Assessing the Impact of Network Compression in Deep CNNs

Andrew Akers
Georgia Institute of Technology
aakers@gatech.edu

Clayton Smith
Georgia Institute of Technology
csmith658@gatech.edu

Ryan Mcbee
Georgia Institute of Technology
rmcbee6@gatech.edu

Joseph Waugh
Georgia Institute of Technology
jwaugh6@gatech.edu

Abstract

Advances in using deep learning for computer vision tasks have been primarily driven by more complex neural network architectures. While this has resulted in material performance gains, it has also led to over-parameterized models that require significant computational resources. Deploying these complex models in resource constrained environments, such as mobile applications, has been challenging. In this research, we explore two techniques for model compression: pruning and quantization, which seek to balance model performance and computational costs. Our experiments prove that both techniques in isolation and a combination of the two are effective ways to significantly reduce model size and complexity without a corresponding degradation in performance on the CIFAR-10 dataset.

1. Introduction/Background/Motivation

Convolutional neural networks (CNN), have enabled tasks that were previously difficult to accomplish without deep learning based approaches. CNNs are particularly useful to robotic systems which rely heavily on visual based sensors to navigate and interact with the world. Adding more advanced visual sensing to robotic systems can provide greater levels of autonomy, and potentially allow for the removal of auxiliary and expensive sensors, such as lidar [11]. Unfortunately, CNN algorithms tend to be characterized by a large number of multiplications and additions to accomplish the matrix multiplication that comprises the algorithm. This limits CNNs to relatively expensive and large processors which restricts the size, weight, and power (SWaP) of robotic systems.

One current solution to reduce computation complexity is to use more efficient CNN architectures such as the mobilenet, which introduced a decoupled type of convolutional layer known as a depth-wise separable convolution[8]. This new design was able to achieve similar accuracy to a full convolution network while having 9 times less parameters, which greatly reduced the total memory and computation used. Another method to reduce computational complexity is to convert the floating point precision that SWaP constrained processors do not have efficient support for to 8-bit integers which these processors have more efficient support for implementing. Converting the floating point values to 8-bit integers is typically done after the floating point version of the CNN is trained, and typically results in minimal decrease in the overall accuracy, more efficient computation, and a 4x decrease in memory.

While the optimizations previously mentioned have enabled CNNs to run on more platforms, there are still a host of computing platforms that still cannot handle the computation/memory demands of CNNs. In this research, we explored two relatively new techniques, parameter pruning and sub 8-bit quantization that are less commonly used but have the potential to greatly increase CNN efficiency. Our research focused on better understanding these two techniques and how they can both be used to further increase the efficiency of CNN algorithms without a substantial decrease in performance. We hope this research will help inform future trade-off considerations between efficiency and accuracy for computer vision algorithms targeted towards low SWaP based systems. The techniques explored in this research are dataset agnostic, so we used the Pytorch default CIFAR-10 and CIFAR-100 datasets to evaluate our research. We chose these datasets because they have a large number of labeled images, are well explored in literature, and are only 32x32 pixels, which allows for quick and easy evaluation of the different models and optimization techniques we explored. A brief analysis of model performance with regards to the number of predictor classes is analyzed with regards to the ResNet and MobileNet architectures used for this report.

2. Approach

2.1. Pruning

Parameter pruning is the process of removing weights which contribute the least to the final accuracy. This type of approach is typically done by removing the smallest weights, which can be thought of as representing those that have the least impact on the model output [5]. Pruning creates a more efficient system by reducing the total memory used for the weights as well as reducing the total number of computations since any weight that is converted to 0 can be ignored.

Pruning strategies for deep CNNs can be separated into two broad categories: unstructured and structured. Unstructured methods remove individual parameters from the 2d convolutional filters across the entire network, creating sparsity. The main drawback of this method is that it requires a deep learning implementation with a sparse linear algebra library to realize its compression benefits. Structured pruning, on the other hand, removes entire channels of filters and output feature maps, which does not require any changes to deep learning implementations, such as Pytorch. Given that one of our main goals is to analyze network compression metrics, such as storage size and inference time, we chose to implement structured pruning.

