The status of research in sparsity/pruning of deep neural networks (DNNs)

by

Nirmala Gnanaratnam

A Master's Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF Engineering

in the Department of Electrical and Computer Engineering

University of Victoria

We acknowledge and respect the lək̓ʷəŋən peoples on whose traditional territory the university stands and the Songhees, Esquimalt and W̱SÁNEĆ peoples whose historical relationships with the land continue to this day.

The status of research in sparsity/pruning of deep neural networks (DNNs)

by

Nirmala Gnanaratnam

Supervisory Committee

Dr. Nikitas Dimopoulos, Supervisor

(Department of Electrical and Computer Engineering)

Dr. Mihai Sima, Departmental Member

(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Nikitas Dimopoulos, Supervisor

(Department of Electrical and Computer Engineering)

Dr. Mihai Sima, Departmental Member

(Department of Electrical and Computer Engineering)

ABSTRACT

Even though deep neural networks (DNNs) were first proposed around 1960s there was a rapid progress in related research starting from about 2012. This was due to the availability of large public datasets, cheap compute that could efficiently run these data driven algorithms, the rise of open source ML platforms and the resulting spread of open source code and models. In addition DNN research has attracted a lot of funding and is of high commercial interest. All of these reasons have contributed to a high volume of research papers; for example in sparsity/pruning DNNs about one paper per couple of days published on arXiv and growing exponentially.

Pruning is about training a network that is larger than necessary and then removing parts that are not needed during inference so that lesser resources are required to store it and less compute to execute the trained network. Even from early days researchers observed that neural networks converge easily while training if the network is large and used it as an experimental heuristic. The published literature on ‘pruning’ show many ways to identify the aforementioned useless parts or removing them before, during or after training. It even turns out that not all kinds of pruning actually allow for accelerating neural networks, which is supposed to be the whole point of pruning.

Moreover, due to the fact that these research areas are quite new and in a rapidly developing stage based mostly on experimental methods there is some concern in the research community about the quality of published research. The purpose of this report is to consider research conducted in deep learning in general and sparsity/pruning of neural networks in particular from the viewpoint of diverse stakeholders in the research community as related to the status of published research, empirical rigor and reporting results and some technical issues related to efficient deployment.

Contents

Chapter 2 contd.

Chapter 3

Some results reported in the sparsity literature

Chapter 6 contd.

List of Tables

List of Figures

Acknowledgements

I thank my supervisor

Professor Nikitas J. Dimopoulos

For accepting me as a MEng student

For his consideration about my general wellbeing throughout my studies at the department

For his immense patience while guiding me through the research related to the MEng project and writing of the report

For his support, kind advice and guidance in all aspects of my life at the Department of Electrical & Computer Engineering including finding work, getting financial support and other academic related activities.

Further, I thank

Professor Mihai Sima for being in the supervisory committee and going through my report at very short notice and for some insightful questions

And

Professor Xiaodai Dong for being kind enough to chair my oral examination at very short notice.

I would also thank all professors in our department who taught me numerous courses and interacted with me in other capacities on various occasions during my study at the department.

Further I thank the staff of the Department of Electrical & Computer Engineering, Engineering co-op office, Financial Aid Office, Faculty of Graduate Studies and the Library for their kind support which made my studies successful at UVic.

I also thank the Professors and the staff at the Department of Computer Science who employed me on numerous occasions (as a TA and as a sessional lecturer) and the Government of BC for employing me in three co-op work-terms which provided me with financial support and work experience at their respective establishments.

I thank the First Nations on whose territories our university stand and the Government of BC for letting me live and study in this beautiful province of British Columbia.

Dedication

I dedicate this work to the remembrance of my late loving parents, Chula Subadra Balasuriya Jayasekara (mother) and Bakthiudayar Isaac Gnanaratnam (father) who instilled in me an appreciation of education and knowledge

And

To the children of the machine-learning age, including my own (Nilmini and Nuwan) who aspire to build artificial systems that try to mimic human beings.

Chapter 1

1.0 Introduction

Deep neural networks (DNNs) using larger neural network models perform more effectively on many artificial intelligence (AI) tasks and are now widely employed in many AI applications such as computer vision, speech recognition, and robotics. Other than academics at universities, numerous firms, such as Google, Microsoft, and Nokia, are actively researching them since they can deliver significant results in many classification and regression problems for available datasets [1]. However, scaling up these DNNs to deliver improved accuracy comes at the expense of additional processing and high computational complexity, which results in high energy consumption. This not only raises data centre operating costs, but also makes it difficult to deploy DNNs on mobile devices and other edge devices such as IoTs with limited energy budgets [2]. For example the large natural language processing (NLP) model GPT-2 has 1.5 billion parameters and takes roughly a week to train on 32 TPUv3 chips and costs over $40,000 on Google's cloud TPU platform [2]. This is only for training a selected model. For research and development in order to select a good model the costs are thousands times more.

Pruning neural network parameters has emerged as a useful strategy for reducing model sizes and compute requirements. There are other approaches for model compression besides sparsification/pruning. Other ways to compressing models to increase computational and memory efficiency, as discussed in [3, page 3], include downsizing models, operator factorization, value quantization, value compression, and parameter sharing.

Despite their well-known empirical success, modern DNNs with millions of parameters require excessive amounts of memory and computational resources to store and conduct inference. A common practice to obtain a small, efficient network architecture is to train an over-parameterized network, prune it by removing certain parts, and re-train the pruned network so that accuracy loss is minimal relative to that of the original network.

In this report we are going to explore

What is network pruning?

How it is done?

Consider the effect of pruning on the DNN and related metrics as given in some sample publications

Some problems identified in the way experiments are conducted and the way results reported in pruning literature

Some technical issues associated with the efficient deployment of pruned networks

In the next section a brief study of pruning DNNs will be undertaken to introduce the related concepts.

1.1 What is sparsification and pruning as related to DNNs?

In this section we hope to give a brief introduction to sparsification and pruning DNNs and associated concepts.

According to the author of [4] who provides an overview of early sparsification techniques until 1993, pruning is about training a network that is larger than necessary and then removing parts that are not needed so that one needs lesser resources to store it and less compute to execute the trained network for inference. In those days researchers observed that neural networks converge easily while training if the network is large and used it as an experimental heuristic. The published literature on ‘pruning’ show countless ways of identifying the aforementioned useless parts or removing them before, during or after training; it even turns out that not all kinds of pruning actually allow for accelerating neural networks, which is supposed to be the whole point of pruning.

When thinking about the cost of neural networks, the count of parameters the most widely used metric, along with FLOPs (floating-point operations per second) which is a measure of the computer’s performance being the other.

In the context of deep neural networks (DNNs) with millions of parameters sparsification means still the same as it was defined in [4] as mentioned in the previous paragraph and is about getting rid of some parts of the network (for example, connections, neurons, filters and transformer heads) by pruning the specific part of the network. Here pruning means making the value of the respective element zero. So, connections could be set to zero by setting to zero the specific element in the weight matrix, etc. using a specific algorithm known as the pruning algorithm. The published literature on pruning explains multiple schemes which will suit multiple objectives. In this section we will consider these different schemes used in pruning.

By removing nonessential elements which will have no or minimum impact on the accuracy of the network; sparsification by pruning reduces the memory space required to store the model as well as reduce the operations to evaluate the model. However, different ways of sparsification lead to different overheads as stated in [3]. For example, in a situation when whole neurons or filters are removed (structured pruning) associativity and distributivity of linear algebra could be used to transform a sparsified structure into a smaller dense structure. But, if random elements of a weight matrix are removed (unstructured weight pruning), then it is necessary to store the indices of the remaining non-zero elements that will need more memory and might not also reduce the operations to execute the model depending on the hardware [3].

Also, pruning parameters does not necessarily reduce the computation time if the majority of the parameters removed are from the fully connected (FC) layers where the computation cost is low. For example, FC layer in VGG-16 occupy 90% of parameters but less than 1% of the computation [5]. Other than that, reducing FLOPs does not necessarily reduce energy cost. Due to the memory bottleneck in normal computing systems 1 access to memory actually is ~1000 more energy consuming than the ADD operation. 16 FP (floating point) Mult (multiplication) takes 1/4 of energy of 32 FP Mult [6]. So, pruning a network will not always help in reducing both the memory utilization and compute.

The first question that should be asked is what can be pruned to reduce cost; which is addressed in the next section.

1.1.1 What could be pruned?

By looking at the recent pruning literature, [3] has identified the following categories for model pruning in terms of elements:

Weights unstructured/fine grained

Weights structured/blocked i.e. in blocks or when you remove rows of weights or columns of weights

Other elements that come under the category of structured pruning are; neurons and neuron like (such as filters, channels and heads) [3].

An example of unstructured weight pruning is explained in [6] where the researchers were able to reduce the number of parameters of AlexNet by a factor of 9×, from 61 million to 6.7 million on the ImageNet dataset, without any accuracy loss and similar experiments with VGG-16 found that the total number of parameters can be reduced by 13×, from 138 million to 10.3 million, again with no loss of accuracy. Their [6] pruning method had a three-step process, which begins by learning the connectivity via normal network training. Unlike under conventional training, the network is not learning the final values of the weights, but rather learning which connections are important. The second step is to prune the low-weight connections. All connections with weights below a threshold are removed from the network — converting a dense network into a sparse network. The final step retrains the network to learn the final weights for the remaining sparse connections. This step was critical. If the pruned network is used without retraining, accuracy is significantly impacted [6].

Yet, unstructured pruning presents a major drawback: most software frameworks and hardware cannot accelerate sparse matrices' computation. Due, to this reason most published pruning papers deal with structured pruning.

Structured pruning could be used to exploit hardware and software optimized for dense computation and is more popular than unstructured pruning. For example, [5] prune entire filters from convolution neural networks and reduced inference costs for VGG-16 by up to 34% and ResNet-110 by up to 38% on CIFAR10 while regaining close to the original accuracy by retraining the pruned networks. The paper [5] proposed to

prune multiple filters at once and retrain once. (as opposed to conventional pruning of one filter at a time and retrain after pruning each filter).An example of channel pruning was considered in [7] where on various image classification datasets for VGGNet, a multi-pass version of their pruned network gave a 20x reduction in model size and a 5x reduction in computing operations. Both models [5, 7] required no special software/hardware accelerators for execution while unstructured pruning leads to irregular sparsity in pruned network, and requires sparse conv libraries and special hardware.

All these element wise model pruning methods effects the forward pass while training or during inference according to [3].

After identifying the elements to prune the next question is what criteria should be used to select these elements which will be handled in the next section.

1.1.2 Scoring or pruning criteria

Next, when one elects to remove elements say connections, filters etc. the question is what criterion should be used to select the specific item.

A criterion that is popularly used is the magnitude of weights which means that one prunes the weights with smallest absolute magnitudes. For example, [6] uses pruning weights with smallest values and is based on unstructured pruning.

The L1-norm of a square matrix is the maximum of the absolute column sums and the L2-norm of a square matrix is the square root of the sum of all the squares of the elements. Therefore, for larger filter weights values both norms will be larger and hence does not matter which norm is used. In the case of structured pruning, one direct way is to order filters depending on their filter weight norm (L 1 or L 2 for example) and zero-out filters with small L1 or L2-norms. The paper [5] used both methods (used both L1 and L2 norms) when pruning filters in their experiments and did not see any significant difference between the two. They concluded that as the important filters tend to have large values for both measures it does not matter which method is used.

In the case of pruning whole channels, without having to compute the combined norm of the relevant parameters, it could be done by inserting a gate which is a learnable multiplicative parameter for each feature map after each set of layers you want to prune. This gate, when reduced to zero, effectively prunes the whole set of parameters responsible for this channel and the magnitude of this gate accounts for the importance of all of them [7]. The method hence consists in pruning the gates of lesser magnitude and applied for example in [7] for channel pruning.

One other pruning criterion used is finding some metrics, derived from the back-propagated gradient. Back in the early 90's a fundamental work in neural network pruning [8] derived a theory based on a Taylor decomposition of the impact of removing a parameter on the loss function, that some metrics, derived from the back-propagated gradient, may provide a good way to determine which parameters could be pruned without damaging the network.

Finally it is required to decide whether the chosen criterion is applied globally to all parameters or filters of the network, or is it computed independently for each layer. Some pruning methods compare scores locally, pruning a fraction of the parameters with the lowest scores within each structural subcomponent of the network (e.g., layers) while others consider scores globally, comparing scores to one another irrespective of the part of the network in which the parameter resides. Even though global pruning has confirmed many times to yield better results sometimes it leads to layer collapse [9, 10].

We discussed the types of elements that could be pruned in Section 1.1.1 and the pruning criteria in this section in the next section we will consider the pruning method.

1.1.3 How to prune parts without harming the network?

In this section we discuss pruning methods or schedules.

Pruning schedules vary in the amount of the network to prune at each step. Some methods used in the literature prune all desired weights at once in a single step. Others prune a fixed fraction of the network iteratively over

several steps or vary the rate of pruning according to a more complex function [10].

For some pruning schedules that involve fine-tuning, it is most common to continue to train the network using the trained weights from before pruning. An alternative scheme include rewinding the network to an earlier state or reinitializing the network entirely [10].

All three example papers [5, 6, 7] mentioned in Section 1.1 used a train, prune and fine tune method as the pruning schedule, which could be thought of as the classic pruning method.

Other than the above there are many other pruning methods discussed in the literature. A brief introduction to these other methods (not discussed in this report) with example papers cited is given in the article [9].

1.1.4 Available frameworks for pruning

Pytorch and TensorFlow with the Keras API provide some basic support for pruning. Other than that Blalock et al. [10] provide in their work an open-source library ShrinkBench in an effort to help the research community normalize how pruning algorithms are compared. This library is based on Pytorch.

ShrinkBench provides standardized and extensible functionality for training, pruning, fine-tuning, computing metrics, and plotting, all using a standardized set of pretrained models and datasets [10].

In Section 1.1 we gave a brief introduction to pruning as related to DNNs and explored some related concepts.

For further reference the interested reader is referred to the article [9] that summarizes some recent published ‘pruning’ papers categorized according to the structure, criterion and the method used for pruning. Other than that [3] provides an extensive discussion about pruning methods in the form of a ‘sparsity tutorial’ presented at a virtual seminar [11].

1.2 Structure of the report

According to Dr. Hoefler & team [3] at least one paper about pruning is being published every couple of days on the arXiv.org. This means a researcher doing a literature review will have to check a large volume of papers to understand the current situation in research.

In a situation like this the authors who wrote the paper 'Distilling Information from a Flood' [12] for instance, have pointed out the importance of use of methods such as meta-analysis and systematic review. Application of such methods could address the challenge of extracting information from research publications when the volume is very large and growing at a rapid rate as in the case of DNNs. In accordance with this recommendation, we selected two papers for further study about research on pruning neural networks; one doing a most recent through systematic review of the field [3] to get an understating of the status of research in pruning DNNs and the other based on a meta-analysis study [10] that identify problems in how pruning experiments are conducted and results reported back to the community with some remediation suggestions.

The report explores the following topics in the given order

In Chapter 2 we will consider the status of research in sparsity/pruning of neural networks as explained in the systematic survey [3].

In Chapter 3 we will consider ten randomly selected papers from a Google Scholar search on filter pruning using VGG16 on CIFAR-10 dataset with their respective metrics for the obtained maximum accuracy extracted from the respective paper to get an understanding of type of the statistics reported.

In Chapter 4 we will describe an experimental result on weight pruning obtained by replicating the experiment using an ipython notebook available from Google research on Colab.

Chapter 5 will be based mostly on the meta-analysis report [10] and consider the current status on the way experiments are performed in pruning neural networks, how they are reported back to the wider community and how to improve the result reporting process.

In Chapter 6 we will consider some technical issues related to efficient deployment of DNNs at the edge in two parts.

In the first part we will consider technical issues related only to pruning

Some recent research found that there are some technical issues associated with pruning networks which recommend that caution be employed when deploying thus pruned networks on edge devices like smart phones.

In the second part we will consider some other common technical issues related to deep neural networks (DNNs)

As reported in the published literature we know that there are some common technical issues in existing deep neural networks (DNNs), associated training datasets and computing systems in which they are implemented. We tried to find out whether pruning networks affect these issues and if yes, how?

The following technical issues will be considered:

- Brittleness (serious when deploying critical DNN applications)
- Issues related to datasets and models

The above technical issues and whether or how pruning networks affect these issues will be discussed in Chapter 5.

Finally Chapter 7 concludes the report.

Chapter 2

The status of sparsity research

2.0 The status of research in sparsity/pruning of neural networks

This topic was explored by referring to a recent literature survey paper on sparsity/pruning [3] carried out at a university about the overall status of research publications in sparsity/pruning and an attempt to classify them according to main characteristics related to pruning and to identify trends in publications. The survey paper is ‘Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks’ [3], a study done at ETH Zurich by Dr. Torsten Hoefler a professor in Computer Science and his team and was published in January 2021.

According to [3], the paper reference [9] provides an overview of early sparsification techniques until 1993 and since then, the literature has evolved significantly.

As mentioned in [3] a second “AI winter” in the late 1980s and early 1990s appears to have significantly reduced the interest in and funding for artificial intelligence research and development and due to this state of affairs activity in neural networks consequently waned for nearly two decades. Deep learning (re-)started its success story around 2012 with convolutional neural networks for image recognition [3]. Since then, more than 266 papers, comprising 4,089 pages focusing on ideas and techniques for sparsity in deep networks appeared, which the authors of [3] attempted to categorize and summarize in their report.

Later a sparsity workshop [14] was held online in September 2021 at ICML 2021 and Dr. Hoefler presented a comprehensive pruning tutorial based on this report at that workshop. The study [3] consists of 429 references of length 124 pages and summarized more than 300 research papers. The main findings of this report are that

- papers published on the topic related to ‘sparsity/ pruning in deep neural networks’ are growing at an exponential rate
- overall number of publications are growing at a rate of one paper published every couple of days on the arXiv.org
- many of the papers reinventing the already published techniques in early 1980s
- many incomparable results that makes it hard to determine the state of the art and whether method A is better than method B
- nearly every basic approach has been invented at least twice

This chapter will be mostly about the sparsity tutorial and some trends in sparse literature as explained in [3].

2.1 A tutorial on sparsity

We did a brief introduction to what is pruning and associated main concepts in Chapter 1. This section will help develop those ideas further towards an extended sparsity tutorial. As we will see in the following sections of this report, authors of [3] employed a different way to categorize the sparsity literature that they scrutinized than is normally employed by other practitioners in the field. Yet, sometimes there might be a slight repetition with the content of Chapter 1 and the content of Chapter 2.

This section attempts to make a summery of the sparsity tutorial introduced in the paper [3] which tries to distill ideas from more than 300 research papers published over four decades. It will cover all the general pruning approaches—from classic ones such as magnitude pruning, to second-order approaches, and regularization and variational approaches. The authors of [3] found that there has been extremely important early work on sparsity, in e.g. the 90s, which introduced several fundamental concepts and techniques. Yet, many of these concepts have been re-invented in recent years. The tutorial attempts to provide a holistic view of all these results, integrating both the early work, as well as recent advances.

The context for this tutorial was the growing energy and performance costs of deep learning models and the need to reduce such costs by reducing the size and compute of DNNs by pruning. The main presenter of the paper [3] Dr. Torsten Hoefler is also the head of the HPC center at ETH, Zurich and

as stated by him; his main interest in their exercise as regards to [3] was the size of modem DNN workloads and the possibility of reducing them in size and flops to run on their servers.

According to the authors of [3], the tutorial is targeted at academics, researchers in industry, and engineers involved in software and hardware design and will be accessible to a wide audience.

The topics covered in the tutorial are addressed in the rest of this section.

2.1.1 What can be pruned?

The first question is which elements of a neural network can be removed.

The authors of [3] try to summarize literature into two main categories, that which propose model or structural sparsification, e.g., zeroing weights, and ephemeral sparsification, e.g. sparsification of gradients and activations. The categorization of elements which can be pruned according to this scheme [3] are as follows:

2.1.1.1 Model/structural sparsification

Model sparsification changes the model but does not change the sparsity pattern across multiple inference or forward passes [3]. In this scheme the two main elements, weights and neurons can be sparsified. Elements in specialized layers, such as filters in convolutional layers or heads in attention layers are similar to neurons in the context of pruning and can be removed as well [3]. Neuron, filter, and head sparsification reduces simple parameters of the model, and can shrink it substantially, and results in a new model that is essentially dense (i.e., can efficiently be executed on the same hardware as the original model) [3].

