The Lottery Ticket Hypothesis: On Sparse, Trainable Neural Networks

by

Jonathan Frankle

B.S.E., Princeton University (2014)
M.S.E., Princeton University (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

Author . . . .
Department of Electrical Engineering and Computer Science
January 23, 2023

Certified by . . . .
Michael Carbin
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

The Lottery Ticket Hypothesis: On Sparse, Trainable Neural Networks

by

Jonathan Frankle

Submitted to the Department of Electrical Engineering and Computer Science
on January 23, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis, I show that, from an early point in training, typical neural networks for computer vision contain subnetworks capable of training in isolation to the same accuracy as the original unpruned network. These subnetworks—which I find retroactively by pruning after training and rewinding weights to their values from earlier in training—are the same size as those produced by state-of-the-art pruning techniques from after training. They rely on a combination of structure and initialization: if either is modified (by reinitializing the network or shuffling which weights are pruned in each layer), accuracy drops.

In small-scale settings, I show that these subnetworks exist from initialization; in large-scale settings, I show that they exist early in training ($< 5\%$ of the way through). In general, I find these subnetworks when the outcome of optimizing them becomes robust to the sample of SGD noise used to train them; that is, when they train to the same convex region of the loss landscape regardless of data order. This occurs at initialization in small-scale settings and early in training in large-scale settings.

The implication of these findings is that it may be possible to prune neural networks early in training, which would create an opportunity to substantially reduce the cost of training from that point forward. In service of this goal, I establish a framework for what success would look like in solving this problem and survey existing techniques for pruning neural networks at initialization and early in training. I find that magnitude pruning at initialization matches state-of-the-art performance for this task. In addition, the only information that existing techniques extract are the per-layer proportions in which to prune the network; in the case of magnitude pruning, this means that the only signals necessary to achieve state-of-the-art results are the per-layer widths used by variance-scaled initialization techniques.

Thesis Supervisor: Michael Carbin
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

The research in this thesis was conducted under the supervision of Prof. Michael Carbin, who was willing to take a risk on a crazy idea from a second-year PhD student who had never done machine learning research. His openmindedness and diligent mentorship are the reasons this thesis and my subsequent career came to be.

All of the research in this thesis was conducted in collaboration with Michael Carbin. The research for Chapters 4 and 5 in this thesis was also conducted in collaboration with Gintare Karolina Dziugaite and Daniel Roy. In particular, the idea to prune neural networks early in training rather than at initialization (Section 4.2) was developed collaboratively with Karolina, and the idea to use linear mode connectivity as the means of comparing pairs of networks trained from the same starting point with different random seeds (Section 4.1) was Karolina's. Both of these ideas were published in our work on *Linear Mode Connectivity and the Lottery Ticket Hypothesis* (Frankle et al., 2020a). Karolina's intellectual contributions to the research in this thesis were essential to its development.

This thesis includes ideas from several other papers I worked on that are not directly covered in this thesis, including research conducted with:

- Alex Renda and Michael Carbin on the efficacy of the IMP procedure (Algorithms 1 and 4) as a general-purpose pruning technique, which serves as the basis for claims in this thesis that magnitude pruning remains at or near the state-of-the-art in unstructured pruning. (Renda et al., 2020)

- Ari Morcos and David Schwab on a variety of experiments exploring the properties of mtaching subnetworks early in training, including the difficulty of training from anything but the exact initialization provided to the subnetworks and the efficacy of self-supervised pre-training to create good subnetwork initializations (Frankle et al., 2020b).

- Davis Blalock, Jose Javier Gonzalez Ortiz, and John Guttag on a survey of the state of the pruning literature as of a couple of years ago. Davis and Jose's

incisive scientific critiques of the pruning literature are woven into the fabric of the narrative and technical aspects of this thesis (Blalock et al., 2020).

- Tianlong Chan, Shiyu Chang, Sijia Liu, Yang Zhang, Atlas Wang, and Michael Carbin, whose collaboration on finding *transferrable* subnetworks in BERT-pretrained networks paved the way for broader claims about the value of winning ticket subnetworks (showing that they're reusable even if they're expensive to find), their presence in natural language processing settings, and how having a good initialization (in this case, a vast amount of self-supervised pre-training on langauge tasks) makes them even easier to find (Chen et al., 2020a).

- Tian Jin, Michael Carbin, and Dan Roy as supervised by Gintare Karolina Dziugaite, which provided a range of insights into why neural network pruning leads to improved generalization, taking the pruning literature back to its original motivation as a generalization-improvement technique, ostensibly via capacity control (Jin et al., 2022).

The research in this thesis required enormous computational resources. At the conclusion of the writing process, I counted 1.1 million saved neural network checkpoints comprising the experiments in this thesis and related research. I am indebted to IBM which, through the MIT-IBM Watson AI Lab, provided access to hundreds of V100s, P100s, and K80s early in this project, despite the fact that it had nothing to do with the original proposal under which the resources were made available, that I racked up large bills in the process, and that I took down the underlying infrastructure (including one incident involving halting transatlantic data traffic). Thank you to David Cox, Mark Weber, and John Cohn at the MIT-IBM Watson AI Lab for their steadfast support.

I am equally indebted to Google which, through the TPU Research Cloud (TRC) program, provided access to hundreds of TPUs for years on end (much longer than the 60 days I was originally promised) that made it possible to scale up the lottery ticket experiments. Thank you especially to Zak Stone, who administers the program and made these resources available to me.

Computational support for this thesis work was also provided in the form of Google Cloud credits from the MIT Quest for Intelligence.

Support for my work during my PhD was provided in-part by:

- A Stata Presidential first-year fellowship, which funded part of my PhD.
- The Internet Policy Research Initiative, which funded part of my PhD.
- ONR award 6937576 (PERISCOPE: Perceptual Reprsentations for Actions, Composition, and Verification), which funded part of my PhD.
- IBM through the MIT-IBM Watson AI Lab, which funded part of my PhD.
- The Alfred P. Sloan Foundation through a Sloan Research Fellowship, which funded part of my PhD.
- Google, which funded me as a summer intern twice (2017 and 2018) and as a part-time researcher during the school year (2018-2019).
- Facebook, which funded me as a summer intern (2019) and as a part-time researcher during the school year (2019-2020).

I am grateful to my colleagues at MosaicML, who patiently allowed me to spend time completing this thesis during my working hours.

Finally, thank you to all of the family, friends, and colleagues whose support and patience through the many ups and downs of this process made it possible for me to reach the end successfully. You know who you are and how essential you were.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Oxford English Dictionary defines pruning as follows:[1]

Reducing the extent of [a neural network] by removing superfluous or unwanted parts.

This thesis is about neural network pruning, and the Oxford English Dictionary provides a useful working definition for what pruning a neural network entails. The superfluous or unwanted parts of a neural network are its parameters, which can be removed individually (e.g., in the form of connections between nodes) or in groups. In this thesis, I explore whether it is possible to prune neural networks early in training and use the resulting, smaller networks in place of the original, unpruned networks.

1.1 Terminology

The pruning literature dates back to the late 1980s, and may different terms have been used to describe methods by which neural networks are pruned and the effect that pruning has on networks. (See Chapter 2 for a lengthy history of this literature as relevant to this thesis.) To establish consistent language for this thesis in line with the contemporary literature, I will use the following terms to describe neural network pruning: A neural network consists trainable *parameters*, many of which take the form of *connections* between *nodes* or *neurons*. Each parameter has a value (e.g., the strength of a connection) specified by its *weight*. The *magnitude* of a

[1]With a small modification on my part.

weight is the weight's absolute value; for brevity, I will sometimes refer to this as the magnitude of a parameter or connection as well. *Pruning* is the process of modifying a neural network by removing parameters (concretely, by permanently fixing the weights of these parameters to 0) and possibly changing the values of the weights of the remaining parameters. There are a number of different *pruning methods*, *pruning techniques*, or *pruning algorithms* (these terms used interchangeably) that specify how to go about removing parameters from the network and setting the weights of the remaining parameters. Many of those—including the instantiation of *magnitude pruning* in this thesis—involve pruning according to a *heuristic* or *saliency metric* and afterwards training the pruned network further (called *re-training* or *fine-tuning*) to recover accuracy that may have been lost when parameters were pruned.

1.2 Motivation

Throughout the decades, the pruning community has been interested in pruning for a wide variety of reasons, ranging from capacity control to prevent overfitting to efficient deployment of trained networks on edge devices. (For a detailed history of these motivations, see Section 2.2.) In this thesis, I am interested in two motivations in particular:

1. Reducing the cost of training neural networks by eliminating parameters and—thereby—the operations required to use and update those parameters.
2. Understanding how neural networks learn and the conditions necessary for them to learn effectively.

Motivation: more efficient training. There is a growing body of work (of which this thesis is a part) focused on using pruning as a tool to make training more efficient. The motivation for this research is that training modern neural networks is expensive by any standard (time, cost, or energy) and that it is growing more expensive at a fast rate. Sevilla et al. (2022) estimate that the number of floating point operations necessary to train the largest scale neural networks has doubling every 10 months since 2015 and every 6 months since the advent of modern deep learning in 2010. Pruning offers the possibility of reducing the number of operations—and potentially the cost—of training.

Using pruning to reduce the cost of training is a newer topic, with the first research (including the work contained in this thesis) released in 2017-2018 and published shortly thereafter (Mocanu et al., 2018; Lee et al., 2019; Frankle & Carbin, 2019). As far as I can determine, the reason that interest in the topic began only recently is that many in the neural network research community believed that it was difficult to use pruning to substantially reduce the cost of training. One of the main contributions of the thesis that follows is to demonstrate that it is indeed possible to train the kinds of neural network architectures that arise from pruning. Specifically, from a point early in training (and, in some cases, at the beginning of training), standard networks can be pruned and still train to the same accuracy as the unpruned network.

Motivation: understanding how practical neural networks learn. Although the work I present in this thesis has practical motivations and implications, I have also conducted this research with the goal of understanding how practical neural networks learn in the real world. We have many theoretical ideas about how neural networks might learn in general, their capabilities as universal function approximators, and why they generalize so effectively. However, the neural networks that we train in practice are a very small subset of all possible neural networks with peculiar properties that defy explanation with existing theory. As I describe in more detail in Section 1.5, I believe that our existing theory is too limited a means by which to understand and improve the large-scale neural networks that we train train in practice on real-world datasets. As such, it is vital that we study these networks empirically, developing and testing beliefs about how they operate much as a neuroscientist, physicist, or other natural scientist might.

In service of this goal, pruning provides a window into many important aspects of how neural networks learn in practice. In this thesis, I will use pruning as a tool to understand:

- The capacity necessary for a neural network to learn a function.
- How that capacity relates to the capacity necessary for a neural network to represent a function.
- The role of initialization in neural network training.
- How neural networks traverse the loss landscape in pursuit of an optimum.
- How these behaviors interlink and the broader picture that emerges.

Intuitively, pruning makes it possible to strip away parts of a neural network in order to improve our understanding of what it really needs to learn or represent a function. It also creates a scarcity of capacity, pushing neural networks into extreme regimes that exaggerate and bring to light important behaviors that occur in general but might otherwise go unnoticed. The results in this thesis are a testament to the efficacy of pruning, not just as a step toward practical improvements in efficiency, but as a scientific tool for understanding and improving complex deep learning systems that are increasingly a part of our daily lives.

1.3 Research Questions and Answers

The central research question that I address in this thesis is as follows:

Under what circumstances would it possible to train a sparse neural network (like that uncovered by pruning after training) in place of a standard dense network?

Embedded within this high-level question are several more specific ones:

What are the sparsest networks that it is possible to train? How do they compare to other benchmarks (such as the sparsity attainable by pruning after training)? How can we relate the capacity necessary for a neural network to represent a function and the capacity necessary for it to learn it?

How many steps are necessary to train these networks successfully (relative to the steps necessary to train the corresponding dense networks)?

What kinds of sparsity patterns and initializations are sufficient for these networks to train successfully?

What properties do these sparse networks have?

This thesis will address these questions from several perspectives. The answer, in short, is as follows:

Neural networks in small-scale, computer vision settings have sparse subnetworks at initialization that can train to full accuracy in the same number of steps as the original network. These networks are about as sparse as those found by standard methods for pruning after training. I show that this is the case by developing a procedure to retroactively uncover such subnetworks after training the full network. The subnetworks I find are sensitive to their initialization, and I only find them to be capable of attaining this performance when each parameter is set back to the value it received at initialization. I refer to the possibility that this observation holds in general as the "lottery ticket hypothesis:" that these subnetworks won the initialization lottery with lucky combinations of initial weights that allowed them to train successfully and that such subnetworks might exist at initialization within many practical neural networks. (Chapter 3)

Neural networks in larger-scale computer vision settings have sparse subnetworks early in training that can complete training and reach full accuracy in the same number of steps as the original network. These networks are about as sparse as those found by standard methods for pruning after training. (There is no evidence that such subnetworks exist even earlier or at initialization, but my results do not rule out this possibility.) The subnetworks I find are again sensitive to the specific values to which parameter of their parameters is set, and I only find subnetworks that are capable of attaining this performance when each parameter is set back to the value it received at the step where the subnetwork was created. (Chapter 4)

The discrepancy between small-scale settings (where such subnetworks can be found at initialization) and larger-scale settings (where such subnewtorks can be found *only after some amount of training) appears to relate to the sensitivity of the subnetworks to SGD noise. In particular, the point in training at which my procedure finds high-accuracy subnewtorks is the same point at which such subnetworks will always reach the same, linearly connected region of the loss landscape regardless of the data order (i.e., the sample of SGD noise) under which they are trained. In small-scale settings, this occurs at initialization; in larger-scale settings, this occurs only after some amount of training. (Chapter 4)*

Although I have established that such subnetworks exist, finding them efficiently at the earliest moments that they are known to come into existence remains a work in progress. Several recently proposed methods for pruning at initialization successfully outperform random pruning, but they do not attain the performance of the subnetworks that I found in the preceding chapters using my retroactive procedure. Moreover, simply performing magnitude pruning at initialization is a state-of-the-art method among this class of methods. In a departure from the preceding results, the sparse subnetworks produced by all of these methods at initialization are sensitive to neither their initializations nor their layerwise sparsity patterns: it is possible to randomly reinitialize the parameters or randomly shuffle which parameters are pruned by these methods within each layer and maintain the (lower-than-baseline) accuracy. Accuracy and sensitivity to reinitialization and shuffling increase as these methods are applied later in training, but the methods do not match the accuracy of the networks found retroactively until much later in training than in the preceding chapters. This raises the question of whether lower accuracy and insensitivity to reinitialization and shuffling are flaws in existing methods or intrinsic properties of pruning early in training that will make it difficult to find desirable subnetworks efficiently in practice. (Chapter 5)

1.4 Challenges

In this thesis, I demonstrate that it is indeed possible to prune neural networks early in training. However, many challenges remain when it comes to turning these findings into practical methods for reducing the cost of training.

Several of these challenges are common to any practical application of pruning, not just those that aim to reduce the cost of training. In order to actually make using the network more efficient via pruning, pruning must occur in patterns conducive to acceleration on existing hardware and underlying software must be written to ensure the hardware actually accelerates those operations. I do not study these questions in this thesis. Rather, I focus on the mere fact that training pruned networks is possible in the first place, which has opened the door for the growing community that studies pruning during training to explore these practical considerations.

Pruning to reduce the cost of training also faces specific challenges. The most significant is that we do not yet know how to prune early in training without reducing the accuracy of the trained network. In this thesis, I show empirical evidence that, early in training or at initialization, unpruned neural networks contain subnetworks capable of completing training and still reaching full accuracy. I find these subnetworks by first training the unpruned network to completion, pruning at that point, and then considering the counterfactual where we had known early in training that those weights would eventually be pruned. This is not an efficient strategy for finding pruned, trainable networks early in training; the unpruned network must still be trained to completion first before pruning can retroactively occur. There is now a vibrant research literature on pruning early in training, and I describe the current state of this research in Chapter 5.

Despite these remaining challenges, the significance of the work I present in this thesis are that these research questions exist as popular research topics today. Without showing that pruned networks can train on their own to full accuracy, it would make no sense to study strategies to prune during training in the first place.

1.5 Research Philosophy

The preceding section states the substantive, technical findings that this thesis will support with experiments and data. This research reflects a broader epistemological perspective on the nature of scientific knowledge in deep learning.

Consider the following metaphor: We know the laws of physics. There are many possible outcomes that might emerge from these laws. One of those outcomes is biology as it exists on Earth. It would be difficult to predict this particular outcome from knowledge of the laws of physics alone. The probability of this particular outcome is infinitesimal, and this outcome contains emergent behavior derived from complexity so vast that it would be nearly impossible to deduce from first principles.

I contend that understanding deep learning is, in this way, like understanding biology.[2] Given the structure of neural networks and the theoretical nature of stochastic gradient descent, many outcomes are possible. All but an infinitesimal portion of those outcomes are uninteresting and worthless. The wrong learning rate or leaving out batch normalization invites catastrophe when training a modern neural network.

Just as biology is a rare but—as a human being on Earth—vitally important outcome of the laws of physics, the neural networks that we use in practice are a rare but vitally important subset of all possible neural networks that we might train. Just as biology involves the study of patterns and motifs that appear time and again in related forms rather than mathematical deduction from first principles, the neural networks that we use in practice are difficult (perhaps impossible) to describe or reason about formally. Just as the laws of physics are often too low-level of an abstraction to efficiently or productively reason about biology, our existing theory can sometimes be too low-level of an abstraction to efficiently or productively reason about the unexpected phenomena that emerge out of the enormously complex neural networks that are productive in practice on real data.

In short, both biology and the neural networks we use in practice are specific, complex outgrowths of simpler systems whose fundamental dynamics we largely un-

[2]Disclaimer: I am not a biologist and the last time I studied biology was in 9th grade.

derstand. Although they are not general cases of those underlying systems, they are cases of paramount practical importance. These cases appear especially resistant to formalization due to their complexity. When it comes to reasoning about their behavior, I see empiricism as a tool as powerful for deep learning as it is for biology.

This thesis embodies this empirical perspective on the nature of scientific knowledge and progress in deep learning. The lottery ticket hypothesis is just a hypothesis. It is not a general theorem about all neural networks, but rather a statement about behavior we observe in practical neural networks trained on real data using hyperparameters that lead to state-of-the-art outcomes. The original statement of the lottery ticket hypothesis—that practical neural networks contain sparse, trainable subnetworks at random initialization (Frankle & Carbin, 2019)—described the original lottery ticket results, but I could find no evidence that it generalized to larger-scale settings. The hypothesis needed revisions to accommodate new evidence (Frankle et al., 2020a), and a more general hypothesis emerged about the relationship between sparse networks found by magnitude pruning and their sensitivity to SGD noise. This is the empirical process of scientific progress at work.

Nowhere in this thesis will you find a theorem or a proof. In order to make formal statements about even the most basic phenomena described in this thesis, Malach et al. (2020) and Pensia et al. (2020) had to radically simplify the problem, eliminating training entirely and reasoning only about the subnetworks of randomly initialized networks that never get optimized. In my view, the fact that these subnetworks are trainable is the most fascinating part of the phenomenon, and these papers—while an important scientific step—must sacrifice the heart of the problem in service of making it formalizable.

To my collaborators and colleagues who began their careers in the natural sciences, the belief that empiricism is a valid and valuable source of scientific knowledge goes without saying. To my collaborators and colleagues who began their careers in computer science, this belief sometimes produces angst. In computer science, we are spoiled that the foundation of much of our scientific knowledge takes the form of proofs—of information that we know to be true with absolute certainty. Aside from areas of computer science that interact with the outside world (privacy, HCI, web measurement) and those that focus on real world performance (applied systems and computer architecture research), empiricism is not a common foundation for knowledge. In my view, establishing a rigorous science of deep learning requires that we broaden our views of what constitutes knowledge in computer science.

Theoretical research is an essential part of the scientific journey toward greater understanding of neural networks, but I contend that it is only one piece of a larger portfolio of approaches within which empiricism will play a central role. This thesis is not only as a statement about a specific neural network phenomenon, but also a case study of the important contributions empiricism can make to scientific progress in deep learning.

Chapter 2

Background on Neural Network Pruning

2.1 A Brief History of (Magnitude) Pruning

The idea of pruning neural networks dates back to time immemorial (in particular, 1989) and has been restated and repopularized time and again over the intervening waves of interest in neural networks. The earliest work I have found on pruning neural networks according to a heuristic (rather than doing so randomly) is that of Janowsky (1989). Although his terminology differs from how we commonly describe these concepts today and his neural networks are optimized differently than contemporary ones (i.e., without backpropagation), Janowsky describes—in essentially its contemporary form—a pruning algorithm that we refer to now as *magnitude pruning*. As Janowsky said:

> *In what we call 'clipping,' weaker bonds are removed prior to strong bonds...So, for example, in a network that was clipped at 50%, the smallest (in absolute value) half of the bonds would all be set to zero.*

Janowsky recognized that pruning the network outside of the optimization process could reduce its accuracy. After all, pruning sets parameters to zero, and the function represented by the neural network after this process will be different from the one that was found by the initial training process. In practice, this generally reduces both the training accuracy of the network and its ability to generalize to unseen data from the same distribution. To address this deleterious side-effect of pruning, Janowsky proposed to train the network further after pruning to recover accuracy, something that is a staple of modern pruning algorithms.

> *Clipping gives us a means of choosing which bonds should be deleted, but what about the remaining bonds? There is no a priori reason why their initial values should remain optimal after the clipping process. Perhaps more learning could take place on the subset of unclipped bonds, in a sense 'tweaking' their values to perform better.*

Despite the thousands of papers on neural network pruning that have been published in the decades since, this algorithm—training a neural network, pruning the connections whose weights have the lowest magnitudes, and further training to recover lost accuracy—remains among the state-of-the-art pruning methods today (Han et al., 2015; Gale et al., 2019; Renda et al., 2020). This pruning algorithm serves as the main scientific tool in this thesis.

Neural network pruning was popular in the late 1980s and early 1990s: LeCun et al. (1990) described the more general pattern that captures many pruning algorithms used today—a generalized version of the algorithm of Janowsky parameterized over an arbitrary *saliency metric* for determining which weights to keep and remove:

> *A simple strategy consists of deleting parameters with small 'saliency,' i.e., those whose deletion will have the least effect on training error...After deletion, the network should be retrained.*

LeCun et al. also recognized that, for best results, pruning algorithms of this form should be repeated, gradually whittling away the network rather than pruning to the target level of sparsity in a single shot.

> *Of course, this procedure can be iterated.*

The concept of iterative pruning—alternating the pruning and re-training steps in the process of gradually reducing the size of the network—is another mainstay of modern pruning strategies, and it also features prominently in this thesis.

Any saliency metric can be inserted into this general algorithm. LeCun et al. mention magnitude pruning in passing (*other things being equal, small-magnitude parameters will have the least saliency*) before proposing a heuristic of their own:

> *[We] move beyond the approximation that 'magnitude equals saliency' and...use[] the second derivative of the objective function with respect to the parameters.*

A note on terminology. The prior work described above has used a variety of different terms to refer to similar concepts. Refer to Section 1.1 in the previous chapter for the terminology used in this thesis, which reflects the terminology used in the contermporary literature at the time of writing this thesis.

By 1993, the topic of neural network pruning became so popular that it merited a survey paper (Reed, 1993). However, a multi-decade lull followed thereafter (mirroring a similar decline in interest in neural networks). In 2015, as the current wave of interest in neural networks was gathering momentum, the idea of pruning neural networks was re-popularized by Han et al. (2015). The procedure of Han et al. follows the familiar refrain of magnitude pruning:

> *Our pruning method...learn[s] the connectivity via neural network training...The second step is to prune the low-weight connections. All connections with weights below a threshold are removed from the network...The final step retrains the network to learn the final weights for the remaining sparse connections. This step is critical. If the pruned network is used without retraining, accuracy is significantly impacted.*

Han et al. also recognized the importance of pruning iteratively.

> *Learning the right connections is an iterative process. Pruning followed by a retraining is one iteration, after many such iterations the minimum number of connections could be found. Without loss of accuracy, this method can boost pruning rate from 5x to 9x on AlexNet compared with single-step aggressive pruning.*

The main finding of the paper was that this pruning algorithm worked quite well on then state-of-the-art benchmarks for computer vision. It reduced the parameter-counts of convolutional networks like LeNet (LeCun et al., 1998) for MNIST (LeCun

& Cortes) and AlexNet (Krizhevsky et al., 2012) and VGG (Simonyan & Zisserman, 2015) for ImageNet (Deng et al., 2009) by about an order of magnitude without hurting test accuracy.[1]

The work of Han et al. marked the birth of the contemporary pruning literature, sparking lines of work on different pruning heuristics, different templates for structuring pruning algorithms (e.g., Molchanov et al., 2017; Zhu & Gupta, 2018; Louizos et al., 2018), and the research described in this thesis. Even among the dozens (if not hundreds) of pruning algorithms that have been proposed in recent years, iterative magnitude pruning remains a standard starting point and a competitive baseline. Recent studies by Gale et al. (2019) and Renda et al. (2020) found that magnitude pruning remains at or near the state of the art in connection pruning and represents a sweet spot in the tradeoff between performance and ease of implementation.

This brief summary of the history of neural network pruning (with an emphasis on magnitude pruning) suffices as context for the thesis that follows. Rather than comprehensively survey three and a half decades of pruning literature myself (which, as Hoefler et al. below found, requires a thesis-length document in its own right), I direct readers interested in the topic to two recent survey papers that have done so very effectively. Hoefler et al. (2021) exhaustively survey the the neural network pruning literature, including common templates that pruning algorithms follow, different pruning heuristics used to instantiate those templates, more recent ideas like dynamic pruning, and the general performance to be expected from pruning algorithms on modern benchmarks. This survey spans 90 pages and cites hundreds of pruning papers dating from the 1980s to the present. Blalock et al. (2020) survey

[1]A neural network can only be pruned so far before pruning starts to affect the network's accuracy. After all, in the limit, a neural network with no parameters remaining cannot learn anything. As the extent of pruning increases, a neural network typically maintains full accuracy to a certain point, after which accuracy begins to drop precipitously. See Figure 3-3 for an example of this phenomenon on LeNet on MNIST. Interestingly, test accuracy often *increases* slightly under moderate levels of pruning, a phenomenon that was a central motivation for pruning in the 1980s (see Section 2.2) and remains of scientific interest today (Bartoldson et al., 2020; Jin et al., 2022). It appears without remark in the results of Han et al. (2015) and in many pruning papers in the modern literature. In my experience, the generalization improvements on well-tuned state-of-the-art networks on large-scale tasks are minimal, so I will not emphasize this phenomenon in this thesis. I do not recommend relying on this phenomenon to improve generalization in any practical context.

pruning papers over the past decade, taking a critical look at the scientific quality of research on pruning, expressing concern about the extent to which the field is genuinely making progress, and making useful suggestions for future pruning research.

2.2 Why Prune?

Why is the research community interested in pruning neural networks in the first place? That motivation has changed substantially over the decades, with early interest in improving generalization, modern interest in reducing the cost of inference, and more recent interest (of which this thesis is an example) in reducing the cost of training and in understanding the science of how neural networks learn in practice.

Historical motivation: improved generalization. The apparent motivation for the earliest neural network pruning research was a concern about overfitting. This concern reflects a classical understanding of the bias-variance tradeoff in machine learning. According to this perspective, when model capacity is too small, the model will fail to generalize because it is unable to adequately fit the training data. Conversely, when model capacity is too large, the model will fail to generalize because it will memorize the training data. LeCun et al. (1990) described the understanding of the bias-variance tradeoff at the time as follows:

> *It is known from theory and experience that, for a fixed amount of training data, networks with too many weights do not generalize well. On the other hand, networks with too few weights will not have enough power to represent the data accurately. The best generalization is obtained by trading off...error and network complexity.*

Even in 1990, neural networks with dozens of parameters were considered especially large and, therefore, prone to overfitting (as preposterous as this may seem by the standards of today's nearly trillion parameter models):

> *As the number of parameters in the systems [of neural networks] increases, overfitting problems may arise, with devastating effects on the generalization performance.*

Pruning was seen as a form of capacity control to reduce overfitting.

One technique to reach this [best generalization] tradeoff is to minimize a cost function composed of two terms: the ordinary training error, plus some measure of network capacity...A time-honored (albeit inexact) measure of complexity is simply the number of non-zero free parameters.

Three years later, Reed (1993) echoed this understanding of the purpose of pruning in his survey of the pruning literature. Figure 1 on the front page of the paper is the classic diagram of the relationship between training time and train/test error present in any machine learning textbook: training error continually decreases over time, while test error decreases and then increases, reflecting initial learning and eventual overfitting. As Reed puts it in the introduction to the survey:

A rule of thumb for obtaining good generalization is to use the smallest system that will fit the data...The approach taken by algorithms in this paper is to train a network that is larger than necessary and then remove parts that are not needed. The large initial size allows the network to learn reasonably quickly...while the reduced complexity of the trimmed down system favors improved generalization.

Improving generalization and mitigating overfitting are no longer the primary motivation for pruning today. Instead, it is typically to minimize the resources necessary to use or create the network by reducing the number of operations that take place during forward and backward propagation (see the following section). However, the interplay between pruning and generalization remains an open and scientifically rich question. Our current understanding of the relationship between model capacity and generalization in deep learning is more complicated than the classical model of the bias-variance tradeoff described by LeCun et al. and Reed. Work on double descent shows that increasing the capacity available to a neural network counter-intuitively leads to *better* generalization (Belkin et al., 2019; Nakkiran et al., 2020).

One could speculate about several different ways that pruning might affect generalization in the context of this richer picture of the bias-variance tradeoff. At the heart of this relationship between pruning and generalization is a question of whether the sparse capacity of a network to which pruning has been applied works in the same manner as the dense capacity of changing the model's width or depth. That is to say, it is possible that removing parts of the network at a smaller granularity (i.e., individual parameters or groups of parameters) and doing so in an informed way (i.e., using a pruning heuristic) may have a different effect on the network than changing its width or depth at the beginning of training. One possible mechanism for any differences in behavior could be that pruning individual connections may leave many or all of the network's nodes intact (albeit with fewer inputs), reducing the number of parameters in the network but not the number of hidden features in the network. Another possible mechanism could be that pruning occurs in an informed way according to a heuristic, potentially removing parameters that are less important to the function that the network has learned to represent as compared to parameters chosen at random.

In practice, we see slight generalization improvements from pruning neural networks to a moderate extent (e.g., Han et al., 2015). This might suggest that pruning a neural network at certain granularities and with certain heuristics does induce a classical bias-variance tradeoff that does not occur when changing model width or depth. However, Bartoldson et al. (2020) contend that these generalization improvements are actually due to the regularization effect of "injecting noise" into the training process by pruning (rather than due to capacity control); this intervention may behave differently than changing the capacity of the network by altering its width or depth, potentially leading to a different relationship with generalization.

Jin et al. (2022) also cast doubt on the idea that pruning improves generalization via capacity control. This paper (of which I am a co-author) involves comparing pruning a neural network with iterative magnitude pruning to following the exact same training schedule (extended training with a cyclic learning rate) but without the pruning. The generalization results are nearly identical, indicating that the training schedule—rather than pruning itself—may be responsible for improved generalization.

From another point of view, perhaps pruning a neural network simply uncovers the capacity that the neural network was actually using. In other words, perhaps neural networks only use a fraction of the parameters available in their dense configurations, and pruning in a sparse fashion eliminates unused parameters, thereby revealing the actual capacity that the networks are using. From this perspective, pruning should not affect the tradeoff that a network occupies between bias and variance; rather, it should only affect our estimate of the actual capacity that a neural network needs to occupy that particular tradeoff.

In summary, although the generalization-centric motivation for pruning is an artifact of the early 1990s and other, efficiency-centric motivations dominate today, the relationship between pruning and generalization is still an open scientific question (perhaps even more so than in the 1990s) that merits further inquiry.

Modern motivation: efficient deployment and inference. Since 2015, pruning research has largely been motivated as a way to make neural network deployment and inference more efficient. This trend began with Han et al. (2015), who open their paper by calculating the power requirements for memory accesses when performing inference on a 1B parameter network. They conclude that this would be "well beyond the power envelope of a typical mobile device." They express two specific motivations for using pruning. The first is to inference itself more efficient:

> *Our goal in pruning networks is to reduce the energy required to run such large networks so they can run in real time on mobile devices.*

The second is to reduce the space required to store the network:

> *The model size reduction from pruning also facilitates storage and transmission of mobile applications incorporating DNNs.*

When seeking to improve the real-world efficiency of inference, it matters not only how many parameters are pruned, but also which ones are pruned and in what patterns they are pruned. For example, modern neural network accelerators (like GPUs and TPUs) typically operate on blocks of weights, multiplying vectors or matrices. Although pruning individual connections in an unstructured manner may lead to hypothetical reductions in the number of floating-point operations required to perform inference, doing so will not necessarily lead to real-world speedup on hardware designed in this way. Even Han et al. implicitly recognized this shortcoming of the unstructured magnitude pruning algorithm they proposed; they clarify that:

We...target[] our pruning method for fixed-function hardware specialized for sparse DNN, given the limitation of general purpose hardware on sparse computation.

In pursuit of speedups on general-purpose neural network accelerators, research that followed Han et al. focused on the patterns in which weights were pruned. For example, papers focused on pruning entire channels from convolutional networks (e.g., Li et al., 2017b; He et al., 2017; Luo et al., 2017; Liu et al., 2017; He et al., 2018a) or attention heads from transformer networks (e.g., Michel et al., 2019). Both of these approaches reduce the size of tensors rather than adding zeros within tensors of the same size. For example, eliminating a convolutional channel reduces the number of convolutional filters in the previous layer and removes one kernel from each filter in the same place in the subsequent layer, both of which result in dense tensors that have a smaller size in the corresponding dimensions. From the perspective of the hardware, it makes it possible to perform the same operation on a smaller tensor, naturally making the operation faster.

These pruning approaches come with a drawback: by requiring greater structure in how pruning occurs, they restrict the number of possible pruning patterns to reach a particular sparsity level. Typically, this means that the network cannot be pruned to as great of an extent before pruning begins to reduce accuracy (Liu et al., 2019).[2]

A popular strategy for navigating this tradeoff has been to choose a middle ground in which weights are pruned in *blocks* that are larger than individual weights (one extreme) but too small to reduce the size of any of a tensor's dimensions (the other extreme). For example, Elsen et al. (2020) prune in contiguous rows of 16 weights from 1x1 convolutions; Elsen et al. determine that doing so provides the best tradeoff between network accuracy and training speed on the mobile ARM processor they target. Elsen et al. find that—with appropriate block sizes/shapes and software

[2]Liu et al. perform a detailed comparison of several structured pruning methods for convolutional networks, and this tradeoff of using structured pruning schemes is on full display. In addition to a variety of structured pruning techniques, they also include numbers for the unstructured magnitude pruning scheme of Han et al. (2015) in Table 6. Accuracies are much higher for the unstructured method of Han et al. than for the structured pruning strategies. For example, when 60% of weights have been pruned from ResNet-50 on ImageNet, the unstructured magnitude pruning scheme maintains full test accuracy. All other pruning schemes in the paper for which a comparison is possible lose one or more percentage points of accuracy with the same number of weights remaining.

kernels—block sparsity can provide 1.3-2.4x improvements in inference wall-clock time. Similarly, Gray et al. (2017) produce CUDA kernels that make it possible to accelerate block-sparse neural networks with block sizes between 8x8 and 32x32 on the NVIDIA Pascal-based accelerators.

One of the biggest challenges of using block sparsity is that the choice of block size and shape is often context-specific. For example, Elsen et al. justify the choice of rows of 16 contiguous weights because:

> *It corresponds to one cache line [on the processor they target]...[T]his allows each strip of 16 spatial locations in the activations to remain in the L1 cache until it is no longer needed.*

The best choice here might differ on another chip, or even on a different version or generation of the same chip. The same is true for the work of Gray et al., which was conducted before the advent of tensor cores in NVIDIA accelerators. In short, block sparsity offers a wider space of tradeoffs between real-world speedup and accuracy, but finding and exploiting the sweet spot in this design space is an undertaking that requires bespoke engineering specific to both the neural network architecture and the target hardware.

Motivation in this thesis: more efficient training. In recent years, the ambitions of pruning researchers have extended beyond making inference efficient on models that have already been trained. Pruning already occurs during training in some pruning methods, implicitly opening the door to accelerating training as well as inference even if the methods themselves are designed to reduce inference costs. For example, Zhu & Gupta (2018) performed an extensive study of *gradual pruning methods* that reach the target parameter-count by the end of training by performing magnitude pruning over training according to a schedule. The goal of this approach was to eliminate a separate fine-tuning phase and reduce the cost of preparing a network for efficient inference.

Using pruning to reduce the cost of training is a new topic, with the first research (including the work contained in this thesis) released in 2017-2018 and published shortly thereafter (Mocanu et al., 2018; Lee et al., 2019; Frankle & Carbin, 2019). As far as I can determine, the reason that interest in the topic began only recently is that many in the neural network research community believed that it was difficult to use pruning to substantially reduce the cost of training.

Han et al. (2015) attempted to train pruned neural networks in place of standard neural networks. To do so, they sampled a new value from the initialization distribution for (i.e., *reinitialized*) each weight in the neural network that survived pruning. Subsequently, they fine-tuned the network. This amounts to conducting a fresh training run, but with the pruned network architecture rather than the original dense network. They found that doing so led to lower accuracy, concluding that:

It is better to retain the weights from the initial training phase for the connections that survived pruning than it is to reinitialize.

Shortly thereafter, in a paper on a structured pruning strategy that pruned entire convolutional kernels, Li et al. (2017a) found the same thing:

Training a pruned model from scratch perform[ed] worse than retraining a pruned model [i.e., performing fine-tuning in the standard way].

On the basis of this result, they hypothesized that

[This finding] may indicate the difficulty of training a network with small capacity.

There was also theoretical backing for the belief that training a neural network might require more capacity than representing the function that it learns. Building on the information bottleneck theory of Tishby et al. (2000), Shwartz-Ziv & Tishby (2017) described the process of neural network as follows:

SGD optimization, commonly used in Deep learning, has two distinct phases: empirical error minimization (ERM) and representation compression.

Shwartz-Ziv & Tishby contend that this behavior is responsible for the surprising generalization performance of overparameterized neural networks: the compression phase that occurs later in training means the network is using far less capacity than it has available, and the network thereby has lower propensity to overfit. A corollary of this belief is that the network may be amenable to pruning only later in training, when the representation is compressed and the network has excess capacity. During Shwartz-Ziv & Tishby's empirical error minimization phase, the network learns an uncompressed representation of the function that requires more capacity to store. Pruning during this phase would damage the function learned by the network, since it is fully using its capacity. As this representation is compressed during Shwartz-Ziv & Tishby's second phase, the network has spare capacity that can be .[3]

One of the main contributions of the thesis that follows is to demonstrate that it is indeed possible to train the kinds of neural network architectures that arise from pruning. Specifically, from a point early in training (and, in some cases, at the beginning of training), standard networks can be pruned and still train to the same accuracy as the unpruned network. In standard academic benchmarks, the level of pruning at which this works is the same or close to the same as the level of pruning achievable by magnitude pruning after training, indicating that—as far as contemporary neural network training procedures are concerned—there is perhaps little or no difference between the capacity necessary for networks to represent the functions they learn and to learn those functions. In short, I show in this thesis that neural networks can still learn effectively even when pruned early in training, opening the door to using pruning to make training more efficient, something previously treated as impossible under the beliefs described in this section.

Motivation in this thesis: understanding how practical neural networks learn. See Chapter 1 for a full elaboration of this motivation.

[3]At the onset of the research trajectory described in this thesis, I asked a faculty member why it might be difficult to train a pruned network if that network is capable of representing a function that reaches the same accuracy as the full network. This information bottleneck explanation is the answer that he gave me. My reaction was a desire to test this belief about pruning empirically, leading to the line of work presented in this thesis. Incidentally, Saxe et al. (2018) found that "there is no evident causal connection between compression and generalization: networks that do not compress are still capable of generalization, and vice versa." They conclude that their findings "put in doubt the generality of the IB theory of deep learning as an explanation of generalization performance in deep architectures," specifically the findings of Shwartz-Ziv & Tishby (2017).

Chapter 3

The Lottery Ticket Hypothesis in Small-Scale Settings

In this section, I study my research question about the existence of sparse, trainable neural networks (Section 1.3) in the context of small-scale settings for computer vision. These settings include small fully-connected and convolutional networks for image classification tasks like MNIST (LeCun & Cortes) and CIFAR-10 (Krizhevsky, 2009). In these settings, the results about sparse, trainable subnetworks are simplest, providing a foundation for the more nuanced results in later chapters.

Motivation. The motivation for this research is two-fold. From a practical perspective, the motivation of this research is to understand the cost reductions that might be possible by training sparse[1] neural networks (like that arise from pruning) in place of the dense networks we typically train. To understand the opportunity that exists to reduce costs in this way, I search for the sparsest possible networks that, during a training run, could have trained to completion on their own. The properties of such networks (i.e., when they come into existence, how sparse they are, how many

[1] I use the term *sparse* to refer to neural networks that have had parameters removed. The *sparsity* of a neural network is the percentage of parameters that have been removed. Pruning results in sparse neural networks, particularly when that pruning occurs in an unstructured way that results in an architecture that has no possible dense representation. (I.e., in contrast to pruning entire neurons, which produces a network that has a smaller dense representation because it has eliminated an entire row or column from a matrix.)

steps they need to train) motivate—or, alternatively, cast doubt on—further research to develop methods for finding and training such subnetworks efficiently, a step along the path toward allowing these networks to serve as a more cost-effective replacement for the dense networks we currently train.

From a scientific perspective, the motivation of this research is to understand the capacity necessary for a neural network to learn a function in comparison to the capacity necessary for it to represent a function. As described in Section 2.2, previous work has argued that training required far more capacity than representing the learned function (e.g., Li et al., 2017a; Shwartz-Ziv & Tishby, 2017), perhaps even to the point where the entire unpruned network was necessary for learning. Insofar as parameter count affects the capacity of a neural network, this belief implies that it should only be possible to prune few parameters (if any) early in training (as compared to the parameters it is possible to prune after training) without affecting the final accuracy of the trained network.

Overview. In Section 3.1, I describe my procedure for finding subnetworks that can train from initialization to completion in these small-scale settings for computer vision. Section 3.3 shows that—using the metrics described in Section 3.2 for the settings described in this chapter—the procedure finds *non-trivially* small *matching subnetworks*: those that can train to completion on their own in the same number of steps as the original network and match its accuracy. A subnetwork is a *trivial matching subnetwork* if (a) it is a matching subnetwork and (b) a subnetwork of the same size found by pruning randomly is also a matching subnetwork.[2] This procedure operates by examining the state of the unpruned network after training and retroactively finding matching subnetworks that existed at initialization.

Each subnetwork found by this procedure is a matching subnetwork down to a similar sparsity to that at which the subnetwork found by pruning after training can

[2]The rationale for this definition is that neural networks are often so overparameterized that it is possible to randomly prune some connections without any effect on accuracy. One trivial strategy for finding subnetworks at ostensibly high levels of sparsity is by starting with an excessively large dense network (i.e., by increasing its width) and randomly pruning it. I focus on more challenging levels of sparsity where the trivial procedure of randomly pruning does not produce matching subnetworks.

maintain full accuracy. **This demonstrates that it is indeed possible to train a pruned network to completion in place of the unpruned network and reach full accuracy, at least in these settings.**

The specific initial weight to which each connection of the subnetwork is essential for its success: these results only hold when the connections of the subnetworks are set back to the same values they received at the beginning of training the unpruned network. In other words, these subnetworks have "won the initialization lottery:" they received lucky initial weights whose connections allowed them to train to high accuracy; sampling a new random initialization leads to lower accuracy. For this reason, I refer to matching subnetworks at initialization as *winning tickets*.

3.1 Methodology: Iterative Magnitude Pruning

In order to find these subnetworks, I use a procedure called *iterative magnitude pruning* (IMP; Han et al., 2015). IMP is one of the two central experimental tools in this thesis (the other being instability analysis as described in Section 4.1). Below, I describe the design space of approaches for finding such subnetworks, the design choices that I use in IMP, and other alternatives. A procedure for finding sparse, trainable networks must return two pieces of information: the connectivity pattern of the sparse network and the initial values for those parameters.

3.1.1 Choosing a Connectivity Pattern

Optimal approach: trying all possible subnetworks. The optimal approach to identifying the sparsest possible subnetworks that can train to full accuracy would be to try all possible connectivity patterns at each level of sparsity until one viable subnetwork is found. Unfortunately, this approach is computationally infeasible for even the smallest networks I consider in this thesis, which have hundreds of thousands of parameters from which to choose sparse subnetworks.

My approach in IMP: magnitude pruning after training. To choose the connectivity pattern for the sparse network, IMP begins with a dense network of which this subnetwork will be a subgraph, trains it to completion, and prunes the least important parameters. The hypothesis behind this approach is that the parameters that survive pruning are those that are most important (at least, according to the pruning heuristic that is employed). The particular pruning heuristic I use to prune is magnitude pruning: prune the parameters with the lowest-magnitude values as compared globally throughout the network (Janowsky, 1989; Han et al., 2015; Gale et al., 2019).

Note that this approach is not a viable way to reduce the cost of training. It cannot find a sparse, trainable subnetwork that can be used in place of the dense network at the moment when such a sparse network comes into existence. After all, the it requires the entire dense network to be trained first, defeating the purpose of replacing this step with something more efficient. However, my goal in this section is not to propose a general solution for efficient sparse training all at once. Instead, it is to develop an oracle to demonstrate that there exist sparse, trainable networks worth seeking out efficiently in the first place. Like trying all possible subnetworks (the optimal approach), IMP is not computationally efficient for real-world use in reducing the cost of training.

Baseline: random pruning. An alternative, naive approach for selecting the sparsity pattern is simply to do so randomly. If this approach produced networks of similar quality to those found by magnitude pruning after training at similar sparsities, it would be strictly preferable: it would be able to find subnetworks without any auxiliary information, whereas magnitude pruning requires the expensive step of first training the entire dense network. On the other hand, if magnitude pruning after training performs better than random pruning, it would indicate that the magnitudes after training indeed contain useful signal for which weights are important for a sparse network. To make it possible to consider these possibilities, I include the baseline where I randomly prune the network globally and where I randomly prune each layer in the same per-layer proportions as magnitude pruning after training.[3]

Alternative methods studied later: magnitude pruning (and other heuristics) before or during training. For a detailed study of performing magnitude pruning (and several other recently proposed heuristics) at initialization and at other points during training, see Chapter 5, which evaluates the current state of research for finding sparse, trainable subnetworks efficiently. I do not describe pruning at these other points in training here, since this section focuses on identifying and describing the best known subnetworks without taking the cost of finding them into account.

3.1.2 Choosing the Initial Values

Optimal approach: trying all possible initializations. The optimal approach to identifying the appropriate initialization for the sparsest possible subnetworks that can train to full accuracy would be to try all possible initializations. In practice, neural networks are composed of 32-bit or 16-bit floating point numbers, so there are finitely-many combinations to try. In addition, follow-up research on this topic by Zhou et al. (2019) and Frankle et al. (2020b) has shown that, in some cases, only the signs of the initial values matter, not the values themselves (so long as the values are at a reasonable scale, such as those in the standard initializations proposed by Glorot & Bengio (2010) and He et al. (2015)). This reduces a space of 32-bit or 16-bit values to one of 1-bit values. Unfortunately, even this space is still computationally infeasible to search for even the smallest networks I consider in this thesis.

My approach in IMP: the original initialization. One example of using an initialization derived from the state of the dense network at some step t of training (i.e., case (a) above) is to set the values of the subnetwork's parameters could be set to the same values that they received when the dense network was initialized.[4] The

[3]For a detailed study of randomly pruning at various granularities, such as within convolutional channels or kernels, see Frankle et al. (2020b).

[4]It will become necessary to use the values from a later step $t > 0$ in training in larger-scale settings. See Chapter 4.

hypothesis behind this approach is that the important parameters contained within the chosen subnetwork received a fortuitous combination of initial weights that allowed them to train effectively. These initial weights predisposed the subnetwork to learn effectively on its own.[5] This view of initialization—that the specific sample of weights proved important for the subnetwork to learn—cuts against the distributional point of view commonly expressed in work on initialization (e.g., Glorot & Bengio, 2010; He et al., 2015; Zhang et al., 2019). That research focuses on the magnitudes or variances of the gradients throughout the network based on the initialization distributions proposed for each layer. The heuristic of using the original initialization for the sparse subnetwork instead draws attention to the importance of the specific sample from these distributions.[6]

Baseline: sampling a new random initialization. The other approach to initializing the subnetwork that could conceivably lead to an efficient realization in practice is to select a function that efficiently generates an initialization (i.e., case (b) above). For example, a naive approach to doing so would be to simply take a new sample from the distribution used to initialize the dense network (i.e., Glorot & Bengio, 2010; He et al., 2015). In contrast to the approach above, which maintains the original *sample* from the initialization distribution, this approach maintains the distribution but selects a new sample. If this approach produced networks of similar quality to those that use the original sample, it would suggest that the specific sample matters less than the overall distribution to be used. In this manner, it functions as a baseline that serves as an opportunity to falsify claims about the importance of the original sample from the initialization distribution. Note that, for this baseline, I will maintain the exact same initialization distribution as for the dense network; I will

[5]There are many possible reasons why these initial weights might predispose the subnetwork favorably in this way. For example, one might speculate that these weights provide a good initial position on the optimization landscape or useful initial features that formed the basis for the functions that the neurons eventually learned. In Chapter 4, I will discuss one property that seems to consistently hold for initializations that lead to matching subnetworks: that the subnetworks always train to the same, linearly connected region of the loss landscape regardless of the sample of SGD noise (i.e., data order).

[6]For an analysis of the effect of adding noise to the original initialization, See Frankle et al. (2020b).

not, for example, modify the parameters of the distribution based on the fan-ins and fan-outs of the sparse connectivity pattern of the subnetwork.

