Neural Pruning with K-Medoids

Rahul Malavalli
rahul@cs.ucla.edu
rahul.malavalli@gmail.com
University of California, Los Angeles
Los Angeles, CA, USA

ABSTRACT

Especially in recent years, technological advancements and the increasing prevalence of larger and larger datasets have ushered in similarly deeper and wider neural networks capable of handling such tasks. Alongside immense performance gains, however, compute time and costs associated with the training, storage, and evaluation of these models have also skyrocketed. As a result, not only are common consumer devices like mobile phones incapable of utilizing these large models, but general training and study often proves prohibitively expensive in compute time and resources. Here, we explore neural pruning with k-medoids to reduce the size of these networks, and potentially improve evaluation and training costs as well. Our algorithm applies k-medoids, inspired by the CRAIG coreset selection technique, to each fully connected and/or convolutional layer in a neural network iteratively from the input to the output; pruned weights are removed entirely from the model. Across both the LeNet-300-100 (fully connected) network trained on MNIST and the VGG16 (CNN) network trained on CIFAR-10, we see that our method consistently performs better than the random baseline after pruning and before any finetuning. After finetuning of LeNet-300-100, we see only a total drop of ~4 points in accuracy at the compression ratio of ~90%; fully connected layer pruning in VGG16 also sees a ~4 point drop in accuracy but at ~82% compression ratio after finetuning. These results suggest that neural pruning with k-medoids would best serve for fully connected layer pruning in situations when finetuning is infeasible, since the algorithm effectively chooses a subset that minimizes immediate loss in accuracy in both tested architectures.

1 INTRODUCTION

Aided by the improving computational capabilities of technology like the graphics processing unit (GPU) and the need to understand an ever-increasing set of big data, neural networks have taken the lead in artificial intelligence research. In the quest for constant advancement, generational leaps in neural network performance are often accompanied by commensurate jumps in neural network depth and width. To learn the MNIST handwriting dataset of 60,000 images from the late 1990s, for example, even a simple fully connected network with a total of ~400 hidden nodes proved sufficient; early convolutional neural networks (CNNs) like LeNet-5 grew to contain ~60,000 parameters [Lecun et al. 1998]. To learn the ImageNet of over 14 million images introduced in 2009 [Deng et al. 2009], CNNs grew to contain 60 million parameters in AlexNet [Krizhevsky et al. 2012], only to be superseeded by the more than 130 million parameters in VGG16 [Simonyan and Zisserman 2015]. However, these CNNs pale in comparison to the most recent advancements in language modeling, which have ranged from ~345 million parameters in BERT [Devlin et al. 2019], to ~1.5 billion and a whopping ~175 billion parameters in the GPT-2 [Radford et al. 2019] and GPT-3 [Brown et al. 2020] models, respectively.

As expected, these increases in model and dataset size drive a similar increase in the computational time and costs needed to train, store, and use these neural networks effectively. Research in model compression techniques is critical to decreasing the time and costs associated with these models. Advancements in neural pruning carry the potential to not only enable use of miniaturized versions of these models on limited end-user devices like mobile phones and other consumer electronics, but to also democratize the training and study of these models on more cost-effective and commonly available technologies.

To this effect, various model compression techniques have been studied over the years. To reduce the size of a neural network, binarization and quantization techniques use strategies such as binary coding to decrease the number of bytes required to represent each weight; note that the actual number of parameters may remain unchanged. Neural network weights can also be analyzed via some form of low-rank factorization, like Singular Value Decomposition (SVD) and variants applied to individual layers, such that only informative parameters are retained. This type of parameter reduction, as well as other methods like sparsity constraints, may also improve the training and evaluation speed of a neural network, if used in an appropriately optimized environment [Cheng et al. 2020].

One of the earliest ideas to decrease model size involved the actual removal of weights in an attempt to find non-redundant parameters via the "optimal brain damage" acceptable in a neural network [LeCun et al. 1990]; neural pruning is also the primary object of study in this paper. Specifically, our paper focuses on the application of previous work in dataset coreset selection with k-medoids [Mirzasoleiman et al. 2020] to neural pruning, with the aim of decreasing the size of a neural network while preserving its test accuracy.

