# Supervised Machine Learning for Ultra-Low Power Systems

Ingy ElSayed-Aly, under the supervision of Prof. Campbell

November 12, 2018

## 1 Introduction

Machine learning has become a very powerful and versatile computing tool, enabling all sorts of novel ways of gaining insight on collected data. Capable of adapting and learning from sensed observations, it provides new ways of managing the complexities of the real world. From more accurate search engines and personal assistants to even autonomous cars, Machine learning is progressively becoming omnipresent in our lives. Many machine learning applications can still find their way in our lives through embedded devices, providing more robust systems and personal devices.

However, most applications choose to move the processing and data to the Cloud or require expensive hardware. Sending all the data collected to the Cloud is source of concern for security and privacy. Although training the model will continue to be computationally expensive, after training, there are several techniques to compress supervised learning models to drastically reduce computation and memory costs [1].

There are many reasons using supervised learning on truly low power devices can be useful. First, for latency and autonomy, now instead of always being connected to the Internet and requiring a stable connection, the device can run operations offline. The ability to run offline will make the system more portable and more robust. Second, the reduced use of the Internet can in turn protect the device from malware and botnet viruses such as Mirai [2]. Moreover, the user can choose to control their privacy better with the option of controlling what data is uploaded to the Cloud. Furthermore, such devices could potentially benefit from the energy harvesting becoming battery-less or last much longer on batteries.

## 2 Problem Statement

The aim of this project is to deploy a Convolutional Neural Network (CNN) inference model for sign language recognition to a Cortex-M processor. This project will detail the steps for the deployment of a CNN from fully trainable model to a quantized inference-only model compatible with the ARM CMSIS Library.

First, we will test the ARM library's machine learning components on a Cortex-M4 processor. Once a solid development foundation has been developed, we will prove that it is possible to make use of ultra-low power devices in the context of machine learning applications for more than proof of concept type problems. This will be done by training a small supervised learning model to detect the number shown by a hand using the sign language convention. This way, we demonstrate that it is possible to obtain reasonable accuracy and efficiency for a reasonable problem on an ultra-low power board. Then, we will deploy this model to our ultra-low power board and analyze performance and accuracy.

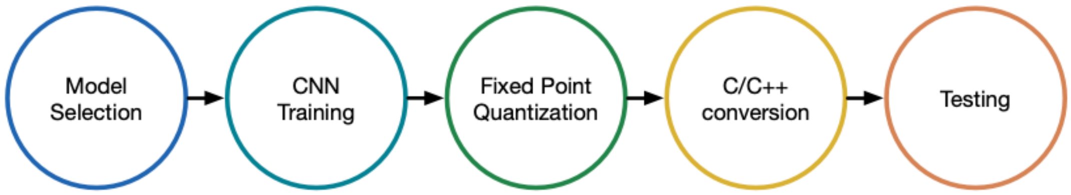


Figure 1: Deployment Workflow to an Edge Device

## 3 Related Work

After a survey of the previous research in the area, we have found that most of the research mentioning machine learning on low power devices in reality use advanced and expensive platforms such as the NVidia Jetson series which include a Cortex-A series processor and even an onboard GPU for acceleration [3, 4]. For reference, one Jetson TX1 costs around 500$. Although the Jetson series has a low power mode and provides a large processing power with some mobility, it can hardly be considered a low power device when running machine learning related tasks with an average consumption of 10W with the GPU running for the TK1 [5]. For reference, the ARM Cortex M4 can consume around 15 mW with the CPU running [6].

However, there is one proof of concept type paper demonstrating a Convolutional Neural Network (CNN) application to solve the CIFAR-10 image classification problem on an ARM Cortex-M7 processor which are cheaper and ultra low power [7]. It is done using an ARM-provided library providing basic support for neural networks CMSIS [7]. The board used for this demonstration is a NUCLEO-F746ZG board which costs around 25$ on DigiKey [8]. Our goal is to prove that it can not only be done in an energy efficient manner on a Cortex-M series but also that it can provide new and practical applications with supervised learning capabilities.

## 4 A Workflow for CNN deployment

This section of the project will detail the different steps from a full fledged Tensorflow CNN model to a quantized model for edge device inference. These steps include Model Selection, CNN Training, Fixed-Point Quantization and Conversion to C/C++ (Figure 1).

### 4.1 Dataset

For the purpose of this project, we chose to select a sign language dataset for the numbers 0 through 9 based on $100 \times 100$ colored images [9]. Each image was resized to a $45 \times 45$ grayscale image in order to take up less memory. After balancing the dataset, it contains 205 samples for each class, and we select 15% of these images for testing. The clarity of the image was an important factor in our decision for size. It is our belief that some CIFAR10 images are too small to clearly identify with the human eye. A sample of the dataset can be seen in Figure 2, the color of the image is a visualization effect to represent the grayscale number.

