Deep Learning Model Compression for Resource-Constrained Applications

Neil Ashtekar
PhD Student
Department of CSE
The Pennsylvania State University
nca5096@psu.edu

Abhishek Dalvi
PhD Student
Department of CSE
The Pennsylvania State University
abd5811@psu.edu

Abstract

Deep learning models have become the de-facto standard of image related-tasks such as classification, de-noising, caption generation, etc. Such systems have become ubiquitous because of their capability to perform tasks accurately at a very rapid speed. In the last few years, deploying such algorithms on embedded systems has been challenging because most neural networks are computationally expensive, memory intensive and power hungry. Hence, to circumvent these issues, there have been techniques to compress large neural networks to be compatible and deployed on embedded systems. In this project, we experiment with three techniques, **clustering**, **quantization**, and **weight pruning** and analyze the reduction in the model's size and the performance degradation after performing the compression. We implement this project in TensorFlow and use TensorFlow Lite package for implementing the mentioned compression techniques. We also use TensorBoard's memory profiler to see how the compression techniques affect the memory usage for the GPU while training. Finally, we provide meaningful insights and visualizations for the experimental observations. The results show that compression techniques can reduce model size by a factor of $2\times - 4\times$ and the decrease in accuracy in most cases is around 1% to 2%.

Our implementations are available here, with most relevant code for VGG model compression in the 'new' folder: `https://drive.google.com/drive/folders/1NpMk528HUo-wXgUclrXUSLfXbLUeD4Gv?usp=share_link`

1 Design

1.1 Motivation

A system which can deploy deep learning models requires bounteous amount of memory and energy. For small portable devices such as cell-phones and cameras such resource demands are limiting. As shown in one of the lecture slides about Processing in Memory, the data movement cost for the DRAM can be $200\times$ the cost of the ALU. As noted by [3] the energy requirement for 32 bit on chip SRAM Cache is about 5 pJ and for a 32 bit DRAM Memory is about 640pJ. Large networks such as RestNet [5] and VGG [12] don't fit on the on chip SRAM, they need to access the DRAM memory. Thus, mobile devises would require large DRAM memory to store such models which is not possible because of the small-form factor of the mobile devices. Even if they were able to store such large models on DRAM, loading and running the model requires a lot of energy. For eg, the device has a model stored on the DRAM and it requires to access the model for some kind of task. Therefore, just accessing/loading a large model (lets say with a billion connection) at $20Hz$ would require $20Hz \cdot 1G \cdot 640pJ = 12.8W$, which is simply infeasible for a mobile device.

To make such devices compatible with large deep learning models; techniques such as clustering, weight pruning and quantization can be used to compress the model. Hence, we experiment and perform analysis on these techniques to see how good is their compression ratio is and the performance post compression.

1.2 Compression Techniques

1.2.1 Quantization

Quantization [2, 6, 11] is a technique to perform computations and storing tensors at lower bit-widths int stead of high precision floating point values. A quantized model executes majority of the tensor operations with integers rather than floating point values. This allows for a compressed model representation and the use of high performance operations on hardware if the hardware is optimized for effiencent and fast INT operations. This technique is in particular useful at the inference time since it saves a lot of inference computation cost without sacrificing too much inference accuracies.

In this project we have used three types of quantizations float16, int8 and dynamic. The first two are static quantization methods which quantize the weights (by simply rounding off to float16 or int8) and activations of the model. It requires calibration with a representative data set to figure out the ideal parameters for activations (mainly finding the correct scaling parameter). Dynamic quatinzation doesn't require any calibration. The weights are quantized before any inference is made but the activations are dynamically quantized on the fly while inference. Thus dynamic quantization have the run-time overhead of quantizing activations on the fly. So, this is beneficial for situations where the model execution time is dominated by memory bandwidth than compute.

1.2.2 Weight Clustering

Weight clustering or weight sharing [13, 14] reduces the number of unique weight values in the model. It does so by clustering the weights at each layer of the model and then use corresponding cluster centroids instead of the weights to reduce the memory footprint of the model. For eg, a layer with 100 weights is clustered into 32 clusters. Instead of using the original weights, we use the cluster centroids of the weight. This results in significant decrease in memory as the number of unique weights in a layer will be at most 32. Although, the total number of weights remain the same, post clustering compression or rather an efficient way to store and retrieve weights should be used for reducing memory footprint of a model.

