QnA: Analysis of Compressing and Deploying the Neural Network Models onto MCU

Qianxi Li*
qianxi@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

Abdullah Al Omar*
aomar3@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

1 INTRODUCTION

The problem we would like to work on is compressing and deploying a MobileNetV2 model to perform the Visual Wake Word (VWW) task on both a microcontroller and a computer. We will compare latency, energy usage, and accuracy between models on different devices. Recently, MobileNet [11], a family of deep neural networks, has drawn a wealth of attention and exhibited substantial improvements in the theoretical and application levels. The first version of MobileNet performed on par with several state-of-the-art approaches with a small model size on image classification tasks in 2017 [16]. Since then, due to its small model size and neat performance, several downstream studies chose MobileNet to infer on mobile and IoT devices. However, there is still a myriad of differences between different IoT devices. One type of IoT device, a Microcontroller Unit (MCU), does not have an operating system and is usually equipped with tiny SRAM and limited computation power. Therefore, making inferences on MCUs is a challenging task. By investigating the compression of the machine learning model, we can extend the capabilities of deep learning on a wide range of tasks from computers to tiny devices, thus leading to the widespread adoption of machine learning in the real world.

2 RELATED WORK

The increasing interest in using microcontrollers for machine learning has led to designing more powerful MCUs in the case of running neural network models. Several works in the literature use different types of MCUs like FRDM-K64F, FS32K142WAT0WLFT, etc.). The flash memory and static random access memory (SRAM) of MCUs are two necessary factors in the case of running models on them [3]. Different use cases have several memory requirements. We will use Arduino Nano 33 BLE [2] as discussed in [7], VWW requires 250 KB SRAM nano has it. Again, this MCU is compatible with tensor flow lite. Hence, Arduino Nano 33 BLE is one of the suitable MCUs for our project.

In the VWW paper [7], along with the published VWW dataset, they also compare the performance of variants of MobileNet, such as MobileNet v1 [16], MobileNet v2 [21], and several other models using both the ImageNet [8] and the VWW dataset on MCUs. However, they do not incorporate the MobileNet, which involves transfer learning, which means using the learned network representations to transfer knowledge from one domain to other similar domains. This is a standard training methodology for machine learning research since training a deep learning model from stretch usually requires countless CPU and GPU cycles; in such cases, one natural thought is to train the model on top of the existing baseline models to speed up training and leverage good feature representations. To tackle this issue, we use a pre-trained MobileNet model trained on the ImageNet and transfer the knowledge from a multi-label classification task to a binary classification task - check the presence of a human in an image. Moreover, to our knowledge, the comparison of MobileNet's energy consumption on MCUs and off-the-shelf computers has not been systematically studied.

Researchers have proposed a wealth of techniques to shrink the size of a deep learning model and satisfy tight hardware constraints. We leave the model compression overview part in the appendix.

3 RESOURCES

3.1 Dataset

We use the Visual Wake Words (VWW) [7], an emerging vision dataset used as a benchmark task on IoT devices. It contains 115k and 8k images for training and validating purposes; correspondingly, each image can be classified as either "human presence" or the inverse, and the boundary boxes of presented people are also annotated. We take advantage of, this dataset to train and validate our deep-learning model on the computer and HPC platforms such as Compute Canada.

The original VWW dataset comes in two parts: (1) the COCO dataset, which contains images without labels (2) .json files which map different images in the COCO dataset into one of the two classes we want to classify.

Since (1) the task of detecting humans' presence is relatively simpler compared to the multi-class classification task, (2) the validation set is unbalanced with the majority of

*Both the authors contributed equally to this report.

images being labeled as "human presence", and (3) for the main deep learning model MobileNetV2 we optimize for, we use pre-trained ImageNet weights to make the training faster and get more accurate results, we only use a small portion of the VWW dataset, which is 20000 images for the training set, 4000 images for the validation set, and 1000 images for the test set, all of those images are split from the large training set. We shuffle all the images before the split to ensure each dataset is representative. For each of the datasets, we also need to specify different image sizes for the model to process, and the datasets are loaded through "tf.keras.utils.image_dataset_from_directory", every image in the pipeline will invoke "tf.image.resize" with default aspect ratio and resizing method. In each dataset, exactly half of the images are labelled as 1, which means "human presence", the rest are labelled as 0, which means "no human presence".

3.2 Hardware

We use Arduino nano 33 BLE [2] to run the compressed version of our model on it. It is one of the updates of Arduino nano based on AtMega 328P. In this upgraded version an updated CPU is used, ARM® Cortex®-M4 running at 64 Mhz., 32 bit. This is a 9-axis inertial measurement unit, it includes-an accelerometer, a gyroscope, and a magnetometer in build on it. However, the power requirement for this unit is lower than other usual/efficient boards, it requires merely 3.3 volts. The most important feature of this unit for our study is the memory, it has 1 MB of flash memory and 256 KB of SRAM. These memories will help us to run the NN model on this tiny board. Moreover, this board is an ideal match for our study because it is compatible with TensorFlow lite.