2 RELATED WORK

2.1 Pruning Techniques

As noted in the introduction, the rising costs and limitations associated with ever-growing neural networks have spawned numerous attempts to reduce neural network sizes and computational times. Many conventional pruning strategies attempt pruning of pre-trained models through a variety of means, usually utilizing some sort of selection or scoring algorithm to decide which neurons, filters, or other functional units to remove. Along this path, other techniques seek to compress models by quantizing network weights [Cheng et al. 2020]. Some research has also established theoretical trade-offs between compression ratio and performance loss with data-independent coreset selections per layer [Mussay et al. 2020].

However, these conventional techniques still require expensive training to be completed prior to pruning; if pruning can be completed beforehand, then training times can also decrease in response to smaller model size. More recent work has shown that some forms of pruning, such as Iterative Magnitude Pruning [Frankle and Carbin 2019][Frankle et al. 2020] and Synaptic Flow [Tanaka et al. 2020], can prune randomly initialized models before training with little loss in test accuracy.

This paper primarily focuses on an approach to prune pre-trained models. However, the algorithm can be directly used for pruning of untrained models, and may be extended to incremental pruning during training, as described in subsection 6.4.

2.2 CRAIG

Introduced by Mirzasoleiman et al., the Coresets for Accelerating Incremental Gradient Descent (CRAIG) algorithm was initially developed to speed up the training of machine learning models and neural networks. To accomplish this, the CRAIG algorithm generates a weighted subset from the original dataset, constructed to most accurately capture the model's training behavior on the full, original dataset; the speed up is then achieved by training the model directly on the smaller CRAIG-obtained coreset. Improvements in training time are most apparent when training on the immensely large datasets common today.

The CRAIG algorithm generates a coreset, S, by finding medoids of the original data, V, in the gradient space based on some similarity matrix, $D \in \mathbb{R}^{|V| \times |V|}$. The coreset consists of all of the cluster centers, where the weight of each center is equal to the number of points in the corresponding cluster. Selection of the next cluster center $c \in V$ is done greedily based on the point c that maximizes the value of $F(S_i \cup \{c\})$, where $F(S)$ is the submodular k-medoids/facility location function defined in Equation 4.

$$CC(x|S) = \operatorname*{argmax}_{c \in S} D[x, c] \tag{1}$$

$$\begin{aligned} CS(c|S) &= \{x \in V \mid c = CC(x|S)\} \\ &= \left\{x \in V \mid c = \operatorname*{argmax}_{c \in S} D[x, c]\right\} \end{aligned} \tag{2}$$

$$\begin{aligned} CW(c|S) &= |CS(c|S)| \\ &= \sum_{x \in V} \mathbb{1}\left(c = CC(x|S)\right) \\ &= \sum_{x \in V} \mathbb{1}\left(c = \operatorname*{argmax}_{c \in S} D[x, c]\right) \end{aligned} \tag{3}$$

$$\begin{aligned} F(S) &= \sum_{x \in V} D\left[x, CC(x|S)\right] \\ &= \sum_{x \in V} \left[\max_{c \in S} \left(D[x, c]\right)\right] \end{aligned} \tag{4}$$

The $F(S)$ in Equation 4 is defined as the sum of the similarities between each original point $x \in V$ and its corresponding cluster center $CC(x|S)$, as defined in Equation 1. This defines each cluster's set, $CS(c|S)$, as the set of points that are more similar to that cluster's center, c, than they are to the center of any other cluster in the coreset S, as described in Equation 2. The coreset weight, $CW(c|S)$, of each cluster center, c, is then equal to the number of points in each cluster, as defined in Equation 3. Intuitively, this coreset is effective because the sum of any weighted subset generally approximates the sum of the full original dataset.

Algorithm 1: CRAIG Pseudocode

```
Input  : $V$ = indices of full dataset
         $D$ = similarity matrix, of size $|V| \times |V|$
         $B$ = target coreset size
Output : Coreset $S$ of size $B$, with weights $W$
1 func CRAIG($V, D, B$):
2     $S \leftarrow \emptyset$
3     while $|S| < B$ do
4         $S \leftarrow S \cup \operatorname{argmax}_{x \in (V \setminus S)} F(S \cup x)$
5     end
6     $W \leftarrow \{CW(c|S)\ \forall c \in S\}$
7     return $S, W$
8 end
```

