Measuring and Visualizing the Effects of Structured Pruning in Neural Networks

Michael Fellner

December 2024

Abstract

Neural networks are effective tools at finding non-linear patterns in data. Neural networks work by performing calculations between trainable parameters over the course of a series of layers. The parameters (weights) change with respect to the network's predictive ability, as measured by a loss function. When the network's parameters are optimized and the network can make good predictions on unseen data, the result is an effective instrument for making accurate predictions given non-linear data. One of the problems with a trained network is that the resultant network may come to not need every initialized parameter in its architecture in order to make accurate predictions – some neurons may matter significantly more than others. Nevertheless, the trained network still needs to take the time to compute using all neurons for each prediction it makes. This paper explores the use of structured pruning as a technique for removing unnecessary neurons in a successfully trained neural network, with the intention of speeding up the time of forward passes. A convolutional network, created using PyTorch, is trained on the MNIST image dataset. The network is then pruned to create a network with the same number of layers but fewer weights as the original. Multiple degrees of pruned networks are created. Each pruned network is then fine-tuned. The results offer a table showing the degree to which each network was pruned, the amount of time the network takes to perform forward passes across the train dataset, how accurate each network is, and the total parameters of each network. Qualitative results are also shared. There appear to be only mild effects of pruning, and discussion of why that may be the case is provided.

1 Introduction

The purpose of this paper is to explore the effects of structured pruning. Structured pruning is a technique for reducing the number of neurons in a trained neural network in order to make the network more memory efficient and perform forward passes more efficiently. This paper explores the effects of structured pruning on a convolution neural network (CNN), trained to predict digits in the

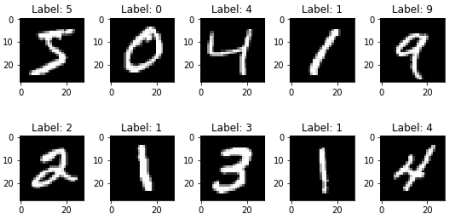


Figure 1: Example Images with Labels from MNIST dataset

MNIST dataset, based on techniques from (Li et al. 2017). What follows is the details of the MNIST dataset, the method of structured pruning, the details of the original CNN architecture, and how the network was trained. Additionally, the tools and means of running attached code are provided, as well as the results and discussion attempting to explain and raise relevant questions regarding the results and approach.

2 Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is a dataset of 28x28 grayscale images of handwritten digits with attached labels of the true digit (Deng 2012). The problem of training a CNN to predict unseen images in the database has become rather trivial, as evidenced by the use of the dataset often being used as a practice task for learning about CNN's (as evidenced by several results on kaggle.com, use in course syllabi, and online tutorials). This dataset was used precisely because it is easy to solve with a CNN. It can help test how few neurons are needed to still solve the task and is simple enough to measure training time without using too much memory or compute time.

3 Structured Pruning

3.1 Pruning Linear Layers

A linear layer of a neural network consists of an input vector $\mathbf{x}$, weight matrix $\mathbf{W}$, bias vector $\mathbf{b}$, and output vector $\mathbf{y}$. The output vector is calculated as:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}.$$

The weight matrix $\mathbf{W}$ has M rows and N columns, where N is also the length of $\mathbf{x}$, and $\mathbf{b}$ is the length of $\mathbf{b}$ and $\mathbf{y}$.

The goal of structured pruning on linear layers is to reduce the size of the weight matrix from $M \times N$ to $(M - a) \times N$ (and from M to $M - a$ for the bias vector), where a is the number of rows to be removed from the weight matrix. Structured pruning aims to remove the a most unimportant rows in each weight

matrix (and bias vectors). The importance of rows is calculated based on a pruning criterion.

In this case, the L_1 norm is used as the pruning criterion. Specifically, the L_1 norm of each row is calculated as:

$$I_i = \sum_{j=1}^{N} |w_{ij}|,$$

where I_i represents the importance score for the i-th row of $\mathbf{W}$, and w_{ij} is the weight in the i-th row and j-th column of $\mathbf{W}$. Rows with the lowest L_1 scores are discarded. The idea is that rows with smaller weights are less important than ones with larger weights.

3.2 Pruning Convolutional Layers

A convolutional layer of a neural network consists of a set of filters, each represented by a 4D weight tensor $\mathbf{W} \in R^{K \times C \times H \times W}$, where: K is the number of output channels (filters). C is the number of input channels. H and W are the height and width of each filter.

