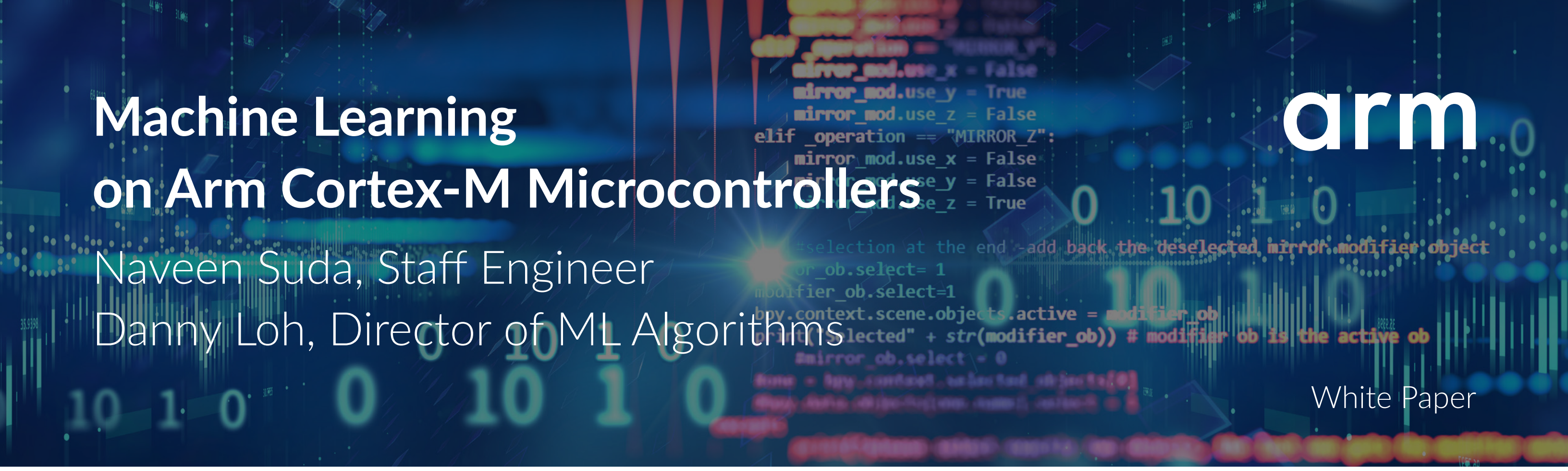


Machine Learning on Arm Cortex-M Microcontrollers

Naveen Suda, Staff Engineer
Danny Loh, Director of ML Algorithms

White Paper

Machine learning (ML) algorithms are moving to the IoT edge due to various considerations such as latency, power consumption, cost, network bandwidth, reliability, privacy and security. Hence, there is an increasing interest in developing neural network (NN) solutions to deploy them on low-power edge devices such as the Arm Cortex-M microcontroller systems. To enable that, we present CMSIS-NN, an open-source library of optimized software kernels that maximize the NN performance on Cortex-M cores with minimal memory footprint overhead. We further present methods for NN architecture exploration, using image classification on CIFAR-10 dataset as an example, to develop models that fit on such constrained devices.

I. 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 [1]. These IoT edge devices typically consist of various sensors collecting data, including audio, video, temperature, humidity, GPS location and acceleration etc. Typically, 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, 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 10's-100's of KB memory available. The entire neural network model, including input/output, weights and activations, has to fit and run within this small memory budget.
- **Limited compute resources:** Many 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 NN architecture exploration. This can improve the quality (i.e., accuracy) of the NNs under fixed memory and computation budgets.

In this paper, we present CMSIS-NN [2] in Section II. 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. In Section III, we present techniques for searching the neural network architectures for the microcontroller memory/compute constraints using image classification application on CIFAR-10 dataset as an example.

II. 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, depth wise 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 intentionally kept simple, so that they can be easily retargeted for any machine learning frameworks such as TensorFlow, Caffe or PyTorch. NNSupportFunctions include utility functions, such as data conversion and activation function tables, which are used in NNFunctions. These 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) cells.