1.2.3 Network Pruning

A common methodology for inducing sparsity in weights and activations is called pruning [4, 10]. Pruning is the application of a binary criteria to decide which weights to prune: weights which match the pruning criteria are assigned a value of zero. Pruned elements are "trimmed" from the model: we zero their values and also make sure they don't take part in the back-propagation process.

The way this is done by deciding a threshold factor; lets say ϵ and then taking the L-1 norm each set of weights and assigning zero to the set of weight if the L1 norm of the the weights is less than ϵ. Note that ϵ should be near zero to preserve the performance of the model. This is because the weights which do not provide a lot of information for the downstream task will be near zero as their contribution to the activation is minuscule. Hence, pruning out those weights will result in a new activation that is not so different from the pre pruning activation.

There is also magnitude based weight pruning which involves removing a fixed proportion of the lowest-magnitude weights from a pertained network and then fine tuning the model without these parameters. We have used this approach in our experiments.

1.3 Benefits on Hardware

Model compression techniques are primarily used to decrease model size, however they offer a host of additional benefits. As discussed in 2, the model original model we compressed was 58 MB. Since most GPUs have at least 4 GB of memory, our original model easily fits. Therefore, we also consider other benefits of compression.

Compressed models are often faster than uncompressed models (i.e. have lower inference latency). For example, pruned models have fewer weights resulting in fewer multiply-accumulate (MAC) operations at inference, while quantized models perform integer rather than floating point operations. Fewer required operations and/or faster operations both result in reduced latency.

In addition, smaller models offer benefits related to device memory hierarchy. Smaller models result in lower data transfer costs between memory and processing units. With smaller weights, more data can fit in caches, thus increasing the amount of relevant data near processing units and decreasing transfer between caches and main memory. Essentially, we are decreasing the size of elements in cache, which has a similar effect to increasing cache size with fixed-size elements. This should result in lower cache miss rates and increased cache utilization.

These benefits contribute to compressed models consuming less power. Compressed models consume less dynamic power since fewer operations are required, and also consume less static/leakage power, since fewer weights must be stored in memory. Similarly, compressed models generate less heat and require less cooling since they are smaller and perform less work than uncompressed models.

2 Methodology

We used a VGG-13 computer vision model trained on the CIFAR-100 dataset [1] The original model size is 58 MB, and its classification accuracies are – top-1: 70.47%, top-3: 85.81%, top-5: 90.20%. Top-k refers to the proportion of the samples in which the correct label is within the model's largest-k predicted activations in its output layer.

The CIFAR-100 dataset contains 32×32 RGB images. There are 100 total class labels. CIFAR-100 is a moderately difficult dataset to classify (not nearly as trivial as MNIST) as state-of-the art models from as recently as 2019 only achieve 93.5% top-1 accuracy after supplementing the training data with synthetic images [8].

We performed model compression using TensorFlow Lite (tflite). Tflite is a library for converting and deploying deep learning models on edge devices. Originally, we had planned to use PyTorch, however after reviewing the documentation, we found that tflite had better support for a wider range of compression methods as compared to the PyTorch equivalent (PyTorch Mobile). We applied three compression methods to VGG-13 using tflite: quantization, weight clustering, and pruning. We evaluated the reduction in model size as well as the performance of the reduced model compared to the original model. Our results are listed in 1, and a comprehensive review of our design choices and their effect on performance is discussed in 3. We conducted most of our experiments using an NVIDIA A100 GPU accessed through Google Colab Pro+. Out of curiosity, we also completed a handful of inference runs on a MacBook Pro laptop.

3 Evaluation and Results

3.1 Quantization

We started by applying three post-training quantization methods: float16 quantization, dynamic range quantization, and int8 quantization. The original model has weights stored in float32 format, so quantizing to float16 and int8 results in model size reductions of roughly $\frac{1}{2}$ and $\frac{1}{4}$ respectively, as shown in 1.

Dynamic range quantization also converts model weights to int8, however the model's activations are quantized dynamically when performing inference. This means that the model's activations are converted to int8 within the network, and the output layer's activations are de-quantized back to floating point values when making predictions. This approach differs from int8 quantization, which is a "full integer quantization" method, meaning that both the model weights and activations are kept as int8 values at all times. Neither float16 nor dynamic range quantization require any data and can

[1] Originally, we had planned to use a ResNet-50 model pretrained on ImageNet, however we found that the ImageNet dataset was too large (roughly 170 GB!) to run our experiments, even when attempting to fine-tune our model rather than train from scratch. We still performed quantization on ResNet-50 and obtained similar size reductions to those given in 1.

