UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

School of Electronics and Computer Science

Pruning ResNet Neural Networks Block by Block

by

Vlad Sebastian Velici

Thesis for the degree of Doctor of Philosophy

Supervisors: Prof. Adam Prügel-Bennett, Prof. Mahesan Niranjan

January 23, 2022

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
School of Electronics and Computer Science

Thesis for the degree of Doctor of Philosophy

PRUNING RESNET NEURAL NETWORKS BLOCK BY BLOCK

by Vlad Sebastian Velici

Neural network pruning has gained popularity for deep models with the goal of reducing storage and computational requirements, both for inference and training. Pruning weights from neural networks to make models sparse has been discussed for a long time starting with in 1990s. Structured pruning is more recent and it aims to minimize the size of neural networks as opposed to only introducing sparsity, having the advantage of not requiring specialized hardware or software. Most structured pruning works focus on neurons or convolutional filters. ResNet is a model architecture composed of many blocks that are linked with residual connections. In our work, we explore pruning larger structures – ResNet blocks, and thoroughly study the feasibility of block by block pruning whilst keeping track of the cost of fine-tuning the pruned networks. We use different block saliency metrics as well as different fine-tuning schedules and parameters. Pruning 27 blocks (50%) from a ResNet-110, in our best configuration, gives 6.48% test error on CIFAR-10, a 0.45% loss from the initial model. When pruning 45 blocks to obtain a similar size to that of a ResNet-20, our best method has a 1.92% loss from initial, 7.95% error. We observe that training small, standard ResNet configurations gives better results than pruning and argue that pruning block by block is only effective for pruning a small number of blocks, or when starting with a model pre-trained elsewhere and the cost of fine-tuning is of concern (fine-tuning alone can be much cheaper than training from scratch and give acceptable results, depending on the pruning configuration). Finally, our pruning work is produced from hundreds of experiments, and a by-product of running, organising and analysing them is our experiment management framework, dbx, aims to simplify this process by storing experiments and their results (as logs) in repositories that can be queried and synchronized between computers.

Contents

List of Figures

List of Tables

Acknowledgements

I am extremely grateful to my supervisors, Prof. Adam Prügel-Bennett and Prof. Mahesan Niranjan, for their guidance, support, and encouragement throughout my PhD, but also for their teaching during my undergraduate degree which has made me consider a PhD in the first place.

Many thanks to the VLC Research Group for a great work environment.

I also wish to thank my family and friends for their support.

Declaration of Authorship

I, Vlad Sebastian Velici , declare that the thesis entitled *Pruning ResNet Neural Networks Block by Block* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: Velici, V. (2021). Results and experiment logs for phd thesis "pruning resnet neural networks block by block". DOI: 10.5281/zenodo.4767180.
  Velici, V. and Prügel-Bennett, A. (2021a). Object detection for crabs in top-view seabed imagery. arXiv preprint arXiv:2105.02964.
  Velici, V. and Prügel-Bennett, A. (2021b). Rotlstm: Rotating memories in recurrent neural networks. arXiv preprint arXiv:2105.00357.

Signed:..

Date:..

Chapter 1

Introduction

Deep learning is a subfield of Machine Learning that has seen significant growth in recent years, both in terms of academic interest, number of publications and the size of conferences and also has shown significant impact in industry and it affects a large number people on a daily basis.

Deep learning techniques have consistently outperformed traditional machine learning methods in computer vision (Russakovsky et al., 2015), speech recognition, natural language processing and understanding, image understanding (LeCun et al., 2015), speech synthesis (Arik et al., 2017), and more. Deep learning methods and the tooling built for one domain can be applied across domains. These techniques can be applied to computer vision, healthcare, financial markets, natural language processing and many other fields.

The downsides of deep learning are the high amounts of data required, the large size of the models produced and the long time it takes to train models. For a number of years, reducing these costs was not a concern of the deep learning community, with trends leading to bigger and bigger networks, trained on bigger and bigger hardware (in all directions: more computing power specialized for deep learning, more memory, more storage), and with larger and larger datasets. This has changed in the last few years. Bigger models are still being developed and trained but there is a very active and growing area of deep learning and machine learning that now focuses on small machine learning that can run on mobile devices and even microcontrollers. Initiatives such as tinyML[1] (inaugural tinyML Summit in 2019) are now popular venues for sharing research in the subfield of small machine learning. Popular deep learning frameworks such as PyTorch with PyTorch Mobile and Tensorflow with Tensorflow Lite now have support for exporting and optimizing models for deployment on mobile phones but also other devices such as the Raspberry Pi.

[1]`https://www.tinyml.org`

The deep learning field is also moving very fast. Participation at, and the total number of papers submitted and accepted in, the top conferences rapidly increased since 2012, the best performing models for many tasks have outperformed previous architectures (with significant margins) many times since the explosion of deep learning, and they continue to do so. Larger and better quality datasets are becoming available as well as observing fast progress in methods to handle lower quality data and unlabelled data. Not all the progress is equal, however. Some of the very successful (and possibly other, less successful) papers in the field of deep learning are hard to reproduce, sometimes missing critical information or lacking public source code, data or experiment logs (Ding et al., 2018; Haibe-Kains et al., 2020).

1.1 Neural network pruning

Neural network pruning is a method of removing parts of a neural network typically with the aim of reducing its size, increasing its performance or for regularization. The idea of pruning neural networks is not new. The most notable classic works in this field are Optimal Brain Damange (LeCun et al., 1990) and Optimal Brain Surgeon (Hassibi et al., 1993) which present pruning techniques that help with increasing generalization.

Small models are ideal for applications where latency is critical, as well as where memory usage and power consumption are of concern. Small models can be easily deployed on many devices like embedded hardware, system-on-a-chip computers, mobile phones, and others. If the accuracy loss is small enough small models can be advantageous on servers as well since CPU computing is cheaper then GPU or other custom AI hardware[2] like TPUs[3].

Early works in pruning explored removing individual weights from a feedforward neural network to create sparse networks. Modern hardware is optimised to perform batch calculations efficiently and sparse neural networks do not efficiently make use of these optimisations. Sparsity does not directly result in smaller and faster models and for efficient use of sparse models there is a need for specialized hardware and software. Sparse representations also have a storage overhead. Pruning structural parts of neural networks, such as entire neurons or convolutional channels, has become popularized in the litarature where larger parts of a neural network are removed instead of single connections (Sze et al., 2017). The removed parts can be feature maps from convolutional networks or entire layers of a neural network. Removing a structural part of a neural network yields non-sparse networks that have a clear and easy to understand efficiency gain and size decrease.

[2]Based on prices of major cloud computing providers as of 21 May 2018, estimated based on online predictions (not batched jobs).

[3]TPU: Google's Tensor Processing Unit: `https://cloud.google.com/tpu/`.

A typical greedy iterative pruning algorithm takes the following form:

Step 1 train a neural network on a dataset,

Step 2 compute saliency of all prunable units,

Step 3 remove the least important prunable unit(s),

Step 4 fine-tune,

Step 5 repeat from **Step 2** until enough prunable units are removed.

There is a large variety of methods for selecting the units to prune, for different choices of prunable unit, presented in the literature. The performance of pruning greatly depends on unit selection, the amount and schedule of fine-tuning and the starting model.

In this thesis we focus on structured pruning of neural networks with residual connections – ResNets. ResNet networks have the ResNet block as the core structure, which is repeated many times across the network. Based on an observation that a few of these blocks can be removed or re-ordered with minimal loss of performance (Veit et al., 2016) we seek to explore a larger prunable unit: an entire ResNet block. Chapters 3 and 4 explore iterative block by block pruning with different block saliency metrics and fine-tuning schedules.

Our research goal is to explore block by block pruning. Specifically we are looking at the following topics:

- Exploration of pruning ResNet models block by block to understand the impact of pruning large structured blocks from a deep neural networks.
- How will different block selection methods for pruning perform, what can be learned from experimentally analysing them, and how important are they for the final impact on pruning?
- To what extent does pruning differ from training from scratch? How do pruned networks train and perform as compared to random networks?
- How much fine-tuning is needed during or after pruning? What is the impact of different fine-tuning schedules to the pruning process?

1.2 Experiment management

A by-product of our research from having over 500 experiments, and having run probably over 1000, is an experiment management framework, `dbx`. As part of the pruning work (and earlier work), many small programs have been written to address specific tasks related to experiment management. Running experiments on multiple machines, copying them all to a central location, querying and filtering them for generating plots and tables, finding whether errors occoured (computer crashes, wrong parameters), finding missing data points, are just a few examples of repetitive, time-consuming tasks.

`dbx` solve the parts about running experiments anywhere and making results available from a central location, as well as simplifying fetching experimental data from experiment logs.

One of the core principles of `dbx` experiments is the log, where all data produced by the experiment goes. Instead of compiling results from the experiment code, the focus is on saving all data that might be required later and separating the plotting, table generation, data extraction from the code that generates the raw data. Users can also write to an experiment log later, after the experiment finished; this can be used to add more data, such as evaluating performance at every pruning step.

It uses the idea of experiment repositories. An analogy to experiment repositories are git, or other code version control, repositories, but with the difference that experiments commonly include large files and require a few features on top of simply managing files, like querying experiments.

The idea of experiment logs has been used for all our pruning experiments, before `dbx` was started, and has proven to be an invaluable tool. We logged data that was not useful at first, such as how long it took to fine-tuning every epoch, and the scores of each block at each pruning step. These were much later used to compare experiments. For all experiments we saved all input parameters (including environment variables, hostname, code version), which has also proven invaluable over time, for fixing errors, re-running experiments, and search.

Many small utilities were written for our pruning work: find unfinished experiments, generate plots and tables, check if saved parameters match expected parameters for a whole directory tree, copy experiments between computers, generate scripts to run batches of experiments on the University's supercomputer or different machines, automatically check if results of generated batches are complete and correct, compare parameters of a list of experiments, and so on. They serve as the inspiration for starting `dbx`, which is presented at length in Chapter 5.

1.3 Thesis structure

In this thesis the topic of pruning neural networks is presented, with a focus on pruning neural networks with skip connections (also called residual connections) and the original ResNet architecture.

Pruning neural networks is not a new idea. It was first popularized by the paper *Optimal Brain Damage* (LeCun et al., 1990), where removing neurons from a feedforward neural network incresed generalization. With modern deep learning, pruning has been shown to be useful for a variety of tasks and many pruning methods have been developed. However, as pointed out by Blalock et al. (2020), the works in the literature on pruning neural networks is hardly comparable due to uses of different datasets, starting models, or measure of model size (for instance: FLOPs, MACs, number of parameters, sparcity, compression rate). There are other dimensions of comparison for pruning works which are qualitative: does the pruning method require a fully trained, partly trained, or randomly initialized model? It the pruning method creating sparse or dense models? Are there measurable time complexity reductions for inference, training, or both? Is the pruning static, applied once and producing a new model, or is it dynamic, choosing sub-networks based on input data at inference time? Chapter 2 explores how pruning is presented and evaluated in the literature and aims to set a common ground for future discussion on the topic, as well as presenting key contributions and results from the pruning literature.

Based on an observation by Veit et al. (2016) that a few parts of ResNet (He et al., 2016) networks can be removed or rearranged with only a small loss of accuracy, Chapter 3 presents different types of pruning ResNet networks block by block (as opposed to pruning smaller units of a network) and discusses the strengths and weaknesses of this method.

The idea of pruning block by block in further explored in Chapter 4, where different pruning settings are discussed: fine-tuning at every step for a few epochs, pruning models from random initialization or little training, group equalisation – making pruned architectures more like standard architectures, phased pruning where blocks are *trained* out of the model using a linearly decreasing scalar, and finally replacement pruning where two blocks are pruned at once and replaced by a randomly initialized one.

Another pillar of this thesis is to highlight the importance of systematically running, logging and desiging experiments that are reproducible. Chapter 5 presents a framework that was developed as part of this work to improve the management of experiments and with the aim of improving the workflow of researchers in the deep learning field. This chapter highlights common patterns in the workflow of a researcher, formulates a problem that can be solved with software and offers a working, open-source solution for machine learning experiment management. It includes a comparison with other similar frameworks.

General conclusions and future plans are presented in Chapter 6.

1.4 Early work

Before starting the work on pruning, two other projects were tackled as part of my PhD work. They are not the core of this thesis but took a significant portion of time. They are mentioned here for completeness and have been published as technical reports.

The first one, *Object detection for crabs in top-view seabed imagery* (Velici and Prügel-Bennett, 2021a), is an eary project aiming to create an object detector with the aim of counting underwater populations of crabs. There are two defining features of the crabs dataset. First, the image quality was low, often showing artefacts of a moving camera or as a result of creating large map-like meshes of the seabed. Second, the species of underwater animals we were interested in counting live in tightly packed colonies, resulting in images with no objects of interest or images that were very crowded. We obtained up to 37.35 mAP for a subset of 3 species (out of the total 6 species available, the 3 species which were left out have in total less than 1000 data samples, in comparison to 45,000 for the selected 3 species), and up to 29.93 mAP on the Pascal VOC dataset.

The other project we briefly explored before pruning, *RotLSTM: Rotating Memories in Recurrent Neural Networks* (Velici and Prügel-Bennett, 2021b), is about adding rotation gates to LSTM and GRU cells, obtaining RotLSTM and RotGRU, respectively. We evaluated the performance of the added gates using a simple architecture on the bAbI tasks for question answering (Weston et al., 2015). Pairwaise 2D rotations added to the LSTM cell (RotLSTM) showed improvements in accuracy of up to 20% over the plain LSTM model on bAbI tasks 5 (three argument relations) and 18 (reasoning about size). The improvement was not seen on all bAbI tasks and was not at all reproduced with the GRU and RotGRU experiments.

1.5 Definitions and clarifications

This section aims to clarify what the meanings of commonly used terms is within this body of work.

Model architecture, or simply **architecture**, is the way a machine learning model is built and does not include its weights or imply any training methods. It is purely how layers and neurons are connected to each other from the input to the output layer.

Model. A model is the archtecture with its weights. A trained model is a model that has been trained on some dataset.

Prunable unit. In this thesis the term **prunable unit** is used to describe what is being pruned from an artificial neural network. It can be a single neuron, a filter from a convolutional layer, an entire layer or a group of layers. Different methods of pruning differ in the choice of prunable unit and the method of choosing which units to prune.

Chapter 2

Review of major neural network pruning methods

Pruning neural networks has been discussed in the machine learning literature for many years and pruning is the area of deep learning has been of interest in the past decade. Many pruning methods have been introduced and have been used for different purposes ranging from the most common uses of reducing model size (Hu et al., 2016; Li et al., 2016) or inference latency (Park et al., 2016; Molchanov et al., 2016), of increasing generalisation (LeCun et al., 1990; Hassibi et al., 1993), to the less common uses of reducing power consumption of neural networks on mobile devices (Yang et al., 2017), or of finding *lucky* random initializations for training (Frankle and Carbin, 2019). Very recent papers combine architecture search and pruning to create performant and efficient networks (Dong and Yang, 2019; Noy et al., 2020).

Regardless of the reason for pruning or specific method, it is important to be able to compare different pruning methods for different tasks. As also brought into light by Blalock et al. (2020), the pruning literature is rather sparse when it comes to comparisons between different pruning methods or standard metrics to use when comparing pruning techniques.

Another lens through which pruning work can be compared is simply the aim of pruning. Different pruning works focus on different goals and as a consequence, different metrics. Commonly used metrics for pruning (typically over loss or accuracy) are FLOPs and inference time when pruning for increased inference latency, sparsity, comparession rate or numbers of parameters when pruning for size reduction.

In this chapter the aim is to present different popular pruning methods and categorise them by the choice of prunable unit, selection method, fine-tuning and whether the pruning is dynamic or not. Equally important, this chapter argues that it is difficult to make direct comparisons between pruning methods and that pruning methods depend greatly on how fine-tuning is performed, the starting weights, how much of a network is pruned and the size of the initial model. Pruning methods also tend to be inconsistent between different starting weights at different target compression ratios.

In Section 2.1 we introduce and explain popular choices of units that can be pruned form an artificial neural network: weights, neurons, and convolutional layers. Different methods of evaluating the importance of these units are discussed in Section 2.3. Fine-tuning is typically performed after pruning and is thoroughly explored in Section 2.4

2.1 Choice of prunable unit

The most critical configuration of a pruning algorithm is what it prunes. In this thesis the term *prunable unit* is used to describe the part of a model that a pruning algorithm removes in one step. Classic works such as LeCun et al. (1990) and Hassibi et al. (1993) prune individual weights of a model. It is possible to prune neurons (all input weights set to 0, equivalent to removing them) or larger parts of a model such as convolutional layer filters (Yang et al., 2017; Li et al., 2016; Luo et al., 2018; Anwar et al., 2017), entire layers of a model, or in the case of architectures with residual connections, entire blocks (Wu et al., 2018).

Pruning weights and neurons in a feedforward neural network is illustrated in Figure 2.1. Notice that if all the input weights of a neuron are pruned (entire row in the weights matrix is 0), then the neuron can safely be pruned since it will not contribute to the end result. Similarly the neuron can also be removed if all its output weights are pruned. This process can be repeated until there are no such neurons left.

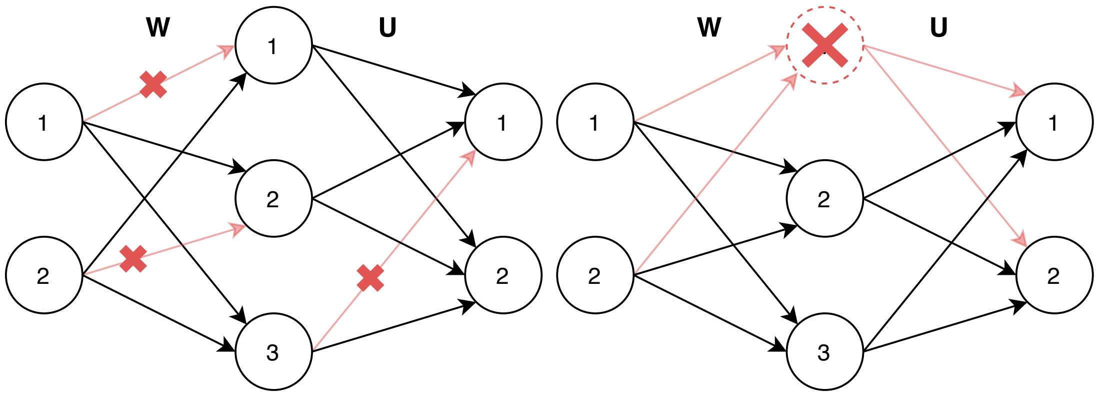


Figure 2.1: A feedforward network with one hidden layer. **W** and **U** denote the weight matrices. Individual weights are pruned on the left (set $\mathbf{W}_{1,1}$, $\mathbf{W}_{2,2}$, $\mathbf{U}_{3,1} = 0$) and the same network with an entire neuron pruned is shown on the right (set $\mathbf{W}_{i,1} = 0$ and $\mathbf{U}_{1,j} = 0$, or removing the first column of **W** and first row of **U**). Pruned units in red marked with an "x".

Convolutional layers can be pruned filter by filter to produce smaller convolutional layers. A visual example is shown in Figure 2.2, where the effects of removing a convolutional filter is displayed[1]. Each convolutional filter has a set of weights for each input channel. When a convolutional filter is removed from layer i, the layer outputs one less channel (feature map) and thus all the weights for this channel must be removed from the filters of layer $i + 1$.

Convolutional layers do not typically store a large number of parameters but they often take most of the computational time. Li et al. (2016) shows that pruning convolutional filters and features maps can reduce inference costs by up 38% for ResNet-110 on CIFAR-10. Their method prunes the filters (and associated feature maps) with the smaller weights magnitude, and they fine-tune the network at the end of pruning. One observation from this work is that, on ResNet, the pruned architectures (with some filters removed, but networks have the same blocks configuration) perform worse when they are trained from scratch than when they are pruned and fine-tuned. Another observation is that when removing a small number of filters the accuracy increases slightly without training.

Yang et al. (2017) shows a method of pruning convolutional neural networks with the goal of reducing energy use on mobile devices. This is achived via estimating the energy of CNNs and using these estimates to guide the pruning process. Energy use of neural networks for mobile devices was also mentioned in Han et al. (2015) as motivation for pruning. Many other works study or use convolutional filter and feature map pruning (Anwar et al., 2017; Molchanov et al., 2016; Ding et al., 2018; He et al., 2020; Anwar and Sung, 2016; van Amersfoort et al., 2020).

When pruning weights, neurons, or convolutional layers the pruning algorithm can select prunable units globally or from a subset of the network. A common choice is to iterate pruning and fine-tuning in a layer by layer fashion, starting from the one closest to the input and finishing with the one closest to the output. For example, in Luo et al. (2018) each layer is pruned to the desired size in one pruning step, followed by an epoch of fine-tuning (except last layer, which has 10 epochs of fine-tuning).

Entire layers of convolutional neural networks can be removed as long as the input and output sizes match, however this type of pruning often result in network configurations that do not perform well for feedforward networks (Veit et al., 2016) and are likely not trainable to recover a reasonable portion of the performance of the original network. Architectures with residual connections are more robust to pruning larger parts of the network; this will be discussed further in Chapter 3.

In the literature the term *structured pruning* is used for any choice of prunable unit that can result in removing parts of the network (as opposed to setting a subset of the weights

[1]This format of drawing convolutional layers was inspired from the excellent write-up from Stanford CS231n, available at `https://cs231n.github.io/convolutional-networks/`.

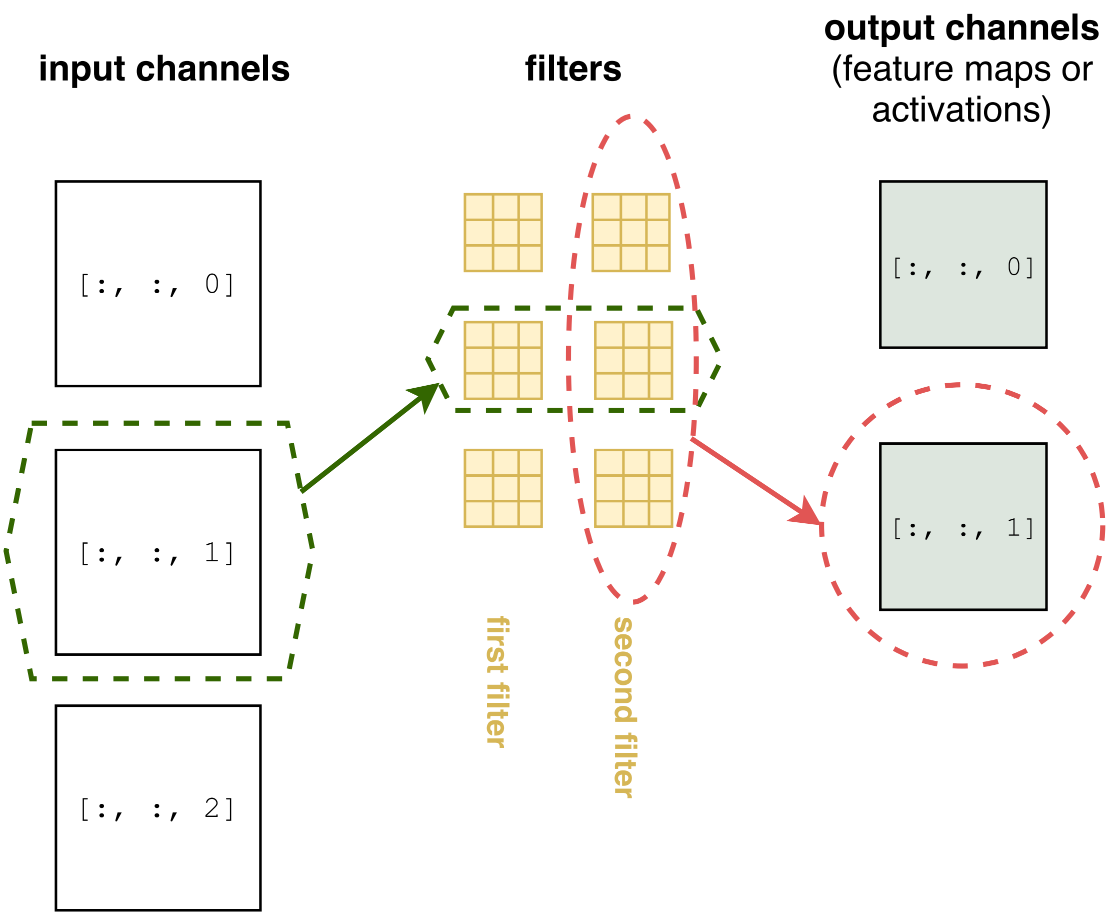


Figure 2.2: The inputs (3 2D channels), filters and outputs (2 2D channels) of a 2D convolutional layer. The red highlight (right part) shows that pruning the second filter of this layer results in removing the second channel of the output. The green highlight (left part) shows that if one of the inputs is removed (for example by removing a filter from the previous layer), the weights for that channel from each filter can be removed.

to 0). Structured pruning methods produce dense models that have less parameters than the initial model. Similarly, *unstructured pruning* is the pruning of individual weights which results in sparse models.

Sparse models do not necessarily yield runtime speed and storage improvements due to issues that arise with storing sparse tensors and performing sparse and sparse-dense operations on modern GPUs using any of the major machine learning frameworks. There has been recent progress developing sparse GPU kernels for deep learning (Gale et al., 2020; Zhu et al., 2019; Ren et al., 2018; Gray et al., 2017), however, at the time of writing, they are not readily available[2] and structured pruning remains the preferred method for increasing speed and reducing storage requirements without the use of specialized libraries or hardware.

[2]Source code and instructions might be publicly available but we argue that easy integration with major deep learning frameworks is required for widespread use.

Unstructured pruning can be successfully used for different applications when sparsity is desirable or as part of other tasks where the focus is not necessarily to reduce model size or speed up inference (other applications include studying importance of weights, interpretability, and pruning to improve generalization as opposed to reduce model size). Equally, unstructured pruning can be turned into structured pruning by pruning items from the same structural unit (e.g. weights for the same neruone or convolutional filter). In this thesis the focus is on structured pruning.

2.2 Global or layer-wise pruning

A very popular method of pruning deep neural networks is layer by layer (layer-wise) (Han et al., 2015; Abbasi-Asl and Yu, 2017; Li et al., 2016; Han et al., 2015; Molchanov et al., 2016), where pruning starts either close to the input or close to the output and is iterated layer by layer until the other end of the network is reached. Typically, a pruning rate is allocated to each layer. Choosing the pruning rate is a hyper-parameter optimization problem that is often solved outside of the pruning loop, with a common choice being an equal fixed pruning rate for all layers that are eligible for pruning. The saliency metric is applied layer by layer. Fine-tuning is performed after each layer was pruned; Li et al. (2016) report that fine-tuning once at the end yields worse accuracies than fine-tuning iteratively.

Global pruning refers to pruning the whole model at once, using a saliency metric to choose which prunable units to remove from anywhere in the model. Global pruning can also be performed iteratively with fine-tuning in between pruning steps. Each pruning step can remove one or more prunable units. Works that use global pruning include van Amersfoort et al. (2020); Lee et al. (2018); Hu et al. (2016).

2.3 Prunable units saliency metric

In this section the main methods of choosing which prunable units to prune at each pruning step are presented. Scoring can also be refered to as evaluating or computing the saliency or importance of prunable units. In this thesis we refer to the algorithm of computing the *score* of each prunable unit as the *saliency metric*. Prunable units with small score are less important and are pruned first. Unless otherwise specified the notation $S(x)$ is used to mean the score of prunable unit x.

There are four main categories of saliency metrics based on the computational demand and whether they require data or not.

- **Based on weights.** Using the weights and the weights alone to compute the saliency metric. No data is required and fastest to evaluate.

- **Based on activations.** Using the activations to evaluate the saliency. Uses data but not backpropagation. Most methods can be efficient but slower than weights-based methods.
- **Based on gradients.** Using gradients to evaluate the saliency metric. Uses data and backpropagation. Similar in efficiency with activation-based methods.
- **Direct evaluation.** Evaluating the model with different weights pruned to estimate saliency. Very inefficient requiring many evaluation runs of the model on a target dataset.

2.3.1 Saliency based on the value of weights

Since pruning a set of weights is equivalent to setting them to zero, the most intuitive way to select which weights to prune is by their absolute value,

$$S(x) = |x|. \tag{2.1}$$

The weights whose values are closest to zero are pruned. The number of weights to prune at each step can be chosen dynamically based on the score (for instance with a threshold) or as a hyperparameter to the pruning algorithm.

This metric can be extended for use with different choices of prunable unit by the use of norms. Denote the weights in a prunable unit as $\mathbf{x} \in \mathbb{R}^n$ where n is the number of weights in $\mathbf{x}$. Then, for any choice of prunable unit we can define a saliency metric based on the absolute mean of the weights (using ℓ^1 norm),

$$S_{\text{mean}} = \frac{1}{n}\|\mathbf{x}\|_1, \tag{2.2}$$

as well as the weights magnitude saliency measure, which is often used in the literature,

$$S_{\text{magnitude}}(\mathbf{x}) = \frac{1}{n}\|\mathbf{x}\|_2. \tag{2.3}$$

Using the ℓ^1 or ℓ^2 norm alone is only valid for comparing prunable units that have the same number of weights n, therefore the score is normalised by n to remediate this issue.

Pruning based on weights alone is an efficient and fast process. It does not require any data to function and, as such, the pruning can be applied in the same way regardless of the target dataset (this holds as long as there is no fine-tuning). This method is widely used in the literature (Frankle and Carbin, 2019; Han et al., 2015; Li et al., 2016).

2.3.2 Saliency based on activation statistics

The main limitation of computing saliency based on weights alone is that it entirely ignores the data. Weights that are small and constantly receive large input are similar to weights that are large but constantly receive small input. Adding data to the equation of computing saliency has the benefit of helping the pruning algorithm to converge to a pruned model that is likely to work well on that target dataset.

The simplest metrics based on activations are the same ones that are defined for weights but adjusted to allow for any number of data inputs. Let ${\mathbf{y}^{(\mathbf{i})}}_k$ be the activation (output) of the prunable unit at index k with input i. For simplicity let $\mathbf{a} \in \mathbb{R}^{n \times m}$ be the concatenation of all $\mathbf{y}^{(\mathbf{i})}$ where m is the total number of data samples used. Equation 2.3 becomes

$$S_{\text{act. magnitude}}(\mathbf{a}) = \frac{1}{n \times m} \|\mathbf{a}\|_2, \tag{2.4}$$

and Equation 2.2 can be similarly rewritten.

Average percentage of zeros (APoZ), introduced in Hu et al. (2016), is another metric which uses the ℓ^0 norm divided by the number of weights in a prunable unit,

$$\text{APoZ}(\mathbf{x}) = 1 - \frac{\|\mathbf{a}\|_0}{n}. \tag{2.5}$$

To use APoZ in our setting we define $S_{\text{APoZ}} = \frac{1}{n}\|\mathbf{a}\|_0 - 1$ (equivalent to average percentage of non-zeros) to be consistent in that the smallest scores are pruned first (APoZ as it was first introduced is used to pruned elements with the largest value).

So far the activation saliency metrics have used the activations of the unit to be pruned alone. In Luo et al. (2018) the idea of using pruning a layer based on the next layer's statistics is evaluated and shown to perfom well. The same idea can be extended such that the impact of removing unit at index i is evaluated by observing the change later in the network, for instance at prunable unit $i+k$ where $k > 1$ is an integer hyperparameter or at a fixed locations close to the end of the network.

2.3.3 Saliency based on gradients

Optimal Brain Damage (LeCun et al., 1990) (OBD) is a method of unstructured pruning that computes saliency of weights using the second derivative of the loss function in regards to the parameters, the salience metric for parameter x_i is $\frac{\partial^2 L}{\partial {x_i}^2}$, where L is the loss function. ODB is an iterative method where the original model is (1) first trained to convergence, (2) saliency is evaluated, (3) a number of weights are removed (set to zero and frozen), (4) the model is then re-trained, and the process is repeated from (2) until target size is met.

Optimal Brain Surgeon (Hassibi et al., 1993) (OBS) is another method of pruning weights. It uses the inverse of the Hessian matrix of the loss function as a saliency metric. For parameter x_i the saliency is $\frac{x_i^2}{2H_{ii}^{-1}}$. The re-training is replaced by updating the weights by subtracting $\frac{x_i H^{-1} e_i}{H_{ii}^{-1}}$ where e_i is the unit vector corresponding to weight x_i.

The bottleneck of OBS is calculating and storing the Hessian and its inverse, making it unpractical on large models. Dong et al. (2017) presents a layer-wise extension of OBS suitable for deep neural networks. It works by calculating a layer by layer error by comparing the output of the layer with its output with weights removed. The saliency for weight x_i at layer l is $\frac{{x_i}^2}{2[H_l^{-1}]_{ii}}$. After a parameter is removed the weights of the layer are updated similarly to OBS.

In SNIP (Lee et al., 2018) the authors add another set of weights, $\mathbf{c}$, of equal size to the original weights of the model $\mathbf{w}$. The weights used in the model are then $w_i' = c_i w_i$. The saliency of weight i is then computed as $\frac{|g_i|}{\sum_k |g_k|}$ with $g_j = \frac{\partial L}{\partial C_j}$. This saliency metric computes the connection sensitivity. The value of the loss function is not required in this formulation, making this method suitable to use before initial training of the network. In practice a pruning mask with values 1 is created to be the same size of the weights. At a forward pass, each weight is multiplied by its respective value in the map. The loss function is calculated with a forward pass and the gradients g_i are obtained by performing a backward pass.

For structured pruning, this is extended such that the mask only has one element corresponding to each prunable unit. For convolutional layers the map has an element for each output channel. For fully connected layers, the mask has one element for each hidden unit. The prunable unit is then pruned based on the gradient of the loss in regards to the mask (van Amersfoort et al., 2020). Both methods (structured and unstructured) are designed as single-shot pruning methods applied before training. In Section 3.3.2 present a similar saliency metric using a mask with one element for each ResNet block.

2.3.4 Learning masks without explicit saliency evaluation

It is possible to learn pruning masks using backpropagation and reach above random accuracies without any training of the actual weights (Zhou et al., 2019). A mask with one element corresponding to each prunable unit (in the case of Zhou et al. (2019) each weight) is interpreted as a bernoulli distribution and trained with backpropagation. The loss is calculated by sampling from the distribution and evaluating the cost function on the target dataset. This method is reported to reach 65.4% accuracy on CIFAR-10 (average of 10 independent samples of the mask) by only learning the mask and without any training of the randomly initialized weights. The accuracy is low in comparison to trained networks on CIFAR-10 but this result illustrates that pruning weights of a randomly initialized network can be effective. The accuracy of the original randomly initialized network was not made public, but a random choice is expected to give an accuracy in the proximity of 10%. A four-layer convolutional network was used, with the configuration: 4 convolutional layers with filters 64, 64, pooling, 128, 128, pooling, followed by fully connected layers of sizes 256, 256, and 10.

2.4 Fine-tuning method

Fine-tuning is the process of training a model after it has been pruned to recover as much of the lost accuracy as possible. Fine-tuning can be applied throughout the pruning process based on different criteria such as accuracy change or number of pruning steps. Fine-tuning is typically applied after certain pruning steps based on different criteria (like a schedule or a threshold) or simply at the end of the pruning process. Alternatively, the initial model can be pruned and trained at the same time resulting in small models without the need to fully train large networks.

2.4.1 Amount of fine-tuning steps

A very popular fine-tuning schedule is the *one-shot fine-tuning*, where only one fine-tuning loop is used at the end of pruning. This method is used by Li et al. (2016), which perform a single run of fine-tuning at the end of the pruning process (on CIFAR-10, they use 40 epochs with a fixed learning rate of 0.001) as fine-tuning is an expensive operation.

In the context of pruning weights, Han et al. (2015) report that training at every pruning step performs better than aggresively pruning once followed and training at the end.

Multiple fine-tuning runs throughout pruning which are triggered by different criteria such as accuracy drop, change or a fixed schedule based on the number of pruning steps or number of parameters pruned. Introducing fine-tuning loops into the pruning process prevents the model from losing too much accuracy and in some situations it might allow for less overall fine-tuning as compared to a one-shot configuration.

In the case of greedy iterative pruning algorithms, multiple runs of fine-tuning also have the benefit of regularly correcting the weights, thus the greedy pruning algorithm can, intuitively, make better local selections of which prunable units to remove. This is based on the intuition that training a pruned network is likely to change the sailency of the remaining weights. Fine-tuning at every pruning, even if done for only a few updates, step also allows the network to gradually adjust to the new shape. Molchanov et al. (2016) apply 1000 (or 100) updates after each pruning step in the context of pruning convolutional filters for transfer learning, which is followed by a larger fine-tuning loop at the end.

Fine-tuning is applied after each pruning step in Hu et al. (2016) but the weights are reset from the initial model before each fine-tuning loop. The authors found that resetting the weights in this way prevents sending the network to a bad local minima but still allow the saliency metric to use an updated (recently fine-tuned) network.

In Luo et al. (2018) convolutional filter pruning is applied iteratively layer by layer. The authors claim that pruning too many units at a time without any fine-tuning damages the network to a state where the lost accuracy is unrecoverable. After each pruning iteration, they apply a small fine-tuning loop of 1 or 2 epochs. A longer fine-tuning loop is performed only at the end of pruning process, after the last layer is pruned.

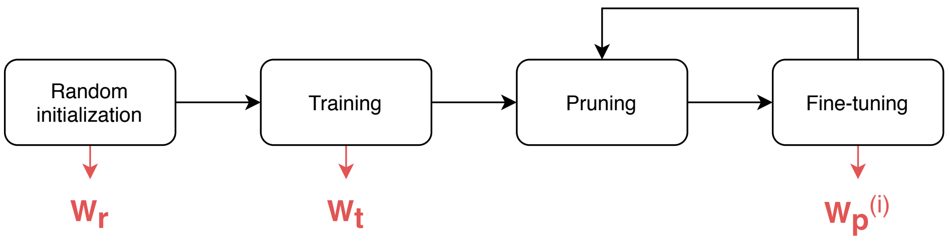
2.4.2 Weights to use for fine-tuning

Most commonly fine-tuning is what follows after pruning, without any other re-initialization or copying of the weights. A small variation is to use the best epoch from fine-tuning according to either the loss or accuracy on a validation set.

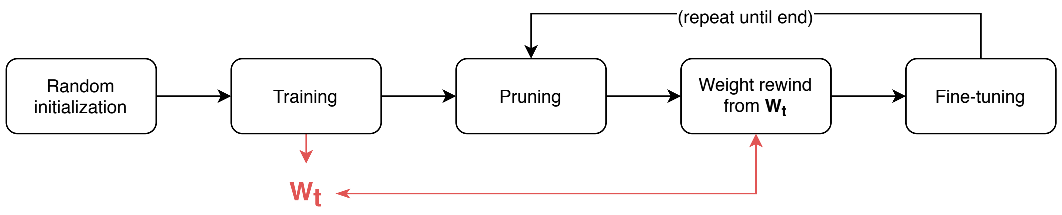
Another approach is to *rewind* the weights at the end or throughout the training process. For notation (Figure 2.3a) let $\mathbf{W_r}$ be the random initalization of a model, $\mathbf{W_t}$ the weights of the model after being fully trained on a dataset and $\mathbf{W_p}^{(i)}$ the (remaining) weights of a model after pruning step i finished (after fine-tuning, if applicable).

In the context of iterative convolutional filter pruning, where fine-tuning loops are applied after pruning, Hu et al. (2016) finds that resetting the remaining weights to $\mathbf{W_t}$ after pruning and before fine-tuning (Figure 2.3b) yields better performance. van Amersfoort et al. (2020) use $\mathbf{W_r}$ with a small amount of data to find smaller networks that train well, eliminating the iterative pruning and fine-tuning process altogether.

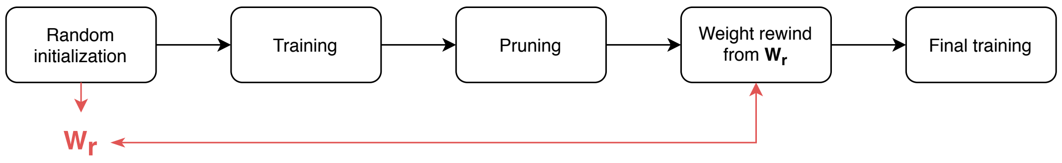
In an unstructured pruning setting, Frankle and Carbin (2019) use the weights that the original model had as initialization, $\mathbf{W_r}$, after pruning, effectively using the pruning process as a means to find *winning lottery tickets* - subnetworks that train as well as the large initial network (Figure 2.3c). Wang et al. (2020) find such subnetworks before training only analysing $\mathbf{W_r}$.



(a) Notation of weights at different steps throughout training and pruning.



(b) Weight rewind before fine-tuning, as done in Hu et al. (2016).



(c) Weight rewind to initial random initalization, as done in Frankle and Carbin (2019).

Figure 2.3: Diagrams showing different methods of resetting weights throughout a pruning process.

2.4.3 Combined pruning and training

Pruning can be categorized based on the method of pre-training the initial model to be pruned. Traditional pruning methods take as input a fully trained model and prune it. Arguably a pruning algorithm must be robust to the method of training for its input model, since in practice it is useful to prune models that one has no control over their initial training (ie. pre-trained models used in transfer learning). On the other hand pruning can be used for the purpose of finding more efficient models or increase accuracy, in which case it is acceptable to prune at any stage and modify the initial training method.

