Due March 24th, 2020

Project 3 - YoloV3 Network Pruning
ECE209 AI On Chip Winter 2020

Bryan Bednarski 005428092
Matt Nicholas 305431314
GitHub: https://github.com/bbednarski9/AIonchip_winter2020_P2-3_Repo

1. Introduction

Object recognition is an exciting field that is evolving at a rapid pace. This field deals with simultaneously classifying objects within an image, while also drawing a bounding box around said objects within the image. Much progress has been made in recent years in creating new networks and integrating them into platforms capable of performing highly accurate, real-time object recognition. However, achieving a high accuracy and speed requires both immense compute power as well as a large amount of memory. This is because the networks that are used to perform object recognition execute huge amounts of floating point operations, and the weights and biases of these large networks must all be stored in memory. This becomes an issue when one desires to deploy an object recognition network in so called 'edge devices'. Usually these devices are low power, lack cutting edge compute power, and have limited memory capacity. As a result, much research has been directed towards methods to reduce the computational and memory loads of these object recognition networks, while simultaneously maintaining an acceptable accuracy. This would allow for the deployment of object recognition networks in all sorts of exciting new systems.

The Yolo-V3 network is an example of an object detection system that has many of these constraints in embedded applications. The purpose of this report is to explore methods of size reduction for this network while analyzing accuracy trade-offs. The previous report dove deep into floating point quantization methods. In this document, we explore the effects of pruning, a method for reducing the memory required for storing a network by removing unnecessary weights, biases, and feature map values from the network structure. We implement two types of pruning, percentage-based and threshold-based, and analyze the performance trade-offs. The overall structure of the report is as follows:

- Section 2 provides necessary background information about pruning methodologies, the YOLO-V3 network, and the data set used in evaluation
- Section 3 describes the basic methods by which evaluation experiments are carried out, as well as the environment and setup for carrying out these experiments - giving a background for results presented later
- Section 4 provides algorithmic descriptions for the two pruning implementations

- Section 5 summarizes our results and provides analysis of what was discovered
- Section 6 provides concluding remarks and suggests potential for future work

2. Background

An Introduction to Pruning

The idea of pruning in deep learning, a parent field to object recognition, was inspired by a real biological process called synaptic pruning. Synapses are structures within the brain that allow for the transmission of electrical and chemical signals across the brain. In newborns, the amount of synapses increases until about age two or three. At that point, the unimportant synapses are gradually removed (or pruned) until early adulthood, where the number of synapses may only be about half of what they once were at age 2 or 3.[1] The idea is essentially the same for deep learning networks. Deep learning networks are composed of connected layers for which an input data structure is passed through. Each layer is composed of a series of weights that perform operations on the input depending on the layer type.

As it turns out, not all weights and biases in a network are important to generating an accurate output. Thus, it is possible to remove certain weights from the network while maintaining an acceptable performance. The statute for 'acceptable performance' of the network varies from case to case depending on the goal and operational requirements of the system. Pruning reduces the memory requirements of the network, and often reduces the time it takes for the network to make an inference on an input. As a result, it becomes easier to deploy pruned networks on low power, performance, or memory devices. This is especially important for the field of object recognition, as these networks are larger and take more time to perform inference than simple deep learning classification networks. This is because object recognition networks are responsible for classifying and drawing a bounding box around all objects that appear within an image, rather than only making a single classification for each image.

YOLOv3 Network and PASCAL VOC Dataset

YOLOv3 is a state-of-the-art, real-time object detection network. It works by applying a single neural network to a full image. This neural network divides the image into regions and predicts bounding boxes and probabilities that an object is present for each region. The bounding boxes are then weighted by the predicted probabilities of object occurrence.[2]

For this report, the PASCAL VOC data set is used to train and evaluate the YOLOv3 network (with and without pruning). This data set contains twenty different classes of objects that the YOLOv3 predicts for. The 2007 Train/validation/test sets are comprised of 9,963 total images and 24,640 annotated objects[3]. The Pascal VOC dataset is used because it is much faster to train than other commonly used object recognition data sets, like the COCO data set, which is considerably larger. This is important because, as will be described in Section 3, the experiments that will be run involve many iterations of retraining YOLOv3.

3. Workflow, Prepossessing, and Environment

Workflow

The basic methodology for creating a pruned model are as follows:

1. A pre-trained YOLOv3 model is loaded into the experimental environment. The YOLOv3 model was provided to us after being trained for 27 epochs and achieves a mean average precision (this metric will be explained in Section Five) of 73.2.

