Machine Learning Compression for On-The-Edge Devices

Anonymous Author [*1]

Abstract

As machine learning enters a wide range of industries, there is a need to reduce both computational power and memory requirements for running machine learning models. Model compression offers a solution to this problem by reducing both the computational power required as well as the memory footprint. This article reviews three methods for compressing machine learning models, including exploring numerical formats and quantization algorithms. The methods proposed by Jacob *et al.* are used to quantize two Deep Convolutional Neural Networks (DCNNs) and are evaluated on three datasets: MNIST, Fashion MNIST, and CIFAR10. The results from this paper are compared with the results found from the two DCNNs. I also implemented my own end-to-end integer-only inference algorithms and compared them against the TensorFlow-Lite implementation of the ideas expressed by Jacob *et al.*

1. Introduction

As machine learning enters new industries, there is a greater need for efficient, accurate, and fast models. Traditional DCNNs, common to image processing applications, suffer from issues of model complexity and computational efficiency making them difficult to deploy on low-resource devices used in Internet of Things (IoT) applications. There are two techniques for solving these issues. First is to exploit novel network architectures that take advantage of computational and memory efficient operations. The second technique, which is the focus of this paper, is known as quantization or model compression (Yao et al., 2019). Quantization takes a traditional machine learning model and reduces the size of the model parameters allowing for lower latency and memory footprints. The papers discussed in this review offer different techniques for model compression. Although not the focus of this paper an example of exploiting novel network architectures is discussed in the paper SparCE: Sparsity Aware General-Purpose Core Extensions to Accelerate Deep Neural Networks. (Sen et al., 2017).

2. Reviews

2.1. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

This article describes methods for quantizing and training neural networks for implementation on edge devices with limited memory capacity or for devices without floating-point support. It provides a *quantization scheme*, *quantized inference framework*, and *quantized training framework* which detail how model compression is done. This article was selected because it is the framework that TensorFlow-Lite uses for its on-the-edge deployment of machine learning models. This article tackles the second solution mentioned in Section 1 providing a baseline for further research on the topic of model compression.

2.1.1. Summary

The *quantization scheme* outlined in this article is a mapping between the original floating-point weights and activations of a machine learning model to a specific range of integers determined by the number of bits allowed in quantization. The *quantzation scheme* is shown by equation (1).

$$r = S(q - Z) \tag{1}$$

Here r represents the original floating-point value, S is an arbitrary floating-point number which is based on the range of a given set of weights or activations, q is the quantized value, and Z is the zero-point which ensures that zero values are mapped to the literal value 0.

The *quantization inference framework* explains how to implement end-to-end integer-only-arithmetic once a model is quantized. In order to achieve integer-only-arithmetic Jacob *et al.* proposed a method for implementing fixed-point multiplication for S. This allows for accurate representations of S while doing integer-only-arithmetic, as shown through Equations (2), (3), (4).

$$r^{(i,k)} = \sum_{j=1}^{N} r^{(i,j)} r^{(j,k)} \tag{2}$$

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1) S_2(q_2^{(j,k)} - Z_2) \tag{3}$$

$$q_3^{(i,k)} = Z_3 + \frac{S_1 S_2}{S_3} \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2) \quad (4)$$

Equation (2) outlines how one value would be calculated in matrix multiplication. If we substitute r as defined in equation (1) then we get equation (3), rearranging equation (3) yields equation (4). In equation (4) the only floating-point value is $\frac{S_1 S_2}{S_3}$. Fixed-point multiplication is used to achieve an equivalent multiplication by $\frac{S_1 S_2}{S_3}$ (Jacob et al., 2017).

The *quantization training framework* establishes a method for insuring minimal accuracy loss when quantizing machine learning models by the methods discussed above. This framework inserts what is called *fake quantization* during training to simulate the loss in precision that occurs during quantization. *Fake quantization* is added to each set of weights and output activations as shown in Figure 1. Algorithm 1 outlines how to achieve the quantized training and inference (Jacob et al., 2017).

Algorithm 1 Quantized graph training and inference

```
1. Create a training graph of the floating-point model.
2. Insert fake quantization TensorFlow operations in locations where tensors will be downcasted to fewer bits during inference.
3. Train in simulated quantized mode until convergence.
4. Create and optimize the inference graph for running in a low bit inference engine.
5. Run inference using the quantized inference graph.
```

2.1.2. Methods

To test the effectiveness of the methods described Jacob *et al.* ran two experiments. Popular algorithms ResNet and InceptionV3 were trained on ImageNet and the floating-point and quantized graph efficiencies were compared. The concept of the latency-vs-accuracy tradeoff is compared between the two models.