Due to the application of the submodular objective function $F(S)$ in the greedy CRAIG algorithm, algorithm 1, Mirzasoleiman et al. prove that the final greedy coreset S_{greedy} always follows Equation 5. This essentially guarantees that $F(S_{greedy})$ is always at least ~63% of $F(S_{optimal})$; in practice, Mirzasoleiman et al. demonstrate much higher performance.

$$F\left(S_{greedy}\right) \geq \left(1 - \frac{1}{e}\right) F\left(S_{optimal}\right) \tag{5}$$

3 NEURAL PRUNING WITH FACILITY LOCATION

This paper utilizes the k-medoids implementation from the CRAIG algorithm to perform neural pruning by mapping the functional units of each layer, such as nodes or filters, to the "dataset" V inputted to algorithm 1. The similarity matrix, D, would be derived from the weights of the layer nodes/filters. This allows the coreset to be a weighted subset of the layer's nodes/filters, which can then be used to prune the respective nodes/filters and weights from that layer, if prunable; the corresponding weights from the next adjustable layer, if any, are also pruned to ensure that the neural network's internal dimensions remain consistent. To preserve the network's output dimension, the last (output) layer is not pruned. The similarity metrics used are discussed in more detail in subsection 4.1.

The version of the pruning algorithm presented in this paper only supports basic fully connected and convolutional neural networks. Psuedocode is provided in algorithm 4, with supporting functions in algorithm 2 and algorithm 3. The PyTorch implementation of this work is available at https://github.com/rahulm/neural-pruning-k-medoids.

Algorithm 2: Neural Network Layer Pruning

```
Input  : $l$ = layer to prune
         $p$ = prune percent, $p \in [0, 1]$
         $m$ = similarity metric function
Output : Pruned layer node coreset $S$
func PruneLayer($l, p, m$):
    if $l$ is a linear layer then
        $V \leftarrow l.nodes$
    end
    else if $l$ is a convolutional layer then
        $V \leftarrow l.filters$
    end
    $D \leftarrow$ m($V$)
    $B \leftarrow \lceil (1 - p) |V| \rceil$
    $S, W \leftarrow$ CRAIG($V, D, B$)
    if $l$ is a linear layer then
        $l.nodes \leftarrow V[S] \cdot W$
    end
    else if $l$ is a convolutional layer then
        $l.filters \leftarrow V[S] \cdot W$
    end
    return $S$
end
```

4 EXPERIMENTS

This neural pruning implementation was tested with a fully connected (FC) LeNet-300-100 architecture [Lecun et al. 1998] trained on the MNIST handwritten digits dataset [LeCun et al. 2010], and with a convolutional neural network (CNN) VGG16 architecture [Simonyan and Zisserman 2015] trained on the CIFAR-10 image recognition dataset [Krizhevsky 2009].

Specifically, experiments consisted of combinations of different similarity metrics and prune percentages per layer type. "Compression ratios" are calculated according to Equation 6, denoting how much of the original model was pruned. Each experiment consists of three steps:

(1) The desired model architecture is fully trained on the given dataset. (This step is only performed once, since subsequent experiments can reuse the same trained model).
(2) The trained model is then pruned according to the pruning parameters. The model size and test accuracy are recorded.
(3) The pruned model is finetuned (re-trained until convergence), recording test accuracy at every epoch.

$$Compression\ Ratio = 1 - \frac{size_{pruned}}{size_{original}} \qquad (6)$$

4.1 Similarity Metrics

As described earlier, the k-medoids function of the CRAIG algorithm requires a similarity matrix comparing all elements of the original set of nodes/filters/weights N against each other per layer. The original CRAIG paper utilized Euclidean distances for its implementation due to its use in corresponding theoretical derivations [Mirzasoleiman et al. 2020]. Here, however, we substitute numerous

