Enabling Deep Learning at the IoT Edge

(Invited Paper)

Liangzhen Lai
Arm Inc.
liangzhen.lai@arm.com

Naveen Suda
Arm Inc.
naveen.suda@arm.com

ABSTRACT

Deep learning algorithms have demonstrated super-human capabilities in many cognitive tasks, such as image classification and speech recognition. As a result, there is an increasing interest in deploying neural networks (NNs) on low-power processors found in always-on systems, such as those based on Arm Cortex-M microcontrollers. In this paper, we discuss the challenges of deploying neural networks on microcontrollers with limited memory, compute resources and power budgets. We introduce CMSIS-NN, a library of optimized software kernels to enable deployment of NNs on Cortex-M cores. We also present techniques for NN algorithm exploration to develop light-weight models suitable for resource constrained systems, using keyword spotting as an example.

1 INTRODUCTION

Connected devices or Internet of Things (IoT) have been rapidly proliferating over the past few years and are predicted to reach 1 trillion across various market segments by 2035 [15]. These IoT edge devices typically consist of sensors collecting data, including audio, video, temperature, humidity, GPS location and acceleration etc. Currently, most data collected from the sensors are processed by analytics tools in the cloud to enable a wide range of applications, such as industrial monitoring and control, home automation and health care. However, as the number of the IoT nodes increases, this places a considerable burden on the network bandwidth, as well as adds latency to the IoT applications. Furthermore, dependency on the cloud makes it challenging to deploy IoT applications in regions with limited or unreliable network connectivity. One solution to this problem is edge computing [14], performed right at the source of data, i.e. the IoT edge node, thus reducing latency as well as saving energy for data communication.

Neural network (NN) based solutions have demonstrated human-level accuracies for many complex machine learning applications such as image classification, speech recognition and natural language processing. Due to the computational complexity and resource requirements, the execution of NNs has predominantly been confined to cloud computing on high-performance server CPUs or specialized hardware (e.g. GPU or accelerators), which adds latency to the IoT applications. Classification right at the source of the data – usually small microcontrollers – can reduce the overall latency and energy consumption of data communication between the IoT edge and the cloud. However, deployment of NNs on microcontrollers comes with following challenges:

Limited memory footprint: Typical microcontroller systems have only tens to few hundred KB of memory available. The entire neural network model, including input/output, weights and activations, has to fit within this small memory budget.

Limited compute resources: Most classification tasks have always-on, and real-time requirement, which limits the total number of operations per neural network inference.

These challenges can be addressed from both the device and the algorithm perspectives. On one hand, we can improve the machine learning capabilities of these microcontrollers by optimizing the low-level computation kernels for better performance and smaller memory footprint when executing neural network workloads. This can enable the microcontrollers to handle larger and more complex NNs. On the other hand, NNs can be designed and optimized with respect to the targeting hardware platform by network architecture exploration. This can improve the quality (i.e., accuracy) of the NNs under a fixed memory and computation budgets.

In this paper, we first introduce CMSIS-NN [10] in Section 2. CMSIS-NN is a collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks on Arm Cortex-M processor cores targeted for intelligent IoT edge devices. Neural network inference based on CMSIS-NN kernels achieves 4.6X improvement in runtime/throughput and 4.9X improvement in energy efficiency. We also present techniques for NN architecture search [19] for keyword spotting application in Section 3. We perform resource-constrained NN architecture search and present comprehensive comparison of different network architectures within a set of compute and memory constraints of typical microcontrollers.

2 CMSIS-NN

The overview of CMSIS-NN neural network kernels is shown in Fig. 1. The kernel code consists of two parts: *NNFunctions* and *NNSupportFunctions*. *NNFunctions* include the functions that implement popular neural network layer types, such as convolution, depthwise separable convolution, fully-connected (i.e. inner-product), pooling and activation. These functions can be used by the application code to implement the neural network inference applications. The kernel APIs are also kept simple, so that they can be easily retargeted for any machine learning framework. *NNSupportFunctions* include utility functions, such as data conversion and activation function tables, which are used in *NNFunctions*. These utility functions can also be used by the application code to construct more complex NN modules, such as Long Short Term Memory (LSTM) or Gated Recurrent Unit (GRU).