We use Arduino nano 33 BLE [2] to run the compressed version of our model on it. It is one of the updates of Arduino nano based on AtMega 328P. In this upgraded version, an updated CPU is used, ARM® Cortex®-M4 running at 64 Mhz., 32 bit. This is a 9-axis inertial measurement unit; it includes-an accelerometer, a gyroscope, and a magnetometer in build on it. However, the power requirement for this unit is lower than other usual/efficient boards; it requires merely 3.3 volts. The most important feature of this unit for our study is the memory; it has 1 MB of flash memory and 256 KB of SRAM. These memories will help us to run the NN model on this tiny board. Moreover, this board is an ideal match for our study because it is compatible with TensorFlow lite. The exciting part of this study is testing the board's limit for different NN models. We devised several models onto the board and tried to measure the scores of human presence through tinyML-based inference. To identify whether it is human and to define some score for it, we use Arducam [1]. This camera module is suitable for Arduino nano 33 BLE and uses a megapixel image senslabeled0.

3.3 Framework

For implementing MobileNet and other NN models, we use TensorFlow since it is one of the most widely used deep learning frameworks. We choose TensorFlow Lite for Microcontrollers to optimize and compress our model, which runs on the MCU.

4 METHODOLOGY

To make sure a deep learning model can be deployed to an MCU, we first describe the workflow from the top level:

(1) Decide the model architecture. Tune some hyperparameters to change the total number of parameters (width multiplier). Decide the input size of the images.
(2) Model training produces a TensorFlow model.
(3) Apply post-training compression techniques (pruning, 8-bit quantization)
(4) Convert to a TensorFlow Lite model using the TensorFlow Lite converter, and produce a model file with the suffix .tflite.
(5) Convert to a C byte array using standard tools (xxd), if the model is smaller than the MCU flash size, proceed to the next step, otherwise go back to step (1).
(6) Copy the C byte array to Arduino's IDE, write the inference logic, and send the code to MCU's flash. If the memory usage already exceeds the SRAM size, go back to step (1). Otherwise, proceed to the next step.
(7) Make the inference and observe the peak memory. Do the experiments and collect results. Figure 1 shows the steps involved in our process.

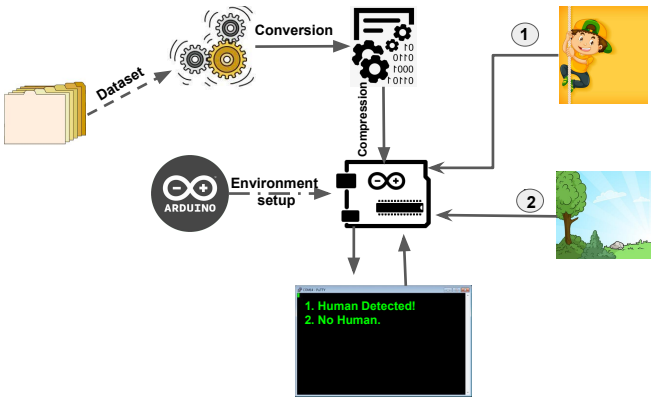


Figure 1: Running NN models in Arduino environment for image classification

4.1 Training

The full training process works as follows:

(1) Load pre-trained MobileNetV2 on ImageNet weights. Remove the classification head since the task has changed.
(2) Add several customized final classification layers for the binary classification task.
(3) Freeze all the layers before the classification head. Train the classification layers simply for a number of epochs, in this case, we use epochs=80. However, we also apply an early-stop callback function during training. So, in practice, the training should stop when the system observes no improvement.
(4) Fine-tuning. Unfreeze many layers in the model, and leverage a small learning rate to fine-tune the model so that the learned representations stay the same. We train for 15 epochs and then obtain the model.

MobileNetV2 defines several valid input image dimensions; we choose image size $\in$ [96, 160, 224], which includes the minimum size, size in the middle, and the maximum size of an image. We will see the difference in using these input image sizes later in the experiments. The batch size is 32. An important hyperparameter that affects the model's arch is the width multiplier. We treat it as one of the built-in model compression techniques and will discuss this later. We also trained a simple convolutional neural network (CNN) with only three convolution layers; two extra convolution layers can be added to the architecture if we specify a hyperparameter called scale. We will compare the performance of the simple CNN and its variants with our MobileNetV2 model.

4.2 Compressing Model

We try multiple model compression methods, including:

(1) Knowledge distillation
(2) Pruning
(3) 8-bit quantization
(4) Width multiplier
(5) TinyEngine [19] for optimizing storage and peak memory.

4.2.1 Knowledge distillation. As a popular deep-learning technique, knowledge distillation has drawn much attention in the literature. However, the process of having a trained teacher model teaches a student model may not be easy, in particular for a complex model like MobileNetV2. We try knowledge distillation in several different settings:

(1) Use a MobileNetV2, ImageNet pre-trained model as the teacher model, and use a CNN model with a small architecture as the student model.
(2) Use a MobileNetV2, ImageNet pre-trained model as the teacher model, and use the same MobileNetV2 architecture as the student model, but with random-initialized weights.
(3) Use a MobileNetV2, ImageNet pre-trained model as the teacher model, and use a MobileNetV2 architecture with fewer layers than the student model.

