On Pruning Methods for Neural Networks

(O Metodach Redukcji Sieci Neuronowych)

Szymon Mikler

Praca magisterska

Promotor: dr hab. Jan Chorowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

27 września 2021

Abstract

Neural networks are becoming larger as our compute capabilities grow. As a side effect, most of them are largely overparametrized. Pruning is a way to reduce this problem. One-shot pruning methods are usually less effective than iterative methods, but the latter have bigger compute requirements. We analyze and organize the most influential works in the topic of iterative pruning. Recently, [Renda et al., 2020] described a simple iterative method that matches and in some cases improves state-of-the-art in neural network pruning. However, [Mikler, 2021] found a flaw in it that causes a severe accuracy degradation in large, highly sparse networks. We examine this flaw to design an iterative pruning method that works just as well or better in every test case. We propose Generalized Iterative Magnitude Pruning (GIMP) as a framework that describes all other methods. Its special case: stable Generalized Iterative Magnitude Pruning (sGIMP) is – to our knowledge – the best performing method for both small and larger architectures.

We explore compute efficient methods, creating our own method that is based on binary masks training: Training Optimal Masks. It is inspired by observations of better performing iterative pruning methods. Our method works just as well or better than competing one-shot pruning methods while having similar compute requirements. Besides, we research sparse masks modifications. Our experiments show that iterative methods create masks that are closely bounded with the network's parameters. Any modifications to either masks or parameters cause an accuracy decrease. We discover an unexpected influence of batch normalization hyperparameters on the iterative pruning performance that only occurs in extremely sparse networks.

Streszczenie

Wraz ze wzrostem możliwości obliczeniowych, sieci neuronowe stają się coraz większe. Modele stają się przeparametryzowane. Redukcja ich rozmiaru poprzez usuwanie połączeń jest jedną z metod zapobiegania temu problemowi. Metody redukujące sieci w jednym kroku są zwykle mniej skuteczne niż metody iteracyjne. Jednak te drugie są bardziej wymagające obliczeniowo. Analizujemy i porządkujemy najbardziej wpływowe prace z dziedziny redukcji iteracyjnej. Niedawna praca [Renda et al., 2020] opisuje prostą iteracyjną metodę która dorównuje lub poprawia wyniki istniejących metod redukcji. Z drugiej strony [Mikler, 2021] zauważa, że ta metoda zawodzi dla większych i mocniej skompresowanych sieci neuronowych niż te oryginalnie przetestowane. W tej pracy potwierdzamy te spostrzeżenia i projektujemy generalizację istniejących metod. Generalizację tę nazywamy *Generalized Iterative Magnitude Pruning*. Jej stabilny wariant *stable Generalized Iterative Magnitude Pruning* (sGIMP) jest oryginalną metodą, którą proponujemy w tej pracy. Osiąga ona równie dobrą, a w niektórych przypadkach lepszą dokładność co dotychczas opublikowane, iteracyjne metody.

Badamy również mniej wymagające obliczeniowo metody. Tworzymy naszą własną metodę bazującą na uczeniu binarnych masek. Jest ona zainspirowana obserwacjami bardziej skutecznych, iteracyjnych metod. Nasza metoda działa równie dobrze lub lepiej niż popularne metody redukcji sieci o podobnych wymaganiach obliczeniowych. Sprawdzamy również jaki wpływ na wyniki sieci neuronowej mają modyfikacje znalezionych masek. Odkrywamy, że maski stworzone przez iteracyjne metody są mocno zależne od konkretnych wartości parametrów. Zarówno po modyfikacji masek i parametrów obserwujemy znaczące różnice w dokładności sieci neuronowych. Zauważamy niespodziewany wpływ parametrów warstwy “batch normalization” na jakość iteracyjnej redukcji sieci neuronowych.

Contents

Chapter 1

Introduction

1.1 Motivation

Neural networks are generally very large. Largest neural language models have as many as hundreds of billion parameters. Research on neural networks [Kaplan et al., 2020] indicates that adding more parameters will improve their capabilities even further. In Table 1.1 we compare the most influential neural network architectures from the domain of computer vision. As one can see, the number of parameters is only loosely correlated with the accuracy. Sometimes large networks are worse than their smaller counterparts, e.g., VGG19 and VGG13. Adding residual blocks improves the accuracy, even if the number of parameters is smaller. Therefore, architectural changes, like residual connections, are one way to improve accuracy. Increasing the number of parameters is another way. The latter, however, is used too often as an easy way to improve the network's accuracy. Nowadays, this leads to terrible parameter efficiency and very large networks. Often so large, that they cannot be used on an average personal computer, even for inference. Computation must be then moved to the cloud.

In reality, parameter efficiency is important for most use cases. Better parameter efficiency means faster training and inference while keeping good accuracy levels. Inference time is especially important on mobile devices with limited compute capabilities. However, even outside mobile applications, it makes a substantial difference whether a neural network can be used on a personal machine or exclusively on specialized hardware. Another aspect, memory footprint, is sometimes just as important as inference speed. The number of parameters in the network corresponds directly to the memory footprint, since each parameter needs to be stored. However, it corresponds indirectly to inference and training time as well. Network efficiency can be described as accuracy per parameter. In this thesis, we are compressing large neural networks and increasing their efficiency. We are often able to compress large networks without losing the original accuracy.

Model	Number of Parameters	CIFAR-10 Accuracy
ResNet-20	272 282	92.5%
ResNet-56	855 578	93.7%
ResNet-110	1 730 522	94.3%
Wide ResNet 16-4	2 752 570	95.0%
Wide ResNet 16-8	10 961 370	95.7%
Wide ResNet 28-10	36 479 194	96.2%
VGG13 (with BN)	9 413 066	94.4%
VGG16 (with BN)	14 724 042	94.2%
VGG19 (with BN)	20 035 018	94.0%
EfficientNet B-7*	∼64 000 000	98.9%
EfficientNetV2-L*	∼121 000 000	99.1%

Table 1.1: Comparison of popular network architectures for computer vision from [He et al., 2016a, Zagoruyko and Komodakis, 2016, Simonyan and Zisserman, 2014]. All of them were originally designed for ImageNet dataset [Russakovsky et al., 2015]. Comparison here show their smaller counterparts designed to work with CIFAR-10 dataset [Krizhevsky et al., 2009]. Excluding EfficientNet, results are from our own, reproduced networks. *EfficientNet is trained using state-of-the-art data augmentation and training tricks (e.g., stochastic depth [Huang et al., 2016]) that could be implemented in other networks as well and it would improve their accuracy.

The topic of neural network compression is important for the environment as well. As [Anthony et al., 2020] and [Patterson et al., 2021] report, with the rapid growth of deep learning, its carbon footprint was growing as well. Between 2012 and 2018, it grew 300000-fold. They suggest that with the current exponential growth, deep learning might become a significant contributor to climate change. Open-source tool shared in [Anthony et al., 2020] allowed to estimate that training of one of the largest neural networks: GPT-3 from [Brown et al., 2020], just once, required as much energy as 126 homes in Denmark per year.

While we do not directly calculate the parameter efficiency, we compare the number of parameters and the accuracy of both dense and sparse networks in Figure 1.1. With this, one can estimate their parameter efficiency. For VGG networks, there is no accuracy difference between dense variants: VGG13, VGG16 and VGG19. We can see that there is no difference in accuracy between their sparse variants as well. Sparse VGG19 works as well as dense VGG16 or VGG13 with the same number of parameters. It is more interesting in the case of residual architectures. Sparse ResNet-56 from [He et al., 2016b] is clearly better than dense ResNet-20 with the same number of parameters. The same is true between Wide ResNet 16-8 and Wide

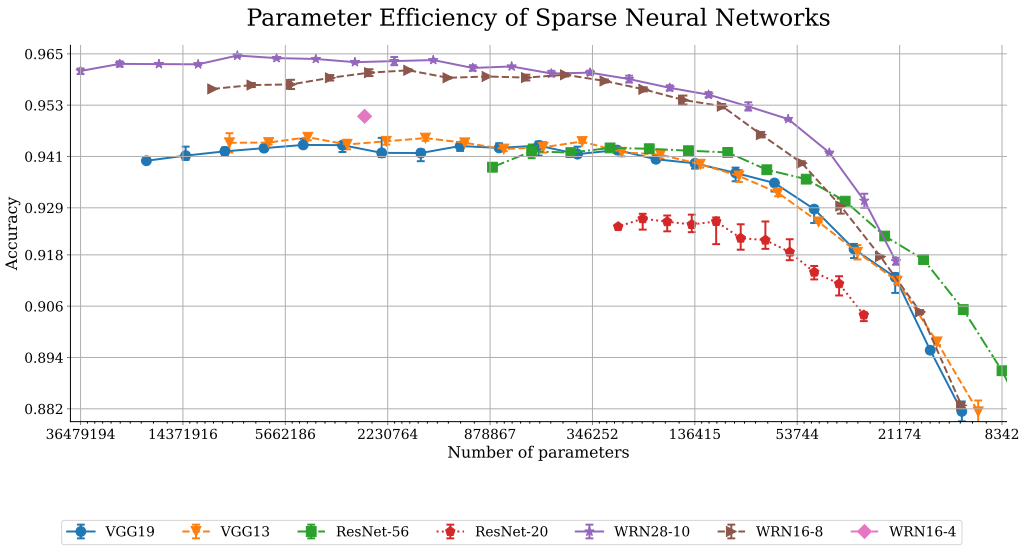

Figure 1.1: Comparison of multiple neural network architectures, in both dense and sparse variants. Each dot is a fully trained trained neural network. Connected dots means the same neural network were pruned to different sparsity levels with an iterative pruning algorithm. We pruned each network with the best – to our knowledge – method. This can be seen as multi-objective optimization. We strive for the smallest number of parameters with the highest possible accuracy. Upper right corner of the plot is the best possible solution and bottom left is the worst. Pruned networks usually have higher parameter efficiency than their smaller, dense counterparts.

ResNet 16-4 from [Zagoruyko and Komodakis, 2016]. We see that it is often better to have a larger network pruned than to use smaller versions of architectures altogether. The exceptions are architectures like VGG, when there is no difference between dense variants in the first place.

There is a branch of deep learning research that focuses on mobile architectures that do not involve pruning [Howard et al., 2017, Tan and Le, 2019]. Some of those architectures offer really high accuracy with a relatively small number of parameters. Most recent architectures, like EfficentNet, are better in terms of parameter efficiency than those presented in Figure 1.1. However, their secret lies not in the network topology, but rather the sophisticated training process. The exact architecture is but an addition. On the other hand, neural network pruning allows for an automatic discovery of an efficient sparse topology. Advanced training methods might be used to further improve sparse networks' performance.

1.2 Contributions

It is difficult to clearly compare pruning methods based on the existing literature. As [Blalock et al., 2020] notices, the results reported in the literature come from multiple different datasets and architectures. It is often the case that two different papers do not have a common combination of a dataset and a neural network architecture. This makes direct comparison impossible. Even when it is not the case, the implementation of neural network and pruning algorithm is rarely the same between papers. This makes any comparison misleading. For this reason, there are no methods that have been shown to outperform all other methods. There is no visible consensus as to which method is leading and should be included for comparison in other pruning papers. In this thesis, we organize the most influential papers from the domain and reproduce their results. We offer a fair comparison and derive conclusions as to which methods are the most effective. In our comparison, we test many different neural network architectures on a single, very popular image recognition dataset – CIFAR10 – with our own implementation of pruning methods. We implement these carefully, so that they exactly match their authors' description.

Moreover, we inspect problems noticed by [Mikler, 2021] regarding one of the papers on iterative magnitude pruning by [Renda et al., 2020]. We offer a solution to these problems in Section 3.2. We describe a generalized algorithm (GIMP) that covers all of the existing methods. Its special case (sGIMP) is our original iterative pruning method which we propose. We examine this new method and compare it to others. This method, to our knowledge, achieves the highest accuracy in the existing literature for large computer vision architectures such as Wide Residual Networks. In each tested scenario, it is just as good or better than existing methods.

Currently, there is a large gap between one-shot and iterative pruning. Sparse networks that iterative pruning produces are much more accurate. However, we recognize the benefits of one-shot pruning. Although it is less accurate, it is much quicker to obtain sparse neural networks using it. In Section 3.3.1, we design our own method that has similar compute requirements as one-shot pruning. It was inspired by our observations of more accurate iterative pruning algorithms. Our method, based on binary masks training, achieves similar performance as competing methods that have similar compute requirements. For VGG19 network, it is as good or better than other methods.

In Section 3.3.2, we perform experiments on modifying the topology of sparse networks. We observe how tightly bounded the sparse masks are to the exact parameters they were found on. In Section 3.3.3 we notice a previously unobserved impact that batch normalization hyperparameters have on the iterative pruning. This suggest that low sparsity iterative pruning is highly sensitive to disturbances.

Chapter 2

Background

2.1 Neural Networks

Neural networks are mathematical models conceptually inspired by biological brains. Similarly, they contain neurons. Artificial neurons contain inputs and outputs. Both inputs and outputs can be represented in the form of vectors. Inputs can be either external data, or they can be outputs of other neurons. Neural networks are parametrized and usually differentiable almost everywhere. Besides neurons, we distinguish layers, which can be seen as groups of neurons. Layers define what operation will be performed on the inputs. There are many different layers, some of them are typical to a specific kind of data, e.g., visual such as pictures. In the simplest case, the operation is a linear transformation – weighted dot product of the neuron's inputs. This is depicted in Figure 2.1.