2.1.3. Results

Jacob *et al.* found that on average the integer-quantized graphs had a 2% accuracy loss compared with their floating-point counterparts and a 12-46% reduction in latency. Further comparison between the results of this paper and my own implementation can be found in Section 3.1.2.

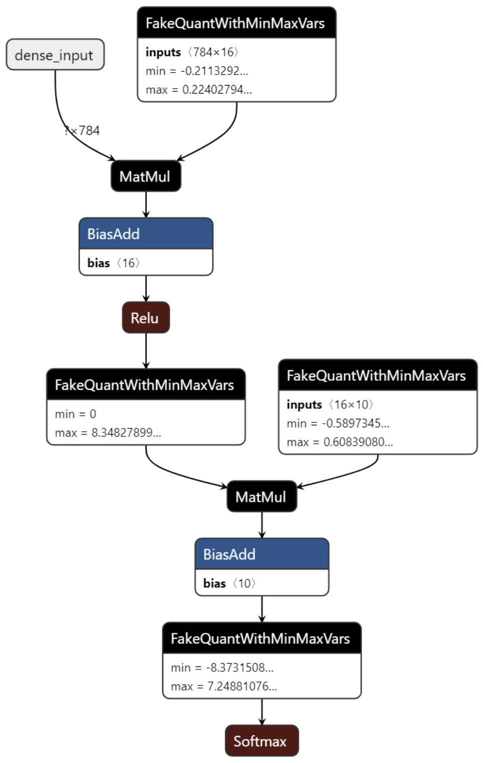

Figure 1. TensorFlow graph with *fake quantization* inserted (Roeder, 2019)

2.1.4. Discussion and Conclusion

This article describes some of the methods TensorFlow-Lite uses for creating machine learning models suitable for integer-only inferencing. The ideas presented in this article provided the foundation for modern research in the area of model compression. Jacob *et al.* provides the necessary references and understanding to implement quantized aware training. I used this information to create a machine learning algorithm for a machine learning accelerator benchmark, discussed in Section 3.2.

2.2. Cheetah: Mixed Low-Precision Hardware & Software Co-Design Framework for DNNs on the Edge

This article offers an analysis of how numerical formats and quantization approaches affect machine learning efficiencies. This article was chosen because it extends the ideas outlined by Jacob *et al.* considering how numerical formats and different quantization methods affect the performance of Deep Neural Networks (DNNs).

2.2.1. Summary

The *Cheetah* framework consists of two software level components and a single hardware level component. The software framework measures how effective various numerical formats and quantization approaches are for mapping 32-bit floating-point models to low-precision models. The hardware framework is a soft-core used for measuring hardware characteristics of multiply-and-accumulate (MAC) operations.

At a high level the software framework does accuracy analyses on various numerical formats and quantizations approaches to determine which is best for DNNs. The software framework begins with a DNN trained using a certain numerical format (floating-point, fixed-point, or posit). Then the network is quantized by one of two methods, rounding or linear. The rounding quantization method simply rounds the floating-point number to the nearest precision that can be represented by the desired quantization bit-width. If the number lies outside that range it is simply clipped to the respective maximum or minimum values. With linear quantization scale values are used to represent the original range of the floating-point values. This is similar to the quantization method expressed by Jacob *et al.* explained in section 2.1.1 but without a Z parameter.

The hardware framework does an energy-delay-product (EDP) analysis to measure how the numerical formats and quantizations approaches effect latency and resource cost for the exact multiply-and-accumulate (EMAC) operations. EMAC operations are implemented to reduce the error due to rounding. Each operation in EMAC is pipelined into different stages: multiplication, accumulation, and rounding.

2.2.2. Methods

The *Cheetah* framework was implemented in Keras and TensorFlow with support for rounding and linear quantization, EMAC operations, floating-point, fixed-point, and posit numbers. Langroudi *et al.* used various feedfoward neural networks and DCNNs to measure how different numerical formats and quantization approaches preformed. These networks were trained on three datasets: MNIST, Fashion MNIST, and CIFAR10, and used an FPGA to implement the hardware framework.

2.2.3. Results

Langroudi *et al.* found that posit numbers consistently outperformed both floating-point numbers and fixed-point numbers in accuracy metrics. When comparing between quantization approaches Langroudi *et al.* showed that posit numbers in combination with linear quantization preformed closest to the floating-point model equivalents. Although posit numbers combined with linear quantization improved accuracy by an average of 23% compared to fixed-point numbers it also increased EDP for EMAC operations. The posit number format had the best latency to accuracy degradation but was significantly slower than the fixed-point number implementation.