For some kernels, such as fully-connected and convolution, different versions of the kernel functions are implemented. A basic

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '18, November 5–8, 2018, San Diego, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3243473

version is provided that works universally, 'as-is', for any layer parameters. We have also implemented other versions which include further optimization techniques with either transformed inputs or with some limitations on the layer parameters. Ideally, a simple script can be used to parse the network topology and automatically determine the appropriate functions to be used.

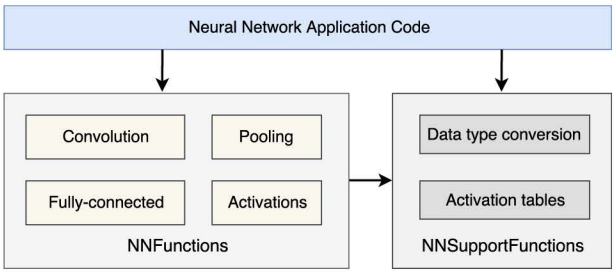

Figure 1: Overview of the neural network kernel structure.

2.1 Fixed-Point Quantization

Traditionally, NN models are trained using 32-bit floating point data representation. However, such high precision is generally not required during inference. Research has shown that NNs work well even with low-precision fixed-point representation [5, 9, 12]. Fixed-point quantization helps to avoid the costly floating-point computation and reduces the memory footprint for storing both weights and activations, which is critical for resource-constrained platforms. Although precision requirements for different networks or network layers can vary [16], it is hard for the CPU to operate on data types with varying bit-width. In this work, we develop the kernels that support both 8-bit and 16-bit data.

The kernels adopt the same data type format as used in CMSIS [2], i.e. $q7\_t$ as $int8$, $q15\_t$ as $int16$ and $q31\_t$ as $int32$. The quantization is performed assuming a fixed-point format with a power-of-two scaling. The quantization format is represented as Qm.n, where the represented value will be $A \times 2^{-1*n}$, where A is the integer value and n is part of Qm.n that represents the number of bits used for the fractional portion of the number, i.e., indicating the location of the radix point. We pass the scaling factors for the bias and outputs as parameters to the kernels and the scaling is implemented as bitwise shift operations because of the power-of-two scaling. We use this type of quantization – instead of the 8-bit quantization used in *TensorFlow* [5] – to avoid the need for floating-point de-quantization in between layers, as some Arm Cortex-M CPUs may not have a dedicated floating point unit (FPU), thus limiting their floating-point computation capabilities. The other benefit of such quantization is that we can use simpler table look-up based activation.

During the NN computation, the fixed-point representation for different data, i.e., inputs, weights, bias and outputs, can be different. The two input parameters, $bias\_shift$ and out_shift, are used to adjust the scaling of different data for the computation. The following equations can be used to calculate the shift values:

$$bias\_shift = n_{input} + n_{weight} - n_{bias} \quad (1)$$

$$out\_shift = n_{input} + n_{weight} - n_{output} \quad (2)$$

where n_{input}, n_{weight}, n_{bias} and n_{output} are the number of fractional bits in inputs, weights, bias and outputs, respectively.

2.2 Software Kernel Optimization

In this section, we highlight some of the optimizations implemented in CMSIS-NN for improving the performance and reducing the memory-footprint.

2.2.1 Matrix Multiplication. Matrix multiplication is the most important computation kernel in neural networks [1]. The implementation in this work is based on the *mat_mult* kernels in CMSIS. Similar to CMSIS implementation, the matrix multiplication kernel is implemented with 2×2 kernels, illustrated in Fig. 2. This enables some data reuse and saves on the total number of load instructions. The accumulation is done with the $q31\_t$ data type and both operands are of $q15\_t$ data type. We initialize the accumulator with the corresponding bias value. The computation is performed using the dedicated MAC instruction *__SMLAD*.