An important consideration in structured pruning is determining which set of filters to remove. In an ideal situation, we would remove the largest set of filters that had a minimal impact on performance. However, there is no optimal way to efficiently solve this problem due to the extremely large possible combinations of filters within a deep network. Further, a greedy approach of pruning one filter at a time with the smallest impact on performance is also not optimal because its impact will likely change as additional filters are pruned in following iterations. Li et al. (2017) [6] created a simple yet effective strategy that selects the smallest filters to prune based on the ℓ_1-norm. The basic idea is that due to the multiplication operations in convolutions, small filters will produce small activations, which will be carried forward through the network and in turn have a small impact on the output. We utilized this method as the foundation of our pruning strategy. In addition, our implementation used some of the code from a python package [10] that the authors published in conjunction with the paper.

Through experimentation, we found that a naïve pruning strategy in which a fixed percentage of filters were removed from all convolutional layers resulted in significant performance degradation. This led to the discovery that the concept of layer sensitivity is key to effective pruning. We define layer sensitivity as the loss in validation accuracy after a percentage of the layer's filters are pruned in isolation. Layer sensitivity for both ResNet18 and MobileNetV2 are displayed in Figures 1 and 2.

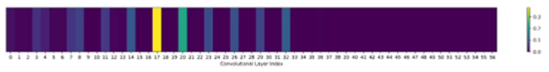


Figure 1. Convolutional Layer Sensitivity - ResNet18

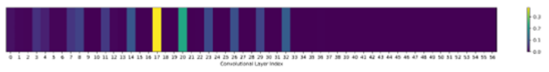


Figure 2. Convolutional Layer Sensitivity - MobileNetV2

Specific convolutional layers in each model are very sensitive to pruning and are essential to performance. We see that this often occurs in earlier layers, whereas layers toward the end of the network are insensitive to pruning. This phenomenon is likely related to the idea of receptive field – later layers have a larger receptive field, and are therefore more likely to have redundant feature activations that can safely be removed. Our overall pruning strategy relies on the realization that some layers can be pruned aggressively, while others should be left untouched. It can be implemented with the following steps:

1. Calculate isolated layer sensitivities for each convolutional layer over a range of pruning percentages (e.g., 0.1, 0.2, etc.).

2. Set the amount of pruning, p_i, for convolutional layer i at the highest percentage in which the layer sensitivity is greater than some small negative constant (e.g., $-1e-3$).

3. Remove $p_i * n_i$ of the filters at layer i for all convolutional layers. Retrain the network until the validation performance stops improving.

2.2. Quantization

As mentioned earlier, quantization is the process of converting the original floating point weights and activations into a fixed point integer representation for greater efficiency. There are two main types of quantization: post training quantization (PTQ) and quantization aware training (QAT) [7]. The first type, PTQ, takes the original floating point values and directly converts them to 8-bit integers without any further training. A small calibration step using unlabeled data is used to readjust the batchnorm statistics. This technique has been shown to work very well for 8-bit quantization, but it performs significantly worse on lower bit precision [4].

The other method, QAT, allows for retraining after the quantization occurs. To make the training work with the quantized values, a new technique called straight through estimator (STE) is used. This technique works by using the quantized weights in the forward pass of the CNN, but during backpropagation [3][12], the floating point weights are updated. This allows the fine updated steps of the backpropagation to work for even the integer quantized values. This process can be improved further by making the quantization levels a learnable parameter allowing for the back propagation to iteratively learn the best quantization level instead of naively setting the quantization level upfront using statistics about the weights. These two ways of implementing quantization have their pros and cons. For PTQ, it is relatively easy to implement, does not require additional training, and does not need any labeled data. The cons of this technique are that it results in much lower accuracy than QAT, especially for lower precision quantizations. For QAT, the pros are that it results in much higher accuracy than PTQ since the weights and activations can adjust after the quantization step. The cons are that this extra quantization step results in significantly more training time and complexity.