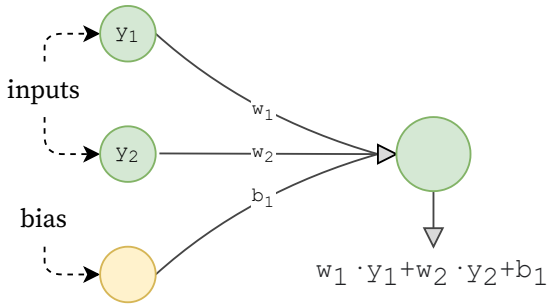

Figure 2.1: Neuron in a simple, linear neural network performs a weighted dot product on its inputs. The result of this operation is passed further as input to other neurons or as the final output.

Neural networks can be seen as directed weighted graphs. There are two factors on which two neural networks can differ: topology and parameters. Topology is determining the general structure of a network: how neurons are connected and how the data flows through the network. Following graph terminology, topology are

nodes and directed edges, while parameters are the graph's weights. In a feedforward network, external inputs are processed by the first layer, then the subsequent layers use outputs of the previous layers as their inputs until the final output is returned. The basic feedforward topology, like in Figure 2.2, is a fully connected, dense network that might consist of multiple linear layers. In such linear layers, each neuron is connected to every neuron from the previous layer and every neuron from the next layer. This means the number of parameters can be precisely calculated if we know the number of input and output neurons. If a layer has N input and M output neurons, there will be $N \cdot M$ connections between them. Often, we include a single bias parameter for each of the M output neurons. This makes the total number of parameters $N \cdot M + M = (N + 1) \cdot M$. An example of a linear neural network with 3 layers, 2 inputs, and 1 output is presented in Figure 2.2. Among the network's parameters, there are weights and biases, the latter being presented as connections from orange neurons with no inputs.

Figure 2.2 shows a really basic neural network and contains only the simplest linear layers. In reality, however, there are other layers specializing in certain domains, e.g., natural language processing or computer vision. In this work, we will be focusing on mainly on computer vision. This means the building block of networks used in this work will be convolutional layers, which will be described in more detail later in Section 2.2.

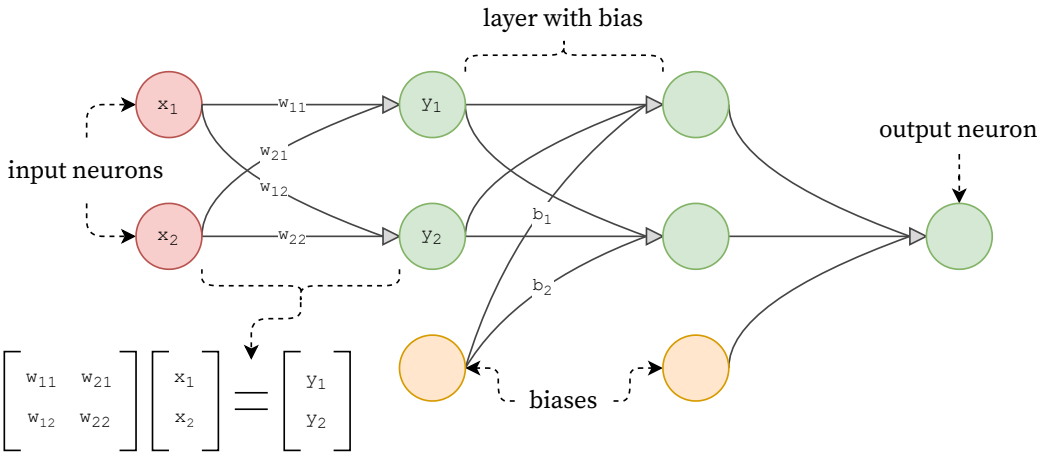

Figure 2.2: Diagram depicting a simple, linear neural network. The neurons are grouped in four layers. Operation performed in the first one is shown in a matrix form. Only some of the parameters are named, e.g., w_{11} or b_2, but every connection (edge) has their own parameter. In the example, there are 13 of them. Layers may or may not include bias.

After the neuron performs its operation, often there is an activation. In example from Figure 2.1, there is a neuron that has three inputs, one of which is a bias, and it outputs:

$$\texttt{output} = w_1 \cdot y_1 + w_2 \cdot y_2 + b_1$$

This operation is a linear transformation (with nonzero constant) of inputs $[y_1, y_2]$. Usually, a linear transformation is not enough for a neural network to express complex functions. For example, it can be proved that it is impossible to express XOR operator using a linear neural network. To give the network more expressive power, the activation is performed. Classically, a sigmoid or tanh functions were used for this purpose [Nwankpa et al., 2018]. Today it is common to use Rectified Linear Units (or ReLU) instead [Nair and Hinton, 2010].

$$\texttt{ReLU}(x) = x \ \texttt{if} \ x > 0 \ \texttt{else} \ 0$$

This non-linearity allows the network to represent XOR operator. For all networks reproduced in this paper, we are using ReLU activation. In case of the example from Figure 2.1, if we use ReLU activation, the output of the neuron will change to: $\texttt{output} = \texttt{ReLU}(w_1 \cdot y_1 + w_2 \cdot y_2 + b_1)$.

Neural networks are differentiable. This allows us to calculate the gradient of a loss function with respect to the network's parameters. The exact choice of the loss function depends on the task. This fact is used for backpropagation [Rumelhart et al., 1985]. In supervised learning, we use this process to iteratively update the network's parameters based on the external data and (often manually) annotated targets. For neural networks, this optimization procedure, when repeated, often leads to convergence. Stochastic Gradient Optimization is a variant of this process, where gradients are calculated based on a small subset of the available data – a batch. We use SGD optimization throughout this thesis.

2.2 Convolutional Neural Networks

In computer vision, the inputs are usually images represented as matrices. Those are often three matrices corresponding to RGB channels. If those were represented as vectors, the previously described linear networks could be used. However, it turns out that we can gain more by keeping their original spatial shape of the images. As [LeCun, 1988a], we utilize it by using convolutional neural networks.

In convolutional networks, the inputs and the intermediate outputs are matrices. Strictly speaking, those are from $[0, 1]^{C \times H \times W}$. For external inputs, it is often the case that $C = 3$, as the images are represented in three RGB channels. H and W correspond to the height and width of the image. In Figure 2.3 there is a data flow example of a convolutional neural network. All H, W and C might change in convolutional layers.

The convolution operation itself needs some explanation as well. In the simplest case, the convolution can be represented by a window sliding through a 2D matrix. Sliding window mechanism is shown in Figure 2.4. The depicted example has no padding and stride equal to 1. This will not always be the case with other convolutional networks.

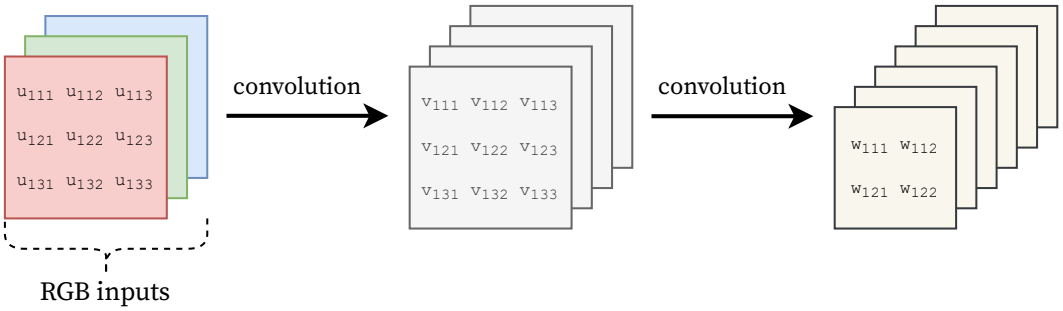

Figure 2.3: Example diagram of a data flow in a convolutional neural network. After the first convolution, only the number of channels changes. After the second convolution, all height, width and number of channels change.

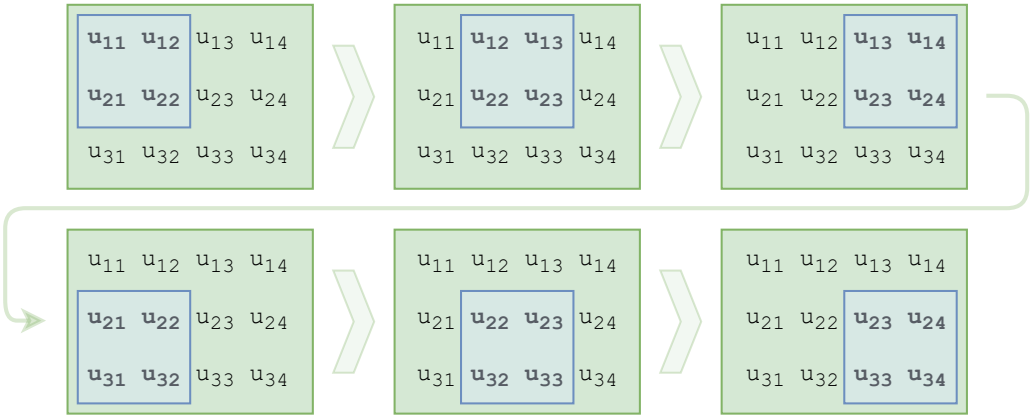

Figure 2.4: Sliding window mechanism in convolution. This particular example uses stride equal to 1 and no padding. Sliding window ends when it reaches the bottom right corner of the input.

Padding is often used to keep the shape of the output constant. Convolution in the Figure 2.4 takes a 2D matrix as an input and returns a 2D matrix as well. However, the inputs and outputs will be of different shapes, unless padding is used.

In convolutional neural networks, convolutional filters have their own learnable parameters. In each step of the sliding window, a dot product between the filter's parameters and input values is calculated. This is depicted in Figure 2.5. The resulting number will be placed in the output matrix. Unless the convolution window is of size 1×1, resulting output matrix will be smaller than the input. One can use zero padding to keep the spatial shape constant. Sometimes one might want to avoid adding meaningless values such as zero. Then reflection padding can be used.

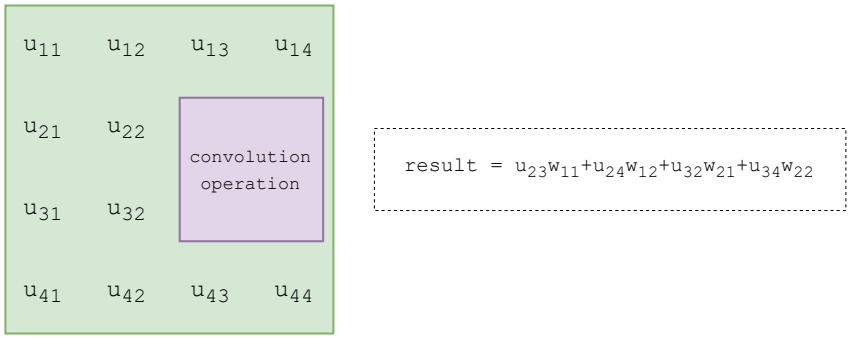

Figure 2.5: In each step of the sliding window, convolutional filter performs a dot product between the inputs and its parameters. Filter in this particular example is parametrized by a vector $[w_{11}, w_{12}, w_{21}, w_{22}]$. Input to the window is changing in each step of the sliding window. During depicted step, input is a vector $[u_{23}, u_{24}, u_{33}, u_{34}]$. Both inputs and parameters can be represented either as vectors or small 2D matrices. We use both representations interchangeably.

2.3 Batch Normalization

Batch normalization [Ioffe and Szegedy, 2015] is a technique that accelerates, stabilizes, and generally improves the performance of neural networks. As described in [Li et al., 2018], batch normalization also has a regularizing effect that is sometimes strong enough to replace entirely another regularization technique – dropout.

Batch normalization is often used in SGD optimization where data is split in batches. It works differently during training and during validation. During training: batch is being standardized using its own mean and standard deviation. During validation: batch is being standardized using moving averages of the mean and standard deviation that have been observed during training. Moving average of both mean and standard deviation is being calculated using the following update rule:

$$x' = \mu x + (1 - \mu)y$$

Where x' is the new moving average, x is the old moving average, y is the newly observed value (either mean or standard deviation) and μ is the momentum hyperparameter. Further discussion and experiments regarding hyperparameter choice are available in Section 3.3.3.

In batch normalization layers, standardization is not the only thing that happens. After the transformations described above and getting a batch with standardized distribution, the distribution is altered again. The batch is multiplied by tensor γ (initialized as 1) and shifted by tensor β (initialized as 0). Both γ and β are learned

during backpropagation. Since shifting the batch by a learnable tensor β is essentially the same as the bias, bias is no longer used by layers if batch normalization follows them.

Summing up, batch normalization returns a modified batch:

$$\gamma \cdot \frac{x - \texttt{mean}(x)}{\sqrt{\texttt{var}(x) + \epsilon}} + \beta$$

Where ϵ is another hyperparameter whose effect is discussed later in Section 3.3.3.