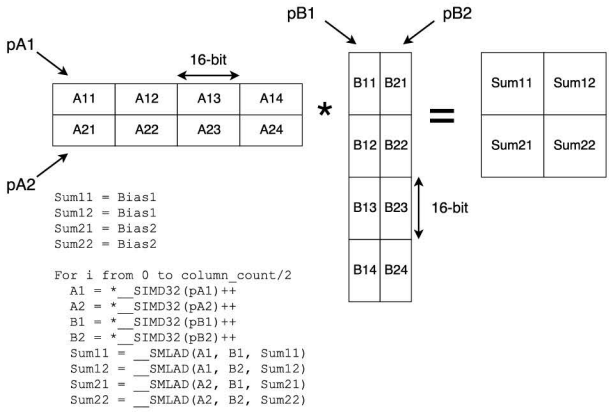

Figure 2: The inner-loop of matrix multiplication with 2×2 kernel. Each loop computes the dot product results of 2 columns and 2 rows, i.e. 4 outputs.

2.2.2 Convolution. A convolution layer extracts a new feature map by computing a dot product between filter weights and a small receptive field in the input feature map. Typically, a CPU-based implementation of convolution is decomposed into input reordering and expanding (i.e. *im2col*, image-to-column) and matrix multiplication operations. *im2col* is a process of transforming the image-like input into columns that represent the data required by each convolution filter. An example of *im2col* is shown in Fig.3.

One of the main challenges with *im2col* is the increased memory footprint, since the pixels in the input image are repeated in the *im2col* output matrix. To alleviate the memory footprint issue while retaining the performance benefits from *im2col*, we implemented a *partial im2col* for our convolution kernels. The kernel will only expand a limited number of columns (e.g. 2), sufficient to get the maximum performance boost from the matrix-multiplication kernels while keeping memory overhead minimal.

The image data format can also affect the performance of convolution, especially *im2col* efficiency [11]. With a batch size of one, the convolution operation is a 2D convolution (i.e. the convolution window can move in two directions) on 3D data, as shown in Fig. 4. The two most common image data formats are Channel-Width-Height (CHW), i.e. channel last, and Height-Width-Channel (HWC), i.e. channel first. The dimension ordering is the same as that of the

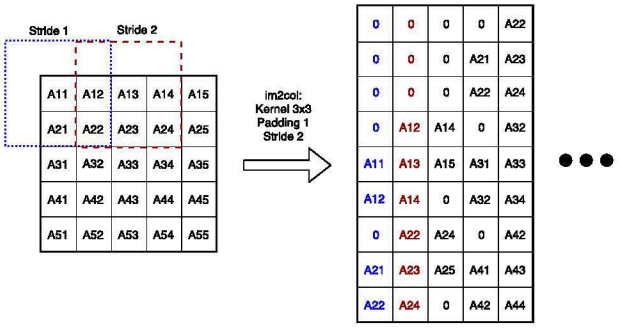

Figure 3: Example of *im2col* on a 2D image with a 3x3 kernel, padding size of 1 and stride size of 2.

data stride. In an HWC format, the data along the channel is stored with a stride of 1, data along the width is stored with a stride of the channel count, and data along the height is stored with a stride of (channel count × image width).

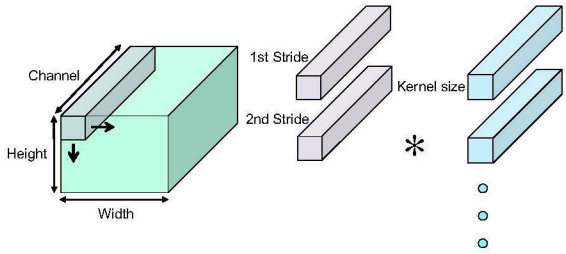

Figure 4: Convolution on 3D data. The image has three dimensions: height, width and channel.