Of these two techniques, we went with the QAT training since we were targeting quantization levels of 5 bits or lower. Using such a low quantization level has two major benefits to improving the overall efficiency of a CNN. First, it allows for the use of more simplified mathematics. On devices that can support custom arithmetic, such as Field Programmable Gate Arrays (FPGA), lower bit precision can help achieve a nearly 2x increase in efficiency of the final computation and less overall resources [1]. Additionally, quantizing the weights and activations uses less overall memory, which will greatly increase the efficiency. By converting the weights and activations from 8-bits to 4-bits, an overall memory saving of 2x is achieved.

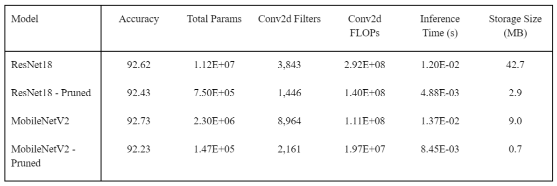
For our specific implementation, we leveraged the code from the PROFIT quantization paper [9]. The quantization process used in this paper is as follows.

1. Train a full precision CNN as normal.

2. Progressively quantize the activation bit precision starting at 8-bit quantization for all activations down to the final bit precision.

3. Progressively quantize the weights starting at 8-bits for all weights and going down to the final bit precision.

For both the weight/activation quantization level, two steps are done. First only the batchnorm and learnable quantization levels are trained, leaving the weights unchanged during the initial step. This gives the network time to reach more stable statistics before training the weights. Next the batchnorm, learnable quantization, and weights are all trained. Since the network has already been trained in full precision, the number of epochs can be significantly less than what were used in the original training. For our research, we used five epochs for the statistics stabilization step, and 15 epochs for the weight training step. For the learning rate scheduler, we found that cosine annealing gave the best results, most likely since it will start off at max value, and decrease quickly for the relatively short number of epochs used in this step.



Model	Accuracy	Total Params	Conv2d Filters	Conv2d FLOPs	Inference Time (s)	Storage Size (MB)
ResNet18	92.62	1.12E+07	3,843	2.92E+08	1.20E-02	42.7
ResNet18 - Pruned	92.43	7.50E+05	1,446	1.40E+08	4.88E-03	2.9
MobileNetV2	92.73	2.30E+06	8,964	1.11E+08	1.37E-02	9.0
MobileNetV2 - Pruned	92.23	1.47E+05	2,161	1.97E+07	8.45E-03	0.7

Figure 3. Pruning Model Results

3. Experiments and Results

3.1. Pruning

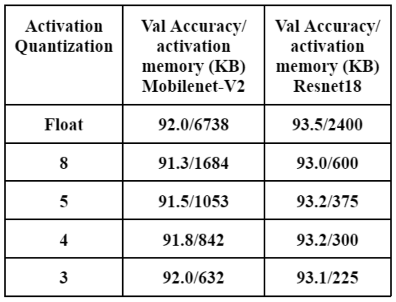
In addition to accuracy degradation, we measured pruning success based on a number of different metrics that helped quantify the pruned model's size and speed. These metrics included the total number of convolutional filters, individual parameters, and floating point operations in the forward pass, the inference time for evaluating a batch, and storage size. From a practical perspective of deploying the model in a resource constrained environment, inference time and storage size are the most important metrics.

Our pruning strategy was quite successful as we were able to remove a significant portion of both MobileNet and ResNet without material performance degradation. In many of the convolutional layers that were insensitive to pruning, we were able to remove up to 60% of the filters, suggesting that the model architectures create highly overparameterized models. The most impactful result was the reduction in storage size. Even though MobileNet was already designed to have a light footprint, we were able to reduce its size from 9.0MB to just 0.7 MB, a 92% reduction. Meanwhile, the validation accuracy fell just 0.5%, from 92.7% to 92.2%. The full results for the original and pruned models are shown in Figure 3.