2.4 Pruning

Neural networks are parametrized. Parameters can be ordered and placed in a vector. Let us call this vector $\mathbf{w} \in \mathbb{R}^n$. To prune the network, we have to calculate the salience (importance) of each parameter. Salience can be usually represented in the form of a function of the network's weights. Some methods, as exceptions, use more inputs besides weights to calculate the salience. Examples are: momentum in Sparse Momentum [Dettmers and Zettlemoyer, 2019] or training data itself in SNIP [Lee et al., 2018]. A vector of saliences is $\mathbf{s} \in \mathbb{R}^n$. Pruning can be seen as modifying (zeroing) weights $\mathbf{w}$. This creates a weight vector transformed in the following way: $\mathbf{w}_i \leftarrow \mathbf{w}_i \cdot \mathbb{1}_{[\mathbf{s}_i > t]}$. Threshold t is calculated in such a way, that only a desired number of elements of $\mathbf{s}$ are larger that t. This will be equal to the number of remaining parameters in the sparse network, so their remaining percentage will be the model's density. One might decide that t is a constant for the whole model, which [Blalock et al., 2020] calls automatic pruning. A different option is to set a desired t for each layer, which is then called predefined pruning. More details on this classification are in Section 2.4.3. The most common way (suggested in *Optimal Brain Damage* [LeCun et al., 1990]) to calculate saliences is what is called magnitude pruning. Saliences are then a function of the model's parameters: $\mathbf{s} \leftarrow |\mathbf{w}|$. This topic is described in more detail in Section 2.4.2.

The history of neural network pruning is long. *Optimal Brain Damage* [LeCun et al., 1990] and *Optimal Brain Surgeon* [Hassibi et al., 1993] were one of the first works on the topic. They were both considering pruning based on Hessian matrix. Magnitude pruning described in [Janowsky, 1989] was reinvented multiple times later. It was reborn in the past years due to the rapid growth of neural networks popularity. The rapid growth of the number of parameters in neural networks was probably another contributing factor.

2.4.1 Structured and Unstructured Pruning

One of the first papers about neural network pruning [LeCun et al., 1990] focused solely on unstructured pruning. In this paper, we focus on unstructured pruning as well. However, current hardware limitations do not allow to take full advantage of unstructured pruning. Structured pruning is worked on as a workaround to this problem. In structured pruning, we remove the basic structure of the network with everything that connects to it. In the case of dense linear neural networks, these structures are neurons and their connections – neuron's inputs and outputs. However, depending on the network's type, this can be something else. In every case, it should be a minimal unit such that the remaining neural network can be represented as a smaller, but dense neural network. In the case of structured pruning of convolutional neural networks, we remove the whole channels and their corresponding parameters in convolutional filters. Structured pruning is a special case of unstructured pruning, but we will not be focusing on it in this thesis.

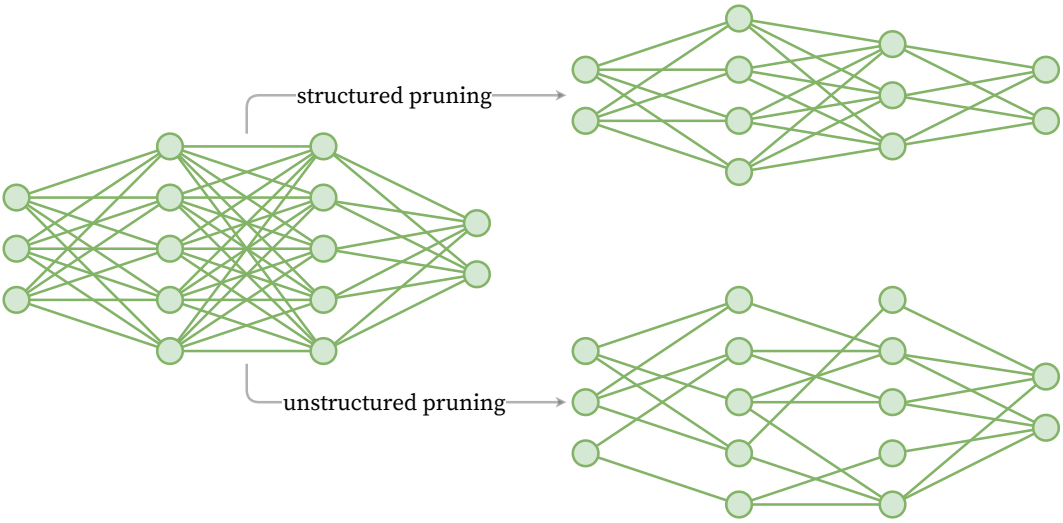

Figure 2.6: Examples of pruned linear neural network topology with three layers. Density in both structured and unstructured examples is 52% with 26 out of 50 connections left. In structured pruning whole neurons with corresponding inputs and outputs were removed. This is why the final network still is a dense neural network.

2.4.2 Pruning Criterion

In each pruning algorithm, both in structured and unstructured pruning, one has to decide which connections to prune. [Janowsky, 1989] and [Frankle and Carbin, 2019] both prune connections with the smallest magnitude. In [Lee et al., 2018] the paper's novelty is a new pruning criterion used before the training, which was supposed to describe the connection's salience better than just their magnitude. Specifically, they

set salience $\mathbf{s} \in \mathbb{R}^n$:

$$\mathbf{s} = |\mathbf{w} \cdot \mathbf{g}|$$

where $\mathbf{g} \in \mathbb{R}$ are the network's gradients. Mathematically, this preserves the connections that have a large impact on the loss value and removes the ones that do not influence it.

2.4.3 Pruning Methods

Pruning is a process of removing parameters from the network. This can be done in many ways. To avoid unnecessary assumptions, we will go through different classifications described in the literature. One can classify pruning methods in a few different ways. One of them is the distinction between:

1. one-shot pruning in Figure 2.7
2. iterative pruning in Figure 2.8

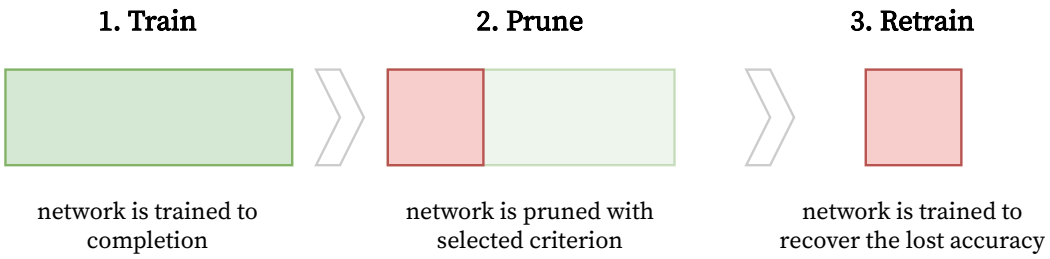

Figure 2.7: One-shot pruning.

There are more variations possible, e.g., [Lee et al., 2018] modifies one-shot pruning by skipping the first step – training – and prunes right after the initialization. On the first look, this is disadvantageous, but it makes the whole procedure quicker. Iterative pruning framework generally requires much more computation, since the networks have to be retrained multiple times. The exact number of iterations depends on the use case and the desired density. In a special case, when using only one iteration, it is one-shot pruning. Therefore, the iterative pruning framework can describe most of the other methods. Each additional iteration costs as much time as it is needed to retrain the network. Higher compute requirements suggest that multi-iteration iterative methods are more effective in terms of final accuracy than one-shot methods. This observation is confirmed by [Frankle et al., 2020a].

There are methods that fail to land in any of those categories. One of the examples is Sparse Momentum [Dettmers and Zettlemoyer, 2019], where the authors dynamically modify the connection pattern during the training. Based on momentum and parameter magnitude, they update the pruning masks multiple times during just one training cycle.

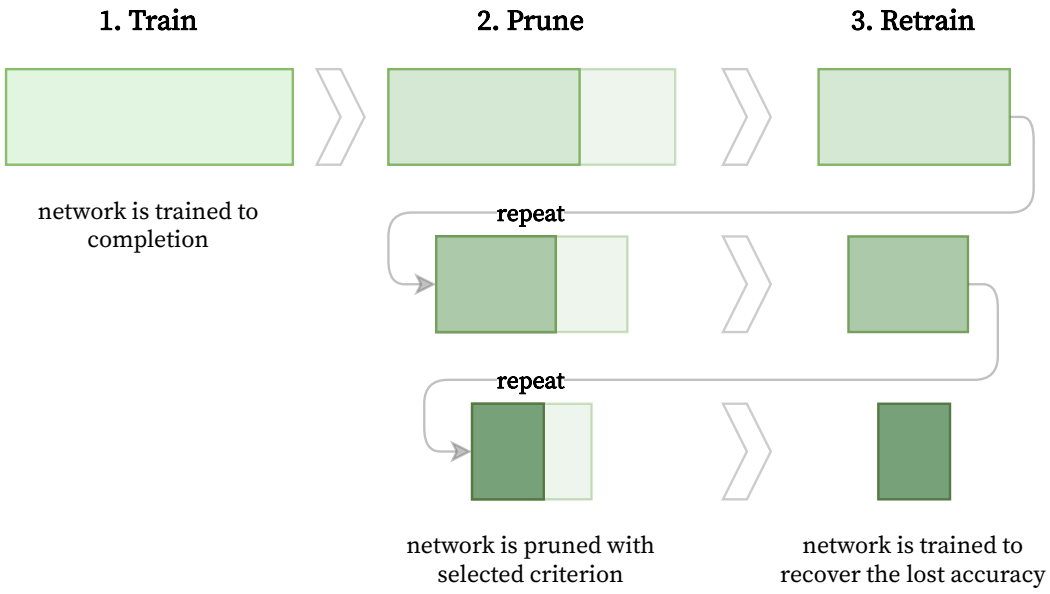

Figure 2.8: Iterative pruning.

In case of one-shot pruning, we prune the network to the desired sparsity all at once. In iterative pruning, we usually select a pruning step size and a desired sparsity. We repeat pruning and retraining until the desired sparsity is achieved. In each pruning step, we prune a fixed percentage of parameters remaining in the network. In [Frankle and Carbin, 2019, Renda et al., 2020] a pruning step of 20% is used. This means the density starts at 100% and is multiplied by 0.8 before each pruning iteration. After two iterations, such network would have 64% density or – equivalently – 36% sparsity.

Another differentiation (described by [Liu et al., 2019b]) is shown in Figure 2.9:

1. automatic pruning
2. predefined pruning

Automatic pruning gives more flexibility to the pruning algorithm. It is rarely the case in automatic pruning that layers are pruned equally, as they usually are in predefined pruning. In most cases, predefined pruning is worse than automatic pruning in terms of the final network's accuracy [Liu et al., 2019b]. Predefined pruning does not necessarily need to sparsify the layers equally. One might plan out specific layers' sparsities to achieve an optimal balance in the network. This is still predefined pruning. In this work, we prefer pruning variants that give more flexibility to the pruning algorithm. For this reason, we select automatic pruning over predefined pruning.

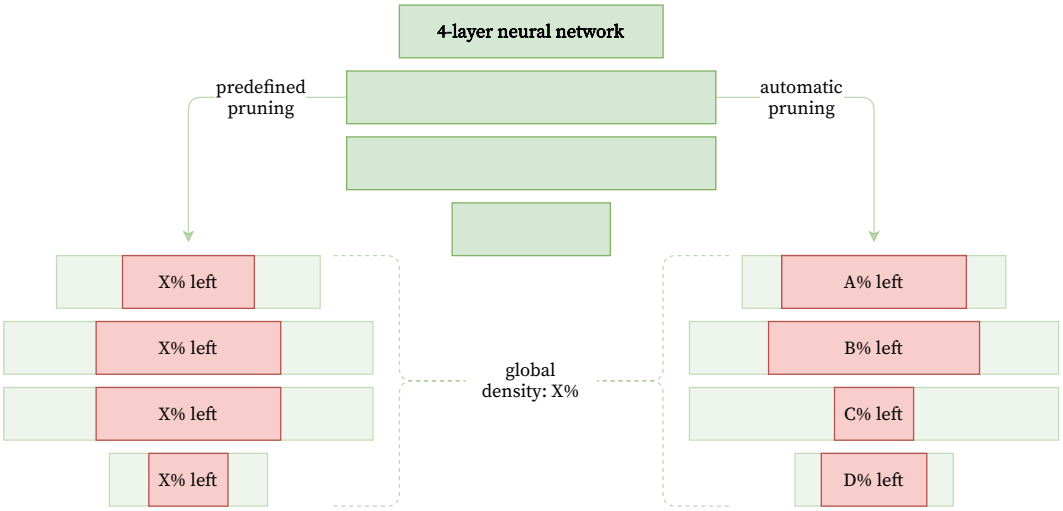

Figure 2.9: On the diagram, there is a feedforward network with four layers. It is pruned in two different ways. Predefined pruning is a process that restricts each layer sparsity to a desired, specifically defined (by an expert) level. Automatic pruning does not have any restrictions besides the global density. This means in automatic pruning, layer might end up with 0 parameters. Unless such layer was residual, this would render the network useless. However, such mistakes do not happen often and automatic pruning usually finds the optimal sparsity proportions.

2.5 Datasets

2.5.1 CIFAR-10

CIFAR-10 [Krizhevsky et al., 2009] is a standard benchmark for computer vision tasks. It contains natural images in low resolution. In consists of 50000 training and 10000 test images. They are split into 10 different classes. Each image is 32×32 pixels and has three color channels. When such image is flattened to a vector, its size is 3072. The task performed on this dataset is image classification. The network's is given an image and one of the ten possible labels is expected as an output.