Algorithm 3: Neural Network Layer Post-Prune Adjustment

```
Input  : $N$ = neural network model
         $O$ = shape of output of each layer in $N$
         $i$ = index of pruned layer in $N$
         $j$ = index of layer to adjust in $N$
         $S$ = subset from previous pruned layer
Output : Model $N$ with adjusted $j^{th}$ layer
func AdjustLayer($N, O, i, j, S$):
    if $N[j]$ is a convolutional layer then
        $N[j].weights \leftarrow N[j].weights[:, S]$
    end
    else if $N[j]$ is a linear layer then
        if $N[i]$ is a linear layer then
            $weightsPerChannel \leftarrow 1$
        end
        else if $N[i]$ is a convolutional layer then
            // find the nearest conv/pooling layer
               output shape
            $k \leftarrow i + 1$
            while $k < j$ do
                if len($O[k]$) $\neq 3$ then
                    break
                end
                // multiply together weights per
                   channel
                $weightsPerChannel \leftarrow \prod O[k][1:]$
                $k \leftarrow k + 1$
            end
        end

        $weightsToKeep \leftarrow []$
        for $s_i \in S$ do
            $weightsToKeep.extend($[
              $weightsPerChannel \cdot s_i$,
              ...,
              $weightsPerChannel \cdot (s_i + 1)$
            ])
        end
        $N[j].weights \leftarrow N[j].weights[:, weightsToKeep]$
    end
    return $N$
end
```

similarity metrics for Euclidean distance since it is easily accepted in the available implementation [Mirzasoleiman 2020]. As described in more detail in later sections, we see that other similarity metrics perform well empirically.

4.1.1 *L2 Norm (Euclidean Distance)*

The common L2 norm, approximated efficiently by scikit-learn via Equation 7a. To obtain a similarity metric (such that similar points have high values), we

Algorithm 4: Neural Network Model Pruning

```
Input  : $N$ = original neural network model
         $O$ = shape of output of each layer in $N$
         $L$ = list of prunable layer types
         $A$ = list of adjustable layer types
         $P$ = prune percent per prunable layer, $P \in \mathbb{R}^{|L|}$
         $M$ = similarity metric per prunable layer, $|M| = |L|$
Output : Pruned neural network, $N$
func PruneNetWithFacilityLocation($N, O, L, A, P, M$):
    $i \leftarrow 0$
    while $i < |N| - 1$ do
        if $N[i] \notin L$ then
            $i \leftarrow i + 1$
            continue // skip layer if not prunable
        end
        $S \leftarrow$ PruneLayer($N[i]$)

        // Then, fix next adjustable layer
        $j \leftarrow i + 1$
        while $j \leftarrow |N|$ do
            if $N[j] \in A$ then
                AdjustLayer($N, O, i, j, S$)
                break
            end
            else
                $j \leftarrow j + 1$
            end
        end
        $i \leftarrow j$
    end
    return $N$
end
```

subtract all the values from the maximum distance, as in equation Equation 7b.

$$D(x, y) = \sqrt{(x \cdot x) - 2 * (x \cdot y) + (y \cdot y)} \tag{7a}$$

$$S(x, y) = \left(\max_{x' \in N, y' \in N} D(x', y') \right) - D(x, y) \tag{7b}$$

4.1.2 ***Radial Basis Function (RBF) Kernel*** The RBF kernel used for various classification algorithms lends itself directly to a similarity metric in Equation 8.

$$S(x, y) = \exp\left(-\gamma \|x - y\|^2\right) \tag{8}$$

$$\gamma = \frac{1}{f} \quad \gamma = \frac{1}{\sqrt{f}} \quad \gamma = \frac{1}{f^2} \tag{9}$$

Experiments were run with the three γ values listed in Equation 9, where f is the number of features/weights per filter or node; for example, $f = 1$ in fully connected layers because each node outputs one value, and $f = C_{in} \cdot H_{filter} \cdot W_{filter}$ in a 2D convolution layer because each output channel/filter is treated as a node.

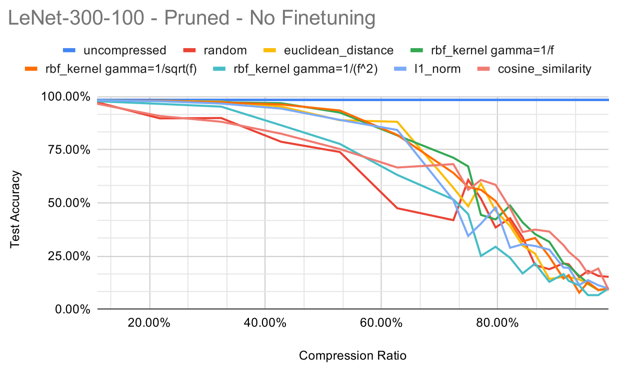

Figure 1: Test accuracy vs compression ratio of LeNet-300-100 trained on MNIST, without finetuning.