Since knowledge distillation is not a built-in TensorFlow technique as pruning and quantization, it is not trivial work to distill knowledge into an empty student model. Unfortunately, we cannot observe any performance improvement in all three cases above. Since applying a combination of several different compression techniques already gives us a good model size and accuracy, for the sake of convenience, we didn't spend much time investigating knowledge distillation. Here are several ideas we think could improve the knowledge distillation performance and would like to try in future work.

(1) Change the Distiller class implementation. We use an example code from Keras, it successfully distills knowledge between two simple CNNs. We want to investigate the customized loss function.
(2) Change teacher model output head. Knowledge distillation prefers the output of a teacher model to be vectors instead of a class label. We can modify the output layers to output feature vectors.

4.2.2 Pruning. TensorFlow has some built-in libraries for model pruning, and they support both structured and unstructured pruning. We met some issues when applying pruning to our models. Since our models are composed of some pre-processing layers, the main body as a sequential or functional object, and a classification head, the main body cannot be pruned as a whole object. In order to make it prunable, we need to consider composing the model with separate layers of the model body instead of a whole object when defining the model. We also leave this to future work.

4.2.3 8-bit quantization. This is the most effective method for model compression in our experiments, it reduces the model size prominently by 3 to 4 times.

4.2.4 Width multiplier. The width multiplier is in fact an argument of the MobileNetV2 model, the authors try to provide a built-in functionality so that the user can determine the right model size they wish to use. The default value is 1 which means the default number of filters from the paper are used at each layer, changing the value will either proportionally decrease or increase the number of filters in each layer. To compress the model, we choose the smallest width multiplier so that the model uses fewer convolution filters.

4.2.5 TinyEngine. The goal of the MCUNet system [19] is to have model-system co-design so that it can search for a small model and use customized hardware system support

to optimize certain operating costs. We wish to use one of their components - TineEngine only, to reduce the memory usage on the MCU. The authors claim that TineEngine can work on different MCUs, they provide an inference tutorial on the STM32F746G-DISCO discovery board. However, we fail to find TinyEngine support on our Arduino nano 33 BLE IDE's library.

In summary, the only two compression methods that work for our design are (1) Width multiplier and (2) 8-bit quantization. We will show the size comparison in the next section.

4.3 Performance Evaluation

4.3.1 *Experiments on the laptop.*

- To measure the accuracy, we use our test set, which contains 1000 images from the VWW dataset. We evaluate the accuracy of our model using the test set, then we report metrics such as confusion matrix, precision, recall, accuracy and F1-score.
- To measure the latency of our model, we use an approximated approach, which is to use the total time of executing the command "model.evaluate()" divided by the total number of images in the test set, which in our case is 1000. Since images can be fed into the function call in different batch sizes, we will also examine the effect of batch sizes on inference latency in the next section.

4.3.2 *Experiments on the MCU.*

- To measure accuracy we will test how often the model can give the correct label to samples, as it is a common way to measure accuracy in computer vision.
- To measure the energy usage by our model on the MCU, we will use a multimeter. There are some code-generated methods to detect energy consumption; however, they are prone to wrong calculations. Hence, we will use a multimeter in this case.
- To measure latency, we record the time delay between the time our models finish inference and the time we start the inference function call. We then report the average and standard deviation of these data.
- At the end of our empirical analysis we will use a confusion matrix, precision, recall and F1-score to report the accuracy of the models, we will also use plots to compare the performance across different models.

5 EXPERIMENTS

5.1 Model Compression Effectiveness

First, since our current compression techniques only include **Quantization** and **Width multiplier**, we will first look at

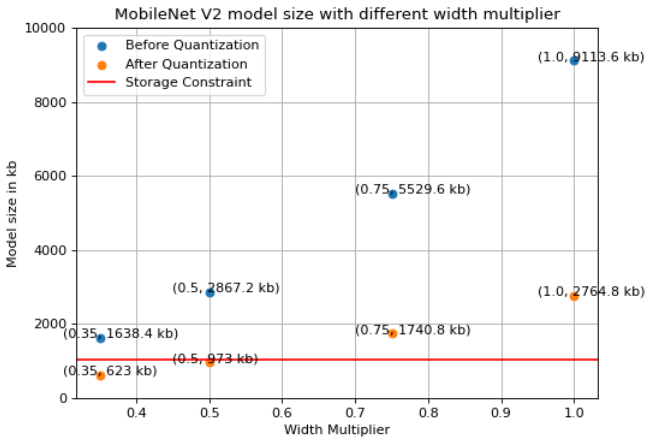


Figure 2: Comparing the storage usage of models with different width multipliers, before and after applying the quantization on a laptop.

Figure 2 and see how effective these two approaches are.

Figure 2 shows the model size comparison of MobileNetV2 with different compression settings, in which the width multiplier $\in [0.35, 0.5, 0.75, 1.0]$ and the presence of quantization. From the plot, we can see the combination usage of the width multiplier 0.35 and quantization manage to make the model size fit the storage constraint, with a model size of 623 KB. The combined usage of the width multiplier 0.5 and quantization also make the model size smaller than 1Mb, but it is closed to cause an overflow. Another thing we can observe from the plot is that both compression methods are guaranteed to shrink the model size. As we mentioned earlier, we also design a simple CNN to test the compression effectiveness and performance, the result is shown in Figure 10.