If arbitrary weights are sparsified, the resulting model may be unstructured and the need to remember indices of none zero elements is required. This adds overheads and leads to less efficient execution on hardware that is optimized for dense matrix computations. Due to this, approaches for structured weight sparsification or pruning of other structures like neurons, etc. have been developed to reduce indexing overheads and improve

efficiency at execution. These approaches typically store contiguous blocks of the weights instead of single elements [3]. The methods mentioned under this category are summarized below:

Model/structural sparsification (which is applied per model)

1. weights
2. neurons
3. Neuron-like (Filters/Channels/Heads) depending on the network

From above weights could be either unstructured (fine-grained) or structured (taken as a block or coarse-grained). All others mentioned, neurons and neurons like elements comes under the structured category in the context of pruning.

All of above mentioned components affects inference and the forward pass through the network.

2.1.1.2 Ephemeral sparsification

This is the second class of elements for sparsification mentioned in [3] and applied during the calculation of each example individually and only relevant for that example.

According to this viewpoint the first set as mentioned in [3]; the most obvious structural sparsification applies to activations, for example the ReLU and SoftMax operators lead to a natural sparsification. Both set values to zero by a fixed threshold (rounding in case of SoftMax). One can also consider random activation sparsity as in dropout.

A second set of ephemeral sparsity elements are related to the gradient-based training values [3]. The back-propagation phase of SGD uses errors and gradients to update the weights. Both can be sparsified to only update weights partially in the forward pass and lead to significant performance improvements, especially in distributed settings [3]. An option explored in the literature is to delay the communication/update of small local gradient contributions until they are significant [3].

A third set of ephemeral techniques are listed under conditional computation, where the model dynamically decides a sparse computation path for each example [3].

These methods related to ephemeral sparsity are listed in point form below:

Ephemeral sparsification (which is applied per example)

1. Dropout (Activations/Weights)
2. Related to gradient-based optimization
    - Gradients
    - Errors
    - Optimizer state
3. Activations (e.g., ReLU)
4. Conditional computation (route each example through a different sparse subnetwork)

From above mentioned elements for ephemeral sparsification, items under 1 and 2 affects only training while elements under 3 and 4 affects inference and forward pass.

2.1.2 What, when, and how to sparsify / remove elements in model sparsification?

The main operation in any sparsification scheme is to select candidate elements to be removed and the most intuitive and most precise data-driven way to select elements for removal is to evaluate the network with and without the elements in question [3]. However, this simple leave-some-out approach to just train the network with and without the neurons or weights removed poses scalability issues as it needs to train networks with n elements in total with k removal candidates and when the value of n is very large as in the case of modern DNNs.

Another simple method is to select elements to be removed at random, which can be quite effective in some settings as mentioned in [3].

However, guiding the removal by some metric of importance has given the best performance to achieve compressed models with high sparsity in practice [3]. In this section, an overview of such selection methods as given in [3] are considered.

Since, comparative studies in pruning up to now have not identified a best method for element removal, authors of [3] only provide an overview of the known methods. They [3] will only provide an understanding of the intuition behind each method without quantifying the efficiency of each method, because that depends on the exact setting of network architecture, hyperparameters, learning rate schedule, learning task etc., and due to this different works can hardly be compared.

2.1.2.1 Structured vs. unstructured element pruning

The section 2.1.1.1 discussed about the disadvantages of fine-grained or unstructured weight removal. Structured sparsity constrains sparsity patterns in the weights and promises a better performance and lower storage overheads but it may lead to worse models because it may limit the degrees of freedom in the sparsification process [3].

One simple example of structured sparsity is the removal of whole neurons in a fully-connected layer: the resulting computations for the forward or backward pass after removing a neuron are simple dense matrix multiplications from which a whole row/column was removed (weights of all incoming and outgoing connections). A similar argument applies to the removal of convolutional filters and transformer heads [3].

Structured pruning often uses similar methods to unstructured pruning, sometimes with minor modifications to prune whole sets of weights [3].

2.1.2.2 Data-free selection methods

One of the simplest but an effective method is removing weights with the smallest absolute magnitude. It is often used together with re-training the sparsified network and training schedules where the sparsity is gradually increased over time [3]. It can be applied to either individual weights or

arbitrary groups of weights using the sum of absolute values in rows/columns in the weight matrix for structured pruning (e.g., blocks or rows/columns for whole neuron pruning) [3]. Also, since the weight values usually follow a normal distribution with a zero mean, pruning by magnitude can remove the bulk of the weights around zero [3]. Different methods are used in the literature to define the threshold below which to prune according to [3]. Magnitude pruning is often used in literature during sparse training schedules to maintain an approximately constant connection density during training [3]. Some papers popularized magnitude pruning for modern deep neural networks as part of neural network compression for inference, while others prune whole filters with the smallest sum of absolute weights in convolutional layers. Several works also used magnitude pruning to prune recurrent neural networks as well as sparse training [3].

There are other data-free methods that does not consider training examples. Some of them base pruning decisions on the structure of the network. Since these methods do not depend on examples, they can be used as a pre- or post-processing step for data-driven methods [3].

A simple scheme compares sets of weights between different neurons and compare the input weights between all neurons which will enable merging of k similar neurons into a single neuron, multiply all weights by k, and add all biases, a method which works well for small networks but prunes less for large networks [3].

While data-free methods, especially magnitude pruning, are often very effective and can provide state-of-the-art results, several works have shown that more precise methods can achieve significantly better results, especially at high sparsity. Furthermore, data-free schemes often require expensive retraining to recover an accuracy as close to the original performance as possible. An obvious way to improve precision is to consider the influence of the training data (and implicitly its distribution) in the pruning method selection [3]. These methods are discussed in the next section.

2.1.2.3 Data-driven selection methods (for inference only)

The first set of methods considered in this section are based on input or output sensitivity and further categorized as input sensitivity and Fourier sensitivity.

These methods consider the statistical sensitivity of the output of neurons or the whole network with respect to the training data. In such methods, a set of examples (potentially all of the training data) is used to determine directly which elements should be removed to maintain or improve prediction accuracy while sparsifying the network. Elements with very small or zero change with respect to deviation of the input examples contribute less in the entire network since their outputs are approximately constant to the variation in their inputs. Thus, such a sensitivity measure can be employed to define the relevance of an element for the function of a network and low-relevance elements can be removed [3].

Such methods can also be applied to filters in convolutional networks. In certain works they define the filter pruning problem in terms of its output sensitivity to the following layer and then prune filters that, across the whole minibatch, change the output of a layer least [3].

In some other works, to find the best number of neurons for the model, they prune neurons based on their output variance across samples from the input distribution. They use Fast Fourier transforms to determine the change in output for inputs that vary within the input distribution. They add neurons and improve model capacity if the mean-square training error exceeds a bound. One benefit of this method is that it suffices to have upper and lower bounds on the features to roughly approximate the input distribution—detangling the selection process from the data [3].

Many of the above mentioned methods can be applied to any neuron in any layer. However, some study the “feature selection problem” to prune input neurons (“features”) [3].

The second set of data-driven selection methods are based on activity and correlation. This is based on one simple observation; which is that, in many networks, some neurons are often activated together, relating to the Hebbian observation “neurons that fire together wire together”. Several sparsification schemes are based on this observation. A simple

sparsification scheme could merge neurons that have very similar output activations and simply adapt their biases and rewire the network accordingly [3].

In some other works, they observed that some neurons are producing very similar outputs for all examples during inference. They identify such pairs of similar-output neurons across the training examples and removed redundant ones [3].

In certain other methods used in the literature, they would strengthen connections between correlated neurons and preferentially drop weights between weakly correlated neurons and maintain connections between strongly correlated neurons. While data-driven sensitivity-based schemes consider the outputs across the examples drawn from the input distribution, they purely aim at minimizing the impact on the input-output behavior of the network. Thus, if the network has a low accuracy, it will not gain from such pruning methods [3].

In the next section, the training loss function itself is considered in the pruning process and used to improve the model accuracy of the pruned network as much as possible.

2.1.2.4 Training-aware (i.e. based on full training) pruning methods

Under this category of pruning methods the first set is based on loss function approximation (1st order, 2nd order) using the Taylor expansion.

Selection based on 1st order Taylor expansion of the training loss function:

Gradient-based first order methods are most successful for learning weights in deep neural networks. Therefore, it is not surprising to apply similar methods to the selection of weights. Since gradients of the weights are computed during the normal optimization process, one can easily re-use those for determining weight importance. Furthermore, gradient computations are generally cheap, so one could employ them together with additional, so called gating elements to select arbitrary elements (weights, neurons, filters, etc.) for removal [3].

Selection based on 2nd order Taylor expansion of the training loss function:

The question of selecting the “least significant” set of weights to remove from a fully-trained model relative to the difference in loss with respect to the current model was considered in some seminal works. These references consider an “optimization” approach to pruning, trying to answer the question of which parameter to remove in order to minimize the corresponding loss increase, under the assumption that the second-order Taylor approximation of the loss around the dense model is exact [3].

The second set is based on regularization (L0, L1, L2).

In these methods based on the technique of regularization, a penalty terms are added to the cost function, for example, L' (x, w) = L(x) + P (w). Here, L(x) is the original loss function and P (w) is a penalty term defined on the weights. Penalty functions can be defined with respect to arbitrary elements in the network (e.g., gating terms for neurons). The penalty will guide the search function to the desired output (e.g., sparse weights) and reduce the complexity of the model. For example, in the case of gating terms for neurons will lead to a sparse, smaller, and potentially faster mode [3]l.

Some disadvantage in this method is that the penalty terms can introduce additional local minima which makes the optimization landscape harder to navigate. Furthermore, tuning the regularization parameters often requires a delicate balancing between the normal error term and the regularization term to guide the optimization process. Even more, regularization may require fine-tuning per layer. Yet, well-tuned regularization terms are essential to deep learning training and sparsification [3].

One of the first penalty terms that was shown to significantly improve generalization was weight decay, where the weight update rule adds a reduction in absolute magnitude: w' = (1 − λ) − αg, with the decay factor λ and the learning rate α. Weight decay is a standard technique for improving generalization today and it can be combined with magnitude pruning for sparsification [3].

Among modern approaches, multiple methods rely on the LASSO (Least Absolute Shrinkage and Selection Operator) to prune weights or groups. Some other methods develop penalties that target weak connections to increase the gap between the parameters to keep and those to prune, so

that their removal has less impact. Some methods show that targeting a subset of weights with a penalization that grows all throughout training can progressively prune them and make their removal seamless [3].

The third set are known as variational selection methods where selecting candidate elements to be removed from the network rely on a Bayesian approach. Here, one can assume a distribution across the elements of a neural network (e.g., over individual weights or neurons), and prune elements based on their variance. The intuition behind this approach is that elements with high variance would have little contribution to the final network performance, and therefore it might be beneficial to remove them. The literature also counts a whole range of methods built around the principle of “Variational Dropout”, a method based on variational inference applied to deep learning. As a pruning method, it was responsible for multiple works that adapt its principle to structured pruning [3].

The methods discussed in the Section 2.1.2 could be summarized as follows:

The simplest scheme to remove elements: leave k elements out – train the remaining part of the model to convergence. Various selection schemes by some importance metric to select the appropriate k elements are given below [3]:

1. Data-free (no model evaluation). Prune
    - neuron-/weight which are similar such that combined effect trivial
    - weight magnitudes approximately equal or zero

2. Data-driven (inference-only)
    - remove trivial elements
    - remove depending on sensitivity
        1. Input sensitivity (do outputs change across examples?)
        2. Fourier sensitivity (which weights do not influence outputs?)
    - Remove correlation / similarity merge based elements
        1. Hebbian (strengthen weights between correlated neurons)

2. Similarity (outputs are all similar?)

3. Training-aware (full training). Remove elements by applying following methods:

- loss function approximation (1st order, 2nd order)
- regularization (L0, L1, L2)
- statistical / variational

2.1.3 When to sparsify? Sparsification schedules for model sparsification

While ephemeral sparsity is dynamically updated for each example and configured with a small number of parameters during inference and training, model sparsity follows a more complex procedure and often trained with a schedule [3]. The authors of [3] differentiate three different classes of training schedules as given in the literature:

1. First train and then sparsify
2. Sparsify during training (including iterative sparsification)
3. Sparse training from the beginning (including regrowth)

Each of those schedules could be used iteratively in an outer train-sparsify loop.

2.1.3.1 Sparsify after training

The train-then-sparsify is the most common schedule type and uses a standard dense training procedure that is run to convergence in T iterations followed by a sparsification of the fully trained model. Beginning from the earliest works, the model is typically re-trained (“fine tuned”) after the sparsification to reach significantly higher accuracy. This schedule type aims at improving performance and/or generalization during inference. It provides the best baseline for model quality because one can always compare the sparsified model quality with the original dense model.

Furthermore, since one starts from a dense model, training does not change such that existing hyperparameter settings and learning schedules can be re-used. Some early works even show that pruning before the model has converged can reduce the final accuracy [3].

2.1.3.2 Sparsify during training

This schedule starts sparsification of the model before it has been trained to convergence and is usually cheaper than the train-then-sparsify schedule. Furthermore, training a dense model to convergence may allow for overfitting that is hard to correct with pruning alone. Schedules that gradually sparsify during training may follow a pruning schedule that also corrects for approximation errors due to premature pruning in early iterations. Such schemes often train the dense model for some iterations before sparsification starts and end with a sparse trained model. In general, sparsifying during training already reaps potential performance benefit of sparsity early on but could lead to less efficient convergence and is often more brittle to configure via hyperparameters. Furthermore, this approach needs to hold the dense model in memory at the beginning of the operation and thus does not enable the use of smaller-capacity devices.

2.1.3.3 Sparse training

The fully-sparse training schedule starts with a sparse model and trains in the sparse regime by removing and adding elements during the training process. Some works showed early that this scheme can even outperform separate growing or pruning approaches for neuron-sparse training of simple MLPs. Weight-sparse training often uses complex hyperparameter settings and schedules. However, it enables to train very high-dimensional models whose dense representations would simply not fit into the training devices [3].

According to [3] the literature could be differentiated between static and dynamic sparsity during sparse training. Dynamic sparsity combines pruning and regrowth of elements during the training process, while static

sparsity prunes once before the training starts and does not update the model structure during training [3].

2.1.3.4 Some other methods used in the train/sparsify schedules

Ephemeral sparsity during training

Most efficient training methods would take advantage of both ephemeral and model sparsity during training. In an empirical study, researchers observed that training is less robust with respect to sparsifying activations in the forward pass and gradients in the backward pass. Based on those findings, they design a method that eliminates small weights during the forward pass and both small weights and activations during the backward pass using a simple top-k method [3].

Sparsify for transfer learning and fine tuning

In transfer learning, large pre-trained networks are specialized to an often narrower task than the original broad training goal. This specialization might require pruning and potentially parameters can be pruned during the process. The schedule for such pruning during fine-tuning is similar to the train and prune schedule: a model is trained to convergence and then pruned. However, the difference is in the training dataset and corresponding distribution. The dataset used for fine-tuning is different from the original dataset—often it corresponds to a specific subset, but sometimes it could represent a distributional shift. So in some sense, the pre-trained network can be seen as a more intelligent (non-random) weight initialization as basis for a shorter learning process. Also, data sets for fine-tuning are often much smaller [3].

2.1.3.5 General Sparse Deep Learning Schedules

The sparse training process can be described as a series of steps, each can be skipped and some steps can be iterated multiple times [3].

Step (1) initializes the network structure, this can either load a description of the network structure from disk or be built using a framework as is

usually done for dense networks. However, it could also generate a random network structure or use a sparse network construction strategy

Step (2) initializes the weights of the network, typically randomly or in transfer learning settings with pre-trained weights. For sparse networks, one could use specialized initialization strategies such as synaptic flow. The different weight values could be either positive, negative or zero.

Step (3) trains the network for a defined number of iterations or until convergence. This training can be done with an unmodified dense training schedule or with a sparsity-inducing schedule (e.g., regularization). This initial training may be run until convergence or stop early for iterative methods.

Step (4) prunes and regrows various elements using the different techniques explained in the Section 2.1.2.

Step (5) may retrain the network either for a fixed number of iterations or to convergence (this step is relatively often skipped but generally improves model accuracy).

Steps (6) and (7) indicate possible loops in the training process. Step (6) is often used in iterative training/sparsification schedules to achieve highest quality. Step (7) could be used to reset weight values, which is sometimes done [3].

A summary of the section 2.1.3 is given below:

There are three main schedules mentioned in literature as given below in [3]:

1. First train and then sparsify
2. Sparsify during training (including iterative sparsification)
3. Sparse training from the beginning (including regrowth)

All above mentioned three methods fit a generic schedule as given below (some stages can be skipped) [3]:

a. initialize structure

b. (re)initialize weights

c. Train the original model

d. prune / regrow

e. retrain the pruned model, if done stop

f. else, iterate between e and d until finish pruning then stop

g. or if you want to reset/rewind then start from b again and continue the process up to e and stop.

2.2 About categorization of literature on sparsity

According to [3] sparsity has intrigued the research community due to the fact that it has lot of open-ended opportunities. They state “We see sparsity as potentially achieving a second significant “jump” in computational capability as, even with current methods, it has the promise to increase computational and storage efficiency by up to two orders of magnitude” [3].

2.2.1 Some information about the popularity of various Pruning techniques

1. Authors of [3] have attempted to categorize the surveyed publications from 1988 to 2020 to following categories of their own making using concepts introduced in the sparsity tutorial discussed in Section 2.1:
   model sparsification for inference
   model sparsification for training
   ephemeral sparsification
   hardware acceleration for sparsity
   software acceleration for sparsity
   According to the above categorization for example, in the year 2020 most papers published were related to model sparsification for inference followed by model sparsification for training. The next most popular theme was hardware acceleration for sparsity [3].

2. Authors of [3] also classified the surveyed papers from 1988 to 2020. in to three different categories: (1) the candidate element to be removed, (2) the method chosen for elements removal, and (3) whether the authors discuss optimizing inference or improving training (type). The different candidate elements considered are neurons, weights, convolutional filters, transformer heads, transformer hidden dimensions, and inputs. The different methods used for pruning are Regularization, Magnitude, Output Sensitivity, First-order, Second-order, Variational, Random, Activation, Leave-one-out and Correlation (as categorized in the report [3])

    - Out of the total number of 159 papers considered 81 were on weight pruning which means that nearly 50% of all papers focused on weight sparsification, and closely followed by neuron sparsification. Also, out of these 81 papers on weight pruning the majority 39 were on magnitude pruning [3].
    - Out of the total number of papers 153 that were considered 93 were on pruning for inference while the balance 60 were pruning related to training. Which means that more than 60% of the papers focused on inference while training was the focus of the remaining papers. Most inference works focused on pruning either neurons, weights, or filters while pruning to improve training largely focused on weights [3].
    - Out of 93 papers on inference 26 used regularization and 19 as magnitude respectively as the pruning criteria. On the other hand out of 60 papers on training 31 used the magnitude as the pruning criteria [3].
    - Overall 50% of the surveyed papers focused on either magnitude pruning or regularization. Further, magnitude pruning is most often used for weights while regularization is equally applied to all other element types such as filters, blocks, and heads all included into a single “structured” category in the report [3].
    - All three classification dimensions illustrated that pruning weights by magnitude was the most popular method addressed in the literature, followed by sensitivity-based neuron pruning [3].

2.3 About the general understanding on pruning networks and how DNNs work

Finally, after reviewing more than 300 papers on pruning in the ‘Authors discussion’ of [3] the researchers state that ‘We do not understand all details of the inner workings of deep neural networks or how pruning influences deep neural networks?

Specifically,

Why can networks be pruned?

What is the best pruning methodology?

Remain as open questions’ [3].

They claim due to above given reasons it is only possible to provide a set of hypotheses, intuitive explanations, etc. as to explain why you got a certain result or why a certain way of pruning leads to a better compressed model.

2.4 Best practices to follow in experiments

This section presents a set of recommendations on practical aspects of pruning identified by the authors of [3] based on the surveyed of literature in the field.

The first observation of [3] is that a flurry of simple approaches enables reaching moderate sparsity levels (e.g., 50–90%) at the same or even increased accuracy. It seems that any non-silly scheme achieves some sparsification and that there is an inherent robustness in the networks themselves. However, reaching higher sparsity levels (e.g., >95%) requires more elaborate pruning techniques where one might be reaching the limit of gradient-based optimization techniques for learning.

The authors of [3] recommends the following best practices in five categories that they think are effective and urge the community to follow when performing pruning in practice:

1. Pruning strategy.

In general, highest sparsity is achieved using regularization methods in combination with iterative pruning and growth schedules. These methods have high computational costs, sometimes causing a five-fold increase in training overheads. Regularization methods are relatively hard to control and require numerous hyperparameters. The simplest training method, magnitude pruning, is easiest to control for target sparsity and accuracy in many practical settings. In most training methods, it is important for the structure search to enable weights to regrow, especially in phase of early structure adaptation at the beginning of training [3].

