Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Sparsity Granularity in Deep Neural Networks

Graham Hutchinson

14316275

April 2019

Supervised by David Gregg

A dissertation submitted in partial fulfillment of the requirements of BAI (Computer Engineering)

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland

Abstract

This dissertation explores the effect of Granularity of Sparsity on the accuracy of neural networks. A sparse tensor is a tensor in which most of the values are zero. The granularity of the sparsity is a measure of how regular the non-sparse elements in a tensor are, ranging from fine-grained sparsity, which involves pruning individual weights, to coarse-grained sparsity, which involves pruning entire channels of a layer of a network.

To achieve this sparse structure, I used pruning. Pruning involves using some metric to determine the least important weight in a layer, and setting that value to zero. This process is repeated until the desired sparsity is reached. Previous research only pruned a small subset of possible shapes from layers in neural networks. In this dissertation I attempted to prune a much larger set of shapes from the layers of a neural network. The pruning metric used for this research was absolute-value pruning.

The architecture I conducted this research on was LeNet, and the dataset I used was MNIST and CIFAR10. Pruning worked very well in the LeNet/MNIST case, with one block maintaining an accuracy of around 90% up to a sparsity level of 0.8 that was 50 times the size of individual weights. However, pruning worked less well in the LeNet/CIFAR10 case, with only a few blocks managing to maintain a decent accuracy. This can likely be attributed to the more complex dataset, the simple network, or the simple pruning metric.

The blocks that performed well were usually either individual weights, blocks that matched one or more of the dimensions of the current layer. Blocks that tended to perform poorly were blocks that had dimensions that did not evenly divide into the dimensions of the current layer. Despite the discrepancy of the performance between the LeNet/MNIST and LeNet/CIFAR10, these qualities were the same in both cases.

Declaration

I, Graham Hutchinson, hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

Signature: Graham Hutchinson

Date: 30/04/2020

Acknowledgements

I would first like to thank my supervisor, David Gregg, for all his help, constructive feedback, and guidance throughout this process.
I would also like to thank Andrew Anderson for the help with the Caffe framework.
Finally, I would like to thank my friends and family for all their love and support.

Contents

1 Introduction

Deep Neural Networks are becoming some of the most widely adopted and popular forms of machine learning used today. However, the size of the networks being used and developed can exceed billions of parameters [1]. This means neural networks can take up a large amount of storage space and can use a large amount of computing power to process. This makes neural networks difficult to deploy on embedded systems, as they are generally low powered devices with a small throughput and a small storage capacity [2]. Each deep neural network is broken up into sections called layers. Each of these layers are 4-dimensional weight tensors. Previous research has attempted to decrease the size of these networks, and convert the layers of the networks into "sparse" tensors [3]. Sparsity is the measure of the number of weights in a weight tensor that are zero, compared with the total number of weights in the tensor. If the sparsity of a network could be increased, then it would require less storage to store, as all of the weights with the value zero would not need to be stored. There is a method that is used to achieve sparsity called pruning. Pruning works by using some metric to determine the least important weights in a tensor, and setting those values to zero, so they can be ignored. There are many different metrics for pruning, for example second-order derivative pruning [4] and absolute value pruning [5]. However, these metrics only prune individual weights. This leaves very irregular patterns of non-zero weights behind, which makes it difficult to produce a fast convolution routine. The regularity of the weights that remain in a sparse network is referred to as a the Granularity of Sparsity. In previous research, the granularity of sparsity was limited to only a small subsection of possible pruning shapes, i.e. individual weights, sub-kernel vectors, kernels and channels [5]. The higher the granularity of sparsity is, the more regular the weight pattern is. This makes developing a fast convolution routine much

easier. In this paper, I attempt to prune a much larger set of shapes from a range of neural networks that were trained on several different datasets. In pruning a larger set of shapes, I hope to discover shapes that perform better than the shapes previously tested, and achieve as high a granularity of sparsity as possible.

2 Background

2.1 Deep Neural Networks

Deep Neural Networks(DNNs) have emerged as some of the most popular forms of machine learning in recent years. They are comprised of large networks containing nodes called neurons, each of which corresponds to some activation function. These neurons are organised in layers, of which there are many types, e.g. Dense, Convolutional, etc.

The uses for DNNs are broad, including image recognition [6] [7] [8] [9], speech recognition [3] and natural language processing [1] [5]. With so many uses for DNNs, there are becoming more widespread in industry, and are being applied to a wide array of problems and applications. However, the DNNs being developed and used today are becoming very large in size, with billions of neurons in the case of BERT and GRT [1], and hundreds of millions of parameters in the case of AlexNet [6]. Networks of this size take up a lot of memory, and can take a long time to compute. This results in DNNs being difficult to implement on embedded systems such as mobile phones and embedded vision applications, and therefore they need specially designed networks for it to be possible to implement a DNN on these systems [2]. Embedded systems i.e. microcontrollers, generally have low memory, and have low processing power. So, implementing a network with billions of parameters can be a difficult task. There has been some research into

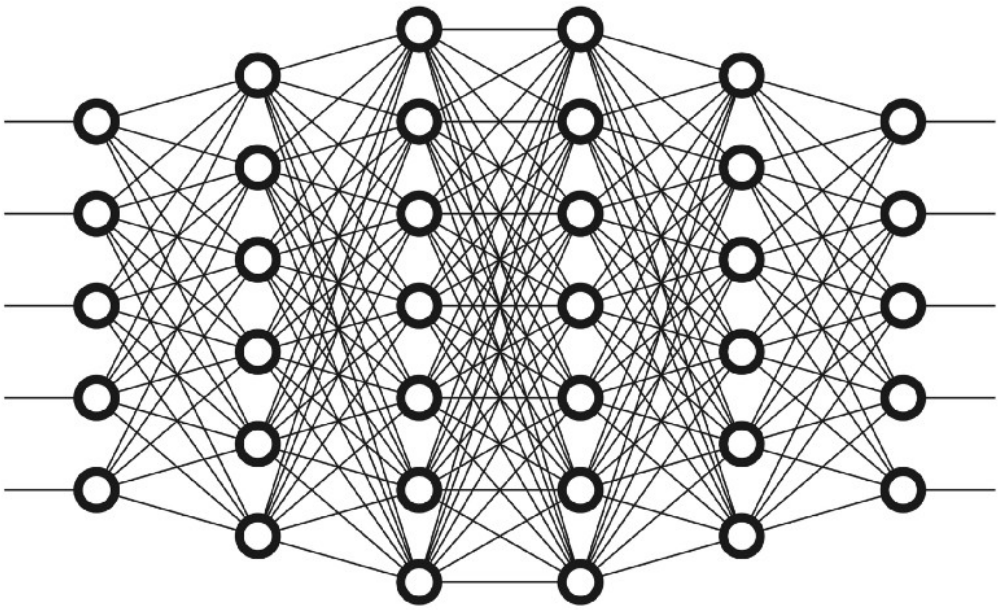

Figure 1: Deep Neural Network Structure

methods of implementing these networks onto embedded systems such as MobileNet:

> *"a streamlined architecture that uses depthwise separable convolutions to build light weight deep neural networks" [2]*

This is an ongoing issue however, as new networks are being developed every day, and as they begin to be deployed more commonly in everyday applications, it is imperative to reduce the footprint that they occupy and to reduce the number of parameters, so that they can be deployed on smaller, lower powered devices.

2.2 Sparsity and Pruning

In order to reduce the size of the network, there needs to be a method of reducing the number of parameters. Some criteria must be developed by which the weights are removed from the networks. The objective of this process is to be left with what is known as a sparse tensor. A sparse tensor is a tensor in which the majority of the weights are zero, as can be seen in figure 2. In this figure, the red squares represent weights that are non-zero, while white squares represent weights that are zero. This type of structure for the convolutional layers in a neural network are desirable because DNNs that have been reduced to sparse tensors have been shown to perform better, with better generalization, along with decreasing the size of the model whilst increasing the speed [3]. Having a sparse structure would also make it easier to implement on embedded systems, due to the reduced size.

In order to reduce the size of these models, I needed to choose some compression method. The method I chose was pruning. Pruning can reduce an over-parametrised network to an appropriately parametrised one [10]. Pruning works by setting a certain number of weights in a network to zero. Pruning can be done at random [3], however, pruning based on some criteria works better [11]. While pruning weights at random will reduce the number of weights in a layer, there are more complex criteria used to induce sparsity [10] [12], that work better than random pruning. The criteria used in this research was pruning based on some threshold. The weights in a layer of a neural network follow a Gaussian distribution, with the mean value being zero and the minimum and maximum value being 1 or -1 respectively [5]. This means that the majority of weights located around the centre are quite close to zero, and therefore, in terms of importance to the network, are the least significant. In order to prune weights, a threshold is set, e.g. 0.4, and then the smallest 40% of the weights are set to zero.

Sparsity is a measure of how sparse a tensor is. A tensor with a sparsity

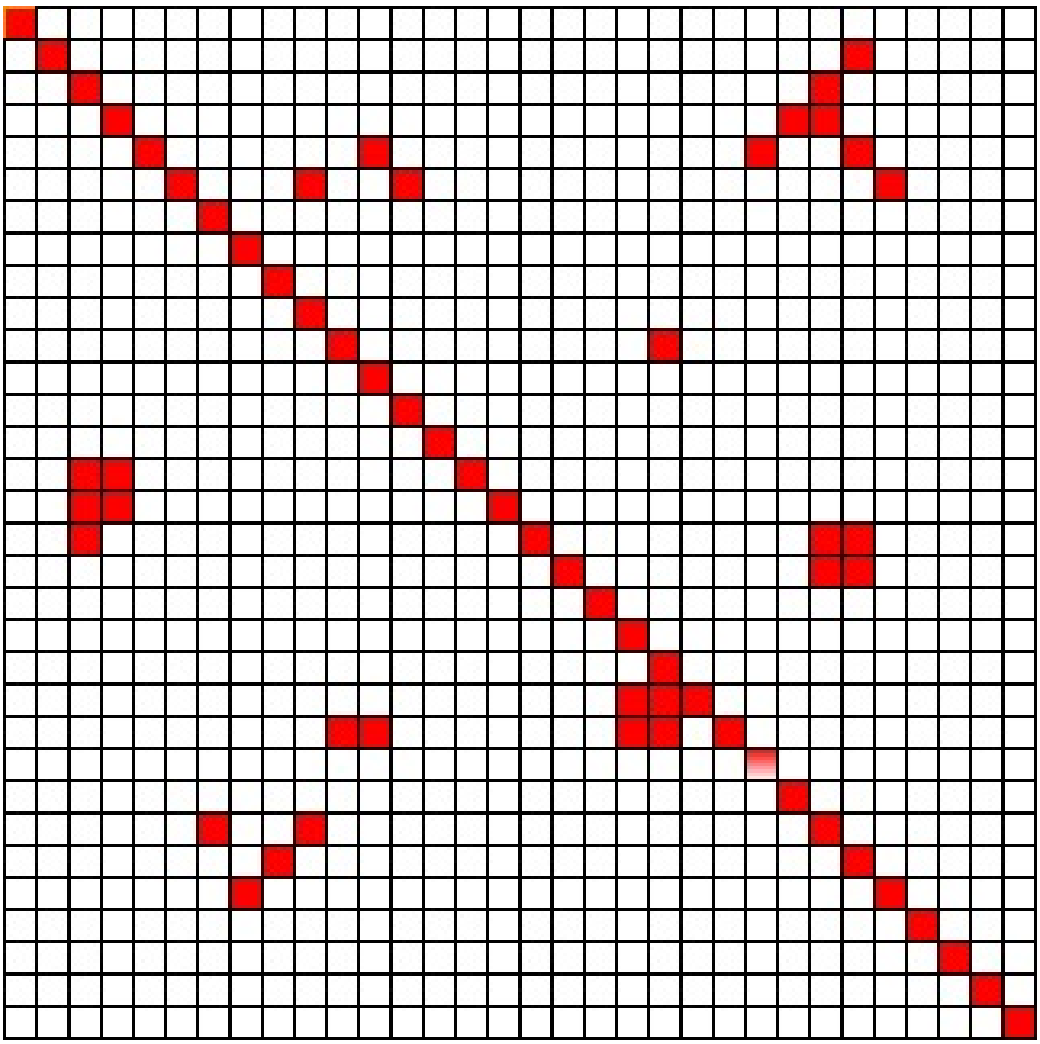

Figure 2: Sparse Network Structure

of 0 has all non-zero weights, while a tensor with a sparsity of 1 has all zero weights. The objective is to achieve a tensor with as high a sparsity value as possible, whilst maintaining the accuracy of the network. There are many frameworks developed that are used in research and industry to produce sparse neural networks, including Neural Network Distiller [13] and Caffe [14], the latter of which was used in this research. The types of neural

nets studied in this research are convolutional neural nets (CNNs), used for image classification. Pruning has been proven to be an effective compression method for creating sparse networks [15] [16] [17] [18] [19] [20].

