Self-Optimizing Networks

Clement Lee

Princeton University

Dept. of Computer Science

advised by Bernard Chazelle

5 May 2017

Acknowledgements

I would like to thank my advisor, Bernard Chazelle, for all of his contributions to my thesis. Despite only having found him after the deadline (much to the ire of the COS department), I could not have asked for a better advisor. Thank you so much for all of your guidance throughout the year, and I hope this thesis is an interesting, if long, read for you.

I additionally give thanks to the Princeton SEAS and COS departments for providing me with funding. This has been incredibly helpful and I don't believe that my experiments or results would have been possible without it.

Lastly, my thanks and my love go out to my wife Belinda Chen, without whom this work would not have been completed. You have been there with me not just through this last hectic week, but also in every aspect of my life. This thesis is dedicated to you, and your otherworldly effort in helping me the whole way. I love you so much.

In summary, if there is one thing I've learned, it's that people with the initials B.C. are helpful.

Contents

Chapter 1

Introduction

In recent years, deep learning has exploded as the forefront of what the New York Times has branded the "A.I. Awakening". It has seen applications in nearly every field of artifical intelligence, and has grown to encompass far more as well. Nearly every application of artifical intelligence has started to adopt deep learning, from classifying different kinds of whales to generating imitation Cézannes. Deep learning is starting to surpass humans in a variety of complex classification tasks. The generational improvements resulting from deep learning rival the improvements made by traditional machine learning methods over a much longer period of time [27].

Furthermore, the advent of deep learning has been ushered in at a pace that has surprised even experts within the field. Traditionally considered to be an unassailable human stronghold, AlphaGo [33] was able to beat modern Go masters. Deep learning has often been seen as an incredibly effective modelling tool, as it involves significantly less manual instruction than other methods of artificial intelligence. Part of the allure of is the potential to have it understand high-level features without relying on domain knowledge, and in fact many deep learning results have demonstrated superior performance to expert humans. While deep networks have not, in general, pointed to significant developments in universal AI, they are quickly becoming the standard method for specialized tasks.

Deep learning has come with additional complexities, however. Typical deep networks, while providing excellent accuracy, have far worse computational efficiency than other methods of machine learning. AlphaGo was reliant on large-scale distributed computing infrastructure in order to achieve peak performance, and in fact the modern-day superiority of deep networks has often been attributed to a maturity of hardware and technology. Furthermore, due to the high computational costs of deep networks, developing new architectures is a very slow and costly process involving long wait times. Combined with a generally confusing literature on ever-changing best practices, deep learning is quickly becoming a research quagmire.

To address these deficiencies, we aim to improve on the current state of network design by making the training process more transparent to the user. Our methods promote stability in the network and improve generalizability of the network's output. Across multiple datasets, we are able to see solid improvements which additionally show significant future promise.

Chapter 2

Background

In this section, we provide a general introduction to the relevant basics of deep learning. We begin by outlining the basics of neural networks and their historical theory. Neural networks are, however, insufficient for most modern day tasks, so we introduce convolutions and convolutional neural networks. We additionally provide some information on common training methods in modern deep learning and how they are applied to our thesis. Finally, we describe residual networks, which we utilize for our experiments.

2.1 Neural Networks

In its purest form, the foundational principle of neural networks is inspired by the biology of the human brain. Neural networks are a subset of the larger field of artificial intelligence (AI). Researchers in AI have often modelled new algorithms after biological phenomena. For example, genetic algorithms are based on evolution and particle swarm optimization is based on animal social behaviors. Historically, neural networks emerged in the very beginnings of artificial intelligence research, and from those times a few core fundamentals still remain.

Firstly, the structure of a basic feedforward network was established. In a general sense, a neural network is a directed graph, with neurons as nodes and weights as edges. Every neuron activates, or outputs, with a strength that is a function of the element-wise multiplication of the inputs with the edge weights. That is, if $w_{i,j}$ is the weight value between nodes i and j, n_i is the activation of node i, and N_j is the list of node indices that are connected to j, then node j will activate with strength

$$n_j = F\left(\sum_{i \in N_j} w_{i,j} n_i\right)$$

This definition relies on an activation function F, which allows the network to produce nonlinear behaviors. We can provide input into the neural network by activating a set of nodes with specific values, and we can similarly read output from any subset of nodes. A feedforward network is then any acyclic neural network graph. These networks are typically organized in layers of neurons, which indicate the depth of each node. In this model, layers are typically fully connected, meaning that all

nodes in one layer are connected to all nodes of the next layer. This allows a computationally-efficient model of weights as a matrix M, taking input vector V to output vector MV.

Throughout modern literature, feedforward networks are an important but rarely examined component; the structure is often considered fixed and serves to provide a final classification. Key limitations of fully-connected layers prevent them from being suitable for use as the sole structure of larger networks. For example, because of the fully connected nature of the layers, they require an immense amount of memory. Such a layer between two sets of just 10000 nodes would require 100 million parameters, while modern networks often have a total of 10 million parameters [11]. This extra capacity, while being inefficient, can also be bad for training in general; there is no sense of locality in such a layer, as every node is treated individually. This means that it is difficult and nearly impossible to train higher level features that should be treated equally across all areas of the input (which is of particular interest to problems like image classification). However, even with these limitations, fully-connected layers remain critical for the task they perform.

The other key insight of neural networks is backpropagation [16], which is an algorithm to let errors accumulated from the output layer of the network propagate backwards through the network, training it in the process. As in the example above, if the network's output is O, but the correct response would be C, we can calculate the error $E = O - C$. From this, we need a cost function that determines how errors are judged; a typical example may be the L_2 loss

$$\text{Cost}(O - C) = \sum_{0}^{n} ||O_i - C_i||^2$$

However, since we know that

$$O = F\left(\sum_{0}^{n} w_i a_i\right)$$

it is possible to figure out the influence each weight had on the error by taking the partial derivative of the cost function with respect to the weight. Utilizing this partial derivative, each weight can be modified as a result of the following layer. This allows the weight updates to propagate backwards through the network, which gives the algorithm its name. Modern training methods are far more advanced, but still rely on the basic algorithm described here, which is often termed gradient descent. LeCun et al.'s seminal work in this field, *Gradient-Based Learning Applied to Document Recognition* [25], provided the first basis of using backpropagation methodologies to train visual classifiers. Even more importantly, it introduced the fundamental structure of the modern visual deep learning network. In its usage of convolutions as a method for extracting high-level features out of larger images, it set the framework for a new style of network that would prove to be far more efficient and scalable.

2.2 Convolutions

A convolution is an operator applied to two functions f and g, which provides a way of interpreting one function in the context of the other. The operation is generally defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(r)g(t - r)dr$$

In the perspective of modern deep learning, we are primarily interested in its usage as a matrix operator; in this context, we limit the range of g to the size of the matrix s such that

$$(f * g)(t) = \int_0^s f(r)g(t-r)dr$$

We refer to g as the convolutional kernel. Using a convolutional kernel to preprocess the image proves to be critical to the performance of modern deep learning methods, as a small kernel can operate over a large image in parallel.

For example, consider the basic edge-detecting matrix

$$E = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

This convolution will perform the element-wise matrix multiplication of the kernel E with the immediate neighbors of each pixel, and then aggregate the elements by summation. That is, if the pixel values around a specific pixel e are

$$P_e = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

then the convolution at that pixel will be

$$\begin{aligned} P_e * E &= 0a + 1b + 0c + 1d - 4e + 1f + 0g + 1h + 0i \\ &= (b + d + f + h) - 4e \end{aligned}$$

Accordingly, it will create a new matrix, with each element representing the convolutional kernel applied at that point. As shown above, the convolution $P_e * E$ will have the strongest activation when there is a strong difference between the pixel e and its neighbors (b, d, f, and h), thus performing a basic localized form of edge detection. Figure 2.1 shows this convolution applied to an arbitrary image.