Different methods of regularization or other training constraints can be used when training a model from scratch in an attempt to make the model more suitable for pruning (Wen et al., 2016; Gomez et al., 2019; Zhou et al., 2016). Regularization can be also combined with fine-tuning throughout the pruning process. Additional fine-tuning can be used as a pre-processing step that is applied to a pre-trained model, in preparation for pruning (Ding et al., 2018).

Another set of works (Frankle and Carbin, 2019) present the idea of pruning trained or partially trained models and only using the resulting architecture, which is then trained from scratch. The critical detail is that, before training from scratch, the model with the pruned architecture is initialized to the same (initially random) weights (leaving out the pruned ones) that the original model was initialized with. Recent works introduce the idea of pruning models from the initial random initialization or after small amounts of training. Others have combined pruning and training so that the benefits of small networks are redeemed in both training and inference (Lee et al., 2020; van Amersfoort et al., 2020; Lee et al., 2018; Wang et al., 2020).

2.5 Static or dynamic pruning

Static pruning is where we start with a model, perform pruning (and fine-tuning), and obtain another model that is smaller. Data is typically used (not always) throughout the pruning process to determine which prunable units to remove.

In dynamic (or runtime) pruning a network is dynamically pruned based on the input data such that only a subset of the weights are active for any given input. There is no traditional pruning step where a new model is created.

BlockDrop (Wu et al., 2018) is a method of dynamically pruning ResNet networks at runtime. It works by training a policy network that is very small in comparison with the large ResNet model. At inference time, the input is first fed through the policy network which outputs a mask, a binary mapping with a value for each block of the base ResNet model. The mask is then applied on the large model, turning off the blocks that have a mask value of 0. The input is then passed through the base model and the output is the final prediction. The results of BlockDrop show that this method of pruning can speed up inference time and the final model accuracy is not significantly affected. It also shows that qualitatively complex inputs (images with small objects, complicated background, or generally more complex images) trigger using more blocks than simpler images (images with larger objects and/or simpler backrounds). The only downside to this approach is that inference has the overhead of running the policy network first, to obtain the block mask. For large ResNet models, however, this overhead is small compared to the savings in computation it gives on average.

Another work presenting a method of dynamic pruning is Lin et al. (2017) where a set of convolutional filters is dynamically selected to be pruned at each layer based on the input being processed. The decision network is a recurrent neural network which is executed in parallel to the backbone CNN. The amount of resources the model should use overall can be adjusted after training by adding a coefficient, therefore the same model can be deployed on systems with different resources available.

2.6 Soft pruning

Soft pruning is the idea of running a pruning algorithm to select prunable units, but instead of removing or setting the weights to zero permanently, the weights are set to zero and are allowed to change during fine-tuning.

Soft pruning can be combined with training a model from scratch such that at each training epoch, a soft pruning iteration is run setting a subset of filters to zero. At the next training epoch the filters are trained normally, thus the capacity of the initial model is not lost. The process is repeated until convergence. The last iteration can be one of hard pruning to actually obtain a model of smaller size. The initial model can be randomly initialized or pre-trained.

Soft pruning was introduced by He et al. (2018) as soft filter pruning (SFP) where filters of convolutional neural networks are soft pruned. It was later expanded by introducing asymptotic soft filter pruning (ASFP) (He et al., 2020) which adds a method of soft pruning different amount of filters as the training progresses.

2.7 Pruning as compression

In the literature pruning is often considered a form of compression. That is, taking a model and pruning it to find a smaller network that is similar to the orignal one in representation and as a result in accuracy. In this case the model is the *data* that gets compressed. Compression algorithms are compared when compressing the same data; similarly, pruning algorithms must be compared on the same initial models and same datasets.

The compression ratio for pruning is defined as

$$\text{compression ratio} = \frac{\text{original size}}{\text{pruned size}}.$$

A model reduced to half its size has a compression ratio of 2. The compression ratio does not include any information about the amount of weights (or other prunable units) removed, it is only a measure of how much has been pruned from the original size. It also does not suggest any easy to understand upper limit (the maximum compression ratio depends on the model architecture and choice of prunable unit). Another source of confusion can come from how the model size is actually calculated. The popular choices for pruning are number of prunable units and number of parameters (which can also include or exclude parts of a model). The use of compression ratio in pruning papers is also pointed out by Blalock et al. (2020) where the authors highlight that different works use different definitions of compression ratio for pruning, making comparing results difficult. For instance Han et al. (2015) uses the compression ratio as defiend above (and reports in the "2×" format); on the other hand, Dong et al. (2017) defines compression ratio as $\frac{\text{pruned size}}{\text{original size}}$ and reports this number as a percentage, except in figures where it is used as a scalar.

Pruning performance changes drastically depending on how much is pruned in absolute terms, not only how much is pruned as a function of the original size. It is true that larger neural networks can be pruned more in absolute terms, a fact that is explained by having a large over-capacity for fitting the training set to start with as well as having a much larger capacity in the pruned model.

Furthermore Blakeney et al. (2020) shows that pruning is not merely a compression method and that even though pruned networks can often match the performance of the initial models, the internal representations of iterative pruning methods are less and less similar as the sparsity increases. Up to 30% sparsity gives similar representations and the similarity decreases as sparsity increases. The representations were analysed using the Singular Vector Canonical Correlation Analysis (SVCCA) tool (Raghu et al., 2017). The authors showed this in an unstructured pruning context. Our intuition is that structured pruning diverges the representation even more.

2.8 Comparing pruning methods

The performance of pruning depends greatly on the starting model and the dataset. As a result it is rather difficult to directly compare against other published works without re-running all the relevant experiments with new starting models. Furthermore, the literature on pruning uses a variation of metrics for evaluating pruning performance such as FLOPs, number of parameters, compression rate, number of prunable units removed, or inference time, which adds more nuances when comparing to other works. We argue that it is critical that the same starting weights and initial model sizes are used when comparing against other pruning methods.

Not only different works use different metrics for measuring how much of a network was pruned, but sometimes the numbers are misleading. For instance, in Liu et al. (2018), the percentage of pruned weights is reported as a percentage of pruned weights from the set of all convolutional weights, whereas other works, such as Li et al. (2016), report the percentage of weights removed from the total weights.

This section presents a weak upper bound and a lower bound that can serve as a first point of comparison or as a guideline for pruning methods. Both the lower bound (random pruning) and the higher bound (oracle pruning) can be used with or without fine-tuning, but for the comparison to be valid, the same amount and schedule of fine-tuning must be used for the lower bound, upper bound and the pruning method that is to be compared. Finally we discuss the importance and significance of comparing against training similarly-sized models from scratch.

2.8.1 Oracle pruning

Given a neural network with the set of weights M, we define perfect pruning as finding the subset M' of a predetermined size s that gives the best performance after fine-tuning.

It is computationally infeasible to perform perfect pruning. The performance after fine-tuning is not strongly correlated to the performance before fine-tuning (discussed in Section 3.3.3 in the context of ResNet block pruning) and different amounts and configurations of fine-tuning can greatly impact the end result, meaning that architectures found by perfect pruning may change based on the amount and schedule of fine-tuning.

The fine-tuning schedule and amount must be kept the same for computing perfect pruning (or an approximation) and for the pruning method that it is to be compared with. Note that the critical finding is not the architecture M', but rather the resulting accuracy of the model, to be used as a comparison.

Three simpler versions of perfect pruning can be easily defined. First, instead of fine-tuning all subsets of size s, only fine-tune the ones that have the best accuracy before fine-tuning; it still has the drawback of having to evaluate a very large number of weight combinations. The second option is to remove fine-tuning altogether and make the comparison without any fine-tuning; this is a weak comparison point since pruning performance greatly depends on fine-tuning. The third option is to perform perfect greedy pruning without fine-tuning at each step until size s is obtained, as done in Molchanov et al. (2016) and refered to as *oracle pruning*.

Oracle pruning is the only feasible calculation for any reasonable choices of initial model, prunable units and datasets, and fine-tuning can be used throughout the process (after each pruning step or similar).

The concept of oracle pruning is introduced to serve as a baseline for comparing pruning methods. It sets an expectation of the performance to obtain via pruning. If a pruning method underperforms the oracle pruning by a significant margin there is clear room for improvement, but oracle pruning is only a weak baseline based on a simple heuristic. The opposite is unknown, it cannot be stated that if a pruning method performs as well as oracle pruning it cannot be improved.

The performance of the initial model can also be a baseline for pruning performance and be used as a weak upper bound, but there are cases where pruned models do outperform their initial models while using less parameters.

2.8.2 Random pruning

Random pruning refers to randomly selecting prunable units.

Oracle pruning is a baseline for pruning a model to a specified size. Random pruning is a lower bound. Random pruning is obtaining a subset M' of size s that is selected at random. It serves as a good lower bound for evaluating pruning methods over a specific starting model and it is cheap to obtain. Random pruning may seem like a trivial target to beat, but in our experiments, random ResNet block pruning is competitive with other selection methods when fine-tuning is enabled. Random block pruning is further discussed in Section 3.3.2.

2.8.3 Training from scratch

Training from scratch is often overlooked, but another comparison to be done beyond the lower and upper bounds described above is to train similarly-sized models from scratch. It can be discovered that training from scratch is better than pruning and fine-tuning, or perhaps that pruning finds useful architectures to be trained from scratch (Liu et al., 2018).

Another outcome, like that presented in *The Lottery Ticket Hypothesis* (Frankle and Carbin, 2019), is that using the initial random weights from the large model combined with the architecture discovered through pruning, is a *winning ticket*, which is possible to train from scratch to a similar accuracy as the bigger inital model even in the case where the pruned model (with weights from the trained initial model) did not train well.

There are pruning works in the literature which omit comparing their results with training similarly sized models from scratch (Abbasi-Asl and Yu, 2017; Han et al., 2015). Li et al. (2016) reports a comparison between pruned architectures trained from scratch and pruned architectures fine-tuned, but does not train similarly sized networks in a *standard*[3] configuration. The sizes of filters after pruning might hinder training from scratch.

It is important to note that fully training a large model, pruning it, and fine-tuning it can be overall more computationally expensive than training a small model from scratch. It is then to be noted that when comparing models small trained from scratch and pruned models, the computenation complexity of both operations must be taken into account. We further discuss training small models from scratch in Section 2.10.

[3]A model with equal filters/layer or where the number of filters doubles when spatial size of input halves (this is a common default configuration known to work reasonably well).

2.9 Pruning performance depends on starting weights

A pruning algorithm can be defined as a function $P_u(I, D, t)$ where I is the initial model (architecture and weights), D is the target dataset, t is the target size or alternatively a stopping criterion, and $u(M)$ is a function that lists all prunable units in the model M. In typical machine learning tasks where we optimize a function f to fit a dataset D the goal is to create a model that generalises for data coming from a similar distribution to that of D, but does not overfit the target dataset. In pruning the starting model is an input to the algorithm as well as the data and it is not sufficient to evaluate a pruning algorithm on a single initial model.

It has been previously shown that pruning performance depends on the starting model (Blalock et al., 2020). In Figure 2.4 a simple experiment is used to further highlight the impact of the starting model on pruning. Baselines A, B, and C have been trained using the same hyper-parameters and data with the only difference being the random initialization. Each basline is a ResNet-110 network trained and pruned on CIFAR-10 in a block by block fashion until it matches the size of a ResNet-56 (halving the number of ResNet blocks). Four different pruning configurations have been used (activation-diff, activation-mean, mean weights and evaluate, explained in detail in Chapter 3) displaying the average and confidence. It is possible to observe that the baselines can be compared to each other in terms of performance at different pruning levels without much overlap on the validation set (the overlaps are larger on the test set but it is still clear that they are comparable).

The lottery ticket hypothesis (Frankle and Carbin, 2019) shows that for any over-parametrized neural network there exists a subset which can be trained to at least match the performance of the original. A network A is randomly initialised with weights $\mathbf{W_0}$, trained to weights $\mathbf{W}$ and pruned obtaining binary mask $\mathbf{M}$ (mask has value 0 for pruned and 1 otherwise, one value for each weight). The pruned network is reset to the random weights $\mathbf{W_0}$ keeping mask $\mathbf{M}$. The newly formed network could be a *winning lottery ticket*: a smaller network that trains to perform at least as well as the original, A, when trained to convergence. This suggests that pruning leads to different subnetworks based on the initial weights of each network. It also hints that the initial random initialization is a critical factor in which weights will be the most important after training. Furthermore keeping only the sign of weights in $\mathbf{W_0}$ and randomly reinitializing works better than simply reinitializing (Zhou et al., 2019).

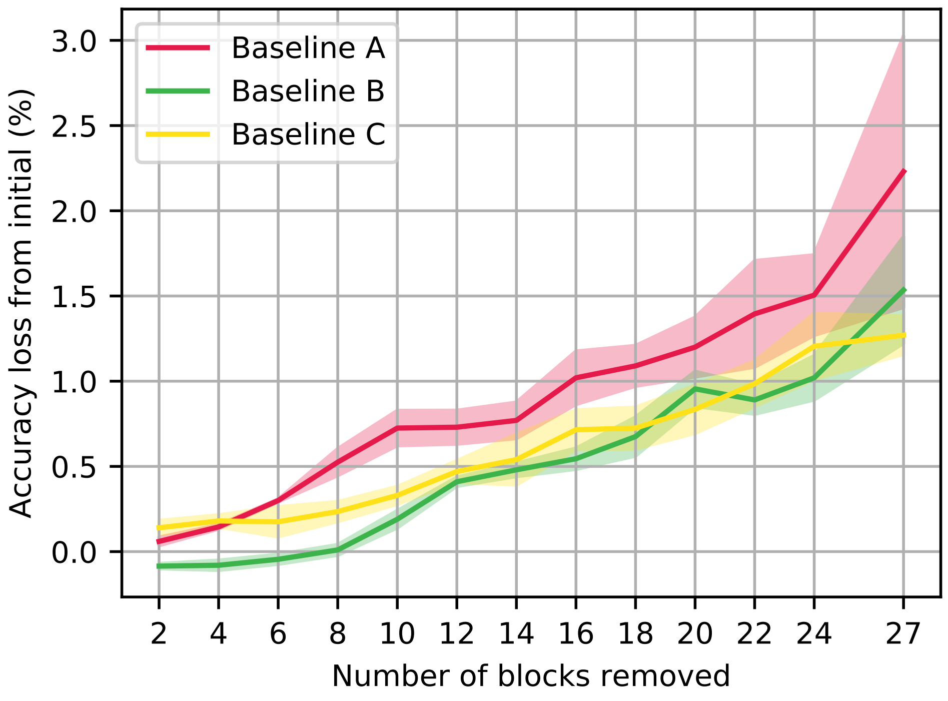


Figure 2.4: Three baselines pruned with four different pruning algorithms displaying the mean and confidence for each baseline. It can be observed that the baselines are *comparable* to each other regardless of pruning configuration, suggesting that pruning is indeed dependent on the starting model. Each baseline is a ResNet-110 trained with identical hyperparameters with different samples of the random initialization, each pruned block by block four times, each with a different method, from 54 blocks to 27 blocks. 50 epochs of fine-tuning is performed at every other step. Dataset: CIFAR-10. Showing the loss from initial model on the validation set.

2.9.1 Trained to be pruned

Dropout (Srivastava et al., 2014) is the idea of turning parts of a model on and off randomly during training as a form of regularization. It is equivalent to applying pruning and restoring at neuron level during training and using the full network on inference. Intuitively networks trained with dropout should be more robust to at least a small amout of pruning at the locations where dropout was applied during training.

Gomez et al. (2019) present targeted dropout where dropout is applied throughout training specifically to weights that are considered of low importance with a simple metric like magnitude of weights, effectively forcing the model to be robust to pruning specifically those weights post training. The results of targeted dropout demonstrate that dropout techniques can be effectively used as regularisation to train models that are robust to subsequent pruning. Targeted dropout yields accuracies of 68.8% top-1 accuracy on single crop ImageNet at 50% sparcity for ResNet-101 (6.7% accuracy loss from baseline) as well as a negligible accuracy loss of 0.05% (92.48% accuracy) for a ResNet-32 baseline on CIFAR-10 pruned at 90% of weights. A similar work is presented in Jia et al. (2018).

Neural networks with skip connections that are trained using training constructs which arbitrarily turn on and off (either fully or by multiplying with a scalar) parts of the network, such as those presented in Huang et al. (2016) and Yamada et al. (2019), intuitively would be more robust to pruning blocks. To test this, we have performed a small set of preliminary experiments using ResNet-110 networks on the CIFAR-10 dataset. They show that this only holds for a small number of pruning steps and fine-tuning after pruning such networks is more difficult.

Figure 2.5 displays the results of pruning 6 ResNet-110 simple baselines and 6 trained with linear decay stochastic depth training (Huang et al., 2016). For the first 15 blocks removed it seems that the stochastic depth training baseline is better suited for pruning, however, this changes as the pruning goes further and the performance degrades more than for the simple baselines. The pruning method is therefore inconsistent in behaviour depending on the amount of the network pruned and the method of initial training.

When fine-tuning is enabled with four training loops at 7, 14, 20 and the final step 27 blocks removed (half of the initial 54) the stochastic depth network is outperformed by the simple baselines by a significant margin in the ballpark of 5% on the validation set, with accuracies of approximatively $88-89\%$ and 93-95%. The large difference is likely caused by the multiplier p — the probability of the block being active — used in stochastic training; before fine-tuning the value of p for every block is reset to 1. A further experiment has been performed where stochastic depth training is performed at fine-tuning time however, once again, this does not improve the accuracy. Finally, an experiment where p was applied throughout fine-tuning as a regular parameter (same as in inference), but without any improvement.

This prelimilary experiment has been performed only to further highlight the importance of the starting weights and not to make a comprehensive statement about stochastic depth training and its impact on pruning. The conclusion is that the starting weights are an important factor which impacts the performance after pruning regardless of pruning method. Similarly, the initial training method does influence the pruning performance and this is highlighted by the difference in accuracy loss between the baselines trained with and without stochastic depth training. Most notably the impact is not consistent with the amount of pruning applied. In our experiment the stochastic training baselines are more robust to pruning a small number of blocks but the simple baselines perform better when more blocks are pruned.

A method of preparing a pre-trained convolutional neural network for pruning at filter level named auto-balanced regularization is contributed by Ding et al. (2018). It works by first running a pre-pruning fine-tuning loop which aims to move the representation capacity of the model to a small subset of its filters. The models is then pruned filter by

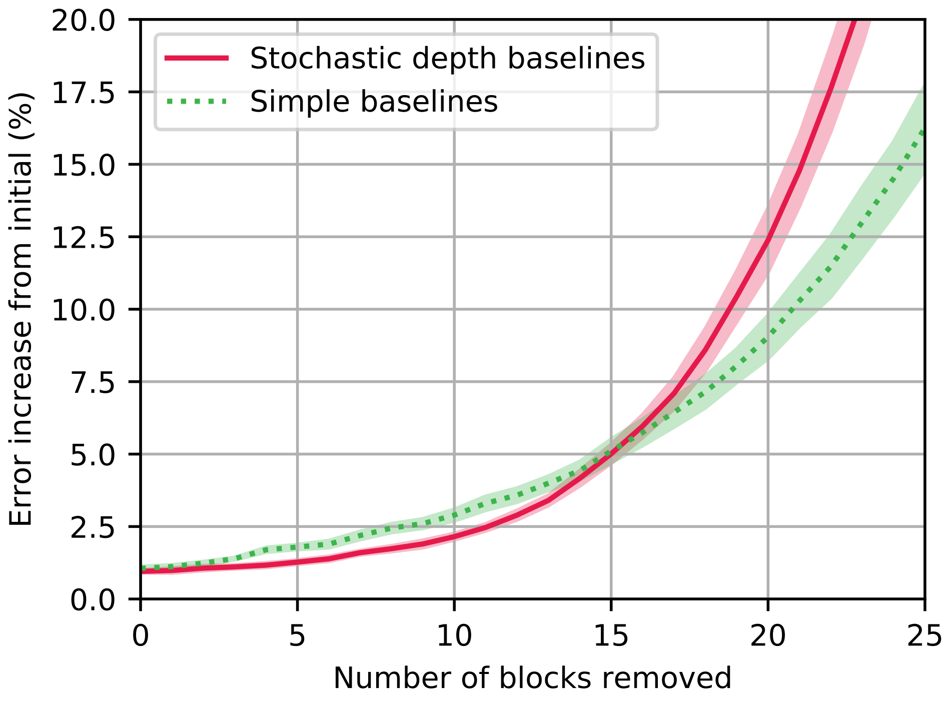


Figure 2.5: Six stochastic depth training and six regular baselines trained on CIFAR-10 and pruned block by block with no fine-tuning. All starting models are ResNet-110 with 54 basic blocks. The curves show the average reduction in test error percentage at each block as compared to the initial baseline with faded background representing the standard error. Observe that the stochastic depth training models are more robust to pruning a small number of blocks but the advantage is lost at around 15 blocks removed where the simple baselines perform better.

filter (with auto-balanced filter pruning and abreast advancing iterative pruning, another contribution of Ding et al. (2018)) and achieves a reduction in FLOPs of ResNet-56 by 60.86% with an accuracy loss of 0.99% on CIFAR-10. The error rate of the pruned network is 7.06% with the starting baseline at 6.07%.

2.10 Training small networks from scratch

Pruning is only a worthwhile investment as long as it outperforms training small networks directly from scratch or provides other benefits such as reduced overall training costs which can be especially visible when used for transfer learning applications where pruning a large pre-trained can be cheaper than training a smaller one from scratch.

It is important to acknowledge that pruning and fine-tuning are not always cheap computationally, especially so in cases where fine-tuning is applied at many steps.

Different types of pruning can also be used to find small architectures that train well on par with how the authors of Frankle and Carbin (2019) found *winning tickets*, the sub-networks that paired with the original random initialisation train to be competitive with the original network. This, however, requires training a large model once initially, restriction which can be relaxed by pruning from random initialization (Wang et al., 2020).

For structured pruning and larger scale datasets (ImageNet as opposed to CIFAR-10 and MNIST) Liu et al. (2018) finds that training from scratch is performing better than pruning in many cases, including in the cases where the architectures obtained by pruning are trained from scratch. The authors find contradictory evidence to that of Frankle and Carbin (2019), where architectures obtained by pruning trained from random initialization does perform better than the *winning ticket* (pruned architecture using original random weights).

Unstructured pruning applied before training is presented in Lee et al. (2018), where unimportant weights are found by using the gradient of a pruning mask. This is extended to structured pruning for convolutional filters and single neurons (van Amersfoort et al., 2020) by using smaller masks where each element corresponds to either a convolution filter output or output of a neuron. The networks are then trained using regular methods and for structured pruning the total training time is halved and inferrence time is reduced 3 times with a trade-off of 0.5% lower accuracy on CIFAR-10 for a VGG-19 model.

2.10.1 Networks designed for compute and size efficiency

Another point of comparison to keep in mind is that of networks that are designed to be small and efficient.

Fast models have been of interest in the field and are still a very active area of research. The speed of training and inference as well as reducing the size of models is critical for real-time applications, battery-powered devices and other applications. Many techniques, such as pruning, are focused on taking a large model and compressing it. Another approach is to design models to be small from scratch or to be able to scale them to the available processing or storage capacity for each application.

MobileNet (Howard et al., 2017), MobileNet V2 (Sandler et al., 2018) and most notably EfficientNets (Tan and Le, 2019), as well as others (Zhang et al., 2018; Ma et al., 2018), were designed with efficiency as a primary factor and aim to yield good trade-offs between accuracy, compute efficiency and model size.

Table 2.1: EfficientNet, MobileNet v2 and ResNet comparison summary. Latency is for a single image inference on CPU. Parameters are in millions, FLOPs in billions. All numbers are from the EfficientNet paper (Tan and Le, 2019).

Model	Top-1	Latency	Parameters	FLOPs
EfficientNet B0	77.1%	-	5.3M	0.39B
ResNet-50	76%	-	26M	4.1B
MobileNet V2	72%	-	-	0.3B
EfficientNet B1	78.8%	0.098s	7.8M	0.7B
ResNet-152	77.8%	0.554s	60M	11B

EfficientNets yield impressive performance per parameter values: EfficientNet B0 trains to 77.1% top-1 ImageNet accuracy with 5.3M parameters compared to 76% for ResNet-50[4] at 26M parameters, also summarized in Table 2.1.

2.11 Changing task or dataset via pruning

Transfer learning is shown to be a good fit for pruning. Rapidly removing the weights that have over-fitted the original target dataset is intuitively a beneficial step to retargeting a model for a new but related task.

Molchanov et al. (2016) focus on pruning for transfer learning — pruning a model trained on a large dataset using a smaller target dataset. It is unclear how this method compares with training a small model from scratch on the large dataset first and then fine-tuning it (without pruning) on the smaller dataset. The *scratch* result in the paper is a small model trained from scratch on the small dataset directly. The *scratch* model is then compared with models pruned using the small target dataset but originally trained on a large dataset (ImageNet). This is an unfair comparison as it is known that transfer learning improves the accuracy on smaller datasets.

In ThiNet (Luo et al., 2018), the authors recommend performing the pruning and fine-tuning on the larger dateset (in their case ImageNet), followed by further fine-tuning (without pruning, only for dataset adaptation) with the target dataset. There is no empirical comparison between this method and pruning directly on the target dataset.

Gordon et al. (2020) observe that BERT (Devlin et al., 2018) can be pruned to remove 30-40% of parameters and will not significantly affect pre-training loss or performance on target tasks. The authors claim that, overall, BERT performs better when pruned whilst pre-training and not when training with the target task.

[4]ResNet-50 as described in He et al. (2016) (for ImageNet) with bottleneck blocks (3 convolutional layers per block) and 4 groups of blocks (3 - 4 - 6 - 3) with 64 base filters for the first group, doubling at each subsequent group. In our pruning chapters we use ResNet networks (as described in the paper under the CIFAR section) with basic blocks, 16 base filters, and 3 groups with equal blocks.

Table 2.2: Summary of results from selected pruning works on CIFAR-10. The numbers are taken from the respective publications. FLOPs are $\times 10^8$. All methods perform convolutional filter pruning.

	Initial model			**Fine-tuning**	**Pruned**				
Name	**Params**	**Err%**	**FLOP**		**Params**	**Params-%**	**Error**	**FLOP**	**FLOP-%**
Li et al. (2016). Abs. weights selection method.									
ResNet-56-Pruned-B	0.85M	6.96	1.25	40 epochs	0.73M	13.7%	6.94	0.909	27.6%
ResNet-110-Pruned-B	1.72M	6.47	2.53	40 epochs	1.16M	32.4%	6.7	1.55	38.6%
VGG-16	15M	6.75	3.13	40 epochs	5.4M	64.0%	6.6	2.06	34.2%
He et al. (2018). L2 of weights selection method.									
SFP		6.32	2.54	1 epoch/it			7.1	1.21	52.30%
He et al. (2020). L2 of weights selection method.									
ASFP-30		6.32	2.53	1 epoch/it			6.63	1.5	40.80%
ASFP-40		6.32	2.54	1 epoch/it			6.9	1.21	52.30%

Table 2.3: Summary of results from selected pruning works on ImageNet. The numbers are taken from the respective publications.

	Initial model			**Fine-tuning**	**Pruned model**			
Name	**Params**	**Accuracy**	**FLOPs**		**Params**	**Params-%**	**Accuracy**	**FLOPs**
Luo et al. (2018). Selection method: statistics in next layer. Prunable unit: conv filters.								
ThiNet-50 ResNet-50	25.56M	72.88% top-1 91.14% top-5	7.72B	1 eppoch per layer and 10 epochs for the final layer	12.38M	51.56%	71.01% top-1 90.02% top-5	3.41B
ThiNet-30 ResNet-50	25.56M	72.88% top-1 91.14% top-5	7.72B		8.66M	66.11%	68.42% top-1 88.30% top-5	2.20B
ThiNet-Conv VGG-16	138.34M	68.34% top-1 88.44% top-5	30.94B		131.44M	4.98%	69.80% top-1 89.53% top-5	9.58B
ThiNet-GAP VGG-16	138.34M	68.34% top-1 88.44% top-5	30.94B		8.32M	93.98%	67.34% top-1 87.92% top-5	9.34B
ThiNet-Tiny VGG-16	138.34M	68.34% top-1 88.44% top-5	30.94B		1.32M	99.04%	59.34% top-1 81.97% top-5	2.01B
Han et al. (2015). Selection method: weight value (thresholding). Prunable units: weights.								
VGG-16	138M	68.5% top1 88.68% top5		At each pruning iteration.	10.3M	92.53%	68.66% top-1 89.12% top-5	
Li et al. (2016). Selection method absolute weights. Prunable unit: conv filters.								
ResNet-34 Pruned-B	21.50M	73.23% top-1	3.64B	20 epochs	19.3M	10.80%	72.17% top-1	2.76B
Hu et al. (2016). Selection method: ApoZ. Prunable unit neuron.								
VGG-16	21.50M	68.36 top-1 88.44 top-5		5k iterations, batch size 256	8.3M	61.39%	70.44 top-1 89.79 top-5	
Molchanov et al. (2016). Selection method "Taylor criterion". Prunable unit: conv filters.								
VGG-16	21.50M	89.3% top-5	30.96B	100 updates/iter and,			87.0% top-5	11.5B
VGG-16	21.50M	89.3% top-5	30.96B	5 epochs at shown rows.			84.5% top-5	8.0B
\cite{NIPS2017_7071}. Selection method L-OBS. Prunable unit: neuron.								
VGG-16	21.50M	68.34% top-1 89.88% top-5		8.63 ×10^4 Iterations	1.61M	92.50%	67.98% top-1 89.03% top-5	
VGG-16	21.50M	68.34% top-1 89.88% top-5		none	1.61M	92.50%	62.68% top-1 85.18% top-5	
Wang et al. (2019). Pruned 50% of blocks (params not reported). Prunable unit ResNet Block. Selection: DBP								
R101-R50 ResNet-101		77.37% top-1					76.97% top-1	

2.12 Chapter summary

In this chapter we have reviewed a few major contributions in pruning and placed pruning algorithms and methods in different categories. The most important aspects of pruning algorithms are the choice of prunable unit — weights, neurons, convolutional filters, or others —, and the fine-tuning settings.

Pruning can be applied in different contexts. It can be applied iteratively layer by layer, or globally all at once. It can be applied on a model that has been pre-trained, or a model that was initialized from random. A pruning method may or may not require a pre-processing step for the model, such as fine-tuning with special regularization.

Fine-tuning to recover lost accuracy can be done at every pruning iteration, only at the end of pruning, or anywhere in between.

We have presented two basic baselines to serve as initial comparison points for pruning method: random pruning and oracle pruning. Random pruning is commonly used, and oracle pruning was introduced and used by Molchanov et al. (2016). It is important to note that random pruning must be done more than once, and with the same fine-tuning schedule as the pruning algorithm to be compared with. Accuracy before fine-tuning is often but not always a good indicator of after fine-tuning performance.

Runtime, or dynamic, pruning is pruning that is applied at inference time. The typical setup is a meta-model which outputs a mask for the large model. If the large model is large enough, the meta model small enough, applying the masks on the large model gives a noticable increase in inference speed. Similar types of runtime or dynamic pruning may also be applied at training time, to increase both training and inference speed.

Empirical findings have been mostly omitted throughout the chapter as the focus is on describing and introducing different pruning techniques. For completeness, some key results from works cited throughout this chapter are included in Table 2.2 for CIFAR-10 and in Table 2.3 for ImageNet.

Chapter 3

Pruning ResNet networks block by block

Deep learning has been proven to be the leading method in terms of the performance for many application domains. Until recently, the trend for new architectures and models that push the state of the art forward was a steady increase in model size (in number of parameters, number of layers, or multiply-add operations) and therefore memory and computational requirements (Khan et al., 2020). Many state of the art deep learning models have the disadvantage of requiring a large amount of memory and processing power to run even in inference, making them hard to use on CPUs or mobile devices. In the last few years the size of deep neural networks has become more widely discussed in the literature, with venues such as TinyML[1], the growing maturity of frameworks that make inference on low power devices accessible, such as Tensorflow-Lite (Abadi et al., 2015) and PyTorch Mobile (Paszke et al., 2019), and a growing body of literature that focuses on small models.

Small models are ideal for applications where latency is critical, as well as where memory usage and power consumption are of concern. Small models can be easily deployed on many devices like embedded hardware, system-on-a-chip computers, mobile phones, and others. If the accuracy loss is small enough small models can be advantageous on servers as well since CPU computing is cheaper than GPU or other custom AI hardware[2] like TPUs[3]. A large and growing body of works on finding efficient deep neural network model architectures exists, focusing on creating model architectures that give good ratios of complexity to accuracy. EfficientNet (Tan and Le, 2019), MobileNet (Howard et al., 2017; Sandler et al., 2018) are great examples of such works.

[1]https://www.tinyml.org

[2]Based on prices of major cloud computing providers as of 21 May 2018, estimated based on online predictions (not batched jobs).

[3]TPU: Google's Tensor Processing Unit: https://cloud.google.com/tpu/.

This chapter presents a simple iterative method of pruning convolutional neural networks with residual connections (architectures based on ResNet (He et al., 2016)). The method removes entire ResNet blocks as opposed to pruning filters or single convolutional layers, which is a bigger choice of prunable unit than commonly found in the literature. This work explores different block selection methods and fine-tuning settings whilst comprehensively comparing the accuracy, computing and storage efficiency of pruned models and models trained from scratch.

Most pruning works in the literature are focused on using smaller prunable units such individual weights or neurons. Pruning work of this last decade adds in various methods of pruning filters and features maps from convolutional layers, as noted in Chapter 2. Residual connections and the structure of ResNet (blocks connected to each other directly and indirectly via residual connections) welcomed the idea of removing entire blocks from a network. It has been noticed (most notably by Veit et al. (2016)) that a small number of ResNet blocks can be removed or reordered with a small impact on performance. The same does not hold true for feedforward networks without any residual connections, like VGG (Simonyan and Zisserman, 2014) where one layer feeds only into the next.

The motivation for block by block pruning and a reminder of the ResNet archtecture are included in Section 3.1. Section 3.2 presents our CIFAR-10 baseline models.

In Section 3.3, seven block saliency metrics used for block by block pruning are defined and evaluated. They are **oracle**, **activation mean**, **activation change**, **activation change plus one**, **block gradient**, **weights mean**, and **random choice**. Section 3.4 evaluates the **oracle** block selection method on checkpoints taken throughout the training of an initial model.

Different methods of placing fine-tuning loops throught the pruning process (fine-tuning schedules) are introduced and evaluated in Section 3.5. Section 3.6 discusses fine-tuning parameters that do not depend on the schedule: learning rates, random noise, and freezing the classifier.

The validation set was used for evaluating the saliency of blocks, which was done as the validation set is often used to select the best models or checkpoints. Nonetheless, this raises the concern of overfitting the validation set. The issue is addressed in Section 3.7.

We compare our results with similar works in Section 3.8, and finally conclude with Section 3.9.

3.1 Intuition and background

A very successful CNN architecture is ResNet (He et al., 2016). It introduces the idea of residual connections for convolutional neural networks. The ResNet architecture is

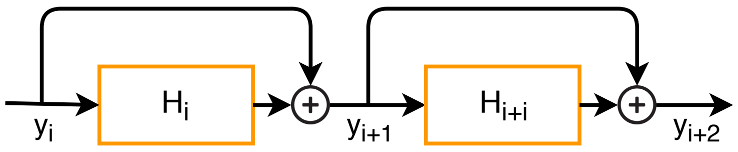


Figure 3.1: Two ResNet blocks, H_i and H_{i+1}, connected to each other showing the basic idea behind the ResNet architecture.

composed of ResNet blocks (also called residual blocks). We denote y_i as the input of the i-th residual block H_i, and its output as

$$y_{i+1} = \sigma(H_i(y_i) + y_i),$$

where $\sigma(\cdot)$ is a non-linear activation function, typically a ReLU. An illustration of two ResNet blocks is presented in Figure 3.1. Many modifications of the ResNet architecture have appeared since it was first presented in 2015, like ResNeXt (Xie et al., 2017) and DenseNet (Huang et al., 2017). Residual connections are now a common feature in recent neural network architectures (Sandler et al., 2018; Xie et al., 2017; Huang et al., 2017; Tan and Le, 2019).

There are two different types of ResNet blocks. The basic block is composed of two convolutional layers of equal filters, both 3x3. The bottleneck block is composed of three convolutional and the first two layers have a smaller number of filters, having sizes 1x1, 3x3, and 1x1, respectively. We use basic blocks in all our experiments.

The first block of each block group performs a downsampling by using 2D convolutions with stride 2. The residual connection needs to also be downsampled for the sizes to match for the element-wise addition. This is achieved by applying a 1x1 convolution with stride 2 on the residual connection, as described in Hu et al. (2016). To reduce any ambiguity, a full representation of the architecture we use in this chapter is shown in Figure 3.2.

Veit et al. (2016) shows an analysis of the ResNet architecture that suggests ResNet models act like an ensemble of 2^N smaller networks, where N is the number of blocks in the network ($N = 16$ for ResNet-50, $N = 32$ for ResNet-101 with Bottleneck blocks; $N = 54$ for ResNet-110 with basic blocks). This is shown by removing blocks from the network and obtaining a slow decrease in performance with the number of blocks cut. We obtain similar results, shown in Figure 3.5. They also observe that re-ordering blocks impacts the performance of the network less than it does for other CNN architectures that have no residual connections like VGG (Simonyan and Zisserman, 2014).

Based on these observations we propose that ResNet architectures are robust to pruning entire blocks, and that with appropriate retraining iterations we can obtain very small networks while not affecting the model performance.

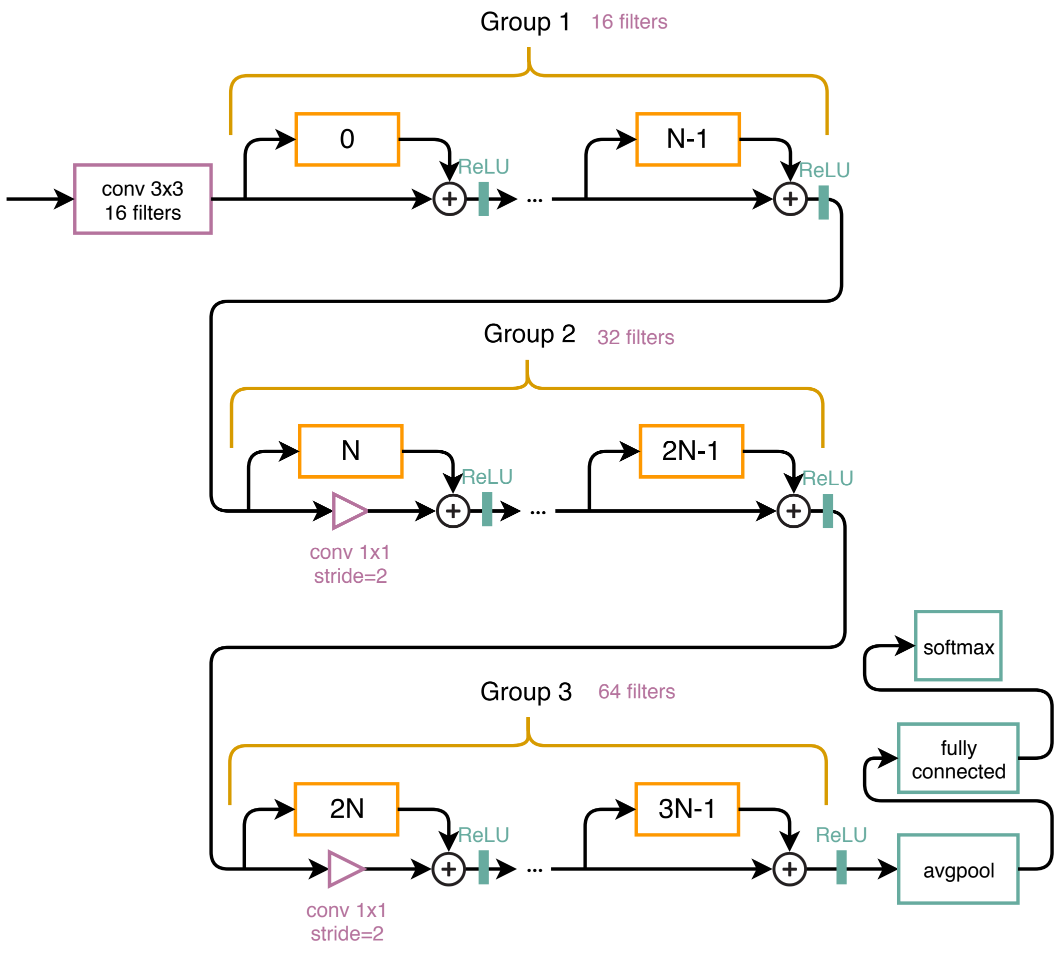


Figure 3.2: A representation of the ResNet model architecture we use in this work for the CIFAR-10 dataset. Batch normalisation omitted from the figure – it is applied before ReLUs.

3.1.1 Pruning ResNet blocks

In this section we present a simple yet effective method of pruning entire blocks from a ResNet[4] network while keeping the accuracy loss to a minimum.

We use the following iterative process:

Step 1 train a ResNet network on a dataset,

Step 2 evaluate the importance of each ResNet block,

Step 3 remove the least important ResNet block(s),

Step 4 optionally, fine-tune the network (re-train for a number of epochs),

[4]This method works with more recent variations of the ResNet network like ResNeXt. In this report we have only used ResNet.