Alternative method: pre-training. One final strategy for generating the initial weights for the subnetwork is to use pre-training. Pre-training is an increasingly popular (and, in some cases, the primary) strategy for initializing networks for a range of downstream tasks. Examples of strategies for initializing networks in this way include BERT (Devlin et al., 2019) for natural language processing and SimCLR (Chen et al., 2020b) and MoCo (He et al., 2020) for computer vision. Pre-training requires a large initial investment in computation to develop the initialization, but that cost can be amortized by repeatedly reusing the initialization for many different downstream tasks. In the context of the results I present in this thesis, it provides an alternate, cost-effective way to initialize sparse models. I do not study self-supervised pre-training of this kind in this thesis, but I, along with my collaborators, have found this to be an effective strategy for initializing sparse networks in both natural language processing (Chen et al., 2020a) with BERT models and in computer vision (Frankle et al., 2020b) with self-supervised rotation (Gidaris et al., 2018). In these experiments, the pre-trained initializations replace the randomly sampled initialization for both the dense network and the sparse subnetwork. In this way, it is a drop-in replacement for the randomly sampled initialization; I use the same initialization for finding the connectivity pattern using the dense network and for initializing the subnetwork. It remains an open question for future research whether a connectivity pattern found with one sample of the pre-trained initialization is compatible with another sample from the pre-trained initialization.

Later in this thesis, however, I do consider a very brief dense training phase (1%-5% of the total training time) at the beginning of training prior to pruning (Chapter 4). Insofar as this phase constitutes pre-training (despite meaningful differences in objective and length of time compared to the self-supervised approaches mentioned above), such pre-training does allow IMP to find sparse subnetworks where it cannot do so at initialization.

3.1.3 The IMP Procedure

Here, I describe the technical details of the procedure I propose for finding matching subnetworks at non-trivial sparsities. It selects the initialization by using the same initial values as the original, dense network used for each parameter that survives pruning. This procedure selects the connectivity pattern by training the dense network and pruning the parameters whose values have the lowest magnitudes; for this reason, I refer to it as *iterative magnitude pruning* (IMP).

In practice, this procedure can be performed in two modes: *one-shot* and *iterative*. In one-shot mode, the dense network is trained to completion, pruned to the target sparsity, and each unpruned parameter is set back to its original initialization, thereby constructing a candidate subnetwork in a single pass In one-shot mode, p is the target sparsity and $N = 1$ in Algorithm 1 below. In iterative mode, the dense network is repeatedly trained to completion, pruned by a fixed amount, and set back to its original initialization until the target sparsity is reached. In this mode, p is the fraction of weights to prune on each iteration (e.g., 20%) and N is selected such that the target sparsity is $1 - (1 - p)^N$ (e.g., $N = 3$ for 48.8% sparsity). The benefit of multiple iterations of pruning is that pruning weights gradually rather than all at once leads to higher accuracy and sparser matching subnetworks (Han et al., 2015). Unless otherwise noted, I will prune 20% of parameters on each iteration of IMP in iterative mode to reach various target sparsities; this is a low enough pruning rate that it maximizes the accuracy of the pruned networks, but a high enough rate that pruning to high sparsities is feasible.

The formal description of IMP is as follows:

First, a dense, randomly initialized network (steps 1 and 2) is trained to completion (the first time that step 4 executes) and pruned such that p% of parameters remain (step 5). This pruning mask is combined with the state of the network at initialization to produce a subnetwork (step 6). If the procedure is run in one-shot mode (i.e., $N = 1$), this is the subnetwork whose performance will be evaluated. If the procedure is run in iterative mode (i.e., $N > 1$), this subnetwork can be trained to completion

Algorithm 1 IMP with resetting to step 0, N iterations, and $p\%$ of parameters pruned on each iteration.

```
1: Create a neural network with randomly initialized parameters $w_0 \in \mathbb{R}^d$ sampled from a
   initialization distribution $\mathcal{D}$.
2: Initialize pruning mask to $m = 1^d$ (i.e., all 1's).
3: for $n \in \{1, \ldots, N\}$ do
4:     Train $(w_0, m)$ (the subnetwork at initialization) from initialization (step 0) to completion
       (step $T$) using the training algorithm $\mathcal{A}$, resulting in trained subnetwork $(w_T, m)$:
       Let $w_T = \mathcal{A}^{0 \to T}(w_0, m)$.
5:     Prune the lowest magnitude $p\%$ of entries of $w_T$.
       Let $m[i] = 0$ if $|w_T[i]|$ is among the lowest magnitude $p\%$ of entries in $w_T$.
6: Return the subnetwork $(w_0, m)$
```

again (step 4), pruned (step 5), reset back to initialization with fewer connections remaining (step 4 again), and potentially trained and pruned several times again.

Global magnitude pruning. One intriguing aspect of IMP is that it uses *global* magnitude pruning: it compares all parameters in the network and removes those with the lowest magnitudes in a manner agnostic to where in the network those parameters resize. As a point of contrast, Han et al. (2015) use layer-wise magnitude pruning in which they prune each layer by an equal amount. He et al. (2018b) use a more advanced technique to determine the layerwise proportions in which to prune the network: a reinforcement-learning based approach. I, and followed on by others (Renda et al., 2020), found that simply performing global magnitude pruning—and allowing the layerwise pruning proportions to arise naturally from there—works just as well as these other approaches and conveniently eliminates a design decision along the way.

It is not obvious a priori that global pruning should work at all. For example, in networks trained with ReLU activations (which includes all of the networks in this thesis), the layerwise scales do not matter; it would be possible to scale up the weights in layer N by 10x and scale down the weights in layer $N + 1$ by 10x and still represent the same function. Altering scale in this way could lead to a network that, when pruned in a global manner, would have an entire layer eliminated in a manner that would sever connectivity. In practice, however, this does not seem to occur, whether due to the initialization distributions, weight decay regularization, batch normalization, or other unidentified dynamics of SGD. This is a rich potential direction to explore for future work.

3.1.4 Baselines

In addition to the IMP procedure, I perform three baselines as described above.[7]

Random reinitialization. The random reinitialization baseline involves sampling a new initialization w'_0 from initialization distribution $\mathcal{D}$ and combining it with the mask m produced by IMP, leading to subnetwork (w'_0, m).

Global random pruning. The global random pruning baseline involves sampling a new mask $m' \in \{0,1\}^d$ with the same sparsity as m but with the locations of the 1's and 0's selected randomly. Equivalently, m' can be generated by randomly shuffling the values of m. This leads to the subnetwork (w_0, m').

Layerwise random pruning. The layerwise random pruning baseline involves sampling a new mask $m' \in \{0,1\}^d$ with the same sparsity as m within sub-units of m (in this case, the parts of m that correspond to individual layers of the network) but with the locations of the 1's and 0's selected randomly. Equivalently, m' can be generated by randomly shuffling the values within the sub-units of m that correspond to individual layers of the network. This leads to the subnetwork (w_0, m').

[7]Although I separately consider random reinitialization and random pruning, I do not consider the two together. This is because random pruning implicitly also imposes random reinitialization, since there is no longer correspondence between the choice of the connectivity pattern (i.e., m) and the initial weights. In other words, in the context of global random pruning, the original initialization w_0 is no different than any other IID sample from the initialization distribution $\mathcal{D}$ with respect to a randomly sampled mask m', so there is no need to also sample a new initialization w'_0. In this sense, global random pruning is a strictly more severe perturbation to the network than reinitialization.

In the case of layerwise random pruning, it is possible that there may be correspondence between the initialization w_0 and the pruning mask m' because the per-layer sparsities of the mask may depend on the specific initialization w_0. However, in practice, all samples from the initialization distribution $\mathcal{D}$ lead to effectively identical per-layer sparsities, so there is no such correspondence in practice. Hence, there is no need to randomly reinitialize in this case either, and layerwise random pruning also implicitly imposes random reinitialization.

Standard pruning (learning rate rewinding). By *standard pruning*, I mean pruning that takes place after training with the goal of reducing the size of the network for inference. Such pruning techniques do not involve any possibility of reducing the cost of the network. However, they provide insight into the number of parameters necessary for the network to represent the function that it learned during training. This addresses the broader research question of how the capacity needed to learn a function compares to the capacity necessary to represent a function of commensurate accuracy it after it has been learned.

For this baseline, I use a specific pruning technique called *learning rate rewinding* (LRR) due to Renda et al. (2020). LRR performs magnitude pruning after training followed by fine-tuning just as described by Janowsky (1989) and Han et al. (2015) as detailed in Chapter 1. Just like IMP, LRR can be performed in one-shot mode (pruning to the target sparsity all at once and fine-tuning thereafter) or in iterative mode (repeatedly pruning and fine-tuning until the target sparsity is reached).

There are two main aspects of LRR that distinguish it from other magnitude pruning techniques. First, it uses global magnitude pruning (see the elaboration in Section 3.1.3). Second, during fine-tuning, it repeats the entire learning rate schedule from the original training phase. In particular, (1) fine-tuning step takes the same length as the entirety of the original training phase and (2) at the beginning of each fine-tuning phase, the learning rate reverts to its value at the beginning of training and follows the same schedule as the training phase thereafter. This means that, in iterative mode, if N iterative pruning phases occur, fine tuning will require N times the number of steps as the original network was trained for. Reverting back to the original learning rate schedule is a departure from prior work, which often uses one of the lower learning rates (or the lowest learning rate) from the original training schedule (Han et al., 2015). Renda et al. found that these design choices made it possible for LRR to recover smaller subnetworks that matched full accuracy as compared to other magnitude pruning strategies.

Concretely, LRR follows Algorithm 2. This is identical to IMP (Algorithm 1) with one important difference: at the end of each training phase, the unpruned parameters retain their weights rather than resetting them back to their values at initialization.

Algorithm 2 LRR with N iterations and $p\%$ of parameters pruned on each iteration.

```
1: Create a neural network with randomly initialized parameters $w \in \mathbb{R}^d$ sampled from a
   initialization distribution $\mathcal{D}$.
2: Initialize pruning mask to $m = 1^d$ (i.e., all 1's).
3: for $n \in \{1, \ldots, N\}$ do
4:     Train $(w, m)$ (the subnetwork using the current weights $w$) from step 0 to completion
       (step $T$) using the training algorithm $\mathcal{A}$, resulting in trained subnetwork $(w_T, m)$:
       Let $w_T = \mathcal{A}^{0 \to T}(w, m)$.
5:     Prune the lowest magnitude $p\%$ of entries of $w_T$.
       Let $m[i] = 0$ if $|w_T[i]|$ is among the lowest magnitude $p\%$ of entries in $w_T$.
6:     Let $w = w_T$
7: Return the subnetwork $(w, m)$
```

Just as with IMP, LRR can run in one-shot mode (where p is the target sparsity and $N = 1$) or iterative mode (where $N > 1$ and $1 - (1 - p)^N$ is the target sparsity). Renda et al. find that LRR finds sparser networks at full accuracy than other variants of magnitude pruning and a variant of IMP for larger-scale networks (see Section 4), although it can lead to greater overfitting due to the larger number of training iterations that each weight experiences (in contrast, under IMP, weights only ever experience the number of training iterations that occur during the original training phases, since weights are reset to their original values before any re-training).

3.2 Methodology: Evaluating Subnetworks

To evaluate the subnetworks studied in this section—both those found using IMP and those generated for the baselines—I will assess whether they are *winning tickets.* The definition of a winning ticket is based on the motivation for this research: determining the upper limits of the cost reductions that might be possible by training sparse networks in place of the dense networks we typically train. The most direct way for a sparse network to be demonstrably useful is for it to be a drop-in replacement for the dense network. That is, it must be able to train to the same level of quality (in the case of the models in this section, that means test accuracy) in the same number of training steps. Even if no procedure is available for exploiting the sparsity of the network for a real-world reduction in cost, it is—at the very least—no worse than using the dense network. With this motivation in mind, I define a winning ticket as:

> *Given an unpruned neural network, a subnetwork of that network (w_0, m) at initialization is a winning ticket if the quality of training the subnetwork (w, m) to completion using the standard strategy for training the unpruned network is at least that of training the unpruned network to completion using the same strategy. Symbolically, $\psi(\mathcal{A}^{0\to T}(w_0, m)) \geq \psi(\mathcal{A}^{0\to T}(w_0))$, where ψ is the quality of a network, $\mathcal{A}^{x\to y}$ is the specified procedure for training from step x of training to step y of training, and T is the final step of training.*

In correspondence with this definition, I will assess subnetworks for both their quality and how quickly they learn. In addition, I will also assess them for their *triviality*: whether they are found at a meaningfully high level of sparsity.

Assessing the quality of subnetworks. To assess the quality of subnetworks, I will train them to completion in isolation and evaluate their quality on both the train set and a hold-out test set to ensure it is at least that of the unpruned network.

Assessing how quickly subnetworks learn. I will take two approaches to assessing how quickly subnetworks learn.

First, I will identify when subnetworks have finished learning, which may occur before or after the standard allotted number of training steps. To do so, I will train the subnetwork for the full number of steps and identify the step at which loss is lowest on a separate hold-out validation set. This is a simple early-stopping heuristic that identifies the point at which the network begins overfitting (if any). I will also evaluate the quality on the test set at this step to ensure it is at least that of the unpruned network.

This procedure works only in contexts where the networks do not follow a set learning rate schedule. When networks do follow a set learning rate schedule, the step of minimum validation loss is likely to only occur when the learning rate schedule allows the learning rate to reach a particularly low value, which may artificially be delayed by the schedule. Since most large-scale networks follow a pre-specified learning rate schedule, I will also use a second technique to assess how quickly the subnetworks learn: I will re-train the subnetworks with the training time decreased or increased and the learning rate schedule dilated accordingly. If the subnetwork trains slower, it will require a longer training time and larger dilation of the learning rate schedule to reach the standard level of quality attained by the unpruned network.

Assessing the non-triviality of subnetworks. For a winning ticket to be remarkable, it must also be *non-trivial*—it must be sparser than the subnetworks that can be found by randomly pruning. Although such randomly pruned subnetworks are indeed be winning tickets at certain sparsities, I am interested in subnetworks that go beyond the sparsities at which this trivial strategy is effective. There are two rationales behind this criterion: (a) I aim to motivate research that goes beyond the current state of the art, of which random pruning is trivially a part, and (b) random pruning generally does not produce particularly sparse matching subnetworks in practice, so going beyond the current state of the art will be necessary if training can occur at sufficiently high sparsities that it could plausibly lead to practical improvements. In order to evaluate the (non-)triviality of subnetworks, I will compare to the random pruning baseline.

3.2.1 Note

All data collected and analysis conducted in this chapter was done freshly for this thesis. It is a replication of the main experiments in the original lottery ticket paper (Frankle & Carbin, 2019) with better infrastructure, more resources, more data, nicer-looking graphs, commentary and analysis reflecting four more years of research maturity, and greater depth in several places (notably the way that I measure the number of steps necessary for subnetworks to train and the learning rate rewinding baseline that we developed a year later (Renda et al., 2020)). Although the findings are the same, there are slight discrepancies in the specific numbers between this replication and the original paper due to completely different implementations and infrastructure.

3.3 Results

3.3.1 Experimental Details

Networks and datasets. In this section, I will examine the following settings for image classfication:

- LeNet-300-100 for MNIST
- Conv-4 for CIFAR-10
- Conv-6 for CIFAR-10
- Conv-8 for CIFAR-10
- ResNet-20 for CIFAR-10
- ResNet-20 Warmup for CIFAR-10[8]
- ResNet-20 Low for CIFAR-10
- VGG-16 for CIFAR-10
- VGG-16 Warmup for CIFAR-10
- VGG-16 Low for CIFAR-10

The LeNet and Conv networks do not have a learning rate schedule; instead, they use Adam at fixed learning rate for a set number of steps. ResNet-20 and VGG-16 decrease the learning rate at fixed points in training according to a schedule.

[8]In the original lottery ticket paper (Frankle & Carbin, 2019), I erroneously referred to ResNet-20 as "ResNet-18", which is actually a different network designed for ImageNet. ResNet-20 has approximately 270K parameters, while ResNet-18 counterintuitively has about 20M parameters due to a larger number of channels per layer.

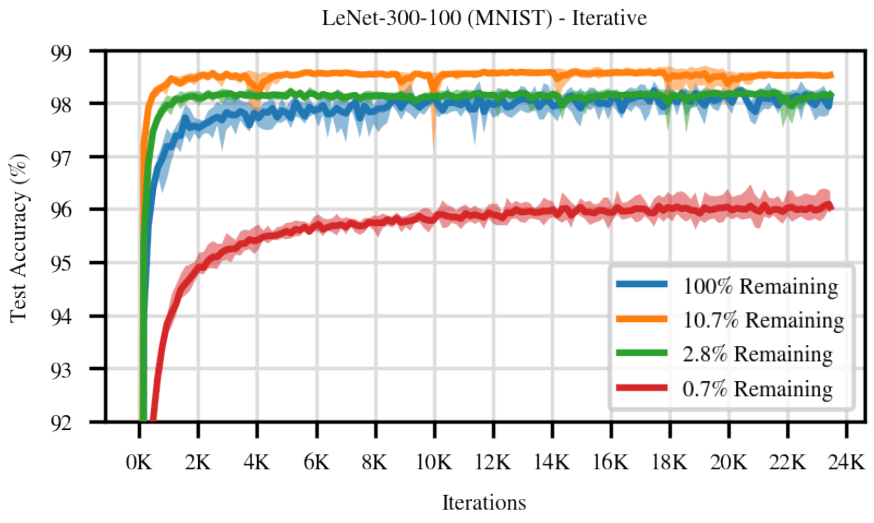


Figure 3-1: The test accuracy throughout training of the subnetworks produced by IMP (iterative mode) on LeNet-300-100 at various levels of pruning when using the original initialization. IMP finds winning tickets: subnetworks that can reach at least the same accuracy as the original network in no more than the same number of training steps.

Infrastructure. All networks were trained using PyTorch on TPUs. The codebase used for these experiments is an expanded version of the OpenLTH codebase I created during my internship at FAIR in the summer of 2019.[9]

Replicates. All results are the average across five replicates with different random seeds and random initializations. Error bars reflect the minimum and maximum value across all replicates.

3.3.2 Illustrative Example

Figure 3-1 shows an illustrative example of the behaviors I will describe in this section on LeNet-300-100 on MNIST. Each line is the test accuracy at the step of training specified on the x-axis. The blue line is when training the full, unpruned network. The other lines are when training IMP subnetworks at various levels of pruning. The network can be pruned until 2.8% of parameters remain while still maintaining at

[9]See: `www.github.com/facebookresearch/open_lth`

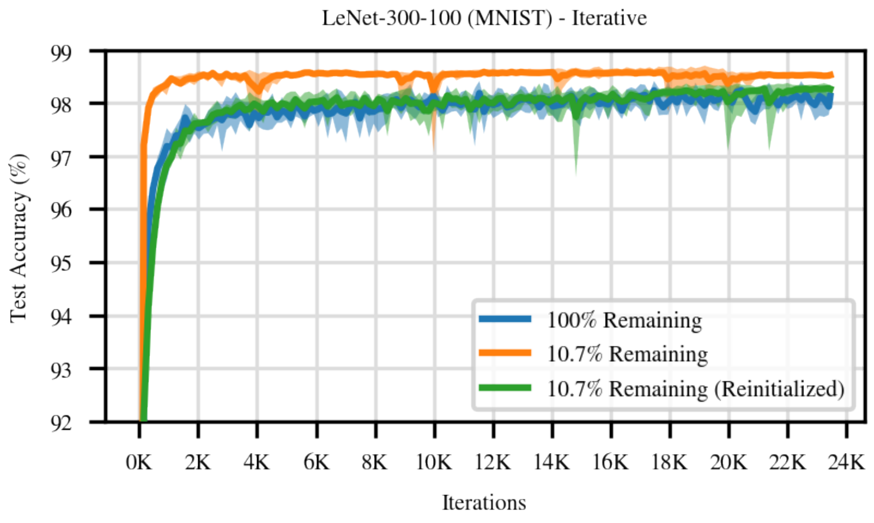


Figure 3-2: The test accuracy throughout training of the subnetworks produced by IMP (iterative mode) on LeNet-300-100 at various levels of pruning. Also includes the test accuracy when sampling a new random initialization for one of the subnetworks. Random reinitialization reduces the accuracy of IMP subnetworks.

least the same accuracy as the original network and training to that accuracy in at most the same number of steps. This trend cannot continue indefinitely: pruning further leads to lower accuracy (red line).

These subnetworks are most successful when using the original initialization provided to each parameter when the unpruned network was trained. Figure 3-2 shows the difference between using the original initialization (orange) and a new initialization sampled from the original distribution (green).

In summary, IMP produces winning tickets on LeNet-300-100 for MNIST, and the success of these winning tickets is dependent on reusing the original initialization.

3.3.3 One-shot vs. Iterative Pruning

As described in Algorithm 1, IMP has two modes: one-shot (which prunes to the target sparsity all at once) and iterative (which repeats the pruning, rewinding, and re-training process, pruning the network gradually to the target sparsity). Figures 3-3 and 3-4 compare these two strategies. The left column of plots shows the test

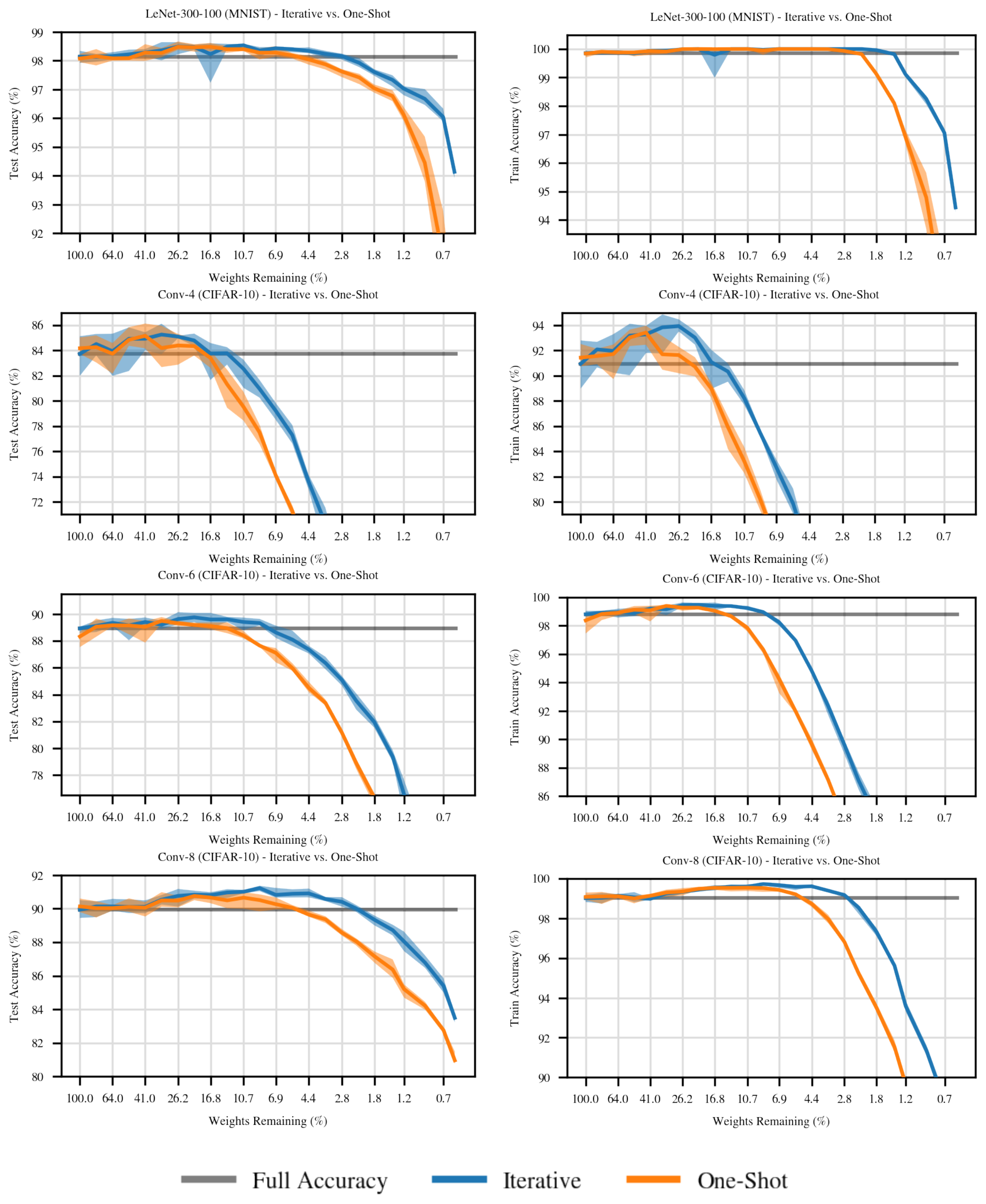


Figure 3-3: The test accuracy (left) and train accuracy (right) when performing IMP in one-shot mode (orange) and iterative mode (blue) for the benchmarks studied in this chapter. Each point along the x-axis is the final accuracy at a different level of pruning (i.e., the rightmost point from each line in Figure 3-1). The gray line is the accuracy when training the unpruned network; a subnetwork is a winning ticket when its test accuracy (represented by the blue or orange line) is at or above that of the unpruned network (the gray line). See Figure 3-4 for additional benchmarks.

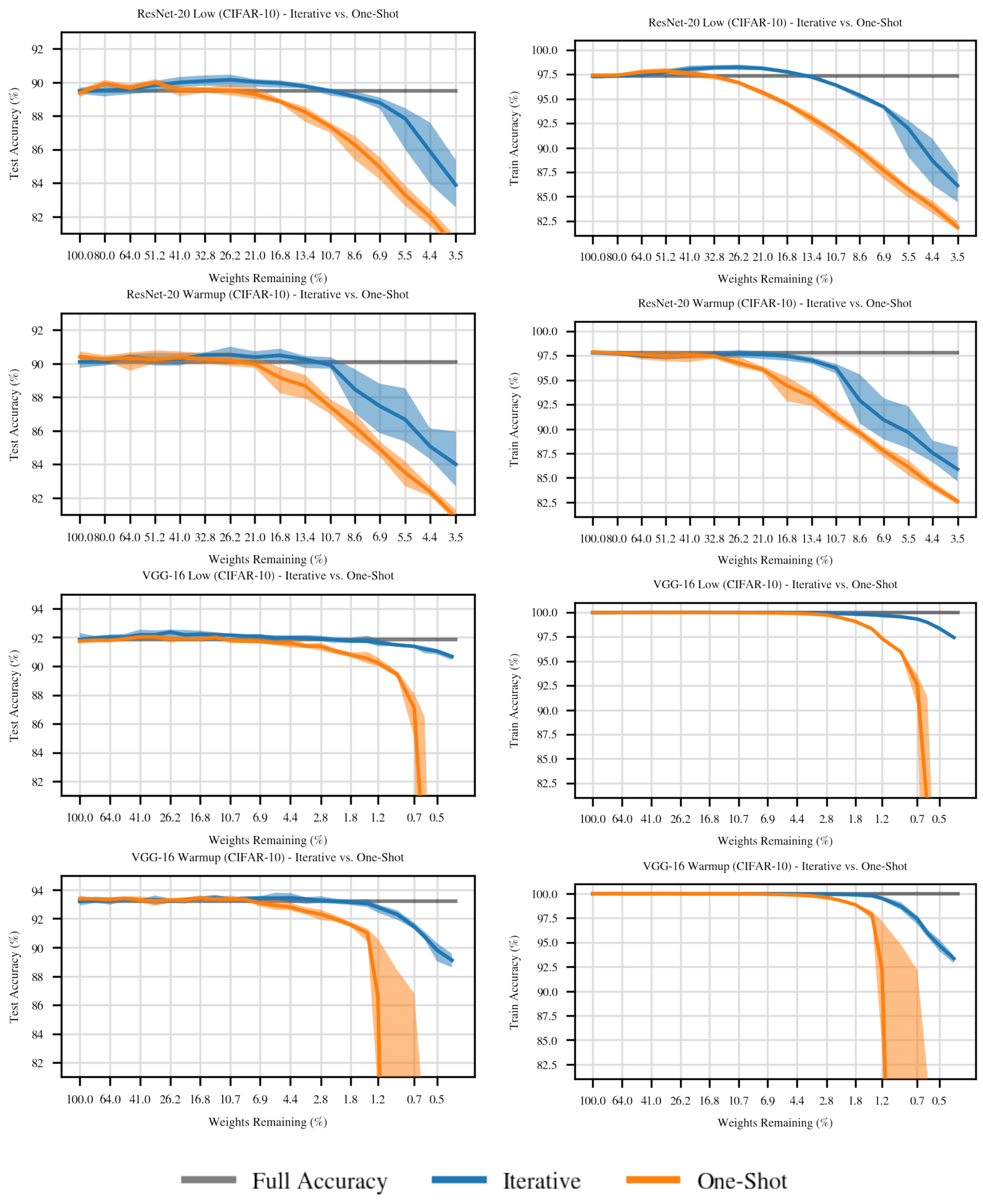


Figure 3-4: A continuation of Figure 3-3.

accuracy of one-shot pruning (orange) and iterative pruning of 20% of weights on each iteration (blue). The right column shows the train accuracy of the same experiment.

How to read and interpret the plots. This style of plot will appear frequently in this thesis. It is a condensed version of the information shown in Figure 3-1. Whereas Figure 3-1 shows the full training curve at all steps of training, Figures 3-3 and 3-4 show only the final accuracy for each network. In this way, each line from Figure 3-1 has been condensed into a single point whose level of pruning is captured by the x-axis of the graph. The y-axis shows the corresponding accuracy. In addition to lines showing the relationship between pruning and accuracy (the blue and orange line in Figures 3-3 and 3-4), these plots also include a horizontal gray line showing the accuracy of the unpruned network as a basis for comparison. The pruned networks in these plots are considered to be *winning tickets* when they reach the same accuracy as the original network (i.e., the blue or orange lines are above the gray line) it at most the same number of steps (in this case, all networks are trained for the same number of steps).

Although this representation of the data sacrifices some information (namely, the accuracy of the networks over the course of training), there are several advantages to portraying it in this way. First, it makes the relationship between sparsity and accuracy easier to see. Second, it makes it possible to fit all possible sparsities, not just a small number of hand-selected sparsities as in Figure 3-1.

Findings. First, these results show that winning tickets exist in all of the benchmarks studied in this section and that IMP uncovers these winning tickets.

The subnetworks found by IMP in iterative mode consistently outperform those found by IMP in one-shot mode. This can be measured in two ways. First, for any given level of pruning, iterative mode subnetworks reach higher accuracy than one-shot mode subnetworks . Second, the iterative mode subnetworks are winning tickets at higher levels of sparsity than the one-shot mode subnetworks: they match the test accuracy of the original network when more weights are pruned. These patterns hold true for both test and train accuracy. Although winning tickets are defined in terms of test accuracy, iterative mode subnetworks also match the train accuracy of the unpruned network at higher sparsities than oneshot-mode subnetworks. The sparsities at which this occurs are similar for test and train accuracy, but they are not identical. In all cases except MNIST, this occurs at higher sparsities for test accuracy than for train accuracy.

In summary, these findings show:

1. Winning tickets exist in all settings studied in this section.

2. IMP finds these winning tickets.

3. Iterative pruning finds sparser winning tickets than one-shot pruning.

Since the goal of this chapter is to identify the sparsest possible subnetworks capable of serving as drop-in replacements for the dense network, I will focus exclusively on iterative pruning going forward.

3.3.4 Capacity to Learn vs. Represent

In the previous section, I showed that winning tickets exist within a variety of small-scale neural networks for computer vision. In this section, I study how sparse these winning tickets are. As described in Section 3.1.4, there are two relevant baselines in this context.

The first baseline is *global random pruning*, in which weights are pruned from the network uniformly at random without regard to where in the network they are pruned from. This baseline is trivial: it is possible to use without any further knowledge about the state of the network, and it is trivially available to use efficiently at initialization. If another method for finding subnetworks does not outperform global random pruning, global random pruning would be a better choice due to its simplicity. A subnetwork is described as *non-trivial* if it outperforms the average accuracy of subnetworks of identical sparsity produced by global random pruning.

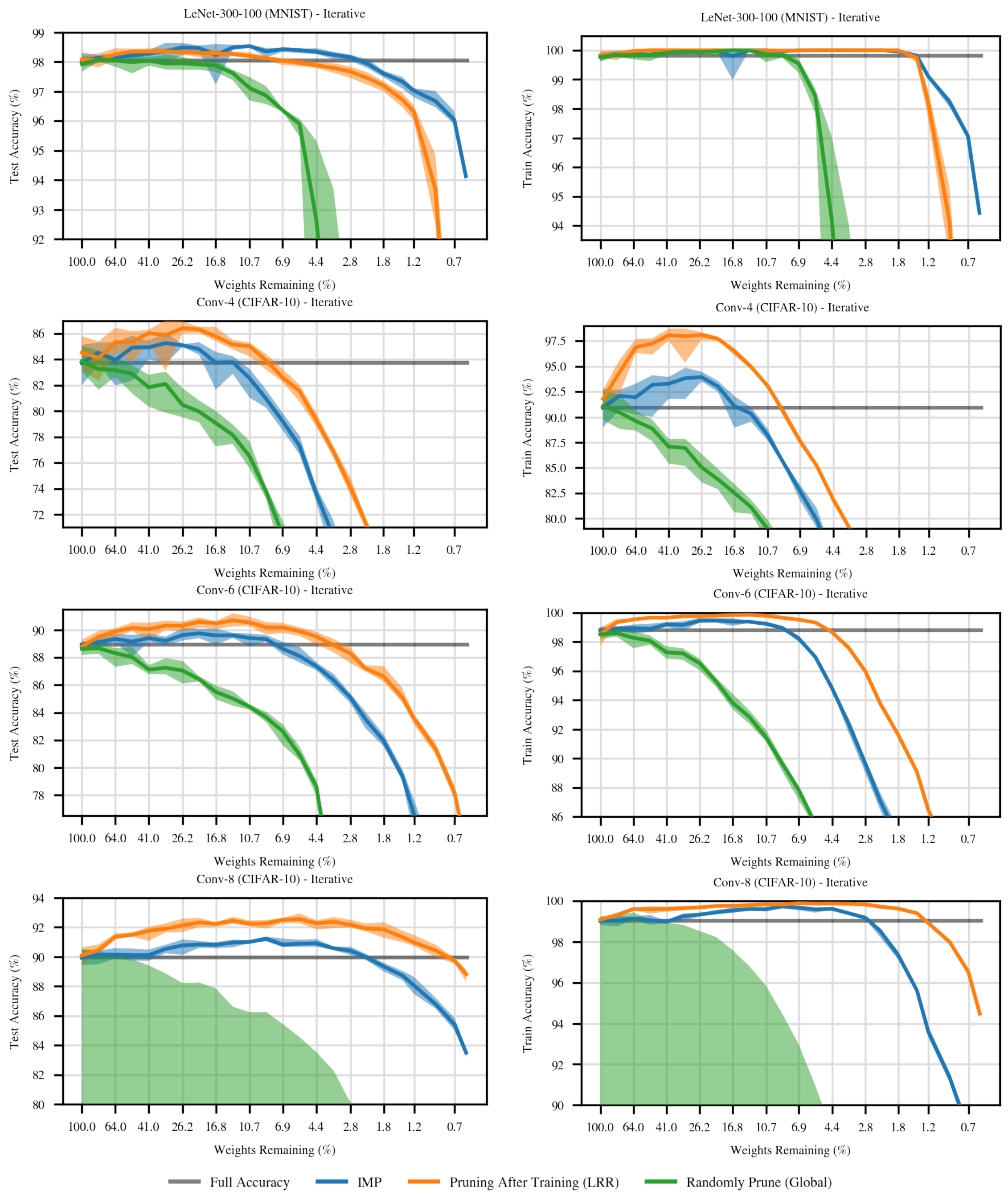


Figure 3-5: The test accuracy (left) and train accuracy (right) when performing IMP in iterative mode (blue), pruning after training via LRR in iterative mode (orange), and global random pruning at initialization (green) for the benchmarks studied in this chapter. The gray line is the accuracy when training the unpruned network. See Figure 3-6 for additional benchmarks.

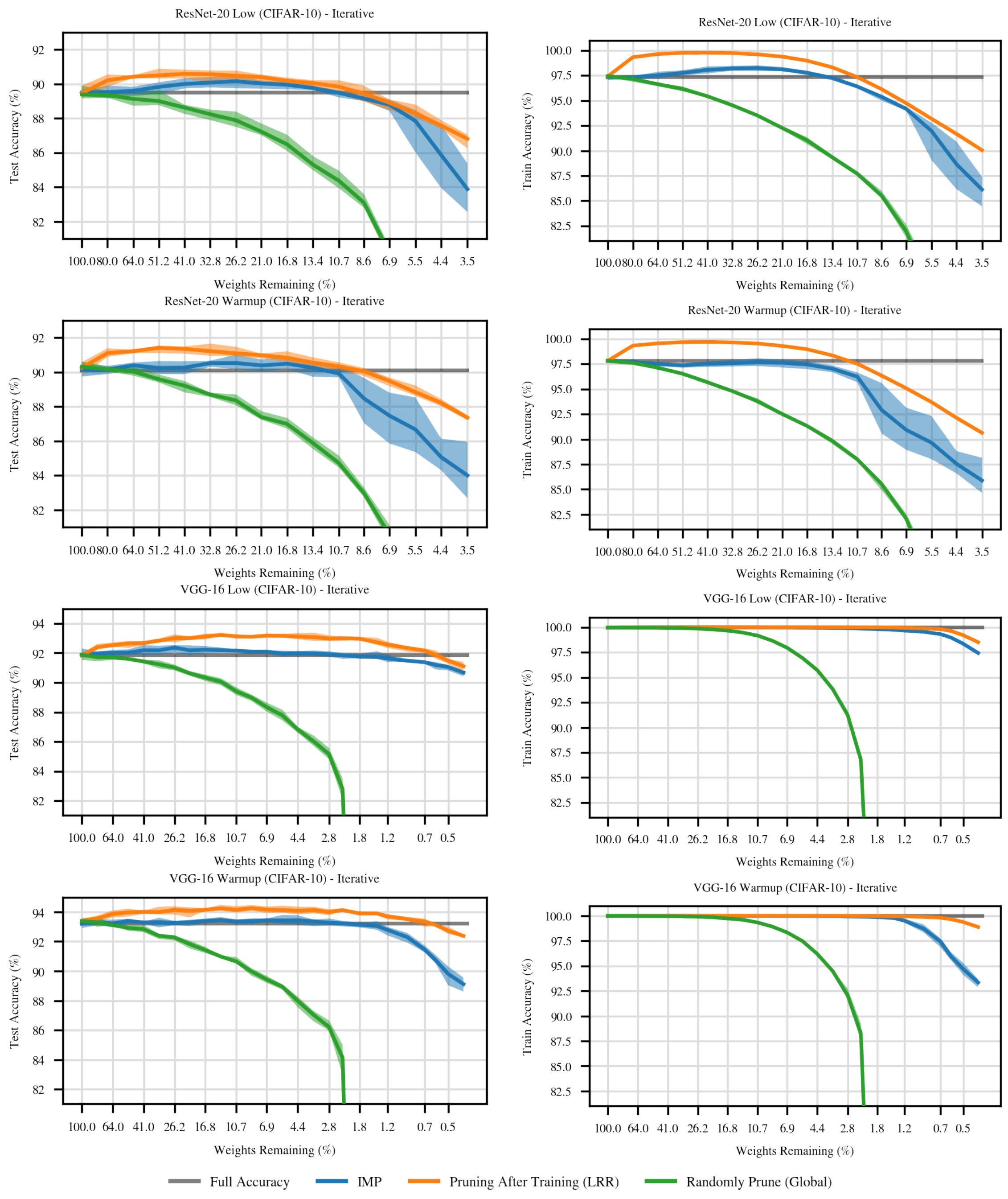


Figure 3-6: A continuation of Figure 3-5.

The second baseline is *standard pruning*, which—in this thesis—refers to the *learning rate rewinding (LRR)* pruning method of Renda et al. (2020) as described in Section 3.1.4 and Algorithm 2. This baseline is a variation of magnitude pruning after training in which the network is (1) trained to completion, (2) pruned, and (3) trained further to recover accuracy lost during pruning (Janowsky, 1989; Han et al., 2015). It is iterative, pruning 20% of weights in the network on each pruning step.

This key distinction between the LRR baseline and the IMP subnetworks are that it does not rewind the weights of the network back to iteration 0. In other words, it does not need to successfully train the sparse network using weights from initialization; it can reuse the representation learned during training, albeit with some weights missing. Insofar as this baseline is state-of-the-art for inference-focused pruning (and Renda et al. argue that it is), it reveals the sparsest known subnetworks capable of *representing* functions that reach a particular level of accuracy. It provides the ability to answer one of our key research questions: a comparison between the capacity necessary to represent high-accuracy functions (via LRR) and to learn such functions (via IMP). If IMP subnetworks reach similar tradeoffs between sparsity and accuracy to those found by LRR, this would indicate that the capacities necessary to learn and represent high-accuracy functions are similar. If there is a substantial difference in the performance of these groups of subnetworks, however, it would not necessarily mean that the capacities necessary to learn and represent high-accuracy functions are different; it could simply be that IMP finds suboptimal winning tickets.

Evaluation. In this experiment and nearly all experiments to follow in this thesis, I focus on the smallest parameter-count at which a subnetwork can match the accuracy of the unpruned network (shown by the gray horizontal lines in each plot). This is the smallest subnetwork that can serve as a drop-in replacement for the original network. While it is possible that even smaller subnetworks may be useful, these subnetworks reach lower accuracy than the original network, meaning there is a tradeoff between accuracy and efficiency. That tradeoff is often context-specific, whereas matching the accuracy of the full network means that the subnetwork is—in the worst case—still no worse than the unpruned network if it were to be used instead.

Findings. The results of this experiment are in Figures 3-5 and 3-6. There are two main results.

First, the winning tickets found by IMP are non-trivial. They can maintain full accuracy at much higher sparsities than random pruning. Concretely, the blue line is higher than the green line in all plots in Figures 3-5 and 3-6 and it crosses the horizontal gray line (representing the accuracy of the unpruned network) at higher sparsities. This difference is substantial in all cases. Global random pruning rarely finds any winning tickets at any sparsities; on the Conv-8 benchmark, the randomly pruned subnetworks often diverge entirely. In a few cases, global random pruning is able to maintain full train accuracy at some sparsities, but IMP can do so at much higher sparsities.

Second, the capacities to learn and represent functions are similar. In all but one case (LeNet on MNIST), LRR finds sparser subnetworks that reach full accuracy than the winning tickets found by IMP. This means that the smallest known subnetworks that can represent full-accuracy functions are smaller than the smallest known subnetworks that can learn full-accuracy functions from scratch. However, these differences are generally small, at most about 2x in terms of parameter count.

In summary, insofar as IMP finds the sparsest possible winning tickets, the capacity necessary to learn a function appears slightly larger than the capacity necessary to represent a function, although these differences are limited to about 2x.

3.3.5 The Role of Initialization and Pruning Pattern

The previous results show that IMP is able to find subnetworks of the state of the randomly initialized network that were capable of training in isolation to the same accuracy as the unpruned networks—at sparsities similar to those at which pruning after training can reach full accuracy, no less. As discussed in Section 3.1, IMP makes two design choices about the composition of each subnetwork:

1. The sparsity pattern of the subnetwork is found by pruning after training.

2. The initialization of the subnetwork is found by setting each weight to the value it received at random initialization when it was part of the unpruned network.

These are strict design choices. Ideally, the conditions necessary to produce winning tickets would be less strict. If satisfying these two conditions is necessary to produce winning tickets, then it suggests that developing an alternative to IMP that can efficiently find winning tickets at initialization could be difficult. Alternatively, if either of these conditions could be relaxed (e.g., if any random initialization or any random sparsity pattern will do), it would reduce the apparent difficulty of efficiently finding such subnetworks at initialization.

In this section, I evaluate the extent to which these design choices are necessary for a subnetwork found by IMP to be a winning ticket. (Note that IMP is not necessarily the only way to find a winning ticket, as outlined in Section 3.1 and related work on alternate methods (Savarese et al., 2020; Sreenivasan et al., 2022).) Specifically, I consider the following baselines as described in Section 3.1:

Random reinitialization (orange). Sampling a new random initial weight for each connection in the subnetwork from the same distribution as was used to initialize the original network. If a network created in this way were to perform (remotely) as well as the winning tickets found by IMP, it would substantially weaken the conditions known to be necessary to create a winning ticket and improve the prospects of developing an efficient algorithm for finding winning tickets at initialization.

Global random pruning (green). Selecting the sparsity pattern by pruning weights uniformly at random from the network. This will produce a sparse subnetwork in which each layer is pruned in approximately equal proportions.[10] This

[10]Note that the initialization of the network does not matter when conducting global random pruning. The only condition under which initializations would matter is if the distribution of weights in the winning ticket is different than the distribution of weights in the initialization distribution for the entire network. Frankle et al. (2020b) show that, even when keeping the same sparsity pattern and shuffling the initial values within each layer (or even within each neuron), accuracy is no better than when using a new initialization. In these experiments, I sample a new initialization when randomly pruning.

experiment is the same as the green line in Figures 3-5 and 3-6. If a network created in this way were to perform (remotely) as well as the winning tickets found by IMP, it would provide an immediate algorithm for finding winning tickets at initialization: globally prune randomly. (This assumes that there are no special distributional properties of the initial values of the weights found by IMP in each layer; see the previous footnote for a discussion of this possibility.)

Layerwise random pruning (red). Selecting the sparsity pattern by pruning the same number of weights as IMP pruned in each layer, but doing so uniformly at random. It is possible that the specific layerwise proportions found by IMP are the essential information necessary to create winning tickets. If a network created in this way were to perform (remotely) as well as the winning tickets found by IMP, it would imply that finding a winning tickets is much easier than previously thought: simply choose the right layerwise proportions. A priori, this is a genuine possibility: many papers have focused on determining the right proportions in which to prune networks after training (e.g., He et al., 2018b; Renda et al., 2020; Su et al., 2020). If this were the case, it would improve the prospect of finding an efficient way to prune neural networks at initialization. There are many useful sources of information about how much to prune each layer, such as its composition (e.g., convolution vs. fully-connected) and its initialization distribution (as determined by its fan-in and fan-out). As I will discuss in Chapter 5, several proposed methods for pruning at initialization rely entirely on the proportions in which they prune the network.[11]

Findings. The results of these experiments are in Figure 3-7. The iterative IMP experiment is in blue, and the aforementioned baselines are in the colors specified above. All of the baselines above substantially degrade the test accuracy of the subnetworks as compared to those found by IMP.

[11]Just as with global random pruning, the specific initialization of the weights when conducting layerwise random pruning does not matter; sampling from the same distribution as was used to initialize the unpruned network performs as well as sampling from the distribution of per-layer initial values from those weights found by IMP (Frankle et al., 2020b). In this experiment, I sample an entirely new initialization.

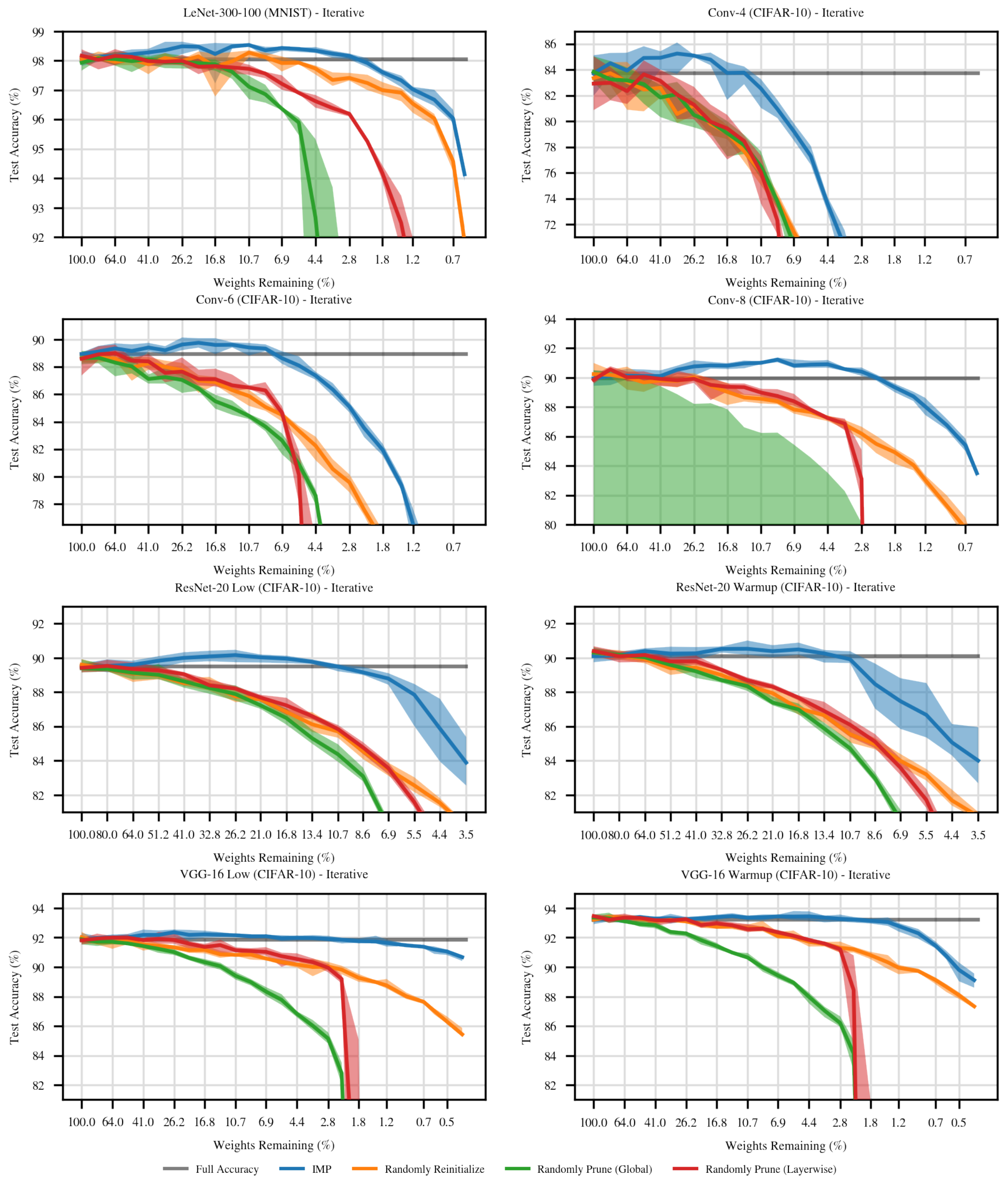


Figure 3-7: The test accuracy when performing IMP in iterative mode (blue) and with various ablations designed to explore the conditions necessary for a subnetwork found by IMP to be a winning ticket for the benchmarks studied in this chapter. The gray line is the accuracy when training the unpruned network.