A convolutional neural network is therefore the product of chaining together convolutions to perform efficient feature extraction with the standard feedforward neural network structure. LeCun's contribution to this structure was showing that the same backpropagation methods used to train other networks could also be applied to convolutional layers, allowing convolutional neural networks (CNNs) to learn their own feature extractors. This allows the CNN to determine what kinds of high-level feature extraction is necessary for the specific problem. More importantly, this allows for networks to automatically chain convolutional layers, in which the initial information can pass through multiple layers of feature extraction, which are all automatically determined from the training data.

2.3 Modern Training

While the basics of neural network training are covered above, there are significant improvements that we highlight in this section.

Figure 2.1: A demonstration of an edge-detecting convolution, from the GIMP User's Manual. [7]

When training a network using gradient descent methods, all of the weights are updated simultaneously. This means that the weight update mechanism can be somewhat chaotic, as there is nothing stabilizing a layer from the changes in the previous and following layers, resulting in a problem called covariate shift. To tackle this, Ioffe and Szegedy introduce batch normalization [21], which helps by computing summary statistics of the training batch and allows a trained parameter to normalize the layer activations. Batch normalization has become a common tool in most deep learning libraries and it is almost universally utilized in modern architectures for improving performance without adding a significant number of trainable parameters. For our work, we partially rewrite common implementations of Ioffe and Szegedy's method in order to accommodate shifting layer capacities.

With network sizes often reaching the millions of parameters, the problem of ensuring that each neuron is contributing and learning something different becomes significantly more difficult. This was quickly observed as deep learning became more popular as a discipline, and larger networks proved nearly impossible to train. Hinton et al. utilize Dropout [17, 34] to solve this issue. This method randomly drops certain neurons during training, ensuring that neurons cannot coadapt to each other. While rather basic in its implementation, dropout has proved to be a crucial part of modern training regimes, and helps dramatically with overfitting errors. For our thesis, we do not rely on dropout because the networks we adapt do not depend on it. In particular, residual network topologies generally do not use dropout, as they would dramatically hinder the intended flow of data through the network.

The nonlinear activation function used by every neuron has also become a significant topic of debate. Originally, many neural networks were designed either with the sigmoid or hyperbolic tangent activations. This is generally seen as effective for smaller networks, but less so for larger networks, as both asymptotically reach ±1, resulting in the gradients becoming zero for large values, which slows training dramatically. Nair and Hinton improve on this using rectified linear units [31], often referred to as ReLUs. This activation function is defined by

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

and despite being very simple, has become an extremely common activation function. Some have noted, however, that the gradients being zero for all negative values may be somewhat problematic,

so a number of solutions have been proposed. Leaky ReLUs [28] were a modification that replace the $x < 0$ case with a constant "leakiness" $l < 1$:

$$f(x) = \begin{cases} x, & x \geq 0 \\ -lx, & x < 0 \end{cases}$$

This was further enhanced with He et al.'s work with PReLU [14], which allow the parameter l to be trained. PReLUs have been seen as a way of introducing small numbers of parameters (as there is only one per node) into a network, rather than modifying the network's architecture significantly. This finally led to ELU (Exponential linear units) by Clevert et al., which are defined as

$$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

Clevert et al. also note, importantly, that negative activations (which are not present in ReLU), are important for preventing extreme network biases, much in the same way as batch normalization. To our knowledge, this is one of the best known modern activation functions, so we utilize it throughout our experiments.

2.4 Residual Networks

As network architectures have changed over time, an ongoing goal has been to develop truly deep architectures. Even as better training algorithms have allowed network depth to increase to tens of layers, it is a generally-held principle that depth, not width, is crucial for allowing a network to learn complex features. At the same time, gradient-descent methods work poorly in networks with significant depth, as the gradient term (the partial derivative of the loss function with respect to the weight) decreases significantly by each layer. This means that after a certain amount of depth in the network, the gradient is so small that it is nearly entirely noise. The issue of disappearing gradient is in some form mitigated by training methods, some of which are more heavily biased towards the sign of the gradient rather than the magnitude. However, regardless of the specific algorithm, gradients are effectively unusable for networks with significant depth for typical network architectures.

To solve this problem, He et al. [15] developed Residual Networks. Within the typical neural network structure, network layers are effectively performing two tasks simultaneously—the transfer of state alongside feature extraction/classification. The former requirement necessitates that the layer learn an encoding of its input, which is inefficient. Therefore, He et al. rewrite the typical neural network architecture to allow the network to focus on the latter task, while the architecture handles the former. This is done by applying a residual (or equivalently, difference) to the inputs. If a typical neural network layer takes input X, it will apply the layer L to produce $L(X)$. A residual network layer takes the input, then outputs the layer's contribution in summation with the original input to produce $L(X) + X$. Beyond the theoretical improvements to the "task" of the layer, it is also crucially important that this identity mapping for X in a residual layer allows the gradient to propagate backwards with its original magnitude. This ensures that the gradient is present at every layer with reasonable strength, allowing the calculations for error on the specific layer operation L to

be done with less noise. He et al. used this structure to produce a 152-layer network, which is almost an order of magnitude increase over previous methods. This result was enough to win ILSVRC in 2015, an industry-standard annual image classification competition, demonstrating the efficacy of the algorithm.

With regards to this thesis, residual networks have a very important property that a layer which is entirely zero (where $L = 0$) results in a layer that simply produces the identity. This means that it is easier to insert and remove residual network layers than in typical architectures. Along these lines, Huang et al. [19] introduce Stochastic Depth, which randomly drops layers during the training phase as a form of regularization and ensuring that every layer learns something different. This is very similar to the usage of Dropout in training, except entire layers are dropped. For the purposes of residual networks, this often helps the testing error.

Further expanding on He et al.'s work, Zaguruyko and Komodakis [39] assert that residual networks are equally suited to creating wide networks as they are for deep ones; their testing indicates that it is possible to use the residual network framework effectively for networks of comparatively shallow depth (16 layers). This is of particular interest because it indicates the difficulty of determining optimal network architectures; the benefits of wide residual networks are dependent on the specific classification problem. This thesis aims to improve on the often-confusing area of architectural optimization by allowing automatic hyperparameter tuning.

Chapter 3

Related Works

In this work, we are primarily interested in optimizing neural network architectures and other hyperparameters. Towards this end, we investigate current findings in the literature. Neural network architecture self-optimization is a very new topic, and many results are preliminary or incomplete. Nevertheless, they provide an important glimpse into the contemporary research space, and are highly motivational to the specific topic of this thesis. The aim of this section is to cover some of the existing work that specifically focuses on network optimization, and to provide some grounding for our contributions.

3.1 Parameter Deletion

Ever since neural networks have been developed, experts have wondered how to make them more efficient. The fixed initial structure required to train a network is one that is inherently overparametrized, because the minimum number of parameters needed is not known ahead of time. Further exacerbating the problem, neural network training is often slow and requires significant computational power, limiting the ability to test out how altering the number of parameters affects the network results. The natural solution is, therefore, to train an oversized network, and then whittle it down to size. Initial practices were based on heuristic deletion; that is, algorithms that deleted all weights w where $w < p$ for some low-pass filter p. These methods generally result in a sparse network (where the network has missing connections), which are difficult to represent and operate on efficiently.