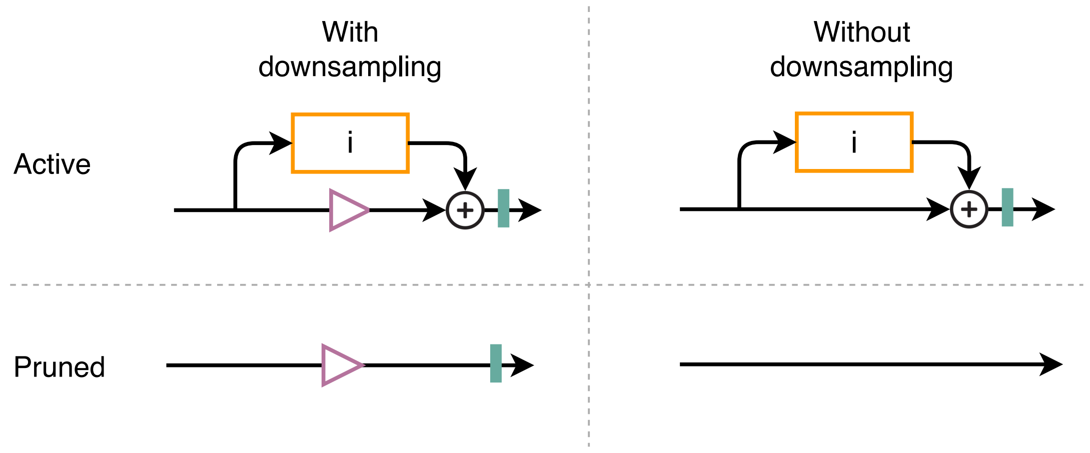


Figure 3.3: The effect of pruning a ResNet block with and without downsampling. The residual connection is drawn at the bottom with the block at the top, suggesting it can be inserted and removed.

Step 5 repeat from **Step 2** until enough units are removed.

The output of a pruned ResNet block is just the output of the previous block, $y_{i+1} = y_i$. In the case of the blocks that perform downsampling, the downsampling is still applied on the residual connection. Visually this is shown in Figure 3.3.

There are two main areas to be explored in our block by block greedy pruning method. The first is the block selection algorithm: given a ResNet, select the next block to prune. The second is fine-tuning the model after it has been pruned. It addresses how much fine-tuning to perform and choosing all the details and hyper-parameters related to fine-tuning, as well as keeping track of the cost of fine-tuning.

3.2 Baselines

For the majority of experiments in this chapter we use the same ResNet configurations as He et al. (2016) uses on CIFAR. For training the models from scratch we we use the same settings as Huang et al. (2016) (constant depth). We train for 500 epochs using the SGD optimiser with weight decay 10^{-4}, Nesterov momentum 0.9, and a learning rate schedule starting with 0.1 and dividing by 10 at epochs 250 and 375. The batch size is 128. We also follow the standard data augmentation for CIFAR datasets, padding by 4 pixels, random crop of 32 by 32 and random horizontal flip. Our results are slightly better than that of He et al. (2016) and Huang et al. (2016) due to using a 1x1 convolutional layer as opposed to a pooling layer for downsampling, and shown in Table 3.1.

For all experiments, unless otherwise stated, the CIFAR-10 validation set is created by taking the first 10% of the data points from the training set. The validation set and the remaining training set are the same for all experiments, and they are roughly balanced

Table 3.1: Baseline points trained on CIFAR-10. Baseline points are our results from training the models from scratch. Huang et al. column shows constant depth results. All numbers are the CIFAR-10 test error %.

	Baseline points	**He et al. (2016)**	**Huang et al. (2016)**
ResNet-110	5.71 (6.0±0.3)	6.43 (6.61±0.16)	6.41
ResNet-56	6.21	6.97	-
ResNet-44	6.63	7.17	-
ResNet-32	6.68	7.51	-
ResNet-20	7.54	8.75	-

Table 3.2: Data points per class after splitting the CIFAR-10 training set. Validation and training sets used by all experiments (unless stated) is shown in the first 2 rows. The next 3 rows, marked with *, show the split for the experiments that included a pruning set of an extra 10% of the original training set. The test set is the same for all experiments.

	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
train	4495	4540	4481	4514	4481	4512	4481	4514	4480	4502
val	505	460	519	486	519	488	519	486	520	498
train*	3995	4026	3968	3984	4001	4063	3970	3999	3975	4019
val*	505	460	519	486	519	488	519	486	520	498
prune*	500	514	513	530	480	449	511	515	505	483
test	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000

(see Table 3.2 and the code to reproduce experiments using the same split). The data taken from the training set and put into the validation or *pruned* set was never used for training or fine-tuning. We acknowledge that a balanced split (4500 training and 500 validation data points per class) may improve the error rates repored throughout this thesis although no experiments were run using that configuration.

A baseline point has been trained for every equal block configuration from the size of ResNet-110 (54 blocks, 18 per group) to ResNet-20 (9 blocks, 3 per group). The results for both the test and validation set of CIFAR-10 are shown in Figure 3.4. Training was performed similarly to the baselines in Table 3.1, except training was performed for 400 epochs only. This reduction in number of epochs was applied to reduce runtime and because we did not see any improvement in performance after epoch 400 in the previous baselines.

We acknowledge that the baseline accuracies obtained are not state of the art and they could be improved by further optimizing the hyperparameters, however our focus is on pruning block by block and not on reaching state of the art results.

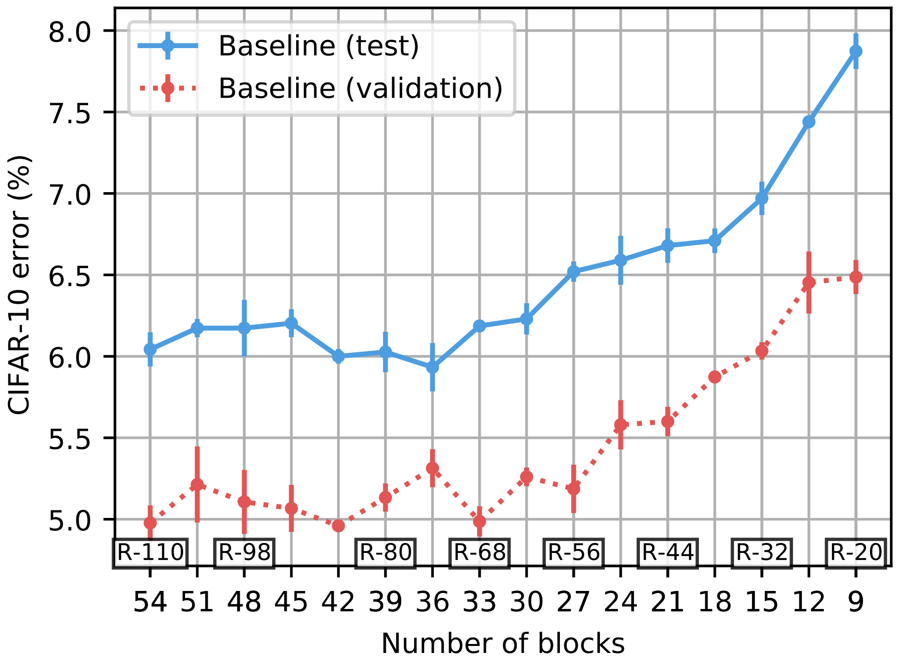


Figure 3.4: Standard ResNet architectures with equal number of blocks per group trained on CIFAR-10 from scratch. Each model has 3 groups of blocks.

3.3 Block selection methods

A simple way to estimate the importance of a ResNet block is to temporarily remove it and evaluate the model performance on the validation set. The higher the accuracy of the model is without a block, the less important the block is. This method of block selection is what we refer to as oracle pruning in Chapter 2. Since oracle pruning is used to mean that the evaluations are performed at every pruning step we also use the term *evaluate* to refer to this method of estimating block saliency but where the saliency is reused.

This method has the disadvantage of requiring an evaluation run for each possible block to remove. For ResNet-110 it requires 54 runs at the first iteration, 53 runs at the second iteration, totalling 2880 evaluations[5]. This process is fast to compute on small datasets like CIFAR-10 (1 minute for all blocks in ResNet-110) and CIFAR-100 (Krizhevsky and Hinton, 2009) but can be problematic on larger datasets such as ImageNet (Deng et al., 2009).

The performance issue of this block selection algorithm can be addressed by reusing the calculated scores for more than one iteration or by using less data to perform the

[5]Each evaluation is slightly cheaper than the previous one since there are less blocks. Allocate the cost of 1 to an evaluation with all 54 blocks and normalise every evaluation by this cost, the total number of *full cost evaluations* is $\sum_{i=10}^{54} \frac{i^2}{54} \approx 994$. This approximation does not take into account that the real cost of evaluation is not only determined by the number of blocks.

evaluation. The feasibility of reusing the scores by only computing them every 2, 4, 8 or 16 steps as well as only computing it once for a full pruning run has been evaluated. On an experiment without fine-tuning, re-calculating the scores every four steps is almost as good as evaluating at every pruning step. The validation and test accuracies on CIFAR-10 for a ResNet-110 (54 blocks) pruned until it reached the same number of blocks as ResNet-20 (9 blocks) are shown in Figure 3.5. Notice that lower frequencies of saliency calculation result in higher error rates and the differences are more noticable when more blocks have been pruned. However, the differences in accuracy are low for re-computing saliency every 2 or 4 steps, suggesting that it is possible to cache saliency computation for a few pruning steps without a major penalty in performance. We deduce that the relative importance of blocks changes as blocks are removed, even without any fine-tuning.

3.3.1 Comparing block selection algorithms

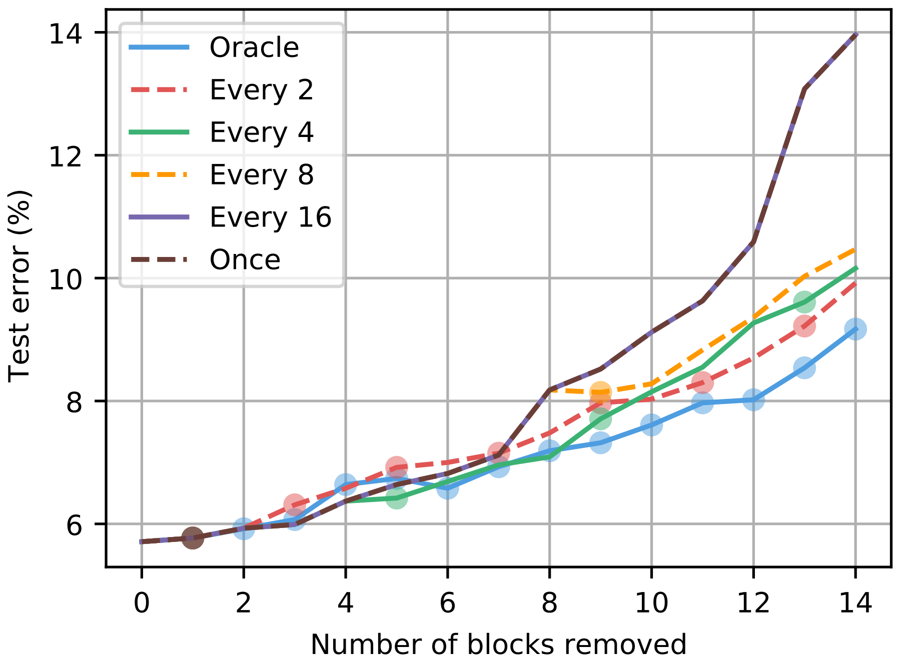
Block selection algorithms, when they are run on the same initial model and without fine-tuning, can be compared by the saliency scores they compute and the blocks they select to prune at each step. At each pruning step one block is removed. At step k there are k removed blocks.

A trivial method of comparing two block selection algorithms is the model accuracy on some dataset after pruning. This is not suitable if the intent is to compare the actual choice of blocks as opposed to model performance. For this purpose the set of removed blocks can be compared between two pruning runs: a similarity score can be computed based on how many common blocks both algorithms selected up to pruning step k as compared to k, the total number of blocks removed.

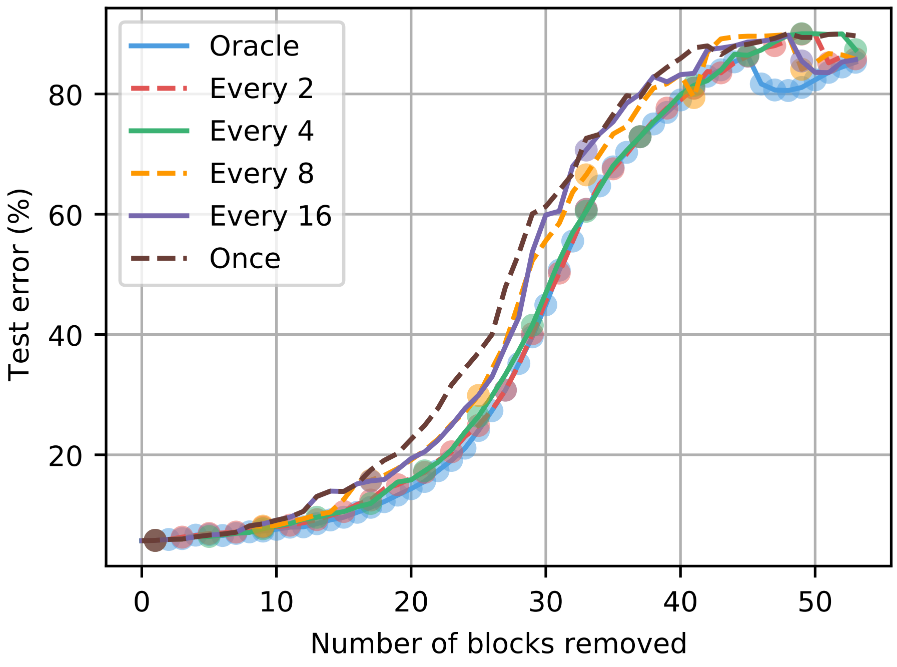
The trivial pruned blocks set comparison above does not take into account the order in which the blocks have been removed, which is important when fine-tuning is used. It also ignores the estimated saliency of the remaining blocks.

For a model with n blocks at step k, the ranking of the blocks for a block selection algorithm is the concatenation of all removed blocks in the order they have been removed followed by the remaining blocks sorted by saliency (at step k the block at position $k+1$ is removed). Using Kendall's tau (Kendall, 1945) as the metric of similarity between two pruning runs and the ranking defined above we obtain a measure of similarity between two block selection algorithms at each pruning step. This ranking allows for comparing a block selection algorithm based on both the past selections (the blocks that have already been removed) and the saliency of the remaining blocks. Kendall's tau metric ranges between 1 for strong agreement (blocks chosen in similar order) and -1 for strong disagreement.

In Figure 3.6 the similarity based on Kendall's tau is used to compare evaluation-based saliency metrics computed at every 2, 4, 8, 16 steps or only once against oracle pruning.



(a) Zoomed in on the first 14 out of 54 blocks removed.



(b) Full view up to 45 out of 54 blocks removed.

Figure 3.5: Oracle pruning of a ResNet-110 on CIFAR-10 without fine-tuning comparing different frequencies of computing the saliency of each block. *Oracle* is computing the score at every step. The error percentage shown is for the test set of CIFAR-10. The validation set was used for block selection. Faded circles highlight the steps where computing of saliency occurred.

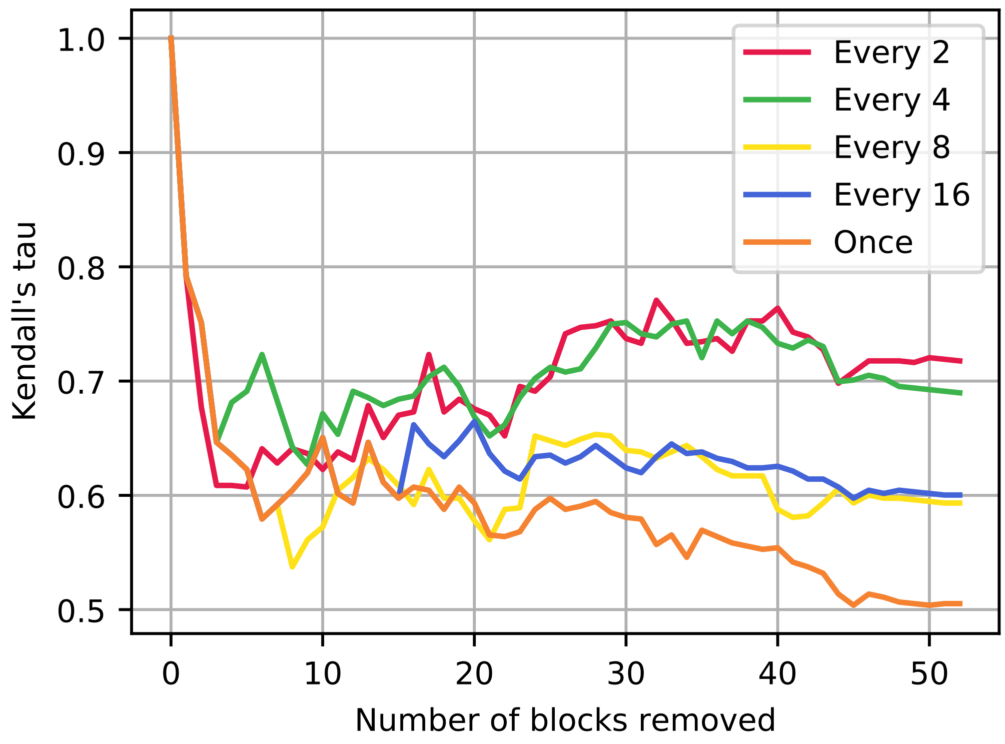


Figure 3.6: Kendall's tau correlation between oracle pruning and reusing the evaluation saliency metric (recomputing once, at every 16, 8, 4, or 2 steps). Kendall's tau similarity ranges from 1 to -1 for strong agreement to strong disagreement, respectively. The first step has a similarity of 1 since all methods shown perform exactly the same saliency metric for the first block.

There is a clear agreement between the metrics compared and it can also be observed that the longer the saliency computation is reused the lower the similarity with the oracle, as expected from the previous accuracy plots.

Another way to estimate the stability and reusability of saliency scores is to compare the ranking of pruning run as defined above with the ranking of the same pruning obtained by the list of all pruned blocks at the end of pruning. If the similarity is high then the saliency scores are highly reusable throughout the pruning process. Another use of this comparison is to check how much the estimated saliency of blocks changes in relation to fine-tuning.

3.3.2 Other block selection methods

More block selection methods and saliency scores have been evaluated as part of this work. This subsection introduces them with a small comparison between them as well as their correlation to oracle pruning.

Weights mean scores the blocks by their mean absolute weights in convolutional layers such that the block with the lowest value is removed first. This method does not require any data.

Activation mean method uses one forward pass with the validation set to compute the average absolute activation of each block. The block with the lowest value is removed first.

Activation change scores the block i based on the impact it has on the input of the next block[6]. This is estimated by computing the mean squared error between y_i (input if block i is pruned) and $ReLU(H_i(y_i) + y_i)$, the input if the block is present. The block with the lowest value is removed first.

Activation change plus one is similar to *activation change* but the mean squared error is applied a block after the removed one, effectively using the output of block $i+1$ to evaluate the saliency of block i.

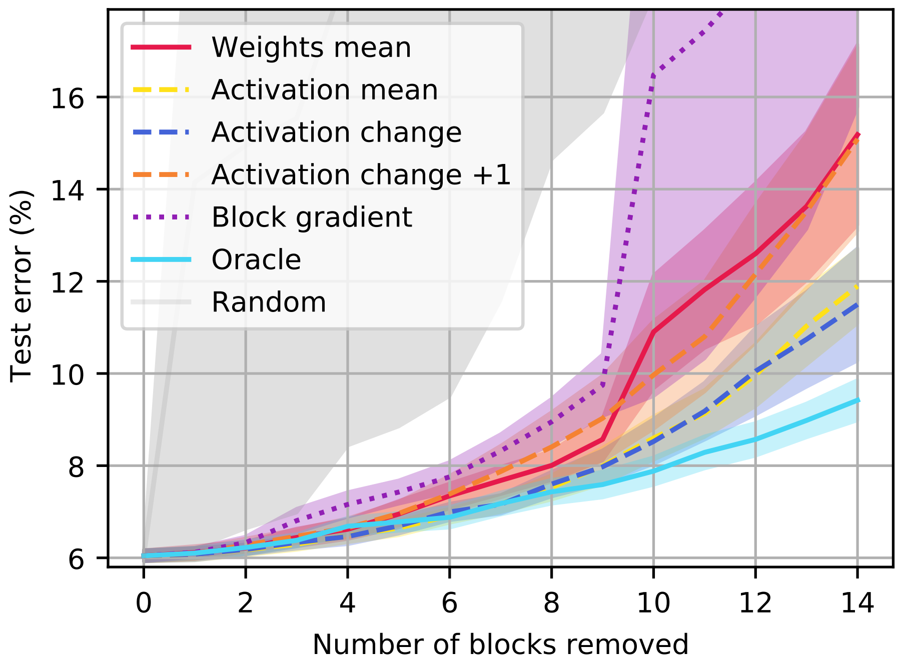
Block gradient magnitude is the method presented by van Amersfoort et al. (2020) but our units are the outputs created by ResNet blocks not individual filters. Our mask $\mathbf{m}$ has one element for each ResNet block. We compute $g = \frac{\partial \text{loss}}{\partial \mathbf{m}}$ and the score for block i is $s_i = \text{abs}(g_i)$, pruning the lowest scores by thresholding or desired target size. This block selection method was introduced as a single-shot pruning before training method and was not originally used in the context of greedy iterative pruning. For iterative greedy pruning, we prune the block with lowest s_i at each step.

Random choice where blocks are simply picked to be removed at random is used in certain parts of this work for comparison.

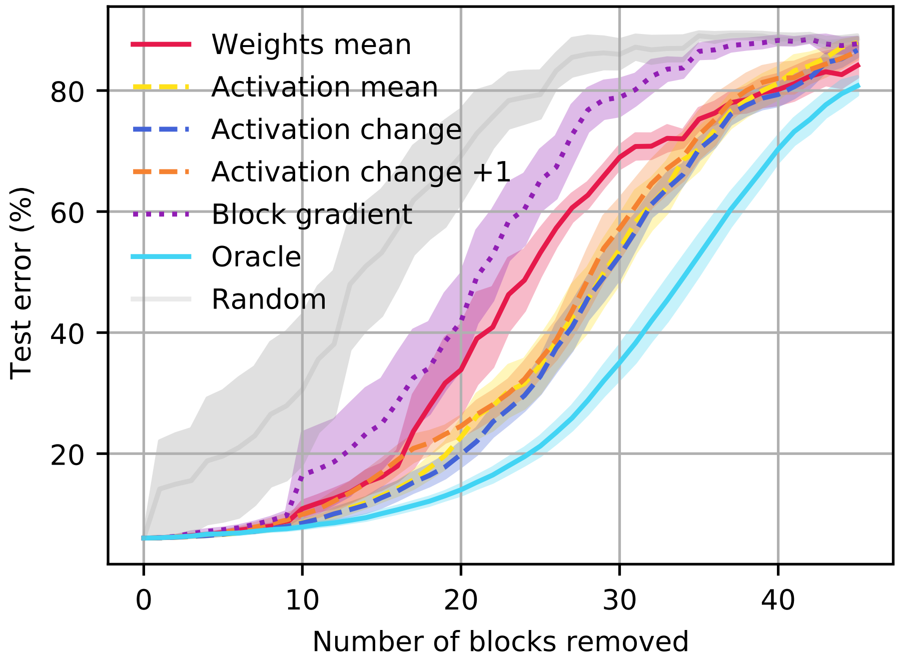
Figure 3.7 illustrates the performance of each of the block picking methods described above in a similar experiment to that of Figure 3.5: pruning a ResNet-110 block by block using each of the above block selection methods with no fine-tuning. Evaluating at every step gives optimal choices but using the activation change method gives a better tradeoff between computational cost and performance. The magnitude of the weights alone gives the worst performing results, confirming the intuition that using data for block selection results in superior choices when compared to no data methods. The Kendall's tau correlation between **weights mean**, **activation mean**, **activation diff**, **block gradient**, and **oracle pruning** is shown in Figure 3.8. As expected from the performance of the pruned models, the saliency metrics that are most similar to oracle pruning are the ones based on activations.

Throughout the rest of this work, unless otherwise specified, we compute the saliency of blocks at each pruning step using the evaluation (oracle) method. We find that the cost

[6]Effectively comparing the activations of the network at the place of output of block i, but to avoid confusion we use the term *input of next block* as opposed to *output of current block* since there is no output of current block when it is removed.



(a) Zoomed in on the first 14 out of 54 blocks removed.



(b) Full view up to 45 out of 54 blocks removed.

Figure 3.7: Pruning a ResNet-110 model on CIFAR-10 without fine-tuning comparing different block selection methods. The error percentage shown is for the test set of CIFAR-10. The validation set was used for block selection. Faded circles highlight the steps where computing of saliency occurred. Lines plotted are average results from 6 different baselines and faded highlight shows confidence.

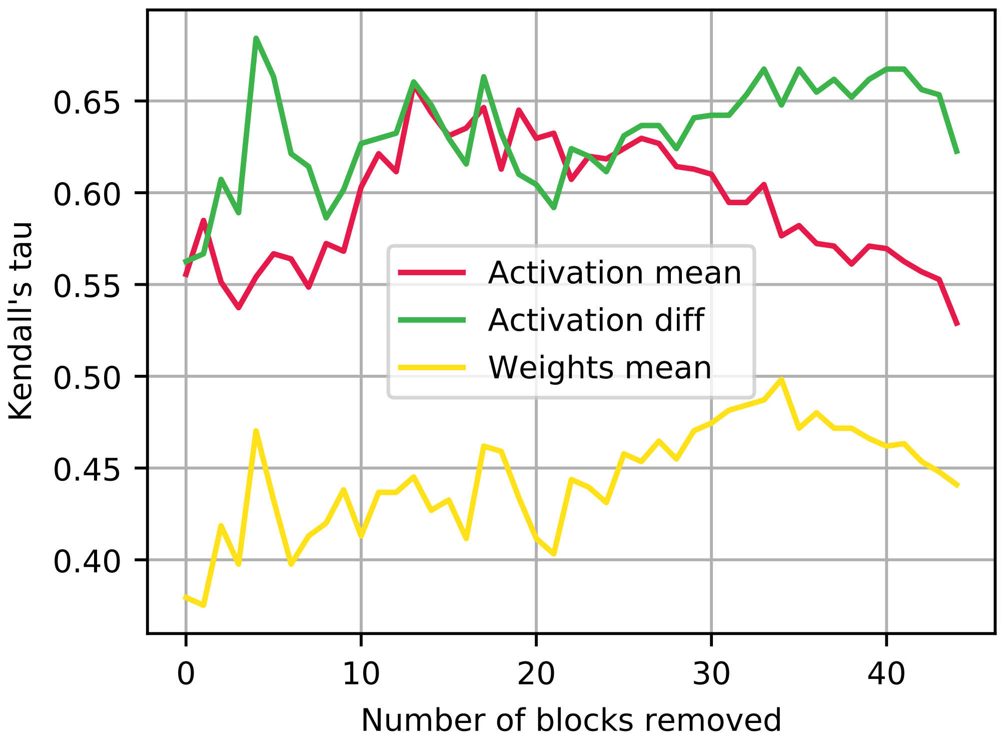


Figure 3.8: Kendall's tau correlation between oracle pruning and other saliency estimation methods: activation diff, activation mean and weights mean.

of this saliency metric is not a bottleneck in our setup since fine-tuning takes significantly more time. Using an NVIDIA GeForce GTX 1080 Ti GPU, for the initial pruning step evaluating 54 blocks, it takes approximatively 1 minute to compute the oracle saliency metric to remove one block. For the last pruning step, from 10 blocks to 9 blocks, it takes 7-8 seconds. As comparison, fine-tuning a network with 9 blocks for 50 epochs takes roughly 10 minutes, or 21 minutes for a network with 27 blocks (pruned 50% from ResNet-110).

When fine-tuning is added, both activation change, oracle pruning, yield similar results. This is illustrated in Figure 3.9 where pruning is performed six times for each selection method, each time with a different initial model. Fine-tuning was performed at every other pruning step for 50 epochs.

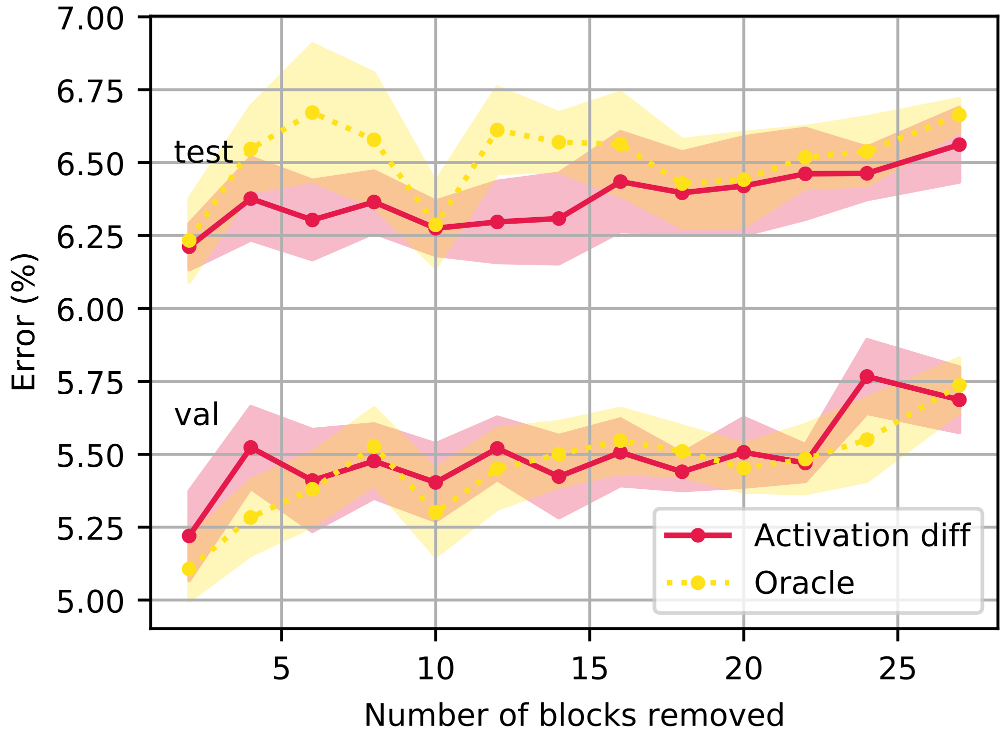


Figure 3.9: Fine-tuning every two pruning steps for 50 epochs using activation diff and oracle group selection methods. Initial model ResNet-110. Observe that their performance levels are similar. Showing the mean and standard error for 6 different pruning runs (different starting model) for each series.

3.3.3 Performance correlation before and after fine-tuning

In this section the hypothesis that the performance of a pruned model before and after fine-tuning are correlated is tested. The purpose of this test is that our *oracle* selection method relies on the performance of a model before fine-tuning to select which blocks to prune.

Since we already had many pruning experiments (264 used, all from ResNet-110) with different block selection methods and different fine-tuning configurations, we aggregated the CIFAR-10 validation accuracy before and after fine-tuning for all data points that we had available, a total of 1,459 fine-tuning loops after filtering for the number of epochs used (reporting best epoch accuracy) to be between 10 and 50, inclusive. A number of 1,311 unique sets of blocks removed were found, from which 494 unique block groups configurations.

A scatter plot of the before fine-tuning and after fine-tuning accuracies is in Figure 3.10. The Pearson correlation is 0.59. Although we have used a large number of data points, the data is biased due to the fact that many of our experiments were performed with oracle pruning (this explains the crowded area in the top right hand side of the plot). Another

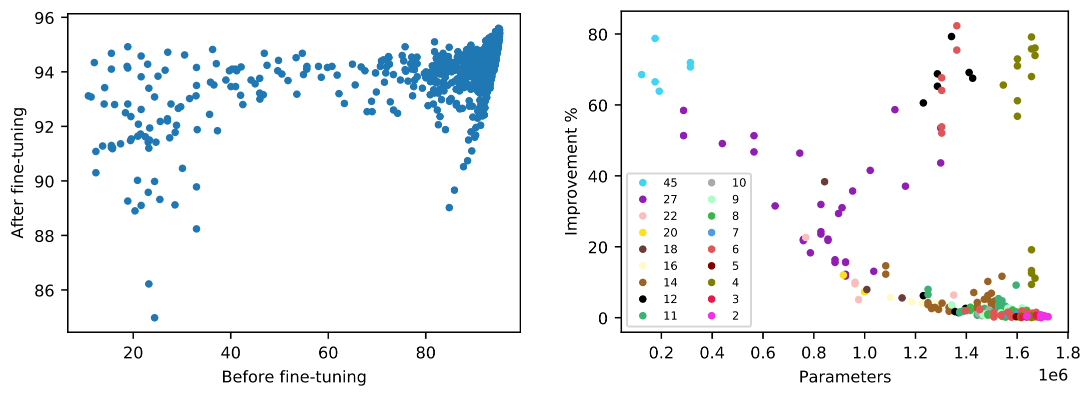


Figure 3.10: Left: CIFAR-10 validation accuracy before and after fine-tuning using a large collection of pruning configurations: 1,459 fine-tuning loops from 264 pruning experiments. Pearson correlation is 0.59. Right: Relative improvement on CIFAR-10 validation set after fine-tuning for 50 epochs (choosing best) over number of parameters of the pruned network. The legend shows the number of blocks removed. Only single-shot runs have been selected.

factor adding bias is the fact that we have more data points with a smaller number of blocks removed, which on average perform better before (and after) fine-tuning.

The observation we make is that a network with a high accuracy before fine-tuning is likely to yield a high accuracy after fine-tuning, but the correlation is not strong. The heuristic of choosing the best performing network before fine-tuning (oracle pruning) gives good results because it is aggressively avoiding possible bad choices, not because it finds good ones.

It is also worth noting that a very large range of before fine-tuning accuracies starting as low as 20% yield after fine-tuning accuracies of 91-95%. The range of after fine-tuning accuracies does not change very much, but there is a small uptrend at very high before fine-tuning accuracies (85%+).

The accuracy of a model before fine-tuning is a weak indicator of the accuracy after fine-tuning, but it does not give information about how likely a subnetwork is to train well.

3.3.4 Block selection methods with fine-tuning enabled

The previous section, Section 3.3.3, highlighted that the correlation between the accuracy of pruned models before and after fine-tuning is weak. In this section we seek to answer whether any of our block selection methods has a clear advantage over others after fine-tuning.

All block selection methods we used are compared with fine-tuning enabled. We used a fixed fine-tuning schedule for each pruning experiment: four fine-tuning loops uniformly spread across all pruning steps from ResNet-110 to ResNet-27. Precisely, fine-tuning at 6, 12, 18, and 27 blocks removed. Six initial ResNet-110 models were used for each of the seven block selection methods compared. In Figure 3.11a we show the results after every fine-tuning iteration, and Figure 3.11b shows the same experiments at every pruning step (for brevity, only a subset of selection methods is plotted in the second figure).

All fine-tuning was done with the same configuration. SGD optimizer with a learning rate schedule starting at 0.1, then 0.01 from epoch 10, 0.001 from epoch 20. Training for 50 epochs in total, and restoring the weights from the best epoch on the validation set. Nesterov momentum 0.9, and weights decay 0.0001.

Three key observations are made:

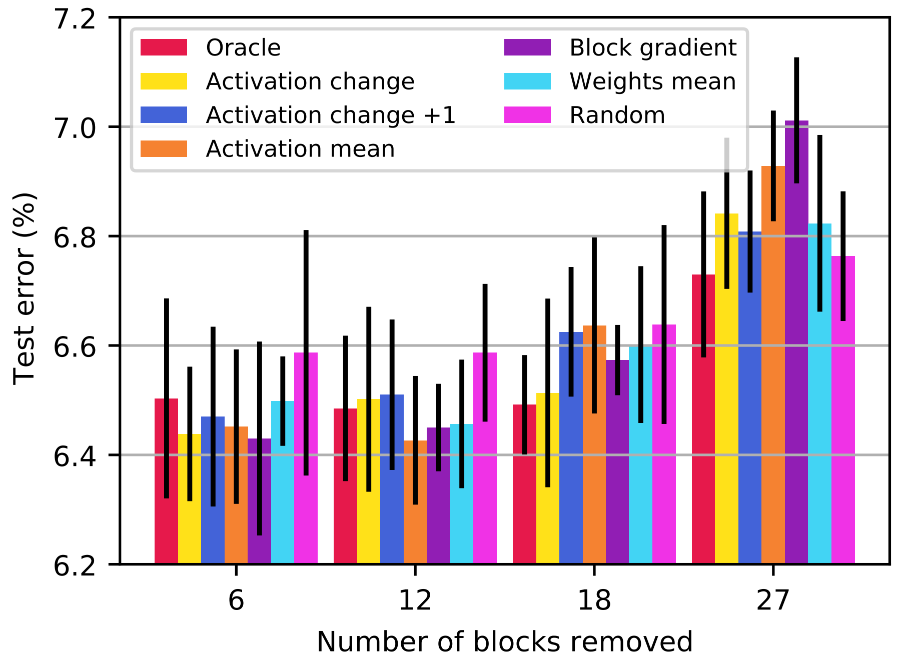
Inconsistency. The performance of each block selection method is not consistent throught the amount of pruning performed. For instance, the **block gradient** selection method is worst at 27 blocks removed but similar with the other methods otherwise; **activation change plus one** is worse than **activation change** at 18 blocks removed, but they are similar at every other pruning amount.

Close together. After fine-tuning, all pruning methods are very close together in mean and their standard errors overlap. There is no significant difference in performance between them. Interestingly, random pruning is similar to all other methods.

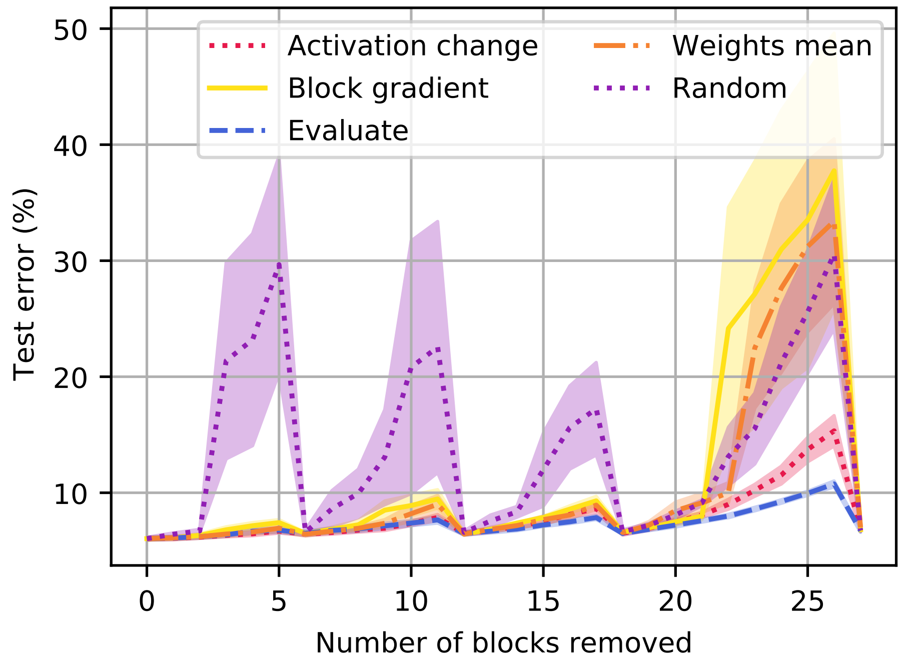
Random is good. The **random** selection method is similar to all other methods in performance after fine-tuning, even though it uses no heuristic at all to pick blocks. Since we have only ran one random run for each baseline, no strong conclusion can be drawn. Nonetheless this is an important finding to highlight.

Based on the above observations it is key to focus on the fine-tuning part of the pruning process as opposed to the block selection method. One explanation is the fact that training can recover the lost error very well in the context of block pruning. Another contributing factor is the small pool of choices of prunable units (54 for ResNet-110) which causes all block selection methods, including random, to likely have a significant overlap with a good subnetwork. This is, however, not likely to hold true for smaller prunable units such as convolutional filters, neurons, or weights. A smaller prunable units means a much larger pool of options.

Not all pruning methods do well, even after fine-tuning, proving that there exist subnetworks that are unsuitable. To show this, we performed the same experiment as above using the **block gradient** method, but instead of using the minimum gradient to prune, we used the maximum. Results in Figure 3.12.



(a) CIFAR-10 test error after fine-tuning.



(b) All pruning steps,with and windout fine-tuning. Lines are the mean and highlighted background is the standard error from using 6 runs with different initial models.

Figure 3.11: All block selection methods pruning 50% of blocks from 6 ResNet-110 initial models with fine-tuning for 50 epochs after pruning 6, 12, 18, and 27 blocks.