4.1.3 ***L1 Norm (Manhattan Distance)*** The common "Manhattan distance" calculated via Equation 10.

$$S(x, y) = \|x - y\|_1 \tag{10}$$

4.1.4 ***Cosine Similarity*** This computes the angle between two points after being projected onto the unit sphere. To do so, the dot product of the points are normalized by their L2 norm, as described in Equation 11.

$$S(x, y) = \frac{xy^\top}{\|x\| \, \|y\|} \tag{11}$$

4.2 Fully Connected Network (LeNet-300-100)

The LeNet-300-100 network takes as input a flattened 28x28 grayscale image, as per the MNIST dataset, and outputs 10 nodes for classification into 10 digits. In between, it contains two hidden fully connected layers with ReLU activation functions: the first contains 300 nodes, and the second contains 100 nodes. The MNIST dataset contains a total of 60,000 images in 10 classes (one for each digit), split up into a training set of 50,000 images and a testing set of 10,000 images. LeNet-300-100 pruning experiments were conducted with all of the similarity metrics mentioned in subsection 4.1, along with target prune percentages per layer ranging from 10% to 99%.

As illustrated in Figure 1, low compression ratios (below around 40%) result in performance at or greater than the uncompressed model performance even before any finetuning with L1 norm, euclidean distance, and RBF kernel for gammas of 1/f and 1/sqrt(f). Performing any finetuning almost immediately restores most, if not all, of the test accuracy to the uncompressed benchmark. In Figure 2, for example, we only see a drop of test accuracy in ~4 points to ~94% at the compression ratio of ~90% in most similarity metrics.

Nearly across the board, we see that the RBF kernel, euclidean distance, and L1 norm similarity metrics tend to significantly outperform the other tested metrics. This is especially apparent in lower compression ratios before finetuning, such as in Figure 1. Interestingly, however, the k-medoids based pruning only seems

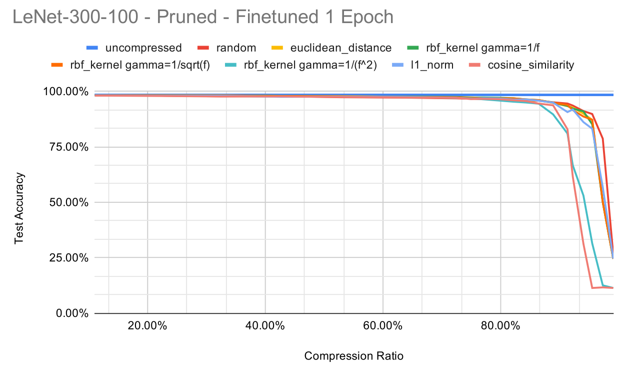

Figure 2: Test accuracy vs compression ratio of LeNet-300-100 trained on MNIST, finetuned for 1 epoch.

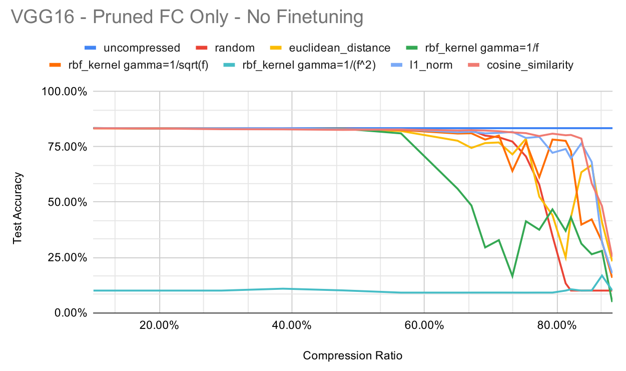

Figure 3: Test accuracy vs compression ratio of VGG16 trained on CIFAR-10, pruned FC layers only, without finetuning.

Compression Ratio	Fine-tuning Epoch	Test Accuracy				
		Random Baseline	Mussay et al.	**Our Method**	Delta (pp) to random	Delta (pp) to Mussay et al.
~80%	none	38.41%	49.24%	58.47%	+20.06	+9.23
	epoch 5	97.44%	96.93%	97.43%	-0.01	+0.50
~90%	none	21.52%	23.28%	30.11%	+8.59	+6.83
	epoch 5	96.24%	89.22%	95.92%	-0.32	+6.70

Table 1: Test accuracy comparison of our k-medoids neural pruning with the data-independent coreset pruning work of Mussay et al., on LeNet-300-100 with MNIST ("pp" stands for "percentage point").

to provide an improvement over the random baseline prior to any finetuning.