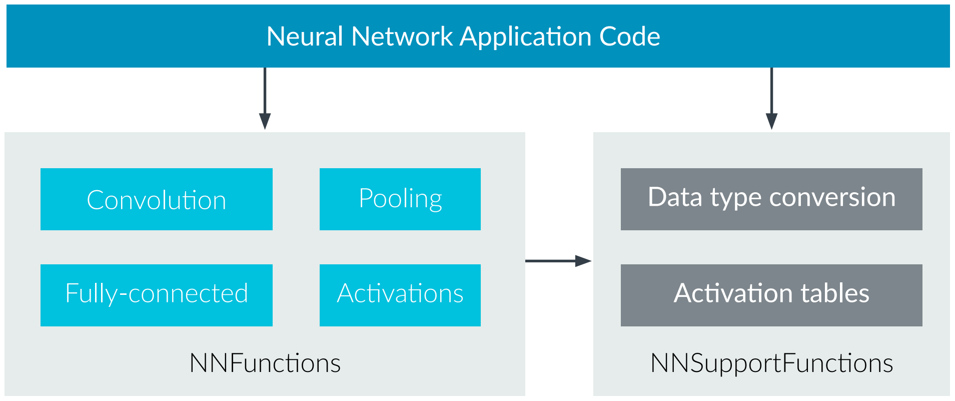


Fig 1: Overview of CMSIS-NN neural network kernels.

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

A. Fixed-Point Quantization

Research has shown that NNs work well even with low-precision fixed-point representation [3]. 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 [4], 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-DSP, 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 $A \times 2^{-n}$, 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.

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:

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

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

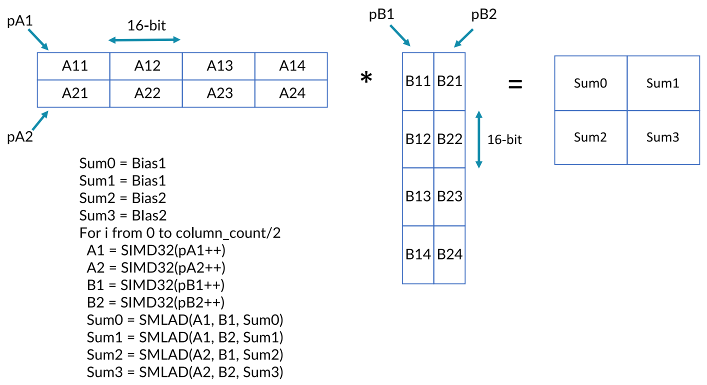
where n_{input}, n_{weight}, n_{bias} *and* n_{output} are the number of fractional bits in inputs, weights, bias and outputs, respectively.

B. 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.

1) *Matrix Multiplication*: Matrix multiplication is the most important computation kernel in neural networks. The implementation in this work is based on the *mat_mult* kernels in CMSIS-DSP. 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 SIMD MAC instruction_SMLAD.

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.



```
Sum0 = Bias1
Sum1 = Bias1
Sum2 = Bias2
Sum3 = BIas2
For i from 0 to column_count/2
  A1 = SIMD32(pA1++)
  A2 = SIMD32(pA2++)
  B1 = SIMD32(pB1++)
  B2 = SIMD32(pB2++)
  Sum0 = SMLAD(A1, B1, Sum0)
  Sum1 = SMLAD(A1, B2, Sum1)
  Sum2 = SMLAD(A2, B1, Sum2)
  Sum3 = SMLAD(A2, B2, Sum3)
```

Fig 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. generate 4 outputs.

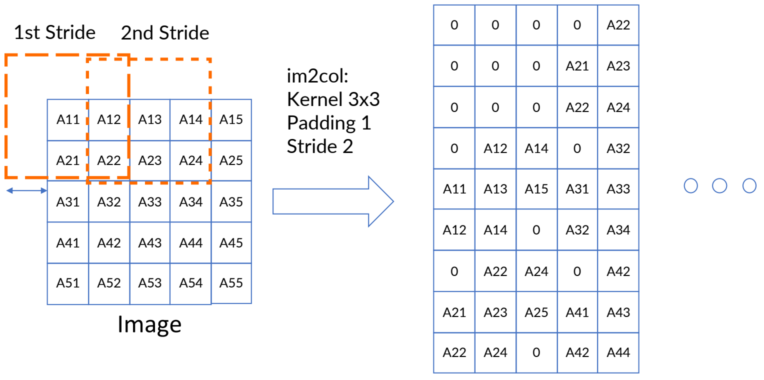


Fig 3: Example of im2col on a 2D image with 3x3 kernel, padding of 1 and stride of 2.