In figure 10 in the appendix, the scale is an argument we designed to determine the architecture of the CNN, when the scale equals 2 or 3, one or two convolution layers will be added to the architecture. We can see from the plot that most of the compression technique combinations can produce a model that satisfies the storage constraint.

5.2 How the batch size will affect inference latency?

After the training process is finished, we obtain a deep-learning model, to evaluate the accuracy on a test set, in TensorFlow we can invoke "model.evaluate()". It accepts a dataset and you can specify a batch size so that a number of inferences can be done in parallel. Here we want to examine how the batch size will affect the inference latency of models.

For different variants of the MobileNetV2 model, we use a batch size $\in$ [1, 16, 32, 64, 128]. The size of the test set is 1000. Figure 11 shows the result of the comparison. From the plot, we can observe that using a batch size of 1 gives the highest latency in all cases, while the other batch sizes give roughly similar latency for all variants of the MobileNetV2. This is not surprising since using batch size=1 does not run inferences in parallel. It will start another inference process after finishing the previous one. When using a large enough batch size, one bottleneck to prevent better parallelism is the memory size, that's the reason we have batch size $\geq$ 16 given roughly the same latency. This also holds for the variants of the CNN model, as shown in Figure 11 in the appendix.

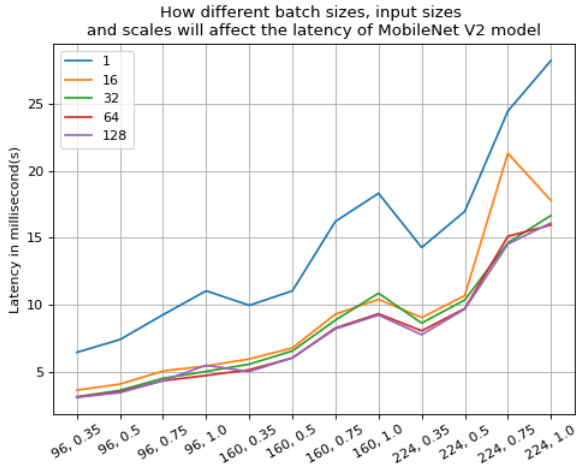


Figure 3: Examine how different batch sizes, input image sizes and width multipliers will affect the latency of a MobileNetV2 model on a laptop.

5.3 Examine the Accuracy and Latency

In section 5.2 we measure the latency of inference on an off-the-shelf laptop and discuss the role of the batch size. However, recall that a Tensorflow model must first be compressed and then converted into .tflite format, then use tools to convert it to a C byte array so that it can be deployed to an MCU. To run inference using a ".tflite" model, we must use an interpreter and do the inference one at a time. The latency and accuracy while using .tflite models are thus needed to be examined.

Figure 4 and 12 show how different image sizes and width multipliers will affect the inference accuracy for MobileNetV2 and our simple CNN. Note that in Figure 4, only MobileNetV2 variants with width multiplier=0.35 or 0.5 and quantization can fit into the constraints of our MCU. From figure 4 we can observe the following things:

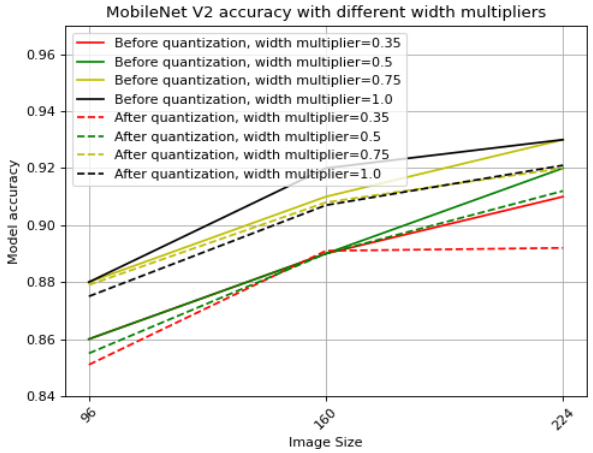


Figure 4: Examine how different image sizes, width multipliers and quantization presence will affect the accuracy of a MobileNetV2 model on a laptop.

(1) The larger the image size is, the more accurate the inference result will be.
(2) Quantization only decreases the inference accuracy slightly.
(3) The larger the width multiplier is, the more accurate the inference result will be.

A larger image size means more information is fed into the neural network, then the convolution layers can extract more accurate representations of a human's features. Quantization reduces the precision of all the weights from 32-bit floats to 8-bit integers, it reduces the model sizes by a factor of 3 4 and still preserves the network performance. A larger width multiplier means the network contains more convolution filters, thus enhancing the ability to capture features. In figure 12, for the CNN case, the accuracy is inferior in general, compared to the MobileNetV2 result. The patterns we observe in figure 4 do not hold for these simple CNNs, the reasons are threefold:

(1) The CNN we use is not deep enough. MobileNetV2 has more than 150 layers in total while our simple CNN merely has no more than 10 layers.
(2) The convolution layers need larger filters.
(3) It is not complex and deep enough, the inference result has more randomness.