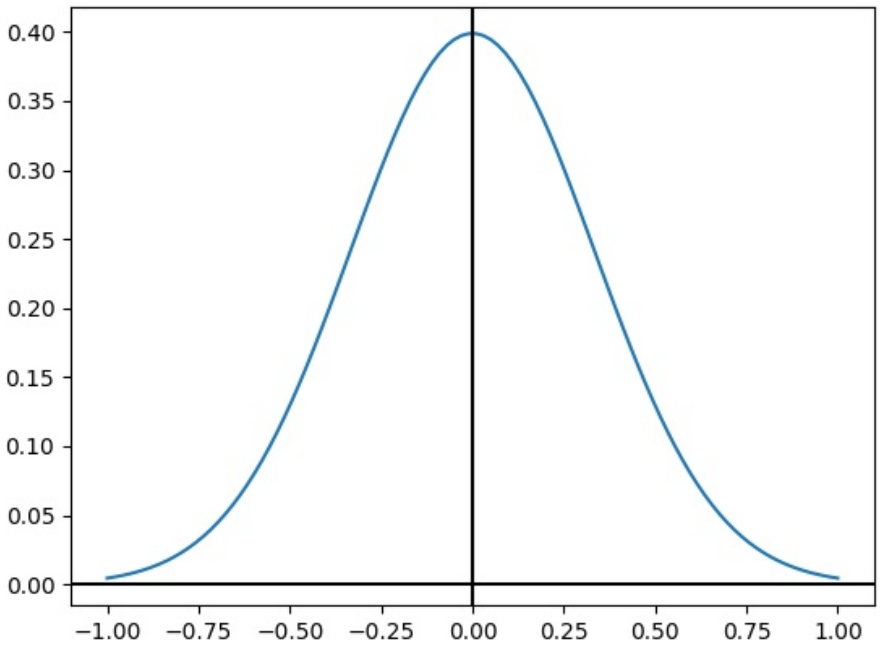

Figure 3: Gaussian Distribution of Weights in a layer

2.3 DNN structure

Deep Neural Networks consist of a series of layers that are connected to one another. Each deep neural network will have an input and an output layer, and then at least two hidden layers. There are many different forms that these hidden layers can take. The two main types of layers that are used in this research are Dense layers and Convolutional layers. Dense layers, also

known as fully connected layers, are layers in which the input is a linear operation on the input vector to that layer. Convolutional layers on the other hand, are layers that are broken up into a set number of filters, in which nearby nodes are grouped together. There are other types of layers such as Pooling layers and Normalisation layers, however, they are not really touched upon in this research.

The first neural network that was used in this research was the LeNet architecture. The LeNet architecture consists of two convolutional layers and three dense layers [21]. The structure of this network can be seen in figure 4 [21].

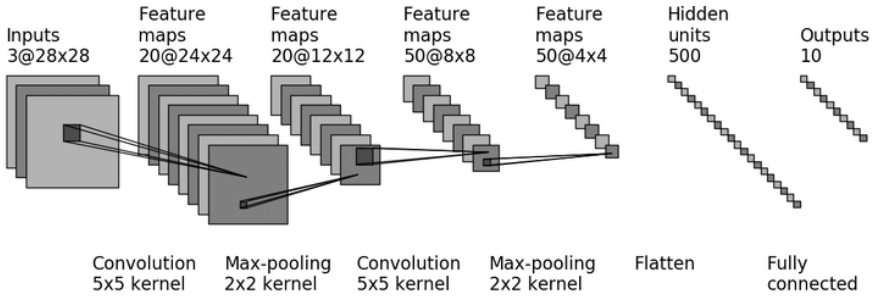

Figure 4: LeNet structure

This architecture was a simple architecture, that would get very accurate results for simple datasets. This architecture had been trained using the MNIST dataset, which can get accuracy of over 90% [21]. The only layers that are pruned in this research are convolutional layers.

The next neural network that was used in this research was AlexNet. AlexNet is a slightly more complex convolutional neural network with five convolutional layers [6]. The structure of AlexNet can be found in figure 5 [6].

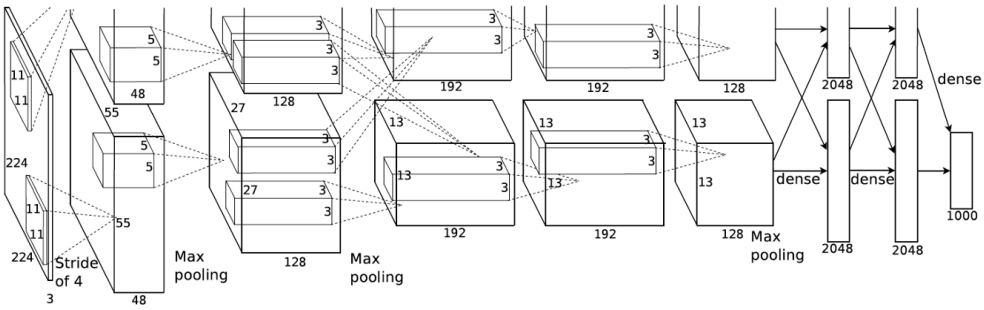

Figure 5: AlexNet structure

2.4 Granularity of Sparsity

In previous research, the most simple pruning algorithms pruned individual weights. However, pruning individual weights results in non-regular patterns of non-zero weights across the network. Having a more regular pattern to the non zero weights in the tensor, makes a faster convolution routine easier to implement [5] [22]. The degree of which the pattern of weights is regular in a sparse tensor is called the granularity of sparsity [5]. The granularity of sparsity ranges from fine-grained sparsity(vanilla sparsity) i.e. individual weights, to coarse-grained sparsity (filter-wise sparsity) i.e. pruning of entire filters of each layer. Fine grained sparsity has been shown to work well on CNNs and Recurrent Neural Networks (RNNs) [5]. Between fine-grained sparsity and coarse-grained sparsity, there is also sub-kernel vector sparsity and channel sparsity. These will be explained in further detail later. Each convolutional layer of a CNN can be viewed as a 4D weight tensor, with dimensions M, C, Y and X, with M corresponding to the number of filters(i.e. the output dimension), C corresponding to the number of channels(i.e. the input dimension), Y corresponding to the y-dimension in the kernel, and X corresponding to the x-dimension of the kernel. When pruning individual weights, each of the four dimensions are required to identify the weight to prune, i.e.

(M, C, Y, X). Increasing the granularity of sparsity and pruning sub-kernel vectors, you no longer need the x-dimension, i.e. $(M, C, Y, :)$. Pruning kernels only require the M and C dimensions, i.e. $(M, C, :, :)$. Finally, when pruning entire filter, only the M dimension is required, i.e. $(M, :, :, :)$. In Dally et.al. [5], the only granularities that were explored were this small subset of possible pruning grain shapes. In each granularity, for each dimension, either the entire dimension of the pruned block was pruned, or 1 element of it was. For example, in a layer with dimensions $(5, 5, 5, 5)$, the only shapes that were pruned are $(1, 1, 1, 1)$, $(1, 1, 1, 5)$, $(1, 1, 5, 5)$ and $(1, 5, 5, 5)$. These shapes were chosen because "they can be and have already been mapped into simple computation logics." [5]. This paper will seek to test the effectiveness of pruning using other shapes that have not been studied in previous research.

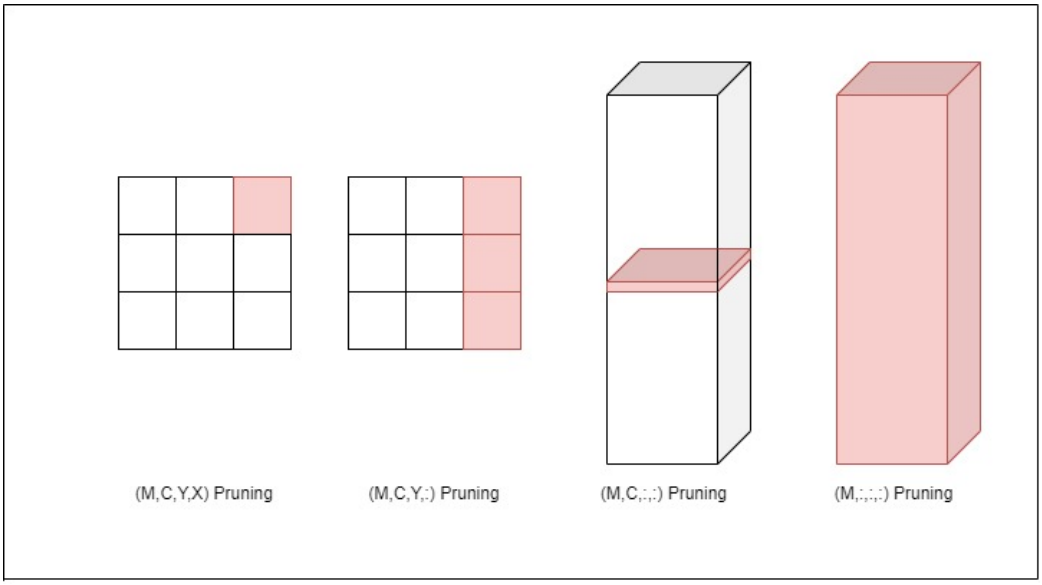

Figure 6: Granularity of Sparsity

The logical next step for pruning would be to attempt to prune different shapes. Using larger shapes not studied in previous research, the granularity

of sparsity could still be increased, but using a different approach. In previous research, only very regimented shapes were considered, either including all the elements of a dimension, or one. This paper proposes using a much larger set of shapes in the pruning process. For example a shape of size $(2, 2, 2, 2)$ could be pruned for the network, as seen in 7. In fact, any shape of the form (M, C, Y, X) could be pruned, as long as each of the dimensions of the block are less than or equal to their respective weight-tensor dimension.

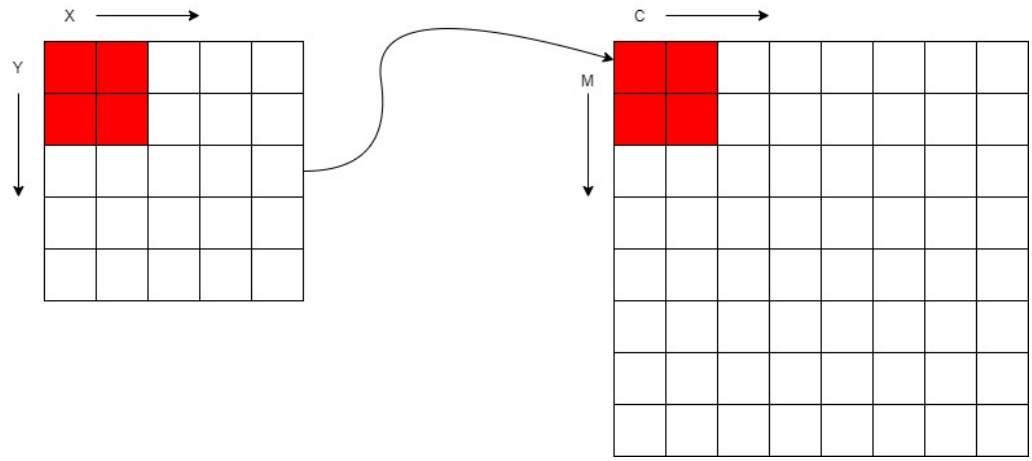

Figure 7: Pruning a Shape of Size 2x2x2x2

2.5 Pruning Metrics

There were many different pruning metrics used to order the weights in terms of importance before the pruning process in previous research, which were outlined in Dally et. al. [5]. These metrics were used so that the least important weights could be pruned first. The first of which was second-order derivative pruning [4]. This type of pruning worked by finding the second order derivatives of each of the activation functions, setting some threshold and then pruning the weights that are below this threshold. The next pruning metric that was used in previous research was using the absolute

value metric [5]. This metric takes the absolute value of each of the weights, orders them,then sets a percentage of the weights that should be pruned, say 20% and then the smallest 20% of the weights are pruned. Finally the last metric that I considered for this research was loss-approximating Taylor expansion. This metric "approximates the change in cost function induced by pruning network parameters" [23]

The pruning metric that I chose to use was the absolute value metric. I chose this because I thought it would be relatively simple to implement, and the aspect of pruning I was most interested in was the effect of using different shapes when pruning, not the pruning metric.