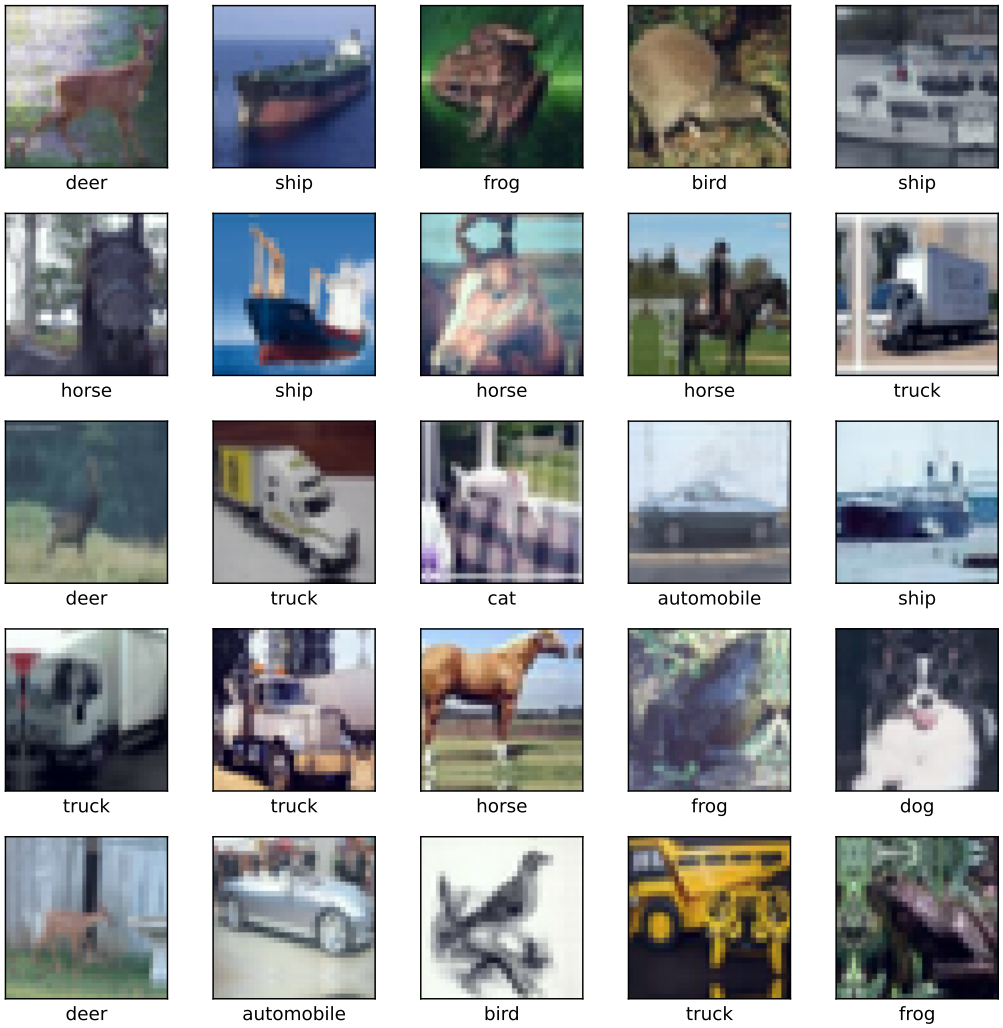

Figure 2.10: Examples of images and their labels from CIFAR-10. Low resolution images are sometimes difficult to recognize even for humans.

Deep learning pipelines are often heavy in preprocessing [Cubuk et al., 2019]. In this work, we use only standard preprocessing, as in [Frankle and Carbin, 2019]. This means we perform light data augmentation. Specifically:

1. Randomly flip the image horizontally,
2. Add 4 pixel padding using image's reflection,
3. Crop to size 32×32,
4. Standardize using training dataset statistics.

After that, the image is processed by the network. Network's output is compared with the ground-truth label and the categorical crossentropy loss is calculated.

Chapter 3

Results

3.1 Iterative Pruning

If we aim for sparse networks with the highest possible classification accuracy, the most proper algorithms to achieve it are iterative pruning algorithms. In those, pruning is repeated many times throughout the training. After each pruning, the accuracy might drop. To recover it, retraining is required. Since pruning and retraining are repeated multiple times, these algorithms are more computationally expensive than one-shot methods.

In the majority of the literature about iterative pruning, the magnitude criterion is used [Frankle and Carbin, 2019]. Thus, in this chapter, we use exclusively `criterion` = `magnitude`. That means in each pruning step we remove connections with the smallest parameter magnitude $|\mathbf{w}|$. In iterative pruning, we usually remove only a small set of parameters in each step. Thus, the ones with the smallest magnitude are often very close to zero and really redundant.

Iterative pruning [Han et al., 2015] is a large improvement over one-shot methods. The algorithm consists of two steps – pruning and retraining. After each step of pruning, a short period of retraining is necessary to recover the lost accuracy. This form of retraining is often called finetuning. In finetuning, one can train the network using a constant, low learning rate. Usually the final learning rate of the network is used. In [Renda et al., 2020], where different retraining procedures are compared, finetuning is used as a baseline method. We include it for reference as well. In this version of iterative pruning, nothing is rewinded after training – this will be the main difference between it and other methods. Instead, we continue the training with a low learning rate until the desired density is achieved. This low learning rate is, in our case, the final learning rate the network was trained with. This method is described step by step in Algorithm 1.

Original Lottery Ticket Pruning described by [Frankle and Carbin, 2019] features an improved version of iterative magnitude pruning. They claim that in a randomly

Algorithm 1 Iterative Magnitude Pruning with Finetuning

Inputs

`trainSteps` ← total number of iterations the model normally needs to converge
`K` $\in [0.5, 1.0)$ ← density multiplier per iteration of pruning, often 0.8
`targetDensity` ← target density, e.g., 0.1 for 10× compression

Procedure

```
density ← K
Train(model, trainSteps)
Prune(model, density)
while density > targetDensity do
    density ← density · K
    Train(model, trainSteps)
    Prune(model, density)
end while
```

initialized neural network, there exists a subnetwork – a *Lottery Ticket* – that trains faster and is as accurate as the original network. One of the novelties in their procedure was rewinding the unpruned parameters of the network to the beginning of the training. In other words, rewinding the unpruned parameters back to their initialization. Previously, no parameters were rewinded. This way proved to work much better than a similar alternative – reinitializing the parameters randomly. This suggests that when pruning after training, the sparsity patterns are bounded to the network's initialization. This algorithm is described step by step in Algorithm 2.

Algorithm 2 Weight Rewinding [Frankle and Carbin, 2019]
Also known as Lottery Ticket Pruning

Inputs

`trainSteps` ← total number of iterations the model normally needs to converge
`K` $\in [0.5, 1.0)$ ← density multiplier per iteration of pruning, often 0.8
`targetDensity` ← target density, e.g., 0.1 for 10× compression

Procedure

```
density ← K
Train(model, trainSteps)          ▷ Train the model for trainSteps iterations
Prune(model, density)             ▷ Remove connections so density is reached
while density > targetDensity do
    density ← density · K
    RewindBy(model, trainSteps)   ▷ Reset parameters to their initial values
    Train(model, trainSteps)      ▷ Retrain until convergence
    Prune(model, density)
end while
```

[Frankle et al., 2020b] is another study on Lottery Ticket Pruning, where a modification of the original algorithm was proposed. They noticed that the originally proposed procedure was not effective when used on deeper neural networks such as ResNets from [He et al., 2016a]. They conclude that it happens due to abrupt changes in the network's weights in the early training phases (the network's lack of stability). As a solution, instead of rewinding the unpruned parameters to their initial weights, they rewind to an early point in the training (0.1% to 7% of total iterations). This variant of iterative magnitude pruning is described in Algorithm 3. Note that the only difference between it and Algorithm 2 is that in each iteration we rewind and retrain for fewer iterations than before.

Algorithm 3 Stable Weight Rewinding [Frankle et al., 2020b]

Inputs

`trainSteps` ← total number of iterations the model normally needs to converge
`K` $\in [0.5, 1.0)$ ← density multiplier per iteration of pruning, often 0.8
`targetDensity` ← target density, e.g., 0.1 for 10× compression
`rewSteps` ← number of steps to rewind (93% to 99.9% of `trainSteps`)

Procedure

```
density ← K
Train(model, trainSteps)
Prune(model, density)
while density > targetDensity do
    density ← density · K
    RewindBy(model, rewSteps)
    Train(model, rewSteps)
    Prune(model, density)
end while
```

[Renda et al., 2020] suggested yet another modification of iterative magnitude pruning. Their proposed Learning Rate Rewinding was reported to match or exceed Stable Weight Rewinding from [Frankle et al., 2020b] in all scenarios. To obtain the method, they do one modification to Algorithm 2. Rewinding the model (`RewindBy`) is decoupled into rewinding weights (`RewindParamBy`) and rewinding learning rate schedule (`RewindSchedBy`). In essence, they use Weight Rewinding procedure, but they skip parameter rewinding. After training and retraining, they do not modify the parameters, they rewind only the training schedule, including the learning rate schedule. They rewind it to the very beginning of the training. Excluding pruning, it resembles the cyclical learning rates from [Smith, 2017]. Since the weights are not rewinded, the training in the next iteration begins from the point of the previous convergence. This method is represented as yet another, but still similar, pseudocode in Algorithm 4.

To simplify, we generalize all those procedures in Algorithm 5. We call this

Algorithm 4 Learning Rate Rewinding [Renda et al., 2020]

Inputs

`trainSteps` ← total number of iterations the model normally needs to converge
`K` $\in [0.5, 1.0)$ ← density multiplier per iteration of pruning, often 0.8
`targetDensity` ← target density, e.g., 0.1 for 10× compression

Procedure

```
density ← K
Train(model, trainSteps)
Prune(model, density)
while density > targetDensity do
    density ← density · K
    RewindParamBy(model, 0)              ▷ Do not modify parameters
    RewindSchedBy(model, trainSteps)     ▷ Restart learning rate schedule
    Train(model, trainSteps)
    Prune(model, density)
end while
```

generalization GIMP from Generalized Iterative Magnitude Pruning. It can express previously described algorithms: Iterative Pruning with Finetuning, Weight Rewinding, Stable Weight Rewinding and Learning Rate Rewinding. The exact algorithm depends on its parameter choice: `rewSteps`, `rewStepsLR`, `retrainSteps`.

Algorithm 5 GIMP (Generalized Iterative Magnitude Pruning)

Inputs

`trainSteps` ← total number of iterations that the model needs for convergence
`K` $\in [0.5, 1.0)$ ← density multiplier per iteration of pruning, often 0.8
`targetDensity` ← target density, e.g., 0.1 for 10× compression
`rewSteps` ← rewind parameters this many steps in each pruning iteration
`rewStepsLR` ← rewind LR schedule this many steps in each pruning iteration
`retrainSteps` ← number of steps for retraining in each pruning iteration

Procedure

```
density ← K
Train(model, trainSteps)
Prune(model, density)
while density > targetDensity do
    density ← density · K
    RewindParamBy(model, rewSteps)
    RewindSchedBy(model, rewStepsLR)
    Train(model, retrainSteps)
    Prune(model, density)
end while
```

To obtain Weight Rewinding from Algorithm 2 in GIMP we would set:

- `rewSteps`, `rewStepsLR`, `retrainSteps` ← `trainSteps`

To obtain Stable Iterative Magnitude Pruning from Algorithm 3 in GIMP:

- `rewSteps`, `rewStepsLR`, `retrainSteps` ← 93% to 99.9% of `trainSteps`

To obtain Learning Rate Rewinding from Algorithm 4 in GIMP:

- `rewStepsLR`, `retrainSteps` ← `trainSteps`
- `rewSteps` ← 0

3.2 Learning Rate Rewinding

We were generally able to confirm the findings of [Renda et al., 2020] and reproduce their experiments with their method – Learning Rate Rewinding from Algorithm 4 – in Figure 3.1.

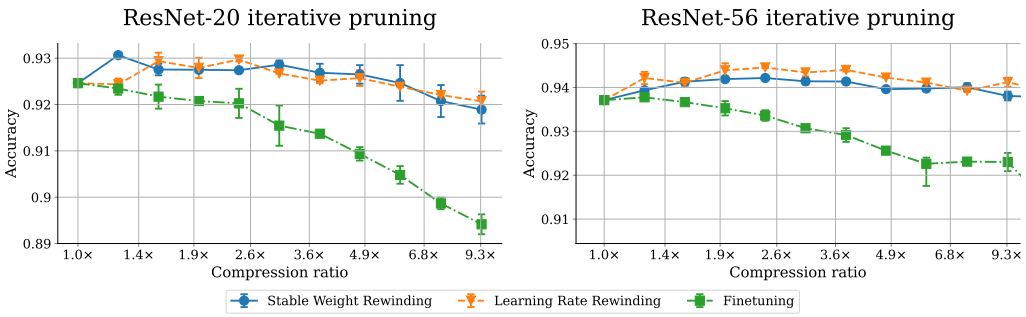

Figure 3.1: We are able to reproduce the experiments from [Renda et al., 2020]. Learning Rate Rewinding either matches or exceeds the accuracy of the other methods. However, the difference between Learning Rate Rewinding and Stable Weight Rewinding is rather slim. It can be observed in case of larger ResNet-56 but not really in case of ResNet-20. Finetuning is generally not an accurate method of retraining after pruning. Excluding finetuning, we observe an initial accuracy raise when pruning to low compression ratios.