2. Retraining/fine-tuning

If the objective of performing sparsity is to improve inference, then retraining/fine-tuning is an essential part of a sparsification schedule. Gradually pruned sparsification schedules perform best and it is most efficient to start each iteration from the most trained/last set of weights [3].

3. Structured pruning

Structured pruning seems to provide a great tradeoff between accuracy and performance on today's architectures. This is partly due to the fact that hardware and frameworks are tuned for dense blocked computations[3].

4. Distribution

The sparsity distribution across layers/operators needs to be considered carefully. For this, one could hand-tune the sparsity levels for each operator type and position in the network. For example, dense layers can often be pruned more than convolutional layers and the first layer in a convolutional network can hardly be pruned. A simpler scheme may use a global sparsity and a learned allocation strategy [3].

5. Combined ephemeral and model sparsity

Any sparse deep neural network should combine both ephemeral and model sparsity. For example, dropout often functions as a "pre-regularizer" and can benefit generalization greatly if enough data is available. Furthermore, ephemeral and model sparsity lead to a multiplicative benefit in terms of needed arithmetic operations [3].

2.5 Summary of Chapter 2

In the report [3], the authors attempted to categorize and summarize more than 266 papers, comprising 4,089 pages which focused on ideas and techniques for sparsity in deep networks which were mostly published from 2012 up to 2020.

The study was also presented as a sparsity tutorial at ICML 2021. This study [3] carried out at ETH, Zurich tried to show the diversity of pruning methods used by the community while pinpointing some weaknesses in existing publications. They also found the current trends where researchers mainly concentrate on weight pruning by magnitude as a model compression technique for inference.

The report [3] concludes by giving some guidelines (best practices) for performing experiments to achieve better results and further lists some open questions to inspire future research.

In this report a brief summary of the sparsity tutorial presented in [3] was attempted in Section 2.1 followed by giving some statistics on popularity of various pruning methods in Section 2.2.

Chapter 3

Some results reported in the sparsity literature

3.0 Introduction

To gain an understanding of the type of statistics reported in actual practise in this chapter we will check ten randomly selected research papers from a Google Scholar search on filter pruning that was trained using the model VGG16 on CIFAR-10 dataset.

We will tabulate the respective metrics related to pruning for the obtained maximum accuracy extracted from the respective paper and try to gain a general understanding of what is reported on filter pruning for the specific model on the given dataset.

One important fact to remind ourselves while we scrutinise these results is that research methodology on DNNs in general and pruning such networks in particular are based on experimental methods with hardly any theory to guide them as was discussed in Chapter 2 based on finding in [3]. Which means that generally the methods used to conduct research in these areas are based on heuristics. According to the Wikipedia article [13] the definition of heuristics is given as" A heuristic, or heuristic technique, is any approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect, or rational, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation" [13].

3.1The summary of ten papers on pruning

Evaluation Protocols used in the table column headings: We use widely-used protocols found in published literature, i.e., number of parameters and required Float Points Operations (denoted as FLOPs), to evaluate model size and computational requirement. To evaluate the task-specific

capabilities, we provide top-1 accuracy of pruned models and the pruning rate (denoted as PR) on CIFAR-10 for the model VGG16.

CIFAR-10 dataset is favored by many acceleration researches. It consists of 50k images for training and 10k for testing in 10 classes.

Explanation of table headings and contents:

Reference – name of the paper and the page from which the data was extracted

Ori E (%) - error in accuracy for the given model and the dataset before pruning as stated in the reference

Pr E (%) – error in accuracy of the pruned network

Δ – Difference in classification error (Pr E - Ori E)

PC – error in accuracy as a percent change (Pr E - Ori E)/ Ori E

PP (%) - percentage of parameters pruned

P – Number of parameters in the model (millions)

FR (%) – percentage in reduction of FLOPS (floating point operations per second)

F – Number of FLOPS (millions)

The best result if given highlighted in the table

N/A is not available

Filter pruning google scholar search results

Model: VGG16, Dataset: CIFAR-10

Table 1. The summary of randomly selected ten papers from a Google Scholar search

Reference	Ori E (%)	Pr E (%)	Δ	PC	PP (%)	P	FR (%)	F	comments
1. Table 1 on page 6 Li et al. (2016) VGG-16 base	6.75					15		313	Most papers compared with this paper. Prune

VGG-16-pruned-A VGG-16-pruned-A scratch-train (in Reference listed as 5)		**6.60** 6.88	**- 0.15**	**0.0222**	**64**	**5.4**	**34.2**	**206**	whole filters with the smallest sum of absolute weights in convolutional layers.
2. Table 1 on page 2785 Zhao et al. (2019) VGG-16 Base VGG-16 Pruned	6.75	 **6.82**	 **0.07**	 **0.0103**	 **73.34**	 14.71 **3.92**	 **39.1**	 313 **190**	Results reported without retraining after pruning
3. Table 2 on page 13 Huang, Qiangui, et al. (2018) VGG-16 Base for drop bound b = 0.5 b =1 b =2 b =4	7.23	 7.35 7.85 8.65 10.15	 **0.6** **1.1** **1.9** **3.4**	 0.0889 0.1629 0.2815 0.5037	 83.3 82.7 86.5 92.8	 15 N/A N/A N/A N/A	 45 55.2 64.5 80.6	 313 N/A N/A N/A N/A	compared with [1] for accuracy drop, GPU and CPU speedup
4. From Tables 3 and 4 on page 4346 He, Yang, et al. (2019) VGG-16 Base Pruned without FT With FT 40 epochs With FT 160 epochs Pruned From Scratch without SA	 6.42 6.42 6.42 6.42	 **19.62** **6.76** **6.0** 6.77	 13.2 0.34 -0.42 0.35	 2.0560 0.0529 0.0654 0.0545	 N/A N/A N/A N/A	 15 N/A N/A N/A N/A	 N/A N/A N/A **35.9**	 313 N/A N/A N/A N/A	As the training setup is not publicly available for [1], they re-implemented the pruning procedure and achieve similar results to the original paper. The result of pruning pre-trained and scratch model is shown in Table 3 and

Pruned From Scratch with SA	6.42	**6.46**	0.04	0.0062	N/A	N/A	34.2	N/A	Table 4, respectively.
5. Table 2 on page 2795 Lin, Shaohui, et al. (2019) VGGNet GAL 0.05 GAL 0.1	 6.04 6.04 6.04	 7.1 7.9	 1.06 1.86	 0.1755 0.3079	 77.6 82.2	 14.98 3.36 2.67	 39.6 45.2	 313.7 189.5 171.9	Filters and other structures (channels) pruned. Average pruned error = 7.1 for lambda 0.05 and 7.9 for lambda 0.1
6. Table 1 on page 1534 Lin, Mingbao, et al. (2020) VGGNet HRank1 HRank2 HRank3	 6.04 6.04 6.04 6.04	 6.57 7.66 8.77	 0.53 1.62 2.73	 0.0877 0.2682 0.4519	 82.9 82.1 92	 14.98 2.51 2.64 1.78	 53.5 65.3 76.5	 313.7 145.6 108.6 73.7	Didn't specifically mention VGG16; model given as VGGNet. But, compares the results with papers (2 and 5 listed here and with two more other papers) using VGG16.
7. Pages 91,93, 95 and Tables 4, 5 and 6 Al Dallal (2021) VGGNet Experiment 1 Experiment 2 Experiment 3	 6.04 6.04 6.04 6.04	 8.5 9.02 9	 2.46 2.98 2.96	 0.4072 0.4934 0.4900	 56.3 64.84 70.66	 14.98 N/A N/A N/A	 56.9 64.8 66.7	 313.7 N/A N/A N/A	Uses the model used in paper [6] and compares results only with [6].
8. Table 1 on page 3 Ayinde, B. O., & Zurada, J. M. (2018) VGG-16 Experiment A Experiment B	 6.2 6.2 6.2	 **6.33** 6.7	 **0.13** 0.5	 **0.0209** 0.0806	 **78.1** 78.1	 14.7 **3.23** 3.23	 **40.5** 40.5	 313 **186** 186	Compares their result with paper 1

9. Tables 5 and 6 on page 5630 Li, Yawei, et al (2019) VGG-16 Experiment 1 (Table 5) Experiment 2 (Table 6)	5.98 5.98 5.98	6.18 6.23	0.2 0.25	0.0334 0.0418	78.2 N/A	14.7 3.21	N/A 76.5	313 N/A	In the VGG16 architecture they used only one FC layer after the last pooling layer (normally it is three FC layers)
10. Table IV at page 682 Carballo, M.V. and Lee, B.K., (2020) VGG-16 experiment	7.19 7.19	7.52	0.33	0.0459	35.21x Or 99.25	527.79 MB 3.93M B	5.66 x Or 82.3	15.48 2.74	Applied quantization in fully connected layers the reduction in parameters size goes up to 47.28x compared to the original model. Due to the use of filter pruning, floating point operations per second (FLOPs) in the model are considerably reduced and is 1.72x

Note: in the above table 1st paper in the References list is 5 and papers 2 to 10 in the References list 14 to 22 respectively.

3.2 Some observations on experimental results

It appears that most papers in the table took the 1st paper listed by Li et al. (2016) as the baseline to compare their results probably because it was one of the best papers published on pruning that appeared early when compared with other works in a chronological order. They tried to show that their respective methods were able to achieve either a better accuracy at a

comparably same pruning rate or a higher accuracy with a higher pruning rate. Some others wanted to achieve a better FLOPS reduction rate than achieved by Li et al. (2016). For example, the paper by He, Yang, et al. (2019) on row four of the table reported only the FLOPS reduction rate. Yet, most probably they did not use the same hardware and the software framework to run their experiments; which means that it might hardly be possible to compare between papers.

The paper by Carballo, M.V. and Lee, B.K., (2020) in the last row of the table, used both pruning and quantization to reduce size and FLOPS.

By taking into account the calculated error in accuracy as a percent change (PC) in the tabulated list, we selected a specific range for PC and then tried to organize the papers in the table by the size and the flops reduction rate of the pruned model in the following manner:

If error in accuracy as a percent change (PC) is in the range 0.001 to 0.18, then

1.

Descending order of the papers according to the size of the pruned network is:

3, 6, 8, 9, 5, 2, 1

According to the above ordering Paper 3 reported the highest amount in pruning parameters of 83% while 1 reported a 64%

The following papers were not considered in the above ordering for the stated reason:

Paper 4 – Did not report the size of the pruned model

Paper 7 – The error in accuracy as a percent change (PC) is far too big and not in the range 0.001 to 0.18

Paper 10 - This paper used quantization as well as pruning to reduce the model size and flop count.

2.

Descending order of the papers according to flop reduction is:

9, 6, 3, 8, 5, 2, 4, 1

According to the stated ordering Paper 9 reported the highest amount in flops reduction of 76.5% while paper 1 the least of 34.2%

The following papers were not considered in the above ordering for the stated reason:

Paper 7 – The error in accuracy as a percent change (PC) is far too big and not in the range 0.001 to 0.18

Paper 10 - This paper used quantization as well as pruning to reduce the model size and flop count

We could gain some insights as above from the tabulated data.

From the above discussion one may conclude that experimental results could be compared provided that experiments were congruent that is, the hyperparameters of each experiment were kept identical. However, should the experimental environments differ, comparisons of the obtained results is not possible. This was identified by the authors of [3] who urged the complete disclosure of the details of the experimental methodology used, and the development of benchmark suites which can be used to disclose the efficacy of new methods.

3.3 Summary of Chapter 3

In this chapter we were able to get an understanding of the type of metrics related to pruning as was actually reported in literature by tabulating those results as given in ten randomly selected papers from a Google Scholar search on filter pruning that was trained using the model VGG16 on CIFAR-10 dataset.

As was mentioned in Section 3.2 the results were not that random because we included the top cited paper which was ranked first in the initial Google Scholar search.

Chapter 4

A replicated experiment on pruning

4.0 Introduction

This chapter examines an experiment on pruning by replicating a publicly available ipython notebook developed by Google Research [23]. The experiment enables to get a basic understanding of how pruning a neural network is implemented in the Python code using the facilities available from the TensorFlow framework running with the Keras API. This software framework hides many of the implementation details from the programmer and provides a smooth interface to perform the function of pruning.

The pruning experiment reported in this chapter is performed on a multi-layer perceptron (MLP) or a feed-forward network with four hidden layers containing 1000, 1000, 500, 200 neurons respectively and trained on the MNIST and FMNIST (fashion MNIST) datasets. This experiment illustrates how pruning will reduce the number of parameters in fully connected layers (FCC) both in connections (weights) and neurons (in two separate experiments) and still maintain the original test accuracy to some extent. The experiment is constructed in a way to view how the value of test accuracy changers with different sparsity levels. Also, the dese layer visualization facility available with the ipython notebook allows one to view how much sparse is each dense layer where the weights are color coded. Additionally, when the ipython notebook is run on Google Colab it gives the experimenter an opportunity to choose the dataset (MNIST or FMNIST), specify which elements to prune (weights or neurons) and set the level of sparsity for the particular MLP model coded and run the experiment any number of times and visualise results i.e. how much sparse is each dense layer.

As explained at the beginning of the notebook [23], the experiment attempts to illustrate the two most popular pruning methods by candidate/element; namely weight pruning by setting individual weights in

the weight matrix to zero which corresponds to deleting connections between neurons in different layers and unit/Neuron pruning: set entire columns in the weight matrix to zero, which will provide the effect of deleting the corresponding output neuron.

In the case of weight pruning to achieve sparsity it is required to rank the individual weights in weight matrix according to their magnitude (absolute value), and then set to zero the smallest. On the other hand for unit/Neuron pruning and to achieve sparsity one needs to rank the columns of a weight matrix according to their L2-norm and delete the smallest. The code in the ipython notebook illustrates how this is actually implemented.

Yet, as expected one will also observe that with a gradual increase in the level of sparsity by deleting more of the network, the task performance will progressively degrade.

The main takeaway from running the experiment is one gets a hands-on experience to gain a basic understanding of how weight or unit pruning is actually implemented for a simple network like a MLP in code.

The replicated experimental code run was saved as a pdf file and is available in the Appendix.

4.1 Some results which demonstrates the effect of pruning on test accuracy

The tables given below list the results of weight and unit pruning of the MLP trained on the MNIST dataset:

MNIST Weight-pruning

k% weight sparsity:	0.0	Test loss: 0.08932	Test accuracy: 98.15 %%
k% weight sparsity:	0.25	Test loss: 0.08758	Test accuracy: 98.16 %%
k% weight sparsity:	0.5	Test loss: 0.07665	Test accuracy: 98.17 %%
k% weight sparsity:	0.6	Test loss: 0.07408	Test accuracy: 97.98 %%
k% weight sparsity:	0.7	Test loss: 0.08247	Test accuracy: 97.72 %%
k% weight sparsity:	0.8	Test loss: 0.19302	Test accuracy: 96.92 %%
k% weight sparsity:	0.9	Test loss: 1.15709	Test accuracy: 87.50 %%
k% weight sparsity:	0.95	Test loss: 2.06444	Test accuracy: 33.11 %%
k% weight sparsity:	0.97	Test loss: 2.24438	Test accuracy: 10.52 %%
k% weight sparsity:	0.99	Test loss: 2.30362	Test accuracy: 09.74 %%

As could be seen from above results for weight pruning the test accuracy increases more than that of the unpruned model up to a sparsity level of 50% and then there is only a slight reduction in the test accuracy up to 90% of sparsity. After that there is a significant decrease in test accuracy.

```
MNIST Unit-pruning

k% weight sparsity:  0.0      Test loss: 0.08932     Test accuracy: 98.15 %%
k% weight sparsity:  0.25     Test loss: 0.07550     Test accuracy: 98.16 %%
k% weight sparsity:  0.5      Test loss: 0.11627     Test accuracy: 97.90 %%
k% weight sparsity:  0.6      Test loss: 0.31343     Test accuracy: 97.60 %%
k% weight sparsity:  0.7      Test loss: 0.95761     Test accuracy: 96.54 %%
k% weight sparsity:  0.8      Test loss: 1.79810     Test accuracy: 90.19 %%
k% weight sparsity:  0.9      Test loss: 2.25901     Test accuracy: 13.84 %%
k% weight sparsity:  0.95     Test loss: 2.30469     Test accuracy: 09.74 %%
k% weight sparsity:  0.97     Test loss: 2.30525     Test accuracy: 09.74 %%
k% weight sparsity:  0.99     Test loss: 2.30526     Test accuracy: 09.74 %%
```

For unit pruning the results are slightly different. Test accuracy slightly increases than that of the unpruned model up to 25% of sparsity and then there is not much reduction in accuracy up to about 80% of sparsity and after which there is a significant degradation in the performance.

The following tables list the results of weight and unit pruning of the MLP trained on the FMNIST dataset:

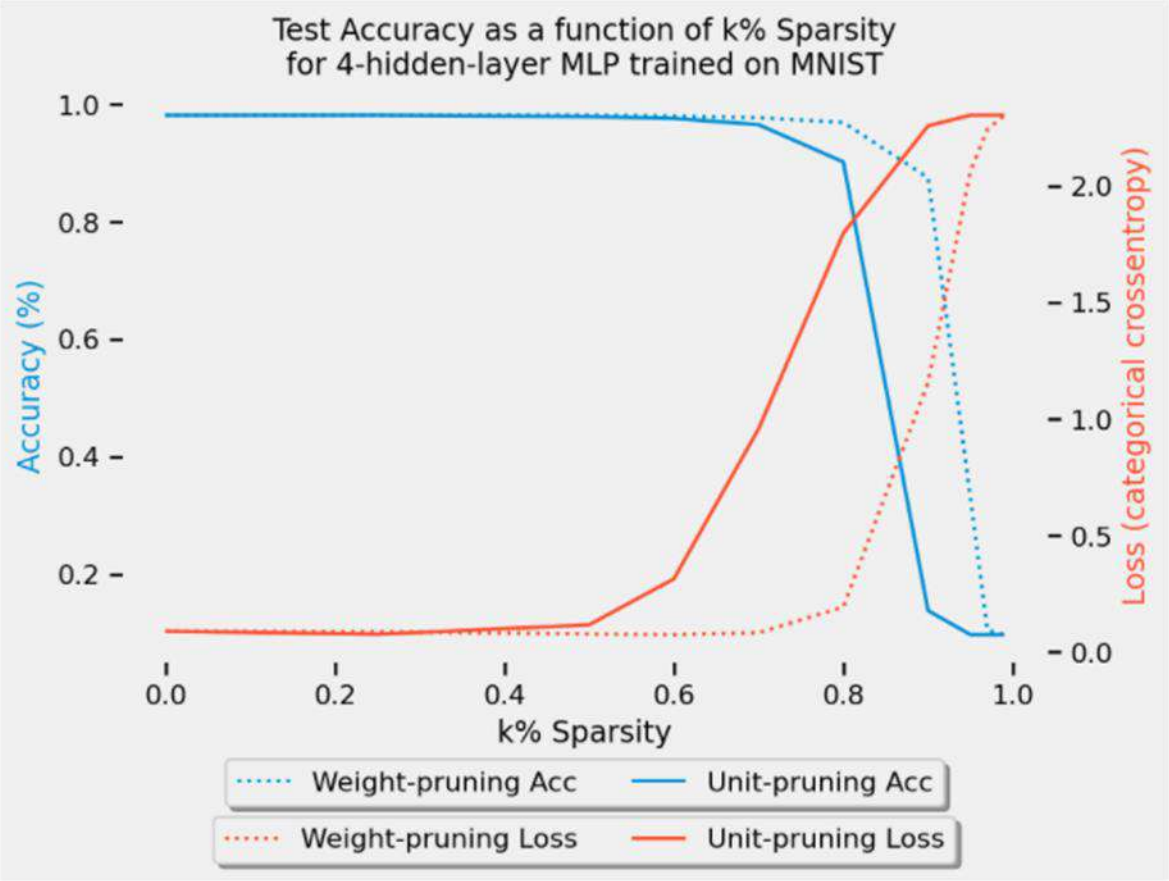
```
FMNIST Weight-pruning

k% weight sparsity:  0.0      Test loss: 0.32409     Test accuracy: 88.58 %%
k% weight sparsity:  0.25     Test loss: 0.32096     Test accuracy: 88.57 %%
k% weight sparsity:  0.5      Test loss: 0.31192     Test accuracy: 88.48 %%
k% weight sparsity:  0.6      Test loss: 0.30943     Test accuracy: 89.01 %%
k% weight sparsity:  0.7      Test loss: 0.33852     Test accuracy: 88.40 %%
k% weight sparsity:  0.8      Test loss: 0.46297     Test accuracy: 86.07 %%
k% weight sparsity:  0.9      Test loss: 1.03374     Test accuracy: 71.64 %%
k% weight sparsity:  0.95     Test loss: 1.79723     Test accuracy: 53.46 %%
k% weight sparsity:  0.97     Test loss: 2.11900     Test accuracy: 38.88 %%
k% weight sparsity:  0.99     Test loss: 2.29444     Test accuracy: 14.85 %%
```

```
FMNIST Unit-pruning

k% weight sparsity:  0.0       Test loss: 0.32409      Test accuracy: 88.58 %%
k% weight sparsity:  0.25      Test loss: 0.31814      Test accuracy: 88.69 %%
k% weight sparsity:  0.5       Test loss: 0.40722      Test accuracy: 87.72 %%
k% weight sparsity:  0.6       Test loss: 0.63275      Test accuracy: 84.65 %%
k% weight sparsity:  0.7       Test loss: 0.90360      Test accuracy: 77.60 %%
k% weight sparsity:  0.8       Test loss: 1.56253      Test accuracy: 60.06 %%
k% weight sparsity:  0.9       Test loss: 2.18632      Test accuracy: 26.99 %%
k% weight sparsity:  0.95      Test loss: 2.29995      Test accuracy: 16.46 %%
k% weight sparsity:  0.97      Test loss: 2.30242      Test accuracy: 10.39 %%
k% weight sparsity:  0.99      Test loss: 2.30491      Test accuracy: 10.00 %%
```

As for the FMNIST pruning experiment weight pruning is some what similar for that obtained for the case of unit pruning with the MNIST dataset discussed previously. In the case of unit pruning drop in test accuracy happens much earlier than for the case with the MNIST dataset.