Global random pruning has the most deleterious effects, likely because—compared to layerwise random pruning—it destroys information about the layerwise proportions in which to prune the network, which appear to be useful for reaching higher accuracy.

In all cases except MNIST LeNet, randomly pruning layerwise and randomly reinitializing have approximately the same accuracy at lower sparsities, suggesting that shuffling the positions of the weights does not have much effect once they have already been reinitialized. At higher sparsities, layerwise random pruning drops off sooner than random reinitialization in several cases, potentially due to severing connectivity in extremely sparse networks.

The behavior of layerwise random pruning and random reinitialization are different only for MNIST LeNet. This is likely because LeNet is the only fully-connected network under consideration; the positions of the unpruned connections in the first layer correspond to specific pixels in the input image, and disturbing those connections may have a particularly severe effect on the network's ability to learn. Lee et al. (2019) show that, for a different pruning method, the weights pruned in MNIST LeNet are located around the periphery of the input, which makes sense intuitively considering all of the numbers in MNIST are centered in the image.

I conclude that, for the winning tickets found by IMP, both the specific initial values and the connectivity pattern of the weights appear necessary for the subnetworks to train to full accuracy at sparsities similar to those of the networks found by pruning after training.

Further reading. For a more fine-grained exploration of reinitializing these subnetworks (shuffling the initial values within structural subcomponents of the network and adding SGD noise), see Frankle et al. (2020b). In that paper, we reach the same conclusions as in this chapter, although with greater confidence given the range of scenarios considered.

For a study of the role that the signs of the weights play in some of the small-scale benchmarks studied in this chapter, see Zhou et al. (2019). They find that the signs of the initial weights alone are enough to create equally accurate winning tickets; the magnitudes of the initial weights can be sampled randomly from the distribution used to initialize the unpruned network. However, for larger-scale settings (specifically, ResNets and VGGs on CIFAR-10), Frankle et al. (2020b) show that this is not the case; the magnitudes are also necessary. We argue that giving each initialization the same sign restricts the variance of the noise that reinitializing the network can impose, and we show that adding Gaussian noise with similar variance has a similar effect on the subnetwork's eventually accuracy.

3.3.6 Training Speed

In order to be a viable drop-in replacement for the unpruned network, a winning ticket must satisfy one additional condition beyond matching its accuracy: it must reach this accuracy in at most the same number of steps as the original network. In this section, I measure how many steps it takes the winning tickets and baselines from the preceding experiments to reach full accuracy.

There is no clear-cut way to measure the amount of time that a network takes to learn. Possible metrics include when the network converges (i.e., when training loss reaches 0 if it ever does) or when the network generalizes to the best of its ability (i.e., when test loss reaches its lowest value, if it ever does). There are many challenges of using such heuristics. Concretely:

1. Convergence typically occurs long after the benchmarks studied in this section have reached peak test accuracy, so it does not capture the metric of interest in the real world: generalization.

2. Looking at when the test loss reaches its lowest value involves deciding an aspect of training (specifically, when to stop training) based on the test set, which goes against the purpose of having a test set.

3. Standard benchmarks (including the ResNet-20 and VGG-16 benchmarks in this chapter) follow learning rate schedules. This means that the network is artificially constrained from reaching the lowest possible test or training loss

until a certain point in the learning rate schedule. It is possible that the network could have reached that point sooner had the learning rate schedule proceeded to lower learning rates sooner.

In light of these challenges, I study the steps necessary for the original networks, the IMP subnetworks, and baseline networks to train to completion using two metrics:

Early stopping metric. This metric looks at the step of training at which the loss on a hold-out set is lowest. In order to avoid involving the test set in the training process, I randomly sample one tenth of the training examples to create a hold-out validation set. This means that the overall test accuracy of each training run is slightly lower than in the preceding experiments, since the networks are trained on fewer examples. This metric addresses Challenge 2 by creating a separate hold-out set. Due to Challenge 3, I only apply this metric to benchmarks that use constant learning rate schedules: all benchmarks except ResNet-20 and VGG-16.[12]

Training dilation metric. In order to address Challenge 3, I will also measure training speed with a second metric. This metric works by dilating the number of training steps and the learning rate schedule of each network.[13] Concretely, if the *dilation factor* $\delta = 0.5$, the network will be trained for half as many steps and the learning rate schedule will proceed twice as quickly in changing the values of the learning rate (e.g., warmup will occur over half as many steps, and the learning rate drops will occur twice as soon). I sweep across many values of δ (both < 1.0 and > 1.0, specifically values between 0.2 and 2.1 at increments of 0.1) and select the minimum value of δ for which the network under consideration matches the

[12]This was the metric that I used in the original lottery ticket paper (Frankle & Carbin, 2019), including the use of the hold-out validation set. The only difference between this thesis and that work is that—in the original lottery ticket paper—I used the hold-out validation set in all experiments, slightly lowering the accuracy of all networks. In that paper, I was not able to measure the training time of ResNet-20 and VGG-16 due to the learning rate schedule challenges. The training dilation metric addresses that gap in the original paper.

[13]The idea of learning rate schedule dilation to assess the speedup of an intervention into the training process is due to Abhi Venigalla, who proposed it for our work at MosaicML. Credit for this idea, which he refers to as *scale schedule*, is his.

accuracy of the unpruned network (if it does so at all), which I refer to here as δ^*. It is possible that δ^* for the unpruned network is < 1.0, meaning that I provided an excessive number of training steps when designing the benchmarks. This sets an even more difficult bar for training time that the winning tickets must match. Since this experiment works both with and without a learning rate schedule, I use it for all benchmarks.[14]

Findings: Early stopping metric. Figure 3-8 shows the result of computing this heuristic (left) and the test accuracy at the step of minimum validation loss (right). This figure includes the unpruned network (gray), the IMP subnetwork (blue), and the random reinitialization (orange), global random pruning (green), and layerwise random pruning (red) baselines. **At sparsities where the IMP subnetworks are winning tickets (blue line above gray line on the right graph), they take at most the same number of steps as the unpruned network to reach minimum validation loss (the blue line is below the gray line in the right graph).** In contrast, the other baselines universally take as many steps or more to reach minimum validation loss (at least until sparsities so high that they diverge entirely and minimum validation loss no longer has any meaning). Based on this experiment, I conclude that the subnetworks found by IMP are indeed drop-in replacements for the unpruned network: they reach at least the same test accuracy in at most the same number of steps, and in many cases they reach higher test accuracy in fewer steps.

Findings: Training dilation metric. Figure 3-9 shows the result of computing this heuristic on all benchmarks considered in this chapter. This figure shows the dilation necessary for the unpruned network to reach full accuracy (gray line), the IMP subnetwork (blue line), and two baselines (the randomly reinitialized baseline in orange and the layerwise randomly pruned baseline in green). Note that, in some

[14]I consider this metric closer to the ideal that I was striving for in the original lottery ticket paper (Frankle & Carbin, 2019). For the sake of completeness, I have still included the early stopping metric for the relevant benchmarks. The training dilation metric is extremely computationally intensive: it requires re-training every sparse network 20 times. It would have been well beyond the computational budget that I had for the original paper, and it stretched the budget even for this thesis.

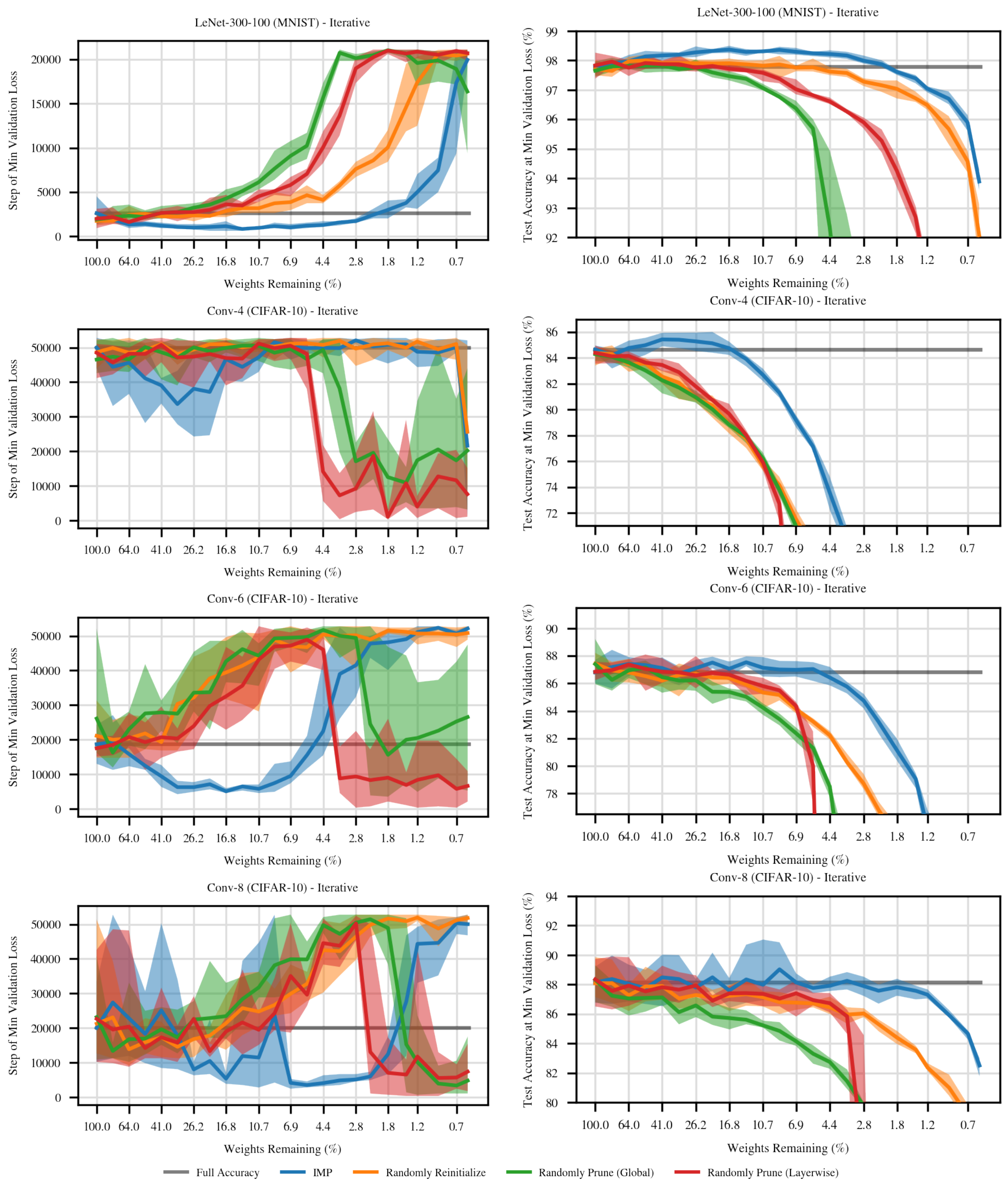


Figure 3-8: The step where validation accuracy is lowest for a holdout validation set comprising 10% of the training examples (left) and the test accuracy at that step (right). This figure only includes benchmarks that have a constant learning rate (i.e., those that do not follow a learning rate schedule). The gray line is the step of minimum validation loss for the unpruned network.

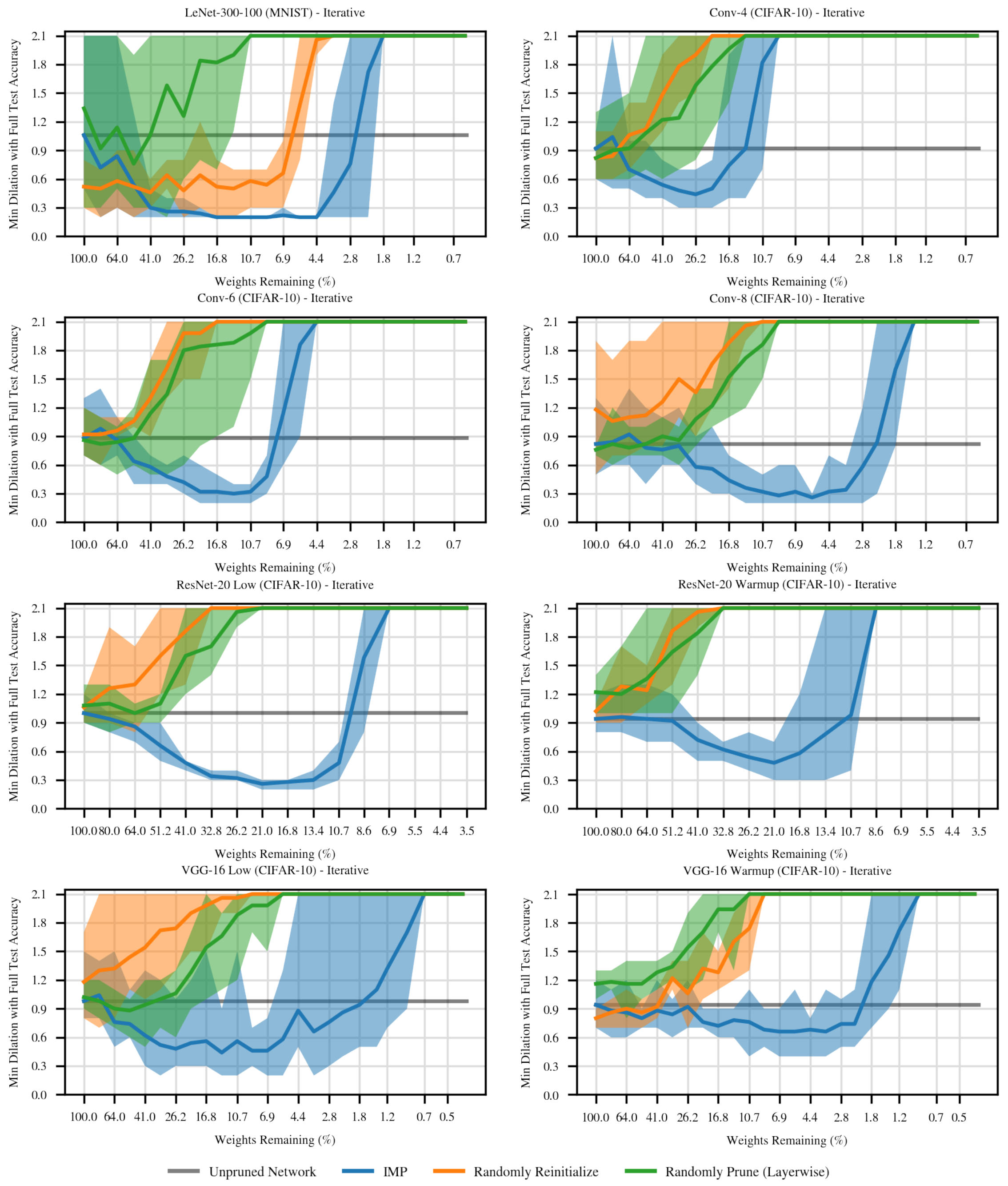


Figure 3-9: The minimum scaling factor δ^* by which the learning rate schedule can be dilated while still allowing the sparse network to match the accuracy of the unpruned network. The gray line is the minimum dilation necessary for the unpruned network to reach full accuracy (defined as dilation factor 1.0). In all cases, the IMP subnetwork (blue line) can reach the same accuracy as the unpruned network in at most the same dilation factor at high sparsities, whereas the baselines can only do so at the very lowest sparsities.

cases, the minimum dilation needed for the unpruned network to reach full test accuracy (δ^*) is slightly less than 1.0, indicating that the learning rate schedule was longer than necessary to reach that accuracy.

In all cases, the IMP-pruned network is able to reach full accuracy with a minimum dilation factor $\delta^* \leq$ that of the unpruned network to high sparsities. **I conclude that these subnetworks are indeed a drop-in replacement for the unpruned network: they can train in isolation to at least the same test accuracy as the unpruned network in at most the same number of steps,** and often far fewer steps (less than half in several cases). In contrast, the baselines are only able to do so at the lowest sparsities if at all. The lone exception is random reinitialization on MNIST, which—as described previously—behaves differently than the other benchmarks due to the fact that it is a fully-connected network where weights in the first hidden layer correspond to specific pixels in the input.

This metric supports the same conclusion as the early stopping metric—that IMP subnetworks on these benchmarks are drop-in replacements for the unpruned networks from the beginning of training—but extends it to cases that the early stopping metric (which was the sole metric in the original lottery ticket paper (Frankle & Carbin, 2019)) cannot address.

3.4 Discussion

3.4.1 Summary of Results

In this chapter, I have demonstrated that—for the small-scale image classification benchmarks under consideration—randomly initialized dense networks contain non-trivially sparse winning tickets: subnetworks that can train from initialization to completion on their own, following the same training procedure as would be used for the dense network, and reach at least the same accuracy as the dense network in at most the same number of steps.

These subnetworks are found by a procedure called IMP (iterative magnitude pruning), which uses the magnitudes of the weights from after training the unpruned network to retroactively create a sparse network with these properties. These subnetworks are non-trivially sparse: they are sparser than subnetworks produced by pruning randomly at initialization. Moreover, they are nearly as sparse as those produced by pruning after training (using learning rate rewinding), a procedure which involves extensive fine-tuning that subjects the weights to an order of magnitude more training than the winning tickets found by IMP. Finally, the specific initialization that each connection received and the specific location of the connections in the network are essential for this performance, at least for the subnetworks found by IMP.

This last observation leads to the *lottery ticket* metaphor: these subnetworks have won the initialization lottery, receiving a lucky sample from the initialization distribution that allowed them to train to completion in this way; reinitializing with a new sample from the distribution severely degrades the subnetwork's performance at non-trivial sparsities.

3.4.2 Answers to Research Questions

Referring back to the research questions from Section 1.3 for the benchmarks studied in this chapter:

> *Under what circumstances would it be possible to train a sparse neural network (like that uncovered by pruning after training) in place of a standard dense network?*

All of the benchmarks studied in this chapter contain sparse subnetworks that do so.

> *What are the sparest networks that it is possible to train? How do they compare to other benchmarks (such as the sparsity attainable by pruning after training)? How can we relate the capacity necessary for a neural network to represent a function and the capacity necessary for it to learn it?*

The sparsest networks that it is possible to train while still reaching full accuracy have one to two orders of magnitude fewer weights than the initial dense networks, at least for the settings found in this chapter. They are nearly as sparse as those found by a state-of-the-art method for pruning after training (specifically, learning rate rewinding (Renda et al., 2020)). Insofar as parameter count is a meaningful measure of neural network capacity, these results show that the capacity necessary to train a neural network is at most slightly higher that required to represent an equally useful function (according to our chosen metric, in this case accuracy) that a dense network would otherwise learn to represent. The gulf between capacity necessary to learn and to represent functions is far smaller than envisioned by previous work, and it is possible that improved methods for finding winning tickets further close the gap.

How many steps are necessary to train these networks successfully (relative to the steps necessary to train the corresponding dense networks)?

The sparse subnetworks found by IMP can reach at least the same accuracy as the unpruned networks in at most the same number of training steps. As the training dilation experiments show, they can often do so in *fewer* steps than the dense networks (anywhere from 20% to 70% fewer steps, depending on the benchmark).

What kinds of sparsity patterns and initializations are sufficient for these networks to train successfully? What properties do these sparse networks have?

The subnetworks in this chapter require each weight to be set to the exact value it received at initialization and for each connection to remain in the same location. They are not robust to eliminating either of these aspects of the subnetworks. Note that, as mentioned in the limitations section below, it is entirely possible that this is due to the way in which the subnetworks were found (namely, using IMP) rather than a general property of all winning tickets that might exist.

3.4.3 The Lottery Ticket Hypothesis

Based on the results in this chapter, I articulated the *lottery ticket hypothesis*, which summarizes the central finding of this research and predicts that it holds in general:

The neural networks we typically train have subnetworks (at non-trivial sparsities) at initialization that can train to full accuracy in the same number of steps as the original network.

Note that this is not a statement of fact; it is a hypothesis about the extent to which the results in this chapter are general. There are a few aspects of this hypothesis.

The neural networks we typically train

This is an empirical statement about the neural networks we train in practice: those that are trained on real-world data and have the design choices (like batch normalization, specific learning rate schedules, architectural choices, overparameterization, etc.) to succeed in these scenarios. This is not intended to be a general statement of all possible neural networks and all possible functions they might represent. There are counterexamples if all possible networks are considered. For example, a neural network that is already of the minimal size to represent a particular function cannot be further compressed in this way and—therefore—does not contain the kinds of sparse subnetworks described in this statement; as a concrete example, consider a neural network with two inputs, two hidden units, and one output unit designed for the binary XOR task. This network has the minimal representation necessary to reach perfect accuracy on this task, and removing any connection would necessarily hurt its performance.

Insofar as we consider the kinds of neural networks typically trained in practice, the ideas described in this chapter have been extended to a variety of different settings, including larger-scale networks for computer vision (Frankle et al., 2020a), GANs (Chen et al., 2021; Kalibhat et al., 2021), natural language processing (Yu et al., 2020; Chen et al., 2020a), and reinforcement learning (Yu et al., 2020; Vischer et al., 2021). The text in the hypothesis is intentionally vague, since we currently lack the formalisms necessary to distinguish the small parts of the neural network design space that work in practice from the rest of the design space.

contain subnetworks (at non-trivial sparsities)

The subnetworks I am interested in are those that are "hard" to find, i.e., those that cannot be found simply by randomly pruning at initialization. Research on pruning after training has long shown that non-trivial subnetworks (according to this definition) exist that can represent the functions learned by neural networks after training. The research described in this thesis is intended to study the extent to which those behaviors extend to networks capable of learning functions, not just representing them. For that reason, the same non-triviality constraint applies.

at initialization

All of the subnetworks found in this chapter exist at initialization. That is to say, from the point that training begins, it would have been possible to dramatically reduce the size (and hypothetical cost, as measured in parameters or FLOPs) of the neural network necessary to complete the learning task. This is the fullest realization of the idea that a sparse neural network can learn functions that are drop-in replacements for those learned by dense networks according to some metric (in this case, accuracy), not just represent them after a dense network has learned them. In the following chapter, I will relax this constraint to extend these findings to larger-scale networks on more challenging tasks: I will consider a hybrid situation where the dense network is permitted to learn for some amount of time before the sparse network is created. According to the current state of knowledge in the field, this is necessary to find sparse subnetworks that are capable of learning on their own from initialization (at least, subnetworks with static sparsity patterns; see Evci et al. (2020)). As the limitations section below describes, this does not mean that winning tickets do not exist at initialization in these settings; it may be that IMP is simply insufficient to find them.

that can train to full accuracy in the same number of steps as the original network.

The subnetworks that this statement hypothesizes to exist have two properties:

1. They can train to the same accuracy as the unpruned network from which they were chosen ("full accuracy").
2. They can reach this accuracy by following the same training procedure as the unpruned network and training for at most the same number of steps as the unpruned network received.

This captures the idea that the subnetworks should be drop-in replacements for the unpruned network. In the very worse case (i.e., if it is difficult to fully exploit this sparsity to speed up training), training with this subnetwork should be no worse than training the original, unpruned network. It should reach the same level of quality (in this case, accuracy, but—in other cases—perhaps transferability (Chen et al., 2020a) or other desirable properties) and do so in the same number of steps.

3.4.4 Limitations

Small image classification networks and tasks. The findings in this chapter are based on experiments on small fully-connected and convolutional neural networks for MNIST and CIFAR-10. The behavior of practical neural networks is known to change demonstrably as the size of the network, the amount of data, and the complexity of the task increase. MNIST, in particular, has stood out in the literature on empirical deep learning as an unrepresentative task. As Chapter 4 shows, the behaviors observed in this chapter are a simpler version of the more general behavior at scale.

On the one hand, the small-scale nature of the benchmarks in this chapter should lead to caution about conjectures that the results are general (e.g., the lottery ticket hypothesis). On the other hand, without having first performed the experiments here, the phenomena to be described in larger-scale settings would have been difficult or impossible to identify. Concretely, while the results in this chapter show that the benchmarks under consideration contain winning tickets (i.e., sparse, trainable subnetworks at initialization), there is no evidence for winning tickets in more challenging settings (like ResNet-50 on ImageNet). Rather, IMP finds sparse, trainable subnetworks only after the dense network has undergone some amount of training. Had I started these experiments on ImageNet, I would likely have considered this entire undertaking a negative result: I would not have found any winning tickets. However, by starting small, identifying a phenomenon, and searching for related behavior at scale, I was able to uncover a general version of the phenomenon.

Subnetworks are found retroactively. A common misinterpretation of this research is that it shows how to efficiently find winning tickets before any training has taken place. To be clear, this is not the claim made in this chapter or in any part of this thesis. There is no known way to efficiently find winning tickets at initialization, as I explore extensively in Chapter 5. Instead, the claim in this chapter is that winning tickets exist, and the experiment used to find them (IMP) finds them retroactively after the unpruned network has been trained (many times in the case of iterative pruning). This existential result changes our understanding of the capacity needed for a neural network to learn with standard training procedures, motivating research on finding winning tickets efficiently.

Exploiting sparsity for better performance requires hardware and software support. On paper, sparse subnetworks should train faster than their dense counterparts. After all, eliminating parameters eliminates operations that must be performed on the forward and backward passes and reduces the amount of memory required to store the network weights and optimizer state.

However, eliminating parameters alone is not enough to provide any speedup at all. In the first place, the number of parameters is not proportional to the number of operations performed on the forward and backward passes. Pruning parameters in layers with convolutions will eliminate more operations than pruning parameters in fully-connected layers (since convolutions are applied repeatedly across the activation map). Similarly, pruning parameters in layers with larger activation maps will eliminate more operations. Both of these effects can be measured by computing the number of FLOPs required to perform a forward or backward pass rather than simply looking at the number of parameters.

Even FLOPs are a poor indicator of real-world speedups. Accelerating sparse neural networks is difficult. Merely representing a sparse tensor requires memory overhead that is not captured by parameter count or FLOP count alone. Hardware is designed for the regular matrix operations used by dense networks, and it is difficult to design hardware and software that can accommodate the irregular patterns that emerge from unstructured sparsity. Accelerating sparsity on contemporary hardware is a research area unto itself, and practitioners often use block-sparse pruning patterns that balance the accuracy benefits of unstructured sparsity with the performance benefits of larger-granularity sparsity (e.g., Gray et al., 2017; Elsen et al., 2020).

In recognition of the broad opportunity that sparsity presents to speed up training, hardware manufacturers have developed new chips that include varying degrees of sparsity support. Starting with the Ampere architecture, NVIDIA includes sparse tensor cores that can double the throughput of inference when every block of four weights has two weights pruned. Cerebras and Graphcore claim that their chips are able to speed up unstructured sparsity as well. CPUs are known to be more conducive to sparsity, and NeuralMagic in particular is exploiting this opportunity. Much of the software support to do this easily in high-level frameworks like PyTorch and TensorFlow has yet to be developed, but—thanks to the work of these vendors—the hypothetical speedups that sparsity provides are moving closer to becoming real.

IMP is only one method for finding winning tickets. The results in this chapter are existential. They show that there exist sparse subnetworks with the properties described in the lottery ticket hypothesis, at least for the settings examined in this chapter. These results merely provide a lower bound on our expectations of what might be possible when it comes to training sparse neural networks. Although these subnetworks are quite sparse, there may exist even sparser winning tickets; other methods have been proposed and claims have been made to this effect (e.g., Savarese et al., 2020). Although these subnetworks appear to train quite fast and reach high accuracy, there may exist faster-training and more accurate winning tickets. Although these subnetworks appear to rely on a fortuitous initialization and connectivity pattern, there may exist other subnetworks that are not bound by these constraints. Although IMP finds these subnetworks only after a significant amount of training, there may exist other methods that can find these subnetworks earlier in training in an efficient way. These results are the beginning of our understanding of the kinds of winning tickets might exist.

Empiricism cannot support any general claim. Although the lottery ticket hypothesis posits that such sparse, trainable subnetworks exist in general, and there has been extensive work showing that they exist in other settings (e.g., Yu et al., 2020;

Chen et al., 2021; Kalibhat et al., 2021; Caron et al., 2020b; Vischer et al., 2021), empiricism can never tell us that this is definitively the case for all neural networks (or even all neural networks that we "typically train" as the lottery ticket hypothesis states). There may be many settings where such subnetworks do not exist, or at least many settings where they do not exist at initialization (see Chapter 4).

Recently, several researchers have studied ways to mathematically formalize the lottery ticket behaviors (Malach et al., 2020; Pensia et al., 2020). However, this research has focused on the performance of sparse subnetworks found by pruning randomly initialized dense networks (Zhou et al., 2019; Ramanujan et al., 2020). Namely, this research proves that, for any given neural network, there exists a larger, randomly initialized neural network that contains a sparse subnetwork that matches the functional behavior of the first network up to some distance ϵ. Although this research does make progress toward understanding the expressive power of sparse neural networks, in my view it does not yet capture the most interesting part of the research described in this chapter: the ability of sparse networks to learn. Instead, it focuses on representation alone, skirting questions of the capacity necessary to learn a function vs. to represent it.

3.4.5 Implications

Can we find these subnetworks efficiently? Creating a method that can uncover these subnetworks at the moment they come into existence in a way that adds little cost to the overall training process is the holy grail of research on lottery tickets. At the time of submission of this thesis, I know of no efficient method for finding subnetworks that reach the same performance as the subnetworks found by IMP after training. By an efficient method, I mean one that finds these subnetworks at the moment they come into existence and meaningfully reduces the overall cost of training (even according to metrics like FLOPs or parameter-epochs, both weaker constraints than wall-clock time). There are several proposed methods that claim to make progress on this question (e.g., Lee et al., 2019; Wang et al., 2020; Tanaka et al., 2020), but none currently meet this standard. Chapter 5 studies this question in detail and summarizes the state of research in this area.

Alternatively, perhaps it is hard to find such subnetworks. For example, perhaps IMP "cheats" by conveying information about the state of the trained network back to initialization and, without this information, it is impossible to determine which parts of the network are superfluous until much or all of training is complete. A hardness result would be equally scientifically interesting, albeit practically disappointing.

Finally, there may be ways to reduce the strength of the conditions necessary to reduce the cost of training using sparsity. For example, perhaps we could create an interim sparsity pattern at initialization, train that subnetwork for a short time, and change the sparsity pattern mid-stream using what was learned during the first phase of training. That would provide an equivalently efficient sparse training strategy, but it might be less difficult to identify the appropriate subnetwork after some training rather than at initialization. Research on *dynamic sparsity* (e.g., Evci et al., 2020) considers this more general problem, and it may provide an alternate path to efficient training via sparsity.

Can we exploit these findings? Even if we cannot find these subnetworks efficiently, are there still ways to make productive use of the findings in this chapter? One immediate approach is to find these subnetworks inefficiently (i.e., with IMP) and amortize this cost by reusing the subnetworks many times. If these subnetworks are able to transfer, e.g., to data with a slightly different distribution than the training data used to find the subnetwork, they could serve as a more efficient platform for future learning. For example, in industrial settings where recommender systems are frequently re-trained to keep up with trending topics, replacing a dense network with a winning ticket could reduce the time needed to deploy these models (assuming it is possible to accelerate sparse computation; see the limitations section below). Morcos et al. (2019) show that winning tickets found for one computer vision setting transfer to other computer vision settings, particularly from more challenging settings like ImageNet to less challenging settings like CIFAR-10. Chen et al. (2020a) show that it is possible to find sparse *transferable* subnetworks within BERT that are capable of transferring to all of the GLUE tasks to the same extent as the unpruned BERT.

The role of weight sparsity in deep learning. The experiments described in this chapter (and all of the experiments in this thesis) involve pruning individual weights from neural networks. There are a variety of other pruning methods that prune at larger granularities like convolutional filters (He et al., 2017); for a detailed survey of many methods for doing so, see Liu et al. (2019). Pruning at larger granularities is valuable because it is easier to exploit these sparsity patterns for real-world speedup. For example, pruning convolutional filters simply reduces the size of tensors in the network, providing an immediate speedup.

The sparse neural networks found by pruning individual weights and by pruning at larger granularities have entirely different properties. For example, concurrent to the original lottery ticket paper (Frankle & Carbin, 2019), Liu et al. (2019) published a paper stating that pruned neural networks could be reinitialized without any effect on accuracy—the opposite finding from the experiments described in this chapter. The difference? Liu et al. focus on pruning at large granularities, while I focused on pruning individual weights. Interpolating between these two settings, Siswanto (2021) shows that, as the granularity of pruning increases (from weight pruning to block-sparse pruning of larger sizes to filter pruning), the performance of the sparse network decreases to the point where it is no better than random pruning and the importance of initialization fades away.

In this sense, either (1) we have yet to develop an effective large-granularity pruning method or (2) neural networks seem naturally amenable to weight sparsity. To embellish (2) further, it seems that the functions learned by neural networks and—as this thesis shows—the learning process itself do not need every connection between neurons. Reducing the internal representation capacity of the network by pruning neurons (e.g., by pruning convolutional filters) seems much more detrimental than pruning the connections between those neurons.

To speculate, perhaps the representations learned by neural networks are inherently sparse, with the concepts learned by individual neurons relying on few other neurons in the preceding layers. If this is the case, the dense neural networks that we typically train inherently have excess capacity, and weight pruning is uncovering the natural structure of the representations. This could be intrinsic to the functions we ask neural networks to learn, it could be a property of design choices we make in architecting neural networks (e.g., using activation functions or finite-bit numbers that limit the capacity of individual neurons), or it could be a tendency of stochastic gradient descent, the algorithm we use to optimize the networks. This is a research topic that is ripe for future investigation, and the results presented in this thesis suggest that sparsity could be natural—not just to the final representations learned by practical neural networks—but to the learning process as well.

What makes a good initialization? The winning tickets studied in this section owe their performance to the specific sample from the initialization distribution. Frankle et al. (2020b) attempted to extract useful distributions from these initializations without success, suggesting that these winning ticket initializations cannot be generated from an easy-to-find distribution. This finding stands in contrast to recent thinking on neural network initializations, which focus on creating distributions from which samples will, in expectation, preserve desirable behaviors at initialization like avoiding vanishing or exploding gradients (Glorot & Bengio, 2010; He et al., 2015; Zhang et al., 2019). The research presented in this chapter suggests that the specific sample from the distribution is important, at least for the sparse subnetworks found by IMP. It is possible that these sparse subnetworks are more sensitive to initialization, and the results in the subsequent chapters lend credence to this hypothesis. Alternatively, it is possible that dense, overparameterized networks are so successful precisely because they contain combinatorially many subnetworks with different (overlapping) samples from the initialization distribution, some of which are bound to exhibit the beneficial properties that make the winning tickets found by IMP so successful. (I referred to this idea as the *lottery ticket conjecture* in the original lottery ticket paper.)

Thinking on initialization has continued to evolve since the original lottery ticket paper was published. Work on self-supervised pre-training is, in my view, really about initialization: creating a good starting point for training on a subsequent task to be successful. Just as with the subnetworks found in this chapter, there is no distribution that describes these initializations; rather, the important part is the specific set of initial weights created by pre-training. These initializations themselves give rise to useful sparse, trainable subnetworks of the kind described in this chapter (see Chen et al. (2020a) for further discussion of this topic).

Chapter 4

Linear Mode Connectivity and the Lottery Ticket Hypothesis

In this chapter, I scale up the observations in Chapter 3 to larger scale settings like ResNet-50 and Inception-v3 on ImageNet. The behaviors observed in the previous chapter were already starting to break down in some of the more challenging settings (datasets like CIFAR-10, deeper networks like VGG and ResNet). At these scales, those behaviors break down completely. In its place, a new, more nuanced story arises: that IMP finds sparse, trainable subnetworks early in training rather than at initialization. In parallel, a deeper understanding of the lottery ticket phenomena also emerges: that sparse networks are sensitive to the stochasticity of SGD.

4.1 Linear Mode Connectivity

Before proceeding to scale up the lottery ticket results, I need to take a detour into the behavior of neural networks on the loss landscape. This investigation begins with a deceivingly simple question:

When training a neural network with mini-batch stochastic gradient descent (SGD), training examples are presented to the network in a random order within each epoch. In many cases, each example also undergoes random data augmentation. This randomness can be seen as noise that varies from training run to training run and

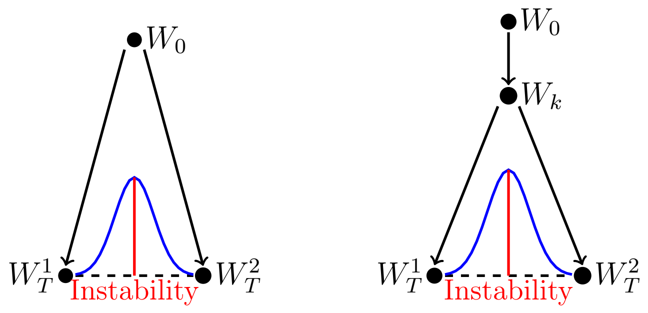


Figure 4-1: A diagram of instability analysis from step 0 (left) and step k (right) when comparing networks using linear interpolation.

alters the network's trajectory through the optimization landscape, even when the initialization and hyperparameters are fixed. How does this SGD noise affect the outcome of optimizing neural networks? What range of outcomes are possible due to different samples of this noise? Is there a point in training at which this noise no longer affects the outcome of training?

Instability analysis. To study these questions, I propose *instability analysis*. The goal of instability analysis is to determine whether the outcome of optimizing a particular neural network is *stable to SGD noise*. Figure 4-1 (left) visualizes instability analysis. First, create a network $\mathcal{N}$ with random initialization W_0. Then train two copies of $\mathcal{N}$ with different samples of SGD noise (i.e., different random data orders and augmentations). Finally, compare the resulting networks to measure the effect of these different samples of SGD noise on the outcome of optimization. If the networks are sufficiently similar according to a criterion, determine $\mathcal{N}$ to be stable to SGD noise. I also study this behavior starting from the state of $\mathcal{N}$ at step k of training (Figure 4-1 right). Doing so allows makes it possible to determine *when* the outcome of optimization becomes stable to SGD noise.

There are many possible ways to compare the networks that result from instability analysis (Appendix 4.1.5). I use the behavior of the optimization landscape along the line between these networks (blue in Figure 4-1). Does error remain flat or even decrease (meaning the networks are in the same, linearly connected minimum), or is there a barrier of increased error? I define the *linear interpolation instability* of

Network	Variant	Dataset	Params	Train Steps	Batch	Accuracy	Optimizer	Rate	Schedule	Warmup	BatchNorm	Pruned Density	Style
LeNet		MNIST	266K	24K Iters	60	98.3 ± 0.1%	adam	12e-4	constant	0	No	3.5%	Iterative
ResNet-20	Standard					91.7 ± 0.1%		0.1		0		16.8%	
ResNet-20	Low	CIFAR-10	274K	63K Iters	128	88.8 ± 0.1%	momentum	0.01	10x drop at32K, 48K	0	Yes	8.6%	Iterative
ResNet-20	Warmup					89.7 ± 0.3%		0.03		30K		8.6%	
VGG-16	Standard					93.7 ± 0.1%		0.1		0		1.5%	
VGG-16	Low	CIFAR-10	14.7M	63K Iters	128	91.7 ± 0.1%	momentum	0.01	10x drop at32K, 48K	0	Yes	5.5%	Iterative
VGG-16	Warmup					93.4 ± 0.1%		0.1		30K		1.5%	
ResNet-50		ImageNet	25.5M	90 Eps	1024	76.1 ± 0.1%	momentum	0.4	10x drop at 30,60,80	5 Eps	Yes	30%	One-Shot
Inception-v3		ImageNet	27.1M	171 Eps	1024	78.1 ± 0.1%	momentum	0.03	linear decay to 0.005	0	Yes	30%	One-Shot

Table 4.1: The networks and hyperparameters I study in this chapter. Accuracies are the means and standard deviations across three initializations. Hyperparameters for ResNet-20 standard are from He et al. (2016). Hyperparameters for VGG-16 standard are from Liu et al. (2019). Hyperparameters for *low*, *warmup*, and LeNet are adapted from Chapter 3. Hyperparameters for ImageNet networks are from Google's reference TPU code (Google, 2018).

$\mathcal{N}$ to SGD noise as the maximum increase in error along this path (red). I consider $\mathcal{N}$ stable to SGD noise if error does not increase along this path, i.e., instability ≈ 0. This means $\mathcal{N}$ will find the same, linearly connected minimum regardless of the sample of SGD noise.

By linearly interpolating at the end of training in this fashion, I assess a linear form of *mode connectivity*, a phenomenon where the minima found by two networks are connected by a path of nonincreasing error. Freeman & Bruna (2017), Draxler et al. (2018), and Garipov et al. (2018) show that the modes of standard vision networks trained from different initializations are connected by nonlinear paths of constant error or loss. Based on this work, one can reasonably expect that all networks we examine are connected by such paths. However, the modes found by Draxler et al. and Garipov et al. are not connected by *linear* paths. The only extant example of linear mode connectivity from the start of training (without inducing the behavior, e.g., (Wortsman et al., 2021a)) is by Nagarajan & Kolter (2019), who train MLPs from the same initialization on disjoint subsets of MNIST and find that the resulting networks are connected by linear paths of constant test error. In contrast, I explore linear mode connectivity from points throughout training, I do so at larger scales, and we focus on different samples of SGD noise rather than disjoint samples of data.

I perform instability analysis on standard networks for MNIST, CIFAR-10, and ImageNet. All but the smallest MNIST network are unstable to SGD noise at initialization according to linear interpolation. However, by a point early in training

(3% for ResNet-20 on CIFAR-10 and 20% for ResNet-50 on ImageNet), all networks become stable to SGD noise. From this point on, the outcome of optimization is determined to a linearly connected minimum.

Context: The lottery ticket hypothesis. In the following subsection, I will show why this is relevant to the main story of this thesis: the Lottery Ticket Hypothesis (LTH). As discussed in the previous chapter, the LTH contains my conjecture that, at initialization, neural networks contain sparse subnetworks that can train in isolation to full accuracy.

Empirical evidence for the LTH consists of experiments using a procedure that I referred to as *iterative magnitude pruning* (IMP) - Algorithm 1. The previous chapter has described how, on small networks for MNIST and CIFAR-10, IMP retroactively finds subnetworks at initialization that can train to the same accuracy as the full network (I call such subnetworks *matching*). Importantly, IMP finds matching subnetworks at *nontrivial* sparsity levels, i.e., those beyond which subnetworks found by trivial random pruning are matching. In more challenging settings, however, there is no empirical evidence for the LTH: IMP subnetworks of VGGs and ResNets on CIFAR-10 and ImageNet are not matching at nontrivial sparsities (Chapter 3 Liu et al., 2019; Gale et al., 2019).

Here, I will show that instability analysis distinguishes known cases where IMP succeeds and fails to find matching subnetworks at nontrivial sparsities, providing the first basis for understanding the mixed results in the literature. Namely, IMP subnetworks are only matching when they are stable to SGD noise according to linear interpolation. Using this insight, I will identify new scenarios where it is possible to find sparse, matching subnetworks at nontrivial sparsities, including in more challenging settings (e.g., ResNet-50 on ImageNet). In these settings, sparse IMP subnetworks become stable to SGD noise *early* in training rather than at initialization, just as found with the unpruned networks. Moreover, these stable IMP subnetworks are also matching. In other words, early in training (if not at initialization), sparse subnetworks emerge that can complete training in isolation and reach full accuracy.

These findings shed new light on neural network training dynamics, hint at possible mechanisms underlying lottery ticket phenomena, and extend the lottery ticket observations to larger scales.

4.1.1 Preliminaries and Methodology

Instability analysis. To perform instability analysis on a network $\mathcal{N}$ with weights W, I make two copies of $\mathcal{N}$ and train them with different random samples of SGD noise (i.e., different data orders and augmentations), producing trained weights W_T^1 and W_T^2. I compare these weights with a function, producing a value I call the *instability* of $\mathcal{N}$ to SGD noise. I then determine whether this value satisfies a criterion indicating that $\mathcal{N}$ is stable to SGD noise. The weights of $\mathcal{N}$ could be randomly initialized ($W = W_0$ in Figure 4-1) or the result of k training steps ($W = W_k$).

Formally, I model SGD by function $\mathcal{A}^{s \to t} : \mathbb{R}^D \times U \to \mathbb{R}^D$ that maps weights $W_s \in \mathbb{R}^D$ at step s and SGD randomness $u \sim U$ to weights $W_t \in \mathbb{R}^D$ at step t by training for $t - s$ steps ($s, t \in \{0, .., T\}$). Algorithm 3 outlines instability analysis with a function $f : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$.

Algorithm 3 Compute instability of W_k with function f.

```
1: Train $W_k$ to $W_T^1$ with noise $u_1 \sim U$: $W_T^1 = \mathcal{A}^{k \to T}(W_k, u_1)$
2: Train $W_k$ to $W_T^2$ with noise $u_2 \sim U$: $W_T^2 = \mathcal{A}^{k \to T}(W_k, u_2)$
3: Return $f(W_T^1, W_T^2)$, i.e., the instability of $W_k$ to SGD noise.
```

Linear interpolation. Consider a path p on the optimization landscape between networks W_1 and W_2. Let the *error barrier height* of p be the maximum increase in error from that of W_1 and W_2 along path p. For instability analysis, I use as the function f the error barrier height along the *linear* path between two networks W_1 and W_2.[1]

Formally, let $\mathcal{E}(W)$ be the (train or test) error of a network with weights W. Let $\mathcal{E}_\alpha(W_1, W_2) = \mathcal{E}(\alpha W_1 + (1 - \alpha) W_2)$ for $\alpha \in [0, 1]$ be the error of the network created by linearly interpolating between W_1 and W_2. Let $\mathcal{E}_{\mathsf{sup}}(W_1, W_2) =$

[1]See Appendix 4.1.5 for alternate ways of comparing the networks.

$\sup_\alpha \mathcal{E}_\alpha(W_1, W_2)$ be the highest error when interpolating in this way. Finally, let $\bar{\mathcal{E}}(W_1, W_2) = \mathsf{mean}(\mathcal{E}(W_1), \mathcal{E}(W_2))$. The error barrier height on the linear path between W_1 and W_2 (which is my function f for instability analysis) is $\mathcal{E}_{\mathsf{sup}}(W_1, W_2) - \bar{\mathcal{E}}(W_1, W_2)$ (red line in Figure 4-1). When performing instability analysis on a network $\mathcal{N}$ with this function, I call this quantity the *linear interpolation instability* (shorthand: *instability*) of $\mathcal{N}$ to SGD noise.

Linear mode connectivity. Two networks W_1 and W_2 are *mode connected* if there exists a path between them along which the error barrier height ≈ 0 (Draxler et al., 2018; Garipov et al., 2018). They are *linearly mode connected* if this is true along the linear path. For instability analysis, I consider a network $\mathcal{N}$ stable to SGD noise (shorthand: *stable*) when the networks that result from instability analysis are linearly mode connected; that is, when the linear interpolation instability of $\mathcal{N} \approx 0$. Otherwise, it is unstable to SGD noise (shorthand: *unstable*). Empirically, I consider instability $< 2\%$ to be stable; this margin accounts for noise and matches increases in error along paths found by Draxler et al. (2018, Table B.1) and Garipov et al. (2018, Table 2). I use 30 evenly-spaced values of α, and I average instability from three initializations and three runs per initialization (nine combinations total).

4.1.2 Instability Analysis of Unpruned Networks

First, I perform instability analysis on the standard networks in Table 4.1 from many points during training. For now, these are the full, unpruned networks. (Pruning will appear in Section 4.2. I will show that, although only LeNet is stable to SGD noise at initialization, every network becomes stable early in training, meaning the outcome of optimization from that point forward is determined to a linearly connected minimum.