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 expand only 2 columns at a time, 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. 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 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).

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. 4, 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 with HWC data layout.

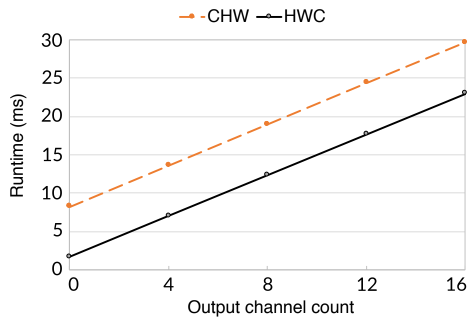


Fig 4: Comparison of convolution execution time with CHW vs. HWC data layouts. Both layouts have the same matrix-multiplication runtime, but HWC has less im2col runtime.

C. 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 runtime is measured on a STMicroelectronics NUCLEO-F746ZG Mbed board with an Arm Cortex-M7 core running at 216 MHz.

Table I: Throughput and energy efficiency improvements.

Layer type	**Baseline runtime**	**CMSIS-NN runtime**	**SRAM Flash**	
			Throughput	**Energy efficiency**
Convolution	443.4 ms	96.4 ms	4.6 X	4.9 X
Pooling	11.83 ms	2.2 ms	5.4 X	5.2 X
ReLU	1.06 ms	0.4 ms	2.6 X	2.6 X
Total	456.4 ms	99.1 ms	4.6 X	4.9 X

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 I 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.

III. Hardware Constrained NN Models

In this section, we use image classification application as an example to highlight the importance of choosing the right neural network architecture for the hardware platform (i.e., a microcontroller) on which the application will be deployed on. In order to do that, we need to understand the hardware constraints of microcontrollers. Microcontrollers typically consist of processor core, an SRAM which acts as a main memory and an embedded flash for storing the code and data. Table-II shows some commercially available microcontroller development boards with Arm Cortex-M cores with different compute and memory capacities.

The amount of memory in the microcontroller system limits the size of the NN model that can be run on the system. Apart from memory constraints, large compute requirements of neural networks pose another critical constraint for running NNs on microcontrollers, as they typically run at low frequencies for low power consumption. Hence NN architectures must be chosen to match the memory and compute constraints of the hardware on which the NN model will be deployed. In order to evaluate the accuracy of the neural networks with different hardware constraints, we choose three system configurations of different sizes and derive the neural network requirements for each configuration as shown in Table-III. Here, we assume a nominal 10 image classification inferences per second (i.e., 10 fps) to derive the NN requirements.

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

Arm Mbed™ Platform	Processor	Frequency	SRAM	Flash
LPC11U24	Cortex-M0	48 MHz	8 KB	32 KB
nRF51-DK	Cortex-M0	16 MHz	32 KB	256 KB
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	1MB
Nucleo F411RE	Cortex-M4	100 MHz	128 KB	512 KB
FRDM-K64F	Cortex-M4	120 MHz	256 KB	1MB
Nucleo F746ZG	Cortex-M7	216 MHz	320 KB	1MB

Table III: Constraints for Neural Networks.

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

A. NN Architectures for Image Classification

1) Convolutional Neural Networks (CNN): CNNs are the most popular neural network architectures that are used in computer vision applications. CNNs consist of multiple convolutional layers, interspersed by normalization, pooling and non-linear activation layers. The convolution layers decompose the input image to different feature maps varying from low-level features such as edges, lines, curves in the intial layers to high-level/ abstract features in the later layers. State of the art CNNs consist of 100's-1000's of these convolutional layers and the final extracted features are classified to the output classes by a fully-connected classification layers. Convolution operation is the most critical operation in CNNs and are very time-consuming with more than 90% time spent in convolutional layers.