4.2 The performance curves for unit and weight pruning

The following figure illustrates the performance curves for both weight and unit pruning obtained by pruning the MLP trained on MNIST

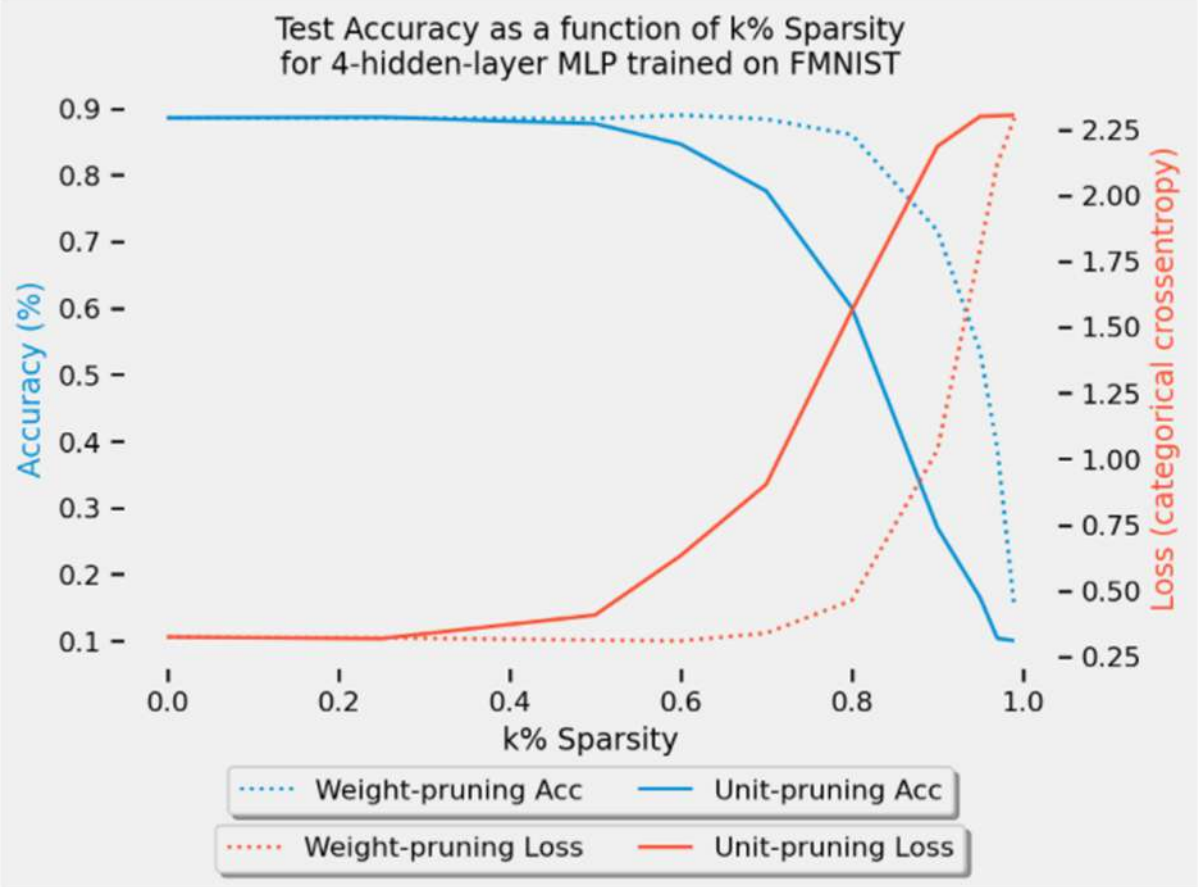


As could be seen the performance curves for unit and weight pruning differ on MNIST.

Pruning weight matrices of dense layers does not result in a significant drops in accuracy or increases in loss until around k=80. Even then, the accuracy does not begin to noticeably decrease until k=90.

For unit-pruning, accuracy begins to fall earlier, around k=70 (with loss beginning to increase around k=60). Even then, both methods are able to effectively remove more than half of the network weights without any dramatic differences in test classification performance.

The following figure illustrates the respective performance curves for both weight and unit pruning obtained by pruning the MLP trained on FMNIST



Even on FMNIST, which has a lower initial accuracy and higher initial loss, the same pattern emerges

Again, pruning weight matrices of dense layers does not result in dramatic drops in accuracy or increases in loss until around k=80. Even then, the accuracy does not begin to noticeably decrease until k=90.

For unit-pruning, the differences come much earlier for FMNIST than in MNIST. Accuracy begins to fall around k=60 (with loss beginning to increase around k=60).

4.3 Visualizing the sparsity in dense layers

As already mentioned in the introduction, running the notebook allows one to visualize the sparsity level of each dese layer since the weights are color coded. The following figures illustrate the level of sparsity in each dense layer (given as weight matrices) for all four dense layers when the MLP was trained on MNIST and the candidate for pruning was set to unit (or

neuron) while the level of sparsity changes from 0%, 50%. The indicated weight values are positive because the L2 norm was used to select weights for pruning. The pruning was done after training the original network and iteratively pruning and retraining later to obtain the accuracy for each sparsity level. The figures depict the magnitude of the weights.

Just to remind ourselves, that we are considering a MLP with four hidden layers. These layers will be dense, fully-connected layers with sizes 1000, 1000, 500 & 200. We will also have a fifth layer for the output logits, in which there will be 10 neurons.

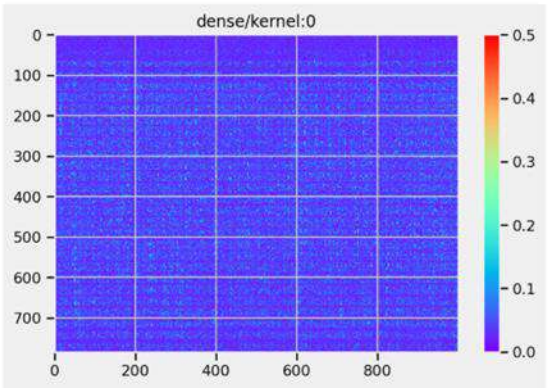
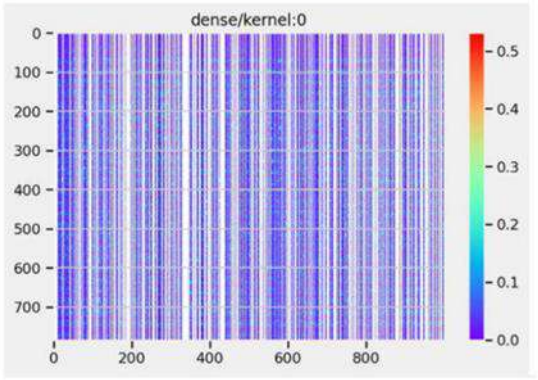
Original model 0% sparsity	50% sparsity
	

Figure 1. Visualization of the weights (connections between the input features and the 1st dense layer) matrix of 784x1000 elements

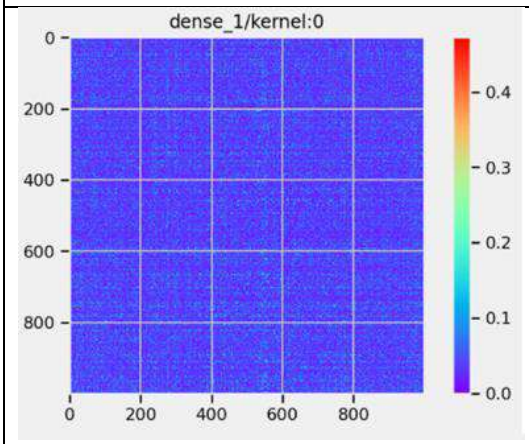
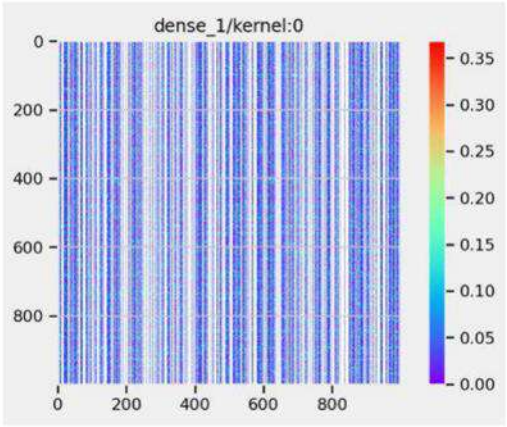
Original model 0% sparsity	50% sparsity
	

Figure 2. Visualization of the weights (connections between the 1st and the 2nd dense layers) matrix of 1000x1000 elements

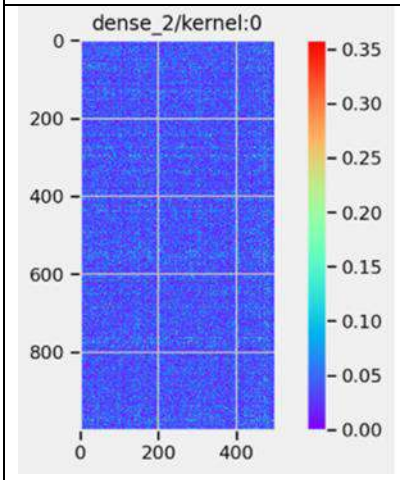
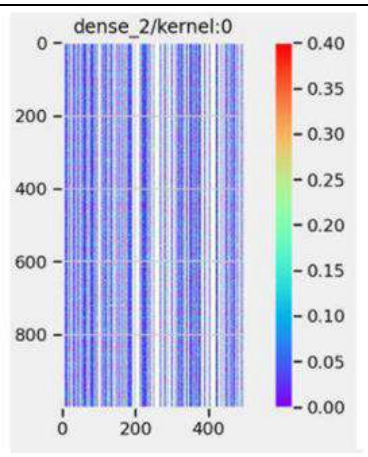
Original model 0% sparsity	50% sparsity
	

Figure 3. Visualization of the weights (connections between the 2nd and the 3rd dense layers) matrix of 1000x500 elements

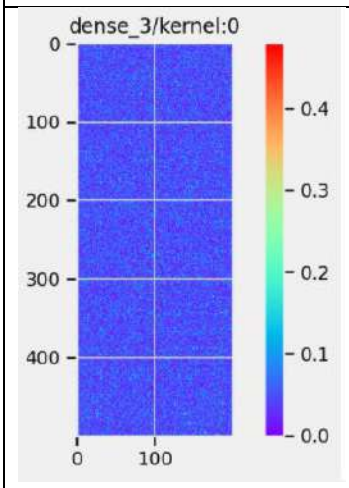
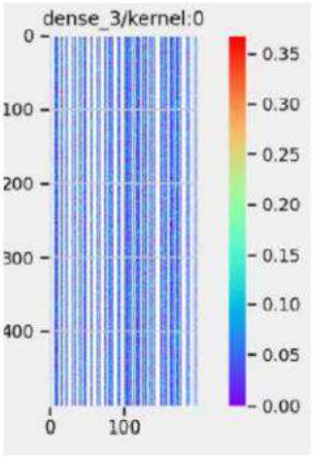
Original model 0% sparsity	50% sparsity
	

Figure 4. Visualization of the weights (connections between the 3rd and the 4th dense layers) matrix of 500x200 elements

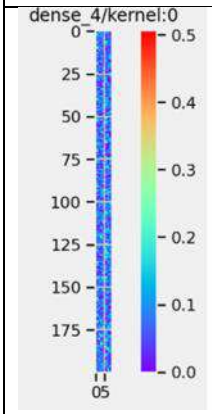
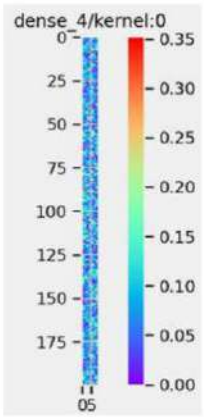
Original model 0% sparsity	50% sparsity
	

Figure 5. Visualization of the weights (connections between the 4th and the output layers) matrix of 200x10 elements

4.4 Loss accuracy of the compressed model

In the final parts of the notebook it is demonstrated how already pruned and saved models are compressed for matrix acceleration.

The following results were obtained for the dense model trained on MNIST with a sparsity level of 95% that compression for matrix acceleration does not have any effect on the model accuracy.

```
k% weight sparsity (unit pruning):  0.95
uncompressed    Test loss: 2.30439      Test accuracy: 09.74 %%
compressed      Test loss: 2.30439      Test accuracy: 09.74 %%
```

Further the following depicts the execution time comparisons for the uncompressed and the compressed models for which the test loss/accuracy were given earlier:

```
[26] %time
     old_score = sparse_model.evaluate(x_test, y_test, verbose=0)

     CPU times: user 5 µs, sys: 0 ns, total: 5 µs
     Wall time: 9.06 µs
```

```
[27] %time
     new_score = compressed_model.evaluate(x_test, y_test, verbose=0)

     CPU times: user 3 µs, sys: 0 ns, total: 3 µs
     Wall time: 6.68 µs
```

As could be seen from the results the compressed model takes less time for execution.

Finally it could be concluded that not only does the compressed model get similar loss and accuracy as the non-compressed, but it also achieves the same result in less time.

4.5 Summary of Chapter 4

In this chapter we explored the results of the execution of an iphython note book [23] publicly available from Google Research to understand the idea

of weight and unit pruning as applied to a dense model trained on both MNIST and FMNIST for results comparison. By looking at the results one could gain an understanding how pruning works in the respective cases and more over how pruning algorithms and related code is implemented in a software framework like TensorFlow.

The results of the code run saved as a pdf file is available in the Appendix.

Chapter 5

Some ideas on empirical rigor

In this chapter we will consider some meta-analysis reports which were done by scholars in the research community to find out about the trends in the quality of research publications in general and certain problems in how experiments are carried out and results reported back to the community. Here we will discuss few possible reasons for this state of affaires as explained by some researchers and some guidelines towards remediation proposed by some others.

As was already pointed out in Chapter 3, the study of DNNs are based on heuristic methods. Due to this, the size and the performance of the pruned networks are not always guaranteed. Other than that, as explained in [24] they require cumbersome ablation studies or manual hyper-parameter tuning. This state of affairs has led some researchers in the community to claim that research in machine learning is like performing ‘alchemy’ in ancient times [25] while some others [9] propose remedial measures such as an open source library to standardise the pruning process.

5.0 Status of published research

5.0.1 Troubling Trends in Machine Learning Scholarship

The authors of the paper [26] observed that as a community the machine learning (ML) researchers are engaged in the creation and dissemination of knowledge about data-driven algorithms. That in a given paper, researchers might aim to achieve following goals:

- to theoretically characterize what is learnable, (try to build a theory about how machines learn)
- to obtain understanding through empirically rigorous experiments
- to build a working system that has high predictive accuracy

By checking publications on machine learning they [26] found the following four patterns appear to be trending in ML scholarship:

1. failure to distinguish between explanation and speculation
2. failure to identify the sources of empirical gains, e.g., emphasizing unnecessary modifications to neural architectures when gains actually stem from hyper-parameter tuning
3. mathiness: the use of mathematics that complicates or impresses rather than clarifies, e.g., by confusing technical and non-technical concepts
4. misuse of language, e.g., by choosing terms with informal meanings or by overloading established technical terms.

They think [26] that possible causes for these patterns might be

1. rapid expansion of the research community
2. the lack of expert reviewer pool as a result
3. misaligned incentives between scholarship and short-term measures of success (e.g. bibliometrics, attention, and entrepreneurial opportunity)

Finally, they comment [26] that papers are most valuable to the community when they aid the reader to obtain foundational knowledge while communicating as clearly as possible.

5.0.2 Ideas for improving publications in the field of Machine Learning

Due to the concerns as stated in [26], some international conferences like NeurIPS commenced special workshops to address these issues.

For example, as an effort towards addressing these problems some guidelines were provided in [27] which summarized discussions from the NeurIPS 2019 Retrospectives Workshop as follows:

- Incentivize openness and alternate forms of scholarship
- Re-structure the review process
- Participation from academia and industry

- Start a discussion on how to train computer scientists to do better science?

5.1 About empirical rigor

Due to the fact that research in deep learning is mainly advanced through experiments without hardly any theory to guide the experimental work, conducting experiments in systemic rigorous manner is an essential requirement.

A presentation done at ICLR 2018 [28] which was an informal meta-analysis based on several published papers each of which individually has found evidence in which a possible lack of rigorous standards for empirical work.

Some problems identified in [28] were

- Empirical studies have become challenges to be “won”, rather a process for developing insight and understanding
- Price of compute - Large research groups (often based in industry) may have the resources, for example, to tune models on 450 GPUs for 7 days but individual researchers may be harder pressed
- Not taking adequate time to perform fine-grained empirical analysis due to publication deadline pressure

Overall, the authors [28] highlighted very important issues related to the empirical studies conducted and reported in the recent past. They [28] emphasized that the lack of standards in empirical rigor is a most recent trend. They also emphasized the standards for reviews and reviewers of conference papers. They [28] noted that the only way to ensure reproducible research is to make available code with the paper and recommended alternative paper formats, including smart notebooks like iPython that include code, data, and analysis along with text.

5.2 The state of neural network pruning experiments done by the wider research community and some recommendations for improvements in the future

To publish the report [10] which is a meta-analysis of the literature on neural network pruning done by Dr. John Guttag and his team at the MIT; the authors gathered results across 81 already published papers and pruned/trained hundreds of models in controlled conditions as was mentioned in the published literature. Their objective was to extract practical lessons for the wider research community through their inquiry. To achieve this, they [10] set about to find answers for example to following questions:

- Which technique achieves the best accuracy/efficiency tradeoff?
- Are there strategies that work best on specific architectures or datasets?
- Which high-level design choices are most effective?

Their [10] main finding was that the state of the published literature is such that the, above mentioned motivating questions are impossible to answer. They explain their frustration in the following manner “Few papers compare to one another, and methodologies are so inconsistent between papers that they could not make these comparisons themselves. They found that for example, a quarter of papers compare to no other pruning method, half of papers compare to at most one other method, and dozens of methods have never been compared to by any subsequent work. In addition, no dataset/network pair appears in even a third of papers, evaluation metrics differ widely, and hyperparameters and other confounders vary or are left unspecified.”

In the report [10] the researchers argued “that existing work tends to

• make it difficult to identify the exact experimental setup and metrics,

• use too few (dataset, architecture) combinations, [use at least 3]

• report too few points in the tradeoff curve for any given combination, and no measures of central tendency, [plot at least 5 points]

• omit comparison to many methods that might be state of-the-art, and

• fail to control the confounding variables.”