3 Methodology

3.1 Requirements

Before I wrote any code or made any decisions in regards to the library that I would use, it was important to decide what was necessary to complete this type of pruning work. The essential components needed were to be able to train deep neural networks, access the weights in a 4-D tensor, apply a pruning algorithm to the weight tensor, and be able to test the accuracy of the deep neural network after a set number of weights has been pruned. There are three different attributes of a deep neural network that can be altered to conduct tests on the effect of pruning. Firstly, there is the architecture of the neural network, which refers to the structure of the neurons inside the layers of the neural network. The networks that will be explored in this project are LeNet and Alexnet. Secondly, the dataset that the neural network is trained on can also be changed. The datasets used in project are MNIST and CIFAR-10. I will discuss both the architectures and the datasets used in this project in more detail later. Finally, the shape of the "grain" could

be changed, i.e. the dimensions of the block that is going to be pruned from the network at each step. The network is comprised of a set of layers, each of which is a 4 dimensional tensor, so the block is also a 4 dimensional object. The dimensions of this block are constrained by the smallest maximum length of the dimensions in any of the layers. For example, if there are 2 layers in a network, with sizes $(20*1*5*5)$ and $(50*20*5*5)$, the maximum dimensions of a block that could be pruned from the network is $20*1*5*5$. If the block to be removed from the network was set to say $20*1*5*6$, this would result in an out of bounds error, because in layer one, the pruning algorithm would be attempting to prune a weight that lay outside of the tensor in the X-dimension. Once the requirements for the project were considered, and an outline had been drawn up on the types of tests that would be attempted during the course of this project, the process of choosing a library to carry out these pruning tests could begin.

3.2 Distiller

The first attempt I made at carrying out conducting pruning tests was by using the Neural Network Distiller package. This is a open source python package that is written in PyTorch and was designed by Intel. The package not only handles pruning algorithms, but many other compression algorithms. The package initially seemed like the best and most complete option for working with pruning, and it appeared to fit the type of research I wanted to conduct quite well. The package contained a simple sample python program called compressclassifier.py, that carried out a basic pruning algorithm according to a pruning scheduler, stored as a YAML file. YAML stands for "YAML Ain't Markup Language" and is a serialisation language that is used to configure files. It serves as a human-readable structured data format. In this application, it is used to inform the pruning script how the weights

should be pruned, i.e, how many should be pruned, in what order, how often etc. There are a few different ways the pruning scheduler could be written, and I will be discussing them in more detail later. The sample script allowed me to specify an array of parameters for the neural network that include: the architecture, the learning rate, the number of batches to process before printing progress, the number of threads being used to load the data, the number of epochs, the dataset to be trained on and the compression scheduler to be used on the network. Additionally, using the YAML file gave me fine control over how the data was pruned, including how often it was pruned, the epoch at which to begin pruning, the epoch to end pruning, etc. The most basic use of the program worked by iterative pruning, that is, starting at the beginning epoch, and finishing at the ending epoch, after each epoch, the network was pruned by a certain threshold, e.g. if the threshold was 0.1, it would remove the smallest 10% of the weights in the network and then continue to train. The script then repeats this at every step until it has completed the set number of epochs. This could be altered by the pruning scheduler file, i.e. the YAML file.

The pruning scheduler file was used to dictate how the model should be pruned, how often it should be pruned, and how much it should be pruned. This allowed the pruning regimen to be changed without changing the code each time. The pruning scheduler file is split up into sections, which can be of several different types, however, I only needed a subset of them, namely, pruners, learning rate schedulers, and policies. Pruners are the section of the pruning scheduler file where the pruning algorithm is specified. Different types of pruning algorithms can be defined, some of which I will discuss later. Learning rate schedulers work in a similar way to pruners, except instead of a pruning algorithm, they define a learning rate decay algorithm. Finally, policies are used to specify when the algorithms defined in pruning scheduler file are to be used. This gave me the ability to only use the pruning algorithm

for certain epochs, or to only prune every second epoch or to prune the epochs between epoch 30 and 40, for example. These different schedules could be combined to create more complex pruning schedules. For each algorithm, a policy is used to dictate when that algorithm is to be used. There are three components to the most basic policies, namely, starting epoch, ending epoch, and frequency. Starting epoch and ending epoch are used to indicate at what epoch the algorithm is question is to begin and end at. The frequency is used to indicate how often the algorithm is used, with a value of one, indicating the algorithm is to be used every epoch, a value of two, indicating the algorithm is to be used every second epoch and so on.

There was only one type of learning rate scheduler used in the project, the exponential learning rate algorithm, which decreases the learning rate exponentially with each passing epoch. However, there are several pruning schedulers that can be declared in the pruning scheduler file. The most basic type of pruning algorithm is a magnitude pruner. This algorithm takes some threshold between zero and one, e.g. 0.6, and then compares each weight in the tensor to this threshold. If the weight's absolute value is larger than this threshold, it is not changed, but if its absolute value is below this threshold, it is set to zero. This is a very simple way to prune a tensor, however, it did not give me much fine control. The weights in a layer of a deep neural network follow a Gaussian distribution, however, if I , for example, wanted to remove 30 percent of the weights in a tensor, using a magnitude pruner would not work very well, because I would have no idea what to set the threshold to in order to achieve this level of sparsity. The next type of pruning algorithm I used is a sensitivity pruner. This type of pruning algorithm first finds the absolute value of each weight in the tensor. Next, I specified for each layer the ratio for pruned weights to total remaining weights to be, e.g. for one convolutional layer, I might set the ratio to be 0.65, which would result in 65 percent of the remaining weights in that layer being pruned.

The level pruning algorithm was the algorithm that best suited my needs. Instead of setting the number of weights to be pruned from the layer at each step, as with the sensitivity pruner, each level was given a target sparsity, i.e. the sparsity that the level should have at the end of the pruning process. This would be difficult to do with the sensitivity pruner, as we would first have to work out what the ratio should be at epoch zero in order to achieve our target sparsity at the end of the pruning process. The level pruner allowed me to skip this step and decide what sparsity I wanted for each level at the end of the pruning step. The framework also included a random pruning algorithm which chose the weights to be pruned at random. The framework also included several structure pruners, which, instead of pruning individual weights, pruned entire structures of the tensor, e.g. kernels, filters and feature maps. These structure pruners could be combined with other pruning algorithms, for example level pruning, or sensitivity pruning. These algorithms woul work in the same way as they would by individual weights. For example, if a structure pruner that was set to prune kernels was combined with a sensitivity pruner that had a threshold set to 0.5, then the program would be order all the kernels in the tensor according to the sum of the absolute value of its weights, and then prune the smallest 50 percent of kernels. In order to carry out the research into pruning new shapes, I would have to use a structure pruner combined with a level pruner.

It was at this point when I ran into the main issue I encountered while working with Distiller. Distiller is a very powerful framework that supported many different type of pruning algorithms. However, in terms of pruning structures, i.e. the type of pruning that I wanted to study, it only supported a small subset of possible pruning shapes, i.e. sub-kernel vectors, kernels, etc. I wanted to study pruning these shapes, but I also wanted to study a much larger subset of possible shapes. I needed to work within a framework that allowed me to choose any $M * C * Y * X$ shape to prune, and then for the

framework to be able to choose the smallest shape of that size in the tensor and prune it, not just $1 * 1 * 1 * X$ shapes, $1 * 1 * Y * X$ shapes, etc. that Distiller supported. So, I decided that while Distiller was quite powerful, and gave me great flexibility in terms of using different pruning algorithms, I needed to choose some other framework that would allow me to be be more flexible in my choice of shapes to prune.

3.3 Caffe

From this point I started to work with the Caffe Deep Learning Framework. Caffe was much more open and less rigid than the Neural Network Distiller package. It allowed me to work with different shapes much more easily and tailor it's functionality to meet my needs. I was given access to the triNNity-caffe package which was a number of scripts being used in the university for research into pruning. I began working with a script that pruned individual weights at random, and then tested the accuracy after each weight was pruned. I decided to modify this script to not just prune individual weights but also take 4 inputs, corresponding to a 4-dimensional block $M * C * Y * X$. In this case M in the number of output feature maps, C in the number of input feature maps, Y is the size of the block in the y dimension of the image, and X is the size of the block in the X direction of the image.

The first thing that needed to be added to this script was the ability to return the smallest weight in terms of absolute value in the tensor, as the script needed to be able to not choose the next weight to be pruned at random, it needed to be able to eliminate the least significant weight, i.e. the one that was having the least impact on deciding the output of the neural network. To do this, the script first set the minimum value stored to be something very large, say 10000, and then the script iterated over the entire 4D tensor, and at each position, it calculated the absolute value of the

weight at that position. The script also stored $M_{min}, C_{min}, Y_{min}$ and X_{min} values that corresponded to the position that the smallest weight that had been found was stored in the 4D tensor, so that it could be pruned once the entire 4D tensor had been iterated through. At each position, the absolute value of the weight at that position is compared to the minimum stored value, and if the absolute value is smaller than the minimum stored value, then the current absolute weight value is stored as the new minimum weight value. Additionally, the position of the new minimum weight value is stored in $M_{min}, C_{min}, Y_{min}$ and X_{min}. After one pass through the 4D tensor, the script will have found the minimum value to prune.

However, in adding this functionality, I encountered a small issue. The way the program was written, it would iterate over the 4D tensor and find the minimum value and store it's location for later pruning. However, the program would then set that new weight to be zero. Then, on the next pass through the tensor, the program would choose the location of the previously pruned weight as its minimum value. And so, the program would repeat indefinitely, because it would be pruning the same weight over and over again and the actual number of weights have been pruned would remain one. So, I needed to add some system to the program in order to keep track of which weights had already been pruned. In order to do this, for each layer, I declared another 4D tensor, a "marked" tensor, i.e. a tensor used to mark whether the weight had been pruned or not. I declared the new marked tensor to be the same shape as the current layer and set every value to zero. I then made it so that whenever the program had iterated through the whole tensor, and had found the minimum value, it pruned that particular weight and then set the value of the marked tensor to 1 instead of 0. Then, at each consecutive pruning step, when the program had found a new minimum value, it would check that location in the marked tensor. If the value was zero, it would set the value in the weight tensor to be the new minimum value

and $M_{min}, C_{min}, Y_{min}$ and X_{min} to be the coordinates of the new minimum weight. Then, at the end of the pruning process, it would mark the marked tensor at the location of the minimum value and the process would repeat.