Method	Model Size Reduction	Accuracy Decrease
Quantization (float16)	2x	0.01%
Quantization (dynamic)	3.97x	1%
Quantization (int8)	3.97x	2%
Weight Clustering (128 centroids)	3.55x	1%
Pruning* (magnitude-based)	3.08x	8%

Figure 1: Results after applying various compression techniques to VGG-13 trained on the CIFAR-100 dataset. Accuracy decrease is measured with respect to Top-1 classification accuracy. Pruning results are given for the CIFAR-10 dataset, as we had difficultly finding a set of pruning hyperparameters resulting in good performance on CIFAR-100.

be applied post-training relatively easily. int8 quantization does require a representative unlabeled data sample used to determine max/min values and set a range used to effectively quantize.

Applying float16 quantization resulted in a negligible accuracy loss. Interestingly, while dynamic range and int8 quantization would appear to significantly reduce precision, model performance remains strong and only decreases by $1 - 2\%$ after compression.

We would expect quantization to reduce inference latency, as float16 and int8 operations are both generally faster than float32 operations. However, during our evaluations, we observed the quantized models had about the same latency as unquantized models, and were sometimes even slower! After further investigation, we found this open issue on GitHub [9] explaining that this latency difference is because tflite models are optimized for ARM chips rather than x86, which require many additional quantization/dequantization operations (explaining the slow latency when performing inference on a MacBook with an Intel Core i9 processor). When running on a GPU, the latencies before and after quantization were about the same. As discussed by GitHub user Namburger in [9], there is a latency decrease by a factor of roughly $\frac{1}{2}$ when running unquantized versus quantized tflite models on an Edge TPU device.

3.2 Weight Clustering

We found that weight clustering provided similar benefits to integer quantization, even though the underlying algorithms are quite different. We suspect that the similarity in performance between the two methods can be explained by the fact that weight clustering only needs to store centroid values and indices rather than a full set of weights. The number of centroids stored is significantly smaller than the number of original weights. In addition, the values used to index the centroids for each weight can be stored using a small number of bits – for example, if 16 clusters are created, then we only need to store 4-bit indices for each weight. This means that the majority of the values stored will be small indices (likely in int8 format), explaining the $3.55\times$ reduction in size compared to the original weights (in float32 format).

We had to experiment with different numbers of cluster centroids in order to achieve good performance. We found that decreasing the number of centroids to be less than 100 resulted in very poor performance, while performance stayed roughly constant for 128 and above, almost matching the original model. Note that the original model includes many 2-dimensional convolutional layers with either 128, 256, or 512 filters of size 3×3, so storing only 128 centroids is a vast decrease in the number of floating point parameters required.

Unlike quantization, weight clustering can benefit from fine-tuning to improve performance. First, the original model's weights are clustered, and next, the resulting centroids are adjusted to maximize performance. This is equivalent to running gradient descent with the additional requirement that the

model can only have a small number of distinct weights which are heavily reused within each layer. Surprisingly, we observed that fine-tuning did not significantly improve performance. Perhaps the original weights cluster "nicely" (i.e. weights form distinct clusters) and/or using 128 centroids was enough to adequately represent the data.

Clustering does not significantly reduce inference latency. This is because clustered models must perform the same number of floating point operations as unclustered models in order to compute activations. Note that there may be some minor speedups due to weight caching – clustered models have a smaller number of unique weights (centroids) as compared to unclustered models, so these centroids could be cached and efficiently reused for many operations. With unclustered models, a larger number of weights would take up more space in the cache leaving less space for training data, potentially resulting in more data transfer and longer runtimes.

3.3 Pruning

Finally, we performed magnitude-based weight pruning. This method involves removing a fixed proportion of the lowest-magnitude weights from the pretrained network, then fine-tuning the network without these parameters. As compared to the quantization and clustering, pruning had many more hyperparameters which we had to experiment with to get good performance. These hyperparameters included:

- Sparsity: how many weights should be removed from the network?
- Pruning schedule: should the weights be removed all at once, or should they be removed iteratively during fine tuning? If they should be removed iteratively, what should the starting and ending sparsity be? What should the decay rate be?
- Layers to be pruned: should all the layers be pruned, or just the deeper layers? Should specific types of layers be pruned?

