one framework. Our core observation is that hardware-aware design choices during training (*e.g.* the way quantization is done) and network architecture (*e.g.* the size of activations in any given layer), while important in other mobile setups, become critical when deploying to MCUs. More specifically:

- We describe challenges and opportunities of MCUs and contrast them with mobile platforms (Section 3)
- We investigate memory hierarchy of MCUs and propose ways of mitigating the impact in latency and energy associated to data movement (Section 5).
- We study different design choices for quantization as a simple form of model compression (Section 6).
- We expose the deficiency of current high-level metrics for assessing the suitability of a (compressed) model for deploying to an MCU (Section 7).

Finally, Section 8 provides a set of deployment guidelines.

EuroMLSys'22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9254-9/22/04.
https://doi.org/10.1145/3517207.3526978

2 Experimental Setup

We work with the Mobilenet_V2_0.35_128 architecture which is commonly used in mobile computer vision applications [29]. To compile our microcontroller software, we use Arm's *gcc-arm-none-eabi-8-2018-q4-major* toolchain along with the CMSIS library and Mbed OS. We set the `-O3` optimization level in Release mode and obtain 1764 KB binaries.

2.1 Model Deployment Pipeline

Modern deep learning frameworks provide easy ways to transfer a neural network from a training environment to a deployment on a GPU / TPU accelerator. However, targeting microcontrollers requires extra manual steps to work around memory and compute limitations discussed previously. In our experiments, we take the following steps on the host machine to prepare the model for deployment on the device.

Simulate quantization on host. We start with a model graph in TensorFlow [1], built using the Keras API [6], and insert TensorFlow's fake quantization operators after each tensor that needs to be quantized. This allows us to fine-tune the model with quantized weights. We use 8-bit quantized tensors throughout the model with 32-bit accumulators for intermediate values. This methodology was first used in Jacob et al. [19]; we use a customized version of `tensorflow.quantize` to rewrite the computation graph.

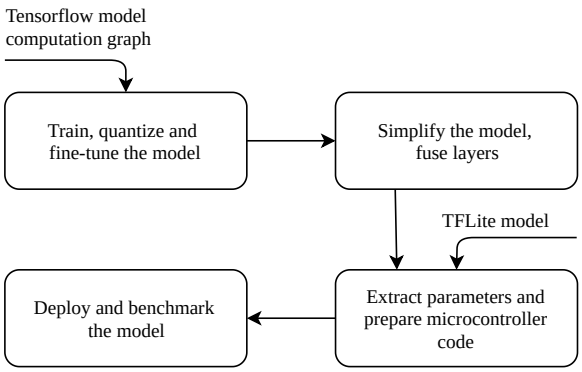

Figure 1. Model deployment pipeline: major conceptual stages of turning a TensorFlow graph into a DNN running on a microcontroller.

Simplify and export the model. The computation graph can be further simplified before deployment since the model is expected to operate only in inference mode. Adjacent layers can be fused to reduce the number of operations and memory accesses at runtime: activations and other element-wise operations are combined with the preceding layer, which also enables them to operate at higher precision intermediate values. Batch normalization layers in inference mode scale and shift their inputs by a learned constant (one multiplication and addition) which can be achieved by weights and biases of the preceding layer. We use a converter provided by TensorFlow Lite to simplify the computation graph, "fold" batch normalization layers and produce a compact model in the TFLite format.

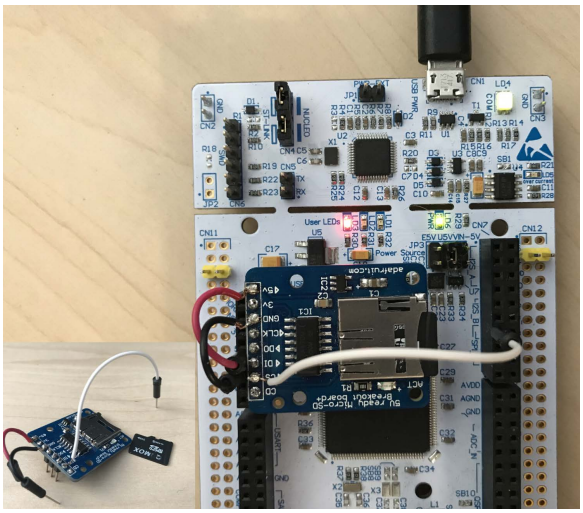

Figure 2. Nucleo-767ZI with attached μSD card shield. This is the board we used in all our experiments.

Generate microcontroller code. Finally, we inspect the serialized TFLite model (produced by the steps above or otherwise) and generate C++ code for the microcontroller. We leverage a custom-built library for NN computations which contains reference implementations of operations for uniform affine quantization and a wrapper around CMSIS-NN for Q-format quantization. We also change the memory layout of tensors for better locality, export weights in both binary and code formats (for storage on SD card and flash respectively) and statically partition the memory to avoid memory management overhead at runtime.