LeCun et al.'s early work from 1989, *Optimal Brain Damage* [26], showed that these heuristic-based methods were inefficient and could irreparably destroy a network. They proposed a method based on error gradients that could more accurately find weights that can be removed with minimal perturbation to the final error. By what is effectively the butterfly effect, the deletion of a weight with small magnitude can actually prove to have a significant impact on the network. By taking the Taylor series of the error function to two terms, they show that it is possible to calculate the two-dimensional Hessian matrix representing the importance of each weight. For reasons of efficiency, LeCun et al. approximate the Hessian with a diagonal matrix, and gradually remove the terms with the smallest saliency. By retraining the network repeatedly after removing connections, they are able to show a significant improvement in performance despite requiring less parameters.

This was taken a step further by Hassibi et al. [12] in their followup work, *Optimal Brain*

Surgeon. By analyzing the Hessian matrix of typical networks, they show that the Hessian matrix is often nondiagonal, and that Optimal Brain Damage can often irreparably destroy small networks. They utilize the full Hessian matrix to better understand the interactions between each pair of weights. Hassibi et al.'s algorithm is among the most detailed methods shown to delete weights from a network, and they find that their algorithm is in fact optimal for specific small networks. However, the calculation of the Hessian is an $O(n^2)$ operation in both space and time, making it largely unsuitable for networks in the modern age, where n (the number of parameters) is often in the millions or tens of millions.

Within the last few years, an increasing amount of literature has been published on parameter deletion, especially as network complexity has grown at a pace that far outpaces the corresponding technological advancements. Han et al. [11] work on reducing existing architectures using a combination of heuristic-based deletion methods and weight regularization, and show that there is significant promise. These methods are more common and rely on deleting weights, which can help optimize performance when running on CPUs. They have limited application on GPUs and especially on convolutions. To solve this problem, Hu et al. [18] trim entire nodes and convolutions from the network, allowing better performance by fully removing them from the layers. They improve on Han et al.'s results by producing networks that are smaller and more accurate, and are even able to see some extremely small improvements over the untrimmed network for some of the largest networks. They hypothesize that this is due to optimizer efficiency. However, we note that despite showing that it is possible to reduce parameter count significantly, no modern work has measurably better overall results. These results are further supported by Murray and Chiang [30], who utilize the same methodologies on natural language modelling and observe similar performance. Instead, the goal is generally to minimize model size while keeping accuracy fixed (or reducing it slightly). In this thesis, we aim to more efficiently utilize existing capacity and achieve improved results over the architectures we start with.

3.2 Specialized Architectures

Another direction researchers have exploited to minimize the number of parameters required is specialized network design. Notable work in this field includes Squeezenet [20] by Iandola et al., which utilizes a number of space-saving tricks to produce a network which has 50 times less parameters. Courbariaux and Bengio further demonstrate that it is possible to constrain a network entirely to binary weights and activations (either +1 or −1) without significant loss in accuracy. Using this method, they are able to construct a convolutional neural network, and optimize it for CPU performance to achieve competitive results. These results are largely corroborated by Rastegari et al. [32], who also use a binarized network and significant usage of the XNOR operation to optimize a wider variety of modern networks. It is important to note, however, that the performance of these methods is still insufficient to reliably overtake GPU networks. Courbariaux and Bengio perform their training against "an unoptimized GPU kernel", while Rastegari et al. perform an efficiency investigation but do not discuss raw performance. While such approaches hint at future promise, in their current state they are more complicated and are still far from seeing general use in modern libraries.

On the other hand, it is not necessary to impose such harsh limits on the network in order to find areas of improvement. Google's Inception network [37], developed by Szegedy et al., has gone

through various iterations, which all involve complex pooling of different convolutional kernel sizes. In their 2016 update to the architecture [38], they focus on tuning the inefficiently large filter sizes used in the previous revision. They note that a 5×5 convolution is effectively the same (covers the same area) as two 3×3 convolutions while requiring more parameters (25 versus $9 \cdot 2 = 18$), dubbing this reduction as filter factorization. In the same vein, it is possible to reduce a 3×3 convolution to a 3×1 convolution followed by a 1×3 convolution, which requires a third less parameters.

There are various benefits to an increased number of smaller layers over one larger layer beyond parameter reduction. Reducing the number of layers allows for the increased application of non-linear activation functions, which are generally regarded as critical for learning complex problems. Furthermore, it allows an increased number of layers with the same number of parameters. Most networks are primarily limited by memory, especially as modern training algorithms require a Even though inference is generally more efficient, it can still remain a difficult problem for more constrained hardware; part of Squeezenet's contribution was the possibility of reducing a model to a size that could be run on modern FPGAs.

3.3 Network Expansion

The counterpart to parameter deletion is network expansion, which tries to add parameters and learning capacity to a network dynamically. One of the important works in this field is Cascade-Correlation Learning by Fahlman and Lebiere [6], which fixes a network and gradually adds single nodes to the network at a time. Their proposed network learns without backpropagation but rather through adding new nodes in order to correct for error. We derive some inspiration for our algorithm in Fahlman and Lebiere's interesting link between introduction of learning capacity and freezing the original network. The main difficulty with their specific algorithm is that it depends on adding individual nodes, which would be prohibitively slow to generate the network sizes that are common in the modern age. Additionally, we are unaware of any existing work that utilizes their findings in a way that is able to take advantage of convolutions.

An alternative way of thinking about this problem was tackled by Chen et al. They develop Net2Net [3], which takes a pretrained smaller network and allows a partial transfer of these learned weights to an expanded network. Noting that modern deep learning research usually requires the training of a number of different networks, they develop an architecture to minimize the amount of repeated computation, and are able to achieve improved results on the ImageNet dataset.

In general, network expansion is an understudied topic in the modern literature, and we believe that there is significant room for improvement, especially with regards to modern deep networks. We aim to provide a form of this by introducing dynamic network capacity, which allows layers to resize up to a fixed size during runtime.

Chapter 4

Methodology

In this section, we provide a high-level description of the algorithms we utilize. First, we construct a network structure that allows dynamic resizing and freezing, a method which has not been investigated on large scale before in the literature. We further develop this algorithm to allow for per-layer capacity tuning, which helps to ensure the optimal utilization of each layer. We then discuss the implementation process and its surrounding details.

4.1 Dynamic Network Capacity

Network design has almost always focused on preferring overparametrization; this principle is clear because underparametrized networks, by definition, simply cannot learn the problem. Our method involves defining the network in such a way that network capacity can be expanded with minimal overparametrization in a way that is unique in the literature. We reshape the underlying architecture to accept two parameters for each layer, representing the fixed capacity and training capacity. We define the fixed capacity of a layer as the number of nodes (or kernels, in the case of a convolutional layer) that are fixed from all training, and the training capacity as the number of nodes that are actively being trained. Crucially, these two capacities can change at any moment, and do not have to sum up to the full capacity of the network, meaning that some nodes can remain entirely unused. This allows a network to be undersized initially, but gradually gain the necessary capacity with minimal overparametrization. In particular, it allows per-layer expansion during runtime while keeping the weights that have already been trained, and performs this efficiently. While this requires upfront allocation of the maximum potential capacity due to library constraints, these requirements are not set in stone. Furthermore, due to increased speed and less chance of overfitting errors, it is generally desirable to train smaller networks if possible. As such, we rework well-known network architecture code to allow for dynamic capacity, which requires a higher-level framework that keeps track of all layer sizes to ensure consistent inputs and outputs. In particular, because we explore shortcut connections as seen in residual networks, this requires that we hold the necessary structure to ensure that the shortcut is projected to the correct dimension.