Next, I needed the ability to prune to a certain threshold. The data structures that house layers in Caffe are called caffe.Net, and they have a built-in function to allow the user to access the weight, namely:

net.layer_dict[layer_name].blobs[0].data.size

This returns an int, corresponding the number of weights stored in net, in the current layer, i.e. layer_name. So for each layer, I now knew the total number of weights. Next, I needed to keep track of the numbers of weights that had been removed from each layer. To do this, I created a dictionary, pruned_weights, that was used to store the total number of weights removed from each layer. So, at the end of a round of pruning, I incremented the pruned_weights dictionary at the current layers location by one, i.e. pruned_weights[layer_name] += 1. I now knew the total number of weights per layer, and I also knew the number of weights that had been pruned by the pruning algorithm for each layer. Now I just needed to create some threshold criteria to test whether each layer should be pruned. Doing this was quite simple, all I had to do was divide the number of pruned weights for that layer by the total number of weights. I now had the ratio of pruned weights to total weights, and I could compare this number to some threshold that I defined in the code. Using a simple if statement, I could then check if the current layer had reached that specified pruning threshold. If it had not yet reached the required threshold, the program would continue with the pruning process. However, if that particular layer had reached the threshold, then from that point on, no more weights would be pruned from the layer. For example, if I chose a threshold of 0.5, and the program was running on a network with one layer of 500 weights, the program would prune the layers weights individually, until the program had pruned around 250 weights, and

then the pruning program would stop pruning that layer. So now, I had a program that could prune individual weights from each layer in a network based on the magnitude of their absolute value, would not prune the same weight over and over, and would only prune the weights of a layer whose ratio of pruned weights to total weights had not exceeded some threshold.

This solves the issue of not pruning a random weight, but the least significant one and also solved the problem of repeatedly pruning the same weight over and over. However, this is only useful for pruning individual weights, not $M * C * Y * X$ blocks. So, next, functionality needed to be added to the script in order to iterate over the 4D tensor and return the $M * C * Y * X$ block with the smallest absolute value. This worked in a similar way to the pruning of individual weights. Recall that for pruning individual weights, I began at first weight in the tensor i.e. $(0, 0, 0, 0)$, and then iterated through the tensor and stored the minimum absolute value of a weight found in the tensor, along with that weights position and it was then pruned. However, I was no working with blocks rather than individual weights, so I had to consider a new way of comparing the block sizes in terms of their weights. In order to do this, I parsed 4 new arguments from the command line using the argparse package, corresponding to the 4 dimensions of the block that I wanted to be pruned. So, I needed a way to measure the total weight of a block. To do this, at each position in the pruning process, I needed to sum the absolute values of all the weights of the $M * C * Y * X$ block at that position. Starting at the current M, C, Y, X position that the pruning process is currently at, using 4 nested for loops, I added up the absolute value of each weight in that current block. This total weight was then compared to the minimum stored weight. Just as before, if the weight was smaller than the minimum weight, the current weight was set to be the new minimum, the location was stored in $M_{min}, C_{min}, Y_{min}, X_{min}$ and the marked tensors value at that position was set to be 1. I now had functionality that could return

the position and weight of the smallest $M * C * Y * X$ block in the tensor.

However, this implementation causes a couple of issues, similar to the previous issue regarding pruning the same weight over and over. Firstly, I needed to make sure that I wasn't pruning shapes that overlapped in anyway. To better explain this, let's consider the problem in two dimensions. Say I wanted to prune a $2 * 2$ block from a layer of size $10 * 10$. If, on the first iteration through the layer, I found that position $(0, 0)$ had the smallest absolute values of combined weights, I would then prune the weight stored at $(0, 0), (0, 1), (1, 0)$ and $(1, 1)$ and the marked tensor would set the value stored at $(0, 0)$ to be equal to 1, i.e. it would indicate to future pruning steps that the block at $(0, 0)$ had been pruned. However, what if on the next iteration, the program chose the weight at $(1, 0)$ to be the smallest block? This block would contain the weights stored at $(1, 0), (1, 1), (2, 0)$ and $(2, 1)$. But two of these weights had already been pruned, so it would give an unfair advantage to the blocks that contained weights that had already been pruned, but had not been marked as pruned. The reason they had not been marked is that only the first weight of each block is marked as pruned. So why not just mark each weight as pruned? This doesn't fully solve the issue, as you could conceivably have weight that had not been marked contain weights that had been marked inside them. Say for example the block at $(1, 1)$ had been pruned. Which means that the weights $(1, 1), (1, 2), (2, 1)$ and $(2, 2)$ would have been pruned and marked as such. Despite being pruned and marked, the program could then choose $(0, 0)$ or $(1, 0)$ for example as the minimum block location and they would contain already pruned weights. Therefore, retreading over already pruned shapes would not work for this process. I would have to come up with some alternate solution of how to prune the smallest possible block, without taking into consideration any previously pruned weights.

The way I solved this issue was by iterated through the tensor according to the dimensions of the block. Let me explain by using the two dimensional

case in the previous paragraph. If I wanted to prune a $(2 * 2)$ block from a $(10 * 10)$ sized tensor, then I started at position $(0, 0)$ and sum the absolute values of the weights in the block at that position i.e. $(0, 0), (0, 1), (1, 0)$ and $(1, 1)$. Now, instead of moving along by one in the x-direction, I moved along by two, i.e., the length of the block shape in the x-direction. I then repeated this in the x-direction, until I reached the end of the tensor in the x-direction. I then did the same in the y-direction, and moved along by two, according to the y-dimension of the block. I then repeated this process for the entirety of the tensor. This method ensured that only regular blocks of the tensor was checked, and that there was no possible way to overlap the blocks. Doing this allowed me to be sure that I was actually finding the minimum values out of all the blocks in the tensor, and made sure that I wasn't selecting the incorrect block, just by virtue of its proximity to a block that had already been pruned. I've included some simple pseudo-code for the pruning individual weights algorithm below.

I now had a program that could choose the minimum $M * C * Y * X$ block of any size and shape (as long as the dimensions of the block were within the dimensions of the layer), prune the weights in that block, and then repeat the process until the desired sparsity level is reached. I also decided to add functionality that cut off the pruning process once it had reached some minimum accuracy. Obviously, when pruning a neural network, it's only useful if it maintains some level of accuracy. So I did not want to continue the pruning process when pruning shapes that resulted in an inaccurate network. To do this, after each block is pruned, the accuracy is tested, and the accuracy of the network is checked if it is below some set value, for example 40%. If the accuracy is below the minimum accuracy, then the pruning process is stopped. Otherwise, the pruning process continues. So now the program can run unabated and will finish when either the target sparsity is reached, or the accuracy of the network falls below the minimum.

Algorithm 1 Pruning

```
1: procedure PRUNING INDIVIDUAL WEIGHTS
2:     while accuracy > 40% do
3:         for layers in layer list do
4:             min value = 10000
5:             m_min, c_min, y_min, x_min = 0
6:             if sparsity < target sparsity then
7:                 for m in range(layer.length(m)) do
8:                     for c in range(layer.length(c)) do
9:                         for y in range(layer.length(y)) do
10:                            for x in range(layer.length(x)) do
11:                                if Weight has not been pruned then
12:                                    if weight > min value then
13:                                        min value = weight
14:                                        m_min = m
15:                                        c_min = c
16:                                        y_min = y
17:                                        x_min = x
18:                Prune(weight)
19:
```

Once this was all completed and I had a script that was capable of carrying out the pruning process, all I needed was data on which to run this script on. The data in this case was trained networks, i.e. predefined architectures such as LeNet, Alexnet, etc. that had been trained on some dataset e.g. MNIST, CIFAR10 etc. I only used predefined architectures for this research, because the topic of this research isn't really to tweak pre-existing architectures to improve them, I was only concerned bout the affect of pruning, so working with architectures that I knew would work well seemed like the most logical choice. I also chose to use some pre-existing datasets for this research, because, like using pre-existing architectures, I didn't have any specific use in mind for the neural networks I was using, I was just concerned with sparsity, so using pre-existing datasets for research would work fine. So now I had two different variables that I could change when carrying out the research: the dataset and the architecture. There is of course a third variable that I could use in this research, which is the shape of the block. From this point I knew that I could work with different variables, datasets, and blocks, so I just had to go about collecting this data.

To run pruning, I needed a trained network, in which each of the convolutional layers is saved as a 4D tensor. Using Caffe, this process was quite simple. Each of the pre-existing architectures that I used were included in Caffe, and were composed of two components: a solver and a model, both of which were stored as prototext files, similar to YAML files. The model definition defines the structure of the neural network i.e. the number of layers and their types e.g. dense layer, convolutional layer, etc. On the other hand, the solver file defines the parameters concerning the training and testing of the neural network, including: learning rate, momentum, number of epochs etc. To train a network, all I needed was a model prototext file, a solver prototext file and a dataset to train the network on. The Caffe framework contained a makefile for each combination of dataset and architecture, and

so to train any combination of dataset and architecture I need only run the makefile specific to that dataset and architecture combination. The output of this program is saved as a .caffemodel file.

From this point, I could train as many architectures and datasets as I wanted and therefore had some data to run pruning on. The next step was to decide which shapes to prune for each architecture. To do this, I needed to modify the pruning program, so that it would first print out the dimensions of each convolutional layer. I did this so as not to prune shapes that were larger in any of the four dimensions M, C, Y, X than the smallest value of these dimensions in the networks layers i.e. when working with LeNet, which contained two convolutional layers of size $20 * 1 * 5 * 5$ and $50 * 20 * 5 * 5$, the maximum block I could prune would be $20*1*5*5$, because 20 is the smallest value in the M dimension, 20 is the smallest value in the C dimension, and so on. This process is not only useful for determining the upper limits of the blocks dimensions, but it also helped me decide which shapes I should consider for pruning. For example, a $1 * 1 * 4 * 4$ shape may not work that well, as it could only prune one shape from each kernel, which in principal would be the same as just pruning a $1 * 1 * 5 * 5$ block. For each architecture and dataset combination, I checked the dimensions of the layers and decided which shapes I would prune for each one.

I had to have some way of reading the results that were produced from the pruning program. To do this, after each block was pruned from the network, I calculated the accuracy of the network, which of course meant I had to test the network in some way. Testing the network involved using a piece of data that was not used in the training data, and checking if the network got the right answer or not. If it got the answer correct, a counter would be incremented. If it didn't get the answer correct, the counter would not be incremented. In the command line arguments, I added an argument for the number of test iterations the program would carry out. The program would

then repeat this testing process for the number of test iterations specified in the command line. After this process, the number stored in the counter would be divided by the total number iterations defined at the command line. This would give me accuracy after each block was pruned. After each pruning step, I then printed the accuracy, the total number of weight pruned (this was kept track of for each layer by iterating a counter associated with that layer each time the pruning function was called), and the total number of weights. I separated each of these values with a comma, and saved the values to a csv file.

After finishing all this code and testing it, I then began the process of choosing which architectures and datasets to use, as well as which shapes to test for each architecture and dataset combination. I then ran each of these combinations and saved them all as csv files, named each according to the dataset, architecture and shape of block. I now had a large set of data and I just had to find a way to visualise the data so that I could view the performance of the blocks compared to one another. Firstly, I needed a way to condense the data slightly. In particular for the smaller blocks, there could be thousands of datapoints, which resulted in a very cluttered plot, as the accuracy of the network was not steady, and would vary quite a bit in consecutive datapoints.In order to combat this, I wrote a very simple python program, that read only the n-th datapoint in a csv file. I could change this n value depending on the size of the block, with a smaller n for larger blocks. After parsing the data, I could then use matplotlib to plot the data I wanted. In the interest of completeness, I would plot all the shapes I had attempted for the particular architecture/dataset combination. From this, I would select groups of blocks that had performed similarly well and attempt to draw some correlations between them, as well as comparing blocks of equal size to see how they performed in comparison to one another.

4 Results

4.1 MNIST and LeNet

The first architecture and dataset combination I worked with was LeNet and MNIST. LeNet was a simple architecture that contained only two convolutional layers. MNIST is a dataset compromised of simple, monochrome images of numbers. I trained the network using the MNIST dataset and then saved the resulting model as caffemodel. I then imported it into the pruning program, and began pruning shapes of different sizes and shapes. Before pruning, I checked what the maximum dimensions were for the blocks (i.e. the minimum dimensions of any layer in the network), and in the case of LeNet, the maximum dimensions allowed for a block was $(20 * 1 * 5 * 5)$. So I came up with a number of shapes to prune and began the pruning process. After I had collected data for all the shapes I wanted to test, I fed the data into the plotting program I had written. Below is a plot of all the shapes accuracy vs. sparsity:

In this plot, several bands of similarly performing blocks are visible. I wanted to know which shapes fit into what band, so that I could look for similarities in the blocks in each band. To do this, I turned the opacity of each plot down from 1 to 0.1, except for one of the blocks, and then noted what bad it was in. I repeated this process for each block in the plot. After I knew what band each block was in, I then turned down the opacity for all the blocks, except for the blocks in band one. I repeated this for all the bands, and saved the plots, along with a legend for all the highlighted blocks.