2. The model is pruned in an unrecoverable, magnitude-based manner. That is, the weights that are pruned cannot be placed back into the model once they are pruned. Weights are pruned depending on their magnitude. There are two different implementations of the pruning function- threshold-based and percentage based. These two methods for pruning are described in detail in Section Four, but are both a subset of magnitude based pruning.

3. The model is retrained, which updates the non-pruned weights of the model.

4. A decision is made about whether to jump back to step 2, or if the model is acceptable as is.

At the end of this process, a pruned and trained model is ready to be evaluated. The decision of when to stop iteratively pruning and retraining depends on a combination of how accurate the model needs to be, and how much memory needs to be saved. For the purpose of these experiments, a model is saved after every epoch so that performance trends over time can be analysed.

Pre-processing and Environment

The pre-processing, training and analysis performed through the remainder of this report, was done in a Google Colaboratory .ipynb notebook mounted to a shared Google drive (except in experiments where it is noted otherwise). The downside of this approach is the slow memory access times within Google Drive for saving and accessing large data sets. However, by using the Google Colab Pro GPU access, we were still able to achieve a massive speedup in training and inference from a single thread CPU-based approach.

Running Python 3.6.9 we ran our GPU enabled code through a remote NVIDIA TESLA P100-TESLA PCIE GPU, driver version 418.67, NVIDIA CUDA V10.1. Overall, this approach enabled us to run training epochs in 2 hours, 21 minutes for the first iteration, and 45 minutes for subsequent iterations. Evaluation on average ran for about 8 minutes. We determined that collecting the magnitude of results presented in this report would have been impossible using a CPU-based system. In our original attempts, training with a CPU required over 12 hours for a single training iteration. The GPU speedup can largely be attributed to the batch sizes of 8 images that we feed to the network simultaneously.

Before running training or evaluation, the Pascal VOC data set must first be loaded, and the voc_label.py script run. This script both relabels paths and converts bounding box annotations to the new image scales for training and evaluation. The script must be run on each of the 5 Pascal VOC data sets that are used for this assignment: 2007 Test, 2007 Train, 2007 Validation, 2012 Test, 2012 Train. Bounding box and image size annotations must be converted, as images are pre-processed according to the Yolo-V3 image input requirements before being fed into the network. This process is described in detail in our earlier report titled 'Project 2 - Yolo-V3 Network Quantization', whereby and input image is zero padded to a square and cropped to a length and width of 416, then resized to 1x416x416x3 before being fed to the initial layers of the Yolo-V3 network.

It is important to note that a common error continues to be triggered when evaluating networks with significant pruning. The error states: "AttributeError: 'NoneType' object has no attribute 'cpu'" when evaluating the network. We believe this error is attributed to pruning out a necessary data path for image evaluation, and only occurs when a vast majority of a model has been pruned.

4. Pruning Implementation

In a typical pruning framework, a weight or bias node that is determined to be pruned is removed from the network entirely. This process reduces the overall memory required to store the network, but also could have some negative side effects in memory regularity as it becomes more difficult to store contiguous memory blocks of different sizes. In the case of this experiment, a symbolic prune is performed by setting the value of all pruned weights to zero. This approach allows the network to act in the same way as if the pruned weights had been removed from the network. In reality, the weight is not actually removed from memory, and instead all pruned weights are still stored in the host device with a value of zero.

This method of symbolic pruning poses two distinct advantages: It is simple to implement, and it gives a reliable measure of the accuracy for a network in which the actual weights are removed. This symbolic nature of pruning also poses some disadvantages. First, it is difficult to determine how much memory will actually be saved when true pruning is implemented. At the end of this report, we do our best to quantify these results by compressing pruned and unpruned models to show how memory can be condensed, since compression algorithms can efficiently compress zeroed values. But these results not reflect the memory reduction that occurs when weights are actually removed from the network. Second, it is difficult to deduce what effect true pruning may have on the speed of inference in a network. A timing analysis of these symbolic methods is presented in the results, but they also do not reflect how a network would perform if weights were actually removed.

Two separate symbolic methods for pruning the Yolo-V3 network were implemented in this assignment: *threshold-based pruning* and *percentage-based pruning*.