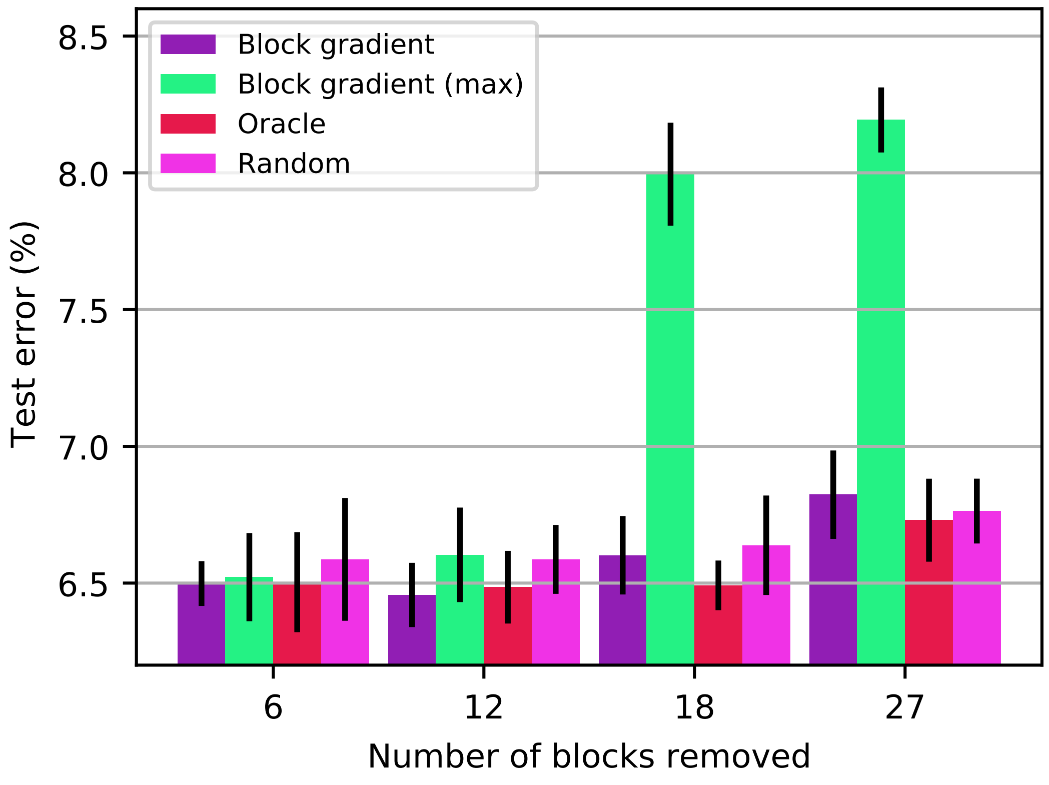


Figure 3.12: Highlighting that block gradient (max) is a selection method that does not perform well after fine-tuning in an experiment identical to that of Figure 3.11: four fine-tuning loops at the steps shown, six pruning runs with different inital models for each block selection method shown.

3.3.5 Blocks removed and parameters removed

Not every block has the same number of weights or MACs (multiply–accumulate operations). The wall clock time inference reduction from removing a block is also different for different blocks. The number of parameters and MACs are the same for blocks that belong to the same group, excluding the first block from each group, but are different for blocks that belong to different groups. We now look at the block selection methods in relation to not only how many blocks are removed and the impact on accuracy, but also in relation to the percentage of parameters that are removed. See Figure 3.13 for a visual representation. We observe that **oracle** pruning tends to preserve the most parameters as the pruning progresses. **Activation change** and **weights mean** selection methods do prune significantly more of the weights for the same number of blocks when compared to oracle pruning. **Block gradient** is similar to or more conservative than oracle in the number of parameters kept per number of blocks removed, however, the accuracy without fine-tuning is much lower.

In a simple preliminary experiment (only one initial model was used) with fine-tuning (four loops of fine-tuning, uniformly placed throughout the pruning process), **oracle**

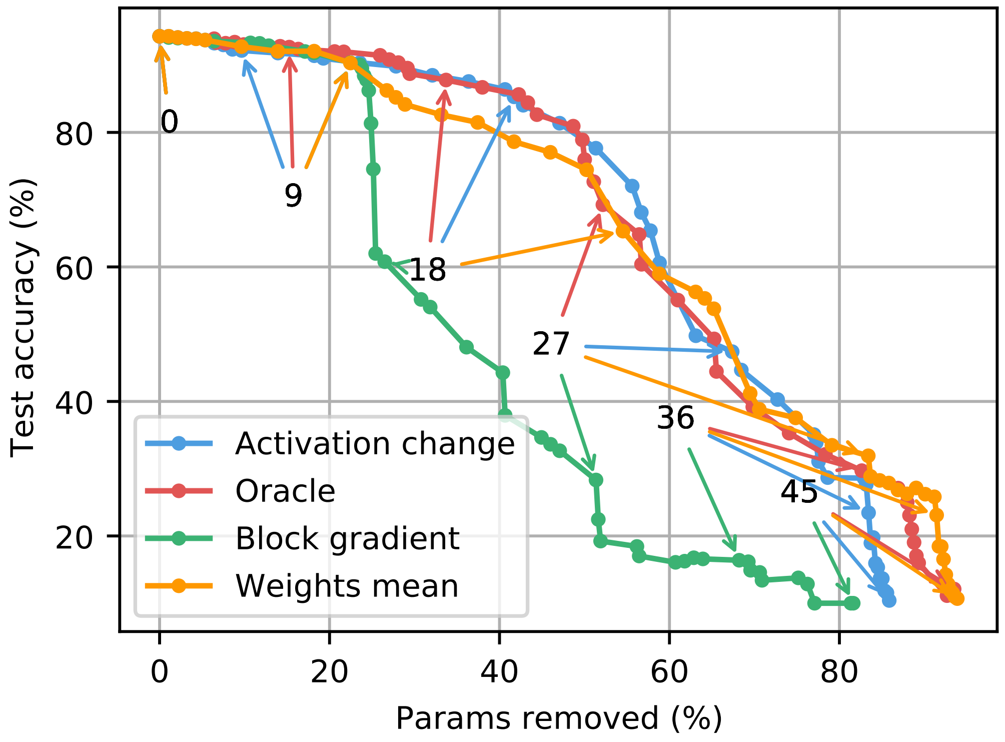
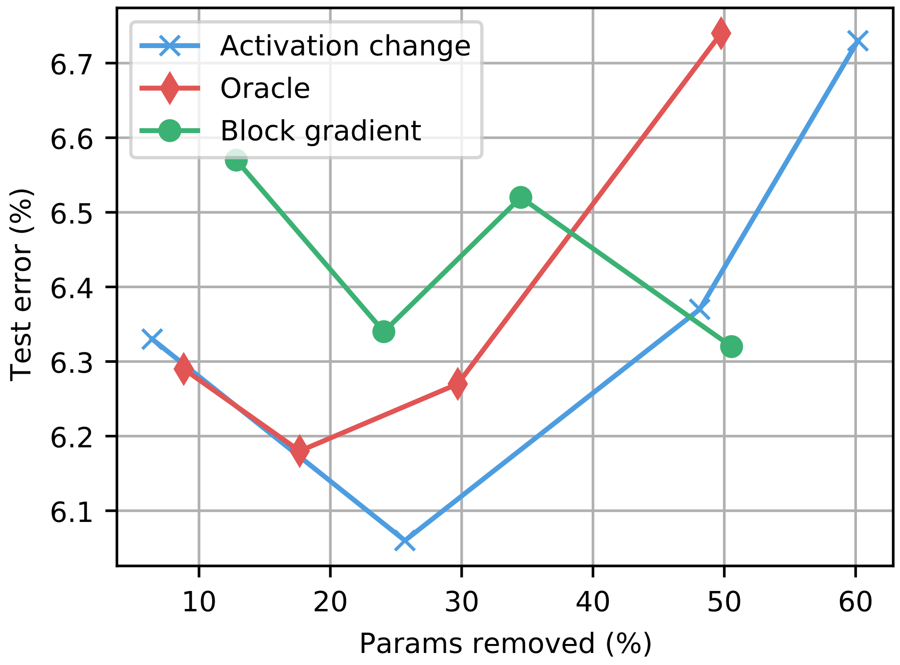


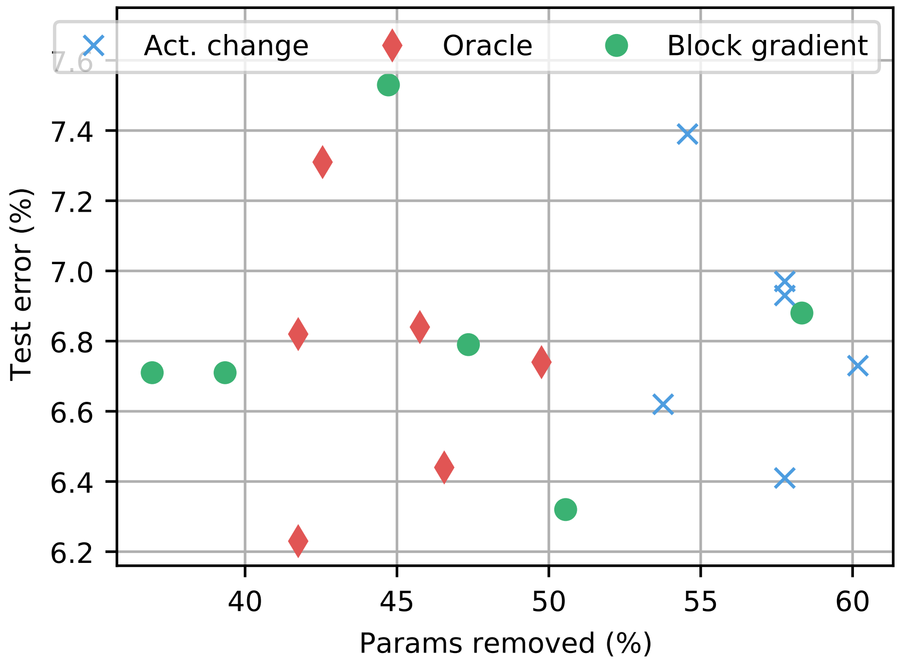
Figure 3.13: CIFAR-10 test accuracy versus percentage of parameters removed when pruning a single initial ResNet-110 with different block selection methods and no fine-tuning. Annotated text with arrows denote the number of blocks removed. Observe that different block selection methods keep different amounts of parameters based on their choice of blocks to prune. The final number of parameters can be a factor that impacts the final accuracy of pruned models and not only the number of blocks removed. Notably, the oracle pruning method is rather conservative in the number of weights pruned when compared to activation change and weights mean.

and **block gradient** selection methods removed a similar amount of parameters, but **Activation change** is similar in accuracy with **oracle** but it removes more parameters (Figure 3.14a). Looking at the same data, but from 6 pruning runs for each block selection method, each starting with a different initial model, we observe that indeed **activation change** prunes more parameters for the same number of blocks and comparable accuracy. **Oracle** and **gradient block** are similar in both accuracy and number of parameters removed but the latter has a much lower time complexity (Figure 3.14b).

The choice of using the number of pruned blocks (our prunable unit) as the main comparison axis makes it easy to compare performance of ResNet networks in relation to the number of pruning iterations that have been performed. Each pruning run, despite having identical configurations, performs differently based on the starting weights. Our experiments were routinely performed 6 times, each with a different starting model (trained identically but with different random initialization; they all have similar before-pruning performance, but different weights). Using the number of blocks removed enables us to



(a) Percentage of parameters removed versus CIFAR-10 test accuracy for a single pruning run for each block selection method. For each line, the connected points are plotted for each fine-tuning loop at 6, 12, 18, and 27 blocks removed from left to right.



(b) Scatter plot of CIFAR-10 test error versus percentage parameters removed for 3 block selection methods. Each block selection method has 6 points, each corresponding to a different initial model. The plot only shows data from 27 blocks removed (50% of blocks of the ResNet-110 inital models).

Figure 3.14: Showing the percentage of parameters removed from ResNet-110 models by different block selection methods. Pruning was performed for 27 blocks with four fine-tuning loops, of 50 epochs each, at 6, 12, 18, and 27 blocks removed, respectively.

have a clear base point for which to create aggregate results (mean and standard error) to compare them with different pruning methods.

It is true that blocks are different in the number of weights, MACs, and impact on inference runtime. This combined pruning with different block selection methods and iterative fine-tuning makes the comparison by number of blocks removed unbalanced: comparing networks of essentially different sizes. This is a known limitation of our approach and analysis. The data to study our findings based on different comparison points is available in our results dataset (Velici, 2021). The choice was made to use number of blocks removed to be able to compare different pruning settings against a set of starting weights. We believe that using different starting weights to report results is essential in pruning, as it is known (see Section 2.9) that pruning performace is affected by the input model. Averaging the same pruning confiruation over mutiple runs with different starting weights does mitigate the unbalance but it does not eliminate it.

Different works in the pruning literature (Lee et al., 2018; Molchanov et al., 2016) avoid using the number of prunable units removed and approach the comparison by setting fixed targets, in number of weights, inference speed, or an approximative maximum accuracy loss. Then they perform the comparison of results at those targets, therefor comparing networks of similar sizes or similar inference speed. Using the inference speed and number of remaining weights could be used in the future for block by block pruning, where the number of blocks pruned is simply part of the results and not used as a predefined target. Molchanov et al. (2016) does not report results for multiple starting weights.

3.4 Block selection and the initial random initialization

The first question asked is how different is the order in which blocks are selected for different fully trained baselines. Six ResNet-110 inital models fully trained on CIFAR-10 were pruned with the oracle block selection method and no fine-tuning. Kendall-tau similarity of the order of the removed blocks (without remaining saliency scores) between each pair reveals that the orders of block removal are not correlated — see Figure 3.15.

Since all the six baselines were trained and pruned identically and the only difference is the initial random initialisation (and subsequent use of randomness during training), the observation that the pruning order is different for each baseline hints at the fact that a key deciding factor for the saliency of blocks is indeed the random initialization of the original model. This begs the question: how early in the training process can the saliency of blocks be determined?

A single model was trained on CIFAR-10 for 500 epochs saving a checkpoint every 10 epochs, including the random initialization. Each of the checkpoints was then pruned

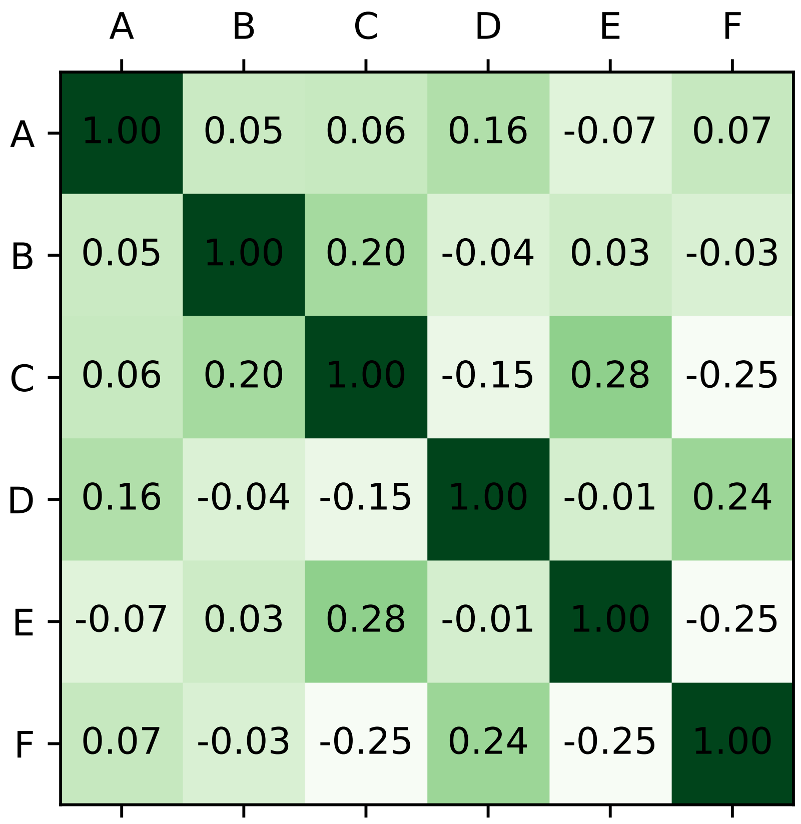
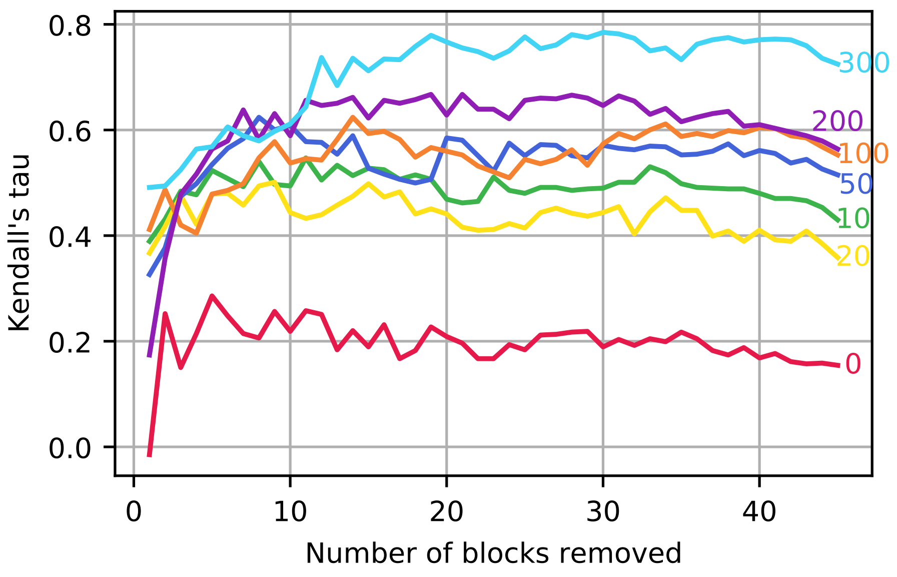


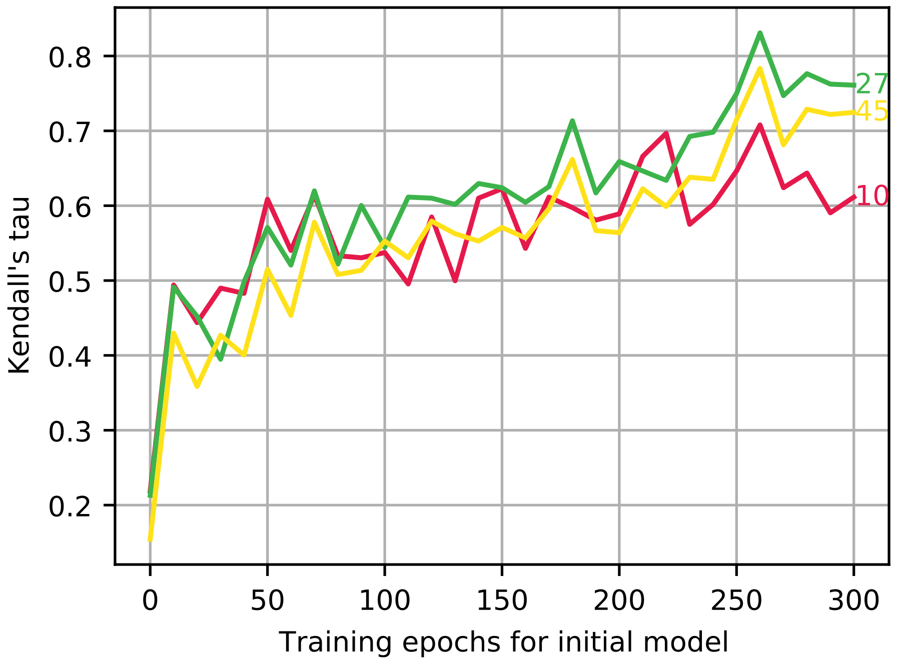
Figure 3.15: Kendall-tau similarity between the order in which blocks have been pruned from 6 different initial models trained on CIFAR-10. Similarity close to zero denotes no correlation between the order in which the blocks were pruned. Overall, this figure suggests that initial randomness is a key deciding factor in the saliency of blocks. Initial models are ResNet-110 and 50% of the blocks were pruned (27 out of 54). The oracle selection method was used with no fine-tuning.

using the oracle method with no fine-tuning to observe the similarity of block choices. The kendall-tau metric is used to compare the block choices between each checkpoint and the final one. Results, in Figures 3.16a and 3.16b, show that the similarity between block choices increases as the training progresses, as expected. There is, however, a sudden increase in similarity early on in training process, which suggests that the most important blocks can be discovered early in the training process, at around 50 epochs or even sooner.

The idea that important parts of a neural network are given by its initial random initialization is not new. The works of Lee et al. (2020); van Amersfoort et al. (2020); Lee et al. (2018); Rosenfeld and Tsotsos (2019); Wang et al. (2020) and others show that pruning at or close to the random initialization is possible and yields promising results. Frankle and Carbin (2019) prune weights of fully trained models and then train the resulting connections starting from the original random initialization obtaining impressive results — matching performace of initial unpruned model at 95+% of weights pruned,



(a) Kendall-tau similarty across all numbers of block removed for a selected number of inital model training epochs (number of epochs for each run is displayed in the figure).



(b) Kendall-tau similarity across initial model training epochs for a selected set of number of blocks removed (number of blocks removed shown next to plotted lines).

Figure 3.16: Kendall-tau similarity of the block pruning order (including saliency scores for non-removed blocks) between pruning the final model (at epoch 354) and pruning a range of checkpoints throughout the training process. Epoch 0 is the random initialization. Observe there is a steeper increase in correlation for the first tens of epochs and it gets shallower as the training progresses, suggesting most important blocks can be identified early in the training process.

and calling the well-performing subnetworks *winning lottery tickets*. Simply training the obtained connections from a new random initialization does not work well, however Zhou et al. (2019) show that only using the sign of the initial random initialization also perfoms better than fully random weights, even if using a constant for the weights. An outstanding result of Zhou et al. (2019) is obtaining 65.4% accuracy on CIFAR-10 by only training a mask and not the original random initialization (a mask is a bernoulli distribution, the result was obtained as average of evaluating 10 independent samples).

3.5 Fine-tuning frequency and amount

Fine-tuning the network after pruning is simply re-training the network for a number of epochs, typically without reinitialising the weights. It can be done after removing each block, when the accuracy drops below a threshold, or by using a fixed schedule. We use the terms training loop and fine-tuning loop interchangeably.

The training threshold is a threshold on the loss of validation accuracy between the current pruning iteration and the last time the network was trained. When the accuracy loss goes below this threshold, a fine-tuning loop is started.

The fine-tuning schedule is forcing a training loop based on the number of blocks pruned at the current iteration. The training schedule that is always present is to trigger a fine-tuning loop when the target size has been reached. Throughout this work we also experiment and evaluate training schedules that add more training loops uniformly spread across the pruning process, or simply training at every pruning step.

After each fine-tuning loop we resume the pruning process using the weights obtained at the end of epoch with highest validation accuracy. This is on par with how an early stopping criterion might be used. Early experiments did not have this reset in place and the pruning was continued by keeping the weights obtained by the last epoch of fine-tuning, however the results were underperforming those with the reset. Unless otherwise stated, the reset from the best performing epoch is present.

3.5.1 The cost of fine-tuning

Fine-tuning is an expensive operation and the goal is to minimize the amount of training required. It is therefore important to keep track of the cost of fine-tuning. Since maximizing accuracy is the primary goal, and increasing the amount of fine-tuning is likely a method of doing so, we focus on finding a reasonable trade-off between the cost of fine-tuning and final pruned model accuracy. A key comparison point in terms of training cost is the cost of training a model of similar size from scratch.

Given a dataset and a model, the training cost can be defined in terms of the total number of epochs of training executed, and this is often a sufficient abstraction for comparing between hyper-parameter choices or similar as long as the model architecture, optimizer algorithm and size of dataset remain the same. For instance, the cost of training a model with one learning rate schedule or another can be directly compared based on how many epochs were required to reach convergence or a target validation accuracy.

For pruning and fine-tuning, the same comparison does not hold as the size of the model is effectively changed (blocks are removed). Therefore an epoch of training with the inital model is more expensive than an epoch of training after one or more pruning iterations[7]. We define the epoch-parameter to be the cost of fine-tuning as the number of epochs trained at step i multiplied by the number of parameters in the model at this stage.

3.5.2 Single shot fine-tuning

In this setting there is a single fine-tuning run at the end of the pruning process. To evaluate sine-shot fine-tuning, a ResNet-110 model fully trained on CIFAR-10, which has 54 initial blocks, was pruned block by block 53 times, with target number of blocks 53, 52, ..., and 1. For comparison, randomly initialized ResNet models with the same block structure were fully trained as well for each of the pruning configurations, as well as ResNet models with *standard* block structure choices (equal number of blocks in each group). Pruning was performed with the oracle block selection method.

Pruning followed by a single fine-tuning loop is beneficial for removing a small number of blocks: 10-12 blocks out of 54 for ResNet-110. Pruning more blocks does not show a consistent decrease in error when compared to simply training standard ResNet block configurations from scratch. The block configurations discovered through pruning do not train well from scratch, being outperformed by both the pruned models and the standard configurations for most choices of number of blocks in our experiment. This is displayed in Figure 3.17.

All training for this section, both from scratch and fine-tuning, shares the same parameters: 400 epochs, SGD optimiser with Nesterov momentum 0.9, weight decay 10^{-4}, learning rate starting at 0.1, then 0.01 from epoch 250, and 0.001 from epoch 375, batch size 128, with the usual data augmentation for CIFAR-10: 4 pixel padding followed by random 32x32 pixel crop and random horizontal flip.

Training for 400 epochs, the same as training from scratch, is expensive for fine-tuning. The goal is to perform as little training as possible and recover the lost accuracy with a the smaller network. A subsequent experiment with a reduced number of fine-tuning epochs to 50 has been performed using the same parameters as before except the learning

[7]Please note that this is not always the case for unstructured pruning or when units are only masked and not removed. In our implementation the pruned blocks are actually removed unless otherwise noted

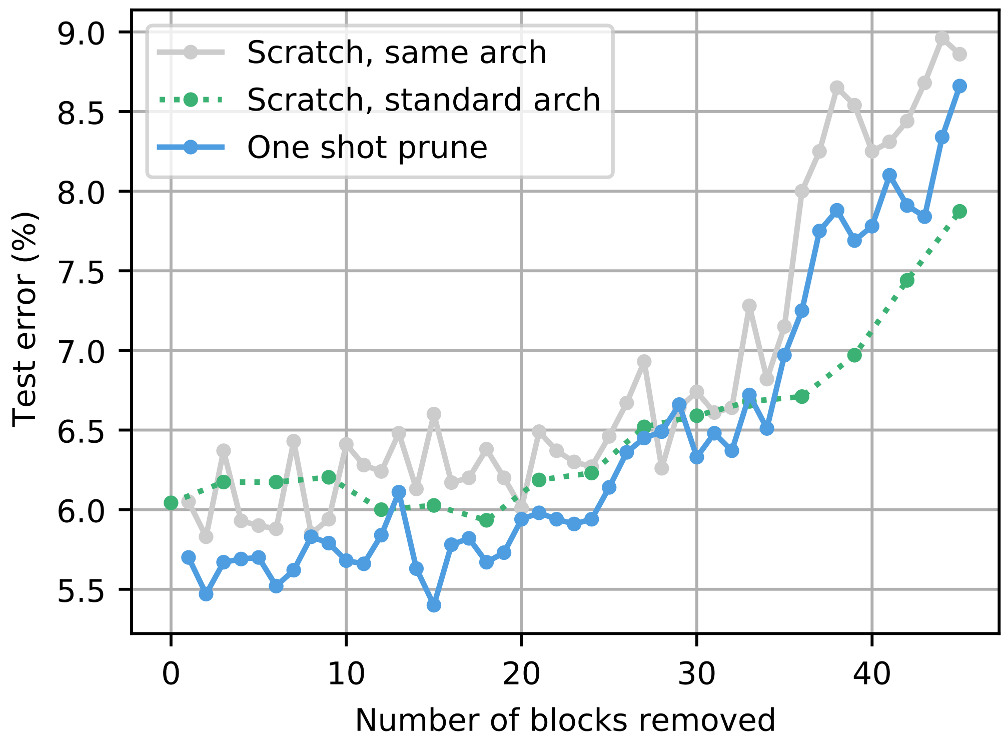


Figure 3.17: One shot fine-tuning from a ResNet-110 (54 blocks) model using CIFAR-10. For pruning, each data point is an independent pruning and fine-tuning run. For scratch, the models are trained from scratch, either reusing the block configuration from the corresponding pruning run (same), or using an equal block per group setting (standard). Both fine-tuning and training is carried out for 400 epochs.

rate starts at 0.1 and is divided by 10 at epoch numbers 10 and 20. The results are in Figure 3.19, where it can be noticed that training similar-sized architectures from scratch constantly outperforms pruning except for up to 6 blocks removed where the accuracies are similar. Although 50 epochs of fine-tuning is an arguably cheap operation, training a large network followed by fine-tunig is more computationally expensive than simply training a small model from scratch.

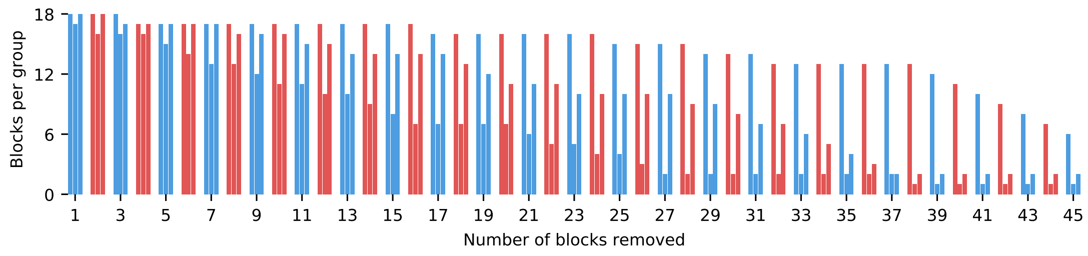


Figure 3.18: Shows the number of blocks per group as the pruning progresses for the pruning runs in Figure 3.17. The ResNet architecture used has 3 groups. Each grouping of 3 bars represents the ResNet configuration for one pruning run, for a fixed number of blocks to be removed. Each bar represents the number of blocks in a ResNet block group. Alternate colouring is for visualisation only. Standard configurations such as ResNet-110 (the initial model for this figure), ResNet-56, ResNet-20 have an equal number of blocks per group, however our pruning methods create unbalanced groups and tend to prune the middle group first.

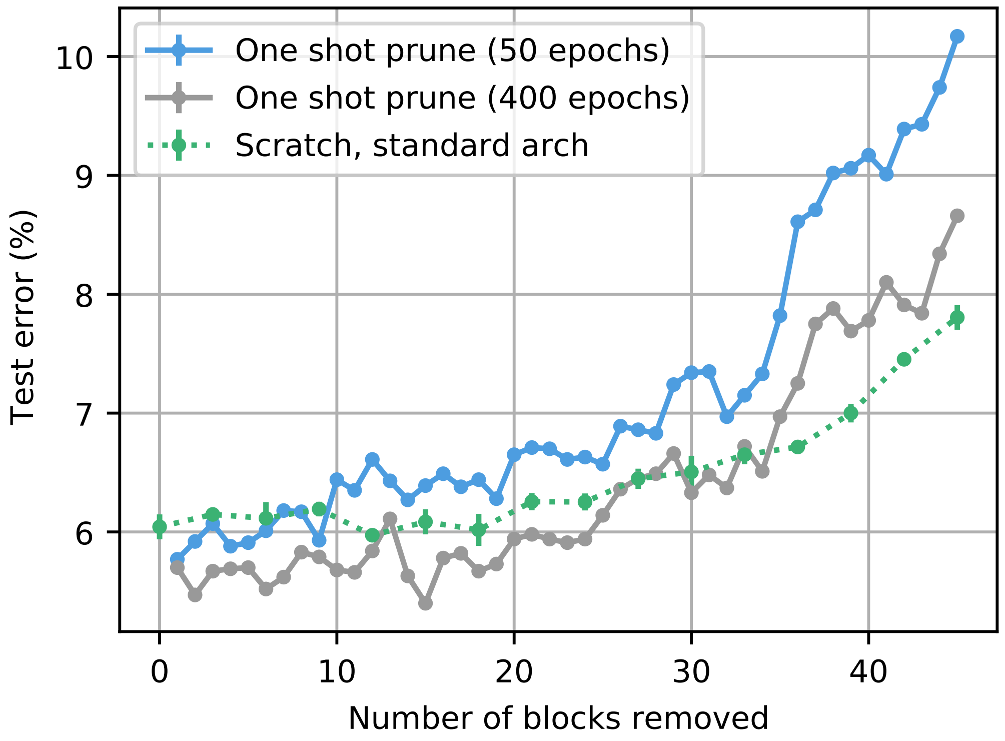


Figure 3.19: One shot fine-tuning from a ResNet-110 (54 blocks) model using CIFAR-10 with 50 epochs of training (with starting learning rate 0.1 and divided by 10 at epochs 10 and 20) for the fine-tuning loop of each run. For pruning, each data point is an independent pruning and fine-tuning run. Scratch training for 400 epochs with equal number of blocks per ResNet group, each point is the mean of 4 runs (6 for ResNet-110) with vertical lines denoting standard error. All pruning runs used the same initial model. The same pruning experiment from Figure 3.17 with 400 epochs of fine-tuning is included for comparison.

3.5.3 More fine-tuning loops

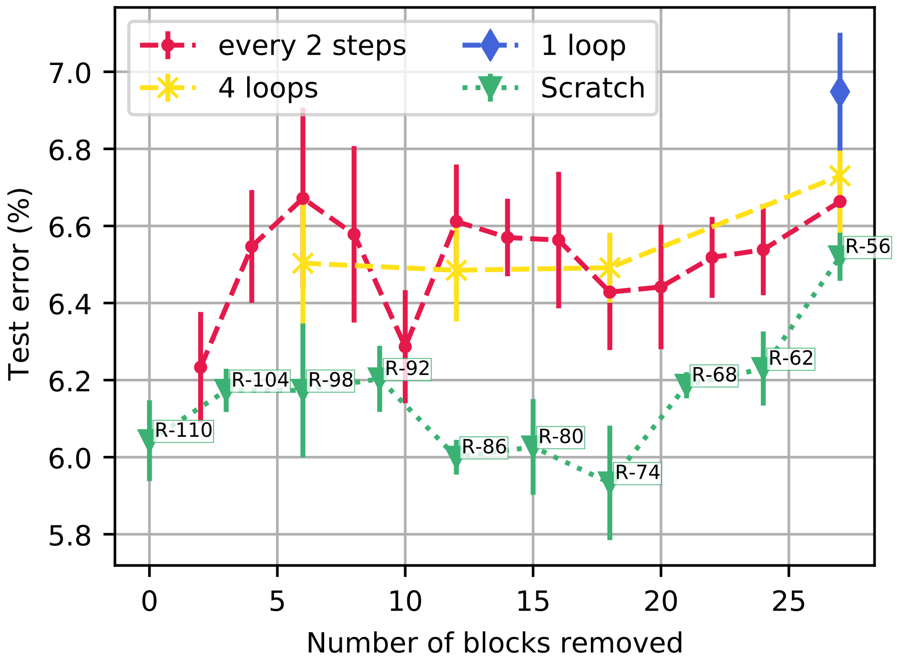
The next question we ask is whether adding more fine-tuning loops will improve the accuracy of pruned models for a lower overall cost of training, as compared to fine-tuning for 400 epochs. Since fine-tuning will be performed at different stages of pruning, the metric of epoch-parameter is used to approximate the cost.

Pruning only a few blocks seems to work well for one shot pruning and the performance decreases as the pruning progresses. More blocks must be removed for pruning to yield impactful model size reductions and inference time speed-ups. We pick pruning 50% of the blocks of a ResNet as the main comparison point - pruning from a ResNet-110 with 54 blocks to ResNet-56 with 27 blocks.

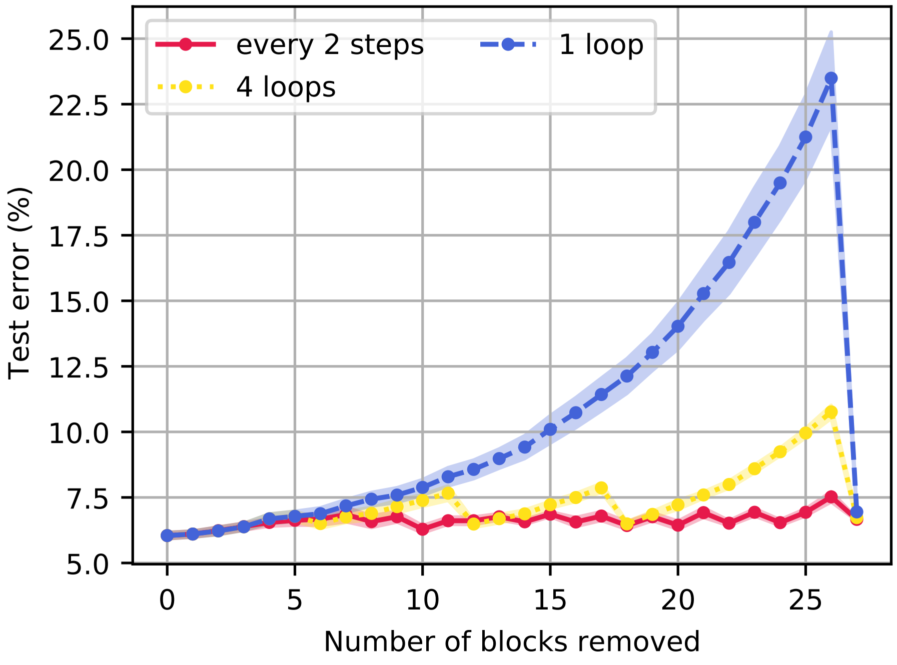
A pruning experiment is setup such that the resulting network size is of 27 blocks and with the following fine-tuning configurations: fine-tuning at every other pruning step, fine-tuning four times in total, and fine-tuning a single time. All for a fixed amount of 50 epochs per fine-tuning step. All other parameters including the learning rate schedule for pruning are the same as for the experiment from Figure 3.19. We find that adding multiple fine-tuning loops does improve the performance of the pruned model, but only by a small margin (Figure 3.20a). Frequent fine-tuning loops keep the error lower even for steps with no fine-tuning (Figure 3.20b). However it appears to be easier to recover the lost accuracy in a single fine-tuning loop when trained for more epochs. Training for more epochs for a single fine-tuning loop gives better performance overall than adding more fine-tuning loops for less epochs, even in the case where the total number of epochs is larger for the latter.

In a different view, Figure 3.21 illustrates the final point of each pruning run from pruning a ResNet-110 using one, two, or four total fine-tuning loops of 50 epochs. The fine-tuning loops are uniformly spread throughout all pruning steps with the constraint that the final pruning step has a fine-tuning loop. Observe that activation change and oracle selection methods perform similarly well for most target network sizes, although training from scratch is still the cheapest and most performant overall. The summary of the results is in Table 3.3. A large table including all numbers of blocks removed and thresholding accuracy loss is in the appendix, Table A.1.

Regardless of the block selection method, adding more fine-tuning loops has a clear benefit for improving the performance of the trained model. After removing 35 blocks, four equally spaced fine-tuning loops of 50 epochs (total 200 epochs) perform at least as well as 400 epochs of fine-tuning at the end of pruning only.

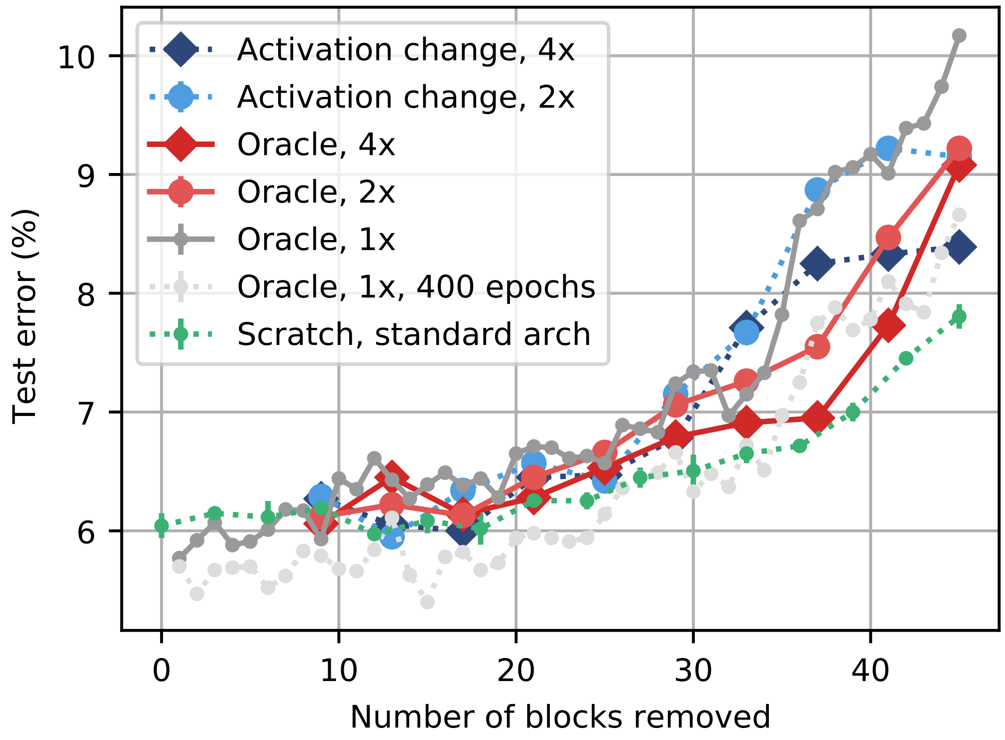


(a) Pruning steps with no fine-tuning are omitted.



(b) Same runs, but includes steps where no fine-tuning was performed. Scratch runs were omitted for brevity.

Figure 3.20: Pruning runs with different fine-tuning schedules, pruning a ResNet-110 from 54 to 27 blocks. Showing CIFAR-10 test error. Each series shows the mean and standard error from 6 runs, each with a different initial model. Note that in this plot fine-tuning cost is cumulative: it costs more for more blocks removed as it includes previous fine-tuning loops from the same pruning run.