To determine how the network should utilize the ability to modify capacity dynamically, we focus primarily on expansion within this thesis. We track the moving average of error rates and gradually resize the network as the error plateaus (indicating that it has been trained to capacity). This process

requires the introduction of a few new hyperparameters to tune the definition of an error plateau, but allows some other ones, such as precise network sizing, to be masked away. We argue that this is a highly beneficial development for deep learning, as it represents a far more visible approach to network architecture as error rates are clear and interpretable. In contrast, beyond some vague sense that larger networks can learn harder problems, the motivation for choosing specific network sizes remains generally unclear. Within the field of residual networks alone, results have been published both demonstrating the superiority of prioritizing depth and prioritizing width, leading to potential confusion as to which one to emphasize.

4.2 Fixed Networks

Reducing parameter count is a difficult problem, and one that is especially complicated because it is difficult to remove weights from a network while maintaining efficiently dense connections. Rather than deal with the numerous details, we regard a different way of modifying the trainable network capacity by freezing parts of the network after they have converged. While this does not decrease model size, it improves the number of parameters that need to be trained, which is a potential point of efficiency. We note that this can be more useful than parameter deletion, which often results in sparse networks. Sparse networks are not very well supported by deep learning libraries, thus oftentimes necessitating that the "deleted" parameters remain in the model but are fixed to zero. We specifically avoid parameter deletion, in order to sidestep these issues. Our algorithm allows the network to utilize the capacity it has learned but avoids calculating a substantial number of gradients, optimizing the bottleneck of the training process. Additionally, this prevents the network from shifting excessively over time, a problem much like the covariate shift that is often covered by batch normalization. We implement this by modifying the layer-specific fixed capacity, which we begin to increase as the network learns the problem. This ensures that some basic knowldge of the problem is always retained and not susceptible to changes by each training minibatch. The specifics of how this fixed capacity increases over time are provided in the experiments chapter.

4.3 Layer-Specific Analysis

In the original paper on Residual Networks, He et al. analyze the relative strengths of each layer activation. To perform this analysis, they record the activations over a minibatch and perform aggregate statistics. In particular, they focus on the standard deviation of the layers as a measure of the relative amount of information in each layer, where the mean is less informative as it is largely influenced by the bias terms of the network. We aim to utilize this methodology to determine where extra capacity is useful. Noting that the standard deviation is not constant across layers, the layers with higher standard deviation are likely contributing more to the end result. While this is beneficial for accuracy, it may also mean that there is an opportunity to increase the capacity of these layers. He et al.'s analysis indicates that deeper networks have lower activation strength in general, which we can observe to also be smoother. We seek to encourage this property by expanding the residual blocks that have the highest activations, noting especially that they tend to occur when the network downsamples the image. Once the network has nearly reached full capacity on all layers, we begin to

selectively increase training capacity for the layers with the strongest activations to ensure that the increased capacity is deployed where it is specifically needed.

4.4 Implementation

We performed all of our experiments within Google's Tensorflow [1] framework. Tensorflow imposes a style of computation which is not immediately adaptable to our experiments, but it was nevertheless chosen due to its prevalence within the current literature. Its popularity has largely affected the number of open-source code samples available, and many current architectures have clear examples in Tensorflow. The thriving ecosystem of open-source contributions around Tensorflow proved to be a highly beneficial factor in providing a variety of existing architectures for experimentation.

Tensorflow operates in a slightly different way than many other libraries. Rather than allowing the user to chain together operations at random, it fixes a computational graph which defines the full model. Google's developers preferred this static model as it is generally well-suited to a lot of deep learning research, while also being flexible enough to allow for distributed computing (of crucial importance to a a cloud company like Google). This, however, poses an obvious problem with our algorithm, which is largely dynamic. Therefore, we had to develop a number of workarounds in order to interface with the static computational graph. While it is possible to use conditional blocks to disable parts of the graph, it is impossible to insert layers or other capacity during runtime. As such, the entire possible network capacity has to be allocated upfront, which potentially reduces the range of experimentation. This additionally means that while parts of the network can be disabled, they still take important parameter space which cannot be reallocated to other parts of the network.

Other deep learning software packages were explored briefly, but they either did not provide the necessary flexibility or lacked a reasonable set of tutorials/examples to facilitate the work within this thesis. For example, one of the more common tools in image-based deep learning has been Caffe [22], which boasts well-tuned performance as well as a public repository of models in the Caffe Model Zoo. Unfortunately, since Caffe is written almost entirely in C++, it is largely unamenable to testing and infrastructure development. Modifying Caffe to implement new training methods typically requires a significant contribution in C++, which requires an overhead not often undertaken except by researchers with significant prior experience. Furthermore, models are loaded in a fixed format, which hampers the ability to dynamically redefine networks. On the other side of the spectrum are libraries like Keras, which usually serve as a higher-level wrapper to other deep learning libraries. They were generally judged as being insufficiently expressive for the type of modifications we performed, so we considered other options.

All experimentation was performed on a GTX 1060, which was provided via a grant from Princeton SEAS. In recent years, GPU computation has become the standard for deep learning computation, as it can increase performance by nearly an order of magnitude. Particularly for models like modern residual networks, which can take days to converge to reasonable accuracy, it is nearly impossible to train neural networks on CPU servers. Tensorflow still utilizes the CPU extensively to coordinate training and perform a significant amount of calculations, but modern-day GPUs are nearly perfectly designed for the type of computations required for convolutions.

A recent glut of libraries aimed at helping automate the deep learning deployment process has led to a variety of different software packages. NVIDIA's CUDA and CUDNN libraries, both of

which are crucial for the performance of modern deep learning libraries, require a complex set of dependencies and installation procedures. To automate these processes, NVIDIA has recently released the `nvidia-docker` tool, which provides an abstraction on top of Docker that is designed to expose the GPU without requiring a complex installation method for the requisite GPU drivers. We use this library to deploy CUDNN v5, as well as the latest GPU drivers and Tensorflow version as of this writing (375.39 and 1.1.0-rc0, respectively).

As Tensorflow's interface is best utilized in Python, we performed some initial testing with the Jupyter application. Jupyter exposes a dynamic notebook interface that allows "cells" of code to be run in an interactive instance, which also shows outputs inline. Despite being relatively useful for basic prototyping, the largely static nature of Tensorflow's graph structure meant that for the larger tests, there was little to no developer-side improvement over traditional coding, which we eventually reverted to. Nevertheless, we note that Jupyter is a useful interface for demonstrating concepts, as many Tensorflow code examples online are in Jupyter `.ipynb` format. In particular, GitHub supports native inline presentation of Jupyter notebooks, which proved to be far more efficient than the typical workflow of downloading code examples, waiting for execution, and parsing terminal output which is often difficult to link to specific code sections. This allows us to cleanly adapt existing code, which helped significantly when doing intial development of the experiments.

4.4.1 Implementation Details & Notes

Throughout our experiments, we utilize the Adam optimizer developed by Kingma and Ba [23] as it allows adaptive training without requiring the careful learning rate tuning that is generally required for straightforward gradient-descent optimization. Many typical optimizers require handholding through epochs to achieve optimal results, while the default parameters for Adam allow far less supervision. In particular, hyperparameter search can often involve determining the correct timings of when to drop learning rate, which "slows" the network's training but also serves to stabilize it. As we aim for our algorithm to be as high-level as possible, this represents yet another dimension of optimization, which lies outside of the scope this thesis.

We also rely on Glorot and Bengio's Xavier initializer [8] to initialize the weights of the network, as it is a common improvement over typical random initialization, but is still relatively simple to use. This poses a small relevant side note to our algorithm; because only parts of the network are initially exposed, the initializer is potentially using incorrect values. Network initialization is crucial to achieving good results (one of the famous papers is this field is humorously entitled *All you need is a good init* [29]), and these initializers are dependent on the shape of the variable to determine properties like the variance of a random distribution. Due to the lack of dynamic initialization in Tensorflow, we do not investigate this issue further. Future work may include developing a custom initializer for this problem.