Instability analysis at initialization. I first perform instability analysis from initialization. I use Algorithm 3 with W_0 (visualized in Figure 4-1 left): train two copies of the same, randomly initialized network with different samples of SGD noise. Figure 4-2 shows the train (purple) and test (red) error when linearly interpolating

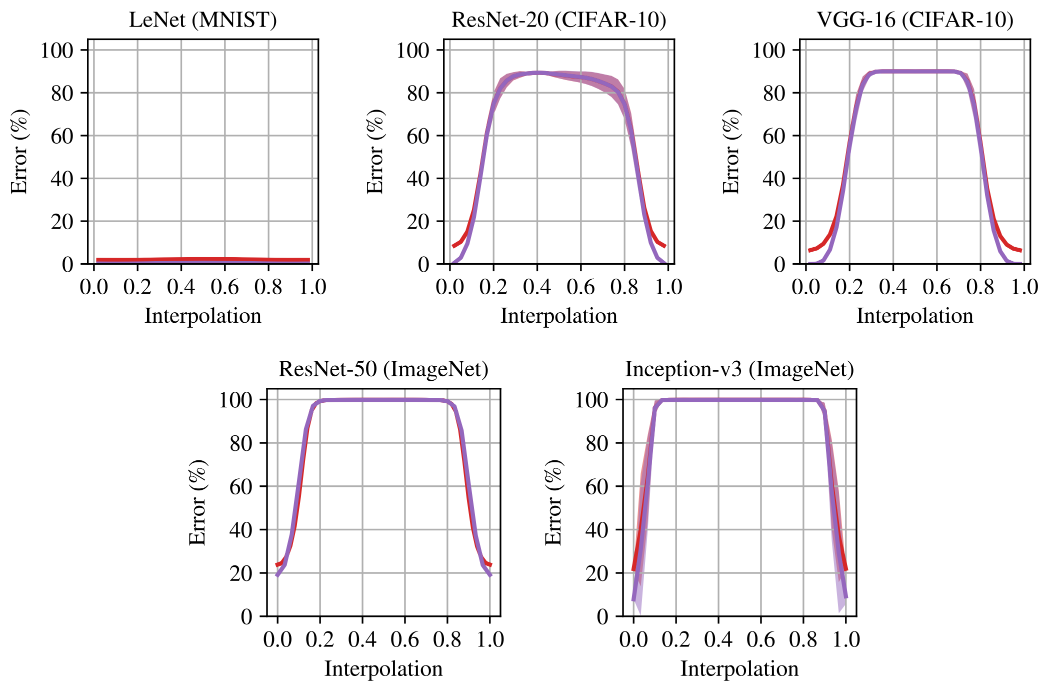


Figure 4-2: Error when linearly interpolating between networks trained from the same initialization with different SGD noise. Lines are means and standard deviations over three initializations and three data orders (nine samples total). Trained networks are at 0.0 and 1.0.

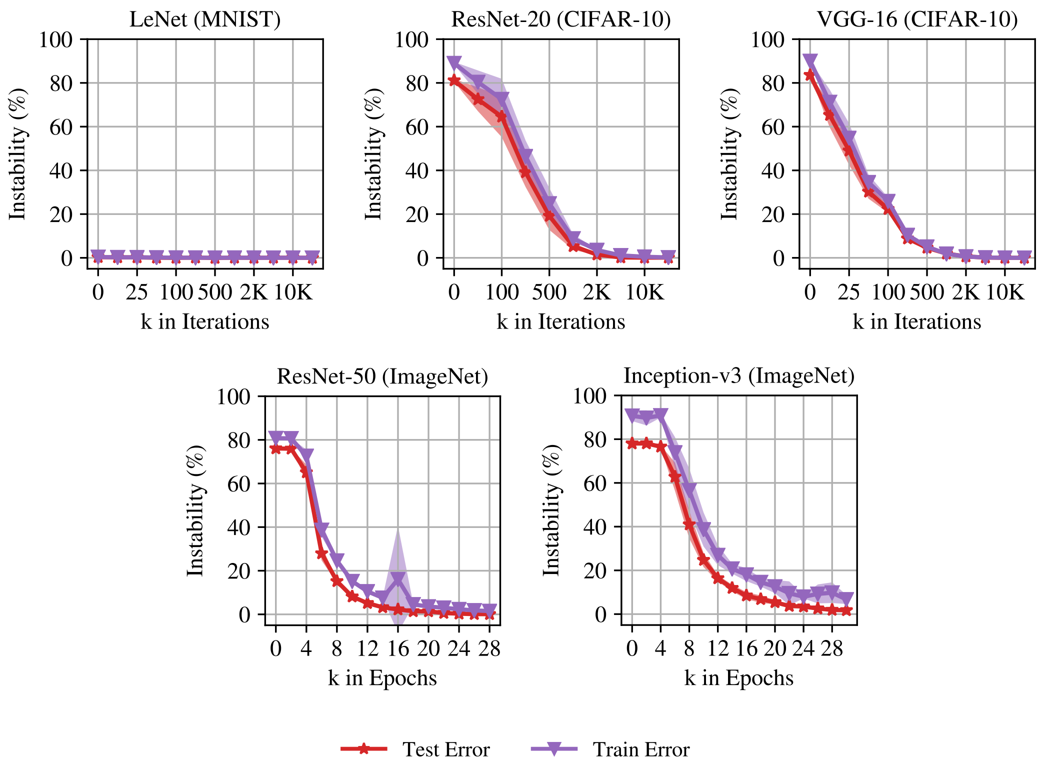


Figure 4-3: Linear interpolation instability when starting from step k. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total).

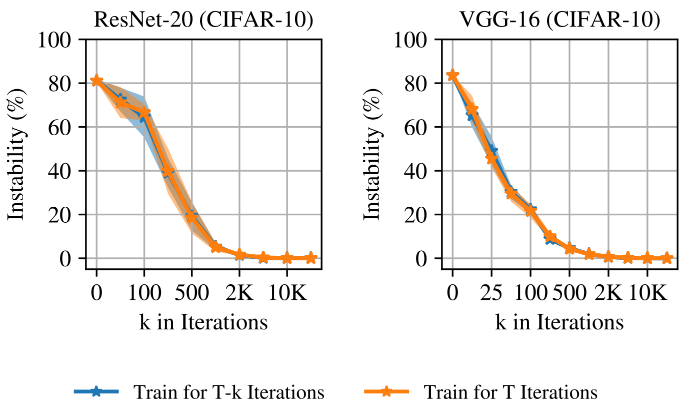


Figure 4-4: Linear interpolation instability on the test set when making two copies of the state of the network at step k and either (1) training for the remaining $T - k$ steps (blue) or (2) training for T steps with the learning rate schedule reset to step 0 (orange).

between the minima found by these copies. Except for LeNet (MNIST), none of the networks are stable at initialization. In fact, train and test error rise to the point of random guessing when linearly interpolating. LeNet's error rises slightly, but by less than a percentage point. The takeaway is that, in general, larger-scale image classification networks are unstable at initialization according to linear interpolation.

Instability analysis during training. Although larger networks are unstable at initialization, they may become stable at some point afterwards; for example, in the limit, they will be stable trivially after the last step of training. To investigate when networks become stable, I perform instability analysis using the state of the network at various training steps. That is, I train a network for k steps, make two copies, train them to completion with different samples of SGD noise, and linearly interpolate (Figure 4-1 right). I do so for many values of k, assessing whether there is a point after which the outcome of optimization is determined to a linearly connected minimum regardless of SGD noise.

For each k, Figure 4-3 shows the linear interpolation instability of the network at step k, i.e., the maximum error during interpolation (the peaks in Figure 4-2) minus the mean of the errors of the two networks (the endpoints in Figure 4-2). In all cases, test set instability decreases as k increases, culminating in stable networks. The steps at which networks become stable are early in training. For example, they do so at iterations 2000 for ResNet-20 and 1000 VGG-16; in other words, after 3% and 1.5% of training, SGD noise does not affect the final linearly connected minimum. ResNet-50 and Inception-v3 become stable later: at epoch 18 (20% into training) and 28 (16%), respectively, using the test set.

For LeNet, ResNet-20, and VGG-16, instability is essentially identical when measured in terms of train or test error, and the networks become stable to SGD noise at the same time for both quantities. For ResNet-50 and Inception-v3, train instability follows the same trend as test instability but is slightly higher at all points, meaning train set stability occurs later for ResNet-50 and does not occur in our range of analysis for Inception-v3.

Disentangling instability from training time. Varying the step k from which I run instability analysis has two effects. First, it changes the state of the network from which two copies are trained to completion on different SGD noise. Second, it changes the number of steps for which those copies are trained: when running instability analysis from step k, the copies are trained under different SGD noise for $T-k$ steps. As k increases, the copies have fewer steps during which to potentially find linearly unconnected minima. It is possible that the gradual decrease in instability as k increases and the eventual emergence of linear mode connectivity is just an artifact of these shorter training times.

To disentangle the role of training time in our experiments, I modify instability analysis to train the copies for T iterations no matter the value of k. When doing so, I reset the learning rate schedule to iteration 0 after making the copies. In Figure 4-4, I compare instability with and without this modification for ResNet-20 and VGG-16 on CIFAR-10. Instability is indistinguishable in both cases, indicating that the different numbers of training steps did not play a role in the earlier results. Going forward, I present all results by training copies for $T - k$ steps as in Algorithm 3.

4.1.3 Instability Throughout Training

In Section 4.1.2, I found stable networks that arrive at minima that are linearly connected. Here, I broaden this question, studying whether the trajectories they follow are also linearly connected. In other words, when training two copies of the same network with different noise, are the states of the network at each step t connected by a linear path over which test error does not increase? In Section 4.1.2, I studied this quantity only at the end of training (i.e., $t = T$). Here, I study it for all iterations t *throughout* training. To study this behavior, I linearly interpolate between the networks at each epoch of training and compute instability.

Figure 4-5 plots instability throughout training for ResNet-20 and VGG-16 from different rewinding iterations k for both train and test error. I begin with the unpruned networks. For $k = 0$ (blue line), instability increases rapidly. In fact, it follows the same pattern as error: as the train or test error of each network decreases, the maximum possible instability increases (since instability never exceeds random guessing). With larger values of k, instability increases more slowly throughout training. When k is sufficiently large that the networks are stable at the end of training, they are generally stable at every epoch of training ($k = 2000$, pink line). In other words, after iteration 2000, the networks follow identical optimization trajectories modulo linear interpolation.

4.1.4 Full Linear Interpolation Data

In Figure 4-3, I plot the instability value derived from linearly interpolating between copies of the same network or subnetwork trained on different data orders. In Figure 4-6, I plot the linear interpolation data from which I derived the instabilities in Figure 4-3. This is the raw data that was used to create the instability plots in Figure 4-3.

4.1.5 Alternate Distance Functions

Instability analysis involves training two copies of the same network on different data orders and comparing the networks that result. In Section 4.1.2, the method of comparison is linear interpolation, which I will later show to offer valuable new insights into neural network optimization and the lottery ticket hypothesis. However, one could parameterize instability analysis with a wide range of other functions for comparing pairs of neural networks. Here, I discuss four alternate methods for which I collected data using the MNIST and CIFAR-10 networks.

L_2 **Distance.** One simple way to compare neural networks is to measure the L_2 distance between the trained weights. The limitation of this function is that there is not necessarily any relationship between L_2 distance and the functional similarity of networks or the structure of the loss landscape. In other words, there is no clear interpretation of L_2 distance.

In Figure 4-7, I plot the L_2 distance function at all rewinding points for the unpruned networks. Distance decreases linearly as we logarithmically increase the

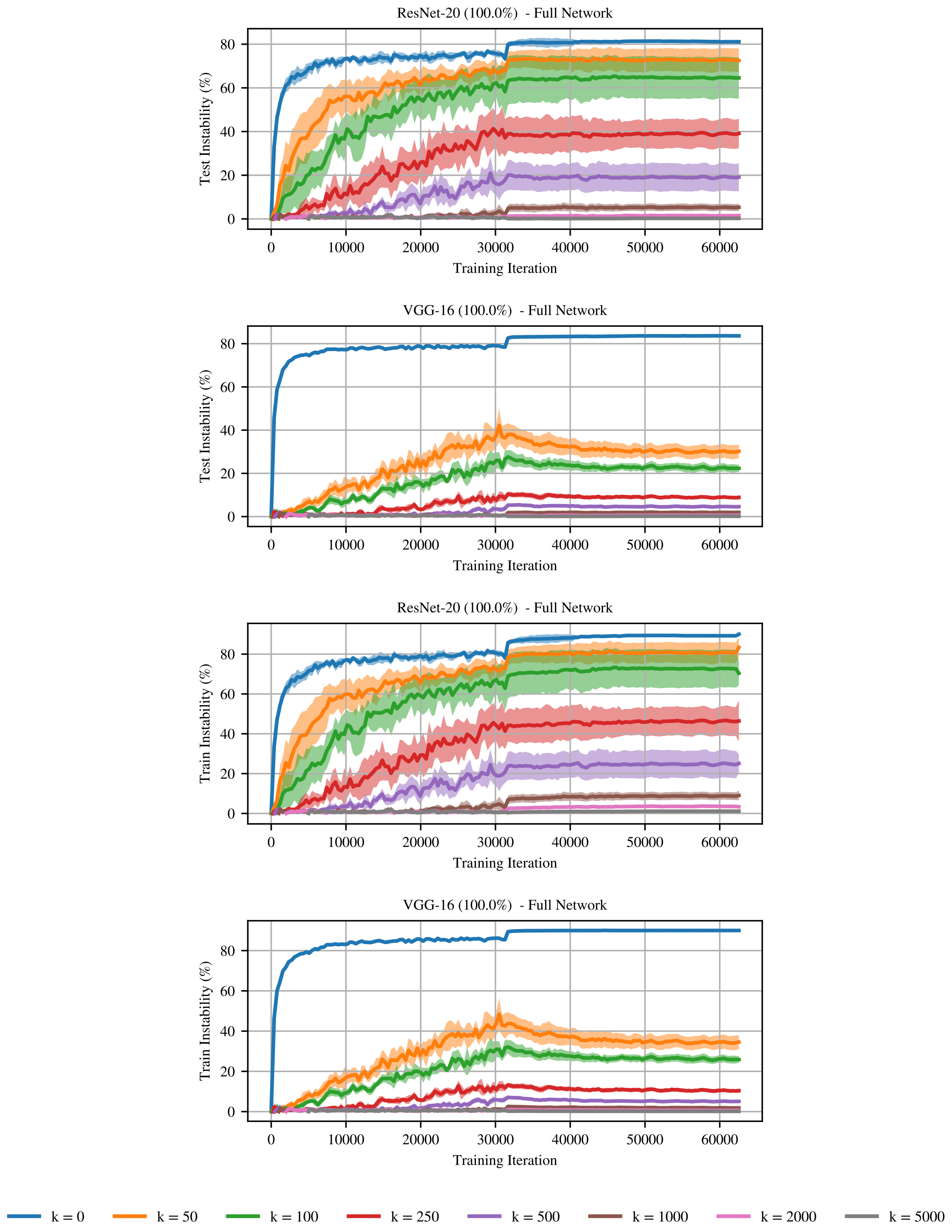


Figure 4-5: Instability throughout training for ResNet-20 and VGG-16 using both the unpruned networks as computed on both the test set and train set. Each line involves training to iteration k and then training two copies on different data orders after. Each point is the instability when interpolating between the states of the networks at the training iteration on the x-axis.

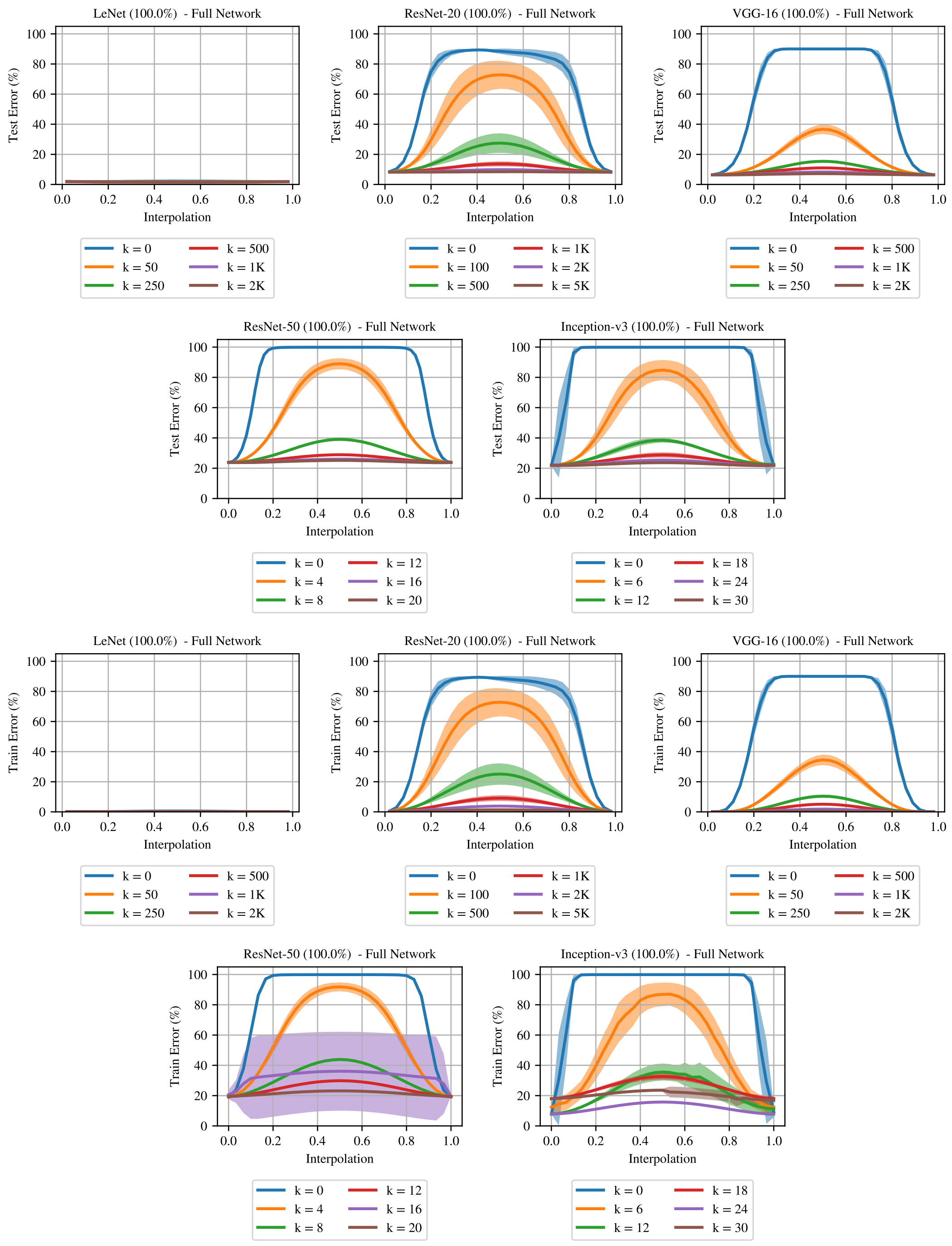


Figure 4-6: The error when linearly interpolating between the minima found by randomly initializing a network, training to iteration k, and training two copies from there to completion using different data orders. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). The errors of the trained networks are at interpolation = 0.0 and 1.0.

rewinding iteration. I see no distinct changes in behavior when the networks become stable, and the L_2 distance remains far from 0 at this point.

Cosine distance. In Figure 4-8, I plot the cosine distance in a manner similar to L_2 distance. The results are similar, and the same interpretation applies.

Classification differences. This distance function computes the number of examples that are classified differently by two networks. Unlike linear interpolation and L_2/cosine distance, this function looks at the functional behavior of the networks rather than the parameterizations. This function is particularly valuable because it allows us to compare the dense and sparse networks directly.

In Figures 4-9 (test set) and 4-10 (train set), the red line shows this function for the networks across values of k. The networks generally classify the same number of examples differently no matter the value of k, although the number of different classifications decreases gradually for the latest rewinding iterations for ResNet-20 low and warmup. I see no relationship between this function and instability.

Loss L_2 distance. This distance function computes the L_2 distance between the vector of cross-entropy losses aggregated by computing the loss on each example. This function again considers only the functional behavior of the networks, but it uses the per-example loss rather than the classification decisions, which may provide more information about the functional behavior of the networks. I plot this data in Figures 4-11 (test set) and 4-12 (train set). It largely mirrors the behavior from the classification difference function, and the same interpretations apply.

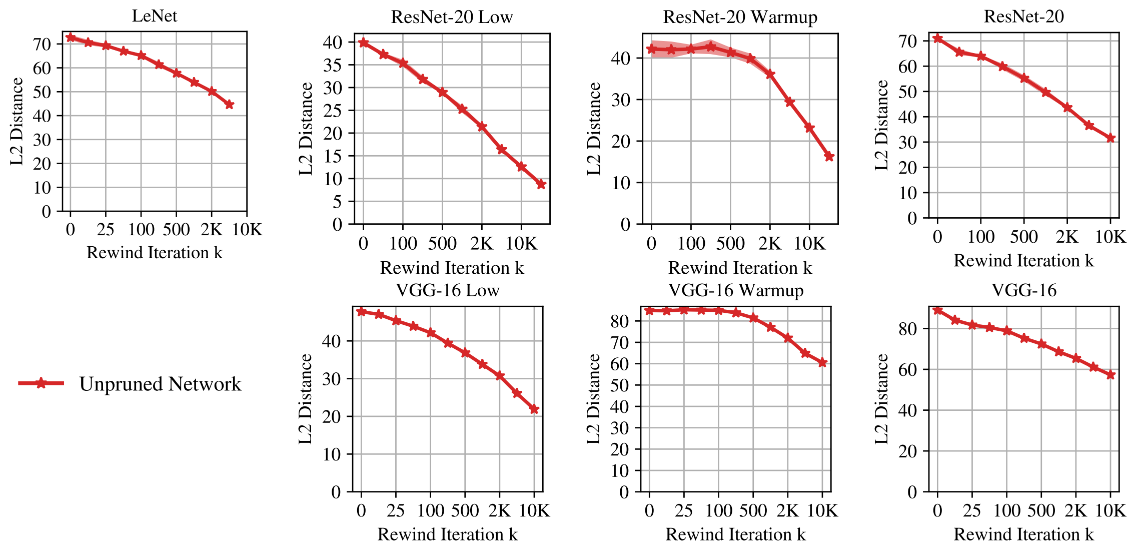


Figure 4-7: The L_2 distance between networks that are created by trained to iteration k, making two copies, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining. I did not compute the train set quantities for the ImageNet networks due to computational limitations.

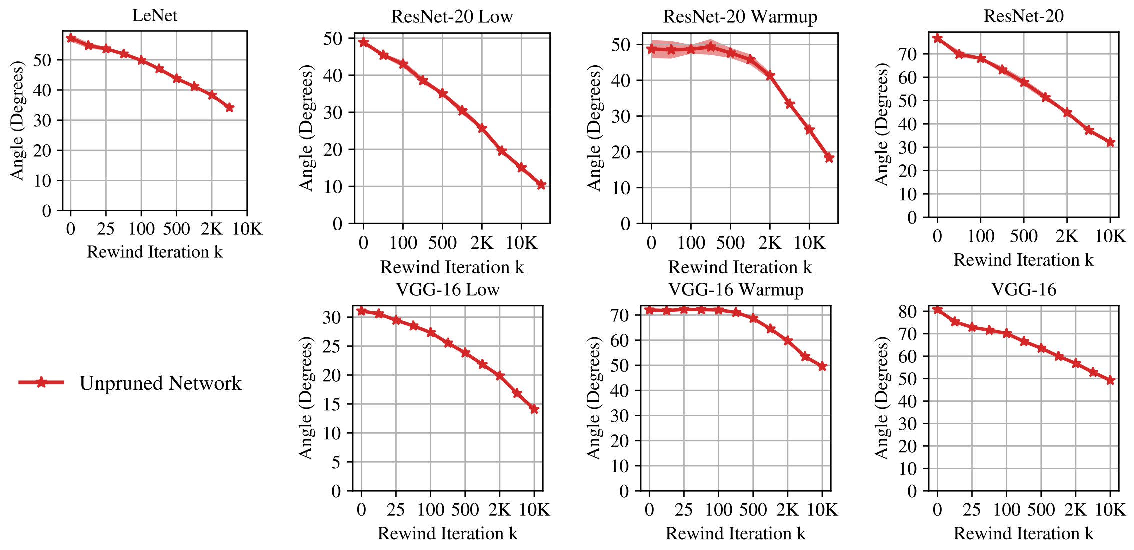


Figure 4-8: The cosine distance between networks that are created by trained to iteration k, making two copies, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining. I did not include the train set for the ImageNet networks due to computational limitations.

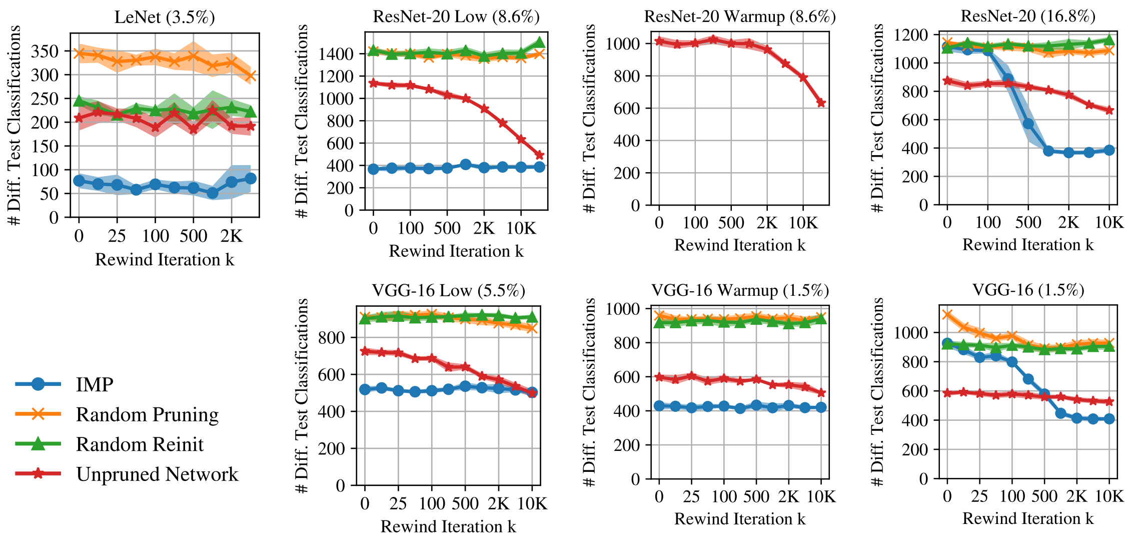


Figure 4-9: The number of different test set classifications between networks that are created by training the full network to iteration k, optionally applying a pruning mask, making two copies, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining.

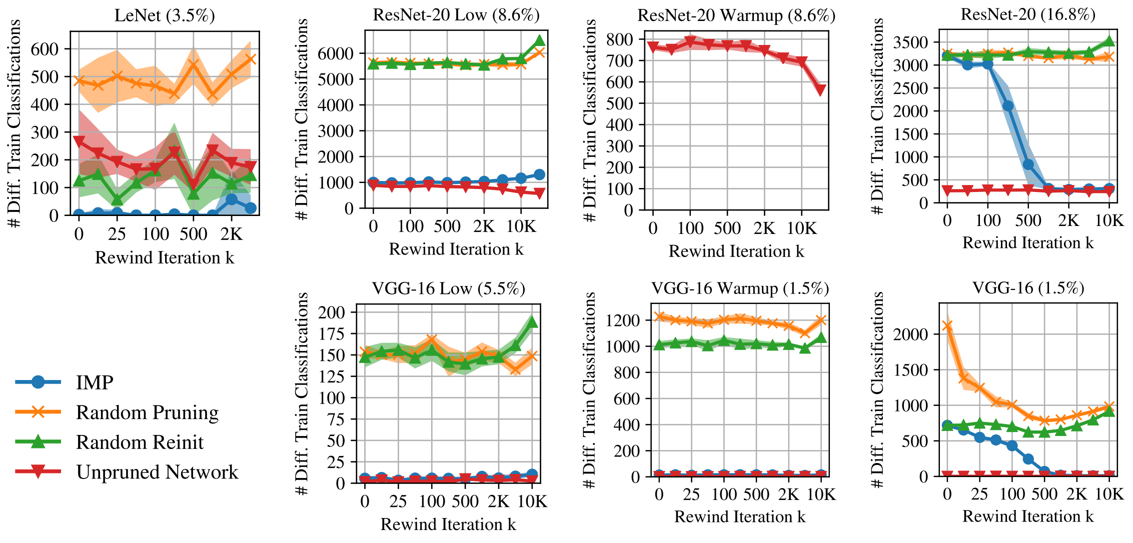


Figure 4-10: The number of different train set classifications between networks that are created by training the full network to iteration k, optionally applying a pruning mask, making two copies, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining.

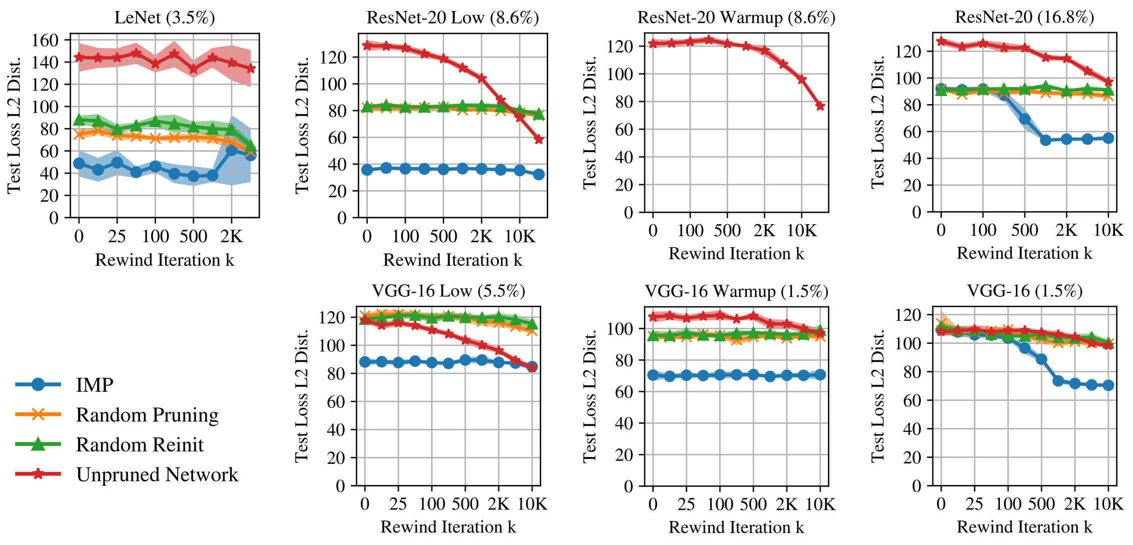


Figure 4-11: The L_2 distance between the per-example losses on the test set for networks that are created by training the full network to iteration k, optionally applying a pruning mask, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining. I did not compute the train set quantities for the ImageNet networks due to computational limitations.

4.1.6 Conclusions on Linear Mode Connectivity

The experiments in this section showed that there is a point in training at which all of the networks studied in this section become robust to SGD noise and consistently find solutions in the same convex region of the loss landscape. Interestingly, this generally occurs—not at the very beginning or end of training—but somewhere in between. Specifically, this takes place during the early part of training in all cases, even the largest-scale ones.

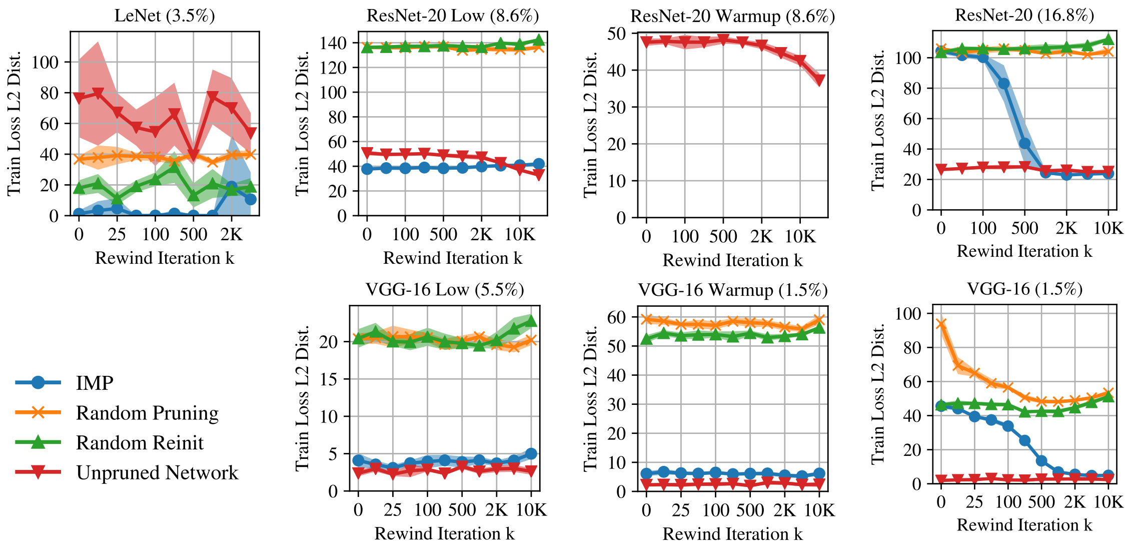


Figure 4-12: The L_2 distance between the per-example losses on the train set for networks that are created by training the full network to iteration k, optionally applying a pruning mask, making two copies, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining. I did not compute the train set quantities for the ImageNet networks due to computational limitations.

Network	Full	IMP	Rand Prune	Rand Reinit	Δ IMP	Matching?
LeNet	98.3	98.2	96.7	97.5	0.1	Y
ResNet-20	91.7	88.5	88.6	88.8	3.2	N
ResNet-20 Low	88.8	89.0	85.7	84.7	-0.2	Y
ResNet-20 Warmup	89.7	89.6	85.7	85.6	0.1	Y
VGG-16	93.7	90.9	89.4	91.0	2.8	N
VGG-16 Low	91.7	91.6	90.1	90.2	0.1	Y
VGG-16 Warmup	93.4	93.2	90.1	90.7	0.2	Y
ResNet-50	76.1	73.7	73.1	73.4	2.4	N
Inception-v3	78.1	75.7	75.2	75.5	2.4	N

Table 4.2: Accuracy of IMP and random subnetworks when rewinding to $k = 0$ at the sparsities in Table 4.1. Accuracies are means across three initializations. All standard deviations are < 0.2.

4.2 The Lottery Ticket Hypothesis at Scale

In this section, I leverage instability analysis and the preceding observations about linear mode connectivity to show how to scale the lottery ticket behavior from Chapter 3 to more challenging settings. In doing so, I reveal further insights into why the sparse subnetworks found by IMP succeed and fail.

4.2.1 Overview

We have long known that it is possible to *prune* neural networks after training, often removing 90% of weights without reducing accuracy after some additional training (e.g., Reed, 1993; Han et al., 2015; Gale et al., 2019). However, sparse networks are more difficult to train from scratch. Beyond *trivial* sparsities where many weights remain and random subnetworks can train to full accuracy, sparse networks trained in isolation are generally less accurate than the corresponding dense networks (Han et al., 2015; Li et al., 2017b; Liu et al., 2019).

However, as shown in the previous chapter, there is a known class of sparse networks that remain accurate at nontrivial sparsities. On small vision tasks, Algorithm 1—*iterative magnitude pruning* (IMP)—retroactively finds sparse subnetworks that were capable of training in isolation from initialization to full accuracy at the sparsities attained by pruning. The existence of such subnetworks raises the prospect of replacing conventional, dense networks with sparse ones, creating new opportunities to reduce the cost of training. However, in more challenging settings, IMP subnetworks perform no better than subnetworks chosen randomly, meaning they only train to full accuracy at trivial sparsities (Chapter 3, Liu et al., 2019; Gale et al., 2019).

In this chapter, I will show that instability analysis offers new insights into the behavior of IMP subnetworks and a potential explanation for their successes and failures. Namely, the sparsest IMP subnetworks only train to full accuracy when they are stable to SGD noise. In other words, when different samples of SGD noise cause an IMP subnetwork to find minima that are not linearly connected, then test accuracy is lower. This insight makes it possible to scale up the lottery ticket observations from

Chapter 3 to more challenging settings by creating subnetworks when they are stable to SGD noise.

4.2.2 Methodology

Iterative magnitude pruning. In this section, I have updated IMP (Algorithm 4) to include the possibility of creating subnetworks at any step k in training rather than just at initialization as in Chapter 3 (Algorithm 1). This change mirrors the structure of instability analysis (Algorithm 3), which considers the instability of a network created at any step k of training.

This updated version of iterative magnitude pruning (IMP) is a procedure to retroactively find a subnetwork of the state of the full network at step k of training. To do so, IMP trains a network to completion, prunes weights with the lowest magnitudes globally, and *rewinds* the remaining weights back to their values at iteration k (Algorithm 4). The result is a subnetwork (W_k, m) where $W_k \in \mathbb{R}^d$ is the state of the full network at step k and $m \in \{0, 1\}^d$ is a mask such that $m \odot W_k$ (where $\odot$ is the element-wise product) is a pruned network. One can run IMP iteratively (pruning 20% of weights, rewinding, and repeating until a target sparsity) or in one shot (pruning to a target sparsity at once). In this chapter, I will one-shot prune ImageNet networks for efficiency and iteratively prune otherwise (Table 4.1).

Chapter 3 focuses on finding sparse subnetworks at initialization; as such, it only uses IMP to "reset" unpruned weights to their values at initialization. One of the main contributions of this chapter is to generalize IMP to *rewind* to any step k. In Chapter 3, I referred to subnetworks that match the accuracy of the full network as *winning tickets* because they have "won the initialization lottery" with weights that make attaining this accuracy possible. When rewinding to iteration $k > 0$, subnetworks are no longer randomly initialized, so the term *winning ticket* is no longer appropriate. However, they are still *matching subnetworks*: they can be trained to completion in isolation and reach the same accuracy as the unpruned network.

Algorithm 4 IMP rewinding to step k and N iterations.

```
1: Create a network with randomly initialization $W_0 \in \mathbb{R}^d$.
2: Initialize pruning mask to $m = 1^d$.
3: Train $W_0$ to $W_k$ with noise $u \sim U$: $W_k = \mathcal{A}^{0 \to k}(W_0, u)$.
4: for $n \in \{1, \ldots, N\}$ do
5:     Train $m \odot W_k$ to $m \odot W_T$ with noise $u' \sim U$:
       $W_T = \mathcal{A}^{k \to T}(m \odot W_k, u')$.
6:     Prune the lowest magnitude entries of $W_T$ that remain.
       Let $m[i] = 0$ if $W_T[i]$ is pruned.
7: Return $W_k, m$
```

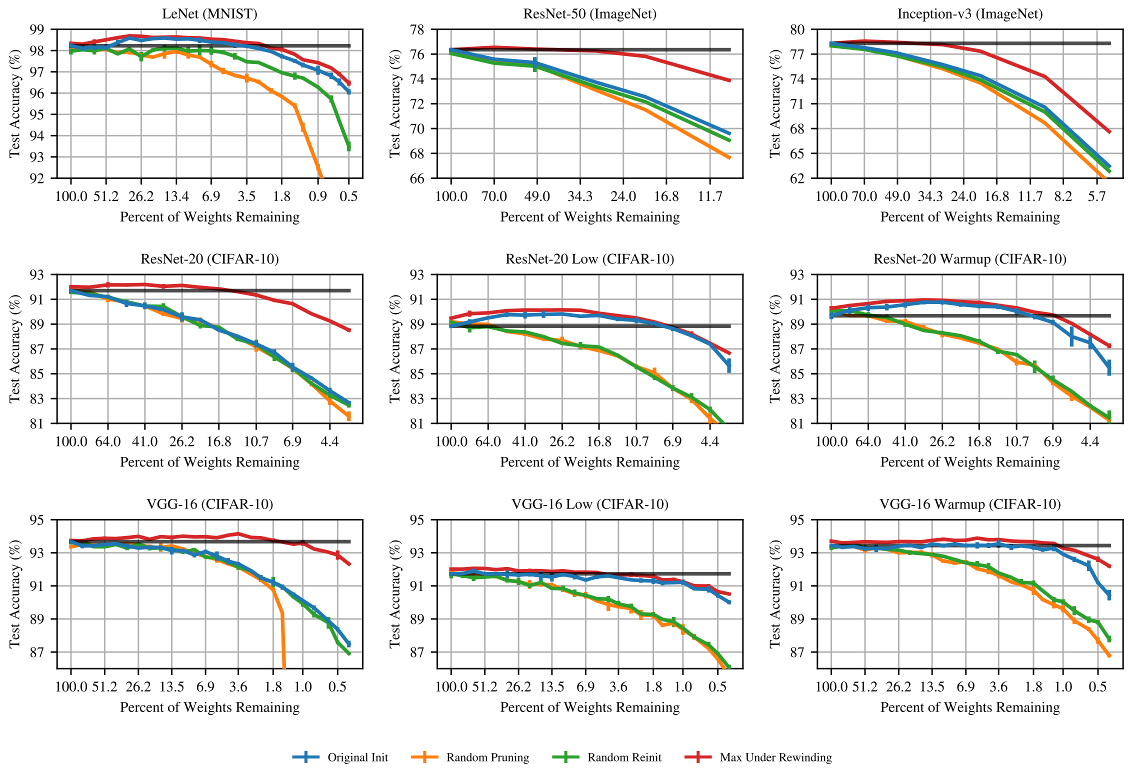


Figure 4-13: An illustration of the methodology by which I select the extreme sparsity levels that we study in Section 4.2. The red line is the maximum accuracy achieved by any IMP subnetwork under any rewinding iteration. The black line is the accuracy of the full network. I use the most extreme sparsity level for which the red and black lines overlap. Each line is the mean and standard deviation across three runs with different initializations.

Sparsity levels. In this section, I will focus on the most extreme sparsity levels for which IMP returns a matching subnetwork at any rewinding step k. These levels are in Table 4.1. These sparsities provide the best contrast between sparse networks that are matching and (1) the full, overparameterized networks and (2) other classes of sparse networks.

I will now describe how I selected the extreme sparsity levels that I examine in Section 4.2.3 (as specified in Table 4.1) for each IMP subnetwork. For each network and hyperparameter configuration, my goal is to study the most extreme sparsity level at which matching subnetworks are known to exist early in training. These are the most challenging circumstances for training sparse subnetworks, so they push the ideas proposed in this section to their limits. To do so, I use IMP to generate subnetworks at many different sparsities for many different rewinding iterations k. I then select the most extreme sparsity level at which any IMP under any rewinding iteration produces a matching subnetwork.

In Figure 4-13, each plot contains the maximum accuracy found by any rewinding iteration in red. The black line is the accuracy of the unpruned network to one standard deviation. For each network, I select the most extreme sparsity for which the red and black lines intersect. As a basis for comparison, these plots also include the result of performing IMP with $k = 0$ according to the procedure in Chapter 3 (blue line), random pruning (orange line), and random reinitialization of the IMP subnetworks with $k = 0$ (green line).

Note that, for computational reasons, ResNet-50 and Inception-v3 are pruned using one-shot pruning, meaning the networks are pruned to the target sparsity all at once. All other networks are pruned using iterative pruning, meaning the networks are pruned by 20% after each iteration of IMP until they reach the target sparsity.

4.2.3 Experiments and Results

Recapping the lottery ticket hypothesis results from Chapter 3. I will begin by revisiting sparse subnetworks rewound to initialization ($k = 0$) as I studied in Chapter 3. I am doing so to provide context for later results using the larger-scale

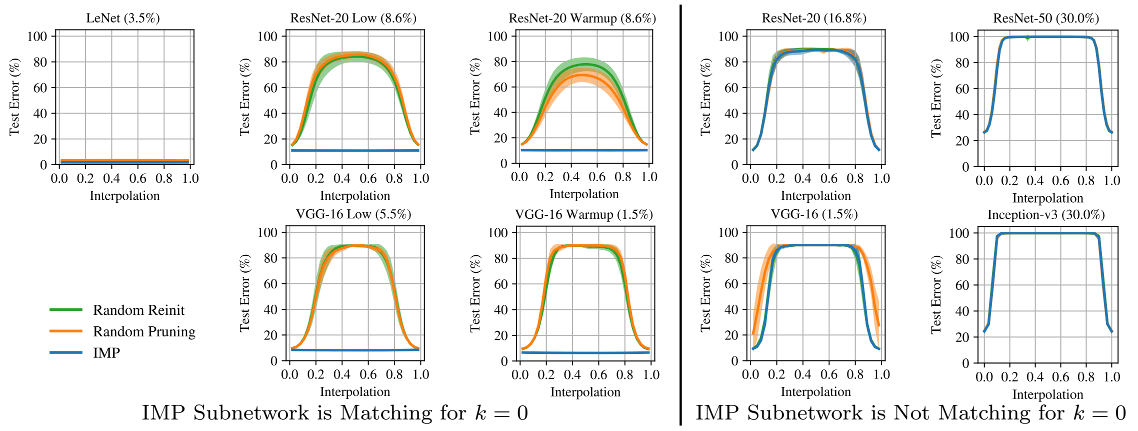


Figure 4-14: Test error when linearly interpolating between subnetworks trained from the same initialization with different SGD noise. Lines are means and standard deviations over three initializations and three data orders (nine in total). Percents are weights remaining.

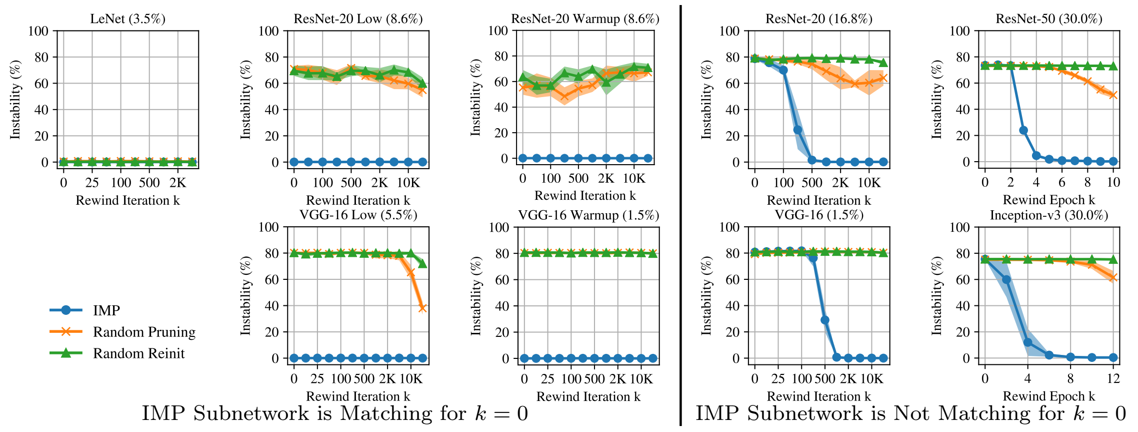


Figure 4-15: Linear interpolation instability of subnetworks created using the state of the full network at step k and applying a pruning mask. Lines are means and standard deviations over three initializations and three data orders (nine in total). Percents are weights remaining.

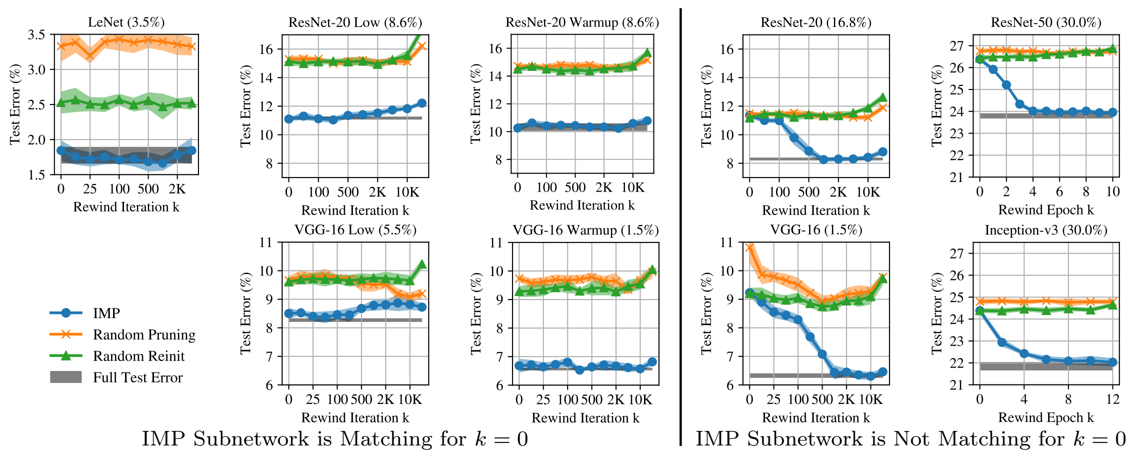


Figure 4-16: Test error of subnetworks created using the state of the full network at step k and applying a pruning mask. Lines are means and standard deviations over three initializations and three data orders (nine in total). Percents are weights remaining.

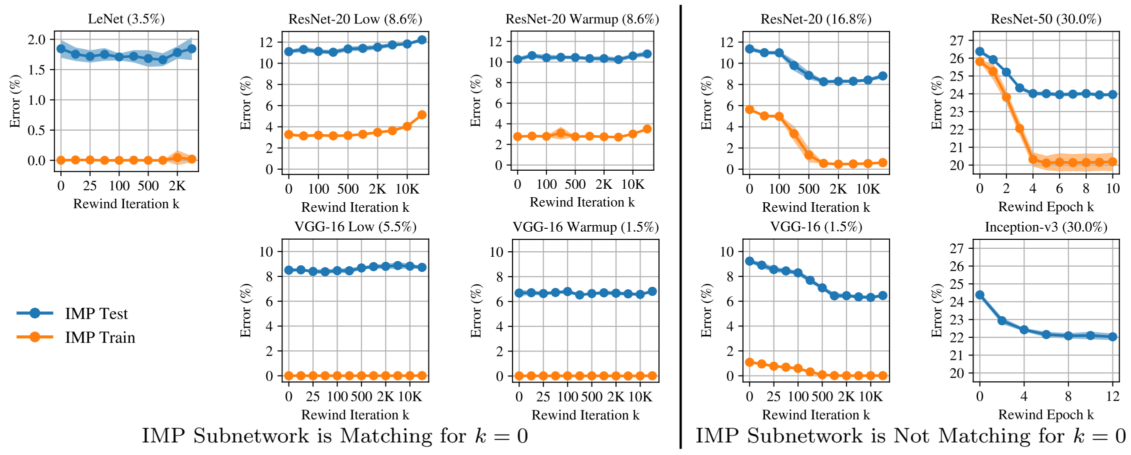


Figure 4-17: Train error of subnetworks created using the state of the full network at step k and apply a pruning mask. Lines are means and standard deviations over three initializations and three data orders (nine in total). Percents are weights remaining. I did not include the train set for Inception-v3 due to computational limitations.

settings in focus in this chapter. As Table 4.2 and Figure 4-13 show, when rewinding to step 0, IMP subnetworks of LeNet are matching, as are variants of ResNet-20 and VGG-16 with lower learning rates or learning rate warmup (changes proposed in Chapter 3 to make it possible for IMP to find matching subnetworks). However, IMP subnetworks of standard ResNet-20, standard VGG-16, ResNet-50, and Inception-v3 are not matching. In fact, they are no more accurate than subnetworks generated by randomly pruning or reinitializing the IMP subnetworks, suggesting that neither the structure nor the initialization uncovered by IMP provides a performance advantage.