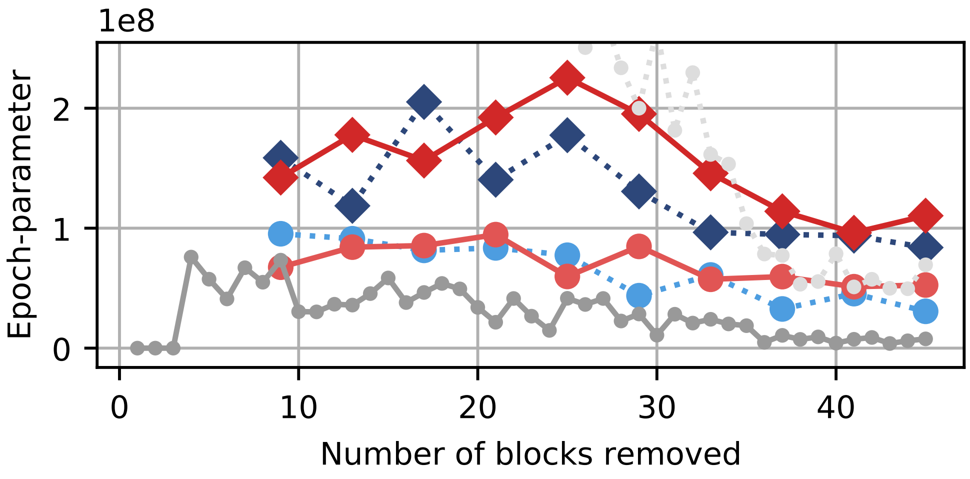


Figure 3.21: **Top**: Comparing one, two, and four fine-tuning loops per pruning run with oracle and activation change block selection methods. Each data point is a full pruning run. **Bottom**: Showing the total cost of fine-tuning is lower for pruning runs with smaller final network size (right hand side). All fine-tuning loops performed for 50 epochs (except 400 epochs run, which is the same as Figure 3.17) with a learning rate of 0.1, 0.01, and 0.001 from the start, epoch 10 and epoch 20, respectively.

Table 3.3: Results table from pruning a ResNet-110 with different fine-tuning schedules and block selection methods. "1x", "2x", "4x" denote the number of uniformly spread fine-tuning loops. t denotes that an accuracy loss threshold was used. Thresholding and "every 2" runs are continuous and others are individual prunings experiments (ie. every row with values under "4x" has four fine-tuning loops). All fine-tuning loops use up to 50 epochs. Cost is in epoch-parameters (1e6) shown based on epochs used when picking best validation error. A * indicates the value is the mean from 6 runs with different initial models. R% is the percentage of parameters removed, #BR is the number of blocks removed. Err is the test error % on CIFAR-10.

	Act. change, 2x			Act. change, 4x			Oracle, 1x		
#BR	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%
9	6.29	95.3	16.06	6.27	158.7	12.85	5.93	73.3	15.25
13	5.95	91.2	26.75	6.05	118.6	29.95	6.43	35.9	25.95
17	6.34	81.5	40.65	6.00	205.3	40.65	6.38	46.4	29.44
21	6.57	83.6	54.54	6.45	140.4	48.14	6.71	21.6	43.33
25	6.42	77.5	58.83	6.47	177.5	62.04	6.57	41.5	50.02
27	-	-	-	6.84*	177.8	57.77	6.86	41.4	52.17
29	7.15	43.7	69.53	6.78	130.6	67.92	7.24	28.5	56.71
33	7.67	61.0	78.62	7.71	96.5	78.62	7.15	24.0	69.81
37	8.87	32.7	87.71	8.25	94.8	83.70	8.71	10.7	86.90
41	9.22	45.4	88.79	8.33	93.8	84.78	9.01	7.4	88.79
45	9.15	30.7	89.87	8.39	83.9	85.86	10.17	7.7	89.87
	Oracle, 1x, e=400			Oracle, 2x			Oracle, 4x		
#BR	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%
9	5.79	413.6	15.25	6.12	67.5	15.25	6.06	142.2	11.25
13	6.11	337.1	25.95	6.22	84.2	25.95	6.45	177.8	21.94
17	5.82	436.0	29.44	6.13	85.5	32.64	6.14	156.4	32.64
21	5.98	301.1	43.33	6.45	94.6	40.13	6.28	192.3	35.32
25	6.14	346.0	50.02	6.66	59.8	49.22	6.53	225.4	49.22
27	6.45	279.0	52.17	-	-	-	6.73*	158.9	46.56
29	6.66	200.0	56.71	7.06	84.8	55.91	6.79	195.2	59.11
33	6.72	161.5	69.81	7.26	57.4	69.00	6.91	145.8	65.00
37	7.75	77.3	86.90	7.55	59.6	82.10	6.95	114.0	74.09
41	8.10	51.0	88.79	8.47	51.2	87.98	7.73	96.1	79.97
45	8.66	69.5	89.87	9.22	52.6	89.06	9.08	110.4	89.87

3.5.4 Fine-tuning at every step

The other extreme of fine-tuning once is to fine-tune at every pruning step. In this context pruning a ResNet-110 to 50% of its blocks requires 27 fine-tuning loops. Pruning down to the size of a ResNet-20 requires 45 fine-tuning loops. For 50 epochs per fine-tuning step, this costs 1350 or 2250 epochs, respectively. Training for an epoch decreases in cost as the pruning progresses, but regardless of the blocks chosen, the cost of fine-tuning at every step is high.

For the purpose of understanding whether this type of fine-tuning helps improve the performance and to what extent, a set of experiments was performed. We have pruned a ResNet-110 initial model with fine-tuning loops at every step using a fixed learning rate of 0.001 and 50 or 20 epochs per step. The results are in Figure 3.22, and we compare than with a similar set of experiments but with 50, 20, 10, or 5 epochs of fine-tuning at every other pruning step. Fine-tuning cost in epoch-parameters and test accuracies are shown in Table 3.4.

Full fine-tuning for 50 epochs at every step of the pruning is too expensive to be considered a useful approach to pruning in practice. Not only it is more expensive than training small models from scratch, but it is also more computationally expensive than

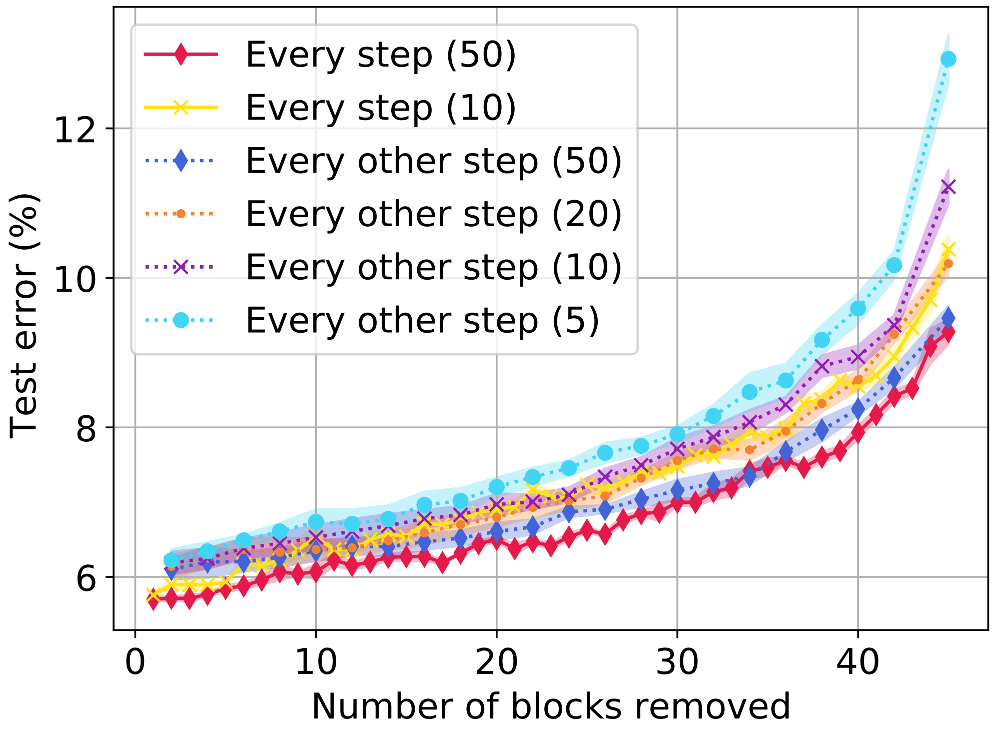


Figure 3.22: Pruning a ResNet-110 with 50, 20, 10, or 5 fine-tuning epochs at every step or every other step. A fixed learning rate of 0.001 was used for all runs.

Table 3.4: Fine-tuning at every pruning step and every other pruning step with different epochs per loop. Fixed learning rate of 0.001 for all runs, and oracle block selection method.

Blocks removed:	10		28		34		45	
Experiment	Error	Cost	Error	Cost	Error	Cost	Error	Cost
Every step (50)	6.07	266.6	6.85	881.4	7.42	1007.5	9.28	1202.2
Every step (10)	6.53	39.6	7.39	139.7	7.94	168.2	10.38	210.2
Every other step (50)	6.35	206.8	7.04	609.2	7.35	709.3	9.46	802
Every other step (20)	6.37	75	7.32	218.4	7.70	254.6	10.19	284.2
Every other step (10)	6.53	25.2	7.49	88.5	8.07	103.2	11.22	124.1
Every other step (5)	6.74	10.4	7.75	40.5	8.47	48.7	12.93	57.8

training *large* models from scratch. On a NVIDIA GeForce 10180 Ti it takes approx. 6 hours to train a ResNet-110 for 500 epochs, and 14 hours to prune (of which approx. 13 hours are fine-tuning only). While only using 10 epochs is cheaper, we find it is still more expensive and less accurate than using a fine-tuning loop schedule (like four uniformly spread loops or using an accuracy loss threshold).

As pruning in many cases is a post-processing step and this approach does not improve results on par with the increase in cost, this approach is only used as a reference point in comparison with other fine-tuning settings. We conclude that fine-tuning at every pruning step is not a useful practice. On the other hand, the idea of fine-tuning at every step for a small number of epochs ($<$3) followed by a larger loop at the end is promising and discussed in Section 4.1.

3.5.5 Thresholding accuracy loss to fire a fine-tuning loop

Fine-tuning for every pruning step is expensive and manually selecting a fine-tuning schedule may not be placing the fine-tuning loops where they are most needed. To address these issues, we introduce a threshold on the accuracy loss from the previous fine-tuning loop or from the initial model at first. This threshold is used to start a fine-tuning loop when the accuracy after pruning dropped by more than the tolerated amount.

The cost of fine-tuning is drastically reduced by this method as the most expensive fine-tuning loops are at the beginning of the pruning process, where the accuracy drop is the lowest. Using the thresholds introduces more fine-tuning loops towards the end of the pruning (where fine-tuning is cheaper) and fewer loops at the beginning, which is illustrated in Figure 3.23. The cost of fine-tuning using total number of used epochs (after resuming from best) for the same pruning runs is available is Table 3.5, along with the total epoch-parameter cost for fine-tuning when removing 27 and 45 blocks. All runs with an accuracy loss threshold have a forced fine-tuning loop at 27 blocks removed, to create a common comparison point at that size.

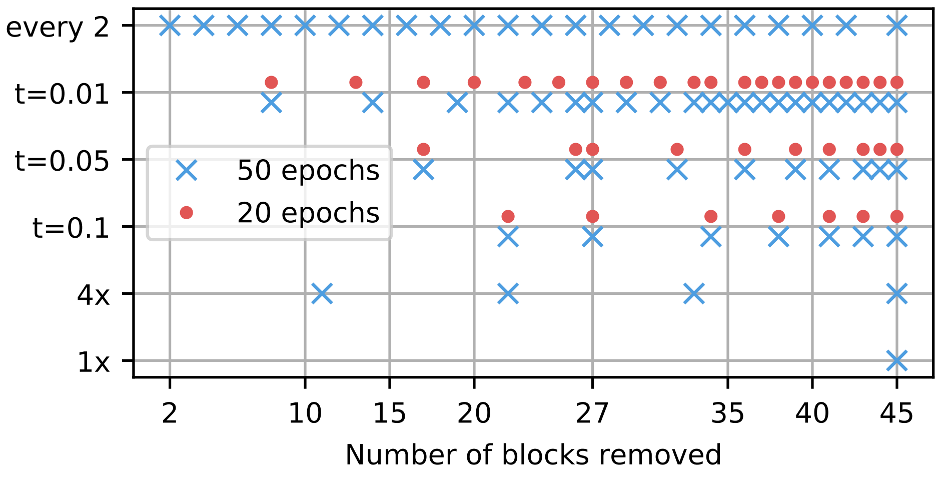


Figure 3.23: Showing where fine-tuning is triggered throught the pruning process when using different accuracy loss thresholds (t). Other common fine-tuning schedules are included: every other step (every 2), four times (4x) and once (1x).

Using a threshold of 0.1 (1% accuracy loss), with 50 epochs of fine-tuning per step, to prune a ResNet-110 to ResNet-20 costs 134.5×10^6 epoch-parameters at results in an average test accuracy (over 3 runs with different initial models) of 8.17%, a similar accuracy from fine-tuning at every two steps, but for a fifth of the fine-tuning cost. A 0.1 threshold starts 10 fine-tuning loops and due to their placement throughout the pruning iterations they produce a more accurate pruned model, 8.17% vs 9.24% test error (respectively), for overall cheaper fine-tuning, 134.5×10^6 vs 143.3×10^6 (respectively), when compared to four uniformly spread fine-tuning loops. For about half the cost of fine-tuning four times at uniformly spread location, a threshold of 0.05 with 20 epochs per fine-tuning step yields an average error of 8.45%, a 0.79% improvement. See Table 3.6 and Figure 3.24 for the breakdown of the results.

Since thresholding puts most of the fine-tuning loops toward the end of pruning, the majority after the 50% (27 blocks removed) mark, our threshold results, with the exception of t=0.01, have less than four training loops for 27 blocks removed, resulting in slightly higher test errors than our 4 loops experiment. However, once again, at a much lower cost of less than half.

A large table containing results from thresholding, four loops, two loop and one loop of fine-tuning is included in the appendix – Table A.1.

Table 3.5: Cost of fine-tuning for different pruning runs. $\sum$ symbol denotes that the value shown is the sum for all fine-tuning steps up to and including the one in each respective row. The cost is the number of epochs multiplied by the number of parameters at each fine-tuning loop. Each group of rows separated by a horizontal line is a single pruning experiment but at different numbers of blocks removed. t=0.1, 0.05, or 0.01 denotes the validation accuracy loss threshold to start a fine-tuning loop for e=50 or 20 maximum epochs. Test error is on CIFAR-10 for a single run. All experiments start with the same initial model, a ResNet-110.

Experiment	Num blocks removed	Params (10^5) (% removed)	$\sum$ epochs	$\sum$ cost (10^6)	Test error %
t=0.1, e=50	27	8.56 (51%)	70	62.9	6.35
t=0.1, e=50	45	3.14 (82%)	242	134.5	7.75
t=0.05, e=50	27	8.70 (50%)	78	77.5	6.61
t=0.05, e=50	45	3.14 (82%)	281	165.8	7.77
t=0.01, e=50	27	8.70 (50%)	235	264.7	6.31
t=0.01, e=50	45	2.59 (85%)	717	523.5	7.79
t=0.1, e=20	27	7.86 (55%)	29	24.7	7.13
t=0.1, e=20	45	2.45 (86%)	101	50.4	8.76
t=0.05, e=20	27	8.56 (51%)	41	38.6	6.83
t=0.05, e=20	45	3.14 (82%)	140	79.9	8.12
t=0.01, e=20	27	9.25 (47%)	93	105.5	6.93
t=0.01, e=20	45	3.14 (82%)	291	214.7	8.16
every step, e=20	27	9.39 (46%)	269	344.1	6.55
every step, e=20	45	2.45 (86%)	526	484.8	8.6
4 steps, e=50	11	13.74 (21%)	41	56.3	6.36
4 steps, e=50	22	10.32 (40%)	91	107.9	6.47
4 steps, e=50	33	6.20 (64%)	134	134.6	6.84
4 steps, e=50	45	1.75 (90%)	184	143.3	9.38

Table 3.6: Average results of pruning 3 ResNet-110 models to the size of ResNet-20 (45 blocks removed) using an accuracy loss threshold t and 50 or 20 maximum epochs e per fine-tuning step. *4 loops* and *every 2* show average and standard error for 6 initial models. The cost of fine-tuning shown is the total up to the shown pruning step, in epoch-parameters, for a single baseline (same as Table 3.5). The *4 loops* and *every 2* rows are individual experiments with target sizes at 27 and 45 blocks removed. Thresholding rows are continued experiments with 45 blocks removed as target size and a forced fine-tuning loop at 27 blocks removed.

27 blocks removed			45 blocks removed		
Experiment	Error %	Cost $\times 10^6$	Experiment	Error %	Cost $\times 10^6$
t=0.01, e=20	7.04 ± 0.06	105.5	t=0.01, e=20	8.07 ± 0.1	214.7
t=0.01, e=50	6.48 ± 0.09	264.7	t=0.01, e=50	**7.97** ± 0.25	**523.5**
t=0.05, e=20	6.87 ± 0.05	38.6	t=0.05, e=20	8.45 ± 0.2	79.9
t=0.05, e=50	6.82 ± 0.3	77.5	t=0.05, e=50	**8.03** ± 0.13	165.8
t=0.1, e=20	7.02 ± 0.06	24.7	t=0.1, e=20	8.67 ± 0.05	**50.4**
t=0.1, e=50	6.69 ± 0.21	62.9	t=0.1, e=50	**8.17** ± 0.23	**134.5**
4 loops, e=50	6.73 ± 0.15	193.5	4 loops, e=50	9.24 ± 0.27	143.3
every 2, e=50	6.66 ± 0.06	525.6	every 2, e=50	8.2 ± 0.16	690.6

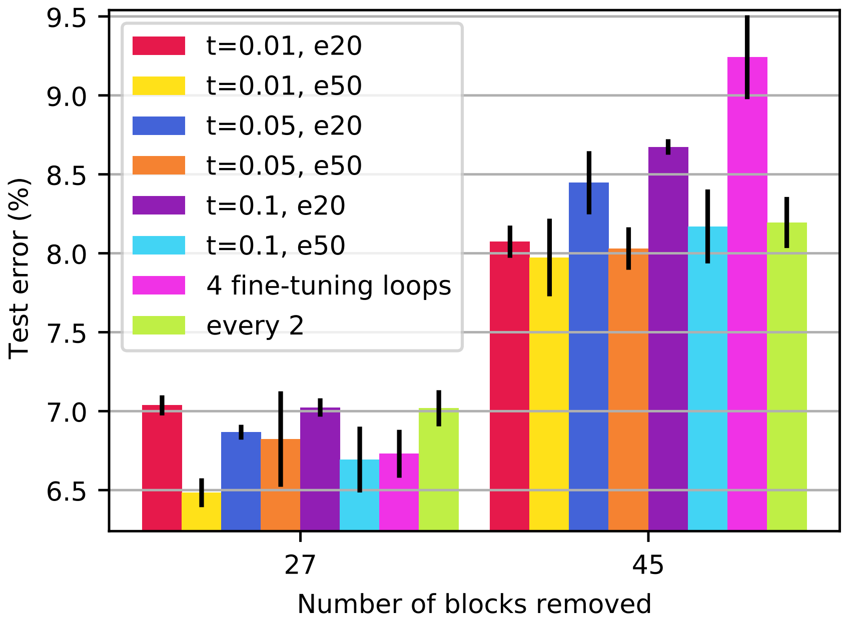


Figure 3.24: Pruning a ResNet-110 with accuracy loss thresholding (t) enabled. Same experiments as in Table 3.6.

3.6 Fine-tuning parameters

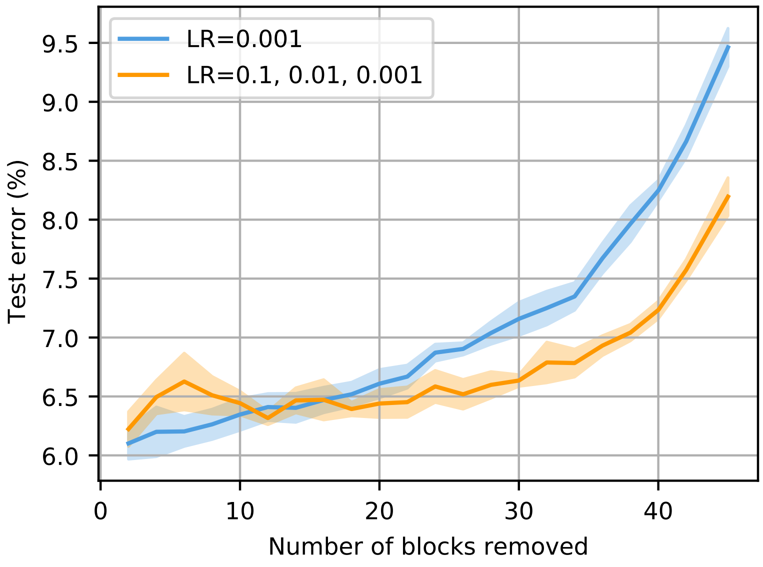
In this section we discuss parameters of the fine-tuning process that are applied at every fine-tuning loop. These settings are independent of the scheduling of fine-tuning loops throughout the pruning process and of the amount of epochs used.

3.6.1 Learning rate schedule

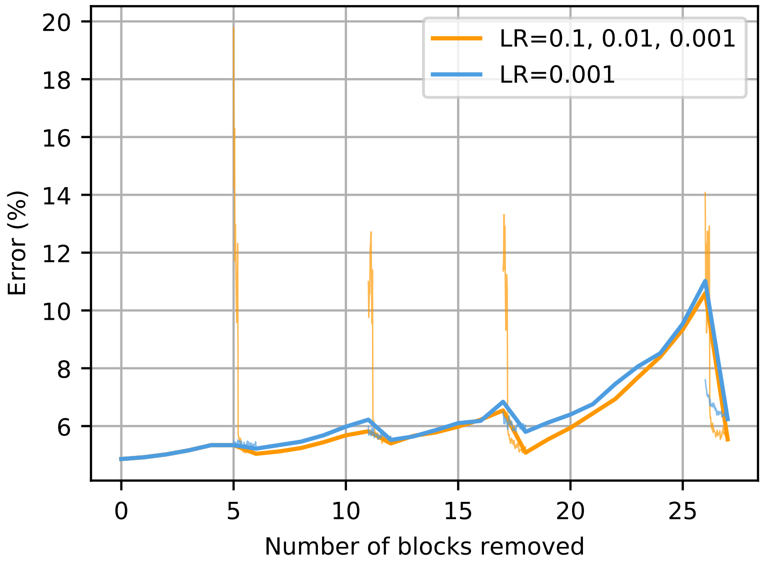
Our initial pruning experiments used a constant learning rate for fine-tuning after pruning. We then identified that adding a learning rate schedule starting with a large learning rate of 0.1 and dividing it by 10 at 10 epochs and 20 epochs (for a 50 epochs fine-tuning run) performed well.

A comparison of the two options when pruning a ResNet-110 from 54 to 9 blocks and fine-tuning for 50 epochs at every other step is shown in Figure 3.25a. We used 6 different initial models for the experiment. Our hypotheis is that pruning pushes the network in some local minima that a small learning rate cannot jump out of, and a high learning rate takes a large enough step allowing the training process to later find an overall better solution.

The impact of training with and without a learning rate schedule is illustrated in Figure 3.25b where it can be seen that the large learning rate at the beginning of the schedule makes the network perform worse and seemingly unstable, and then better later in the training process when the learning rate is reduced, supporting our local minima hypotheis.



(a) Pruning 6 ResNet-110 models with fine-tuning at every other pruning step. Showing mean and standard error for 6 runs, each with a different initial model.



(b) Pruning a ResNet-110 model from 54 to 27 blocks, with four total fine-tuning loops. Showing validation accuracy, thin lines are the accuracies throughout training from first to last epoch (50) at each fine-tuning loop.

Figure 3.25: Using a constant learning rate of 0.001 or a learning rate schedule starting with 0.1, from epoch 10 using 0.01 and finally 0.001 from epoch 20. Total 50 epochs per step for both configurations. Block selection method is oracle.

3.6.2 Noise the weights

Another approach for pushing the network out of a local minima is to add noise to the weights before fine-tuning. For the weights of a convolutional layer we sample the noise from a Gaussian distribution with mean 0 and the same standard deviation as the weights (all weights of the layer) scaled by a factor η. η is a new parameter for which a value must be chosen.

We add noise before every fine-tuning loop in combination with using a learning rate schedule as described in Section 3.6.1 — starting with 0.1, then 0.01 and 0.001 at a set schedule based on the epoch number, making a learning rate change at epochs 10 and 20, respectively.

An experiment with four fine-tuning loops at 6, 12, 18, and 27 blocks removed (pruning a ResNet-110 initial model) with 10 different values of η ranging from 0.001 to 0.5 shows that adding noise to the weights do improve the final pruned model accuracy when pruning a moderate number of blocks. Smaller values of η work better. Figure 3.26 illustrates the results. Three pruning runs were performed for each value of η on a single model; the *same baseline* bar represents pruning this model with the same parameters, but no noise. The *no noise* bar shows the average and standard error of pruning six different initial models, including the one used for the noise grid search. All pruning in this section is done with the block gradient selection method.

The evidence suggests that adding a small amount of noise to the weights improves performance of pruned model, but not consistently. To further test this hypothesis we use the same pruning configuration and prune all 6 initial models with $\eta = 0.0001$, 0.1112, and with no noise. The results, in Figure 3.27, illustrate that adding a small amount of noise slightly improves the average error by 0.1% to 0.2% when compared to no noise. However there is a large overlap in the standard errors and as a result it cannot be concluded that adding noise improves performance.

It remains as future work to explore the impact of sampling the noise in relation to the standard deviation of each individual convolutional kernel, as opposed to the whole convolutional layer.

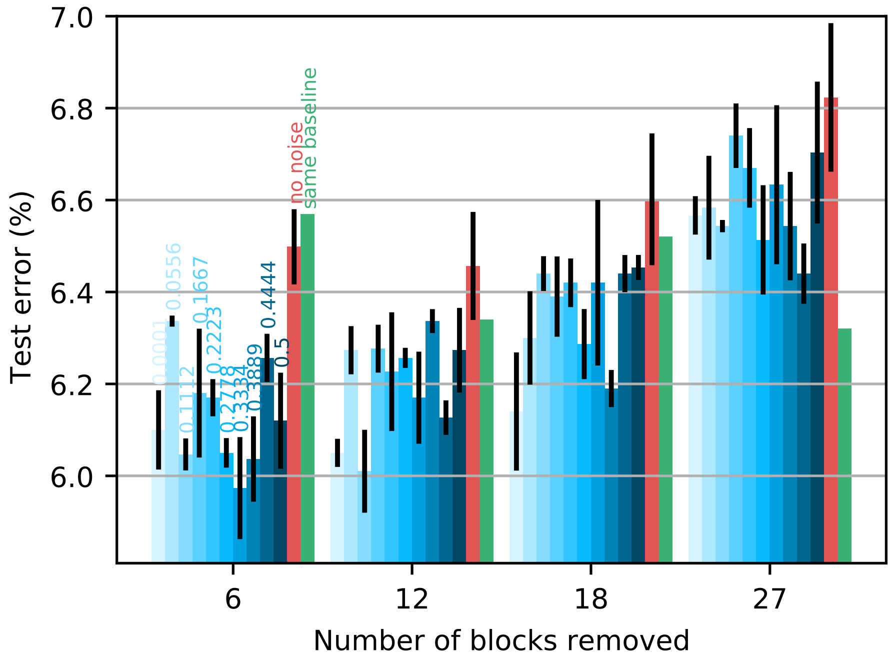


Figure 3.26: Pruning a ResNet-110 to the size of a ResNet-56 with four fine-tuning loops (at plotted pruning steps) with different noise applied to weights using different values of η (number shown on top of bars). For the noise grid search, the value plotted is the mean and standard error of 3 runs with the same initial model. *No noise* is the mean and standard error of the same pruning configuration without any noise for 6 initial models, including the *same baseline* for the noise search, which is also plotted separately for reference.

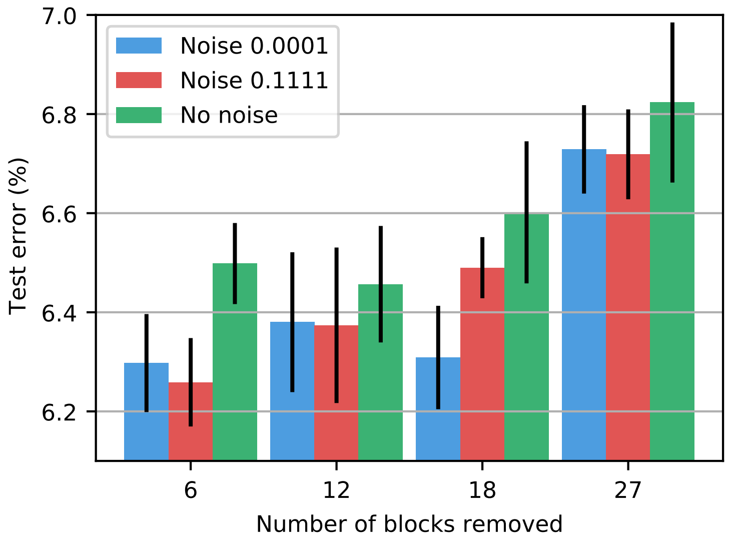


Figure 3.27: Pruning 27 blocks (50%) from 6 initial ResNet-110 with four steps of fine-tuning of 50 epochs. Different amounts of noise (scaled by 0.0001 or 0.1112) has been added to the weights before each fine-tuning loop. Each bar is the mean of 6 runs with standard error bars. Experiments with noise have slightly lower error rates on average, but their error bars overlap with *no noise*.

3.7 Dataset used for block selection

Throughout this chapter the validation set was used for all block selection methods that require data. This is justified by the fact that the validation set is normally used for checking for overfitting the training set, and to select the best model in both the context of early stopping or choosing from multiple training runs. Pruning can be thought of as selecting a model. In all places where it was possible, we report test set accuracies for our results.

There is a remaining concern that using the validation set for block selection causes the pruned model to overfit the validation set. To address this concern, a subset of our experiments has been ran with using a subset of the training set (10%, same size as the validation set) for block selection. We refer to it as the pruning set. For the consistency of comparison the same initial models were used, and as a result the pruning set is included in the initial training of the model, but not subsequent fine-tuning.

When no fine-tuning is enabled, the performance difference between using the validation set or the pruning set is negligible (Figure 3.28). However, when fine-tuning is enabled using the pruning set performs worse than using the validation set by a noticeable margin for oracle, activation change, weights mean and block gradient selection methods (Figure 3.29). The performance gap could be explained by the fact that when using the pruning set, the training set is 10% smaller as we did not reuse data from the pruning set for training. Figure 3.30 shows the same block choices as the made with the pruning set, but fine-tuning with the full training set, confirming that the gap in accuracy was due to a reduced training set.

To conclude, using the validation set for block selection does not seem to make the model to overfit.

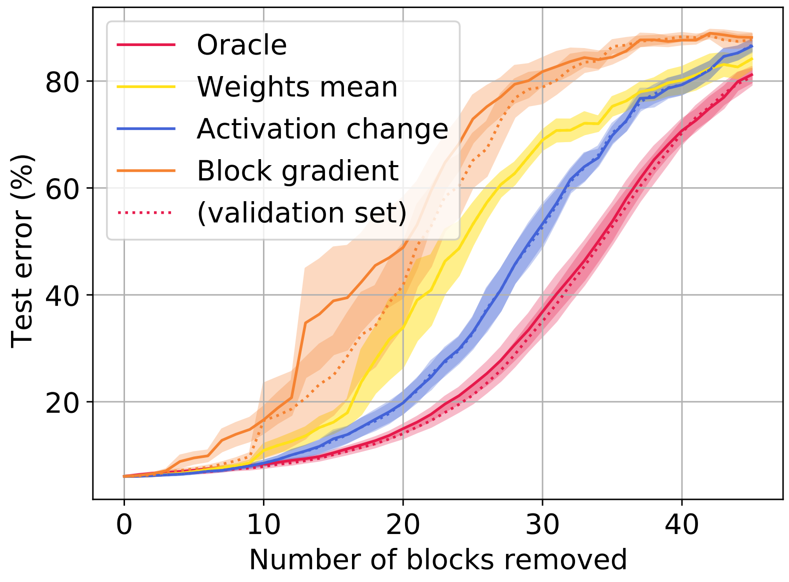


Figure 3.28: Pruning ResNet-110 models using the validation set (dotted lines) and the pruning set (solid lines) with 4 different block selection methods and no fine-tuning.

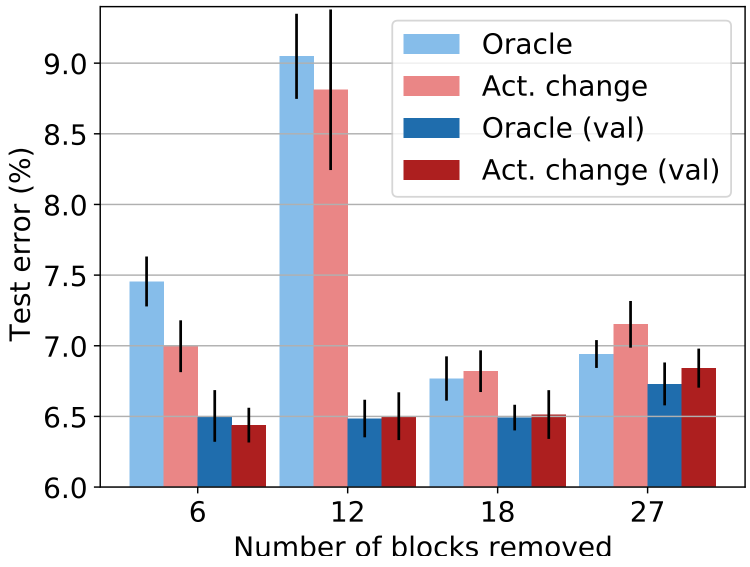


Figure 3.29: Pruning ResNet-110 models using the validation set (marked with *(val)*) and the pruning set. Four fine-tuning loops of 50 epochs are used.

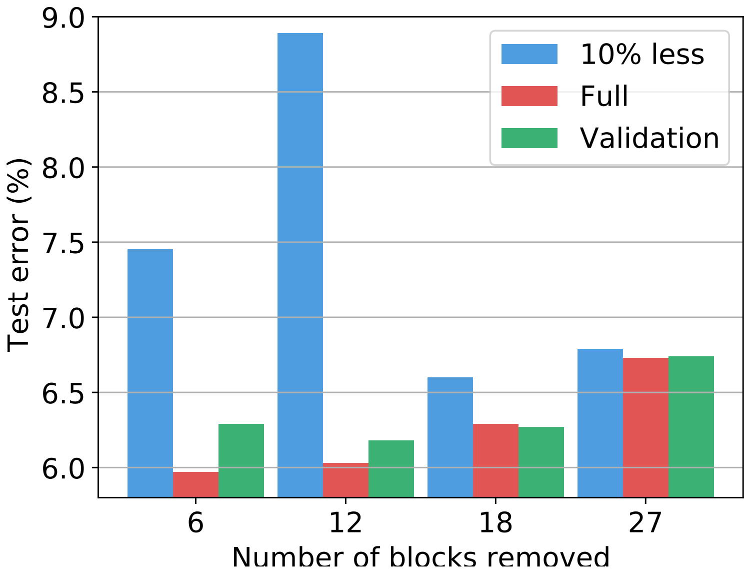


Figure 3.30: Pruning 27 blocks from a single ResNet-110. *10% less* and *full* prune the same blocks in the same order, but *10% less* does not use the pruning set for training. *Validation* is the regular oracle pruning using the validation set to pick blocks and the full training set for fine-tuning.

3.8 Comparison with related works

Table 3.7: Results summary comparing some of our methods with other works. Ours-A is a R-110, single shot, oracle, with 18 blocks removed. Ours-B is a R-110, four shot, oracle, 27 blocks removed.

Model	**Error %**	**Orig. Err %**	**# params**	**Orig. # params**	**Pruned %**	**Fine-tuning**
ResNet-110-pruned-A Li et al. (2016))	6.45	6.47	1.68 ×10ˆ6	1.72 ×10ˆ6	2.3	40 epochs
ResNet-110-pruned-B Li et al. (2016)	6.7	6.47	1.16 ×10ˆ6	1.72 ×10ˆ6	32.40	40 epochs
Ours-A[8]	6.44	5.71	1.14 x 10ˆ6	1.73 x 10ˆ6	33.71	50 epochs
ResNet-110 DBP-0.5 Wang et al. (2019)	6.75	6.03			50% of blocks	300 epochs (3x100)
Ours-B	6.74	5.71	0.93 x 10ˆ6	1.73 x 10ˆ6	49.75	200 epochs (4x50)
ResNet-56 scratch (ours)	6.44		0.85 x 10ˆ6			

Pruning ResNet networks block by block is not thoroughly discussed in the literature, but structured pruning of convolutional neural is a popular approach for making networks more efficient. In Chapter 2 we gave an overview of pruning in the literature focusing on structured neural network pruning, and discussing different major approaches. In this section we compare parts of our results and method.

In Li et al. (2016), a method of pruning convolutional filters layer by layer is introduced using the magnitude of the weights as the saliency metric. For CIFAR-10, the pruned networks are then fine-tuned (at the end of pruning) for 40 epochs with a learning rate of 0.001. The authors compare thier method against random pruning and using activations (feature maps) of each filter. We argue that the comparison included in the paper is not sufficient to draw a strong conclusion as it is only performed without fine-tuning. In the context of block pruning, we have shown that it is not always the case that configurations that are better before fine-tuning are always better after, and we believe the same holds for smaller prunable units.

When pruning a ResNet-110, at 33.71% parameters removed with a similar training schedule (50 epochs, single fine-tuning loop) we obtain a test error of 6.44% on CIFAR-10. Li et al. (2016) results in 6.7% error for 32.4% parameters removed.

Discrimination Block Pruning (DBP) (Wang et al., 2019) is a block selection method that evaluates saliency of blocks by attaching a linear classifier after each block in the large network (to our interpretation, 54 classifiers for ResNet-110). Each classifier is trained independently, with the rest of the network frozen, for 3 epochs with a learning rate of 0.1, 0.01, 0.001 for each epoch, respectively. The authors note that the accuracy of each classifier increases with the depth of the blocks, with the exception of certain blocks where the accuracy drops (or does not change). Denote A_i as the accuracy of the classifier attached to block i. DBP scores block i as $A_i - A_{i-1}$ (the change in accuracy from block i to block $i{+}1$), pruning the blocks with lowest scores. The saliency evaluation is performed only once.

With DBP, pruning a ResNet-110 to 50% of blocks (27 blocks removed) gives 6.75% error on CIFAR-10, similar to our results with four fine-tuning steps of 50 epochs and oracle pruning, 6.73%. DBP results used three fine-tuning loops of, we infer, in the proximity of 100 epochs (reported a fifth of baseline model). Our similarly-sized ResNet-56 baseline trained from scratch outperforms both DBP and our result obtaining 6.44% error. Our thresholding (t=0.01, e=50) experiment is closer to the baseline at 6.48%.

The authors of DBP also evaluate their method using knowledge distillation for fine-tuning, althought we had preliminary experiments with knowledge distillation (not included in final work) we do not have results that are comparable. They achieve 7.09% test error on CIFAR-10 when pruning a ResNet-164 to the size of ResNet-20, lower than our scratch baselines and best runs (our runs start with ResNet-110). Our best results for pruning a ResNet-110 to ResNet-20 are in the region of 7.97% error, with our ResNet-20 baselines trained from scratch averaging at 7.8%.

The results above are summarized in Table 3.7. ResNet block-level pruning was also discussed in Huang and Wang (2018) and Wu et al. (2018).

3.9 Conclusions for this chapter

We have presented a simple method of pruning entire ResNet blocks and empirically evaluated its performance on the CIFAR-10 dataset. Our simple method can cut 50% of the blocks of a ResNet-110 while only losing 0.48% accuracy on the test set. This is a good reduction in size and performance loss for the model, however, training a standard ResNet-56 (which has a similar size) from scratch gives similar results. Training from scratch is computationally cheaper if the cost of the initial training of a large model is taken into account.

Seven different block selection methods were compared, including random, and we conclude that they yield similar error rates for pruned models when fine-tuning is enabled. **Oracle**, **activation change**, and **block gradient** are the best performing, however this is not a strong conclusion. **Block gradient** is the fastest to compute from the three (not including **random** and **mean weights**) as it only requires a single forward and backward pass with one mini-batch of data. **Activation change** requires a forward pass and to save the activations of each block, using the full validation set. **Oracle** is the most expensive, requiring one full evaluation for each currently active block, at each pruning step (although shown to be reusable, we did not reuse saliency metrics).

A notable observation is that **random** performs reasonably well when fine-tuning is enabled. We argue this is because the choice pool is not too large and, although hard-to-train configurations exist, their number is not higher than that of trainable configurations. By using a purpose-built saliency metric (opposite of block gradient, choosing maximum instead of minimum gradient) we showed that bad choices that do not train well exist, resulting in significantly higher error rates.

Different fine-tuning options were explored. First, using a learning rate schedule throughout fine-tuning significantly increases the accuracy of pruned models. As a result, a learning rate schedule was used for most of our experiments. Adding noise before fine-tuning and freezing the classifier when fine-tuning did not show promising improvements.

We introduced the epoch-parameter metric for estimating the cost of fine-tuning. It is a metric that takes into account the size of the model at each fine-tuning loop.