We fix portions of the network by using Tensorflow's `tf.stop_gradient` method. Support for freezing whole layers is a generally universal feature across deep learning libraries, but our investigation showed that none supported partial freezing—that is, the ability to train part of a layer while keeping the other part fixed. Notably, because we need to modify the amount of the layer that is fixed during runtime, it is impossible to decompose this problem into two separate layers. Our implementation involves deconstructing a variable into slices before reassembling it; a quick

demonstration in pseudocode is presented in Listing 4.1. This kind of workaround for a lack of inbuilt dynamicism is a typical example of what was necessary to build the desired structure into Tensorflow.

```
# x:              input, full-size variable
# fix_capacity:   what portion of x to freeze
# train_capacity: what porition of x to train,
#                 assumed to be greater than fix_capacity
f(x, fix_capacity, train_capacity):
    # slice X according to each capacity
    fixed = x[:fix_capacity]
    train = x[fix_capacity:train_capacity]

    # freeze fixed
    fixed = stop_gradient(fixed)

    # reassemble
    new_x = concat(fixed, train)

    return new_x
```

Listing 4.1: Variable Deconstruction

In following with a common Tensorflow workflow, we separate the testing and training models rather than running them under the same code but with different inputs. This has the key benefit of allowing inference testing to be completely independent of the training loop. In our case, this allows us to run inference on the CPU, as it is both less computationally intensive and less time dependent; we note that this could be further extended to allow the testing dataset to be run on a completely different machine. However, this practice also comes with a few downsides, as training error becomes largely separated from testing error. Without careful matching of the epoch count between the two methods, it becomes impossible to compare the two except during runtime observation. We did not perform this matching due to a lack of time, so we do not report training and testing errors on the same timescales for our results. We believe our current data is sufficiently representative of our algorithm, but note that this could allow more discussion on generalizability, which we expand on in a later section.

Chapter 5

Experiments

In this section, we present our findings. We expect that our algorithm will improve the training process for a regular deep learning user, both in the perspective of training performance and in final accuracy. This hypothesis is borne out in the smaller datasets, but remains somewhat inconclusive for larger datasets. Nevertheless, we note that there are significant points of interest that are raised as a result, and the potential for improvement on this algorithm is encouraging.

5.1 Function Regression

Our first experiment is relatively simplistic, but is also indicative of the basic algorithm's performance. In this experiment, we approximate the trigonometric sine function in the range of $[-2\pi, 2\pi]$. Our architecture for this experiment is extremely simple: it is merely a feedforward network with one hidden layer consisting of up to 500 nodes. This is sufficient capacity to learn the sine function with great accuracy, but can still be heavily affected by the training regime applied to it. We investigate the importance of the hidden layer's capacity by testing static networks of 100 and 500 nodes, then compare these results to a dynamically sizing network.

For this experiment, we split the network into fifths, and initialize them all before any training begins. We initially only let the network use the first fifth of its capacity, making it equivalent to the 100-node static test. We track the moving average of the error, and after it fails to rise within the last 10 batches, we proceed to increase the capacity of the network by a fifth, but also freeze the existing capacity. This serves to force the additional capacity to learn the mistakes of the existing capacity, rather than just adding additional parameters that would change alongside the original. We see the results of the experiment in Figure 5.1, in which our method is labelled "adaptive."

It is immediately clear that our method is able to converge to a much better solution than simple training methods. Importantly, we plot the y-axis on the logarithmic scale, so small improvements are in fact extremely significant. Firstly, we note that increasing the capacity of the network from 100 to 500 nodes does not have a significant impact on final error. In fact, the 500-node result would generally be considered worse, as it exhibits far noisier behavior. On quick inspection, the adaptive method is generally superior in every way. Surprisingly, however, the adaptive network is initially slower to train compared to the 100-node network, despite utilizing same capacity. We suspect this difference has to do with the potential variances in optimizer and initializer performance discussed

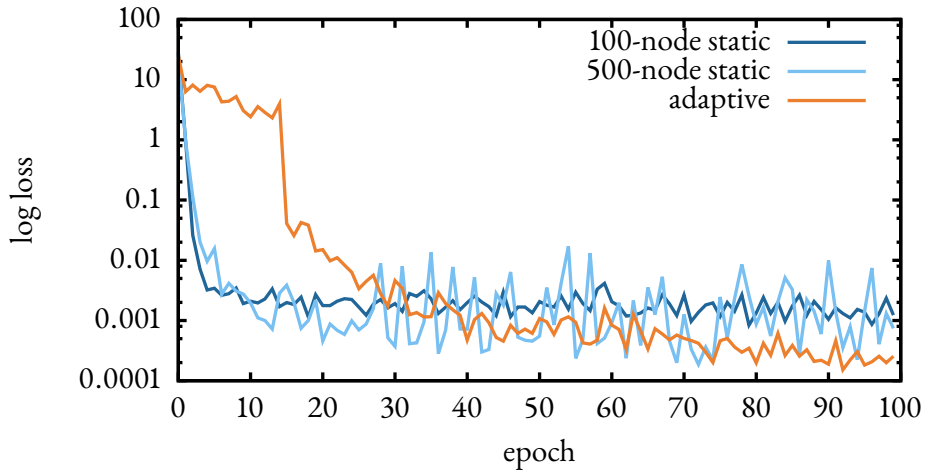

Figure 5.1: Sine function approximation by different methods.

above. Regardless of its early-stage performance, we can see that it benefits tremendously from increased capacity, and overtakes the both of the static networks by around epoch 35. There are some small spikes in error which we attribute to the sudden increase in capacity, but overall the performance is consistently better than that of the static methods.

Table 5.2: Final-10 errors for various methods.

Network	Mean	Standard Deviation
100-node	0.00138	0.000389
500-node	0.00234	0.003261
Adaptive	0.00024	0.000084

To measure these results quantitatively, we consider both the average and the standard deviations of loss over the final 10 samples. These results are presented in Table 5.2. We can see that the noisier results of the 500-node network are actually noticeably worse when averaged—it has nearly twice the error of the smaller 100-node network. Furthermore, the standard deviation is an order of magnitude larger over the 100-node network., which is extremely poor as it is larger than the average error. In contrast, the adaptive method exhibits both lower mean and standard deviation in the long run. This indicates an improvement not just in learning capacity, but also in stability, which is a crucial property for training as noisy behavior is indicative of a number of other problems. Chief amongst these is the simple problem that noisy behavior makes it difficult to decide when an experiment has concluded. In any case, throughout this experiment, the adaptive method has offered significant improvements in performance over either static network.

5.2 MNIST Classifier

Modern deep learning algorithms have generally tended to be developed for image classification purposes, in part due to the original usage of convolutional neural networks. LeCun et al.'s original work with CNNs [25] was in designing a classifier for the MNIST dataset, a collection of monochrome handwritten digits. These images have been preprocessed for regularity, and have all been resized to a standard 28×28 resolution. MNIST is a well-regarded small image classification dataset which serves as a useful benchmark for this algorithm.

Tensorflow includes MNIST support as part of the base installation as part of its example code, so we are able to rely on a simple interface to download, load, and process the image data according to standard image augmentation purposes. For the purposes of this experiment, we follow the example convolutional neural network provided by Google [9], and modify it so we can apply our expansion algorithm. This is a typical structure, with two convolutional layers utilizing 7×7 kernels, and then a fully-connected layer of 1024 nodes. While this is far from the best known architectures for MNIST, it serves as a good baseline and is easily accessible. Once again, we apply our methodology of training the network in portions. This time, due to the increased capacity of the network, we instead train it in tenths, once again expanding when the moving average of accuracy begins to stall or decrease.