Threshold-Based Pruning

Threshold-based pruning is implemented by selecting some magnitude for which all weights less than that magnitude will be pruned. This magnitude is selected before the run-time of the program that trains/prunes the model, and it is static throughout execution. The basic steps for implementing this function are as follows:

Algorithm 1: Threshold-Based Pruning

```
1: threshold = 'some value' {set threshold to desired value}
2: for layer in YoloV3 do
3:     mask = abs(layer) > threshold {Bitwise mask for each layer}
4:     pruned_layer = mask * layer
5:     replace layer with pruned_layer in YoloV3
6: end for
```

Before even using this method on a real network, it is possible to see a few possible advantages and disadvantages. The most apparent disadvantage is that an acceptable threshold must be determined experimentally. Without examining all the weights, it is not possible to know how much of the network will be pruned. Additionally, by having a static threshold for pruning, it is possible that certain layers are disproportionately pruned. Some layers with low weights could be almost entirely pruned, while other layers could be left entirely as is. Depending on the situation this could be either an advantage or a disadvantage. It could be an advantage in a situation where certain layers that are of high importance are rightfully left as is. The magnitude of a weight is generally proportional to its predictive power in the network, and high magnitude weights will never be pruned using this method (with proper threshold choice). However, on the other hand, if the threshold is too high, or if a layer is composed entirely of low magnitude weights (relative to other layers), it is possible that too much of some layer is pruned (that is composed of weights that are nearly all below the threshold). The resulting model will suffer a far greater loss in accuracy than if the same amount of weights were pruned, but spread more evenly across the entire network. Another advantage of this implementation is its simplicity of implementation, and its relative speed compared to the next method (which involves sorting). An evaluation of this pruning method and an analysis of these advantages and disadvantages will be carried out in section four.

Percentage-Based Pruning

Percentage-based pruning is implemented so that the programmer has more control over exactly how much the network is Pruned. Instead of using a static threshold for every layer, a new threshold is determined for each layer in the network. This results in a predefined fraction of the weights in each layer being pruned. This is implemented as follows:

Algorithm 2: Percentage-Based Pruning

```
1: perc_prune = 'some percentage' {set to percentage desired}
2: for layer in YoloV3 do
3:     flatten and sort weights in layer- Store in a 1d tensor layer_altered
4:     threshold_index = perc_prune $*$ length(layer_altered)
5:     threshold value = layer_altered[threshold_index]
6:     mask = abs(layer) $>$ threshold {Bitwise mask for each layer}
7:     pruned_layer = mask $*$ layer
8:     replace layer with pruned_layer in YoloV3
9: end for
```

The advantage of this implementation is that each layer is proportionally pruned the same amount. This ensures that there is no layer that is pruned a far more than another, which could act as a bottleneck in the network, and drastically reduce accuracy (this situation is possible in the previous implementation). However, it is possible that this method could perform worse than the previous method in certain situations. For instance, sometimes it could be essential to keep an entire layer in tact if it is composed entirely of high magnitude weights (relative to other layers in the network). In this case, the implementation above would still prune those higher magnitude weights in that layer, even though it may result in a more accurate model if that layer was left as is, and additional lower magnitude weights were pruned in other layers. However, if given a priori knowledge of which layers should not be pruned, this implementation could be easily altered to exclude those layers. Additionally, this problem could also potentially be dealt with by implementing a clamping function that does not allow for a weight to be pruned above a certain threshold. However, if this were the case, then the pruning function would not be pruning the amount the user specified (the function is passed a percentage to be pruned), and additional weights would need to be pruned in other layers to reduce the model to the size the user specified.

5. Results and Analysis

The experiments that were carried out fall under two categories- pruning without retraining, and pruning with retraining. For each of these methods, we iterate through a number of different threshold and percentage-based pruning function calls, to demonstrate our results and analysis. We use two different metrics to define the success of our model at these different stages: mean average precision (mAP), and validation loss.

Primarily, the Yolo-V3 network's performance is evaluated using a mean average precision (mAP) with a minimum intersection over union of 0.5. This evaluation criteria is discussed in depth in our previous report titled 'Project 2 - Yolo-V3 Network Quantization'. The mAP evaluation criteria requires the label around an object that is labeled in the training data to be correct, in addition to the bounding box placed during evaluation having at least 50% overlap with the bounding box in the labeled original. If these criteria are not met for an object in an image that is labeled in the training set, the failed result is averaged into the results for the image as a whole.

We additionally use intermediate validation loss as another evaluation criteria during training. This metric is used to evaluate the network's accuracy on the data in the training set while the network is training. Typically cross-validation splits the training data into temporary subsets of randomized training data, and validations data. Images that are designated as validation data in one training step, can be training images in the next, and visa versa. A negative side effect of this is that validation loss can misguide the engineering to over fitting the network, as it continue to decrease while evaluation results may be worsening.