2.2 Hardware Setup

We use the Nucleo-F767ZI board (see Table 1 for an overview of its hardware specifications). To enable the deployment of models larger than the available on-device Flash memory, we make use of the μSD card adaptors from AdaFruit connected over SPI. This adaptor is attached to the Nucleo boards using pin-to-pin data and clock connections to minimise the impact of cabling (see Figure 2). We use a 2 GB Class 4 (C4) μSD card in all our experiments and, unless stated otherwise, the operating frequency is set to 1 MHz (default). We use the Monsoon HV Power Monitor to power the board at 3.3V and sample energy consumption at 5 KHz. We rely on the internal timers of the device to obtain latency measurements.

This setup allows us to carefully control how we obtain our measurements. The absolute values of metrics would differ from a production device (due to it being more optimized), but the relative results will still hold in the closer-to-product implementations.

3 Deploying Deep Neural Networks on Microcontrollers

A typical microcontroller is comprised of a processing core, on-chip static random access memory (SRAM) and NOR-Flash memory, where the program code lives. It is several orders of magnitude more resource-constrained than today's average smartphone, especially in terms of on-chip RAM. In our study, we consider the family of Cortex-M microcontrollers from Arm. Some of these currently commercially available MCUs are shown in Table 1.

MCUs are an attractive application platform:

Low power usage. Microcontrollers have small SRAM chips and relatively slow cores which helps to achieve power consumption of 5–150μW/MHz, depending on the model and input voltage.

DSP capabilities. For Cortex-M devices, the CMSIS-DSP libraries provide highly efficient low-level implementations of signal processing primitives, which are equally usable for DNNs. More recently, the CMSIS-NN [22] library was released to provide optimized kernels which leverage SIMD instructions on supported MCUs.

Affordability. Current prices range between $0.75 per unit for a less capable Cortex-M0 to $10 per unit for a top-tier Cortex-M7.[1] Such low prices are achieved by using simple chip designs with very few components and employing old-generation lithography—MCUs are often produced using the 40nm or 90nm specification [32].

Cheap on-device computation. MCUs, data movement is the primary source of energy consumption[33]. Inter-device data movement is costly, even when relying on low-power close-range communication mechanisms. For example, transmission costs using BLE or ZigBee are in the order of hundreds of milliwatts [30]. Unlike displays and radios, processing cores and sensors use considerably less power: real-time audio processing at 384KHz requires 10–25mW [14] on a Cortex-M4; low-resolution image sensors for object tracking consume 277μW [28]. This shows that the communication cost of offloading can exceed the computation cost itself, had it been done locally. This makes MCUs an attractive deployment target for applications that do not necessarily require low-latency or real-time processing.

[1] As catalogued by Mouser Electronics (Feb 2019) for an order of a thousand units of just cores without development boards.

Model	Processor	Clock	SRAM	Flash	FPU
F767ZI	Cortex-M7	216 MHz	512 KB	2 MB	✓
F746ZG	Cortex-M7	216 MHz	320 KB	1 MB	✓
F429ZI	Cortex-M4	180 MHz	256 KB	1 MB	✓
F207ZG	Cortex-M3	120 MHz	128 KB	1 MB	×
F091RC	Cortex-M0	48 MHz	32 KB	256 KB	×

Table 1. Selected commercially available MCUs from Arm in development STM32 Nucleo boards form factors.

3.1 Mapping DNNs to MCUs

From a computational point of view, a deep neural network is a sequence of linear algebra operations, along with some element-wise non-linearities, grouped together into "layers".

The deployment of neural networks to MCUs involves deciding where the weights, biases and activations lie, how the compute operations are scheduled and how the data gets accessed and processed. Executing a model that does not fit into on-chip memory could then take the following steps for each layer in turn:

1. Load a binary blob representing the compressed weights of the layer into RAM from the storage.
2. Revert any compression of weights, *e.g.* revert Huffman encoding or K-means clustering, as suggested by Han et al. [10]. If the decompressed tensor does not fit into RAM, attempt to store it in the Flash memory or, if it fails, onto the SD card.
3. Execute the computation of the layer (inputs are assumed to be already in RAM). If decoded weights were stored in the backing storage, load them in chunks as needed.
4. Store the result in a pre-allocated buffer in SRAM. This will become an input to the following layer.

In order to assess how data flow impacts latency or energy consumption, we study the costs of different types of data movement, data formats and encoding schemes in detail.

3.2 The Challenges of MCUs

Besides model size, inference latency and energy consumption, MCUs are subject to their unique set of challenges:

Working set may not fit in RAM. In the context of DNNs, the working set of a layer is the amount of memory needed to store its weights, biases and activations. An inability to hold the working set in SRAM will significantly deteriorate both latency and battery life. We further discuss this in Section 4.

Data movement is costly. Data movement is the primary source of energy consumption on MCUs. In Section 5 we study the costs of reading and writing DNN data to and from SRAM, flash and μSD card.