2.2.4. Discussion and Conclusion

This article was the first of its kind to use a software and hardware co-design to measure the effectiveness of numerical formats and quantization approaches on DNNs. Its expansion of the ideas expressed by Jacob *et al.* offers a better insight as to the best method for model quantization for low-resource devices.

2.3. Efficient Implementation of Convolutional Neural Networks With End to End Integer-Only Dataflow

This article gives yet another quantization method for implementing integer-only dataflow for DCNNs. Yao *et al.* uses a unified layer representation to efficiently condense the separate operations of a convolutional layer into one layer operation. A retaining scheme is introduced to help reduce accuracy loss from the proposed quantization. This article was selected to gain another view point in the area of model compression to further validate the claims mentioned by both Jacob *et al.* and Langroudi *et al.*

2.3.1. Summary

Yao *et al.* introduces a unified layer representation for traditional convolutional layers combining convolutional operations, biases, and activations into one operation denoted by equation (5).

$$O = \text{ReLU}(\text{MAC}(I, W) * A + B) \tag{5}$$

Here I represents the layer inputs, W represent the weights for the layer, A represents an affine transformation for implementing batch normalization, and B represents the bias term.

Yao *et al.* then goes on to discuss the method used for quantizing the layer parameters. The quantization method is defined by equation (6) and (7).

$$\begin{cases} P = \begin{cases} 0, & (abs_{max} \geq 2^{R-1}) \\ R-1, & (2^{R-1} > abs_{max} \geq 1) \\ 31, & (1 > abs_{max} \geq 0) \end{cases} \\ k = P - bitLen(round(abs_{max} * 2^{P})) + R - 1 \end{cases} \tag{6}$$

$$\begin{cases} W_q = round(W * 2^{w}),\ Q(E, w) \\ A_q = round(A * 2^{a}),\ Q(F, a) \\ B_q = round(B * 2^{b}),\ Q(F, b) \end{cases} \tag{7}$$

Here $abs_{max} = max(abs(W), abs(A), abs(B))$ where W, A, B represent the weights, activations, and biases from a given layer, R is the bit width, $bitLen$ represents the effective binary length of the integer value $round(abs_{max} * 2^P)$, Q is the symbol for integer quantization, $E = F = R = 8$, and the exponent w, a, b is a fixed value for each layer quantized.

The methods for implementing quantization for the arithmetic results are beyond the scope of this review. ResNet and DenseNet include two-path fusion, where two convolution layers are fused to one layer. This poses a problem for quantization because each layer needs to have the same exponent parameter from equation (7). Yao *et al.* introduces a *fusion* equation to help solve this by unifying the two layers exponents into one common exponent.

A retaining scheme is introduced to help reduce the accuracy loss post-quantization. Yao *et al.* describes what sounds similar to post-training quantization where the model is retrained to accommodate integer arithmetic operations.

2.3.2. Methods

The quantization methods discussed in this article were tested on a variety of popular DCNNs and trained on two datasets: ImageNet and PASCAL VOC. The accuracies of each are compared to the baseline model trained using 32-bit floating-point numbers and to the quantization methods used by TensorFlow.

2.3.3. Results

Yao *et al.* found that their proposed quantization method preformed with about 0.278% less accuracy than the TensorFlow implementation. It was also found that it achieved around a 1% loss in accuracies for models TensorFlow was unable to quantize.

2.3.4. Discussion and Conclusion

This article expands on the concepts discussed in Section 2.1 giving new methodology for quantizing 32-bit floating-point models to 8-bit integer models. The unified layer representation simplified arithmetic operations within DCNNs which offers machine learning accelerators more opportunity for improving optimizations for low-resource devices.

3. Implementation

I chose to use the ideas outlined by Jacob *et al.* to implement model quantization on two different DCNNs trained on MNIST, Fashion MNIST, and CIFAR10. I compared the accuracy and latency of the floating-point models with the 8-bit integer models. I show that the TensorFlow-Lite algorithms for model compression do not guarantee that the latency will decrease and that accuracy loss is significantly less for smaller models. I also compare my results with the results found by Jacob *et al.*

I also explore how to implement end-to-end integer-only dataflow using the *quantized inference framework* discussed in section 2.1.1. I compare the latency and accuracy of my own implementation of with that of TensorFlow-Lite.