3.2. Quantization

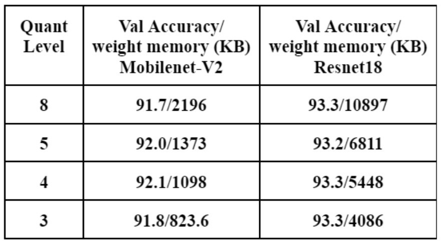
Because sub 8-bit quantization requires special hardware to get inference time speed ups, we only looked at total memory usage of each of the different quantizations. In table Figure 4, we show the validation accuracy for full model activation quantizations with full precision weights for both Mobilenet-V2 and Resnet18. In table Figure 5, we have the validation accuracy for both models where the activations were fixed at 3-bits and the weights were progressively quantized.

There are some interesting takeaways from these results. First, it makes sense that from the floating point model to the 8-bit model the validation accuracy decreases just slightly. But what is surprising is that as the quantizations progress, the accuracy increases. There are two possible explanations for this occurrence. The first and most likely



Activation Quantization	Val Accuracy/ activation memory (KB) Mobilenet-V2	Val Accuracy/ activation memory (KB) Resnet18
Float	92.0/6738	93.5/2400
8	91.3/1684	93.0/600
5	91.5/1053	93.2/375
4	91.8/842	93.2/300
3	92.0/632	93.1/225

Figure 4. Validation accuracy for activation quantization only



Quant Level	Val Accuracy/ weight memory (KB) Mobilenet-V2	Val Accuracy/ weight memory (KB) Resnet18
8	91.7/2196	93.3/10897
5	92.0/1373	93.2/6811
4	92.1/1098	93.3/5448
3	91.8/823.6	93.3/4086

Figure 5. Validation accuracy for weight quantization using 3-bit activations

answer is that the hyper parameters for how long to train each quantization level were too short and resulted in each quantization level not regaining the full accuracy. The second explanation is that the lower bit precision resulted in a greater regularization of the network that allowed the it to generalize better. While outside the scope of this research, tuning the hyper parameters could result in improved performance for the different quantization levels.

One other interesting feature is the difference between which method uses more activation memory and which uses more weight memory. Many of the previous papers focused on quantization only examined the reduction in weight memory. While this is helpful in making the overall CNN more efficient by not having to move weights around as much, one factor that is heavily overlooked is how much memory is used to store the intermediate memory. As we can see between the two models, these two factors can vary wildly between models that achieve similar accuracy. For ResNet18, it uses significantly less activation memory than MobileNetV2. On the other hand, ResNet18 uses significantly more memory to store the weights than does MobileNetV2, with almost a 5x difference in weight memory between ResNet18 and MobilNetV2. These differences in where memory is used should be factored into the decision of which model to use based on which custom computing platform you are using.

3.3. Combination of Pruning and Quantization Methods

In order to further optimize model accuracy versus computational footprint, we combined both model compression methods detailed above using three separate strategies: pruning then quantizing, quantizing then pruning, and an interleaved method. The interleaved method breaks the quantization operations into steps and progressively quantizes and prunes at the same time, choosing the best operation (in terms of accuracy) at each step. We target 3-bit quantization in four steps (8,5,4,3) and scale back the pruning amount by a factor of four. We use these methods on both ResNet18 and MobileNetV2 to determine if model structure impacts performance in terms of compression and accuracy. The model compression metrics used here are the in-memory footprint (calculated by multiplying the number of model parameters with their size), and the size of the model on disk.

