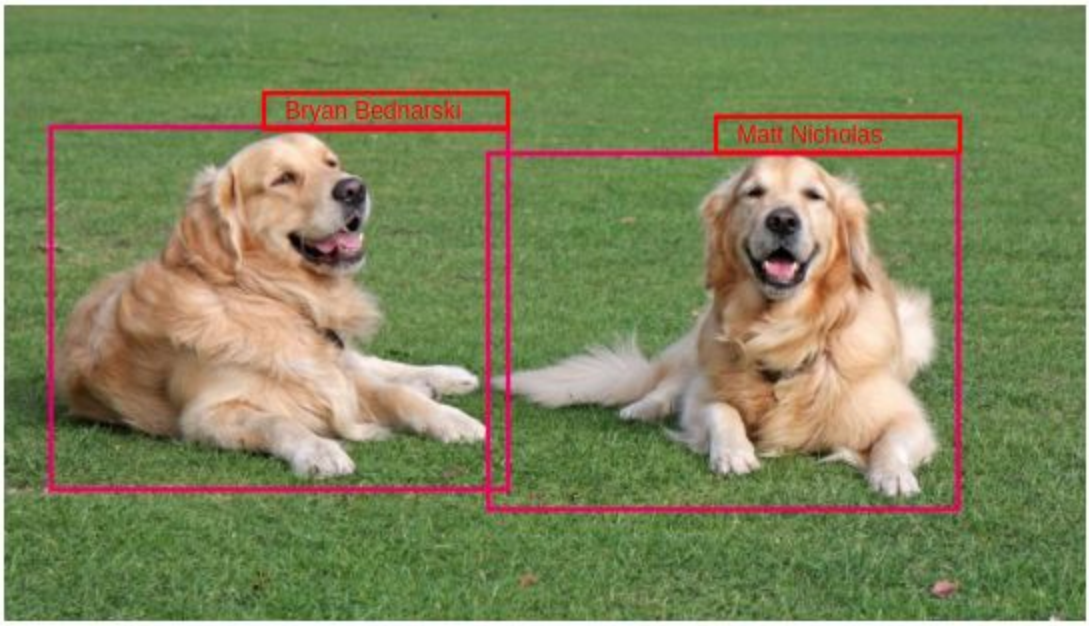
Exploring the Optimization of Deep Convolutional Neural Networks for Object Detection and Bounding

Matthew Nicholas, Bryan Bednarski

Department of Electrical Engineering, UCLA

June 12, 2020

https://github.com/bbednarski9/CS259_Final



UCLA

Abstract

Convolutional neural networks comprise the backbone of modern image recognition platforms. Since Yan LeCun's LeNet in 1995, numerous networks like UNET and Yolo-V3 have created opportunities for innovation in deep learning vision applications. Recently, newer developments in image recognitions with GANs and semantic segmentation in robotics have pushed the capabilities of these systems to their operational limits. In this report, we aim to explore some of the most promising optimizations for convolutional neural networks: pruning and weight sharing. We aim to show the impact of these network modifications on the accuracy and latency of a trained Yolo-V3 network and provide an intuition as to how these optimizations impact performance on different hardware platforms. Therefore, throughout this report we will interweave analysis of both pruning (layer and global-based) and weight sharing algorithms, with runtime analysis on three different NVIDIA GPU platforms: Tesla V100 (server), 1080 GTX and Jetson Nano development board. Our aim is to provide intuition towards getting complex and cumbersome deep CNNs to run efficiently on a variety of research platforms.

Introduction

Neural networks require large amounts of computational power, memory bandwidth, and storage space to be practically deployed. These constraints make high-end deep learning applications difficult to deploy on hardware constrained edge and mobile devices. Cloud computing can sometimes be used to remedy these issues, especially during a network's training stage where computational throughput for mini-batch training is often a limiting factor for systems. However, during inference, communication between the target device and the cloud datacenters may introduce unacceptable delays in timing critical applications. Additionally, certain applications cannot risk the privacy concerns that arise when sending sensitive data to cloud datacenters. As a result, there has been significant research directed towards finding techniques to reduce the memory and computational overhead for neural networks while maintaining acceptable accuracy.

One such technique that does just that is described in *Deep Compression* [1]. In this paper, researchers prune, retrain, quantize, and compress a neural network. While these techniques result in considerable memory saving, a dedicated accelerator must be used to achieve energy saving and inference speedup.

This paper will make three primary contributions. First, it will analyze and explore new techniques that build upon those outlined in Deep Compression to achieve improved theoretical memory savings while maintaining accuracy. Second, these techniques will be applied to YoloV3, a state-of-the-art object recognition system. This class of network was not evaluated in Deep Compression, and it will give insights into the generalizability of the techniques. Third, the object recognition system will be deployed on three different hardware platforms to determine if (and if so, why?) the implemented techniques impact speed of inference.

Deep Compression

In *Deep Compression* Han et. al. implement a three stage pipeline to reduce the storage requirements for neural networks. The first stage of the pipeline is to prune unnecessary weights from the network and retrain[1]. Pruning is performed by setting a global threshold and removing all weights from the network that have a smaller absolute value. This results in reducing the number of parameters by 9x for Alenet and 13x for VGG-16 without a drop in accuracy from the baseline.

This paper will implement a similar pruning and retraining technique with a few changes. Rather than determining a threshold for the absolute value of a weight, a threshold is determined for the 'L1' norm of a weight. Additionally, a second method for pruning is implemented. This method is "layer-based" and prunes an equivalent percentage of weights within each layer. For each of these methods an overall percentage is set to be pruned. For the global method, the smallest set percentage of weights are pruned with respect to the entire network.

The second stage of the *Deep Compression* Pipeline is to reduce the number of bits required to represent each weight by applying weight sharing to the pruned network. For each convolutional and fully connected layer in the network, k-means clustering is performed to find the shared weights. The shared weights are stored in an array, and an index into the shared weights array is stored in the weights matrix. Eight bits were used for indexing the shared weights in the convolutional layers (256 shared weights) and 5 bits were used for fully connected layers (32 shared weights). This resulted in an additional 27x-31x reduction in size without drop in accuracy.

This paper will also implement similar weight sharing techniques with additional changes. First, YoloV3 does not have fully connected layers, so weight sharing will only be applied to convolutional layers. Second, the number of bits used to index the shared weights will be altered layer by layer to determine if further memory savings can be achieved without a drop in accuracy. Additionally, global weight sharing will be implemented. Using this method a single shared weights matrix will be used across all convolutional layers.

The third stage of the *Deep Compression* pipeline is huffman encoding, which will not be implemented.