Now let's look at the latency comparison of the MobileNetV2 and CNN in .tflite format when using an interpreter. In figure 5 and figure 13, we use the log-latency instead of latency since the latency of quantized models can surpass 1000 ms while the latency of un-quantized models can be 20 ms. The reason for having prohibitively high latency for the quantized models is discussed in one of TensorFlow's GitHub

issues, the quantized and optimized .tflite model is only optimized for mobile devices. We can observe several patterns in figure 5 and figure 13:

(1) When doing inference on a laptop, quantized models give higher latency than the unquantized models.
(2) Larger input image size leads to higher latency.
(3) Larger total number of parameters (more complex network architecture) leads to higher latency.

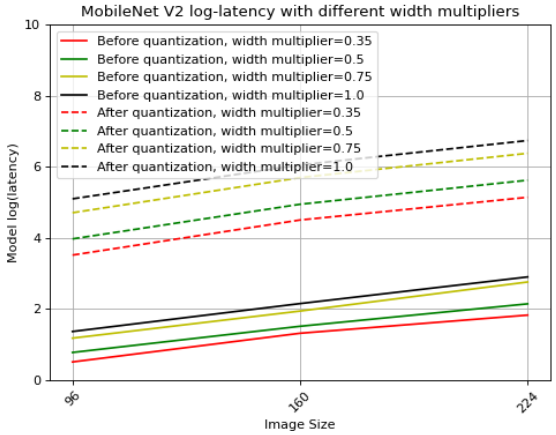


Figure 5: Examine how different image sizes, width multipliers and quantization presence will affect the latency of a MobileNetV2 model on a laptop.

5.4 Byte array Analysis of Neural Networks

Figure 6 illustrates different neural networks and their byte array sizes. We use several notations in our MCU-based experiments. *S_CNN* refers to the small CNN, *Q* or *NQ* means the Quantized or Non-Quantized mode, respectively, *MV* is MobileNet Version (1 or 2), and for our experiment, we have specified types of several NN models based on their size.

In figure 6, we can notice intriguing characteristics of Arduino Nano 33 BLE. Here, the orange points are the NN models overflowing from the MCU. On the other side, blue points are NN models that run on the MCU and provide us with expected inference. We trained several models to test the limit for the Arduino Nano 33 BLE. In this analysis, we have found that 800 KB of Byte array length is the highest model size we can put on the board for making inferences. To the best of our knowledge, this limit-wise analysis of a board is absent in the literature. We also found that MCUs preserve some place for operations; we can only use part of the 1 MB. Byte array size depends on the quantization of the model. *Q* model's byte array is smaller than *NQ* models.

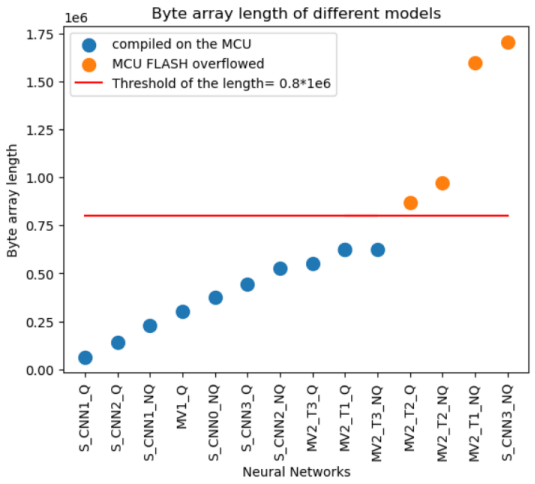


Figure 6: Different NN model and the byte array length

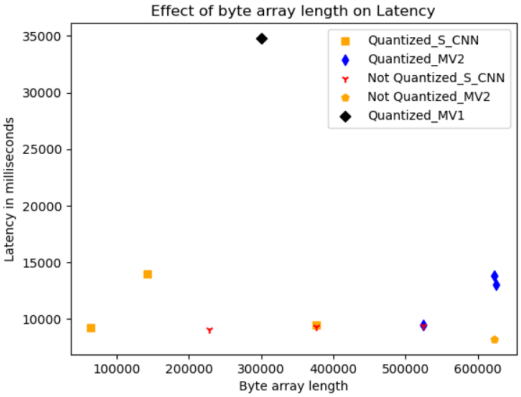
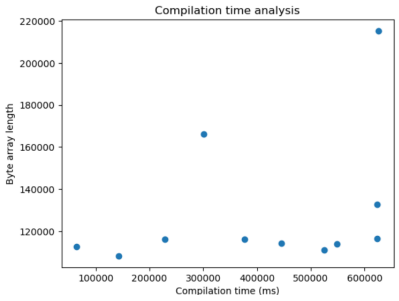


Figure 7: Different NN model and the latency on the board

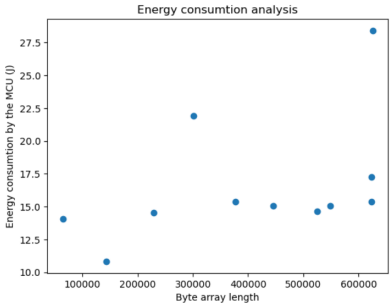
In Figure 7, we notice some intriguing findings; we find that the board takes 15 seconds on average as latency. In our setting, latency refers to the time between activating the camera function and getting an inference score in human detection through the serial monitor of the Arduino IDE. We also find in this analysis that quantization may not affect latency as we get the same latency for Quantized *S_CNN* (orange square shaped) and not quantized *S_CNN* (red y shaped). By some theoretical analysis, we claim that the complexity of the network architecture (number of parameters) may affect the latency.