Instability analysis of subnetworks at initialization. When performing instability analysis on these subnetworks, I find that they are only matching when they are stable to SGD noise (Figure 4-14). The IMP subnetworks of LeNet, ResNet-20 (low, warmup), and VGG-16 (low, warmup) are stable and matching (Figure 4-14, left). In all other cases, IMP subnetworks are neither stable nor matching (Figure 4-14, left). The low and warmup results are notable because—in Chapter 3—I selected these hyperparameters specifically for IMP to find matching subnetworks; that this change also makes the subnetworks stable adds further evidence of a connection between instability and accuracy in IMP subnetworks.

No randomly pruned or reinitialized subnetworks are stable or matching at these sparsities except those of LeNet: LeNet subnetworks are not matching but error only rises slightly when interpolating. For all other networks, error approaches that of random guessing when interpolating.

Instability analysis of subnetworks during training. I just showed that IMP subnetworks are matching from initialization only when they are stable. In Section 4.1.2, I found that unpruned networks become stable only after a certain amount of training. Here, I combine these observations: I study whether IMP subnetworks become stable later in training and, if so, whether improved accuracy follows.

Concretely, I perform IMP where I rewind to iteration $k > 0$ after pruning. Doing so produces a subnetwork (W_k, m) of the state of the full network at iteration k. I

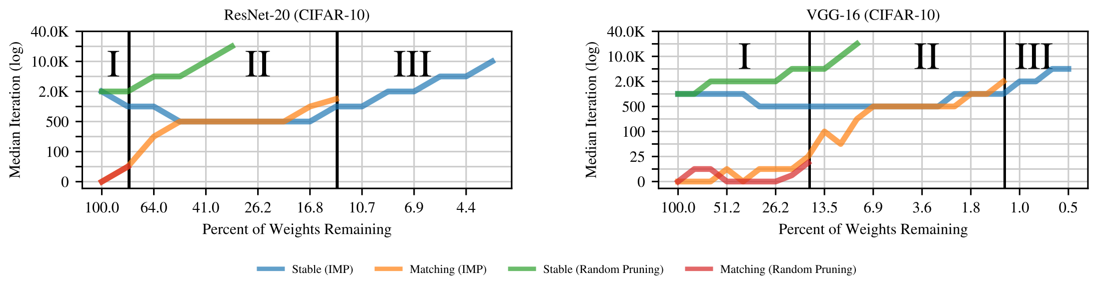


Figure 4-18: The median rewinding iteration at which IMP subnetworks and randomly pruned subnetworks of ResNet-20 and VGG-16 become stable and matching. A subnetwork is stable if instability $< 2\%$. A subnetwork is matching if the accuracy drop $< 0.2\%$; we only include points where a majority of subnetworks are matching at any rewinding iteration. Each line is the median across three initializations and three data orders (nine samples in total).

then run instability analysis using this subnetwork. Another way of looking at this experiment is that it simulates training the full network to iteration k, generating a pruning mask, and evaluating the instability of the resulting subnetwork; the underlying mask-generation procedure involves consulting an oracle that trains the network many times in the course of performing IMP.

The blue dots in Figure 4-15 show the instability of the IMP subnetworks at many rewinding iterations. Networks whose IMP subnetworks were stable when rewinding to iteration 0 remain stable at all other rewinding points (Figure 4-15, left). Notably, networks whose IMP subnetworks were *unstable* when rewinding to iteration 0 become stable when rewinding later. IMP subnetworks of ResNet-20 and VGG-16 become stable at iterations 500 (0.8% into training) and 1000 (1.6%). Likewise, IMP subnetworks of ResNet-50 and Inception-v3 become stable at epochs 5 (5.5% into training) and 6 (3.5%). In all cases, the IMP subnetworks become stable sooner than the unpruned networks, substantially so for ResNet-50 (epoch 5 vs. 18) and Inception-v3 (epoch 6 vs. 28).

The test error of the IMP subnetworks behaves similarly. The blue line in Figure 4-16 plots the error of the IMP subnetworks and the gray line plots the error of the full networks to one standard deviation; subnetworks are matching when the lines cross. Networks whose IMP subnetworks were matching when rewinding to step 0 (Figure 4-16, left) generally remain matching at later iterations (except for ResNet-20 low and VGG-16 low at the latest rewinding points). Notably, networks whose IMP subnetworks were *not* matching when rewinding to iteration 0 (Figure 4-16, right) become matching when rewinding later. Moreover, these rewinding points closely coincide with those where the subnetworks become stable. In summary, at these extreme sparsities, IMP subnetworks are matching when they are stable.

Other observations. Interestingly, the same pattern holds for the train error: for those networks whose IMP subnetworks were not matching at step 0, train error decreases when rewinding later. For ResNet-20 and VGG-16, rewinding makes it possible for the IMP subnetworks to converge to 0% train error. These results suggest stable IMP subnetworks also optimize better.

Randomly pruned and reinitialized subnetworks are unstable and non-matching at all rewinding points (with LeNet again an exception). Although it is beyond the scope of this chapter, this behavior suggests a potential broader link between subnetwork stability and accuracy: IMP subnetworks are matching and become stable at least as early as the full networks, while other subnetworks are less accurate and unstable for the sparsities and rewinding points we consider.

Train set data. In the preceding experiments, I only measured instability and error on the test set. Here, I present the corresponding data on the training set. Figures 4-19 and 4-20 examine the instability and error of the same IMP subnetworks as Figure 4-15, but it shows both the train and test sets. I did not compute the train set quantities for Inception-v3 due to computational limitations.

Train set and test set instability are nearly identical, just as found in Section 4.1.2. Interestingly, the two coincide more closely for IMP subnetworks of ResNet-50 than they do for the unpruned networks in Section 4.1.2.

For networks that are unstable at rewinding iteration 0, train error and test error follow similar trends, starting higher when the subnetworks are unstable and dropping when the subnetworks become stable. That is, the unstable IMP subnetworks are not able to fully optimize to 0% train error, while the stable IMP subnetworks are.

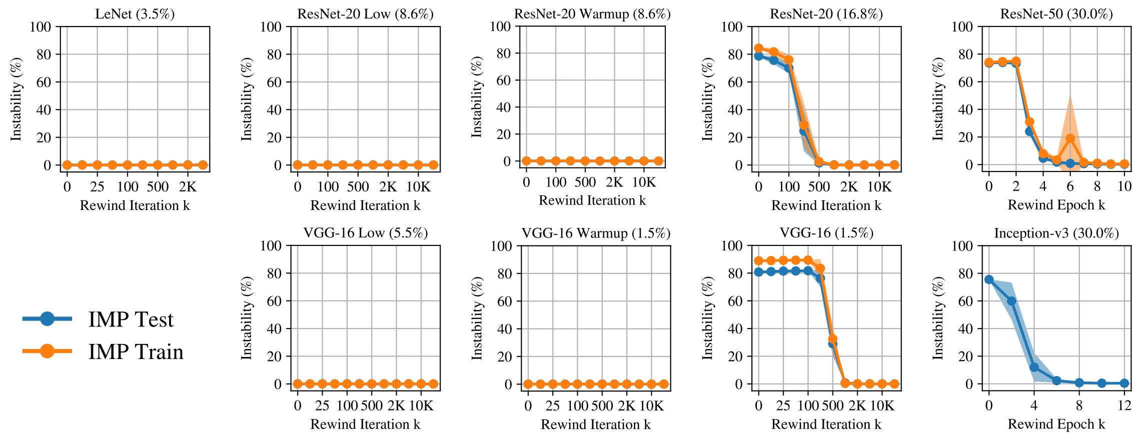


Figure 4-19: The train and test set instability of subnetworks that are created by using the state of the full network at iteration k, applying the pruning mask found by performing IMP with rewinding to iteration k, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining. I did not include the train set for Inception due to computational limitations.

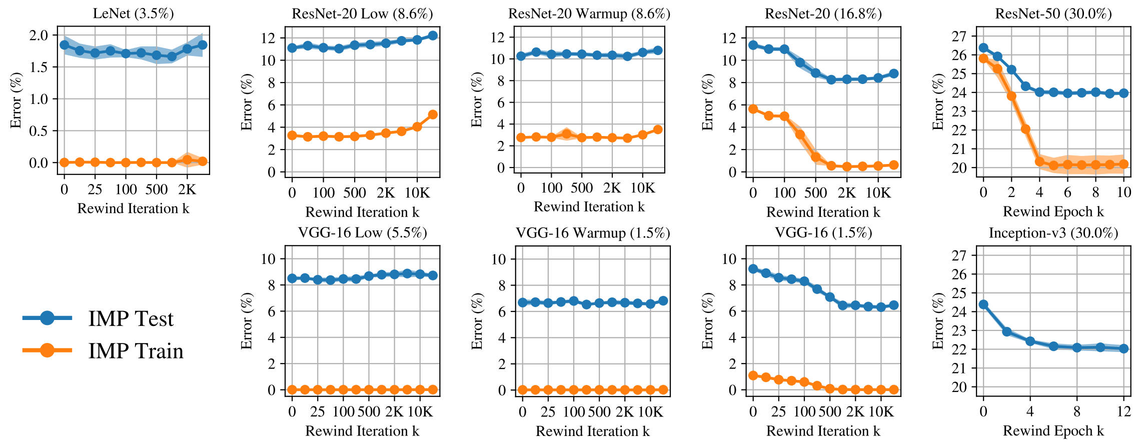


Figure 4-20: The train and test set error of subnetworks that are created by using the state of the full network at iteration k, applying the pruning mask found by performing IMP with rewinding to iteration k, and training on different data orders from there. Each line is the mean and standard deviation across three initializations. Percents are percents of weights remaining. I did not include the train set for Inception due to computational limitations.

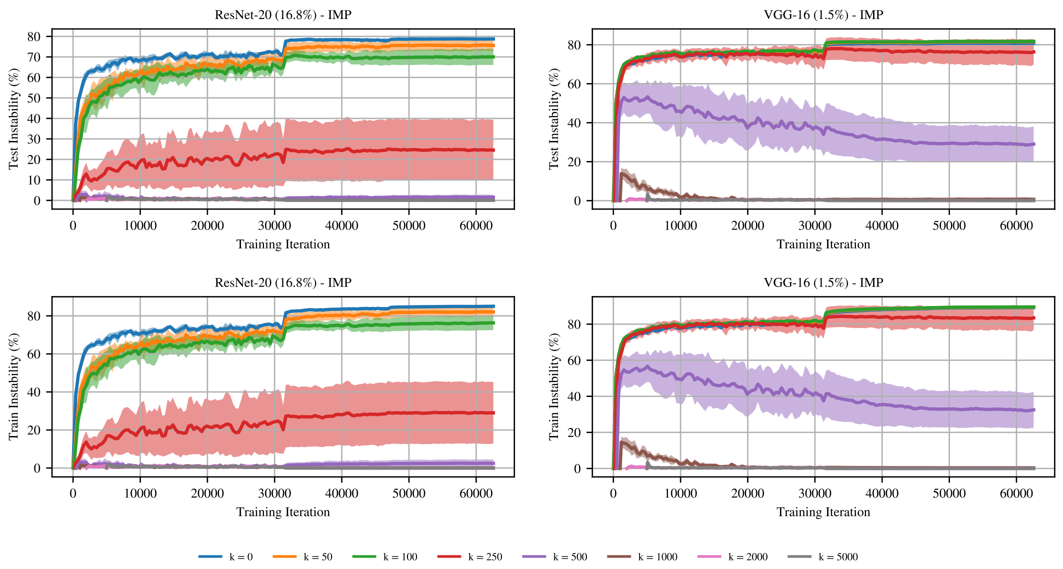


Figure 4-21: Instability throughout training for ResNet-20 and VGG-16 using the IMP-pruned networks as computed on both the test set and train set. Each line involves training to iteration k and then training two copies on different data orders after. Each point is the instability when interpolating between the states of the networks at the training iteration on the x-axis.

Instability throughout training. As in Section 4.1 Figure 4-5, I include the plots of instability over time for the sparse networks in Figure 4-21. The IMP subnetworks of ResNet-20 exhibit the same behavior as the unpruned network: when the network is stable at the end of training, it is stable throughout training, meaning two copies of the same network follow the same optimization trajectory up to linear interpolation. The IMP subnetworks of VGG-16 exhibit sightly different behavior at rewinding iterations 500 and 1000: instability initially spikes (meaning the networks rapidly become separated by a loss barrier) but decreases gradually thereafter. For rewinding iteration 1000, it decreases to 0, meaning the networks are stable by the end of training. For all other rewinding iterations, being stable at the end of training corresponds to being stable throughout training, so it is possible that rewinding iteration 1000 represents a transition point between the unstable rewinding iterations earlier and the stable rewinding iterations later.

Full linear interpolation data. In Figure 4-15, I plot the instability value derived from linearly interpolating between copies of the same network or subnetwork on

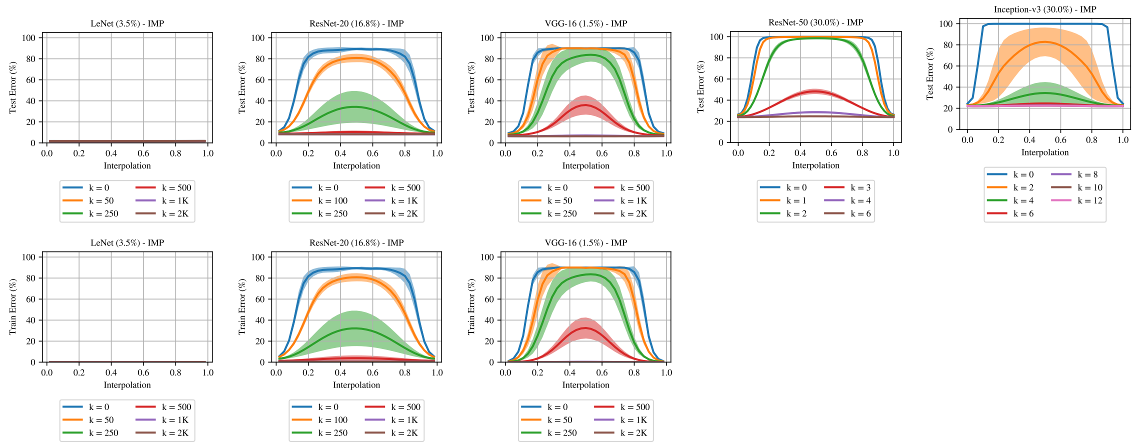


Figure 4-22: The error when linearly interpolating between the minima found by randomly initializing a network, training to iteration k, pruning according to IMP, and training two copies from there to completion using different data orders. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). The errors of the trained networks are at interpolation = 0.0 and 1.0. I did not interpolate using the training set for the ImageNet networks due to computational limitations.

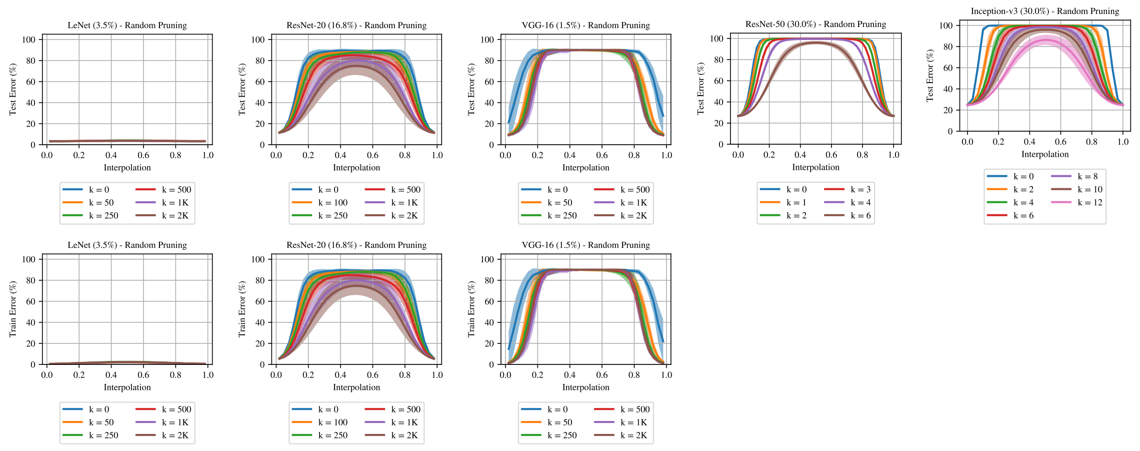


Figure 4-23: The error when linearly interpolating between the minima found by randomly initializing a network, training to iteration k, pruning randomly in the same layerwise proportions as IMP, and training two copies from there to completion using different data orders. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). The errors of the trained networks are at interpolation = 0.0 and 1.0. I did not interpolate using the training set for the ImageNet networks due to computational limitations.

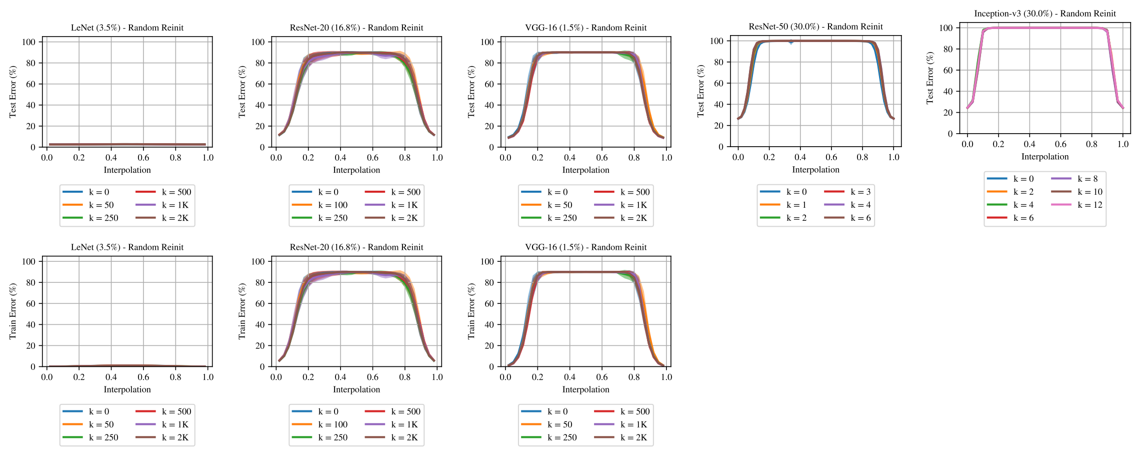


Figure 4-24: The error when linearly interpolating between the minima found by randomly initializing a network, training to iteration k, pruning according to IMP, randomly reinitializing, and training two copies from there to completion using different data orders. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). The errors of the trained networks are at interpolation $=$ 0.0 and 1.0. I did not interpolate using the training set for the ImageNet networks due to computational limitations.

different data orders. In Figures 4-22, 4-23, and 4-24, I plot the linear interpolation data from which I derived the instabilities in Figure 4-15. This is the raw data that was used to create the instability plots in Figure 4-15.

4.2.4 Alternate Distance Functions

As described in Section 4.1.5, one could parameterize instability analysis with a wide range of other functions for comparing pairs of neural networks other than linear interpolation. Here, I consider the same four functions as in Section 4.1.5.

L_2 **Distance.** In Figure 4-25, I plot the L_2 distance function at all rewinding points for all three classes of sparse networks. L_2 distance mirrors the behavior of instability. In cases where the IMP subnetworks are stable at all rewinding points (ResNet-20 low/warmup, VGG-16 low/warmup, and LeNet), the L_2 distance is at a lower level than the L_2 distance between the other baselines (random pruning and random reinitialization) and is consistent across rewinding points. In cases where the IMP subnetworks are unstable at initialization but become stable later (ResNet-20 and VGG-16),

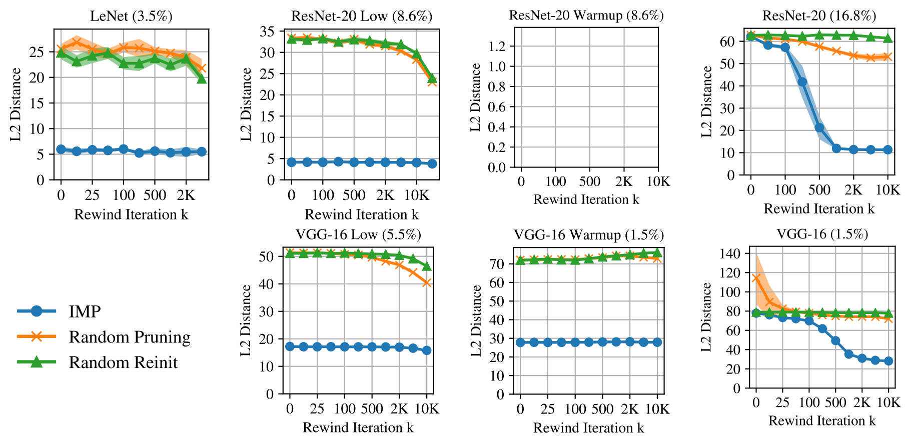


Figure 4-25: The L_2 distance between subnetworks that are created by using the state of the full network at iteration k, applying a pruning mask, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining. I did not compute the train set quantities for the ImageNet networks due to computational limitations.

the L_2 distance begins high (at the same level as the L_2 distance for the randomly pruned and randomly reinitialized baselines) and drops when the subnetworks become stable, settling at a lower level. Although stable IMP subnetworks are closer in L_2 distance than unstable IMP subnetworks and the baselines, the L_2 distance remains far from zero. In general, it is difficult to translate the results of this function into higher-level statements about the relationships between the networks.

Cosine distance. In Figure 4-26, I plot the cosine distance in a manner similar to L_2 distance. The results are similar, and the same interpretation applies.

Classification differences. Figures 4-9 (test set) and 4-10 (train set), include the number of examples classified differently by the networks for all three varieties of sparse networks across rewinding iterations (in addition to the unpruned networks from Section 4.1).

The behavior of the IMP sparse networks better matches instability as compared to the unpruned networks. IMP subnetworks that are stable from initialization (ResNet-20 low and warmup, VGG-16 low and warmup, LeNet) consistently have the same

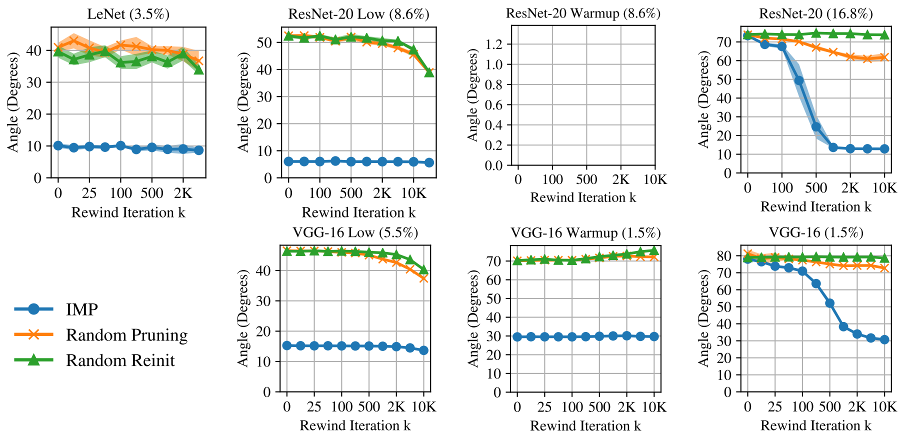


Figure 4-26: The cosine distance between subnetworks that are created by using the state of the full network at iteration k, applying a pruning mask, and training on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples in total). Percents are percents of weights remaining. I did not compute the train set quantities for the ImageNet networks due to computational limitations.

distance no matter the rewinding iteration. This distance is lower than that for the randomly pruned and randomly reinitialized baselines.

IMP subnetworks that are unstable at iteration 0 (ResNet-20 and VGG-16) have the same number of different classifications as the baselines when rewinding to iteration 0. When the networks become stable, the number of different classifications drops substantially to a lower level.

One challenge with using this distance function is that it is inherently entangled with accuracy. As the accuracy of the networks improves, the number of different classifications might decrease simply because the networks will classify more examples correctly (and thereby, the same way). Consider the IMP subnetworks of ResNet-20 on the CIFAR-10 test set (the graph in the upper right of Figure 4-9, blue line). At rewinding iteration 0, the networks have about 11% error on the test set, meaning there are at most 2200 examples they could classify differently.[2] In Figure 4-9, the networks are classifying about 1100 examples differently.

When ResNet-20 IMP subnetworks are stable, error decreases to 8.5%, meaning

[2]In the worst case, all examples that one network misclassifies will be classified correctly by the other. Since each network misclassifies 1100 examples, 2200 examples will be classified differently.

at most 1700 examples can be classified differently. However, in Figure 4-8, we see that only about 350 examples are being classified differently. Although this number is lower than the 1100 differences at rewinding iteration 0 in absolute terms, accuracy has improved as well, so one must consider these differences in context. At rewinding iteration 0, classification differences are 50% of their maximum possible value, while at rewinding iteration 1000, classification differences are at 21% of their maximum possible value. In summary, as the IMP subnetworks become stable, they behave in a more functionally similar fashion, even considering accuracy improvements.

Loss L_2 distance. I plot this data in Figures 4-11 (test set) and 4-12 (train set). Just as with the unpruned networks, it largely mirrors the behavior from the classification difference function, and the same interpretations apply.

4.2.5 Results at Other Sparsity Levels

Thus far, I have performed instability analysis at only two sparsities: unpruned networks (Section 4.1.2) and an extreme sparsity (Section 4.2.3). Here, I examine sparsities between these levels and beyond for ResNet-20 and VGG-16. Figure 4-18 presents the median iteration at which IMP and randomly pruned subnetworks become stable (instability $< 2\%$) and matching (accuracy drop $< 0.2\%$, allowing a small margin for noise) across sparsity levels.

Stability behavior. As sparsity increases, the iteration at which the IMP subnetworks become stable decreases, plateaus, and eventually increases. In contrast, the iteration at which randomly pruned subnetworks become stable only increases until the subnetworks are no longer stable at any rewinding iteration.

Matching behavior. I separate the sparsities into three *ranges* where different sparse networks are matching.

In sparsity range I, the networks are so overparameterized that even randomly pruned subnetworks are matching (red). These are sparsities we refer to as *trivial.*

This range occurs when more than 80.0% and 16.8% of weights remain for ResNet-20 and VGG-16.

In sparsity range II, the networks are sufficiently sparse that only IMP subnetworks are matching (orange). This range occurs when 80.0%-13.4% and 16.8%-1.2% of weights remain in ResNet-20 and VGG-16. For part of this range, IMP subnetworks become matching and stable at approximately the same rewinding iteration; namely, when 51.2%-13.4% and 6.9%-1.5% of weights remain for ResNet-20 and VGG-16. In Section 4.2.3, I observed this behavior for a single, extreme sparsity level for each network. Based on Figure 4-18, I conclude that there are many sparsities where these rewinding iterations coincide for ResNet-20 and VGG-16.

In sparsity range III, the networks are so sparse that even IMP subnetworks are not matching at any rewinding iteration we consider. This range occurs when fewer than 13.4% and 1.2% of weights remain for ResNet-20 and VGG-16. According to the analysis to follow, the error of IMP subnetworks still decreases when they become stable (although not to the point that they are matching), potentially suggesting a broader relationship between instability and accuracy.

Results across all sparsities. In Figure 4-15 and the preceding analysis, I showed the relationship between sparsity level and rewinding iteration. Here, I present the raw data for that analysis: the full data on the relationship between rewinding iteration and instability/test error for all levels of sparsity for standard ResNet-20 (Figures 4-27 and 4-28) and VGG-16 (Figures 4-29 and 4-30) on CIFAR-10.

This data begins with 80% of weights remaining and includes sparsities attained by repeatedly pruning 20% of weights (e.g., 64% of weights remaining, 51% of weights remaining, etc.). I include these levels in particular because we use IMP to prune 20% of weights per iteration, meaning we have sparse IMP subnetworks for each of these levels. I include data for every sparsity level displayed in Figure 4-13, including those beyond the extreme sparsities studied in Section 4.2.3.

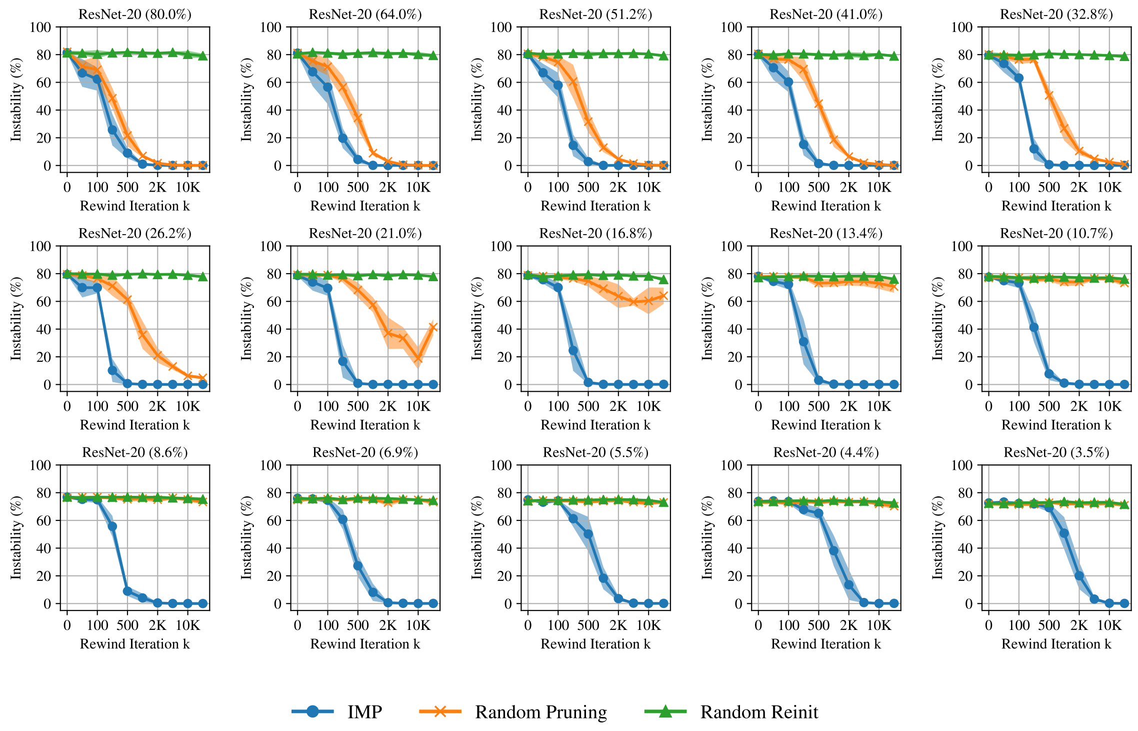


Figure 4-27: The instability of subnetworks of ResNet-20 created using the state of the full network at iteration k and trained on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples total). Percents are percents of weights remaining.

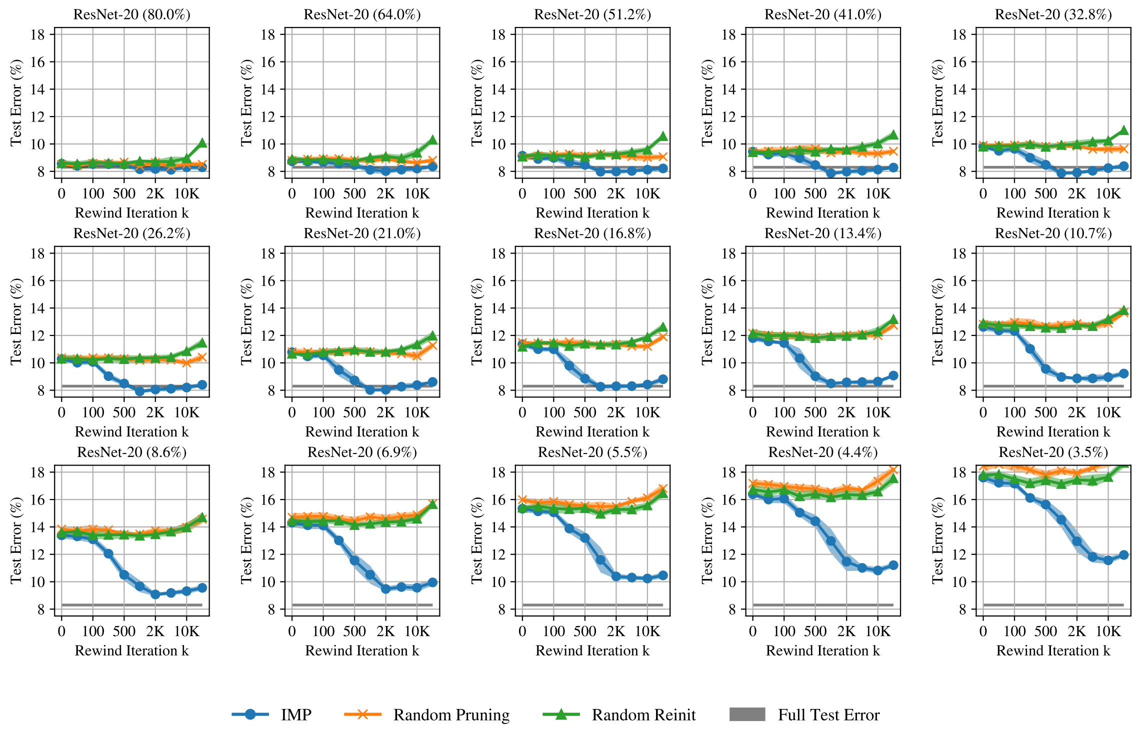


Figure 4-28: The test error of subnetworks of ResNet-20 created using the state of the full network at iteration k and trained on different data orders from there. Each line is the mean and standard deviation across three initializations. Gray lines are the accuracies of the full networks to one standard deviation. Percents are percents of weights remaining.

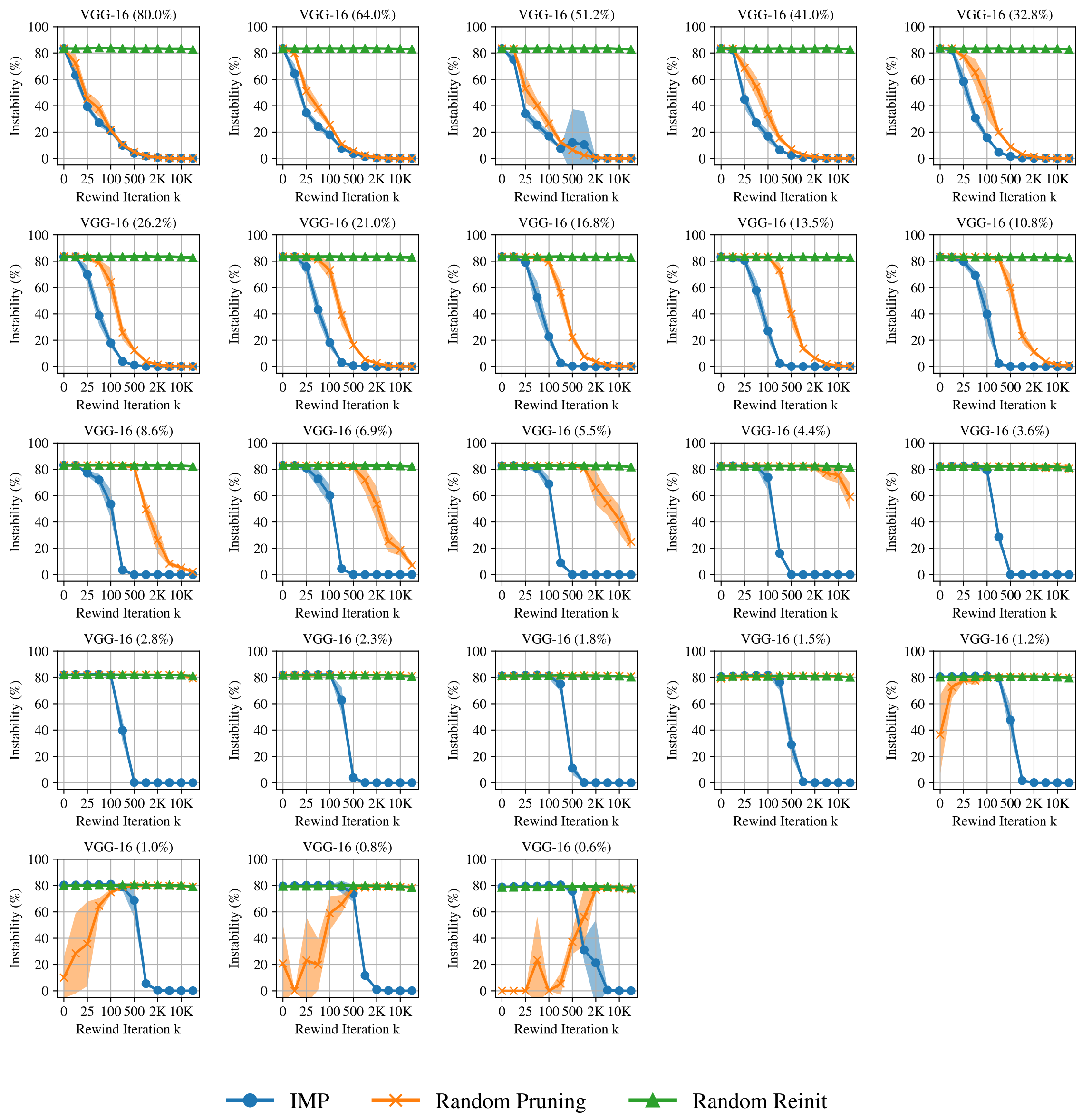


Figure 4-29: The instability of subnetworks of VGG-16 created using the state of the full network at iteration k and trained on different data orders from there. Each line is the mean and standard deviation across three initializations and three data orders (nine samples total) Percents are percents of weights remaining.

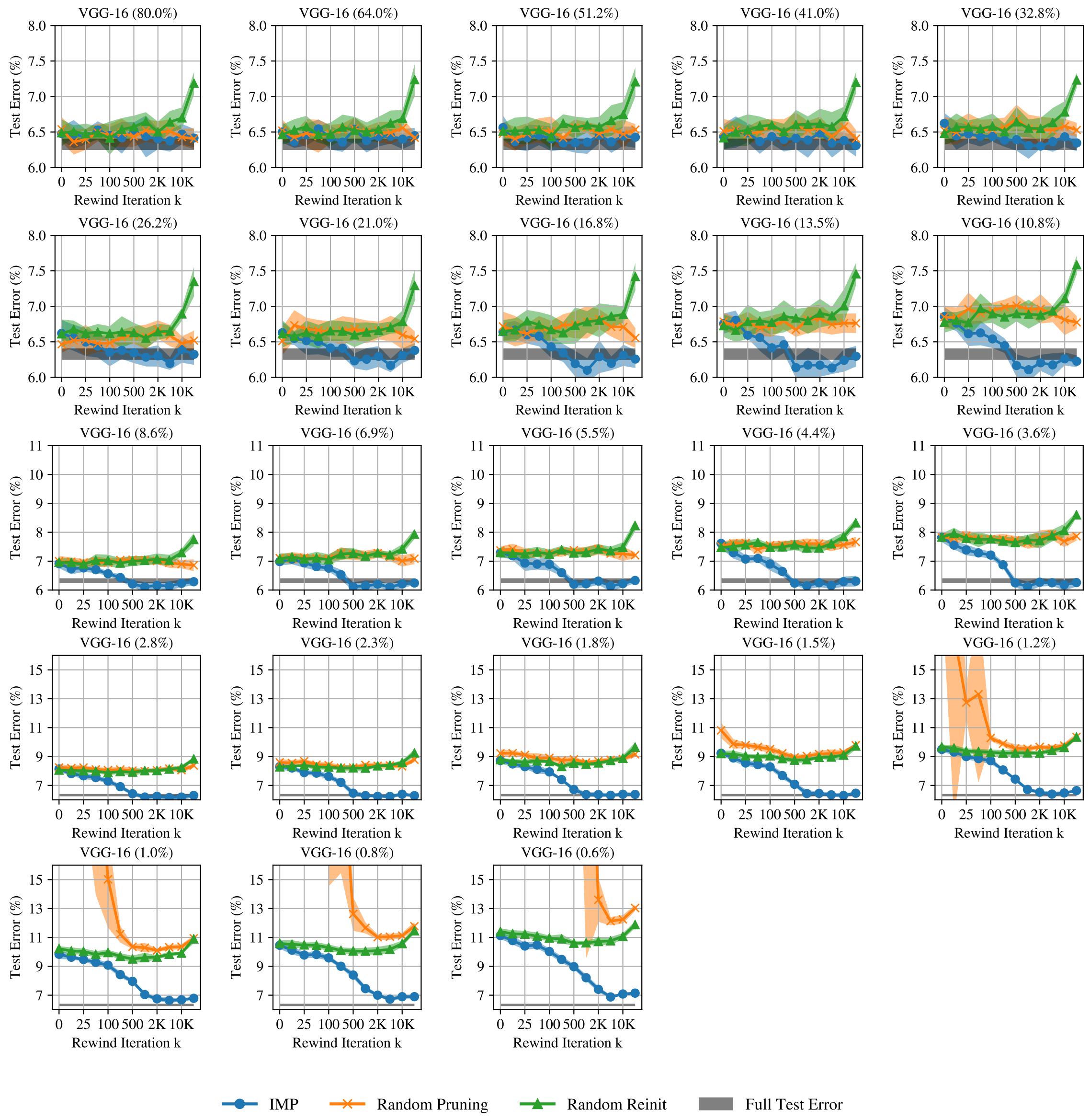


Figure 4-30: The test error of subnetworks of VGG-16 created using the state of the full network at iteration k and trained on different data orders from there. Each line is the mean and standard deviation across three initializations. Gray lines are the accuracies of the full networks to one standard deviation. Percents are percents of weights remaining.

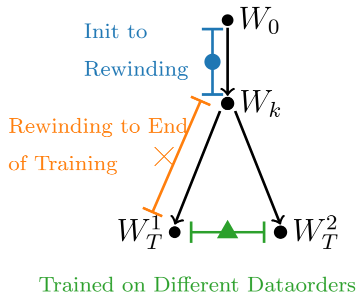
4.3 The State of the Network at Rewinding

4.3.1 Methodology

In the preceding sections of this Chapter, I performed instability analysis by training to step k, making two copies of the network, optionally apply a pruning mask, and training these two copies to completion under different samples of SGD noise. I found that, for a sufficiently large value of k, the trained networks will find the same, linearly connected minimum. In this section, I address the following question: what is the state of the network at the step k from which this linear connectivity results? Are the networks so far along in training that they are virtually fully optimized? Have they traveled the vast majority of the distance from initialization to the eventual minimum? In this sense, is the iteration at which the network becomes stable "trivial?" I address these questions in two ways.

Error at rewinding. In Figure 4-31, I present the error of the unpruned network at each rewinding iteration from the preceding sections. With this data, I investigate how close the network has come to its full accuracy when it becomes stable.

L_2 **distances.** In Figures 4-32 and 4-33, I measure various L_2 distances that capture how close the network is to initialization and to the end of training. In particular, I measure three distances as shown in the diagram below (which is an annotated version of Figure 4-1).



First, I measure the L_2 distance in parameter space from initialization to step k (blue circle in the diagram above and in Figures 4-32 and 4-33); for the sparse

IMP subnetworks, I measure the L_2 distance after applying the pruning mask to both initialization and the state of the network at iteration k. This quantity captures the distance that the network has traversed from initialization by step k.

Second, I measure the distance from the state of the network at step k to its state at the end of training under one data order (orange x in the diagram above and in Figures 4-32 and 4-33). This quantity captures the distance that the network traverses after step k. If the network is very close to the minimum by the time it becomes stable, then one would expect this quantity to be small compared to the L_2 distance between initialization and iteration k; that would indicate that the network has already traversed a large distance and has a relatively smaller distance to go.

Finally, I measure the distance between the final states of networks trained from step k under different data orders (green triangle in the diagram above and in Figures 4-32 and 4-33). This quantity captures the size of the linearly connected minimum found by the networks. I am most interested in how this distance compares to the distance traveled by the networks and how this quantity changes as the rewinding iteration varies.

4.3.2 Results

Error at rewinding. These results appear in Figure 4-31. Recall that the unpruned networks become stable at a different (typically later) iteration than the IMP subnetworks, so we consider two rewinding points for each network.

Unpruned networks. ResNet-20 and VGG-16 become stable to SGD noise at iterations 2000 and 1000, at which point test error is about 25% (compared to final error 8.3%) for ResNet-20 and 20% (compared to final error 6.3%) for VGG-16. Train error is at a similar value to test error at these points; in both cases, train error eventually converges to 0%. I conclude that, at the iteration at which they become stable to SGD noise, these networks have not fully converged but are much closer to their final errors than to random guessing.

There is similar behavior for the unpruned ResNet-50 and Inception networks, which become stable to SGD noise at epochs 18 and 28. Test error is 55% (compared to final error 24%) for ResNet-50 and 33% (compared to final error 22%) for Inception-v3. Both networks are most of the way to their final performance.

IMP pruned subnetworks. The IMP pruned subnetworks become stable to SGD noise earlier than the unpruned networks. ResNet-20 and VGG-16 become stable to SGD noise at iterations 500 and 1000, at which point error is 30% (compared to final error 8.3%) for ResNet-20 and 35% (compared to final error 6.3%) for VGG-16. These networks have not fully converged but are closer to their final errors than to random guessing. IMP subnetworks of ResNet-50 and Inception-v3 become stable to SGD noise much earlier than the unpruned networks—at epoch 5 and epoch 6, respectively. At these points, error is much higher—55% for ResNet-50 and 40% for Inception-v3—leaving these networks substantial room to further train. I did not evaluate the train accuracy at these checkpoints for the ImageNet networks due to storage and computational limitations.

L_2 **distances.** These results appear in Figures 4-32 and 4-33.

Unpruned networks. ResNet-20 and VGG-16 become stable to SGD noise at iterations 2000 and 1000, at which point they are closer to their initial weights than to their final weights. This indicates that they still have a substantial distance to travel on the optimization landscape and are still far from their final weights. This result is particularly remarkable considering the previous observation that stable networks follow the same, linearly connected trajectory throughout training (according to test error); the L_2 distance data suggests that they do so for a substantial distance.

The unpruned ResNet-50 and Inception-v3 networks are closer to their final weights than their initial weights when they become stable to SGD noise. In fact, it appears that distance from initialization begins to plateau and distance to the final weights only decreases slowly. This may indicate that the networks will make much slower progress for the remaining 80% of training iterations.

The green triangles in these plots show the distance between the weights of copies of the network trained from a rewinding iteration to completion on different data orders. In all cases, the distance between these copies is substantial, even after the networks become stable. As a point of comparison, we use the distance that the networks travel between initialization and the final weights, which is captured by the orange x for rewinding iteration 0. For ResNet-20, the distance between copies trained on different data orders from iteration 2000 (when it becomes stable) is more than half the distance that the network travels during the entirety of training. The same is true for VGG-16 from iteration 1000 (when it becomes stable). For ResNet-50 and Inception-v3, this distance is about a quarter and half (respectively) of the distance the networks travel over the course of training. These are remarkably large distances considering that any network on this line segment reaches full test accuracy.

IMP pruned subnetworks. I show the same data for the IMP subnetworks in Figure 4-33. Each L_2 distance in this figure is measured after applying the pruning mask to all weights. When ResNet-20 and VGG-16 become stable to SGD noise (iterations 2000 and 1000, respectively), they are about 2x (ResNet-20) and 3x (VGG-16) closer to their initial weights than their final weights. ResNet-50 and Inception-v3 are about equal distances from both points for the epochs at which they become stable.

Unique to the IMP subnetworks, I observe here and in the sections on alternate distance metrics that the L_2 distance between copies trained on different data orders drops alongside instability, plateauing at a lower value when training from the rewinding iteration at which the subnetworks becomes stable. Even this lower distance is still a substantial fraction of the overall distance the network travels: 25%, 45%, 27%, and 28% for ResNet-20, VGG-16, ResNet-50, and Inception-v3.

4.4 Discussion

Instability analysis. I introduce instability analysis as a novel way to study the sensitivity of a neural network's optimization trajectory to SGD noise. In doing so, I uncover a class of situations in which *linear* mode connectivity emerges, whereas previous examples of mode connectivity (e.g., between networks trained from different initializations) at similar scales required nonlinear paths (Draxler et al., 2018; Garipov et al., 2018).

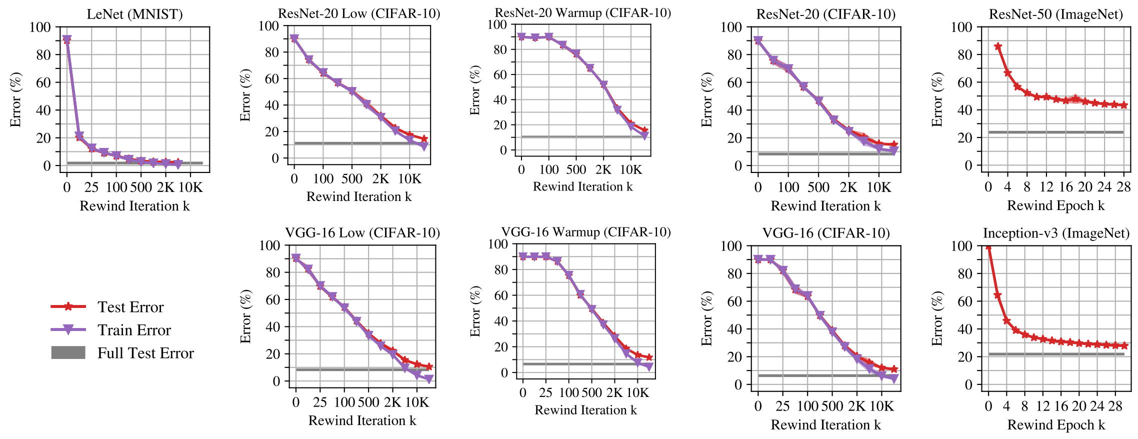


Figure 4-31: The error of the full networks at the rewinding iteration specified on the x-axis. For clarity, this is the error of the network at that specific iteration of training, before any copies are made or further training occurs. Each line is the mean and standard deviation across three initializations.