They concluded that these problems often make it difficult or impossible to assess the relative effectiveness of different pruning methods. To enable direct comparison between methods in the future, they [10] suggested “the following practices:

- Identify the exact architectures, datasets, and metrics that were used, preferably in a systematic manner that isn't spread across the results section.
- At least three (dataset, architecture) pairs should be used, with one of them being modern and large-scale. MNIST and toy models are excluded. The architectures AlexNet, CaffeNet, and Lenet-5 are no longer considered current.
- Report both the compression ratio and the theoretical speedup for each pruned model. The original size divided by the new size is the compression ratio. The original number of multiply-adds (MACS) divided by the new number represents the theoretical speedup. It's worth noting that there is no reason to report only one of these metrics.
- Report both Top-1 and Top-5 accuracy for ImageNet and other many-class datasets. There's no reason to only report one of these.
- Any metrics reported for a pruned model should also be reported for an appropriate control model (usually the original model before pruning).
- Plot the tradeoff curve for a particular dataset and architecture along with the curves for competing techniques.
- Use at least 5 operating points across a range of compression ratios when plotting tradeoff curves. A good set of ratios to use is 2, 4, 8, 16, and 32.
- Whenever appropriate, report and plot means and sample standard deviations rather than one-off measurements.
- To the greatest extent possible, ensure that the methods being compared use identical libraries, data loading, and other code.

It was also suggested that when examining papers that claim to give a better approach of pruning neural networks, reviewers at international conferences require a much higher level of rigour.”

In general, the authors of [10] discovered that the research community lacks consistent benchmarks and measurements, making it difficult to compare pruning strategies or evaluate how far the discipline has progressed over the last three decades. They provided an efficient way to publish results by establishing an open-source pruning library called ShrinkBench as a solution to this challenge.

ShrinkBench employs a standardised collection of pretrained models and datasets to provide uniform and extendable functionality for training, pruning, fine-tuning, computing metrics, and graphing. , which, according to the study [10] is available on GitHub.

ShrinkBench is built on PyTorch and is intended to make evaluating methods with arbitrary scoring functions, layer pruning, and sparsity structures simple. ShrinkBench will automatically apply pruning, update the network according to a standard training or fine-tuning setup, and compute metrics across many models, datasets, random seeds, and levels of pruning when given a callback defining how to compute masks for a model's parameter tensors at a given iteration. They [10] direct readers to the project's documentation for a breakdown of ShrinkBench's implementation and API.

5.3 Findings of some other experiments that replicated already published research

Other than the already mentioned meta-analysis reports considered in this chapter; this study ‘The State of Sparsity in Deep Neural Networks’ [29] done in 2019 is also considered here to highlight problems about reported results in published papers. Please note this report [29] is a preprint available from arXiv.org and was directed by researchers at Google Brain.

This study was done by the researchers while at a Google AI residency program. They replicated thousands of experiments details of which were already published and found some current trends such as simple

magnitude pruning works best for larger networks and emphasized the need for large-scale benchmarks. In addition to the above findings, while exploring the existing literature they also found that there is some disagreement in the research community on how to explain sparsity in neural networks.

In their report [29] Gale et. al. (2019) mentions that few other multiple independent studies have recently proposed that the value of sparsification in neural networks has been misunderstood. They further state that these studies suggest that sparsification can be viewed as a form of neural architecture search, but the researchers disagree on what is required to achieve this objective. Specifically one study [30], Liu et al. (2018) re-trained learned sparse topologies with a random weight initialization, whereas the other study [31] Frankle & Carbin (2018) postulate that the exact random weight initialization used when the sparse architecture was learned is needed to match the test set performance of the model sparsified during optimization. So, Gale et. al. (2019) [29] replicated the experiments performed by (Frankle & Carbin, 2018) [31] and (Liu et al., 2018) [30] at scale and shows that unstructured sparse architectures learned through pruning cannot be trained from scratch to the same test set performance as a model trained with joint sparsification and optimization. Other than above replication experiments Gale et. al. (2019) [29] rigorously evaluated three state-of-the-art techniques for inducing sparsity in deep neural networks on two large-scale learning tasks: Transformer trained on WMT 2014 English-to-German translation, and ResNet-50 trained on ImageNet. Further, performing thousands of experiments, Gale et. al. (2019) [29] demonstrated that complex techniques (Molchanov et al., 2017 [32]; Louizos et al., 2017) [33] shown to produce high compression rates on smaller datasets perform inconsistently, and that simple magnitude pruning approaches achieve comparable or better results for bigger networks. Gale et. al. (2019) [29] concluded that these results highlight the need for large-scale benchmarks in the field of model compression.

5.4 Summary of Chapter 5

We considered in this chapter some ideas expressed about the status of machine learning publications by some scholars in the community based on meta analysis reports. Some other ideas discussed at recent

international conferences such as at the Retrospectives Workshop in NeurIPS 2019, might also help improve future research and publications.

The scholars in the research community emphasised the need for empirical rigor and the study from the MIT team [10] had an interest to guide the research community to perform more rigorous experiments and showed how to report their findings in research publications in a comprehensible manner.

The final study [29] carried out by Gale et. al. (2019), a study undertaken by the researchers while at a Google AI Residency by replicating experiments thousands of times for rechecking as the researchers claim; stressed the need for benchmarks for large-scale networks and which also showed that better results were reported by simple magnitude pruning for larger networks than using complicated pruning schemes as reported in the existing research. The approach taken in [29] might be a way to advance research in the deep learning field.

Chapter 6

Efficient deployment of DNNs

There are some technical issues that hinder the efficient deployment of DNNs.

In the first part of this chapter we will consider such technical issues specific to pruning and in the second part some common issues associated with DNNs or datasets on which they are trained and how pruning will affect these technical issues.

In Sections 6.0 we will consider three recent studies that highlight some technical issues identified in neural networks due to pruning. Based on their experimental findings the researchers of the last two studies advice to be cautious when deploying such models in mission critical systems.

Starting from Section 6.1 we will consider few common technical issues and whether pruning will have an effect on them.

6.0.1 About getting high errors on a small subset of examples for a pruned model

The paper [34] published in December 2020 which reports a study undertaken by researchers at Google Brain found that even though techniques like pruning achieve high levels of compression with negligible impact on top-line metrics (top-1 and top-5 accuracy) this overall accuracy hides disproportionately high errors on a small subset of examples; which they called Compression Identified Exemplars (CIEs). Further they found for these CIE examples, compression amplifies existing algorithmic bias. These examples were typically less well represented in the training data. The experiment was done on using a pruned model of CNNs on a classification task on the ImageNet dataset using the unstructured iterative magnitude pruning method. The researchers of [34] found further, that compared with pruning; quantization results in a much smaller impact to different classes.

In addition, they found that pruned models are significantly more brittle under distribution shifts, such as corrupted images in ImageNetC [Hendrycks and Dietterich 2019] [35] or naturally adversarial images in ImageNet-A [Hendrycks et al. 2021] [36].

They [34] concluded that pruning disproportionately impacts performance on underrepresented features and that even though these CIEs are a relatively small subset but they greatly contributes to errors in the model. So, as a remedy they proposed its use as a human-in-the-loop auditing tool to identify a tractable subset of the dataset for further inspection or annotation by a domain expert.

6.0.2 About ‘pruning identified exemplars’ (PIEs) and their impact on pruning

The next study [37] which we will be discussing in this section was again undertaken by researchers at Google Brain who were partially involved in the previous study [34] discussed in 6.0.1 and was published in September 2021.

The researchers state what prompted them to do this study was that the ability to prune networks with apparently so little degradation to generalization performance seemed to them to be puzzling.

So, they [37] re-evaluated the quantization and pruning techniques which are known to be already widely used in production systems and integrated with popular deep learning libraries. What they found was the reliance on top-line metrics such as top-1 or top-5 test-set accuracy (which is the normally used metric to evaluate performance) hides critical details in the ways that pruning impacts model generalization.

While experimenting they [37] did thousands of large scale experiments and established consistent results across multiple datasets— CIFAR-10, CelebA and ImageNet on widely used pruning and quantization techniques, and model architectures.

Their results show that the increased errors on certain classes caused by pruning can amplify existing algorithmic biases. On CelebA, a dataset of celebrity faces with significant correlations between demographic groups, pruning increases errors on underrepresented subgroups. For example,

pruning a model trained to identify people with blond hair to 95% sparsity increased the average false-positive rate for men by 49.54%, but by only 6.32% for others. So they point out the biases and brittleness introduced by pruning may limit the utility of pruned models, especially in situations that often deal with protected attributes and are sensitive to fairness, such as facial recognition or healthcare.

Through their research they propose a formal framework to identify the classes and images where there is a high level of disagreement or difference in generalization performance between pruned and non-pruned models. They found that certain examples, which they named as ‘pruning identified exemplars’ (PIEs), and classes are steadily more impacted by the introduction of sparsity. Some examples of PIEs are shown in the paper [37]. Please note that Compression Identified Exemplars (CIEs) as mentioned in [34] and PIEs mentioned in [37] are the same. Probably, the researchers thought ‘pruning identified exemplars’ (PIEs) is a better name because it highlights from where the problem comes; specifically from pruning.

So they recommend “Caution should be used before deploying compressed models to sensitive domains such as hiring, health care diagnostics, self-driving cars, facial recognition software. For these domains, the introduction of pruning may be at odds with the need to guarantee a certain level of recall or performance for certain subsets of the dataset.” [37]

They also found pruning significantly reduces robustness to image corruptions and natural adversarial images as found in [34].

A plot of both absolute % change in class recall and the normalized accuracy relative to change in overall top-1 accuracy caused by pruning for different levels of pruning (30%, 50%, 70% and 90%) could be visually seen as an animation found on the website [38] done by one of the authors of [34] and [37].

6.0.3 A set of easy-to-follow guidelines for safe pruning in practice

The next study [39] was done at the Computer Science and Artificial Intelligence Lab at MIT and was published in March 2021 and later presented at the Proceedings of Machine Learning and Systems 3 (MLSys 2021). In this paper the authors wanted to re-examine the common assumptions as to why pruning works well such as stated in [39] ‘While the ability to prune large portions of a network is not obvious at first sight, common wisdom attributes the apparent overparameterization of modern deep learning architectures as one of the key reasons why pruning such large portions of the network is possible without harming the performance of the network.’

So, they [39] undertook to rigorously assess how pruning using state-of-the-art prune/retrain techniques affects the function represented by a neural network, including the similarities and disparities exhibited by a pruned network with respect to its unpruned counterpart.

Their [39] findings are as follows:

- The pruned models are functionally similar to the uncompressed parent model, which enables to distinguish the parent of a pruned network for a range of prune ratios.
- Despite the similarity between the pruned network and its parent, it could be observed that the prune potential of the network varies significantly for a large number of tasks. Here an example of a task is autonomous driving.
- A pruned model may be of similar predictive power as the original one when it comes to test accuracy, but may be much more brittle when faced with out-of-distribution data points
- This raises concerns about deploying pruned models on the basis of accuracy alone, in particular for safety-critical applications such as autonomous driving), where unforeseen, out-of-distribution, or noisy data points commonly arise
- Highlight the need to consider task-specific evaluation metrics during pruning which hold even when considering robust training objectives,

prior to the deployment of a pruned network to, e.g., safety-critical systems
- These results also question the common assumption that there exists a significant amount of “redundant” parameters to begin with and provide a robust framework to measure the amount of genuine overparameterization in networks.[39]

Based on above observations they [39] formulated a set of easy-to-follow guidelines for pruning neural networks in practice for efficient deployment:

1. Don’t prune if unexpected shifts in the data distribution may occur during deployment.

2. Prune moderately if you have partial knowledge of the distribution shifts during training and pruning.

3. Prune to the full extent if you can account for all shifts in the data distribution during training and pruning.

4. Maximize the prune potential by explicitly considering data augmentation during retraining.

In the next section of this chapter (second part) we will consider following common technical problems found in deep neural networks (DNNs) and associated datasets:

- Brittleness (serious for critical DNN applications)
- Errors in test datasets

We will explore each one of them and check whether pruning neural networks will alleviate or worsen these problems or does not have any effect on them at all?

6.1.0 Brittleness of DNNs and pruning

The article named ‘Why deep-learning AIs are so easy to fool?’ [40] explained how deep neural networks could be easily broken. It cited and showed example situations where researchers found how easy it is to fool these networks and recommended caution when deploying these networks in mission critical situations.

As already explained in the previous section 6.0.2; the study [37] done through extensive experimentation by some researchers at Google Brain in September 2021 found that that pruning significantly reduces robustness to image corruptions and natural adversarial images. This means that pruning worsens the already brittle nature of existing DNNs.

6.1.1 Issues related to datasets and models

6.1.1.1 Label Errors in test datasets

A study named ‘Pervasive Label Errors in Test Sets Destabilize Machine Learning Benchmarks’ [41] found label errors on ten popular benchmark test datasets.

They found [41]

- lower capacity models may be practically more useful than higher capacity models in real-world datasets with high proportions of erroneously labeled data
- on ImageNet with corrected labels: ResNet-18 outperforms ResNet-50 if the prevalence of originally mislabeled test examples increases by just 6%
- On CIFAR-10 with corrected labels: VGG-11 outperforms VGG-19 if the prevalence of originally mislabeled test examples increases by just 5%

These test set errors across the 10 datasets can be viewed at https://labelerrors.com

Pruning is desirable and optimizes the network whether there are labelling errors or not. Yet, correcting errors in test datasets especially on large datasets like ImageNet will be desirable because if there are no errors, then pruning will be more efficient since the network will not need to remember the exceptions introduced by mislabels.

6.1.1.2 The need for large-scale benchmarks in model compression

As discussed in Chapter 4 Section 4.3 after replicating thousands of experiments as reported in the published literature, the authors of the study ‘ The State of Sparsity in Deep Neural Networks ‘[29] found that community lacked large-scale benchmarks in the field of model compression.

Without proper benchmarks for large scale networks it might be difficult to know the effectiveness of pruning.

6.2 Summary of Chapter 6

In this chapter we considered some technical issues that will effect the efficient deployment of DNNs.

In the first part of the chapter we considered three studies that highlighted some problems that pruned networks face at inference and all three studies cautioned that these issues should be considered when deploying pruned models in mission critical systems.

The researchers at Google Brain of [37] did thousands of large scale experiments and established consistent results across multiple datasets—CIFAR-10, CelebA and ImageNet on widely used pruning and quantization techniques, and model architectures and reconfirmed the results that they obtained previously in [34].

Both studies [34] and [37] also found through extensive experimentation that pruning significantly reduces robustness to image corruptions and natural adversarial images.

The study [39] formulated a set of easy-to-follow guidelines for pruning neural networks in practice (for example when considering mission critical applications) so that problems highlighted in the studies could be minimized.

In the second part of the chapter we considered some known technical issues associated with DNNs and their representative datasets. In addition we discussed whether pruning the DNN will mitigate, worsen or have no effect on the particular technical issue considered.

Chapter 7

Conclusion

In this report we considered the status of research in machine learning/deep learning in general and sparsity/pruning neural networks in particular from various stakeholder's point of view in the research community.

In Chapter 2 we attempted to summarize a recent survey paper [3] which considered mostly sparsity literature published from 2012 to 2020 covered in about 300 research papers. Moreover, it was found in [3] that most researchers have published papers reinventing the already published techniques in early 1980s and that nearly every basic approach has been invented at least twice. They also showed the current trends in publications for instance in sparsity and pruning networks where the researchers mainly concentrate on weight pruning by magnitude as a model compression technique.

In Chapter 3, we tried to summarize ten randomly selected research papers found on Google Scholar related to a search on filter pruning of VGG16 model trained on the CIFER-10 dataset. This was done by tabulating the results reported as related to metrics associated with pruning. The tabulated data enabled us to observe certain trends in result reporting. Many researchers attempted to report their finding with comparison to some other work already published stating that their method helped to gain a less loss in accuracy with a higher rate of pruning with respect to the size and flops than the previous work. We also attempted to organize the tabulated papers in some order according to the size and the reduction in flops of the pruned network when the error in accuracy as a percent change (PC) is in some specific range.

As these research areas are quite new and in a rapidly developing stage based mostly on experimental methods most researchers especially from academia highlighted the problems in the way the experiments are conducted and reported in existing research publications. They, emphasised the need to conduct experiments with empirical rigor and report the results in a systemic manner if the community is to benefit from

the research conducted. In an effort to validate the already reported results; from tech companies, for example at Google AI residency programs researchers replicated experiments thousands of times to check the validity of experimental details in already published literature [29]. Besides that, we ourselves replicated an experiment by using a public code note book available from Google Research which was explained in Chapter 4.

The replication experiment done with a well written ipython note book [23], in Chapter 4 gave us a basic understanding of how pruning algorithms are actually implemented in code from the view provided for a software developer. What should be noted here is that most of the algorithmic code is hidden in the software framework and to get a deeper understanding it might be required to have a programmer's view across the whole computer architecture stack from the application layer down to the device layer.

In Chapter 5 we further explored some researcher's views on published literature. Some researchers gave specific guidelines on how to conduct experiments and suggested that the method of reporting results should be standardised. Some other scholars were concerned about the trends in published research such as; failure to distinguish between explanation and speculation, failure to identify the sources of empirical gains, unnecessary use of mathematics and misuse of language. They recommended training reviewers so that reviewer pool becomes bigger to handle the vast number of publications submitted at international conferences and also suggested guidelines to review such submissions.

Some technical issues related to efficient deployment were considered in Chapter 6. First we considered technical issues specifically related to pruning networks. Of which some researchers were concerned and cautioned about deploying pruned networks on edge devices for mission critical systems. Some other researchers proposed guidelines for pruning networks to be deployed in such situations. Further, we considered few common technical problems of neural networks, associated datasets and computing systems in which they are deployed as reported in the literature. We checked how pruning networks will effect these problems.

Even though we should have a holistic view about technology as engineers, we did not consider environmental, political, legal or ethical

issues related to technologies associated with deep learning in general or pruning in particular in this report.

Finally, it should be noted that the approach used, the papers selected for consideration and their interpretation as given in the report might be biased towards our current view on these topics.

By writing this report, in the process we were able to gain some understanding of the current status of research as reported by some scholars in the community as related to pruning neural networks in terms of how experiments are done, the way results are reported back to the community, the current understanding about pruning methods and some technical issues that might effect efficient deployment of such networks. Probably, the approach taken in compiling the report could be taken as a guide to cover similar topics related to hardware and associated technical issues which were not addressed in the current report.

References

1. Sarker, Iqbal H. "Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions." SN Computer Science 2, no. 6 (2021): 1-20.

2. Guo, Cong, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. "Accelerating sparse dnn models without hardware-support via tile-wise sparsity." In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-15. IEEE, 2020.

3. Torsten Hoefler, Dan Alistarh, Tan Ben-Nun, Nikoli Dryden, Alexandra Peste: Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks Journal of Machine Learning Research. Vol 22, Nr. 241, pages 1-124, Sep. 2021.

4. R. Reed, "Pruning algorithms-a survey," in IEEE Transactions on Neural Networks, vol. 4, no. 5, pp. 740-747, Sept. 1993, doi: 10.1109/72.248452.

5. Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. arXiv preprint arXiv:1608.08710, 2016.

6. Han, S., Pool, J., Tran, J., & Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. arXiv preprint arXiv:1506.02626.

7. Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. Learning efficient convolutional networks through network

slimming. In Proceedings of the IEEE International Conference on Computer Vision, pp. 2736–2744, 2017.

8. Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In Advances in neural information processing systems, pages 598–605, 1990.

9. Tessier, H, *Neural Network Pruning 101*, September 8 2021, accessed 16 December 2021, <https://towardsdatascience.com/neural-network-pruning-101-af816aaea61>.

10. Blalock, Davis, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. "What is the state of neural network pruning?". *arXiv preprint arXiv:2003.03033* (2020).

11. *Sparsity in Neural Networks - Advancing Understanding and Practice* 2021, online workshop, Google sites, accessed 10 December 2021, < https://sites.google.com/view/sparsity-workshop-2021/home >.

12. Henderson, Peter, and Emma Brunskill. "Distilling information from a flood: A possibility for the use of meta-analysis and systematic review in machine learning research." *arXiv preprint arXiv:1812.01074* (2018).

13. Wikipedia. “Heuristic.” Last modified November 23, 2021. https://en.wikipedia.org/wiki/Heuristic.

14. Zhao, Chenglong, Bingbing Ni, Jian Zhang, Qiwei Zhao, Wenjun Zhang, and Qi Tian. "Variational convolutional neural network pruning." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2780-2789. 2019.

15. Huang, Qiangui, Kevin Zhou, Suya You, and Ulrich Neumann. "Learning to prune filters in convolutional neural networks." In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 709-718. IEEE, 2018.

16. He, Yang, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. "Filter pruning via geometric median for deep convolutional neural networks acceleration." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4340-4349. 2019.

17. Lin, Shaohui, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. "Towards optimal structured cnn pruning via generative adversarial learning." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2790-2799. 2019.