First, the results for pruning without retraining will be presented. This means that the pre-trained model was loaded, pruned, and then evaluated without ever retraining.

Table 1: Percentage Based Pruning

Percent Pruned	mAP (mean average Precision)
0%	76.5
5%	68.4
10%	36.8
20%	3.4

Table 2: Threshold-Based Pruning

Threshold	Resulting % of Weights Pruned	mAP (mean average Precision)
0	0%	76.5
1e-6	5.2%	76.5
2.5e-6	9.99%	76.5
6.5e-5	19.9%	76.5
5e-3	50%	76.5
1e-2	65%	76.1
2e-2	83%	60.6
3e-2	90%	30.7

The first three thresholds in *Table 2* were chosen so that it resulted in pruning the same amount of weights as the percentage-based pruning (5%, 10%, and 20% respectively). This allows for a direct comparison of the threshold-based method and the percentage-based method. It is surprising how much better the threshold-based pruning performed than the percentage-based. Using a threshold that pruned 20% of the weights resulted in identical performance to a un-pruned model, while pruning 20% using the percentage-based method led to a drop from 76.5 mAP to 3.4 mAP. At 10% and 5% the threshold-based method again resulted in identical performance to the original model, while the percentage-based method achieved an mAP of 36.8 and 68.4, respectively. Since the threshold-based method of pruning at 5, 10 and 20 percent maintained the same mAP as the original model, additional experiments with higher thresholds were carried out to determine when the performance begins to drop. From the experiments, the mAP did not begin to drop until 65% (1e-2 threshold) of the model was pruned, and there was still respectable performance when 83% (2e-2 threshold) of the model was pruned.

The high performance of the model after threshold-based pruning suggests that there is a huge number of extremely low magnitude weights that do not hold a high predictive power. It is important to note that this does not mean that the YOLO-V3 network is unnecessarily large, and that it always contains a plethora of unneeded weights. It means that this particular model, trained and evaluated using this methodology, in this environment, resulted in a large number of unneeded weights. It is possible that if it were trained or evaluated using different methods- whether that be a different optimizer, batch size, or even data set (like COCO), that a far higher percentage of the weights would need to be utilized to make accurate predictions.

There is a stark difference in performance between the two methods of pruning, shown in *Table 1* and *Table 2*. This could be attributed to a number of different factors. As mentioned in Section 4, a possible disadvantage of the percentage-based method is that it would prune high magnitude weights if there were layers that were composed entirely of high magnitude weights (relative to other layers in the network), since it works by pruning 'x'% of every layer. After examining these results, a manual inspection of the weights of various layers within the network was carried out. Through this manual inspection, certain layers were detected that had significantly higher weights than most layers, which supports that hypothesized disadvantage of percentage-based pruning in Section 4. To confirm this hypothesis a statistical analysis for the 10% percentage-based method was carried out. A script was created that kept track of the threshold that was chosen for each layer in the network when percentage-based pruning was executed. The median threshold was 0.0065, and the mean threshold was 0.1658. If this median and average were used as the threshold for threshold-based pruning, it would result in pruning 55% and 99% of the weights in the model, respectively. Two primary conclusions can be drawn from this- it further supports the hypothesis that certain layers contained almost entirely high magnitude weights, and that these high magnitude weights have high predictive performance. Next it can be concluded that it is for this reason that percentage-based pruning performs worse than threshold-based pruning: percentage-based pruning will always prune the specified amount from a layer, even if it contains entirely high importance, high magnitude weights.

Seeing the relative poor performance of percentage-based pruning when compared to threshold-based pruning, it was decided to perform iterative pruning and retraining using percentage-based pruning to see if the mAP could be improved (this methodology is outline in Section 3). The following is a graph which shows the mAP and validation loss after every epoch of training for 10% percentage-based pruning. Note- a total of 10% of the model is pruned during iterative training (not 10% every pruning call).