The first band that I isolated was the band containing the best performing blocks, located at the top of the plot. I wanted to see if these blocks had any similarities, or common structural properties. Below is the plot of all blocks that are contained within band one.

As is to be expected, pruning individual weights i.e. $(1 * 1 * 1 * 1)$ sized

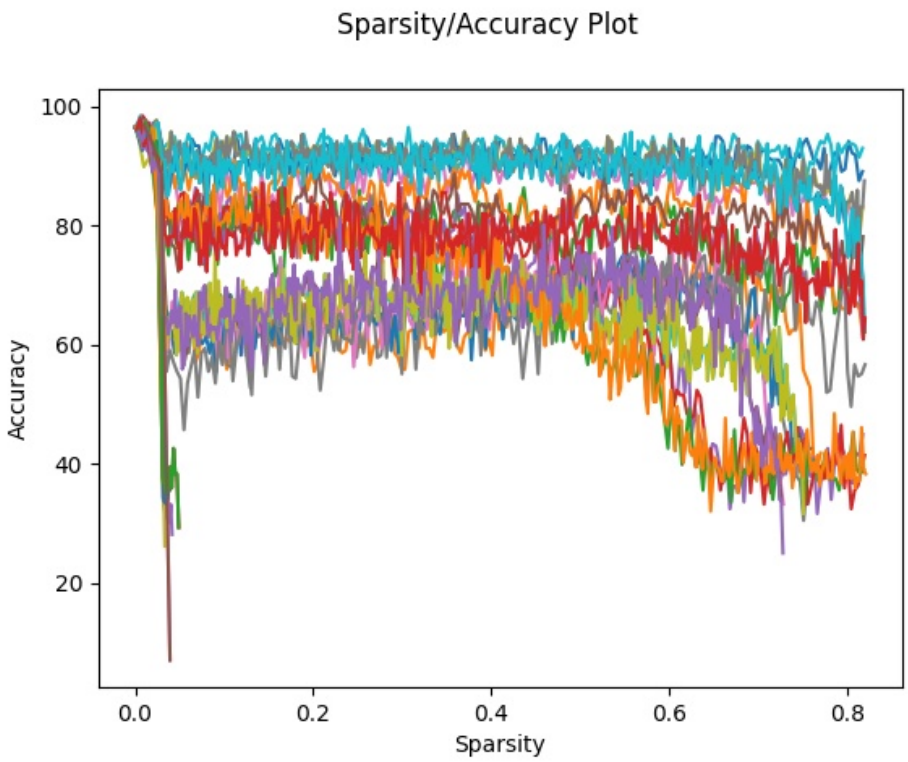

Figure 8: All shapes tested using LeNet and MNIST.

blocks, performed very well. Although pruning individual weights results in a high performing network, the non-zero weights remaining are not in regular patterns, and so this pruned network would be difficult to implement into a fast convolution routine. The larger the blocks that are pruned, the more regular the remaining non-zero elements remaining are. The other blocks in this band tend to follow similar shape patterns. Generally, if the block is of shape $(M * 1 * 1 * 1)$ or $(M * 1 * 1 * 5)$, they are contained in this block. It appears that increasing the block in the M-direction (the number of output feature maps) does not have a major impact on the performance of the block. This may have something to do with that fact that same (X, Y) coordinates of a feature map are likely to have the same level of importance

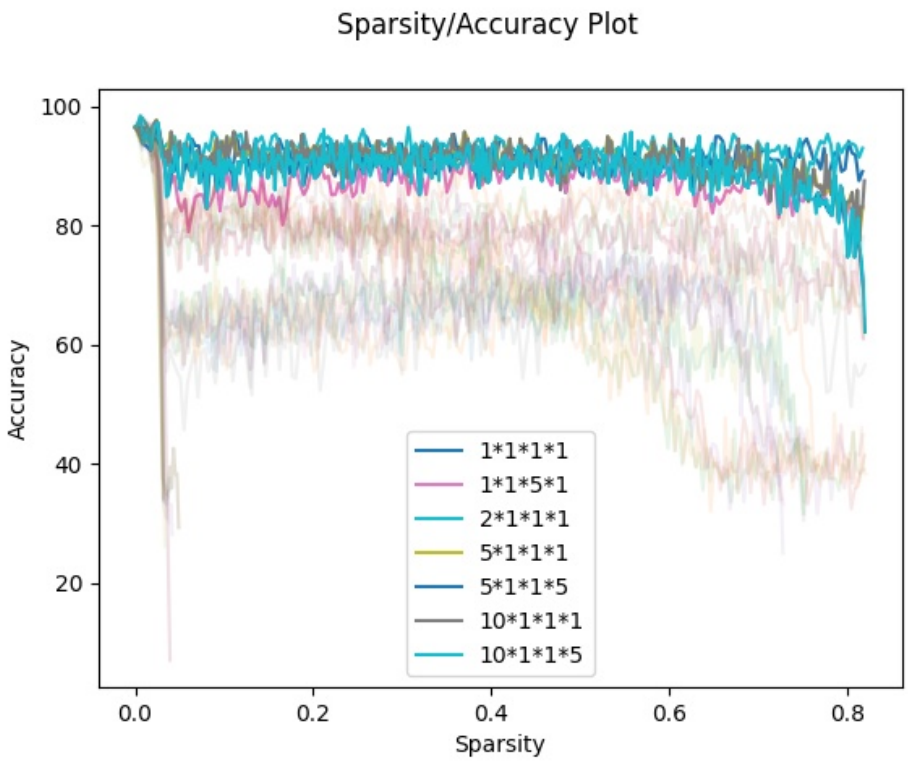

Figure 9: LeNet and MNIST: Band One

compared with that same coordinate in other feature maps. For example, say at position $(0, 0)$ in the original image is just a black pixel(i.e. without any visual information for a monochromatic image), then that corresponding pixel in the feature maps will likely be of similar importance. So instead of pruning just one weight from one feature map, pruning 1 weight each from 10 feature maps with the same (X, Y) coordinates performs just as well and also maintains a more regular weights pattern in the network. This combined with the fact that $(N * 1 * 1 * 5)$ blocks tend to perform well, means that $(10 * 1 * 1 * 5)$ blocks can be pruned from the network whilst maintaining a high accuracy. These blocks are 50 times the size of regular blocks, so the pruning process is much faster, and the remaining network structure is much

more regular.

Next, I isolated the second band. This band contained blocks that performed similarly to the first band, except the accuracy of blocks in this band were approximately 10% less than the accuracies in the first band. The blocks in this band however, despite being less accurate, did not lose much accuracy as the sparsity of the network increased. Below is the plot of the second band:

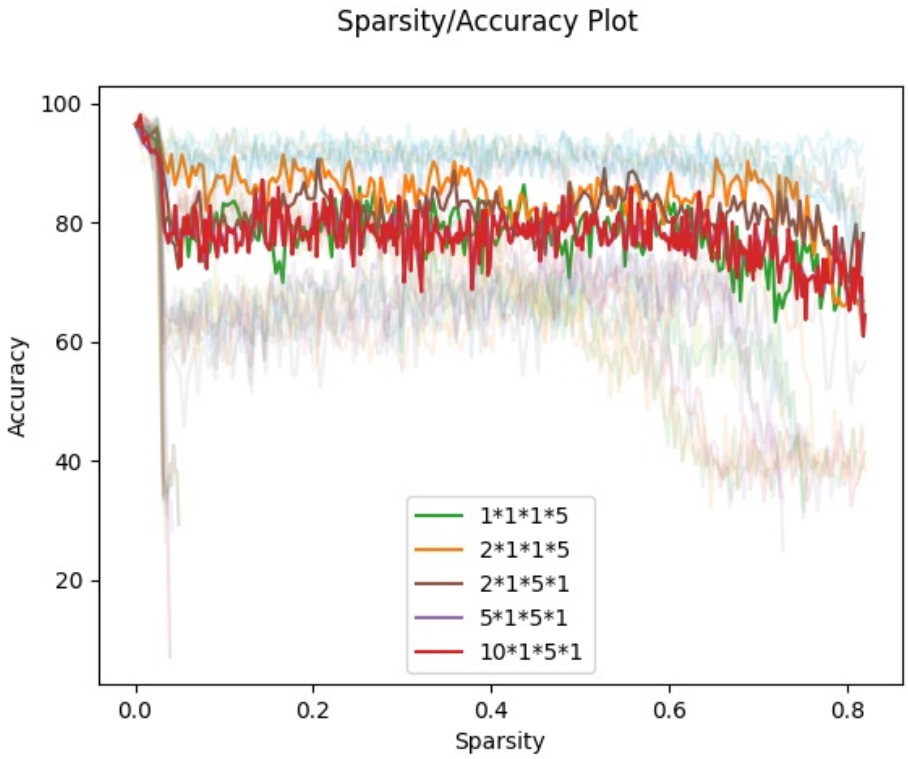

Figure 10: LeNet and MNIST: Band Two

The performance of the blocks in the second band varied more than in the first band, but generally, their performance dropped sharply after the first few pruning steps and from there their accuracy remained fairly consistent, up until around 70 % sparsity, at which point, some blocks became unstable,

and their performance started to drop off slightly. The shapes of the blocks in this band tended to be of the shape $(N * 1 * 5 * 1)$ (i.e. shapes that pruned blocks of maximum size in the y-direction), and some of the shape $(N * 1 * 1 * 5)$, most of which were found in the first band. It is also worth noting that there is some ambiguity as to which band some of the blocks were a part of. For example, the blocks $(1 * 1 * 5 * 1)$ and $(2 * 1 * 1 * 5)$, had some overlap, and it was difficult to determine which band each of the blocks belonged to. A closer look at these two bands can be found below:

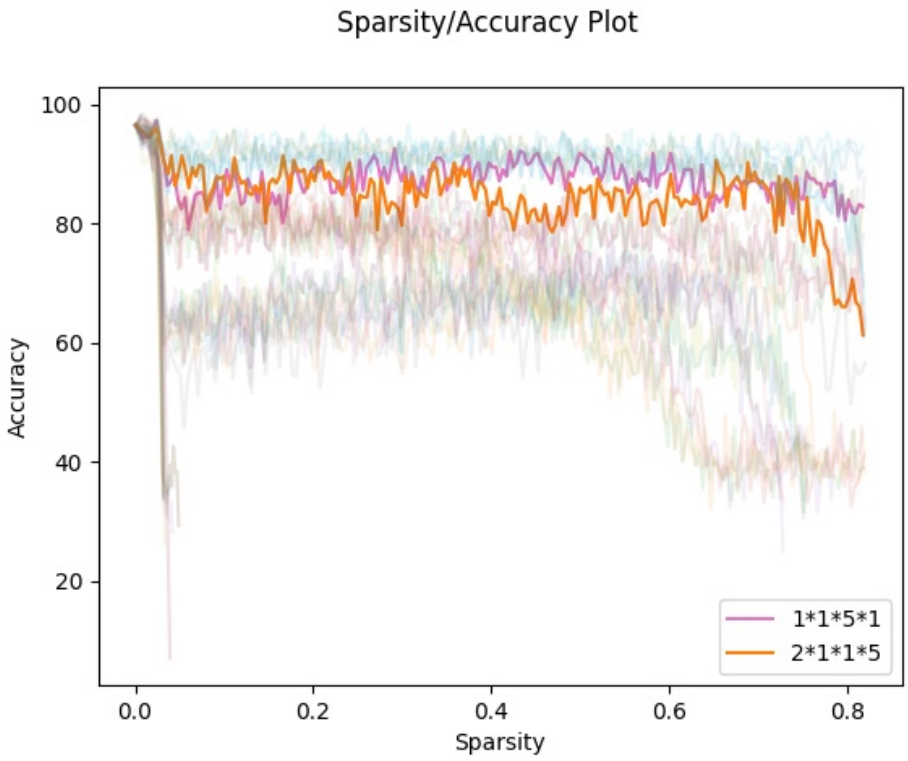

Figure 11: LeNet and MNIST: $(1 * 1 * 5 * 1)$ and $(2 * 1 * 1 * 5)$ blocks

As can be seen in the plot of these two blocks, they overlap for much of the plot, however the $(2 * 1 * 1 * 5)$ drops off toward at around 70% sparsity, so I included it in the second band. The $(1 * 1 * 5 * 1)$ block is of the form

$(N * 1 * 5 * 1)$, so that would explain why it is close to the second band, as blocks of this form tended to be in the second band, however the blocks were still relatively small, with a size of only five, and pruning smaller blocks tends to lead to be a better accuracy. I would have expected $(2 * 1 * 1 * 5)$ to perform better, as most of the blocks of shape $(N * 1 * 1 * 5)$ tended to be in the first band.