2) Recent Efficient NN architectures: To reduce the compute complexity of the CNNs, depthwise separable convolution layer have been proposed in [5] as an efficient alternative to the standard convolution operation. By replacing a standard 3-D convolution with a 2-D depthwise convolution followed by a 1-D pointwise convolution, an efficient class of NN called as MobileNets are proposed in [6]. ShuffleNets [7] utilize depthwise convolutions on shuffled channels along with groupwise 1x1 convolutions to improve the accuracy with compact models. MobileNets-V2 [8] further improved the efficiency by adding shortcut connections, which help in convergence in deep networks. Overall, there have been many efficient neural network architectures proposed, which can be leveraged when developing a NN model specific for our hardware budget.

B. Hardware constrained NN Model Search

We use MobileNet architecture with shortcut connections similar to those in ResNet model for the hardware constrained neural model search. The number of layers, number of features in each layer, the convolution filter dimensions, stride are sued as the hyperparameters the search. Training all combinations of these hyperparameters would be time-consuming and not practical. So, we iteratively perform exhaustive search of the hyperparameters, compute the memory/compute requirements of the models and train only those models which fit within the hardware budgets. This is followed by a selection of hyperparameters from the previous pool to narrow down the search space to continue the next iteration of model search. Fig. 5 shows an example of hyperparameter search, which shows the accuracy, number of operations and parameters of each model.

After a few iterations, the models with the highest accuracy within the hardware budgets are shown in Table IV. Note that since this is not an exhaustive search through all the hyperparameters, there may be some neural network models with higher accuracy within our hardware constraints that aren't captured during our search space exploration.

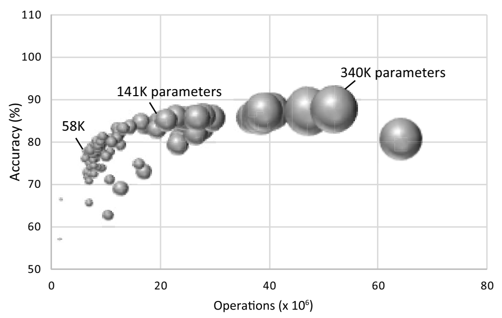


Fig 5: Neural network hyperparameter search for image classification on CIFAR-10 dataset, showing accuracy vs. operations with the number of parameters shown as size of the bubble.

Table IV: Summary of best neural networks from hyperparameter search.

NN model	Accuracy	Memory	Operations
S (80 KB, 6 MOps)	77.8%	58 KB	5.8 MOps
M (200 KB, 20 MOps)	84.7 %	141 KB	19.8 MOps
L (500 KB, 80 MOps)	87.7 %	340 KB	51.9 MOps

The results show that the models scale up well and accuracy saturates at different levels for the different hardware budgets. For example, for the memory/compute budget of 200KB, 20MOps, the model accuracy saturates around ~85% and is bound by the compute capability of the hardware. Understanding whether the neural network accuracy is bound by compute or memory resources provides key insights into different tradeoffs in the hardware platform selection.

IV. Conclusion

Machine learning algorithms are proven to solve some of the complex cognitive tasks demonstrating human-level performance. These algorithms are slowly moving to the IoT edge aided by the new efficient neural network architectures and optimized NN software to enable neural networks to run efficiently on the edge devices. We presented techniques to perform NN model search within a set of memory/compute constraints of typical microcontroller devices, using image classification as an example. Further we presented methods used in optimizing the NN kernels in CMSIS-NN to maximize the performance of neural networks on Cortex-M cores with minimal memory footprint. The CMSIS-NN library is available at https://github.com/ARM-software/CMSIS_5.

References

[1] Philip Sparks. "The route to a trillion devices," – available online: https://community.arm.com/iot/b/blog/posts/whitepaper-the-route-to-a-trillion-devices.

[2] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for Arm Cortex-M CPUs," arXiv preprint arXiv:1801.06601.

[3] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," In International Conference on Machine Learning (ICML) 2016, pp. 2849–2858.

[4] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," In ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) 2018, pp. 764-775.

[5] François Chollet, "Xception: Deep learning with depthwise separable convolutions," arXiv preprint arXiv:1610.02357, 2016.

[6] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," arXiv preprint arXiv:1704.04861, 2017.

[7] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," arXiv preprint arXiv:1707.01083, 2017.

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