The data layout has no impact on the matrix-multiplication operations, as long as the dimension order of both weights and images is the same. The *im2col* operations are performed along the width and height dimensions only. The HWC-style layout enables efficient data movement, as data for each pixel (i.e. at the same x,y location) is stored contiguously and can be copied efficiently with SIMD instructions. To validate this, we implemented both CHW and HWC versions and compared the runtime on an Arm Cortex-M7. The results are highlighted in Fig. 5, where we fixed the HWC input to be 16x16x16 and swept the number of output channels. When the output channel value is zero, it means that the software performs only *im2col* and no matrix-multiplication operation. Compared to CHW layout, HWC has less *im2col* runtime with the same matrix-multiplication performance. Therefore, we implement the convolution kernels assuming that the data layout is in HWC format.

2.3 CMSIS-NN Results

We tested the CMSIS-NN kernels on a CNN trained on the CIFAR-10 dataset, consisting of 60,000 32x32 color images divided into 10 output classes. The network topology is based on the built-in example provided in *Caffe*, with three convolution layers and one fully-connected layer. All the layer weights and activation data are quantized to *q7_t* format. The layer parameters and the detailed runtime results using the CMSIS-NN kernels are shown in the

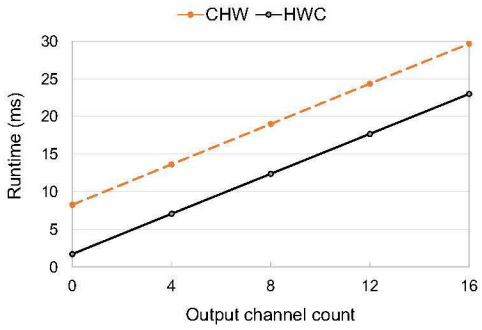

Figure 5: Experiment results with CHW and HWC data layout. Both data layout styles have the same matrix-multiplication runtime. HWC has less *im2col* runtime.

Table 1. The runtime is measured on a NUCLEO-F746ZG Mbed board [4] with an Arm Cortex-M7 core running at 216 MHz.

The entire image classification takes about 99.1 ms per image (the equivalent of 10.1 images per second). The compute throughput of the CPU is about 249 MOps per second for running this network. The pre-quantized network achieves an accuracy of 80.3% on the CIFAR-10 test set. The 8-bit quantized network running on Arm Cortex-M7 core achieves 79.9% accuracy. Maximum memory footprint using the CMSIS-NN kernels is ~133 KB, where convolutions are implemented with *partial im2col* to save memory, followed by matrix-multiplication. Memory footprint without partial *im2col* would be ~332 KB and the neural network would not fit on the board.

To quantify the benefits of CMSIS-NN kernels over existing solutions, we also implemented a baseline version using a 1D convolution function (*arm_conv* from CMSIS-DSP), *Caffe*-like pooling and ReLU. For the CNN application, Table 2 summarizes the comparison results of the baseline functions and the CMSIS-NN kernels. The CMSIS-NN kernels achieve 2.6X to 5.4X improvement in runtime/throughput over the baseline functions. The energy efficiency improvement is also in line with the throughput improvement.

Using keyword spotting application as an example, the next section presents the importance of finding the neural network architecture that is suitable for the constrained edge devices.

3 KEYWORD SPOTTING (KWS)

Keyword spotting is a critical component in initiating voice-based user interactions with smart devices at the edge. A typical KWS system consists of a feature extractor followed by a neural network based classifier as shown in Fig. 6. First, the input audio of length L is windowed into overlapping frames of length l with a stride s, giving a total of $T = \frac{L-l}{s} + 1$ frames. From each frame, F FFT-based speech features (e.g. MFCC or LFBE) are extracted, generating a total of $T \times F$ features. The extracted speech features are fed into a neural network based classifier, which computes the probabilities for the output classes. In a typical deployment, KWS inference is run continuously on the incoming audio stream and the output probabilities are averaged over a fixed time window to improve the overall prediction confidence.

Table 1: Layer parameters and performance for the CIFAR-10 CNN.