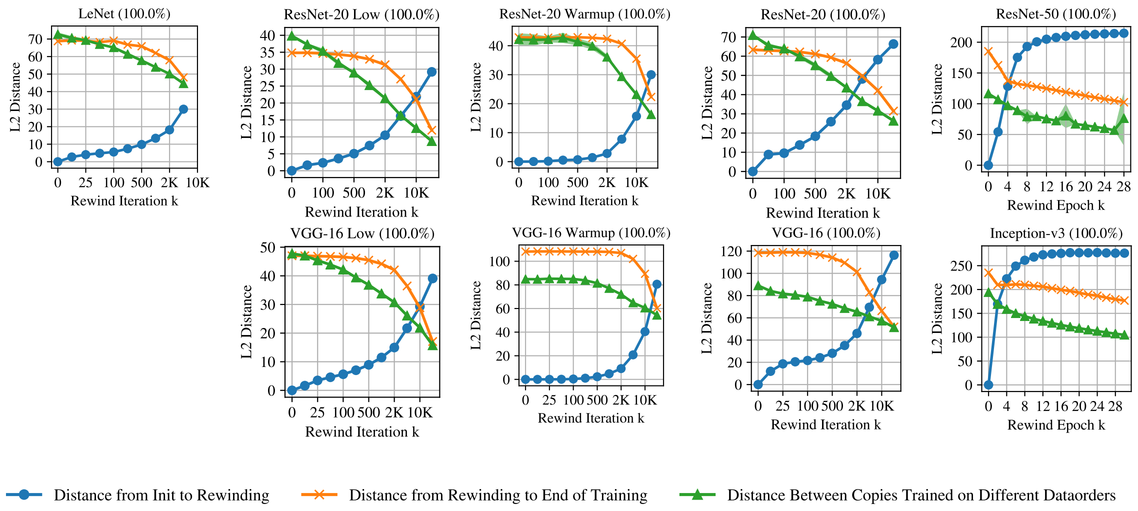


Figure 4-32: L_2 distances for the full networks at the rewinding iteration on the x-axis. Each line is the mean and standard deviation across three initializations.

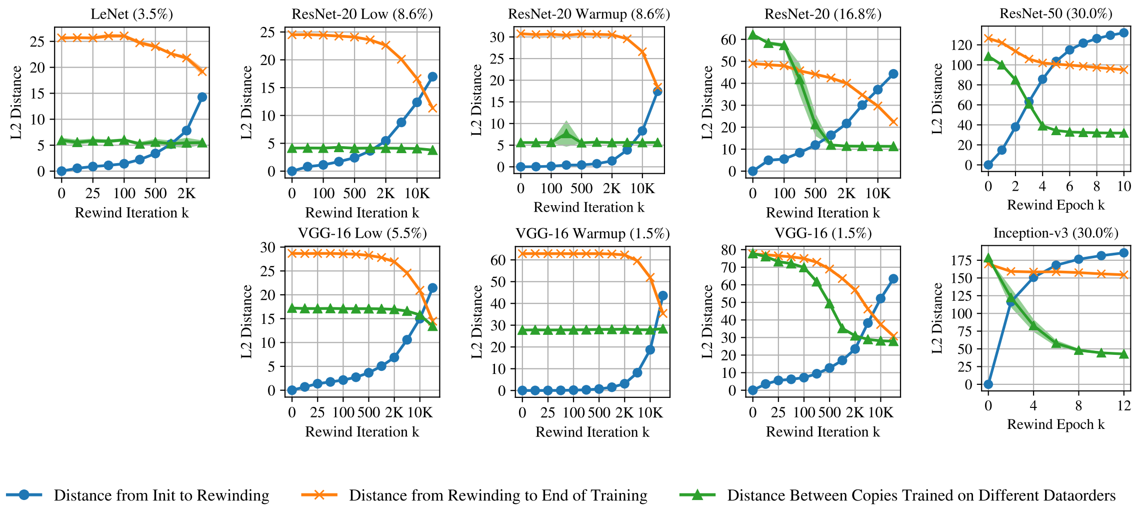


Figure 4-33: Various L_2 distances for the IMP subnetworks at the rewinding iteration specified on the x-axis. Each line is the mean and standard deviation across three initializations. Each L_2 distance is computed after applying the pruning mask to the states of the networks in question.

The unpruned network results in Section 4.1.2 divide training into two phases: an unstable phase where the network finds linearly unconnected minima due to SGD noise and a stable phase where the linearly connected minimum is determined. The finding that stability emerges early in training adds to work suggesting that training comprises a noisy first phase and a less stochastic second phase. For example, the Hessian eigenspectrum settles into a few large values and a bulk (Gur-Ari et al., 2018), and large-batch training at high learning rates benefits from learning rate warmup (Goyal et al., 2017).

One way to exploit these findings is to explore changing aspects of optimization (e.g., learning rate schedule or optimizer) similar to Goyal et al. (2017) once the network becomes stable; instability analysis can evaluate the consequences of doing so. I also believe instability analysis provides a scientific tool for topics related to the scale and distribution of SGD noise, e.g., the relationship between batch size, learning rate, and generalization (LeCun et al., 2012; Keskar et al., 2017; Goyal et al., 2017; Smith & Le, 2018; Smith et al., 2018) and the efficacy of alternative learning rate schedules (Smith, 2017; Smith & Topin, 2018; Li & Arora, 2020).

The lottery ticket hypothesis. As framed in Chapter 3, my lottery ticket hypothesis entails the conjecture that any "randomly initialized, dense neural network contains a subnetwork that—when trained in isolation—matches the accuracy of the original network." This work is among several recent papers to propose that merely sparsifying at initialization can produce high performance neural networks (Mallya et al., 2018; Zhou et al., 2019; Ramanujan et al., 2020; Evci et al., 2020). In Chapter 3, I supported the lottery ticket hypothesis by using IMP to find matching subnetworks at initialization in small vision networks. However, follow-up studies show (Liu et al., 2019; Gale et al., 2019) and the current chapter confirm that IMP does not find matching subnetworks at nontrivial sparsities in more challenging settings. I use instability analysis to distinguish the successes and failures of IMP as identified in previous work. In doing so, I make a new connection between the lottery ticket hypothesis and the optimization dynamics of neural networks.

Practical impact of rewinding. By extending IMP with rewinding, I show how to find matching subnetworks in much larger settings than in previous work, albeit from *early* in training rather than initialization. This technique has already been adopted for practical purposes. Morcos et al. (2019) show that subnetworks found by IMP with rewinding transfer between vision tasks, meaning the effort of finding a subnetworks can be amortized by reusing it many times. Renda et al. (2020) show that IMP with rewinding prunes to state-of-the-art sparsities, matching or exceeding the performance of standard techniques that fine-tune at a low learning rate after pruning (e.g., Han et al., 2015; He et al., 2018b). Other efforts use rewinding to further study lottery tickets (Yu et al., 2020; Frankle et al., 2020b; Caron et al., 2020a; Savarese et al., 2020).

Pruning. In larger-scale settings, IMP subnetworks only become stable and matching after the full network has been trained for some number of steps. Recent proposals attempt to prune networks at initialization (Lee et al., 2019; Wang et al., 2020), but our results suggest that the best time to do so may be after some training. Likewise, most pruning methods only begin to sparsify networks late in training or after training (Han et al., 2015; Gale et al., 2019; He et al., 2018b). The existence of matching subnetworks early in training suggests that there is an unexploited opportunity to prune networks much earlier than current methods.

4.5 Conclusions

In this chapter, I proposed instability analysis as a way to shed light on how SGD noise affects the outcome of optimizing neural networks. I found that standard networks for MNIST, CIFAR-10, and ImageNet become *stable* to SGD noise early in training, after which the outcome of optimization is determined to a linearly connected minimum.

I then applied instability analysis to better understand a key question at the center of the lottery ticket hypothesis: why does iterative magnitude pruning find sparse networks that can train from initialization to full accuracy in smaller-scale settings (e.g., MNIST) but not on more challenging tasks (e.g., ImageNet)? I found that extremely sparse IMP subnetworks only train to full accuracy when they are stable to SGD noise, which occurs at initialization in some settings but only after some amount of training in others. Most importantly, this observation made it possible to identify such subnetworks in much larger-scale settings than the original formulation of this experiment, scaling up the lottery ticket results to a much wider range of settings. Others continue to find new settings where these observations apply.

In addition, instability analysis and the linear mode connectivity criterion have contributed to a growing range of empirical tools for studying and understanding the behavior of neural networks in practice. Although that is not the focus of this thesis, there is now a vibrant literature on this topic (Mirzadeh et al., 2021; Entezari et al., 2022; Wortsman et al., 2021a, 2022; Li et al., 2022).

Chapter 5

The State of the Art in Pruning Neural Networks Early in Training

Since the 1980s, we have known that it is possible to eliminate a significant number of parameters from neural networks without affecting accuracy at inference-time (Reed, 1993; Han et al., 2015). When the goal is to reduce inference costs, pruning often occurs late in training (Zhu & Gupta, 2018; Gale et al., 2019) or after training (LeCun et al., 1990; Han et al., 2015). As the preceding chapters have shown, there is reason to believe it may be possible to prune early in training without affecting final accuracy. Specifically, from early in training (although often after initialization), there exist subnetworks that can train in isolation to full accuracy (Figure 5-1, red line). These subnetworks are as small or nearly as small as those found by inference-focused pruning methods after training, raising the prospect that it may be possible to maintain this level of sparsity for much or all of training. Thus far, however, this thesis has only showed that these subnetworks exist; it has not suggested a way to find these subnetworks without first training the full network.

The pruning literature offers a starting point for finding such subnetworks efficiently. Standard networks are often so overparameterized that pruning randomly has little effect on final accuracy at lower sparsities (green line). Moreover, many existing pruning methods prune during training (Zhu & Gupta, 2018; Gale et al., 2019), even if they were designed with inference in mind (orange line).

Recently, several methods have been proposed specifically for pruning at initialization. SNIP (Lee et al., 2019) aims to prune weights that are least salient for the loss. GraSP (Wang et al., 2020) aims to prune weights that most harm or least benefit gradient flow. SynFlow (Tanaka et al., 2020) aims to iteratively prune weights with the lowest "synaptic strengths" in a data-independent manner with the goal of avoiding *layer collapse* (where pruning concentrates on certain layers).

In this chapter, I assess the efficacy of these pruning methods at initialization. How do SNIP, GraSP, and SynFlow perform relative to each other and naive baselines like random and magnitude pruning? How does this performance compare to methods for pruning after training? What is the state of our ability to prune neural networks early in training—to realize the promise of the preceding chapters? Looking ahead, are there broader challenges particular to pruning at initialization? My purpose is to clarify the state of the art, shed light on the strengths and weaknesses of existing methods, understand their behavior in practice, set baselines, and outline an agenda for the future.

I specifically focus at and near *matching sparsities*: those where magnitude pruning after training matches full accuracy.[1] I do so because: (1) these are the sparsities typically studied in the pruning literature, and (2) for magnitude pruning after training, this is a tradeoff-free regime where a user does not have to balance the benefits of sparsity with sacrifices in accuracy. The experiments considered in this chapter (summarized in Figure 5-2) and findings are as follows:

The state of the art for pruning at initialization. The methods for pruning at initialization (SNIP, GraSP, SynFlow, and magnitude pruning) generally outperform random pruning. No single method is SOTA: there is a network, dataset, and sparsity where each pruning method (including magnitude pruning) reaches the highest accuracy. SNIP consistently performs well, magnitude pruning is surprisingly effective, and competition increases with improvements made to GraSP and SynFlow.

Magnitude pruning after training outperforms these methods. Although

[1]Tanaka et al. design SynFlow to avert *layer collapse*, which occurs at higher sparsities than I consider. However, they also evaluate at matching sparsities, so I believe this is a reasonable setting to study SynFlow.

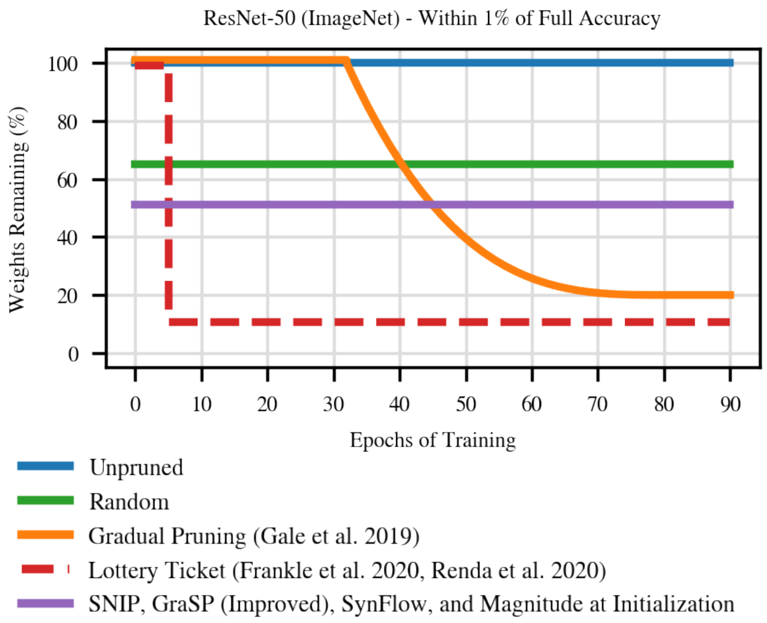


Figure 5-1: Weights remaining at each training step for methods that reach accuracy within one percentage point of baseline top-one accuracy ResNet-50 on ImageNet (baseline accuracy is defined as 76.1%). The dashed line is a result that is achieved retroactively using IMP with rewinding as described in the preceding chapters.

this result is not necessarily surprising (after all, these methods have less readily available information upon which to prune), it raises the question of whether there may be broader limitations to the performance achievable when pruning at initialization. In the rest of the chapter, I study this question, investigating how these methods differ behaviorally from standard results about pruning after training.

Methods prune layers, not weights. The subnetworks that these methods produce perform equally well (or better) when randomly shuffling the weights they prune in each layer; it is therefore possible to describe a family of equally effective pruning techniques that randomly prune the network in these per-layer proportions. The subnetworks that these methods produce also perform equally well when randomly reinitializing the unpruned weights. These behaviors are not shared by state-of-the-art weight-pruning methods that operate after training; both of these ablations (shuffling and reinitialization) lead to lower accuracy (Han et al., 2015).

These results appear specific to pruning at initialization. There are two possible reasons for the comparatively lower accuracy of these methods and for the

Method	Early Pruning Methods					Baseline Methods			Ablations		
	SNIP	GraSP	SynFlow	Magnitude	Random	LTR	Magnitude (After)	Other	Reinit	Shuffle	Invert
SNIP	—	—	—	✗	✗	✗	✗	✗	✓	✗	✗
GraSP	✓	—	—	✗	✓	✓	✗	✓	✗	✗	✗
SynFlow	✓	✓	—	✓	✓	✗	✗	✗	✗	✗	✗

Figure 5-2: Comparisons made in the in the SNIP, GraSP, and SynFlow papers. Dashed lines represent comparisons that are impossible due to the order in which the papers were published. Does not include MNIST experiments. SNIP lacks baselines beyond MNIST. GraSP includes random pruning, LTR, and other methods; it lacks magnitude pruning at initialization and ablations. SynFlow has the other methods at initialization but lacks baselines or ablations.

fact that the resulting networks are insensitive to the ablations: (1) these behaviors are intrinsic to subnetworks produced by these methods or (2) these behaviors are specific to subnetworks produced by these methods at initialization. I eliminate possibility (1) by showing that using SNIP, SynFlow, and magnitude to prune the network after initialization leads to higher accuracy (Section 5.10) and sensitivity to the ablations (Appendix 5.11). This result means that these methods encounter particular difficulties when pruning at initialization.

Looking ahead. These results raise the question of whether it is generally difficult to prune at initialization in a way that is sensitive to the shuffling or reinitialization ablations. If methods that maintain their accuracy under these ablations are inherently limited in their performance, then there may be broader limits on the accuracy attainable when pruning at initialization. Even work on lottery tickets, which has the benefit of seeing the network after training, reaches lower accuracy and is unaffected by these ablations when pruning occurs at initialization.

Although accuracy improves when using SNIP, SynFlow, and magnitude pruning after initialization, it does not match that of the full network unless pruning occurs nearly halfway into training (if at all). This means that (1) it may be difficult to prune until much later in training or (2) we need new methods designed to prune early in training (since SNIP, GraSP, and SynFlow were not intended to do so).

5.1 Background

Until recently, pruning research focused on improving efficiency of inference. However, methods that *gradually prune* throughout training provide opportunities to improve the efficiency of training as well (Zhu & Gupta, 2018; Gale et al., 2019). Lottery ticket work as presented in the preceding chapters shows that there are subnetworks or early in training that can reach full accuracy. Several recent papers have proposed efficient ways to find subnetworks at initialization that can purportedly train to high accuracy. SNIP (Lee et al., 2019), GraSP (Wang et al., 2020), SynFlow (Tanaka et al., 2020), and NTT (Liu & Zenke, 2020) prune at initialization. De Jorge et al. (2020) and Verdenius et al. (2020) apply SNIP iteratively; Cho et al. (2020) use SNIP for pruning for inference. Work on *dynamic sparsity* maintains a pruned network throughout training but regularly changes the sparsity pattern (Mocanu et al., 2018; Dettmers & Zettlemoyer, 2019; Evci et al., 2019); to match the accuracy of standard methods for pruning after training, Evci et al. needed to train for five times as long is standard for training the unpruned networks. You et al. (2020) prune after some training; this research is not directly comparable to any of the aforementioned papers as it prunes channels (rather than weights as in all work above) and does so later (20 epochs) than SNIP/GraSP/SynFlow (0 epochs) and lottery tickets (1-2 epochs).

5.2 Methods

Pruning. Consider a network with weights $w_\ell \in \mathbb{R}^{d_\ell}$ in each layer $\ell \in \{1, \ldots, L\}$. Pruning produces binary *masks* $m_\ell \in \{0,1\}^{d_\ell}$. A pruned *subnetwork* has weights $w_\ell \odot m_\ell$, where $\odot$ is the element-wise product. The *sparsity* $s \in [0,1]$ of the subnetwork is the fraction of weights pruned: $1 - \sum_\ell m_\ell / \sum_\ell d_\ell$. In this chapter, I study pruning methods $\mathsf{prune}(W, s)$ that prune to sparsity s using two operations. First, $\mathsf{score}(W)$ issues *scores* $z_\ell \in \mathbb{R}^{d_\ell}$ to all weights $W = (w_1, \ldots, w_L)$. Second, $\mathsf{remove}(Z, s)$ converts scores $Z = (z_1, \ldots, z_L)$ into masks m_ℓ with overall sparsity s. Pruning may occur in *one shot* (score once and prune from sparsity 0 to s) or *iteratively* (repeatedly score unpruned weights and prune from sparsity $s^{\frac{n-1}{N}}$ to $s^{\frac{n}{N}}$ over iterations $n \in \{1, \ldots, N\}$).

Re-training after pruning. After pruning at step t of training, I subsequently train the network further by repeating the entire learning rate schedule from the start (Renda et al., 2020). Doing so ensures that, no matter the value of t, the pruned network will receive enough training to converge. An alternative would be to train after pruning for $T - t$ steps (where T is the total number of training steps). This would ensure all networks receive T total training steps combined before and after pruning. However, a network pruned late in training (e.g., iteration $T-1$) would receive insufficient training after pruning to recover the accuracy lost from pruning (e.g., one iteration).

Early pruning heuristics. In this chapter, I study the following methods for pruning early in training.

Random pruning. This method issues each weight a random score $z_\ell \sim \mathsf{Uniform}(0, 1)$ and removes weights with the lowest scores. Empirically, it prunes each layer to approximately sparsity s. Random pruning is a naive method for early pruning whose performance any new proposal should surpass.

Magnitude pruning. This method issues each weight its magnitude $z_\ell = |w_\ell|$ as its score and removes those with the lowest scores. Magnitude pruning is a standard way to prune after training for inference (Janowsky, 1989; Han et al., 2015) and is an additional naive point of comparison for early pruning.

SNIP (Lee et al., 2019). This method samples training data, computes gradients g_ℓ for each layer, issues scores $z_\ell = |g_\ell \odot w_\ell|$, and removes weights with the lowest scores in one iteration. The justification for this method is that it preserves weights with the highest "effect on the loss (either positive or negative)." For full details, see Section 5.3.

GraSP (Wang et al., 2020). This method samples training data, computes the Hessian-gradient product h_ℓ for each layer, issues scores $z_\ell = -w_\ell \odot h_\ell$, and removes weights with the highest scores in one iteration. The justification for this method is that it removes weights that "reduce gradient flow" and preserves weights that "increase gradient flow." For full details, see Section 5.4.

SynFlow (Tanaka et al., 2020). This method replaces the weights w_ℓ with $|w_\ell|$. It computes the sum R of the logits on an input of 1's and the gradients $\frac{dR}{dw_\ell}$ of R. It issues scores $z_\ell = |\frac{dR}{dw_\ell} \odot w_\ell|$ and removes weights with the lowest scores. It prunes iteratively (100 iterations). The justification for this method is that it meets criteria that ensure (as proved by Tanaka et al.) it can reach the maximal sparsity before a layer must become disconnected. For full details, see Section 5.5.

Benchmark methods. I use two benchmark methods to illustrate the highest known accuracy attainable when pruning to a particular sparsity in general (not just at initialization). Both methods reach similar accuracy and match full accuracy at the same sparsities. Note: throughout this chapter, I will use one-shot pruning, so accuracy is lower than in work that uses iterative pruning and for the iterative pruning results in previous chapters. I use one-shot pruning to make a fair comparison to the early pruning methods, which do not get to train between iterations.

Magnitude pruning after training. This baseline applies magnitude pruning to the weights at the end of training. Magnitude pruning is a standard method for one-shot pruning after training (Renda et al., 2020). If it is possible to prune early in training and reach the same accuracy as pruning after training, then this baseline represents the corresponding target accuracy. I compare the early pruning methods at initialization against this baseline.

Lottery ticket rewinding (LTR). This baseline uses the mask from magnitude pruning after training and the weights from step t. This is the IMP-with-rewinding technique described in preceding chapters; I refer to it as *lottery ticket rewinding* (LTR) here to clearly distinguish it from the other methods. In the preceding chapters, I have showed that, for t early in training and appropriate sparsities, these subnetworks reach the full accuracy of the unpruned network. One can think of this baseline as emulating an algorithm that prunes at step t using an oracle that possesses information from after training. As I showed in the preceding chapters, accuracy under this pruning method improves for $t > 0$, saturating early in training (Figure 5-12, blue). I compare the early pruning methods after initialization against this baseline.

Sparsities. I divide sparsities into three ranges as in the preceding chapters.

Trivial sparsities are the lowest sparsities: those where the network is so overparameterized that randomly pruning at initialization can still reach full accuracy.

Matching sparsities are moderate sparsities: those where the benchmark methods can match the accuracy of the unpruned network.

Extreme sparsities are those beyond.

I will focus on matching sparsities and the lowest extreme sparsities. Trivial sparsities are addressed by random pruning. Extreme sparsities require making subjective or context-specific tradeoffs between potential efficiency improvements of sparsity and severe drops in accuracy.

Networks, datasets, and replicates. In this chapter, I will continue to study image classification. It is the main (SNIP) or sole (GraSP and SynFlow) task in the papers introducing the early pruning methods and in the papers introducing modern magnitude pruning (Han et al., 2015) and LTR (as in the preceding chapters). I study these methods on ResNet-20 and VGG-16 on CIFAR-10, ResNet-18 on TinyImageNet, and ResNet-50 on ImageNet. I repeat all experiments five (CIFAR-10) or three (TinyImageNet and ImageNet) times with different seeds and plot the mean and standard deviation.

5.3 Replicating SNIP

In this section, I describe and evaluate the replication of SNIP used in this chapter (Lee et al., 2019). Comparing pruning methods across papers and setups is notoriously difficult (Blalock et al., 2020). Rather than use existing code, I have reimplemented SNIP in the same codebase as the other lottery ticket experiments in this thesis. This section serves to evaluate whether that reimplementation appears to be correct insofar as it matches the results from the original paper and other implementations in the literature.

5.3.1 Algorithm

SNIP introduces a virtual parameter $c_i \in 0, 1$ as a coefficient for each parameter w_i. Initially, SNIP assumes that $c_i = 1$.

SNIP assigns each parameter w_i a score $s_i = \left|\frac{\partial L}{\partial c_i}\right|$ and prunes the parameters with the lowest scores.

This algorithm entails three key design choices:

1. Using the derivative of the loss with respect to c_i as a basis for scoring.

2. Taking the absolute value of this derivative.

3. Pruning weights with the lowest scores.

Lee et al. (2019) explain these choices as follows: "if the magnitude of the derivative is high (regardless of the sign), it essentially means that the connection c_i has a considerable effect on the loss (either positive or negative) and it has to be preserved to allow learning on w_i."

5.3.2 Implementation Details

Algorithm. It is possible to rewrite the score as follows. Let a_i be the incoming activation that is multiplied by w_i. Let z be the pre-activation of the neuron to which w_i serves as an input.

$$s_i = \left|\frac{\partial L}{\partial c_i}\right| = \left|\frac{\partial L}{\partial z}\frac{\partial z}{\partial c_i}\right| = \left|\frac{\partial L}{\partial z}a_i w_i\right| = \left|\frac{\partial L}{\partial z}\frac{\partial z}{\partial w_i}w_i\right| = \left|\frac{\partial L}{\partial w_i}w_i\right|$$

In summary, one can rewrite s_i as the gradient of w_i multiplied by the value of w_i. This is the formula used in the implementation in this chapter.

Selecting examples for SNIP. Lee et al. (2019) use a single mini-batch for SNIP. To create the mini-batch used for SNIP, I follow the strategy used by Wang et al. (2020) for GraSP: create a mini-batch composed of ten examples selected randomly from each class.

Reinitializing. After pruning, Lee et al. (2019) reinitialize the network.[2] I do not.

Running on CPU. To avoid any risk that distributed training might affect results, I run SNIP computation on CPU and then train the pruned network on TPU.

5.3.3 Networks and Datasets

Lee et al. consider the following networks for computer vision:

SNIP Name	Name in This Thesis	Dataset	GitHub	Replicated	Notes
LeNet-300-100	LeNet-300-100	MNIST	✓	✗	Fully-connected
LeNet-5	—	MNIST	✓	✗	Convolutional
AlexNet-s	—	CIFAR-10	✓	✗	
AlexNet-b	—	CIFAR-10	✓	✗	
VGG-C	—	CIFAR-10	✓	✗	VGG-D but some layers have 1x1 convolutions
VGG-D	VGG2-16	CIFAR-10	✓	✓	VGG-like but with two fully-connected layers
VGG-like	VGG-16	CIFAR-10	✓	✓	VGG-D but with one fully-connected layer
WRN-16-8	WRN-14-8	CIFAR-10	✗	✓	
WRN-16-10	WRN-14-10	CIFAR-10	✗	✓	
WRN-22-8	WRN-20-8	CIFAR-10	✗	✓	

5.3.4 Results

A GitHub repository associated with the paper[3] includes all of those networks except for the wide ResNets (WRNs). I have made my best effort to replicate a subset of these networks in the research framework used in this thesis The table below shows the results from this replication. Each of my numbers is the average across five replicates with different random seeds.

[2]See discussion on OpenReview.
[3]`https://github.com/namhoonlee/snip-public/`

Name	Unpruned Accuracy		Sparsity	Pruned Accuracy	
	Reported	This Thesis		Reported	This Thesis
VGG-16	91.7%	93.6%	97%	92.0% (+0.3)	92.1% (−1.5)
VGG2-16	93.2%	93.5%	95%	92.9% (−0.3)	92.3% (−1.2)
WRN-14-8	93.8%	95.2%	95%	93.4% (−0.4)	93.4% (−1.8)
WRN-14-10	94.1%	95.4%	95%	93.6% (−0.5)	93.9% (−1.4)
WRN-20-8	93.9%	95.6%	95%	94.1% (+0.3)	94.3% (−1.2)

Unpruned networks. The unpruned networks implemented by Lee et al. (2019) appear poorly tuned such that they do not achieve standard performance levels. The accuracy of my VGG-16 is 1.9 percentage points higher, and the accuracies of the wide ResNets from this thesis are between 1.3 and 1.7 percentage points higher. In general, implementation details of VGG-style networks for CIFAR-10 vary widely (Blalock et al., 2020), so some differences are to be expected. However, ResNets for CIFAR-10 are standardized (He et al., 2016; Zagoruyko & Komodakis, 2016), and the accuracies from this thesis identical to those reported by Zagoruyko & Komodakis in the paper that introduced wide ResNets.

Pruned networks. After applying SNIP, the accuracies from this thesis more closely match those reported in the paper. However, since those networks started at higher accuracies, the SNIP-pruned accuracy values represent much larger drops in performance than reported in the original paper. Overall, the results after applying SNIP appear to match those reported in the paper, giving some confidence that the implementation used in this thesis is correct. However, since the accuracies of the unpruned networks in SNIP are lower than standard values, it is not certain.

5.3.5 SNIP Results from GraSP Paper

The paper that introduces GraSP (Wang et al., 2020) also replicates SNIP.[4] The table below contains a comparison of their reported results with the results from the codebase used for this thesis on the networks studied in Section 5.4.

Name	Unpruned Accuracy		Results		
	Reported	This Thesis	Sparsity	Reported	This Thesis
VGG-19	94.2%	93.5%	90%	93.6% (-0.6)	93.5% (-0.0)
			95%	93.4% (-0.8)	93.4% (-0.1)
			98%	92.1% (-2.1)	diverged
WRN-32-2	94.8%	94.5%	90%	92.6% (-2.2)	92.5% (-2.0)
			95%	91.1% (-3.7)	91.0% (-3.5)
			98%	87.5% (-7.3)	87.7% (-6.8)
ResNet-50	75.7%	76.2%	60%	74.0% (-1.7)	73.9% (-2.3)
			80%	69.7% (-6.0)	71.2% (-5.0)
			90%	62.0% (-13.7)	65.7% (-10.5)

Unpruned networks. See Section 5.4 for a full discussion of the unpruned networks. The VGG-19, WRN-32-2, and ResNet-50 results from this thesis reach slightly different accuracies from those of Wang et al. (2020), but the performance differences are much smaller than for the unpruned networks from the SNIP paper.

Pruned networks. Although performance of the unpruned VGG-19 network from this thesis is lower than that of Wang et al., the SNIP performances are identical at 90% and 95% sparsity. This outcome is similar to the comparison to the original SNIP paper, with similar pruned accuracy despite different unpruned accuracy.

On ResNet-50 for ImageNet, the unpruned and SNIP accuracies from this thesis are higher than those reported in by Wang et al. (2020). This may be a result of

[4]Although Wang et al. (2020) have released an open-source implementation of GraSP, this code does not include their implementation of SNIP. It is unclear which hyperparameters they used for SNIP.

different hyperparameter choices for training the network; see Section 5.4 for full details. This may also be a result of different hyperparameter choices for SNIP. My implementation may use a different number of examples than Wang et al. to compute the SNIP gradients and may select these examples differently (although, since Wang et al. did not release their SNIP code, it is impossible to be certain); see Section 5.4 for full details.

5.4 Replicating GraSP

In this section, I describe and evaluate the replication of GraSP used in this chapter (Wang et al., 2020). Comparing pruning methods across papers and setups is notoriously difficult (Blalock et al., 2020). Rather than use existing code, I have reimplemented GraSP in the same codebase as the other lottery ticket experiments in this thesis. This section serves to evaluate whether that reimplementation appears to be correct insofar as it matches the results from the original paper and other implementations in the literature.

5.4.1 Algorithm

Scoring parameters. GraSP is designed to preserve the gradient flow through the sparse network that results from pruning. To do so, it attempts to prune weights in order to maximize the change in loss that takes place after the first step of training. Concretely, let $\Delta L(w)$ be the change in loss due to the first step of training:[5]

$$\Delta L(w) = L(w + \eta \cdot \nabla L(w)) - L(w)$$

where η is the learning rate. Since GraSP focuses on gradient flow, it takes the limit as η goes to 0:

[5]I believe this quantity should instead be specified as $L(w) - L(w - \eta \cdot \nabla L(w))$. The gradient update goes in the negative direction, so one should subtract the expression $\eta \cdot \nabla L(w)$ from the original initialization w. I expect loss to decrease after taking this step, so—if $\Delta L(w)$ should capture the improvement in loss—the updated loss must be subtracted from the original loss.

$$\Delta L(w) = \lim_{\eta \to 0} \frac{L(w + \eta \cdot \nabla L(w)) - L(w)}{\eta} \approx \nabla L(w)^\top \nabla L(w) \tag{5.1}$$

The last expression emerge by taking a first-order Taylor expansion of $L(w + \eta \cdot \nabla L(w))$. GraSP treats pruning the network as a perturbation δ transforming the original parameters w into perturbed parameters $w+\delta$. The effect of this perturbation on the change in loss can be measured by comparing $\Delta L(w + \delta)$ and $\Delta L(w)$:

$$C(\delta) = \Delta L(w + \delta) - \Delta L(w) = \nabla L(w + \delta)^\top \nabla L(w + \delta) - \nabla L(w)^\top \nabla L(w) \tag{5.2}$$

Finally, GraSP takes the first-order Taylor approximation of the left term about w, yielding:

$$\begin{aligned} C(\delta) &\approx \nabla L(w)^\top \nabla L(w) + 2\delta^\top \nabla^2 L(w) \nabla L(w) + O(||\delta||_2^2) - \nabla L(w)^\top \nabla L(w) \\ &= 2\delta^\top \nabla^2 L(w) \nabla L(w) + O(||\delta||_2^2) \\ &= 2\delta^\top H g \end{aligned} \tag{5.3}$$

where H is the Hessian and g is the gradient. Pruning an individual parameter w_i at index i involves creating a vector $\delta^{(i)}$ where $\delta_i^{(i)} = -w_i$ and $\delta_j^{(i)} = 0$ for $j \neq i$. The resulting pruned parameter vector is $w - \delta^{(i)}$; this vector is identical to w except that w_i has been set to 0. Using the analysis above, the effect of pruning parameter w_i in this manner on the gradient flow is approximated by $C(-\delta^{(i)}) = -w_i(Hg)_i$. GraSP therefore gives each weight the following score:

$$s_i = C(-\delta^{(i)}) = -w_i(Hg)_i \tag{5.4}$$

Using scores to prune. To use s_i for pruning, Wang et al. make the following interpretation:

GraSP uses [Equation 5.3] as the measure of the importance of each weight.

Specifically, if $C(\delta)$ is negative, then removing the corresponding weights will reduce gradient flow, while if it is positive, it will increase gradient flow.

In other words, parameters with lower scores are more important (since removing them will have a less beneficial or more detrimental impact on gradient flow) and parameters with higher scores are less important (since removing them will have a more beneficial or less detrimental impact on gradient flow). Since the goal of GraSP is to maximize gradient flow after pruning, it should prune "those weights whose removal will not reduce the gradient flow," i.e., those with the highest scores.

5.4.2 Implementation Details

Algorithm. The implementation of GraSP in this thesis follows the PyTorch implementation provided by the authors on GitHub[6] (which computes the Hessian-gradient product according to Algorithm 2 of the paper).

Selecting examples for scoring parameters. To create the mini-batch that used to compute the GraSP scores, I follow the strategy used by Wang et al. (2020) for the CIFAR-10 networks in their implementation: randomly sample ten examples from each class. The implementation in this thesis uses this approach for both CIFAR-10 and ImageNet; on ImageNet, this means I use 10,000 examples representing all ImageNet classes.

It is not entirely clear how Wang et al. select the mini-batch for the ImageNet networks in their experiments. In their configuration files, Wang et al. appear to use one example per class (1000 in total covering all classes). In their ImageNet implementation (which ignores their configuration files), they use 150 mini-batches where the batch size is 128 (19,200 examples covering an uncertain number of classes).

Reinitializing. I do not reinitialize after pruning.

[6]`https://github.com/alecwangcq/GraSP`

Running on CPU. To avoid any risk that distributed training might affect results, all GraSP computation is run on the CPU, and the pruned network is subsequently trained on TPU.

5.4.3 Networks and Datasets

Wang et al. consider the networks for computer vision in the below. They use both CIFAR-10 and CIFAR-100 for all CIFAR-10 networks, while I only use CIFAR-10. Note that the hyperparameters for ResNet-50 on ImageNet in this thesis differ from those in the GraSP implementation (likely due to different hardware): the implementation in this thesis uses a larger batch size (1024 vs. 128) and a higher learning rate (0.4 vs. 0.1).

GraSP Name	My Name	Dataset	GitHub	Replicated	Notes
VGG-19	VGG-19	CIFAR-10	✓	✓	
ResNet-32	WRN-32-2	CIFAR-10	✓	✓	Wang et al. use twice the standard width.
ResNet-50	ResNet-50	ImageNet	✓	✓	ResNets for CIFAR-10 and ImageNet are different.
VGG-16	—	ImageNet	✗	✗	VGGs for CIFAR-10 and ImageNet are different.

5.4.4 Results

The GitHub implementation of GraSP by Wang et al. (2020) includes all of the networks from the above table except VGG-16. I have made my best effort to replicate these networks in the research framework used for this thesis. The table below shows the results from this replication. Each numbers is the average across five replicates with different random seeds.

Name	Unpruned Accuracy		Sparsity	Pruned Accuracy	
	Reported	This Thesis		Reported	This Thesis
VGG-19	94.2%	93.5%	90%	93.3% (−0.9)	92.8% (−0.7)
			95%	93.0% (−1.2)	92.5% (−1.0)
			98%	92.2% (−2.0)	91.6% (−1.9)
WRN-32-2	94.8%	94.5%	90%	92.4% (−2.4)	92.2% (−2.3)
			95%	91.4% (−3.4)	90.9% (−3.6)
			98%	88.8% (−6.0)	88.3% (−6.2)
ResNet-50	75.7%	76.2%	60%	74.0% (−1.7)	73.4% (−2.8)
			80%	72.0% (−6.0)	71.0% (−5.2)
			90%	68.1% (−7.6)	67.0% (−9.2)

Unpruned networks. My unpruned networks perform similarly to those of Wang et al. (2020). Although I made every effort to replicate the architecture and hyperparameters of the GraSP implementation of VGG-19, the average accuracy is 0.7 percentage points lower.[7] Accuracy on WRN-32-2 is closer, differing by only 0.3 percentage points. Accuracy on ResNet-50 for ImageNet is higher by half a percentage point, likely due to using different hyperparameters.

Pruned networks. The pruned VGG-19 and WRN-32-2 also reach lower accuracy than those of Wang et al.; this difference is commensurate with the difference between the unpruned networks. On VGG-19, the accuracies of the pruned networks are lower than those of Wang et al. by 0.5 to 0.6 percentage points, matching the drop of 0.7 percentage points for the unpruned network. Similarly, on WRN-32-2, the accuracies of the pruned networks are lower than those of Wang et al. by 0.2 to 0.5 percentage points, matching the drop of 0.3 percentage points for the unpruned networks. In both cases, the decrease in performance after pruning (inside the parentheses in the table above) is nearly identical between the two papers, differing by no more than 0.2 percentage points at any sparsity. I conclude that the behavior of the implementation used in this thesis matches that of Wang et al., although starting from slightly lower baseline accuracy.

[7]VGG networks for CIFAR-10 are notoriously difficult to replicate (Blalock et al., 2020).

The accuracy of the pruned ResNet-50 networks is less consistent with that of Wang et al.. Despite starting from a higher baseline, the accuracy after pruning is lower by 0.6 to 1.1 percentage points. These differences are potentially due to different hyperparameters: as mentioned previously, the implementation in this thesis selects examples for GraSP differently than Wang et al. and trains with a different batch size and learning rate.

5.5 Replicating SynFlow

Here, I describe and evaluate my replication of SynFlow (Tanaka et al., 2020).

5.5.1 Algorithm

SynFlow is an iterative pruning algorithm. It prunes to sparsity s over the course of N iterations, pruning from sparsity $s^{\frac{n-1}{N}}$ to sparsity $s^{\frac{n}{N}}$ on each iteration $n \in \{1, ..., N\}$. On each iteration, it issues scores to the remaining, unpruned weights and then removes those with the lowest scores.

Synflow scores weights as follows:

1. It replaces all parameters w_ℓ with their absolute values $|w_\ell|$.

2. It forward propagates an input of all ones through the network.

3. It computes the sum of the logits R.

4. It computes the gradient of R with respect to each weight $|w|$: $\frac{dR}{dw}$.

5. It issues the score $|\frac{dR}{dw} \cdot w|$ for each weight.

Tanaka et al. (2020) explain these choices as follows. Their goal is to create a pruning technique that "provably reaches Maximal Critical Compression," i.e., a pruning technique that ensures that the network remains connected until the most extreme sparsity where it is possible to do so. As they prove, any pruning technique that is iterative, issues positive scores, and is *conservative* (i.e., the sum of the incoming and outgoing scores for a layer are the same), then it will reach maximum critical compression (Theorem 3). The iterative requirement is that scores are recalculated after each parameter is pruned; in their experiments, Tanaka et al. use 100 iterations in order to make the process more efficient.

5.5.2 Implementation Details

Iterative pruning. I use 100 iterations, the same as Tanaka et al. (2020) use (as noted in the appendices).

Input size. I use an input size that is the same as the input size for the corresponding dataset. For example, for CIFAR-10, I use an input that is 32x32x3; for ImageNet, I use an input that is 224x224x3.

Reinitializing. After pruning, Tanaka et al. (2020) do not reinitialize the network. I do not reinitialize either.

Running on CPU. To avoid any risk that distributed training might affect results, I run all SynFlow computation on CPU and subsequently train the pruned network on TPU.

Double precision floats. I compute the SynFlow scores using double precision floats. With single precision floating point numbers, the SynFlow activations explode on networks deeper than ResNet-44 (CIFAR-10) and ResNet-18 (ImageNet).

5.5.3 Networks and Datasets

Tanaka et al. consider the settings in Table 5.1. Of the settings that I replicated, my unpruned network performance is in Table 5.2.

The VGG-11 and VGG-16 CIFAR-10 results are identical between my implementation and that of Tanaka et al..

SynFlow Name	My Name	Dataset	GitHub	Replicated	Notes
VGG-11	VGG-11	CIFAR-10	✓	✓	A shallower version of my VGG-16 network
VGG-11	VGG-11	CIFAR-100	✓	✗	A shallower version of my VGG-16 network
VGG-11	VGG-11 (Modified)	TinyImageNet	✓	✗	Modified for TinyImageNet
VGG-16	VGG-16	CIFAR-10	✓	✓	Identical to my VGG-16 network
VGG-16	VGG-16	CIFAR-100	✓	✗	Identical to my VGG-16 network
VGG-16	VGG-16 (Modified)	TinyImageNet	✓	✗	Modified for TinyImageNet
ResNet-18	ResNet-18 (Modified)	CIFAR-10	✓	✓	Modified ImageNet ResNet-18; first conv is 3x3 stride 1; no max-pool
ResNet-18	ResNet-18 (Modified)	CIFAR-100	✓	✗	Modified ImageNet ResNet-18; first conv is 3x3 stride 1; no max-pool
ResNet-18	ResNet-18 (Modified)	TinyImageNet	✓	✓	Modified ImageNet ResNet-18; first conv is 3x3 stride 1; no max-pool

Table 5.1: Networks and datasets examined in the SynFlow paper (Tanaka et al., 2020)

Network	Dataset	Reported	Replicated	Notes
VGG-11	CIFAR-10	~92%	92.0%	Hyperparameters and augmentation are identical to ours
VGG-16	CIFAR-10	~94%	93.5%	Hyperparameters and augmentation are identical to ours
ResNet-18 (Modified)	CIFAR-10	~95%	93.7%	Hyperparameters reported by Tanaka et al. (lr=0.01, batch size=128, drop factor=0.2)
ResNet-18 (Modified)	CIFAR-10	—	94.6%	Hyperparameters reported by Tanaka et al. (lr=0.2, batch size=256, drop factor=0.1)
ResNet-18 (Modified)	TinyImageNet	~64%	58.8%	Hyperparameters reported by Tanaka et al. (lr=0.01, batch size=128, epochs=100)
ResNet-18 (Modified)	TinyImageNet	—	64%	Modified hyperparameters (lr=0.2, batch size=256, epochs=200)

Table 5.2: Top-1 accuracy of unpruned networks as reported by Tanaka et al. (2020) and as replicated. Tanaka et al. showed plots rather than specific numbers, so reported numbers are approximate.

I modified the standard ImageNet ResNet-18 from TorchVision to match the network of Tanaka et al.. I used the same data augmentation and hyperparameters on TinyImageNet, but accuracy was much lower (58.8% vs. 64%). By increasing the learning rate from 0.01 to 0.2, increasing the batch size to 256, and increasing the number of training epochs to 200, I was able to match the accuracy reported by Tanaka et al.. I also used the same data augmentation and hyperparameters on CIFAR-10, but accuracy was lower (95% vs. 93.6%). Considering that I needed different hyperparameters on both datasets, I believe that there is an unknown difference between my ResNet-18 implementation and that of Tanaka et al..

5.5.4 Results

Tanaka et al. compare to random pruning, magnitude pruning at initialization, SNIP, and GraSP at 13 sparsities evenly spaced logarithmically between 0% sparsity and 99.9% sparsity (compression ratio 10^3). In Figure 5-3, I plot the same methods at sparsities between 0% and 99.9% at intervals of an additional 50% sparsity (e.g., 50% sparsity, 75% sparsity, 87.5% sparsity, etc.).

Tanaka et al. present graphs rather than tables of numbers. As such, in Figure 5-3, I compare my graphs (left) to the graphs from the SynFlow paper (right). On the VGG-style networks for CIFAR-10, my results look nearly identical to those of Tanaka et al. in terms of the accuracies of each method, the ordering of the methods, and when certain methods drop to random accuracy. The only difference is that SNIP encounters layer collapse in my experiments, while it does not in those of Tanaka et al.. Since these models share the same architecture and hyperparameters as in the SynFlow paper and the results look very similar, I have confidence that my implementation of SynFlow and the other techniques matches that of Tanaka et al..

My ResNet-18 experiments look quite different from those of Tanaka et al.. The ordering of the lines is different, SynFlow is not the best performing method at the most extreme sparsities, and magnitude pruning does not drop to random accuracy. Considering the aforementioned challenges replicating Tanaka et al.'s performance on the unpruned ResNet-18, I attribute these differences to an unknown difference in my model or training configuration. Possible causes include different hyperparameters (which may cause both the original model and the pruned networks to perform differently). Another possible cause is a different initialization scheme: I initialize the γ parameters of BatchNorm uniformly between 0 and 1 and use He normal initialization based on the fan-in of the layer (both PyTorch defaults) while Tanaka et al. initialize the γ parameters to 1 and use He normal initialization based on the fan-out of the layer. Although these differences in the initialization scheme are small, they could make a substantial difference for methods that prune at initialization. This difference in results, despite the fact that both unpruned ResNet-18 networks reach the same accuracy on TinyImageNet, suggest that there may be a significant degree of brittleness in SynFlow, at least at the most extreme sparsities.

The GraSP implementation of Tanaka et al. (2020) contains a bug: it uses batchnorm in evaluation mode rather than training mode, which may also explain some of the differences seen in GraSP performance, especially on Modified ResNet-18.

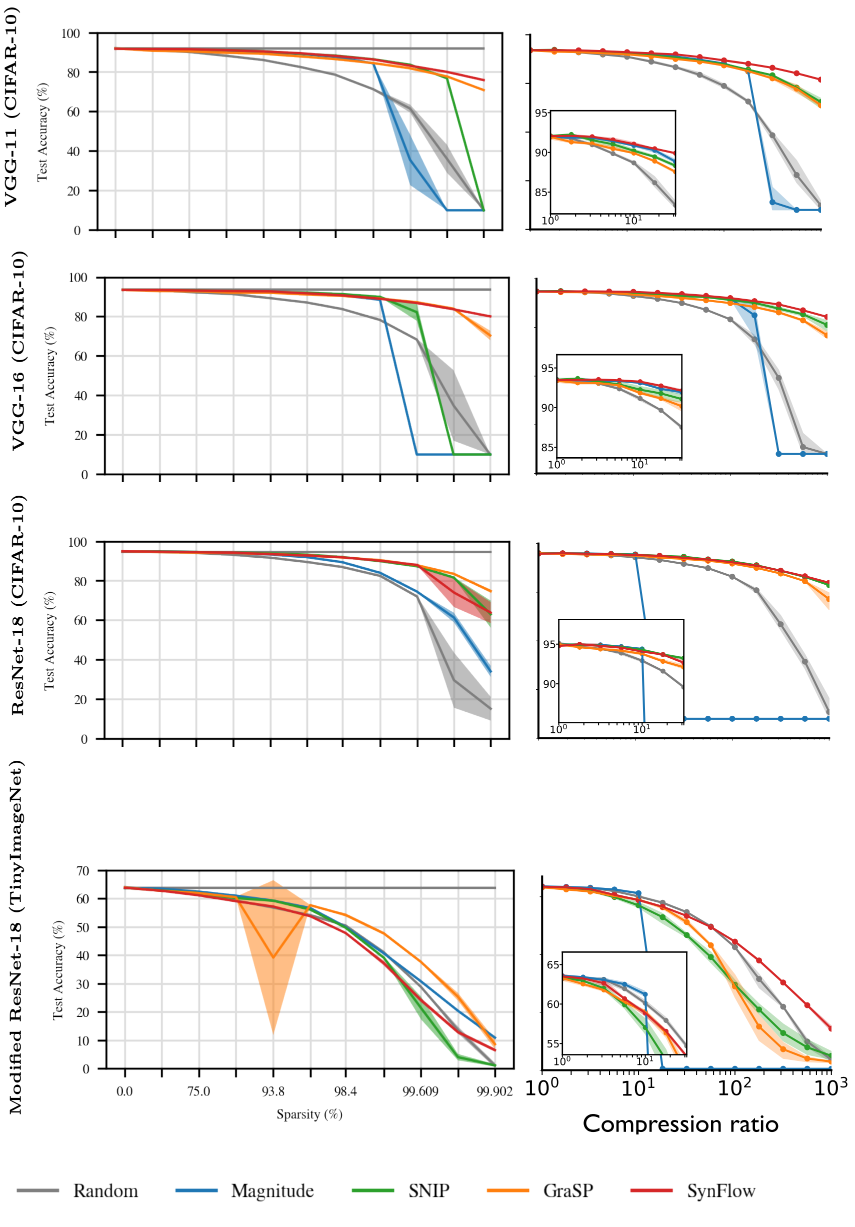


Figure 5-3: Synflow replication experiments.

