

https://dspace.org/

On sparsity in deep learning: the benefits and pitfalls of sparse neural networks and how to learn their architectures

Tessera, Kale-ab

2021
https://hdl.handle.net/10539/33097

Downloaded from a DSpace repository. https://dspace.org/

WITS UNIVERSITY
University of the Witwatersrand, Johannesburg
School of Computer Science and Applied Mathematics

On Sparsity in Deep Learning: The Benefits and Pitfalls of Sparse Neural Networks and How to Learn Their Architectures

Kale-ab Tessera
Supervised by: Prof. Benjamin Rosman



WITS
UNIVERSITY

A research report submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science.

April 2021

Declaration

I declare that this research report is my own, unaided work, except where otherwise acknowledged. It is being submitted for the Degree of Master of Science to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.



(Signature of candidate)

27 day of July 20 21 at Johannesburg.

Abstract

Overparameterization in deep learning has led to many breakthroughs in the field. However, overparameterized models also have various limitations, such as high computational and storage costs, while also being prone to memorization. To address these limitations, the field of sparse neural networks has gained a renewed focus.

Training sparse neural networks to converge to the same performance as dense neural architectures has proved to be elusive. Recent work suggests that initialization is the key. However, while this research direction has had some success, focusing on initialization alone appears to be inadequate. In this work, we take a broader view of training sparse networks and consider the role of regularization, optimization, and architecture choices on sparse models. We propose a simple experimental framework — *Same Capacity Sparse vs Dense Comparison* (`SC-SDC`) — that allows for a fair comparison of sparse and dense networks. Furthermore, we propose a new measure of gradient flow — *Effective Gradient Flow* (`EGF`) — that better correlates to performance in sparse networks. Using top-line metrics, `SC-SDC` and `EGF`, we show that the default choices of optimizers, activation functions and regularizers used for dense networks can disadvantage sparse networks.

Another issue with sparse networks is the lack of efficient, flexible methods for learning their architectures. Most current approaches only focus on learning convolutional architectures. This limits their application to Convolutional Neural Networks (CNNs) and results in a large search space, since each convolutional layer requires learning hyperparameters such as the padding, kernel, and stride size. To address this, we use techniques that leverage Neural Architecture Search (NAS) methods to learn sparse architectures in a simple, flexible, and efficient manner. We propose a simple NAS algorithm — *Sparse Neural Architecture Search* (`SNAS`) — and a flexible NAS search space that we use to learn layer-wise density levels (percentage of active weights). Due to the simplicity of our approach, we can learn most architecture types, while also having a smaller search space. Our results show that we can consistently learn sparse Multilayer Perceptrons (MLPs) and sparse CNNs that outperform their dense counterparts, with considerably fewer weights. Furthermore, we also show that the learned architectures are competitive with state-of-the-art architectures and pruning methods.

Based upon these findings, we show that reconsidering aspects of sparse architecture design and the training regime, combined with simple search methods, yields promising results.

To my family — Bruk, Emebet and Abebe, and to my partner — Maxine.

Acknowledgements

I would like to start by thanking my supervisor, Prof. Benjamin Rosman, without whom this work would not have been possible. Benji, I am incredibly grateful for your guidance, invaluable feedback, and willingness to let me explore exciting ideas. I am also thankful that, without fail, you always made time in your busy schedule for our weekly meetings and for your unwavering support throughout the process.

I would also like to acknowledge Sara Hooker, a colleague and collaborator on some of the presented work. Thank you, Sara, for your collaboration, support, and indispensable guidance. It was truly a pleasure to work with you, an opportunity I sincerely appreciate and that has greatly helped develop my skills as a researcher.

I am also indebted to the communities that helped foster my growth. Notably, the Deep Learning Indaba, an organization that aims to strengthen African Machine Learning. I found my passion for machine learning at the annual Indaba conference, which is also where I met my collaborators. Furthermore, together with Microsoft, the Indaba allowed me to attend NeurIPS (Neural Information Processing Systems) 2019, a major global machine learning conference. I am genuinely thankful for the warm community that the Indaba has created and its significant impact on my research path. In addition, I would like to thank the RAIL (Robotics, Autonomous Intelligence and Learning) Lab at the University of the Witwatersrand (Wits). It was a fantastic community of like-minded individuals where I could share and discuss fascinating topics.

I would also like to thank Shunmuga Pillay and the MSS (Mathematical Sciences Support) team for maintaining and upgrading the Wits computing cluster, without which most of the large-scale experiments would not have been possible.

Finally, I would like to express my deepest appreciation to my family. To my parents, Abebe and Emebet, thank you for the sacrifices you have made to ensure your kids get the best education possible. Thank you for your constant encouragement and support. To my brother, Bruk, thank you for always believing in me and reassuring me in my times of doubt. To my partner, Maxine, thank you for your love and support. It truly helped me get through the difficult times.

Contents

1 Introduction **9**
- 1.1 Overview . . . 9
- 1.2 Contributions . . . 11
- 1.3 Research Report Structure . . . 11

2 Background Work **13**
- 2.1 Introduction . . . 13
- 2.2 Feedforward Neural Networks . . . 13
  - 2.2.1 Perceptron . . . 13
  - 2.2.2 Logistic Regression . . . 14
  - 2.2.3 Multilayer Perceptron (MLP) . . . 15
  - 2.2.4 Convolutional Neural Networks (CNNs) . . . 15
- 2.3 Activation Functions . . . 16
  - 2.3.1 Sigmoidal Activations . . . 16
  - 2.3.2 ReLU . . . 17
- 2.4 Optimization in Neural Networks . . . 18
  - 2.4.1 Cost Function . . . 18
  - 2.4.2 Optimization Algorithms . . . 19
- 2.5 Regularization and Normalization . . . 21
  - 2.5.1 Regularization . . . 21
  - 2.5.2 Batch Normalization . . . 22
- 2.6 Different Notions of Sparsity . . . 22
- 2.7 Pruning . . . 23
  - 2.7.1 When to Prune . . . 23
  - 2.7.2 What to Prune . . . 24
- 2.8 Neural Architecture Search . . . 25
  - 2.8.1 Introduction . . . 25
  - 2.8.2 Search Space . . . 26
  - 2.8.3 Search Strategy . . . 27
  - 2.8.4 Performance Estimation . . . 30
- 2.9 Conclusion . . . 30

3 Keep the Gradients Flowing: Using Gradient Flow to Study Sparse Network Optimization **31**
- 3.1 Introduction . . . 31
- 3.2 Methodology . . . 33
  - 3.2.1 Same Capacity Sparse vs Dense Comparison (`SC-SDC`) . . . 33
  - 3.2.2 Measuring Gradient Flow . . . 34
- 3.3 Empirical Setting . . . 37
  - 3.3.1 Average `EGF` . . . 37

3.3.2 Architecture, Normalization, Regularization and Optimizer Variants . . . 37
3.3.3 SC-SDC MLP Setting . . . 38
3.3.4 Extended CNN Setting . . . 38
3.4 Results and Discussion . . . 38
3.4.1 Comparison of Dense and Sparse Interventions Using SC-SDC . . . 38
3.4.2 Generalization of Results Across Architecture Types - Wide ResNet-50 . . . 42
3.4.3 Generalization of Results From Random Pruning to Magnitude Pruning . . . 43
3.5 Conclusions . . . 43

4 Learning Sparse Neural Network Architectures 45
4.1 Introduction . . . 45
4.2 Motivation . . . 47
4.3 Sparse Neural Architecture Search . . . 48
4.3.1 Search Space . . . 48
4.3.2 Search Strategy . . . 50
4.3.3 Performance Evaluation and Estimation . . . 50
4.3.4 Sparse Neural Architecture Search (SNAS) Algorithm . . . 51
4.4 Results . . . 52
4.4.1 Experiment Setup . . . 52
4.4.2 Searching for MLP Architectures . . . 54
4.4.3 Searching for CNN Architectures . . . 55
4.5 Limitations . . . 57
4.6 Conclusion . . . 57

5 Related Work 59
5.1 Introduction . . . 59
5.2 Pruning From Scratch . . . 59
5.2.1 Connection Sensitivity . . . 59
5.2.2 Random Pruning . . . 60
5.3 Neural Architecture Search for Sparse Networks . . . 60
5.4 Sparse Network Dynamics . . . 61
5.4.1 Sparse Network Optimization as Pruning Criteria . . . 61
5.4.2 Sparse Network Optimization to Study Network Dynamics . . . 61
5.5 Conclusion . . . 62

6 Conclusions and Future Work 63

Appendices 78

A Sparse Network Optimization 79
A.1 SC-SDC . . . 79
A.1.1 SC-SDC Implementation Details . . . 79
A.1.2 Benefits of SC-SDC . . . 80
A.2 Gradient Flow . . . 81
A.3 Adaptive Methods . . . 81
A.4 Activation Functions . . . 83
A.5 Detailed Results for `SC-SDC` . . . 84
A.5.1 Detailed Results . . . 84

B Learning Sparse Architectures 96
B.1 Detailed SNAS Results . . . 96

List of Figures

2.1 **An Illustration of an MLP.** We provide an illustration of an MLP showing input, hidden and output layers. Bias nodes are not included in the figure. 15
2.2 **Sparse Connectivity in CNNs [Goodfellow et al., 2016].** We show sparse connectivity in convolutional layers of CNNs (top), compared to dense connectivity found in dense layers (below). 16
2.3 **Plot of Sigmoid and Tanh Activation Functions.** We plot the Sigmoid and tanh activation functions and their derivatives using an input range of [-5,5]. . . 17
2.4 **Sparse Propagation of Activations and Gradients of ReLU [Glorot et al., 2011].** ReLU's sparse formulation results in sparse activations, where some activations are inactive (shown in white). 18
2.5 **Plot of the ReLU Activation Function.** We plot the ReLU activation function and its derivatives using an input range of [-5,5]. 18
2.6 **Sparse Activity (left) and Sparse Connectivity (right).** In sparse activity, we have sparse activations (left), while in sparse connectivity, we have sparse weights (right). 22
2.7 **Different Components of NAS [Elsken et al., 2018b].** We show the different components of Neural Architecture Search (NAS) and how they interact with each other. 26
2.8 **An Illustration of the Different Kinds of NAS Chain-structured Search Spaces.** We show a simple chain only dependent on its parent node (left), a chain with skip connections (middle), and a more complex chain with branching and skip connections (right). 27
2.9 **Best NAS Cells Learned by NASNet [Zoph et al., 2018].** We show the best performing convolutional cells, that were learned by NASNet-a on CIFAR-10. 28
2.10 **NAS Using Reinforcement Learning [Zoph and Le, 2016].** We illustrate how reinforcement learning can be used as a NAS search algorithm. 28

3.1 **Same Capacity Sparse vs Dense Comparison (SC-SDC).** `SC-SDC` is a simple framework that fairly compares sparse and dense networks. This is done by ensuring that the compared sparse and dense networks have the same number of active (nonzero) weights in each layer, and that these active weights are initially sampled from the same distribution. 33
3.2 **Simple Comparison of Sparse and Dense Networks** We show a simple illustration of how we ensure sparse and dense networks have the same parameter count (the sparse network has more hidden nodes than the dense network, but it has the same number of active connections/weights). 35
3.3 **Test Accuracy and Gradient Flow in Sparse and Dense MLPs.** We study the effect of different regularization and optimization methods on test accuracy and average gradient flow, across different learning rates. We see that for Adam, a higher gradient flow tends to correlate to poor performance. The results for all optimizers can be found in Figures A.14 and A.16. 41

3.4 **Effect of Activation Functions on Accuracy and Gradient Flow on CIFAR-100, With a Low Learning Rate (0.001).** We see that Swish is the most promising activation function across most optimizers. The results across all optimizers and learning rates are shown in Figure A.15 and A.17. 42
3.5 **Wide ResNet-50 Test Accuracy on CIFAR-100.** We see that the results achieved on MLPs, using `SC-SDC`, are also consistent in CNNs. The densities range from 1% to 100% (fully dense) and the gradient flow results can be found in Figure A.19. 43
3.6 **Accuracy and Gradient Flow for Magnitude Pruning.** We see that similarly to randomly pruned networks, magnitude pruned networks trained with Adam and $L2$ lead to high `EGF` and poor performance. 44

4.1 **An Illustration of a Simple Neural Network.** We show a simple neural network with 24 weights. 48
4.2 **MLP Sparse Search Space.** We show our sparse ResNet-like cells and how they can be chained to form Sparse ResNet Architectures. In our Sparse ResNet Architecture diagram, we omit the BatchNorm and ReLU cells for conciseness, and D represents our learned density vector, where d_i is the density of weight layer i. 50
4.3 **CNN Sparse Search Space.** We show how we can use an existing CNN architecture and learn the densities for each convolutional and linear layer. D represents our learned density vector, where d_i is the density of weight layer i (convolutional or linear layer). 51
4.4 **Test and Train Accuracy of SNAS MLPs Across Various Depths.** We show the test and train accuracy of the best MLPs learned at different depths, compared to the dense baselines. We see that across all depths, `SNAS` consistently finds better performing MLP architectures than the dense baselines. 53
4.5 **Number of Active Weights in SNAS MLPs.** We compare the number of active weights in the best performing learned MLPs to their dense counterparts. We see that `SNAS` consistently finds much smaller architectures, especially for deeper networks. 54

List of Tables

3.1 **The Average Correlation Between Gradient Flow Measures and Generalization Performance.** We compare the average, absolute Kendall Rank correlation [Kendall, 1938] between different formulations of gradient flow and generalization (test loss and test accuracy). The subscripts (1 or 2) denote the p-norm ($l1$ or $l2$ norm). We see that for sparse networks, our proposed measure, `EGF` (Equation 3.6), consistently has higher absolute correlation to performance compared to standard gradient flow measures ($||\boldsymbol{g}||_1$ and $||\boldsymbol{g}||_2$, Equation 3.5). For each dataset, we highlight the measure with the highest correlation to performance in bold. 36

3.2 **Network Configurations.** Different network configurations for Sparse and Dense Comparisons. 37

3.3 **Wilcoxon Signed Rank Test Results for MLPs with Four Hidden Layers, Trained on CIFAR-100.** We show the results using different optimization and regularization methods, across various sparsity levels as mentioned in Section 3.3.3. We use a p-value of 0.05, with the bold values indicating where we can be statistically confident that sparse networks perform better than dense (reject H_0 from 3.4). We also use a continuous colour scale to make the results more interpretable. This scale ranges from green (0 - likely that sparse networks perform better than dense) to yellow (0.5 - 50% chance that sparse networks perform better than dense) to red (1 - highly likely that sparse networks do not outperform dense - cannot reject H_0 from 3.4). The performance results for all these networks are present in Appendix A.5. 40

4.1 **Network Configurations.** Different network configurations used in `SNAS`. . . . 52

4.2 **Best SNAS MLP Compared to SOTA MLPs.** We see that `SNAS` produces competitive architectures compared to SOTA MLP methods, while learning the smallest models for MLPs trained with SGD. 56

4.3 **SNAS Learning Sparse CNN Architectures.** `SNAS` finds smaller, yet better performing ResNet-32 and Wide ResNet-50 architectures. 56

4.4 **SNAS Compared to Other Pruning Methods.** We compare `SNAS` to other pruning methods. Although this is not a fair comparison, since other pruning methods are constrained to a specific sparsity level, we see that `SNAS` achieves competitive performance. These performance metrics are retrieved from Su et al. [2020]. 57

We have published an arXiv paper based on the work presented in Chapter 3 and this was accepted to the Sparsity in Neural Networks: Advancing Understanding and Practice workshop.

K.A. Tessera, S. Hooker, B. Rosman. Keep the Gradients Flowing: Using Gradient Flow to Study Sparse Network Optimization. Sparsity in Neural Networks: Advancing Understanding and Practice, July 2021.

Chapter 1

Introduction

1.1 Overview

Deep learning has gained immense popularity in the last decade, with breakthroughs in fields such as strategic games [Silver et al., 2016, Schrittwieser et al., 2020, Vinyals et al., 2019, OpenAI et al., 2019], robotics [Andrychowicz et al., 2020], machine translation [Vaswani et al., 2017, Devlin et al., 2018, Brown et al., 2020], computer vision [Karras et al., 2020, Krizhevsky et al., 2012, Russakovsky et al., 2015, He et al., 2016] and scientific discovery [Deepmind, 2020]. These advances in deep learning have occurred recently, even though most of the fundamental principles, such as backpropagation and shallow neural networks, have existed for decades [Rosenblatt, 1958, Werbos, 1982]. The main catalysts for these advances were the availability of more data and the usage of larger neural networks, combined with the emergence of Graphics Processing Units (GPUs) and more sophisticated software infrastructure [Goodfellow et al., 2016].

Driven by the successes of overparameterized deep neural networks (DNNs), a "bigger is better" race in the number of model parameters has gripped the field of machine learning [Amodei et al., 2018, Thompson et al., 2020]. Additional parameters improve top-line metrics (such as test accuracy), but drive up the cost of training [Horowitz, 2014, Strubell et al., 2019] and increase the latency and memory footprint at inference time [Warden and Situnayake, 2019, Samala et al., 2018, Lane and Warden, 2018]. Moreover, without adequate regularization and training techniques, overparameterized networks can be more prone to memorization [Zhang et al., 2016, Hooker et al., 2019]. This is because the training data can be memorized by the large number of network parameters, instead of learning a model that generalizes well.

To address some of these limitations, there has been a renewed focus on compression techniques that preserve top-line performance, while improving efficiency. A considerable amount of research focus has centred on pruning, where weights estimated to be unnecessary are removed from the network at the end of training [Louizos et al., 2017, Wen et al., 2016, Cun et al., 1990, Hassibi et al., 1993b, Ström, 1997, Hassibi et al., 1993a, Zhu and Gupta, 2017, See et al., 2016, Narang et al., 2017]. Pruning has shown a remarkable ability to preserve top-line performance metrics, even when removing the majority of weights [Gale et al., 2019, Frankle and Carbin, 2019]. Furthermore, pruned networks also have faster training [Dettmers and Zettlemoyer, 2019, Luo et al., 2017] and inference [Molchanov et al., 2016, Luo et al., 2017] times, while also being more robust to noise [Ahmad and Scheinkman, 2019]. Even with the substantial benefits of pruning, most pruning techniques still require training a large, overparameterized model before pruning a subset of weights.

Due to the drawbacks of starting dense prior to introducing sparsity, there has been a recent focus on methods that allow networks that *start* sparse at initialization, to converge to similar performance as dense networks [Frankle and Carbin, 2019, Frankle et al., 2019b, Liu et al., 2018c]. These efforts have focused disproportionately on understanding the properties of initial sparse weight distributions that allow for convergence. However, while this work has had some success, focusing on initialization alone has proved to be inadequate [Frankle et al., 2020, Evci et al., 2019]. Furthermore, certain aspects of sparse networks are poorly understood, such as their sensitivity to learning rates, notably higher ones [Frankle and Carbin, 2019, Liu et al., 2018c, Frankle et al., 2019b]. This hints to optimization in sparse networks not being well understood, even though it appears to be critical to designing well-performing sparse networks.

In this research report, we firstly aim to study sparse network optimization. We take a broader view of why training sparse networks to converge to the same performance as dense networks has proved to be elusive. We reconsider many of the basic building blocks of the training process, such as optimization, regularization, and architecture components, and ask whether they disadvantage sparse networks or not. Furthermore, we provide tooling tailored to the analysis of these networks. Specifically, we propose an experimental framework — *Same Capacity Sparse vs Dense Comparison* (`SC-SDC`), that compares sparse networks to their equivalent capacity dense networks (same number of active connections, depth, and weight initialization).

Recent work has suggested that poor gradient flow is an exacerbated issue in sparse networks [Wang et al., 2020a, Evci et al., 2020]. To this end, we go beyond merely comparing top-line metrics by also measuring the impact on gradient flow of each intervention. To accurately measure gradient flow in sparse networks, we propose a normalized gradient flow measure — *Effective Gradient Flow* (`EGF`). This measure normalizes gradient flow by the number of active weights. Thus it is better suited for studying sparse optimization. We also show that this measure correlates better to top-line metrics in sparse networks than other formulations of gradient flow. We use `EGF` in conjunction with `SC-SDC` to critically study sparse network optimization.

Lastly, we move on from studying sparse network optimization, to briefly discussing and proposing methods on how to learn sparse architectures. As mentioned, there are various drawbacks to starting dense before introducing sparsity. We thereby focus on learning sparse architectures from scratch, without first training overparameterized, dense models. We do this by leveraging Neural Architecture Search (NAS) methods to learn sparse architectures in a simple, flexible, and efficient manner.

Most current NAS methods are restricted to learning Convolutional Neural Network (CNN) architectures [Zoph et al., 2018, Real et al., 2019, Baker et al., 2016, Suganuma et al., 2017], which limits their application. Contrary to this, we propose a simple NAS algorithm — *Sparse Neural Architecture Search* (`SNAS`) — and a flexible NAS search space that we use to learn layer-wise density levels (percentage of active weights per layer) across various architectures. We propose variants of our search space that can be used to learn Multilayer Perceptron (MLP) and CNN architectures. However, due to its flexibility, it can also be adapted to any architecture that contains linear or convolutional layers such as Recurrent Neural Networks (RNNs) or Transformers [Vaswani et al., 2017]. Furthermore, our `SNAS` algorithm is also adaptable. It currently uses random search and Bayesian Optimization as search algorithms, but these methods can be replaced by more sophisticated search algorithms, without modifying `SNAS`. The performance evaluation and estimation techniques in `SNAS` can also easily be replaced with other approaches.

Using our search space and `SNAS`, our results show that we can consistently learn sparse MLPs

and sparse CNNs that outperform their dense counterparts, with considerably fewer weights. The MLPs learned range from networks with two hidden layers to networks with thirty-two hidden layers, while our learned CNN architectures are based on ResNet-32 [He et al., 2016] and Wide ResNet-50 [Zagoruyko and Komodakis, 2016]. Finally, we also show that the learned architectures are competitive with state-of-the-art architectures and pruning methods.

Based on these findings, we show that going beyond initialization and studying sparse network optimization yields insight into how to better tailor training to sparse networks. This better understanding of sparse network training dynamics, combined with simple architecture search methods, yields promising results.

1.2 Contributions

Our main contributions can be enumerated as follows:

1. ***New Tooling to Study Sparse Network Optimization*** (Chapter 3): We conduct large-scale experiments to establish a better understanding of sparse network optimization. Consequently, we propose a new experimental framework — *Same Capacity Sparse vs Dense Comparison* (`SC-SDC`) — that fairly compares sparse networks to their equivalent capacity dense networks (same number of active connections, depth, and weight initialization). Furthermore, we propose a new measure of gradient flow, *Effective Gradient Flow* (`EGF`), that we show to be a stronger predictor, in sparse networks, of top-line metrics than current gradient flow formulations.

2. ***Insights into Sparse Network Optimization*** (Chapter 3): With `SC-SDC`, `EGF` and test accuracy, we show insights into sparse network optimization. We show that batch normalization (BatchNorm) [Ioffe and Szegedy, 2015] is statistically more critical for sparse networks than for dense networks, suggesting that gradient instability is a crucial obstacle for sparse networks. Furthermore, we show that certain optimizers are sensitive to higher gradient flow, specifically ones that use an exponentially weighted moving average (EWMA) to obtain an estimate of the variance of the gradient, such as Adam [Kingma and Ba, 2014] and RMSProp [Hinton et al., 2012]. This results in these methods being sensitive to methods such as $L2$ regularization. Finally, we also benchmark various activation functions and show that Swish [Ramachandran et al., 2017] and PReLU [He et al., 2015] are promising activation functions, particularly in the sparse regime.

3. ***Simple, flexible way to learn well-performing Sparse Architectures*** (Chapter 4): In our brief look at learning sparse architectures, we propose a flexible NAS search space and algorithm — `SNAS` — that learns competitive sparse MLP and CNN architectures. These sparse architectures consistently outperformed their dense variants, while having considerably fewer weights. They were also competitive with state-of-the-art architectures and pruning methods. Due to the flexibility of our approach, our proposed method can also be used to learn most architecture types, with a variety of search algorithms, evaluation, and performance estimation methods.

1.3 Research Report Structure

Having discussed our problem area and our main contributions, the rest of this research report is organized as follows:

- In Chapter 2, we present an overview of the background work, specifically related to the fundamentals of deep learning, pruning and Neural Architecture Search (NAS). This work

provides the foundations and necessary knowledge required for the rest of the research report.

- In Chapter 3, we use our proposed measure of gradient flow, `EGF`, and our `SC-SDC` framework to study sparse network optimization across different optimizers, activation functions, architectures, learning rates and regularizers.

- In Chapter 4, we propose a simple, flexible method to learn sparse architectures. We give the details of the formulation of our sparse NAS search spaces and `SNAS` algorithm. Furthermore, we compare our learned sparse models to the base dense architectures, other state-of-the-art architectures, and pruning methods.

- In Chapter 5, we present related work, specifically focused on pruning, sparse architecture search and sparse network dynamics.

- Finally, in Chapter 6, we summarize our main contributions and propose possible avenues for future work.

Chapter 2

Background Work

2.1 Introduction

In this chapter, we introduce the background work that will form the foundations for studying sparse network optimization (Chapter 3) and learning sparse architectures (Chapter 4). Firstly, in Section 2.2, we provide the formulation of neural networks. Having provided their formulation, we discuss activation functions (Section 2.3) and neural network optimization (Section 2.4). In Section 2.5, we consider regularization and normalization techniques. Moving on from the foundational components of neural networks, we look at the different kinds of sparsity in neural networks (Section 2.6) and pruning methods (Section 2.7). Finally, in Section 2.8, we discuss Neural Architecture Search (NAS).

2.2 Feedforward Neural Networks

This section outlines some of the basic properties of neural networks. We start by outlining the formulation of a perceptron and logistic regression, and then we build up to Multilayer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs).

2.2.1 Perceptron

A perceptron is a type of artificial neuron and linear binary classifier, that was the first model that could learn weights from examples of inputs [Rosenblatt, 1958]. It takes one or more values as inputs and produces a single binary output.

It is formulated as follows:

$$z = \sum_{i=1}^{D} \theta_i x_i + b = \theta^{\intercal} \cdot x + b \quad , \tag{2.1}$$

where D is our input dimension, $x \in R^D$ and refers to the inputs, $\theta \in R^D$ and refers to the trainable weights, $b \in R^1$ is our bias unit, $\theta^{\intercal} \cdot x$ is the dot product $\sum_{i=1}^{D} \theta_i x_i$ (using vector notation) and $z \in R^1$ is our intermediate output. This is the formulation for the output of Linear Regression [Goodfellow et al., 2016].

The perceptron uses a Heaviside step function [Abramowitz et al., 1972] as an activation function. This function returns one for all positive arguments greater than zero and returns

zero otherwise. It is expressed as follows:

$$step(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \quad , \tag{2.2}$$

where *step* is the Heaviside step function and x is our input to this function.

Thereby, our full perceptron formulation is as follows:

$$\hat{y} = step(z) \quad , \tag{2.3}$$

where z follows from Equation 2.1, *step* is the Heaviside step function (Equation 2.2) and $\hat{y}$ is the binary output of our function.

2.2.2 Logistic Regression

A related model to a perceptron is logistic regression. Logistic regression has the same form as the perceptron model, except that it uses the Sigmoid activation function [Verhulst, 1838] (in two-class logistic regression) instead of the Heaviside step function. The Sigmoid activation function is defined as follows:

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

The Sigmoid activation function squishes the output of the linear function (Equation 2.1) to be between zero and one. Furthermore, the outputs are continuous, which allows them to be interpreted as a probability; thereby, logistic regression can account for uncertainty [Goodfellow et al., 2016].

While Sigmoid allows us to do binary classification, for multiclass logistic regression we use the Softmax activation function [Luce, 2012]. It returns probabilities per class and is defined as follows [Goodfellow et al., 2016]:

$$softmax(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)} \quad , \tag{2.5}$$

where x is a vector, $softmax(x)_i$ is the probability of class i and K is the number of classes. The vector returned by the softmax function can be interpreted as conditional probabilities per class, given an input, e.g. $softmax(x)_i = p(y = i|x)$.

Following from Equation 2.4, the output of two-class logistic regression is defined as follows:

$$\hat{y} = Sigmoid(z) \quad , \tag{2.6}$$

where z follows from Equation 2.1, *sigmoid* is the Sigmoid activation function (Equation 2.4), and $\hat{y} \in R^1$ and is the output.

While the output for multiclass logistic regression is defined as follows:

$$\hat{y} = softmax(z) \quad , \tag{2.7}$$

where z follows from Equation 2.1, $softmax$ is the softmax activation function (Equation 2.5) and our output $\hat{y} \in R^K$, where K is the number of classes.

2.2.3 Multilayer Perceptron (MLP)

Neural networks, also known as Multilayer Perceptrons (MLPs), can be considered an extension of logistic regression, with one or more intermediary layers between the input and output layers, known as hidden layers [Reed and MarksII, 1999]. Furthermore, MLPs use various activation functions (see Section 2.3), while logistic regression uses the sigmoid activation function.

MLPs can be defined recursively as follows:

$$a^l = \phi\left((\theta^l)^T \cdot a^{l-1} + b^l\right) \quad , \tag{2.8}$$

where ϕ is our nonlinear activation function, which can be a variety of activation functions for intermediary layers and is usually the softmax activation (Equation 2.5) for the final layer (in the case of classification), a^l represents the output of layer l, with $a^0 = x$ (the input to the network), and θ^l represents the weights in layer l. In Figure 2.1, we present an illustration of an MLP.

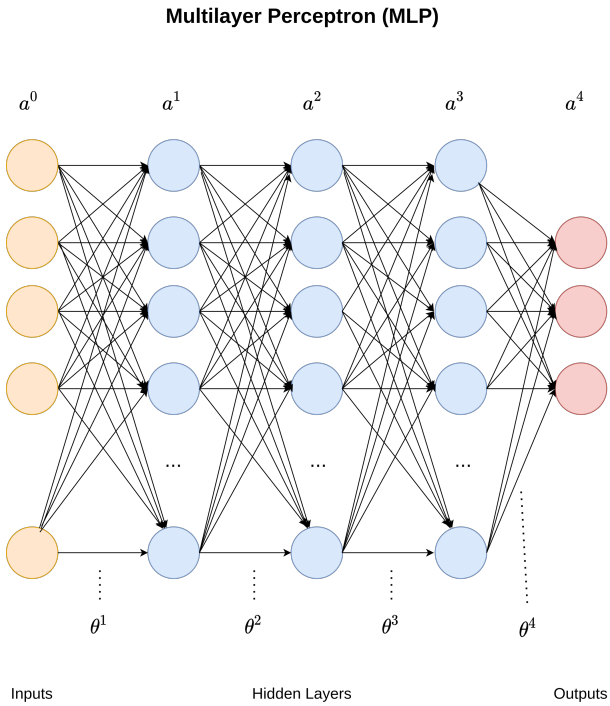


Figure 2.1: **An Illustration of an MLP.** We provide an illustration of an MLP showing input, hidden and output layers. Bias nodes are not included in the figure.

2.2.4 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) [LeCun et al., 1999] are a specialized type of feedforward neural network. They are often used for data with grid-like topologies, such as images [Goodfellow et al., 2016], as they exploit local structure in inputs and assume that spatially nearby pixels are correlated [LeCun et al., 1999]. CNNs typically have three components — convolutional layers, nonlinear activation functions (e.g. ReLU), and pooling layers. Convolutional layers perform convolution operations, which tiles inputs, and multiplies them with filters to produce an output feature map [Brownlee, 2019]. These convolutional layers have hyperparameters such as the number of filters, filter size, stride, and padding. On the other hand, pooling layers apply a summary statistic to the output of the activation layer, for example, max pooling, which chooses the max value from nearby cells, and average pooling, which selects the average value of a group of cells [Goodfellow et al., 2016].

CNNs have various benefits. Firstly, CNNs are translation invariant, which means the representations derived from inputs are unchanged when specific translations occur [LeCun et al., 2015], making them robust models for computer vision. Furthermore, because the convolutional kernels are usually smaller than the input image, CNNs are sparsely connected (sparse connectivity in Figure 2.2). This means the output cells comprise of only a subset of the input cells, as opposed to densely connected layers where the outputs are comprised of all the inputs (dense connectivity in Figure 2.2). Sparse connectivity results in lower memory requirements and better computational efficiency [Goodfellow et al., 2016]. Another benefit of CNNs is that they have reduced memory requirements because of parameter sharing, where each weight of the filter is used for multiple inputs, unlike in fully connected layers where each weight is only used once when computing the output [Goodfellow et al., 2016].

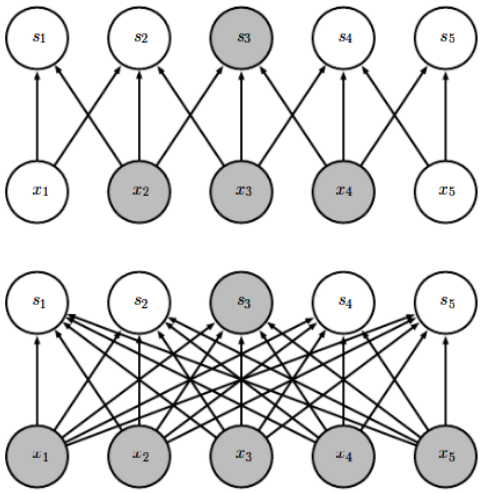


Figure 2.2: **Sparse Connectivity in CNNs [Goodfellow et al., 2016].** We show sparse connectivity in convolutional layers of CNNs (top), compared to dense connectivity found in dense layers (below).

2.3 Activation Functions

Activation functions are fundamental components of neural networks. Their primary purpose is to squeeze function outputs into a desired range or interval.