### 4.2 Model Architecture and Training

For this project, two different models were trained based on two image input sizes. Initially, our model took a $64 \times 64$ grayscale image as input. However, after converting to C/C++ code compatible with the device, it appeared that the CMSIS function accepted a limited number of parameters for the activation functions (less than $65,536$, which corresponds to a 16bit

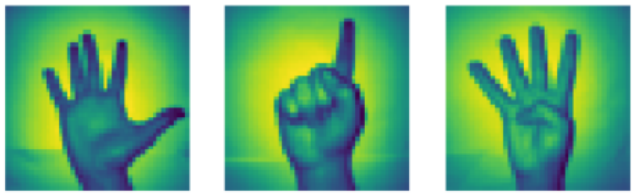


Figure 2: Image samples from the converted dataset

| Layer | Type | Kernel Shape | Output Shape |
|---|---|---|---|
| Input | - | - | $45 \times 45 \times 1$ |
| Layer 1 | Convolution | $5 \times 5 \times 32$ | $45 \times 45 \times 32$ |
| Layer 2 | Max Pooling | $2 \times 2$ | $23 \times 23 \times 32$ |
| Layer 3 | Convolution | $5 \times 5 \times 32$ | $23 \times 23 \times 32$ |
| Layer 4 | Max Pooling | $2 \times 2$ | $12 \times 12 \times 32$ |
| Layer 5 | Convolution | $5 \times 5 \times 16$ | $12 \times 12 \times 16$ |
| Layer 6 | Max Pooling | $2 \times 2$ | $6 \times 6 \times 16$ |
| Layer 7 | Dense | $6 \times 6 \times 16 \times 10$ | 10 |

Table 1: Model Architecture

unsigned integer). Therefore, only the second more successful network architecture with a $45 \times 45$ input image will be discussed (the original model can be found as an appendix).

For this CNN, we used only ReLu activation functions after each convolution and a Softmax activation for the Dense layer in which each neuron represents the class probability (10 output neurons). The 5x5 kernel for convolution enables us to efficiently extract features without reducing the dimensions and the 2x2 max pooling kernels reduce the dimensions by half while keeping the channels intact. Moreover, in the training phase, we added dropouts of 25% after each pooling layer to limit overfitting. After 193 epochs of training, we achieve a very reasonable validation accuracy of 92.5% over the balanced dataset with 10 classes.

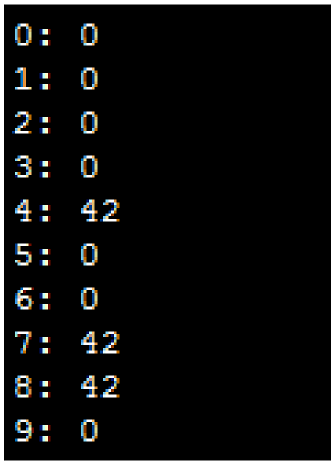
### 4.3 Fixed Point Quantization

In order to achieve high accuracy while training the CNN model weights are all stored as 32 or 64 bit floating point representation. All computations during the training involving weights are done with computationally expensive Floating Point Unit (FPU) operations. However, not all Cortex-M processors have an FPU. Moreover, floating point precision may not be needed for inference and CMSIS Library supports fixed-point quantized weights only [7, 10].

Fixed point quantization refers to the representation of a floating point value using two integers. In this case, we use power-of-two scaling, for example: $200.305 \approx 25 \times 2^3$. Then 200.305 which would be stored as a 32 or 64 bit float can now be stored in as little as two 8 bit integers in which the 3 is the scaling factor. Furthermore, power of two scaling is useful for simplifying computation, especially for activations [7]. In order to perform the quantization faster, we implemented a Python script to read the model weights and save them in the form of a header file compatible with the CMSIS NN component with quantized weights.

### 4.4 Conversion to C/C++

After generating the quantized weights, we needed to store input images and all layer parameters in header files, which was done using a Python script. Finally, the code for the inference was written and the Makefile modified to reflect the new changes. For this project, we used RTT for debugging. After debugging the code for the Sign Language Recognition



```
0: 0
1: 0
2: 0
3: 0
4: 42
5: 0
6: 0
7: 42
8: 42
9: 0
```

Figure 3: RTT Viewer output

CNN compiles and runs succesfully on the nrf52 development board. The output of the code is represented in figure 3, each line indicates a class (0-9) and the probability of it being that class. Total time for inference for one image is less than a second and the total size of the program including test image, weights, parameters and code is 89KB.