18. Lin, Mingbao, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. "Hrank: Filter pruning using high-rank feature map." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1529-1538. 2020.

19. Al Dallal, Ahmed. "Filter Pruning in Convolutional Neural Networks Using Structural Similarity Based K-Means." (2021).

20. Ayinde, Babajide O., and Jacek M. Zurada. "Building efficient convnets using redundant feature pruning." *arXiv preprint arXiv:1802.07653* (2018).

21. Li, Yawei, Shuhang Gu, Luc Van Gool, and Radu Timofte. "Learning filter basis for convolutional neural network

compression." In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 5623-5632. 2019.

22. Carballo, Marina Villalba, and Byeong Kil Lee. "Accuracy-aware Structured Filter Pruning for Deep Neural Networks." In *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 679-682. IEEE, 2020.

23. Keras Model pruning Exploration, accessed 2 January 2022, <https://colab.research.google.com/github/matthew-mcateer/Keras_pruning/blob/master/Model_pruning_exploration.ipynb>.

24. Liebenwein, Lucas, Cenk Baykal, Harry Lang, Dan Feldman, and Daniela Rus. "Provable filter pruning for efficient neural networks." *arXiv preprint arXiv:1911.07412* (2019).

25. Ali Rahimi - NIPS 2017 Test-of-Time Award presentation, 2017, accessed 10 December 2021, <https://www.youtube.com/watch?v=ORHFOnaEzPc>.

26. Lipton, Z. C., Steinhardt, J. Troubling trends in machine learning scholarship. Published in acmqueue, Volume 17, issue 1, April 24, 2019, accessed 16 December 2021, https://queue.acm.org/detail.cfm?id=3328534

27. Sodhani, S., Jaiswal, M. S., Baker, L., Sinha, K., Shneider, C., Henderson, P., ... & Lowe, R. (2020). Ideas for Improving the Field of Machine Learning: Summarizing Discussion from the NeurIPS 2019 Retrospectives Workshop. *arXiv preprint arXiv:2007.10546*. (2020).

28. Sculley, D., Snoek, J., Wiltschko, A., Rahimi, A.: Winner's curse? on pace, progress, and empirical rigor. In: International

Conference on Learning Representations Workshop track (2018), published online: iclr.cc

29. Gale, Trevor, Erich Elsen, and Sara Hooker. "The state of sparsity in deep neural networks." *arXiv preprint arXiv:1902.09574* (2019).

30. Liu, Zhuang, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. "Rethinking the value of network pruning." *arXiv preprint arXiv:1810.05270* (2018).

31. Frankle, Jonathan, and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks." *arXiv preprint arXiv:1803.03635* (2018).

32. Molchanov, Dmitry, Arsenii Ashukha, and Dmitry Vetrov. "Variational dropout sparsifies deep neural networks." In *International Conference on Machine Learning*, pp. 2498-2507. PMLR, 2017.

33. Louizos, Christos, Max Welling, and Diederik P. Kingma. "Learning sparse neural networks through $ L_0 $ regularization." *arXiv preprint arXiv:1712.01312* (2017).

34. Hooker, Sara, Nyalleng Moorosi, Gregory Clark, Samy Bengio, and Emily Denton. "Characterising bias in compressed models." *arXiv preprint arXiv:2010.03058* (2020).

35. Dan Hendrycks and Thomas Dietterich. 2019. Benchmarking neural network robustness to common corruptions and perturbations. In Proceedings of the Seventh International Conference on Learning Representations. arXiv:cs.LG/1903.12261

36. Hendrycks, D., Zhao, K., Basart, S., Steinhardt, J., & Song, D. (2021). Natural adversarial examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 15262-15271).

37. Hooker, Sara, Aaron Courville, Gregory Clark, Yann Dauphin, and Andrea Frome. "What do compressed deep neural networks forget?." *arXiv preprint arXiv:1911.05248* (2019).

38. *Selective Brain Damage: Measuring the Disparate Impact of Model Compression*, n.d., github.io, accessed 10 December 2021, <https://weightpruningdamage.github.io/> .

39. Liebenwein, Lucas, Cenk Baykal, Brandon Carter, David Gifford, and Daniela Rus. "Lost in pruning: The effects of pruning neural networks beyond test accuracy." *arXiv preprint arXiv:2103.03014* (2021).

40. Heaven, Douglas. "Why deep-learning AIs are so easy to fool." (2019) Nature Publishing Group: 163-166.

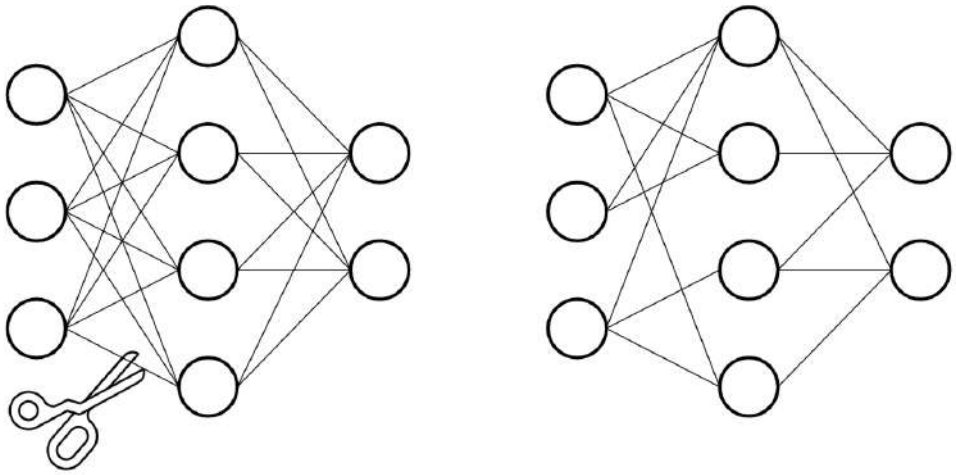
41. Northcutt, Curtis G., Anish Athalye, and Jonas Mueller. "Pervasive label errors in test sets destabilize machine learning benchmarks." *arXiv preprint arXiv:2103.14749* (2021).

APPENDIX

The execution of the ipython notebook ‘Keras Model pruning Exploration’ available from Google Research saved as a pdf file

Keras Model pruning Exploration

There are multiple ways of optimizing neural-network-based machine learning algorithms. One of these optimizations is the removal of connections between neurons and layers, and thus speeding up computation by reducing the overal number of parameters.



Networks generally look like the one on the left: every neuron in the layer below has a connection to the layer above; but this means that we have to multiply a lot of floats together. Ideally, we'd only connect each neuron to a few others and save on doing some of the multiplications; this is called a "sparse" network.

Given a layer of a neural network $ReLU(xW)$ are two well-known ways to prune it:

- **Weight pruning:** set individual weights in the weight matrix to zero. This corresponds to deleting connections as in the figure above.
    - Here, to achieve sparsity of k we rank the individual weights in weight matrix W according to their magnitude (absolute value) $|wi, j|$, and then set to zero the smallest k.
- **Unit/Neuron pruning:** set entire columns to zero in the weight matrix to zero, in effect deleting the corresponding output neuron.
    - Here to achieve sparsity of k we rank the columns of a weight matrix according to their L2-norm $|w| = \sqrt{\sum_{i=1}^{N}(x_i)^2}$ and delete the smallest k.

Naturally, as you increase the sparsity and delete more of the network, the task performance will progressively degrade. Here, we will be implementing **both weight and unit pruning** and compare the performance across both the MNIST and FMNIST datasets.

Dependency Installations

```
!mkdir images
!mkdir models


# Selecting Tensorflow version v2 (the command is relevant for Colab only).
%tensorflow_version 2.x


# Load the TensorBoard notebook extension.
# %reload_ext tensorboard
%load_ext tensorboard


import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sn
import numpy as np
import pandas as pd
import math
import datetime
import platform

print('Python version:', platform.python_version())
print('Tensorflow version:', tf.__version__)
print('Keras version:', tf.keras.__version__)
```

```
Python version: 3.7.12
Tensorflow version: 2.7.0
Keras version: 2.7.0
```

```
#! pip uninstall -y tensorflow
#! pip uninstall -y tf-nightly
#! pip install -U tf-nightly-gpu
! pip install tensorflow-model-optimization
!pip install tensorflow as tf
import math
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.axes as axes
import numpy as np
import os
import pandas as pd
import seaborn as sns
import tempfile
import tensorboard
#import tensorflow as tf
import keras
```

```
import timeit
import zipfile
#import keras
from IPython.core.pylabtools import figsize
from numpy import linalg as LA
from tensorflow.keras.models import load_model
from tensorflow_model_optimization.sparsity import keras as sparsity

#tf.enable_eager_execution()
plt.style.use('fivethirtyeight')
sns.set_context('notebook')
pd.set_option('display.max_rows', 30)
np.random.seed(1337)
%config InlineBackend.figure_format = 'retina'
%load_ext tensorboard
```

```
Collecting tensorflow-model-optimization
  Downloading tensorflow_model_optimization-0.7.0-py2.py3-none-any.whl (213 kB)
     |████████████████████████████████| 213 kB 5.0 MB/s
Requirement already satisfied: dm-tree~=0.1.1 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: numpy~=1.14 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: six~=1.10 in /usr/local/lib/python3.7/dist-packages (fro
Installing collected packages: tensorflow-model-optimization
Successfully installed tensorflow-model-optimization-0.7.0
Requirement already satisfied: tensorflow in /usr/local/lib/python3.7/dist-packages (2.
ERROR: Could not find a version that satisfies the requirement as (from versions: none)
ERROR: No matching distribution found for as
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

Dataset

We'll set up a function for loading both MNIST and FMNIST

```
def load_dataset(dataset='mnist'):
    """
    Loads and preprocesses the data for this task.
    Args:
      dataset: the name of the dataset to be used for this classification task.
        (mnist | fmnist)
    Returns:
      x_train: Features for training data
      x_test: Features for test data
      y_train: Labels for training data
      y_test: Labels for test data
      num_classes: Number of classes for the dataset
    """
    # input image dimensions (equal for both MNIST and FMNIST)
    img_rows, img_cols = 28, 28
```

```
    if dataset=='mnist':
        # Number of classes in the data
        num_classes = 10

        # the data, shuffled and split between train and test sets
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

    elif dataset=='fmnist':
        # Number of classes in the data
        num_classes = 10

        # the data, shuffled and split between train and test sets
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()

    else:
        print('dataset name does not match available options \n( mnist | keras )')

    x_train = x_train.reshape(x_train.shape[0], img_rows*img_cols)
    x_test = x_test.reshape(x_test.shape[0], img_rows*img_cols)
    input_shape = (img_rows*img_cols*1,)

    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train /= 255
    x_test /= 255
    print('x_train shape:', x_train.shape)
    print(x_train.shape[0], 'train samples')
    print(x_test.shape[0], 'test samples')

    # convert class vectors to binary class matrices
    y_train = tf.keras.utils.to_categorical(y_train, num_classes)
    y_test = tf.keras.utils.to_categorical(y_test, num_classes)

    return x_train, x_test, y_train, y_test, num_classes, input_shape

# We will load separate tensors for the MNIST and FMNIST data.

mnist_x_train, mnist_x_test, mnist_y_train, mnist_y_test, num_classes, input_shape = load_dat

fmnist_x_train, fmnist_x_test, fmnist_y_train, fmnist_y_test, num_classes, input_shape = loac
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
x_train shape: (60000, 784)
60000 train samples
10000 test samples
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train
32768/29515 [=================================] - 0s 0us/step
40960/29515 [=========================================] - 0s 0us/step
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train
26427392/26421880 [==============================] - 0s 0us/step
26435584/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-
16384/5148 [================================================================================
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-
4423680/4422102 [==============================] - 0s 0us/step
4431872/4422102 [==============================] - 0s 0us/step
x_train shape: (60000, 784)
60000 train samples
10000 test samples
```

Training a keras model without pruning

We will construct a $ReLU$-activated neural network with four hidden layers. These layers will be dense, fully-connected layers with sizes 1000, 1000, 500, & 200.We'll also have have a fifth layer for the output logits, which we will have 10 (**Note:** Since these connect directly to the output layer, these will be spared from any and all pruning).

For the sake of simplicity, we will also omit Dropout layers, Convolutional layers, Batch Normalization Layers, and Avg Pooling Layers.

```
l = tf.keras.layers

def build_model_arch(input_shape, num_classes, sparsity=0.0):
    """
    Builds the model architecture
    Args:
      input_shape: The tuple describing the input shape
      num_classes: how many classes the data labels belong to
      sparsity: For compressing already sparse models, how much sparsity was used
    Returns:
      model: an un-compiled TF.Keras model with 4 hidden
        dense layers with shapes [1000, 1000, 500, 200]
    """

    model = tf.keras.Sequential()

    model.add(l.Dense(int(1000-(1000*sparsity)), activation='relu',
                      input_shape=input_shape)),
    model.add(l.Dense(int(1000-(1000*sparsity)), activation='relu'))
    model.add(l.Dense(int(500-(500*sparsity)), activation='relu'))
    model.add(l.Dense(int(200-(200*sparsity)), activation='relu'))
    model.add(l.Dense(num_classes, activation='softmax'))

    return model
```

```
# The architectures are the same, but we are initializing 2 different sequential
# models. One is for MNIST, and one is for FMNIST

mnist_model_base = build_model_arch(input_shape, num_classes)
fmnist_model_base = build_model_arch(input_shape, num_classes)
```

Load TensorBoard to monitor the training process

```
logdir = tempfile.mkdtemp()
print('Writing training logs to ' + logdir)
```

```
Writing training logs to /tmp/tmpk8f1gxgn
```

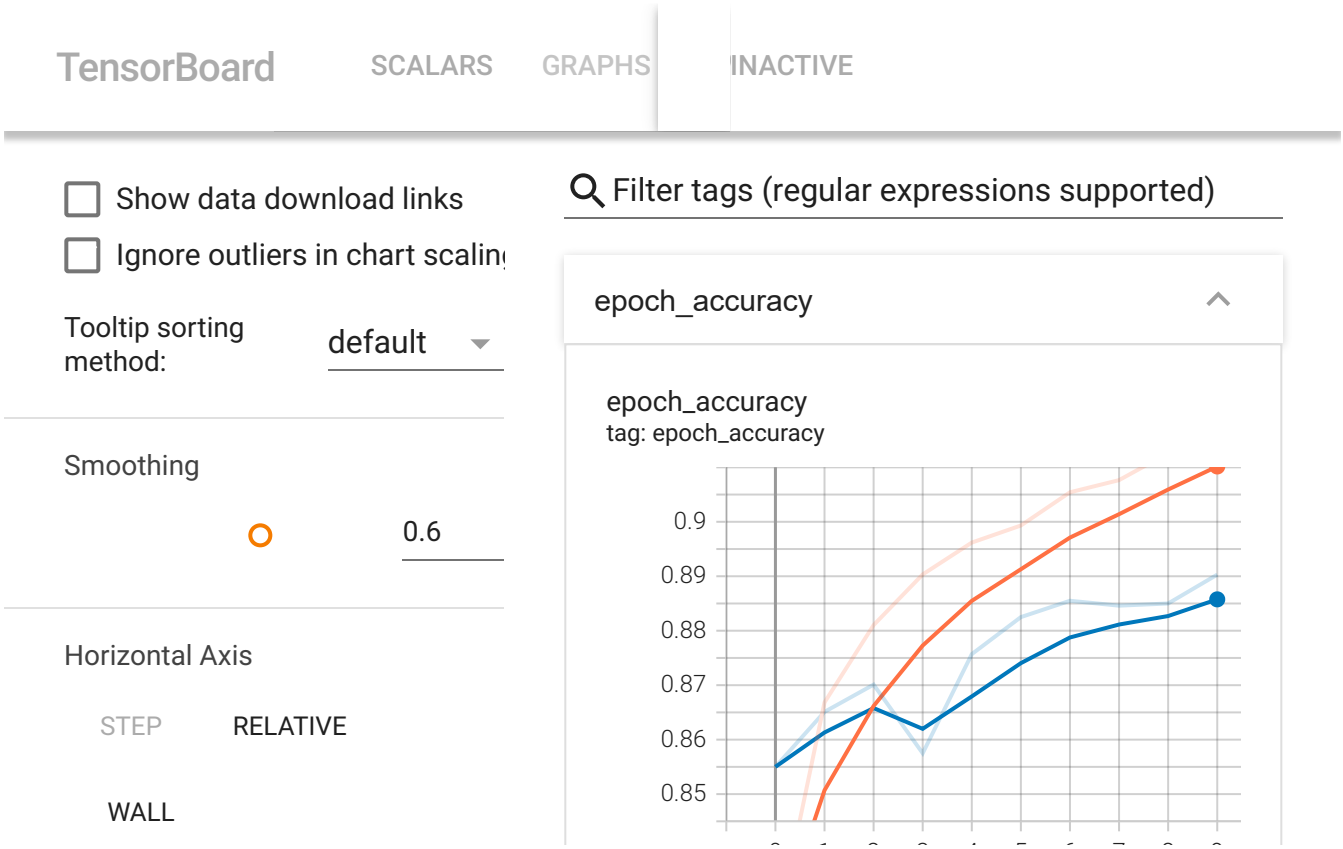
Training the model

```
def make_nosparse_model(model, x_train, y_train, batch_size,
                        epochs, x_test, y_test):
    """
    Training our original model, pre-pruning
    Args:
      model: Uncompiled Keras model
      x_train: Features for training data
      y_train: Labels for training data
      batch_size: Batch size for training
      epochs: Number of epochs for training
      x_test: Features for test data
      y_test: Labels for test data
    Returns:
      model: compiled model
      score: List of both final test loss and final test accuracy
    """
    callbacks = [tf.keras.callbacks.TensorBoard(log_dir=logdir, profile_batch=0)]

    model.compile(
        loss=tf.keras.losses.categorical_crossentropy,
        optimizer='adam',
        metrics=['accuracy'])

    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              verbose=1,
              callbacks=callbacks,
              validation_data=(x_test, y_test))
    score = model.evaluate(x_test, y_test, verbose=0)
    print('Test loss:', score[0])
```

```
    print('Test accuracy:', score[1])

    return model, score

batch_size = 128
epochs = 10

mnist_model, mnist_score = make_nosparse_model(mnist_model_base,
                                               mnist_x_train,
                                               mnist_y_train,
                                               batch_size,
                                               epochs,
                                               mnist_x_test,
                                               mnist_y_test)
print(mnist_model.summary())

fmnist_model, fmnist_score = make_nosparse_model(fmnist_model_base,
                                                 fmnist_x_train,
                                                 fmnist_y_train,
                                                 batch_size,
                                                 epochs,
                                                 fmnist_x_test,
                                                 fmnist_y_test)

print(fmnist_model.summary())

      dense_1 (Dense)             (None, 1000)              1001000

      dense_2 (Dense)             (None, 500)               500500

      dense_3 (Dense)             (None, 200)               100200

      dense_4 (Dense)             (None, 10)                2010

     =================================================================
     Total params: 2,388,710
     Trainable params: 2,388,710
     Non-trainable params: 0
     _________________________________________________________________
     None
     Epoch 1/10
     469/469 [==============================] - 26s 54ms/step - loss: 0.4789 - accuracy:
     Epoch 2/10
     469/469 [==============================] - 26s 54ms/step - loss: 0.3608 - accuracy:
     Epoch 3/10
     469/469 [==============================] - 26s 55ms/step - loss: 0.3227 - accuracy:
     Epoch 4/10
     469/469 [==============================] - 26s 55ms/step - loss: 0.2960 - accuracy:
     Epoch 5/10
     469/469 [==============================] - 26s 55ms/step - loss: 0.2788 - accuracy:
     Epoch 6/10
     469/469 [==============================] - 26s 56ms/step - loss: 0.2690 - accuracy:
     Epoch 7/10
     469/469 [==============================] - 28s 59ms/step - loss: 0.2514 - accuracy:
```

```
Epoch 8/10
469/469 [==============================] - 26s 55ms/step - loss: 0.2443 - accuracy:
Epoch 9/10
469/469 [==============================] - 26s 55ms/step - loss: 0.2316 - accuracy:
Epoch 10/10
469/469 [==============================] - 25s 54ms/step - loss: 0.2206 - accuracy:
Test loss: 0.32523465156555176
Test accuracy: 0.8902999758720398
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense_5 (Dense)              (None, 1000)              785000

 dense_6 (Dense)              (None, 1000)              1001000

 dense_7 (Dense)              (None, 500)               500500

 dense_8 (Dense)              (None, 200)               100200

 dense_9 (Dense)              (None, 10)                2010

=================================================================
Total params: 2,388,710
Trainable params: 2,388,710
Non-trainable params: 0
_________________________________________________________________
None
```

```
%tensorboard --logdir={logdir}
```



Pruning

Now that the model is trained, we are going to prune away (set to zero) weigths of the trained model. Two pruning methods will be used for this:

- **Weight pruning:** set individual weights in the weight matrix to zero. This corresponds to deleting connections as in the figure above.
    - Here, to achieve sparsity of k we rank the individual weights in weight matrix W according to their magnitude (absolute value) $|w_{i,j}|$, and then set to zero the smallest k.
- **Unit/Neuron pruning:** set entire columns to zero in the weight matrix to zero, in effect deleting the corresponding output neuron.
    - Here to achieve sparsity of k we rank the columns of a weight matrix according to their L2-norm $|w| = \sqrt{\sum_{i=1}^{N}(x_i)^2}$ and delete the smallest k.

The $k\%$ of weights using weight and unit pruning for k in $[0, 25, 50, 60, 70, 80, 90, 95, 97, 99]$. Neither of these pruning methods will prune the weights leading to the softmax layer.

Pruning the layers

```
def weight_prune_dense_layer(k_weights, b_weights, k_sparsity):
    """
    Takes in matrices of kernel and bias weights (for a dense
      layer) and returns the unit-pruned versions of each
    Args:
      k_weights: 2D matrix of the
      b_weights: 1D matrix of the biases of a dense layer
      k_sparsity: percentage of weights to set to 0
    Returns:
      kernel_weights: sparse matrix with same shape as the original
        kernel weight matrix
      bias_weights: sparse array with same shape as the original
        bias array
    """
    # Copy the kernel weights and get ranked indeces of the abs
    kernel_weights = np.copy(k_weights)
    ind = np.unravel_index(
        np.argsort(
            np.abs(kernel_weights),
            axis=None),
        kernel_weights.shape)

    # Number of indexes to set to 0
    cutoff = int(len(ind[0])*k_sparsity)
    # The indexes in the 2D kernel weight matrix to set to 0
    sparse_cutoff_inds = (ind[0][0:cutoff], ind[1][0:cutoff])
    kernel_weights[sparse_cutoff_inds] = 0.