2.3.1 Sigmoidal Activations

Sigmoidal functions are asymptotically bounded from above and below [Kalman and Kwasny, 1992]. Before the introduction of Rectified Linear Units (ReLU) [Nair and Hinton, 2010, Glorot et al., 2011], these functions were the most popular choices for activation units [Goodfellow et al., 2016].

As shown in Equation 2.4, Sigmoid is used in two-class logistic regression to ensure the output is in the range $[0, 1]$. Furthermore, it is smooth and differentiable everywhere. A closely related activation to Sigmoid is the Hyperbolic Tangent (tanh) function. Tanh can even be formulated in terms of the Sigmoid function as follows, $tanh(x) = 2sigmoid(2x) - 1$ [Goodfellow et al., 2016].

The formulation for tanh itself is as follows:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.9}$$

Furthermore, tanh is easier to train than Sigmoid. When the input to the function is close to 0, $tanh(0) = 0$ while $sigmoid(0) = \frac{1}{2}$ (as can be seen from Figure 2.3), meaning that tanh closely resembles an identity function around this region. This property makes tanh functions easier to train since if activations are kept small, deep tanh networks would resemble linear models [Goodfellow et al., 2016].

(a) Activation with inputs [-5,5]

(b) Derivative with inputs [-5,5]

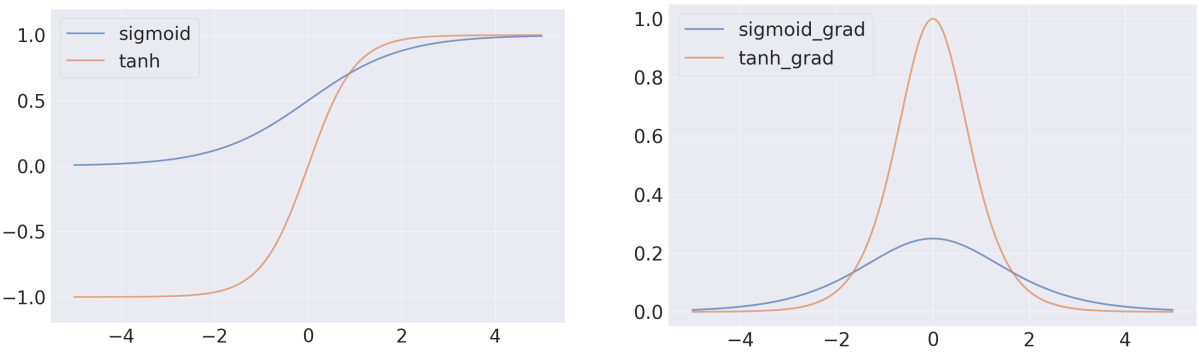


Figure 2.3: **Plot of Sigmoid and Tanh Activation Functions.** We plot the Sigmoid and tanh activation functions and their derivatives using an input range of [-5,5].

2.3.2 ReLU

Currently, Rectified Linear Unit (ReLU) [Nair and Hinton, 2010, Glorot et al., 2011] is the most common activation function [Goodfellow et al., 2016, Zhang et al., 2019a]. It is defined as follows:

$$ReLU(x) = max(x, 0) \tag{2.10}$$

As can be seen from the formulation, ReLU transforms negative inputs to zero, while applying no transformations to positive inputs. Furthermore, as shown in Figure 2.4, ReLU naturally leads to sparse representations, which is more consistent with biological networks [Glorot et al., 2011]. These sparse representations achieved by ReLU have various benefits, including [Glorot et al., 2011]:

- **Variable Sized Representations** - Sparse representations allow the model to control the effective dimension of the representations since the number of active neurons can be learned.
- **Linear Separability** - Sparse representations are more likely to be linearly separable (even with fewer active nonlinearities) because the information is usually represented in higher dimensions.
- **Information Disentanglement** - Dense representations are highly entangled because almost any input change modifies most of the weights, while sparse representations result in the sparse weights usually being conserved when there are small changes in the input.

One of the main reasons for ReLU's popularity is that its gradients are particularly well behaved. Either they are zero (vanish) or let the value through (as shown in Figure 2.5). Furthermore, ReLU is easy to optimize because it is similar to linear units, with the only difference being that ReLU outputs zero across half its domain [Goodfellow et al., 2016]. Variants of ReLU exist, such as leaky ReLU [Maas et al., 2013] and parametric ReLU (PReLU) [He et al., 2015],

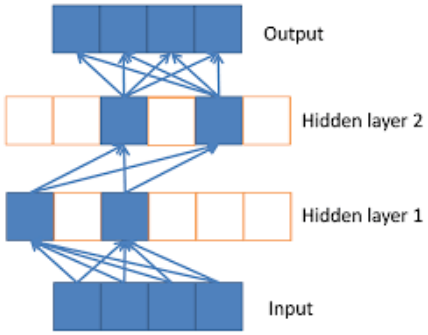
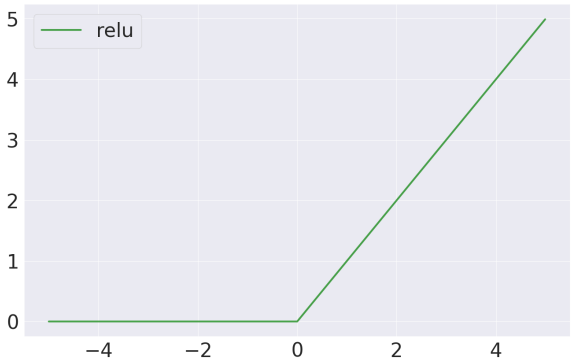


Figure 2.4: **Sparse Propagation of Activations and Gradients of ReLU [Glorot et al., 2011].** ReLU's sparse formulation results in sparse activations, where some activations are inactive (shown in white).

(a) Activation with inputs [-5,5]



(b) Derivative with inputs [-5,5]

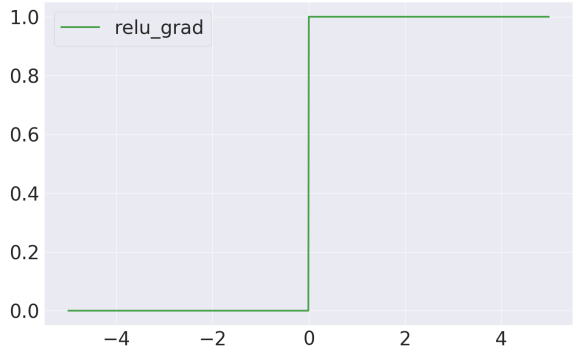


Figure 2.5: **Plot of the ReLU Activation Function.** We plot the ReLU activation function and its derivatives using an input range of [-5,5].

where negative inputs still allow some information to get through (i.e. not zero when the input is negative).

2.4 Optimization in Neural Networks

We have discussed the formulations of feedforward neural networks (Section 2.2) and their activation functions (Section 2.3). In this section, we briefly discuss how optimization in neural networks is achieved, i.e. how we train neural networks.

2.4.1 Cost Function

Optimization in neural networks refers to how we update a neural network's weights to minimize a cost function. This cost function gives us a way to measure how well we perform on a specific task/dataset. Formally, optimization is about finding parameters θ that minimize a cost function $J(\theta)$. This is usually done indirectly as the cost function is based on the training set, while we wish to improve performance on the test set. This relies on the assumption that reducing the training error will reduce the test error [Goodfellow et al., 2016].

We can define our cost function as the average training loss as follows:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\mathrm{data}}} L(\hat{y}, y) \quad , \tag{2.11}$$

where L is the loss per example, $\hat{y}$ is our model's output, y is the actual label (ground truth), and $\hat{p}_{\text{data}}$ is the training data's distribution. We want to optimize the cost function with respect to the underlying data distribution p_{data}, but we often cannot do this because we do not know the true distribution; therefore we optimize our training data distribution ($\hat{p}_{\text{data}}$). We thereby optimize the training error (also known as empirical risk) [Goodfellow et al., 2016].

Empirical risk can be defined as the average loss over the training data:

$$\mathbb{E}_{(\boldsymbol{x},\text{y})\sim\hat{p}_{\text{data}}}[L(\hat{y},y)] = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)},y^{(i)}\right) \quad , \tag{2.12}$$

where m is the number of training samples and L is our loss function.

The loss function L, can be formulated in various ways. In regression, a standard loss function is mean squared error (MSE) [Laplace, 1820], which is formulated as follows [1] :

$$L(\hat{y},y) = (\hat{y}-y)^2 \quad , \tag{2.13}$$

which tells us the square of the error (square of the difference between our predicted output $\hat{y}$ and our actual label y).

For classification problems, we use cross-entropy loss [Kullback and Leibler, 1951]:

$$L(\hat{y},y) = -\sum_{j=1}^{K} y_j \log \hat{y}_j \quad , \tag{2.14}$$

where K is the number of classes. As opposed to MSE, which compares two real numbers, cross-entropy tells us the difference between our classes' true probability distribution y and the distribution predicted by our model $\hat{y}$.

Unfortunately, empirical risk minimization is prone to overfitting as sufficiently large models can memorize the whole training set [Goodfellow et al., 2016]. This is apparent as we can achieve 0% training error and still get inferior results on the test set (overfitting). Overfitting has resulted in various regularization techniques, as discussed in Section 2.5.

2.4.2 Optimization Algorithms

While the cost function gives us a way to measure our performance, optimization algorithms minimize this objective function. In this section, we briefly discuss optimization algorithms, starting with gradient descent and moving on to momentum and adaptive optimization methods.

2.4.2.1 Gradient Descent

Gradient descent is a first-order optimization algorithm used to minimize an objective function $J(\theta)$, by taking small steps in the parameter space θ, according to the derivative of the objective function [Goodfellow et al., 2016, Ruder, 2016].

Gradient descent can be formulated as follows:

$$\theta_t = \theta_{t-1} - \eta\nabla_\theta J(\theta) \quad , \tag{2.15}$$

where θ_t is our updated weights, η is our learning rate and $\nabla_\theta J(\theta)$ is the derivative of the objective function w.r.t. to the parameters θ.

[1] Note for the full MSE, L gets plugged into Equation 2.12.

There are three primary variants of gradient descent. Batch gradient descent computes the gradient updates using the *whole* dataset before doing a single weight update. Stochastic gradient descent computes these updates using *each* example before updating the weights, and minibatch stochastic gradient descent (SGD) computes the gradient updates over a *minibatch of size n* before updating the weights [Ruder, 2016].

Although larger batches provide a more accurate estimate of the gradient, currently minibatch methods are more popular because batch gradient methods are slow, require a lot of memory, and do not scale well with larger datasets, while stochastic methods have high variance since updates are made based on each sample [Ruder, 2016, Goodfellow et al., 2016].

2.4.2.2 Momentum

Although minibatch stochastic gradient descent is a popular choice, it can sometimes be slow [Goodfellow et al., 2016]. This has led to the use of momentum [Polyak, 1964] to speed up the algorithm. Momentum takes an exponentially decaying moving average of past gradients (from past steps) and continues to move in their direction, accelerating SGD in the relevant direction and aids in dampening oscillations [Goodfellow et al., 2016, Ruder, 2016]. This is done by including velocity variable v, which is initialized as zero. The gradient update is as follows:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta_t &= \theta_{t-1} - v_t \quad , \end{aligned} \tag{2.16}$$

where γ is our momentum term that determines our rate of decay and is usually set to 0.9, and v is a velocity term that represents the gradient that is retained from previous iterations.

2.4.2.3 Adaptive Methods

Deciding on the right learning rate is a difficult task. A learning rate that is too small results in little to no progress, while having one too large results in oscillations and could even cause the cost function to diverge [Zhang et al., 2019a, Goodfellow et al., 2016]. This has resulted in the use of learning rate schedules that have different learning rates (usually decayed) for different periods of training.

Another approach to handling learning rates is using adaptive optimization methods. These methods learn parameter specific learning rates, which change over time as opposed to consistent learning rates with non-adaptive methods. We briefly discuss the most popular adaptive methods.

One of the first adaptive optimization methods introduced was Adagrad [Duchi et al., 2011]. It adapts the learning rate individually for each parameter θ, where the learning rate of more frequently occurring features are rapidly decreased, while the learning rate of infrequent features has a slower decrease. This follows the intuition that when an infrequent feature is encountered, the algorithm should take notice and thus have a higher learning rate at that time as opposed to frequent features. This results in Adagrad working well for sparse features [Ruder, 2016]. RMSProp (root mean square propagation) [Hinton et al., 2012], was later proposed as a modification of Adagrad, which works better in nonconvex settings by replacing the gradient accumulation with an exponentially weighted moving average [Goodfellow et al., 2016].

Another algorithm that uses an adaptive learning rate is Adam (Adaptive Moment Estimation) [Kingma and Ba, 2014]. Like RMSProp, Adam stores an exponentially decaying average of past squared gradients (v_t in Equation 2.17). Adam also incorporates an exponentially decaying average of past gradients, similar to a momentum term (m_t in Equation 2.17) [Ruder, 2016].

Adam has proved to be robust and compares favourably to other adaptive methods [Ruder, 2016, Goodfellow et al., 2016].

For brevity, we define $g_t = \nabla_\theta J(\theta)$, and can thereby present the formulation of these adaptive methods below:

Adagrad	RMSProp	Adam
$\mathbf{v}_t = \mathbf{v}_{t-1} + \mathbf{g}_t^2$	$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + (1-\gamma)\mathbf{g}_t^2$	$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\,\mathbf{g}_t$
$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\mathbf{v}_t + \epsilon}} \odot \mathbf{g}_t$	$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\mathbf{v}_t + \epsilon}} \odot \mathbf{g}_t$	$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2$
		$\mathbf{m}_t = \frac{\mathbf{m}_t}{1-\beta_1}$
		$\mathbf{v}_t = \frac{\mathbf{v}_t}{1-\beta_2}$
		$\mathbf{g}'_t = \frac{\eta \hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}$
		$\theta_t = \theta_{t-1} - \mathbf{g}'_t$

(2.17)

where $\mathbf{g}_t$ are the gradients of a network, $\mathbf{m}_t$ is the estimate of the first moment of the gradient, $\mathbf{v}_t$ is the estimate of the second moment of the gradient, η is the learning rate, β_1, β_2, γ are weighting parameters.

2.5 Regularization and Normalization

Regularization and normalization are critical components of modern deep learning. We briefly discuss these methods below.

2.5.1 Regularization

The ability for models to perform well on unseen data is known as generalization. Any modifications made to learning algorithms to reduce this generalization error, while possibly increasing the training error, is known as regularization [Goodfellow et al., 2016]. These methods attempt to reduce overfitting and prevent models from memorizing the training set. They are thereby usually only applied during the training phase.

We briefly discuss the most popular regularization techniques: data augmentation, $L1$ regularization, $L2$ regularization and dropout.

2.5.1.1 Data Augmentation

Data augmentation techniques alter the training data by introducing random, yet minor augmentations to the training data's inputs. These techniques are most popular in object recognition tasks, where images are randomly flipped, cropped, translated or they have their colour transformed [Shorten and Khoshgoftaar, 2019, Krizhevsky et al., 2012].

2.5.1.2 Regularization

Two common regularization methods are $L1$ and $L2$ (weight decay) regularization. These methods both add a regularization term Ω to the cost function J, which drives weight values to be smaller, i.e. closer to the origin [Goodfellow et al., 2016]. They can be represented in the following form:

$$\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \alpha\Omega(\boldsymbol{\theta}) \quad , \tag{2.18}$$

where α is a regularization factor that weights the contribution of each norm penalty, and $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1$ for $L1$ regularization, and $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{\theta}\|_2^2$ for $L2$ regularization.

While both $L1$ and $L2$ regularization encourage weight values to be smaller, $L1$'s regularization term results in more values being zero, i.e. it leads to sparser solutions. This results in $L1$ sometimes being used as a feature selection method [Goodfellow et al., 2016].

2.5.1.3 Dropout

Another popular regularization technique is dropout. This technique randomly drops nodes and weights during training time to prevent strong co-adaption between weights, which could lead to overfitting [Srivastava et al., 2014]. During training, each input element is trained on a different randomly sampled subset of the network referred to as a thinned network. During test time, the learned weights from all thinned networks are scaled and combined into one neural network.

2.5.2 Batch Normalization

Batch normalization (BatchNorm) [Ioffe and Szegedy, 2015] is a method that normalizes activations in neural networks, leading to more stable gradient propagation and faster training with larger learning rates. BatchNorm uses the statistics from each minibatch to normalize activations as follows [Zhang et al., 2019a]:

$$\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta \quad , \tag{2.19}$$

where $\mathbf{x}$ are the activations of a layer, $\hat{\mu}$ is the minibatch's mean, $\hat{\sigma}$ is the minibatch's variance and γ, β are learned parameters. BatchNorm results in these activations having a mean close to zero and unit variance (not exactly zero due to the constants added for numerical stability and the scaling and shifting which occurs after normalization).

2.6 Different Notions of Sparsity

In neural network architectures, sparsity can occur in two distinct forms, namely sparse activity and sparse connectivity [Thom and Palm, 2013]. As shown in Figure 2.6, sparse activity is when only a fraction of neurons are active, while sparse connectivity refers to when neurons are connected to only a subset of other neurons. Both of these forms of sparsity also occur in biological neural networks [Barlow, 1972, Hubel and Wiesel, 1959].

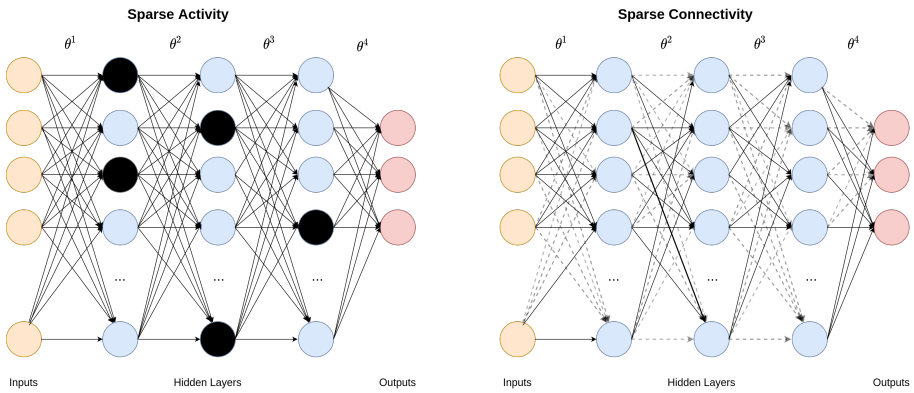


Figure 2.6: **Sparse Activity (left) and Sparse Connectivity (right).** In sparse activity, we have sparse activations (left), while in sparse connectivity, we have sparse weights (right).

Modern neural network architectures make use of both sparse activity and sparse connectivity. Sparse activity can be introduced by using an activation function such as ReLU, which sets the output to zero when the weighted sum of inputs is less than zero (as shown in Section 2.3.2). On the other hand, sparse connectivity can be a result of various methods such as L1 regularization (Section 2.5.1.2), or form part of the chosen architecture, as is the case in Convolutional Neural Networks (CNNs) where it occurs as a result of the receptive fields of convolutional kernels. These receptive fields ensure that only a subset of inputs are used to form an output, as opposed to fully connected layers. Pruning is also a common method for introducing both forms of sparsity, as discussed in Section 2.7.

Outside of neural network architectures, sparsity can be present in the training data, as the inputs and the outputs can be sparse. The input data can be naturally sparse (e.g. recommender system data) or made sparse using sparse coding (also referred to as sparse dictionary learning). Sparse coding [Olshausen and Field, 1997] was originally a way of describing how mammalian brains process images. In modern times, sparse coding is used to reference a class of unsupervised learning methods, which learn sparse latent representations of input data by mapping it into a sparse, linear combination of basis functions [Ng et al., 2013]. Output data can also be naturally sparse or it can be as a result of one-hot encoding, which is a method for representing categorical data.

2.7 Pruning

As mentioned in Section 2.6, network pruning is a popular method for introducing sparse activity and connectivity. This is done by removing unimportant components from a network's architecture [LeCun et al., 1989]. In this section, we discuss the main distinguishing factors for pruning methods, specifically when to prune and what to prune.

2.7.1 When to Prune

Sparsity can be introduced at various periods during a network's lifecycle, namely — before any training has occurred (pruning from scratch), during training (dynamic sparsity), or after training (post-training pruning) [Hoefler et al., 2021].

2.7.1.1 Pruning After Training (Post Training Pruning)

The typical pruning workflow usually occurs in three stages - training, pruning, and fine-tuning [Liu et al., 2018c]. Firstly, in the training stage, dense, overparameterized models are trained. Then in the pruning phase, weights are ranked and those deemed unimportant are removed. Finally, in the fine-tuning phase, the remaining weights are trained and recalibrated. The most common form of pruning — magnitude pruning [Zhu and Gupta, 2017, Han et al., 2015, Hertz, 2018] — follows this three-step process. Although this form of pruning is the simplest to tune since no changes are required to the original dense network's hyperparameter and learning rate configuration [Hoefler et al., 2021], it still requires pre-training an overparameterized, dense network before pruning. This pre-training process is computationally costly and the benefits of sparsity cannot be leveraged during training.

2.7.1.2 Pruning Before Training (Pruning From Scratch)

Methods that prune at initialization aim to start sparse, instead of first pre-training an overparameterized network and then pruning. These methods use certain criteria to estimate at initialization, which weights should remain active. These criteria include using connection sensitivity [Lee et al., 2018], gradient flow (via the Hessian vector product) [Wang et al., 2020a],

conservation of synaptic saliency [Tanaka et al., 2020], and even data-independent, random sparsity [Su et al., 2020, Bourely et al., 2017]. These methods have a lower computational cost than post-training pruning methods since they do not require the training of an overparameterized, dense network before pruning.

2.7.1.3 Pruning During Training (Dynamic Sparsity)

Another pruning approach is to prune during training, also sometimes referred to as Dynamic Sparsity. These methods use information gathered during the training process to dynamically (and at times iteratively) update the sparsity pattern of networks [Mostafa and Wang, 2019, Bellec et al., 2017, Dettmers and Zettlemoyer, 2019, Evci et al., 2019, Ding et al., 2019]. Notably, Mocanu et al. [2018] introduce Sparse Evolutionary Training (SET), a method inspired by biological neural networks [Strogatz, 2001, Pessoa, 2014], that starts with an Erdős–Rényi random graph [Erdős and Rényi, 1960] and then iteratively prunes and grows connections, while maintaining the same number of connections.

Another branch of methods that prune during training are methods that incorporate a sparsity-inducing term in the loss/penalty function [Chauvin, 1988, Collins and Kohli, 2014, Carreira-Perpinán and Idelbayev, 2018, Louizos et al., 2017, Neyshabur, 2020]. A concern with some of these methods ($L1$ and $L2$ based pruning methods) is that they are ineffective when used with batch normalization (BatchNorm) [Azarian et al., 2020, Hoffer et al., 2018, Van Laarhoven, 2017]. This is due to BatchNorm's rescaling of activations (see Section 2.5.2), meaning input weights into activations can become arbitrarily small without influencing the loss, thereby preventing the regularization effect of $L1$ and $L2$ penalties. This is problematic since BatchNorm is a critical part of modern neural architectures.

Dropout techniques have also been used to prune networks during training. Molchanov et al. [2017] used variational dropout, where they had per-parameter dropout rates. Then after training, they removed weights with high dropout rates. Gomez et al. [2019] introduce Targeted Dropout, where dropout is explicitly applied to weights deemed less useful. They rank weights during training according to their magnitude and then they apply dropout to weights with the smallest magnitudes.

Although dynamic sparsity appears to be a good balance between pre-training and post-training pruning methods, computing which elements to prune at run-time, especially if this is done iteratively, is a computationally costly process [Liang et al., 2021]. Furthermore, it is not immediately obvious how to leverage hardware acceleration and sparse storage if the structure continuously changes, as opposed to static variants of sparsity, where there are established methods for storing and accelerating sparse matrices [Tinney and Walker, 1967, Buluç et al., 2009, Duff et al., 1989].

2.7.2 What to Prune

Once the pruning period has been decided, the next vital consideration is which components of a network to prune.

2.7.2.1 Structured vs Unstructured Pruning

One of the key decisions to make when pruning is whether to use unstructured or structured pruning. In unstructured pruning, the redundant components are the weights of a neural network [LeCun et al., 1989, Han et al., 2015, Zhu and Gupta, 2017], while in structured pruning, we remove whole neurons, layers, channels or filters [Liu et al., 2017, Li et al., 2016, Changpinyo et al., 2017, Zhou et al., 2016, Wen et al., 2016].

Since structured pruning only removes whole network units (layers or channels), the architectures remain dense. Thus these methods have training and inference speed ups without dedicated software libraries or hardware [Liu et al., 2017]. However, structured pruning has limited flexibility, since only whole units can be removed. This restricts the sparsity levels that structured pruning can learn, while unstructured pruning methods do not have this constraint and so they are commonly able to learn very sparse networks (95% - 99% sparse, only 1% - 5% of weights are active) [Wang et al., 2020a, Lee et al., 2018, Su et al., 2020, Frankle and Carbin, 2019]. Furthermore, advances in sparse matrix computation and storage [Zhao et al., 2018, Merrill and Garland, 2016, Ma et al., 2019, Zhang et al., 2019b, 2020] could further motivate for the increased adoption of unstructured pruning methods. Due to the potential and flexibility of unstructured sparse networks, we focus on their dynamics and the learning of their architectures.

2.7.2.2 Local vs Global Pruning

Another distinction between pruning methods is whether pruning is applied locally or globally. In local pruning, units (connections or weights) are compared exclusively to units in the same layer, before applying the pruning criteria. In global pruning, all units across all layers are compared together before pruning [Morcos et al., 2019].

2.7.2.3 Pruning Criteria

Pruning criteria are a critical component of pruning, one that is used to determine which network components are unimportant to performance.

Classical pruning methods used the impact of certain components (neurons [Mozer and Smolensky, 1989] and weights [Karnin, 1990]) on the loss or the Hessian of the loss function [LeCun et al., 1989, Hassibi et al., 1993a] as pruning criteria. In the modern era, magnitude pruning [Zhu and Gupta, 2017, Han et al., 2015], where the smallest weights are pruned, is the most common pruning criterion. Although these methods have achieved successful results, they are post-training pruning methods that require training a dense network first. To avoid this initial pre-training period, we focus on pruning from scratch methods.

Pruning from scratch methods use information from the beginning or early in training to estimate which components should be removed. This work is partly inspired by methods that have shown that sparse networks can be trained to competitive performance, without first training an overparameterized network [Frankle and Carbin, 2019, Liu et al., 2018c]. Prominently, Frankle and Carbin [2019] proposed the Lottery Ticket Hypothesis (LTH), where they showed that it is possible to train these sparse subnetworks from scratch as long as you have the correct initialization. We discuss these pruning from scratch methods in more detail in Section 5.2.

2.8 Neural Architecture Search

2.8.1 Introduction

Creating novel and well-performing neural network architectures is a complex task, as there are many design decisions to be made. This is especially the case with modern deep, convolutional, and recurrent neural networks. Neural architecture search (NAS) is a subset of hyperparameter search that focuses on automatically learning neural architectures and their hyperparameters from the input data. NAS algorithms can logically be categorized according to these three components [Elsken et al., 2018b]:

- **Search Space** - architecture space that determines what possible networks can be represented.
- **Search Strategy** - search methods that are used to sample architectures from the search space.
- **Performance Estimation** - techniques that are used to estimate the performance of different architectures, preferably without having to fully train and evaluate each architecture.

In Figure 2.7, we show how these components interact with each other.

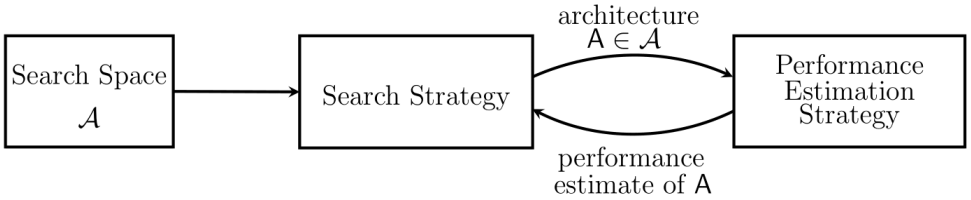


Figure 2.7: **Different Components of NAS [Elsken et al., 2018b].** We show the different components of Neural Architecture Search (NAS) and how they interact with each other.

2.8.2 Search Space

NAS search spaces represent all possible architectures that can be learned; hence this is a vital decision when implementing NAS methods. The search space of all possible architectures is intractable, and so often prior knowledge is used to decrease the size of this search space. Including prior knowledge can also be detrimental as this adds human biases on what possible architectures can be learned [Elsken et al., 2018b]. There should be a balance between the search space's flexibility and size.

2.8.2.1 Chain-Structured Neural Network

NAS search spaces can be represented in various ways, but one of the most popular methods is representing them as computational graphs. In this search space, each node's output is only dependent on the single node before it (the parent node), as illustrated in Figure 2.8 (left). Formally, it is defined as follows [Wistuba et al., 2019]:

$$\boldsymbol{z}^{(k)} = o^{(k)}\left(\left\{z^{(k-1)}\right\}\right) \quad , \tag{2.20}$$

where $\boldsymbol{z}^{(k)}$ represents the value of the current node/layer, $\boldsymbol{z}^{(k-1)}$ is the value of the previous layer/parent layer, and $\boldsymbol{o}^{(k)}$ represents an operation on $\boldsymbol{z}^{(k-1)}$. Examples of operations can include applying convolutions, pooling, or concatenations.

The chain-structured search space was used by Baker et al. [2016], with the possible operations being convolutions, pooling, linear transforms (fully connected dense layer), and termination. These operations have their own hyperparameters, such as the number of filters, kernel size and stride size, for convolutional layers.

Succeeding this work, Zoph et al. [2018] extended this chain-structure to allow for skip connections as illustrated in Figure 2.8 (middle). They also restricted their search space to only search for convolutional architectures, but they included skip connections. With skip connections, each node in the search space is dependent on a parent node and possibly another ancestor node.

This can be formulated as follows [Wistuba et al., 2019]:

$$\boldsymbol{z}^{(k)} = o^{(k)}\left(\left\{\boldsymbol{z}^{(k-1)}\right\} \cup \left\{\boldsymbol{z}^{(i)} | \alpha_{i,k} = 1, i < k-1\right\}\right) \quad , \tag{2.21}$$

where α specifies the set of parents and operations for each node.

Chain-structured search spaces have also been extended to include multi-branch networks, as shown in Figure 2.8 (right).

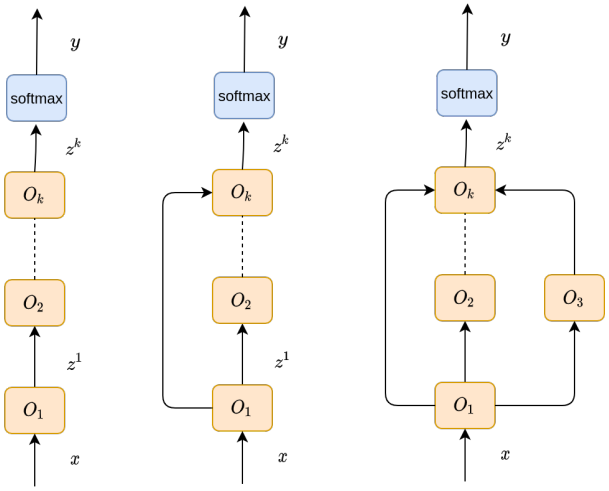


Figure 2.8: **An Illustration of the Different Kinds of NAS Chain-structured Search Spaces.** We show a simple chain only dependent on its parent node (left), a chain with skip connections (middle), and a more complex chain with branching and skip connections (right).

2.8.2.2 Cell-Based Architectures

Zoph et al. [2018] noticed that many learned architectures were composed of many repeating structures and proposed learning cells (best convolutional layers), instead of learning whole architectures. To achieve this, they proposed a cell-based approach for learning architectures, referred to as NASNet. NASNet contains two kinds of cells — a normal cell that maintains the dimension of the feature map, and a reduction cell, which reduces a feature map's spatial dimension by a factor of two. The best convolutional layers that were learned are shown in Figure 2.9. The learned architectures were composed of the same cells, but different weights, meaning the problem changed from learning architectures to learning the best cell structure. Once the best cells were learned, their overall architecture (the combination of reduction and normal cells) was manually chosen. These cells were also transferable. An architecture learned on CIFAR-10 was transferred to ImageNet (with more copies of the cell being stacked to increase capacity) and achieved state-of-the-art performance on both datasets.

2.8.3 Search Strategy

Once we have our search space, we need to decide how to explore and search for architectures in this space. The most popular strategies are reinforcement learning [Zoph and Le, 2016, Baker et al., 2016], random search [Sciuto et al., 2019, Li and Talwalkar, 2019], Bayesian Optimization [Mendoza et al., 2019, White et al., 2019], and evolutionary methods [Real et al., 2017, Elsken et al., 2017].

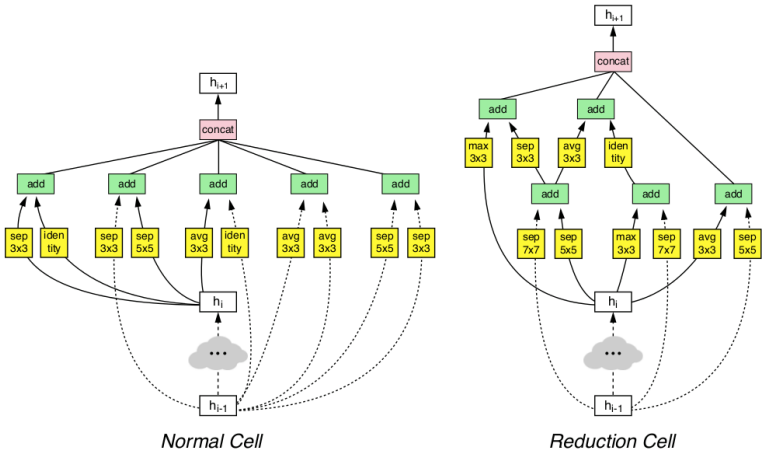


Figure 2.9: **Best NAS Cells Learned by NASNet [Zoph et al., 2018].** We show the best performing convolutional cells, that were learned by NASNet-a on CIFAR-10.