3.1. Quantization Aware Training

In order to create a DCNN with quantization aware training I needed to follow Algorithm 1 discussed in Section 2.1.1. TensorFlow supports quantization aware training so inserting *fake quantization* is as easy as calling a function shown in Figure 2. Figure 1 shows where *fake quantization* is inserted into the model. Once *fake quantization* is inserted I trained the DCNN using Keras on one the datasets mentioned above. Using TensorFlow-Lite's converter I created a TensorFlow-Lite model suitable for integer-only inferencing. Once this model was created I tested it and compared the results to the floating-point version.

```
# inserts fake quantization
# for training
tf.contrib.quantize. \
create_training_graph()
# inserts fake quantization
# for evaluation
tf.contrib.quantize. \
create_eval_graph()
```

Figure 2. TensorFlow functions for inserting fake quantization

3.1.1. Methods

I chose to use two different DCNNs to look at how the size of the model effects both accuracy and latency. The first DCNN used has about 1.7M trainable parameters, which is about 10% of the size of the models used in Jacob's paper. The second DCNN had around 500K trainable parameters. When measuring metrics such as latency it is important to ensure that all variables stay constant except for the machine learning model. To do this I tested the floating-point model and 8-bit integer model on the same machine and tested with a batch size of one so I could measure the average time it takes to infer one image.

3.1.2. Results

Table 1 compares the accuracies between the two models over the three datasets: MNIST, Fashion MNIST, and CIFAR10. On average the quantized model was about 0.75% more accurate than the floating-point models. This likely occurred due to the loss in precision propagating through the model, which can yield unexpected results. I assume that the level of uncertainty in the results correlate to the

Table 1. compares floating-point and uint8 quantized model accuracy

	MNIST	Fashion MNIST	CIFAR10	MNIST	Fashion MNIST	CIFAR10
	Floating-point model			UINT8 quantized model		
Large DCNN (%)	99.16	92.48	73.17	99.19	92.46	73.82
Small DCNN (%)	98.32	90.71	61.05	98.39	90.64	64.93

Table 2. compares floating-point and uint8 quantized model latency expressed as an average time per inference

	MNIST	Fashion MNIST	CIFAR10	MNIST	Fashion MNIST	CIFAR10
	Floating-point model			UINT8 quantized model		
Large DCNN (ms)	5.82	5.23	6.88	18.84	17.47	24.72
Small DCNN (ms)	2.19	2.13	2.44	1.82	1.82	3.30

size of the model because the loss of precision propagates less allowing for less error accumulation. This result would explain why the results found by Jacob *et al.* experienced an average loss of 2.42% across their experiments when comparing floating-point with 8-bit models. Jacob *et al.* tested models significantly larger than the ones tested in this review allowing the error due to precision loss to have a greater effect.

Table 2 compares the average time per inference of floating-point models to 8-bit integer models. The quantized models were on average 7.21ms slower than the floating-point models. The results shown in Table 2 were largely unexpected. My intuition was that the quantized models would run faster due to the use of integer math. However, after careful consideration I have thought of a few reasons why this might not be the case. Modern processors can run floating-point math just as fast, if not faster than integer math due to its abundant use. I ran these tests on my desktop PC which has significantly more resources than an average microcontroller these quantized models would be deployed on. This means my desktop PC was likely able to hold most if not all of the machine learning parameters in its memory without needing to reallocate space during inferencing. When working on a low-resource device large machine learning networks will almost never be able to fit inside a microcontroller's cache, therefore the reduction in memory footprint due to quantization will allow the microcontroller to hold more parameters at a time reducing the need to reallocate memory speeding up the overall inference time.

3.1.3. Conclusions

Quantization allows for machine learning models to decrease its memory footprint by four fold giving on-the-edge devices the opportunity to reduce the need for reallocating memory and eliminating the need for communication between a host computer or server, dramatically increasing the time for response. This implementation showed the uncertainty created when quantizing models and explored what causes reduced latency in machine learning inferencing for quantized models.

3.2. Integer-Only Inferencing

The Purdue Vertically-Integrated-Projects System-on-Chip Extension Technologies (SoCET) team is creating their own microcontroller and one of the teams implemented sparsity aware core extensions (SparCE) to the RISC-V core (Sen et al., 2017). My task for the semester was to implement a machine learning algorithm for testing the efficiency improvements of the optimizations. The microcontroller is still in development and is currently limited to integer-only arithmetic, so my task was to find a way of creating a machine learning algorithm capable of integer-only inferencing. After reading about the quantization methods outlined in by Jacob *et al.* I was able to create a DCNN suitable for testing our newly added machine learning accelerator. The algorithm will soon be tested on an FPGA to compare latency, accuracy, and power consumption with and without the machine learning accelerator.