All of the size reduction techniques that will be implemented are "symbolic"- the actual architecture of the network is never changed. Symbolic approaches allow for a true analysis of the resulting accuracy of these techniques, but the timing statistics are not equivalent to what they would be if the architecture of the network was actually changed (sparse matrix multiplication is super slow!). For pruning, a mask is applied to each layer. When performing inference, the network uses the mask to determine which weights have been pruned so that they do not impact accuracy. For weight sharing, rather than each weight in a layer storing an index into a shared weights array, the shared weight that is being indexed is simply put in the layer in place of the original weight.

All of the pruning and weight sharing techniques will be evaluated separately. Then, the pruning and weight sharing techniques that produce the best results will be combined and deployed together.

Yolo-V3 Object Detection Network

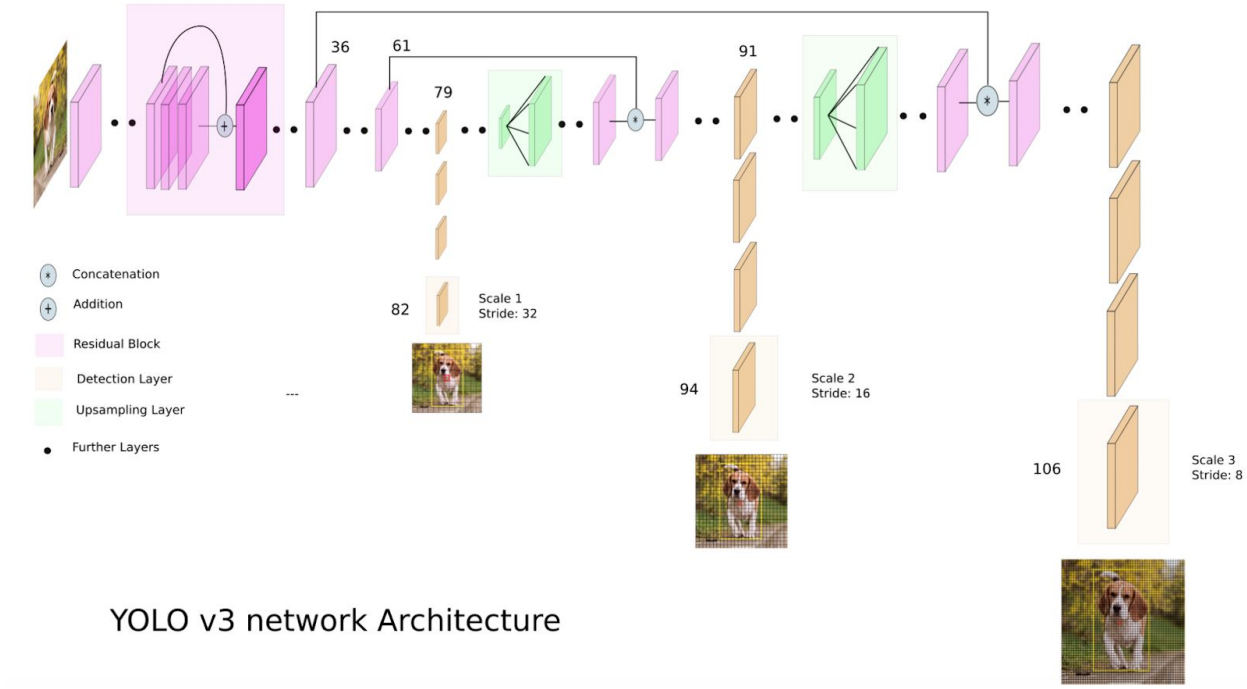


Figure 1: The Yolo-V3 network architecture consists of two Darknet-53 backbone networks connected in series (second reversed) and provides bounding box prediction at three different feature map scales.

The Yolo-V3 object detection network is a convolutional neural network, built on the Darket-53 CNN backbone structure. This backbone comes pretrained (bootstrapped) on the ImageNet database, but still needs to be retrained for its application. Yolo-V3 has demonstrated best-in-class object detection accuracy of 57.9/55.3/51.5 mAP (mean average precision) at 20/34.4/45.5 fps (frames per second) using the COCO dataset for training and inference. These results outperform other well-regarded networks like SSD and RetinaNet in speed and accuracy in most cases[2].

The Yolo-V3 network totals 106 layers and consists of two Darket-53 backbones, the first is oriented normally while the second is reversed for the up-sampling effect. At layers 79, 91

and 102, feature maps are extracted for anchor box prediction at three different scales to detect objects of different sizes with similar accuracy. At each of these different scales, the image is divided into a grid and predictions are made for each class that the network has been trained to evaluate for. The dimensions of these boxes are important because they are eventually scaled by a log-space transformation that scales the pre-set bounding box to predict the size of the object. During training, when the network is prompted to predict bounding boxes, all bounding boxes that have a minimum intersection over union score (IoU) of 0.5 are considered, and the top 6 (if they exist) are selected. From these 6 bounding boxes, the network passes boxes through a non-max suppression function to combine and reduce the prediction to a single bounding box. This final bounding box is compared with the intended label and bounding box, and evaluated for its mean average precision (mAP), which is the standard form for evaluating accuracy of object detection networks[3].

Another significant feature of the Yolo-V3 network are the residual layers that connect each of the back-to-back Darknet-53 backbone networks to prevent information loss between downsampling-upsampling for scale. Residual blocks from layers 36 and 61 perform this transfer of information, and are considered a critical aspect of this network. We suspect that this method has drawn inspiration from the ResNet architecture presented by Microsoft's He et. al in 2015, which won first place in the 2015 ILSVRC classification competition[4]. See **Figure 1** for a visualization of all features described here.

Deep Compression only reports results for AlexNet, Lenet, and VGG, which are relatively out of date and simple image classification networks (deep learning has been advancing quickly!). Deploying novel pruning and weight sharing techniques on YoloV3 will give interesting insights into the tractability of these techniques for a larger network designed to perform objective recognition rather than image classification.

Dataset and Initial Training

The standard dataset for training and validating most high-performance object detection systems is the COCO test-val dataset. This dataset contains over 1.5 million images with highly-refined and peer reviewed labels for 80 object categories. This dataset is massive however, totalling out at over 42.7GB, which is too large for the type of training and validation we are performing here. Instead we use a custom dataset, the PASCAL VOC 2007 and 2012

datasets, which combined is only ~25,000 images and contains bounding boxes and labels for 20 common objects, which is much more manageable for deploying on all three systems.

With this new dataset, the first step in deploying our system is to pre-train it until we maximize performance on the validation test set, VOC 2007 (a subset of 2500 of 25000 total images). The results below in **Figure 2** show the inference performance on this validation dataset as we train our baseline Yolo-V3 network on the training images.