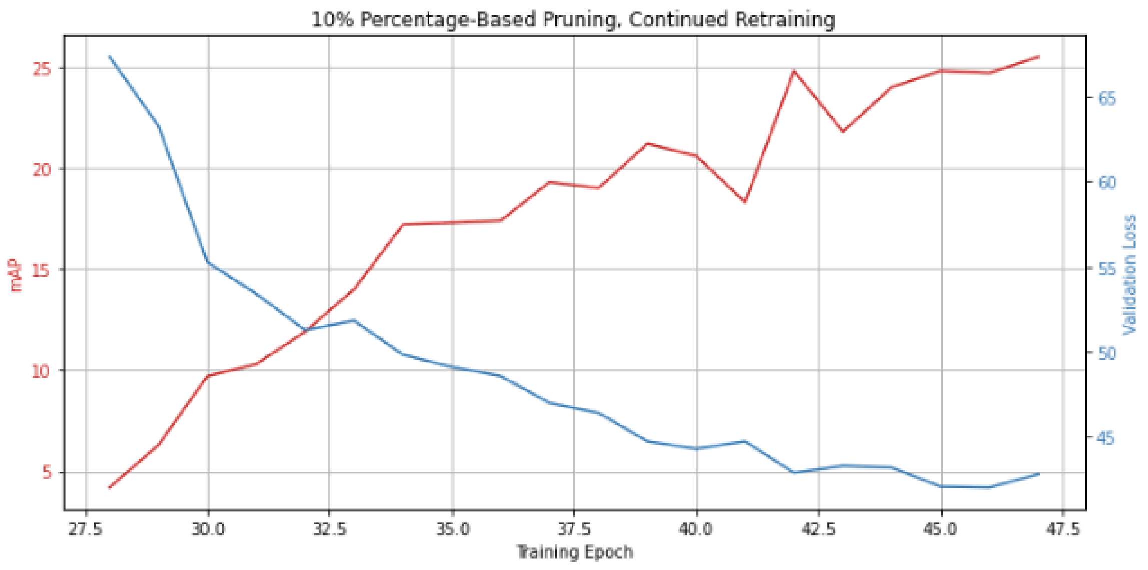

Figure 1: mAP and Validation Loss for 10% percentage-based iterative pruning and retraining.

From the results shown in 1 we see that the mAP improves rapidly at each epoch at first, before gradually reaching its peak at around epoch 45. Similarly, the validation loss decreases rapidly, before it too levels off at about epoch 45. What is surprising about this data, is the massive drop in accuracy at the first new epoch (epoch 28). After one epoch of pruning and retraining, the mAP dropped all the way to 4.2. Seeing this result in solidarity, it would be easy to attribute this to the pruning. It is not unbelievable that pruning 10% of a model and only retraining for one epoch could result in a large drop in accuracy. After all, after many epochs of training, the mAP reaches a respectable level of about 25. However, with the added context of *Table 1* above, the big drop in mAP at the first epoch (which is epoch 28) is highly unexpected. One would expect that with one epoch of pruning/retraining, an mAP $\geq$ 36.8 would be achieved. This is because the 10% result from *Table 2* were obtained by pruning the same model with the same method, but no retraining was performed. It does not make intuitive sense that a model that was pruned and immediately evaluated would perform better than a model that was pruned the same amount and then retrained as well, but that is the situation that occurs here. Furthermore, it would make sense that further retraining would result in a mAP far higher 36.8, but it only ever reaches 25.

Upon this realization, another experiment was carried out, this time performing iterative threshold-based pruning and re-training. The goal was to determine whether this drop in mAP was repeatable across different pruning methods. This time, the pre-trained YOLOv3 model was pruned using threshold-based pruning with a threshold of 2.5E-6, which prunes 9.99% of the weights from the pre-trained model.