Next, I plotted all of the blocks in the third band. The blocks in this band immediately drop down to about 60% accuracy and remain around this accuracy until the network reaches a sparsity of about 0.6, at which point most of the blocks start to lose some accuracy. Like the second band, the accuracy of the blocks in this band vary more than in the first band, however, once they drop down in accuracy initially, they tend to stay quite consistent, as the level of sparsity increases. Below is the plot of the third band:

Interestingly, the accuracy of blocks in this band actually start to slightly increase between a sparsity level 0 and 0.6, before dropping off. The blocks in this layer tended to have a either 2 or 5 in the x or y dimension, but not both. The low accuracy in comparisons to other block shapes may have to do with the fact that 2 does not evenly divide into the dimension in the X and Y direction of each layer, i.e. the (X, Y) dimensions of the kernels in this network are $(5, 5)$, therefore, the fifth weight in either dimension cannot be pruned if the X or Y dimension is 2. That is to say that the only two combinations of weights the program could prune is the first and the second, or the third and the fourth. So this inability to actually choose the fifth position in the X or Y direction may account for the low accuracy. As was the case for the first and second band, increasing the dimensions in the M direction does not appear to have an effect on the accuracy as sparsity increases. This is more evidence to indicate that if one wanted to use a bigger block size, they may want to increase the block size in the M direction first.

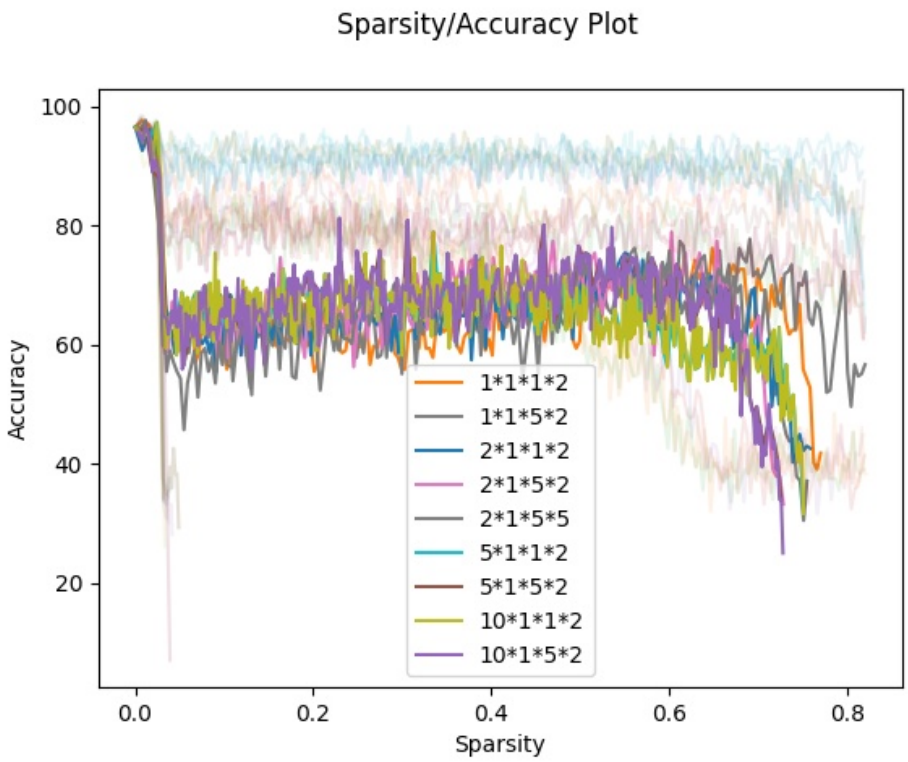

Figure 12: LeNet and MNIST: Third Band

After this I plotted the fourth band. Blocks in this band dropped in accuracy immediately to around 80%, and from there decreased in a curve that slightly resembled a inverse exponential curve, until it reached a sparsity level of around 0.6, at which point it leveled off at a accuracy of around 40%. Obviously, shapes like this should be avoided, because pruning blocks that result in a network with an accuracy that low are not very useful in the real world. Below is the plot for the fourth band:

The shapes in this band all had the exact same structure, that is $(N * 1 * 2 * 2)$. Not only that. but every block I tested with this shape appeared in this band. Blocks of this shape appear to exhibit the same issue as the blocks in the third band. Namely, they don't divide evenly into the dimensions of

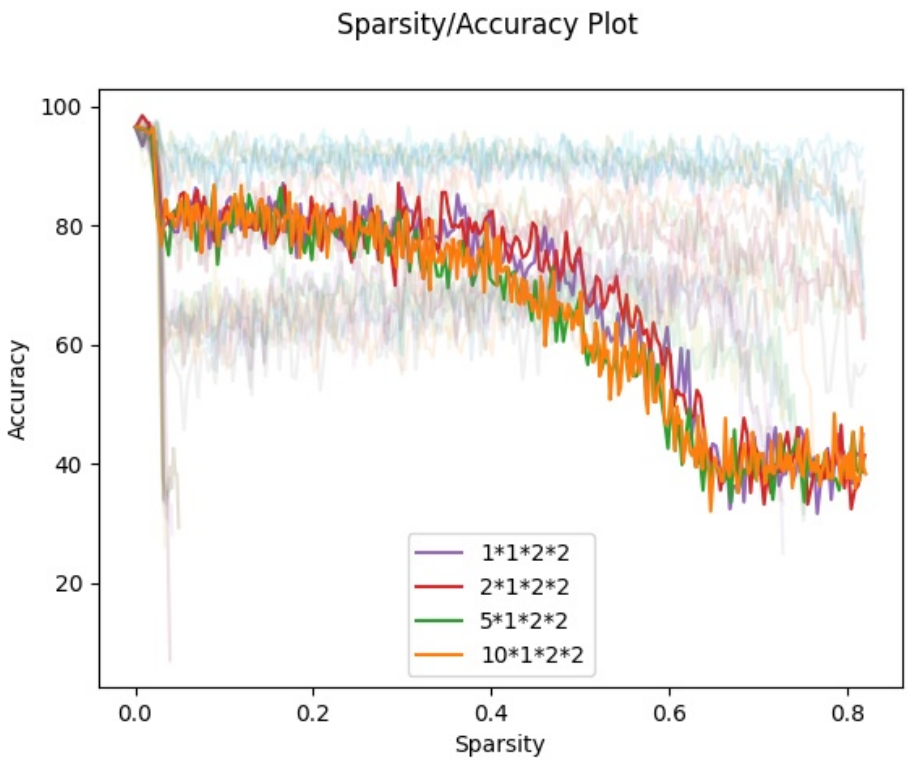

Figure 13: LeNet and MNIST: Fourth Band

the kernel of each layer. The following visualisation of a two dimensional case, explains this better:

The shapes in this band all had the exact same structure, that is $(N * 1 * 2 * 2)$. Not only that. but every block I tested with this shape appeared in this band. Blocks of this shape appear to exhibit the same issue as the blocks in the third band. Namely, they don't divide evenly into the dimensions of the kernel of each layer. The following visualisation of a two dimensional case, explains this better, as can be seen in figure 7. The coloured squares of size $(2 * 2)$, each represent the possible blocks that be pruned from the network. The white squares represent the weights in the kernel that are not accessible by the program and therefore the weights that cannot be pruned.

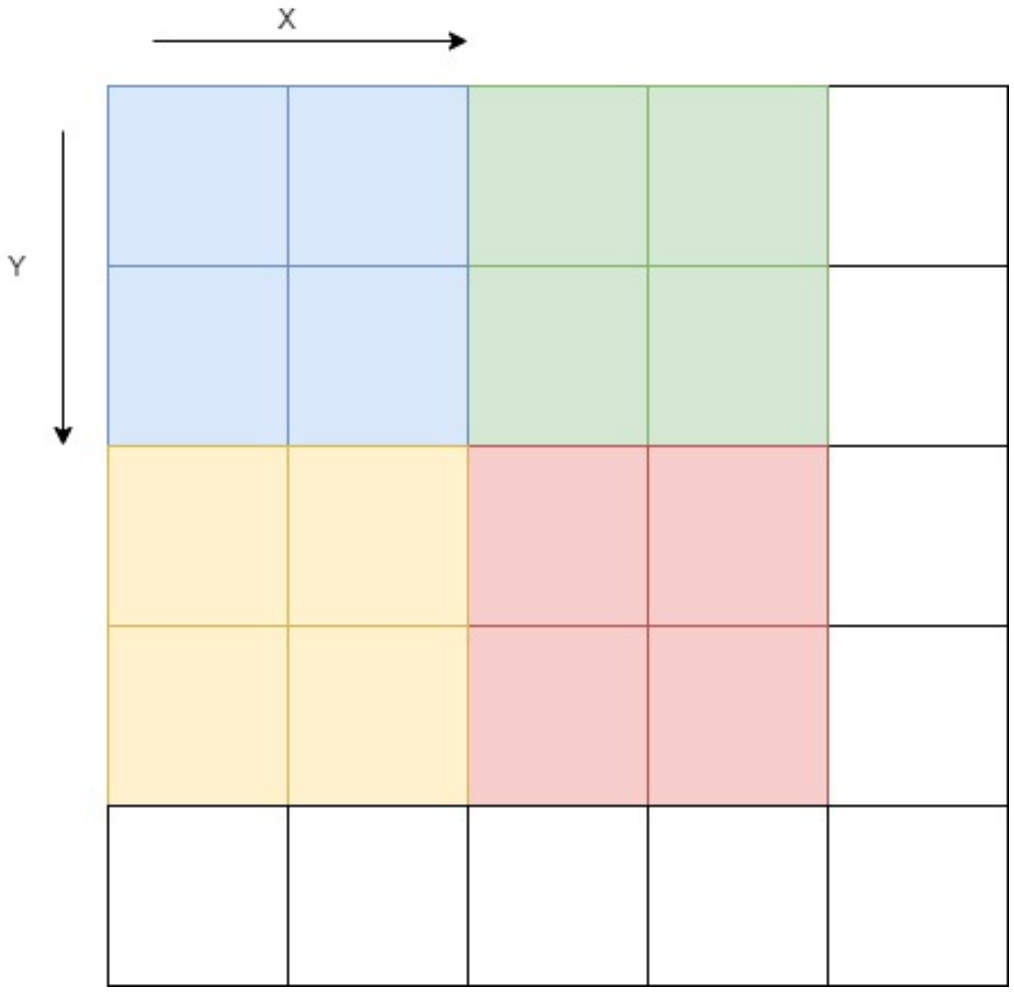

Figure 14: Pruning a (2*2) block from a (5*5) kernel

This would explain why these shapes actually appear to work reasonably well at the start, as 16/25 of the weights (i.e 64%) are able to be pruned, and so the minimum being accessible is more likely that not. However, as the sparsity increases, the likelihood that the minimum weights are in a position that is able to be by the program decreases, and therefore, the accuracy of the network begins to drop more sharply as the sparsity increases. The performance of these blocks indicates that choosing block dimensions that fit

evenly into the dimensions of the kernel is imperative for maintaining a high accuracy as the sparsity increases.