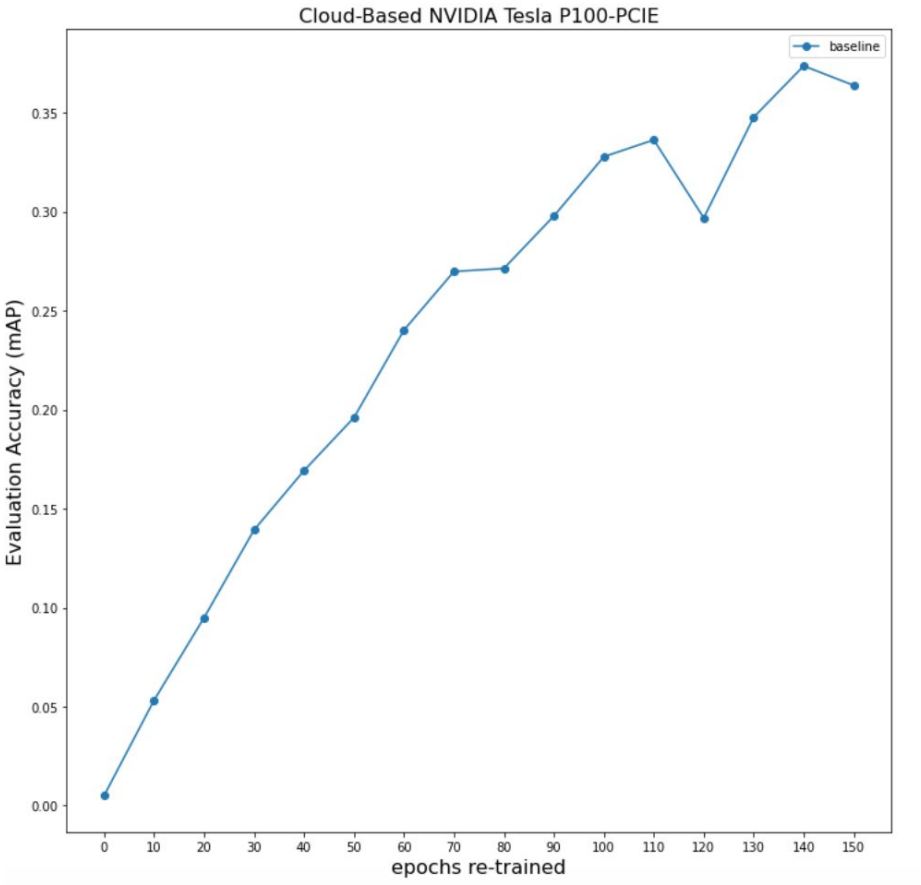


Figure 2: Yolo-V3 baseline training on VOC 2007 & 2012 training datasets on Google Colab Tesla P100

Hardware Platforms

Since all of the size reduction techniques that will be implemented are “symbolic”, the timing statistics are not equivalent to what they would be if the architecture of the network was actually changed. Nevertheless, Yolov3 will be deployed on three different hardware platforms for two reasons- (1) To determine if (and if so, why?) symbolic pruning and weight sharing has an impact on the speed of inference. (2) To try and identify reasons for performance differences

between hardware platforms (specifically with hardware architecture in mind) that run YoloV3 with the implemented techniques.

YoloV3 will be trained and evaluated on the VOC2007 validation dataset on the three different hardware platforms. It is our aim to use these runtime comparisons as a road map to how these optimizations will be applicable to improve runtimes, memory efficiency and accuracy when these systems are deployed on edge devices. Below we provide some system specifications for each of the platforms that we test on.

Platforms:	**GPU Specifications**					
	CUDA Cores	**GPU Clock**	**Comp. Throughput**	**Memory Bandwidth**	**Memory Size**	**Memory Clock**
CPU: Google Colab Server GPU: (Tesla) P100-PCIe	3584	1189 MHz	21,000 GFLOPS	Up to 732 GB/s	16 GB	715 MHz
CPU: Quad Core Intel i7-7700K GPU: (Pascal) 1080 GTX	2560	1607 MHz	8228 GFLOPS	320 GB/s	8 GB	1251 MHz
CPU: Quad Core ARM A57 GPU: 128 Core Maxwell	128	921 MHZ	472 GFLOPS	25.6 GB/s	4 GB	1600 MHz

Table 1: Hardware platform comparison, specifications provided by NVIDIA [4]

Later in this report, we show runtimes for a variety of pruning algorithms on each platform to determine the limiting factors on each system with the pruned Yolo-V3 algorithm. For the 1080 and Jetson, these runtimes can be expected to accurately reflect the memory and compute bound times of these platforms. However, we cannot claim that the same holds for the P100, due to the server latency and complex cache hierarchy that are inherent in server-based computation. The Google Colab server was a great tool during training, but when performing validation/inference, we found that the execution timing was highly dependent on the number of identical iterations that had been previously performed. This observation is shown below in **Figure 3**, which shows the average inference frequency (in fps) over 16 models as we trained 160 epochs to prepare our model for optimization analysis. In the first of 16 runs, the server is only able to evaluate at 1.2 FPS yet by the end we reach our fastest inference and top out at 43.9 FPS. We include inference results from the Colab server later in this report for

completeness, but keep in mind that these are not entirely indicative of the hardware's computational workload potential.