It should be noted that the simplicity of the MNIST dataset and the LeNet-300-100 architecture may be a major factor behind the fast convergence and high performance in finetuning experiments like Figure 2. Furthermore, the similarity metrics found to be effective for this specific architecture may not may be directly applicable to different layer or model types; initial experimentation with different layer types is explored in subsection 4.3.

For the fully connected LeNet-300-100, initial tests were also completed to compare our k-medoids neural pruning method to the data-independent coreset pruning from Mussay et al., which implements an algorithm with a theoretically provable trade-off between compression rate and approximation error. As noted in Table 1, our neural pruning method outperformed the algorithm from Mussay et al. at high compression ratios, both before and after finetuning.

4.3 Convolutional Neural Network (VGG16)

The VGG16 network takes as input a 32x32 color (RGB) image, as per the CIFAR-10 dataset, and outputs 10 nodes for classification into 10 classes. The CIFAR-10 dataset contains a total of 60,000 images with 6,000 images per class, split up into a training set of 50,000 images and a testing set of 10,000 images. This work's VGG16 architecture used 5 convolutional+pooling blocks, each with 2 to 3 convolutional layers followed by ReLU activation functions and a max pooling layer, followed by 2 hidden fully connected layers containing 4096 nodes each, and a fully connected output layer; more implementation details are available through PyTorch at https://github.com/pytorch/vision/blob/master/torchvision/models/vgg.py.

To understand the differences between pruning of layer types in VGG16, two sets of experiments were completed: one for pruning of only fully connected layers, and one for pruning of only convolutional layers, as detailed below. Both sets of experiments were conducted with all of the similarity metrics mentioned in subsection 4.1, along with prune percentages per layer ranging from 10% to 99%.

4.3.1 *Fully Connected Layer Pruning*

Like in the fully connected LeNet-300-100 experiments, we see in Figure 3 that the k-medoids based pruning outperforms the random baseline before finetuning, although the random baseline seems to join the other similarity metrics after finetuning as in Figure 4. Before finetuning, for example, the cosine similarity metric achieves only a ~3 point drop in accuracy to ~80% at an 82% compression ratio, whereas the random baseline dropped to ~10% accuracy at the same compression ratio. We also see, to a greater extent, that the RBF kernel with gamma of 1/(f^2) performs poorly compared to other similarity metrics both with and without finetuning.

4.3.2 *Convolutional Layer Pruning*

We again see in Figure 5 that most similarity metrics greatly outperform the random baseline before finetuning, with a maximum difference of ~50 points at ~20% compression. Unlike the other experiments, however, we see that the random baseline actually greatly overtakes all similarity metrics in Figure 6 after finetuning. Furthermore, both the random baseline and k-medoids pruning exhibit larger decreases in performance coupled with lower effective compression ratios; this may be explained by the smaller size of convolutional layers relative to the VGG16 fully connected layers, which in turn suggests lower redundancy available for reduction in the convolutional layers.

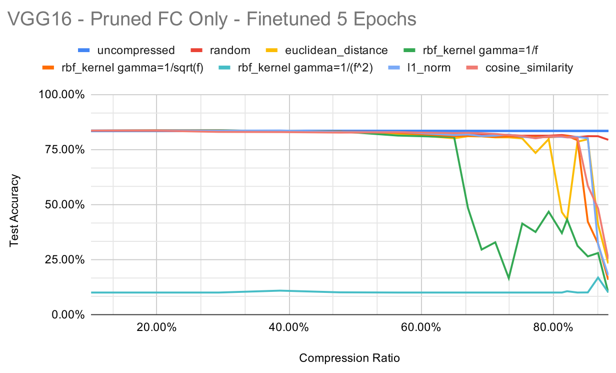

Figure 4: Test accuracy vs compression ratio of VGG16 trained on CIFAR-10, pruned FC layers only, finetuned for 5 epochs.