Finally I plotted the fifth band. This band was comprised of the blocks that performed particularly badly. I decided to plot these, as I felt it was important to try an identify any similarities between the blocks that performed badly. There was quite a large number of blocks in this band, and all of them fell well below the cut-off threshold for accuracy before any of them had reached a sparsity level of 0.1. These blocks would obviously have no practical uses for this combination of architecture and dataset. Below is the sparsity/accuracy plot for the blocks in the fifth band:

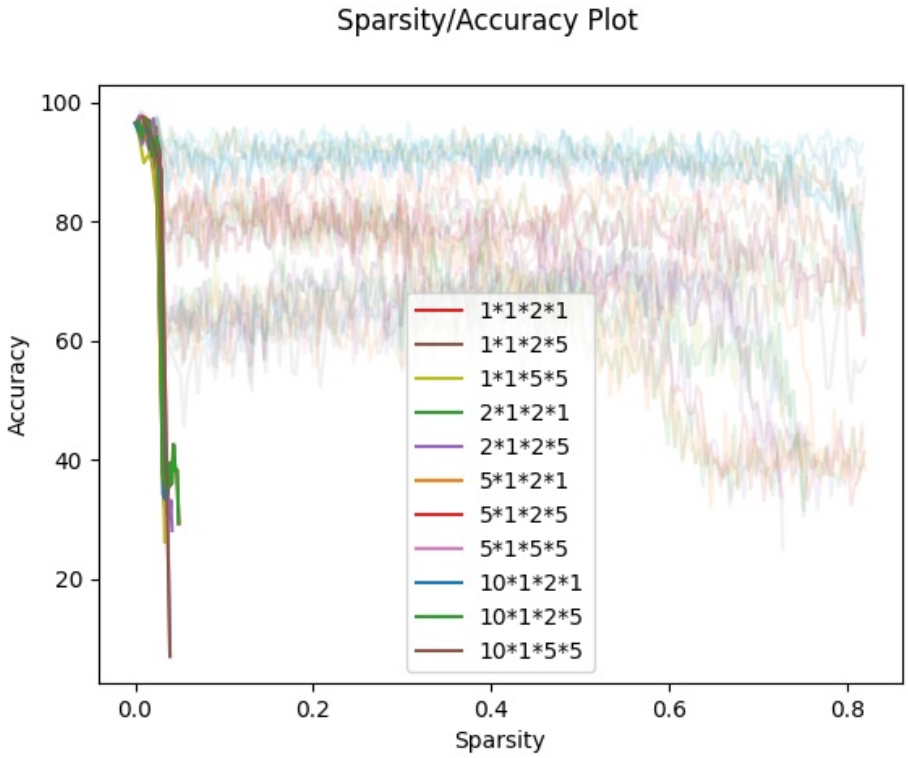

Figure 15: LeNet and MNIST: Fifth Band

The blocks in this band generally were either very large blocks e.g. $(10 *$

$1 * 5 * 5)$, $(5 * 1 * 5 * 5)$, which are of size 250 and 125 respectively, blocks that had dimensions in the y direction that did not divide evenly into the y dimension of the kernel of each layer, or were of the form $(N * 1 * 5 * 5)$. It is to be expected that extremely large blocks would perform badly, because as the size of these blocks increases, the pruning process would be removing such large clusters of weights that the chance of not removing a weight that was important to the network was slim. On the other hand, as with the previous two bands, some of the block shapes found in this band were of the form $(N * 1 * 2 * M)$. So, having a pruning shape with a y dimension that does not divide evenly into the y dimension of the kernel proves to be disastrous for pruning, as 73% of the blocks in the fifth band are of this form. Finally, there were several block shapes in this band of the form $(N * 1 * 5 * 5)$, which was surprising to me, as the block $(2 * 1 * 5 * 5)$, appeared in the third block, and so I would have expected shapes of this form to perform better than they did, especially $(1 * 1 * 5 * 5)$, as that was smaller in size than $(2 * 1 * 5 * 5)$.

On the topic of block size (in this case, size is referring to the number of weights in the block), I was also interested in testing if the size of a block had much of an effect on the performance. It would of course have some effect, but I was interested if blocks of the same size performed similarly at all, or if there were other factors that were impacting the performance of the block. I decided to plot the performance of blocks that had size 20 and 10, as these two shape sizes were the most common among the blocks I had tested. Below are the two accuracy sparsity plots for blocks of size 10 and size 20:

As can be seen in these two plots, the performances of blocks of the same size can vary wildly. In the case of blocks of size 10, $(1 * 1 * 2 * 5)$ and $(5 * 1 * 2 * 1)$ both performed extremely badly, however, on the other hand, $(10 * 1 * 1 * 1)$, managed to keep an accuracy above 90% until a sparsity level of around 0.8, which is of course a great performance. Similarly, the performance of blocks of size 20 also varied greatly, with some blocks, such

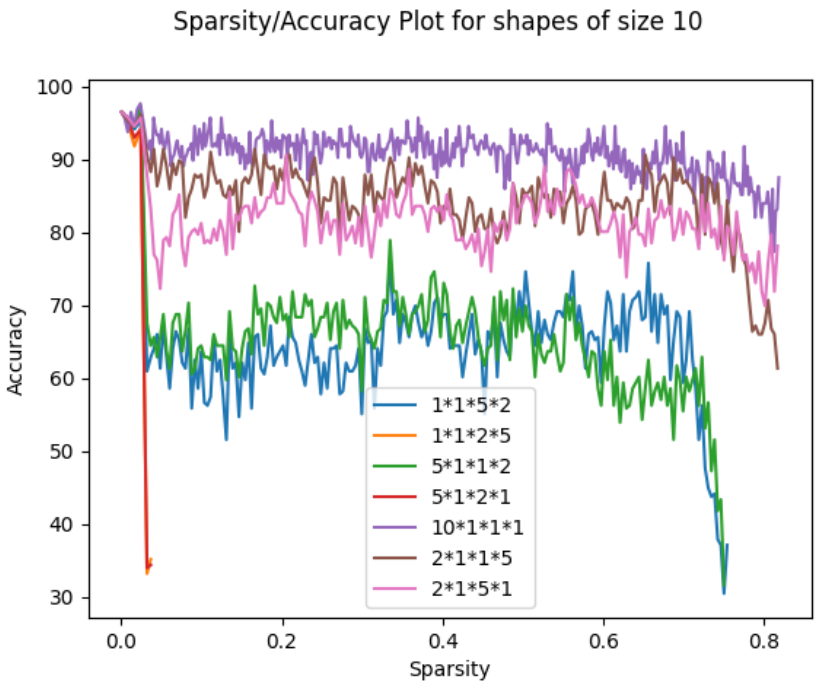

Figure 16: LeNet and MNIST: Shapes of size 10

as $(2 * 1 * 5 * 2)$ and $10 * 1 * 1 * 2)$, maintaining a reasonable accuracy until a sparsity level of around 0.7 in the case of $(2 * 1 * 5 * 2)$ and 0.6 in the case of $(10 * 1 * 1 * 2)$. On the other hand $(2 * 1 * 2 * 5)$ and $(10 * 1 * 2 * 1)$ performed very badly. They both immediately fell below the accuracy threshold for the pruning program. It is worth noting that although the performances of shapes can vary wildly, on average, smaller shapes still do tend to maintain accuracy better, as can be seen in the two plots, as on average, blocks with smaller sizes result in better performance.

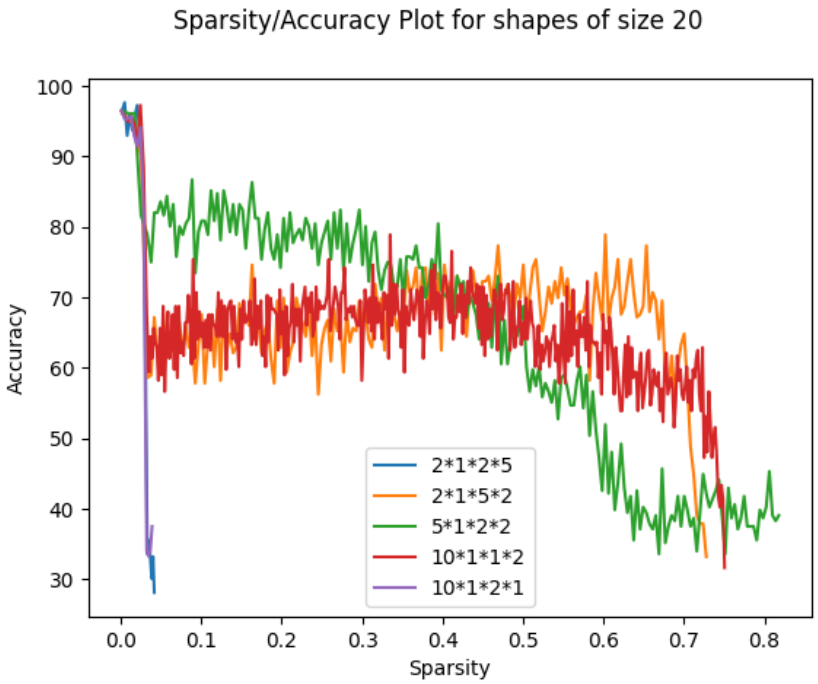

Figure 17: LeNet and MNIST: Shapes of size 20

4.2 CIFAR10 and LeNet

The next dataset and architecture combination that I pruned was a LeNet architecture trained on a the CIFAR10 dataset. The CIFAR10 dataset is a collection of images split into 10 categories. Each of the images in in colour, and so has 3 input channels for red, green and blue. The LeNet architecture is the same as the one used before, however, due to the 3 colour channels, the input dimension in the C direction has increased from 1 to 3. This dataset is therefore slightly more complex than the MNIST dataset, and the same architecture can be expected to perform slightly worse than compared to the MNIST dataset. Before I began the pruning process, I first trained the LeNet architecture on the CIFAR10 dataset using Caffe and saved the model as a caffemodel. I then imported the network to the pruning program.

The maximum dimensions of the pruned block of a LeNet architetcure trained on a CIFAR10 dataset is larger than the previous case, that is $(20 * 3 * 5 * 5)$. The only dimension that increases is C, because this corresponds to the input channels, i.e. RGB. This means that the number of possible pruning shapes has doubled from the previous example, as I could prune shapes of the form $M * 1 * Y * X)$ and $M * 3 * Y * X)$, rather than just $M * 1 * Y * X)$. This effectively doubles the number of possible pruning shapes. As was the case with pruning a $1*1*2*2)$ block from a $20*1*5*5)$ layer, it didn't make sense to prune shapes of size $M*2*Y*X)$, as it would be the same as pruning $M*3*Y*X)$ blocks. Once I had decided on all the shapes I wanted to prune, I pruned the network with these shapes, and after each pruning step, I recorded the sparsity and the accuracy, and saved in all in csv file for each shape, as before. I then plotted the accuracy and sparsity relationship for each shape, as can be seen in figure 18.

The first thing worth noting about the performance of these blocks is that the base accuracy has been lowered from approximately 90% to approximately 75%. If I left the accuracy threshold at 40%, many of the blocks

with interesting results would have been cut off too soon. In order to combat this, I lowered the accuracy threshold to 20% accuracy. The lower accuracy of this network can likely be attributed to the more complex nature of the data, with 3 times as many input channels as before. With this in mind, the performances of pruning, both by weight and in blocks, can expected to be less effective, as LeNet is a relatively simple network, and so pruning can be expected to have a more drastic effect on the performance as sparsity increases. As can be seen in figure 18, many of the pruning blocks that I pruned from this network fell quite sharply at a low level of sparsity (i.e. between 0.1 and 0.2). Although there were many shapes in the previous case, a significant amount of them failed to maintain accuracy as sparsity increased. In fact, approximately 86% of the shapes pruned from this network failed to make it beyond a sparsity level of 0.2 before their accuracy dipped below the accuracy threshold, (i.e. 20% in this case). Obviously, this means that more complex datasets result in a smaller subset of pruning shapes viable. It also may be attributed to the fact that LeNet is quite a simple network, and when it is trained on a more complex dataset i.e. CIFAR10, the starting accuracy isn't as high (75% compared to 90%), as the training isn't as effective as when training on MNIST. The dataset may have just been too complex for the LeNet architecture. If this was the case, then there may have been too many essential weights, that pruning any large blocks resulted in a large drop in accuracy.