2.8.3.1 Reinforcement Learning

NAS gained popularity in a research context once Zoph and Le [2016] achieved impressive performance on Penn Treebank and CIFAR-10. However, this performance came at a cost, with more than 800 concurrent GPUs trained for three to four weeks, with a total of 12 800 architectures evaluated [Elsken et al., 2018b].

Reinforcement learning approaches to NAS first convert the architecture search problem to a Markov Decision Process (MDP), where A (action space) refers to the search space in NAS, with generating a specific architecture being equivalent to taking an action a. In this case, the reward signal R is the test accuracy. The policy is represented by a Recurrent Neural Network (RNN) that sequentially samples a string, which represents a neural network architecture, and predicts the next action (architecture component) based on previous actions [Zoph and Le, 2016]. This can be seen in Figure 2.10.

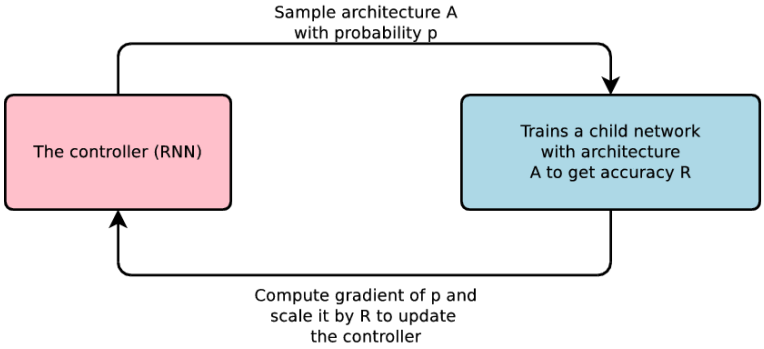


Figure 2.10: **NAS Using Reinforcement Learning [Zoph and Le, 2016].** We illustrate how reinforcement learning can be used as a NAS search algorithm.

Initially, Zoph and Le [2016] used the REINFORCE policy gradient algorithm to find a controller that maximizes the expected reward as follows:

$$J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R] \quad , \tag{2.22}$$

where $a_{1:T}$ is the predicted architecture represented as a sequence of actions, θ_c are parameters of the RNN, and R is the test accuracy. In follow-up work, Zoph et al. [2018] used Proximal Policy Optimization (PPO). Q-learning has also been used to train a policy to sequentially choose layers and their hyperparameters [Baker et al., 2016].

In these approaches, the state of the environment is the actions that have been sampled so far, the reward is only obtained at the final action (no immediate reward) and there is no interaction with an environment. Thereby it has been argued that a multi-armed bandit would be a better approach to these problems [Elsken et al., 2018b].

2.8.3.2 Evolutionary Algorithms

Another search strategy used in NAS is neuro-evolution. Early neuro-evolutionary approaches in NAS used genetic algorithms to optimize architectures and the weights of a neural network [Angeline et al., 1994, Stanley and Miikkulainen, 2002]. Although certain work has shown that genetic algorithms are competitive with gradient methods in certain domains with high variance, such as reinforcement learning domains [Such et al., 2017], modern neuro-evolution approaches in NAS use genetic algorithms only to optimize the neural architectures, but use gradient-based methods for optimizing the weights [Real et al., 2017, Suganuma et al., 2017, Elsken et al., 2018a].

Evolutionary methods in NAS evolve populations, which are groups of chromosomes. Each chromosome represents an encoded representation of a neural network architecture. The fitness of these architectures are evaluated by looking at their accuracy.

Evolutionary algorithms are applied to NAS as follows:

- *Parents from the current population are chosen.* In NAS, the most common approach is tournament selection [Real et al., 2017, Wistuba, 2018, Real et al., 2019].
- *Mutations are applied.* In the context of NAS, examples of mutations are operations such as adding convolutions, altering kernel size, number of channels, strides, learning rates and adding or removing skip connections [Real et al., 2017]. No indication has been given that recombination or crossover of two individuals will result in offspring with better fitness in NAS approaches [Wistuba et al., 2019].
- *The fitness of an architecture is evaluated.* The fitness of an architecture is represented by its test accuracy, which is calculated by training a network or using performance estimation techniques.
- *The surviving generation is chosen.* This is done by various methods, such as elitism (selecting only the best remaining individuals) [Xie and Yuille, 2017] or merely allowing the whole population to survive to ensure diversity [Wistuba, 2018].

2.8.3.3 Bayesian Optimization

Bayesian Optimization (BayesOpt) refers to a class of optimization methods with the following goal:

$$\max_{x \in A} f(x), \tag{2.23}$$

where f is the objective function and A is the feasibility set/domain. BayesOpt is used when f is expensive to evaluate and can only be accessed via point estimates. Furthermore, f is usually continuous since it is often modeled using Gaussian Process regression [Frazier, 2018]. A Gaussian Process (GP) is a collection of random variables, where any finite number of these variables have joint Gaussian distributions [Rasmussen, 2003]. A GP can be fully specified by its mean function $m(x)$ and covariance function $\kappa(x, x')$, which tells us the similarity between two points.

In BayesOpt, our objective function is unknown and so we place a Gaussian process prior

over f. As we get more data from function evaluations, we update our prior and form a posterior distribution over f. This posterior is then used to compute an acquisition function, which determines which areas of our search space to sample next. Examples of acquisition functions include Upper Confidence Bounds (UCB), Probability of Improvement and Expected Improvement [Hoffman et al., 2011].

In the context of NAS, f is the test accuracy of a specific neural architecture x, which we wish to maximize. Furthermore, new covariance functions κ have been derived for computing similarity between two network architectures, where if $\kappa(x, x')$ is large, then $f(x)$ and $f(x')$ are similar architectures [Kandasamy et al., 2018].

2.8.4 Performance Estimation

Once we have our search strategy, we need to evaluate the performance of the chosen architectures. The simplest way of doing this is to train an architecture on the training data and fully evaluate it on test data. This is computationally expensive as each architecture has to be fully evaluated.

Performance estimation methods have been introduced to improve efficiency. These techniques range from early stopping [Zoph et al., 2018, Real et al., 2019], neural network parameter sharing [Pham et al., 2018], dataset sampling [Klein et al., 2016], or using lower quality images [Chrabaszcz et al., 2017]. These approaches rely on these proxy networks or simpler datasets being reasonable estimates of performance on the desired dataset.

Another performance estimation technique is transfer learning, which uses information learned from previous tasks and applies it to a new, somewhat similar domain [West et al., 2007]. Examples in NAS include learning an architecture on a simpler dataset (e.g. CIFAR-10) and transferring it to another dataset (e.g. ImageNet) [Zoph et al., 2018], or using weights from previous performance estimations to initialize a new, yet similar architecture [Wei et al., 2016].

2.9 Conclusion

This chapter discussed various background work related to neural networks — specifically, we discussed their formulation, activation functions, optimization, regularization, and normalization techniques. We also discussed sparsity in neural networks and different aspects of pruning methods. Finally, we introduced Neural Architecture Search (NAS) and its various components such as the search space, search strategy and performance estimation. This relevant background information provides the foundations for studying sparse network optimization and learning sparse architectures.

Chapter 3

Keep the Gradients Flowing: Using Gradient Flow to Study Sparse Network Optimization

3.1 Introduction

Training sparse networks to converge to the same performance as dense neural architectures has proved to be elusive. Most current approaches for learning well-performing sparse architectures from scratch, focus on finding special weight initializations or "lottery tickets" [Frankle and Carbin, 2019, Frankle et al., 2019b, Liu et al., 2018c]. Although this research direction has had some success, focusing on initialization alone has proved to be inadequate [Frankle et al., 2020, Evci et al., 2019].

In this work, we take a broader view of training sparse networks. We reconsider many of the basic building blocks of the training process, such as regularization, optimization, and architecture choices, and ask whether they disadvantage sparse networks or not. Our work focuses on the behaviour of networks with random, fixed sparsity at initialization and we aim to gain further intuition into how these networks learn. Furthermore, we provide tooling tailored to the analysis of these networks.

To study sparse network optimization in a controlled environment, we propose an experimental framework, *Same Capacity Sparse vs Dense Comparison* (`SC-SDC`). Contrary to most prior work comparing sparse to dense networks, where overparameterized dense networks are compared to smaller sparse networks [Frankle and Carbin, 2019, Frankle et al., 2019a, Lee et al., 2019, Evci et al., 2019], `SC-SDC` compares sparse networks to their equivalent capacity dense networks (same number of active connections and depth). This ensures that the results are a direct result of sparse connections themselves and not due to having more or fewer weights (as is the case when comparing large, dense networks to smaller, sparse networks).

Historically, exploding and vanishing gradients were a common problem in neural networks [Hochreiter et al., 2001, Hochreiter, 1991, Bengio et al., 1994, Glorot and Bengio, 2010, Goodfellow et al., 2016]. Recent work has suggested that poor gradient flow is an exacerbated issue in sparse networks [Wang et al., 2020a, Evci et al., 2020]. To this end, we go beyond simply comparing top-line metrics by also measuring the impact on gradient flow of each intervention. To accurately measure gradient flow in sparse networks, we propose a normalized measure of gradient flow, which we term *Effective Gradient Flow* (`EGF`) – this measure normalizes gradient

flow by the number of active weights and is thus better suited to studying the training dynamics of sparse networks. We use this measure in conjunction with `SC-SDC` to see where sparse optimization fails, and consider where this failure could be due to poor gradient flow.

Contributions Our contributions from this chapter are enumerated as follows:

1. **New Tooling to Study Sparse Network Optimization** We conduct large-scale experiments to evaluate the role of regularization, optimization and architecture choices on sparse models. We first introduce a new experimental framework — *Same Capacity Sparse vs Dense Comparison* (`SC-SDC`) — that fairly compares sparse networks to their equivalent capacity dense networks (same number of active connections, depth, and weight initialization). We also propose a new measure of gradient flow, *Effective Gradient Flow* (`EGF`), that we show to be a stronger predictor, in sparse networks, of top-line metrics such as accuracy and loss than current gradient flow formulations.

2. **Batch Normalization Plays a Disproportionate Role in Stabilizing Sparse Networks** We show that batch normalization (BatchNorm) [Ioffe and Szegedy, 2015] is statistically more critical for sparse networks than for dense networks, suggesting that gradient instability is a key obstacle to starting sparse.

3. **Not All Optimizers and Regularizers Are Created Equal** We show that optimizers that use an exponentially weighted moving average (EWMA) to obtain an estimate of the variance of the gradient, such as Adam [Kingma and Ba, 2014] and RMSProp [Hinton et al., 2012], are sensitive to higher gradient flow. This results in these methods, at times, having poor performance when used with $L2$ regularization or data augmentation.

4. **Changing Activation Functions Can Benefit Sparse Networks** We benchmark a wide set of activation functions, specifically ReLU [Nair and Hinton, 2010] and non-sparse activation functions, such as PReLU [He et al., 2015], Swish [Ramachandran et al., 2017], ELU [Clevert et al., 2015], SReLU [Jin et al., 2015] and Sigmoid [Neal, 1992]. Our results show that Swish is a promising activation function when using adaptive optimization methods, while when using stochastic gradient descent (SGD), PReLU and Swish both show promise. We also show that due to Swish's non-monotonic formulation, it leads to better gradient flow, which helps performance in the sparse regime.

Implications Our work is timely as sparse training dynamics are poorly understood. Most training algorithms and methods have been developed to suit training dense networks. Our work provides insight into the nature of sparse optimization. It suggests the need for a broader viewpoint beyond initialization to achieve better performing sparse networks. Our proposed approach provides a more accurate measurement of the training dynamics of sparse networks. This can be used to inform future work on the design of networks and optimization techniques that are tailored explicitly to sparsity.

This chapter is structured as follows: Section 3.2 describes our methodology by introducing our `SC-SDC` framework and `EGF` formulation. Then, in Section 3.3, we describe the details of our empirical setting, such as the different architecture and optimization configurations we use in our experiments. Having described our methodology and empirical setting, we present our results and discussion in Section 3.4. Finally, in Section 3.5, we conclude with our main findings from studying sparse network optimization.

3.2 Methodology

In this work, we study sparse network optimization and measure which architecture and optimization choices favour sparse networks relative to dense networks. To this end, in the following section, we introduce *Same Capacity Sparse vs Dense Comparison* (`SC-SDC`) , a framework that we use to fairly compare sparse and dense networks (Section 3.2.1) and *Effective Gradient Flow* (`EGF`), a better measure of gradient flow in sparse networks to study network optimization (Section 3.2.2). We use both `SC-SDC` and `EGF` in conjunction with test loss and accuracy to study the behaviour of these networks and better understand their training dynamics (Section 3.4).

3.2.1 Same Capacity Sparse vs Dense Comparison (`SC-SDC`)

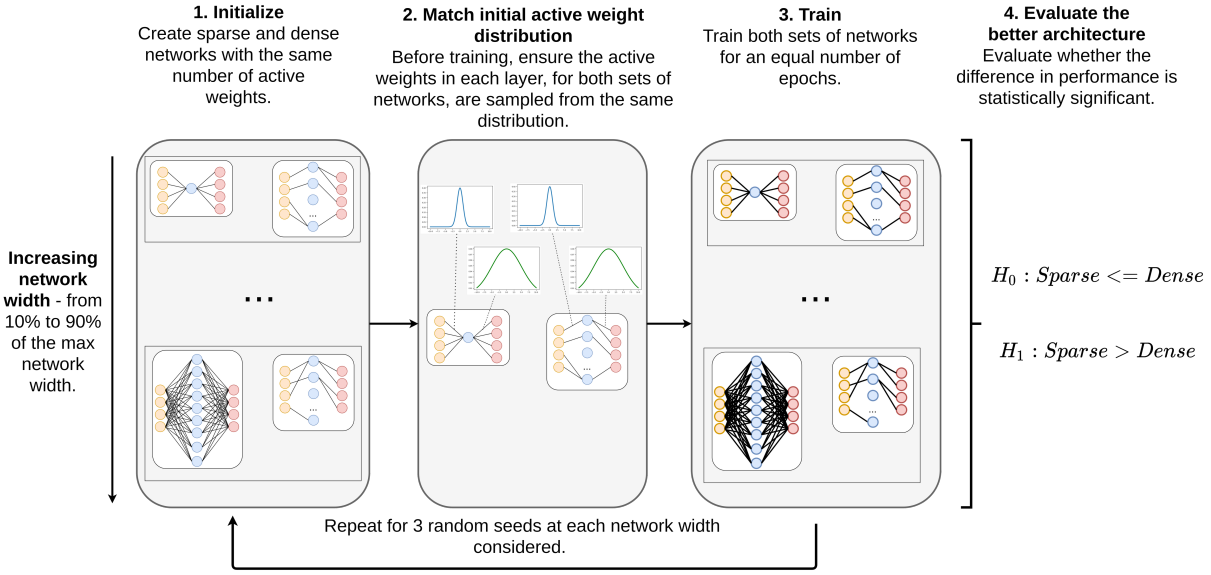


Figure 3.1: **Same Capacity Sparse vs Dense Comparison (SC-SDC).** `SC-SDC` is a simple framework that fairly compares sparse and dense networks. This is done by ensuring that the compared sparse and dense networks have the same number of active (nonzero) weights in each layer, and that these active weights are initially sampled from the same distribution.

To fairly compare sparse and dense networks, we propose *Same Capacity Sparse vs Dense Comparison* (`SC-SDC`), a simple framework that allows us to study sparse network optimization and identify which training configurations are not well suited for sparse networks.

`SC-SDC` can be summarized as follows (see Figure 3.1 for an overview):

1. Initialize For a chosen network depth (number of layers) L and a maximum network width N_{MW}, we compare sparse network S and dense network D at various widths, while ensuring they have the same parameter count. Initially, we mask the weights $\boldsymbol{\theta}_S$ of sparse network S:

$$\boldsymbol{a}_S^l = \boldsymbol{\theta}_S^l \odot m^l \quad , \quad \boldsymbol{a}_D^l = \boldsymbol{\theta}_D^l, \quad \text{for} \quad l = 1, \ldots, L \quad , \tag{3.1}$$

where $\boldsymbol{\theta}_S^l \odot m^l$ denotes an element-wise product of the weights $\boldsymbol{\theta}_S$ of layer l and the random binary matrix (mask) for layer l, m^l, $\boldsymbol{a}_S^l$ is the nonzero weights in layer l of sparse network S, and $\boldsymbol{a}_D^l$ is the nonzero weights in layer l of dense network D (all the weights since no masking occurs).

For a fair comparison, we need to ensure the same number of nonzero weights for sparse

network S and dense network D , across each layer L.

$$||\boldsymbol{a}_S^l||_0 = ||\boldsymbol{a}_D^l||_0, \quad \text{for} \quad l = 1, \ldots, L \tag{3.2}$$

For illustrative purposes, in Figure 3.2, we show a simple example of how we ensure sparse and dense networks have the same parameter count. We provide more implementation details of how we achieve this in Appendix A.1.1.

2. Match active weight distributions Following prior work [Liu et al., 2018c, Gale et al., 2019], we ensure that the nonzero weights at initialization of the sparse and dense networks are sampled from the same distribution at each layer as follows:

$$\boldsymbol{a}_S^l \sim P^l \quad , \quad \boldsymbol{a}_D^l \sim P^l, \quad \text{for} \quad l = 1, \ldots, L \quad , \tag{3.3}$$

where P^l refers to the initial weight distribution at layer l.

This is done by using a normal (or uniform) distribution, with of the same mean (zero mean when using He initialization [He et al., 2015]) and scaling the variance of the sparse network (fan-ins/fan-outs) to the same variance as its equivalent dense network. We use the dense network's number of input neurons (fan-in) and output neurons (fan-outs) when calculating the variance. This ensures that both sets of active weights (sparse and dense) are initially sampled from the same distribution.

3. Train We then train the sparse and dense networks for the same number of epochs, allowing for convergence (500 epochs for Fashion-MNIST, 1000 epochs for CIFAR-10 and CIFAR-100).

4. Evaluate the better architecture We gather the results across the different widths and conduct a paired, one-tail Wilcoxon signed-rank test [Wilcoxon, 1945] to evaluate the better architecture. We have a sample size of 30 networks [1] for each test configuration. This comprises of five different network widths × three independent training runs (different random seeds) × two architectures (sparse and dense variant).

Our null hypothesis (H_0) is that sparse networks have similar or worse test accuracy than dense networks (lower or the same median), while our alternative hypothesis (H_1) is that sparse networks have better test accuracy performance than dense networks of the same capacity (higher median). This can be formulated as:

$$H_0 : Sparse <= Dense \quad , \quad H_1 : Sparse > Dense \tag{3.4}$$

3.2.2 Measuring Gradient Flow

Gradient flow (GF) is used to study optimization dynamics and is typically approximated by taking the norm of the gradients of the network [Pascanu et al., 2013, Nocedal et al., 2002, Chen et al., 2018, Wang et al., 2020a, Evci et al., 2020]. Looking at gradient flow is particularly relevant in the context of sparse networks, since they are sensitive to poor gradient flow [Wang et al., 2020a, Evci et al., 2020]. Hence this would be a useful analysis tool for their optimization.

We consider a feedforward neural network $f : \mathbb{R}^D \to \mathbb{R}$, with function inputs $\mathbf{x} \in \mathbb{R}^D$ and network weights $\boldsymbol{\theta}$. The gradient norm is usually computed by concatenating all the gradients

[1] When calculating the p-values, we use the exact distribution, which is more accurate for smaller sample sizes.

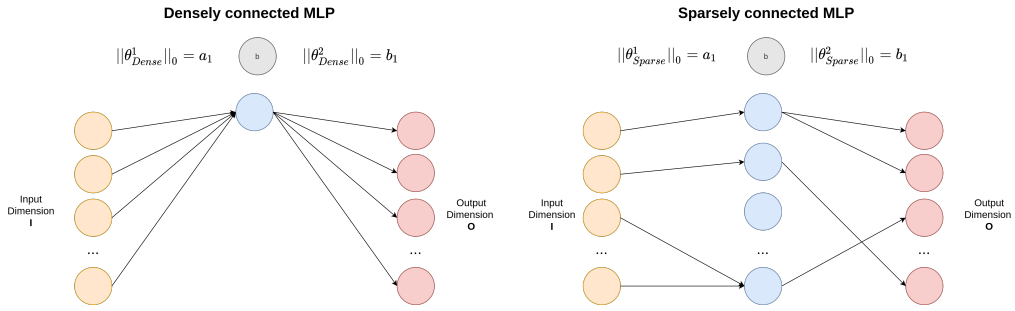


Figure 3.2: **Simple Comparison of Sparse and Dense Networks** We show a simple illustration of how we ensure sparse and dense networks have the same parameter count (the sparse network has more hidden nodes than the dense network, but it has the same number of active connections/weights).

of a network into a single vector, $\boldsymbol{g} = \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}}$, where C is our cost function. Then the vector norm is taken as follows:

$$gf_p = ||\boldsymbol{g}||_p \quad , \tag{3.5}$$

where p denotes the pth-norm and gf_p is the gradient flow. Traditional gradient flow measures take the $L1$ or $L2$ norm of all the gradients [Chen et al., 2018, Pascanu et al., 2013, Evci et al., 2020]. Unless gradients are masked before calculating the gradient flow, the standard gradient flow formulation could result in gradients of masked weights, which do not influence the forward pass, being included in the formulation. Furthermore, computing the $L1$ or $L2$ gradient norm by concatenating all the gradients into a single vector gives disproportionate influence to layers with more weights.

Effective Gradient Flow To address these shortcomings with the standard gf_p measures, we propose a simple modification of Equation 3.5, which we term *Effective Gradient Flow* (`EGF`), that computes the average, masked gradient (only gradients of active weights) norm across all layers.

We calculate `EGF` as follows:

$$\boldsymbol{g} = (\frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}^1} \odot m^1, \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}^2} \odot m^2, \ldots, \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}^L} \odot m^L) \quad , \tag{3.6}$$

$$EGF_p = \frac{\sum_{n=1}^{L} ||\boldsymbol{g}_n||_p}{L} \quad , \tag{3.7}$$

where L is the number of layers. For every layer l, $\frac{\partial \mathcal{C}}{\partial \theta^l} \odot m^l$ denotes an element-wise product of the gradients of layer l, $\frac{\partial \mathcal{C}}{\partial \theta^l}$, and the mask m^l applied to the weights of layer l. For a fully dense network, m^l is a matrix of all ones, since no gradients are masked.

EGF has the following favourable properties:

- **Gradient Flow Is Evenly Distributed Across Layers** `EGF` distributes the gradient norm across the layers equally. This prevents layers with many weights from dominating the measure, and prevents layers with vanishing gradients from being hidden in the formulation, as is the case with equation 3.5 (when all gradients are appended together).

- **Only Gradients of Active Weights Are Used** `EGF` ensures that for sparse networks, only gradients of active weights are used. Even though weights are masked, their gradients are

	Measure	Sparse		Dense	
		Test Loss	Test Accuracy	Test Loss	Test Accuracy
FMNIST	$\|\|\boldsymbol{g}\|\|_1$	0.355	0.316	**0.365**	**0.354**
	$\|\|\boldsymbol{g}\|\|_2$	0.282	0.292	0.285	0.329
	EGF_1	**0.419**	**0.373**	**0.365**	**0.354**
	EGF_2	0.360	0.323	0.298	0.320
CIFAR-10	$\|\|\boldsymbol{g}\|\|_1$	0.440	0.327	**0.380**	0.251
	$\|\|\boldsymbol{g}\|\|_2$	0.447	0.308	0.355	**0.290**
	EGF_1	0.371	0.300	**0.380**	0.252
	EGF_2	**0.451**	**0.332**	0.363	0.287
CIFAR-100	$\|\|\boldsymbol{g}\|\|_1$	0.355	0.385	0.325	0.319
	$\|\|\boldsymbol{g}\|\|_2$	0.373	0.393	0.357	**0.385**
	EGF_1	0.358	0.320	0.325	0.319
	EGF_2	**0.402**	**0.396**	**0.359**	0.382

Table 3.1: **The Average Correlation Between Gradient Flow Measures and Generalization Performance.** We compare the average, absolute Kendall Rank correlation [Kendall, 1938] between different formulations of gradient flow and generalization (test loss and test accuracy). The subscripts (1 or 2) denote the p-norm ($l1$ or $l2$ norm). We see that for sparse networks, our proposed measure, `EGF` (Equation 3.6), consistently has higher absolute correlation to performance compared to standard gradient flow measures ($||\boldsymbol{g}||_1$ and $||\boldsymbol{g}||_2$, Equation 3.5). For each dataset, we highlight the measure with the highest correlation to performance in bold.

not necessarily zero since the partial derivative of the weight w.r.t. the loss is influenced by other weights and activations. Therefore, even though the weight is zero, its gradient can be nonzero.

- **Possibility for Application in Gradient-based Pruning Methods** Tanaka et al. [2020] showed that gradient-based pruning methods like GRASP [Wang et al., 2020a] and SNIP [Lee et al., 2018], disproportionately prune large layers and are susceptible to layer-collapse, which is when an algorithm prunes all the weights in a specific layer. Since `EGF` is evenly distributed across layers, maintaining `EGF` (as opposed to standard gradient norm) could be used as pruning criteria. Furthermore, current approaches measuring or approximating the change in gradient flow during pruning in sparse networks [Wang et al., 2020a, Evci et al., 2020, Lubana and Dick, 2021], could benefit from this new formulation.

To evaluate `EGF` against other standard gradient norm measures, such as the $L1$ and $L2$ norm, we empirically compare these measures and their correlation to test loss and accuracy. We take the average of the absolute Kendall Rank correlation [Kendall, 1938], across the different experiment configurations. We follow a similar approach to Jiang et al. [2019], but unlike their work which has focused on correlating network complexity measures to the generalization gap, we measure the correlation of gradient flow to performance (accuracy and loss). We measure gradient flow at points evenly spaced throughout training.

Our results from Table 3.1 show that in sparse networks, `EGF` consistently has a higher average absolute correlation to both test loss and accuracy. We see that `EGF` has similar correlation to standard measures of gradient flow in dense networks. This shows that the benefits of using `EGF` are more apparent when used in conjunction with sparse networks, since `EGF` only considers the gradients of nonzero weights. Due to the comparative benefits of `EGF` in sparse networks, we use it for the remainder of the paper to measure the impact of interventions. We include

experimental results using other gradient flow measures in Appendix A.2 for completeness.

Configuration	Variants
Optimizers	Adagrad, Adam, RMSProp, SGD and SGD with mom (0.9).
Regularization/Normalization	No Regularization (NR), Weight Decay (L2), Data Augmentation (DA), Skip Connections (SC) and BatchNorm (BN).
Number of hidden layers	1, 2 and 4.
Dense Width	308, 923, 1538, 2153 and 2768.
Activation functions	ReLU, PReLU, ELU, Swish, SReLU and Sigmoid.
Learning rate	0.001 and 0.1.
Datasets	Fashion-MNIST, CIFAR-10 and CIFAR-100.

Table 3.2: **Network Configurations.** Different network configurations for Sparse and Dense Comparisons.

3.3 Empirical Setting

3.3.1 Average `EGF`

To measure gradient flow, we use the Average `EGF` calculated at the end of 11 epochs, evenly spread throughout the training. For example, when using 1000 epochs, we calculate `EGF` at the end of epoch 0, 99, 199, 299, 399, 499, 599, 699, 799, 899 and 999, and compute the average. This is done because it would be computationally costly to calculate gradient norms at the end of every epoch.

3.3.2 Architecture, Normalization, Regularization and Optimizer Variants

We briefly describe our key experiment variants below and also include for completeness all unique variants in Table 3.2.

1. **Activation Functions** ReLU networks [Nair and Hinton, 2010] are known to be more resilient to vanishing gradients than networks that use Sigmoid or Tanh activations, since they only result in vanishing gradients when the input is less than zero, while on active paths, due to ReLU's linearity, the gradients flow uninhibited [Glorot et al., 2011]. Although most experiments are run on ReLU networks, we also explore different activation functions, namely PReLU [He et al., 2015], ELU [Clevert et al., 2015], Swish [Ramachandran et al., 2017], SReLU [Jin et al., 2015], and Sigmoid [Neal, 1992].

2. **Batch Normalization and Skip Connections** We empirically explore the relative benefits of BatchNorm [Ioffe and Szegedy, 2015] and skip connections [Srivastava et al., 2015, He et al., 2016] across dense and sparse networks.

3. **Regularization Techniques** We evaluate various popular regularization methods: weight decay/$L2$ regularization ($\lambda = 0.0001$) [Krogh and Hertz, 1992, Hanson and Pratt, 1989] and data augmentation (random crops and random horizontal flipping [Krizhevsky et al., 2012]).

4. **Optimization Techniques** We benchmark the impact of the most widely used optimizers such as minibatch stochastic gradient descent (SGD) [Robbins and Monro, 1951], minibatch stochastic gradient descent with momentum (momentum term - 0.9) [Sutskever et al., 2013, Polyak, 1964], Adam [Kingma and Ba, 2014], Adagrad [Duchi et al., 2011],

and RMSProp [Hinton et al., 2012].

3.3.3 SC-SDC MLP Setting

We firstly use the `SC-SDC` empirical setting (Section 3.2.1) to evaluate which choices of optimizer, regularization and architecture choices result in a statistically significant performance difference between sparse and dense networks. We train 600 MLPs for 500 epochs on Fashion-MNIST (FMNIST) [Xiao et al., 2017] and more than 10 000 MLPs for 1000 epochs on CIFAR-10 and CIFAR-100 [Krizhevsky et al., 2009]. We compare sparse and dense networks across various widths, depths, learning rates, regularization, and optimization methods as shown in Table 3.2.

Dense Width Following from `SC-SDC`, these networks are compared at various network widths, specifically a width of 308, 923, 1538, 2153, 2768 (10%, 30%, 50%, 70% and 90% of our maximum width $N_{MW}(3076)$ when using CIFAR datasets) as shown in Table 3.2. We use the term **dense width** to refer to the width of a network if that network was dense. For example, when comparing sparse and dense networks at a Dense Width of 308, this means the dense network has a width of 308, while the sparse network has a width of N_{MW} (3076), but has the same number of active connections as its dense counterpart. We provide more details on dense width and other components of the `SC-SDC` implementation in Appendix A.1.1.

3.3.4 Extended CNN Setting

We also extend our experiments to CNNs and train 200 Wide ResNet-50 (WRN) models, specifically the WRN-28-10 variant [Zagoruyko and Komodakis, 2016] for 200 epochs on CIFAR-100. We use the same network training parameters used in the original paper (an initial learning rate of 0.1, dropped by 0.2 at epochs 60,120 and 160, weight decay λ set to 0.0005 and a momentum term of 0.9).

3.4 Results and Discussion

We present our results studying sparse network optimization in MLPs, using `SC-SDC` and `EGF`. Furthermore, we extend our results outside of `SC-SDC`, from MLPs to CNNs and from random pruning to magnitude pruning.

For brevity, we use a shorthand notation for our different regularization/normalization methods — no regularization (*NR*), weight decay (*L2*), data augmentation (*DA*), skip connections (*SC*) and BatchNorm (*BN*). When multiple of these methods are combined, we chain their abbreviations such as *DA_BN*, which represents data augmentation and BatchNorm combined.

Furthermore, for our discussion, we make a distinction between optimization methods that use exponential weighted moving averages, also known as leaky averaging, for estimates of the variance of their gradients (*EWMA optimizers*) — specifically Adam and RMSProp, and methods which do not — Adagrad, SGD and SGD with momentum. We provide details of this distinction in Appendix A.3.

3.4.1 Comparison of Dense and Sparse Interventions Using SC-SDC

In this section, we use the results from `SC-SDC` to identify which optimization choices are currently well suited for sparse networks and which are not. Most of the results discussed are achieved using four hidden layers on CIFAR-100. While we provide the full set of results for Fashion-MNIST, CIFAR-10, and CIFAR-100 in Appendix A.5.

3.4.1.1 Batch Normalization Plays a Disproportionate Role in Stabilizing Sparse Networks

BatchNorm ensures that the distribution of the nonlinearity inputs remains stable as the network trains, which was hypothesized to help stabilize gradient propagation (gradients do not explode or vanish) [Ioffe and Szegedy, 2015].

In Non-EWMA Optimizers, BatchNorm Favours Sparse Networks From Table 3.3a and 3.3b, we see that in non-EWMA optimizers (Adagrad, SGD and SGD with momentum), BatchNorm is statistically more significant for sparse network performance than it is for dense networks. Without BatchNorm (configurations *NR*, *DA*, *L2* and *SC*), we see that sparse networks do not outperform their dense counterparts in test accuracy. With the addition of BatchNorm, across high and low learning rates and at most configurations (except in some cases where *L2* is used), there is strong evidence that sparse networks outperform dense networks.

In EWMA Optimizers, BatchNorm Brings Sparse and Dense Networks Closer in Performance When Adam or RMSProp are used as optimization algorithms, we see that sparse networks outperform dense networks without BatchNorm (*NR* from Table 3.3a and 3.3b). When BatchNorm is added to these methods (*BN*), we see that sparse networks lose their advantage and they perform similarly or slightly better than their dense counterparts in test accuracy and gradient flow, as can be seen from Figure 3.3.

At Shallow Depths, BatchNorm is More Critical to Performance than Skip Connections Figures 3.3a and 3.3c show that BatchNorm (*BN*) is more critical in terms of test accuracy than skip connections (*SC*). Furthermore, we see that BatchNorm (*BN*) performs similarly, in terms of test accuracy, to skip connections and BatchNorm combined (*SC_BN*). This suggests that skip connections do not provide a performance benefit when applied with BatchNorm. We believe this is due to the shallow depth of these networks (four hidden layers), as skip connections have proved to be critical for deeper networks, even with BatchNorm [Balduzzi et al., 2017, Yang et al., 2019, Labatie, 2019].