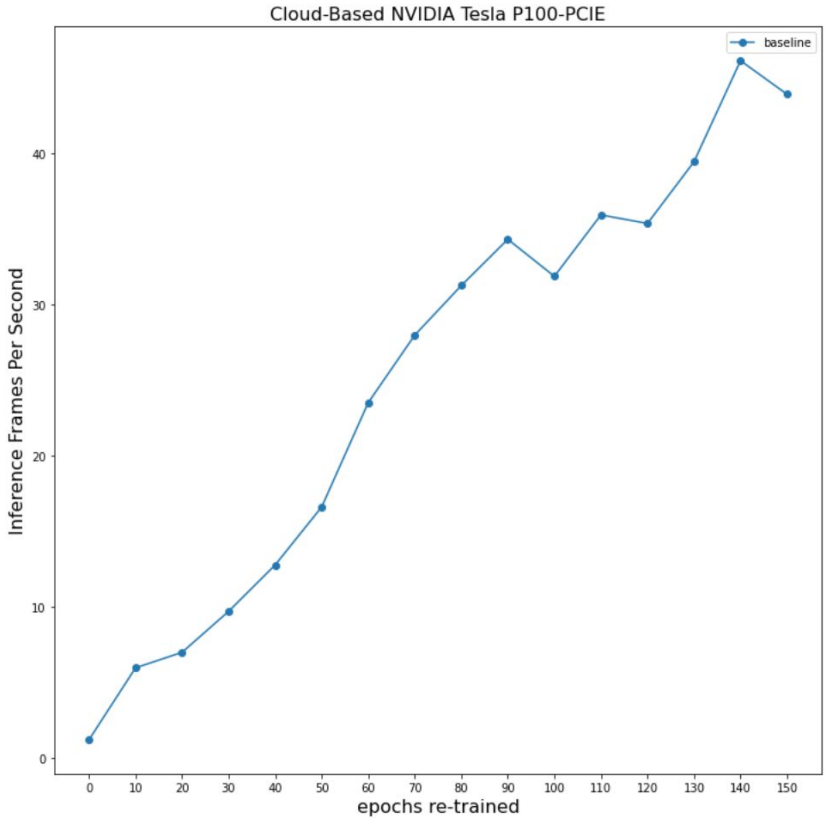


Figure 3: Yolo-V3 baseline inference speed (in fps) on Google Colab Tesla P100, rises with subsequent iterations due to complex caching hierarchy and network latency.

Pruning Accuracy Analysis

In this section we compare the inference accuracy of Yolo-V3 after performing layer-based and global weight pruning on the Conv-2D layers. In total, we ran 6 major iterations of pruning, showing 10, 50, 70, and 90% layer-based pruning and 70 and 90% global-based pruning. We initially ran the four combinations of layer-based pruning to analyze the general accuracy impact of these different percentages, and from these initial findings we observed that 10 and 50% pruning was not necessary to run as a global-prune, because the range where accuracy was significantly affected was between 70-90%. All of our testing results for each platform are provided below in the **Appendix.**

Below, in **Figure 4** we summarize the accuracy results for performing these 6 iterations of pruning and retraining on the network for an additional 50 epochs. After the initial 150 epochs of training, our network was performing at a baseline mAP accuracy of 0.3639. In this plot, we can see that the only pruning percentages that were able to maintain this accuracy were 10% (layer), 50% (layer) and 70% (global). We did not re-run global pruning for 50% because after 30 epochs of retraining, we saw similar performance from the 70% globally pruned network as the 50% layer pruned network (best at the time). Most importantly, we observe that global pruning seems to generally outperform layer-based pruning of the same percentage. At 0 epochs of re-training, we see 70% layer at mAP of 0.277, whereas 70% global performs at 0.329, an ~18% improvement. This margin is maintained throughout retraining. Additionally, we see that the global method for 90% maintains the same 24% margin of improvement over the comparable layer-based method throughout retraining.

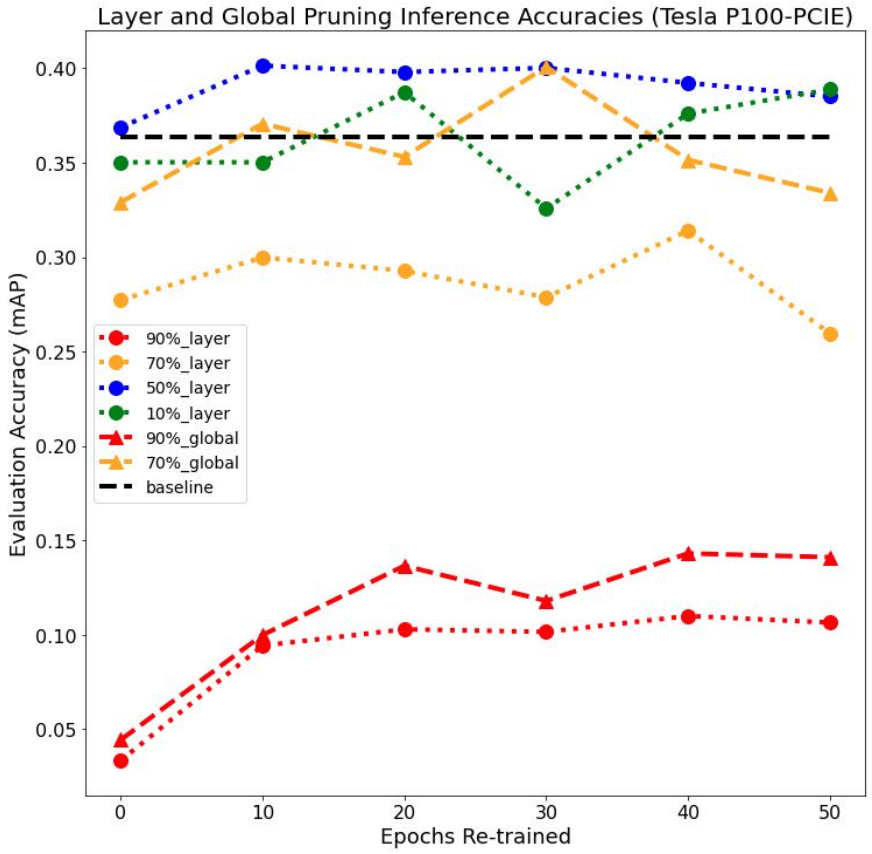


Figure 4: Pruning and retraining shows that global pruning generally outperforms layer-based methods.

Pruning Runtime Analysis

In order to compare the performance of each platform with this network, we ran each of the six pruned model variations on each hardware platform and compared the results in terms of the number of images that inference was computed for each second, frames per second. **Figure 5** below shows our initial results for inference on the Google Colab cloud-based NVIDIA P100 GPU. From these results, we interestingly observed that the inference time decreased inversely to the proportion of pruning that was done on the network with 90% performing worse than the other models. We did not expect this result due to the symbolic nature of our pruning. However, for the Google Colab runs, we cannot entirely trust the computational throughput of the server rack due to caching complexity and none of the prioritization guarantees that we have with local hardware.