## 5 Limitations

Because of the novelty of the CMSIS NN component and our use of the GNU-embedded-tools instead of an IDE like Keil, it was not always obvious how ARM researchers came to certain parameters or how they converted Caffe or Tensorflow weights and parameters to their C/C++ code. Although the code for our CNN runs succesfully, the classification on the board is incorrect. This could be due to two factors, our manual quantization method is not the same as what the CMSIS code expect, ours is based only on right-shift, they may be expecting left shifted quantization for bias and right shifted quantization for kernel weights. Moreover, it could be due to an error in weight ordering, the ARM paper describes weight interleaving but it is unclear whether it applies to the preprocessing and conversion to C/C++ or if it is done automatically by the CMSIS library.

Finally, the use of 16 bit unsigned integers to store the number of parameters even if the board is capable of storing that many parameters seems like a general limitation of the CMSIS library. This limitation greatly reduces the possible size of inputs. Moreover, even using fixed-point quantization, some information from the weights is loss when migrating to the inference only model.

## 6 Conclusion

In summary, we were successful in deploying a different and a viable CNN to the nrf52 Ultra-Low Power development board for inference with the support of the CMSIS library. The CNN model we developed solves a reasonable real world applications problem. Moreover, we have demonstrated that inference models can be used on a much smaller scale than previously thought (Cortex-M4). Although, the weights conversion still needs some adjustment to correct the classification, the program runs in very little time with a very small memory footprint of 89KB. It is our hope that with this demonstration and the CMSIS library, more research will be directed in this area.

## References

[1] F. N. Iandola and K. Keutzer, “Keynote: Small neural nets are beautiful: Enabling embedded systems with small deep-neural-network architectures,” *CoRR*, vol. abs/1710.02759, 2017. [Online]. Available: http://arxiv.org/abs/1710.02759

[2] “Krebsonsecurity,” https://krebsonsecurity.com/tag/mirai-botnet/, accessed: 2018-09-20.

[3] M. Malik and H. Homayoun, “Big data on low power cores: Are low power embedded processors a good fit for the big data workloads?” in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, Oct 2015, pp. 379–382.

[4] B. Wu, F. N. Iandola, P. H. Jin, and K. Keutzer, “Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving,” *CoRR*, vol. abs/1612.01051, 2016. [Online]. Available: http://arxiv.org/abs/1612.01051

[5] “Jetson/graphics performance,” https://elinux.org/Jetson/Graphics_PerformancePower_Use_-_Overview, accessed: 2018-09-20.

[6] “nRF52840 board specification,” http://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.0.pdf, accessed: 2018-09-20.

[7] N. S. Liangzhen Lai and V. Chandra, “Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,” Jan 2018. [Online]. Available: https://arxiv.org/pdf/1801.06601.pdf

[8] “NUCLEO-F746ZG digi-key overview page,” https://www.digikey.com/product-detail/en/stmicroelectronics/NUCLEO-F746ZG/497-16282-ND/5806779, accessed: 2018-09-20.

[9] “Sign language digits dataset,” https://www.kaggle.com/ardamavi/sign-language-digits-dataset, accessed: 2018-09-30.

[10] L. Lai, N. Suda, and V. Chandra, “Deep convolutional neural network inference with floating-point weights and fixed-point activations,” *CoRR*, vol. abs/1703.03073, 2017. [Online]. Available: http://arxiv.org/abs/1703.03073

| Layer | Type | Kernel Shape | Output Shape |
|---|---|---|---|
| Input | - | - | $64 \times 64 \times 1$ |
| Layer 1 | Convolution | $5 \times 5 \times 32$ | $64 \times 64 \times 32$ |
| Layer 2 | Max Pooling | $2 \times 2$ | $32 \times 32 \times 32$ |
| Layer 3 | Convolution | $5 \times 5 \times 32$ | $32 \times 32 \times 32$ |
| Layer 4 | Max Pooling | $2 \times 2$ | $16 \times 16 \times 32$ |
| Layer 5 | Convolution | $5 \times 5 \times 16$ | $16 \times 16 \times 16$ |
| Layer 6 | Max Pooling | $2 \times 2$ | $8 \times 8 \times 16$ |
| Layer 7 | Dense | $8 \times 8 \times 16 \times 128$ | 300 |
| Layer 8 | Dense | 300 | 10 |

Table 2: Model Architecture 1

## A Original Model Architecture

This table summarizes the original model architecture attempted with a larger image input.