Hidden costs. Since running a DNN on an MCU may become an I/O-bound task, methods aimed at reducing the number of parameters or MACs at the cost of more data movement may actually hurt performance. Extra costs introduced by such methods have been insufficiently studied in existing literature—we discuss them further in Section 7.

4 Working Set

The working set profile of a DNN often displays significant variation across layers. Relative to the aggregate total model

size, it gives a richer overview of the minimum resources needed to run a network. Figure 3 shows the working set of the baseline MobileNet, the standard in mobile sensing. We see that the working set exceeds the available MCU SRAM at the 6th and the 68th layers. The latter, even though serious is less concerning since reading from an external memory source is much cheaper than writing to it. Thus, even though the network fails to fit in the SRAM at two points only one of these failures prohibits realistic deployment.

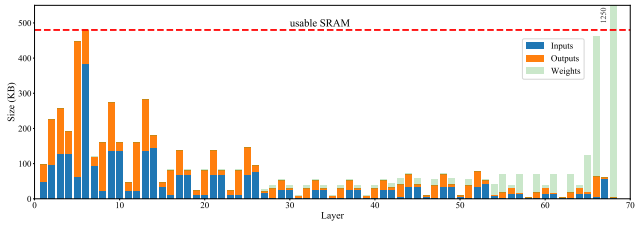

Figure 3. Per-layer working set of MobileNet_V2_0.35.

There is a fundamental difference between those layers whose working set sizes are close to the SRAM limit and those with working sets at a comfortable distance from it. An experiment in Figure 4 demonstrates the impact of extra filters added to every layer of the baseline MobileNet. The left panel reports the increased working set sizes. The right panel reports the per-layer increase in forward pass latency that resulted from this network expansion.

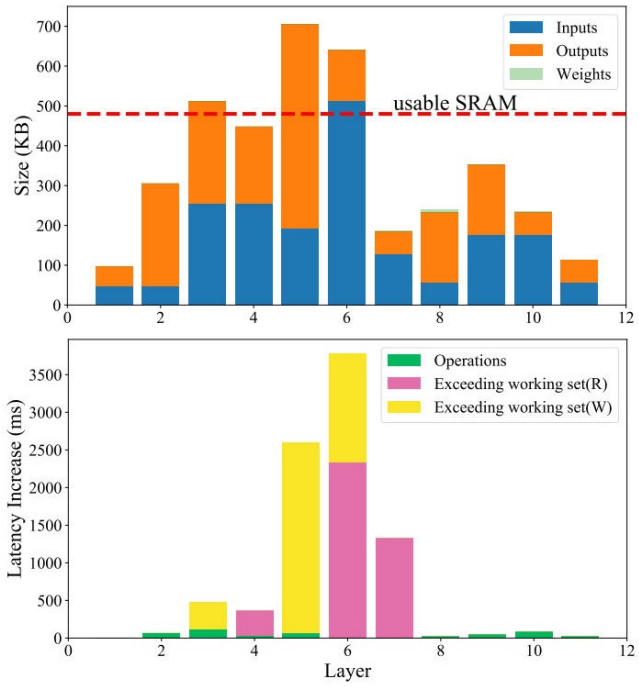

Figure 4. Impact of adding 32 more filters to each of the first ten layers of MobileNet_V2_0.35.

The latency due to the added operations (green) is dwarfed by those caused by writing out and storing off-SRAM the excess output at layers 3, 5, and 6 (yellow) and then reading it back at the next layer (pink). Note that layers 3 and 5 have only write-related latency penalty, as they exceed the SRAM size themselves but were not preceded by a layer that would do the same and similarly layers 4 and 7 have read-only.

Relative to these impacts the other layers are affected only marginally since the operation penalty is marginal. For this reason we omitted layers far away from the SRAM limit.

In conclusion, when networks exceed the amount of on-chip RAM they face a significant amount of data movement, which, as would be further studied in the following section, results in latency and energy penalties [2].

5 Data Movement

The data movement costs are a function of the number of bytes to move and the position of the origin or destination memory in the system's memory hierarchy. MCUs often rely on L2 SRAM chips as their main general purpose memory. Bigger pieces of data can be stored on non-volatile memories such as NAND-Flash based SSDs and SD cards. Accessing these requires two to three orders of magnitude more energy than accessing the MCU's SRAM. SD cards are versatile for on-the-wild deployments [14], easy to use, and work well in compact form factors suitable for MCUs.

At inference, each MAC requires three memory reads (one per buffer in the working set *WS*) and one memory write to update the output. Having an entire high-performance model in SRAM is not realistic. Therefore, in the worse case, each MAC requires three $3 \times \mu$SD card read accesses and one write. To provide some context, the reference MobileNet_V2 model requires 300 million MACs per inference pass.

5.1 Implications for Inference

We evaluate the impact of: independently relying on SRAM, on-chip Flash (NOR-Flash) and μSD card storage (NAND-Flash), the asymmetric nature of reading/writing from/to flash, as sectors need to be erased prior writing on them; and the impact of the operation frequency of μSD card.