The layer performs a convolution operation on an input tensor $\mathbf{X} \in R^{C \times H_{in} \times W_{in}}$, producing an output tensor $\mathbf{Y} \in R^{K \times H_{out} \times W_{out}}$:

$$\mathbf{Y}[k, :, :] = \sum_{c=1}^{C} \mathbf{W}[k, c, :, :] * \mathbf{X}[c, :, :], \quad \forall k \in [1, K],$$

where $*$ denotes the convolution operation.

The goal of structured pruning for convolutional layers is to reduce the number of filters K, resulting in a smaller weight tensor $\mathbf{W}' \in R^{(K-a) \times C \times H \times W}$, where a is the number of filters to be removed. This also reduces the corresponding number of output feature maps in $\mathbf{Y}$ to $(K - a)$.

Similar to linear layers, structured pruning identifies the least important filters based on a pruning criterion. In this case, the L_1 norm is used again, only this time over each filter. The importance of the k-th filter is calculated as:

$$I_k = \sum_{c=1}^{C} \sum_{h=1}^{H} \sum_{w=1}^{W} |w_{k,c,h,w}|,$$

where $w_{k,c,h,w}$ represents the weights in the k-th filter. Filters with the smallest L_1 norms are pruned.

4 Tools and Approach

4.1 Original Model Architecture

The original CNN network consists of 5 convolutional layers and 2 fully connected layers, totaling 6.8 million trainable parameters. Results are pooled after

each convolutional layer, and dropout is applied after the second and fourth convolutional layer, as well as on the first linear layer. The total trainable parameters for the original CNN is larger than required to complete the classification task in order to make the effects of pruning more apparent. To create pruned models, the layers kept at each threshold are saved. A new model is then created using the saved layers with the appropriate dimensions of each weight matrix and bias vector calculated based on the dimensions of the saved layers.

4.2 Training Method

Each model trains on 60k training examples for 10 epochs using stochastic gradient descent with a batch size of 64. In the case of the pruned models, their training is a case of fine-tuning the detached layers from the original CNN. There are 10k test examples. The dataset is normalized prior to use. The learning rate is 0.001 and negative log-likelihood loss is used (cross-entropy loss could have been used instead but this is a remnant from working with a different approach initially).

4.3 Libraries Used and How to Run Code

To run the attached notebook file, upload it to Google Colab. Change the runtime to GPU. The code makes use of the PyTorch, Numpy, Pandas, Time, and Matplotlib libraries, all of which are already integrated with colab. The notebook can simply be run in Colab with no additional setup required. The dataset is downloaded through the PyTorch Datasets library. I have not tested locally and do not intend to do to lack of GPU.

5 Results and Outcome

5.1 Quantitative Results

Table 1: Convolutional Layers Pruning Only Results (10 epochs of training data)

Pruning Percentage	**Trainable Parameters**	**Training Time (s)**	**Test Accuracy**
0%	6,816,010	205.94	99.45%
80%	1,300,958	179.75	99.42%
90%	661,911	176.98	99.13%
95%	332,746	179.43	98.86%

Table 1 shows the results of pruning each convolutional layer by different thresholds. The training time measures how long it takes the network to forward pass through the 60,000 training sample in the MNIST dataset 10 times. There is a mild jump in training time from 0 to 80% pruning, with test accuracy

suffering only midly as pruning increases. Train time does not appear to change in any patterend way from 80 to 95% pruning.

Table 2: Pruning on all layers

Pruning Percentage	**Trainable Parameters**	**Training Time (s)**	**Test Accuracy**
0%	6,816,010	206.20	99.45%
80%	271,858	183.18	99.41%
90%	69,526	181.24	99.16%

Table 2 shows the result of pruning both convolutional layers and linear layers in the network. When pruning linear layers, only up to 90% pruning could be completed without causing an error. Despite reaching far fewer trainable parameters when pruning linear layers in addition to convolutional layers, the difference in train time and test accuracy appears to be negligible. Potential reasons for why are provided in the discussion.

5.2 Qualitative Results

This section examines the life of a sample input data and how it is transformed by each filter in the first convolutional layer of each network. We can intuitively see how pruned networks appear to have a richer result on average for applying each filter.

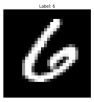


Figure 2: Original input with label 6.

On the original CNN, intuitively, it appears that the feature maps of applying the first convolutional layer with higher $L1$ scores look richer than those with lower $L1$ scores. The map with the lowest $L1$ score is almost black and it makes sense why it may be less useful than the one with the highest $L1$ score in the top left. These intuitive results offer an explanation on why pruning was able to work and still get similar accuracies as the original CNN; it appears that redundant layers existed in the original CNN and that they can be removed.