Scheduling fine-tuning loops in the pruning process is a key factor in the final accuracy of pruned models, however too much fine-tuning results in pruning runs that are too expensive, even more expensive than training from scratch. Fixed uniformly spread fine-tuning loops are a quick way to obtain acceptable results for costs that are easy to estimate. For an improved final accuracy, thresholding the accuracy loss enables starting of fine-tuning loops when required and often gives better error rates than fixed schedules. Depending on the threshold, also lower costs. Thresholding the accuracy loss schedules fine-tuning loops towards the end of the prunig process, where the network is smaller and where the biggest accuracy gains are obtained from training. This allows thresholding to keep the cost of fine-tuning low with more fine-tuning loops.

Comparing with related work, our results are equivalent, but a more thorough comparison is left for future work. Finally, we addressed the issue of using the validation set for evaluating the saliency of models, concluding it is an acceptable choice.

In Chapter 4, we discuss more block by block pruning options including using a small number of epochs for each fine-tuning loop followed by regular fine-tuning, and pruning models that were not fully trained.

Chapter 4

Different pruning strategies

This chapter further explores pruning ResNet architectures block by block and analyses different pruning strategies. This chapter includes additional configurations and settings for pruning and fine-tuning that were omitted in Chapter 3.

Each section of this chapter introduces and evaluates a single pruning idea, as well as discusses its motivations, pros, and cons.

The chapter starts with the low and high training schedule (Section 4.1), enabling small fine-tuning loops at every pruning step to mitigate the accuracy loss and reap the benefit of frequent fine-tuning loops at a much lower cost. Next, in an attempt to help the network preserve more of the accuracy of the initial model, the phased pruning method (Section 4.2) scales each pruned block by a decreasing scalar at fine-tuning. Block equalisation, in Section 4.3, limits the pool of blocks vailable for pruning at each step to a single ResNet group, cycling through the groups as the pruning progresses. The idea is to keep the pruned architectures similar to the standard ResNet architecture that are known to train well form scratch. In Section 4.4 we look at pruning before training and pruning after only training the initial model for a small amount. Replacement pruning, in Section 4.5, uses the idea of soft filter pruning to re-initialize pruned blocks instead of removing them, in an attempt to increase model performance. Finally Section 4.6 concludes this chapter.

4.1 Low and high training schedule

Fine-tuning often throughout the pruning process was shown to be a useful setting in Section 3.5. The limitations were the cost of fine-tuning for a significant number of epochs at every pruning step. To reduce this cost but also fine-tune at every pruning step, we introduce a low and a high training schedule. The *high* schedule is the regular fine-tuning as previously done: 50 epochs of training at predetermined pruning steps.

The *low* schedule is fine-tuning for a small number of epochs at every pruning step the *high* schedule isn't active, effectively replacing no training with small training loops. This method is also used in the pruning literature especially in the context of pruning filters from convolutional layers (Molchanov et al., 2016; Luo et al., 2018).

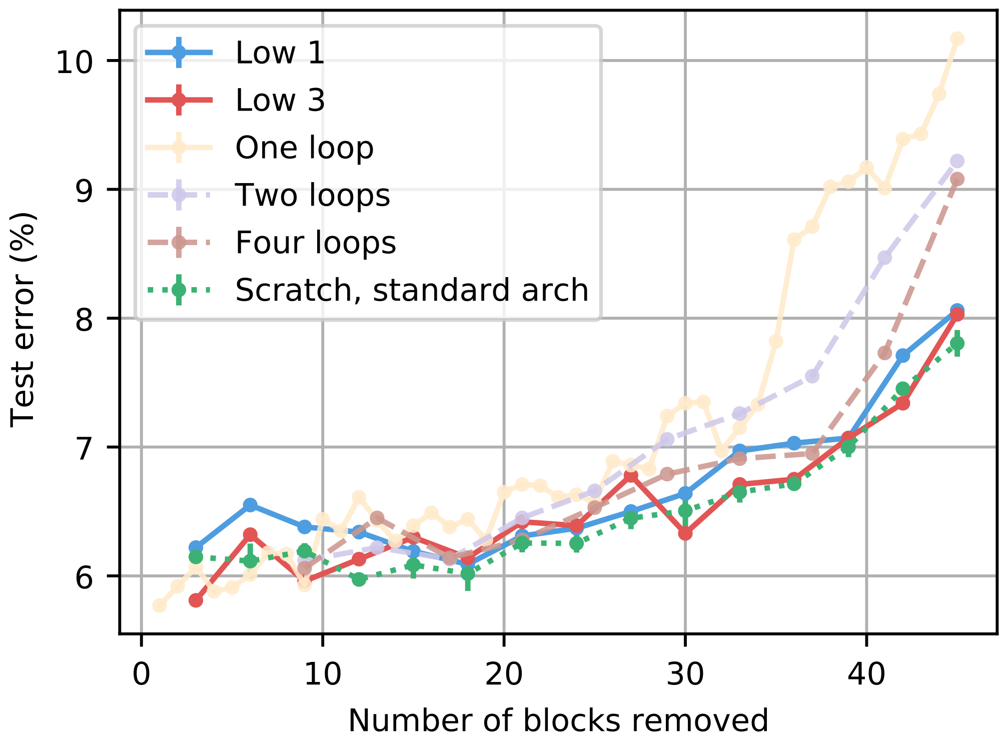
Adding the extra *low* schedule results in a significant increase in pruned network performance when compared to one shot experiments as well as two and four uniformly spread fine-tuning loops. This holds true even for only one epoch per *low* fine-tuning step and a single fine-tuning loop of 50 epochs at the end. The results are illustrated in Figure 4.1a using CIFAR-10, starting with a ResNet-110 model and pruning it for a select number of target network sizes up to removing 45 blocks. Each data point is a full pruning experiment.

The epoch-parameter cost of using a low schedule of one epoch is on par with having two fine-tuning loops for all target sizes in our experiments, but the obtained accuracy is higher for smaller pruned networks. For 3 epochs of low schedule fine-tuning, the cost is lower than four full training loops up to 30 blocks removed for a similar accuracy. See Figure 4.1 for costs and performance.

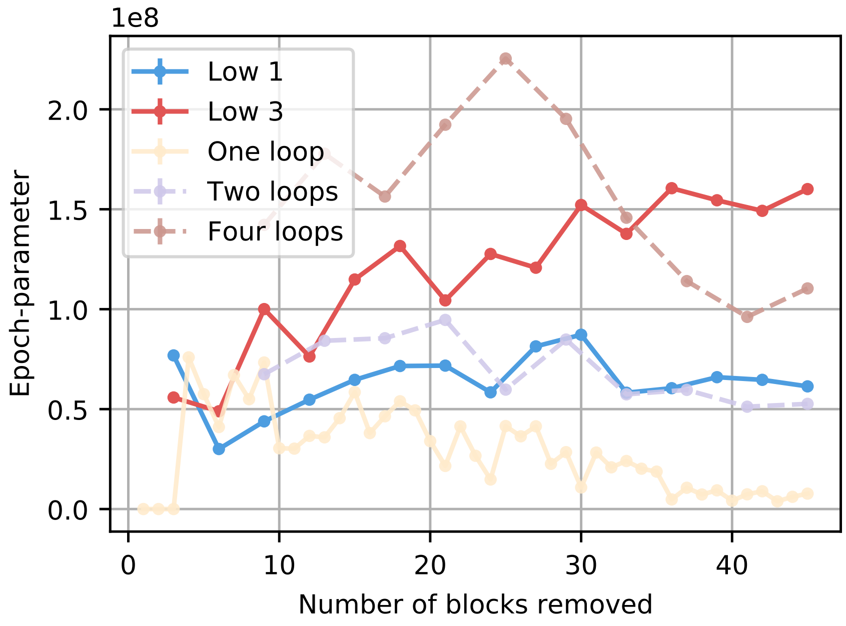
At 45 blocks removed from ResNet-110, a low training schedule of one epoch is similar in accuracy with a 0.1 accuracy loss threshold, but for half the fine-tuning cost. Three epochs of low training is on par with a 0.05 threshold for 45 blocks removed both in fine-tuning cost and accuracy. Table 4.1 and Figure 4.2 show a detailed breakdown of the accuracy and error for 1 and 3 epochs of low fine-tuning, accuracy loss thresholding, and fixed fine-tuning schedules. Thresholding results are from continued experiments with 45 blocks removed as target size and a forced fine-tuning loop at 27 blocks removed, all other runs are independent pruning experiments with target network size at 27 and 45 blocks removed.

Using a low fine-tuning schedule has the advantage of not having to manually select a threshold, which, for best results, must be separately tuned for each individual target size of the pruned model.

A low fine-tuning schedule can also be combined with thresholding, where the number of epochs for the *low* configuration runs at every pruning step, and if the accuracy loss is above the threshold after the *low* loop, the loop will continue up to the *high* schedule. Empirically evaluating this combination is left as future work.

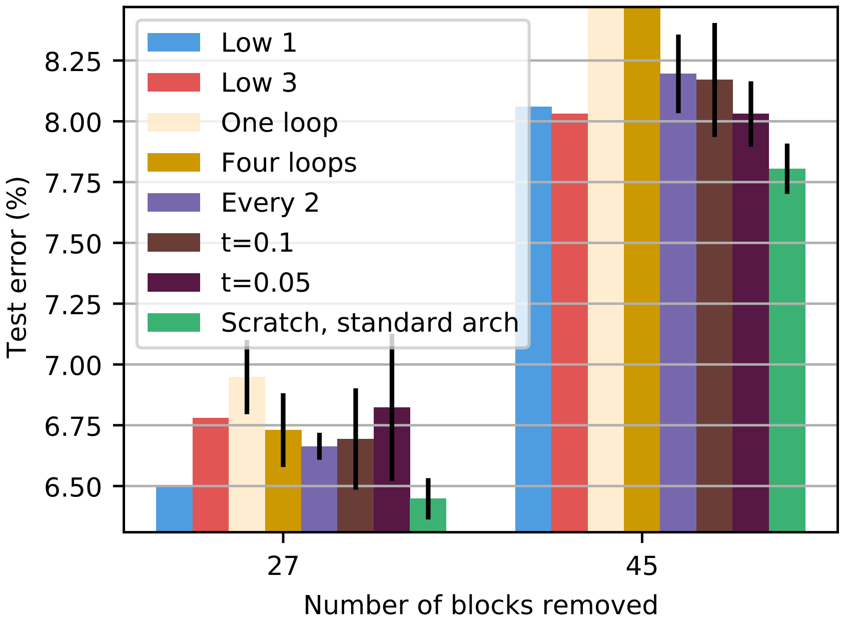


(a) CIFAR-10 test accuracy of each pruning run.



(b) The epoch-paramter fine-tuning cost of each pruning run.

Figure 4.1: Fine-tuning with a *low* fine-tuning schedule of 1 or 3 epochs, followed by 50 epochs of fine-tuning at the last step pruning. One, two, and four loops do not have a low schedule, only a 50 epoch loop once, twice or four times, respectively. Each data point is a full pruning experiment.



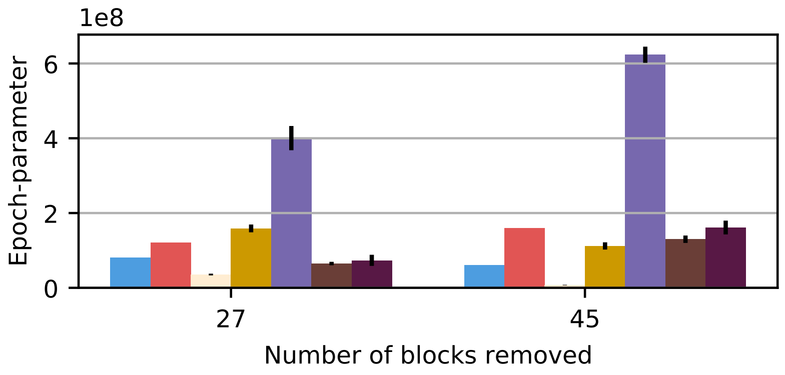


Figure 4.2: CIFAR-10 test error (top) and cost (bottom) of low fine-tuning schedule (with 1 and 3 epochs) compared with accuracy loss thresholding (t) and fixed fine-tuning schedules.

Table 4.1: Cost and CIFAR-10 error of low fine-tuning schedule (with 1 and 3 epochs) compared with accuracy loss thresholding (t) and fixed fine-tuning schedules. *Low* experiments are single runs, others are averages over 6 runs with different starting models.

27 blocks removed			45 blocks removed		
Experiment	Test error %	Cost $\times 10^6$	Experiment	Test error %	Cost $\times 10^6$
Low 1	6.50	81.4	Low 1	8.06	61.4
Low 3	6.78	120.7	Low 3	8.03	160.1
One loop	6.95 ± 0.15	35.6	One loop	10.56 ± 0.55	7.4
Four loops	6.73 ± 0.15	158.9	Four loops	9.24 ± 0.27	112
Every 2	6.66 ± 0.06	400.4	Every 2	8.20 ± 0.16	623.4
t=0.1	6.69 ± 0.21	64.9	t=0.1	8.17 ± 0.23	130
t=0.05	6.82 ± 0.30	73.5	t=0.05	8.03 ± 0.13	161.3

4.2 Phased fine-tuning

We introduce the idea of phased block pruning. After a block i has been selected for pruning, instead of simply removing it, we start a training cycle of S epochs and define a linearly decaying factor

$$\gamma_e = 1 - \frac{e}{S},$$

where e is the epoch number (starting at 1). We rewrite the output of the i-th block as

$$y_{i+1} = ReLU(\gamma_e H_i(y_i) + y_i).$$

During the phased pruning training epochs the weights of the i-th block are frozen so that they do not adjust to γ_e but let the rest of the network adjust to removing the block.

Intuitively this gives the network more time to adjust to the missing block. After finishing the S epochs of phased pruning, the block(s) are removed and we fine-tune by re-training.

Lazy phased pruning is similar to phased pruning but it is applied for more blocks at the same time. We perform pruning without training or phased pruning for k blocks, then we restore the k blocks and perform phased pruning for S epochs by setting γ_e for all the k blocks, after which we finally remove the blocks from the model. Lazy phased pruning is triggered only before regular fine-tuning is scheduled to run, and if training is done at every pruning step, lazy phased pruning is the same as regular phased pruning.

Phased pruning intuitively helps the network adjust to pruned blocks by giving the model a smoother transition. The number of epochs of phased pruning, S, is a new parameter that needs to be picked. We prune a ResNet-110 with a training threshold of 0.01 and a training schedule that triggers re-training at all baseline steps. We train for 50 epochs at every training step, from which S epochs will be for lazy phased pruning (and regular fine-tuning for $50 - S$ epochs). The results for S=10, 20, 30, and 40 are shown in Figure 4.3, and detailed results are in Table A.2. We observe that a value of S=10 shows an increase in performance over simple pruning, however we also notice that, on average, it uses more training epochs after picking the best epoch on the validation set, see Table 4.4.

Figure 4.4a presents phased pruning results with S=10 on the test and validation sets, while training at every pruning step for a total of 50 epochs. Phased pruning only shows a negligible increase in performance compared to simple pruning in this scenario, and uses more training epochs after picking the epoch with best validation accuracy. The results are listed in Table 4.3.

Table 4.2: Simple and phased pruning with different values of S for a total of 10 epochs of training at every pruning step. Showing average test accuracy from 5 runs. Initial model ResNet-110. Dataset used CIFAR-10.

Blocks	**removed**	**Simple**	**S=1**	**S=2**	**S=3**	**S=4**	**S=5**
ResNet-56	27 (50%)	92.73	**92.82**	92.60	92.56	92.52	92.76
ResNet-44	33 (61%)	92.23	92.27	92.22	92.27	92.35	**92.51**
ResNet-32	39 (72%)	91.37	91.65	91.53	**91.75**	91.66	91.22
ResNet-20	45 (83%)	89.62	89.81	89.52	89.82	**88.89**	89.53

Table 4.3: Simple and phased pruning for a total of 50 epochs of training at every pruning step. Showing average accuracy from 5 runs. Initial model ResNet-110. Dataset used CIFAR-10.

Blocks	**removed**	**Simple**	**S=10**
ResNet-56	27 (50%)	93.24 ± 0.06	93.20 ± 0.05
ResNet-44	33 (61%)	92.81 ± 0.10	92.92 ± 0.05
ResNet-32	39 (72%)	92.31 ± 0.06	92.12 ± 0.08
ResNet-20	45 (83%)	90.72 ± 0.15	90.92 ± 0.08

Reducing the total amount of training to 10 epochs per step and using phased pruning with S=1 and S=2 does improve the performance of pruned models (compared to simple pruning), as illustrated in Figure 4.4b. Table 4.2 shows more detailed results with different values of S.

Although intuitively phased pruning forces the network to adjust to a block being removed, our results show that it does not give an advantage over simple fine-tuning. It is possible that a more fruitful approach is to introduce controlled stochasticity in the fine-tuning process. This can be done by making γ_e a binary mask that takes the value 1 with probability $1 - \frac{e}{S}$. Sampling of γ_e can be done at every epoch or minibatch. Each block to be pruned in the lazy version can have a different value for the mask. This approach is left as future work.

Table 4.4: Number of epochs performed while pruning a ResNet-110 on CIFAR-10 using a training threshold of 0.01 and training schedule for each baseline point. Total of 50 epochs per training step (S phased pruning + 50-S fine-tuning). Picked column shows total number of epochs used after choosing the best validation accuracy at every pruning step. Showing average values from 5 runs.

Arch (blocks removed)		**Phased**	**Fine-tuning**	**Picked**	**Total**	**# steps trained**	**Accuracy**
ResNet-56 (27 blocks)	S=0	0	243	243	350	7.00	93.06 ± 0.05
	S=10	68	199	268	342	6.83	93.09 ± 0.07
	S=20	133	135	268	333	6.67	92.96 ± 0.05
	S=30	204	80	284	340	6.80	92.95 ± 0.05
	S=40	280	41	321	350	7.00	92.97 ± 0.05
ResNet-44 (33 blocks)	S=0	0	373	373	560	11.20	92.62 ± 0.06
	S=10	113	335	448	567	11.33	92.76 ± 0.07
	S=20	220	229	449	550	11.00	92.72 ± 0.10
	S=30	324	123	447	540	10.80	92.52 ± 0.10
	S=40	440	58	498	550	11.00	92.59 ± 0.10
ResNet-32 (39 blocks)	S=0	0	570	570	860	17.20	92.17 ± 0.10
	S=10	173	492	665	867	17.33	92.18 ± 0.07
	S=20	340	358	698	850	17.00	92.17 ± 0.05
	S=30	504	190	694	840	16.80	91.92 ± 0.10
	S=40	680	87	767	850	17.00	92.11 ± 0.10
ResNet-20 (45 blocks)	S=0	0	781	781	1160	23.20	90.88 ± 0.05
	S=10	233	647	880	1167	23.33	90.70 ± 0.10
	S=20	460	472	932	1150	23.00	90.69 ± 0.03
	S=30	684	273	957	1140	22.80	90.59 ± 0.12
	S=40	920	119	1039	1150	23.00	90.51 ± 0.08

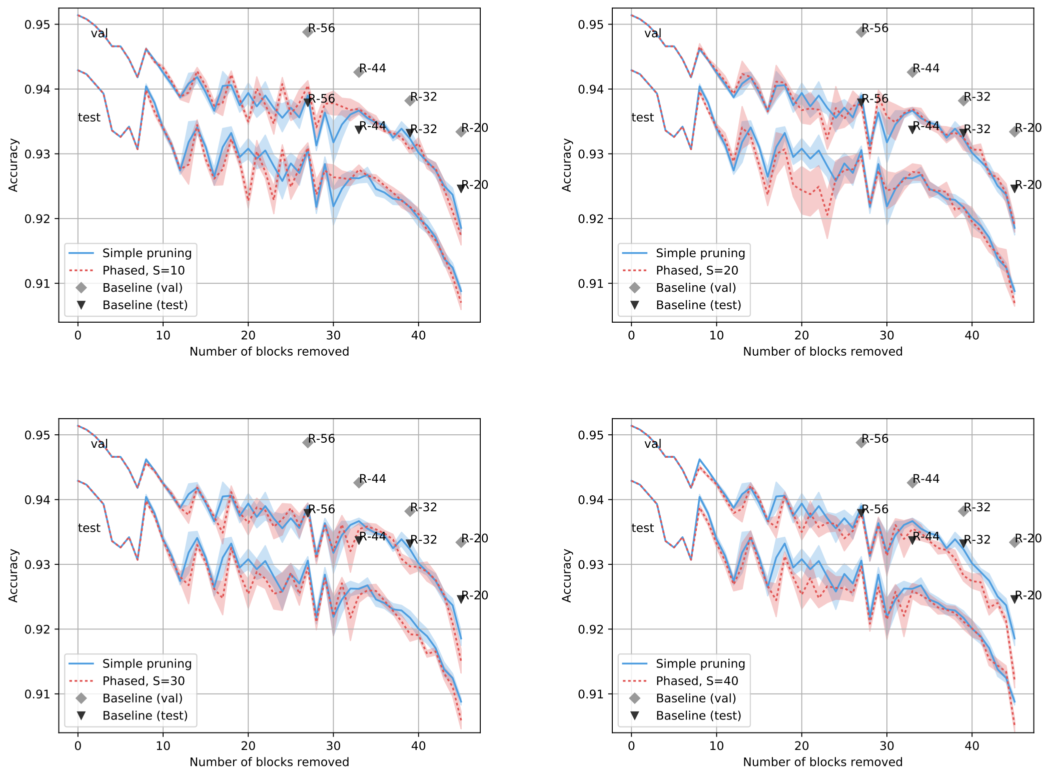
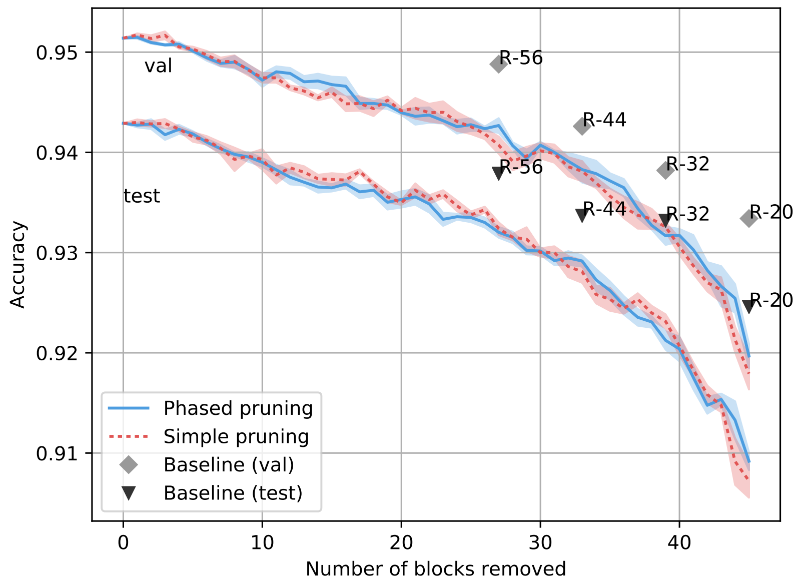
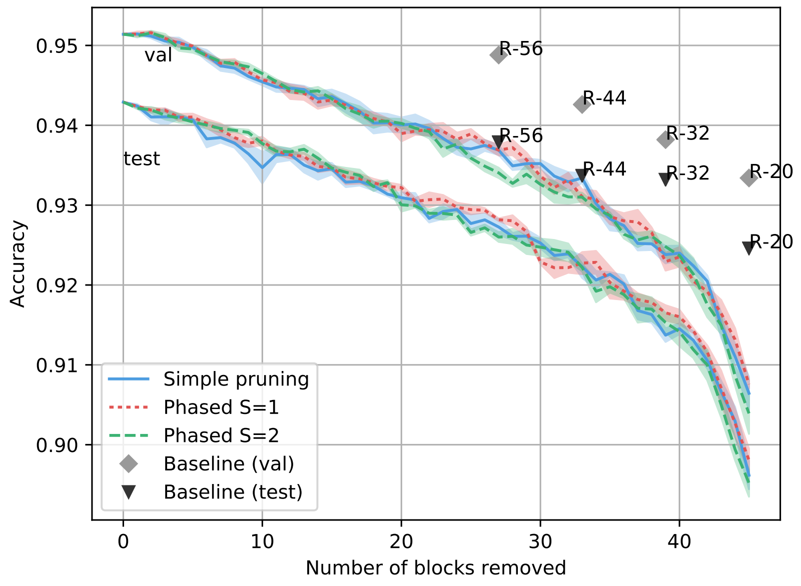


Figure 4.3: Lazy phased pruning with different values of S. Using a training threshold of 0.01, and a schedule for all baseline points. Total number of epochs at every training loop is 50, S for lazy phased pruning and 50-S for fine-tuning. Initial model is ResNet-110 trained on CIFAR-10. Baseline points are the standard ResNet architectures trained from scratch. Lines show mean accuracy for 5 runs.



(a) Phased pruning S=10 followed by fine-tuning for 40 epochs. Simple pruning with fine-tuning for 50 epochs at every step.



(b) Phased pruning after each block removed with S=1, S=2 followed by 9 and 8 epochs of fine-tuning, respectively. Simple pruning performed with fine-tuning for 10 epochs after each block removed.

Figure 4.4: Phased pruning with different values of S and different amounts of fine-tuning. Initial model is ResNet-110 trained on CIFAR-10. Baseline points are the standard ResNet architectures trained from scratch. Lines show mean test accuracy for 5 runs and coloured background is mean±standard deviation.

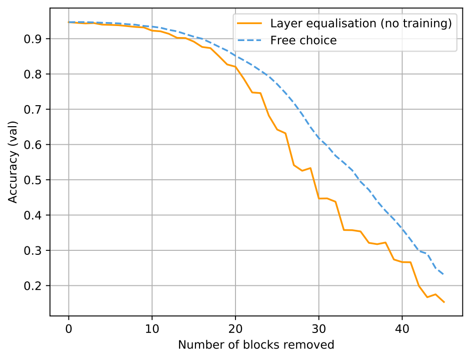
4.3 Block equalisation

We observe that the ResNet architectures with unequal number of groups per blocks obtained by pruning do not perform as well as the standard equal blocks per group configurations[1], even when trained from scratch (see Figure 3.17).

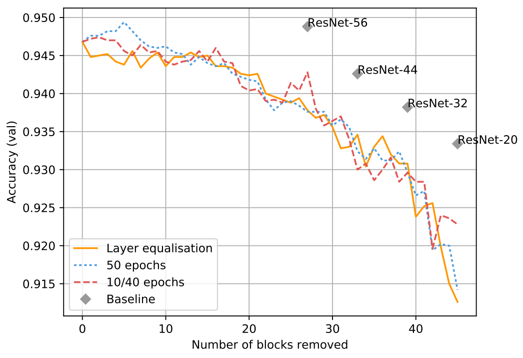
A new constraint has been added to the pruning method such that the blocks picked for pruning belong to a given group. At each pruning iteration the chosen group is changed such that the number of layers per group is equal every 3 pruning steps. Figure 4.5a shows this *block equalisation* technique with no training using oracle pruning, and as expected it is performing slightly worse than when choosing blocks freely. Figure 4.5b compares the block equalisation method with free block choice with fine-tuning enabled. Block equalisation with fine-tuning is comparable with the regular pruning method with fine-tuning, both being outperformed by the baseline models trained from scratch.

We conclude that block equalisation does not help to improve the accuracy of pruned models despite the forced constraint to use architectures that are known to work well when trained from scratch. From this it can also be noted that the weights themselves are more important than the ResNet architecture configuration.

[1]Note that *standard* (as used in He et al. (2016)) choices for ResNet configurations trained on ImageNet do not have equal numbers of blocks per group, but the ones for used for CIFAR-10 do.



(a) Block equalisation and free choice pruning without training.



(b) Block equalisation trained at every pruning step for 40 epochs and 10 epochs of phased pruning. 10/40 denotes the same training schedule but with a free block choice.

Figure 4.5: Block equalisation method for obtaining pruned ResNet configurations similar to the baselines.

4.4 Pruning before training

In this section we ask whether pruning can be effective without a pre-trained model.

We took a randomly initialized ResNet-110 and trained it to convergence using 500 epochs on CIFAR-10, same as all our baselines in Section 3.2. It has 5.51% test accuracy for the best performing epoch on validation (475). A checkpoint was saved with the weights of the random initialization, after first epoch, and after every 10 epochs of training thereafter. The test accuracies of selected checkpoints are in Table 4.5 (column **Initial error**).

Every checkpoint was pruned using the oracle block selection method with a single fine-tuning loop at the end of pruning. Two target sizes were used, ResNet-56 (27 blocks removed) and ResNet-20 (45 blocks removed). They are computed in independet experiments. Fine-tuning was performed such that the total training amount is 500 epochs, for instance the checkpoint *200* was originally trained for 200 epochs and fine-tuned for 300 epochs. A learning rate of 0.1 was used for the start of fine-tuning, and multiplied by 0.1 as the training progressed, at 50% and 75% marks in the training process (e.g. epochs 50 and 75 for 100 epochs of fine-tuning).

We observe that, excluding the random initialization, pruning models that were not trained to converge with a large number of epochs (but combined initial and pruning training epochs to be equal to 500, the same as our scratch experiments), is on par or better with pruning models with a small fine-tuning budget (one fine-tuning loop of 50 epochs, *one loop* in the table), that have been trained to convergence.

Training the initial model as little as one epoch improves pruning performace by over 1% at 27 blocks removed and 4.5% for 45 blocks removed. This is likely explained by the fact that even one epoch of training changes the weights in such a way that our *oracle* saliency metric can effectively avoid removing the most important blocks. The last two checkpoints we have used, 400 and 450 epochs of training for the initial model, do not perform very well when pruned. The explanation is that only 100 or 50 epochs, respectively, are left for the final fine-tuning. We have previously shown that a single loop of fine-tuning is often worse than other fine-tuning schedules for a small number of fine-tuning epochs.

When compared with one loop fine-tuning with 400 epochs, checkpoints from epochs 30 to 350 seem to perform similarly well, with the rest of epochs being slightly worse. The total cost is, however, much lower when starting with checkpoints from early epochs since fully training the initial model is not required.

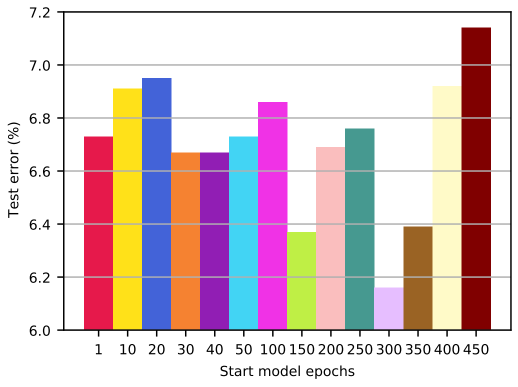
Table 4.5: Pruning a ResNet-110 from different checkpoints taken from the initial training, starting at the random initialization. The *Checkpoint* column denotes the epoch when the checkpoint was taken. Showing test error on CIFAR-10. Each checkpoint was pruned to two different target sizes: 27 and 45 blocks removed, and a single fine-tuning loop at the end of pruning.

Checkpoint	Initial error %	Fine-tuning epochs	Error % at 27 blocks pruned	Error % at 45 blocks
Random init	90.00	500	7.80	13.77
1	80.47	499	6.73	9.25
10	32.41	490	6.91	8.70
20	18.22	480	6.95	8.47
30	18.57	470	6.67	8.67
40	13.87	460	6.67	**8.25**
50	13.12	450	6.73	8.86
100	19.75	400	6.86	8.41
150	11.22	150	**6.37**	8.38
200	12.73	200	6.69	8.51
250	6.89	250	6.76	9.45
300	5.75	200	6.16	8.79
350	5.66	350	6.39	9.14
400	5.58	100	6.92	9.85
450	5.55	50	7.14	9.88
Scratch			6.21	7.54
Low 3			6.78	8.03
One (e=50)			6.95	10.56
One (e=400)			6.45	8.66

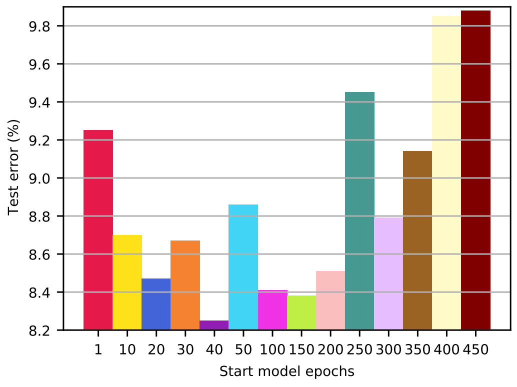
The results are shown in Figure 4.6 as well as detailed in Table 4.5. Note that in this section we used the pruning set for block selection, and as a result the training set for fine-tuning was 10% smaller[2].

The learning rate for training the initial model started at 0.1, changed to 0.01 at epoch 250, and to 0.001 at epoch 375. This explains the big decrease in error for the initial model between epochs 200 and 250. Another experiment can be design such that more models are trained with different epoch budgets, chosen steps ranging from 1 to 500, same as the previous setup, but for each budget a learning rate and learning rate schedule is picked independently to best train the model within the given epoch budget. We expect this to give better results overall, for initial models and for pruned models, especially for initial training budgets in the region between 50 and 250 epochs. This is, however, left as future work.

[2]Unfortunately, this was due to a mistake on our end and time constraints prevented us from re-doing the experiments.



(a) Target size: 27 blocks removed.



(b) Target size: 45 blocks removed.

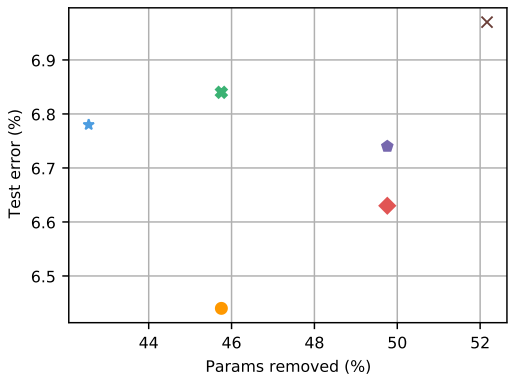
Figure 4.6: CIFAR-10 test error for pruning different checkpoints taken at different epochs from the training of a ResNet-110 model. The random initialization checkpoint was omitted for brevity. Fine-tuning performed once at the end of pruning such that the total number of epochs is 500. Pruning was performed with the *oracle* block selection method.

Another area of future exploration is using the gradient block selection method in the same context as van Amersfoort et al. (2020) – pruning all blocks in one saliency evaluation from random initialization. Also to investigate is using different fine-tuning schedules, such as a low and high schedule or accuracy loss thresholding when pruning from different checkpoints taken throughout the initial training of a large model.

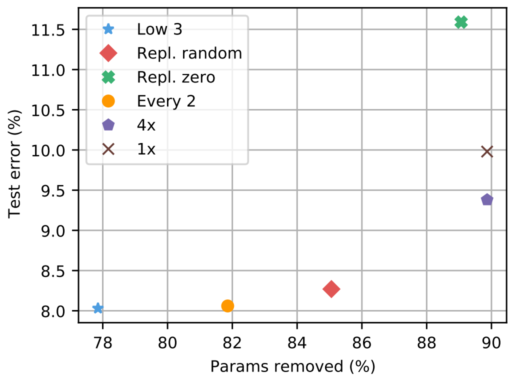
4.5 Replacement pruning

In this type of pruning we keep the number of blocks for the network a constant. When a block is selected to be pruned, instead of removing it, it is reinitialized with random weights or set to zero (effectively, replaced). This is effectively not removing any blocks. However, instead of choosing to prune a single block at a time, we choose a group of k neighbouring blocks and replace them with one single randomly initialized or zero-weighted one.

A preliminary experiment indicates that replacement pruning may be a promising method (see Figure 4.7a), however it is not consistent throughout the pruning process (Figure 4.7b). In this experiment, $k = 2$ blocks that are next to each other are pruned at a time. The saliency metric used is *oracle*, but in this context it evaluates the accuracy of the network by iterating over (and temporarily pruning) groups of two consecutive blocks. We use a *low* training schedule of 3 epochs followed by 50 epochs at the end of pruning (with the usual learning rate schedule) for both target sizes (27 and 45 blocks removed are independent runs).



(a) 27 blocks removed.



(b) 45 blocks removed.

Figure 4.7: Replacement pruning $k = 2$ consecutive blocks with a randomly initialized block or with a block with weights set to zero. 4x and 1x are four and one fine-tuning loops, respectively. Each data point is an independent pruning run. Test error is on CIFAR-10.

The performance of replacement pruning in our experiment may be affected by the fact that two blocks are pruned at once, instead of one, possibly forcing the pruning process to prune blocks that would otherwise be avoided. Another factor can be the training schedule: a randomly initialized block might benefit from isolated training (freezing the rest of the network for a small number of epochs) before the regular fine-tuning loop.

Replacement prunig was inspired by Soft Filter Pruning (He et al., 2018), where convolutional neural networks are soft pruned at first (weights of filters reset to 0, but remaining trainable), and then hard pruned (filters removed). Our experiment is different in setup from Soft Filter Pruning, as we do not perform soft pruning. Instead, we remove more than one prunable units and replace with one, such that the network size decreases at every step. It is left as future work to experiment with a more similar approach: preprocessing the large model with block by block soft pruning *before* starting to prune the model block by block as in our regular setup.

4.6 Conclusions

In this chapter we have investigated five strategies applied in the context of pruning ResNet models block by block.

The *low* and *high* fine-tuning schedule where training for a small number of epochs is done at every step, and a regular fine-tuning loop at the end is evaluted. The results suggest a low training schedule can be a cost effective way to prune without any extra hyper-parameter (like an accuracy loss threshold), and it works the same for all pruning steps. Using a low training schedule of 3 epochs followed by 50 epochs at the end of pruning gives an error rate for pruned models similar to that of training small models from scratch, for all targer pruning sizes.

Phased fine-tuning is the idea of gradually pruning a block, allowing the network to adjust to the change. This is achieved by scaling the output of the pruned block by a linearly decreasing scalar at fine-tuning time, for a subset of the total fine-tuning epochs. The results show a slight improvement in accuracy for certain target sizes, but not consistently throughout the pruning process.

Block equalisation guides the pruning so that pruned models are similar in block configuration to the standard ResNet models know to train well from scratch. Despite this constraint and the fact that pruned configurations do not train well from scratch, this constraint impairs pruning performance before fine-tuning, and is similar to choosing blocks freely when fine-tuning is enabled.

Pruning a ResNet-110 starting from different checkpoints taken during the inital training process, including the random initialization, shows that block by block pruning is possible without training the initial model to convergence. After only one epoch of initial training, followed by pruning and fine-tuning of 499 epochs (total epochs same as our baselines), the error rates are within 2% of the pruning runs which start a fully trained model. This can be further experimented with different fine-tuning schedules with the purpose of reducing the total training cost, motivated by previous results which suggest that adding fine-tuning loops throughout training improves performance for lower cost.

Finally, inspired by soft filter pruning, we introduced replacement pruning, where instead of selecting one block at a time for pruning, we select two. One is pruned, and the other one's weights are reset to either zero or random. Our preliminary experiment shows this method might be worth further consideration, perhaps in a setup more similar to soft filter pruning. The accuracy is similar ($\pm 0.2\%$) to that of other pruning methods when using reset to random, however, we have not studied the effect of pruning two consecutive blocks at a time in an isolated setting, which is also left as future work.

Chapter 5

An experiment management framework

Research and development of machine learning systems often involves running a large number of experiments, some of which take a long time to finalise. Effectively managing the process of implementing a new idea, running experiments to confirm or dismiss it and analysing the results is a challenging task when the number of experiments is large. To make it even harder, software bugs and human coding mistakes are inevitable when writing code, especially so in research where the code is considered a means to testing theories and is often not written by following good software engineering principles.

As part of my pruning work I ran over 1000 experiments, probably many more if all the ones that have been deleted due to human errors or bugs were counted. At times, I would batch about 100 experiments on the University's supercomputer. To be able to handle this number of experiments I started writing small scripts to automate different tasks, such as finding all experiments in a directory tree that have a specific parameter, or finding experiments that have similar parameters to a given experiment, and so on. I have also written utility tools to parse the logs of experiments and derive useful data from them. Early on in my pruning work I made the decision to save plenty of data the experiments were generating as structured logs (a simple format: one JSON object per

Figure 5.1: The logo of the dbx experiment management tool.

line). At the time, there was no plan to use all the data saved in these logs, but later it proved to be useful. This log format along with a few other experiment storage ideas have been extremely helpful tools in managing my experiments. The project presented in this chapter, `dbx`, is the evolution of my utility scripts and experiment storage format in the hope that they will become useful for others and to more directly address the most major pain points in managing machine learning experiments.

This chapter starts with identifying the tasks that need to be performed when doing machine learning research and working with many experiments, including designing experiments, running them in a reproducible way, storing the results and being able to efficiently and effectively make use of these results. Some of the required tasks for analysing results include searching for experiments by parameters and code versions, querying, plotting and comparing experiment results, and being able to quickly re-run an experiment with or without changing some of the parameters or code versions.

A set of key requirements for a machine learning experiment software will be presented and followed by the design of the `dbx` toolkit and the current implementation, along with some examples of how it works and our plans for the future. A comparison with other experiment management solutions is also provided, focusing on comparing with the solutions that are most similar to ours.

Our toolkit, `dbx` (Figure 5.1), is a suite of software tools that could aid researchers with experiment management and analysis. It is a set of small tools that aim to be agnostic to any machine learning framework, flexible to use in different environments and non-intrusive such that it allows researchers to work as they normally do.

This work was presented at the Workshop on MLOps Systems[1], part of the Third Conference on Machine Learning and Systems (MLSys) 2020[2] conference in Austin, Texas, USA.