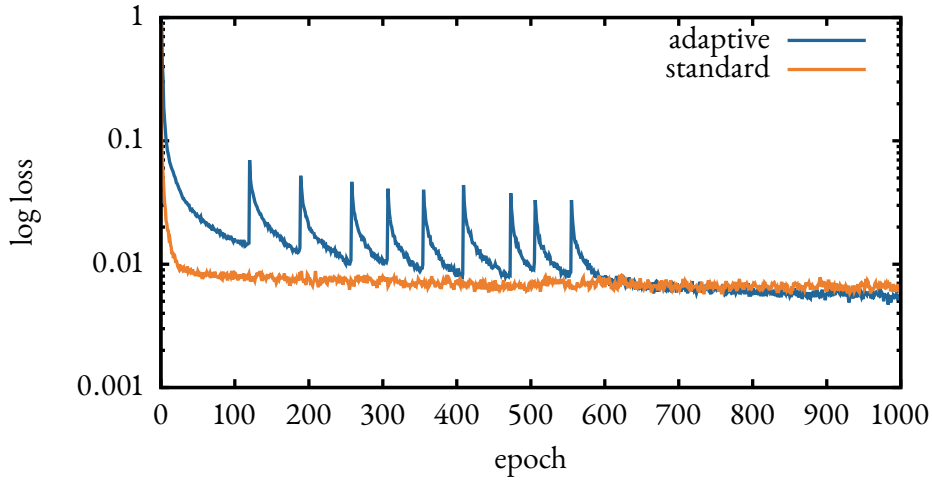

Figure 5.3: MNIST loss trained by different methods.

We show the results of the MNIST classifier in Figure 5.3. In this experiment, there are significant spikes to the loss when new capacity is added, indicating that the extra capacity produces a significant shock on the network. We hypothesize that this is due to the fact that multiple layers are all increasing in capacity simultaneously, which leads to a much more pronounced change in outputs. Whereas the previous experiment only involved a single layer, the interactions of additional units that are connected to the original network changes the dynamics significantly. This still shows an improvement over the

standard network, although the difference is much smaller than previously demonstrated. This is likely due to the fact that the error is very low in both examples; typical algorithms achieve over 99% accuracy on MNIST. It appears that the adaptive network is still improving over time but may be limited by the length of the experiment, while the standard network has converged to its best potential. We were not able to test this theory more fully due to time constraints, but leave it as a potential point of interest. We also report the mean and standard deviation statistics for this experiment in Table 5.4. This corroborates the close performance between the two networks, but also demonstrates the improvements our adaptive method produces over a standard network of the same final size. Both the error and the standard deviation are lower, indicating a mild improvement.

Table 5.4: Comparison on MNIST.

Network	Mean	Standard Deviation
standard	0.00686	0.000369
adaptive	0.00633	0.000248

5.3 CIFAR-100

One of the common modern image classification datasets is CIFAR-100, a set of 60000 images collected by researchers at the University of Toronto. It consists of 20 classes, each with 5 subclasses. For each of the 100 subclasses, there are 500 training images and 100 testing images. The images are in color, but are of low resolution at 32×32; the small size of the dataset makes it especially attractive as an experimental problem; a few sample images are shown in Figure 5.5. Larger image classification datasets exist, such as the commonly used ImageNet, but due to its over 150GB download size and consequently longer training times, it was not considered for this thesis. Most modern deep learning papers include results on both CIFAR-10 (a smaller version of the same problem) and CIFAR-100. Because state of the art performance on CIFAR-10 is over 90% which leads to a closer and less separable grouping of experimental results, we choose CIFAR-100. In doing this, we hope to avoid the problem seen on the MNIST dataset, where it is extremely difficult to improve on results that are already nearly perfect.

A particular point of interest with CIFAR-100 is that there are relatively few images per class. This means that it is a dataset for which overfitting is a critical concern. Typical algorithms, without any specially designed methods, can often achieve around 60% accuracy on the testing dataset. This, however, tends to represent a hard limit. Training accuracy will usually hit nearly 100% accuracy, meaning that the network has learned all it can from the training dataset. The difference between testing and training accuracy, especially with the limited data available, is the primary area of improvement for modern algorithms.

We perform a base experiment on a residual network with 30 residual blocks (60 layers), with channel sizes starting at 16 and increasing by a factor of 2 every 10 layers. This structure was originally built for the ImageNet dataset. However, we adapt it to CIFAR-100, which has much smaller images and therefore requires less capacity in the network. Again utilizing common practice, we use

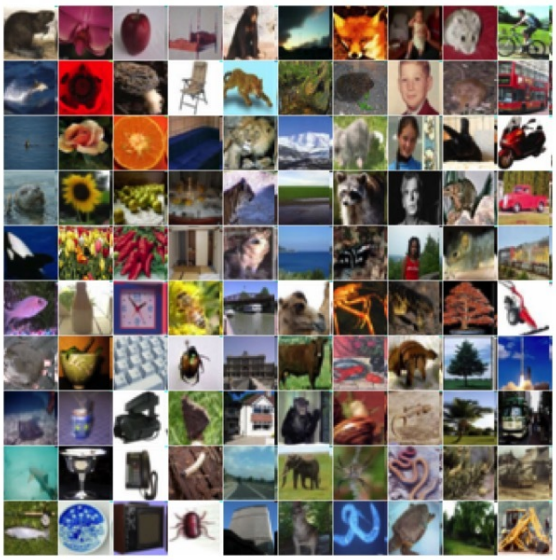

Figure 5.5: An example set of images from CIFAR-100. From [13]

minibatches of size 128, and sample training accuracy every 100 steps. We allow training to proceed until error rates drop inperceptibly over a reasonably significant period of time. We see the expected training accuracy go to 100%, but the testing accuracy hovers around 65%, which is also in line with expectations.

In our first experiment, we again use the same basic algorithm first developed for the sine experiment and let the network learn in tenths. Observing the error rates from the static experiment, we let the network expand every 25 samples (representing 2500 steps). This is an extremely restrictive network, and our results show that this is perhaps overconstrained for the problem; the training error peaks at 80% while the testing error hovers around 58%. While this is a noticeably poorer result, it is still interesting to note that the generalization appears to be better in this experiment, as the difference between the training and testing error falls from 35% to 22%. Even in the prior static baseline, there was never a comparably small difference between the training and testing error. This leads us to consider how we can better understand the generalizability of our algorithm, which we discuss in the following chapter.

This performance is summarized in Figure 5.6. We include results from our final experiment, which we discuss below. Unfortunately, these results are based on the training loss and accuracy rather than testing accuracy, due to a few bugs in the software. Nevertheless, we provide this figure to demonstrate the difficulties in optimizing the standard network, as the adaptive methods are unable to achieve the same results. This also furthers our discussion on generalization. While the training error does not decrease as significantly, we are still able to tune the algorithm to surpass the standard network in our second experiment.