BatchNorm Stabilizes Gradient Flow Across all optimizers and learning rates, we see that when BatchNorm is used, it results in a lower, more stable `EGF` (Figures 3.3b and 3.3d), compared to the original network without BatchNorm (comparing *NR* to *BN*, *L2* to *L2_BN*, *DA* to *DA_BN* and *SC* to *SC_BN*). This further emphasizes the importance of BatchNorm in stabilizing gradient flow.

3.4.1.2 EWMA Optimizers Are Sensitive to High Gradient Flow

In EWMA Optimizers, $L2$ Regularization can hurt Network Performance For networks without BatchNorm and with $L2$ regularization (configuration *L2*), we can see from Figure 3.3a that when using adaptive methods (Adagrad, Adam and RMSProp), $L2$ regularization mainly hurts sparse network performance. This occurs particularly at high sparsity levels (a dense width of less than 2153). Conversely, equivalent capacity and configured dense networks achieve a performance improvement with $L2$ regularization, compared to their base configuration (comparing *NR* to *L2*).

For networks with BatchNorm, trained with EWMA optimizers, $L2$ regularization adversely affects both sparse and dense network performance. From Figures 3.3c and 3.3d, we see that the addition of $L2$ (configurations *L2_BN* and *DA_L2_SC_BN*) drastically decreases the accuracy of these networks. When we analyse their `EGF`, from Figure 3.3d, we see that the addition of $L2$ consistently across all optimizers results in distinctively larger `EGF` values. This hints at EWMA optimizers being more sensitive to larger gradient norms than other optimizers.

(a) Different Regularization Methods - Low learning rate (0.001)

	NR	DA	L2	SC	BN	DA_BN	L2_BN	SC_BN
Adagrad	1.000	1.000	0.998	0.239	**0.006**	**0.002**	**0.001**	**0.003**
Adam	**0.000**	0.055	0.198	**0.003**	0.079	0.051	0.254	0.166
RMSProp	**0.001**	**0.000**	0.300	0.166	0.117	**0.021**	0.914	0.541
SGD	1.000	1.000	1.000	0.248	**0.000**	**0.000**	**0.001**	**0.003**
Mom (0.9)	1.000	1.000	1.000	0.999	**0.001**	**0.000**	**0.007**	**0.008**

(b) Different Regularization Methods - High learning rate (0.1)

	BN	DA_BN	L2_BN	SC_BN	DA_SC_BN	DA_L2_SC_BN
Adagrad	**0.000**	**0.002**	0.963	**0.002**	**0.023**	**0.014**
Adam	0.070	**0.002**	**0.043**	**0.004**	0.191	0.377
RMSProp	**0.002**	**0.027**	**0.008**	0.562	0.894	**0.002**
SGD	**0.001**	**0.000**	**0.048**	**0.005**	**0.001**	**0.013**
Mom (0.9)	**0.001**	**0.002**	0.488	**0.003**	**0.005**	0.212

(c) Different Activation Functions - High learning rate (0.1)

	ReLU	Swish	PReLU	SReLU	Sigmoid	ELU
Adagrad	**0.023**	**0.005**	**0.050**	0.182	0.568	**0.003**
Adam	0.191	0.182	**0.039**	0.062	**0.005**	**0.000**
RMSProp	0.894	0.167	**0.002**	**0.012**	0.997	0.153
SGD	**0.013**	**0.027**	**0.005**	0.078	**0.030**	0.056
Mom (0.9)	0.212	**0.013**	**0.001**	0.078	**0.001**	0.973

Colour Scale based on p-values : 0 (S > D) .5 1 (S ≤ D)

NR - No Regularization, BN - Batchnorm, SC - Skip Connections, DA - Data Augmentation, L2- weight decay, D - Dense Networks and S - Sparse Networks.

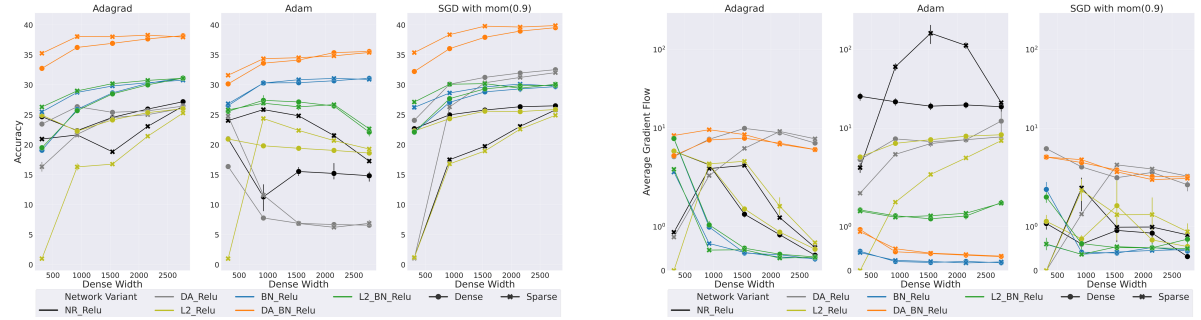
Table 3.3: **Wilcoxon Signed Rank Test Results for MLPs with Four Hidden Layers, Trained on CIFAR-100.** We show the results using different optimization and regularization methods, across various sparsity levels as mentioned in Section 3.3.3. We use a p-value of 0.05, with the bold values indicating where we can be statistically confident that sparse networks perform better than dense (reject H_0 from 3.4). We also use a continuous colour scale to make the results more interpretable. This scale ranges from green (0 - likely that sparse networks perform better than dense) to yellow (0.5 - 50% chance that sparse networks perform better than dense) to red (1 - highly likely that sparse networks do not outperform dense - cannot reject H_0 from 3.4). The performance results for all these networks are present in Appendix A.5.

The poor performance of adaptive methods with $L2$ regularization (specifically EWMA optimizers) agrees with Loshchilov and Hutter [2017]. They proposed a different formulation of weight decay for Adam, named AdamW, since the current $L2$ regularization formulation for adaptive methods could lead to weights with large gradients being regularized less. We experimentally verified this in Figure A.18, showing that the AdamW's weight decay formulation has a lower `EGF` than the standard $L2$ formulation used in Adam, and this correlates to better network performance in AdamW.

Data Augmentation Favours Non-EWMA Optimizers When networks are trained with

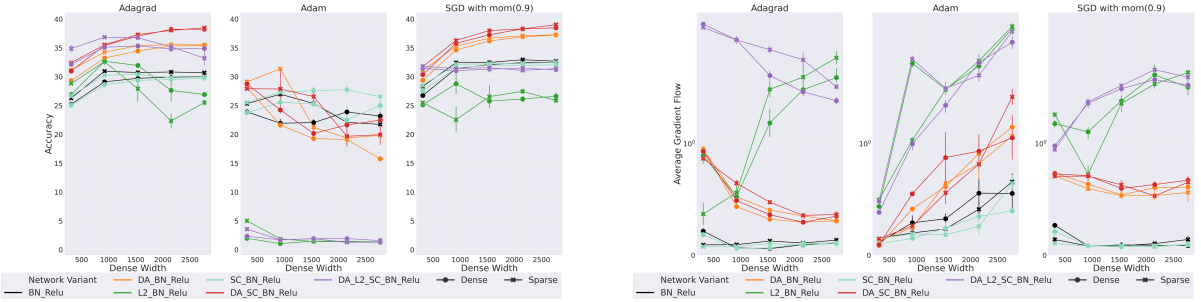
(a) Test Accuracy - Low learning rate (0.001)

(b) Gradient Flow - Low learning rate (0.001)



(c) Test Accuracy - High learning rate (0.1)

(d) Gradient Flow - High learning rate (0.1)



NR - No Regularization, BN - Batchnorm, SC - Skip Connections, DA - Data Augmentation and L2- weight decay.

Figure 3.3: **Test Accuracy and Gradient Flow in Sparse and Dense MLPs.** We study the effect of different regularization and optimization methods on test accuracy and average gradient flow, across different learning rates. We see that for Adam, a higher gradient flow tends to correlate to poor performance. The results for all optimizers can be found in Figures A.14 and A.16.

data augmentation and without BatchNorm (configuration *DA*), we see poor test accuracy for EWMA optimizers across sparse and dense networks (Figure 3.3a). With the addition of BatchNorm (configuration *BN_DA*), when using a low learning rate, we see that data augmentation benefits all optimizers (Figure 3.3a). This behaviour differs when using a high learning rate (Figure 3.3c), where data augmentation results in a higher gradient flow in all optimizers (comparing *BN* to *DA_BN*). This then results in lower performance in EWMA optimizers, further emphasizing the sensitivity of these optimizers to higher gradient flow.

For non-EWMA optimizers, data augmentation consistently increases performance, even with a higher learning rate and higher *EGF*, which suggests data augmentation is better suited for these methods.

EWMA Optimizers Struggle with High Gradient Flow If we take a closer look at the gradient flow, through the average `EGF` (Figures 3.3b and 3.3d), we see that for EWMA optimizers the worst performing variants (*L2*, *NR*, *DA*, *L2_BN* and *DA_L2_SC_BN*) consistently have a relatively high `EGF`, while the best performing interventions (*BN*, *BN_SC* and *DA_BN_SC*) have a lower `EGF`. This is also true of different activation functions, where the worst performing activation function when using Adam, SReLU, also has the highest `EGF` (Figure 3.4).

In non-EWMA optimizers, a higher gradient flow does not consistently lead to poor per-

(a) Test Accuracy (b) Gradient Flow

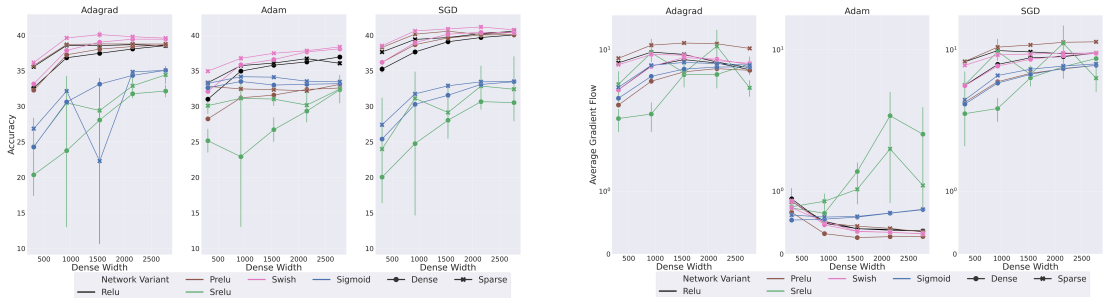


Figure 3.4: **Effect of Activation Functions on Accuracy and Gradient Flow on CIFAR-100, With a Low Learning Rate (0.001).** We see that Swish is the most promising activation function across most optimizers. The results across all optimizers and learning rates are shown in Figure A.15 and A.17.

formance. This hints that the higher effective learning rates of EWMA optimizers can be problematic during training, primarily when used in conjunction with methods that result in high `EGF`. The sensitivity of networks trained with EWMA optimizers, particularly sparse networks, to $L2$ and data augmentation suggests that these methods are not adequate for sparse networks in their current formulation.

3.4.1.3 The Potential of Non-Sparse Activation Functions - Swish and PReLU

We also explore the impact of different activation functions - specifically PReLU [He et al., 2015], ELU [Clevert et al., 2015], Swish [Ramachandran et al., 2017], SReLU [Jin et al., 2015] and Sigmoid [Neal, 1992] - on network performance. The best regularization configuration for each optimizer was chosen.

From Table 3.3c and A.3b, we see that Swish and PReLU consistently favour sparse networks across most optimizers in a statistically significant manner. From the performance results shown in Figures 3.4a, A.15 and A.17, we see that Swish and PReLU are the most promising activation functions, with Swish being the best performing activation function in adaptive methods, and PReLU achieving promising results for networks trained with SGD.

In Appendix A.4, we plot the different activation functions (Figure A.9a) and their gradients (Figure A.9b). We see that although most activation functions, apart from ReLU, are non-sparse (do not have zero gradients), Swish is the only activation function that allows for the flow of negative gradients due to its non-monotonicity. This leads to Swish having a lower, more stable gradient flow (Figures 3.4b, A.15 and A.17), which could explain some of its success, particularly for EWMA optimizers. We continue to see a consistent trend in EWMA methods that higher `EGF` values, for example in SReLU, correspond to poor performance, while promising methods, such as Swish, result in a lower `EGF`.

We also see that the behaviours mentioned in this section are also present in CIFAR-10 and FMNIST (Table A.1 and A.2, Figures A.10 and A.14).

3.4.2 Generalization of Results Across Architecture Types - Wide ResNet-50

We move on from `SC-SDC` and extend our results to more complicated, convolutional architectures. We train Wide ResNet-50 (the WRN-28-10 variant) [Zagoruyko and Komodakis, 2016] on CIFAR-100. We note from Figure 3.5 that most of our results from `SC-SDC` also hold on

Wide ResNet-50, specifically that $L2$ regularization (even with BatchNorm) hurts performance for adaptive methods (Adagrad and Adam) and also results in higher `EGF` values (Figure A.19). Furthermore, we see that Swish is a promising activation function for adaptive methods and leads to lower `EGF` (Figure A.19). Finally, we see that the combination of Swish and AdamW (configuration *Swish (AdamW)*) achieve good performance for Adam, showing that the AdamW (weight decay) results from `SC-SDC` are also consistent in Wide ResNet-50. These results show that the `SC-SDC` results are not constrained to small scale experiments and that they can be used to learn about the dynamics of larger, more complicated networks.

3.4.3 Generalization of Results From Random Pruning to Magnitude Pruning

We briefly validate if some of our results achieved through random pruning, extend to magnitude pruning [Zhu and Gupta, 2017, Han et al., 2015]. For a comparable experimental setting to section 3.4.2, we train dense Wide ResNet-50 for 100 epochs (50% of the training time) and use magnitude pruning to achieve the desired sparsity, and then fine-tune for the remaining 100 epochs. From Figures 3.6a and 3.6b, we see that although it appears magnitude pruned networks are more susceptible to vanishing gradients, these networks behave similarly to randomly pruned networks. In these networks, $L2$ regularization leads to high `EGF` values (*L2_BN* and *DA_L2_SC_BN*), which correlates to poor performance for EWMA optimizers (Adam). This provides evidence that some of the results achieved on random, sparse networks also extend to magnitude pruned networks.

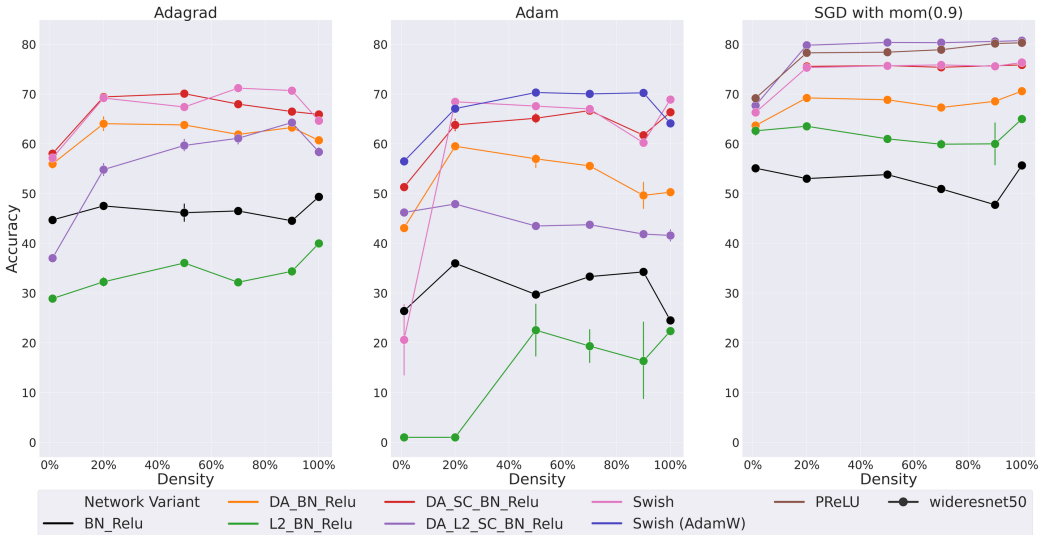


Figure 3.5: **Wide ResNet-50 Test Accuracy on CIFAR-100.** We see that the results achieved on MLPs, using `SC-SDC`, are also consistent in CNNs. The densities range from 1% to 100% (fully dense) and the gradient flow results can be found in Figure A.19.

3.5 Conclusions

In this work, we take a more comprehensive view of sparse optimization strategies and introduce appropriate tooling to measure the impact of architecture and optimization choices on sparse networks (`EGF`, `SC-SDC`).

Our results show that BatchNorm is critical to training sparse networks, more so than for dense networks, as it helps stabilize gradient flow. Furthermore, we show that EWMA optimizers (Adam [Kingma and Ba, 2014] and RMSProp [Hinton et al., 2012]) are sensitive to high

(a) Test Accuracy - Magnitude Pruning (b) Gradient Flow - Magnitude Pruning

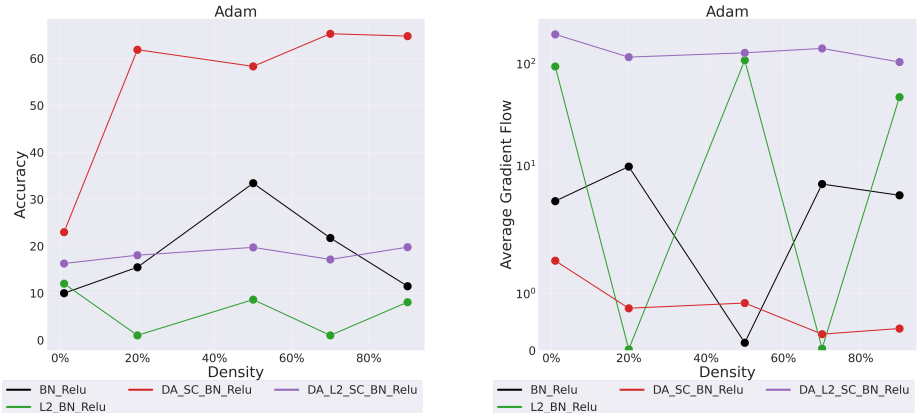


Figure 3.6: **Accuracy and Gradient Flow for Magnitude Pruning.** We see that similarly to randomly pruned networks, magnitude pruned networks trained with Adam and $L2$ lead to high `EGF` and poor performance.

gradient flow (`EGF`). This results in these optimizers, at times, performing poorly when used with $L2$ regularization or data augmentation. We also show the potential of non-sparse activation functions for sparse networks such as Swish [Ramachandran et al., 2017] and PReLU [He et al., 2015], with Swish's non-monotonic formulation allowing for better gradient flow. Finally, we show that our results extend to more complicated models, like Wide ResNet-50 [Zagoruyko and Komodakis, 2016] and popular pruning methods, such as magnitude pruning [Zhu and Gupta, 2017, Han et al., 2015].

We trust that this work emphasizes that initialization is simply one piece of the sparse network puzzle and that a broader view of sparse network training is necessary for the benefits of sparsity to be fully realized. Needless to say, this research is simply the start at investigating sparse network optimization and does not cover all questions related to it. Future directions could include using `EGF` and `SC-SDC` to study other regularization methods, such as Dropout [Srivastava et al., 2014] or $L1$ regularization [Tibshirani, 1996], or other normalization methods, such as Layer Normalization [Ba et al., 2016]. Furthermore, these tools could inspire novel pruning, regularization, or optimization techniques, that are grounded in stabilizing the gradient flow in sparse networks.

Chapter 4

Learning Sparse Neural Network Architectures

4.1 Introduction

Chapter 3 explored sparse network optimization by using and proposing appropriate tooling to analyse sparse networks. In this chapter, we explore how to learn these sparse architectures.

As mentioned in Chapter 1, overparameterized networks have various drawbacks, such as increased training costs [Horowitz, 2014, Strubell et al., 2019], higher latency and memory requirements [Warden and Situnayake, 2019, Samala et al., 2018, Lane and Warden, 2018] and a higher chance of memorization [Zhang et al., 2016, Hooker et al., 2019]. To address these limitations, there has been an increased focus on model compression techniques that reduce the number of model parameters, while maintaining or improving performance.

Network pruning is a popular model compression technique that aims to remove unimportant components from a network's architecture [LeCun et al., 1989]. The typical pruning workflow usually occurs in three stages - training, pruning, and fine-tuning [Liu et al., 2018c]. Firstly, in the training stage, dense, overparameterized models are trained. Then in the pruning phase, weights are ranked and those deemed unimportant are removed. Finally, in the fine-tuning phase, the remaining weights are trained and recalibrated. Even with the success of pruning [Zhu and Gupta, 2017, Han et al., 2015], this initial training phase is a computationally costly process.

Contrary to post-training pruning methods, Frankle and Carbin [2019] proposed the Lottery Ticket Hypothesis (LTH), where they showed that it is possible to train these sparse subnetworks from scratch as long as you have the correct initialization. These initializations, or "lottery tickets", are found by training a model, pruning the weights below a certain threshold and then rewinding the remaining weights to their initial value or a value early in training [Frankle et al., 2019b, Frankle and Carbin, 2019]. Although this work showed that with the correct initialization it is possible to train a subnetwork from scratch to similar performance as the original dense network, it was still necessary to find the correct initialization, which can be computationally costly. Furthermore, finding the correct initialization often requires starting dense, as is the case with iterative magnitude pruning (IMP) [Frankle and Carbin, 2019, Frankle et al., 2019b].

To avoid having to train an overparameterized, dense model first, we use techniques that leverage Neural Architecture Search (NAS) methods to learn sparse architectures from scratch, in a simple and efficient manner. Concretely, we learn the densities (percentage of active weights) of each linear and convolutional layer in a network. Our approach differs from most current NAS methods, which are restricted to learning Convolutional Neural Network (CNN) architectures [Zoph et al., 2018, Real et al., 2019, Baker et al., 2016, Suganuma et al., 2017]. This limits their application to specific problems with structural information such as local structure, which assumes that spatially nearby pixels are correlated [LeCun et al., 1999]. Furthermore, these NAS methods have a large search space since they need to search through hyperparameters such as filter size, stride size, choice of padding, and choice of pooling for each convolutional layer.

Although the number of active weights in each layer is learned, the connections themselves are randomly chosen, meaning we learn random, sparse networks. In parallel work, Su et al. [2020], Frankle et al. [2020] showed that for popular pruning from scratch methods, changing the preserved weights while keeping the number of active weights in each layer constant, does not affect final performance. This hints that for these methods, the layer-wise densities are more critical for performance than the specific active weights. This motivates searching for random, sparse networks, while simply learning their layer-wise densities.

Su et al. [2020] propose "smart-ratios", which are predefined ratios that represent the layer-wise density levels of a network. These ratios are based on heuristics inspired by other pruning from scratch methods (such as a density of 0.3 for linear layers, decaying these ratios for deeper layers or setting these ratios as a function of network depth). This differs from our work as we completely learn these density percentages, leveraging NAS methods. To this end, we propose a flexible, sparse search space (Section 4.3.1) and a simple NAS algorithm for learning sparse architectures - Sparse Neural Architecture Search (`SNAS` - Section 4.3.4).

He et al. [2018] also proposed learning layer-wise density ratios for CNNs, with their AutoML for Model Compression (AMC) algorithm. They train a reinforcement learning (RL) agent (using Deep Deterministic Policy Gradient (DDPG) [Lillicrap et al., 2015]) to predict the sparsity ratios in a network, one layer at a time. Although our work is similar in principle, there are fundamental differences:

- `SNAS` randomly prunes weights at initialization according to a sparsity ratio, while AMC does magnitude pruning according to the sparsity level after training an overparameterized network.
- The choice of a DDPG agent limits AMC's flexibility since it forces AMC to be temporal and learn the sparsity of each layer, one layer at a time, whereas `SNAS` learns the whole network's sparsity at once.
- There are various hyperparameters to set and tune when using a DDPG agent, while our methods have considerably fewer hyperparameters to tune (no hyperparameters for random search and one hyperparameter for Bayesian Optimization).

Using our search space and `SNAS`, our results show that we can consistently learn sparse Multi-layer Perceptrons (MLPs) and sparse Convolutional Neural Networks (CNNs) that outperform their dense counterparts, with considerably fewer weights. Furthermore, we show that the learned architectures are competitive with state-of-the-art architectures and pruning methods. This section aims not to necessarily optimize for best test accuracy using `SNAS`, but rather to show the potential of `SNAS` and our search space, even with naive search methods and random sparsity.

Contributions Our contributions from this chapter are enumerated as follows:

1. **Introduction of a Sparse NAS Space** We propose a flexible NAS search space that can be used to learn sparse neural networks, across various architectures. We provide two variants of this search space, an MLP variant that uses sparse ResNet-like cells to learn chained architectures, and a CNN variant that leverages existing CNN architectures and simply learns the densities across the layers. Due to the flexibility of this search space, it can also be adapted to any architecture that contains linear or convolutional layers such as Recurrent Neural Networks (RNNs) or Transformers [Vaswani et al., 2017].

2. **Sparse Neural Architecture Search (SNAS) Algorithm** We also propose a simple algorithm to learn sparse architectures - Sparse Neural Architecture Search (`SNAS`). We show that `SNAS`, in conjunction with our sparse NAS search space, can be used to consistently learn well-performing architectures, with significantly fewer weights than the original dense network.

This chapter is structured as follows: In Section 4.2, we describe our motivation for learning sparse architectures in a flexible, simple manner. Then in Section 4.3.4, we describe our `SNAS` algorithm and our sparse search spaces. In Section 4.4, we show our empirical results and compare them to other architectures and methods. Having described our methodology and discussed our empirical results, in Section 4.5 we present the limitations of our work. Finally, in Section 4.6, we conclude with our main findings from learning sparse architectures.

4.2 Motivation

We briefly describe our motivation to learn sparse architectures in a simple and flexible manner:

- **Sparsity**: As mentioned in the introduction (Section 4.1), smaller, sparse models are advantageous in terms of memory and training speed, while also at times resulting in better accuracy.

- **Flexibility**: As mentioned in Section 5.3, most NAS approaches focus on learning CNN architectures by learning CNN specific hyperparameters such as filter size, stride size, choice of padding, and choice of pooling. This is problematic because focusing purely on learning CNN architectures assumes some structural relevance in the data, which is not always the case. This restricts these methods to certain domains, such as computer vision. Moreover, this is a relatively large, combinatorial search space, which affects the efficiency of the search.

  To address these issues, we propose learning architectures based on densities (percentage of active weights) in linear and convolutional layers. This simpler approach allows for better flexibility and can be used to learn various types of architectures, such as standard MLPs, CNNs, Recurrent Neural Networks (RNNs), or any architecture type with weight layers (linear or convolutional), with no change to the search space or algorithm.

- **Uniformity**: We formulate our search space based only on the densities of each layer, as opposed to having combinations of different variables (such as filter size or stride length) and kinds of variables (discrete and continuous). This means we have a consistent, uniform search space that can be exploited for a more efficient search process, as discussed in Section 4.3.1.1.

- **Possible Generalization**: Since our learned architecture is represented as a vector of

densities, it is possible to use the same learned architecture and see if it generalizes well to other problems, datasets, or domains. For example, we can learn an architecture on CIFAR-10 [Krizhevsky et al., 2009] and use the same learned density vector on ImageNet [Russakovsky et al., 2015]. This saves computational time since CIFAR-10 is a significantly smaller dataset than ImageNet.

- **Efficiency**: Learning sparse architectures is primarily learning which weight values to set to zero and which ones we allow to stay active. The number of possible combinations of active weights is 2^n, where n is the number of weights in a neural network [1].

  Even with simple networks, the problem of searching all possible combinations of this space proves intractable. For example, if we have a simple neural network as shown in Figure 4.1, with one hidden layer and 24 weights, we have 2^{24} possible combinations of active weights, which is 16 777 216 combinations. Adding one more hidden node results in 1 073 741 824 (2^{30}) different combinations.

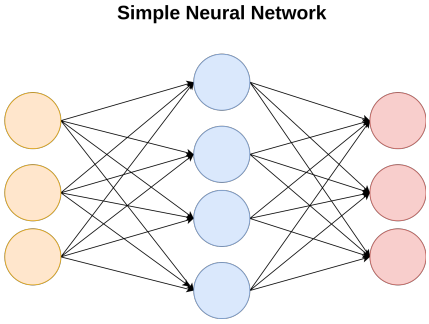


Figure 4.1: **An Illustration of a Simple Neural Network.** We show a simple neural network with 24 weights.

We therefore need to find more efficient ways to search this space. Searching for random, sparse networks, means we can simply learn the number of weights per layer that should be active, as opposed to which specific weights. This drastically reduces our search space.

4.3 Sparse Neural Architecture Search

In this section, we describe how we learn sparse neural architectures. In Section 4.3.1, we describe our simple search space that allows us to learn MLP and CNN architectures. We then briefly discuss our chosen search algorithms/strategies (Section 4.3.2) and our performance estimation techniques (Section 4.3.3). Finally, we describe our complete algorithm for learning sparse architectures (Section 4.3.4).

4.3.1 Search Space

As mentioned in Section 2.8.2, an essential aspect of NAS is the search space as it determines the space of possible architectures that can be learned. One of the main contributions of this research is finding a simple search space for sparse neural networks, that can be used to learn sparse architectures across various network types such as MLPs, CNNs, and RNNs.

Firstly, we discuss our search space representation, both the uniform and non-uniform sparsity variant. Secondly, we discuss and propose two simple search space formulations. The first

[1] Using the fact that a finite set, with n elements, has 2^n subsets, including the empty set [Halmos, 2017]

variant is an MLP search space, which consists of sparse ResNet-like cells that are chained together. The second variant takes predefined convolutional architectures and learns the sparsity levels of each convolutional and linear layer. We use our MLP search space to find architectures for CIFAR-10 and CIFAR-100, while we use our CNN search space to find architectures for CIFAR-100.

4.3.1.1 Search Space Representations

Our search space is simply represented as a vector of densities, where each element of the vector represents the percentage of active weights in a layer. We have two variants of this representation - non-uniform and uniform sparsity.

Non-uniform Sparsity Our non-uniform sparsity representation is described as follows:

$$D = (d_1, d_2, ..., d_L) \in [0, 1]^L \quad , \tag{4.1}$$

where L is the number of layers. For every layer l, our density for that layer, d_l, corresponds to the percentage of active weights in layer l, that is to say, weight matrix θ_l has d_l percentage of active weights compared to a dense/fully-connected layer.

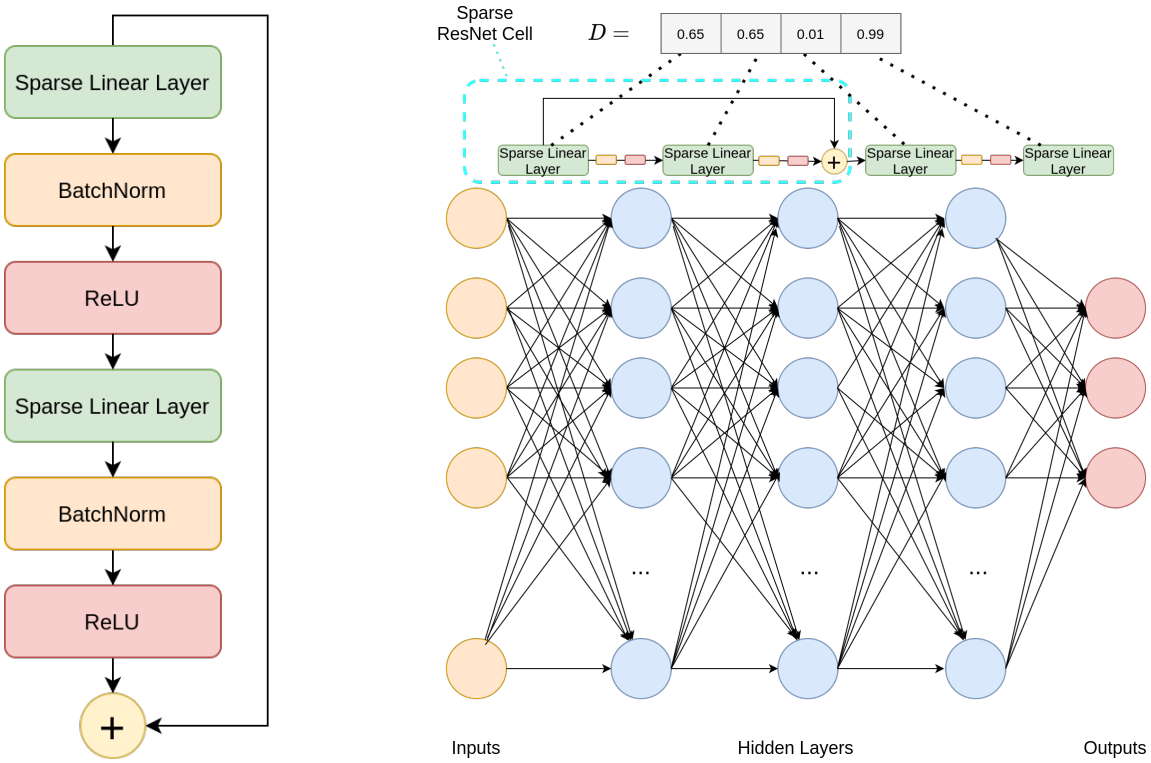
Uniform Sparsity Due to this search space's uniformity (as mentioned in Section 4.2), we also propose a variant of our search space representation where all layers have the same density, which drastically reduces the size of our search space. We refer to this as uniform sparsity. It is represented as a single number/scalar, $d_1 \in [0, 1]$, which corresponds to the density of all the layers (each layer has d_1 percentage of active weights).