5.5 Time and Energy analysis on MCU

In Figure 8a, we analyzed the compilation time of each Neural Network. We found that the compilation time can be similar for increasing byte array length. For instance, the last three points on the graph are on top of each other, which means

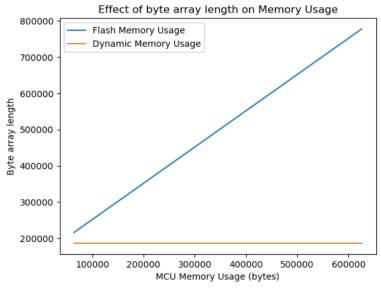


(a) Compilation time of several Neural Networks

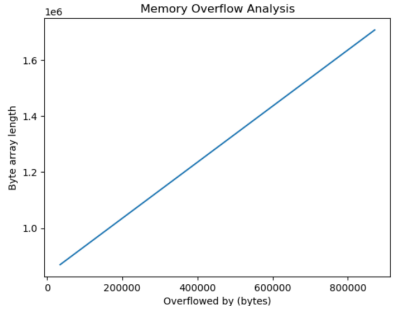


(b) Energy consumption by the MCU

Figure 8: Energy consumption and compiling time Analysis on MCU



(a) Flash and Dynamic Memory usage on the board



(b) Memory overflow by the .cc array on the MCU

Figure 9: Dynamic and Flash memory analysis of MCU

they have the same compilation time; however, their byte array length is different. The top holds the highest byte array length; the bottom two hold the lesser byte array length. Again, sometimes the hardware might behave slightly differently, so a small byte array length-based NN may provide a higher compilation time, like the middle point in the graph.

To measure the energy consumption by the MCU, we take the help of theoretical information based on Arduino.cc. The V_o for our board is 3.3 v, and the current is 40 mA as the board needs to use its highest amount of resources in the case of TinyMl-related works. Moreover, our analysis shows that we reached a peak in some cases concerning flash memory. Thus, we have selected the peak current usage by the board (empirically 40 mA). We calculated the power, which is 0.132 watts. Later, we used the power and compilation time to measure energy consumption reported in Figure 8b. We can see from Figure 8a and Figure 8b that in our setting, the graphs show similar attributes in all points.

5.6 Memory usage (Dynamic & Flash) and overflow analysis

Figure 9a delineates the memory usage of the byte array of different NNs. It is evident through the illustration that the usage of Dynamic and Flash memory is different for each NN. For the same NN, the memory usage as Flash is always higher than Dynamic memory usage. Moreover, Flash memory usage is linearly increasing with the Byte array length. On the other hand, Dynamic memory usage is similar (71%) for all the NNs. We found this similarity in Dynamic memory usage due to the exact program definition in the Arduino environment.

After the threshold length of 800 KB (as shown in Figure 6), we can not input anything on the MCU. As depicted in Figure 9b, the board keeps some space for the operation. Additionally, image capturing and decoding the image require some flash memory. We anticipate that MCU keeping the 200KB of memory for these operations of our program.

6 FUTURE WORK

Although we tried several different methods, due to the high re-training cost we didn't modify the way we compose a TensorFlow to apply pruning techniques. And since knowledge distillation is a hard problem, we tend to look for more promising methods to compress the model and satisfy the constraints. We obtained access to HPC like Compute Canada at the end of this work but no longer had time to re-train new models. We want to do several things in future work:

(1) Compose TensorFlow models layer by layer, instead of having the main body of the model as a functional

object or sequential object which hampers the applicability of the pruning techniques.
(2) Spend more time investigating knowledge distillation and applying it to the MobileNetV2.
(3) Measuring accuracy on MCU solving the IDE issue based on memory blockage in our system.
(4) Cross-platform analysis of the Neural Networks.
(5) Consider a mobile device and see the potential of using TinyML-based models on it.

7 CONCLUSIONS

In this work, we try several different methods to compress both a MobileNetV2 model and a simple CNN model. We compare the model sizes, accuracy, and latency before and after the compression on the laptop. We also compare how using different batch sizes for inference will affect the inference latency on the laptop.

Our main motivation for the work is to evaluate the limit of a microcontroller board for TinyML operations. We used Arduino Nano 33 BLE in our experiments and tried to infer the human detection and give some score on it. We successfully implemented the models on MCU and analyzed some important points like memory usage, both dynamic and flash, Latency of the NNs and the byte size, compilation time, and last but not least, energy consumption based on our setting.[1] We hope our work can shed some light on the process of compressing and deploying a deep-learning model onto a microcontroller.