We also note that that accuracy on the dataset is just one measure of precision. During our experiments, we also log the cross-entropy loss, which provides a sense of not just how likely the network was to get the right answer, but how confidently it did so. That is, while the accuracy is determined by picking the category with the highest activation, this choice may have been a very

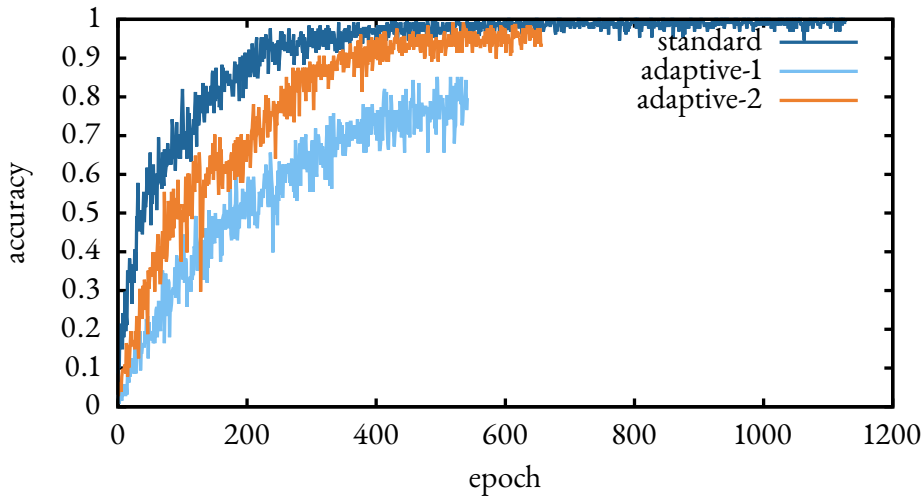

Figure 5.6: CIFAR-100 training accuracy.

close decision. For example, on a binary classification problem with classes 0 and 1, where the correct response is nearly always 1, a network with constant output activations $[0.49, 0.51]$ would have extremely low error but high cross-entropy loss. Using this metric, we note that the cross-entropy loss is lower at the same training error for our methodology, indicating that its outputs are more regularized.

To improve the performance of this algorithm, we perform another experiment where we gradually allow a limited proportion (50%) of the network to unfreeze over time. We believe that this can be helpful in unlocking some of the limited capacity in the network; in particular, we note that the first few convolutional layers are fed into a very narrow pipeline of only 16 filters at full capacity. Our training algorithm limits this further, which may overly constrict the flow of information through the network. By leaving at least half of the capacity to be trained, we attempt to allow sufficient flow for better accuracy. This hypothesis is borne out by the results, which show that the new network (entitled "Adaptive-2" in the figures) is able to demonstrate a small but significant improvement over the standard network. We achieve a stable testing accuracy of 68%, but interestingly, our training accuracy stabilizes at a lower point at around 95%. The results of our experiments are summarized in full in Table 5.7.

5.4 Performance

We note that our algorithm involves nearly no overhead over the original architecture; a simple timing benchmark over 1000 epochs of MNIST indicates a performance difference of 3.5% (37.9 seconds versus 36.6 seconds), which is well within the margin of error. Furthermore, by limiting the capacity

Table 5.7: CIFAR-100 results

Network	Testing Accuracy	Training Accuracy	Training Loss
Standard	0.659	0.9927	0.0221
Adaptive-1	0.582	0.8001	0.6595
Adaptive-2	0.681	0.9557	0.1379

of the network, we are able to achieve far faster initial training. The initial timing experiment was performed by applying the algorithm but forcing it to use the full capacity of the network initially; this is far from the original intent. By utilizing it in the same way as developed for the experiments, the first 1000 epochs of MNIST actually take 11.9 seconds, which is a huge improvement. This performance boost can make a significant difference over the course of a training cycle.

While the algorithm takes more epochs to converge, the increased speed of working through the initial epochs is a significant boon. In general, any decrease in performance can likely be attributed to the more intricate methods required to perform basic variable operations, recalling Listing 4.1. These are generally considered to be minor; in fact, for most researchers, the choice of deep learning library is rarely made for performance reasons, especially for single-GPU servers. Nearly all of the time spent is within the intricacies of the CUDNN module which interfaces directly with the GPU. Our algorithm adds effectively no stress to the GPU, and can speed up training even when running at full capacity by fixing portions of the network, thus eliminating the need to perform the expensive gradient calculations.

Chapter 6

Discussion

Our results are highly promising, and additionally show some other tendencies, which we highlight in this chapter. Our algorithm appears to work as a regularizer, ensuring that a network does not devolve into a suboptimal state. We are also able to see an improvement in generalization compared to standard training regimes, which is a key benefit. Finally, we list a number of limitations and provide guidance and direction for potential future work.

6.1 Regularization

An important aspect of fixing part of the network capacity is that it prevents the network from diverging significantly. This is, in effect, a form of regularization, which we can see most clearly in the function regression results. By fixing the majority of the network, the capability of the network to produce noisy results is far more limited. This may be an important property even if the network is unable to achieve significant improvements on a dataset, stability is an important goal of any training algorithm. This stability may also allow the algorithm to perform better, as the moving average becomes less susceptible to noise. We believe that this virtuous cycle can be further exploited by a more advanced extrapolation of the error curve. Anecdotal results have indicated that the algorithm does indeed pick more opportune times to expand the network as the fixed capacity increases (and with it, the strength of the regularization), but further work would have be done to verify this effect.

Furthermore, this corroborates the known literature that network capacity is being used inefficiently. The ability for a network to function well despite only being able to train on a fraction of its capacity indicates a potential overparametrization of the original network. It would be interesting to apply parameter deletion methods to the frozen capacity, as they generally try to involve minimal perturbation to the network. This would allow an efficient network to be constructed in-place, without requiring a significant amount of retraining. We had previously attempted a version of the algorithm that gradually unfreezes the network as an attempt to improve late-stage error, but were unable to detect any major differences between this algorithm and standard training. This indicates that the retraining process during most parameter deletion methods may be unnecessarily noisy, and we believe that our fixed capacity may help solve this problem. By utilizing extra capacity to correct for and smooth the errors of the fixed portion, the network is given what is potentially a simpler problem. We note that this is different from boosting or ensemble architectures due to the high degree

of interconnection—as capacity is introduced, it is fully connected to all of the available capacity of the previous and next layers. This means that the learning is far more organized as a single unit rather than as small substructures.

6.2 Generalization

As we noted briefly in the experiments, one of the interesting trends of the adaptive network was that it tended to overfit less, even if the full results were not as good. Interestingly, its generalization performance was better than the standard network at the same training error, indicating that it had potentially learned the problem better under a certain metric. Part of the hope with the algorithm is inspired by the idea that using less parameters prevents overfitting and allows for a model to be more general; this seems to be borne out by the results. Through various versions of the algorithm, we were able to perform improvements to the testing error consistently; this is despite the standard network having fully utilized the training set, reaching 0% error. Our methods have shown consistent improvement, and most interestingly, our final CIFAR-100 test has lower training accuracy but higher testing accuracy than the standard network.

This is an interesting improvement, as the imperfect training accuracy may indicate that further gains could be made on both accuracies, perhaps by better informing the optimizer. Crucially, the final fully-connected layer's capacity is currently modified along with the last convolutional layer, which could significantly impact the outputs. In keeping with modern trends that focus on maximizing convolutions throughout the network, residual networks do not depend significantly on fully-connected layers. However, we believe that for the purposes of this algorithm, our results demonstrate the potential need for additional capacity that does not change as often. Along with our good generalization results, performing further higher-level architectural optimizations may help improve results even further.

6.3 Limitations and Future Work

There are number of topics that were unfortunately outside the realm of reasonable exploration during the course of this thesis. Many of these pertain to the specifics of our algorithm, which could see significant fine tuning. While our results are generally good, we believe that that are still major gains to be achieved by continuing along the same directions established by our work. This thesis provides an interesting result, but also opens up a variety of questions for future investigation.