	Layer Type	Filter Shape	Output Shape	Ops	Runtime
Layer 1	Convolution	5x5x3x32 (2.3 KB)	32x32x32 (32 KB)	4.9 M	31.4 ms
Layer 2	Max Pooling	N.A.	16x16x32 (8 KB)	73.7 K	1.6 ms
Layer 3	Convolution	5x5x32x32 (25 KB)	16x16x32 (8 KB)	13.1 M	42.8 ms
Layer 4	Max Pooling	N.A.	8x8x32 (2 KB)	18.4 K	0.4 ms
Layer 5	Convolution	5x5x32x64 (50 KB)	8x8x64 (4 KB)	6.6 M	22.6 ms
Layer 6	Max Pooling	N.A.	4x4x64 (1 KB)	9.2 K	0.2 ms
Layer 7	Fully-connected	4x4x64x10 (10 KB)	10	20 K	0.1 ms
Total		87 KB weights	55 KB activations	24.7 M	99.1 ms

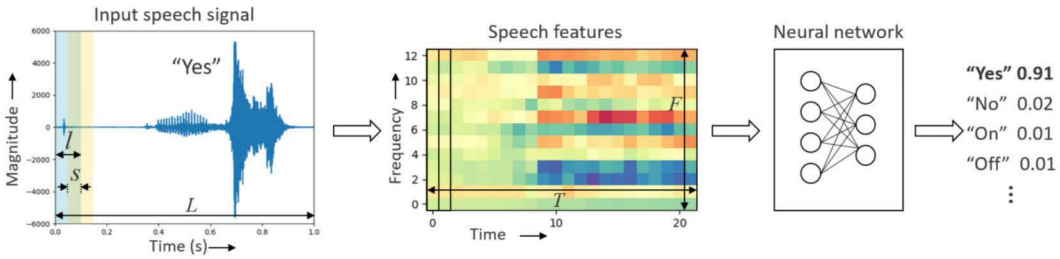

Figure 6: Keyword spotting pipeline.

Table 2: Throughput and energy efficiency improvements by layer types

Layer type	Baseline runtime	New kernel runtime	Improvement	
			Throughput	Energy Efficiency
Convolution	443.4 ms	96.4 ms	4.6X	4.9X
Pooling	11.83 ms	2.2 ms	5.4X	5.2X
ReLU	1.06 ms	0.4 ms	2.6X	2.6X
Total	456.4ms	99.1 ms	4.6X	4.9X

3.1 Neural Network Architectures for KWS

Several neural network architectures have been proposed for keyword spotting application such as deep neural networks (DNN) [7], convolutional neural networks (CNN) [13], recurrent neural networks (RNN) [17], convolutional recurrent neural networks (CRNN) [6] and. Depthwise separable convolutional neural networks (DS-CNN) [19]. DNNs consist of standard feed-forward neural networks with fully-connected layers separated by activation layers. CNN models utilize convolution layers to exploit the correlation between the time-domain and frequency-domain features by considering the extracted speech features as a 2-D image and performing convolution operations on them. RNN models with long short-term memory (LSTM) cells or gated recurrent units (GRU) further exploit the temporal correlation in the input features by capturing the long-term dependencies using "gating" mechanism. CRNNs, a hybrid of CNNs and RNNs,exploit the local temporal/spatial correlation using convolution layers and global temporal dependencies in the speech features using recurrent layers. Depthwise separable convolutional neural networks (DS-CNN), inspired from the efficient convolutions in Mobilenet architecture [8], decompose standard 3-D convolutions into 2-D convolutions (or depthwise separable convolutions) followed by 1-D convolutions (or pointwise convolutions), which allow deeper networks to fit on resource-constrained devices.

To compare the performance of neural network architectures from literature [6, 7, 13, 17], we trained them on the Google speech commands dataset [18]. Table 3 summarizes the accuracy, memory requirements and operations per inference for the network architectures for KWS from literature [6, 7, 13, 17]. The operations per inference is the number of total multiplications and additions in convolutions and fully-connected layers (including the recurrent layers), which acts as a good proxy for the neural network execution time. The memory requirement assumes 8-bit quantized weights and activations, which is generally sufficient to achieve the original model accuracy. Furthermore, we assume that the memory required for activations is reused across the neural network layers.