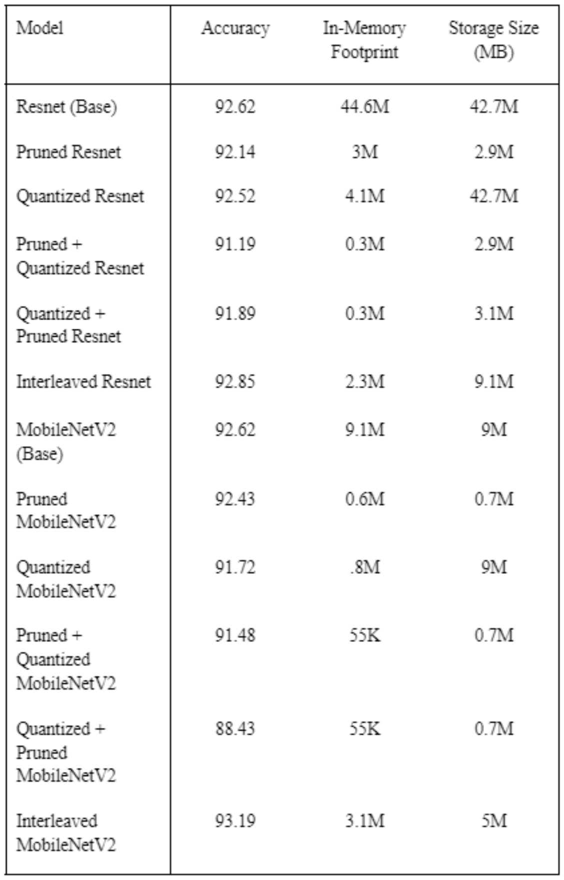
In terms of accuracy, the interleaved methods perform the best even compared to the base models (92.85% accuracy and 93.19% accuracy for the ResNet18 and MobileNetV2 models, respectively) while the basic pruning and quantization combinations fall slightly short in terms of accuracy relative to the standalone pruning and quantization methods. The pruning then quantizing method appears to have the higher accuracy of the two methods. This is likely due to the fact that the pruning selection algorithm struggles to determine what to prune at low resolution since the network is already densely encoded.

Despite slightly lower accuracy scores, the combination methods eliminate the tradeoff between quantization and pruning. Both quantized models have substantially lower memory usage (ResNet18 with a 99.3% reduction, MobileNetV2 with a 99.3% reduction) as well as a substantially lower storage size (ResNet18 with a 93.2% reduction, MobileNetV2 with a 92.3% reduction). The ordering had a negligible effect on compression, with only the quantized-then-pruned resnet showing slightly increased storage space. This suggests that the accuracy optimal ordering should generally be selected, which for both models appears to be pruning and then quantizing.

While the interleaved models don't offer as substantial a reduction in memory and storage, it is very interesting to see that they perform better than the base models. This suggests that the interleaving compression process could potentially be used to allow models to generalize better while at the same time generating a more compact network.

3.4. Model Accuracy: CIFAR-10 and CIFAR-100

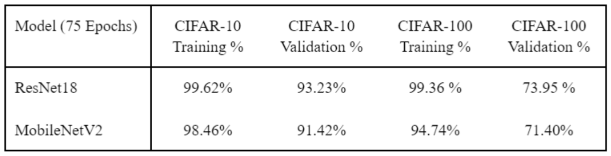
In order to validate the results for quantization and pruning methods described above, a comparison between the obtained results for the pruned ResNet18 and MobileNetV2 neural network architectures were tested against two datasets: the CIFAR-10 (the original dataset used above) and CIFAR-100 datasets. This will showcase the effect of increasing the number classes has on these architectures compared to the original results above. The initial