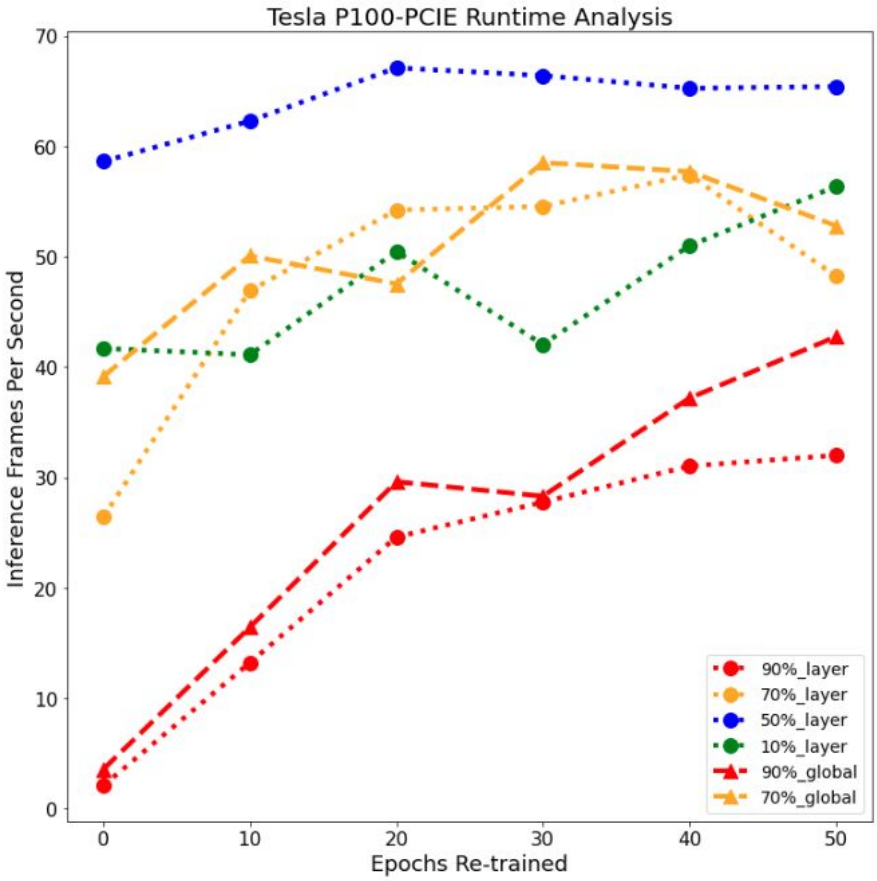


Figure 5: Inference frame rate of Yolo-V3 running on Google Colab Tesla P100

Next, we ran inference on the same models, using a local 1080 GTX GPU. As we suspected would happen, we observed faster inference times for mostly all models (except 50%

pruned by layer). We prove here, that even through the computational throughput complexity of the 1080 is significantly less than the P100 (8,228 vs. 21,000 GFLOPS), by removing internet latency and simplifying the caching/locality hierarchy for computation, we managed to outperform the Colab server consistently with the local 1080. Another interesting observation was that we actually show greater potential to improve inference speed with additional retraining (as seen in **Figure 5**'s depiction of 90% global and local pruning), than improvement in inference accuracy. This behavior remains unexplained currently. We expected the pruned models to generally outperform non-pruned models even with additional parameters for symbolic masking, because of the simplified computational logic in masking most of the values with a multiply by zero. We observe this bump in performance, comparing **Figure 3** with **Figures 5 & 6**.

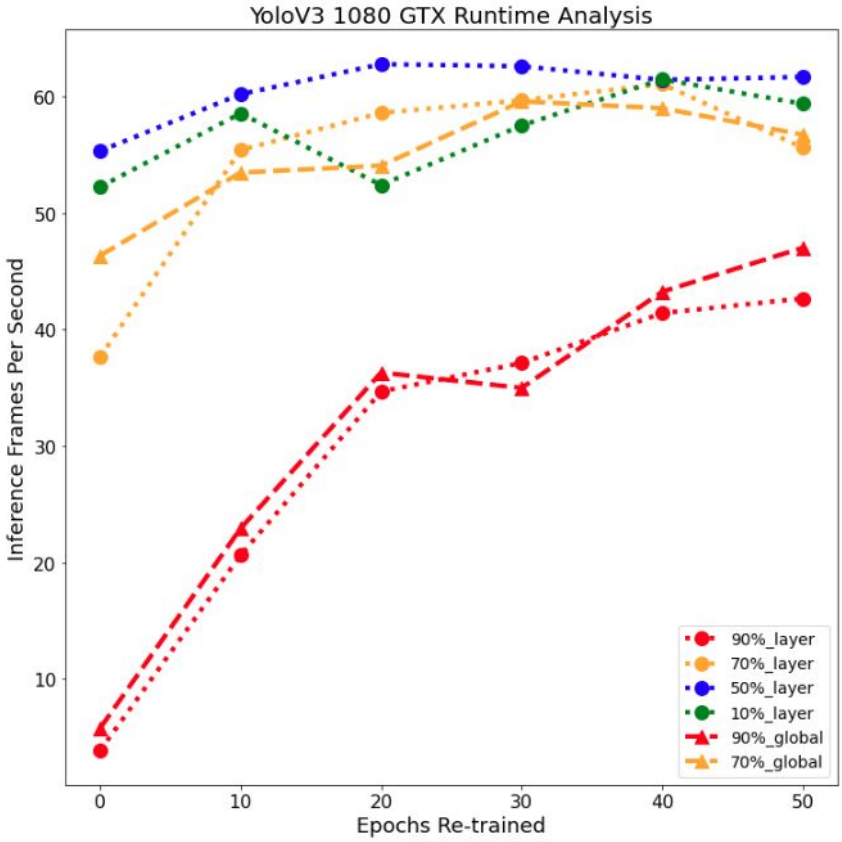


Figure 6: Inference frame rate of Yolo-V3 running on local 1080 GTX

Due to time constraints, we we're not able to run all of our model iterations on the Jetson Nano, as it was evaluating images at a devastatingly slow rate around 1-2 FPS on average, and our validation training set is ~2,500 images. Therefore, we ran only the fully-retrained models

(50 additional epochs) for inference on the Jetson. **Table 2** below summarizes our results with the Jetson.

Model Type (percentage, global/layer epochs retrained)	Inference Frame Rate (frames per second)		
	Jetson Nano	1080 GTX (from figure 6)	Tesla P100 (from figure 5)
10%, layer, 50	1.35	**59.38**	56.36
50%, layer, 50	1.28	61.69	**65.42**
70%, layer, 50	1.09	**55.62**	48.23
90%, layer, 50	1.05	**42.68**	32.01
70%, global, 50	1.33	**52.81**	**52.81**
90%, global	1.36	**47.02**	42.74

Table 2: Comparison of pruned Yolo-V3 networks, after 50 epochs of retraining shows the range of performance on each platform.