As noted, whenever possible, we have preferred to maximally utilize the currently available methods rather than performing significant rewrites specific to our problem. This helped our work maintain its focus on the specifics of improving deep learning training, rather than work on significant reimplementations that would likely introduce new bugs into the system. At the same time, this could be an area of future work, as we have briefly discussed the potential limitations of using off-the-shelf initializers and optimizers. These restrictions are generally due to the lack of knowledge within the system, which could significant effects on the training capability of the network. We noted that our initial sine function approximation network began to learn far slower than expected, which we conjectured to be due to poor initialization. Furthermore, we were not able to independently verify

whether the fixed weights were entirely frozen. Due to the intracacies of the optimizer, it is possible that momentum terms, or just other modifications from standard gradient descent led our network to keep changing even when it was fixed. We expect that Tensorflow's ability to stop gradients from flowing through the fixed sections should have effectively done the job, but it may have taken a few more epochs. In order to solve these issues, we would have to perform custom implementations of Adam, Xavier, and perhaps other components as well that are aware of the capacity limitations we impose. This would likely complicate the codebase significantly, as Tensorflow operations are not as easy to develop. Some algorithmic questions also remain, such as the correct initialization values for added capacity. These would involve significant new testing and theory to determine.

We also believe that sparsity is an interesting topic but were unable to cover it within this thesis. In his work on spatially-sparse convolutional neural networks [10], Graham noted that there are potential improvements in architecture by performing sparse convolutions. Tensorflow does not support such functionality at the moment, although it appears that they may be planning its development for the future [36]. For our experiments, we continue to rely on densely-connected convolutional layers. Apart from the natural computational efficiency, we note that sparse networks are generally utilized in problems where the problem is seen as less compact or able to exploit the sparse connections—such is not often the case for image classification, which is our primary subject in this thesis. We do note, however, that this would be a very interesting way of implementing dynamic network capacity that extends beyond our current implementation. Importantly, this may allow the network to suffer less shock as additional capacity is added by initially minimizing the number of connections between the original or fixed section and the newly-added training section. In this way, the sections can be trained somewhat like an ensemble of networks that gradually begins to learn some capacity for communication. Therefore, controlling this dynamic would be an extremely powerful tool.

Another area of interest is the amount of dynamicism in the network architecture. Srivastava et al. explore Highway Networks [35], which allow learned connections to form between any two layers. This is obviously a far more complex architecture, requiring additional connections in the network, but may be thought of as a higher-level abstraction over residual networks. Allowing the network to develop not just in per-layer capacity but also in layer connections could allow better mutability. The results regarding the training accuracy have shown that fixing significant capacity limits the ability of the network to overcome some of the errors caused by limitations of capacity. However, it is possible that increased connections between each layer would provide the necessary adaptations to learn the problem well, without adding a significant number of parameters to the whole network.

While the hyperparameters for our algorithm were generally chosen on inspection of the testing baseline, we note that it may be possible to develop a reasonable set of defaults for an average user. This would be highly beneficial, as it further removes the necessity of tuning. Apart from edge cases which would be known to the user, it seems that basic analysis can indicate when convergence is beginning, and the algorithm can adjust accordingly. A improved algorithm would perhaps entail a more detailed analysis of previous errors beyond a simple moving average, which would allow it to be smarter about when a resize is necessary, as opposed to occasionally falling for noise in the error. Many small tweaks could be developed as a result of more extensive testing of the algorithm to better understand its behavior on a wider range of learning problems.

Another direction we see is in the potential for live user intervention during training. In general, most modern methods do not allow any changes to the architecture, meaning that if certain parameters

are set poorly but go unnoticed, significant time can be lost as the network will have to start training from scratch. Technically, this functionality is available in a very crude sense in our current software, as the data is mostly saved into checkpoints that could be loaded and overwritten. This means that by overwriting the current state variables of the algorithm, subsequent runs would then adopt the updated values. Especially with Google's Tensorboard software, which allows a Tensorflow network to show its computational graph, log various properties, and much more, we see the potential for users to gradually tune a network on-the-fly. In conjunction with our algorithm providing suggestions on network changes, it would be interesting to allow a more technical user to query specific statistics about the network, then make decisions on tuning without necessitating a new and costly training cycle. This kind of fine-grained architectural control during runtime is completely new to the literature, so we see our work as a foundational first step.

Chapter 7

Conclusion

This thesis introduces the concept of dynamic network capacity, which allows for the fine-grained tuning of layer capacities. No work in the literature has done this before. We develop a training algorithm to utilize this structure, and combine it with fixed networks to encourage additional learning. Our algorithm leads to large improvements on all of the the datasets we test on, and we observe potentially significant gains in efficiency. Essentially, we improve on existing network architectures with minimal extra human intervention, and we believe that there are more accessible improvements in this direction that would even further maximize performance. The success of our method allows us to abstract away the opaque hyperparameters involved in modern training and replace them with visible, understandable variables.

Deep learning is an extremely active area of development, and it is likely to proliferate in even more fields in the future. Its recent explosion as a popular research field has resulted in a flurry of new knowledge being created, which requires better and more consistent training processes that are transparent to the user. Our work can be applied to a wide range of existing network architectures, and is a significant boost to the architectural flexibility without a dramatic increase in complexity. We believe our algorithm has far-reaching implications and future possibilities in making neural network models both easier to interface with and more efficient. Our framework for working with dynamic network capacity pioneers a field of exploration that has been generally dormant in the literature, and we demonstrate that there are significant benefits to this problem in the modern day.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

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

[3] Chen, T., Goodfellow, I., and Shlens, J. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641* (2015).

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

[5] Courbariaux, M., and Bengio, Y. Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[6] Fahlman, S. E., and Lebiere, C. The cascade-correlation learning architecture.

[7] GIMP. Convolution matrix. `https://docs.gimp.org/en/plug-in-convmatrix.html`.

[8] Glorot, X., and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Aistats* (2010), vol. 9, pp. 249–256.

[9] Google. Deep MNIST for Experts. `https://www.tensorflow.org/get_started/mnist/pros`. From the Tensorflow documentation, r1.1.

[10] Graham, B. Spatially-sparse convolutional neural networks. *arXiv preprint arXiv:1409.6070* (2014).

[11] Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems* (2015), pp. 1135–1143.

[12] Hassibi, B., Stork, D. G., et al. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems* (1993), 164–164.

[13] Hastie, T. CIFAR-100 image database. `https://web.stanford.edu/~hastie/CASI_files/DATA/cifar-100.html`.

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

[15] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 770–778.

[16] Hecht-Nielsen, R., et al. Theory of the backpropagation neural network. *Neural Networks 1*, Supplement-1 (1988), 445–448.

[17] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* (2012).

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

[19] Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. Q. Deep networks with stochastic depth. In *European Conference on Computer Vision* (2016), Springer, pp. 646–661.

[20] Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 1mb model size. *arXiv preprint arXiv:1602.07360* (2016).

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

[22] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (2014), ACM, pp. 675–678.

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

[24] Lebedev, V., and Lempitsky, V. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2554–2564.

[25] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[26] LeCun, Y., Denker, J. S., Solla, S. A., Howard, R. E., and Jackel, L. D. Optimal brain damage. In *NIPs* (1989), vol. 2, pp. 598–605.

[27] Lewis-Kraus, G. The great a.i. awakening. `https://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html`.

[28] Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML* (2013), vol. 30.

[29] Mishkin, D., and Matas, J. All you need is a good init. *arXiv preprint arXiv:1511.06422* (2015).

[30] Murray, K., and Chiang, D. Auto-sizing neural networks: With applications to n-gram language models. *arXiv preprint arXiv:1508.05051* (2015).

[31] Nair, V., and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010), pp. 807–814.

[32] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision* (2016), Springer, pp. 525–542.

[33] 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. Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[34] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

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

[36] Staker, J. Feature request: Implementing spatially-sparse conv networks in tensorflow. `https://github.com/tensorflow/tensorflow/issues/1604`. From the Tensorflow Github repository.

[37] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), pp. 1–9.

[38] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2818–2826.

[39] Zagoruyko, S., and Komodakis, N. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).

This paper represents my own work in accordance with University regulations.
/s/Clement Lee