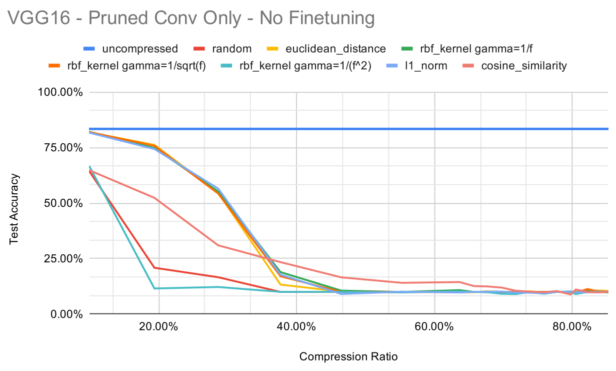

Figure 5: Test accuracy vs compression ratio of VGG16 trained on CIFAR-10, pruned convolutional layers only, without finetuning.

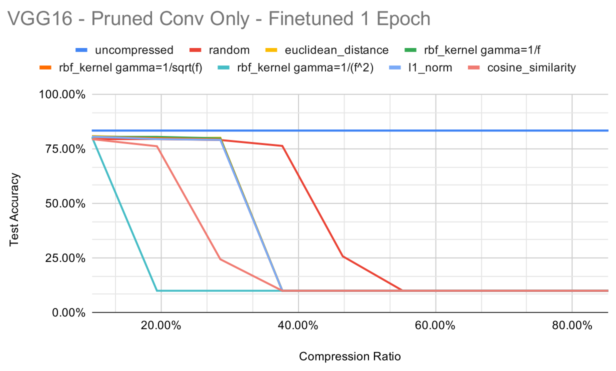

Figure 6: Test accuracy vs compression ratio of VGG16 trained on CIFAR-10, pruned convolutional layers only, finetuned for 1 epoch.

5 CONCLUSION

Overall, we see that fully connected layer pruning provides the highest compression ratios while preserving high test accuracy within short convergence times on both LeNet-300-100 and VGG16. This suggests that fully connected layers tend to contain more redundant information than convolutional layers, at least in the model architectures and datasets tested. However, experiments would need to be run with more complex models and datasets to better understand general behavior and optimal pruning configurations.

Prior to finetuning, k-medoids neural pruning seems to outperform the random baseline in all experimental setups. Performing any finetuning, however, seems to quickly correct imperfections in the remaining weights, such that the random baseline matches our neural pruning method's results. This suggests that the CRAIG algorithm powering our neural pruning is selecting a coreset that effectively estimates the original model's outputs, but that selection is overridden by any finetuning. So, k-medoids neural pruning is most suitable for situations in which post-pruning finetuning is infeasible; extremely large modern Transformer-based language models trained on immense datasets, for example, may not be finetunable/re-trainable with limited computational resources, and thus require an effective pruning step.

Although pruning experiments were successful in fully connected layers for both LeNet-300-100 and VGG16, pruning of convolutional layers exhibited a much more substantial decrease in test accuracy at lower effective compression ratios. Because even the random baseline provided poor convolutional pruning accuracies before and after finetuning, these results indicate that convolutional layers may be generally averse to pruning due to their information density relative to fully connected layers. Nevertheless, convolutional pruning can preserve most of a model's performance at low compression ratios if used carefully, as seen below ~30% compression ratio in Figure 6.

6 FUTURE WORK

6.1 Architecture Support

This work focused on fully connected layers and convolutional layers because of their prevalence in a wide variety of neural networks. Other popular layers, like BatchNorm, should also be supported. More exotic model types, such as Recurrent Neural Networks (RNNs) and Transformers, should also be experimented with. It should be noted, though, that these initial results suggest that fully connected layer pruning may be sufficient for most cases.

6.2 Baseline Comparisons

In this paper, results were mainly compared to the random baseline. It would be beneficial to more thoroughly examine the efficacy of neural pruning with k-medoids through comparison to other baselines, such as magnitude-based pruning, and other research in the field. For example, initial comparisons in Table 1 were done against the data-independent coreset pruning algorithm by Mussay et al.; similar experiments should be conducted with different models and datasets to better evaluate the method developed here.

Combining other baselines along with different datasets and architectures, as mentioned in subsection 6.1, could shed more light onto the results of this paper; for example, the seemingly high performance of random pruning after finetuning can be investigated. Furthermore, insight could be provided into possible explanations for other results, such as whether the models were overparametrized for the given datasets. That is, the advantage of our k-medoids neural pruning over the random (and other) baselines may become more pronounced with models that contain less redundant info compared to the size of their datasets; the pairings of models to datasets in this paper may rely on architectures that contain more learning power than is necessary for the relatively simple MNIST and CIFAR-10 datasets, and thus may contain more redundancies that were less sensitive to pruning.