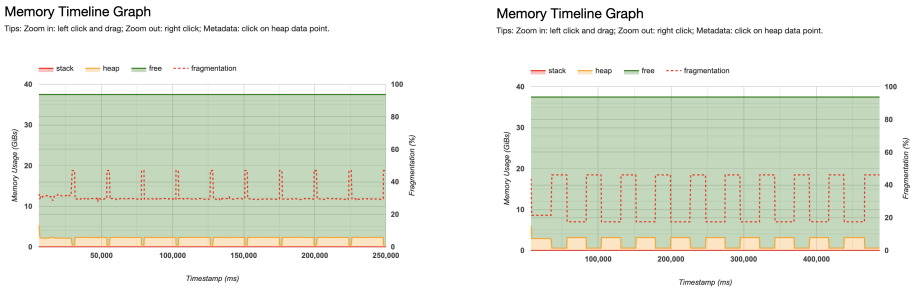
In addition to these hyperparameters, all the hyperparameters related to fine-tuning must also be determined (i.e. number of epochs, batch size, learning rate, etc). After many attempts using different combinations of hyperparameters (varying sparsity from 20% to 80% in steps, only pruning the later layers, only pruning the convolutional layers, etc) we were only able to achieve about 45% top-1 accuracy on the CIFAR-100 dataset. Note that this is still somewhat decent performance – since there are 100 classes, the baseline accuracy from randomly guessing is in the single-digits. Instead, we applied pruning to the easier CIFAR-10 dataset, and were able to sparsify the model by 80% – i.e. only 20% of the compressed model's weights are non-zero – with less than 10% drop in top-1 accuracy. Note that sparsifying the model by 80% does not mean that the compressed model size drops by 80% (this was incorrectly stated in our presenation slides). First, the biases are not sparsified, as these terms do not contribute to model complexity, and second, there are preexisting bottlenecks built into the way TensorFlow stores models.

We were surprised to learn that pruned tflite models do currently benefit from latency reduction (this is also mentioned explicitly in the tflite documentation [1]). This is surprising because pruning should result in faster inference, as fewer operations are required since most weights have been removed. Again, this is likely because of preexisting design choices made in earlier versions of TensorFlow which introduce incompatibility with pruning-based latency optimization. The documentation mentions that future version of tflite will provide latency reduction to pruned models.

3.4 Extra: Memory Profiling

In addition to evaluating various model compression techniques, we also used TensorBoard for memory profiling. Originally, we had planned to use TensorBoard to profile compressed versus uncompressed models at inference, however we were unable to do so as memory profiling is primarily designed for debugging during training (see this open GitHub issue [7]). Instead, we thought it would be interesting to profile quantization-aware training (QAT).

As the name suggests, QAT performs quantization during training, specifically in the forward pass of backpropagation. This allows the backward pass to adjust parameters in a way which accounts for the loss of precision introduced by quantization, thus improving quantized model performance. Note

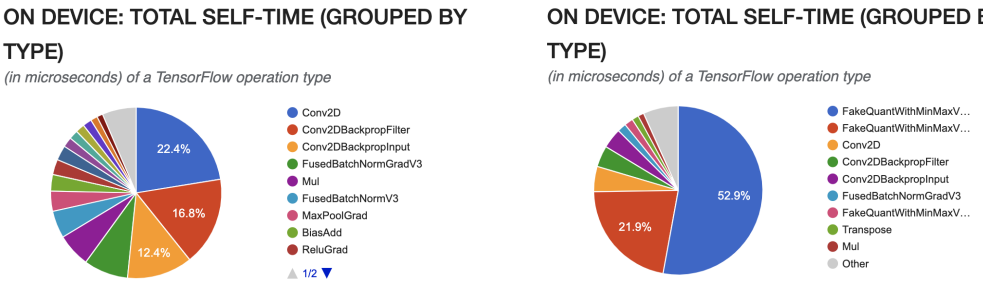


(a) Original Model (b) Quantization-Aware Model

Figure 2: Memory usage on GPU during training

that the methods evaluated in 3.1 are all post-training quantization, not QAT. QAT generally offers the best performance at the highest computational cost during training.