Model	Accuracy	In-Memory Footprint	Storage Size (MB)
Resnet (Base)	92.62	44.6M	42.7M
Pruned Resnet	92.14	3M	2.9M
Quantized Resnet	92.52	4.1M	42.7M
Pruned + Quantized Resnet	91.19	0.3M	2.9M
Quantized + Pruned Resnet	91.89	0.3M	3.1M
Interleaved Resnet	92.85	2.3M	9.1M
MobileNetV2 (Base)	92.62	9.1M	9M
Pruned MobileNetV2	92.43	0.6M	0.7M
Quantized MobileNetV2	91.72	.8M	9M
Pruned + Quantized MobileNetV2	91.48	55K	0.7M
Quantized + Pruned MobileNetV2	88.43	55K	0.7M
Interleaved MobileNetV2	93.19	3.1M	5M

Figure 6. Pruning Quantization Impact on Memory

results were obtained below:



Model (75 Epochs)	CIFAR-10 Training %	CIFAR-10 Validation %	CIFAR-100 Training %	CIFAR-100 Validation %
ResNet18	99.62%	93.23%	99.36 %	73.95 %
MobileNetV2	98.46%	91.42%	94.74%	71.40%

Figure 7. CIFAR-10 vs CIFAR-100 accuracy

In looking at the results above, there's a clear decrease in validation accuracy when using the CIFAR-100 dataset instead of CIFAR-10. The relative decrease in validation accuracy is within 19-20% for both model architectures, thus showing that the impact is very similar. These model architectures generate an output by selecting the argmax of a class prediction for each class in the dataset.

As a result, we shift from selecting the argmax of 10 predictions, to now the argmax of 100 predictions in choosing the class prediction. By shifting this number of classes upwards, the confidence in distinct classes is decreased as the distinction between these classes is reduced with a larger number of potential predictions. Prior research has supported this theory [2], and the strong decrease in these validation accuracy scores can potentially be linked to the different classes featuring strongly significant features that cause the model to struggle when generating these confidence scores for each class. Pairing this with an increase in the number of classes means that the margin of error in predicting the correct class among several classes that may be similar in terms of the loss function outputs results in misclassifications.

References

[1] Convolutional neural network with int4 optimization on xilinx devices white paper. 2014. 3

[2] Felix Abramovich and Marianna Pensky. Classification with many classes: challenges and pluses, 2015. 5

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013. 2

[4] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018. 2

[5] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989. 2

[6] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets, 2016. 2

[7] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021. 2

[8] Eunhyeok Park and Sungjoo Yoo. Profit: A novel training method for sub-4-bit mobilenet models. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 430–446, Cham, 2020. Springer International Publishing. 1

[9] Eunhyeok Park and Sungjoo Yoo. Profit: A novel training method for sub-4-bit mobilenet models, 2020. 3

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. 2

[11] Zhiqing Wei, Fengkai Zhang, Shuo Chang, Yangyang Liu, Huici Wu, and Zhiyong Feng. Mmwave radar and vision fusion for object detection in autonomous driving: A review, 2021. 1

[12] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets, 2019. 2

Student Name	Contributed Aspects	Details
Ryan McBee	Quantization Implementation and Coding; Introduction/Background/Motivation Section	Ryan McBee worked on implementing the low-precision quantization used in this research and implemented the training code for the quantization training. He also wrote the introduction/background/motivation section.
Andrew Akers	Model Pruning Implementation and Coding; Model Training Code	Andrew Akers implemented the model pruning and layer-by-layer pruning strategy based on layer sensitivity. He also wrote the basic Pytorch model training code.
Clayton Smith	Pruning and Quantization Methodology and Coding	Clayton Smith worked on combining the pruning and quantization methods and implementing the interleaved compression method.
Joseph Waugh	CIFAR-10 vs CIFAR-100 research and analysis; Report Formatting; Meeting Coordination	Joseph Waugh worked on comparing the non-pruning and quantized base model results and quantized model results on both the original CIFAR-10 dataset and CIFAR-100 dataset to see if an increase in the number of classifications has an impact on validation accuracy. He also compiled the draft of the report into LaTeX format using the provided template.

Table 1. Contributions of team members.