6.3 Error-Bounded Pruning

The current k-medoids neural pruning algorithm removes a certain number of a layer's nodes based on a supplied target pruning percentage. Treating the pruning percentage as a hyper-parameter in this fashion relies on either prior knowledge from the model designer and/or some form of grid search.

Instead, the prune percentage can be replaced by an "error bound" hyper-parameter that denotes the highest acceptable amount of deviation in the next layer's output after pruning the current layer. Specifically, if layer i is being pruned, then the error would be the absolute difference in the outputs of layer $i + 1$ from before and after layer i's pruning; this output can be calculated by passing a matrix of ones through layer i, essentially computing the sum of the weights entering each node in layer $i + 1$. The algorithm can then attempt to automatically select the largest prune percentage that results in an error below the error bound, allowing optimization of the prune percentage without requiring a full evaluation, finetuning, and/or grid search.

In the simplest case, a set or range of prune percentage choices can be provided as another hyper-parameter. A more intelligent search algorithm, such as a binary search, could also be used to avoid a linear time check of all prune percentage options, under some assumptions.

6.4 Incremental In-Training Pruning

All of the experiments conducted in this paper involve pruning of pre-trained models. To avoid an extra pruning and finetuning stage, this pruning may be bundled in to the original training process as a step after every epoch, or some other interval. Each step could either be taken with a specified, and possibly decaying, prune percentage per layer, or as an implementation of the error bounded pruning in subsection 6.3. Utilizing a specified constant or decaying prune percentage per layer may force the algorithm to remove unnecessarily large portions of the model per iteration, possibly decreasing the model size until it can no longer learn the desired dataset. On the other hand, error bounded pruning can dynamically adjust the target pruning percentage and automatically stop pruning when the error bound is reached, essentially converging to an optimal model while reducing susceptibility to a poorly designed prune percentage hyper-parameter.

Ideally, this incremental pruning would not only remove the necessity of a post-training pruning stage, but also speed up training (due to a decreasing model size per step) and create more space-efficient models with less redundancy.

7 ACKNOWLEDGEMENTS

This MS capstone project was completed with the guidance of my faculty advisor, Dr. Baharan Mirzasoleiman (http://web.cs.ucla.edu/~baharan/), and with reviews from my capstone committee consisting of Dr. Kai-Wei Chang (https://web.cs.ucla.edu/~kwchang/) and Dr. Lin Yang (http://drlinyang.net/), all associated with UCLA.

This work used computational and storage services associated with the Hoffman2 Shared Cluster provided by UCLA Institute for Digital Research and Education's Research Technology Group. Cloud-based computational instances were also used from Lambda Labs (https://lambdalabs.com/).

REFERENCES

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2020. A Survey of Model Compression and Acceleration for Deep Neural Networks. arXiv:1710.09282 [cs.LG]

Jia Deng, Wei Dong, Richard Socher, Li-Jai Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. arXiv:1803.03635 [cs.LG]

Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M. Roy, and Michael Carbin. 2020. Stabilizing the Lottery Ticket Hypothesis. arXiv:1903.01611 [cs.LG]

Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc., 1097–1105. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *ATT Labs [Online]* 2 (2010). http://yann.lecun.com/exdb/mnist/

Yann LeCun, John Denker, and Sara Solla. 1990. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*, D. Touretzky (Ed.), Vol. 2. Morgan-Kaufmann, 598–605. https://proceedings.neurips.cc/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf

Baharan Mirzasoleiman. 2020. Data-efficient Training of Machine Learning Models: GitHub Repo. https://github.com/baharanm/craig

Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. 2020. Coresets for Data-efficient Training of Machine Learning Models. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, Virtual, 6950–6960. http://proceedings.mlr.press/v119/mirzasoleiman20a.html

Ben Mussay, Margarita Osadchy, Vladimir Braverman, Samson Zhou, and Dan Feldman. 2020. Data-Independent Neural Pruning via Coresets. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1gmHaEKwB

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9. https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV] https://arxiv.org/abs/1409.1556

Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli. 2020. Pruning neural networks without any data by iteratively conserving synaptic flow. arXiv:2006.05467 [cs.LG]