4.3.1.2 Search Space Formulations

MLP Search Space Our MLP search space consists of sparse ResNet-like cells, as shown in Figure 4.2a. These cells are inspired from the Residual building blocks from He et al. [2016]. They contain BatchNorm layers and skip-connections (skip-connections every two layers), as these are a robust combination for well-performing neural networks [Labatie, 2019]. Each linear layer has an associated density value learned during the architecture search process. This learned density then translates to a number of active weights, that are randomly selected from that layer. These cells are then chained together to a certain depth specified, as shown in Figure 4.2b.

CNN Search Space We also adapt our sparse search space to work for convolutional architectures, which have the added complexity of containing convolutional layers. Similarly to dense layers, each convolutional layer has a density percentage, representing a percentage of active weights in a particular layer. The only difference between the sparse linear and sparse convolutional layers is that we use a three-dimensional (3D) random mask when masking convolutional layers instead of a two-dimensional (2D) mask for linear layers. This is because we flatten the image for linear layers to convert the input from a matrix to a vector, since linear layers can only handle vectors. While for convolutional layers, we maintain the original input matrix and keep the three image channels (red, green, and blue).

Due to the complexity of designing various convolutional architectures, we take existing architectures, such as ResNet-32 [He et al., 2016] and Wide ResNet-50 [Zagoruyko and Komodakis, 2016], and simply mask each of the convolutional and linear layers. In Figure 4.3, we show how this is done - by learning a density value for each weight layer. This allows our CNN search space to use strong priors that have been built upon through years of computer vision research. Furthermore, due to the simplicity of this approach, we can replace our base CNN architectures with the latest CNN architectures without changing the search space or algorithm.



(a) Sparse ResNet Cell

(b) Sparse ResNet Architecture

Figure 4.2: **MLP Sparse Search Space.** We show our sparse ResNet-like cells and how they can be chained to form Sparse ResNet Architectures. In our Sparse ResNet Architecture diagram, we omit the BatchNorm and ReLU cells for conciseness, and D represents our learned density vector, where d_i is the density of weight layer i.

4.3.2 Search Strategy

NAS search strategies/algorithms determine which populations of architectures get sampled from the search space (see Section 2.8.3). We use random search and Bayesian Optimization (Section 2.8.3.3) as search algorithms, mainly due to their simplicity, but they have also proved to be competitive search algorithms [Li and Talwalkar, 2019, Bergstra and Bengio, 2012, Kandasamy et al., 2018, Snoek et al., 2012].

For our Bayesian Optimization implementation, we use BayesOpt [Nogueira, 2020], an open source Bayesian Optimization library. We do not set any informative priors over the density of the layers (thereby using the default specified by BayesOpt - a random uniform sample in the range [0,1]) and we use Upper Confidence Bound (UCB) [Brochu et al., 2010], with a kappa of 10, as an acquisition function. This high kappa value encourages exploration and proved to be the best performing kappa value while doing a crude search of the hyperparameter space.

4.3.3 Performance Evaluation and Estimation

As mentioned in Section 2.8.4, evaluating the performance of every proposed architecture can be computationally expensive. The simplest performance evaluation method is to train the model and evaluate its performance on a test set, but this is rather inefficient and computationally costly. Some methods reduce training time by using performance estimation techniques (see Section 2.8.4).

For our performance estimation algorithm, we use the Asynchronous Successive Halving Al-

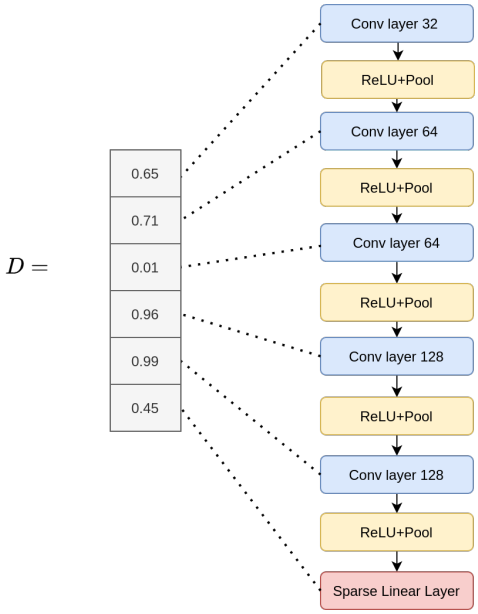


Figure 4.3: **CNN Sparse Search Space.** We show how we can use an existing CNN architecture and learn the densities for each convolutional and linear layer. D represents our learned density vector, where d_i is the density of weight layer i (convolutional or linear layer).

gorithm (ASHA) [Li et al., 2018], implemented in Tune [Liaw et al., 2018], a Python library for distributed hyperparameter search. ASHA aggressively leverages early stopping to quickly terminate poor performing trials. ASHA is an asynchronous version of the Successive Halving Algorithm (SHA) [Karnin et al., 2013, Jamieson and Talwalkar, 2016]. SHA works by initially allocating equal resources to all hyperparameter configurations, then promoting the top 50% of candidate architectures, and then iteratively repeating the process. This use of aggressive early stopping means that even though we might have many samples we want to evaluate, we only fully evaluate the most promising architectures, while quickly stopping the rest.

4.3.4 Sparse Neural Architecture Search (SNAS) Algorithm

In Algorithm 1, we present our complete Sparse Neural Architecture Search (SNAS) algorithm. This is a simple, flexible algorithm that we use to learn the density vector required for our MLP and CNN search spaces.

Algorithm 1: Sparse Neural Architecture Search

Input: Number of samples $\mathbf{S}$, model $\mathbf{m}$, evaluation function $\mathbf{eval}(\mathbf{x})$ and a chosen search algorithm, *search_algorithm* .

1. For $\mathbf{s}$ in $\mathbf{S}$ samples,
   (a) Sample density vector from search algorithm, while passing history of sampled density vectors, $\mathbf{d_{hist}}$, and history of model performance, $\mathbf{score_{hist}}$.
   $\mathbf{d_s} = \textit{search_algorithm}(\mathbf{m}, \mathbf{d_{hist}}, \mathbf{score_{hist}})$
   (b) Apply mask to model $\mathbf{m}$, $\mathbf{m_s} = \textit{mask_model}(\mathbf{d_s}, \mathbf{m})$
   (c) For each candidate architecture, evaluate their performance, $\mathbf{score_s} = \mathbf{eval}(m_s)$
   (d) Update history of performance and density vectors, $\mathbf{d_{hist}}$ and $\mathbf{score_{hist}}$.

Output: Return best model, $\mathbf{m}^*$, and best score, $\mathbf{score}^*$.

As input, our algorithm takes in a specified number of samples $\mathbf{S}$, a chosen model $\mathbf{m}$, an evaluation function $\mathbf{eval}(\mathbf{x})$ (that returns the accuracy of architecture $\mathbf{x}$ after training), and a search algorithm, *search_algorithm*. As mentioned in Section 4.3.2, we use random search and

Bayesian Optimization as search algorithms, but any search algorithm can be used to learn the density vector.

Once we have sampled a density vector, $\mathbf{d_s}$, we mask our model according to this density vector. Then, using the $\mathbf{eval}(x)$ function, we evaluate the performance of our model, $\mathbf{m_s}$. For our $\mathbf{eval}(x)$ function, we use test accuracy, but this could be replaced with any other reasonable performance measure, such as test loss. During execution, we also keep track of the history of density vectors sampled, $\mathbf{d_{hist}}$, and history of model evaluations, $\mathbf{score_{hist}}$. This allows the search algorithm to pick future density vectors based on the performance history.

Finally, for our outputs, we return the best performing model, $\mathbf{m}^*$, and best score, $\mathbf{score}^*$.

4.4 Results

In this section, we discuss the results we achieved using our `SNAS` algorithm to learn sparse MLPs and CNNs. We show that `SNAS` is effective when used with MLPs and CNNs, as it consistently achieves better performance than the dense baselines, while finding smaller, sparser models. Furthermore, we show that the architectures found are competitive with other learned MLP and CNN architectures.

Our experiments are run on both the non-uniform and uniform sparsity representations of our search space. The results shown in this section refer to the best performing architectures from these representations, with layer-wise density levels and other details presented in Appendix B.1.

4.4.1 Experiment Setup

We benchmark our `SNAS` algorithm on both our MLP and CNN search space. We briefly describe some of the key components of our experiment setup below and we also include for completeness, our full MLP configuration in Table 4.1a and our full CNN configuration in Table 4.1b.

Configuration	Variants
Optimizer	SGD with mom (0.9).
Number of Hidden Layers	2,4,8,16 and 32
Width	3076
Activation functions	PReLU
Epochs	500
Learning rate	0.01 (decayed by 10 at epoch 250, 375)
Datasets	CIFAR-10 and CIFAR-100.
Number of Samples	50
Architecture Choices	l2(0.0005), Dropout, Skip Connections, Batch Normalization, Data Augmentation

(a) MLP Configuration

Configuration	Variants
Optimizer	SGD with mom (0.9) and AdamW (0.005 - l2).
Architecture	ResNet-32 and Wide ResNet-50
Activation functions	ReLU
Epochs	200
Learning rate	0.1 (decayed by 20 at epoch 60, 120, 160)
Datasets	CIFAR-100.
Number of Samples	25
Architecture Choices	l2(0.0005), Dropout, Skip Connections, Batch Normalization, Data Augmentation

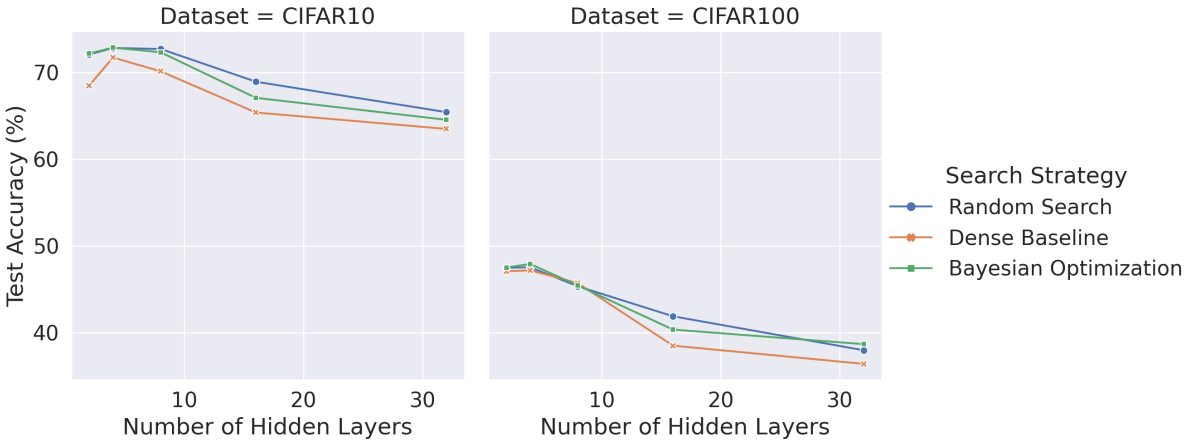
(b) CNN Configuration

Table 4.1: **Network Configurations.** Different network configurations used in `SNAS`.

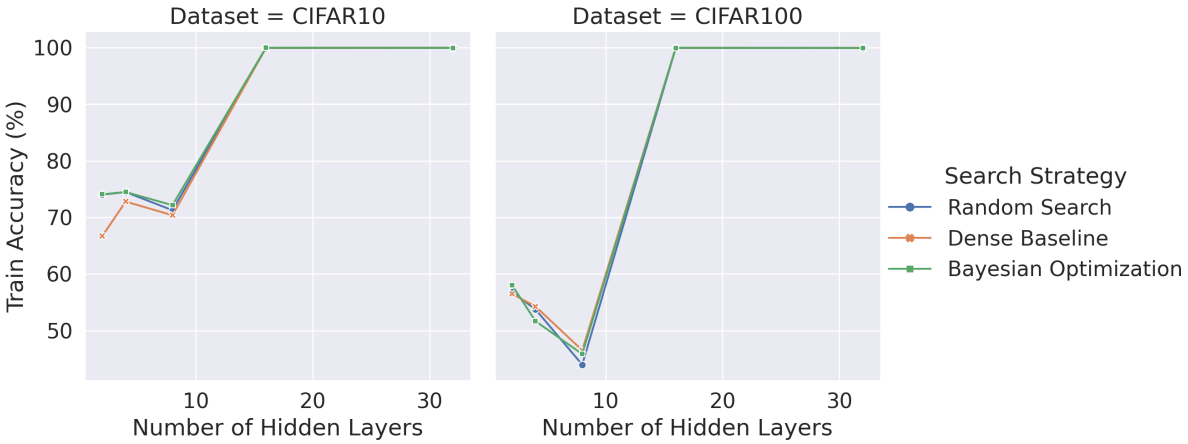
For our MLP search space, we sample 50 architectures ($\mathbf{S} = 50$), our chosen models, $\mathbf{m}$,

are MLPs with a varying number of hidden layers - $2, 4, 8, 16$ and 32, while using a width of 3076. This width was chosen because Lu et al. [2017] proved a universal approximation theorem for width-bounded ReLU networks, with width bounded to $n + 4$, where n is the input dimension ($n = 3072$ in the case of CIFAR datasets). These networks are trained with minibatch stochastic gradient descent (SGD) with momentum (momentum term is 0.9). The search space is composed of the sparse ResNet-like cells as described in Section 4.3.1, with the addition of L2 regularization, data augmentation and dropout. For deeper architectures (16 and 32 hidden layers), we train these models without dropout, as experimentally dropout led to a decrease in accuracy and it has been shown to limit the trainable depth of neural networks [Schoenholz et al., 2016]. For our activation function, we use PReLU [He et al., 2015], because in Section 3.4.1 we showed that PReLU is a promising activation function when training models with SGD.

In our CNN search space, we sample 25 architectures ($\mathbf{S} = 25$), while using ResNet-32 [He et al., 2016] and Wide ResNet-50 [Zagoruyko and Komodakis, 2016] as the base architectures, $\mathbf{m}$. We train these networks using SGD with momentum and AdamW [Loshchilov and Hutter, 2017], a variant of Adam [Kingma and Ba, 2014] with weight decay decoupled from gradient updates.



(a) Test Accuracy



(b) Train Accuracy

Figure 4.4: **Test and Train Accuracy of SNAS MLPs Across Various Depths.** We show the test and train accuracy of the best MLPs learned at different depths, compared to the dense baselines. We see that across all depths, `SNAS` consistently finds better performing MLP architectures than the dense baselines.

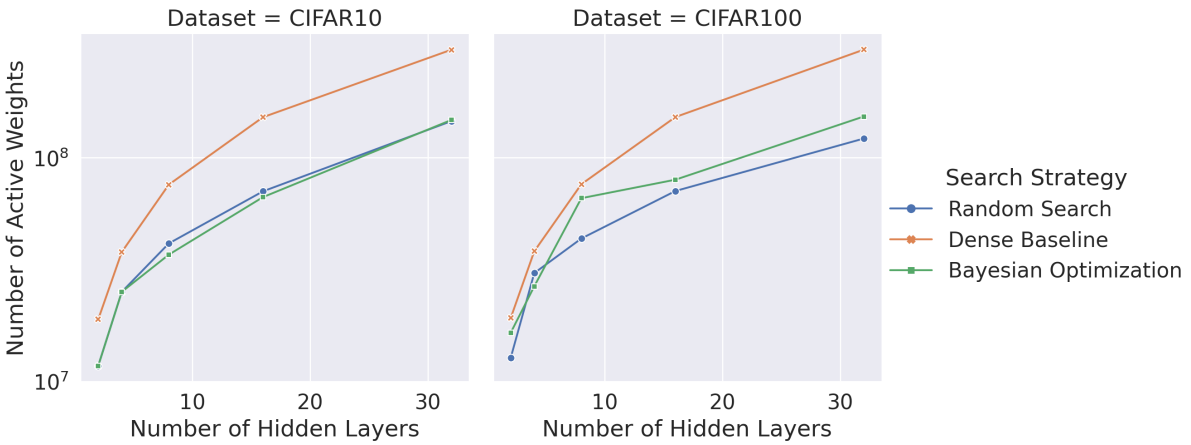


Figure 4.5: **Number of Active Weights in SNAS MLPs.** We compare the number of active weights in the best performing learned MLPs to their dense counterparts. We see that `SNAS` consistently finds much smaller architectures, especially for deeper networks.

4.4.2 Searching for MLP Architectures

In this section, we explore the effectiveness of our `SNAS` algorithm at learning sparse MLP architectures.

4.4.2.1 In Terms of Best Accuracy, Random Search Performs Similarly to Bayesian Optimization

When comparing the best accuracy found ($score_*$) with a limited amount of samples, we see that random search and Bayesian Optimization achieve similar performance across most depths (Figure 4.4a). At times, random search even outperforms Bayesian Optimization, notably at a depth of 16 hidden layers, where random search has a 1.86% higher test accuracy on CIFAR-10, and 1.53% higher test accuracy on CIFAR-100. This agrees with Li and Talwalkar [2019], who showed that random search is a competitive NAS baseline compared to other search algorithms.

Although in terms of best search accuracy, random search is competitive to Bayesian Optimization, looking at mean test accuracy or other measures of search efficiency might prove to favour Bayesian Optimization. In our current setting, due to aggressive early stopping, we cannot fairly compare the mean test accuracy of these methods. Furthermore, Bayesian Optimization requires the tuning of certain hyperparameters. These hyperparameters include setting the prior, which in our case refers to the prior over the layer-wise densities, and setting an exploration strategy. Bayesian Optimization methods also often rely on custom kernels [Jin et al., 2019, Kandasamy et al., 2018] to perform well. It is possible that with better tuning of these hyperparameters, Bayesian Optimization could perform better than random search for our search space.

4.4.2.2 Shallower MLPs Outperform Deeper Ones

When looking at test accuracy (Figure 4.4a), we see that the best performing architectures for CIFAR-10 and CIFAR-100 are MLPs with two or four hidden layers. Across both datasets, deeper architectures perform worse than shallow ones. When we look at the train accuracy in Figure 4.4b, we see that deeper networks are overfitting and memorizing the training data since they achieve 100% train accuracy. Furthermore, these networks have a high generalization gap since their test accuracy is much lower than their train accuracy. Although dropout is a

common technique to reduce overfitting [Srivastava et al., 2014], it has been shown to limit the trainable depth of neural networks [Schoenholz et al., 2016]. In our case, it empirically hurt the test accuracy of these deeper networks and hence it was not applied in our final implementation for 16 and 32 hidden layers. This motivates for the development of regularization techniques that are more catered for deep, wide MLPs.

4.4.2.3 SNAS Consistently Finds Smaller, yet Better Performing Sparse Architectures

From Figure 4.4a, we see across all depths on CIFAR-10 and CIFAR-100, that `SNAS` consistently finds better performing architectures than the standard dense configuration. This performance improvement peaks at a 3.69% absolute test accuracy increase, which occurs with 2 and 16 hidden layers on CIFAR-10, and 16 hidden layers for CIFAR-100.

Not only does `SNAS` find better performing architectures, but the sparse architectures found are also substantially smaller than their dense counterparts (Figure 4.5). This is especially apparent in deeper networks (16 and 32 hidden layers), where we see that the networks found have between 50% and 60% fewer weights than their dense counterparts, while achieving better performance.

4.4.2.4 SNAS Produces Competitive MLP Architectures

Although the goal of this section was not necessarily to optimize training accuracy using `SNAS`, but rather to show the flexibility of the search space and `SNAS` algorithm, we compare `SNAS` to state-of-the-art (SOTA) MLP architectures in Table 4.2.

From a performance perspective, `SNAS` does not achieve SOTA performance, but we see that the architectures learned by `SNAS` are competitive when compared to SOTA MLP architectures. The Z-LIN Network [Lin et al., 2015] and Student-Teacher MLP [Urban et al., 2016] outperform our models, but these methods require pre-training. Mocanu et al. [2018] train competitive MLPs, but they use SET (Sparse Evolutionary Training), a dynamic sparsity approach that adapts the active weights during training, as opposed to standard SGD. Neyshabur [2020] train the best performing MLPs according to our knowledge, but these models are trained for 4000 epochs (eight times the training period used in `SNAS`), while the best performing architecture is trained with β-lasso, which is a variant of Lasso regression with soft-thresholding.

From a model size perspective (number of parameters), we see that the architectures found by `SNAS` are smaller than all other MLP architectures that were trained with SGD. However, SET MLPs [Mocanu et al., 2018] and MLPs trained with β-lasso [Neyshabur, 2020] have considerably fewer parameters because their optimization procedure encourages smaller models.

Although in terms of performance and model size, `SNAS` does not outperform SOTA, `SNAS` is competitive, especially when we consider that `SNAS` requires no pre-training, it is trained with conventional SGD, and each architecture is only trained for 500 epochs. In future work, `SNAS` could be used to rather learn the architectures of MLPs trained with SET or β-lasso. Furthermore, we can adapt `SNAS` to find sparser architectures by setting the objective function ($score_s$ in Algorithm 1) to be a combination of performance and model size, as opposed to only performance (test accuracy).

4.4.3 Searching for CNN Architectures

In this section, we move on from MLPs and evaluate our `SNAS` algorithm's effectiveness at learning CNN architectures.

Architecture	Training		Test Accuracy		# Params (M)	
	Optimization	Epochs	CIFAR-10	CIFAR-100	CIFAR-10	CIFAR-100
MLP [Neyshabur, 2020]	SGD.	400	72.77	47.72	256.00	256.00
MLP (SNAS - Ours)	SGD.	500	72.84	47.92	22.4.	25
MLP [Urban et al., 2016]	SGD. Pre-trained convolutional teacher network.	-	74.30	-	31.60	-
Z-LIN Network [Lin et al., 2015]	SGD. Pre-trained autoencoder layers.	1000	78.62	-	112.00	-
MLP [Mocanu et al., 2018]	SET (Sparse Evolutionary Training).	1000	78.84	-	0.30	-
MLP [Neyshabur, 2020]	SGD.	4000	78.63	51.43	256.00	256.00
MLP [Neyshabur, 2020]	β-lasso.	4000	**85.19**	**59.56**	10	11

Table 4.2: **Best SNAS MLP Compared to SOTA MLPs.** We see that `SNAS` produces competitive architectures compared to SOTA MLP methods, while learning the smallest models for MLPs trained with SGD.

4.4.3.1 SNAS Is Effective at Learning CNN Architectures

From Table 4.3, we see that `SNAS` is also effective when using a CNN search space. It achieves better performing CNNs across the different CNN architectures and across networks trained with AdamW and SGD. For ResNet-32, the absolute performance increase is approximately 0.5% across SGD and AdamW. For Wide ResNet-50, we see a 0.10% increase when trained with SGD and a 0.88% increase when trained with AdamW. Along with better performance, these models have from 5% up to 40% fewer weights than their dense counterparts.

Although we see a performance increase when learning CNN architectures, this increase is not as drastic as the increase we saw when learning sparse MLPs. We believe that this can be attributed to the fact that in these CNN architectures, all learned sparse layers are convolutional layers except for the final layer. Due to weight sharing in convolutional layers (see Section 2.2.4), these layers already have drastically fewer parameters than linear layers. Furthermore, convolutional layers already employ structured sparsity due to local connectivity (see Section 2.2.4). It appears that sparsity is more beneficial when there are more weights available and there is no form of prior sparsity, as is the case with linear layers. These two factors could be why `SNAS` appears to be more effective for improving MLP architecture performance than for CNNs.

4.4.3.2 SNAS Is Competitive to Other Pruning Methods

In Table 4.4, we show the best `SNAS` architecture compared to other pruning methods. This is not a fair comparison since other pruning methods are constrained to a specific sparsity (for example, 90% sparsity or conversely, 10% density), while `SNAS` learns the sparsity without this constraint. However, we show that `SNAS` finds architectures that are competitive relative to other pruning methods. In the future, constrained versions of `SNAS` could be formulated, where the maximum density can be set and we can more fairly compare it to other pruning methods.

	ResNet-32				**Wide ResNet-50**			
Optimization	SGD		AdamW		SGD		AdamW	
	Accuracy	Sparsity	Accuracy	Sparsity	Accuracy	Sparsity	Accuracy	Sparsity
Dense	75.09	0.00	69.18	0.00	81.25	0.00	73.88	0.00
SNAS (Ours)	**75.63**	13.38	**69.71**	40.13	**81.34**	13.38	**74.76**	4.93

Table 4.3: **SNAS Learning Sparse CNN Architectures.** `SNAS` finds smaller, yet better performing ResNet-32 and Wide ResNet-50 architectures.

Pruning Method	ResNet-32	
	Accuracy	Sparsity
LT [Frankle and Carbin, 2019]	69.6 ± 0.26	90
SNIP [Lee et al., 2018]	69.97 ± 0.17	90
GraSP [Wang et al., 2020a]	70.12 ± 0.15	90
Random Tickets [Su et al., 2020]	69.70 ± 0.48	90
SNAS (Ours)	**75.63**	13.38

Table 4.4: **SNAS Compared to Other Pruning Methods.** We compare `SNAS` to other pruning methods. Although this is not a fair comparison, since other pruning methods are constrained to a specific sparsity level, we see that `SNAS` achieves competitive performance. These performance metrics are retrieved from Su et al. [2020].

4.5 Limitations

Our technique for learning sparse architectures is simple: learn the layer-wise density levels in a network. However, our approach has much room for improvement.

Firstly, in the current setting, our learned densities cannot be constrained. For example, limiting the maximum density of a network or layer to a fixed value, such as 0.1 (a maximum of only 10% of the weights can be active), is not currently possible. Incorporating a restriction to the algorithm will allow for a fair comparison to other pruning methods and better use in practice, since the maximum parameter budget can be set.

Another concern is that our MLP and CNN search space relies on manually configured architectural priors. We manually specify the desired depth for our MLP search space, which constrains the learned architectures to that depth. At the same time, our CNN search space relies on previously established CNN architectures as base architectures, limiting the scope of architectures that can be learned. Although these manual decisions provide a strong architectural prior for these search spaces, it could be beneficial to allow more diverse architectures to be learned.

Finally, our last and possibly most critical concern is efficiency. Although we do not have to train an overparameterized, dense network, we often have to sample numerous sparse networks to find well-performing sparse architectures. The number of samples can be set in `SNAS`, and we use aggressive early stopping. This means that we only fully train the most promising architectures, but our search is still a computationally costly process. To improve efficiency, more sophisticated performance estimation techniques should be used (see Section 2.8.4), in conjunction with more advanced search strategies (see Section 2.8.3).

4.6 Conclusion

This chapter showed that our `SNAS` algorithm and our NAS search spaces, both MLP and CNN variants, are effective for learning sparse MLP and CNN architectures. The learned architectures consistently perform better than their dense counterparts, while having considerably fewer weights. This is the case even with naive search algorithms and random sparsity. Moreover, we show that the learned architectures are competitive with state-of-the-art architectures and pruning methods. Apart from the empirical performance improvements, we have also shown that `SNAS` provides a flexible and simple way to learn sparse architectures, which can be used for most architecture types and search algorithm choices.

This work emphasizes that simple, flexible methods for learning sparse architectures have potential. As mentioned in Section 4.5, there are various limitations to our approach, which could be improved upon by future work, namely, using more advanced NAS methods could further improve our search efficiency. It is also possible that looking at different kinds of sparse connections, as opposed to random ones, could be beneficial, for example, learning Erdős–Rényi random connections [Erdős and Rényi, 1960] as was done by Xie et al. [2019]. We could also use `SNAS` to learn networks trained with promising optimization methods, such as SET (Sparse Evolutionary Training) [Mocanu et al., 2018] or β-lasso [Neyshabur, 2020].

Chapter 5

Related Work

5.1 Introduction

The field of sparsity in machine learning is vast and encompasses various compression and pruning techniques. In this chapter, we discuss work related to our study of sparse network training dynamics (Chapter 3) and the learning of sparse architectures (Chapter 4). Notably, we discuss pruning from scratch methods (Section 5.2), Neural Architecture Search (NAS) for sparse networks (Section 5.3), and approaches to studying sparse network dynamics (Section 5.4).

5.2 Pruning From Scratch

As mentioned in Section 2.7.1.2, pruning from scratch methods use information from the beginning or early in training to estimate which network components should be removed. In this section, we discuss pruning from scratch methods and distinguish our Sparse Neural Architecture Search (SNAS) algorithm (Chapter 4) from these methods.

5.2.1 Connection Sensitivity

One of the first successful implementations of pruning from scratch methods was Single-shot Network Pruning based on Connection Sensitivity (SNIP) [Lee et al., 2018]. The authors proposed SNIP, a pruning method that uses a saliency criterion that measures the sensitivity of connections by looking at their influence on a network's loss, regardless of whether the influence is positive or negative. Due to the disregard of the sign of the influence, this criterion is not dependent on the actual loss value, thereby removing the need for pre-training a dense network before pruning. This differs from previous methods [Mozer and Smolensky, 1989, Karnin, 1990], which relied on finding weights/neurons that resulted in the smallest decrease in performance, which required pre-training.

Wang et al. [2020a] noted that since SNIP independently considers the gradients of each weight, it could remove weights that are critical to gradient flow in the network. They show at high sparsity levels, SNIP removes nearly all the weights in certain layers, making gradient propagation an issue. To address this, the authors propose Gradient Signal Preservation (GraSP), a pruning criteria based on the first-order approximation of the change in the gradient norm if a specific weight was pruned. GRASP aims to maintain or improve gradient flow, by first removing weights whose removal will not reduce gradient flow/gradient norm.

Tanaka et al. [2020] showed that gradient-based pruning methods like GRASP and SNIP, disproportionately prune large layers and are susceptible to layer-collapse, which is when an algorithm prunes all the weights in a specific layer. They propose Iterative Synaptic Flow Pruning (SynFlow), a pruning algorithm that provably avoids layer-collapse and achieves state-of-the-art pruning performance. Another pruning from scratch method was proposed by Liu and Zenke [2020], where they used the Neural Tangent Kernel (NTK) to characterize sparse training dynamics and bring these dynamics closer to dense training dynamics. Their resulting Neural Tangent Transfer (NTT) initialization showed performance improvements over random initialization and SNIP, but the calculation of NTK matrices remains computationally expensive.

Pruning methods based on connection sensitivity differ from our work as we learn random sparse networks using NAS, without any gradient or connection-specific information.

5.2.2 Random Pruning

SNIP, GRASP and SynFlow use different forms of connection sensitivity as pruning criteria. These methods search for the specific, essential weights that need to remain active, while pruning unimportant weights. Another branch of pruning is random pruning, where the connections that need to remain active are randomly chosen. Although random pruning has had some success [Changpinyo et al., 2017, Mittal et al., 2018], it was often believed to be inferior to other more informed methods and was mostly used as a pruning baseline.

Contrary to this, Su et al. [2020] and Frankle et al. [2020] showed that for pruning from scratch methods (SNIP, GRASP and SynFlow), shuffling the preserved weights, while keeping the number of active weights in each layer constant, does not affect final performance. This motivates for rather learning the layer-wise densities (number of active connections per layer), as opposed to the specific important weights as is commonly done in pruning methods. Su et al. [2020] propose "smart ratios", which are ratios that represent the layer-wise density levels of a network. These ratios are based on predefined heuristics and schedules inspired by other pruning from scratch methods, such as a density of 0.3 for linear layers or decaying these ratios as a function of network depth. This differs from our work as we completely learn these density percentages, leveraging NAS methods, without any hardcoded layer densities.

5.3 Neural Architecture Search for Sparse Networks

Neural Architecture Search (NAS) is the process of automating the design of neural network architectures [Elsken et al., 2018b] (see Section 2.8). NAS methods augment network topology by using various search methods, while pruning methods alter a network's architecture by removing network units (weights or neurons) that are deemed irrelevant (often based on a component's influence on the loss or a sensitivity criteria). Pruning methods have also been considered a generalization of NAS [Hoefler et al., 2021].

NAS methods have been successful at learning CNN architectures [Zoph et al., 2018, Zoph and Le, 2016, Real et al., 2019, Baker et al., 2016, Suganuma et al., 2017, Liu et al., 2018a,b], with the current state-of-the-art ImageNet architecture, EfficientNet [Tan and Le, 2019], having leveraged NAS. Although these methods have proved to be successful, they focus disproportionately on learning CNN architectures and their hyperparameters (such as kernel/stride size), whereas we propose `SNAS` and our search space as a general approach to learning sparse networks, across a wide variety of architectures.

There has been work that has combined NAS and pruning approaches [Fedorov et al., 2019,

Noy et al., 2020, Wu et al., 2020, Wang et al., 2020b]. However, these methods structure architectures as a directed graph and focus on pruning of structural components, as opposed to pruning weights or neurons of a network. For example, Xie et al. [2019] used directed graphs to learn randomly wired CNNs, where random connections are between nodes consisting of ReLU, convolution and BatchNorm operations, while each edge is just a flow of data between nodes. As opposed to using NAS and pruning network topology, Dong and Yang [2019] use NAS to search for the appropriate depth and width for a pruned network.

The work of He et al. [2018] is the closest related work to our `SNAS` algorithm and search space (Chapter 4). They propose AutoML for Model Compression (AMC), where they train a reinforcement learning (RL) agent (using Deep Deterministic Policy Gradient (DDPG) [Lillicrap et al., 2015]) to predict the sparsity ratios in a network. This is done one layer at a time, where the agent takes layer t's embedding s_t and outputs a sparsity ratio a_t. Then, according to a_t, the agent prunes the smallest a_t percent of weights in layer t (according to their magnitude). The agent then moves on to the next layer $t+1$ and receives the next state s_{t+1}. Finally, after going through all the layers in the network, the validation accuracy and FLOPs are returned as a reward.