5.6 Pruning at Initialization

In this section, I evaluate the early pruning methods at initialization, the point in training where Chapter 3 indicates it may be possible to find matching subnetworks and the point in training where SNIP, GraSP, and SynFlow were designed to be used. As summarized in Figure 5-2, this is the first apples-to-apples comparison between all of these early pruning methods. The SynFlow paper only includes other early pruning methods (Tanaka et al., 2020), the GraSP paper does not compare against magnitude pruning at initialization (Wang et al., 2020), and the SNIP paper does not include any baselines for its main CIFAR-10 results. Moreover, the SNIP and GraSP papers include only specific hand-picked sparsities rather than the full range of possible sparsities (Blalock et al., 2020). Figure 5-4 shows the performance of magnitude pruning (green), SNIP (red), GraSP (purple), and SynFlow (brown) at initialization. For context, it also includes the accuracy of pruning after training (blue), random pruning at initialization (orange), and the unpruned network (gray).

Matching sparsities. For matching sparsities,[8] SNIP, SynFlow and magnitude all dominate in at least one sitaution depending on the network. On ResNet-20, all methods are similar in performance but magnitude pruning at initialization reaches the highest accuracy. On VGG-16, SynFlow slightly outperforms SNIP until 91.4% sparsity, after which SNIP overtakes it; magnitude pruning at initialization and GraSP are at most 0.4 and 0.9 percentage points below the best method. On ResNet-18, SNIP and SynFlow remain even until 79.0% sparsity, after which SNIP dominates; magnitude pruning at initialization and GraSP are at most 2.1 and 2.6 percentage points below the best method. On ResNet-50, SNIP, SynFlow, and magnitude pruning at initialization perform similarly, 0.5 to 1 percentage points above GraSP.

In some cases, methods are able to reach full accuracy at non-trivial sparsities. On VGG-16, SNIP and SynFlow do so until 59% sparsity (vs. 20% for random pruning and 93.1% for magnitude pruning after training). On ResNet-18, SNIP does so until

[8]As a reminder, in this chapter, matching sparsities are defined those where magnitude pruning after training matches full accuracy. These are sparsities $\leq$ 73.8%, 93.1%, 96.5%, and 67.2% for ResNet-20, VGG-16, ResNet-18, and ResNet-50.

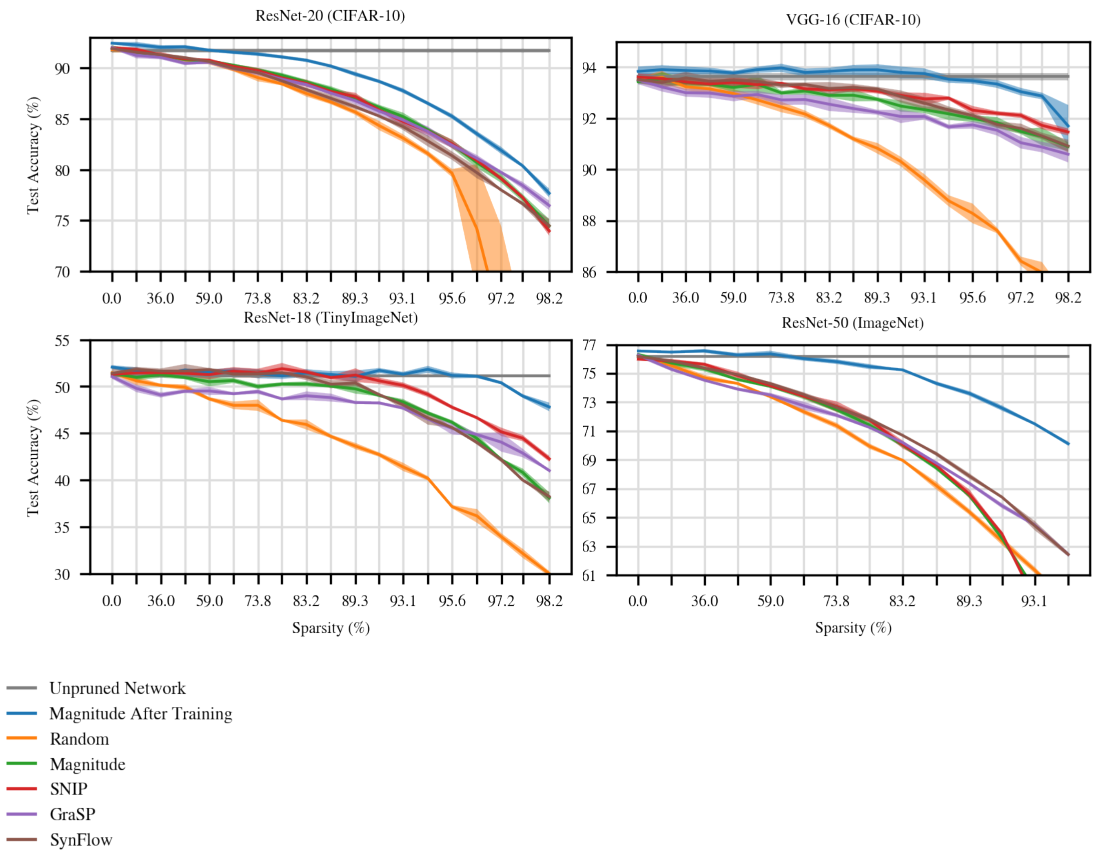


Figure 5-4: Accuracy of early pruning methods when pruning at initialization to various sparsities.

89.3% sparsity and SynFlow until 79% sparsity (vs. 20% for random pruning and 96.5% for magnitude pruning after training). On ResNet-20 and ResNet-50, no early pruning methods reach full accuracy at non-trivial sparsities; one explanation for this behavior is that these settings are more challenging and less overparameterized in the sense that magnitude pruning after training drops below full accuracy at lower sparsities.

Extreme sparsities. At extreme sparsities,[9] the ordering of methods is the same on VGG-16 but changes on the ResNets. On ResNet-20, magnitude pruning at initialization and SNIP drop off, GraSP overtakes the other methods at 98.2% sparsity, and SynFlow performs worst; ResNet-50 shows similar behavior except that SynFlow performs best. On ResNet-18, GraSP overtakes magnitude pruning at initialization and SynFlow but not SNIP.

Summary. No one method is the highest-performance choice in all settings and sparsities. SNIP consistently performs well, and SynFlow frequently competitive. Magnitude pruning at initialization is surprisingly effective against more complicated heuristics. GraSP performs worst at matching sparsities but does better at extreme sparsities. Overall, the methods generally outperform random pruning; however, they cannot match magnitude pruning after training in terms of either accuracy or the sparsities at which they match full accuracy. There remains a gap in performance.

5.7 Ablations at Initialization

In this section, I evaluate the information that each method extracts about the network at initialization in the process of pruning. My goal is to understand how these methods behave in practice and gain insight into why they perform differently than magnitude pruning after training.

[9]As a reminder, in this chapter, extreme sparsities are those beyond which magnitude pruning after training reaches full accuracy.

5.7.1 Random Shuffling

I first consider whether these pruning methods prune specific connections. To do so, I randomly shuffle the pruning mask m_ℓ within each layer. If accuracy is the same after shuffling, then the per-weight decisions made by the method can be replaced by the per-layer fraction of weights it pruned. If accuracy changes, then the method has determined which parts of the network to prune at a smaller granularity than layers, e.g., neurons or individual connections.

Overall. All methods maintain accuracy or improve when randomly shuffled (Figure 5-5, orange line). In other words, the useful information these techniques extract is not which individual weights to remove, but rather the layerwise proportions by which to prune the network.[10] Although layerwise proportions are an important hyperparameter for inference-focused pruning methods (He et al., 2018b; Gale et al., 2019), proportions alone are not sufficient to explain the performance of those methods. For example, as the bottom row of Figure 5-5 shows, magnitude pruning after training makes pruning decisions specific to particular weights; shuffling in this manner reduces performance. The same is true in the IMP and IMP-with-rewinding experiments presented in the preceding chapters: randomly pruning in the same layerwise proportions as IMP substantially reduces accuracy at matching sparsities. In general, the community knows of no state-of-the-art weight-pruning methods (among the many that exist (Blalock et al., 2020)) with accuracy robust to this ablation. This raises the question of whether the insensitivity to shuffling for the early pruning methods may limit their performance.

Magnitude pruning at initialization. Since the masks from magnitude pruning at initialization can be shuffled within each layer, its pruning decisions are sensitive only to the per-layer initialization distributions that determine the magnitudes of the

[10]Note that there are certainly pathological cases where these proportions alone are not sufficient to match the accuracy of the pruning techniques. For example, at the extreme sparsities studied by Tanaka et al. (2020), pruning randomly could lead the network to become disconnected. An unlucky random draw could also lead the network to become disconnected. However, I do not observe these behaviors in any of the experiments.

weights in each layer. In the experiments presented here, these distributions are chosen using He initialization: normal with a per-layer variance determined by the fan-in or fan-out (He et al., 2015). These variances alone, then, are sufficient information to attain the performance of magnitude pruning—performance often competitive with SNIP, GraSP, and SynFlow. Without this information, magnitude pruning performs worse (purple line): if each layer is initialized with variance 1, it will prune all layers by the same fraction no differently than random pruning. This does not affect SNIP,[11] GraSP, or SynFlow, showing a previously unknown benefit of these methods: they maintain accuracy in a case where the initialization is not informative for pruning in this way.

SynFlow. On ResNet-20 and ResNet-18, SynFlow accuracy counter-intuitively improves at extreme sparsities when shuffling. This is surprising, considering that all other examples of this ablation in this thesis hurt accuracy and shuffling would seem to destroy information. I hypothesize that this behavior is connected to a pathology of SynFlow that I term *neuron collapse*: SynFlow prunes entire neurons (in this case, convolutional channels) at a higher rate than other methods. Figure 5-6 shows the fraction of neurons with sparsity $\geq s\%$ (for $s \in [0, 100]$) for each early pruning method. At the highest matching sparsities, SynFlow prunes 31%, 52%, 69%, and 29% of neurons on ResNet-20, VGG-16, and ResNet-18, and ResNet-50, respectively. In contrast, SNIP prunes 5%, 11%, 32%, and 7% of neurons; GraSP prunes 1%, 6%, 14%, and 1% of neurons; and magnitude pruning at initialization prunes 0%, 0%, $<$ 1%, and 0% of neurons. Shuffling SynFlow layerwise reduces these numbers to 1%, 0%, 3.5%, and 13%[12] (orange line). I hypothesize that, by pruning neurons, SynFlow reduces the representation capacity; randomly shuffling restores these neurons and the lost capacity, improving performance.

Neuron collapse may be inherent to SynFlow. From another angle, SynFlow works

[11]Note that initialization can still affect the performance of SNIP. In particular, Lee et al. (2020) demonstrate a scheme for adjusting the initialization of a SNIP-pruned network after pruning that leads to higher accuracy.

[12]This number remains high for ResNet-50: half of the pruned neurons are in layers that get pruned entirely, namely skip connections that downsample using 1x1 convolutions (see Section 5.9).

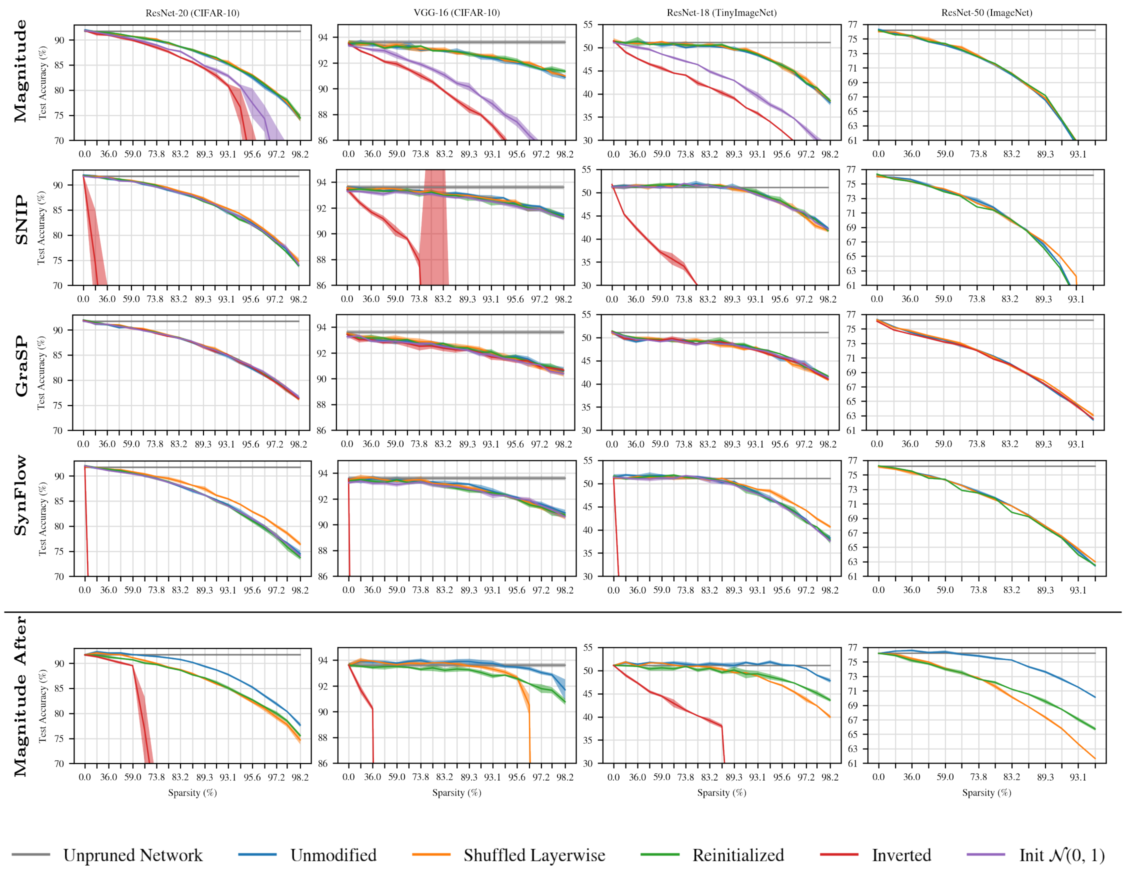


Figure 5-5: Ablations on subnetworks found by applying magnitude pruning, SNIP, GraSP, and SynFlow at initialization. The bottom row shows the effect of these ablations on magnitude pruning after training as a basis for comparison. For all methods that prune at initialization, it is possible to randomly shuffle which connections are pruned in each layer or randomly reinitialize the unpruned weights and still reach the same accuracy, indicating that these methods are not sensitive to which specific connections are pruned within each layer or their weights. This is in stark contrast to magnitude pruning after training, which is sensitive to both of these ablations. (This figure includes limited ablations on ResNet-50 due to resource limitations.)

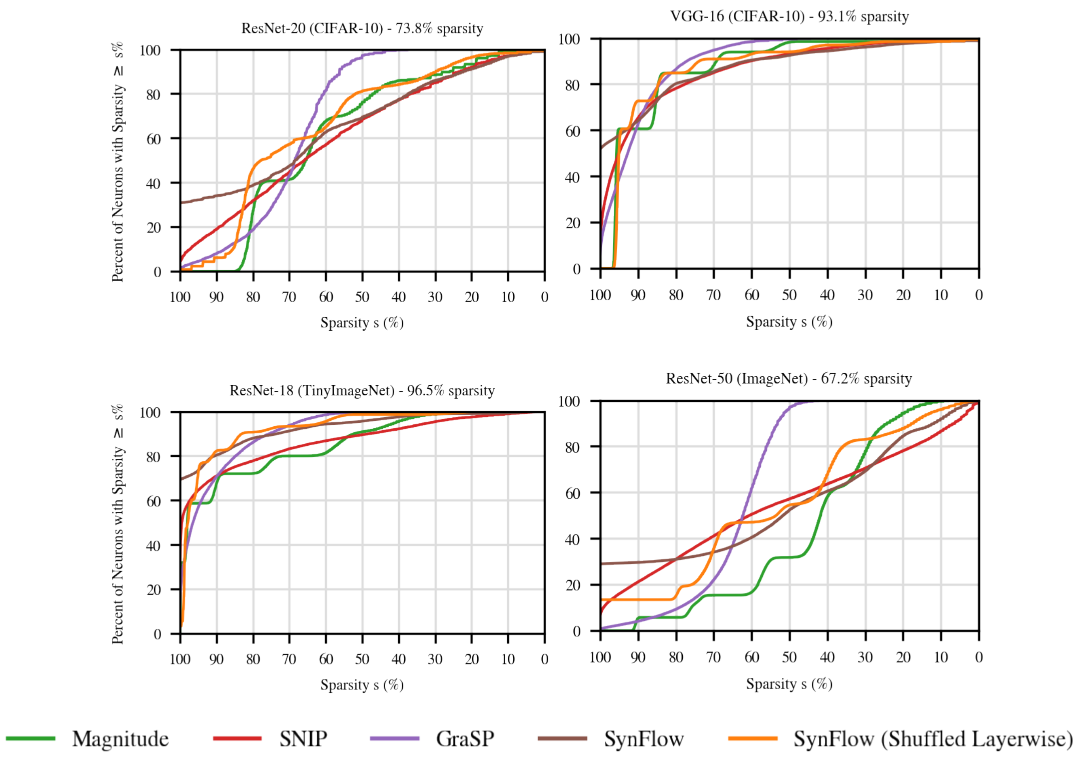


Figure 5-6: Percent of neurons (specifically, convolutional channels in this case) with sparsity $\geq s\%$ at the highest matching sparsity. SynFlow prunes far more neurons than the other methods at any given sparsity.

as follows: consider all paths $p = \{w_p^{(\ell)}\}_\ell$ from any input node to any output node. The SynFlow gradient $\frac{dR}{dw}$ for weight w is the sum of the products $\prod_\ell |w_p^{(\ell)}|$ of the magnitudes on all paths containing w. This is the weight's contribution to the network's ($\ell_{1,1}$) *path norm* (Neyshabur et al., 2015). SynFlow prunes iteratively, removing a small portion of weights, recalculating these values, and repeating. Once an outgoing (or incoming) weight is pruned from a neuron, all incoming (or outgoing) weights are in fewer paths, decreasing $\frac{dR}{dw}$; they are more likely to be pruned on the next iteration, potentially creating a vicious cycle that prunes the entire neuron. Similarly, SynFlow heavily prunes skip connection weights, which participate in fewer paths.

In fact, randomly shuffling the weights to mitigate this neuron collapse improves the performance of SynFlow to the point where it overtakes the other methods (Figure 5-1 right). Between 73.8% and 95.6% sparsity, randomly shuffled SynFlow outperforms magnitude pruning (the leading method in this sparsity range) by 0.2 to 0.4 percentage points. Beyond these sparsities, it matches the performance of GraSP (the leading method in this sparsity range).

5.7.2 Reinitialization

I next consider whether the networks produced by these methods are sensitive to the specific initial values of their weights. That is, is performance maintained when sampling a new initialization for the pruned network from the same distribution as the original network? Magnitude pruning after training and LTR (shorthand for IMP-with-rewinding) are known to be sensitive this ablation: when reinitialized, pruned networks train to lower accuracy (Figure 5-5 bottom row; Appendix 5.11; Han et al., 2015), and there is no known state-of-the-art weight-pruning method with accuracy robust to this ablation. However, all early pruning techniques are unaffected by reinitialization (green line): accuracy is the same whether the network is trained with the original initialization or a newly sampled initialization. As with random shuffling, this raises the question of whether this insensitivity to initialization may pose a limit to these methods that restricts performance.

5.7.3 Inversion.

SNIP, GraSP, and SynFlow are each based on a hypothesis about properties of the network or training that allow a sparse network to reach high accuracy. Scoring functions should rank weights from most important to least important according to these hypotheses, making it possible to preserve the most important weights when pruning to any sparsity. In this ablation, I assess whether the scoring functions successfully do so: I prune the *most* important weights and retain the *least* important weights. If the hypotheses behind these methods are correct and they are accurately instantiated as scoring functions, then *inverting* in this manner should lower performance.

Magnitude pruning at initialization, SNIP, and SynFlow behave as expected: when pruning the most important weights, accuracy decreases (red line). In contrast, GraSP's accuracy does not change when pruning the most important weights. This result calls into question the premise behind GraSP's heuristic: one can keep the weights that, according to Wang et al. (2020), *decrease* gradient flow the most and get the same accuracy as keeping those that purportedly increase it the most. Moreover, I find that pruning weights with the lowest-magnitude GraSP scores improves accuracy (Section 5.8.2).

5.7.4 Summary

When using the methods for pruning at initialization, it is possible to reinitialize or layerwise shuffle the unpruned weights without hurting accuracy. This suggests that the useful part of the pruning decisions of SNIP, GraSP, SynFlow, and magnitude at initialization are the layerwise proportions rather than the specific weights or values. In the broader pruning literature and in the preceding chapters, shuffling and reinitialization are associated with lower accuracy, and—as an example—Figure 5-5 shows that this is the case for magnitude pruning after training in otherwise identical settings. Since these ablations do not hurt the accuracy of the methods for pruning at initialization, this raises the question of whether these methods may be confined to this lower stratum of performance.

This behavior under the ablations also raises questions about the reasons why SNIP, GraSP, and SynFlow are able to reach the accuracies in Section 5.6. Each paper poses a hypothesis about the qualities that would allow a pruned network to train effectively and derives a heuristic to determine which specific weights should be pruned based on this hypothesis. In light of these ablation results, however, it is unclear whether the performance achieved by these methods is attributable to these hypotheses. For example, SNIP aims to “identify important connections” (Lee et al., 2019); however, accuracy does not change if the pruned weights are randomly shuffled. SynFlow focuses on paths, “taking the inter-layer interactions of parameters into account” (Tanaka et al., 2020); accuracy improves in some cases when discarding path information by shuffling and is maintained if altering the “synaptic strengths flowing through each parameter” by reinitializing. In addition to these concerns, GraSP performs identically when inverted. My overall recommendation is that future early pruning research should use these and other ablations to evaluate whether the proposed heuristics behave according to the claimed justifications.

5.8 Variants of Pruning Methods

Since the release of SNIP, GraSP, and SynFlow, other variations of these methods have been proposed. In this section, I consider two of them.

5.8.1 Iterative SNIP

In Section 5.7, I consider SNIP as it was originally proposed by Lee et al. (2019), pruning weights from the network in one shot. Recently, however, de Jorge et al. (2020) and Verdenius et al. (2020) have proposed variants of SNIP in which pruning occurs iteratively (like SynFlow). In iterative variants of SNIP, the SNIP scores are calculated, some number of weights are pruned from the network, and the process repeats multiple times with new SNIP scores calculated based on the pruned network. Here, I compare one-shot SNIP (the same experiment as before) and iterative SNIP (in which I prune iteratively over 100 iterations, with each iteration going from sparsity

$s^{\frac{n-1}{N}}$ to sparsity $s^{\frac{n}{N}}$—the same strategy as I use for SynFlow).

Figure 5-7 below shows the performance of SNIP (red) and iterative SNIP (green) at the sparsities I consider. At these sparsities, making SNIP iterative does not meaningfully alter performance. It is possible that making SNIP iterative matters most at the especially extreme sparsities studied by de Jorge et al. (2020) and Tanaka et al. (2020) rather than at the matching and relatively less extreme sparsities I focus on in this section.

Figure 5-8 shows the shuffling and reinitialization ablations for SNIP (top) and iterative SNIP (bottom). The ablations do not meaningfully affect accuracy.

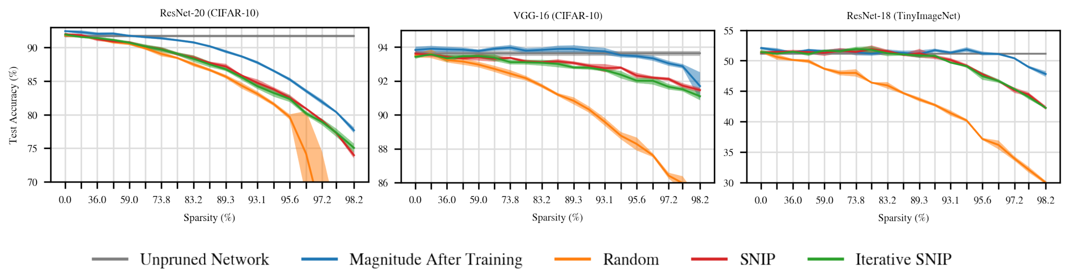


Figure 5-7: A comparison of SNIP and iterative SNIP (100 iterations).

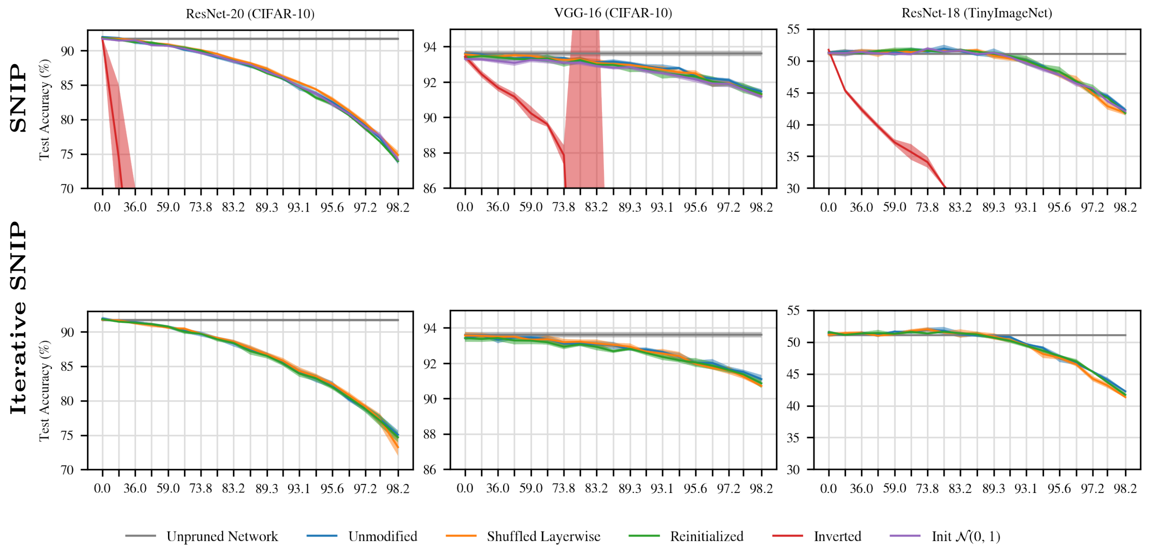


Figure 5-8: A comparison of the ablations for SNIP (from Section 5.7 Figure 5-5) and for iterative SNIP. (ResNet-50 is not included due to computational limitations.)

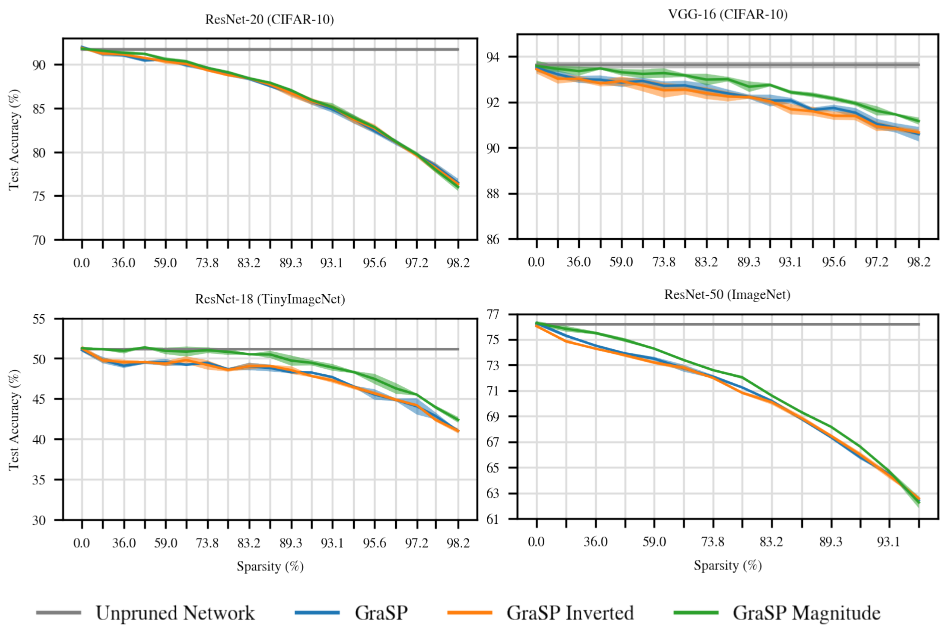


Figure 5-9: Accuracy of three different variants of GraSP

5.8.2 Variants of GraSP

In Figure 5-9, I show three variants of GraSP: pruning weights with the highest scores (the version of GraSP from Wang et al.), pruning weights with the lowest scores (the inversion experiment from Section 5.7), and pruning weights with the lowest magnitude GraSP scores (my proposal for an improvement to GraSP as shown in Figure 5-10). This figure is intended to make the comparison between these variants clearer; Figure 5-5 is too crowded for these distinctions to be easily visible.

The main takeaway from this experiment is that GraSP appears to perform tangibly better on three of the four benchmarks when pruning the connections with the lowest GraSP score *magnitudes* rather than values as proposed by Wang et al. (2020).

5.8.3 Comparisons to Improved GraSP and SynFlow

In Figure 5-10 below, I compare the early pruning methods with my improvements to GraSP and SynFlow. This figure is identical to Figure 5-4, except that I modify GraSP to prune the weights with the lowest-magnitude GraSP scores (Section 5.8.2) and I modify SynFlow to randomly shuffle the per-layer pruning masks after pruning

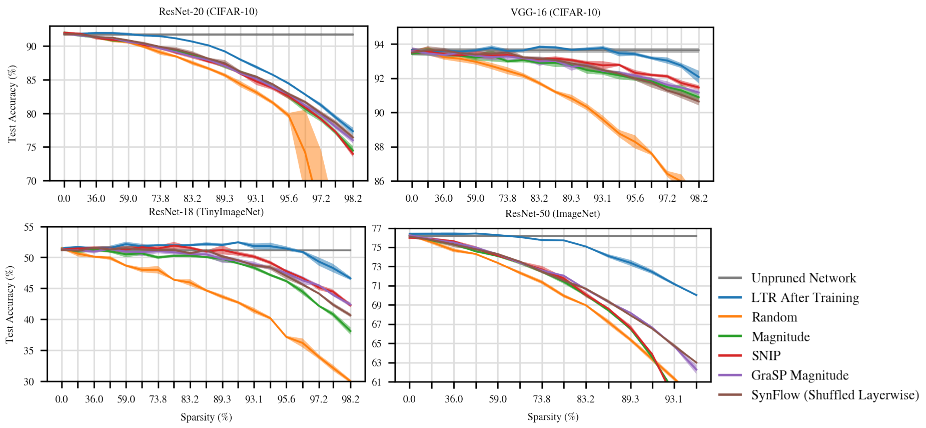


Figure 5-10: The best variants of pruning methods at initialization from my experiments. GraSP and SynFlow have been modified as described in Section 5.7 and Section 5.8.2.

(Section 5.7). The overall findings remain similar (none of the methods are comparable to magnitude pruning after training), but modified SynFlow and GraSP are more competitive and perform especially well at extreme sparsities on ResNet-50.

5.9 Context: Layerwise Pruning Proportions

In Figure 5-11, I plot the per-layer sparsities produced by the unmodified, unablated versions of each pruning method at the highest matching sparsity. Each sparsity is labeled with the corresponding layer name; layers are ordered from input (left) to output (right) with residual shortcut/downsample connections placed after the corresponding block. I make the following observations.

Different layerwise proportions lead to similar accuracy. At the most extreme matching sparsity, the early pruning methods perform in a relatively similar fashion: there is a gap of less than 1, 1.5, 2.5, and 1 percentage point between the worst and best performing early pruning methods on ResNet-20, VGG-16, ResNet-18, and ResNet-50. However, the layerwise proportions are quite different between the methods. For example, on ResNet-20, SynFlow prunes the early layers to less than 30% sparsity, while GraSP prunes to more than 60% sparsity. SNIP and SynFlow tend to prune later layers in the network more heavily than earlier layers, while GraSP

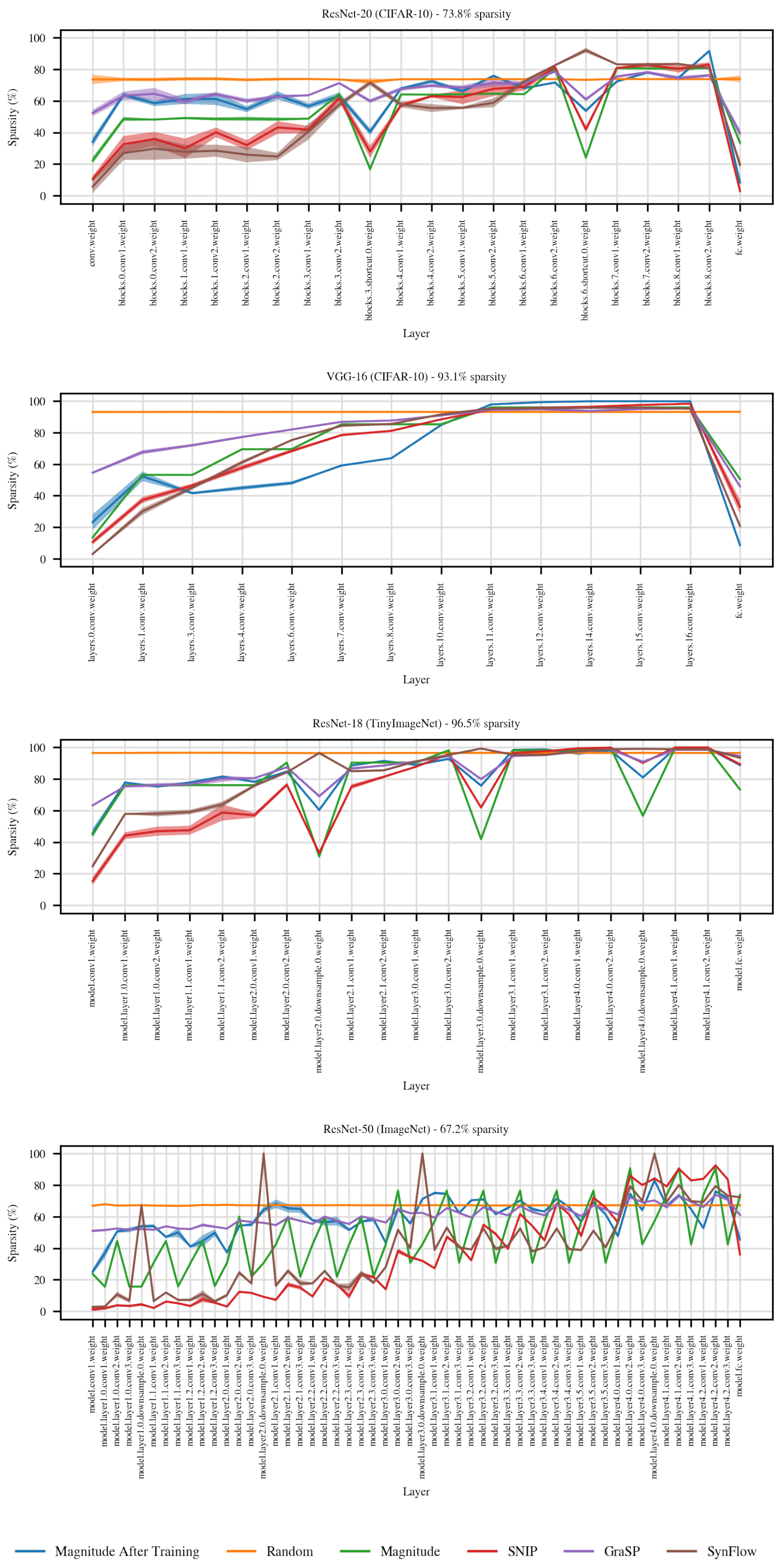


Figure 5-11: Per-layer sparsities produced by each pruning method at the highest matching sparsity.

tends to prune more evenly.

On the ResNets, the GraSP layerwise proportions most closely resemble those of magnitude pruning after training, despite the fact that GraSP is not the best-performing method at the highest matching sparsity on any network.

These results are further evidence that determining the layerwise proportions in which to prune (rather than the specific weights to prune) may not be sufficient to reach the higher performance of the benchmark methods. The early pruning methods produce a diverse array of different layerwise proportions, yet performance is universally limited.

Skip connections. When downsampling the activation maps, the ResNets use 1x1 convolutions on their skip connections. On ResNet-20, these layer names include the word `shortcut`; on ResNet-18 and ResNet-50, they include the word `downsample`. SynFlow prunes these connections more heavily than other parts of the network; in contrast, all of the other methods prune these layers to similar (ResNet-50) or much lower (ResNet-20 and ResNet-18) sparsities than adjacent layers. On ResNet-50, SynFlow entirely prunes three of the four downsample layers, eliminating the residual part of the ResNet for the corresponding blocks.

The output layer. All pruning methods (except random pruning) prune the output layer at a lower rate than the other layers. These weights are likely disproportionately important to reaching high accuracy since there are so few connections and they directly control the network outputs.

5.10 Pruning After Initialization

In Section 5.6, I showed that pruning at initialization using any of the existing proposals leads to lower accuracy than magnitude pruning after training. In Section 5.7, I showed that this accuracy is invariant to ablations that hurt the accuracy of magnitude pruning after training. In this section, I seek to distinguish whether these behaviors are (1) intrinsic to the pruning methods or (2) specific to using the

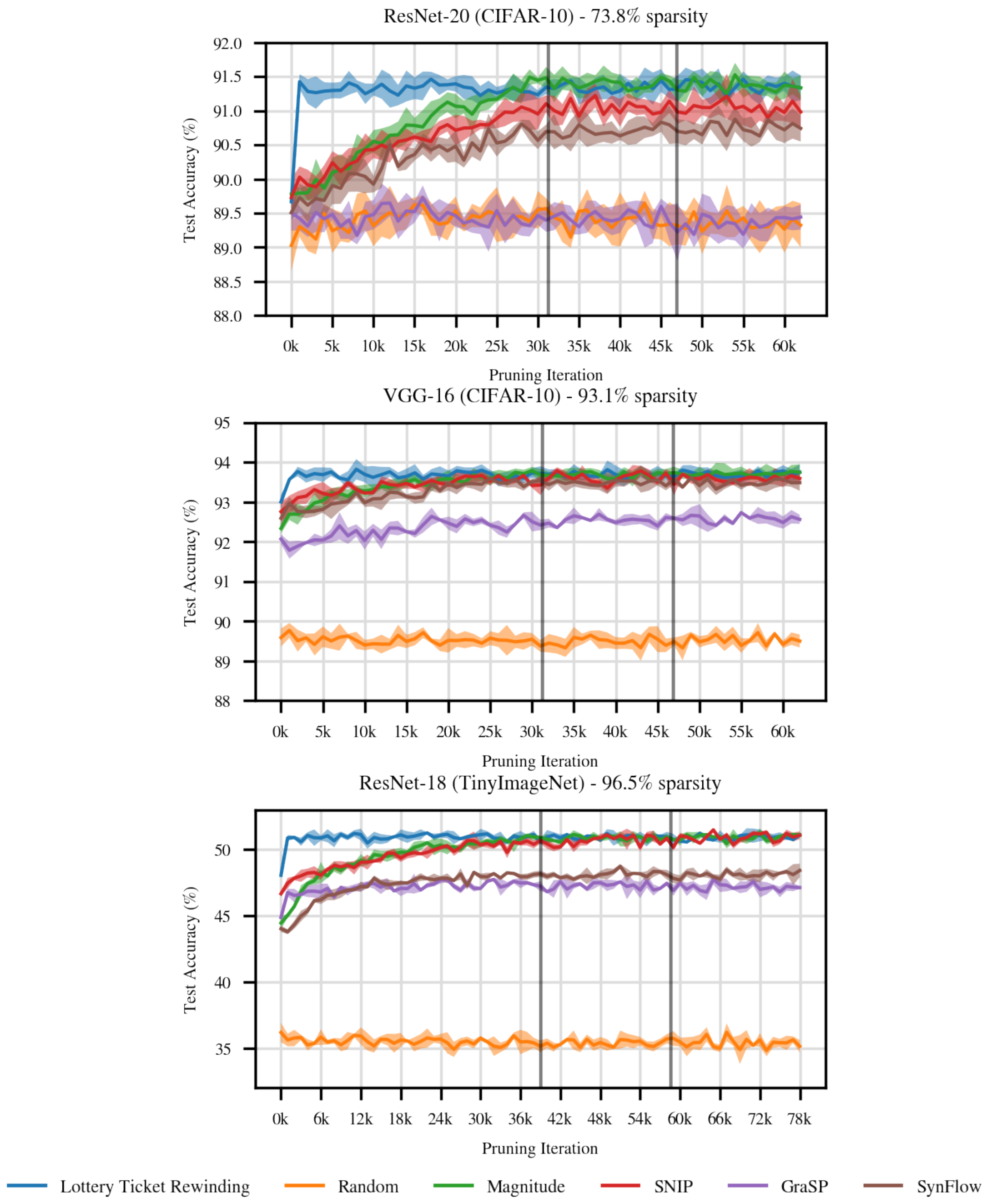


Figure 5-12: Accuracy of early pruning methods when pruning at the iteration on the x-axis. These experiments involve pruning at one fixed sparsity for each network: namely, the highest matching sparsities. For computational reasons, I had to focus on one sparsity, and this is the most difficult sparsity at which any method might be expected to produce subnetworks that reach full accuracy. The inclusion of LTR (IMP with reminding) prunes after training and initializes to the weights from the specified iteration. Vertical lines are iterations where the learning rate drops by 10x.

pruning methods at initialization. I have already shown evidence in support of (2) for magnitude pruning after training: the subnetworks it finds are less accurate and maintain their accuracy under shuffling and reinitialization at initialization but not when pruning after training (Figure 5-5 in Section 5.7). Moreover, as the preceding chapters have shown, LTR performs best when pruning early in training rather than at initialization, potentially pointing to broader difficulties specific to pruning at initialization.

To eliminate possibility (1), one would need to demonstrate that there are circumstances where these methods reach higher accuracy than they do when pruning at initialization and where they are sensitive to the ablations. I do so by using these methods to prune later in training: I train for k iterations, prune using each technique, and then train further for the entire learning rate schedule (Renda et al., 2020).[13] If accuracy improves when pruning at iteration $k > 0$ and the methods become sensitive to the ablations, then it would show that the behaviors observed in Sections 5.6 and 5.7 are not intrinsic to the methods. Note that SNIP, GraSP, and SynFlow were not designed to prune after initialization; as such, I focus on whether accuracy improves rather than the specific accuracy itself.

Figure 5-12 shows the effect of this experiment on accuracy. I include random pruning as a lower baseline that every method should match or surpass. I also include LTR as an upper baseline: is the best accuracy known to be possible when applying a pruning mask early in training, even if that mask was found using an expensive procedure that required a substantial amount of additional training.

Random pruning also provides a control to show that pruning after the network has trained for longer (and has reached higher accuracy) does not necessarily cause the pruned network to reach higher accuracy; no matter when in training I randomly prune, accuracy is the same. This means that we should not assume that pruning later in training to lead to better accuracy simply because the network has learned more. If a method reaches improved accuracy when applied later in training, that

[13]Due to the expense of this experiment, I examine a single sparsity: the most extreme matching sparsity, which is the most difficult setting in which any method for pruning early in training might be expected to produce subnetworks that complete training and reach full accuracy.

means that the method extracted useful information from the network.

Magnitude pruning, SNIP, and SynFlow improve when applied later in training, with magnitude pruning and SNIP approaching the performance of LTR, and SynFlow is not far behind LTR. This means that the performance gap in Section 5.6 is not intrinsic to the heuristics these methods employ, but rather it is due to using them at initialization.

Interestingly, LTR reaches higher accuracy than the pruning methods early in training: magnitude pruning does not match the accuracy of LTR at iterations 1K, 2K, and 1K until iterations 25K, 26K, and 36K on ResNet-20, VGG-16, and ResNet-18. This gap in accuracy is the known opportunity still remaining to develop better methods for pruning early in training. All of the methods considered here understandably underperform LTR; after all, they were designed with a different purpose in mind (either pruning at initialization or pruning after training). We may need to develop entirely new methods to close this gap. Alternatively, if this gap in accuracy reflects a broader challenge inherent to pruning early in training, then these results suggest it may be difficult to prune, not just at initialization, but after as well.

5.11 Ablations After Initialization

To further understand the behavior of pruning after initialization, I perform the ablations from Section 5.7 (shuffling and reinitialization) on the pruning methods when applied *after* initialization. My goal is a continuation of the experiment in Section 5.10: to study whether the pruning methods become more sensitive to the ablations as pruning occurs later in training.

5.11.1 Ablations at the End of Training

In Figure 5-13, I perform the ablations from Section 5.7 when pruning at the end of training. This figure parallels Figure 5-5, where I performed these ablations when pruning at initialization.

Magnitude. The results in Figure 5-13 confirm the well-known result that shuffling or reinitializing subnetworks found via magnitude pruning after training causes accuracy to drop (Han et al., 2015, previous chapters of this thesis). On ResNet-20, shuffling and reinitializing perform similarly; they match full accuracy until 49% sparsity vs. 73.8% for the unmodified network. On VGG-16 and ResNet-18, random pruning is better initially, and reinitializing is better at higher sparsities.

SNIP. On SNIP, there are smaller differences in accuracy between the unmodified network and the ablations. On ResNet-20, the unmodified network performs slightly better. On VGG-16, it performs approximately as well as when randomly shuffled. On ResNet-18, there are more substantial differences. These results indicate that SNIP is not inherently robust to these ablations, but rather that the point in training at which SNIP is applied plays a role. Note that, at the highest sparsities on ResNet-20 and VGG-16, SNIP did not consistently converge, leading to the large error bars.

GraSP. On GraSP, the ablations have a limited effect on the performance of the network, similar to pruning at initialization. On ResNet-20, the unmodified networks and the ablations perform the same. On VGG-16, the shuffling ablation actually outperforms the unmodified network (which performs the same as when reinitialized) at lower sparsities. On ResNet-18, all three experiments perform similarly at lower sparsities, and shuffling performs lower at extreme sparsities.

SynFlow. On SynFlow, the ablations do affect performance, but in different ways on different networks. On ResNet-20, shuffling improves accuracy; on VGG-16 and ResNet-18 it decreases accuracy at higher sparsities.

5.11.2 Ablations After Initialization

In Figure 5-14, I perform the ablations at all points in training at a single sparsity for each network. This figure shows the ablations corresponding to Figure 5-12 in Section 5.10.

Magnitude. Magnitude pruning becomes sensitive to the ablations early in training (after iteration 5,000 for ResNet-20, 10,000 for VGG-16, and 4,000 for ResNet-18). The performance of the ablations does not improve after the very earliest part of training; the performance without the ablations continues improving after this point, while the performance of the ablations remains the same after this point. On VGG-16, shuffling outperforms reinitialiation, suggesting the initialization matters more tan the structure of the sparse network. On ResNet-18, the reverse is true.

SNIP. The behavior of SNIP is very similar to the behavior of magnitude pruning.

GraSP. GraSP improves little when pruning after initialization. It only sensitive to the shuffling ablation on VGG-16 and ResNet-18, and the changes in accuracy under this ablation are small.

SynFlow. SynFlow is sensitive to the ablations, but in different ways on different networks. On ResNet-20 and VGG-16, the accuracy under shuffling improves alongside the accuracy of unmodified SynFlow. On ResNet-18, the accuracy under shuffling is lower and does not improve. In all cases, accuracy is lower under reinitialization.

5.11.3 Summary

Overall, these results support the hypothesis that it may be especially difficult to prune in a connection-specific or initialization-sensitive manner at initialization. At initialization, magnitude pruning, SNIP, and SynFlow maintain or improve upon their accuracy under random shuffling and reinitialization. After initialization, shuffling and reinitialization hurt accuracy to varying degrees.

5.12 Discussion

The state of the art in pruning at initialization. This chapter establishes the following findings about the current state-of-the-art in pruning at initialization.

Surpassing random pruning. All methods surpass random pruning at some or all matching sparsities, and, in certain settings, some methods maintain full accuracy at non-trivial sparsities.