NN Architecture	Accuracy	Memory	Operations
DNN [7]	84.3%	288 KB	0.57 MOps
CNN-1 [13]	90.7%	556 KB	76.02 MOps
CNN-2 [13]	84.6%	149 KB	1.46 MOps
LSTM [17]	88.8%	26 KB	2.06 MOps
CRNN [6]	87.8%	298 KB	5.85 MOps

Table 3: Neural network model accuracy. CNN-1, CNN-2 are (*cnn-trad-fpool3, cnn-one-fstride4*) architectures from [13].

Table 3 shows that DNNs have very less number of operations but can be very memory-intensive. CNNs achieve good accuracy but have very large compute requirements. LSTMs/CRNNs strike a good balance between compute and memory resources without sacrificing much accuracy.

3.2 Hardware-Constrained NN Model Search

Microcontrollers typically consist of a processor core, SRAM based main memory and an embedded flash for storing the program binary, which is loaded onto memory at runtime. Table 4 shows some commercially available microcontroller development boards with Arm Cortex-M cores with different compute capabilities and memory.

The amount of memory (SRAM) in the microcontroller limits the size of the NN model that can be run on that system. Apart from memory footprint, performance (i.e., operations per second) can be another constraint for running neural networks on microcontrollers. Hence NN architectures need to be chosen considering the memory and compute constraints of the hardware on which the NN model will be deployed. In order the evaluate the neural network architectures with different constraints, we broadly derive three sets of memory/compute constraints based on the typical microcontroller system configurations. Table 5 shows these constraints, targeting small, medium and large microcontroller systems running KWS at 10 inferences per second.

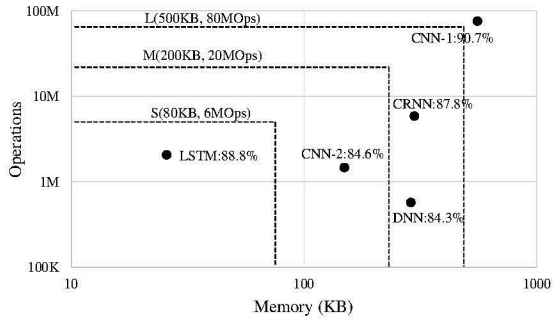

Figure 7: Number of operations vs. memory vs. test accuracy of NN models from prior work [6, 7, 13, 17] trained on the speech commands dataset [18].

Figure 7 shows the number of operations per inference, memory requirement and test accuracy shown in Table 3 overlaid with the hardware constraints in Table 5. From the figure, it is evident that all models apart from LSTM model do not fit into the small

Arm Mbed^{TM} platform	Processor	Frequency	SRAM	Flash
Mbed LPC11U24	Cortex-M0	48 MHz	8 KB	32 KB
Nordic nRF51-DK	Cortex-M0	16 MHz	32 KB	256 KB
Mbed LPC1768	Cortex-M3	96 MHz	32 KB	512 KB
Nucleo F103RB	Cortex-M3	72 MHz	20 KB	128 KB
Nucleo L476RG	Cortex-M4	80 MHz	128 KB	1 MB
Nucleo F411RE	Cortex-M4	100 MHz	128 KB	512 KB
FRDM-K64F	Cortex-M4	120 MHz	256 KB	1 MB
Nucleo F746ZG	Cortex M7	216 MHz	320 KB	1 MB

Table 4: Off-the-shelf Arm Cortex-M Mbed platforms.

NN size	NN memory limit	Ops/inference limit
Small (S)	80 KB	6 MOps
Medium (M)	200 KB	20 MOps
Large (L)	500 KB	80 MOps

Table 5: Hardware constraints assuming 10 inferences per second and 8-bit weights/activations.