Memory hierarchy impact. We measured the latency and energy consumed while evaluating the last layer of our MobileNet_V2 network when weights reside in either SRAM, on-chip Flash or external μSD card. The input to this convolutional layer is a $[1 \times 4 \times 4 \times 112]$ tensor and the weights are of shape $[1280 \times 1 \times 1 \times 112]$. Retrieving the weights from μSD card results in almost a two orders of magnitude increase in latency. When weights are either in SRAM or Flash, latency is identical when measured at a resolution of 1 ms, but energy consumption is marginally (+0.07mJ) higher when using the weights in Flash.

[2] See Siu et al. [31] for a a study of domain-specific working-set trade-offs.

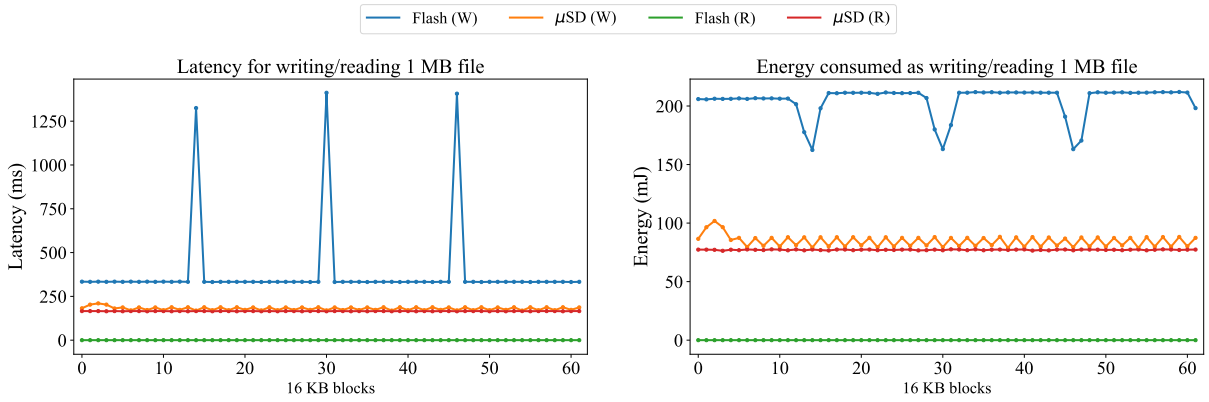

Figure 5. Latency and energy costs of writing/reading 1 MB to/from on-chip flash and μSD card.

Flash re-usage impact. In this experiment we investigate the cost of off-loading excessive working set onto the SD card. We measure the latency and energy required to store and read 1 MB of data in SRAM and the SD card. This is a realistic setup as inference would be performed multiple times on the device. We measure the costs of reading/writing such amount of data every 16 KB blocks. In Figure 5 we observe that, while μSD card read/write is comparably costly, there is a big disparity between the costs associated to Flash read/write operations. Consequently, is to only worth writing to on-chip flash during runtime if the data to be stored is planned to be used multiple times during network inference.

Operation frequency of μSD card. From the previous experiment, we argue that NAND-Flash (e.g. μSD card) is preferable when needing to temporarily offload some data from system memory. In this experiment we study the impact in energy consumption that the operating frequency of the μSD card has in the overall system. We measure the energy required to write to μSD card arrays of different dimensions, resulting in file size ranging from 10 Bytes to 200 KB. In our development setup, we observe power consumptions of ~460mW when setting the frequency to 100KHz. Power goes up to ~590mW when operating at the highest frequency supported by Mbed OS, 25 MHz (see Figure 6). Although higher frequencies translate into an up to 25% power drawn, the overall energy consumed suggests that using higher frequencies is more economical. We highlight with the operation frequencies at which the writing operation results in the least energy consumed.

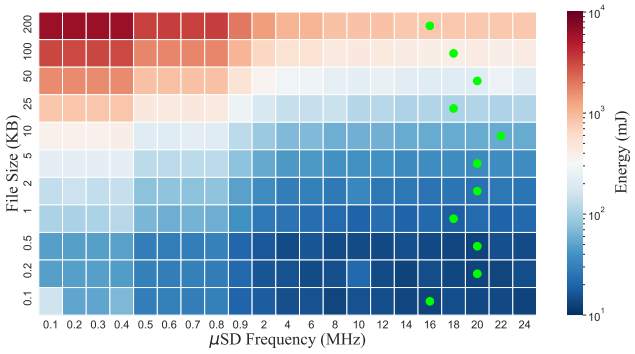

Figure 6. Energy costs of writing activation arrays to the external μSD card when varying its operation frequency.

6 Quantizing Neural Networks

Quantization represents model's parameters with fewer bits than the 32-bit used in training. The 8-bit quantization was shown to reach comparable performance to full-precision models [19, 20], to be deployable on off-the-shelve mid- to low-end hardware, and to result in a direct 4× model size reduction and up to 116× and 27.5× chip area reduction [16].

6.1 Dimensions to Quantization

We present the three dimensions of quantization: the bit-width, the granularity and the implementation.