Although our work is similar in principle to He et al. [2018], there are two fundamental differences. Firstly, `SNAS` randomly prunes weights at initialization according to a sparsity ratio, while AMC does magnitude pruning according to the sparsity level *after* training an overparameterized network. The random pruning removes the need to pre-train a dense network. Secondly, the search strategy is fundamentally different. AMC uses a DDPG agent (RL), while `SNAS` currently uses random search and Bayesian Optimization (and can be adapted to use most search algorithms). The choice of a DDPG agent limits AMC's flexibility, since it forces AMC to be temporal and learn the sparsity of each layer, one layer at a time, while `SNAS` learns the sparsity of the whole network at once. Furthermore, the choice of DDPG means multiple hyperparameters have to be tuned and architectural decisions made, such as the architecture of the actor and critic network, batch size, number of episodes, replay buffer configuration, exploration vs exploitation trade-off, and other RL specific configuration decisions. In contrast, we do not have to tune any hyperparameters when using random search and only have to tune kappa (determines the exploration vs exploitation trade-off) when using Bayesian Optimization.

5.4 Sparse Network Dynamics

5.4.1 Sparse Network Optimization as Pruning Criteria

Optimization in sparse networks has often been neglected in favour of studying network initialization. However, there has been work that has looked at sparse network optimization from different perspectives, mainly as a guide for pruning criteria. This includes using gradient information [Mozer and Smolensky, 1989, LeCun et al., 1989, Hassibi et al., 1993a, Karnin, 1990], approximates of gradient flow [Wang et al., 2020a, Dettmers and Zettlemoyer, 2019, Evci et al., 2020], and Neural Tangent Kernel (NTK) [Liu and Zenke, 2020] to guide the introduction of sparsity.

5.4.2 Sparse Network Optimization to Study Network Dynamics

Apart from being used as pruning criteria, optimization information has been used to investigate aspects of sparse networks, such as their loss landscape [Evci et al., 2019], how they are impacted by SGD noise [Frankle et al., 2019a], the effect of different activation functions [Dubowski, 2020] and their weight initialization [Lee et al., 2019]. Our work differs from these approaches as we consider more aspects of the optimization and regularization process in a controlled experimental

setting — *Same Capacity Sparse vs Dense Comparison* (`SC-SDC`), while using *Effective Gradient Flow* (`EGF`) to reason about some of the results.

5.5 Conclusion

While our work is motivated by the same goal as most pruning and sparsity-related work — achieving well-performing, sparse networks — our approaches to achieve this are different.

As opposed to most pruning methods which use pruning criteria dependent on pre-training a dense network, we use NAS to learn the layer-wise sparsity ratios of random networks at initialization. Our work also differs from sparse NAS approaches, specifically the work of He et al. [2018] as they learn sparsity ratios for magnitude pruning (requiring pre-training of a dense network). Furthermore, they use RL to learn these ratios, one layer at a time, while we learn the ratios for the whole architecture at once, using simpler search methods.

In the context of sparse training dynamics, to our knowledge, our work is the first to take a broader view of sparse training dynamics, considering various aspects of training such as learning rates, architectures, optimization methods, regularization methods, and activation functions.

Chapter 6

Conclusions and Future Work

Sparse neural networks form a critical aspect of modern deep learning. As models get deeper and datasets get larger, sparsity provides a mechanism for designing better networks. These networks can have reduced memory requirements, better computational efficiency, and faster training and inference times, while maintaining or even improving their performance compared to dense networks. Even with the various possible benefits of sparse networks, they are poorly understood. Therefore, in this research report, we aim to aid in the understanding of sparse networks. This was done by looking at two critical aspects of these networks - their training dynamics, and how to learn their architectures. In both circumstances, we provide new tooling or frameworks that contribute to the landscape of methods used for sparse networks.

Firstly, in Chapter 3, we took a more expansive view of sparse optimization strategies and introduced appropriate tooling to measure the impact of architecture and optimization choices on sparse networks — Effective Gradient Flow (`EGF`) and Same Capacity Sparse vs Dense Comparison (`SC-SDC`). Our results showed that batch normalization (BatchNorm) is critical to training sparse networks, more so than for dense networks, as it helps stabilize gradient flow. We also showed that optimizers that use an exponentially weighted moving average when estimating the variance of the gradient (EWMA optimizers - Adam [Kingma and Ba, 2014] and RMSProp [Hinton et al., 2012]) are sensitive to high gradient flow (`EGF`). This results in these optimizers, at times, performing poorly when used with $L2$ regularization or data augmentation. Additionally, we showed the potential of non-sparse activation functions, such as Swish [Ramachandran et al., 2017] and PReLU [He et al., 2015], with Swish's non-monotonic formulation allowing for better gradient flow. Finally, we extended our results to more complicated models, like Wide ResNet-50 [Zagoruyko and Komodakis, 2016], and to popular pruning methods, such as magnitude pruning [Zhu and Gupta, 2017, Han et al., 2015].

Secondly, in Chapter 4, we leverage Neural Architecture Search (NAS) approaches to learn sparse architectures. We propose a simple, sparse architecture search algorithm, Sparse Neural Architecture Search (`SNAS`), and a new NAS search space. Using our `SNAS` algorithm and our NAS search space, we effectively learned sparse Multilayer Perceptron (MLP) and Convolutional Neural Network (CNN) architectures. These architectures consistently outperformed their dense variants, while having considerably fewer weights. Furthermore, the learned architectures were competitive with state-of-the-art architectures and pruning methods. We also ensured that our approach was flexible so that the same methods could be used for most architecture types and search algorithm choices.

Our work on sparse training dynamics and learning of sparse architectures gives some insight into these architectures. Naturally, this work is simply the start at understanding and improving sparse architectures. There is much room for future work. In terms of studying sparse training dynamics, possible avenues of future work could include using `EGF` and `SC-SDC` to study other regularization methods, such as Dropout [Srivastava et al., 2014] or $L1$ regularization [Tibshirani, 1996], or other normalization methods, such as Layer Normalization [Ba et al., 2016]. Furthermore, these tools could inspire novel pruning, regularization, or optimization techniques, that are grounded in stabilizing the gradient flow in sparse networks.

Our work on learning sparse architectures also has many avenues for future work. We could formulate our search space in a manner that allows it to be constrained to a specific density. Furthermore, we could improve the diversity of learnable architectures, by learning the depth in our MLP search space and learning the full CNN architecture in our CNN search space, as opposed to manually configuring these options. We also suspect that leveraging more sophisticated NAS methods, such as choosing more advanced search algorithms or performance estimation techniques, could vastly improve our search efficiency and performance results.

In conclusion, these are exciting times for sparse networks. Once methods of initialization, training and learning of sparse architectures have matured, combined with promising work in sparse matrix calculations and storage [Zhao et al., 2018, Merrill and Garland, 2016, Ma et al., 2019, Zhang et al., 2019b, 2020], there will be vast opportunities to further scale deep learning and make promising scientific progress.

Bibliography

Abramowitz, M., Stegun, I. A., et al. (1972). *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*, volume 55. National bureau of standards Washington, DC.

Ahmad, S. and Scheinkman, L. (2019). How can we be so dense? the benefits of using highly sparse representations. *arXiv preprint arXiv:1903.11257*.

Amodei, D., Hernandez, D., Sastry, G., Clark, J., Brockman, G., and Sutskever, I. (2018). Ai and compute.

Andrychowicz, O. M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., et al. (2020). Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, 5(1):54–65.

Azarian, K., Bhalgat, Y., Lee, J., and Blankevoort, T. (2020). Learned threshold pruning. *arXiv preprint arXiv:2003.00075*.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Baker, B., Gupta, O., Naik, N., and Raskar, R. (2016). Designing neural network architectures using reinforcement learning. *CoRR*, abs/1611.02167.

Balduzzi, D., Frean, M., Leary, L., Lewis, J., Ma, K. W.-D., and McWilliams, B. (2017). The shattered gradients problem: If resnets are the answer, then what is the question? *arXiv preprint arXiv:1702.08591*.

Barlow, H. B. (1972). Single units and sensation: a neuron doctrine for perceptual psychology? *Perception*, 1(4):371–394.

Bellec, G., Kappel, D., Maass, W., and Legenstein, R. (2017). Deep rewiring: Training very sparse deep networks. *arXiv preprint arXiv:1711.05136*.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2).

Bourely, A., Boueri, J. P., and Choromonski, K. (2017). Sparse neural networks topologies.

arXiv preprint arXiv:1706.05683.

Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCand lish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language Models are Few-Shot Learners. *arXiv e-prints*.

Brownlee, J. (2019). *Deep Learning for Computer Vision: Image Classification, Object Detection, and Face Recognition in Python*. Machine Learning Mastery.

Buluç, A., Fineman, J. T., Frigo, M., Gilbert, J. R., and Leiserson, C. E. (2009). Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244.

Carreira-Perpinán, M. A. and Idelbayev, Y. (2018). "learning-compression" algorithms for neural net pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8532–8541.

Changpinyo, S., Sandler, M., and Zhmoginov, A. (2017). The power of sparsity in convolutional neural networks. *arXiv preprint arXiv:1702.06257*.

Chauvin, Y. (1988). A back-propagation algorithm with optimal use of hidden units. In *Proceedings of the 1st International Conference on Neural Information Processing Systems*, pages 519–526.

Chen, Z., Badrinarayanan, V., Lee, C.-Y., and Rabinovich, A. (2018). Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning*, pages 794–803. PMLR.

Chrabaszcz, P., Loshchilov, I., and Hutter, F. (2017). A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv preprint arXiv:1707.08819*.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*.

Collins, M. D. and Kohli, P. (2014). Memory bounded deep convolutional networks. *arXiv preprint arXiv:1412.1442*.

Cun, Y. L., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann.

Deepmind (2020). Alphafold : a solution to a 50-year-old grand challenge in biology. https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology. Accessed: 2021-02-17.

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30.

Dettmers, T. and Zettlemoyer, L. (2019). Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*.

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

Ding, X., Ding, G., Zhou, X., Guo, Y., Han, J., and Liu, J. (2019). Global sparse momentum sgd for pruning very deep neural networks. *arXiv preprint arXiv:1909.12778*.

Dong, X. and Yang, Y. (2019). Network pruning via transformable architecture search. *arXiv preprint arXiv:1905.09717*.

Dubowski, A. (2020). Activation function impact on sparse neural networks. B.S. thesis, University of Twente.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159.

Duff, I. S., Grimes, R. G., and Lewis, J. G. (1989). Sparse matrix test problems. *ACM Transactions on Mathematical Software (TOMS)*, 15(1):1–14.

Eldan, R. and Shamir, O. (2016). The power of depth for feedforward neural networks. In *Conference on learning theory*, pages 907–940.

Elsken, T., Metzen, J.-H., and Hutter, F. (2017). Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*.

Elsken, T., Metzen, J. H., and Hutter, F. (2018a). Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081*.

Elsken, T., Metzen, J. H., and Hutter, F. (2018b). Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*.

Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60.

Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. (2019). Rigging the Lottery: Making All Tickets Winners. *arXiv e-prints*, page arXiv:1911.11134.

Evci, U., Ioannou, Y. A., Keskin, C., and Dauphin, Y. (2020). Gradient flow in sparse neural networks and how lottery tickets win. *arXiv preprint arXiv:2010.03533*.

Evci, U., Pedregosa, F., Gomez, A., and Elsen, E. (2019). The difficulty of training sparse neural networks. *arXiv preprint arXiv:1906.10732*.

Fedorov, I., Adams, R. P., Mattina, M., and Whatmough, P. N. (2019). Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *arXiv preprint arXiv:1905.12107*.

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

Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. (2019a). Linear mode connectivity and the lottery ticket hypothesis. *arXiv preprint arXiv:1912.05671*.

Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. (2019b). Stabilizing the lottery ticket hypothesis. *arXiv preprint arXiv:1903.01611*.

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

Frazier, P. I. (2018). A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*.

Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192.

Gale, T., Elsen, E., and Hooker, S. (2019). The State of Sparsity in Deep Neural Networks. *arXiv e-prints*, page arXiv:1902.09574.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.

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

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Halmos, P. R. (2017). *Naive set theory*, page 20. Courier Dover Publications.

Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pages 1135–1143, Cambridge, MA, USA. MIT Press.

Hanson, S. J. and Pratt, L. Y. (1989). Comparing biases for minimal network construction with back-propagation. In *Advances in neural information processing systems*, pages 177–185.

Hassibi, B., Stork, D. G., and Com, S. C. R. (1993a). Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 5*, pages 164–171. Morgan Kaufmann.

Hassibi, B., Stork, D. G., and Wolff, G. J. (1993b). Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pages 293–299 vol.1.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

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

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. (2018). Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference*

on Computer Vision (ECCV), pages 784–800.

Hertz, J. A. (2018). *Introduction to the theory of neural computation*. CRC Press.

Hinton, G., Srivastava, N., and Swersky, K. (2012). Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8).

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1).

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. (2021). Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*.

Hoffer, E., Banner, R., Golan, I., and Soudry, D. (2018). Norm matters: efficient and accurate normalization schemes in deep networks. *arXiv preprint arXiv:1803.01814*.

Hoffman, M. D., Brochu, E., and de Freitas, N. (2011). Portfolio allocation for bayesian optimization. In *UAI*, pages 327–336. Citeseer.

Hooker, S., Courville, A., Clark, G., Dauphin, Y., and Frome, A. (2019). What Do Compressed Deep Neural Networks Forget? *arXiv e-prints*, page arXiv:1911.05248.

Hornik, K., Stinchcombe, M., White, H., et al. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Horowitz, M. (2014). 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14.

Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248. PMLR.

Jiang, Y., Neyshabur, B., Mobahi, H., Krishnan, D., and Bengio, S. (2019). Fantastic generalization measures and where to find them. *arXiv preprint arXiv:1912.02178*.

Jin, H., Song, Q., and Hu, X. (2019). Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956.

Jin, X., Xu, C., Feng, J., Wei, Y., Xiong, J., and Yan, S. (2015). Deep learning with s-shaped rectified linear activation units. *arXiv preprint arXiv:1512.07030*.

Kalman, B. L. and Kwasny, S. C. (1992). Why tanh: choosing a sigmoidal function. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 4, pages 578–581. IEEE.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. (2018). Neural architecture search with bayesian optimisation and optimal transport. *arXiv preprint arXiv:1802.07191*.

Karnin, E. D. (1990). A simple procedure for pruning back-propagation trained neural networks. *IEEE transactions on neural networks*, 1(2):239–242.

Karnin, Z., Koren, T., and Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246. PMLR.

Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., and Aila, T. (2020). Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8110–8119.

Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika*, 30(1/2):81–93.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2016). Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*.

Krizhevsky, A., Nair, V., and Hinton, G. (2009). Cifar-10 and cifar-100 datasets. *URl: https://www. cs. toronto. edu/kriz/cifar. html*, 6:1.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86.

Labatie, A. (2019). Characterizing well-behaved vs. pathological deep neural networks. In *International Conference on Machine Learning*, pages 3611–3621. PMLR.

Lane, N. D. and Warden, P. (2018). The deep (learning) transformation of mobile and embedded computing. *Computer*, 51(5):12–16.

Laplace, P. S. (1820). *Théorie analytique des probabilités*. Courcier.

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

LeCun, Y., Denker, J. S., and Solla, S. A. (1989). Optimal Brain Damage. In *NIPS*, pages 598–605. Morgan Kaufmann.

LeCun, Y., Haffner, P., Bottou, L., and Bengio, Y. (1999). Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer.

Lee, N., Ajanthan, T., Gould, S., and Torr, P. H. (2019). A signal propagation perspective for pruning neural networks at initialization. *arXiv preprint arXiv:1906.06307*.

Lee, N., Ajanthan, T., and Torr, P. H. (2018). Snip: Single-shot network pruning based on

connection sensitivity. *arXiv preprint arXiv:1810.02340*.

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

Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Hardt, M., Recht, B., and Talwalkar, A. (2018). Massively parallel hyperparameter tuning.

Li, L. and Talwalkar, A. (2019). Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*.

Liang, T., Glossner, J., Wang, L., and Shi, S. (2021). Pruning and quantization for deep neural network acceleration: A survey. *arXiv preprint arXiv:2101.09671*.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. (2018). Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, Z., Memisevic, R., and Konda, K. (2015). How far can we go without convolution: Improving fully-connected networks. *arXiv preprint arXiv:1511.02580*.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2018a). Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34.

Liu, H., Simonyan, K., and Yang, Y. (2018b). Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

Liu, T. and Zenke, F. (2020). Finding trainable sparse networks through neural tangent transfer. *arXiv preprint arXiv:2006.08228*.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. (2017). Learning Efficient Convolutional Networks through Network Slimming. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2755–2763.

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

Loshchilov, I. and Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Louizos, C., Welling, M., and Kingma, D. P. (2017). Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*.

Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. (2017). The expressive power of neural networks: A view from the width. In *Advances in neural information processing systems*, pages 6231–6239.

Lubana, E. S. and Dick, R. (2021). A gradient flow framework for analyzing network pruning. In *International Conference on Learning Representations*.

Luce, R. D. (2012). *Individual choice behavior: A theoretical analysis*. Courier Corporation.

Luo, J.-H., Wu, J., and Lin, W. (2017). Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066.

Ma, Y., Li, J., Wu, X., Yan, C., Sun, J., and Vuduc, R. (2019). Optimizing sparse tensor times matrix on gpus. *Journal of Parallel and Distributed Computing*, 129:99–109.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3.

McDonald, J. H. (2009). *Handbook of biological statistics*, volume 2. sparky house publishing Baltimore, MD.

Mendoza, H., Klein, A., Feurer, M., Springenberg, J. T., Urban, M., Burkart, M., Dippel, M., Lindauer, M., and Hutter, F. (2019). Towards automatically-tuned deep neural networks. In *Automated Machine Learning*, pages 135–149. Springer.

Merrill, D. and Garland, M. (2016). Merge-based parallel sparse matrix-vector multiplication. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 678–689. IEEE.

Mittal, D., Bhardwaj, S., Khapra, M. M., and Ravindran, B. (2018). Recovering from random pruning: On the plasticity of deep convolutional neural networks. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 848–857. IEEE.

Mocanu, D. C., Mocanu, E., Stone, P., Nguyen, P. H., Gibescu, M., and Liotta, A. (2018). Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12.

Molchanov, D., Ashukha, A., and Vetrov, D. (2017). Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2016). Pruning Convolutional Neural Networks for Resource Efficient Inference. *ArXiv e-prints*.

Morcos, A. S., Yu, H., Paganini, M., and Tian, Y. (2019). One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. *arXiv preprint arXiv:1906.02773*.

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

Mozer, M. C. and Smolensky, P. (1989). Skeletonization: A technique for trimming the fat from a network via relevance assessment. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1*, pages 107–115. Morgan-Kaufmann.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *ICML*.

Narang, S., Elsen, E., Diamos, G., and Sengupta, S. (2017). Exploring Sparsity in Recurrent Neural Networks. *arXiv e-prints*, page arXiv:1704.05119.

Neal, R. M. (1992). Connectionist learning of belief networks. *Artificial intelligence*, 56(1):71–

113.

Neyshabur, B. (2020). Towards learning convolutions from scratch. *Advances in Neural Information Processing Systems*, 33.

Ng, A., Ngiam, J., Foo, C. Y., Mai, Y., Suen, C., Coates, A., Maas, A., Hannun, A., Huval, B., Wang, T., et al. (2013). Unsupervised feature learning and deep learning.

Nocedal, J., Sartenaer, A., and Zhu, C. (2002). On the behavior of the gradient norm in the steepest descent method. *Computational Optimization and Applications*, 22(1):5–35.

Nogueira, F. (2020). Bayesian optimization: Open source constrained global optimization tool for python (2014–). *URL https://github. com/fmfn/BayesianOptimization*.

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

Olshausen, B. A. and Field, D. J. (1997). Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325.

OpenAI, :, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318.

Pessoa, L. (2014). Understanding brain networks and brain organization. *Physics of life reviews*, 11(3):400–435.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*, 7.

Rasmussen, C. E. (2003). Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019). Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. (2017). Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org.

Reed, R. and MarksII, R. J. (1999). *Neural smithing: supervised learning in feedforward artificial*

neural networks. Mit Press.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

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

Samala, R. K., Chan, H.-P., Hadjiiski, L. M., Helvie, M. A., Richter, C., and Cha, K. (2018). Evolutionary pruning of transfer learned deep convolutional neural network for breast cancer diagnosis in digital breast tomosynthesis. *Physics in Medicine & Biology*, 63(9):095005.

Schoenholz, S. S., Gilmer, J., Ganguli, S., and Sohl-Dickstein, J. (2016). Deep information propagation. *arXiv preprint arXiv:1611.01232*.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609.

Sciuto, C., Yu, K., Jaggi, M., Musat, C., and Salzmann, M. (2019). Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*.

See, A., Luong, M.-T., and Manning, C. D. (2016). Compression of Neural Machine Translation Models via Pruning. *arXiv e-prints*, page arXiv:1606.09274.

Shorten, C. and Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Highway networks. *arXiv preprint arXiv:1505.00387*.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Strogatz, S. H. (2001). Exploring complex networks. *nature*, 410(6825):268–276.

Strubell, E., Ganesh, A., and McCallum, A. (2019). Energy and policy considerations for deep

learning in nlp.

Ström, N. (1997). Sparse connection and pruning in large dynamic artificial neural networks.

Su, J., Chen, Y., Cai, T., Wu, T., Gao, R., Wang, L., and Lee, J. D. (2020). Sanity-checking pruning methods: Random tickets can win the jackpot. *arXiv preprint arXiv:2009.11094*.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*.

Suganuma, M., Shirakawa, S., and Nagao, T. (2017). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 497–504. ACM.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147.

Tan, M. and Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR.

Tanaka, H., Kunin, D., Yamins, D. L., and Ganguli, S. (2020). Pruning neural networks without any data by iteratively conserving synaptic flow. *arXiv preprint arXiv:2006.05467*.

Thom, M. and Palm, G. (2013). Sparse activity and sparse connectivity in supervised learning. *Journal of Machine Learning Research*, 14(Apr):1091–1143.

Thompson, N. C., Greenewald, K., Lee, K., and Manso, G. F. (2020). The Computational Limits of Deep Learning. *arXiv e-prints*, page arXiv:2007.05558.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.

Tinney, W. F. and Walker, J. W. (1967). Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809.

Urban, G., Geras, K. J., Kahou, S. E., Aslan, O., Wang, S., Caruana, R., Mohamed, A., Philipose, M., and Richardson, M. (2016). Do deep convolutional nets really need to be deep and convolutional? *arXiv preprint arXiv:1603.05691*.

Van Laarhoven, T. (2017). L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Verhulst, P.-F. (1838). Notice sur la loi que la population suit dans son accroissement. *Corresp. Math. Phys.*, 10:113–126.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

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

Wang, N., XIANG, S., Pan, C., et al. (2020b). You only search once: Single shot neural architecture search via direct sparse optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Warden, P. and Situnayake, D. (2019). *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, Incorporated.

Wei, T., Wang, C., Rui, Y., and Chen, C. W. (2016). Network morphism. In *International Conference on Machine Learning*, pages 564–572.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. (2016). Learning structured sparsity in deep neural networks. *arXiv preprint arXiv:1608.03665*.

Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer.

West, J., Ventura, D., and Warnick, S. (2007). Spring research presentation: A theoretical foundation for inductive transfer. *Brigham Young University, College of Physical and Mathematical Sciences*, 1:32.

White, C., Neiswanger, W., and Savani, Y. (2019). Bananas: Bayesian optimization with neural architectures for neural architecture search. *arXiv preprint arXiv:1910.11858*.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83.

Wistuba, M. (2018). Deep learning architecture search by neuro-cell-based evolution with function-preserving mutations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 243–258. Springer.

Wistuba, M., Rawat, A., and Pedapati, T. (2019). A survey on neural architecture search. *CoRR*, abs/1905.01392.

Wu, Y., Liu, A., Huang, Z., Zhang, S., and Van Gool, L. (2020). Neural architecture search as sparse supernet. *arXiv preprint arXiv:2007.16112*.

Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

Xie, L. and Yuille, A. (2017). Genetic cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1379–1388.

Xie, S., Kirillov, A., Girshick, R., and He, K. (2019). Exploring randomly wired neural networks for image recognition. *arXiv preprint arXiv:1904.01569*.

Yang, G., Pennington, J., Rao, V., Sohl-Dickstein, J., and Schoenholz, S. S. (2019). A mean field theory of batch normalization. *arXiv preprint arXiv:1902.08129*.

Zagoruyko, S. and Komodakis, N. (2016). Wide residual networks. *CoRR*, abs/1605.07146.

Zaheer, M., Reddi, S., Sachan, D., Kale, S., and Kumar, S. (2018). Adaptive methods for nonconvex optimization. In *Advances in neural information processing systems*, pages 9793–

9803.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2019a). *Dive into Deep Learning*. http://www.d2l.ai.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2016). Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*.

Zhang, J.-F., Lee, C.-E., Liu, C., Shao, Y. S., Keckler, S. W., and Zhang, Z. (2019b). Snap: A 1.67—21.55 tops/w sparse neural acceleration processor for unstructured sparse deep neural network inference in 16nm cmos. In *2019 Symposium on VLSI Circuits*, pages C306–C307. IEEE.

Zhang, Z., Wang, H., Han, S., and Dally, W. J. (2020). Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274. IEEE.

Zhao, Y., Li, J., Liao, C., and Shen, X. (2018). Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 94–108.

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

Zhu, M. and Gupta, S. (2017). To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*.

Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710.

Appendices

Appendix A

Sparse Network Optimization

A.1 SC-SDC

In this section, we provide more information about `SC-SDC` and its benefits.

A.1.1 SC-SDC Implementation Details

Wilcoxon Signed Rank Test `SC-SDC` uses the Wilcoxon Signed Rank Test as the statistical test to compare sparse and dense networks. This test is a non-parametric statistical test that compares dependent or paired samples, without assuming the differences between the paired experiments are normally distributed [McDonald, 2009, Demšar, 2006].

Random Sparsity Our work focuses on the training dynamics of random, sparse networks. We achieve random sparsity, by generating a random mask for each layer and then multiply the weights by this mask during each forward pass. The sparsity is distributed evenly across the network. For example, a 20% sparse MLP has 20% of the weights remaining in each layer.

Dense Width A critical component to how we specify our experiments is a term we define as **dense width**. In order to fairly compare sparse and dense networks, we need them to have the same number of active connections at each depth. In the case of sparse networks, this means ensuring they have the same number of active connections as the dense networks, while remaining sparse. **Dense width** refers to the width of a network if that network was dense. This process of comparing sparse and dense networks at different **dense widths** is illustrated in Figure A.2.

Fair Comparison of Sparse and Dense Networks As can be seen from Figure A.2, `SC-SDC` ensures the exact same active parameter count, but the sparse networks will be connected to more neurons. It is possible that the increased number of activations being used can lead to sparse networks having higher representational power. However, most work on expressivity of neural networks looks at this from a depth perspective and proves certain depths of networks are universal approximators [Eldan and Shamir, 2016, Hornik et al., 1989, Funahashi, 1989]. To this end, we ensure these networks have the same depth, but we believe going forward an interesting direction would be ensuring they have a similar amount of active neurons.

SC-SDC Comparison Details For completeness, we provide more details of how we ensure sparse and dense networks are of the same capacity.

Following from Equation 3.2, to ensure the same number of weights in sparse and dense

networks, we can ensure they have the same number of active weights at each layer as follows:

$$||\boldsymbol{a}_S^l||_0 = ||\boldsymbol{a}_D^l||_0, \quad \text{for} \quad l = 1, \ldots, L \quad , \tag{A.1}$$

where $\boldsymbol{a}_S^l$ is the nonzero weights in layer l of sparse network S and $\boldsymbol{a}_D^l$ is the nonzero weights in layer l of dense network D (all the weights since no masking occurs).

This is achieved by masking each of the weight layers of sparse network S:

$$\boldsymbol{a}_S^l = \boldsymbol{\theta}_S^l \odot m^l \quad \text{for} \quad l = 1, \ldots, L \quad , \tag{A.2}$$

where m^l is a random binary matrix (mask) for layer l, s.t. $||m^l||_0 = ||\boldsymbol{a}_D^l||_0$, where $||\boldsymbol{a}_D^l||_0$ is the number of active (nonzero) weights in the dense network and is determined by the chosen comparison width.

For `SC-SDC`, we need a maximum network width N_{MW} and comparison width N_W. We choose a max network width N_{MW} of $n+4$, where n is the input dimension of the network. In the case of CIFAR, $n = 3072$ and so our maximum width $N_{MW} = 3076$. The choice of $n+4$ follows from Lu et al. [2017], where the authors prove a universal approximation theorem for width-bounded ReLU networks, with width bounded to $n + 4$. Our comparison width, N_W, is equivalent to **dense widths** we vary in our experiments - 308, 923, 1538, 2153, 2768 (10%, 30%, 50%, 70% and 90% of our maximum width $N_{MW}(3076)$ when using CIFAR datasets).

The dimensions of each of layers of the different networks, S (sparse) and D (dense), are as follows:

1. First Layer:

$$\boldsymbol{\theta}_D^1 \in \mathbb{R}^{I \times N_W} \quad , \quad \boldsymbol{\theta}_S^1 \in \mathbb{R}^{I \times N_{MW}} \quad , \quad m^1 \in \{0,1\}^{I \times N_{MW}} \tag{A.3}$$

2. Intermediate Layers:

$$\boldsymbol{\theta}_S^1 \in \mathbb{R}^{N_W \times N_W} \quad , \quad \boldsymbol{\theta}_S^1 \in \mathbb{R}^{N_{MW} \times N_{MW}} \quad , \quad m^{\{2,\ldots,L-1\}} \in \{0,1\}^{N_{MW} \times N_{MW}} \tag{A.4}$$

3. Final Layer:

$$\boldsymbol{\theta}_S^L \in \mathbb{R}^{N_W \times O} \quad , \quad \boldsymbol{\theta}_S^L \in \mathbb{R}^{N_{MW} \times O} \quad , \quad m^L \in \{0,1\}^{N_{MW} \times O} \quad , \tag{A.5}$$

where N_{MW} is maximum width of the sparse layer, N_W is the comparison width, I is the input dimension, O is output dimension, L is the number of layers in the network, $\boldsymbol{\theta}_S^l$ is the weights in layer l of sparse network S and $\boldsymbol{\theta}_D^l$ is the weights in layer l of dense network D.

This process would be the same for convolutional layers, but there would be a third dimension to handle the different channels. In Figure A.1, we provide an illustrative example showing how to ensure sparse and dense networks are compared fairly.

A.1.2 Benefits of SC-SDC

The benefits of `SC-SDC` can be summarized as follows:

- **We can better understand sparse network optimization.** `SC-SDC` allows us to identify which optimization or regularization methods are poorly suited to sparse networks in a controlled setting, ensuring the results are a direct result of the sparse connections themselves.

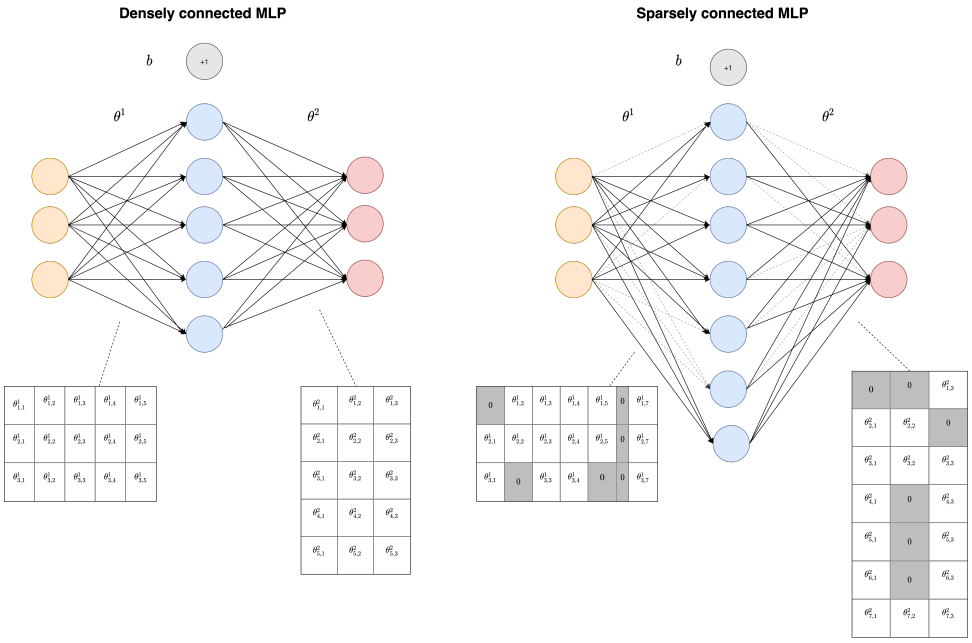


Figure A.1: **Fair Comparison of Sparse and Dense Networks.** We fairly compare sparse and dense networks, by ensuring they have the same number of connections per layer and turning off the bias unit.

- **Learn at what parameter and size budget, sparse networks are better than dense.** Comparing sparse and dense networks of the same capacity allows us to see which architecture is better at different configurations. In configurations where sparse architectures perform better, we could exploit advances in sparse matrix computation and storage [Zhao et al., 2018, Merrill and Garland, 2016, Ma et al., 2019, Zhang et al., 2019b, 2020] to simply default to sparse architectures.

A.2 Gradient Flow

In the main body of the paper, we use EGF_2 (Equation 3.6) as our notion of gradient flow for its comparative benefits with other measures of gradient flow. In this section, for completeness, we present the full set of results using the different formulations of gradient flow on CIFAR-100. Namely, we show $||\boldsymbol{g}||_1$ (Equation 3.5) (Figure A.4 and A.7) ,$||\boldsymbol{g}||_2$ (Equation 3.5) (Figure A.5 and A.8) and EGF_1 (Equation 3.6) (Figure A.3 and A.6).