    # Copy the bias weights and get ranked indeces of the abs
    bias_weights = np.copy(b_weights)
    ind = np.unravel_index(
        np.argsort(
            np.abs(bias_weights),
            axis=None),
        bias_weights.shape)

    # Number of indexes to set to 0
    cutoff = int(len(ind[0])*k_sparsity)
    # The indexes in the 1D bias weight matrix to set to 0
    sparse_cutoff_inds = (ind[0][0:cutoff])
    bias_weights[sparse_cutoff_inds] = 0.

    return kernel_weights, bias_weights



def unit_prune_dense_layer(k_weights, b_weights, k_sparsity):
    """
    Takes in matrices of kernel and bias weights (for a dense
      layer) and returns the unit-pruned versions of each
    Args:
```

```
      k_weights: 2D matrix of the
      b_weights: 1D matrix of the biases of a dense layer
      k_sparsity: percentage of weights to set to 0
    Returns:
      kernel_weights: sparse matrix with same shape as the original
        kernel weight matrix
      bias_weights: sparse array with same shape as the original
        bias array
    """

    # Copy the kernel weights and get ranked indeces of the
    # column-wise L2 Norms
    kernel_weights = np.copy(k_weights)
    ind = np.argsort(LA.norm(kernel_weights, axis=0))

    # Number of indexes to set to 0
    cutoff = int(len(ind)*k_sparsity)
    # The indexes in the 2D kernel weight matrix to set to 0
    sparse_cutoff_inds = ind[0:cutoff]
    kernel_weights[:,sparse_cutoff_inds] = 0.

    # Copy the bias weights and get ranked indeces of the abs
    bias_weights = np.copy(b_weights)
    # The indexes in the 1D bias weight matrix to set to 0
    # Equal to the indexes of the columns that were removed in this case
    #sparse_cutoff_inds
    bias_weights[sparse_cutoff_inds] = 0.

    return kernel_weights, bias_weights
```

Pruning across an entire model

```
def sparsify_model(model, x_test, y_test, k_sparsity, pruning='weight'):
    """
    Takes in a model made of dense layers and prunes the weights
    Args:
      model: Keras model
      k_sparsity: target sparsity of the model
    Returns:
      sparse_model: sparsified copy of the previous model
    """
    # Copying a temporary sparse model from our original
    sparse_model = tf.keras.models.clone_model(model)
    sparse_model.set_weights(model.get_weights())

    # Getting a list of the names of each component (w + b) of each layer
    names = [weight.name for layer in sparse_model.layers for weight in layer.weights]
    # Getting the list of the weights for each component (w + b) of each layer
```

```
    weights = sparse_model.get_weights()

    # Initializing list that will contain the new sparse weights
    newWeightList = []

    # Iterate over all but the final 2 layers (the softmax)
    for i in range(0, len(weights)-2, 2):

        if pruning=='weight':
            kernel_weights, bias_weights = weight_prune_dense_layer(weights[i],
                                                                    weights[i+1],
                                                                    k_sparsity)
        elif pruning=='unit':
            kernel_weights, bias_weights = unit_prune_dense_layer(weights[i],
                                                                  weights[i+1],
                                                                  k_sparsity)
        else:
            print('does not match available pruning methods ( weight | unit )')

        # Append the new weight list with our sparsified kernel weights
        newWeightList.append(kernel_weights)

        # Append the new weight list with our sparsified bias weights
        newWeightList.append(bias_weights)

    # Adding the unchanged weights of the final 2 layers
    for i in range(len(weights)-2, len(weights)):
        unmodified_weight = np.copy(weights[i])
        newWeightList.append(unmodified_weight)

    # Setting the weights of our model to the new ones
    sparse_model.set_weights(newWeightList)

    # Re-compiling the Keras model (necessary for using `evaluate()`)
    sparse_model.compile(
        loss=tf.keras.losses.categorical_crossentropy,
        optimizer='adam',
        metrics=['accuracy'])

    # Printing the the associated loss & Accuracy for the k% sparsity
    score = sparse_model.evaluate(x_test, y_test, verbose=0)
    print('k% weight sparsity: ', k_sparsity,
          '\tTest loss: {:07.5f}'.format(score[0]),
          '\tTest accuracy: {:05.2f} %%'.format(score[1]*100.))

    return sparse_model, score
```

Weight-and-unit pruning across all $k\%$ sparsities

Loop over each k% sparsity in the list

```
#@title Loop over each k% sparsity in the list

# list of sparsities
k_sparsities = [0.0, 0.25, 0.50, 0.60, 0.70, 0.80, 0

# The empty lists where we will store our training r
mnist_model_loss_weight = []
mnist_model_accs_weight = []
mnist_model_loss_unit = []
mnist_model_accs_unit = []
fmnist_model_loss_weight = []
fmnist_model_accs_weight = []
fmnist_model_loss_unit = []
fmnist_model_accs_unit = []

dataset = 'mnist'
pruning = 'weight'
print('\n MNIST Weight-pruning\n')
for k_sparsity in k_sparsities:
    sparse_model, score = sparsify_model(mnist_model
                                         y_test=mnis
                                         k_sparsity=
                                         pruning=pru
    mnist_model_loss_weight.append(score[0])
    mnist_model_accs_weight.append(score[1])

    # Save entire model to an H5 file
    sparse_model.save('models/sparse_{}-model_k-{}_{
    del sparse_model


pruning='unit'
print('\n MNIST Unit-pruning\n')
for k_sparsity in k_sparsities:
    sparse_model, score = sparsify_model(mnist_model
                                         y_test=mnis
                                         k_sparsity=
                                         pruning=pru
    mnist_model_loss_unit.append(score[0])
    mnist_model_accs_unit.append(score[1])

    # Save entire model to an H5 file
    sparse_model.save('models/sparse_{}-model_k-{}_{
    del sparse_model

dataset = 'fmnist'
pruning = 'weight'
print('\n FMNIST Weight-pruning\n')
for k_sparsity in k_sparsities:
    sparse_model, score = sparsify_model(fmnist_mode
```

```
                                        y_test=fmni
                                        k_sparsity=
                                        pruning=pru
    fmnist_model_loss_weight.append(score[0])
    fmnist_model_accs_weight.append(score[1])

    # Save entire model to an H5 file
    sparse_model.save('models/sparse_{}-model_k-{}_{
    del sparse_model


pruning='unit'
print('\n FMNIST Unit-pruning\n')
for k_sparsity in k_sparsities:
    sparse_model, score = sparsify_model(fmnist_mode
                                        y_test=fmni
                                        k_sparsity=
                                        pruning=pru
    fmnist_model_loss_unit.append(score[0])
    fmnist_model_accs_unit.append(score[1])

    # Save entire model to an H5 file
    sparse_model.save('models/sparse_{}-model_k-{}_{
    del sparse_model
```

```
  MNIST Weight-pruning

 k% weight sparsity:  0.0        Test loss: 0.07312      Test accuracy: 98.20 %%
 k% weight sparsity:  0.25       Test loss: 0.07199      Test accuracy: 98.21 %%
 k% weight sparsity:  0.5        Test loss: 0.06560      Test accuracy: 98.25 %%
 k% weight sparsity:  0.6        Test loss: 0.06628      Test accuracy: 98.16 %%
 k% weight sparsity:  0.7        Test loss: 0.08017      Test accuracy: 98.06 %%
 k% weight sparsity:  0.8        Test loss: 0.21426      Test accuracy: 97.55 %%
 k% weight sparsity:  0.9        Test loss: 1.24332      Test accuracy: 88.60 %%
 k% weight sparsity:  0.95       Test loss: 2.10263      Test accuracy: 32.43 %%
 k% weight sparsity:  0.97       Test loss: 2.25835      Test accuracy: 09.91 %%
 k% weight sparsity:  0.99       Test loss: 2.30494      Test accuracy: 09.74 %%

  MNIST Unit-pruning

 k% weight sparsity:  0.0        Test loss: 0.07312      Test accuracy: 98.20 %%
 k% weight sparsity:  0.25       Test loss: 0.06709      Test accuracy: 98.19 %%
 k% weight sparsity:  0.5        Test loss: 0.13468      Test accuracy: 97.60 %%
 k% weight sparsity:  0.6        Test loss: 0.36788      Test accuracy: 97.27 %%
 k% weight sparsity:  0.7        Test loss: 0.97559      Test accuracy: 96.65 %%
 k% weight sparsity:  0.8        Test loss: 1.85384      Test accuracy: 52.28 %%
 k% weight sparsity:  0.9        Test loss: 2.25347      Test accuracy: 17.24 %%
 k% weight sparsity:  0.95       Test loss: 2.30439      Test accuracy: 09.74 %%
 k% weight sparsity:  0.97       Test loss: 2.30591      Test accuracy: 09.74 %%
 k% weight sparsity:  0.99       Test loss: 2.30591      Test accuracy: 09.74 %%
```

```
 FMNIST Weight-pruning

k% weight sparsity:  0.0       Test loss: 0.32523      Test accuracy: 89.03 %%
k% weight sparsity:  0.25      Test loss: 0.32418      Test accuracy: 89.14 %%
k% weight sparsity:  0.5       Test loss: 0.31160      Test accuracy: 89.15 %%
k% weight sparsity:  0.6       Test loss: 0.30990      Test accuracy: 88.93 %%
k% weight sparsity:  0.7       Test loss: 0.33070      Test accuracy: 88.53 %%
k% weight sparsity:  0.8       Test loss: 0.43838      Test accuracy: 86.86 %%
k% weight sparsity:  0.9       Test loss: 1.02177      Test accuracy: 74.18 %%
k% weight sparsity:  0.95      Test loss: 1.87324      Test accuracy: 44.08 %%
k% weight sparsity:  0.97      Test loss: 2.17840      Test accuracy: 22.28 %%
k% weight sparsity:  0.99      Test loss: 2.29463      Test accuracy: 10.88 %%

 FMNIST Unit-pruning

k% weight sparsity:  0.0       Test loss: 0.32523      Test accuracy: 89.03 %%
k% weight sparsity:  0.25      Test loss: 0.31479      Test accuracy: 89.04 %%
k% weight sparsity:  0.5       Test loss: 0.38848      Test accuracy: 87.70 %%
k% weight sparsity:  0.6       Test loss: 0.55010      Test accuracy: 87.46 %%
k% weight sparsity:  0.7       Test loss: 0.95942      Test accuracy: 81.72 %%
k% weight sparsity:  0.8       Test loss: 1.57718      Test accuracy: 62.26 %%
k% weight sparsity:  0.9       Test loss: 2.17769      Test accuracy: 18.96 %%
k% weight sparsity:  0.95      Test loss: 2.29922      Test accuracy: 10.31 %%
k% weight sparsity:  0.97      Test loss: 2.30454      Test accuracy: 10.00 %%
k% weight sparsity:  0.99      Test loss: 2.30598      Test accuracy: 10.00 %%
```

Organizing these results into one Pandas DataFrame

```
# Convert the lists to numpy arrays
k_sparsities = np.asarray(k_sparsities)
mnist_model_loss_weight = np.asarray(mnist_model_loss_weight)
mnist_model_accs_weight = np.asarray(mnist_model_accs_weight)
mnist_model_loss_unit = np.asarray(mnist_model_loss_unit)
mnist_model_accs_unit = np.asarray(mnist_model_accs_unit)
fmnist_model_loss_weight = np.asarray(fmnist_model_loss_weight)
fmnist_model_accs_weight = np.asarray(fmnist_model_accs_weight)
fmnist_model_loss_unit = np.asarray(fmnist_model_loss_unit)
fmnist_model_accs_unit = np.asarray(fmnist_model_accs_unit)

# Stack the arrays so they can be used in the DataFrame
sparsity_data = np.stack([k_sparsities,
                          mnist_model_loss_weight,
                          mnist_model_accs_weight,
                          mnist_model_loss_unit,
                          mnist_model_accs_unit,
                          fmnist_model_loss_weight,
                          fmnist_model_accs_weight,
                          fmnist_model_loss_unit,
                          fmnist_model_accs_unit])

# Defining the Pandas DataFrame
sparsity_summary = pd.DataFrame(data=sparsity_data.T,     # values
```

```
sparsity_summary = pd.DataFrame(data=sparsity_data.T,    # values
                                columns=['k_sparsity',   # Column names
                                         'mnist_loss_weight',
                                         'mnist_acc_weight',
                                         'mnist_loss_unit',
                                         'mnist_acc_unit',
                                         'fmnist_loss_weight',
                                         'fmnist_acc_weight',
                                         'fmnist_loss_unit',
                                         'fmnist_acc_unit'])
sparsity_summary.to_csv('sparsity_summary.csv')
sparsity_summary
```

	k_sparsity	mnist_loss_weight	mnist_acc_weight	mnist_loss_unit	mnist_acc_unit	fm
0	0.00	0.073117	0.9820	0.073117	0.9820	
1	0.25	0.071988	0.9821	0.067093	0.9819	
2	0.50	0.065600	0.9825	0.134678	0.9760	
3	0.60	0.066279	0.9816	0.367883	0.9727	
4	0.70	0.080165	0.9806	0.975589	0.9665	
5	0.80	0.214255	0.9755	1.853837	0.5228	
6	0.90	1.243321	0.8860	2.253467	0.1724	
7	0.95	2.102632	0.3243	2.304388	0.0974	
8	0.97	2.258347	0.0991	2.305907	0.0974	
9	0.99	2.304936	0.0974	2.305907	0.0974	

Plotting

Visualizing Sparsity

```
def visualize_model_weights(sparse_model):
    """
    Visualize the weights of the layers of the sparse model.
    For weights with values of 0, they will be represented by the color white
    Args:
      sparse_model: a TF.Keras model
    """
    weights = sparse_model.get_weights()
```

```
    names = [weight.name for layer in sparse_model.layers for weight in layer.weights]

    my_cmap = matplotlib.cm.get_cmap('rainbow')
    my_cmap.set_under('w')

    # Iterate over all the weight matrices in the model and visualize them
    for i in range(len(weights)):
        weight_matrix = weights[i]
        layer_name = names[i]
        if weight_matrix.ndim == 1: # If Bias or softmax
            weight_matrix = np.resize(weight_matrix,
                                      (1,weight_matrix.size))
            plt.imshow(np.abs(weight_matrix),
                       interpolation='none',
                       aspect = "auto",
                       cmap=my_cmap,
                       vmin=1e-26); # lower bound is set close to but not at 0
            plt.colorbar()
            plt.title(layer_name)
            plt.show()
        else: # all other 2D matrices
            plt.imshow(np.abs(weight_matrix),
                       interpolation='none',
                       cmap=my_cmap,
                       vmin=1e-26);
            plt.colorbar()
            plt.title(layer_name)
            plt.show()
```

```
#@title Dense Layer visualization { run: "auto"

#@markdown In Google Colab, this file becomes inter
#@markdown
#@markdown All weights with values of `0.0` will be
#@markdown
#@markdown Which dataset?
dataset = 'mnist' #@param ['mnist', 'fmnist']
#@markdown k sparsity
sparsity = "0.5" #@param ['0.0', '0.25', '0.5', '0.6
#@markdown Pruning method.
pruning = 'unit' #@param ['unit', 'weight']

sparse_model = load_model('models/sparse_{}-model_k-

visualize_model_weights(sparse_model)
```

Dense Layer visualization

In Google Colab, this file becomes interactive, and you can select the sparse model you want to retrieve.

All weights with values of `0.0` will be color-coded weight. 1D Bias layers will be auto-scaled to the dimensions of the 2D plots.

Which dataset?

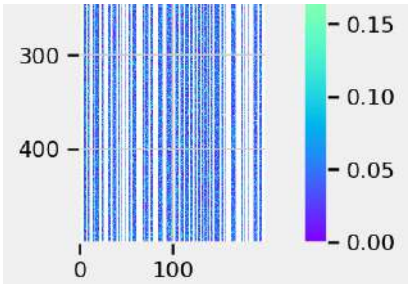
dataset: mnist

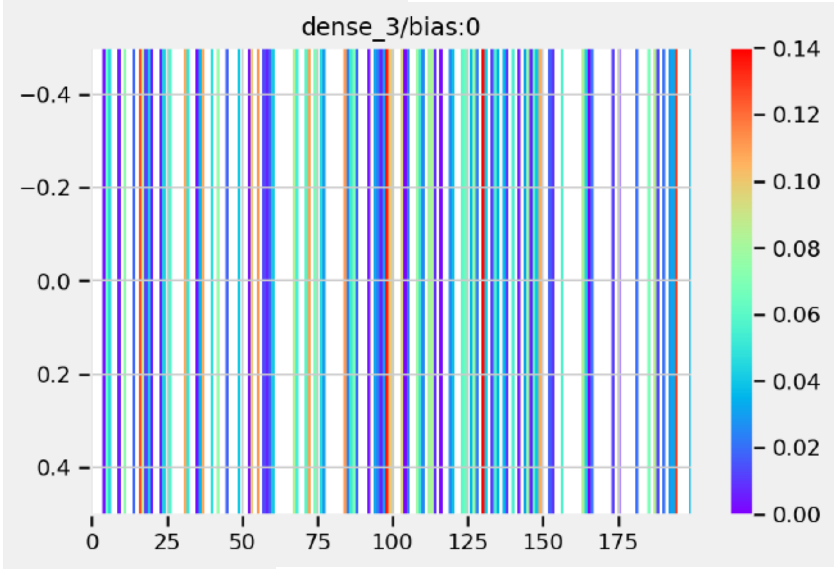
k sparsity

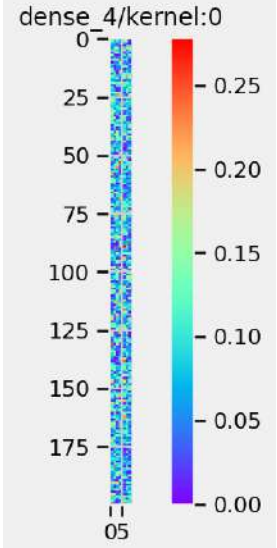
sparsity: 0.5

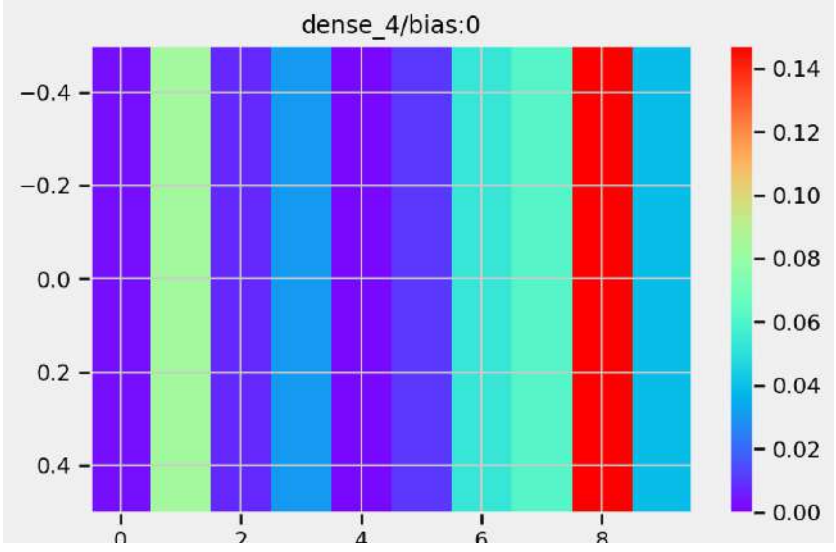
Pruning method.

pruning: unit


300
400
0
100
0.15
0.10
0.05
0.00


dense_3/bias:0
−0.4
−0.2
0.0
0.2
0.4
0
25
50
75
100
125
150
175
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00


dense_4/kernel:0
0
25
50
75
100
125
150
175
0
5
0.25
0.20
0.15
0.10
0.05
0.00


dense_4/bias:0
−0.4
−0.2
0.0
0.2
0.4
0
2
4
6
8
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00

Visualizing sparse model performance on MNIST and FMNIST