We observe that Learning Rate Rewinding matches or exceeds the results of Stable Weight Rewinding (Algorithm 3) on ResNet architecture. In our implementation, we tightly follow procedures from [Renda et al., 2020]. We do not reset nor rewind the parameters after the training, but rather keep the parameters' values from convergence. Doing that, we observe an initial drop in accuracy. We suppose this is due to the large learning rate used at the beginning of the training. However,

later in the training the accuracy converges much quicker than if we reset the weights and start training from scratch. Thanks to this procedure, the final accuracy is often higher in Learning Rate Rewinding than in Stable Weight Rewinding. Another benefit over both Weight Rewinding and Stable Weight Rewinding is that the networks are often converging much faster during the training. This is because they start from an already converged set of weights. This fact might be used to create iterative methods which are much faster while still being as accurate. That being the case, during the experiments, both [Renda et al., 2020] and us retrain our networks for the full training time, although it is usually not necessary for the best accuracy.

However, in this work, we discover that there is more to Learning Rate Rewinding than originally presented. There is a previously unnoticed accuracy improvement over Stable Weight Rewinding in case of extremely sparse networks. Therefore, when pruning the networks to extreme sparsity values, we are achieving much better results with Learning Rate Rewinding. This is shown in Figure 3.2.

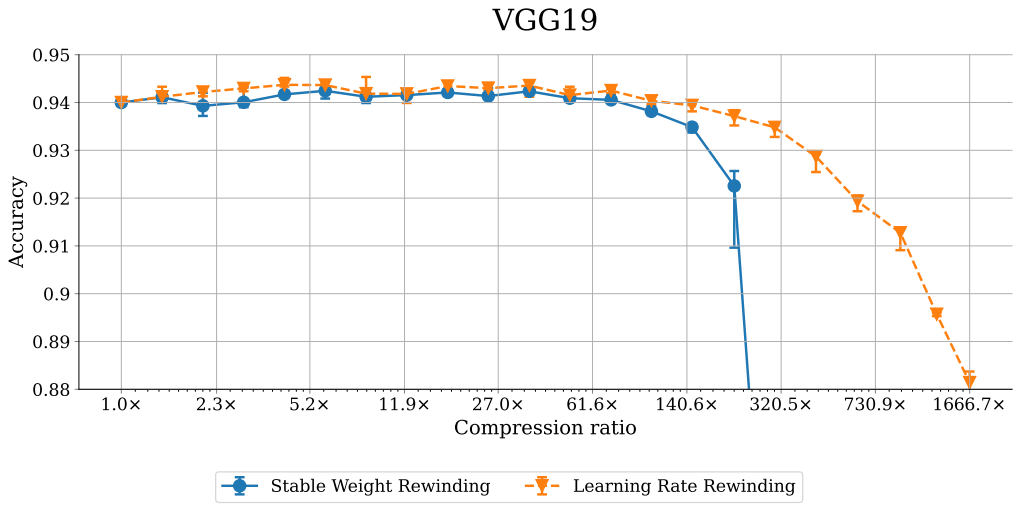

Figure 3.2: Learning Rate Rewinding superiority is especially visible in high sparsity domain. There's a clear gap between Stable Weight Rewinding and Learning Rate Rewinding methods, but the results only start diverging after many pruning iterations and in extreme sparsity domains. Extreme sparsity comparison between these methods was not shown in the literature before. Experiments in [Renda et al., 2020] focused on low sparsity pruning.

We recognize that Learning Rate Rewinding has many advantages over previous methods. However, as [Mikler, 2021] discovered, this method is not working well on larger neural networks. We repeated a similar set of experiments as [Renda et al., 2020] but with a different set of architectures. We used Wide ResNets from [Zagoruyko and Komodakis, 2016], which are much larger architectures than ResNets. For example, WRN-16-8, although shallower than all commonly used ResNet architectures, has almost 13 times as many parameters as the most popular ResNet-56. This is a result of it being 8 times wider (width corresponds to the

number of channels in a convolutional network). We found out that Wide ResNets are not working well with Learning Rate Rewinding. The pruning method proposed by [Renda et al., 2020] has catastrophic effects, it was shown in Figure 3.3. We refer to this effect as a catastrophic degradation of iterative Learning Rate Rewinding on large neural architectures.

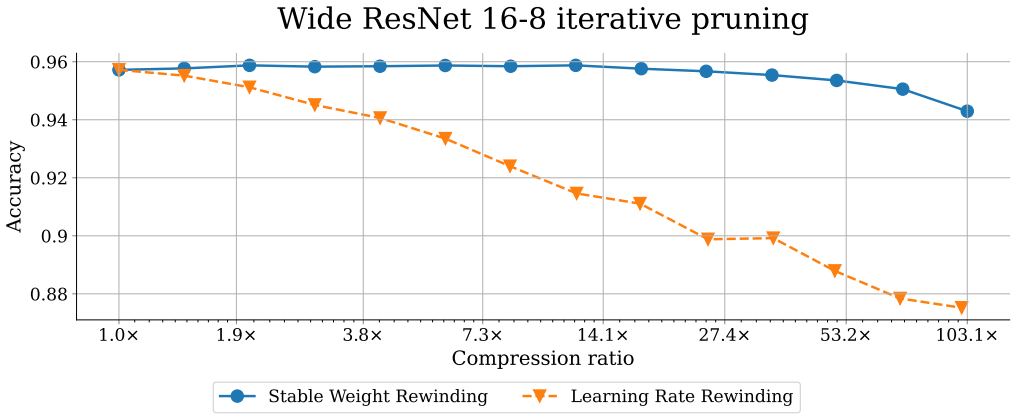

Figure 3.3: We observe a catastrophic degradation when following Learning Rate Rewinding procedure on Wide ResNet architecture, but we don't see it using the stable variant of Weight Rewinding from older [Frankle et al., 2020b]. Learning Rate Rewinding, which was supposed to be an improvement, is clearly worse than its predecessor.

We can inspect this problem further by comparing one-shot pruning at different sparsity levels. As a remainder – one-shot pruning is equivalent to one iteration long iterative pruning. We expect to see some effects of catastrophic degradation even after just one step of pruning. The following question is interesting: is Learning Rate Rewinding bad for all step sizes in case of Wide ResNet? As [Mikler, 2021] suggested, it is not and only low sparsity Learning Rate Rewinding is catastrophic for Wide ResNet. This can be seen in Figure 3.4. For higher sparsity domains, Learning Rate Rewinding is better than Stable Weight Rewinding. However, iterative pruning is a process that performs multiple low sparsity prunings. This is why the negative effect stacks up, causing the larger catastrophic degradation of iterative Learning Rate Rewinding.

On large architectures, Learning Rate Rewinding is showing a degradation when compared to Stable Weight Rewinding (Algorithm 3) or even to their predecessor – original Weight Rewinding (Algorithm 2) also known as Lottery Ticket Pruning [Frankle and Carbin, 2019]. We suggest that this is because in Learning Rate Rewinding each retraining starts from a fully converged neural network. Large networks tend to overfit the training data a lot and overfitted networks tend to be unstable. Such networks might not be a good starting point for the next training cycle. We propose ways to cope with this problem while keeping the advantages of

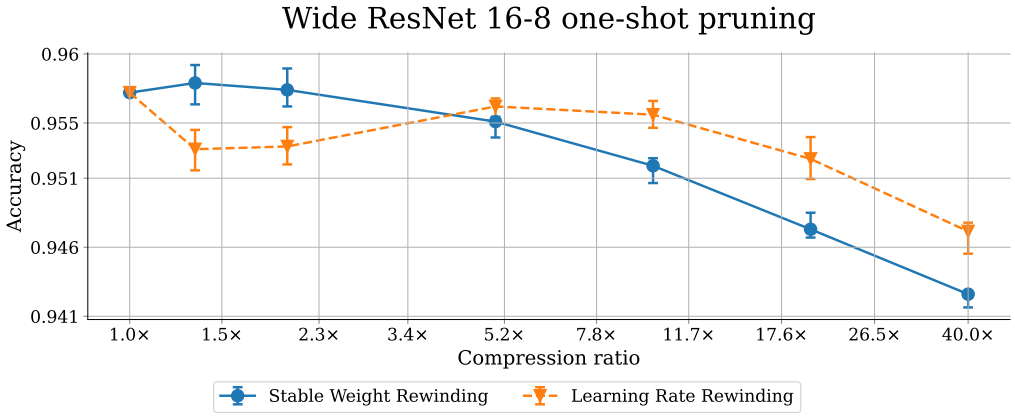

Figure 3.4: There is a visible accuracy degradation of Learning Rate Rewinding (Algorithm 4) in certain sparsity ranges. This effect does not occur in case of Stable Weight Rewinding (Algorithm 3). Moreover, for high sparsity (high compression) pruning, Learning Rate Rewinding turns out to be better, which is rather interesting. This suggest that high sparsity one-shot Learning Rate Rewinding still works better than Stable Weight Rewinding, even for large architectures.

Learning Rate Rewinding.

We remind that GIMP is a generalization of Lottery Ticket Pruning (Algorithm 2) and Learning Rate Rewinding (Algorithm 4). For clearness, we will explain the exact parameter choice to obtain these from GIMP framework. In the original Lottery Ticket Pruning from Algorithm 2, `rewSteps` is equal to 100% of `trainSteps`, so every training iteration is fully reversed before retraining. This means we start retraining from a totally random set of parameters. This is problematic due to instability at the beginning of the training as discovered by [Frankle et al., 2020b]. They claim Lottery Ticker Pruning does not work well on larger architectures, due to large parameter changes during the early phase of gradient-based network training. We can obtain the other method – Learning Rate Rewinding from Algorithm 4 by using $\texttt{rewSteps} = 0$ in our GIMP framework. However, this version seems to be problematic for even larger architectures, most likely due to their heavy overfitting. Overfitted networks tend to be unstable and are not a good starting point for parameter retraining.

Unlike in Learning Rate Rewinding, in the GIMP framework which we propose, the starting point for the subsequent training cycle is not necessarily the final state of the previous network. By modifying `rewSteps`, we can rewind the network to different phases before the retraining. We will investigate what are the benefits of using values $\texttt{rewSteps} \neq 0$ and $\texttt{rewSteps} \neq 100\%$ of `trainSteps`. We will observe that some values of `rewSteps` solve the accuracy degradation problem observed in Wide ResNet architectures.

To fix the aforementioned problem, we generalized the approaches from previous works in GIMP. Now we propose using the following variables in GIMP (Algorithm 5):

$$\texttt{rewStepsLR}, \texttt{retrainSteps} \leftarrow \texttt{trainSteps}$$
$$\texttt{rewSteps} \leftarrow [0\%, 100\%] \cdot \texttt{trainSteps} \text{ (recommended 50\% to 98\%)}$$

Our proposition is a stable variant of GIMP with learning rate rewinding. It can be considered a mix between Learning Rate Rewinding and Stable Weight Rewinding. For simplicity, we will call it sGIMP as stable Generalized Iterative Magnitude Pruning. As above, we do not suggest a precise choice of `rewSteps`. The recommended interval – $[50\%, 98\%] \cdot$ `trainSteps` worked well in all our experiments. Generally, we define sGIMP(rewind `X%`) as an iterative pruning algorithm where in each pruning iteration we:

1. reset the learning rate to the beginning of the training,
2. rewind the parameters by `X%` of `trainSteps` (total number of steps in the training).

Algorithm 6 sGIMP(rewind `X%`)

Inputs
`trainSteps` ← total number of iterations that the model needs for convergence
`K` $\in [0.5, 1.0)$ ← density multiplier per iteration of pruning, often 0.8
`targetDensity` ← target density, e.g., 0.1 for 10× compression

Procedure
```
density ← K
Train(model, trainSteps)
Prune(model, density)
while density > targetDensity do
    density ← density · K
    RewindParamBy(model, X% · trainSteps)
    RewindSchedBy(model, trainSteps)
    Train(model, trainSteps)
    Prune(model, density)
end while
```

As a first examination of our sGIMP method, we present Figure 3.5. For Wide ResNet 16-8, we can see that there is a clear accuracy difference between

sGIMP(rewind 0%) and other values. In the case of VGG19 for the presented compression ratios, the accuracy does not go down as much for neither of these methods. However, for some of them, it goes up by a significant amount during iterative pruning. We observe some differences between different sGIMP variants. Accuracy is clearly the worst for sGIMP(rewind 100%). Therefore, it is better to rewind to a nonzero iteration of the previous training. In other words, in the case of VGG19, it is best to not use sGIMP(rewind 100%). On the other hand, for Wide ResNet 16-8, sGIMP(rewind 100%) yields similar results as other variants. For WRN architecture, it is best to not use sGIMP(rewind 0%), which is equivalent to Learning Rate Rewinding from [Renda et al., 2020]. Summing up, rewind 100% does not work well for VGG network and rewind 0% does not work well for WRN network. Everything in between seems like a sensible choice for both networks.