8 TASK DISTRIBUTION

Qianxi Li: Prepare dataset, write code for VWW task, train and evaluate MobileNet model using VWW dataset, investigate model compression approaches, compress MobileNet model to fit into the memory of the MCU and examine the memory usage and latency for running the models on the laptop.

Abdullah Al Omar: Select an MCU platform for testing, and write code for it. Compare and analyze the result of variants of models by energy usage, memory usage, and latency. Take necessary decisions on the results and report accordingly.

REFERENCES

[1] Arducam. 2022. Arducam Mini 2MP Plus - OV2640 SPI Camera Module for Arduino UNO Mega2560 Board Raspberry Pi Pico. https://www.arducam.com/product/arducam-2mp-spi-camera-b0067-arduino/ Accessed: 2022-11-18.

[2] Arduino.cc. 2022. Arduino® Nano 33 BLE. https://docs.arduino.cc/resources/datasheets/ Accessed: 2022-10-15.

[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. *Proceedings of Machine Learning and Systems* 3 (2021), 517–532.

[4] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. 2018. Scalable Methods for 8-bit Training of Neural Networks. https://doi.org/10.48550/ARXIV.1805.11046

[5] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-All: Train One Network and Specialize it for Efficient Deployment. https://doi.org/10.48550/ARXIV.1908.09791

[6] Brian Chmiel, Liad Ben-Uri, Moran Shkolnik, Elad Hoffer, Ron Banner, and Daniel Soudry. 2020. Neural gradients are near-lognormal: improved quantized and sparse training. https://doi.org/10.48550/ARXIV.2006.08173

[7] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. 2019. Visual wake words dataset. *arXiv preprint arXiv:1906.05721* (2019).

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. *IEEE conference on computer vision and pattern recognition.*

[9] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. 2017. Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon. https://doi.org/10.48550/ARXIV.1705.07565

[10] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2018. Neural Architecture Search: A Survey. (2018). https://doi.org/10.48550/ARXIV.1808.05377

[11] Zehaos Zehao Shi et al. 2022. MobileNet: MobileNet build with Tensorflow. https://github.com/Zehaos/MobileNet Accessed: 2022-10-15.

[12] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR* abs/2103.13630 (2021). arXiv:2103.13630 https://arxiv.org/abs/2103.13630

[13] Babak Hassibi and David G. Stork. 1992. Second Order Derivatives for Network Pruning: Optimal Brain Surgeon. In *Proceedings of the 5th International Conference on Neural Information Processing Systems* (Denver, Colorado) *(NIPS'92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 164–171.

[14] 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. https://doi.org/10.48550/ARXIV.1802.03494

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. https://doi.org/10.48550/ARXIV.1503.02531

[16] 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. https://doi.org/10.48550/ARXIV.1704.04861

[17] Zehao Huang and Naiyan Wang. 2017. Data-Driven Sparse Structure Selection for Deep Neural Networks. *CoRR* abs/1707.01213 (2017). arXiv:1707.01213 http://arxiv.org/abs/1707.01213

[18] Edgar Liberis, Lukasz Dudziak, and Nicholas D. Lane. 2020. μNAS: Constrained Neural Architecture Search for Microcontrollers. *CoRR* abs/2010.14246 (2020). arXiv:2010.14246 https://arxiv.org/abs/2010.14246

[19] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. https://doi.org/10.48550/ARXIV.2007.10319

[20] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. https://doi.org/10.48550/ARXIV.1802.05668

[21] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. https://doi.org/10.48550/ARXIV.1801.04381

[1] Here's the code for this work:
https://github.com/liqianxi/compress_mobilenet

A APPENDIX

A.1 Plots

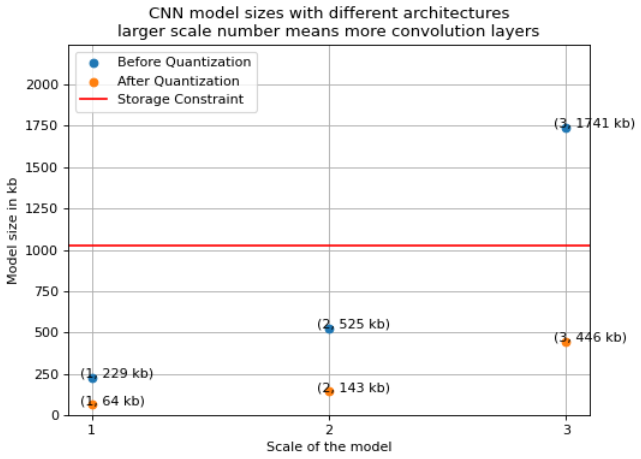


Figure 10: Comparing the storage usage of models with different network architectures, before and after applying the quantization on a laptop.

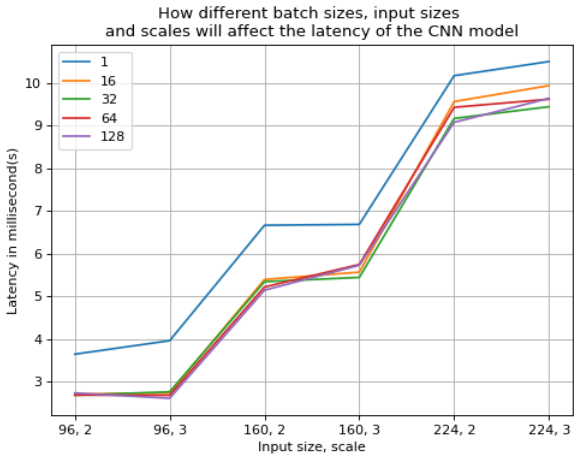


Figure 11: Examine how different batch sizes, input image sizes and network architecture will affect the latency of a CNN model on a laptop.