Bit-width. This refers to the number of bits used to represent the weights of a neural network. 32-bits, 16-bits, 8-bits, and 1-bit are popular options due to being well-supported by existing hardware.

Granularity. Options include tuning the bit-width of *each weight* (fine-grained quantization), setting the bit-width for *each layer* or using the same precision *throughout the model* (coarse-grained quantization). Intuitively, finer quantization is more flexible and should approximate the full-precision model better. However, it requires additional logic and book-keeping due to the (potential) mismatch in bit-width of layer's operands, weights and activations. A common compromise is to perform layer-wise quantization [20, 25, 27, 37].

Implementation. Most common implementations are:

Q-format. It is a method that splits the n bits representing a real value into integer and fractional parts. It is denoted $Q_{a.b}$, where a is the number of bits for the integer part, b

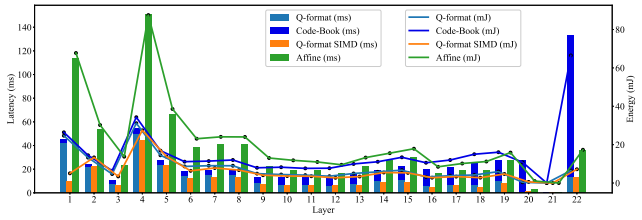

Figure 7. The latency (ms) and energy (mJ) of running a quantized MobileNet_V2 network on the Nucleo-F767ZI.

those for the fractional part and, $n = a+b$. For 8-bit representations, n is usually set to 7 and the 8-th bit becomes the sign bit. Operating with Q-formatted data is well supported in most hardware as it is an efficient method to perform operations that would otherwise require floating point hardware support. The main overhead is keeping track of the decimal position when performing operations in which operands assign different number of bits for fractional or integer part. This scenario is common in neural networks as the output of layer L_i becomes the input of layer L_{i+1} and these are likely to be quantized independently. The main limitation of this quantization implementation is the fact that it homogeneously splits the $[a, b)$ range for $Q_{a.b.}$, which becomes a problem for long-tailed or heavily asymmetric distributions.

Affine. This implementation represents a real number $r \in \mathbb{R}$ as the *affine mapping* of an integer q using parameters S and Z. This is defined as $r = S(q - Z)$, where S is a FP32 value and scales the zero-centred quantized number q. This centering is performing by applying offset Z, which is quantized to the same number of bits as q. These extra parameters result in extra arithmetic operations during *e.g.* matrix multiplication. We refer the interested reader to Jacob et al. [19] for a detailed description. A common practice is to learn a single $\{Z, S\}$ pair for each layer.

Code-book. This term groups most of the compression frameworks proposed for DNNs to date. While lossless approaches exists, such as those based on Huffman [17] codes or Lempel-Ziv [38] (e.g. LZ77 or LZ78) algorithms, the majority of these are lossy and therefore trade compression ratio with accuracy degradation. For our study we chose a combination of lossy and lossless compression and concatenate K-means clustering and Huffman encoding on 8-bit Q-format data. In order to operate with such data representation we need to (1) decode (i.e. revert Huffman encoding) and (2) *de-cluster* (i.e. revert K-means) the weights.

6.2 Quantized DNNs for MCUs

We measure the impact in performance of each of the quantization implementations on a MobileNet_V2_0.35 network. The results, as shown in Figure 7, verify that the simplicity of a Q-format implementation results in lower latency, specially when exploiting SIMD operations, compared to the alternatives. Code-book quantization underperforms significantly due to the latency needed to unpack its codebook. The affine quantization, similarly, requires longer processing times. Finally, we observe that, regardless of the quantization implementation, there is a direct correlation between latency and energy consumption.

7 The Weakness of High-level Metrics

Inherent to all quantization implementations presented in Section 6 is the need for additional parameters in order to give meaning to the quantized weights. These parameters are *decimal point* location, in the case of Q-format quantization; and *scale* and *zero point* values, for affine quantization. They often represent a negligible portion of the total model size and their existence will not therefore constrain the deployability of such models. However, their introduction does translate into an increase in latency, particularly in the case of affine quantization, as it was observed in Figure 7.

Code-book approaches, which have been the predominant form of model compression [10, 18, 35], suffer from additional overheads. These overheads come in the form of frequent lookups to interpret their weight encoding. These costs are *hidden* from users because: (1) they rarely get reported when a new compression framework is presented or when compared against the previous state of the art; and (2) their impact during model inference in platforms where the fully uncompressed model does not fit in memory, as it is likely to happen in MCUs, has not been studied.