Weight sharing Results

Unlike the networks evaluated in *Deep Compression*, YoloV3 does not contain any fully connected layers. As a result, weight sharing is only performed on the convolutional layers. The performance of weight sharing is marked by the resulting compression rate and the achieved mAP by evaluating the "weight shared" YoloV3 on the Pascal VOC dataset. As mentioned earlier, a "symbolic" weight sharing was implemented. The compression rate calculated represents the theoretical memory savings had the weights been stored as they were in *Deep Compression*. This symbolic weight sharing still results in the accuracy that would be seen with "true" weight sharing implemented.

$$Compression\ Rate = \frac{original\ \#\ of\ bits\ needed\ to\ store\ convolutional\ weights}{\#\ of\ bits\ needed\ to\ store\ convolutional\ weights\ after\ weight\ sharing}$$

$$Compression\ Rate = \frac{(\#\ weights) * (32\ bits)}{(\#\ of\ shared\ weights) * (32\ bits) + (\#\ weights) * (indexing\ bits)}$$

First, weight sharing was applied as a standalone technique (pruning was not performed prior). The number of bits used to index into the shared weights array was varied from 4 to 8 (16

to 256 shared weights). On these runs, each layer has its own set of shared weights, and the number of shared weights for each layer was held constant.

# Bits to index shared weights	Evaluation Accuracy (mAP)	Compression Rate
4 (16 shared weights)	0.179739	7.998
5 (32 shared weights)	0.227401	6.38
6 (64 shared weights)	0.359801	5.33
7 (128 shared weights)	0.366143	4.57
8 (256 shared weights)	0.365186	3.99
Baseline	0.363926	1

Table 3: weight sharing analysis scaling number of bits representing 2D-conv layers and analyzing the effect on network inference accuracy.

Far fewer shared weights can be used when applied to convolutional layers in YoloV3 than suggested by *Deep Compression.* Using 6 bits (64 shared weights) resulted in only a negligible drop in mAP. In *Deep Compression*, accuracy dropped when fewer than 8 bits were used for convolutional layers. It is possible that object recognition systems are less sensitive to weight sharing techniques.

Next, a novel technique was devised to achieve additional memory savings while maintaining accuracy. This technique dynamically selects the size of the shared weights array for each layer in the neural network. The intuition being that some layers could sufficiently be represented by less bits than others. Each shared weight array was chosen to be indexed by 5, 6, or 7 bits. This choice was made for each layer depending on statistics about those layers. First, standard deviation was used. If the standard deviation of a layer was among the bottom third standard deviations of all layers, 5 bits was chosen for the layer. If the standard deviation of a layer was among the middle third, 6 bits was chosen. If the standard deviation was in the top third, 7 bits was chosen. Next, the same process was repeated, but with the range (max weight - min weight) of the layer used to determine the number of bits. Finally, this process was repeated once more, except using the total number of weights in the layer to decide the number of bits to select.

Dynamic Bit Selection Technique 5, 6, or 7 bits	Evaluation Accuracy (mAP)	Compression Rate
Stdev	0.3628	5.7000
Range	0.3648	4.9670
Length	0.2292	4.6998
Baseline	0.363926	1

Using the standard deviation and range method, the dynamic bit selection technique resulted in maintaining the baseline accuracy, but improving the compression rate further than selecting a static number of shared weights. Using the standard deviation method, a compression rate of 5.7 was achieved without a drop in accuracy. This sits in between the compression rate when 5 and 6 bits were used for all layers (6.38 and 5.33, respectively). For the Range method, the compression rate was not as high, but accuracy was still maintained. Using length, accuracy dropped to a level similar to when 5 bits were chosen for all layers, but the compression rate was not as high. This makes clear that the standard deviation and range of a layer are far more informative than the size of a layer when deciding how many shared weights should be used for a layer. Since no drop in accuracy was seen for the range and standard deviation method, the number of bits selected were switched from 5, 6, or 7 bits to 4, 5, or 6 bits.

Dynamic Bit Selection Technique 4, 5, or 6 bits	Evaluation Accuracy (mAP)	Compression Rate
Stdev	0.359	6.93
Range	0.356	5.88
Baseline	0.363926	1

This again resulted in even further compression while essentially maintaining baseline accuracy (only off by 0.004 and 0.007 for stdev and range). For the original implementation, when 5 bits was used, the mAP dropped to 0.22 and a compression rate of 6.38. Using the dynamic standard deviation technique, the compression was increased from 6.38 to 6.93, and

the mAP only dropped a negligible amount. This strongly reinforces that using a different number of bits for each layer results in higher compression rates and better accuracy than when using a single number of bits for each layer.

Lastly a global weight sharing method was designed and implemented. It worked by initializing one set of shared weights across all the layers, rather than a different set of shared weights for each layer. Unfortunately, it was not possible to obtain results, as the hardware platforms did not enjoy performing K-means clustering for 60+ million weights.

Combination Results

Pruning and weight sharing was then applied successively to the YoloV3 network. First the network was pruned 70% using the global prune method. Next, weight sharing was applied to the remaining weights in the network. The number of bits used to index the shared weights was varied.

Pruned Percentage	# bits to index shared weights	mAP
70	**2**	0.00816330174
70	**3**	0.1979804792
70	**4**	0.3259599299
70	**5**	0.3336276743
70	**6**	0.3363692174
70	**7**	0.3358983629
baseline	**baseline**	0.363926

Combining pruning and weight sharing led to surprising results. Using only four bits to index into the shared weights for every layer led to a drop of only 0.037 in mAP when compared to the baseline. This is far better performance than reported in *Deep Compression*, where using fewer than eight bits to index into the shared weights resulted in a significant drop in performance.

Future work should be dedicated to determining why performance varied so much between these techniques deployed on YoloV3 vs the classification networks in *Deep*

Compression. Furthermore, researchers should continue to hone the novel weight sharing techniques outlined in this paper. This could be done by finding additional metrics to use to choose the number of weights that a layer should use, and by altering the ratio of layers that are selected to use a particular number of bits for shared weights (for this report, ⅓ of layers were given the smallest number of bits, ⅓ the middle, and ⅓ the highest number of bits depending on stdev, range, or length of the layer. This ratio could potentially be changed to further improve compression).

Hardware Analysis

The first major difference restricting the runtime of Yolo-V3 on each of these platforms is the amount of memory onboard each GPU. For the Colab (16GB RAM) and 1080 (8GB RAM) runtimes, we were able to run these validation tests with a batch size of 8 images in parallel. However, on the NVIDIA Jetson Development board (4GB RAM), we were forced to run using a batch size of 4. By feeding images in parallel, we need to make multiple sets of intermediate parameters for each image passed to the GPU for SIMT computation. Therefore, even if that RAM is large enough to hold all images in the batch and the initial model and its parameters, it may not support all tensors. This can be understood by the size of each model that needs to be loaded from main memory, to the GPU. We see from **Table 4** that the model, model parameters and 8 images will require less than 1.5GB. However, all of the duplicate parameters that are too large for the scratchpad will occupy space in RAM. From trial and error we found that the maximum batch size of 4 can be passed to the JETSON to maximize computational throughput.