```
# Visualizing performance on MNIST
fig = plt.figure()
ax1 = fig.add_subplot(1, 1, 1)
plt.grid(b=None)
ax2 = ax1.twinx()
plt.grid(b=None)
plt.title('Test Accuracy as a function of k% Sparsity\nfor 4-hidden-layer MLP trained on MNIS
ax1.plot(sparsity_summary['k_sparsity'].values,
         sparsity_summary['mnist_acc_weight'].values,
         '#008fd5', linestyle=':', label='Weight-pruning Acc')
ax1.plot(sparsity_summary['k_sparsity'].values,
         sparsity_summary['mnist_acc_unit'].values,
         '#008fd5', linestyle='-', label='Unit-pruning Acc')
ax2.plot(sparsity_summary['k_sparsity'].values,
         sparsity_summary['mnist_loss_weight'].values,
         '#fc4f30', linestyle=':', label='Weight-pruning Loss')
ax2.plot(sparsity_summary['k_sparsity'].values,
         sparsity_summary['mnist_loss_unit'].values,
         '#fc4f30', linestyle='-', label='Unit-pruning Loss')

ax1.set_ylabel('Accuracy (%)', color='#008fd5')
ax2.set_ylabel('Loss (categorical crossentropy)', color='#fc4f30')
ax1.set_xlabel('k% Sparsity')
ax1.legend(loc='upper center', bbox_to_anchor=(0.5, -0.15), shadow=True, ncol=2);
ax2.legend(loc='upper center', bbox_to_anchor=(0.5, -0.25), shadow=True, ncol=2);
plt.savefig('images/MNIST_sparsity_comparisons.png')
```

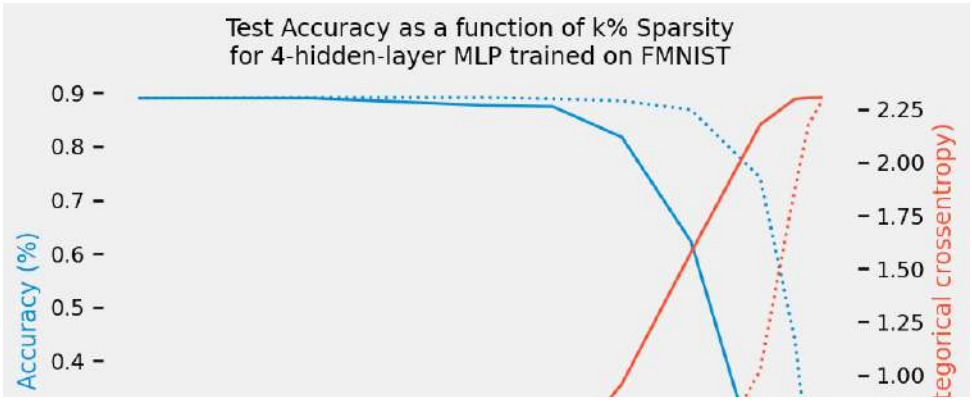
The performance curves for unit and weight pruning differ on MNIST.

Pruing weight matrices of dense matrices does not result in dramatic drops in accuracy or increases in loss until around $k = 80$. Even then, the accuracy does not begin to noticeably decrease until $k = 90$.

For unit-pruning, accuracy begins to fall earlier, around $k = 70$ (with loss beginning to increase around $k = 60$). Even then, both methods are able to effecively remove more than half of the network weights without any dramatic differences in test classification performance.

How does this work on FMNIST?


```
# Visualizing performance on KMNIST
fig = plt.figure()
ax1 = fig.add_subplot(1, 1, 1)
plt.grid(b=None)
ax2 = ax1.twinx()
plt.grid(b=None)
plt.title('Test Accuracy as a function of k% Sparsity\nfor 4-hidden-layer MLP trained on FMNI
ax1.plot(sparsity_summary['k_sparsity'].values,
         sparsity_summary['fmnist_acc_weight'].values,
         '#008fd5', linestyle=':', label='Weight-pruning Acc')
ax1.plot(sparsity_summary['k_sparsity'].values,
         sparsity_summary['fmnist_acc_unit'].values,
         '#008fd5', linestyle='-', label='Unit-pruning Acc')
ax2.plot(sparsity_summary['k_sparsity'].values,
         sparsity_summary['fmnist_loss_weight'].values,
         '#fc4f30', linestyle=':', label='Weight-pruning Loss')
ax2.plot(sparsity_summary['k_sparsity'].values,
         sparsity_summary['fmnist_loss_unit'].values,
         '#fc4f30', linestyle='-', label='Unit-pruning Loss')

ax1.set_ylabel('Accuracy (%)', color='#008fd5')
ax2.set_ylabel('Loss (categorical crossentropy)', color='#fc4f30')
ax1.set_xlabel('k% Sparsity')
ax1.legend(loc='upper center', bbox_to_anchor=(0.5, -0.15), shadow=True, ncol=2);
ax2.legend(loc='upper center', bbox_to_anchor=(0.5, -0.25), shadow=True, ncol=2);
plt.savefig('images/FMNIST_sparsity_comparisons.png')
```



Even on FMNIST, which had a lower initial accuracy and higher initial loss, the same pattern emerges

Again, pruing weight matrices of dense matrices does not result in dramatic drops in accuracy or increases in loss until around $k = 80$. Even then, the accuracy does not begin to noticeably decrease until $k = 90$.

For unit-pruning, the differences come much earlier for FMNIST than in MNIST. Accuracy begins to fall around $k = 60$ (with loss beginning to increase around $k = 60$).

Step 7

Why do you think that is/isn't?

The two methods use different strategies of finding the least useful weights. The weight-pruning finds the absolute values ($|w|$) of individual weights within the weight matrices. The unit-pruning finds L2 norms across entire columns of weight matrices. This difference is in part a difference between fine-grained and coarse-grained weight pruning.

Both the weight-pruning and unit-pruning are forms of saliency mapping. The pruning functions are going through the weight matrices and identifying the weights that would have minimal impact if they were multiplied with input data being fed into the layer. Given that some weights are orders of magnitude smaller than the largest ones, the impact of removing them is minimal. What is suprising is how much of the weight matrices are made up of these low-saliency weights.

Do you have any hypotheses as to why we are able to delete so much of the network without hurting performance (this is an open research question)?

This demonstration shows that neural network parameter counts of trained networks can be decreased by over 80% without hurting perofrmance. Other researchers have demonstrated pruning techniques that can decrease parameter counts by over 90%.

This bears some similarity to the "optimal brain damage" hypothesis by Yann LeCun. In addition to being based on practical observations of differences in weight activations in neural networks, this

bears similarity to ideas of how biological neural networks (like those in the neocortex) specialize. .

[Frankle & Carbin, 2019] proposed the "Lottery Ticket hypothesis". This hypthesis articulates that dense, randomly-initialized, feed-forward networks contain subnetworks ("winning tickets") that - when trained in isolation - reach test accuracy comparable to the original network in a similar number of iterations. It is named for the fact that these subnetworks, when initialized with random initialization, just happened to get initial weights that make training marticularly effective (i.e., they have won the "training lottery") For the task of MNIST, both weight-pruning and unit-pruning suggest that $2,388,710$ trainable parameters is supervluous compared to the truly optimal subnetwork that captures the concise classification function.

In finding ways of scaling the weights (or columns of weights), and removing the $k\%$ closest to 0, we are trying to filter out all but the weights that make the input features maximally separable. [Zhang et al., 2018] compared the use of this kind of separation to Grassmannian Manifolds. Given that there is no universal formula to figuring out provably optimal subspace packings, the Grassmannian model of neural networks frames them as approximators of the solutions. Given that this is a less than optimal packing, some of the subspaces of the weights (e.g., those represented either as individual weights or vectors of weights) can easily be removed without impacting the performance of the classification. At some point, there are no more values to remove save for those satisfying the subspace packing needed for the maximum separability of the classes.

The latent variable model of neural network function would suggest that neural networks work by trying to learn the requisite latent variables (David ha's blog post is a great example of this) needed to capture the epistemological essense of the given class. As David Ha's example shows, very high resolution images of MNIST digits can be produced from a generator with a vector containing just 32 real numbers.

For finding minimum subnetwork (as framed by the "lottery-ticket hypothesis"), the optimal subspace packing (as framed by the), it is also possible that there is more room for improvement beyond weight-pruning. Weight pruning looks at individual weight values in isolation. While the Unit pruning seems to suggest that looking at weights in combination does not produce better results, this is only one such filter for choosing weights to set to 0. Uber AI tested out multiple masks for finding the "lottery-ticket" subnetworks.

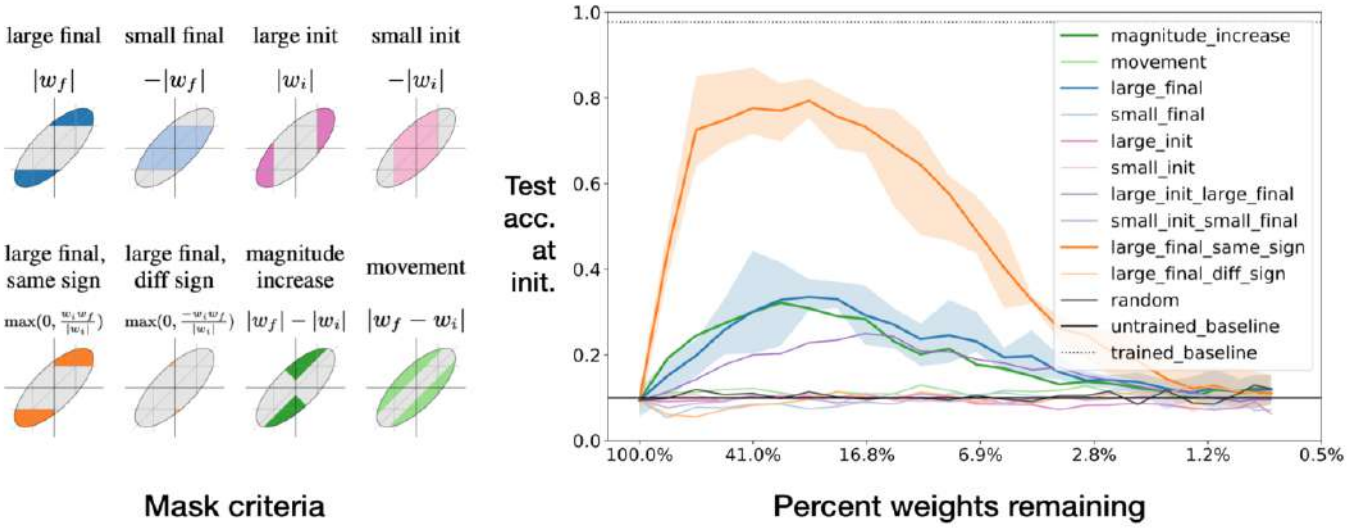


Figure from Uber AI's ICML 2019 poster [Deconstructing Lottery Tickets: Zeros, Signs, and the Supermask]

It is also important to note that this pruning method only involves setting weight values below a

Reducing model runtime with Sparsity

We will also test how much our models can be compressed using this new sparsity, and if this will speed up execution.

Compressing model *without* TF model optimization toolkit

```
def compress_sparse_weigths(sparse_model_weight_list, unique_columns=None):
    """
    Given a list of weight matrices of the layers, compreses all sparse matrices
    save for the last two layers
    Args:
      sparse_model_weight_list: List of weight matrices for kernel and bias
        weights.
    Returns:
      compressed_weight_list: list of weight matrices similar to the input list
        of matrices, except the dimensions have been reduced by omitting columns
        populated exclusively by 0s.
    """
    compressed_weight_list = []

    for i in range(0,len(sparse_model_weight_list)-2,2):
        # If this is just after the input layer, the matrix with input
        # dimensions will be added on unchanged. Otherwise the matrix will only
        # retain the columns that are not entirely 0s
```

```
        if unique_columns==None:
            kernel_weights = sparse_model_weight_list[i]
        else:
            kernel_weights = sparse_model_weight_list[i][unique_columns,:]
        bias_weights = sparse_model_weight_list[i+1]

        # a tuple of two arrays: 0th is row indices, 1st is cols
        indices = np.nonzero(kernel_weights)
        columns_non_unique = indices[1]
        unique_columns = sorted(set(columns_non_unique))
        kernel_weights = kernel_weights[:,unique_columns]

        bias_weights = bias_weights[unique_columns]

        # Adding the new kernel and bias weights to the new list
        compressed_weight_list.append(kernel_weights)
        compressed_weight_list.append(bias_weights)

    # Adding the softmax and modifying the pre-softmax weight matrices
    compressed_weight_list.append(sparse_model_weight_list[-2][unique_columns,:])
    compressed_weight_list.append(sparse_model_weight_list[-1])

    return compressed_weight_list
```

```
#@title Compressing sparse models (from saved files)

#@markdown Which dataset?
dataset = 'mnist' #@param ['mnist', 'fmnist']
#@markdown k sparsity
sparsity = "0.95" #@param ['0.0', '0.25', '0.5', '0.6', '0.7', '0.8', '0.9', '0.95', '0.97',
#@markdown We are only using sparse models that have undergone unit pruning
pruning = 'unit'
sparse_model = load_model('models/sparse_{}-model_k_{}_unit-pruned.h5'.format(dataset, sparsi

# List of weights from the loaded file
sparse_weight_list = sparse_model.get_weights()

# Creating a list of dense weights from the sparse weight list
compressed_weight_list = compress_sparse_weights(sparse_weight_list)

# Getting a 4-layer neural network with layer dimensions that match those of
# the new dense weight matrices (e.g., at 50% sparsity, layer size of 1000 is
# reduced to 500)
compressed_model = build_model_arch(input_shape, num_classes, sparsity=float(sparsity))
compressed_model.compile(loss=tf.keras.losses.categorical_crossentropy,
                         optimizer='adam', metrics=['accuracy'])
compressed_model.set_weights(compressed_weight_list)

# Printing the model summaries and comparing the weights of the original sparse
# model and the dense model
```

Compressing sparse models (from saved files)

Which dataset?

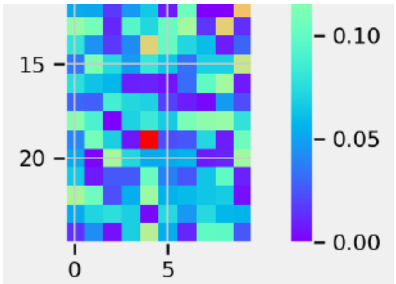
dataset: mnist

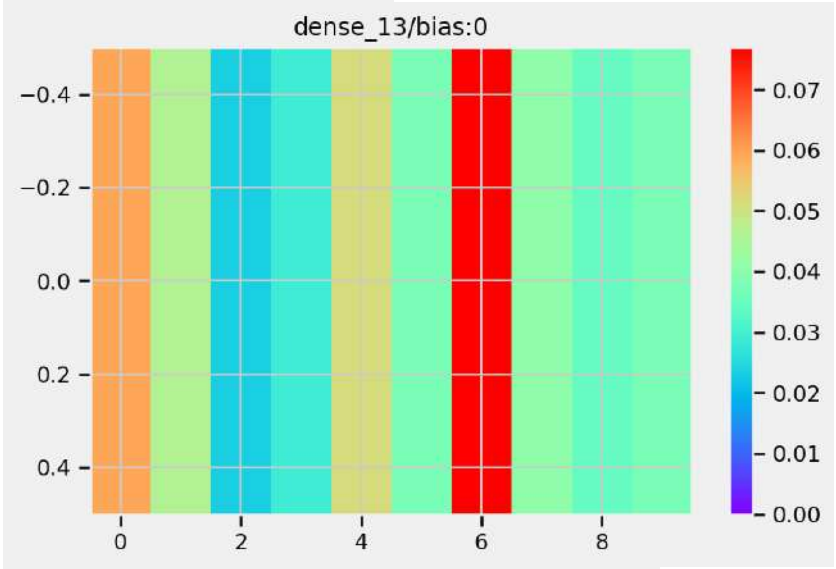
k sparsity

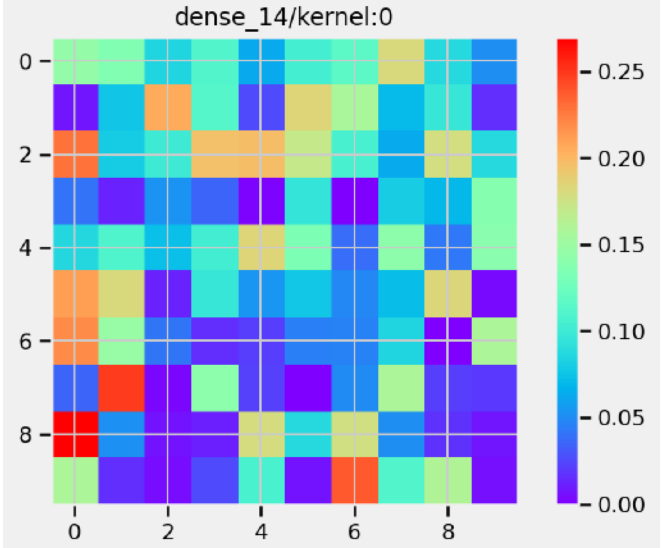
sparsity: 0.95

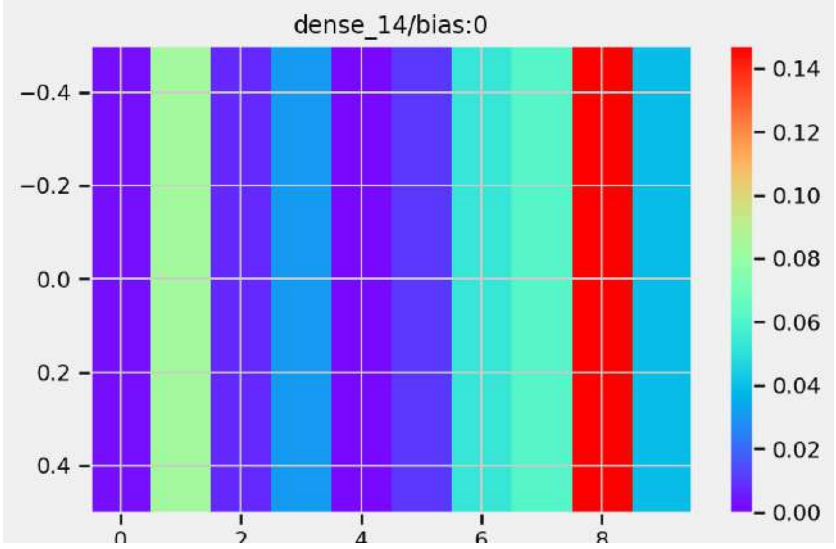
We are only using sparse models that have undergone **unit** pruning

```
print(compressed_model.summary())
print('\nVISUALIZING ORIGINAL SPARSE MODEL')
visualize_model_weights(sparse_model)
print('\nVISUALIZING COMPRESSED MODEL')
visualize_model_weights(compressed_model)
```


15
20
0
5
0.10
0.05
0.00


dense_13/bias:0
−0.4
−0.2
0.0
0.2
0.4
0
2
4
6
8
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00


dense_14/kernel:0
0
2
4
6
8
0.25
0.20
0.15
0.10
0.05
0.00


dense_14/bias:0
−0.4
−0.2
0.0
0.2
0.4
0
2
4
6
8
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00