However, despite the large number of failed blocks, there were several blocks that did perform well. The blocks that performed well were of a similar structure as the previous dataset and architecture combination's high performing blocks. I isolated the blocks that performed well and plotted them. The resultant plot can be seen in figure 19.

As expected, the pruning shape that performed the best was the $(1 * 1 * 1 * 1)$ block, which is of course just the pruning of individual weights.

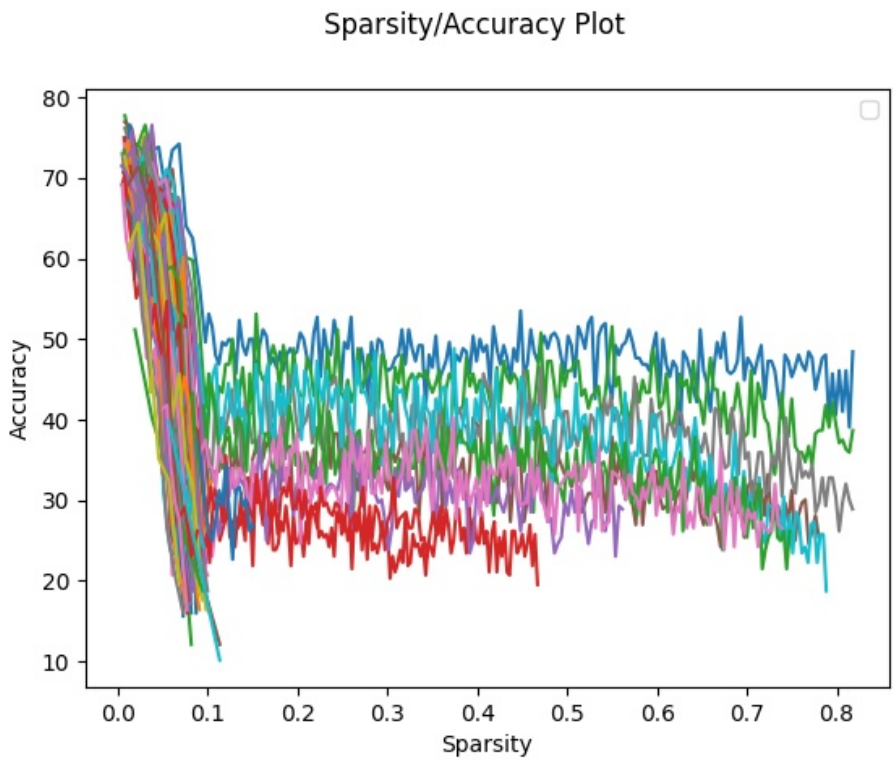

Figure 18: LeNet and CIFAR10: All shapes

Most of the weights that remained were of the form $(M * C * 5 * 1)$ or $(M * C * 1 * 5)$, which is in line with the behaviour of shapes from the LeNet/MNIST case. Additionally, no blocks had 2 in either the X or Y dimension. This makes sense, as even in the LeNet/MNIST case, when the accuracy was much higher, pruning shapes with 2 in the X or Y dimension resulted in a very sharp drop in accuracy. So, in the LeNet/CIFAR10 case, which is more sensitive to pruning, it makes sense that the network would be very intolerant of pruning shapes of this form. A difference between the LeNet/CIFAR10 case and the LeNet/MNIST case is that changes in the pruning blocks M-dimension seem to have a much larger effect on the accuracy as sparsity increase in the LeNet/CIFAR10 case as compared to

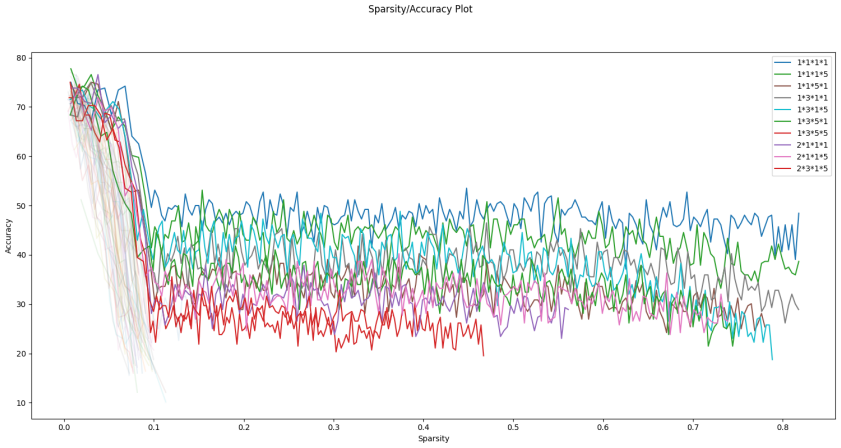

Figure 19: LeNet and CIFAR10: High performing shapes

the LeNet/MNIST case. In the LeNet/MNIST case, the images that were being used were very simple, so it is more likely that the respective weight in another filter would be of relatively the same importance to the pruning process, so this worked quite well in that example. However, due to the more complex nature of the images in the CIFAR10 dataset, this being the case in the CIFAR10 dataset seems unlikely, and that is reflected in the results. As the size of the blocks pruned increased, generally, the accuracy drops more sharply, and more or less maintains that accuracy up to high levels of sparsity.

This set of results was less promising than LeNet/MNIST results. I would have expected similar results to the LeNet/MNIST results, whereby pruning the smallest shapes would result in the same accuracy being maintained as the degree of sparsity increased. However, even pruning individual weights resulted in a sharp drop of accuracy by approximately 20%. Once this accuracy was reached, the network maintained a relatively steady accuracy as the sparsity increases. In fact, all of the blocks that didn't drop below the accuracy threshold had a large initial drop, with larger blocks having a

larger initial drop, and smaller blocks having a smaller one, but after that initial drop, the majority of them maintained the accuracy that they fell to. While not entirely useful in this case, as the accuracy of the network dropped below an accuracy that would be useful in any way, the principle of sparsity remained intact, ignoring of course the initial drop. Perhaps for larger networks, pruning results from the LeNet/MNIST case would be easier to replicate.

5 Conclusion

5.1 Conclusion

The pruning process worked very well in the LeNet and MNIST case. Generally, pruning shapes that were small in size, maintained a better accuracy as the sparsity increased than shapes that were large in size. However, increasing the size of the block in different dimensions had different effects, so accuracy could be maintained for high sparsity levels for larger shapes than might be expected. For example, pruning a block of size $(10 * 1 * 1 * 5)$ maintained an accuracy of approximately 90% up to a sparsity level of 0.8. These blocks were 50 times larger than pruning weights, which is a significant improvement in terms of the regularity of the blocks that remained. This also means that the pattern that remains in the network is 50 times more regular than pruning individual weights. The more regular the pattern of weights that remain in a network are, the easier it is to develop a fast convolution routine, which in turn makes it easier to implement on embedded systems. For this architecture and dataset combination, removing weights of the form $(M * 1 * 1 * 1)$ and $(M * 1 * 1 * 5)$ appeared to work best. The shapes that tended to not maintain accuracy very well were shapes that had either their X or Y values not divide evenly into the dimensions of the layer. This makes

sense as it effectively eliminates 20% of possible weights that can be pruned. Blocks that had either 1 or 5 in these dimensions consistently outperformed blocks that had 2 in either of these dimensions. In particular, blocks of length 2 in the Y dimension performed particularly badly, and dropped off in accuracy almost immediately. Also, very large blocks, such a $(10*1*5*5)$ performed poorly, and their accuracy dropped very sharply too. Blocks of this size are so large that it can be expected that pruning would result in removing essential weights, and so would result in a poor performance. However, pruning in general did not work as well with CIFAR10 and LeNet. A much larger proportion of the shapes experienced a sharp drop in accuracy compared with MNIST and LeNet. This can likely be attributed to the fact that CIFAR10 is a more complex dataset compared with MNIST, and due to the fact that I had used the same network for both CIFAR10 and MNIST. The data being more complex likely meant that more of the weights in the network were essential for it to be accurate, so pruning large shapes resulted in a large drop in accuracy. The base accuracy in this case was also lower than in the previous case, which likely was a factor also. Some shapes managed to perform decently in CIFAR10 and LeNet however, and these shapes were of a similar shape to the shapes that performed well in the MNIST and LeNet case. Like before, pruning individual weights performed the best out of any shape, as is to be expected. Shapes of the form $(M * C * 1 * 1)$ and $(M * C * 1 * 5)$ also performed well. However, unlike in the MNIST and LeNet case, increasing the size of block in the M direction did not work as well in the CIFAR10 and LeNet case. This could be attributed to the fact that CIFAR10 had three input channels, for red green and blue, and also that the data is more complex in general. The CIFAR10 data may have needed a more complex architecture in order to see the benefits of pruning more clearly. Additionally, a more complex and robust pruning metric may have been necessary for these more complex datasets and architectures.

5.2 Future Research

I had initially planned to prune a multitude of shapes from the AlexNet architecture, however, AlexNet is so large that I could not run a pruning routine on the architecture using my machine. So for future research, I'd like to see more architectures and datasets pruned, as the larger the network being pruned, the larger set of possible shapes that could be pruned.

I'd also like to see more pruning metrics tested, as the metric used in this research was quite simple, and a more robust and complex pruning metric may lead to better weights being pruned than the weights chosen by the metric used in this research.

References

[1] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, "Training large neural networks with constant memory using a new execution algorithm," *ArXiv*, vol. abs/2002.05645, 2020.

[2] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[3] D. Yu, F. Seide, G. Li, and L. Deng, "Exploiting sparseness in deep neural networks for large vocabulary speech recognition," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4409–4412, 2012.

[4] Y. L. Cun, J. S. Denker, and S. A. Solla, *Optimal Brain Damage*, p. 598–605. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[5] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, p. 84–90, May 2017.

[7] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, and G. Wang, "Recent advances in convolutional neural networks," *CoRR*, vol. abs/1512.07108, 2015.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[10] M. S. Zhang and B. C. Stadie, "One-shot pruning of recurrent neural networks by jacobian spectrum evaluation," *ArXiv*, vol. abs/1912.00120, 2019.

[11] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," *CoRR*, vol. abs/1705.08922, 2017.

[12] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," *ArXiv*, vol. abs/1902.09574, 2019.

[13] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A python package for dnn compression research," *ArXiv*, vol. abs/1910.12232, 2019.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *CoRR*, vol. abs/1408.5093, 2014.

[15] M. Thom and G. Palm, "Sparse activity and sparse connectivity in supervised learning," *Journal of Machine Learning Research (JMLR)*, vol. 14, 04 2013.

[16] I. Lemhadri, F. Ruan, and R. Tibshirani, "A neural network with feature sparsity," *ArXiv*, vol. abs/1907.12207, 2019.

[17] A. Kusupati, V. Ramanujan, R. Somani, M. Wortsman, P. Jain, S. M. Kakade, and A. Farhadi, "Soft threshold weight reparameterization for learnable sparsity," *ArXiv*, vol. abs/2002.03231, 2020.

[18] Kevin, Park, Fredrick, Zhang, Shuai, Qi, Xin, and Jack, "ℓ_0 regularized structured sparsity convolutional neural networks," *ArXiv*, Dec 2019.

[19] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, "Fast sparse convnets," *ArXiv*, vol. abs/1911.09723, 2019.

[20] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *CoRR*, vol. abs/1608.03665, 2016.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278 – 2324, 12 1998.

[22] X. XIAO, Z. Wang, and S. Rajasekaran, "Autoprune: Automatic network pruning by regularizing auxiliary parameters," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, eds.), pp. 13681–13691, Curran Associates, Inc., 2019.

[23] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," *CoRR*, vol. abs/1611.06440, 2016.