Relying on high-level metrics such as model size or total number of (high-level network architecture) MACs needed for model inference could misslead the choice of an architecture given an application and target platform. To illustrate why these costs are relevant to the scenario of deploying DNNs on MCUs we use as an example the SqueezeNet architecture compressed with the Deep Compression [10] framework. As reported Iandola et al. [18], this network results in 0.47MB in model size while maintaining the same accuracy as AlexNet [21] on ImageNet [8]. The SqueezeNet network is 510× smaller than AlexNet. A sub-megabyte model such as this one would fit in all but the most constrained MCU from Table 1. However, this compact model-representation cannot be used directly to classify an input. It needs to be unpacked first. This process will transform the model from the initial 0.47MB up to 4.8MB, which no longer fits in memory or flash and therefore will result in a significant impact in both latency and energy consumption.

We show in Figure 8 that high-level metrics alone might be misleading when assessing the suitability of a network or a compression framework foran MCU. We show that the stress in memory due to the sizes of the input and output tensors of a layer is far greater than that resulting from the tensors

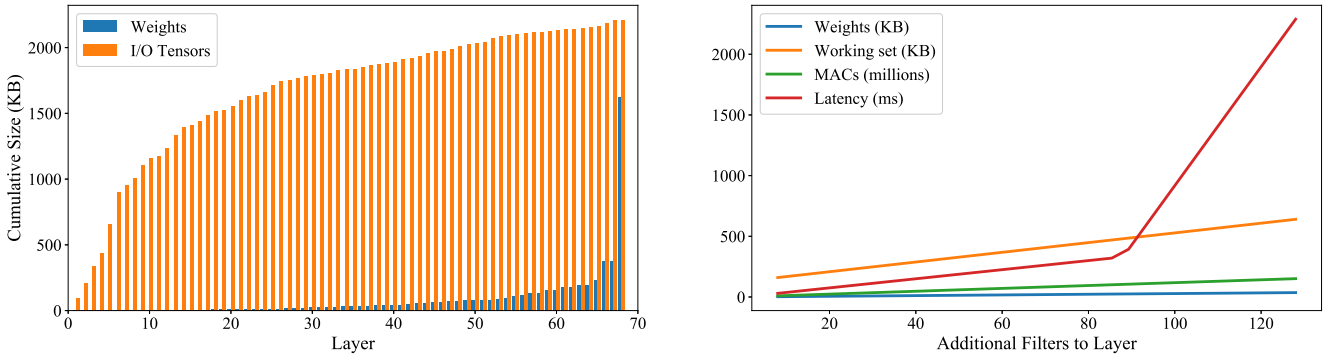

Figure 8. Left: per-layer cumulative memory usage of MobileNet_V2. Right: WS, MACs and Latency vs additional layer filters.

containing the weights (which is precisely what the model size metric measures). We also show that, linearly increasing the model size, does not translate necessarily into a linear increase in latency. We can observe how other parameters of the hardware (in this case the amount of usable SRAM) significantly limits the throughput of the application when exceeding the maximum working set limit.

8 Guidelines for MCU ML Developers

MCUs, just as mobile devices, are an appealing application platform. They are highly energy efficient, affordable, and are virtually omnipresent with trillions presently deployed. However, their resource limitations are a much more extreme compared to mobile devices. These additional constraints have to be carefully considered at design stage, making the complexity of moving DNNs from mobile platforms to MCUs comparable to the difficulty experienced with migrating models from server to mobile platforms.

Mobile devices led the way on deploying deep neural networks (DNN) in constrained setups. They are limited in RAM size, Flash size, and Op-s per second relative to DNN's original server platforms. But, they offer better setup (hardware cost) as well as operation (energy consumption) affordability. Moreover, easier accessibility of mobile devices allows wider access to the services DNNs can offer.

Developers can leverage existing tools and techniques aimed at mobile devices to get most of the way to a successful deployment on an MCU. However, to make the model run at reasonable levels of accuracy, power use and latency, the following points have to be considered:

Working set size. Due to little amount of on-chip memory in MCUs, the working set size becomes the bottleneck, forcing large model to page tensors to external storage. Avoiding interaction with external storage will improve overall inference latency and power consumption.

Data movement. Data movement makes up a significant proportion of overall energy consumption on an MCU. This can quickly exceed the amount of energy consumed by compute operations: incredibly, moving 1 byte across memories can equate to many floating point operations. Minimising the amount of data movement should be one of the primary considerations for a model designer.

Quantization. Quantization is an attractive technique to significantly reduce the model and working set sizes. It may be a required step if the target device does not have DSP circuitry and hence does not support floating point operations. The methods offer different trade-offs between representation accuracy and bookkeeping and implementation complexity. Using a natively supported bit-width would allow to leverage SIMD on supported devices.

Beware of misleading metrics. Commonly, lightweight model design and compression optimize for the total model size and multiply-accumulate operations. While these metrics allow for meaningful comparisons on server and mobile platforms, on MCUs they can hide data movement costs.

9 Related Work

This paper is best seen in two contexts. The first is the hardware context in which it is closely related to the literature on hardware acceleration. The second is the modelling context which benefits from the paper's lessons learned.

In the hardware space, a close cousin of the here-deployed MCUs are the application-specific integrated circuits (ASICs) and the field-programmable gate arrays (FPGAs). Researchers have, in the recent years, made a considerable progress towards running deep models on both of them.