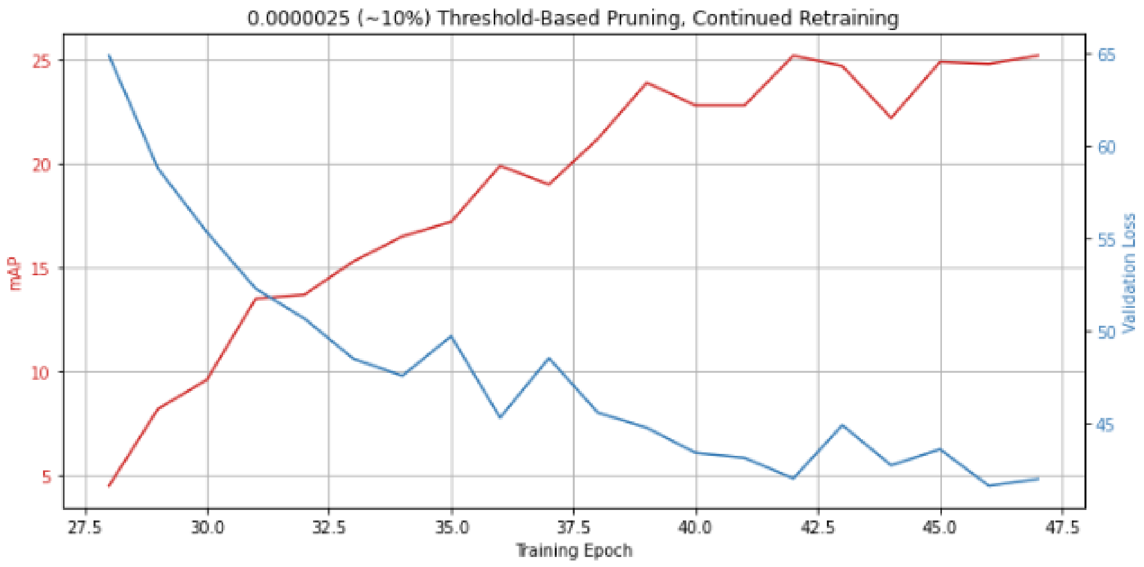

Figure 2: mAP and Validation Loss for 0.0000025 (10%) threshold-based iterative pruning and training.

The results in Figure 2 mirror those in Figure 1. The pre-trained model achieves a mAP of 75.6, but after pruning and retraining for one epoch, the new model achieves a mAP of only 4.5. Just like the past experiment, the mAP rises similarly every epoch trained. Once again the same counter-intuitive result is observed here: The model that is pruned and retrained performed worse than the model that was pruned, but not retrained at all. Furthermore, for this example, using threshold-based pruning with 2.5E-6 as the threshold and no retraining, the same mAP as the original pre-trained model was achieved. These results spurred a thorough examination of the training process in order to find the cause of the drop in accuracy when retraining is performed.

Particular layers within the network were printed out at two parts of the program. First, after the weights were updated in the training step (after Optimizer.step() is called), and second, after the pruning function was called. It was noticed that the weights were properly being set to zero as specified by the pruning function call, but, after the training update, those weights that were set to zero previously were actually still updated to a small value (often times the value of the learning rate). According to the assignment specification, setting the weights to zero stopped the gradients in the optimizer from updating those pruned weights. However, by printing out the actual weights in the specified locations, it was observed that this was not the reality. Upon further investigation, it was realized that the training environment (which was cloned from a given GitHub repository and not written for this project) was using the ADAM optimizer. The reason that setting the weights to zero did not result in the cessation of updates for those weights is because the ADAM optimizer accumulates gradients from previous steps. Thus, the gradients from past epochs (before pruning) still influence the optimizer, and so the pruned weights are still updated after being set to zero. While this was clever detective work, it was not clear that this update in the pruned weights were the reason for the drop initial drop in accuracy when the model was retrained. In fact, it likely had very little effect on the performance. The pruned weights were updated with such small values that they are always immediately pruned again when the next pruning function was called. Nevertheless, switching the Optimizer to one that does not accumulate past gradients could fix this issue, but since switching would have required retraining and evaluating all of the past data, and it likely wouldn't have improved performance, it was not implemented due to timing constraints.

Instead, in order to confirm the suspicion that this slight flaw in the optimizer was not responsible for the large drop in mAP when first retraining, a new experiment was designed. This time, the pre-trained model was loaded and trained for more epochs without pruning- that is, when the pruning function was called, the weights were left as is.