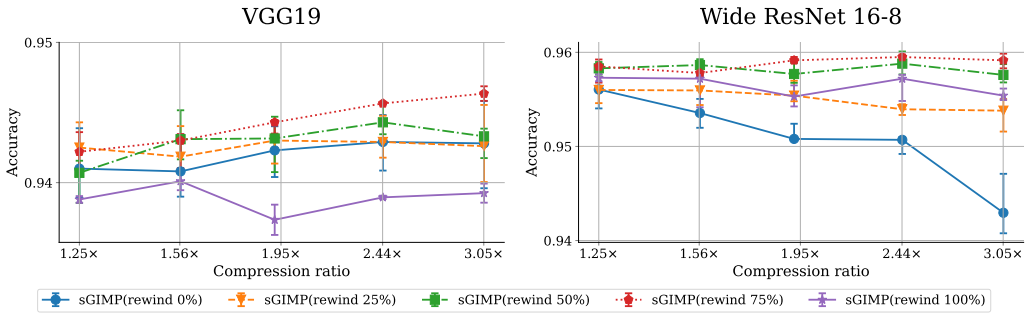

Figure 3.5: GIMP with `rewSteps` varying between 0% and 100% of `trainSteps`. Special cases: sGIMP(rewind 0%) and sGIMP(rewind 100%) are Learning Rate Rewinding (Algorithm 4) and Weight Rewinding (Algorithm 2). We see that for VGG19 network rewind 100% does not look promising. But for Wide ResNet it is another value – rewind 0% – that is the worst. This experiment focuses exclusively on low compression ratios.

So far, we saw small differences between different sGIMP variants. We examine if those small differences stack up in case of high sparsity iterative pruning. On Figure 3.6 we compare high sparsity iterative pruning of Wide ResNet 16-8. We arrive to a conclusion that – at least in the case of large architectures – our method is superior to Learning Rate Rewinding, which suffers from catastrophic accuracy degradation. In the upcoming figures, one should notice that our sGIMP works just as well or better than competing methods. In some cases, the advantage of sGIMP is clear. In others, we at least match the existing results. While sGIMP requires a choice of an additional parameter, we tested that sGIMP(rewind `X`%) for $\texttt{X} \in [50, 98]$ works well in all cases. As a safety measure, one can avoid extreme values from this range and use $\texttt{X} \in [75, 95]$. In our experiments, we used $\texttt{X} = 75$ most often.

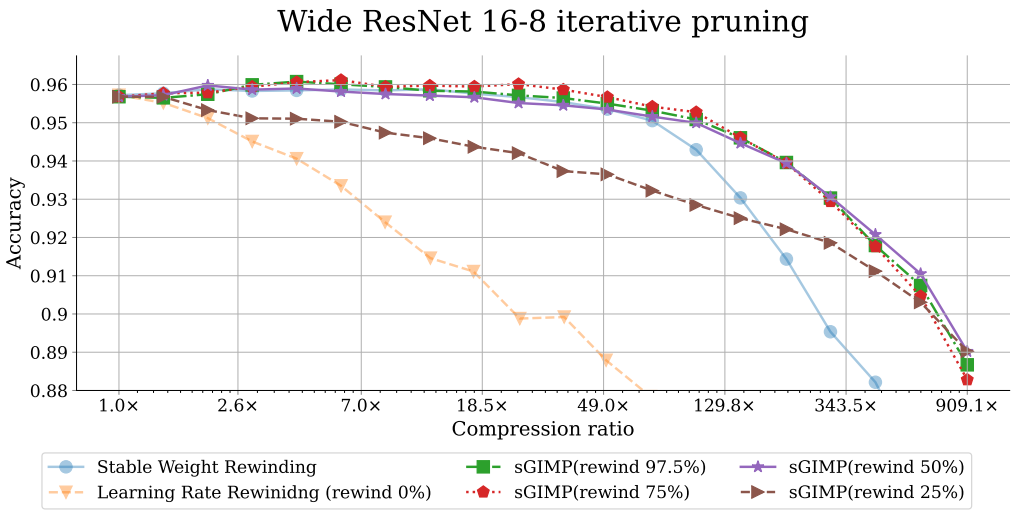

Figure 3.6: Our method obtains more accurate variants of sparse Wide ResNet 16-8 at the same sparsity levels as existing methods. sGIMP is, to our knowledge, the best method for pruning large neural networks in terms of both accuracy and compression ratio. In our iterative pruning method, it is usually irrelevant to what point of the training we rewind. Result worsening starts when we rewind to near the end of the training, where the network reaches convergence. Results would worsen as well if we rewound to the beginning of the training. Rewinding 0% (to the end of the previous training) is equivalent to Learning Rate Rewinding. Rewinding 100% (to the random initialization) is like original Lottery Ticket Pruning procedure from Algorithm 2. Everything in between is our generalized method. Here we can clearly see that rewinding 50%, 75% and 97.5% of the total number of iterations outperform other methods. Rewinding 25% show first signs of degradation, probably because it rewinds to a state too close to the end of the training and the convergence.

Results from Figure 3.6 and 3.8 make us wonder: how to recognize a point to which we can safely rewind to? In our practice, we were always able to safely rewind to an early point in the training, for example, by rewinding 75% of training steps. It worked well on all architectures we tested so far. It might be impractical and risky to rewind to a later point in the training (so rewind a lower number of iterations, like 25% or 50%). At the same time, we do not see any advantages of rewinding to a later point in the training. We suspect that catastrophic degradation occurs due to the network's overfitting. If the network starts converging and overfitting too early, it will be in risk of severe catastrophic degradation as we have seen in Figure 3.3.

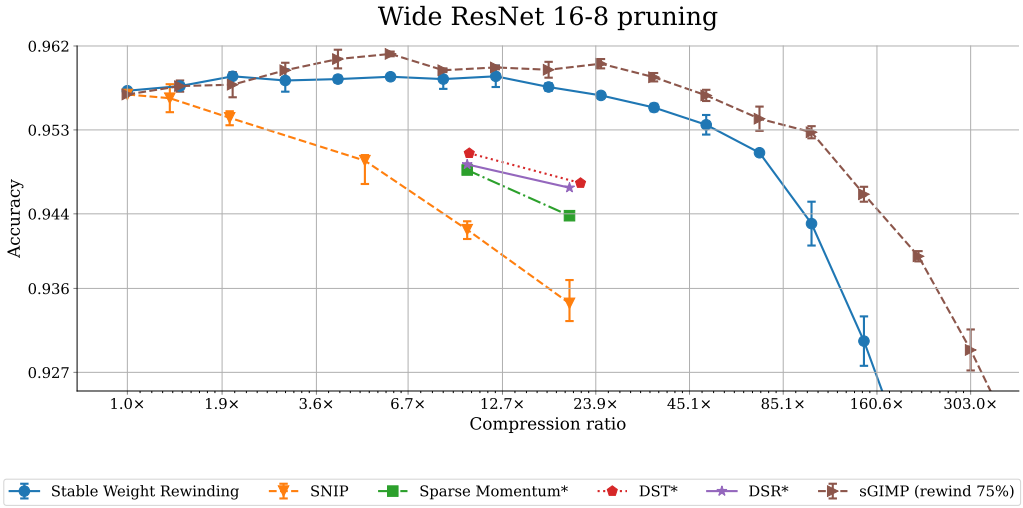

Figure 3.7: Our method compared to others, including non-iterative methods. SNIP [Lee et al., 2018], Sparse Momentum [Dettmers and Zettlemoyer, 2019], DST (Dynamic Sparse Training) [Liu et al., 2019a], DSR (Dynamic Sparse Reparametrization) [Mostafa and Wang, 2019], Stable Weight Rewinding [Frankle et al., 2020b]. Learning Rate Rewinding [Renda et al., 2020] was not included due to especially bad results as shown in Figure 3.3. In this particular comparison our method is rewinding 75% of the training updates. In this figure, both iterative and one-shot methods are compared. Best performing methods – ours and Stable Weight Rewinding are more computationally expensive than the rest. *Sparse Momentum, DST and DSR were not reproduced by us, but results were taken from the original papers.

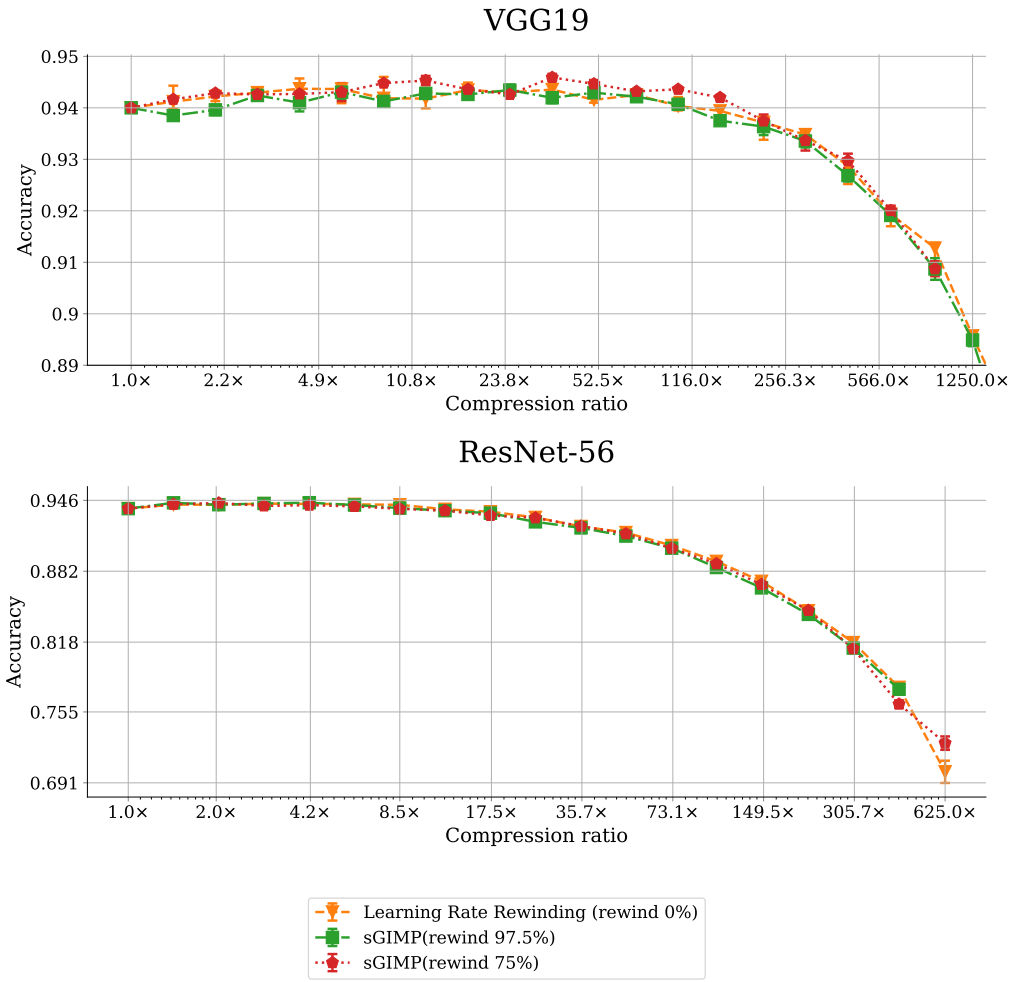

Figure 3.8: In case of smaller networks, like VGG and ResNet, step to which we rewind is irrelevant. In contrary to WRN16-8, rewinding 0% works just as well as other values. As a remainder, rewinding 0% is equivalent Learning Rate Rewinding from Algorithm 4. One should assume that rewinding any number from range $[0\%, 97.5\%]$ would work just as well for those architectures. An important conclusion is that we see no negative effect of our sGIMP in comparison to Learning Rate Rewinding for both VGG and ResNet architectures.

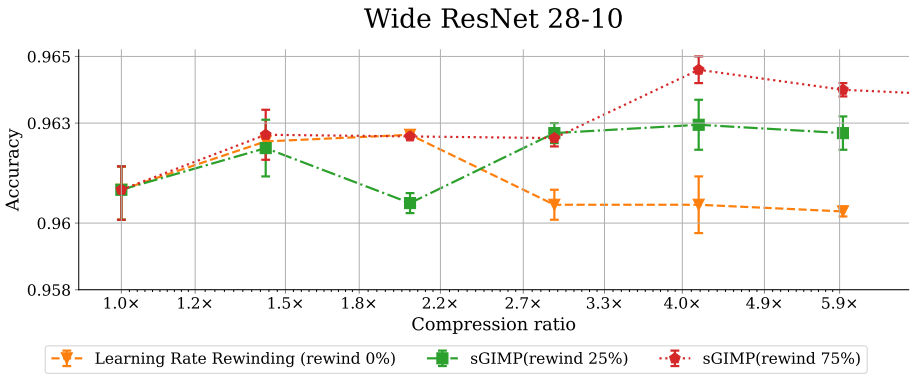

Figure 3.9: For Wide ResNet 28-10, similarly as for Wide ResNet 16-8, we observe that our method works better than Learning Rate Rewinding from [Renda et al., 2020]. Interestingly, for this architecture the degradation is not as big as in Figure 3.3 and 3.6. Also, we once again confirm that the best performing parameter for sGIMP(rewind `X%`) is from range $[50\%, 97.5\%]$.

3.3 More Results

3.3.1 Training Optimal Masks

Main advantage of one-shot methods over iterative is the amount of compute needed to obtain neural networks with desired sparsity. The disadvantage is that the quality of such spare networks is lower [Frankle et al., 2020a]. The accuracy of networks pruned with one-shot pruning algorithms is lower than those pruned with iterative algorithms, as it is shown in Figure 3.7. As described in Section 2.4.3, there are methods that are neither one-shot nor iterative. However, these are close to one-shot in terms of their compute needs and accuracy. There is yet to be discovered a method as accurate as iterative pruning, but less computationally demanding.

Inspired by binary networks from [Courbariaux et al., 2016] we design our own compute efficient method, trying to match the accuracy of iterative pruning. In [Zhou et al., 2020] there is a concept of supermasks. They found out that by taking an untrained neural network and applying proper pruning, we obtain a sparse network with better than random accuracy. They train these masks with backpropagation and then use them with random networks. We observed the same phenomenon independently. We observe that, at least in the case of small architectures, just by using the masks found by iterative pruning, we get a network with better than random accuracy even though it has random parameters.