5.1 Background

Managing experiments and results in machine learning is a challenging task and more often than not it is the responsibility of the researcher to organise the work in a systematic and useful manner. Failing to do so results in loss of work, human and computer time and resources, and delays in the research project at hand. Manually organizing experiments and results is, however, a tedious and time-consuming task and grows in difficulty with the number of experiments to keep track of.

Managing machine learning experiments and results is a common problem faced by many researchers in the field. Although there are tools available to address this problem, many

[1] https://mlops-systems.github.io/

[2] https://mlsys.org

Table 5.1: Work mode categories for machine learning research projects.

	Quick experimentation	**Comprehensive study**
Core focus	Speed of iteration, learning.	Detailed understanding, comprehensive analysis.
Experiments	Mostly for finding promising results or validating ideas. Learning quickly.	Understanding and explaining results or hypothesis, reproducing early results on different conditions (e.g. datasets or model architectures.)
Code	The code is written for quick experimentation. Code quality is often not a priority.	The code quality is important. Bugs or other errors are of high concern. Code is likely to be reused or even published.
Metrics and analysis	Small number of metrics, quick analysis. More metrics could be useful but only if easily obtained. Metrics may change often between old and new experiments.	More metrics and detailed analysis. The metrics used are now consistent between experiments.
Reproducibility	Not of high concern at this stage, but could be useful if easily obtained.	Needs to be taken into account. Reproducing results on different conditions (e.g. datasets or model architectures) becomes important at this stage.
Visualisations	Visualisations that are easy to build and help iterate quickly. Publication quality visualisations are not necessary. New types of visualisations or interpretable outputs are produced as the project progresses. Visualisations are debugging and learning tools.	The quality of these outputs becomes important, especially if the work is to be published. Iteration on the actual formats of the visualisations and other similar outputs becomes part of the process. It is important that all required data is easily available. Visualisations are communication tools.

have different drawbacks like requiring too much setup, not being available for free or being too restrictive in the way experiment code needs to be written and run. As a result there is no de facto solution.

The typical work mode of a machine learning researcher can be split in two categories: the quick experimentation mode and the comprehensive experimentation mode. In quick experimentation, the goal is to try many ideas and iterate quickly until the work converges to something that can be explored in detail. The comprehensive mode is when a possibly fruitful idea is found through quick experimentation and it requires more extensive validation. These modes are described in the most vague terms since each project or idea will required different tasks in each mode. An illustration to aid differentiate between the two modes is shown in Table 5.1. A researcher or a team might switch between the two work modes many times. There is no clear separation that can be defined except that the focus will lean towards either speed of iteration or quality and comprehensiveness of results.

An experiment management framework must be useful for both of these work modes, and must not add unnecessary processes or limitations. In both work modes there are tasks that are unavoidable and involve saving and reading data about experiments and other artefacts. A few examples include: saving experiment results to files or databases, generating different visualisations, moving data from one format to another or from one place to another for further processing or understanding, computing aggregate or some sort of summary results, copying results or artifacts from one computer to another, and many more. These tasks are not about machine learning research, they are about managing experiments and results. These are specifically the tasks that experiment management tools aim to automate or otherwise simplify, and `dbx` is no different.

In the quick experimentation work mode, dbx improves the iteration speeds by automating tedious tasks, such as copying results from one computer to another and keeping an easy to query record of all hyper-parameters and code versions used for each experiment. In the comprehensive work mode where the focus is on quality, the tools make it easier to organize and analyse results, produce visualisations, export results in formats that can be read by others and ensure the experiments are reproducible.

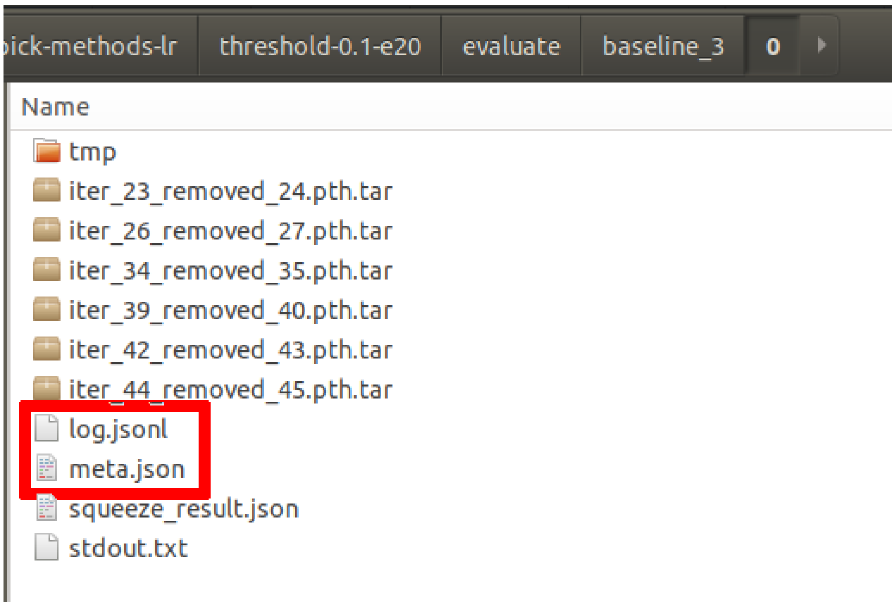
Most of the tasks that dbx automates or provides tools for are tasks that the individual researcher or team would typically implement themselves. Either by reusing work from previous projects or implementing tools one by one as they are needed.

This section has justified the importance and potential usefulness of experiment management tools and frameworks, but does not yet justify why building *yet another one*. In the following subsection the main problems and pain points are clearly identified. They are followed by a list of software requirements derived from them for what we found (through experience and discussions) makes a good software solution for experiment management. Later in this chapter we compare our framework to other works that are available.

5.1.1 Motivation

This project started as small utilities written as part of earlier work, including a large part of the work on pruning. The goal was to collect all these utilities, package them into a standalone project and improve upon them to create a fully featured experiment management framework. The motivation to undertake this comes from the lack of other suitable solutions and the requirements were drafted and iterated on based on my own needs. This section will illustrate a selection of the utilities created as part of earlier works and highlight the problems that they solve.

Early in the pruning work I have made the decision to save experiments in a specified format. Each experiment is stored in its own folder, using the directory tree to keep related experiments together. Each experiment has a metadata file and a structured log as well as all other outputs it produces (example in Figure 5.2). The log must be



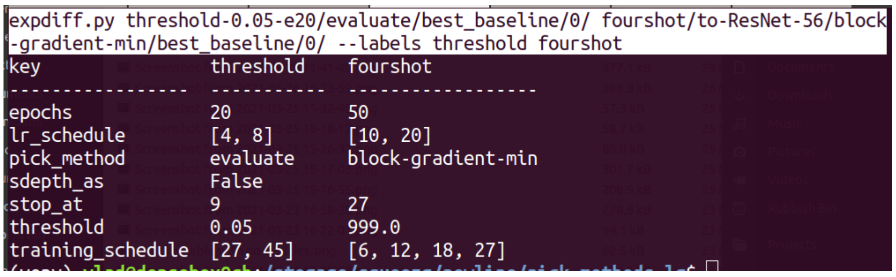
```
Name
tmp
iter_23_removed_24.pth.tar
iter_26_removed_27.pth.tar
iter_34_removed_35.pth.tar
iter_39_removed_40.pth.tar
iter_42_removed_43.pth.tar
iter_44_removed_45.pth.tar
log.jsonl
meta.json
squeeze_result.json
stdout.txt
```

Figure 5.2: Files in a pruning experiment highlighting the experiment log and the metadata.

sufficient to analyze all results and the metadata must be sufficient to reproduce the experiment. This method of organizing is kept for `dbx` experiments as it has proven to be extremely useful. Below is a list of some of the utilites that were written and were very useful for managing the experiments for the pruning work.

`getcmd.py <exp>` outputs the command needed to run the experiment again, formatted for easy editing.

`expdiff.py` prints the parameters that are different for all experiments given (Figure 5.3).



```
expdiff.py threshold-0.05-e20/evaluate/best_baseline/0/ fourshot/to-ResNet-56/block
-gradient-min/best_baseline/0/ --labels threshold fourshot
key                threshold    fourshot
-----------------  -----------  ------------------
epochs             20           50
lr_schedule        [4, 8]       [10, 20]
pick_method        evaluate     block-gradient-min
sdepth_as          False
stop_at            9            27
threshold          0.05         999.0
training_schedule  [27, 45]     [6, 12, 18, 27]
```

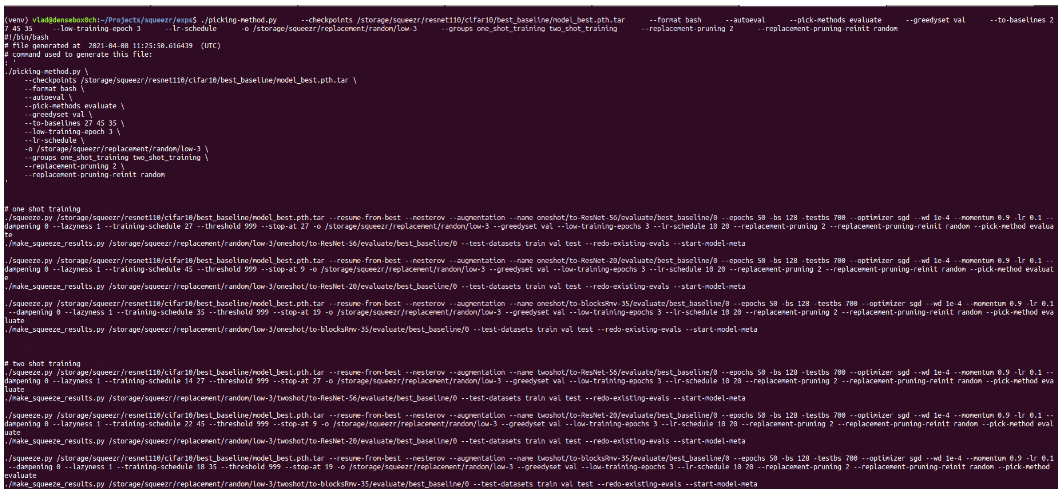
Figure 5.3: The command **expdiff** which prints out the parameters that are different between two or more experiments.

`findlike.py` finds experiments that have similar parameters to the given experiment. It helps to answer questions such as *"are there results for this but with more epochs per pruning step?"*

`list_runs.py` shows a table of all or a subset of experiments, with selected summary values such as final accuracy or pruning block selection method.

`plot_*.py`. There are 21 utilities written as part of the pruning work to plot parts of experiments. All plotting utilities take experiments in the format above and extract required information from the log or other experiment artefacts. The organization in directory trees is used for grouping experiments together for computing aggregated results (such as the mean and standard error).

Experiment list generators were written to create sets of experiments to be run. All generators are based on a set of common functions which enable them to set a base configuration, rotate hyper-parameters (example: generate experiments for oracle and activation change selection methods with and without a learning rate schedule, fine-tuning at every 2, 4, and 8 steps). The output of those generated configurations is a script that could be scheduled with `sbatch` or run locally. The script has a reference to re-generate itself in the header (as a comment) and there is an option (add the `-testing` flag) to test whether all the experiments were run and correct; this is best explained with the screenshots in Figures 5.4 and 5.5.



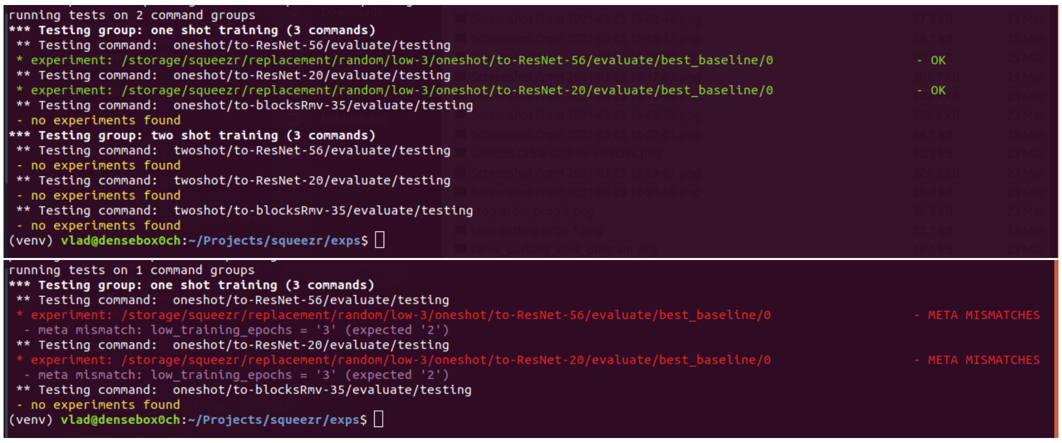
```
(venv) vlad@densebox0ch:~/Projects/squeezr/exps$ ./picking-method.py      --checkpoints /storage/squeezr/resnet110/cifar10/best_baseline/model_best.pth.tar      --format bash      --autoeval      --pick-methods evaluate      --greedyset val      --to-baselines 2
7 45 35      --low-training-epoch 3      --lr-schedule      -o /storage/squeezr/replacement/random/low-3      --groups one_shot_training two_shot_training      --replacement-pruning 2      --replacement-pruning-reinit random
#!/bin/bash
# file generated at  2021-04-08 11:25:50.616439  (UTC)
# command used to generate this file:
: '
./picking-method.py \
     --checkpoints /storage/squeezr/resnet110/cifar10/best_baseline/model_best.pth.tar \
     --format bash \
     --autoeval \
     --pick-methods evaluate \
     --greedyset val \
     --to-baselines 27 45 35 \
     --low-training-epoch 3 \
     --lr-schedule \
     -o /storage/squeezr/replacement/random/low-3 \
     --groups one_shot_training two_shot_training \
     --replacement-pruning 2 \
     --replacement-pruning-reinit random
'


# one shot training
./squeeze.py /storage/squeezr/resnet110/cifar10/best_baseline/model_best.pth.tar --resume-from-best --nesterov --augmentation --name oneshot/to-ResNet-56/evaluate/best_baseline/0 --epochs 50 -bs 128 -testbs 700 --optimizer sgd --wd 1e-4 --momentum 0.9 -lr 0.1 --
dampening 0 --lazyness 1 --training-schedule 27 --threshold 999 --stop-at 27 -o /storage/squeezr/replacement/random/low-3 --greedyset val --low-training-epochs 3 --lr-schedule 10 20 --replacement-pruning 2 --replacement-pruning-reinit random --pick-method evalua
te
./make_squeeze_results.py /storage/squeezr/replacement/random/low-3/oneshot/to-ResNet-56/evaluate/best_baseline/0 --test-datasets train val test --redo-existing-evals --start-model-meta

./squeeze.py /storage/squeezr/resnet110/cifar10/best_baseline/model_best.pth.tar --resume-from-best --nesterov --augmentation --name oneshot/to-ResNet-20/evaluate/best_baseline/0 --epochs 50 -bs 128 -testbs 700 --optimizer sgd --wd 1e-4 --momentum 0.9 -lr 0.1 --
dampening 0 --lazyness 1 --training-schedule 45 --threshold 999 --stop-at 9 -o /storage/squeezr/replacement/random/low-3 --greedyset val --low-training-epochs 3 --lr-schedule 10 20 --replacement-pruning 2 --replacement-pruning-reinit random --pick-method evaluat
e
./make_squeeze_results.py /storage/squeezr/replacement/random/low-3/oneshot/to-ResNet-20/evaluate/best_baseline/0 --test-datasets train val test --redo-existing-evals --start-model-meta

./squeeze.py /storage/squeezr/resnet110/cifar10/best_baseline/model_best.pth.tar --resume-from-best --nesterov --augmentation --name oneshot/to-blocksRmv-35/evaluate/best_baseline/0 --epochs 50 -bs 128 -testbs 700 --optimizer sgd --wd 1e-4 --momentum 0.9 -lr 0.1
 --dampening 0 --lazyness 1 --training-schedule 35 --threshold 999 --stop-at 19 -o /storage/squeezr/replacement/random/low-3 --greedyset val --low-training-epochs 3 --lr-schedule 10 20 --replacement-pruning 2 --replacement-pruning-reinit random --pick-method eva
luate
./make_squeeze_results.py /storage/squeezr/replacement/random/low-3/oneshot/to-blocksRmv-35/evaluate/best_baseline/0 --test-datasets train val test --redo-existing-evals --start-model-meta


# two shot training
./squeeze.py /storage/squeezr/resnet110/cifar10/best_baseline/model_best.pth.tar --resume-from-best --nesterov --augmentation --name twoshot/to-ResNet-56/evaluate/best_baseline/0 --epochs 50 -bs 128 -testbs 700 --optimizer sgd --wd 1e-4 --momentum 0.9 -lr 0.1 --
dampening 0 --lazyness 1 --training-schedule 14 27 --threshold 999 --stop-at 27 -o /storage/squeezr/replacement/random/low-3 --greedyset val --low-training-epochs 3 --lr-schedule 10 20 --replacement-pruning 2 --replacement-pruning-reinit random --pick-method eva
luate
./make_squeeze_results.py /storage/squeezr/replacement/random/low-3/twoshot/to-ResNet-56/evaluate/best_baseline/0 --test-datasets train val test --redo-existing-evals --start-model-meta

./squeeze.py /storage/squeezr/resnet110/cifar10/best_baseline/model_best.pth.tar --resume-from-best --nesterov --augmentation --name twoshot/to-ResNet-20/evaluate/best_baseline/0 --epochs 50 -bs 128 -testbs 700 --optimizer sgd --wd 1e-4 --momentum 0.9 -lr 0.1 --
dampening 0 --lazyness 1 --training-schedule 22 45 --threshold 999 --stop-at 9 -o /storage/squeezr/replacement/random/low-3 --greedyset val --low-training-epochs 3 --lr-schedule 10 20 --replacement-pruning 2 --replacement-pruning-reinit random --pick-method eval
uate
./make_squeeze_results.py /storage/squeezr/replacement/random/low-3/twoshot/to-ResNet-20/evaluate/best_baseline/0 --test-datasets train val test --redo-existing-evals --start-model-meta

./squeeze.py /storage/squeezr/resnet110/cifar10/best_baseline/model_best.pth.tar --resume-from-best --nesterov --augmentation --name twoshot/to-blocksRmv-35/evaluate/best_baseline/0 --epochs 50 -bs 128 -testbs 700 --optimizer sgd --wd 1e-4 --momentum 0.9 -lr 0.1
 --dampening 0 --lazyness 1 --training-schedule 18 35 --threshold 999 --stop-at 19 -o /storage/squeezr/replacement/random/low-3 --greedyset val --low-training-epochs 3 --lr-schedule 10 20 --replacement-pruning 2 --replacement-pruning-reinit random --pick-method
evaluate
./make_squeeze_results.py /storage/squeezr/replacement/random/low-3/twoshot/to-blocksRmv-35/evaluate/best_baseline/0 --test-datasets train val test --redo-existing-evals --start-model-meta
```

Figure 5.4: This command generates a script that runs (or schedules using sbatch) a set of experiments with varying parameters. The beginning of the script contains the command to re-generate itself as a comment.

5.1.2 Identified issues and requirements

Designing and running experiments. Sets of experiments are designed to test a hypothesis or generate baseline results. It is important for reproducibility that the same set of experiments can be re-run easily, with or without minor changes (for example to use a new code version that includes a bug fix, or a slightly different parameter to create a comparison).

Setting up sets of experiments to run or re-run should be a quick and easy process. From experiece we suggest that running experiments should be done in a way the researcher



```
running tests on 2 command groups
*** Testing group: one shot training (3 commands)
 ** Testing command:  oneshot/to-ResNet-56/evaluate/testing
 * experiment: /storage/squeezr/replacement/random/low-3/oneshot/to-ResNet-56/evaluate/best_baseline/0                  - OK
 ** Testing command:  oneshot/to-ResNet-20/evaluate/testing
 * experiment: /storage/squeezr/replacement/random/low-3/oneshot/to-ResNet-20/evaluate/best_baseline/0                  - OK
 ** Testing command:  oneshot/to-blocksRmv-35/evaluate/testing
 - no experiments found
*** Testing group: two shot training (3 commands)
 ** Testing command:  twoshot/to-ResNet-56/evaluate/testing
 - no experiments found
 ** Testing command:  twoshot/to-ResNet-20/evaluate/testing
 - no experiments found
 ** Testing command:  twoshot/to-blocksRmv-35/evaluate/testing
 - no experiments found
(venv) vlad@densebox0ch:~/Projects/squeezr/exps$
```

```
running tests on 1 command groups
*** Testing group: one shot training (3 commands)
 ** Testing command:  oneshot/to-ResNet-56/evaluate/testing
 * experiment: /storage/squeezr/replacement/random/low-3/oneshot/to-ResNet-56/evaluate/best_baseline/0                  - META MISMATCHES
  - meta mismatch: low_training_epochs = '3' (expected '2')
 ** Testing command:  oneshot/to-ResNet-20/evaluate/testing
 * experiment: /storage/squeezr/replacement/random/low-3/oneshot/to-ResNet-20/evaluate/best_baseline/0                  - META MISMATCHES
  - meta mismatch: low_training_epochs = '3' (expected '2')
 ** Testing command:  oneshot/to-blocksRmv-35/evaluate/testing
 - no experiments found
(venv) vlad@densebox0ch:~/Projects/squeezr/exps$
```

Figure 5.5: The generator script when the **-testing** flag is included. It checks whether the experiments in the output folder match the expected configuration and that they are all present. Top shows the output for correct and missing experiments and the bottom shows sample output for parameter mismatches.

typically runs experiments and the experiment management tool should not impose specific styles. An experiment management tool can however aid in running experiments by providing complementary tools. For instance, regardless of how a user initially runs experiments, the experiment saved should store everything necessary to re-run it: code version, input paths, hyper-parameters and so on.

Different styles of running experiments refer to how parameters are set and how the experiment program is run. Parameters can be set through hardcoded variables, configuration files, environment variables, command line arguments, and so on. Apart from discouraging hardcoded variables (because they are impossible to change without editing the code), an experiment management framework should not favour any specific style. Experiments can be run directly by executing file (ie. `./train.py`), inside containers (ie. using docker), or through a *proxy program* (ie. `./proxy ./train.py`). Experiment management software should not add a proxy program to the mix, nor assume containerisation is to be used or not used. It should simply not interfere with how experiments are run.

Using a proxy program or requiring containerisation for experiment management limits the availability and user friendliness of the experiment management tools. On some systems that experiments are run the researcher may not have full access to freely install software, nor have a container engine available. Even if the user is allowed to install a proxy program, it may be difficult to make it available in the `PATH`, making running experiments significantly more complicated than it should be.

Different styles of running experiments may be used for the same project, especially between quick experimentation and comprehensive modes. In quick experimentation, hand-written command line arguments might be sufficient but config files (of some sort) might prove more useful for a comprehensive study.

Saving artefacts and results. Experiments often take a long time to run. It is important that each experiment configuration, hyper-parameter, and result is saved such that it can be easily accessed, searched for, analysed and reproduced. It is also important for reproducibility to keep strict versioning of artefacts that are used by experiments, like starting weights for a model.

Code versions and bug fixes. Code changes. Bugs get discovered and fixed. Saving a reference to the code versions that an experiment was ran with and cross-referencing with future bug fixes is critical to ensure correctness of results and research conclusions. It also improves reproducibility by enabling the use of exactly the same source code to re-run past experiments.

Two useful debug features are code diffs (similar to `git diff`) between two experiments or an experiment and the current version of code, and checking out the exact version of code that an experiment used, perhaps in a new branch in the current project.

A researcher may wish to confirm that all the experiments used in a report or publication use the version of code that is also made publicly available, or at least be able to check that no significant changes that may affect results have been made.

Availability of experiment results. Experiments are often run on many computer systems (clusters, servers, development workstations) and it is critical that experimental results do not get lost or forgotten but are easily available for analysis through a central interface. Some of the computer systems may not be able to share experiment results in real time (for example due to network restrictions on compute nodes) hence synchronizing experiments between different locations is a critical feature.

Add-on metrics. In a research project it is often discovered at a later stage that not all required metrics, or not all required evaluations have been run for a set of experiments. It should be easily possible to run extra programs for each experiment that would fill in missing values. An example would be running a second script that evaluates the checkpoints saved at every N epochs on a different dataset split.

The experiment management software should be able to allow for running extra scripts and adding more information to the experiment log at a later stage, but also to be able to find experiments that lack such information.

Results. Filtering, sorting and searching experiments by their results (or other core metrics) is often required. The results, unlike hyper-parameters or other configuration, change as the experiment progresses or new scripts have been run. The toolkit must support and save results in a useful format.

Portability and flexiblity. All the tools must be usable regardless of the end users' computer systems: a good experiment management tool need not impose restrictions on how experiments are run or on what machines. For example, experiments might run on machines with restricted network access and this must be supported by the experiment management tools.

Making results useful. Saving and organizing the experiments and their results is the first step, but not a very useful step unless those results are easy to query and analyse. The most important issues are finding experiments by name, hyper-parameters, results, code versions or a combination thereof, and extracting only the information that is required from experiment logs (example: validation accuracy vs epoch number).

Easily extensible. Each research project is different and an experiment management tool needs to be flexible and extensible to allow use cases not thought of before. This can be achieved by embracing the UNIX philosophy and splitting the toolset into many independent tools that communicate through well-specified protocols and file formats rather than creating a monolithic project.

Each tool can have different points where it can be extended. A web UI can have the possiblity to add custom pages or visualisations. The command line tool and loggers should be extensible to allow new storage back-ends. The whole ecosystem can be extended with additional tools that use the common APIs and file formats.

5.1.3 Related work

The most similar project to ours is Weights and Biases (Biewald, 2019). It offers a solution to save experiments, experiment metadata (via `wandb.config`), an experiment log similar to ours and files related to each experiment. The cloud solution comes with a handy web-based interface where users can see various plots generated from logs and analyse results as well as access all the saved experiments, metadata and files. It comes with handy integrations for popular machine learning frameworks like PyTorch (Paszke et al., 2017) and TensorFlow (Abadi et al., 2015) making the logging very simple. The two apparent downsides are the fact that it is not open-source, thus it cannot be easily customised or self-hosted [3] and it seems that experiments need to be run on machines with full internet access[4].

Neptune.ML (Inc., 2019) is a similar cloud service that offers a simple python experiment and results tracker easy to integrate with any machine learning platform but it is not open-source (but it has a free tier). Like most cloud services it lacks the ability to run experiments on machines on restricted networks.

Signac (Adorf et al., 2018, 2019) is a Python framework for tracking experiments and workflows and is released as open-source but is not focused on machine learning. It offers

[3]Hosting on premise might be available for enterprise customers.

[4]Or by storing the experiment data locally and manually importing it when online, however this functionality is not supported via the Weights and Biases libraries and will have to be implemented by the user

a simple method to track parameters and values for different experiments but seems to lack many of the required functionalities that would be needed for effective machine learning research, like a system for storing experiment logs or synchronizing experiment results between different machines or storing large files.

ModelDB (Vartak et al., 2016) is an model and experiment tracker that aims to save all the experiment metadata and results and make them available via a web user interface. The documentation seems to be incomplete and the focus to have changed to a new version, currently unreleased, ModelDB 2 which might also become available as a commercial service via the `verta.ai` website. The current ModelDB seems to focus on integrations with the spark.ml (Meng et al., 2016) and scikit-learn (Pedregosa et al., 2011) frameworks.

DVC (Iterative, 2019) is an approach to experiment management by tightly integrating the machine learning research workflow with the source code version control system git. DVC works by creating branches for different experiments and allowing researchers to define workflows to run experiments. DVC tracks large files, metrics as well as source code. It is different from our approach since we create a tool entirely separated from source version control systems, and we belive that tracking experiments and their results have fundamentally different requirements from tracking source code.

Sacred (Klaus Greff et al., 2017) is a python framework that is similar to our proposal in the way that it stores experiment metadata and results in a database to make them easily available later. To use this tool the researcher needs to structure the experiment code in a specific manner to integrate with the framework, which is a disadvantage especially when porting existing projects. It also imposes restrictions in how the python scripts for the experiments are run and how their command line arguments are passed. A considerable advantage is the availability of many front-end interfaces that work with Sacred: Omniboard, Incense, Sacredboard, Neptune, and SacredBrowser.

Trains.ai (Allegro.ai, 2019) is an open-source python tool that integrates with popular machine learning frameworks and logs experiments and results by sending data to a server (also provided as open-source). It lacks the possiblity of running experiments in a constrained network environment. The logger supports saving custom data as well as automatically collecting plenty of useful information. The integration with existing code is done with a few lines of code.

Studio.ML (Studio.ML, 2019) is an open-source experiment manager that can save the environment and metadata, but it seems to require users to run experiments in a certain way (e.g. using the studio command). Similarly, Datmo (Acusense Technologies, 2019) is an open-source tool for model management focused on managing the workflow from training models to deploying them to production. Datmo forces users to run experiments in Docker containers and seems to lack the ability to store structured logs.

Other similar projects include FGLab, Guild.AI (TensorHub, 2019), MLFlow, ModelChimp. There are various cloud services providing experiment management solutions, like Comet.ml (2019) and valohai.com (2019). FORGE (Kosiorek, 2019) is a small tool that helps with defining and separating models, datasets and experiments, but it is not enough for effective experiment tracking and management.

TensorBoard (Abadi et al., 2015) is a visualization tool that can aid with experiment management. It is tightly integrated with the TensorFlow framework but packages exist to save TensorBoard logs from other frameworks (like tensorboard-x for PyTorch). TensorBoard can show logs from many experiments and has experiment and log filtering features. It works well for comparing a small number of experiments but it slows down for a large number and lacks other experiment management features like showing experiment metadata, search by hyperparameters. Visdom (Facebook Research, 2019) is a tool for visualizing live data and works well with PyTorch and Numpy. It can be used to plot live updates from running machine learning experiments.

5.2 Requirements

The issues that need to be addressed are presented in 5.1.2. Based on these issues we propose the following requirements for a software solution that aids experiment management and machine learning research.

5.2.1 Experiment repository abstraction

Machine learning projects are made of many experiments. A key requirement for experiment management is to be able to perform operations with sets of experiments: searching through them by different criteria, comparing aggregate results and comparisons, adding new experiments to the set, copying the set to a different location (or machine), and other related operations.

A repository of experiments is an abstraction used to represent a set of experiments. It must be easy to clone and synchronise with other repositories, such that distributed work (in a team or using many computer systems) can be easily handled.

Partial repository imports and exports must be supported. For example, a compute node may create a new local repository where it saves all experiments it runs, and it imports all these experiments into a central repository at a later stage.

5.2.2 Experiments and results

An experiment is loosely defined as running one or more programs that generates one or more related artefacts. A simple example is training a neural network on a dataset producing a trained model. A second script could be evaluating the model from the previous step on a test dataset. Both scripts are part of the same experiment.

For good management of experiment results we propose that each experiment must have:

- a human-readable name,
- a type (user-defined type, ie. *training*, *pruning*, *baseline*),
- a unique ID,
- a set of parameters,
- a results object,
- one or more runs (invocations of a script or program),
- one or more logs,
- attachments.

The experiment metadata is a key-value store for saving information about the experiment, such as the parameters used, other user-specified configuration, experiment name, kind and so on are saved in the metadata. Each run also has metadata saving information about the version of code used, system environment, timestamps, command line arguments, and anything else that might be required for reproducibility.

The experiment log is for storing all the events that happen during the experiment. The experiment log is an append-only list of immutable events. Each event has a name and arbitrary key-value data supplied by the user. Files stored outside of the log can be referenced. The experiment log is the main resource of information for analysing experiments and their results.

Experiment attachments can be external or tracked by the tool but must always have a checksum stored within the experiment (metadata or log, or anywhere else they are referenced). The checksum is critical for reproducibility, since external files may be changed or moved.

5.2.3 Code tracking

The code for any software project changes and research code is no exception. Each experiment run is using a specific version of the codebase and it is important to track it for debugging, reproducibility and understanding. To address this, each experiment saves a reference to the version of code that it uses when it is run. This can be done by saving the VCS version (e.g. git commit checksum, branch and repository) along with a patch containing all uncommited changes if there are any.

This way the experiment can be reproduced by using exactly the same source code.

The code used for an experiment should be easily compared with the current codebase or the code used for another experiment. Command line utilities or simple to use interfaces for comparing code must be provided. Integrations with common VCS must be done through small external programs or isolated packages such that more integrations can be added with ease.

5.2.3.1 Compiled languages

Although interpreted languages are by far the most widely used for machine learning research, it is important to support compiled languages throughout the experiment management framework.

With compiled languages the source code does not need to co-exist with the binary being executed. Also the compiled binary does not need to be produced with the current version of the source code. Thus tracking the source code may be meaningless in this setup.

Depending on the project setup, saving the compiled binary as an experiment attachment (which can then be compared by checksum with other experiments), tracking the code, compilation commands or a combination thereof can be a sufficient workaround. An addition can be having the binary output the code version (ie. commit hash from compile time) when a special command line flag is used (e.g. `./program -version`) and logging this output in the experiment manager.

5.2.4 Minimally invasive

Each project has different requirements and uses different frameworks, programming languages and tools. Each researcher, department or company has different computer systems available but also different preferences. A system that helps rather than hinders experiment management will not impose any requirements on how experiment scripts are written or run, but rather aids the researcher with minimal required setup in the code and plenty of optional helpers.

Experiment management frameworks must not force the users to adopt any particular method of setting experiment parameters, running experiments, loading data or referencing, packaging code, saving files, logging, tuning hyperparameters or producing visualisations and other figures. The job of the experiment management tool is aiding most of the aforementioned tasks by providing methods to simplify auxiliary tasks such as fetching the correct data for a visualisation or providing methods for verifying the integrity of saved or loaded files.

5.2.5 Extendable

The system must be based on APIs, protocols and file formats rather than specific software solutions. The system must be usable wholly or partially and it must allow for third party extensions and applications to integrate seamlessly. Users must be able to modify existing parts or create new parts to fit their particular needs.

5.3 Our proposed solution

Our solution, `dbx`, is a set of tools that, when used together, provide a full platform for machine learning experiment management that fulfills all of the identified requirements. First we present the abstract ideas of what an experiment is, what the experiment log is and then we detail the architecture of the software solution.

5.3.1 Core objects

The most important objects of `dbx` are illustrated in Figure 5.6, which shows a simplified schema of the system. The following subsections present each object type in more detail.

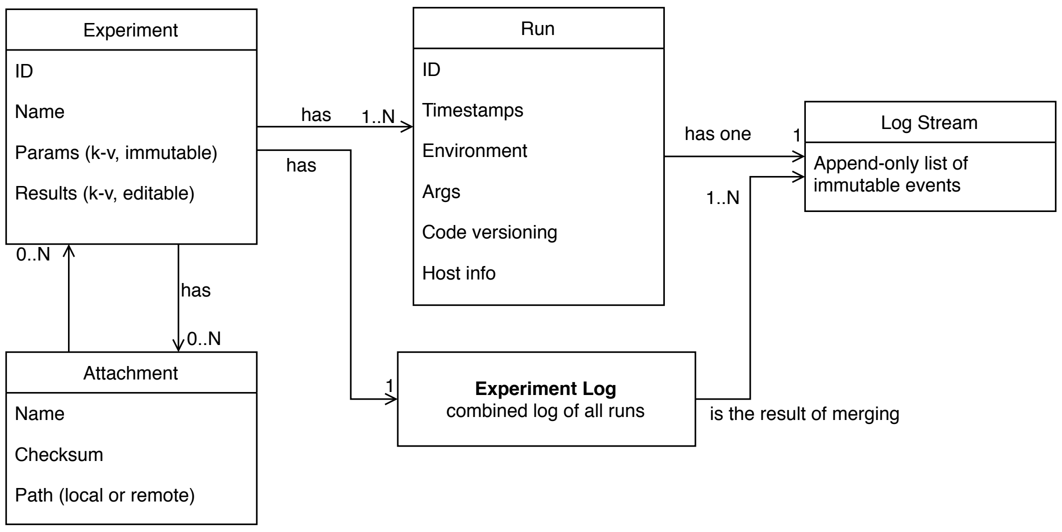


Figure 5.6: Core objects of the dbx system, simplified. k-v stands for key-value. Attachments can be referenced from logs, runs or results, not just experiments, but arrows are omitted in the figure for brevity.

5.3.1.1 Experiment

At the core of `dbx` is the **Experiment** object. It represents a machine learning experiment, and can be composed of one or more **Run**s. It provides all information about the experiment: name, kind, parameters, the log, and results. It is identified by a randomly generated unique ID.

The experiment name is a human-readable string that aims to help recognize experiments but also to organize them. The name accepts the forward slash character `/` as a separator and experiments can be looked up by partial names – similar to how files and folders work. The experiment names need not be unique, experiments are uniquely identified only by their ID. This naming scheme allows us to build an interactive experiment browser that behavies in a familiar way (like the filesystem). In local repositories, experiments are stored in eponymous folders followed by their unique id.

The experiment kind is also a human-readable string aiding organization of experiments, but only aiming to group them by a single category, typically one word. The kinds of experiments to be set are project-specific and the following examlpes show the intent of this feature: *train*, *prune*, *baseline*, *baseline-state-of-art*.

The experiment parameters are all the inputs required to reproduce this experiment. All hyper-parameters and other configurations go here. The parameters are saved as any valid JSON object and can be used in filtering experiments. Nested JSON objects are permitted, althought it is recommended for readability to keep the hierarchies small in depth. We use dot-notation to nagivate through hierarchies of parameters; `optimizer.lr` attempts to fetch `parameters['optimizer']['lr']`. As such, the dot character `.` is not accepted in parameter names. The parameters are immutable and only set when the experiment is initially created.

The results are similar to parameters in how they are stored and accessed, but they are mutable and written to as results are obtained. The results can also be used for filtering experiments, and are useful for storing summary results for quick access.

5.3.1.2 Experiment run

Each experiment can have more runs. A **Run** represents a program that ran for an experiment and saves information that is helpful to execute the **Run** again. This includes the command line arguments including the script name, code version (git commit hash) and local patch if there are uncommited changes, and environment variables.

There is one log allocated for each run.

5.3.1.3 The log

The experiment log is the concatenation of the logs of all runs that belong to that experiment. The log is an append-only list of events. Each event is immutable and can store any information (as a JSON[5] object). Log events have a name and unique ID.

The events are stored in the order they have been created and are hierarchical by name. The name hierarchy is similar to that of a filesystem and uses the character `/` as a separator. It enables grouping of events by name and not only by the order in which they have been created and serves as a basis for easily quering relevant information from the log. Events with the same name are treated as one with the most recent data taking priority, allowing runs (same or different) to override or add information to different log events after they have been initially logged. Since logs are append-only no information is lost and overriden values can still be accessed.

Hierarchical event names are not mandatory but recommended for easy filtering and querying when analysing results. Event names such as `training/epoch/0/train` and `training/epoch/0/eval` are more useful than `training-epoch` and `training-epoch-eval` for the purpose of analysing such logs.

When saved on disk, a log is a single file with the extension `jsonl`, with one JSON object per line. Each object represents a log event. The keys `event` and `id` are reserved for saving the event name and its ID, respectively.

5.3.1.4 Attachments

Attachments represent blobs that are saved outside of the results, params or the log and are referenced from them. An attachment can be referenced from many experiments (ie. an experiment that starts from a checkpoint generated by another experiment).

Attachments may or may not be managed by `dbx`. Managed attachments are the ones that are copied when experiment repositories are synchronized and can be downloaded with `dbx`.

An attachment is referenced by its path and a checksum. The path of an attachment can be a local path on the filesystem (in which case it is not automatically copied on sync), or a path inside a repository.

[5]The MongoDB Extended JSON v2 (relaxed) is used for representing values not available in JSON, such as infinity and NaN.

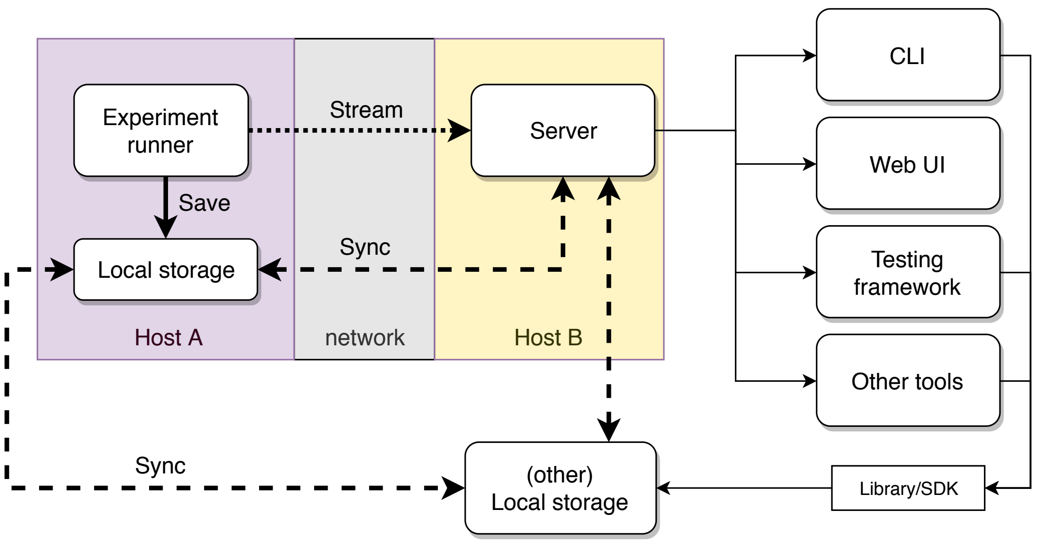


Figure 5.7: System architecture showing one experiment running that saves data both locally and streams data to a server. It shows that the experiment repository can be synchronized local-to-local, server-to-local and local-to-server and also that various tools can directly work on the local repository without a server via using the library or SDK. None of the components need to be on the same machine.