We trained our VGG-13 model with and without QAT for 10 epochs on the CIFAR-100 dataset using a GPU. The respective memory profiles are shown in 2. Both plots have 10 clean segments in which memory is allocated for each epoch, as expected. There are two notable differences between the plots: first, the original model takes about half the time to train as compared to the QAT model (see x-axis timestep), and second, the QAT model has larger time gaps between each epoch. The first observation is easily explained by the fact that QAT requires more operations than standard training, hence the longer runtime. The second observation is not as obvious – perhaps QAT performs many quantization/dequantization operations on a small subset of the weights between epochs.



(a) Original Model (b) Quantization-Aware Model

Figure 3: Breakdown of GPU time spent performing various types of operations

In addition, we compared the time spent performing various operations on the original model and QAT model, as shown in 3. As expected, the standard training spends most time on convolution operations, while the QAT model spends most time performing so-called "FakeQuant" operations. FakeQuant nodes are introduced in QAT to first perform quantization and then dequantization for the model's forward and backward passes. We can see that FakeQuant operations dominate the training time, confirming our hypothesis from 2.

4 Extra: Research Exploration

The final component of our project involved open-ended research exploration. We focused on the problem of teacher-student federated distillation, in which the local student models are resource-constrained and therefore much smaller than the global teacher model. Both models are assumed to be some kind of neural network. The global teacher may not access the local data of each student, though they may communicate via privacy-preserving proxies such as weights or gradient updates. We would like to update the global teacher model based on the local student model updates in an online fashion.

We were not able to devote as much time to this problem as we would have liked due to other end-of-semester commitments (i.e. NeurIPS conference), however we had some preliminary ideas. First, we suspect that this problem can be reduced the problem of finding the correspondence between the weights of a large teacher model and small student model trained on the same data. If this correspondence can be determined, then it appears fairly straightforward to derive the correspondence in gradient updates between the two models. This correspondence function could be used to map the gradient updates from the student model to the teacher model, or vice versa. Note that the reduced problem is not at all trivial – this is just one potential direction.

Another idea is to use principal component analysis (PCA) to map between the weights of the two models. This could be used if the models have the same number of layers, but the student model has fewer nodes or weights per layer as compared to the teacher model. We could learn PCA on the weights of the teacher model, then apply PCA to the weights to update the student model. Alternatively, we could map from student to teacher by applying the transpose of the learned PCA matrix to student's weights or gradients. This idea is motivated by the fact that PCA "summarizes" data nicely in a reduced dimension, though problems may arise when dealing with nonlinear activation functions.

Another idea is to somehow learn the mapping between the student and teacher (i.e. treat this as a meta-supervised learning problem). We feel that this is a interesting problem and the ideas we proposed are still very preliminary would require a lot of additional investigation.

5 Contributions

5.1 Neil

- Coded implementations in tflite
- Wrote Methodology 2, Evaluation and Results 3, and Research Exploration 4 sections
- (Team) Discussed compression techniques and planned experiments
- (Team) Brainstormed ideas for research exploration

5.2 Abhishek

- Research compression methods in detail
- Wrote Abstract and Design 1 sections
- (Team) Discussed compression techniques and planned experiments
- (Team) Brainstormed ideas for research exploration

References

[1] Tensorflow for mobile and edge: Model optimization, 2021.

[2] Sungmin Bae, Piotr Zielinski, and Satrajit Chatterjee. A closer look at hardware-friendly weight quantization. *ArXiv*, abs/2210.03671, 2022.

[3] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *arXiv: Computer Vision and Pattern Recognition*, 2015.

[4] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural network. *ArXiv*, abs/1506.02626, 2015.

[5] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015.

[6] Lu Hou and James Tin-Yau Kwok. Loss-aware weight quantization of deep networks. *ArXiv*, abs/1802.08635, 2018.

[7] Geonhwa Jeong. Inference profiling, 2020. https://github.com/tensorflow/profiler/issues/24.

[8] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning, 2019.

[9] Mieszko Kłusek. Slow quantized tflite, 2020. https://github.com/tensorflow/tensorflow/issues/40183.

[10] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *ArXiv*, abs/1608.08710, 2016.

[11] Sang-Il Seo and Juntae Kim. Efficient weights quantization of convolutional neural networks using kernel density estimation based non-uniform quantizer. *Applied Sciences*, 2019.

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[13] Sanghyun Son, Seungjun Nah, and Kyoung Mu Lee. Clustering convolutional kernels to compress deep neural networks. In *European Conference on Computer Vision*, 2018.

[14] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, and Yingyan Lin. Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions. *ICML*, 2018.