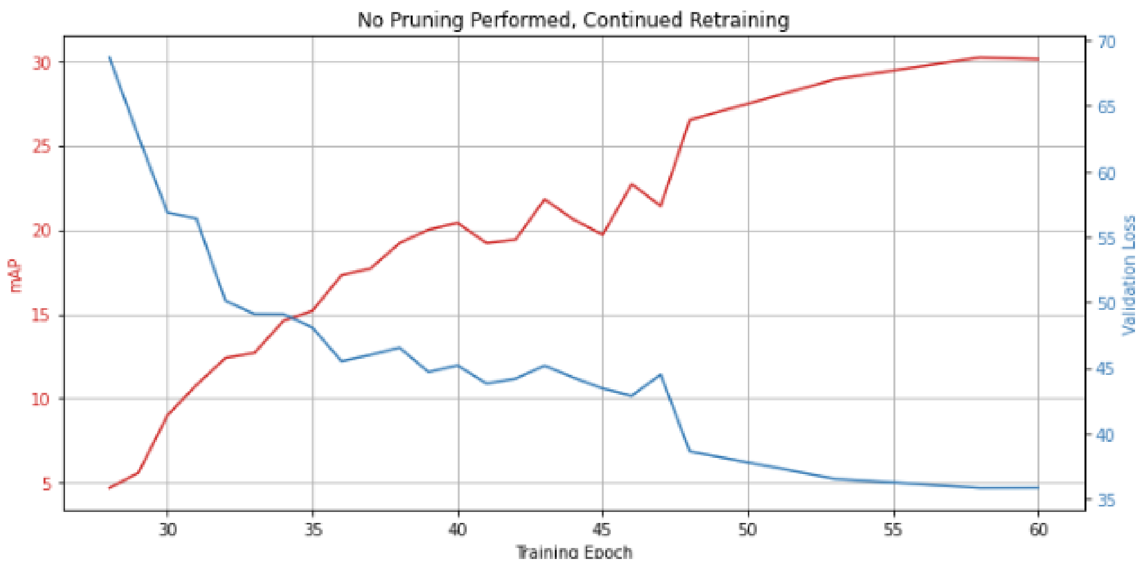

Figure 3: mAP and Validation Loss Results for no pruning pruning and additional training.

The results in Figure 3 confirm what was hypothesized previously. The slight flaw in the optimizer was not what was causing a degradation in the retrained models. The pre-trained model was loaded and subsequently trained further for multiple epochs with no pruning, but the same drop in accuracy from the pre-trained model to the next epoch was seen, despite no pruning. This makes clear that both the pruning implementation and the optimizer flaw were not responsible for the drop in performance when re-training. Unfortunately, it only clarifies that there is a serious issue with the training setup that was inherited via this GitHub, rather than point out what this issue may be. Since the project was not meant to deal with implementing the actual training environment, excellent results had already been gathered from pruning with no retraining, and because of limited amounts of time, it was decided to continue into other aspects of the analysis rather than spend additional time on this issue.

Next, experiments were designed and carried out that analyzed the impact that the pruning implementation had on timing (for both CPU configuration and GPU configuration) and on memory.

Timing Improvements

In order to gauge whether the symbolic pruning techniques had an impact on the speed of inference, evaluation was timed for the first 20 batches of images using a variety of pruning parameters. These evaluation were performed on both a CPU and GPU (in Colab) configuration, and are compared to the evaluation time for the pre-trained model with no pruning.

Table 3: Timing Analysis (20 batches) Threshold-Based

Threshold	Resulting % of Weights Pruned	CPU Time (s)	GPU Time (s)
0 (pre-trained, no pruning)	0%	390	4.128
1E-6	5.2%	30	3.60
2.5E-6	9.99%	28.9	4.35
6.5E-5	19.9%	28.7	4.20
5E-3	50%	28.8	3.65
1E-2	65%	28.8	3.81
2E-2	83%	28.9	5.90

Table 4: Evaluation Timing Analysis (20 batches) Percentage-Based

Threshold	CPU Time (s)	GPU Time (s)
0 (pre-trained, no pruning)	390	4.128
5%	314	3.20
10%	213	3.96

For threshold-based pruning on a CPU, the speed of performing inference increases dramatically when pruning is implemented. The time it takes to evaluate 20 batches of images drops from 390 seconds with no pruning, to about 28 seconds with pruning. The value of the threshold during pruning does not seem to alter the evaluation time, as it takes around the same amount of time with a threshold of 1E-6 to 2E-2. For percentage-based pruning on a CPU, there is also a noticeable speedup, albeit not as dramatic as with threshold-based. Evaluation time is reduced from 390 seconds to 314 seconds at 5% pruning, and 213 second at 10% pruning.

When evaluating on the GPU, there is no difference in the speed of inference when pruning is implemented using percentage-based or threshold-based. This could be because the bottleneck for timing with the GPU on Google Colab is the latency sending information to and from their servers, rather than the actual compute time on the GPU. If others attempt to recreate these timing results, it is important to note that in the Google Colab environment, when running an evaluation for the first time on an instance, the evaluation is far slower than on subsequent runs. It is likely that this is a result of long memory latency when accessing the images/labels/weights/etc from the Google Drive. On subsequent runs this is sped up likely in part because this data is stored in a higher hierarchical memory location with lower latency access times. Thus, when attempting to recreate these timing statistics, make sure that timing data is not gathered from the first evaluation run on a Google Colab instance.