Model Type	Model Size	Model Parameters
Original, unpruned	470 MB	235 MB
Global/layer, symbolic pruned	940 MB	470 MB

Table 4: model and parameter sizes before and after symbolic pruning

Next we want to determine generally, whether each hardware platform is computation bandwidth, memory bandwidth or communication bandwidth bound. In order to do so, we first consider the number of operations that must be performed to evaluate a single image with our trained network. In its original form, without any pruning applied, the baseline YoloV3 network

contains 62.6M parameters, and requires 65.71B floating point operations to evaluate. When we apply our symbolic pruning masks to this network, the number of parameters doubles to over 120M, and we estimate a total 130.5B floating point operations to evaluate the network for each image[5].

Because each of our pruned models requires the same number of parameters to represent weights and pruning masks, and therefore essentially the same number of computations to evaluate inference for a single input image, we average the inference frequencies for each device from Table 2 to use as a general throughput measure. Table 5 below shows these averaged values. We then scale these averaged values by the theoretical computational throughput of each device, and compare to the theoretical Yolo inference times. Additionally, we know from previous experimentation that the Google Colab-based Tesla P100 is likely communication bound because each GPU has greater memory and computational potentials than the local 1080, and achieves similar performance in practice.

From this analysis, we conclude that both the Jetson Nano and the 1080GTX are memory bandwidth bound because their observed computational throughput is significantly less than the theoretical maximum. We would expect to perform significantly closer to the theoretical max if there were no other issues, because Yolo-V3 consists of many regular Conv-2D computations, with linear math to determine bounding and anchor boxes. With the masking, we take advantage of a much higher percentage of simple computations as we multiply every weight by 1 or 0 for each 2D Convolutional Layer.

Platform	Average Inference Frequency (Hz) ref: table 2	Average Observed Compute Throughput (GFLOPS)	Theoretical Compute Throughput (GFLOPS) ref: table 1	System Bound
Jetson Nano	1.24	161.82	472	Memory Bandwidth
1080 GTX	53.20	6,942.60	8228	Memory Bandwidth
Tesla P100	49.60	6472.80	21,000	Communication Latency

Table 5: Inference platform bottleneck analysis summary

Representing our ‘edge device’ we are particularly concerned with the slow performance of the full network on the Jetson Nano. Model parameters for our pruned network total up to almost 0.5 GB. The Jetson Nano GPU has only 4GB of GPU main memory. Therefore when the batch size was greater than 4, the Jetson would have segmentation faults likely due to illegal allocations of memory on the GPU. Similarly, on this system the ‘/swp’ partition allocated was also only 4GB. In future experiments, it it worth investigating whether allocating a larger partition for ‘/swp’ would improve performance. Still, we will likely still be memory bound by the small size of the GPU RAM. A more appropriate fix for this issue would be to simplify the network to reduce the total number of computations for single evaluation. After all, when this algorithm is being used in practice, it is fed a steam of images to evaluate before the next arrives. Future optimizations should be performed with a single image.

Conclusion

In this report, we have experimented with tradeoffs between different pruning and weight sharing techniques on a variety of hardware platforms. The results acquired from these experiments shed light on many aspects of neural net memory reduction. First, global pruning was shown to outperform layer-based pruning. Second, it was discovered that fewer bits could be used to index into the shared weights array while still maintaining accuracy than suggested by *Deep Compression* (6 bits vs 8 bits). Third, a new weight sharing technique that dynamically selects the number of shared weights to use in each layer was designed and implemented. This resulted in additional memory savings without a drop in accuracy. Fourth, when combining weight sharing and pruning, the number of shared weights needed to maintain accuracy dropped even further (only 4 bits to index into the shared weights array).

By deploying these techniques on three different hardware platforms it is clear that the performance of the YoloV3 object detection algorithm is largely influenced by the hardware resources available, due to its high computational demands. The runtime investigation into pruning requires much further development- even with the wide range of computational horsepower that we had at our disposal for demonstrating the results of this investigation, we found it difficult to predict the performance of symbolic pruning on our inference speed. While accuracy remained consistent across platforms as expected, it was difficult to predict when speed-ups due to simplications of the math (masking) would outweigh slow downs caused by the additional size (and total computations) that accompanied the symbolic prunes.

References

[1] Han S., Mao,H., Dally, W., Deep Compression: Compression Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv, 2015. https://arxiv.org/pdf/1510.00149.pdf

[2] Redmon, J., Farhadi, H., Yolov3: An Incremental Improvement. arXiv 2018. https://arxiv.org/pdf/1804.02767.pdf

[3] Python Lessons. Yolo v3 theory explained. Medium. July 4, 2019. https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193

[4] Dustin Franklin. Jetson Nano Brings AI Computing to Everyone. NVIDIA Developer Blogs. March 28, 2018. https://devblogs.nvidia.com/jetson-nano-ai-computing/

[5] Zhang, P. Zhong, Y., Li, X. SlimYolov3: Narrower, Faster and Better for Real-Time UAV Applications. Beijing Institute of Technology, https://arxiv.org/pdf/1907.11093.pdf.

Statement of Work

Work was divided and performed evenly by Bryan and Matt. See the following lists of responsibilities.

Bryan

- Yolo-V3 initial setup on google Colab and integration with custom VOC dataset
- Initial pre-training of network + benchmark analysis
- Implementation of layer-based and global pruning methods
- Accuracy and runtime analysis of pruning methods on all three hardware platforms
- Respective sections in report

Matt

- Implementation of all shared weights method
    - Static # bits
    - Layer by layer selection based on stdev, range, length

- Implementation of of combined weight sharing and pruning
- Accuracy analysis of weight sharing and combined weight sharing/pruning methods
- Introduction, deep compression, weight sharing, combined weight sharing, ½ conclusion sections of report

Appendix

Platform: Google Colab - NVIDIA Tesla P100-PCIE			
Model: YoloV3, No Pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.005123	2115.829000	0.841660
10	0.053279	424.159900	0.168230
20	0.094944	361.885600	0.143465
30	0.139281	260.964497	0.103290
40	0.169256	198.482514	0.078424
50	0.196050	152.886918	0.060286
60	0.240251	108.418100	0.042574
70	0.269858	91.422694	0.035773
80	0.271403	81.997120	0.032015
90	0.297875	75.891585	0.029159
100	0.327856	80.527437	0.031402
110	0.336348	71.748712	0.027850
120	0.296941	72.816109	0.028293
130	0.347819	66.580514	0.025366
140	0.373747	65.087551	0.021687
150	0.363926	63.852519	0.022785