Firgure 4 shows the feature maps on the 80% pruned network after fine tuning. Even after fine tuning, it appears that some feature maps are richer than others. This may be why the network was still able to be pruned even more.

With only 3 and 2 filters respectively at 90% and 95% pruning, each filter appears to capture distinct features.

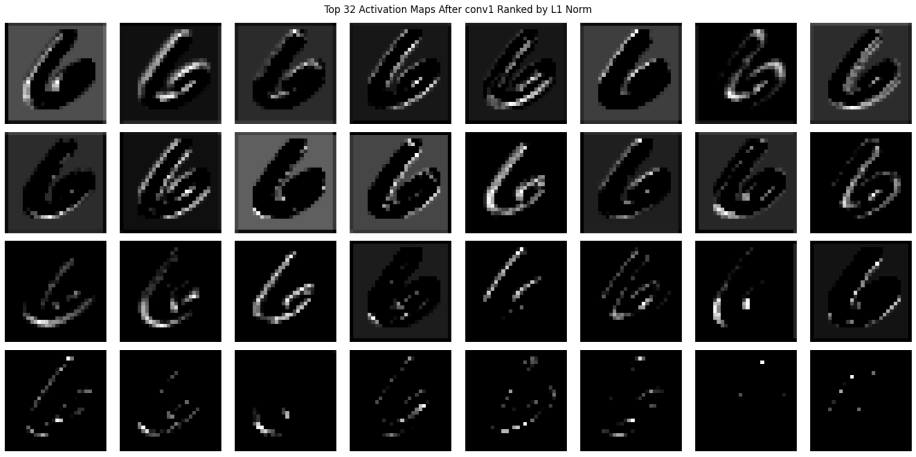
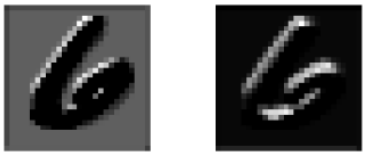


Figure 3: Feature maps ranked by $L1$ score (highest in top left, lowest in bottom right) on original CNN.

Figure 4: Ranked feature maps on 80% pruning.

(a) Ranked feature maps on 90% pruning.



(b) Ranked feature maps on 95% pruning.

Figure 5

6 Discussion

Compared to the results in Li et al 2016, these results are worse. Li et al prune two CNN networks (VGG-16 and ResNet-56) by 34% and 38% respectively for

about a 30% increase in inference speed. My results achieve the same benefits in using less memory while achieving similar test accuracy, however the inference costs appear to only increase by about 5%.

One reason why my results may be different despite pruning even more than Li et al, is due to hardware and scale. Running on Colab may not allow for the effects of pruning at the scale of 6.8 million parameters to show. Google Colab provides virtualized environments with shared resources that may not fully leverage the benefits of model pruning and the underlying hardware might not be optimized for sparse computations, leading to limited improvements in inference speed. Specialized hardware accelerators have been shown to better exploit spare pruned networks (Sui et al. 2023) and using such hardware may lead to better results than running on Colab.

Another reason my resutls may be different is due to software reasons. The use of PyTorch may be optimized in such a way that the number of layers affects the speed more than the amount of rows or kernels of weights in each layer, causing pruning speed effects to not emerge at this scale.

Another pruning techique called weight quanitization reduces network memory by reducing the floating point bit size of each weight (Krishnamoorthi 2018). This would have been interesting to compare to, however this was too difficult to set up for this time.

7 Future Directions

To go further on this project, I would learn more about the hardware and software reasons for why my pruning only increased inference speed by 5%. I would also look into the effects of the feature maps of applying the 2nd, 3rd, and 4th convolutional layers as well in order to learn more about the qualitative effects of pruning.

8 Conclusion

This project showed that pruning can be a good option for reducing memory of a network without reducing test accuracy significantly. It showed that pruning can increase inference time at least a little bit. And most originally, it showed how the feature maps of more pruned convolutional layers appear to be of richer average quality than less pruned, or unpruned layers. This project was a learning experience on creating a data prediction pipeline, on learning more about CNN's, and about how to take pieces of existing neural networks and use them to build a new pruned network.

References

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, *29*(6), 141–142.

Li, H., Kadav, A., Durdanovic, I., Samet, H., & Graf, H. P. (2017). Pruning filters for efficient convnets. https://arxiv.org/abs/1608.08710

Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*.

Sui, X., Lv, Q., Zhi, L., Zhu, B., Yang, Y., Zhang, Y., & Tan, Z. (2023). A hardware-friendly high-precision cnn pruning method and its fpga implementation. *Sensors*, *23*(2), 824.