Our conclusion is, *proper pruning is training*. This leads us to the idea that we can obtain good masks (sparse structure) by training the masks directly. We can do this by following research on binary networks, where all parameters of the network are binary $\{0, 1\}$ (sometimes teritary: $\{-1, 0, 1\}$). Our case is similar – the network is parametrized by parameters $\mathbf{w} \in \mathbb{R}^n$ and masks $\mathbf{c} \in \{0, 1\}^n$.

Pretrain the Parameters

We do not start by training the masks. Following observations from [Frankle et al., 2020b], we want to avoid instability issues. Those issues occur in networks in their early training phases, where randomly initialized parameters are being rapidly modified. To avoid that, we pretrain the dense network's parameters until stability is achieved. For that, we use networks trained for 10% of their total number of iterations. After this, the parameters are frozen.

Train the Masks

Besides masks $\mathbf{c} \in \{0, 1\}^n$, we create binary distributions $\mathbf{b} \in \mathbb{R}^n$ for each parameter. During training: in each step, we draw a binary number from the distribution. We

set $\mathbf{c}$ randomly according to:

$$P(c_i = 1) = \sigma(b_i)$$
$$P(c_i = 0) = 1 - \sigma(b_i)$$

Where $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$ is a sigmoid function. Thanks to sigmoid's properties, we know that $\sigma(b_i) \in [0, 1]$, so we can use it as probability. Following straight-through estimator from [Bengio et al., 2013], we calculate the gradients of the loss function (categorical crossentropy) with respect to masks: $\frac{\delta L}{\delta \mathbf{c}}$, but we use them to update $\mathbf{b}$. Before that, we apply gradient clipping to $[-0.1, 0.1]$ range. Summing up, in every training step we update $\mathbf{b}$ in the following way:

$$\mathbf{b}' = \mathbf{b} - \alpha \cdot \texttt{clip}(\frac{\delta L}{\delta \mathbf{c}}, -0.1, 0.1)$$

Also, we find that it is good to clip b at extreme values, such as values outside range $[-15, 15]$. During validation: in each step, we select c value according to:

$$c_i = 1 \texttt{ if } b_i \geq x_t \texttt{ else } 0$$

Where x_t is a value such that exactly $t\%$ of $\{b_1, b_2, ..., b_n\}$ are larger than x_t. This allows us to choose the desired sparsity by choosing t. Network's sparsity during validation is then $t\%$. Usually, L2 or L1 penalty is needed to see p decaying. During masks training, we expect to see the accuracy close to that of the dense network's accuracy. For example, for VGG19 network, we achieve almost 90% accuracy just by training the masks with over 95% sparsity.

Train the Parameters

After mask training, we can get rid of the optimized distributions $\mathbf{b}$. We use them to obtain final masks $\mathbf{c}$. At this point, we have settled on specific masks, thus the sparse network's topology. As the last step, we train the parameters as usually, for the full training time, until convergence.

The results are rather promising. When used in a simple neural network as presented in Table 3.1, we found out that this is the only method that reaches a performance similar to iterative pruning. We achieve $(98.639 \pm 0.039)\%$ accuracy on MNIST dataset with our Training Optimal Masks, whereas we get $(98.624 \pm 0.039)\%$ in case of iterative pruning. In Figure 3.10 we can see that our method matches or beats the competing one-shot pruning methods on a larger CIFAR-10 dataset and VGG19 architecture. In the figure we see that the accuracy of iterative methods is still out of reach. Those, however, are much more computationally expensive.

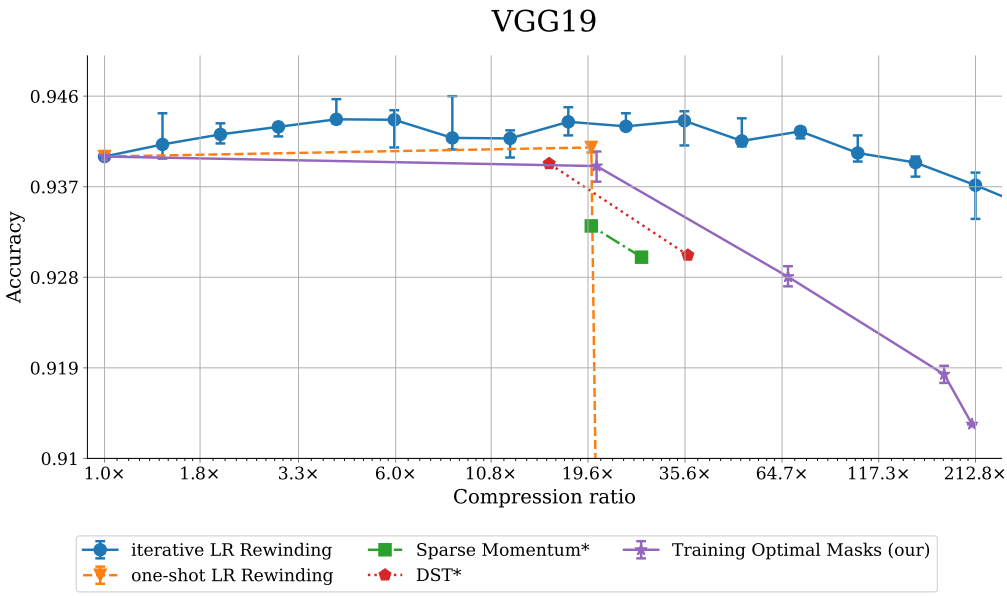

Figure 3.10: Our Training Optimal Masks has worse accuracy than iterative methods. However, it performs as good or better as existing one-shot methods, while having similar compute requirements. We can see that one-shot Learning Rate Rewinding fails to find any trainable networks under 95% sparsity. *Sparse Momentum [Dettmers and Zettlemoyer, 2019] and DST [Liu et al., 2019a] were not reproduced by us, but results were taken from the original papers.

3.3.2 Sparse Mask Modifications

Sparse networks might be implemented by masking pruned parameters. In this implementation, besides parameters $\mathbf{x} \in \mathbb{R}^n$ there is mask $\mathbf{c} \in \{0,1\}^n$ for each parameter. If the parameter's mask is equal to 0, the parameter is considered pruned and will not be used anymore. This is equal to zeroing parameter's weight. After the process of pruning, we might consider the updated parameters being $\mathbf{x}' = \mathbf{c} \cdot \mathbf{x}$. All pruning algorithms, both one-shot and iterative, return masks $\mathbf{c}$ in the end. As [Frankle and Carbin, 2019] has shown, masks in iterative algorithms are most effective for a specific set of parameters. This is true even in Algorithm 2, where parameters are rewinded to their values from the initialization, which were random. If we use masks from this algorithm, but reinitialize the parameters randomly again, we would observe a decrease in accuracy.

We conduct experiments on this matter. We are interested in whether we can modify either masks or weights without losing accuracy. We ask how tightly the parameters and masks are coupled in a sparse neural networks. To do so, we design a small feedforward neural network and train it using MNIST digit classification dataset [LeCun, 1988b]. Our neural network has 784 inputs, since MNIST dataset

contains 28×28 images. Then there are three 400-neuron layers, each finished with ReLU activation. The network ends with 10 outputs, as this is the number of classes in MNIST digit classification task. This makes the total number of parameters $784 \cdot 400 + 2 \cdot 400 \cdot 400 + 400 \cdot 10 = 637\,600$, excluding biases. We prune this network using Weight Rewinding from [Frankle and Carbin, 2019] (Algorithm 2) to 96% sparsity. This leaves 25 504 nonzero kernel parameters.

Modification	**MNIST Accuracy (%)**	**Difference**
None	98.624 ± 0.035	-0.000
Shuffle parameters within masks (A.)	98.315 ± 0.056	-0.309
Reinitialize parameters	98.259 ± 0.084	-0.365
Shuffle masks and weights in layer (B.)	97.350 ± 0.050	-1.274
Shuffle masks in layer and reinitialize	96.897 ± 0.230	-1.727

Table 3.1: Results of sparse (96% sparsity) feedforward neural network on MNIST classification dataset. Modifications were included. First two modifications do not modify the sparse topology and modify only parameters. Either by shuffling existing values or introducing new ones by reinitialization. Last two modifications do modify the sparse topology by shuffling masks in each layer. Such shuffling keeps each layer's sparsity intact. One can see shuffling examples and their visualization in Figure 3.11.

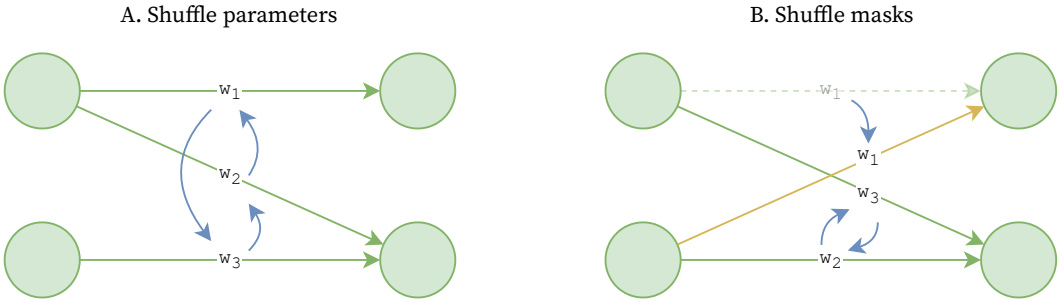

Figure 3.11: Shuffling examples from Table 3.1 visualized. Option A keeps the sparse topology intact. Option B shuffles masks, modifying the topology. Shuffling operation is random, diagrams show only an example of it.

From Table 3.1 we can see that every modification has a negative effects on the masks found by Lottery Ticket Pruning. However, the modification type matters. We see that when we keep the sparse topology intact, the accuracy does not decrease nearly as much as when we modify it. Shuffling parameters within masks checks whether it is the general distribution of parameters that is important (e.g., large values) or their specific values. It turns out, specific values are important. Moreover, there is no significant difference between shuffling parameters and reinitializing them.

If we modify the sparse topology, even when keeping the sparsity constant in each layer, we observe a higher drop in accuracy. In this scenario, the drop is lesser if we keep the original distribution of parameters. Reinitialization has a clear negative impact on the sparse network's accuracy.

3.3.3 Batch Normalization Parameters

Batch normalization description, including its hyperparameter, was described in detail in Section 2.3. We discover an unexpected impact of batch normalization parameters on iterative pruning. The exact choice of batch normalization momentum μ and ϵ has no clear effect on the performance of dense networks and many papers choose different values [He et al., 2016a, Zagoruyko and Komodakis, 2016]. Furthermore, neural network frameworks have different default values: TensorFlow [Abadi et al., 2015] has $\mu = 0.99, \epsilon = 10^{-3}$ and PyTorch [Paszke et al., 2019] has $\mu = 0.9$, $\epsilon = 10^{-5}$. Although it does not usually seem to have any influence on dense networks, in the case of iterative pruning, we observed an unexpectedly large impact of those hyperparameters.

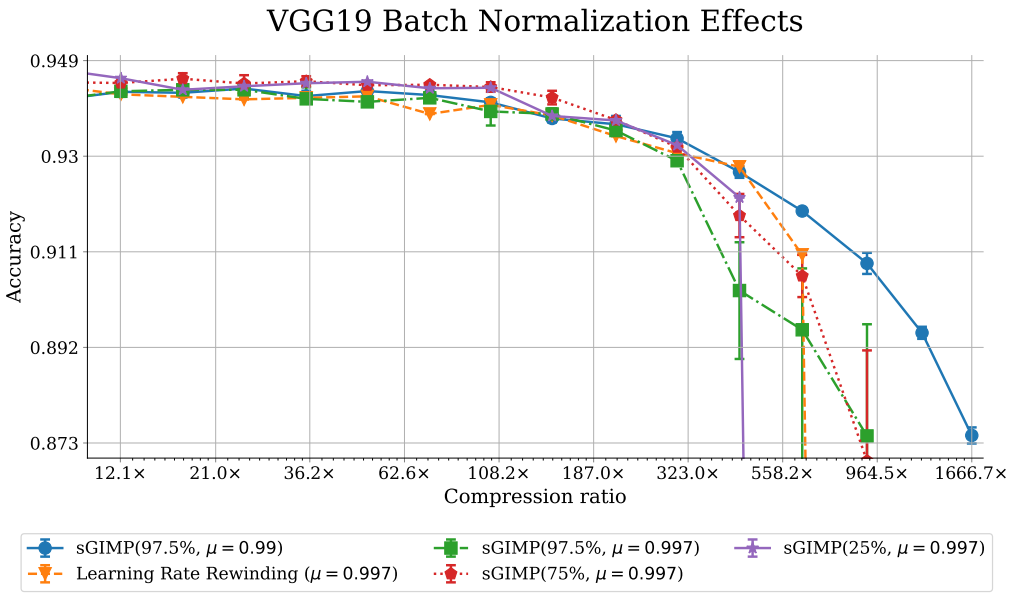

Figure 3.12: Testing the effect of μ, ϵ hyperparameters choice on iterative pruning algorithms. Here we compare $\mu = 0.99, \epsilon = 10^{-3}$ and $\mu = 0.997, \epsilon = 10^{-5}$. These μ and ϵ values were used in conjunctions. If $\mu = 0.997$ then $\epsilon = 10^{-5}$ and if $\mu = 0.99$ then $\epsilon = 10^{-3}$.