Platform: Google Colab - NVIDIA Tesla P100-PCIE			
Model: YoloV3, 90% layer-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.033453	1260.860555	0.474819

10	0.094299	191.374079	0.075594
20	0.102942	103.575415	0.040648
30	0.101611	91.912306	0.036011
40	0.110015	82.491807	0.032223
50	0.106527	80.072756	0.031241

Platform: Google Colab - NVIDIA Tesla P100-PCIE

Model: YoloV3, 70% layer-based pruning

Epochs Trained	Evaluation Accuracy (mAP)	Total Evaluation Time	Avg. Time Per Image
0	0.277266	96.802842	0.037864
10	0.299874	64.287429	0.021312
20	0.292728	63.507498	0.018441
30	0.278766	63.312476	0.018332
40	0.314049	62.677006	0.017417
50	0.259637	63.087641	0.020727

Platform: Google Colab - NVIDIA Tesla P100-PCIE

Model: YoloV3, 50% layer-based pruning

Epochs Trained	Evaluation Accuracy (mAP)	Total Evaluation Time	Avg. Time Per Image
0	0.368723	476.314965	0.017056
10	0.401428	52.520370	0.016053
20	0.398020	52.390390	0.014902
30	0.400240	53.019237	0.015060
40	0.392262	52.430050	0.015324
50	0.385107	52.674382	0.015287

Platform: Google Colab - NVIDIA Tesla P100-PCIE

Model: YoloV3, 10% layer-based pruning

Epochs Trained	Evaluation Accuracy (mAP)	Total Evaluation Time	Avg. Time Per Image
0	0.350259	65.940952	0.023986
10	0.350259	65.403690	0.024320
20	0.387343	64.772025	0.019812
30	0.325825	65.302851	0.023786

40	0.376031	64.570112	0.019612
50	0.388796	63.902751	0.017744

Platform: Google Colab - NVIDIA Tesla P100-PCIE			
Model: YoloV3, 90% global-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.044438	706.091188	0.278728
10	0.099932	154.408403	0.060830
20	0.136612	86.528368	0.033768
30	0.118012	90.390629	0.035326
40	0.143161	69.768911	0.026892
50	0.141140	65.697801	0.023395

Platform: Google Colab - NVIDIA Tesla P100-PCIE			
Model: YoloV3, 70% global-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.329078	1822.979223	0.025535
10	0.370681	67.847296	0.019968
20	0.353116	67.089293	0.021036
30	0.400798	66.548414	0.017086
40	0.351482	67.432030	0.017328
50	0.334031	66.771339	0.018937

Platform: NVIDIA 1080 GTX			
Model: YoloV3, 90% layer-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.033453	649.971736	0.258422
10	0.094299	122.447421	0.048430
20	0.102942	73.211381	0.028827
30	0.101611	68.440729	0.026929
40	0.110015	61.455624	0.024142
50	0.106527	59.672098	0.023434

Platform: NVIDIA 1080 GTX			
Model: YoloV3, 70% layer-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.277266	67.512475	0.026566
10	0.299874	46.097512	0.018046
20	0.292728	43.651199	0.017065
30	0.278766	42.851734	0.016754
40	0.314049	41.906824	0.016374
50	0.259637	45.941696	0.017980

Platform: NVIDIA 1080 GTX			
Model: YoloV3, 50% layer-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.368723	46.165695	0.018074
10	0.401428	42.481402	0.016610
20	0.398020	40.768449	0.015928
30	0.400240	40.917889	0.015974
40	0.392262	41.668069	0.016278
50	0.385107	41.508853	0.016210

Platform: NVIDIA 1080 GTX			
Model: YoloV3, 10% layer-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.350259	48.844478	0.019141
10	0.387343	43.658007	0.017074
20	0.325825	48.702331	0.019081
30	0.376031	44.425769	0.017385
40	0.388796	41.631534	0.016267
50	0.370462	43.056876	0.016840

Platform: NVIDIA 1080 GTX			
Model: YoloV3, 90% global-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.044438	435.555592	0.173039
10	0.099932	110.378724	0.043595
20	0.136612	70.018301	0.027536
30	0.118012	72.650745	0.028575
40	0.143161	58.941726	0.023119
50	0.141140	54.285632	0.021266

Platform: NVIDIA 1080 GTX			
Model: YoloV3, 70% global-based pruning			
Epochs Trained	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
0	0.329078	55.020278	0.021577
10	0.370681	47.815460	0.018699
20	0.353116	47.294956	0.018493
30	0.400798	42.984287	0.016781
40	0.351482	43.412133	0.016949
50	0.334031	45.120113	0.017625

Platform: NVIDIA Jetson Nano					
Models: YoloV3, 6 primary pruning models, 50 epochs retrained					
Pruning Type	**%**	**Epochs Retrained**	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**
layer	10	50	0.371551	1858.329820	0.736714
layer	50	50	0.381950	1818.538397	0.720415
layer	70	50	0.256882	2312.398800	0.915616
layer	90	50	0.105829	2388.000000	0.946288
global	70	50	0.140498	1900.064016	0.751044
global	90	50	0.334615	1850.782997	0.732243

Model: Yolov3 Weight Sharing (independent, no pruning) Raw Results				
Conv Bits	**Evaluation Accuracy (mAP)**	**Total Evaluation Time**	**Avg. Time Per Image**	**Compression Index**
4	0.179739	48.711909	0.019043	7.998
5	0.227401	50.391176	0.019725	6.38
6	0.359801	49.070025	0.019199	5.33
7	0.366143	49.209112	0.019252	4.57
8	0.365186	49.260447	0.019270	3.99
Baseline	0.363926	n/a	n/a	n/a

Combined Pruning and Weight Sharing Raw Results				
Conv Bits	**pruned %**	**mAP**	**Total Eval Time**	**Average Time Per Image**
2	yolov3_ckpt_70percglobal_50.pth	0.00816330174	40.67915344	0.01582016203
3	yolov3_ckpt_70percglobal_50.pth	0.1979804792	44.69784069	0.01744933237
4	yolov3_ckpt_70percglobal_50.pth	0.3259599299	47.66325235	0.01862479035
5	yolov3_ckpt_70percglobal_50.pth	0.3336276743	47.60053086	0.01860350627
6	yolov3_ckpt_70percglobal_50.pth	0.3363692174	47.82790184	0.01868716158
7	yolov3_ckpt_70percglobal_50.pth	0.3358983629	49.15561128	0.01919865845