ASICs: as their name suggests, these are fixed pieces of silicon fully-optimized for a very specific task. Consequently, they can't be reprogrammed. Cavigelli et al. [5], proposed the first such accelerator aimed at running convolutions. It avoided the problem of fitting a deep model by focusing on the simpler issue or fitting a single convolution layer on the hardware. This method was then improved on and finally measured on silicone to provide up to 196 GOp/s on 3.09 mm2 in UMC 65-nm technology and to achieve a power

efficiency of 803 GOp/s/W [4]. Andri et al. [2] argued that this performance, impressive as it may be for a first-one-ever, fell short in its energy efficiency to be deployable on an actual mobile device. They proposed an extension based on network binarization - the process of quantizing network weights with only one bit.

FPGAs: are configurable integrated circuits that offer the developer the opportunity program their logic blocks. Han et al. [11] proposed a compression method that allows a LSTM-based speech recognition model to run on the Xilinx XCKU060 FPGA. This is running at 200MHz with a performance of 282 GOPS and running the same model has 40 and 11.5 times higher energy requirement on server-grade CPU and GPU respectively. The flexibility that facilitated the deployment came at a considerable monetary premium.

Overall, accelerators, both ASIC and FPGA, have demonstrated their ability to run some form of deep models. This, however comes with significant strings attached.

ASICs are application bound, meaning that they have to be re-developed for every architecture design separately. Moreover, creating each new ASIC requires longer development cycles including sending the chip schematics to foundry - a process with challenges in itself (e.g. fabrication process). [4] proposed that this cost can be alleviated by building accelerators for specific layer types as opposed to networks in their entirety (such as their convolution Origami). This, while true of marginally different networks (i.e. those that share basic features) is not true of the field at large. Specifically, the differences between state of the art in any given Deep Learning domain were not due to different organization of broadly similar layer types. To the contrary, often the development of the state of the art was driven by the introduction of novel, very unique, innovations in the types of deployed network layers - for instance the residual connections in the bottleneck layers [12]. Therefore, while the pressure on the chip development can be partially alleviated, for most part the costs related to model deployment on ASICs remain large and network-bound.

FPGAs, are much more flexible in that the hardware can be re-programmed, potentially even at the inference time. Consequently, at first glance they are much more viable and palatable alternative to ACISs. This flexibility, however, comes at a significantly increased per-unit cost. For many products this can be prohibitive.

In the modelling space there have been successful attempts at using reinforcement learning and smart searching to either compress or design MCU-depolyable architectures. In the compression space [34] developed a Reinforcement Learning agent-powered quantization method. This method quantizes network's layers such that its hardware simulator-derived energy and latency metrics are minimized.

NAS: In the architecture design space Banbury et al. [3], Lin et al. [26] used neural architecture search (NAS) to discover tiny DL architectures deployable on MCUs. The earlier work, MCUNet, jointly learns the constrained architecture and the lightweight inference engine that is to power it. It is based on a standard NAS design whereby the search space is first constrained and then explored [26]. Banbury et al. [3] observe that in the constrained NAS search space of the MCU-deployable architectures model latency varies linearly with model operation. That means, they argue, that operation count can be used as a proxy for on-device latency in the search process. This, they conclude, lends itself well to the use of differentiable NAS (DNAS) which optimizes a continuous objective, rather than just searches the search space. This observation allowed their search algorithm to significantly improve on the Pareto-frontier achieved by MCUNet. Further advancements in the field of NAS-learned architectures were achieved by expanding the set of objectives the agent aims to optimize and further refining the search space [23]. These are very promising steps forward in this space. It is the authors' hope that the here-presented observations and measurements can further aid in the construction of these network construction machines.

Finally there have been attempts at achieving deployment through alternative means. Operator reordering was shown to be able to save memory at inference [24]. The aforementioned MCUNet learned a memory-efficient inference engine [26]. For a good overview of the broader context in this space see Warden and Situnayake [36]. And for a strong development platform for DL on edge devices see TFLM [7].

10 Conclusion and Future Work

We discussed the benefits and difficulties of using microcontrollers as a target platform for applications powered by deep neural networks. We described ways to deploy and run a DNN and explained problems that primarily stem from the lack of memory and compute resources and a different memory hierarchy compared to server and mobile platforms. We believe that our work demonstrates the need for the development of new metrics tailored to MCU-type hardware, which we hope would facilitate the development of new neural network architectures and compression methods, which would optimise data movement in the context of the small fixed memory budgets found on MCUs.