These timing statistics could be useful to someone who wishes to deploy the YOLO-V3 network in a CPU system with strict timing requirements. These results suggest that implementing these symbolic pruning techniques would help speed up the evaluation. However, it is important to note the shortcomings of these results. First, the evaluation uses batches of images, rather than single images from a video (which may be the case in a real-time system). Second, the results could vary widely depending on the specific CPU and system used. Third, the variability of timing for evaluating single images was not analysed, so it is unknown if the upper bound on evaluation time is the same as the pre-trained network. Even with the shortcomings stated, it is likely that these symbolic pruning methods could help speed up performance in a real system, and it is an area of work which should be pursued further. These statistics do not reflect timing differences that may be seen when true pruning is implemented on a network (the weights are removed from memory, which could drastically affect memory regularity). Those timing results would be dependent on the implementation of the pruning algorithms, and the hardware used to evaluate.

Next, the improvements in memory were analyzed using the different pruning methods.

Memory Improvements

A number of pruned models were saved after different epochs of training in order to analyze their size.

Pruning & Retraining Status:	**no compression**	**bzip**	**gzip**	**xzip**
Epoch 27, Un-Pruned:	246.8MB	237.3MB	231.2MB	227.8MB
Epoch 28, Percentage-Based (10%):	246.8MB	215.2MB	212.5MB	207.3MB
Epoch 47, Percentage-Based (10%):	246.8MB	215.7MB	214.2MB	207.9MB
Epoch 28, Threshold-Based (10%):	246.8MB	210.7 MB	207.6 MB	204.2 MB
Epoch 47, Threshold-Based (10%):	246.8MB	**175.5 MB**	**175.8 MB**	**170.6 MB**

The table above shows the results of a simple compression analysis on the trained Yolo-V3 network models. There are a number of things that can be extrapolated from these results. First, we can see that we are in fact doing a symbolic prune. All models, whether pruned or retrained are the same size before compression. In this compression analysis, any 32-bit weights that have been pruned to 0 can be easily compressed to a smaller bit-width, and we see the effects when using 'bzip2', 'gzip' or 'xzip' generic lossless compression algorithms. From this, we can determine that for percentage-based pruning, the number of weights that are pruned in the first iteration remain pruned 20 epochs later as the size of the files at epoch 28 are the same as epoch 47. Conversely, we can see that for threshold-based pruning, we continue to shave off unimportant weights and biases in successive pruning and retraining iterations, as the compressed model for epoch 28 is larger than that for epoch 47. These results confirm that our pruning functions are working as expected. While these results do not reflect the actual amount of memory that would be saved in a real pruning implementation, where weights are removed from memory, they do give us a relative comparison for size reduction that we could expect from a non-symbolic pruning methodology.

6. Conclusion

In this report, two methods of symbolic pruning are designed and implemented on the YOLOv3 object recognition network. Threshold-based pruning works by setting all weights within the network below a certain threshold to zero. Percentage-based pruning works by dynamically selecting a threshold for each layer, such that the same percentage of weights are set to zero in each layer in the network. Both of these pruning implementations were evaluated using two general methods. First, pruning was performed on a pre-trained model with no retraining. Second, iterative pruning and retraining was performed on the same pre-trained model (evaluated at every epoch). It was concluded that the threshold-based method performed far better than the percentage-based method when there was no re-training. When there was iterative pruning and retraining, a fault in the inherited training environment was discovered that resulted in a significant decrease in performance when retraining for the first epoch. However, after retraining for many epochs, a respectable accuracy was once again achieved. An analysis of the two pruning methods effects on the speed of evaluation was carried out, and it was shown that the threshold-based method of pruning resulted in better evaluation speed. Finally, a memory analysis was performed using both pruning methods, which confirmed pruning functionality, and gave an idea of the relative reduction in size on a real system.

The results in this report offer a comprehensive analysis of the two symbolic pruning methods. Going forward, three areas of future work that build directly upon this report have been identified: First, the fault in the inherited training environment (which was supplied for this report) should be identified and fixed. Second, it would be exciting to implement these symbolic pruning functions in a real system, and compare results with this report. Third, it would be interesting to implement 'real' pruning function (where weights are removed from memory), and again compare those results with this report.

References

[1] I. of Medicine, N. R. Council, From Neurons to Neighborhoods: The Science of Early Childhood Development, The National Academies Press, Washington, DC, 2000. doi:10.17226/9824.

[2] J. Redmon, A. Farhadi, Yolov3: An incremental improvement, arXiv.

[3] M. Everingham, Pascal voc dataset 2007.
URL `http://host.robots.ox.ac.uk/pascal/VOC/`