3.2.1. Methods

When exploring TensorFlow-Lite algorithms I learned that it uses a low-precision matrix multiplication library called gemmlowp. This library can be implemented on any system which supports C++11 and POSIX. Unfortunately for me our microcontroller supports neither so I ended up using the gemmlowp library as a reference for writing my own matrix multiplication and fixed-point multiplication algorithms (Google, 2019). Due to the memory limitations of the microcontroller and the fact that I am writing my own matrix multiplication algorithms I needed to keep the network small with relatively low complexity. The model was trained on a scaled-down version of MNIST (scaled-down to 14x14 pixel images and flattened to 196 element arrays). This flattened image is sent into a Conv1D layer with eight

	TensorFlow-Lite	My Implementation
Accuracy (%)	92.58	93.03
Latency (ms)	0.250	0.262

Table 3. compares TensorFlow-Lite matrix multiplication algorithms against my own

filters. This is flattened and sent into a Fully Connected layer with 16 nodes. Finally, a Fully Connected layer of 10 nodes is used for softmax prediction. When implementing the DCNN I decided to exclude the softmax layer from my algorithm because it was extremely convoluted due to the excessive use of fixed-point multiplication. Excluding the softmax layer does not effect accuracy because the maximum argument of the inputs to the softmax layer is the same as the maximum argument of the outputs of the softmax layer.

3.2.2. Results

At the time of writing this paper I am still working on running the algorithm on the simulated RISC-V core. However, according to Sen *et al.*, the SparCE optimization will improve latency by 19-31% and reduce power consumption by 17-29% without effecting model accuracy. Table 3 illustrates how my implementation compares to TensorFlow-Lite.

The latency in Table 3 refers to the average time it took for the model to infer on one image. The increase in accuracy between the two models is likely due to different rounding methods used by TensorFlow-Lite and myself, the rounding methodology I used was not better but introduced a degree of uncertainty which for this model increased accuracy. The increase in latency between the two implementations is as expected. TensorFlow-Lite has a lot of optimizations to ensure that the algorithm is performed as fast as possible. These are not included in my algorithm due to its complex nature. Despite my algorithm not including the softmax layer I still expected it to be slower due to these reasons.

3.2.3. Conclusions

The *quantization scheme*, *quantized inference framework*, and *quantized training framework* outlined by Jacob *et al.* gave me the understanding to implement a machine learning model that uses quantization aware training to generate a model capable of integer-only inferencing. This model and handmade machine learning implementation will be used to benchmark SparCE optimization added to a RISC-V core. Once tested I should see an improvement in latency with little to no loss in accuracy.

4. Final Thoughts

Each of the articles used in this review offer techniques for reducing the memory footprint of machine learning models, increasing inference time, eliminating the need for external communication, and allowing for deployment on integer-only processors. The ideas expressed in these papers will help the IoT revolution as industry finds new applications for machine learning on-the-edge.

References

Google. gemmlowp: low-precision matrix multiplication. `https://github.com/google/gemmlowp`, 2019.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. G., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017. URL `https://arxiv.org/pdf/1712.05877.pdf`. 153 citations.

Langroudi, H. F., Carmichael, Z., Pastuch, D., and Kudithipudi, D. Cheetah: Mixed low-precision hardware software co-design framework for dnns on the edge. *CoRR*, abs/1908.02386, 2019. URL `https://arxiv.org/pdf/1908.02386.pdf`. 0 citations.

Roeder, L. Netron: visualizer for neural network, deep learning and machine learning models. `https://github.com/lutzroeder/Netron`, 2019.

Sen, S., Jain, S., Venkataramani, S., and Raghunathan, A. Sparce: Sparsity aware general purpose core extensions to accelerate deep neural networks. *CoRR*, abs/1711.06315, 2017. URL `http://arxiv.org/abs/1711.06315`.

Yao, Y., Dong, B., Li, Y., Yang, W., and Zhu, H. Efficient implementation of convolutional neural networks with end to end integer-only dataflow. In *2019 IEEE International Conference on Multimedia and Expo (ICME)*, pp. 1780–1785, 2019. URL `https://ieeexplore.ieee.org/document/8784721`. 0 citations.