(*S*) bounding box with 80KB/6MOps constraints. The model with highest accuracy (i.e. CNN-1) doesn't fit in any of the bounding boxes at all, whereas the other models fit within the bounds of M and L constraints. Hence a systematic search is needed to find the neural network architecture and model that fits within the hardware constraints and achieve high accuracy. In order to do that, one needs to understand the impact of the other components of the application pipeline on the model size and operations. For example, feature extraction parameters i.e. number of features per frame (F) and the frame stride (S) (Fig. 6) impact the model size, operations per inference and accuracy. Increasing the F increases the number of weights in DNN and RNN models, whereas decreasing the S decreases the total operations in the model. The aim of the NN architecture search is be to find a model within the memory/compute constraints that has maximum accuracy.

The NN architectures and the corresponding hyperparameters explored in this work are summarized in Table 6. The LSTM model mentioned in the table has peephole connections and an output projection layer to reduce the model size, similar to that in [17], whereas the basic LSTM model does not include those. CRNN uses a convolution layer followed by multi-layer GRU cells for the recurrent layers. We perform exhaustive search of feature extraction parameters and NN architectural hyperparameters to find the models that yield maximum accuracy within the constraints in Table 5. Figure 8 shows an example of hyperparameter search we performed on CRNN architecture. The final best performing models for each neural network architecture along with their memory requirements and operations are summarized in Table 7 and Fig. 9. The NN model definitions can be seen in [19]. The *TensorFlow* models and embedded code to deploy them on Arm Cortex-M cores using CMSIS-NN are open-sourced at [3].

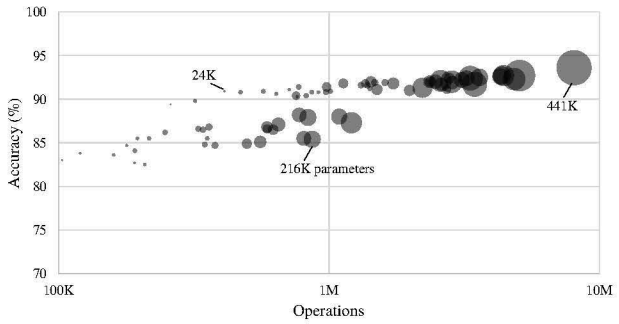

Figure 8: Hyperparameter search for CRNN architecture showing the model accuracy vs. operations, with the number of parameters shown as size of the circle.

From the results in Table 7 we can see that all the NN architectures scale up well but saturate at different accuracies. Most of the models are are limited by the memory constraint. DS-CNN architecture achieve higher accuracies compared to other NN models, because of their deeper architecture enabled by the efficient depthwise separable conovlutions.

4 CONCLUSION

Deep learning algorithms are moving to the IoT edge due to power, latency, cost, network bandwidth, reliability and privacy considerations. There is an increasing interest in deploying the deep learning

NN model	Model hyperparameters
DNN	Number of fully-connected (FC) layers and size of each FC layer
CNN	Number of Conv layers: features/kernel size/stride, linear layer dim., FC layer size
Basic LSTM	Number of memory cells
LSTM	Number of memory cells, projection layer size
GRU	Number of memory cells
CRNN	Conv features/kernel size/stride, Number of GRU and memory cells, FC layer size
DS-CNN	Number of DS-Conv layers, DS-Conv features/kernel size/stride

Table 6: Neural network architectural hyperparameters.

NN model	S(80KB, 6MOps)			M(200KB, 20MOps)			L(500KB, 80MOps)		
	Acc.	Mem.	Ops	Acc.	Mem.	Ops	Acc.	Mem.	Ops
DNN	84.6%	80.0KB	158.8K	86.4%	199.4KB	397.0K	86.7%	496.6KB	990.2K
CNN	91.6%	79.0KB	5.0M	92.2%	199.4KB	17.3M	92.7%	497.8KB	25.3M
Basic LSTM	92.0%	63.3KB	5.9M	93.0%	196.5KB	18.9M	93.4%	494.5KB	47.9M
LSTM	92.9%	79.5KB	3.9M	93.9%	198.6KB	19.2M	94.8%	498.8KB	48.4M
GRU	93.5%	78.8KB	3.8M	94.2%	200.0KB	19.2M	94.7%	499.7KB	48.4M
CRNN	94.0%	79.7KB	3.0M	94.4%	199.8KB	7.6M	95.0%	499.5KB	19.3M
DS-CNN	94.4%	38.6KB	5.4M	94.9%	189.2KB	19.8M	95.4%	497.6KB	56.9M

Table 7: Summary of best neural networks from the hyperparameter search.