A.3 Adaptive Methods

Not all adaptive methods are created equally. We briefly show the different update rules of the adaptive learning rate methods we use in our experiments. In Equations A.6, we highlight that Adam and RMSProp have the same second moment estimate $\mathbf{v}_t$, which uses an exponential weighted moving average (EWMA) of past squared gradients to obtain an estimate of the

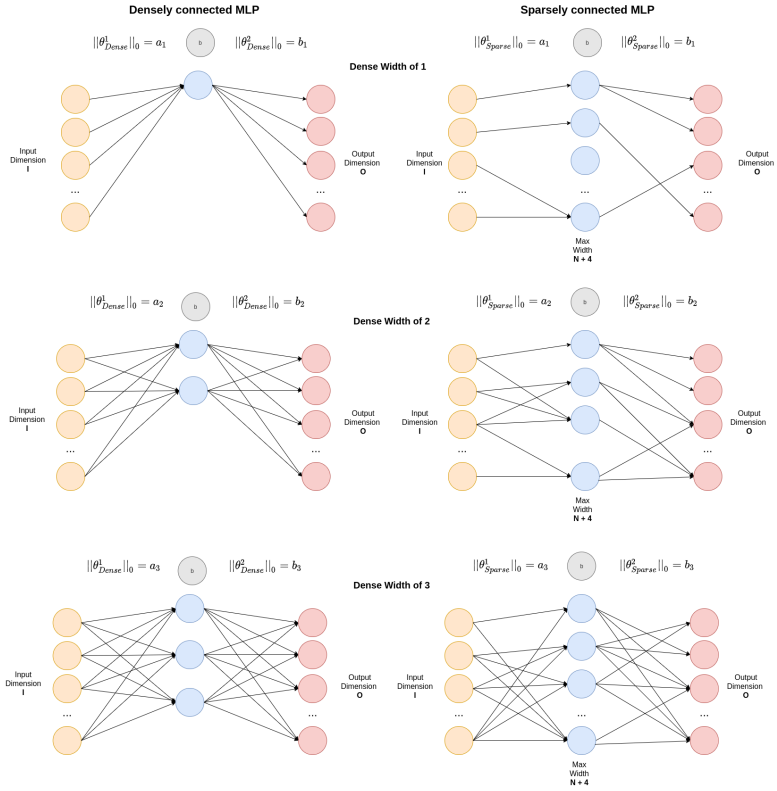


Figure A.2: **Fairly Comparing Sparse and Dense Networks at Different Widths.** We show how we ensure that sparse and dense networks have the same parameter count, at different widths.

variance of the gradient, which differs from Adagrad's $\mathbf{v}_t$.

Adagrad

$$\mathbf{v}_t = \mathbf{v}_{t-1} + \mathbf{g}_t^2$$
$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{v}_t + \epsilon}} \odot \mathbf{g}_t$$

RMSProp

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + (1-\gamma)\mathbf{g}_t^2$$
$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{v}_t + \epsilon}} \odot \mathbf{g}_t$$

Adam

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\,\mathbf{g}_t$$
$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2$$
$$\mathbf{m}_t = \frac{\mathbf{m}_t}{1-\beta_1}$$
$$\mathbf{v}_t = \frac{\mathbf{v}_t}{1-\beta_2}$$
$$\mathbf{g}'_t = \frac{\eta \hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}$$
$$\mathbf{w}_t = \mathbf{w}_{t-1} - \mathbf{g}'_t \tag{A.6}$$

where $\mathbf{g}_t$ are the gradients of a network, $\mathbf{m}_t$ is the estimate of the first moment of the gradient, $\mathbf{v}_t$ is the estimate of the second moment of the gradient, η is the learning rate, β_1,β_2,γ are weighting parameters.

This difference between EWMA optimizers (Adam and RMSProp) and Adagrad is two-fold:

1. Adagrad's $\mathbf{v}_t$ is evenly weighted between the current squared gradient $\mathbf{g_t^2}$ and the previous value of $\mathbf{v}_t$. Adam and RMSProp's $\mathbf{v}_t$ use 0.9 [Hinton et al., 2012] or 0.999 [Kingma and Ba, 2014] as a weighting parameter for the past $\mathbf{v}_t$, which weights past $\mathbf{v}_{t-1}$ more than $\mathbf{g}_t$.

Figure A.3: Gradient Flow in CIFAR-100 using EGF_1, with low learning rate (0.001)

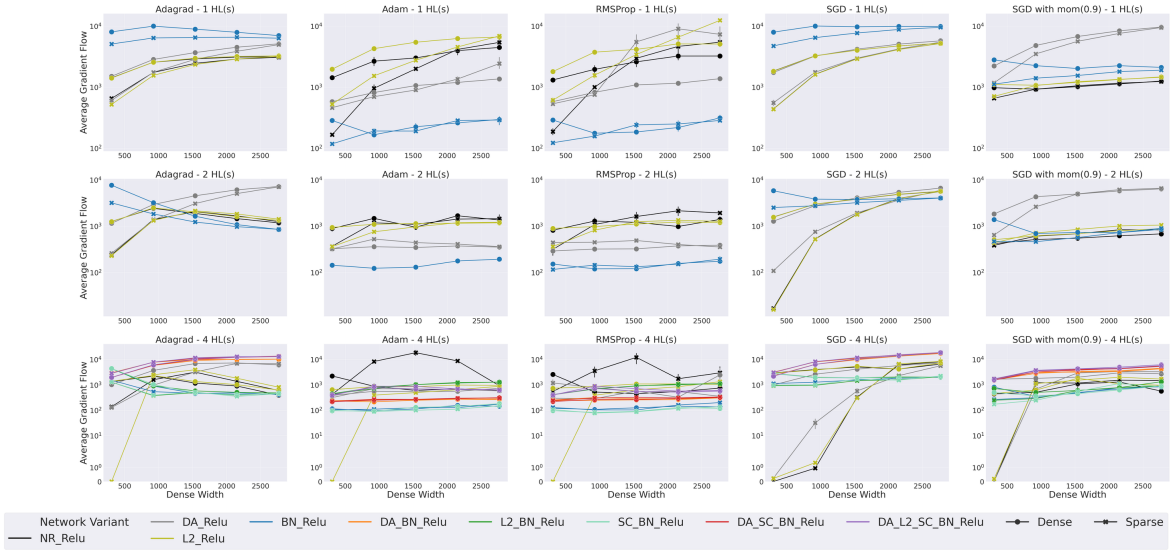
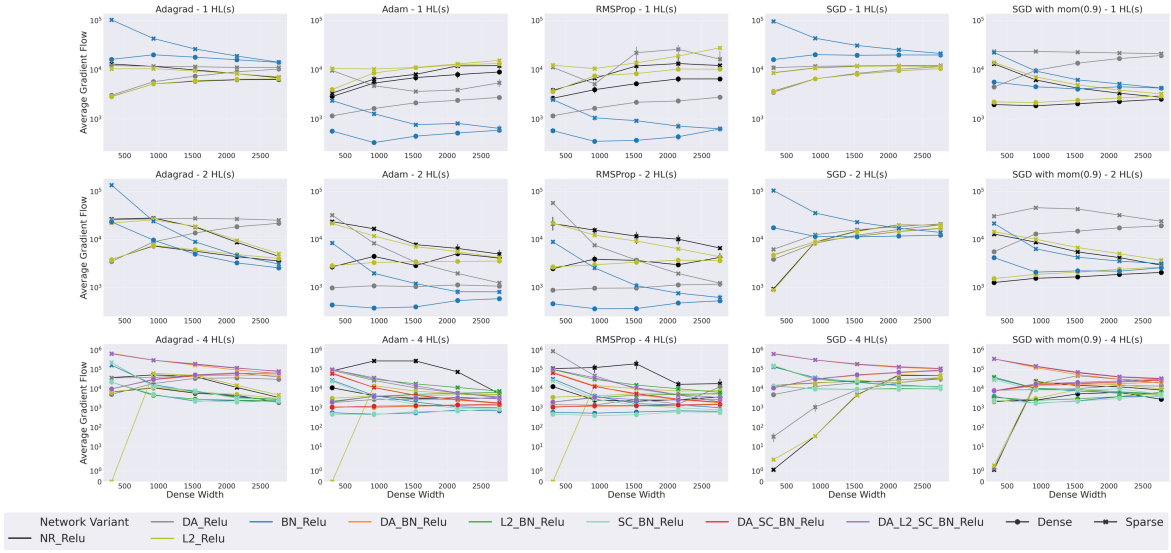


Figure A.4: Gradient Flow in CIFAR-100 using $||\boldsymbol{g}||_1$, with low learning rate (0.001)



2. Adagrad's $\mathbf{v}_t$ grows almost linearly over time [Zhang et al., 2019a], which results in fast decline in learning rate (a higher $\mathbf{v}_t$ results in a smaller effective learning rate since η is divided by $\mathbf{v}_t$). Since Adam and RMSProp's $\mathbf{v}_t$ is multiplied by a weighting factor, this value decreases over time. This occurs even if we choose a weighting factor of 0.5, since both $\mathbf{v}_t$ and $\mathbf{g}_t$ will be multiplied by 0.5. A smaller $\mathbf{v}_t$, results in a higher effective learning rate, meaning learning rates stay higher for longer.

We believe the higher effective learning rate (point 2) is why Adam and RMSProp behave similarly in some contexts. Furthermore, it has been noted that at times, when the $\mathbf{v}_t$ estimate blows up, it can cause optimizers to fail to converge [Zaheer et al., 2018, Zhang et al., 2019a]. $\mathbf{v}_t$ is also present in the formulation of Adadelta [Zeiler, 2012].

A.4 Activation Functions

We plot the activation functions we used (Figure A.9a) and their gradients (Figure A.9b).

Figure A.5: Gradient Flow in CIFAR-100 using $||\boldsymbol{g}||_2$, with low learning rate (0.001)

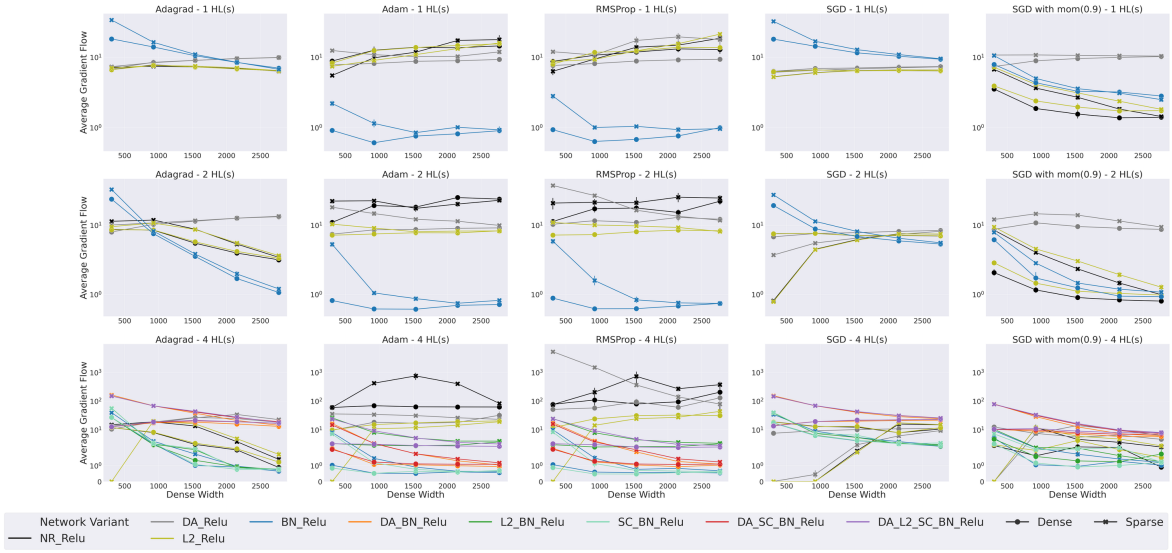
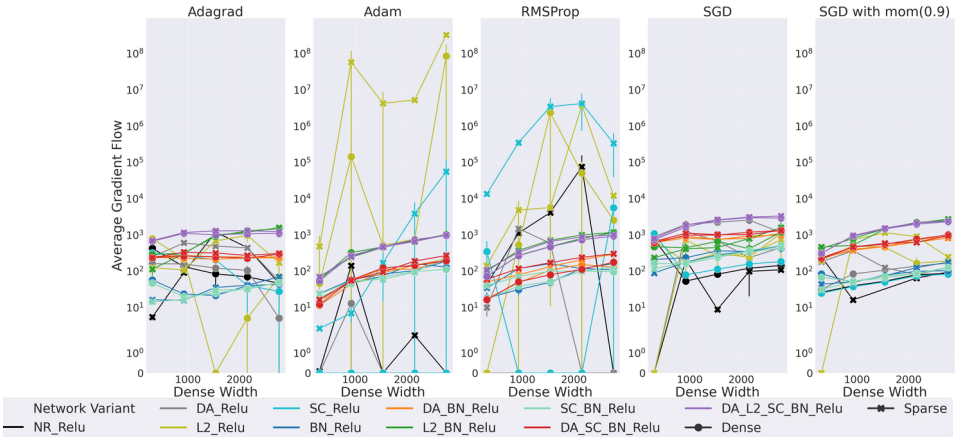


Figure A.6: Gradient Flow in CIFAR-100 using EGF_1, with high learning rate (0.1)



A.5 Detailed Results for SC-SDC

In this section, we present the detailed results for our experiments. For the tables, we use the following colour scale, depending on the p-value from the Wilcoxon Signed Rank Test.

Colour Scale based on p-values: 0 (S > D) .5 1 (S ≤ D)

where S refers to sparse networks and D refers to dense networks.

A.5.1 Detailed Results

We present the detailed results as follows:

1. FMNIST

   (a) All SC-SDC results for Four Hidden Layers with different regularization methods - Table A.1.

Figure A.7: Gradient Flow in CIFAR-100 using $||\boldsymbol{g}||_1$, with high learning rate (0.1)

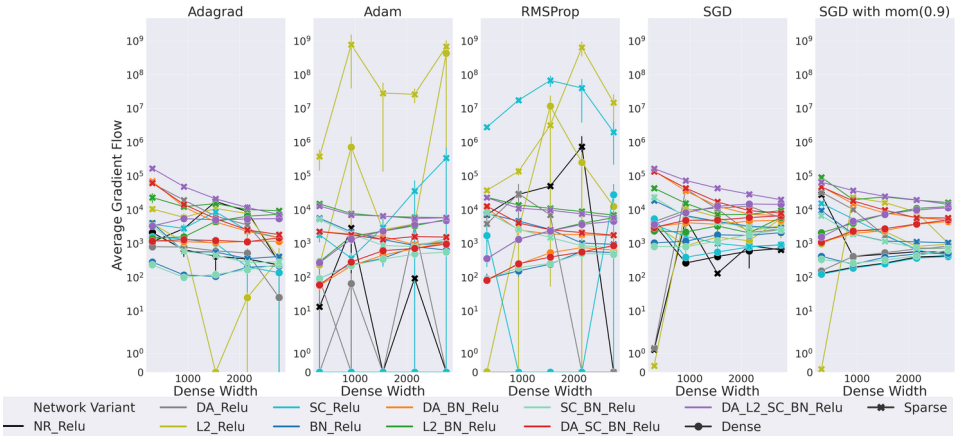
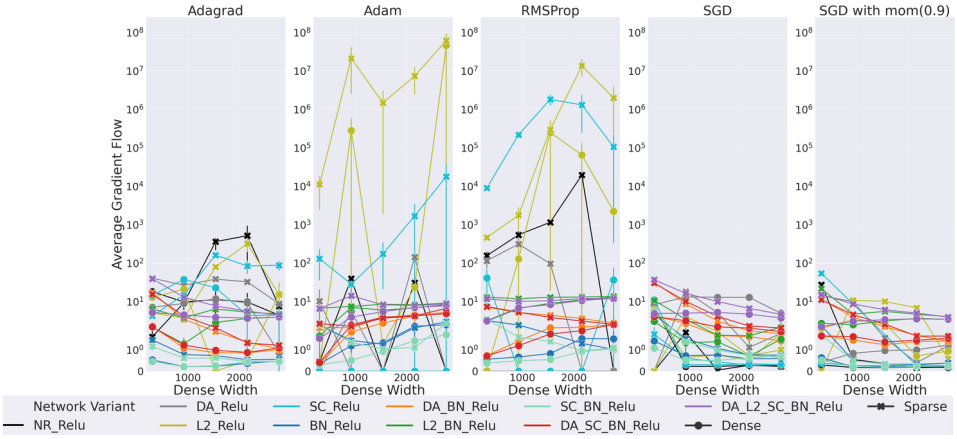


Figure A.8: Gradient Flow in CIFAR-100 using $||\boldsymbol{g}||_2$, with high learning rate (0.1)



(b) Effects of Regularization with a low learning rate (0.001) - Figure A.10.

2. CIFAR-10

   (a) All `SC-SDC` results for Four Hidden Layers with different regularization methods - Table A.2.

   (b) Effects of Regularization with a low learning rate (0.001) - Figure A.11.

   (c) Effects of Regularization with a high learning rate (0.1) - Figure A.12.

   (d) Effects of Activations with a low learning rate (0.001) - Figure A.13.

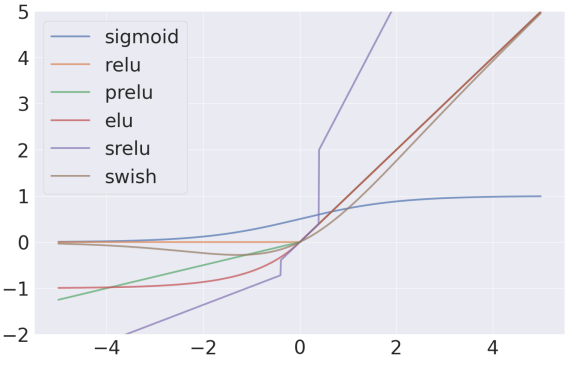
3. CIFAR-100

   (a) Detailed Results with a low learning rate (0.001)

      i. One, Two and Four hidden layers showing all forms of regularization - accuracy and gradient flow (Figure A.14).

      ii. Four Hidden Layers with different activation functions - accuracy and gradient flow (Figure A.15).

(a) Activation Function with inputs [-5,5]



(b) Derivative of Activation Function with inputs [-5,5]

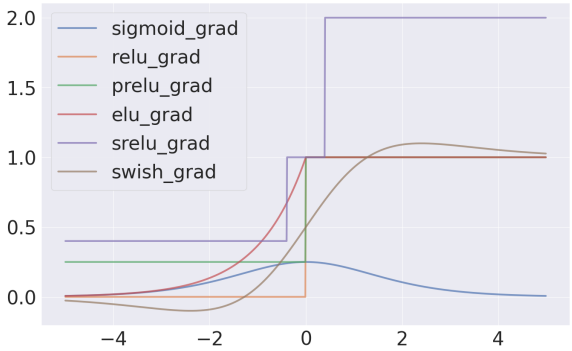


Figure A.9: **Plot of Activation Functions.** We plot the activation functions we used and their derivatives. We see that Swish is the only activation function that allows for the flow of negative gradients, due its non-monotonicity.

iii. All `SC-SDC` results for Four Hidden Layers with different regularization and activation methods- Table A.3.

(b) Detailed Results with a high learning rate (0.1)

i. Four Hidden Layers with BatchNorm - accuracy and gradient flow (Figure A.16).

ii. Four Hidden Layers with different activation functions - accuracy and gradient flow (Figure A.17).

iii. Adam vs AdamW - accuracy and gradient flow (Figure A.18).

iv. Gradient Flow for Wide ResNet-50 - Figure A.19.

	NR_Relu	L2_Relu	DA_Relu	SC_Relu	BN_Relu	DA_SC_BN_Relu	DA_L2_SC_BN_Relu
Adagrad	0.993	0.992	1.000	0.941	**0.001**	**0.001**	**0.000**
Adam	0.170	0.995	0.088	**0.012**	**0.020**	**0.000**	0.076
Mom (0.9)	1.000	0.991	1.000	0.924	**0.000**	**0.008**	**0.008**

Table A.1: **Wilcoxon Signed Rank Test Results for MLPs With Four Hidden Layers, With a Low Learning Rate (0.001), trained on FMNIST.**

(a) Different Regularization Methods - Low learning rate (0.001)

	NR	DA	L2	SC	BN
Adagrad	0.555	0.681	0.999	0.271	**0.000**
Adam	**0.024**	**0.001**	0.681	**0.000**	**0.000**
RMSProp	**0.000**	0.598	0.866	**0.001**	**0.005**
SGD	1.000	1.000	1.000	**0.000**	**0.000**
Mom (0.9)	1.000	0.862	1.000	0.995	**0.001**

(b) Different Regularization Methods - High learning rate (0.1)

	BN	DA_BN	L2_BN	SC_BN
Adagrad	**0.001**	0.063	0.953	**0.000**
Adam	**0.009**	**0.001**	0.074	**0.000**
RMSProp	0.227	0.375	0.219	**0.037**
SGD	**0.000**	0.500	0.289	**0.000**
Mom (0.9)	**0.000**	**0.024**	**0.004**	**0.000**

(c) Effect of Different Activation Functions - High learning rate (0.001)

	ReLU	SReLU	Swish	PReLU
Adam	**0.004**	**0.006**	**0.002**	**0.005**
Mom (0.9)	**0.000**	0.053	**0.002**	**0.001**

Table A.2: **Wilcoxon Signed Rank Test Results for MLPs with Four Hidden Layers, trained on CIFAR-10.** Wilcoxon Signed Rank Test Results for CIFAR-10, using a p-value of 0.05, with the bold values indicating where we can be statistically confident that sparse networks perform better than dense (reject H_0 from 3.4). We also use a continuous colour scale to make the results more interpretable. This scale ranges from green (0 - likely that sparse networks perform better than dense) to yellow (0.5 - 50% chance that sparse networks perform better than dense) to red (1 - highly likely that sparse networks do not outperform dense - cannot reject H_0 from 3.4).

(a) Effect of Different Regularization Methods

	NR	DA	L2	SC	BN	DA_BN	L2_BN	SC_BN	DA_SC_BN	DA_L2_SC_BN
Adagrad	1.000	1.000	0.998	0.239	**0.006**	**0.002**	**0.001**	**0.003**	**0.001**	**0.004**
Adam	**0.000**	0.055	0.198	**0.003**	0.079	0.051	0.254	0.166	**0.039**	0.118
RMSProp	**0.001**	**0.000**	0.300	0.166	0.117	**0.021**	0.914	0.541	0.096	**0.023**
SGD	1.000	1.000	1.000	0.248	**0.000**	**0.000**	**0.001**	**0.003**	**0.001**	**0.004**
Mom (0.9)	1.000	1.000	1.000	0.999	**0.001**	**0.000**	**0.007**	**0.008**	**0.001**	**0.003**

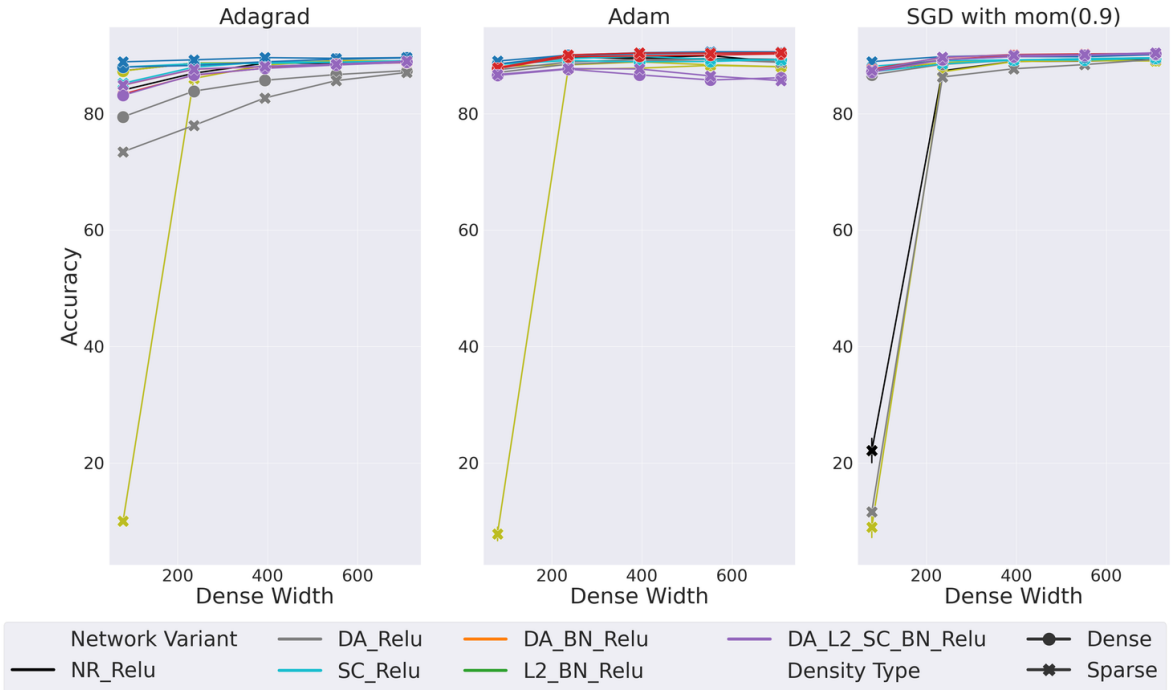
(b) Effect of Activation Functions

	relu	swish	srelu	prelu	elu	sigmoid
Adagrad	**0.000**	**0.001**	0.104	**0.001**	**0.000**	**0.011**
Adam	**0.042**	**0.003**	0.076	**0.001**	**0.004**	**0.000**
RMSProp	0.104	0.165	0.065	**0.047**	**0.013**	**0.001**
SGD	**0.002**	**0.000**	0.180	**0.000**	**0.001**	**0.000**
Mom (0.9)	**0.001**	**0.000**	0.054	**0.000**	**0.000**	0.054

Table A.3: **Wilcoxon Signed Rank Test Results for MLPs with Four Hidden Layers, with low learning rate (0.001), trained on CIFAR-100.** Wilcoxon Signed Rank Test Results for CIFAR-100, using a low learning rate. We use a p-value of 0.05, with the bold values indicating where we can be statistically confident that sparse networks perform better than dense (reject H_0 from 3.4). We also use a continuous colour scale to make the results more interpretable. This scale ranges from green (0 - likely that sparse networks perform better than dense) to yellow (0.5 - 50% chance that sparse networks perform better than dense) to red (1 - highly likely that sparse networks do not outperform dense - cannot reject H_0 from 3.4).

Figure A.10: Effect of Regularization on Accuracy and Gradient Flow for Dense and Sparse Networks, with Four Hidden Layers on FMNIST, with low learning rate (0.001)

(a) Test Accuracy for Dense and Sparse Networks on FMNIST



(b) Gradient Flow for Dense and Sparse Networks on FMNIST

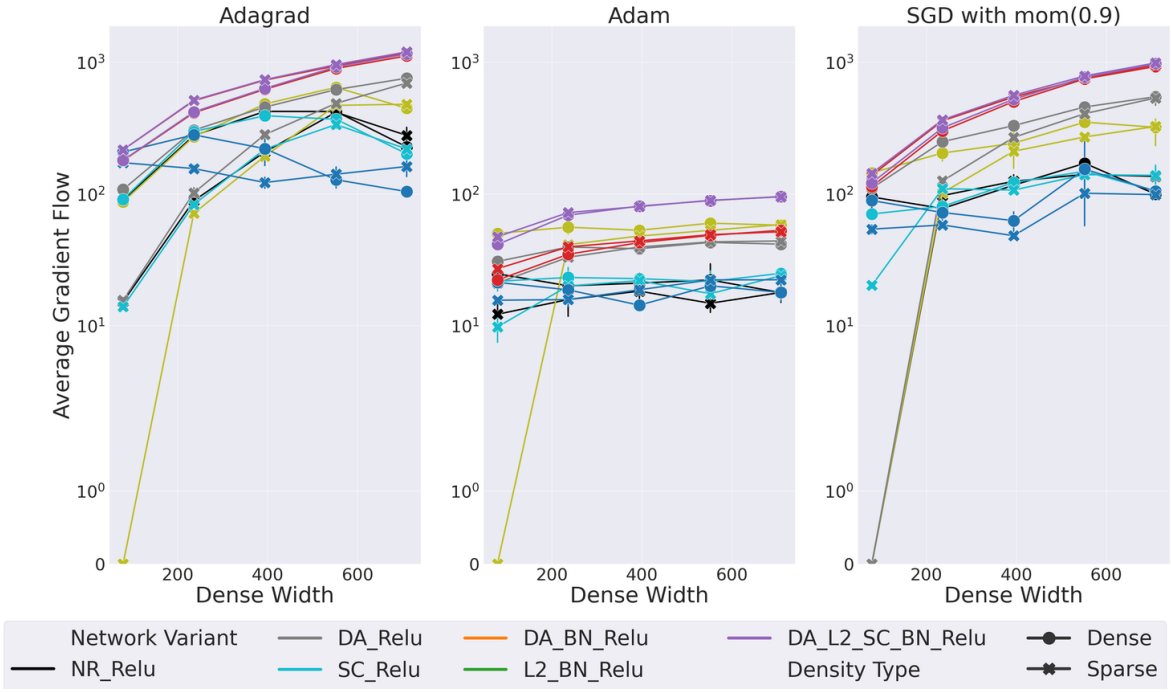
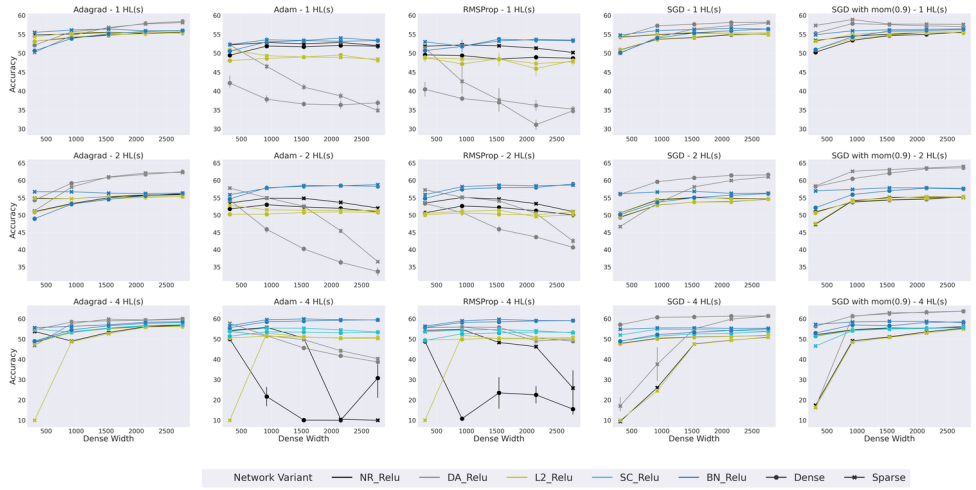


Figure A.11: Effect of Regularization on Accuracy and Gradient Flow for Dense and Sparse Networks, on CIFAR-10, with low learning rate (0.001)

(a) Test Accuracy for Dense and Sparse Networks



(b) Gradient Flow for Dense and Sparse Networks

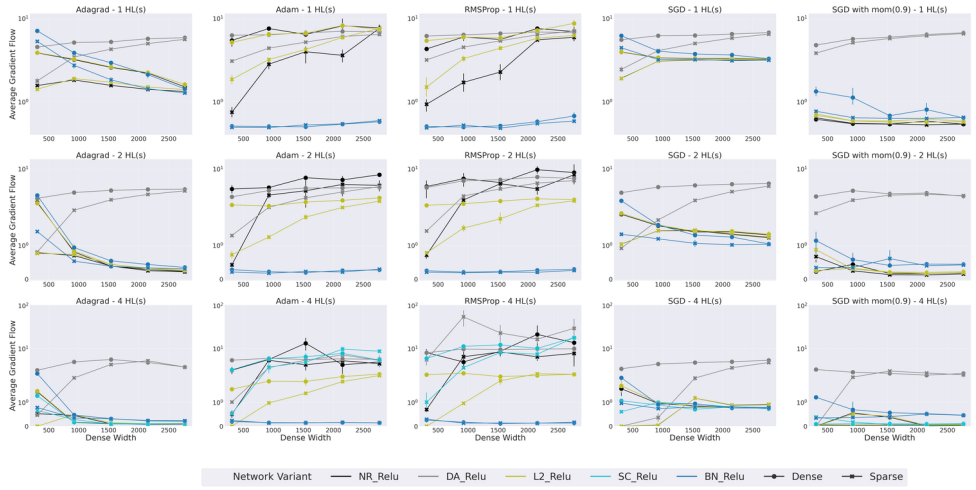
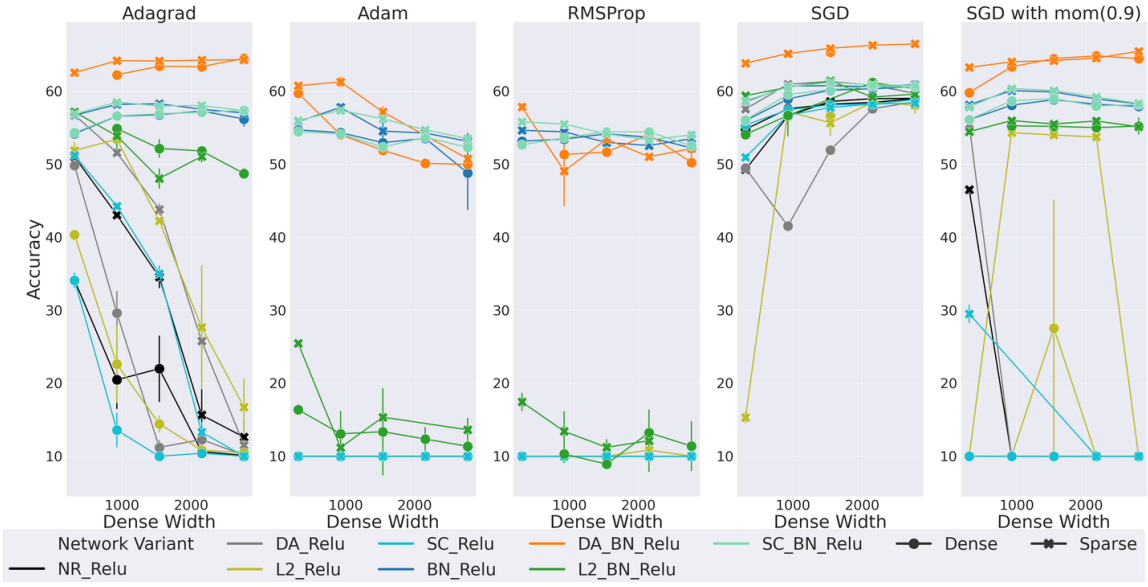