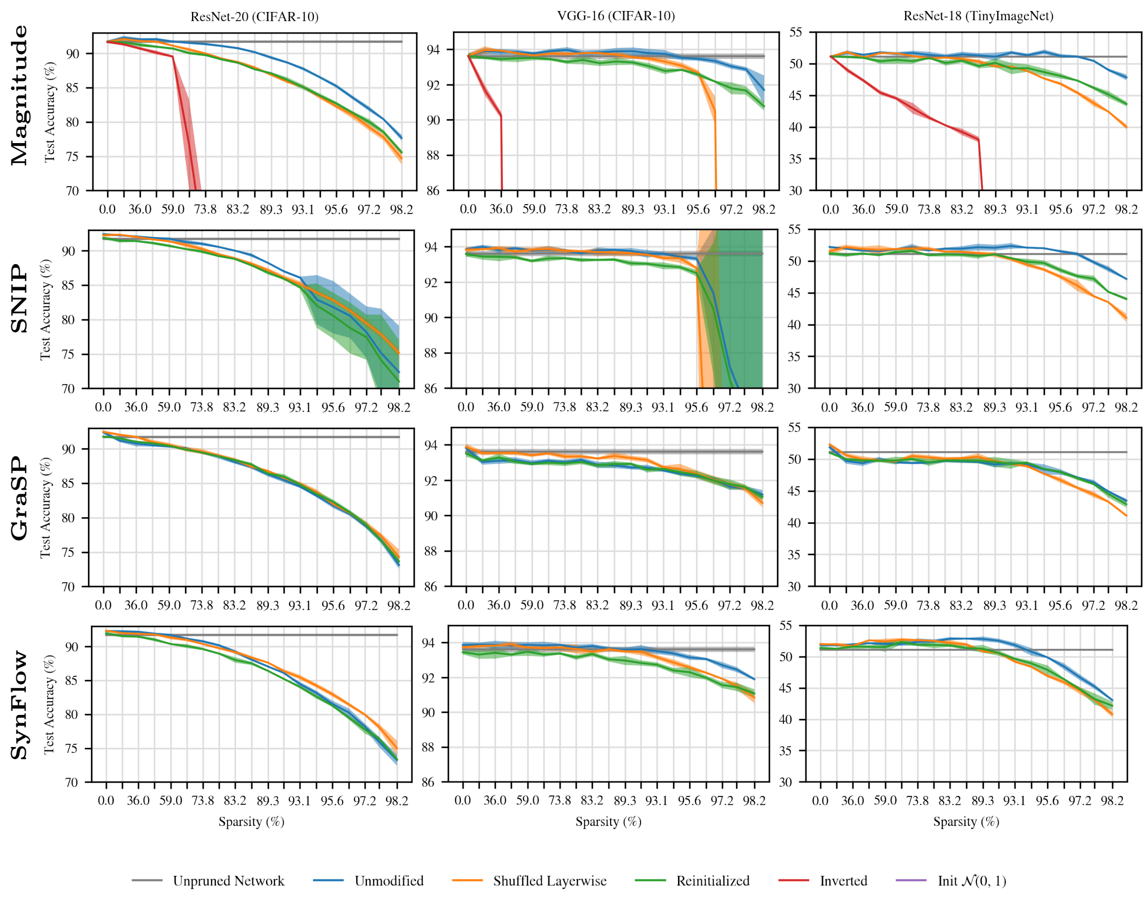


Figure 5-13: Ablations on subnetworks found by applying magnitude, SNIP, GraSP, and SynFlow after training. (I did not run this experiment on ResNet-50 due to resource limitations.)

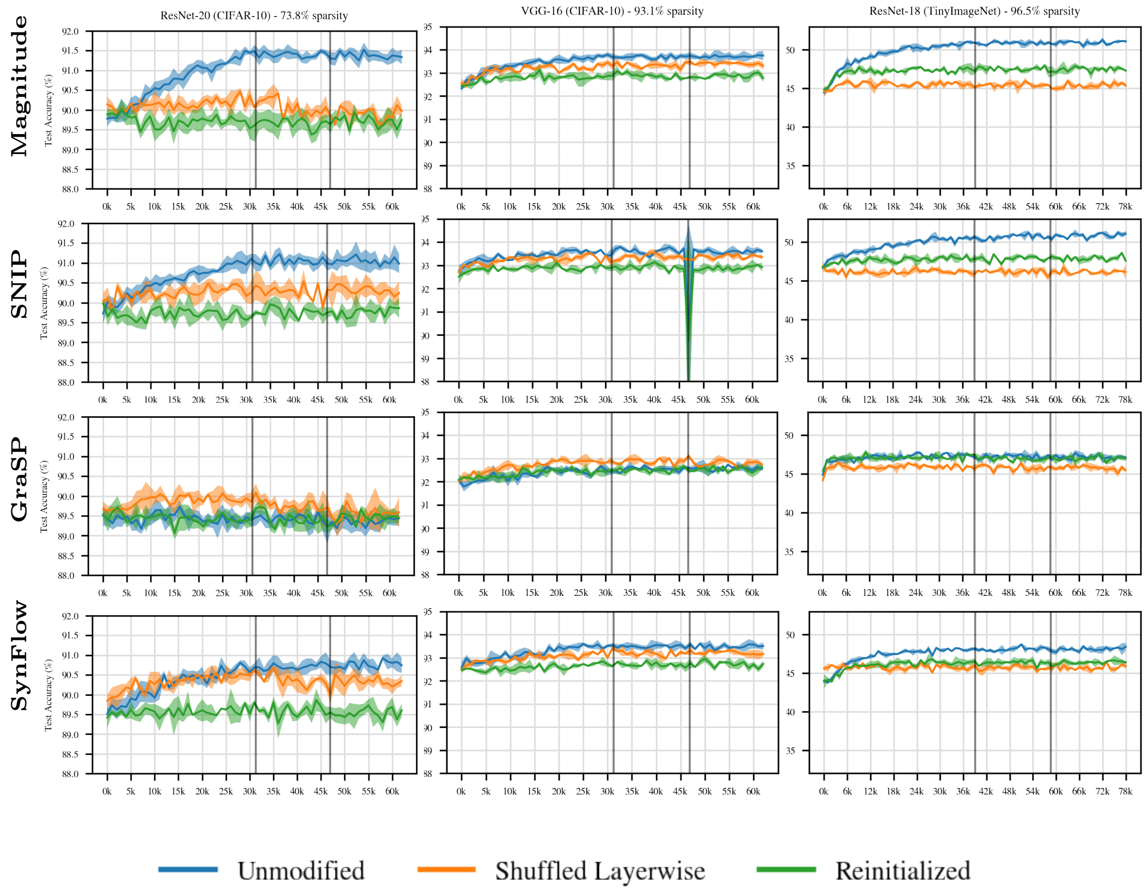


Figure 5-14: Accuracy of early pruning methods and ablations when pruning at the iteration on the x-axis. Sparsities are the highest matching sparsities. Vertical lines are iterations where the learning rate drops by 10x.

No single method is dominant at initialization. Depending on the network, dataset, and sparsity, there is a setting where each early pruning method reaches the highest accuracy. Enhancements to GraSP (Figure 5-9 in Appendix 5.8.2) and SynFlow (Figure 5-5) further tighten the competition. For matching sparsities, the best and worst methods differ by no more than 0.3, 0.9, and 2.1 percentage points for ResNet-20, VGG-16, and ResNet-18. In comparison, the best method differs from pruning after training by at most 1.5, 0.7, and 4.5 percentage points and the worst method differs from random pruning by at most 0.4, 3.0, and 8.2 percentage points. SNIP consistently performs best or among the best, despite the fact that multiple papers have built on it since its publication. Magnitude pruning performs surprisingly well at initialization, especially on ResNet-20 and ResNet-50.

Data is not currently essential at initialization. SynFlow and magnitude pruning are competitive at initialization without using any training data. Like robustness to shuffling and reinitialization, however, data-independence may only be possible for the limited performance of current methods. In contrast, magnitude pruning after training and LTR rely on data for both pruning and initialization insofar as they determine which connections to prune using a trained network and apply those pruning decisions to a network that has already undergone some amount of training.

Below the performance of pruning after training. All methods for pruning at initialization reach lower accuracy than magnitude pruning after training. Moreover, note that I have only compared against one-shot magnitude pruning. Iterative pruning methods can match full accuracy at higher sparsities, presenting an even more challenging comparison.

The challenge of pruning at initialization. It is striking that methods that use such different signals (magnitudes; gradients; Hessian; data or lack thereof) reach similar accuracy, behave similarly under ablations, and improve similarly (except GraSP) when pruning after initialization.

Ablations. None of the methods I have examined are sensitive to the specific weights that are pruned or their specific values when pruning at initialization. When pruning after training, these ablations are associated with lower accuracy. This contrast raises the question of whether methods that maintain accuracy under these ablations are inherently limited in the accuracy they can reach, and whether this is a property of these methods or of pruning at initialization in general. It may yet be possible to design a method that reaches higher accuracy and is sensitive to these ablations at initialization, but it may also be that the failures of four methods to do so (and their improvements after initialization in Section 5.10 and Appendix 5.11) point to broader challenges in pruning at initialization.

Why is it challenging? I do not identify a cause for why these methods struggle to prune in a specific fashion at initialization (and often only at initialization), and I believe that this is an important question for future work. Perhaps there are properties of optimization that make pruning specific weights difficult or impossible at initialization (Evci et al., 2019). For example, training occurs in multiple phases (Gur-Ari et al., 2018; Jastrzebski et al., 2020); perhaps it is challenging to prune during this initial phase. Further study of this question may reveal whether this challenge is related to shared characteristics of these methods or whether it is intrinsic to pruning at initialization.

Looking ahead. I will close this section by discussing the implications of these findings for future pruning research.

Why prune early in training? There are many reasons to do so. This includes the scientific goals of studying the capacity needed for learning (i.e., the main topic of this thesis) and the information needed to prune (Tanaka et al., 2020) and the practical goals of reducing the cost of finding pruned networks for inference (Lee et al., 2019) and "saving resources at training time" (Wang et al., 2020).

In this section, I have focused on the practical goal of reducing the cost of training, since the results show that pruning at initialization is not a drop-in replacement for pruning after training. I found that existing methods for pruning at initialization require making a tradeoff: reducing the cost of training entails sacrificing some amount of accuracy. Looking ahead, the most compelling next step for pruning research is developing tradeoff-free ways to reduce the cost of training.

Pruning after initialization. In Section 5.10, SNIP, SynFlow, and magnitude improve gradually after initialization but LTR improves much faster. However, these methods were designed for initialization; focusing early in training may require new approaches. Alternatively, it may be that, even at iterations where LTR succeeds in Section 5.10, the readily available information is not sufficient reach this performance without consulting the state of the network after training. One way to avoid this challenge could be to dynamically change the mask to exploit signals from later in training (Mocanu et al., 2018; Dettmers & Zettlemoyer, 2019; Evci et al., 2020).

New signals for pruning. It may be possible to prune at initialization or early in training, but signals like magnitudes and gradients (which suffice late in training) may not be effective. Are there different signals one should use early in training? Is it reasonable to expect signals that work early in training to work late in training (or vice versa)? For example, second order information should behave differently at initialization and convergence, which may explain why GraSP struggles later.

Measuring progress. We typically evaluate pruning methods by comparing their accuracies at certain sparsities. In the future, we will need to extend this framework to account for tradeoffs in different parts of the design space. At initialization, we must weigh the benefits of extreme sparsities against decreases in accuracy. This is especially important for methods like SynFlow and FORCE (de Jorge et al., 2020), which are designed to have lower but non-random accuracy at the most extreme sparsities. In this chapter, I deferred this tradeoff by focusing on matching sparsities.

When pruning after initialization, we will need to address an additional challenge: comparing a method that prunes to sparsity s at step t against a method that prunes to sparsity $s' > s$ at step $t' > t$. To do so, we will need to measure overall training cost. That might include measuring the area under the curve in Figure 5-1, FLOPs (as, e.g., Evci et al. (2019) do), or real-world training time and energy consumption on software and hardware optimized for pruned neural networks. Mere accuracy/sparsity pareto frontiers will no longer be capable of telling the full story.

Chapter 6

Conclusions, Influence, and Implications

6.1 Conclusions

As stated in the introduction, the central research question that I addressed in this thesis was as follows:

Under what circumstances would it possible to train a sparse neural network (like that uncovered by pruning after training) in place of a standard dense network?

In Chapter 3, I showed that, in smaller-scale settings, sparse, trainable subnetworks exist at initialization. These subnetworks are capable of training in isolation from initialization to the same accuracy as the unpruned network in at most the same number of steps. In many cases, these subnetworks actually reached this accuracy in fewer steps than the original network. They could do so at sparsities similar to those attainable when pruning after training the full network.

In Chapter 4, I scaled up these results. The technique I used to find these subnetworks, iterative magnitude pruning (IMP), did not find winning tickets at initialization in larger-scale and more challenging settings. Instead, I had to modify the procedure, pruning early in training (1%-5% of the way in) rather than at initialization. This made it possible to find sparse subnetworks capable of completing training in isolation and reaching the same accuracy as the original network in at most the same number of steps. In doing so, it still preserved nearly all of the opportunity to improve the efficiency of neural network training. Underlying this change in behavior between pruning at initialization and pruning early in training were my observations about linear mode connectivity: IMP subnetworks that consistently train to the same convex region of the loss landscape (regardless of the sample of SGD noise) also reach higher accuracy.

Finally, in Chapter 5, I evaluated and extended the state of the art in practical methods for efficiently pruning neural networks at initialization. I found that simply pruning the lowest magnitude weights at initialization performs as well as several other proposed methods. Although this procedure does not yield winning tickets or matching subnetworks at sparsities as high as those at which IMP can find them, it does represent progress toward the eventual goal of making the lottery ticket results practical.

In conclusion, it is possible to train a sparse neural network (like that uncovered by pruning after training) in place of a standard dense network in nearly all circumstances. There are caveats (e.g., in larger scale settings, we may only be able to do so from early in training, and we do not know how to find these subnetworks efficiently), but the results in this thesis have established that training sparse networks in this way is possible and have opened the door to the exciting new research problem of practical sparse training. Concurrently, they have offered a variety of new scientific insights into the nature of neural network training.

6.2 Influence

The line of work discussed in this thesis has interacted with and energized the literatures on several related topics. In this section, I summarize several of those literatures, describe important research in those areas, and provide commentary.

6.2.1 Practical Algorithms for Pruning at Initialization

Concurrently to and in the time since the original lottery ticket paper was published, a number of strategies have been proposed for pruning neural networks at initialization and finding performant subnetworks (possibly including winning tickets). This topic is discussed in-depth in Chapter 5, where I evaluate three prominent algorithms, compare them to baselines, and ablate the importance of various aspects of the decisions they make about which weights to prune. At the time of writing, there is no known efficient (cost $< 10\%$ of the overall cost of training) strategy for finding winning tickets at initialization or matching subnetworks early in training at sparsities that are comparable to those in this thesis. Chapter 5 discusses the key challenges to doing so. For further details on this literature, see Chapter 5.

6.2.2 Dynamic Sparse Training

Summary of the literature. Algorithms for pruning at initialization have not yet been effective at realizing the opportunity to improve the efficiency of training through pruning. However, the approaches discussed so far represent only one part of the design space for training sparse networks. In particular, they focus on *static* sparsity: that, once the network is pruned, the sparsity pattern remains fixed from that point forward.

In parallel, another literature has emerged that takes on the problem of sparse training in a different way. Research on *dynamic sparsity* explores the possibility that, by changing the sparsity pattern throughout training, it may be possible to train a network to full accuracy that is sparse from beginning to end, perhaps in the same number of steps as the unpruned network. Doing so would realize the same efficiency improvements as training using a winning ticket, assuming the overhead of changing the sparsity pattern is minimal.

Dating back to 2018, several algorithms have been proposed for dynamic sparse training (Mocanu et al., 2018; Bellec et al., 2018; Mostafa & Wang, 2019; Dettmers & Zettlemoyer, 2019; Evci et al., 2020). This literature predates my research on the lottery ticket hypothesis. However, the two literatures subsequently began to interact, and the most successful work on the subject by Evci et al. (2020) entitles the paper *Rigging the Lottery* and refers to the central algorithm as *RigL*.

RigL involves repeatedly training the network, removing the lowest magnitude weights, and regrowing new weights according to gradients on the latest minibatch. The layerwise sparsities are set according to a modification of the Erdos-Renyi formulation of the expected connectivity in random graphs; this modification is to make it more suitable for convolutions. The main hyperparameters to set in RigL are the fraction of weights to prune and regrow on each pruning step and the frequency of the pruning steps. On ResNet-50 for ImageNet, Evci et al. change the sparsity pattern every 100 steps for the 32,000 steps of training, and they gradually decrease the fraction of weights changed such that the sparsity pattern converges late in training.

Commentary. The headline result of Evci et al. is that RigL can indeed train networks that are sparse from the start (at 90% sparsity for ResNet-50 on ImageNet) and nearly reach commensurate accuracy to the unpruned network. However, to do so, the network must be trained for 5x as many steps as usual, reducing the overal FLOP-count by only half. On a per-FLOP basis, this outperforms the methods discussed in Chapter 5, but it does not fulfill the criterion of a winning ticket that it must train in the same number of steps as the original network. In other words, it does not present a trade-off free scenario in which the sparse training strategy is a same-cost drop-in replacement for the training the unpruned network even in the absence of any way to exploit sparsity for speedup.

These results are still promising, and—in my view—they suggest that there are further opportunities to develop better methods that might allow for sparse-from-the-start training to full accuracy in the same number of steps as the original network—i.e., something equivalent to possessing a winning ticket. (Utku Evci has expressed more pessimism in personal correspondence; he conjectured that sparse networks may intrinsically need more steps in order to learn.) The recipe of Evci et al. is purposefully simplistic, and more fine-grained exploration of the design space could yield further improvements.

In my view, one underexplored area of dynamic sparsity is research on how effective we might hope it to be. The lottery ticket work gave us a belief of how well we might expect any pruning strategy to perform at initialization or early in training. I think it is important that we do something similar for dynamic sparsity. Can we produce an existential result at any cost that provides information on the fewest number of sparsity pattern changes that might be necessary and the least number of weights that might need to be swapped when the sparsity pattern changes? Can we learn anything about how those new weights should be reinitialized?

Algorithm 5 One-shot Dynamic IMP with sparsity s and iterations $n_1, n_2, \ldots, n_k$ at which the sparsity pattern is modified.

```
1: Create a network with randomly initialization $W_0 \in \mathbb{R}^d$.
2: Train $W_0$ to $W_{n_1}$: $W_{n_1} = \mathcal{A}^{0 \to k}(W_0)$
3: Let $n_0 = 0$.
4: for $i, n \in \{(1, n_1), \ldots, (k, n_k), (k+1, N)\}$ do
5:     Initialize pruning mask to $m_i = 1^d$.
6:     Prune the lowest magnitude entries of $W_n$ that remain.
       Let $m_i[j] = 0$ if $W_n[j]$ is pruned.
7:     Train $m_i \odot W_{n_{i-1}}$ to $m_i \odot W'_{n_i}$: $W'_{n_i} = \mathcal{A}^{n_{i-1} \to n_i}(m_i \odot W_{n_{i-1}})$.
8:     If $i \leq k$: grow new weights. Let $W'_i[j] = V[j]$ if $m_i[j] = 0$,
       where $V$ is a new random initialization.
```

I have conducted some preliminary experiments in this direction. Although they are too early to include as a results section in this thesis, they do shed some intriguing light on these least upper bounds. I use the procedure described in Algorithm 5. Written as a formal algorithm, the procedure looks quite complicated. A simplified version with one change to the sparsity pattern and one-shot pruning works as follows:

1. Create a randomly initialized network W_0.
2. The first phase of dynamic sparsity.
    (a) Train the network W_0 to step k, producing weights W_k.
    (b) Prune the lowest magnitude weights to the desired sparsity in one-shot, creating a mask m_1.

   (c) Go back to the beginning, this time training the sparse network $m_1 \odot W_0$ to step k, producing weights W'_k.

3. Grow new weights in every location where a weight was previously pruned. That is, wherever $m_1 = 0$, sample a new value from the initialization distribution and set the corresponding weight of W'_k to that value. Call this new unpruned network (a hybrid between the sparse network W'_k and a randomly initialized network) W''_k.

4. The second phase of dynamic sparsity.

   (a) Train the network W''_k from step k to the end of training, producing weights W_N.

   (b) Prune the lowest magnitude weights to the desired sparsity in one-shot, creating a mask m_2.

   (c) Go back to step k, this time training the sparse network $m_2 \odot W''_k$ to the end of training, producing weights W'_N.

This procedure performs one-shot IMP twice, first when training from iteration 0 to iteration k and again when training from iteration k to the end of training. In between, weights are regrown wherever they were pruned during the first phase of IMP. Importantly, every pruned weight is regrown, meaning the second phase of IMP begins with a dense network that is a hybrid of the sparse network from the first phase and new weights sampled from a random initialization. This simulates regrowing all weights, subsequently letting the second phase of IMP select which of these regrown weights should be kept.

The end result is two pruning masks, m_1 and m_2, with the sparsity patterns for each phase of dynamic pruning, and a set of connections $g = ((\sim m_1) \& m_2) \odot W'_k$ that are regrown along with their initializations. The exact number of weights to be regrown is determined by the second phase of IMP. It is possible that m_1 and m_2 overlap significantly or not at all.

I have run this experiment on ResNet-20 with CIFAR-10 for several values of k toward the middle of training. Preliminary results include:

- One change in sparsity pattern is sufficient to produce at set of masks m_1 and m_2 and a set of regrown weights g such that it is possible to train sparse-from-the-start to completion in (a) the same number of steps as the unpruned network (b) to the same accuracy (c) at the same sparsities as the best matching subnetworks found by IMP with rewinding. In other words, dynamic sparsity makes it possible to improve upon the hypothetical efficiency of the sparse subnetworks found in Chapter 4: the dense phase at the beginning of training is no longer necessary.

- The subnetworks differ significantly between phase 1 and phase 2, and increasing the overlap between the subnetworks degrades performance. The masks m_1 and m_2 hardly overlap; 80% of the weights in m_2 are not present in m_1 and vice versa. Only a small number of trained weights from phase 1 are necessary to ensure that phase 2 completes successfully. Manually imposing a criterion that m_1 and m_2 must overlap more reduces the accuracy of the trained subnetwork, suggesting that having many new connections is important to the success of dynamic sparse training.

- The regrown weights are not sensitive to initialization. They can be randomly reinitialized without affecting the accuracy of the subnetwork in phase 2. Combined with the previous observation, this means that the tiny number of trained weights carried over from phase 1 to phase 2 are sufficient to allow phase 2 to complete successfully; without those trained weights, the subnetwork in phase 2 would be entirely randomly initialized, which would produce far worse results according to the experiments from Chapters 3 and 4.

- Performance only degrades slightly if the initial sparsity pattern is chosen by random pruning rather than the first IMP phase. Performance barely degrades at all if the initial sparsity pattern is chosen by magnitude pruning at initialization (as described in Chapter 5) rather than the first IMP phase.

- The proposed criterion for growing connections in RigL (use the gradients on a single mini-batch of data) performs no better than randomly growing connections, which performs far worse than using the second phase of IMP to choose which connections to grow.

Combining these observations together, these preliminary results have two implications.

1. The hyperparameters that were used for RigL are very different from those suggested by this experiment. RigL changes the sparsity pattern slightly (30% of weights) hundreds of times throughout training, whereas these experiments require only one change where dramatically more weights are swapped. This does not necessarily mean that these hyperparameters will work better within the context of the RigL algorithm, but they do suggest there may be significnat opportunities to further improve upon those results.

2. The criterion for growing weights in RigL appears to be no more effective than randomly growing weights, implying that there may be an opportunity for further improvement.

If these results hold in general (and, for emphasis, these results are preliminary), then they suggest a different path toward practical sparse-from-the-start training. First, select an initial sparsity pattern by pruning weights with the lowest magnitudes at initialization. Next, train to a point midway through. Then prune the lowest magnitude 80% of weights, re-grow weights in the right places, and randomly initialize them. Finally, train to completion.

Whereas the results presented in the main chapters of this thesis require developing a function that determines which connections to prune at initialization or early in training in order to be practical, this approach requires only a function for which new connections to grow at a point midway through training. Much more information is available at midway through training that at initialization or early in training, so developing a practical heuristic for which connections to grow seems more attainable.

6.2.3 Pruning as Training

Summary of the literature. The lottery ticket hypothesis as described in this thesis focuses on finding sparse networks at initialization that are capable of training to completion in isolation and reaching the same accuracy as the unpruned network. A literature has branched off of this work studying whether high-accuracy pruned networks can be found at initialization *without* training. That is to say, this literature treats pruning itself as a form of training.

This line of work began when Zhou et al. (2019) made the observation that the MNIST lottery tickets were reaching nontrivial accuracy even at initialization. The mere act of applying the pruning mask from after training was improving accuracy. (Note: This phenomenon does not occur on models larger than LeNet for MNIST at high sparsities, including cases where pruning occurs early in training rather than at initialization (Frankle et al., 2020b).)

This finding inspired Zhou et al. to develop pruning strategies that explicitly seek high-accuracy subnetworks within randomly initialized dense networks, which they refer to as *supermasks*. To do so, they use an algorithm in which each parameter of the network is augmented with a mask parameter m, which passes through a sigmoid and, on each step, provides the probability for a sample from a bernoulli distribution that determines whether the parameter is masked. These m parameters are trained with SGD, keeping the weights of the network frozen at their original initializations. This results in surprisingly high accuracy on MNIST and CIFAR-10, although lower than when training the dense networks.

Ramanujan et al. (2020) further extended this work, scaling it up to ImageNet with an algorithm they refer to as *edge-popup*. This algorithm assigns each connection a score s. On the forward pass, only the connections with the top $k\%$ of scores are used. On the backward pass, a straight-through estimator is used to update every score. This algorithm improved upon the results of Zhou et al. and reached 68% top-1 accuracy for ResNet-50 on ImageNet (compared to 76% top-1 accuracy when training all weights in the network).

Commentary. This work has fewer direct applications than producing trainable subnetworks, but it has already found a useful application for efficient model storage at inference time. Wortsman et al. (2020) have shown that supermasks can allow a single randomly initialized network to efficiently store trained models for many tasks. The weights of the randomly initialized network can be produced from a deterministic pseudo-random number generator, meaning the only storage needed is the random seed. Each task can be stored as a pruning mask of the randomly initialized network, taking up much less storage space than that necessary for the weights of a trained model.

This line of research is similar to work on binarized neural networks (BNNs; Hubara et al., 2016), which seeks to train neural networks whose weights can be stored as binary values (e.g., ± 1). Doing so makes it possible to substitute multiply operations for bitwise operations, dramatically reducing the cost of inference. The central distinctions between these two training strategies are that (1) supermasks begin with a randomly initialized network with a variety of weight values, whereas binarized networks set all weights to ± 1 and (2) supermasks have sparse networks, whereas BNNs have all weights set to one of two non-zero values (at least in this conception of BNNs). The optimization procedures for edge-popup and BNNs are superficially similar; both use a straight-through estimator on the backward pass. To the best of my knowledge, these two strategies have not been compared directly, so there is no data on whether the greater variety of weight values in work on supermasks leads to better final networks.

One interesting aspect of the work on supermasks is that the subnetworks produced by edge-popup do not reach higher accuracy when the weights are allowed to train. (According to personal correspondance with Mitchell Wortsmann.) It appears that the act of finding a performant subnetwork within a randomly initialized network does not lead to an object that can optimize effectively. That is to say, the edge-popup algorithm is not an alternative to IMP when it comes to finding matching subnetworks of the kind discussed in this thesis. It is interesting that there is such a difference between the optimizability of subnetworks found by IMP and those found by edge-popup.

6.2.4 Theory About Sparsity

Summary of literature. Inspired by the ideas on pruning as training and supermasks, a literature has emerged on *Proving the Lottery Ticket Hypothesis* (Malach et al., 2020). Malach et al. (2020) prove the statement that, with high probability, any sufficiently large, randomly initialized neural network contains a subnetwork whose functional behavior is close to the behavior of a chosen smaller network.

This statement is clearly true in the abstract: if the randomly initialized network is big enough, it will eventually have a subnetwork whose weights are increasingly identical to those in the smaller network. The challenge of this work is to prove that the large network need not be too large. In the case of Malach et al., the larger network needs to be $O(d^4l^2)$ wider than the smaller network of width d and depth l, and it also needs to be twice as deep. Malach et al. consider this a good result, since it is only polynomially larger than the original network. Pensia et al. (2020) improve upon this result by "connecting pruning random ReLU networks to random instances of the SubsetSum problem;" they show that the width need only be $O(\log(dl))$ wider and twice as deep.

Commentary. This line of work is the best knowledge we have on the lottery ticket hypothesis in the form of formal theory. However, it remains for future work to address the most interesting aspects of the empirical results presented in this thesis:

1. This work still assumes that the sparse subnetworks are found within a larger network than the dense network to which their performance is compared.

2. This work does not yet address optimization: the subnetworks found in this thesis are capable of training to high accuracy. It may be that capturing the effect of training will make it possible to address point 1.

In my view, the optimization story of the sparse networks described in this thesis as the most interesting part of the lottery ticket phenomenon. As studied in-depth in Chapter 4, the success of subnetworks appears to relate to the trajectories they follow on the optimization landscape. In cases where test accuracy improves when the rewinding iteration k is set later in training, the train accuracy also improves. The subnetworks are optimizing better, not just generalizing better; in many cases, the generalization gap worsens. I hope that future work will capture this facet of the experiments in this thesis and bring us closer to a formal understanding of the lottery ticket hypothesis.

6.2.5 Ensembling Through Averaging

The linear mode connectivity results in Chapter 4 have contributed to a separate literature on loss landscape analysis and using weight averaging as a way to ensemble models. My work builds on a number of existing papers that made observations about the structure of the loss landscape. This includes research on Stochastic Weight Averaging (SWA; Izmailov et al., 2018), which proposes a strategy to improve generalization by averaging multiple copies of a network from the late stages of training. This technique only works because the network finds a convex region of the loss landscape toward the end of training in which averaging iterates produces a coherent result.

In parallel, other research showed that the optima found by neural networks at the end of training are connected by nonlinear (but not linear) paths over which loss and error do not increase, a phenomenon known as *mode connectivity* (Draxler et al., 2018; Garipov et al., 2018). These paths exist even between networks trained from different initializations, and they can be simplified down to piecewise linear paths with two segments.

My work on linear mode connectivity built on these results by showing that, from an early point in training, the outcome of optimization appears to be constrained to a single convex region regardless of the sample of SGD noise. This extended the work on SWA by showing that networks are constrained to this convex region from much earlier in training than SWA leverages. It also extended the work on mode connectivity by identifying circumstances where *linear* mode connectivity occurs, making it possible to use weight averaging to ensemble models.

Wortsman et al. (2021b) built on the linear mode connectivity results by developing a technique to train entire subspaces of networks from the beginning of training. Concretely, Wortsman et al. trains pairs of randomly initialized networks such that the result of training is a line segment of networks that (a) reach the target accuracy and (b) are functionally diverse. To accomplish (a), Wortsman et al. sample a value $\alpha \in [0, 1]$ and train the interpolation of the two networks on each step. To accomplish (b) and avoid having the two networks collapse into a single network, Wortsman et al. apply a diversity-inducing regularizer. This technique accomplishes both goals in various image classification tasks. Moreover, the technique works for more than two models (producing entire subspaces of trained networks, hence the title of the paper) and—when averaging two models—does better than SWA.

The line of work on model averaging has continued to ensemble increasingly ambitious combinations of models, including those trained in different ways and on different tasks. Neyshabur et al. (2020) observed that, after conducting large-scale pre-training, networks fine-tuned on the same task will always experience linear mode connectivity: they will always find the same linearly-connected region of the loss landscape. Wortsman et al. (2022) built on that observation to use weight averaging to ensemble networks fine-tuned with different hyperparameters (e.g., those resulting from hyperparameter search in the fine-tuning phase of training), which they found to generalize better.

Most recently, Li et al. (2022) used weight averaging to combine copies of a partly-trained language model that were subsequently trained on different domains, producing a model proficient in all of those domains. The implication of this work is that it may be possible to teach a language model about a new domain simply by training a copy of it on that domain and averaging it into an existing, more expensive-to-train model. Moreover, it may be possible to mix and match these domain-specific models depending on the task at hand.

Finally, a line of work has emerged on the origins of the linear mode connectivity phenomenon. Specifically, Entezari et al. (2022) and Ainsworth et al. (2022) have explored the extent to which barriers in the loss landscape emerge from permutations in the model representations and, thereby, whether linear mode connectivity—and the possibility to ensemble via weight averaging—can be induced in any pair of networks by modifying these permutations. Entezari et al. go to heroic lengths to falsify this hypothesis and do not succeed, and Ainsworth et al. propose a technique for resolving these permutation differences between networks that reduces the height of error barriers when linearly interpolating in small-scale settings.

6.3 Implications and Final Thoughts

6.3.1 Sparsity and Deep Learning

The origins of sparsity. This thesis has shown that sparsity (fixing parameters to zero) can emerge earlier than previously understood in the neural networks we train in practice: not only do neural networks learn functions that are representable sparsely, but—when the right parameters are removed—they can train in a sparse fashion as well. These observations shed new light on existing questions and suggest many new questions at the heart of our understanding of deep learning. Why does sparsity emerge in the first place? Is it intrinsic to neural networks in general (e.g., do activations form bottlenecks that can only take advantage of a limited number of inputs)? Or is it a product of choices we make in practice (e.g., the nature of real-world data, the architectures we use in practice, or implicit biases of SGD (Neyshabur et al., 2015)? Does sparsity imply that neural network optimization occurs in a lower-dimensional subspace (Gur-Ari et al., 2018) and, if so, can we identify it (Larsen et al., 2022)?

Sparsity for understanding deep learning. This thesis has shown that—even when sparsity itself is not desirable—sparsifying neural networks can reveal new insights into how all networks—including dense ones—learn and behave in practice. Sparsity puts networks into particularly capacity-scarce regimes that produce extreme behaviors whose residues can also be found when examining dense networks. The most prominent example of this scientific process in this thesis is in Chapter 4, where the challenges with scaling the small-scale lottery ticket behaviors in Chapter 3 forced me to study the optimization beahvior of sparse networks, leading to the linear mode connectivity insights that have spawned a literature of their own.

Sparsity has the potential to make many further such contributions to our broader understanding of deep learning. For example, sparse networks sensitive to initialization in ways that the dense neural networks we typically train are not. Studying this behavior further may yield further insights into the nature of "good" and "bad" optima and why neural networks tend to generalize well even when low-generalization optima are known to exist. This ties to broader questions about the role of overparameterization (where a neural network has enough capacity to memorize the training set) in deep learning, particularly as it pertians to the doule descent phenomenon (Belkin et al., 2019; Liu et al., 2020). Can we only sparsify neural networks in this overparameterized, second-descent regime? Is optimizing a sparse neural network intrinsically more difficult than doing so for dense networks, or does sparsity simply require a different approach to initialization and optimization? With a better understanding of the emergence of sparsity and the nature of sparse optimization problems, perhaps we may eventually be able to identify and train sparse networks from scratch without needing to resort to pruning at all.

6.3.2 Empiricism

More broadly, I hope that the research in this thesis has demonstrated the value of empiricism as a tool for identifying new properties of neural networks. As discussed at the end of Chapter 1, empiricism makes it possible to investigate questions, behaviors, and settings that arise in practice but may be beyond the reach of existing theory. The properties that make deep neural networks exciting rely on intricate decisions about hyperparameters, data, objectives, and fine-tuning strategies. Until we can formally distinguish between good and bad choices for these decisions—e.g., a learning rate that is just right vs. one that is slightly too high—the assumptions underlying our theory will necessarily be too coarse-grained for the proofs to provide insights into the properties that make practical neural networks special.

The research in this thesis is an example of scientific approach. It shares the same goal as theoretical research: to understand the properties of neural networks and the nature of why they work. However, it relies on empiricism rather than mathematical formalism. As a product of this empirical approach, the properties of neural networks enumerated in this thesis are inherently limited. None of them are assuredly and universally true in the way that a proof confers certainty upon a theorem. To the contrary, the properties identified in Chapter 3 were *not* universally true, requiring substantial changes to the core lottery ticket claims and experiments.

However, the properties identified in this thesis hold to some degree in enough real-world instances to have important implications for our understanding of deep learning and for how we train neural networks in practice. In general, a handful of practical scenarios (ResNets for image classification and repurposed for other vision tasks, and various flavors of language model transformers) capture the vast majority of the use cases for neural networks in research and practical settings. Making statements about these networks is useful, even if those statements may not hold in general for all neural networks we use in practice, let alone all neural networks on all possible datasets. In that sense, empiricism is a satisfactory foundation for building useful knowledge about neural networks, even if it is different than the formal mathematical foundation we typically rely on in computer science.

6.3.3 Faster Neural Neural Network Training, Algorithmically

Although the results in this thesis relate to neural network sparsity, they epitomize a broader perspective on neural network efficiency: the way that we currently train neural networks is less efficient than it could be, and—by studying the properties of how those networks learn in practice—we can eliminate extraneous parts of the training process and reduce the cost of training.

Neural network training is approximate computing. We seek to create network with a certain level of quality on some metric (e.g., accuracy or perplexity) while (a) minimizing the time or cost necessary train it and (b) maintaining the same cost in other parts of the machine learning lifecycle (e.g., inference). There are many different recipes for training neural networks that reach similar performance on a chosen set of metrics. The recipes we use today are typically drawn directly from papers that originated these recipes. These recipes are difficult to devise, which may explain reluctance to alter them. However, as the work in this thesis shows, these recipes are inefficient. The work described here shows that many standard neural networks can be trained from at or near the beginning to full accuracy with 80%-90% fewer parameters than standard recipes suggest. This opportunity for improvement is almost certainly true about other aspects of our existing training recipes.

In the time since completing the research in this thesis, I have been working on a startup called MosaicML, whose mission is to identify these opportunities to modify and improve the recipes we use to train practical neural networks. We do so by identifying dozens of the interventions into neural network training (of which sparsity is just one) and putting these interventions together into better training recipes. The results suggest that our existing baselines contain enormous headroom for improvement, with training speedups (measured in terms of wall-clock time to standard quality) of 7x on ResNet-50 on ImageNet, 5x on DeepLab-v3 on ADE20K (an image segmentation benchmark), and 3x-4x on BERT pre-training.

These results rest on the same central insight of this dissertation: that, in the approximate computing problem that is deep learning, there are significant unexploited opportunities to change the workload to reduce cost without affecting the quality of the outcome.

In computer science, we often treat correctness as paramount and our workload as fixed. There are many ways to speed up neural networks while maintaining correctness and leaving the workload unaffected, including building specialized hardware and developing better compilers. However, the efficiency improvements achieved in this way are small compared those attainable when modifying the workload by changing the training recipe. In the years to come, I believe that the results described in this thesis will be one example of a vast collection of algorithmic improvements to neural network training that make it possible for anyone to have access to capabilities that, today, are in the hands of just a few well-resourced organizations.

Bibliography

Samuel K Ainsworth, Jonathan Hayase, and Siddhartha Srinivasa. Git re-basin: Merging models modulo permutation symmetries. *arXiv preprint arXiv:2209.04836*, 2022.

Brian Bartoldson, Ari Morcos, Adrian Barbu, and Gordon Erlebacher. The generalization-stability tradeoff in neural network pruning. *Advances in Neural Information Processing Systems*, 2020.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias-variance trade-off. *Proceedings of the National Academy of Sciences*, 2019.

Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. Deep rewiring: Training very sparse deep networks. In *International Conference on Learning Representations*, 2018.

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In *Proceedings of Machine Learning and Systems 2020*. 2020.

Mathilde Caron, Ari Morcos, Piotr Bojanowski, Julien Mairal, and Armand Joulin. Finding winning tickets with limited (or no) supervision, 2020a.

Mathilde Caron, Ari Morcos, Piotr Bojanowski, Julien Mairal, and Armand Joulin. Pruning convolutional neural networks with self-supervision. *arXiv preprint arXiv:2001.03554*, 2020b.

Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. In *Advances in Neural Information Processing Systems*, 2020a.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 2020b.

Xuxi Chen, Zhenyu Zhang, Yongduo Sui, and Tianlong Chen. Gans can play lottery tickets too. In *International Conference on Learning Representations*, 2021.

Minsu Cho, Ameya Joshi, and Chinmay Hegde. Espn: Extremely sparse pruned networks. *arXiv preprint arXiv:2006.15741*, 2020.

Pau de Jorge, Amartya Sanyal, Harkirat S Behl, Philip HS Torr, Gregory Rogez, and Puneet K Dokania. Progressive skeletonization: Trimming more fat from a network at initialization. *arXiv preprint arXiv:2006.09081*, 2020.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2009.

Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*. Association for Computational Linguistics, 2019.

Felix Draxler, Kambis Veschgini, Manfred Salmhofer, and Fred A Hamprecht. Essentially no barriers in neural network energy landscape. In *International Conference on Machine Learning*, 2018.

Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

Rahim Entezari, Hanie Sedghi, Olga Saukh, and Behnam Neyshabur. The role of permutation invariance in linear mode connectivity of neural networks. In *International Conference on Learning Representations*, 2022.

Utku Evci, Fabian Pedregosa, Aidan Gomez, and Erich Elsen. The difficulty of training sparse neural networks. *arXiv preprint arXiv:1906.10732*, 2019.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, 2020.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*, 2020a.

Jonathan Frankle, David J. Schwab, and Ari S. Morcos. The early phase of neural network training. In *International Conference on Learning Representations*, 2020b.

C Daniel Freeman and Joan Bruna. Topology and geometry of half-rectified network optimization. In *International Conference on Learning Representations*, 2017.

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

Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P Vetrov, and Andrew G Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. In *Advances in Neural Information Processing Systems*, pp. 8789–8798, 2018.

Spyros Gidaris, Praveer Singh, and Nikos Komodakis. Unsupervised representation learning by predicting image rotations. In *International Conference on Learning Representations*, 2018.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010.

Google. Networks for Imagenet on TPUs, 2018. URL `https://github.com/tensorflow/tpu/tree/master/models/`.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training Imagenet in 1 hour, 2017.

Scott Gray, Alec Radford, and Diederik P Kingma. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 3:2, 2017.

Guy Gur-Ari, Daniel A Roberts, and Ethan Dyer. Gradient descent happens in a tiny subspace. *arXiv preprint arXiv:1812.04754*, 2018.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural networks. In *Advances in neural information processing systems*, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. IJCAI'18. AAAI Press, 2018a.

Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, 2017.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 784–800, 2018b.

Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 2021.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, pp. 876–885. Association For Uncertainty in Artificial Intelligence (AUAI), 2018.

Steven A Janowsky. Pruning versus clipping in neural networks. *Physical Review A*, 1989.

Stanislaw Jastrzebski, Maciej Szymczak, Stanislav Fort, Devansh Arpit, Jacek Tabor, Kyunghyun Cho*, and Krzysztof Geras*. The break-even point on optimization trajectories of deep neural networks. In *International Conference on Learning Representations*, 2020.

Tian Jin, Michael Carbin, Daniel M Roy, Jonathan Frankle, and Gintare Karolina Dziugaite. Pruning's effect on generalization through the lens of training and regularization. In *Advances in Neural Information Processing Systems*, 2022.

Neha Mukund Kalibhat, Yogesh Balaji, and Soheil Feizi. Winning lottery tickets in deep generative models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 8038–8046, 2021.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations*, 2017.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

Brett W Larsen, Stanislav Fort, Nic Becker, and Surya Ganguli. How many degrees of freedom do we need to train deep networks: a loss landscape perspective. In *International Conference on Learning Representations*, 2022.

Yann LeCun and Corinna Cortes. URL `http://yann.lecun.com/exdb/mnist/`.

Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, 1990.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pp. 9–48. Springer, 2012.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. SNIP: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations*, 2019.

Namhoon Lee, Thalaiyasingam Ajanthan, Stephen Gould, and Philip H. S. Torr. A signal propagation perspective for pruning neural networks at initialization. In *International Conference on Learning Representations*, 2020.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations*, 2017a.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations*, 2017b.

Margaret Li, Suchin Gururangan, Tim Dettmers, Mike Lewis, Tim Althoff, Noah A Smith, and Luke Zettlemoyer. Branch-train-merge: Embarrassingly parallel training of expert language models. *arXiv preprint arXiv:2208.03306*, 2022.

Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. In *International Conference on Learning Representations*, 2020.

Chaoyue Liu, Libin Zhu, and Mikhail Belkin. Toward a theory of optimization for over-parameterized systems of non-linear equations: the lessons of deep learning. *arXiv preprint arXiv:2003.00307*, 2020.

Tianlin Liu and Friedemann Zenke. Finding trainable sparse networks through neural tangent transfer. In *International Conference on Machine Learning*, 2020.

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, 2017.

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2019.

Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l0 regularization. In *International Conference on Learning Representations*, 2018.

Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, 2017.

Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*. PMLR, 2020.

Arun Mallya, Dillon Davis, and Svetlana Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 67–82, 2018.

Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 2019.

Seyed Iman Mirzadeh, Mehrdad Farajtabar, Dilan Gorur, Razvan Pascanu, and Hassan Ghasemzadeh. Linear mode connectivity in multitask and continual learning. In *International Conference on Learning Representations*, 2021.

Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12, 2018.

Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*. PMLR, 2017.

Ari Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. In *Advances in Neural Information Processing Systems*, pp. 4932–4942, 2019.

Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *International Conference on Machine Learning*, pp. 4646–4655. PMLR, 2019.

Vaishnavh Nagarajan and J Zico Kolter. Uniform convergence may be unable to explain generalization in deep learning. In *Advances in Neural Information Processing Systems*, pp. 11615–11626, 2019.

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. In *International Conference on Learning Representations*, 2020.

Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. Norm-based capacity control in neural networks. In *Conference on Learning Theory*, 2015.

Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. What is being transferred in transfer learning? *Advances in neural information processing systems*, 33:512–523, 2020.

Ankit Pensia, Shashank Rajput, Alliot Nagle, Harit Vishwakarma, and Dimitris Papailiopoulos. Optimal lottery tickets via subset sum: Logarithmic over-parameterization is sufficient. *Advances in Neural Information Processing Systems*, 2020.

Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's hidden in a randomly weighted neural network? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11893–11902, 2020.

Russell Reed. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 1993.

Alex Renda, Jonathan Frankle, and Michael Carbin. Comparing rewinding and fine-tuning in neural network pruning. In *International Conference on Learning Representations*, 2020.

Pedro Savarese, Hugo Silva, and Michael Maire. Winning the lottery with continuous sparsification. *Advances in Neural Information Processing Systems*, 33:11380–11390, 2020.

Andrew Michael Saxe, Yamini Bansal, Joel Dapello, Madhu Advani, Artemy Kolchinsky, Brendan Daniel Tracey, and David Daniel Cox. On the information bottleneck theory of deep learning. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=ry`$_WPG-A-$.

Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning. *arXiv preprint arXiv:2202.05924*, 2022.

Ravid Shwartz-Ziv and Naftali Tishby. Opening the black box of deep neural networks via information. *arXiv preprint arXiv:1703.00810*, 2017.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

Arlene Elizabeth Siswanto. *Block sparsity and weight initialization in neural network pruning*. PhD thesis, Massachusetts Institute of Technology, 2021.

Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464–472. IEEE, 2017.

Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of residual networks using large learning rates. In *International Conference on Learning Representations*, 2018.

Samuel L. Smith and Quoc V. Le. A bayesian perspective on generalization and stochastic gradient descent. In *International Conference on Learning Representations*, 2018.

Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018.

Kartik Sreenivasan, Jy-yong Sohn, Liu Yang, Matthew Grinde, Alliot Nagle, Hongyi Wang, Kangwook Lee, and Dimitris Papailiopoulos. Rare gems: Finding lottery tickets at initialization. *arXiv preprint arXiv:2202.12002*, 2022.

Jingtong Su, Yihang Chen, Tianle Cai, Tianhao Wu, Ruiqi Gao, Liwei Wang, and Jason D Lee. Sanity-checking pruning methods: Random tickets can win the jackpot. *Advances in Neural Information Processing Systems*, 33:20390–20401, 2020.

Hidenori Tanaka, Daniel Kunin, Daniel LK Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *arXiv preprint arXiv:2006.05467*, 2020.

Naftali Tishby, Fernando C Pereira, and William Bialek. The information bottleneck method. *arXiv preprint physics/0004057*, 2000.

Stijn Verdenius, Maarten Stol, and Patrick Forré. Pruning via iterative ranking of sensitivity statistics. *arXiv preprint arXiv:2006.00896*, 2020.

Marc Aurel Vischer, Robert Tjarko Lange, and Henning Sprekeler. On lottery tickets and minimal task representations in deep reinforcement learning. *arXiv preprint arXiv:2105.01648*, 2021.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020.

Mitchell Wortsman, Vivek Ramanujan, Rosanne Liu, Aniruddha Kembhavi, Mohammad Rastegari, Jason Yosinski, and Ali Farhadi. Supermasks in superposition. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2020.

Mitchell Wortsman, Maxwell C Horton, Carlos Guestrin, Ali Farhadi, and Mohammad Rastegari. Learning neural network subspaces. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 11217–11227. PMLR, 18–24 Jul 2021a.

Mitchell Wortsman, Maxwell C Horton, Carlos Guestrin, Ali Farhadi, and Mohammad Rastegari. Learning neural network subspaces. In *International Conference on Machine Learning*, pp. 11217–11227. PMLR, 2021b.

Mitchell Wortsman, Gabriel Ilharco, Samir Ya Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, et al. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *International Conference on Machine Learning*, pp. 23965–23998. PMLR, 2022.

Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G. Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Toward more efficient training of deep networks. In *International Conference on Learning Representations*, 2020.

Haonan Yu, Sergey Edunov, Yuandong Tian, and Ari S. Morcos. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp. In *International Conference on Learning Representations*, 2020.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Hongyi Zhang, Yann N. Dauphin, and Tengyu Ma. Residual learning without normalization via better initialization. In *International Conference on Learning Representations*, 2019.

Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *Advances in neural information processing systems*, 2019.

Michael H. Zhu and Suyog Gupta. To prune, or not to prune: Exploring the efficacy of pruning for model compression, 2018.