A.2 Model Compression Overview

To shrink the size of a deep learning model and satisfy tight hardware constraints, researchers have proposed a wealth of techniques. Their efforts can be categorized as follows:

(1) Design a small deep learning to fit into the constraints directly

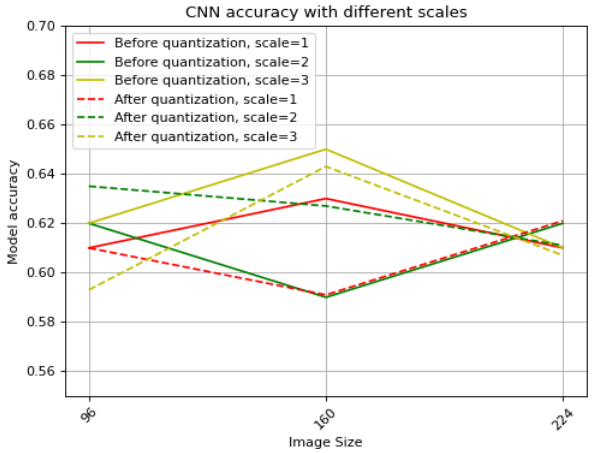


Figure 12: Examine how different image sizes, architecture and quantization presence will affect the accuracy of a CNN model on a laptop.

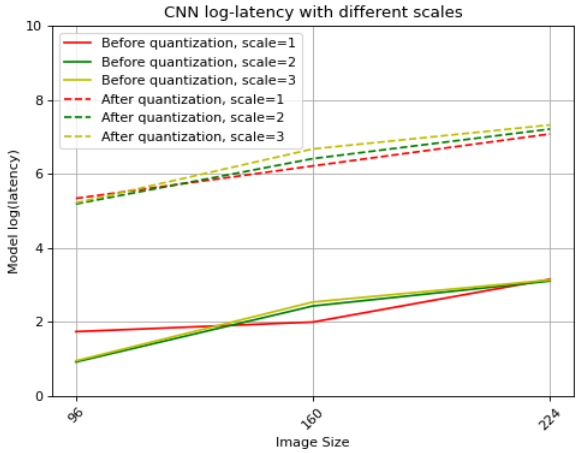


Figure 13: Examine how different image sizes, architecture and quantization presence will affect the latency of a CNN model on a laptop.

One line of work is to design Automated machine learning (AutoML) and neural architecture search (NAS) methods. Given the constraints of model size, depth, width and memory usage, this class of methods aims to find a perfect candidate neural network architecture so that all the constraints are satisfied [18]. A more complete survey of NAS techniques can be found here [10].

(2) Search for NN architecture and optimize for hardware together

Sometimes to keep metrics like peak memory usage and

energy consumption below a certain level, searching for a neat architecture is not enough [5]. We also want some common operations in a neural network to be optimized with hardware support. MCUNet [19] as an architecture and hardware co-design system searches for suitable candidates using search techniques and uses a hardware component called TinyEngine, which uses special-designed memory scheduling, specialized computational kernel and optimized convolution operations to reduce memory usage further.

(3) Pruning

Pruning is a technique that removes tiny weights from a network, which results in a sparse computational graph. With a large sparsity, more compression techniques can be applied to save the storage space of a model. Pruning can also be categorized into structured pruning [14, 17] and unstructured pruning [9, 13]. Structured pruning will remove a group of parameters, such as the whole convolution filters from the architecture, whereas unstructured pruning will remove neurons with a tiny weight that contribute less to the inference. Structured pruning is done aggressively and often causes a drop in accuracy, while unstructured pruning usually doesn't lose much of the generalization of a model. NAS can also be used for pruning proposes.

(4) Knowledge distillation

Knowledge distillation is first proposed in [15], it involves training a large teacher model, then using the soft probabilities produced by the teacher model to teach a simpler student model to learn. Compared to techniques like pruning and quantization which give at least 4 times smaller model sizes and still maintain a satisfying performance, knowledge distillation methods tend to have accuracy degradation. As a combination of these methods, [20] has shown great success.

(5) Quantization

Quantization is a technique that has shown to be remarkably effective [4, 6, 12], and it also has a long history in digital computing. The idea is that instead of using high-precision floating point numbers as weights, we convert them all into low-precision integer numbers with some fixed conversion rules. In this way, since all the weights required fewer storage spaces, the model size drops drastically. The way of applying quantization also includes: (1) post-training quantization(PTQ), which uses a small calibration dataset to compute the clipping ranges and the scaling factors, then quantize the model using calibration result. And (2) Quantization-Aware Training (QAT) quantizes a pre-trained model first, then re-trains the quantized model using a training set.