Acknowledgements

This work was supported by the UK's Engineering and Physical Sciences Research Council (EPSRC) with the following grants: EP/S001530/1 (the MOA project) and EP/R018677/1 (the OPERA project). Additionally, it was supported by the European Research Council (ERC) via the REDIAL project (Grant Agreement ID: 805194).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. 2018. YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 1 (Jan 2018), 48–60. https://doi.org/10.1109/TCAD.2017.2682138

[3] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 517–532. https://proceedings.mlsys.org/paper/2021/file/a3c65c2974270fd093ee8a9bf8ae7d0b-Paper.pdf

[4] L. Cavigelli and L. Benini. 2017. Origami: A 803-GOp/s/W Convolutional Network Accelerator. *IEEE Transactions on Circuits and Systems for Video Technology* 27, 11 (Nov 2017), 2461–2475. https://doi.org/10.1109/TCSVT.2016.2592330

[5] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. 2015. Origami: A Convolutional Network Accelerator. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI* (Pittsburgh, Pennsylvania, USA) *(GLSVLSI '15)*. ACM, New York, NY, USA, 199–204. https://doi.org/10.1145/2742060.2743766

[6] François Chollet et al. 2015. Keras. https://keras.io.

[7] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. 2021. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 800–811. https://proceedings.mlsys.org/paper/2021/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[9] Daniel K Fisher and Hirut Kebede. 2010. A low-cost microcontroller-based system to monitor crop temperature and water status. *Computers and Electronics in Agriculture* 74, 1 (2010), 168–173.

[10] Song Han et al. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015). arXiv:1510.00149 http://arxiv.org/abs/1510.00149

[11] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/3020078.3021745

[12] Kaiming He et al. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

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

[14] Andrew P. Hill, Peter Prince, Evelyn Piña Covarrubias, C. Patrick Doncaster, Jake L. Snaddon, and Alex Rogers. 2018. AudioMoth: Evaluation of a smart open acoustic device for monitoring biodiversity and the environment. *Methods in Ecology and Evolution* 9, 5 (2018), 1199–1211. https://doi.org/10.1111/2041-210X.12955

[15] Fred H Holmes, Kevin Baxter, and Ken Fisher. 2010. Systems and methods of identifying/locating weapon fire using envelope detection. US Patent 7,710,278.

[16] M. Horowitz. 2014. 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)*. 10–14. https://doi.org/10.1109/ISSCC.2014.6757323

[17] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers* 40, 9 (September 1952), 1098–1101.

[18] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[19] 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 *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[20] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR* abs/1806.08342 (2018). arXiv:1806.08342 http://arxiv.org/abs/1806.08342

[21] Alex Krizhevsky et al. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 1097–1105.

[22] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *CoRR* abs/1801.06601 (2018). arXiv:1801.06601

[23] Edgar Liberis, ukasz Dudziak, and Nicholas D Lane. 2021. μNAS: Constrained Neural Architecture Search for Microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 70–79.

[24] Edgar Liberis and Nicholas D. Lane. 2019. Neural networks on microcontrollers: saving memory at inference via operator reordering. *CoRR* abs/1910.05110 (2019). arXiv:1910.05110 http://arxiv.org/abs/1910.05110

[25] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2015. Fixed Point Quantization of Deep Convolutional Networks. *CoRR* abs/1511.06393 (2015). arXiv:1511.06393

[26] Ji Lin, Wei-Ming Chen, Yujun Lin, john cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 11711–11722. https://proceedings.neurips.cc/paper/2020/file/86c51678350f656dcc7f490a43946ee5-Paper.pdf

[27] Christos Louizos, Karen Ullrich, and Max Welling. 2017. Bayesian Compression for Deep Learning. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 3288–3298.

[28] M. Rusci, D. Rossi, E. Farella, and L. Benini. 2017. A Sub-mW IoT-Endnode for Always-On Visual Monitoring and Smart Triggering. *IEEE Internet of Things Journal* 4, 5 (Oct 2017), 1284–1295. https://doi.org/10.1109/JIOT.2017.2731301

[29] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[30] M. Siekkinen, M. Hiienkari, J. K. Nurminen, and J. Nieminen. 2012. How low energy is bluetooth low energy? Comparative measurements with ZigBee/802.15.4. In *2012 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. 232–237. https://doi.org/10.1109/WCNCW.2012.6215496

[31] K. Siu, D. M. Stuart, M. Mahmoud, and A. Moshovos. 2018. Memory Requirements for Convolutional Neural Network Hardware Accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 111–121. https://doi.org/10.1109/IISWC.2018.8573527

[32] STMicroelectronics. 2019. STM32 High Performance MCUs. (2019).

[33] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *CoRR* abs/1703.09039 (2017). arXiv:1703.09039

[34] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2018. HAQ: Hardware-Aware Automated Quantization. *CoRR* abs/1811.08886 (2018). arXiv:1811.08886

[35] Yunhe Wang, Chang Xu, Shan You, Dacheng Tao, and Chao Xu. 2016. CNNpack: Packing Convolutional Neural Networks in the Frequency Domain. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 253–261.

[36] Pete Warden and Daniel Situnayake. 2019. *TinyML*. O'Reilly Media, Incorporated.

[37] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. 2017. Adaptive Quantization for Deep Neural Network. *CoRR* abs/1712.01048 (2017). arXiv:1712.01048

[38] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY* 23, 3 (1977), 337–343.