5.3.2 Experiment repository

In `dbx` all experiments are saved as part of an experiment repository that can be either local or remote. Local repositories are saved on disk as plain text files where each experiment is saved in its own folder. A remote repository is saved remotely, experiment data is sent and accessed through the network.

5.3.3 Architecture

In this section we present the architecture of our solution and how all the tools work together. A visual summary can be seen in Figure 5.7.

Experiments, their results and artefacts can be stored either locally in a local repository located directly on the filesystem, or remotely via a server. Experiment repositories can be synchronized local-to-remote, remote-to-local and local-to-local by using a command line tool. The synchronization does not create duplicate experiments even if they are stored in different folder hierarchies or there are experiment name duplicates.

In local repositories, experiment are stored each in their own folder that contains, at a minimum, a `meta.json` file with all the experiment metadata, a `log-N.jsonl` file for the log and a `run-N.json` file for each run. Additional files can be saved if required - large files or files required by extensions to the core libraries. Server-side, the experiments and logs are saved in MongoDB; each log event is a document. Large files are stored in a content-addressable way either in the local filesystem (where the server runs) or on a networked drive or using a cloud provider API.

5.3.4 Logging and saving experiments

A python[6] package is provided that allows users to easily define experiments by passing a dict of metadata, a local path and optionally a server URL.

The package takes logs and metadata from the experiment program and saves them into the local experiment repository. If a server URL is passed it also streams the logs to the server. The logging happens in a background process so it will not block the experiment code waiting for I/O.

The logger can also save information in the experiment results and update the experiment state and progress. A python interface similar to that of the tqdm package is provided for sharing progress with the server.

5.3.5 Synchronization

Synchronization can be performed using the command line tool and it is ideal for situations where a server cannot be used or cannot be accessed from the machine that runs the experiments. Other use cases are collaboration and sharing results with others but also for quickly analysing experiment files with standard unix tools.

Synchronization can be performed in any direction from local to server, server to server or local to local.

5.3.6 Analysing results

The command line tool can also list, filter and search experiments by name or metadata. It can also extract data from logs by using event names and custom conditions on any of the fields saved to events. These operations can be combined with other tools to easily extract required information, create visualisations and plots and compares experiments with each other.

The server exposes an API that is capable of performing queries on logs (filter events and select parameters), and these queries can be ran once or as a listener. When they are run once, the filtering and selecting is done once and the result is returned. When a query is registered as a listener it can be used to create live visualisations or other analysis as new data comes in from the experiment runners.

[6]Python is the most popular machine learning language. Loggers for other languages can be added later.

Experiment searching and filtering is used to get a reference to all experiments that are needed and log querying and merging is then used to select the final values required for analysis.

5.3.6.1 Searching experiments

Sets of experiments from a repository can be fetched by applying filters on any of the metadata, parameters or results that the experiment has. An example of a complex query: get all experiments with kind `prune`, name prefixed by `idea45/` that were generated in the last week, have the parameter `optimizer` set to `adam` and the result `best_val_acc` less than `0.8`.

5.3.6.2 Querying logs

The experiment log is intended to be the primary source of data for analysis. The logs can be selected and filtered by name and parameters. Querying logs is for extracting information about the experiment. The following concepts allow for a powerful and simple to use method of extracting log data.

Basic filtering. The log can be filtered by event name or any other key-value element events have. This includes checks for existance of a key and comparing a value to a given parameter (equality check, less than, greater than, regular expression match for strings). All the basic filtering can be composed using logical operations AND, OR, NOT.

Name query. The first selection of logs is via the name of the event. Event name querying can be done by path (separated by `/`), where a query can be matching event names fully or partially and can include special parameters:

- **wildcard** `*`, matches anything in a single hierarchy level (equivalent to the `[^/]` regular expression),
- **variable** `:name`, same as the wildcard but the matching value will be available as `name` in the results object),
- **path wildcard** `**`, matches none or more hierarchy levels (equivalent to any number of wildcards and attempts to match as many as possible).

Selector. Similar to the well-known `SELECT` keyword from SQL, the selector allows to specify what parameters from log events to return. It is also possible to use variables specified in the name filter, as well as variables obtained from the state.

Event merging. Events can be merged (by name query) to yield a concatenation of data (union of JSON objects). Events in a group are processed in the order they were logged. If data is overriden the values from the latest event is used. If new data is added, the result is treated as a larger object. An example use case is fetching all events with a similar name pattern where the events with the same name are grouped; this is particularly useful in the case where different runs log the same event name but with different data (e.g. training script logs the loss per epoch, another script logs training, validation and test accuracies for all saved checkpoints).

Position-based query. Each event has a position in the log. The events in the vicinity of a selected event can be used for additional information when querying data from logs. An example use case is to fetch the latest validation or test accuracy obtained and the epoch it was obtained at assuming it does not get evaluated at every epoch in a training loop. Another example in the context of our pruning work is to select all logged events before or after a given pruning step.

Compose log queries. The above operations are applied on a log, and they produce another log, which can then be queried again.

Log merging. One or more logs (not log events) can be merged by applying an aggregate function. This can be for the purpose of computing aggregates per event (e.g. computing mean and standard error accuracies per epoch or pruning step) or simply concatenating more logs.

5.3.7 Creating extensions and plugins

No research project is the same and the core features provided by any software management solution might need to be augmented. Our system is built from many independent parts (loggers, command line utilities, a server that exposes an API, a web interface and a testing framework) that can be replaced or adapted, or new parts can be added.

The server APIs can be used to create new user interfaces, testing or verification frameworks, experiment design scripts, and so on.

New loggers can be created to further automate and simplify the logging of various metrics and deeply integrate with popular machine learning frameworks. Custom importers can be created such that logs from TensorBoard or other machine learning experiment management systems can be imported to the server.

5.3.8 Implementation details

The project is split into many programs, each fulfilling a certain task and each being usable as standalone.

The server. A server written in Go that exposes a REST and a gRPC API that can perform synchronization operations, accept incoming streams and exposes the experiment metadata and logs. It can also perform queries (select and filter) on logs, both on streaming logs (by registering a query as a listener) and on logs that are finished. The API exposes methods to search experiments by name or hyperparameters and find experiments similar to another.

The server can be configured to use different storage mechanisms for large files. For experiment data excluding the logs a MongoDB database is used and Redis streams are used for logs.

Loggers. A logger is the software package that is used in the experiment code to send logs to the server or save them to disk locally (or both at the same time). A python logger library is released and available at `github.com/vladvelici/dbxlogger`, where the experiment repository format is specified such that third party implementations and integrations are possible. A logger simply saves data produced by the experiment script as well as information about the runtime environment and code version.

Command line tool. The command line tool can be used as a primary interface to the system and it can work with the server or directly with local experiment repositories. It has commands for searching for experiments by metadata or name, filtering logs by event name and selecting specific fields, synchronizing experiment repositories to and from the server or each other, and more.

Web interface or GUI. An web user interface that allows easy exploration, comparison and analysis of the experiments and results. It can plot data from experiment logs and compare experiments between each other as well as aid in saving notes for each experiment.

5.4 Project status and future work

This project is a work in progress. The Python logger is currently available at `github.com/vladvelici/dbxlogger`. Real-time capabilities are not yet implemented. Synchronization is currently done via the `rsync` utility, however it has certain limitations especially when it comes to synchronizing logs and deleted experiments. Querying and filtering experiments is not part of the released logger. The functionality described is under construction as a command line tool, originally written in Go but it will likely be ported to python or implemented at the same time as python bindings. There are plans to combine the logger and the "fetcher" (log and experiment queries, downloading checkpoints or large files) to be available via a single Python package, allowing users to not only write data to `dbx` from experiment code, but also fetch data (e.g. for resuming training or resolving assets like start checkpoints from experiment or asset IDs and not file paths).

5.4.1 Study design and verification framework

Repetitively running the same experiments with slightly different parameters is prone to human error. Mistakes like running all experiments with a slightly different parameter than intended are hard to catch but fortunately automated tests that can verify experiments after they ran are possible.

The first step is to design the *Study*, the set of experiments to be run and have a program that generates the commands required to run those experiments. This program can then be used to generate the expected metadata or certain constraints on the experiment logs and a testing framework will verify that all experiments in the Study comply. If a human error was made when designing the Study, simply fixing it and running the test framework will highlight all experiments affected by the mistake.

Such a tesing framework can be implemented to work with the system described above by using the library behind the command line tool to fetch experiment data from both with the server and local experiment repositories.

5.4.1.1 Experiment runner

A separate tool that can schedule experiments to be run and an extension for the server to keep track of *planned* experiments as opposed to only those that were already run. Studies of many experiments can be designed further in advance and they will be scheduled on computer resources when decided they need to be run.

This is not a job queue but a tool that can be used to design sets of jobs to be sent to a job queue system where they get executed.

5.4.1.2 Hyperparameter tuner framework

Hyperparameter tuning is a common feature of experiment management frameworks. A hyperparameter tuner framework can be easily built on top of our server and command line tools. Using our server, a hyper-parameter tuner tool could schedule experiments to be run and then register log listeners on them. If experiments underperform they can be stopped and replaced with other configurations; this can be used to implement different early stopping crtieria but also to wait until experiments finish and use partial results to generate new configurations.

Our framework can be the connecting platform between different implementations of hyperparameter tuning strategies.

5.4.1.3 Plot and table generator

An independet utility to generate publication-ready and reproducible plots and tables directly from data, either from `dbx` or other source. In our pruning work, most plotting scripts take many command line arguments as input, fetch data directly from experiments, and generate a plot, a LaTeX table, and a script. The script is human-readable, easy to edit, and is used to regenerate the plots, for example if data has changed or style changes must be applied.

5.5 Conclusion

We have identified a list of issues to be addressed to help researchers with managing machine learning experiment results and written requirements for software solutions. We reviewed a list of projects that tackle this problem, both in the open-source community and commercial options and highlighted that no existing solution satisfies all the requirements.

We then propose a software architecture that addresses the majority of the issues and requirements with the possiblity to be extended later via plugins and new tools to improve in the areas not thoroughly solved by the core tools. Our framework `dbx` is under construction with some parts available as open-source[7] whilst others are still being developed.

[7] At https://github.com/vladvelici/dbxlogger

Chapter 6

Conclusions and future work

In this thesis we presented a thorough exploration of pruning ResNet models block by block, using seven block selection methods and many configurations of fine-tuning schedules and amounts.

We implemented and evaluated 7 block saliency metrics: **oracle**, **activation mean**, **activation change**, **activation change plus one**, **block gradient**, **weights mean**, and **random choice**. Oracle is the best performing by a noticeable margin when no fine-tuning is used and randomly picking blocks is worst, however, when fine-tuning is enabled the difference between block selection methods diminishes. On the other hand, it is possible to design a badly performing saliency metric (we used block gradient, and selected the maxium gradients instead of minimum); this resulted in pruned models that did not train to recover the lost accuracy, with errors significantly higher than other block selection methods, including random.

How will different block selection methods for pruning perform, what can be learned from experimentally analysing them, and how important are they for the final impact on pruning? Some block selection methods perform better than others when no fine-tuning is enabled. This is easily understood by the fact that we prune greedily, and the before fine-tuning performance is simply the best local choice given the selection method. It is no surprise that *oracle* pruning performs the best, followed by block gradient magnitude, activation-based methods, and finally mean of weights. At the same time, when fine-tuning is used, even randomly pruning blocks is relatively on par with other selection methods. From this we conclude that most block selection methods *avoid bad choices* as opposed to making optimal ones. We showed that bad choices do exist by using the block gradient (max) saliency metric.

To what extent does pruning differ from training from scratch? How do pruned networks train and perform as compared to random networks? For pruning only a few blocks, pruning can perform better without fine-tuning or with very little fine-tuning. For pruning more blocks, training from scratch often outperforms pruning. We argue that pruning block by block is better for pruning a small number of blocks, or when starting with a model pre-trained elsewhere and the cost of fine-tuning is of concern. Fine-tuning alone can be much cheaper than training from scratch and give acceptable results, depending on the pruning configuration. Pruned networks are best fine-tuned with a learning rate schedule similar to that of the original training; we obtain significantly better results when starting fine-tuning with a large learning rate of 0.1 for the first 10 (out of 50) epochs (or 2 epochs in some configurations). This helps the model to *jump out* of any local minima the pruning process liekly pushed it into.

How much fine-tuning is needed during or after pruning? What is the impact of different fine-tuning schedules to the pruning process? We introduced a method of estimating the cost of fine-tuning, the epoch-parameter, which takes into account the network size at fine-tuning steps throughout pruning. We reported the cost of fine-tuning for different fine-tuning configurations, and we conclude that a small budget can be effective, but not the best.

The location and amount of fine-tuning is critical for the final pruned model performance, and we showed that using a threshold on the accuracy loss to start a fine-tuning loop is a simple yet effective way to keep the cost of fine-tuning under control and obtain efficient models. Thresholding schedules more fine-tuning loops toward the end of the pruning process, when the model is smaller and faster to train. Also, the biggest accuracy gains from training are also seen closer to the end of pruning.

The downside of thresholding is having to choose a threshold, which to be effective depends on the target network size. An alternative is to schedule a small number of epochs (we called it the *low schedule*) at every pruning step, and a longer fine-tuning loop at the end (the *high schedule*). This results in costs and performance levels on par with thresholding.

Is pruning block by block an effective way of pruning ResNet models? Pruning 27 blocks (50%) from a ResNet-110, in our best configuration, gives 6.48% test error on CIFAR-10, a 0.45% loss from the initial model. When pruning 45 blocks to obtain a similar size to that of a ResNet-20, our best method has a 1.92% loss from initial, 7.95% error. We observe that training small, standard ResNet configurations gives better results than pruning.

Block by block pruning can be effective and useful if larger pre-trained models are available from elsewhere (e.g. model zoos) and small models are required at a low cost of fine-tuning. If model complexity or size is of high concern, pruning a few blocks without fine-tuning is a viable option for preparing a large model for further pruning or compression by other means.

Pruning block by block has the advantage of being simple and easy to implement, highly effective for a small number of blocks, and it is possible to further combine it with other pruning or network compression methods.

Another observation is that pruning block by block is effective even when starting with models that have not been trained to convergence, showing promising final results for pruned networks even if the initial model was only trained for one epoch.

6.1 Future work

This section highlights key areas of improvement for this work and possible future directions.

6.1.1 Evaluation on multiple datasets

This thesis is introducing the idea of pruning ResNet networks block by block. Throughout this thesis we have only used a small dataset, CIFAR-10, for our experiments. Rerunning key experiments on ImageNet for image classification, and expanding the work to include other tasks such as object detection on the Pascal VOC as well as MS COCO datasets would help strengthen our findings.

6.1.2 Multiple base architectures

Our purning work is focused on removing blocks from ResNet networks but it can be easily extended to use other base architectures with residual connections, whilst keeping the goal of pruning large parts of a network at once.

6.1.3 Combined prunable units

Another area to be explored is combining more pruning methods. Intuitively a hybrid pruning method which first selects large prunable units and moves to smaller and smaller ones as pruning progresses might be able to yield similar results as pruning smaller units from the beginning, but in fewer steps. There is clear advantage of pruning large units (ResNet blocks) of a network at once: a big reduction in complexity in a single iteration. The limitation is that not too many blocks can be removed until the network is not performing as well as the inital model, but if as many ResNet blocks as possible are removed the pruning process can likely continue with a smaller prunable unit (convolutional filters) without significant loss of accuracy.

6.1.4 Transfer learning

Adjusting a model to a new, smaller dataset seems to be a good fit for removing parts of a neural network. Starting with a pre-trained model trained on a large dataset (such as ImageNet) and pruning it to fit smaller tasks (such as CIFAR-10, Birds-100, SVHN), is something to be explored. A key question to ask is whether fine-tuning the model to fit the new dataset should be done before, throughout, or after pruning, and whether the cost of training can be minimized. The accuracy of each of the three choices is to be compared also with training a small model from scratch on the large dataset and fine-tuning it for the small dataset.

6.1.5 Knowledge distillation

Knowledge distillation is a technique for training small (student) networks bsaed on the *learnt knowledge* of (bigger) teacher networks. We had minimal preliminary work to use knowledge distillation for fine-tuning pruned networks, but it was not sufficiently examined to be included in the thesis. We leave as future work to thoroughly explore fine-tuning pruned networks with knowledge distillation using the initial model. A small number of experiment results are available in the published dataset (Velici, 2021) in the folder `organised/distillation`.

6.1.6 Local training

Removing a ResNet block *damages* the model. This damage should be repaired in order for the model to recover the lost performance. Intuitively, the damage caused by removing a block is local to where the block is and, through fine-tuning, the network should be able to adjust the weights in the vicinity of the damage.

At each fine-tuning step, a selected part of the network is trained and the remaining is kept frozen. The intuition is that we aim to fix the *local damange* relative to where blocks have been pruned. In phased local training we gradually expand this *vicinity* until it eventually includes the full network.

Two new parameters, B_L and B_R, are introduced. They represent the number of blocks to fine-tune at the left and at the right of the pruned block, respectively. Local fine-tuning is performed for a small number of epochs at every pruning step, similar to the low training schedule in Section 4.1, and is followed by regular fine-tuning at the end of the pruning process.

Obtaining empirical data evaluating local training with different choices of B_L and B_R is left as future work.

We have done preliminary experiments for local training but has not been thoroughly explored and it was left out of the thesis. A small number of experiment results are available in the published dataset (Velici, 2021) in the folder `organised/local_training`.

6.1.7 Pruning from random

We have explored pruning from random weights or little training in Section 4.4, and the topic can be investigated further. Namely, different fine-tuning schedules can be evaluated in an attempt to reduce the total training cost and perhaps further improve model performance.

Appendix A

Extra results for pruning

A.1 Multiple fine-tuning loops

A results table (Table A.1) is included here for completeness for pruning a ResNet-110 with different fine-tuning schedules and block selection methods. "1x", "2x", "4x" denote the number of uniformly spread fine-tuning loops. t denotes that an accuracy loss threshold was used. Thresholding and "every 2" runs are continuous and others are individual prunings experiments (ie. every row with values under "4x" has four fine-tuning loops). All fine-tuning loops use up to 50 epochs. Cost is in epoch-parameters (1e6) shown based on epochs used when picking best validation error. Cost of 0 means training didn't improve accuracy. A * indicates the value is the mean from 6 runs with different initial models.

Table A.1: Results table from pruning a ResNet-110 with different fine-tuning schedules and block selection methods.

	Activation change, 2x			Activation change, 4x			Oracle, 1x			Oracle, 1x, 400 epochs			Oracle, 2x			Oracle, 4x			Oracle, every 2 *			Oracle, t=0.01 *			Oracle, t=0.1 *		
# blocks removed	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%	Err	Cost	R%
1	-	-	-	-	-	-	5.77	0.0	1.07	5.70	660.9	1.07	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	5.92	0.0	2.14	5.47	640.2	2.14	-	-	-	-	-	-	6.22	0.0	0.54	-	-	-	-	-	-
3	-	-	-	-	-	-	6.07	0.0	6.42	5.67	615.5	6.42	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	5.88	75.9	6.69	5.69	549.1	6.69	-	-	-	-	-	-	6.50	24.5	2.68	-	-	-	-	-	-
5	-	-	-	-	-	-	5.91	57.5	7.76	5.70	509.2	7.76	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	6.01	41.0	8.83	5.52	615.3	8.83	-	-	-	-	-	-	6.63	49.0	4.83	-	-	-	-	-	-
7	-	-	-	-	-	-	6.18	67.0	9.91	5.62	575.4	9.91	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	6.17	55.0	14.18	5.83	458.9	14.18	-	-	-	-	-	-	6.51	78.8	10.18	-	-	-	-	-	-
9	6.29	95.3	16.06	6.27	158.7	12.85	5.93	73.3	15.25	5.79	413.6	15.25	6.12	67.5	15.25	6.06	142.2	11.25	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	6.44	30.4	16.33	5.68	543.1	16.33	-	-	-	-	-	-	6.44	123.1	15.52	-	-	-	-	-	-
11	-	-	-	-	-	-	6.35	30.2	20.60	5.66	504.3	20.60	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	6.61	36.6	21.67	5.84	401.3	21.67	-	-	-	-	-	-	6.32	163.5	16.87	-	-	-	-	-	-
13	5.95	91.2	26.75	6.05	118.6	29.95	6.43	35.9	25.95	6.11	337.1	25.95	6.22	84.2	25.95	6.45	177.8	21.94	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	6.27	45.5	27.02	5.63	490.1	27.02	-	-	-	-	-	-	6.47	204.2	18.21	-	-	-	-	-	-
15	-	-	-	-	-	-	6.39	58.5	28.09	5.40	441.8	28.09	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	6.49	38.0	29.17	5.78	431.5	29.17	-	-	-	-	-	-	6.47	253.3	23.56	-	-	-	-	-	-
17	6.34	81.5	40.65	6.00	205.3	40.65	6.38	46.4	29.44	5.82	436.0	29.44	6.13	85.5	32.64	6.14	156.4	32.64	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	6.44	53.9	33.71	5.67	383.2	33.71	-	-	-	-	-	-	6.39	290.5	28.90	-	-	-	-	-	-
19	-	-	-	-	-	-	6.28	49.4	37.98	5.73	355.3	37.98	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	6.65	34.0	42.26	5.94	307.8	42.26	-	-	-	-	-	-	6.44	334.8	33.45	-	-	-	-	-	-
21	6.57	83.6	54.54	6.45	140.4	48.14	6.71	21.6	43.33	5.98	301.1	43.33	6.45	94.6	40.13	6.28	192.3	35.32	-	-	-	-	-	-	-	-	-
22	-	-	-	-	-	-	6.70	41.4	44.40	5.94	358.9	44.40	-	-	-	-	-	-	6.45	371.8	34.79	-	-	-	-	-	-
23	-	-	-	-	-	-	6.61	26.6	48.68	5.91	342.0	48.68	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
24	-	-	-	-	-	-	6.63	14.8	49.75	5.94	301.8	49.75	-	-	-	-	-	-	6.59	404.0	39.33	-	-	-	-	-	-
25	6.42	77.5	58.83	6.47	177.5	62.04	6.57	41.5	50.02	6.14	346.0	50.02	6.66	59.8	49.22	6.53	225.4	49.22	-	-	-	-	-	-	-	-	-
26	-	-	-	-	-	-	6.89	36.4	51.09	6.36	250.5	51.09	-	-	-	-	-	-	6.52	440.4	44.68	-	-	-	-	-	-
27	-	-	-	6.84*	177.8	57.77	6.86	41.4	52.17	6.45	279.0	52.17	-	-	-	6.73*	158.9	46.56	-	-	-	6.69	64.9	50.56	6.48	239.0	49.76
28	-	-	-	-	-	-	6.83	22.6	56.44	6.49	233.7	56.44	-	-	-	-	-	-	6.60	472.6	46.02	-	-	-	-	-	-
29	7.15	43.7	69.53	6.78	130.6	67.92	7.24	28.5	56.71	6.66	200.0	56.71	7.06	84.8	55.91	6.79	195.2	59.11	-	-	-	-	-	-	-	-	-
30	-	-	-	-	-	-	7.34	10.8	60.99	6.33	266.0	60.99	-	-	-	-	-	-	6.64	505.7	47.37	-	-	-	-	-	-
31	-	-	-	-	-	-	7.35	28.3	65.26	6.48	181.6	65.26	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
32	-	-	-	-	-	-	6.97	20.9	65.53	6.37	229.7	65.53	-	-	-	-	-	-	6.79	532.0	55.92	-	-	-	-	-	-
33	7.67	61.0	78.62	7.71	96.5	78.62	7.15	24.0	69.81	6.72	161.5	69.81	7.26	57.4	69.00	6.91	145.8	65.00	-	-	-	-	-	-	-	-	-
34	-	-	-	-	-	-	7.33	20.2	74.08	6.51	153.4	74.08	-	-	-	-	-	-	6.78	551.0	61.26	-	-	-	-	-	-
35	-	-	-	-	-	-	7.82	18.7	78.35	6.97	103.8	78.35	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
36	-	-	-	-	-	-	8.61	4.8	82.63	7.25	78.5	82.63	-	-	-	-	-	-	6.93	567.5	61.80	-	-	-	-	-	-
37	8.87	32.7	87.71	8.25	94.8	83.70	8.71	10.7	86.90	7.75	77.3	86.90	7.55	59.6	82.10	6.95	114.0	74.09	-	-	-	-	-	-	-	-	-
38	-	-	-	-	-	-	9.02	7.3	87.98	7.88	53.3	87.98	-	-	-	-	-	-	7.04	588.9	67.15	-	-	-	-	-	-
39	-	-	-	-	-	-	9.06	9.4	88.25	7.69	55.5	88.25	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
40	-	-	-	-	-	-	9.17	4.2	88.52	7.78	78.7	88.52	-	-	-	-	-	-	7.23	603.6	71.69	-	-	-	-	-	-
41	9.22	45.4	88.79	8.33	93.8	84.78	9.01	7.4	88.79	8.10	51.0	88.79	8.47	51.2	87.98	7.73	96.1	79.97	-	-	-	-	-	-	-	-	-
42	-	-	-	-	-	-	9.39	8.9	89.06	7.91	57.4	89.06	-	-	-	-	-	-	7.57	615.0	76.24	-	-	-	-	-	-
43	-	-	-	-	-	-	9.43	3.9	89.33	7.84	49.9	89.33	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
44	-	-	-	-	-	-	9.74	6.1	89.60	8.34	49.5	89.60	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
45	9.15	30.7	89.87	8.39	83.9	85.86	10.17	7.7	89.87	8.66	69.5	89.87	9.22	52.6	89.06	9.08	110.4	89.87	8.20	623.4	85.06	8.17	130.0	81.86	7.97	505.5	85.06

A.2 Phased block pruning

Results for phased pruning with a threshold of 0.01 at every training point are shown in Table A.2. They were used for Section 4.2, Figure 4.3. In the main body only the accuracies for networks with the same number of blocks as the standard baseline networks were shown.

Table A.3 shows a detailed comparison between phased pruning and simple pruning for 50 epochs of total training for each block removed, starting from a ResNet-110. Simple pruning and phased pruning perform about the same overall.

Table A.2: Average test accuracy for 5 runs of phased pruning for different values of S. Phased pruning for S epochs and fine-tuning for 50-S epochs at every training loop. Training threshold 0.01 and a schedule for all baseline points. Initial network is ResNet-110. Dataset is CIFAR-10. Only showing the points where all runs performed a training loop.

Blocks removed		**Simple**	**S=10**	**S=20**	**S=30**	**S=40**
	8	$\mathbf{94.04 \pm 0.03}$	93.98 ± 0.03	93.96 ± 0.03	93.98 ± 0.02	93.88 ± 0.03
ResNet-56	27	93.06 ± 0.05	$\mathbf{93.09 \pm 0.07}$	92.96 ± 0.05	92.95 ± 0.05	92.97 ± 0.05
ResNet-44	33	92.62 ± 0.06	$\mathbf{92.76 \pm 0.07}$	92.72 ± 0.10	92.52 ± 0.10	92.59 ± 0.10
	34	92.68 ± 0.10	92.66 ± 0.03	$\mathbf{92.71 \pm 0.10}$	92.60 ± 0.07	92.52 ± 0.13
	35	92.46 ± 0.09	$\mathbf{92.62 \pm 0.05}$	92.44 ± 0.10	92.59 ± 0.09	92.44 ± 0.08
	36	92.40 ± 0.07	$\mathbf{92.51 \pm 0.05}$	92.43 ± 0.04	92.44 ± 0.07	92.33 ± 0.09
	37	92.31 ± 0.08	92.40 ± 0.07	$\mathbf{92.41 \pm 0.12}$	92.28 ± 0.04	92.30 ± 0.06
	38	$\mathbf{92.29 \pm 0.07}$	92.28 ± 0.10	92.14 ± 0.10	92.12 ± 0.08	92.25 ± 0.09
ResNet-32	39	92.17 ± 0.10	$\mathbf{92.18 \pm 0.07}$	92.17 ± 0.05	91.92 ± 0.10	92.11 ± 0.10
	40	92.01 ± 0.11	$\mathbf{92.07 \pm 0.08}$	91.95 ± 0.18	91.91 ± 0.05	92.01 ± 0.11
	41	$\mathbf{91.89 \pm 0.10}$	91.82 ± 0.04	91.81 ± 0.09	91.62 ± 0.05	91.88 ± 0.04
	42	$\mathbf{91.71 \pm 0.07}$	91.66 ± 0.09	91.59 ± 0.08	91.66 ± 0.05	91.54 ± 0.09
	43	91.38 ± 0.10	91.41 ± 0.10	$\mathbf{91.46 \pm 0.07}$	91.32 ± 0.09	91.44 ± 0.08
	44	91.24 ± 0.06	91.10 ± 0.09	91.25 ± 0.22	91.11 ± 0.13	$\mathbf{91.34 \pm 0.08}$
ResNet-20	45	$\mathbf{90.88 \pm 0.05}$	90.70 ± 0.10	90.69 ± 0.03	90.59 ± 0.12	90.51 ± 0.08
Times highest		5	6	3	0	1

Table A.3: Phased and simple pruning starting from a ResNet-110, using CIFAR-10 dataset. Fine-tuning was performed for 50 epochs at each block removed, and for phased pruning 10 epochs for phased pruning and 40 for fine-tuning. Showing mean test accuracy from 5 runs.

Blocks removed		Simple	Phased S=10
	1	94.30 ± 0.03	94.27 ± 0.02
	2	94.28 ± 0.02	94.28 ± 0.04
	3	94.29 ± 0.04	94.18 ± 0.05
	4	94.23 ± 0.02	94.23 ± 0.04
	5	94.15 ± 0.04	94.18 ± 0.02
	6	94.12 ± 0.04	94.10 ± 0.03
	7	94.04 ± 0.05	94.04 ± 0.03
	8	93.93 ± 0.10	93.98 ± 0.02
	9	93.96 ± 0.02	93.95 ± 0.04
	10	93.93 ± 0.09	93.90 ± 0.04
	11	93.77 ± 0.08	93.83 ± 0.05
	12	93.85 ± 0.04	93.75 ± 0.06
	13	93.80 ± 0.05	93.70 ± 0.03
	14	93.74 ± 0.04	93.65 ± 0.05
	15	93.73 ± 0.06	93.65 ± 0.03
	16	93.72 ± 0.05	93.68 ± 0.04
	17	93.81 ± 0.02	93.61 ± 0.06
	18	93.68 ± 0.02	93.62 ± 0.05
	19	93.56 ± 0.05	93.50 ± 0.06
	20	93.50 ± 0.02	93.53 ± 0.07
	21	93.62 ± 0.06	93.56 ± 0.07
	22	93.53 ± 0.04	93.49 ± 0.08
	23	93.58 ± 0.03	93.33 ± 0.06
	24	93.47 ± 0.05	93.36 ± 0.03
	25	93.38 ± 0.03	93.35 ± 0.04
	26	93.43 ± 0.03	93.30 ± 0.04
ResNet-56	27	93.24 ± 0.06	93.20 ± 0.05
	28	93.15 ± 0.03	93.15 ± 0.05
	29	93.13 ± 0.11	93.02 ± 0.04
	30	93.00 ± 0.04	93.02 ± 0.03
	31	93.00 ± 0.05	92.92 ± 0.04
	32	92.86 ± 0.09	92.95 ± 0.06
ResNet-44	33	92.81 ± 0.10	92.92 ± 0.05
	34	92.58 ± 0.08	92.73 ± 0.10
	35	92.53 ± 0.10	92.62 ± 0.08
	36	92.44 ± 0.05	92.47 ± 0.08
	37	92.54 ± 0.05	92.35 ± 0.07
	38	92.40 ± 0.06	92.31 ± 0.05
ResNet-32	39	92.31 ± 0.06	92.12 ± 0.08
	40	92.07 ± 0.08	92.04 ± 0.11
	41	91.83 ± 0.07	91.75 ± 0.11
	42	91.58 ± 0.08	91.48 ± 0.07
	43	91.48 ± 0.06	91.54 ± 0.05
	44	90.92 ± 0.21	91.33 ± 0.17
ResNet-20	45	90.72 ± 0.15	90.92 ± 0.08

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Abbasi-Asl, R. and Yu, B. (2017). Structural compression of convolutional neural networks based on greedy filter pruning. *arXiv preprint arXiv:1705.07356*, 21.

Acusense Technologies, I. (2019). Datmo.

Adorf, C. S., Dodd, P. M., Ramasubramani, V., and Glotzer, S. C. (2018). Simple data and workflow management with the signac framework. *Comput. Mater. Sci.*, 146(C):220–229.

Adorf, C. S., Ramasubramani, V., Dice, B. D., Henry, M. M., Dodd, P. M., and Glotzer, S. C. (2019). glotzerlab/signac.

Allegro.ai (2019). Trains.

Anwar, S., Hwang, K., and Sung, W. (2017). Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):1–18.

Anwar, S. and Sung, W. (2016). Compact deep convolutional neural networks with coarse pruning. *arXiv preprint arXiv:1610.09639*.

Arik, S. O., Chrzanowski, M., Coates, A., Diamos, G., Gibiansky, A., Kang, Y., Li, X., Miller, J., Raiman, J., Sengupta, S., et al. (2017). Deep voice: Real-time neural text-to-speech. *arXiv preprint arXiv:1702.07825*.

Biewald, L. (2019). Weights and biases.

Blakeney, C., Yan, Y., and Zong, Z. (2020). Is pruning compression?: Investigating pruning via network layer similarity. In *The IEEE Winter Conference on Applications of Computer Vision*, pages 914–922.

Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. (2020). What is the state of neural network pruning? In *Proceedings of Machine Learning and Systems 2020*, pages 129–146.

Comet.ml (2019). Comet.ml.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Ding, X., Ding, G., Han, J., and Tang, S. (2018). Auto-balanced filter pruning for efficient convolutional neural networks. In *AAAI*, volume 3, page 7.

Dong, X., Chen, S., and Pan, S. (2017). Learning to prune deep neural networks via layer-wise optimal brain surgeon. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 4857–4867. Curran Associates, Inc.

Dong, X. and Yang, Y. (2019). Network pruning via transformable architecture search. In *Advances in Neural Information Processing Systems*, pages 760–771.

Facebook Research (2019). Visdom.

Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*.

Gale, T., Zaharia, M., Young, C., and Elsen, E. (2020). Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*.

Gomez, A. N., Zhang, I., Kamalakara, S. R., Madaan, D., Swersky, K., Gal, Y., and Hinton, G. E. (2019). Learning sparse networks using targeted dropout. *arXiv preprint arXiv:1905.13678*.

Gordon, M. A., Duh, K., and Andrews, N. (2020). Compressing bert: Studying the effects of weight pruning on transfer learning. *arXiv preprint arXiv:2002.08307*.

Gray, S., Radford, A., and Kingma, D. P. (2017). Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 3.

Haibe-Kains, B., Adam, G. A., Hosny, A., Khodakarami, F., Waldron, L., Wang, B., McIntosh, C., Goldenberg, A., Kundaje, A., Greene, C. S., et al. (2020). Transparency and reproducibility in artificial intelligence. *Nature*, 586(7829):E14–E16.

Han, S., Pool, J., Tran, J., and Dally, W. (2015). Learning both weights and connections for efficient neural network. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 28, pages 1135–1143. Curran Associates, Inc.

Hassibi, B., Stork, D. G., and Wolff, G. J. (1993). Optimal brain surgeon and general network pruning. In *Neural Networks, 1993., IEEE International Conference on*, pages 293–299. IEEE.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

He, Y., Dong, X., Kang, G., Fu, Y., Yan, C., and Yang, Y. (2020). Asymptotic soft filter pruning for deep convolutional neural networks. *IEEE Transactions on Cybernetics*, 50(8):3594–3604.

He, Y., Kang, G., Dong, X., Fu, Y., and Yang, Y. (2018). Soft filter pruning for accelerating deep convolutional neural networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2234–2240. International Joint Conferences on Artificial Intelligence Organization.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Hu, H., Peng, R., Tai, Y.-W., and Tang, C.-K. (2016). Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*.

Huang, G., Liu, Z., Weinberger, K. Q., and van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3.

Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. Q. (2016). Deep networks with stochastic depth. In Leibe, B., Matas, J., Sebe, N., and Welling, M., editors, *Computer Vision – ECCV 2016*, pages 646–661, Cham. Springer International Publishing.

Huang, Z. and Wang, N. (2018). Data-driven sparse structure selection for deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 304–320.

Inc., N. L. (2019). Neptune.ml.

Iterative, I. (2019). Dvc.

Jia, H., Xiang, X., Fan, D., Huang, M., Sun, C., Meng, Q., He, Y., and Chen, C. (2018). Droppruning for model compression. *CoRR*, abs/1812.02035.

Kendall, M. G. (1945). The treatment of ties in ranking problems. *Biometrika*, pages 239–251.

Khan, A., Sohail, A., Zahoora, U., and Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516.

Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber (2017). The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, and Pacer, M., editors, *Proceedings of the 16th Python in Science Conference*, pages 49 – 56.

Kosiorek, A. R. (2019). Forge.

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605.

Lee, N., Ajanthan, T., Gould, S., and Torr, P. H. S. (2020). A signal propagation perspective for pruning neural networks at initialization. In *International Conference on Learning Representations*.

Lee, N., Ajanthan, T., and Torr, P. H. (2018). Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*.

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

Lin, J., Rao, Y., Lu, J., and Zhou, J. (2017). Runtime neural pruning. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 2181–2191. Curran Associates, Inc.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. (2018). Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*.

Luo, J., Wu, J., and Lin, W. (2018). Thinet: A filter level pruning method for deep neural network compression. In *2017 IEEE International Conference on Computer Vision (ICCV)*, volume 00, pages 5068–5076.

Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. (2018). Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2016). Pruning convolutional neural networks for resource efficient inference.

Noy, A., Nayman, N., Ridnik, T., Zamir, N., Doveh, S., Friedman, I., Giryes, R., and Zelnik, L. (2020). Asap: Architecture search, anneal and prune. In *International Conference on Artificial Intelligence and Statistics*, pages 493–503. PMLR.

Park, J., Li, S., Wen, W., Tang, P. T. P., Li, H., Chen, Y., and Dubey, P. (2016). Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830.

Raghu, M., Gilmer, J., Yosinski, J., and Sohl-Dickstein, J. (2017). Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In *Advances in Neural Information Processing Systems*, pages 6076–6085.

Ren, M., Pokrovsky, A., Yang, B., and Urtasun, R. (2018). Sbnet: Sparse blocks network for fast inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8711–8720.

Rosenfeld, A. and Tsotsos, J. K. (2019). Intriguing properties of randomly weighted networks: Generalizing while learning next to nothing. In *2019 16th Conference on Computer and Robot Vision (CRV)*, pages 9–16. IEEE.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

Studio.ML (2019). Studio.ml.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329.

Tan, M. and Le, Q. V. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*.

TensorHub, I. (2019). Guild ai.

valohai.com (2019). Valohai.

van Amersfoort, J., Alizadeh, M., Farquhar, S., Lane, N., and Gal, Y. (2020). Single shot structured pruning before training. *arXiv preprint arXiv:2007.00389*.

Vartak, M., Subramanyam, H., Lee, W.-E., Viswanathan, S., Husnoo, S., Madden, S., and Zaharia, M. (2016). M odel db: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM.

Veit, A., Wilber, M. J., and Belongie, S. (2016). Residual networks behave like ensembles of relatively shallow networks. In *Advances in Neural Information Processing Systems*, pages 550–558.

Velici, V. (2021). Results and experiment logs for phd thesis "pruning resnet neural networks block by block". *DOI: 10.5281/zenodo.4767180*.

Velici, V. and Prügel-Bennett, A. (2021a). Object detection for crabs in top-view seabed imagery. *arXiv preprint arXiv:2105.02964*.

Velici, V. and Prügel-Bennett, A. (2021b). Rotlstm: Rotating memories in recurrent neural networks. *arXiv preprint arXiv:2105.00357*.

Wang, C., Zhang, G., and Grosse, R. (2020). Picking winning tickets before training by preserving gradient flow. *arXiv preprint arXiv:2002.07376*.

Wang, W., Zhao, S., Chen, M., Hu, J., Cai, D., and Liu, H. (2019). Dbp: Discrimination based block-level pruning for deep model acceleration. *arXiv preprint arXiv:1912.10178*.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. (2016). Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29:2074–2082.

Weston, J., Bordes, A., Chopra, S., and Mikolov, T. (2015). Towards ai-complete question answering: A set of prerequisite toy tasks. *CoRR*, abs/1502.05698.

Wu, Z., Nagarajan, T., Kumar, A., Rennie, S., Davis, L. S., Grauman, K., and Feris, R. (2018). Blockdrop: Dynamic inference paths in residual networks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8817–8826.

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2017). Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5987–5995. IEEE.

Yamada, Y., Iwamura, M., Akiba, T., and Kise, K. (2019). Shakedrop regularization for deep residual learning. *IEEE Access*, 7:186126–186136.

Yang, T.-J., Chen, Y.-H., and Sze, V. (2017). Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5687–5695.

Zhang, X., Zhou, X., Lin, M., and Sun, J. (2018). Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856.

Zhou, H., Alvarez, J. M., and Porikli, F. (2016). Less is more: Towards compact cnns. In *European Conference on Computer Vision*, pages 662–677. Springer.

Zhou, H., Lan, J., Liu, R., and Yosinski, J. (2019). Deconstructing lottery tickets: Zeros, signs, and the supermask. In *Advances in Neural Information Processing Systems*, pages 3597–3607.

Zhu, M., Zhang, T., Gu, Z., and Xie, Y. (2019). Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 359–371, New York, NY, USA. Association for Computing Machinery.