In Figure 3.12 we tested effects of choosing either $\mu = 0.99, \epsilon = 10^{-3}$ or $\mu = 0.997, \epsilon = 10^{-5}$ in VGG19 network. We found out that $\mu = 0.99, \epsilon = 10^{-3}$ offered stable and high accuracy in case high sparsity pruning, which cannot be said in case of $\mu = 0.997, \epsilon = 10^{-5}$.

While μ is not used during training, only during validation, ϵ is used in both training and validation. It was described in detail in Section 2.3. We observe that even such small disturbances cause a rather large difference in performance. This suggests that sparse neural networks are rather unstable and more sensitive to disturbances than their dense counterparts.

3.4 Methodology

All tested networks and training algorithms were implemented by ourselves and are available in our repository[1]. Our implementation is in TensorFlow [Abadi et al., 2015] under Python 3. For GPU computing, we use multiple RTX 3090 and RTX 3080 GPUs.

To implement the architectures, we follow the directions from the papers that introduced them. The exception is VGG (also often found in the literature under the names VGG-B or VGG-C). For VGG, we use batch normalization layers which were not originally used and we finish the network with a global pooling and only a single classification layer.

Regarding hyperparameters, we try to match the most popular or most efficient values known to us. We follow [He et al., 2016a, He et al., 2016b, Zagoruyko and Komodakis, 2016, Frankle and Carbin, 2019] where applicable.

We use L2 regularization for every model. In the case of WRN models, we use with the coefficient 0.0002, whereas in other models we use 0.0001. There is a possibility of confusing L2 coefficient with weight decay, which describes the same thing, but the coefficient value means something else. We will explain this difference.

Let L be a loss function and α be learning rate. Usual parameter update in neural network training can be described as:

$$\mathbf{w}' = \mathbf{w} - \alpha \frac{\delta L(\mathbf{w})}{\delta \mathbf{w}}$$

Weight decay is defined as follows:

$$\mathbf{w}' = \mathbf{w} - \alpha \left(\frac{\delta L(\mathbf{w})}{\delta \mathbf{w}} + \beta_1 \mathbf{w} \right)$$

L2 penalty is added directly to the loss, before differentiation:

$$\mathbf{w}' = \mathbf{w} - \alpha \frac{\delta L(\mathbf{w}) + \beta_2 \sum \mathbf{w}^2}{\delta \mathbf{w}}$$

Those are almost equivalent, because:

$$\frac{\delta \beta_2 \sum \mathbf{w}^2}{\delta \mathbf{w}} = 2\beta_2 \mathbf{w}$$

[1] `https://github.com/gahaalt/on-pruning-methods-for-neural-networks`

Thus, only for $\beta_1 = 2\beta_2$ both regularization methods are exactly equivalent. In this work, we state the coefficients in terms of L2 regularization and not weight decay.

Weight decay is often used in the popular PyTorch framework [Paszke et al., 2019]. L2 regularization is often used in another popular framework – TensorFlow [Abadi et al., 2015].

In all ResNet experiments, we use batch normalization momentum equal to 0.997 with epsilon 10^{-5}. For the rest, including WRN and VGG networks, we use batch normalization momentum equal to 0.99 with epsilon 10^{-3}.

Learning rate changes during our trainings. We follow a piecewise constant decay, which decays the learning rate in a stepwise manner. We train ResNet architectures for 72 000 iterations starting with learning rate 0.1. We decrease the learning rate tenfold after 36 000 and 54 000 iterations, finishing the training with learning rate 0.001. In the case of VGG architectures, we train for 80 000 iterations starting with learning rate 0.1, then decaying it five-fold three times at iterations 32 000, 48 000 and 64 000. In case of WRN architectures, we train for 80 000 steps starting with learning rate 0.2. During training, we decrease it five-fold three times at 32 000, 48 000 and 64 000 step. In the case where the training is prolonged, like in finetuning from Algorithm 1, we continue the training with the final learning rate.

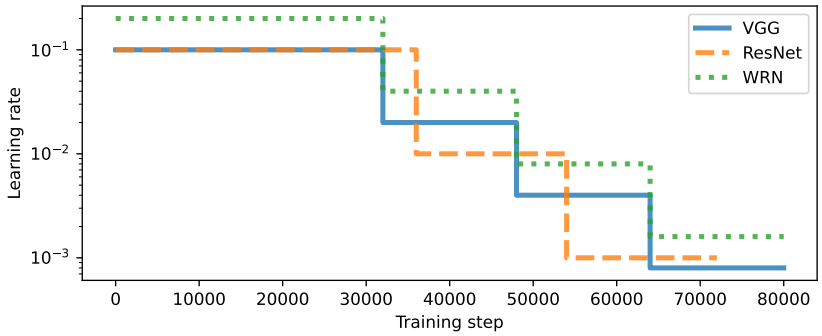

Figure 3.13: Learning rate schedule of different architectures used in this work.

We use all 50 000 images from CIFAR-10 for training. During Stochastic Gradient Optimization, we use batches of size of 128 examples.

During iterative pruning, we use a pruning step of 30% instead of 20% used in [Frankle and Carbin, 2019] as it makes our computations a little bit faster. With this pruning step, the desired sparsity is achieved quicker. We did not see any negative effect of this change.

3.5 Conclusions

We see that iterative pruning methods allow us to obtain extremely sparse neural networks. They have only a fraction of the parameters of their dense counterparts, while being as accurate or almost as accurate. In some cases, we reduce the network's size even $1000 - 1500$ times. Networks that are less, but still significantly sparse, e.g., with $50\times$ compression ratio, are often better than their dense counterparts. Our proposed Generalized Iterative Magnitude Pruning (GIMP) from Algorithm 5 describes many different algorithms in one framework. Its variant, stable Generalized Iterative Magnitude Pruning or sGIMP for short, is better than existing state-of-the-art pruning methods for all tested architectures. Our recommendation for iterative pruning is to rewind the networks' parameters by $[75\%, 95\%]$ of networks' total training iterations after each pruning step. All values in the recommended range seem to work just as well, but the higher spectrum of it is most likely more stable.

Our proposition for time-efficient pruning is called Training Optimal Masks. The main point in this method is training sparsity masks directly, using gradient updates. After the last phase of this procedure, parameter retraining, it achieves better accuracy than existing time-efficient pruning methods. It is visible especially in high sparsity ranges. Training Optimal Masks works well, but is rather complicated. It requires an additional mask training phase compared to classical one-shot pruning. We found out that this is the only method that matches the iterative pruning results on our small feedforward network example. However, time-efficient methods, like Training Optimal Masks, are still far from iterative pruning for most architectures. It would be an interesting topic of future research to check why there is such a gap in accuracy between the methods.

It turns out that the sparse masks discovered in iterative pruning processes are only working well when coupled with the original networks' parameters. Sparse networks are sensitive to parameter shuffling operations, even if the masks stay intact. Moreover, we observed an unexpected impact of batch normalization hyperparameters on iterative pruning. This leads us to a conclusion that sparse neural networks are highly volatile to disturbances in the training process. The robustness of sparse neural networks should be further examined.

Bibliography

[Abadi et al., 2015] 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.

[Anthony et al., 2020] Anthony, L. F. W., Kanding, B., and Selvan, R. (2020). Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *arXiv:2007.03051 [cs, eess, stat]*. arXiv: 2007.03051.

[Bengio et al., 2013] Bengio, Y., Léonard, N., and Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv:1308.3432 [cs]*. arXiv: 1308.3432.

[Blalock et al., 2020] Blalock, D., Ortiz, J. J. G., Frankle, J., and Guttag, J. (2020). What is the state of neural network pruning? arXiv: 2003.03033 Citation Key: Blalock2020.

[Brown et al., 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., and et al. (2020). Language models are few-shot learners. *arXiv:2005.14165 [cs]*. arXiv: 2005.14165.

[Courbariaux et al., 2016] Courbariaux, M., Bengio, Y., and David, J.-P. (2016). Binaryconnect: Training deep neural networks with binary weights during propagations. *arXiv:1511.00363 [cs]*. arXiv: 1511.00363.

[Cubuk et al., 2019] Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. (2019). Autoaugment: Learning augmentation policies from data. *arXiv:1805.09501 [cs, stat]*. arXiv: 1805.09501.

[Dettmers and Zettlemoyer, 2019] Dettmers, T. and Zettlemoyer, L. (2019). Sparse networks from scratch: Faster training without losing performance. (1):1–14.

[Frankle and Carbin, 2019] Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. *7th International Conference on Learning Representations, ICLR 2019*, page 1–42. arXiv: 1803.03635 Citation Key: Frankle2019.

[Frankle et al., 2020a] Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. (2020a). Pruning neural networks at initialization: Why are we missing the mark? arXiv: 2009.08576.

[Frankle et al., 2020b] Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. (2020b). Stabilizing the lottery ticket hypothesis.

[Han et al., 2015] Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. *arXiv:1506.02626 [cs]*. arXiv: 1506.02626.

[Hassibi et al., 1993] Hassibi, B., Stork, D., and Wolff, G. (1993). Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, page 293–299 vol.1.

[He et al., 2016a] He, K., Zhang, X., Ren, S., and Sun, J. (2016a). Deep residual learning for image recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-Decem:770–778. arXiv: 1512.03385 Citation Key: He2016 ISBN: 9781467388504.

[He et al., 2016b] He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Identity mappings in deep residual networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9908 LNCS:630–645. arXiv: 1603.05027 ISBN: 9783319464923.

[Howard et al., 2017] 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: 1704.04861 Citation Key: Howard.

[Huang et al., 2016] Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. (2016). Deep networks with stochastic depth. *arXiv:1603.09382 [cs]*. arXiv: 1603.09382.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167 [cs]*. arXiv: 1502.03167.

[Janowsky, 1989] Janowsky, S. A. (1989). Pruning versus clipping in neural networks. *Physical Review A*, 39:6600–6603. Citation Key: 1989PhRvA..39.6600J ADS Bibcode: 1989PhRvA..39.6600J.

[Kaplan et al., 2020] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv:2001.08361 [cs, stat]*. arXiv: 2001.08361.

[Krizhevsky et al., 2009] Krizhevsky, A., Nair, V., and Hinton, G. (2009). Cifar-10 (canadian institute for advanced research).

[LeCun, 1988a] LeCun, Y. (1988a). Handwritten Digit Recognition with a Back-Propagation Network. In *Neural Information Processing Systems*. American Institute of Physics.

[LeCun, 1988b] LeCun, Y. (1988b). Handwritten digit recognition with a back-propagation network. In *Neural Information Processing Systems*. American Institute of Physics.

[LeCun et al., 1990] LeCun, Y., Denker, J., and Solla, S. (1990). Optimal brain damage. In Touretzky, D., editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann.

[Lee et al., 2018] Lee, N., Ajanthan, T., and Torr, P. H. S. (2018). SNIP: single-shot network pruning based on connection sensitivity. *CoRR*, abs/1810.02340.

[Li et al., 2018] Li, X., Chen, S., Hu, X., and Yang, J. (2018). Understanding the disharmony between dropout and batch normalization by variance shift. *arXiv:1801.05134 [cs, stat]*. arXiv: 1801.05134.

[Liu et al., 2019a] Liu, J., Xu, Z., Shi, R., Cheung, R. C. C., and So, H. K. H. (2019a). Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers.

[Liu et al., 2019b] Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. (2019b). Rethinking the value of network pruning. *7th International Conference on Learning Representations, ICLR 2019*, page 1–21. arXiv: 1810.05270 Citation Key: Liu2019a.

[Mikler, 2021] Mikler, S. (2021). Reproducibility study: Comparing rewinding and fine-tuning in neural network pruning. *arXiv:2109.09670 [cs]*. arXiv: 2109.09670.

[Mostafa and Wang, 2019] Mostafa, H. and Wang, X. (2019). Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. *arXiv:1902.05967 [cs, stat]*. arXiv: 1902.05967.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. page 8.

[Nwankpa et al., 2018] Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *arXiv:1811.03378 [cs]*. arXiv: 1811.03378.

[Paszke et al., 2019] 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.

[Patterson et al., 2021] Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., and Dean, J. (2021). Carbon emissions and large neural network training. *arXiv:2104.10350 [cs]*. arXiv: 2104.10350.

[Renda et al., 2020] Renda, A., Frankle, J., and Carbin, M. (2020). Comparing Rewinding and Fine-tuning in Neural Network Pruning. arXiv: 2003.02389.

[Rumelhart et al., 1985] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). *Learning Internal Representations by Error Propagation:*.

[Russakovsky et al., 2015] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv: 1409.1556 Citation Key: Simonyan2014.

[Smith, 2017] Smith, L. N. (2017). Cyclical learning rates for training neural networks.

[Tan and Le, 2019] Tan, M. and Le, Q. V. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. *36th International Conference on Machine Learning, ICML 2019*, 2019-June:10691–10700. arXiv: 1905.11946 Citation Key: Tan2019 ISBN: 9781510886988.

[Zagoruyko and Komodakis, 2016] Zagoruyko, S. and Komodakis, N. (2016). Wide residual networks. In Richard C. Wilson, E. R. H. and Smith, W. A. P., editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press.

[Zhou et al., 2020] Zhou, H., Lan, J., Liu, R., and Yosinski, J. (2020). Deconstructing lottery tickets: Zeros, signs, and the supermask. *arXiv:1905.01067 [cs, stat]*. arXiv: 1905.01067.