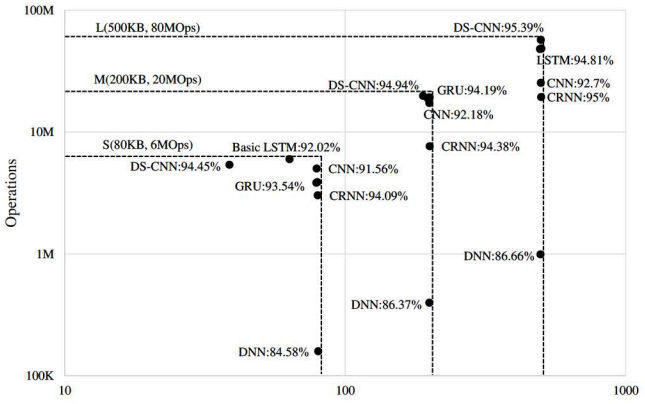

Figure 9: Memory vs. Ops/inference of the best models described in Table 7.

algorithms on low-power edge devices such as Arm Cortex-M microcontroller systems. To enable that, we presented CMSIS-NN, a library of optimized software kernels that maximize the performance of neural networks on Cortex-M cores with minimal memory footprint. We further presented techniques to perform NN architecture search within the memory/compute constraints of typical microcontrollers on a keyword spotting application. The CMSIS-NN library and keyword spotting models are open-sourced at [2, 3].

REFERENCES

[1] [n. d.]. https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/. ([n. d.]).

[2] [n. d.]. CMSIS_5. https://github.com/ARM-software/CMSIS_5. ([n. d.]).

[3] [n. d.]. ML-KWS-for-MCU. https://github.com/ARM-software/ML-KWS-for-MCU. ([n. d.]).

[4] [n. d.]. NUCLEO-F746ZG development board. http://www.st.com/en/evaluation-tools/nucleo-f746zg.html. ([n. d.]).

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[6] Sercan O Arik, Markus Kliegl, Rewon Child, Joel Hestness, Andrew Gibiansky, Chris Fougner, Ryan Prenger, and Adam Coates. 2017. Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting. *arXiv preprint arXiv:1703.05390* (2017).

[7] Guoguo Chen, Carolina Parada, and Georg Heigold. 2014. Small-footprint keyword spotting using deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 4087–4091.

[8] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[9] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2017. Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations. *arXiv preprint arXiv:1703.03073* (2017).

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

[11] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 633–644.

[12] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*. 2849–2858.

[13] Tara N Sainath and Carolina Parada. 2015. Convolutional neural networks for small-footprint keyword spotting. In *Sixteenth Annual Conference of the International Speech Communication Association*.

[14] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[15] Philip Sparks. [n. d.]. The route to a trillion devices. https://community.arm.com/iot/b/blog/posts/white-paper-the-route-to-a-trillion-devices. ([n. d.]).

[16] Naveen Suda et al. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 16–25.

[17] Ming Sun, Anirudh Raju, George Tucker, Sankaran Panchapagesan, Gengshen Fu, Arindam Mandal, Spyros Matsoukas, Nikko Strom, and Shiv Vitaladevuni. 2016. Max-pooling loss training of long short-term memory networks for small-footprint keyword spotting. In *Spoken Language Technology Workshop (SLT), 2016 IEEE*. IEEE, 474–480.

[18] Pete Warden. 2017. Speech Commands: A public dataset for single-word speech recognition. *Dataset available from http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz* (2017).

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