Figure A.12: Effect of Regularization on Accuracy and Gradient Flow for Dense and Sparse Networks, with Four Hidden Layers on CIFAR-10, with high learning rate (0.1)

(a) Test Accuracy for Dense and Sparse Networks



(b) Gradient Flow for Dense and Sparse Networks

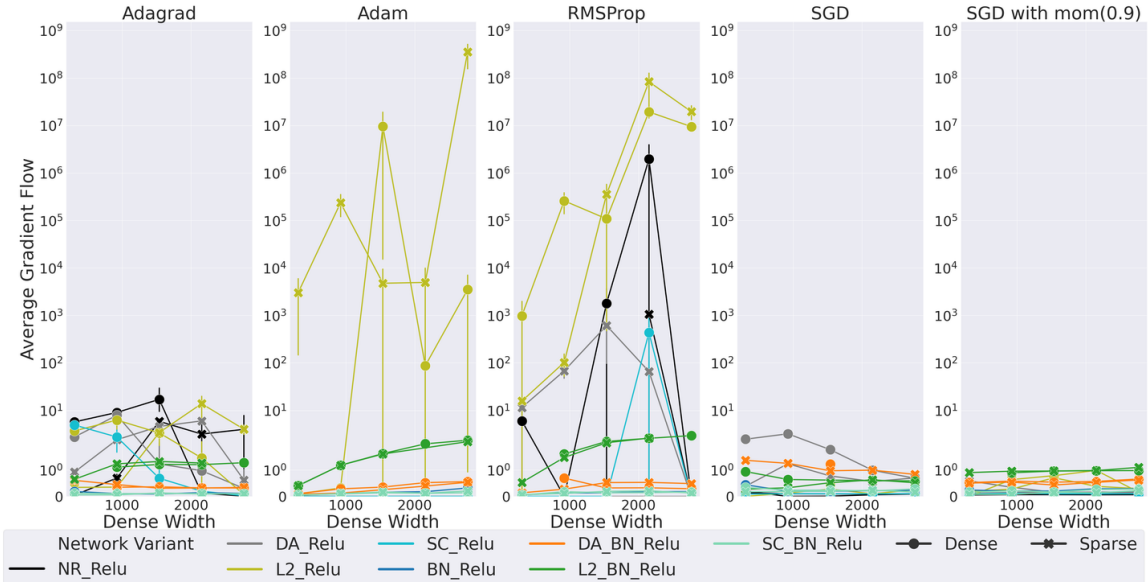
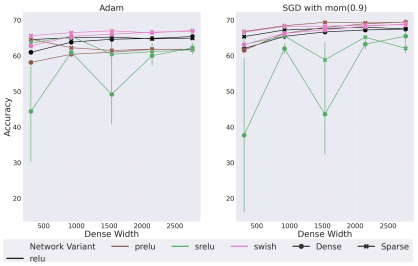
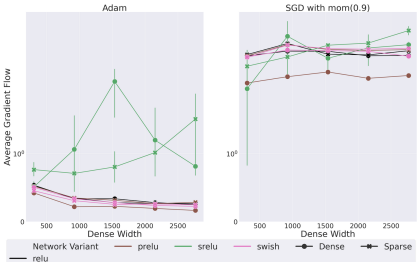


Figure A.13: Effect of Activation Functions on Accuracy and Gradient Flow for Dense and Sparse Networks with Four Hidden Layers on CIFAR-10, with low learning rate (0.001)



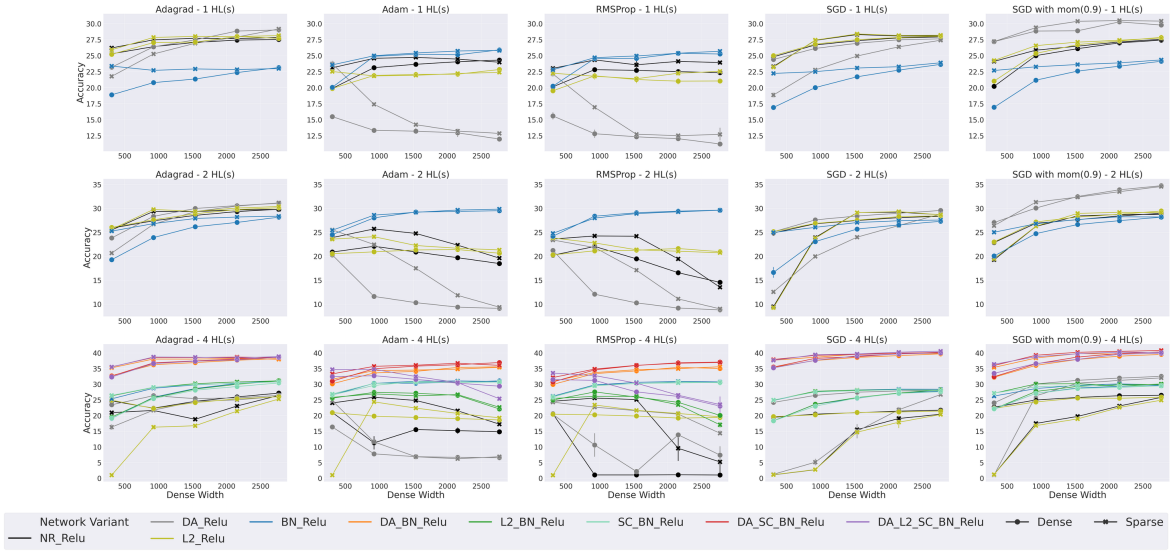
(a) Test Accuracy for Dense and Sparse Networks on CIFAR-10



(b) Gradient Flow for Dense and Sparse Networks on CIFAR-10

Figure A.14: Effect of Regularization on Accuracy and Gradient Flow for Dense and Sparse Networks with One, Two and Four Hidden Layers on CIFAR-100, with low learning rate (0.001)

(a) Test Accuracy for Dense and Sparse Networks on CIFAR-100



(b) Gradient Flow for Dense and Sparse Networks on CIFAR-100

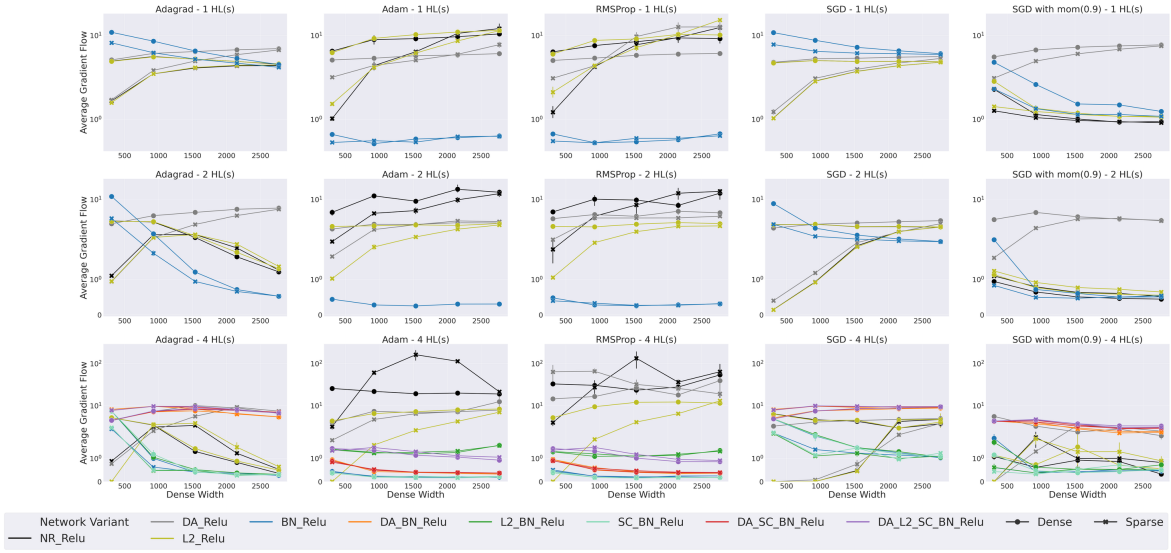
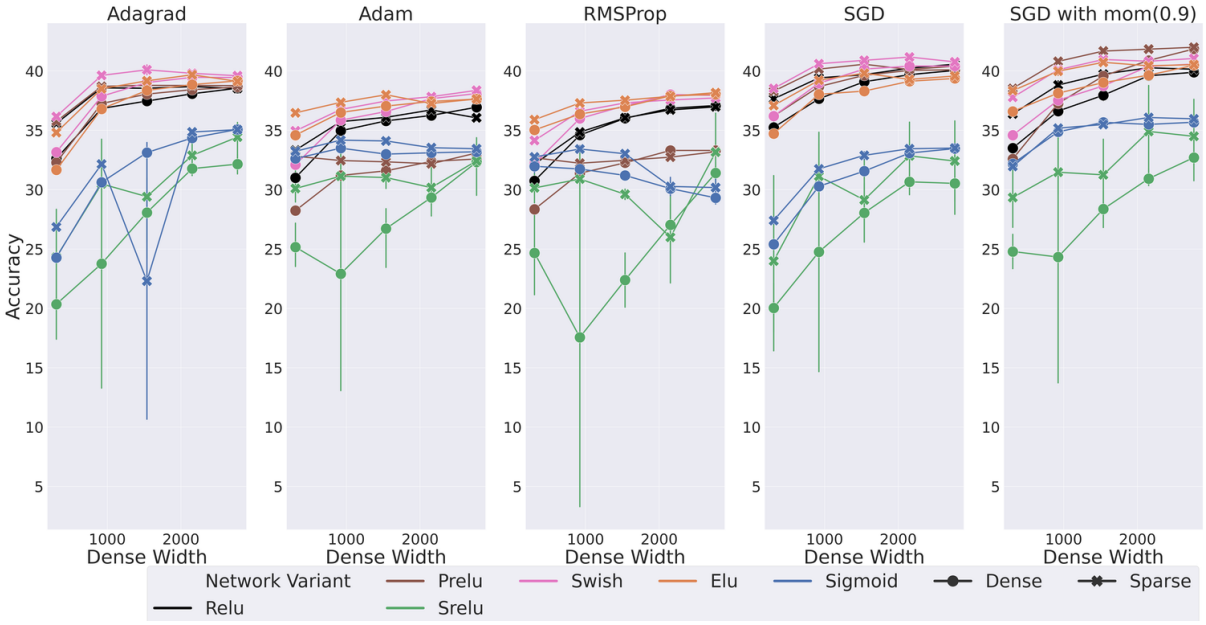


Figure A.15: Effect of Activation Functions on Accuracy and Gradient Flow for Dense and Sparse Networks with Four Hidden Layers on CIFAR-100, with low learning rate (0.001)

(a) Test Accuracy for Dense and Sparse Networks on CIFAR-100



(b) Gradient Flow for Dense and Sparse Networks on CIFAR-100

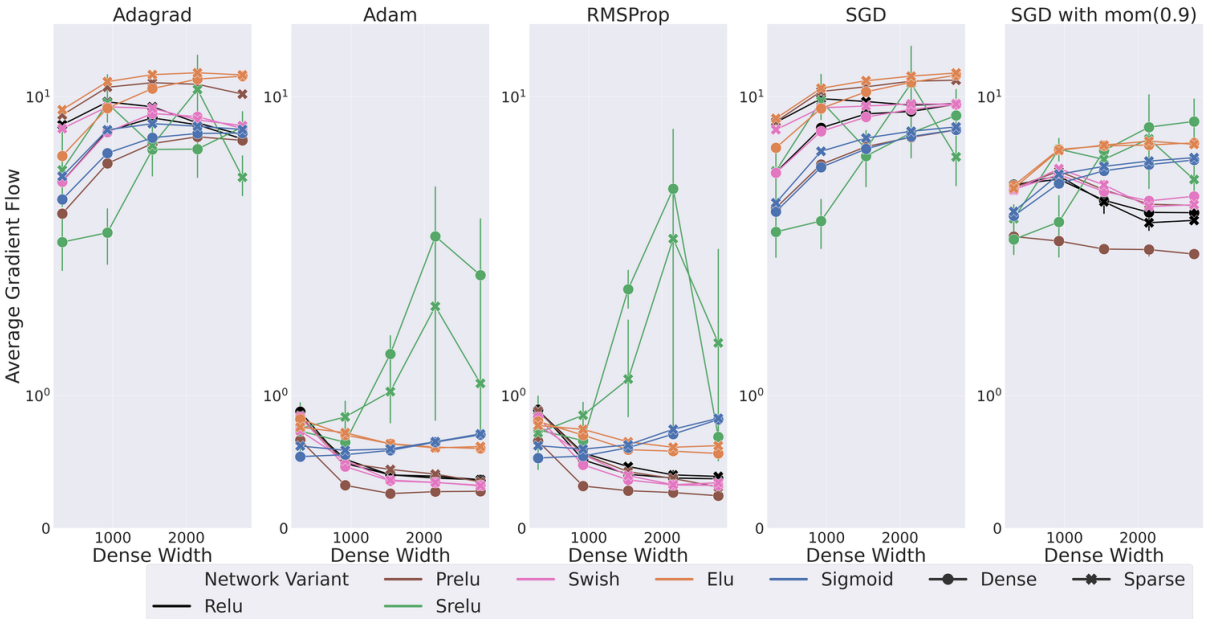
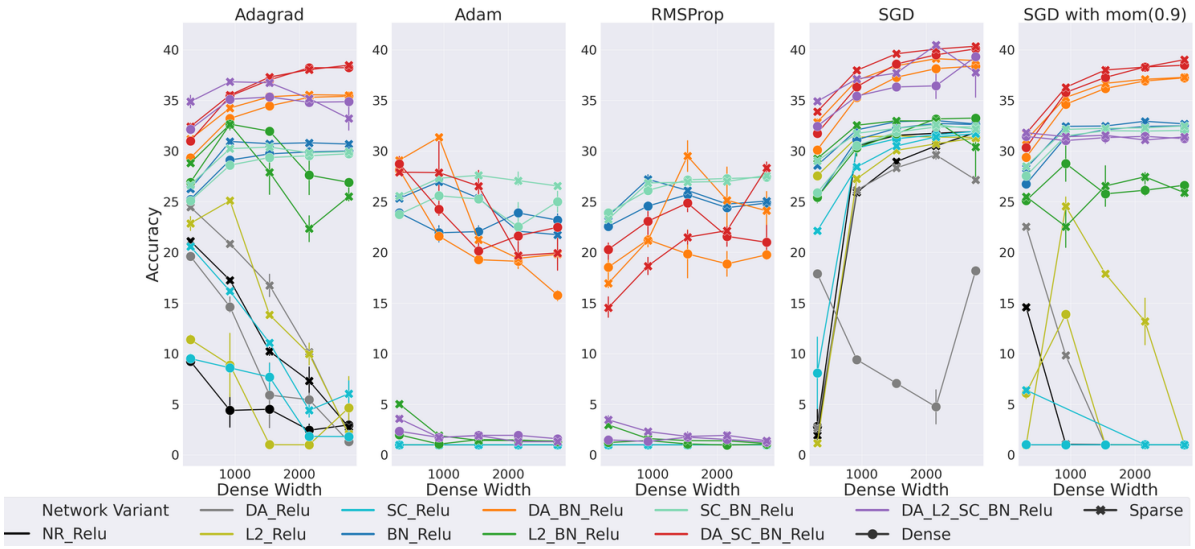


Figure A.16: Effect of Regularization (*with BatchNorm*) on Accuracy and Gradient Flow for Dense and Sparse Networks, with Four Hidden Layers on CIFAR-100, with high learning rate (0.1)

(a) Test Accuracy for Dense and Sparse Networks on CIFAR-100



(b) Gradient Flow for Dense and Sparse Networks on CIFAR-100

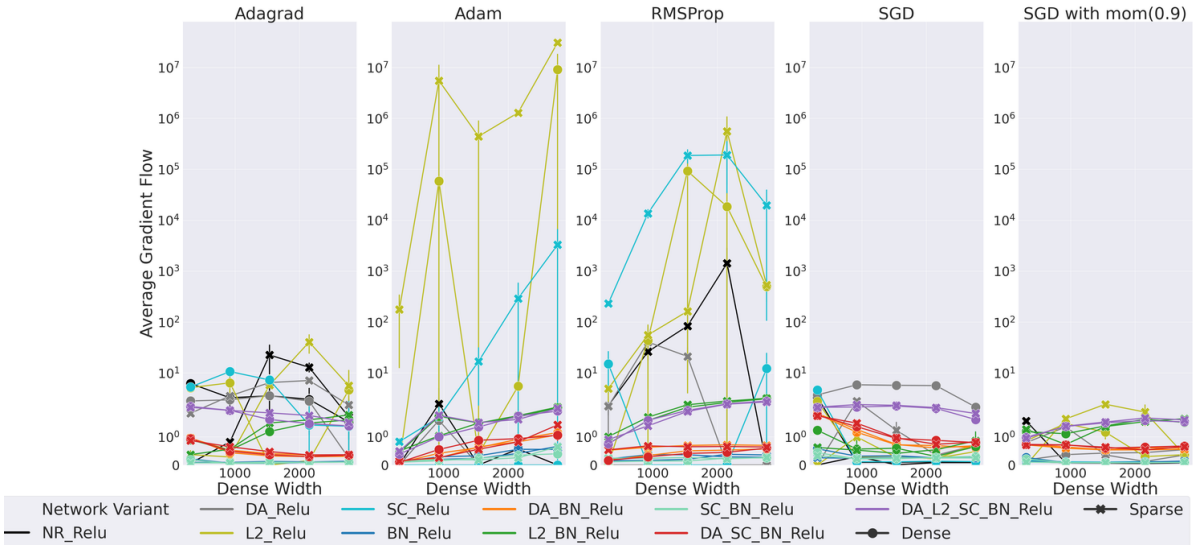
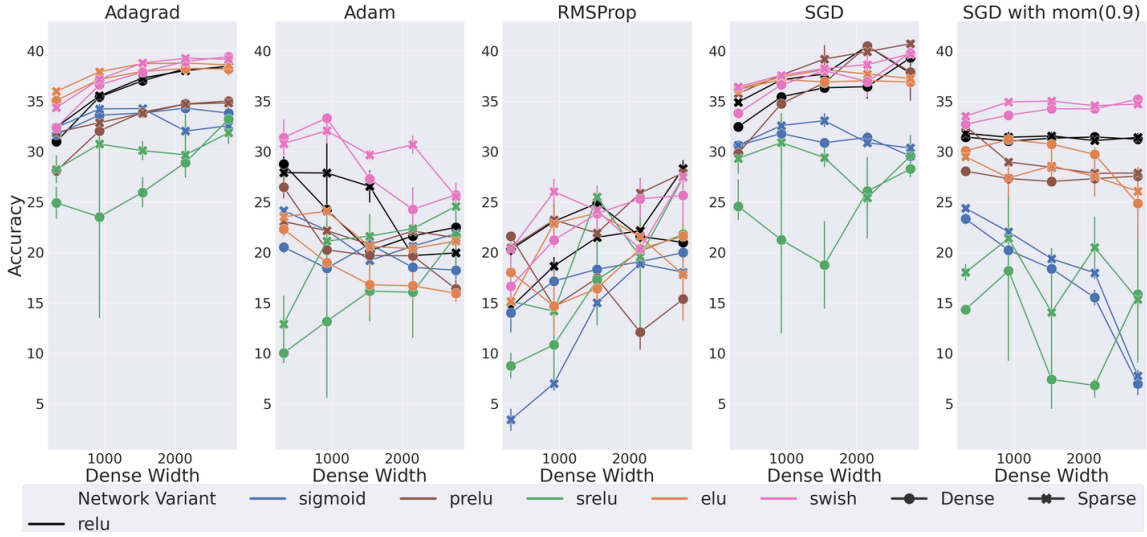


Figure A.17: Effect of Activation Functions on Accuracy and Gradient Flow for Dense and Sparse Networks with Four Hidden Layers on CIFAR-100, with high learning rate (0.1)

(a) Test Accuracy for Dense and Sparse Networks on CIFAR-100



(b) Gradient Flow for Dense and Sparse Networks on CIFAR-100

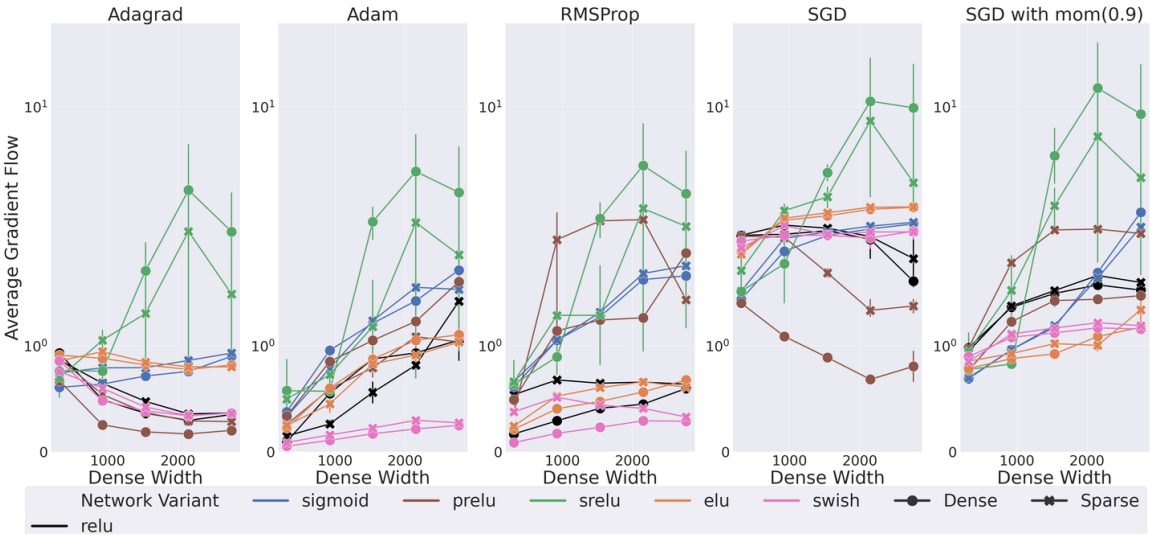
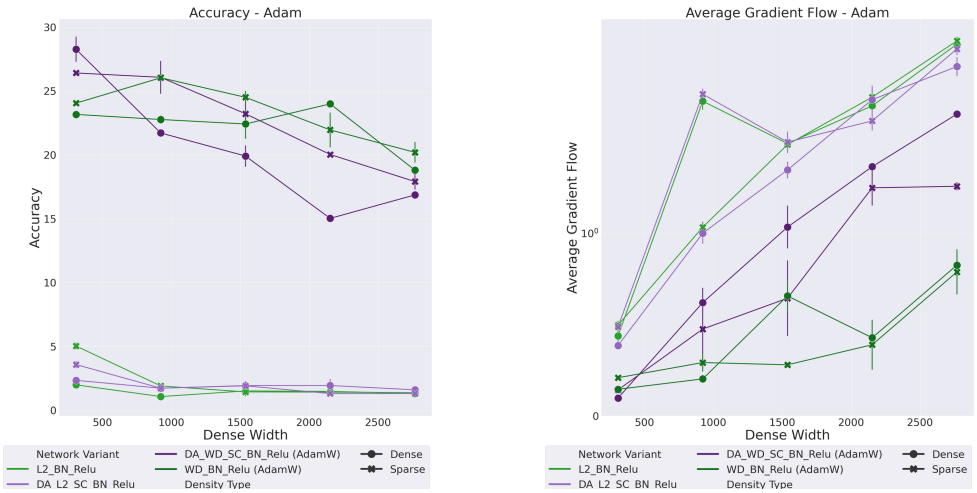


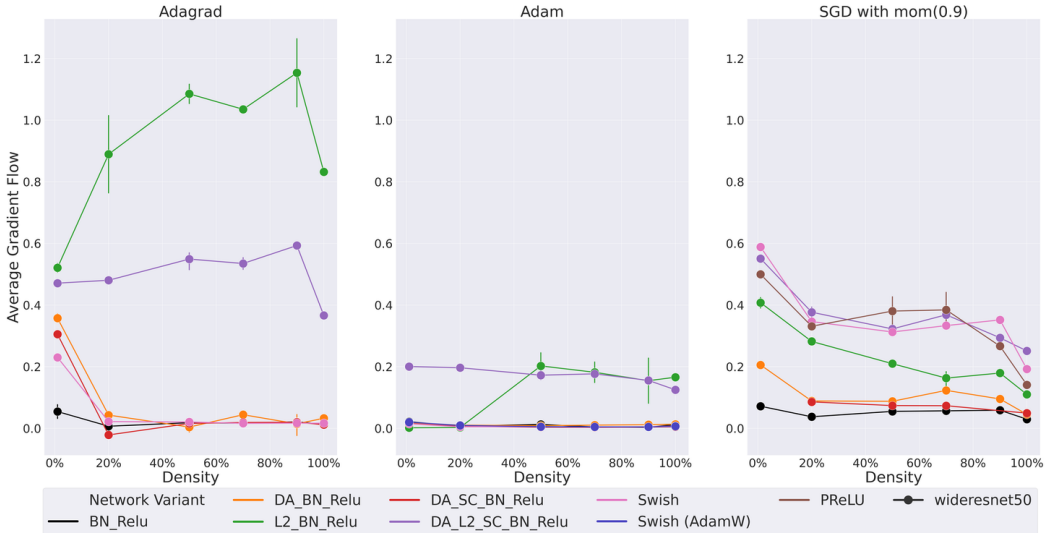
Figure A.18: **Comparison of Adam and AdamW for Dense and Sparse Networks.** We compare the performance of Adam and AdamW for dense and sparse Networks with four hidden layers on CIFAR-100, with a high learning rate (0.1). We see that AdamW's weight decay formulation has a lower `EGF` than the standard L2 formulation used in Adam and this correlates to better network performance in AdamW



(a) Test Accuracy for Dense and Sparse Networks on CIFAR-100

(b) Gradient Flow for Dense and Sparse Networks on CIFAR-100

Figure A.19: **Wide ResNet-50 Gradient Flow on CIFAR-100.** We illustrate the gradient flow of Wide ResNet-50 on CIFAR-100, with the density ranging from 1% to 100%. The accuracy results can be found in Figure 3.5.



Appendix B

Learning Sparse Architectures

B.1 Detailed SNAS Results

We present the detailed results of the best MLPS (Table B.1) and best CNNs (Table B.2) found by our `SNAS` algorithm.

# HL	Search Alg	Accuracy	# Weights (M)	Δ Accuracy	% Dense	Density
2.0	Dense Baseline	68.48	18.9	0.00	100.000	[1.0]
2.0	Random	72.06	11.8	3.58	62.017	[0.456, 0.785, 0.2]
2.0	Bayes	72.17	11.8	3.69	62.017	[0.456, 0.785, 0.2]
4.0	Dense Baseline	71.70	37.9	0.00	100.000	[1.0]
4.0	Random	72.80	25.1	1.10	66.377	[0.375, 0.951, 0.732, 0.599, 0.156]
4.0	Bayes	72.84	25.1	1.14	66.377	[0.375, 0.951, 0.732, 0.599, 0.156]
8.0	Dense Baseline	70.12	75.7	0.00	100.000	[1.0]
8.0	Random	72.69	41.4	2.57	54.625	[0.12, 0.713, 0.761, 0.561, 0.771, 0.494, 0.523, 0.428, 0.025]
8.0	Bayes	72.31	36.9	2.19	48.675	[0.375, 0.951, 0.732, 0.599, 0.156, 0.156, 0.058, 0.866, 0.601]
16.0	Dense Baseline	65.37	151.5	0.00	100.000	[1.0]
16.0	Random	68.92	70.7	3.55	46.659	[0.006, 0.815, 0.707, 0.729, 0.771, 0.074, 0.358, 0.116, 0.863, 0.623, 0.331, 0.064, 0.311, 0.325, 0.73, 0.638, 0.887]
16.0	Bayes	67.06	66.6	1.69	43.962	[0.075, 0.987, 0.074, 0.358, 0.116, 0.863, 0.623, 0.331, 0.064, 0.311, 0.772, 0.199, 0.006, 0.815, 0.707, 0.729, 0.771]
32.0	Dense Baseline	63.48	302.9	0.00	100.000	[1.0]
32.0	Random	65.41	145.4	1.93	48.001	[0.075, 0.987, 0.772, 0.199, 0.006, 0.815, 0.707, 0.729, 0.771, 0.074, 0.358, 0.116, 0.863, 0.623, 0.331, 0.064, 0.311, 0.325, 0.73, 0.638, 0.887, 0.472, 0.12, 0.713, 0.761, 0.561, 0.771, 0.494, 0.523, 0.428, 0.025, 0.108, 0.031]
32.0	Bayes	64.53	147.1	1.05	48.576	[0.108, 0.031, 0.161, 0.318, 0.818, 0.861, 0.007, 0.511, 0.417, 0.222, 0.636, 0.314, 0.509, 0.908, 0.249, 0.41, 0.756, 0.229, 0.077, 0.29, 0.93, 0.808, 0.633, 0.871, 0.804, 0.187, 0.893, 0.539, 0.807, 0.896, 0.11, 0.228, 0.427]

(a) Results on CIFAR-10

# HL	Search Alg	Accuracy	# Weights (M)	Δ Accuracy	% Dense	Density
2.0	Dense Baseline	47.07	19.2	0.00	100.000	[1.0]
2.0	Random	47.47	12.8	0.40	66.403	[0.375, 0.951, 0.732]
2.0	Bayes	47.50	16.5	0.43	85.880	[0.734, 0.997, 0.45]
4.0	Dense Baseline	47.19	38.2	0.00	100.000	[1.0]
4.0	Random	47.54	30.5	0.35	79.895	[0.428, 0.967, 0.964, 0.853, 0.294]
4.0	Bayes	47.92	26.6	0.73	69.730	[0.346, 1.0, 0.873, 0.59, 0.068]
8.0	Dense Baseline	45.71	76.0	0.00	100.000	[1.0]
8.0	Random	45.35	43.5	-0.36	57.223	[0.118, 0.649, 0.746, 0.583, 0.962, 0.375, 0.286, 0.869, 0.224]
8.0	Bayes	45.44	65.8	-0.27	86.622	[0.866]
16.0	Dense Baseline	38.51	151.7	0.00	100.000	[1.0]
16.0	Random	41.89	70.9	3.38	46.736	[0.006, 0.815, 0.707, 0.729, 0.771, 0.074, 0.358, 0.116, 0.863, 0.623, 0.331, 0.064, 0.311, 0.325, 0.73, 0.638, 0.887]
16.0	Bayes	40.36	79.7	1.85	52.501	[0.012, 0.247, 0.894, 0.06, 0.583, 0.983, 0.747, 0.247, 0.493, 0.506, 0.691, 0.958, 0.564, 0.889, 0.015, 0.493, 0.98]
32.0	Dense Baseline	36.41	303.2	0.00	100.000	[1.0]
32.0	Random	37.98	121.6	1.57	40.106	[0.037, 0.472, 0.565, 0.066, 0.776, 0.453, 0.524, 0.441, 0.401, 0.56, 0.155, 0.182, 0.862, 0.946, 0.373, 0.271, 0.644, 0.409, 0.025, 0.156, 0.716, 0.659, 0.027, 0.222, 0.231, 0.672, 0.02, 0.104, 0.8, 0.179, 0.653, 0.238, 0.099]
32.0	Bayes	38.68	152.5	2.27	50.303	[0.004, 0.379, 0.924, 0.289, 0.215, 0.145, 0.579, 0.824, 0.584, 0.144, 0.094, 0.334, 0.706, 0.593, 0.662, 0.328, 0.707, 0.796, 0.819, 0.434, 0.33, 0.722, 0.753, 0.365, 0.149, 0.331, 0.911, 0.656, 0.69, 0.438, 0.275, 0.925, 0.005]

(b) Results on CIFAR-100

Table B.1: Detailed Results of `SNAS` on MLPS

Model	Optim	Search Alg	Accuracy	# Weights (M)	Δ Accuracy	% Dense	Density
Resnet32	Adamw	Dense Baseline	69.180	1.900	0.000	100.000	[1.0]
Resnet32	Adamw	Bayes	69.710	1.100	0.530	59.976	[0.599]
Resnet32	Sgd	Dense Baseline	75.090	1.900	0.000	100.000	[1.0]
Resnet32	Sgd	Bayes	75.630	1.600	0.540	86.654	[0.866]
Wideresnet50	Adamw	Dense Baseline	73.880	36.500	0.000	100.000	[1.0]
Wideresnet50	Adamw	Random	74.310	17.000	0.430	46.557	[0.389, 0.271, 0.829, 0.357, 0.281, 0.543, 0.141, 0.802, 0.075, 0.987, 0.772, 0.199, 0.006, 0.815, 0.707, 0.729, 0.771, 0.074, 0.358, 0.116, 0.863, 0.623, 0.331, 0.064, 0.311, 0.325, 0.73, 0.638, 0.887]
Wideresnet50	Adamw	Bayes	74.760	34.700	0.880	95.074	[0.951]
Wideresnet50	Sgd	Dense Baseline	81.250	36.500	0.000	0	[1.0]
Wideresnet50	Sgd	Random	81.130	30.400	-0.120	83.253	[0.832]
Wideresnet50	Sgd	Bayes	81.350	31.600	0.100	86.624	[0.866]

Table B.2: Detailed Results of `SNAS` on CNNs