Contents

Chapter 1

Introduction

Deep learning has exploded in recent years as the forefront of what the New York Times has referred to as the "A.I. Awakening."

< a whole lot more to go here >

AlphaGo shocked experts worldwide when, in a best of 7 series against 9-dan Go master Lee Sedol, it was able to win 4-1. Go had often been seen as an example of a problem where humans had an unsurmountable advantage. In the modern day, chess programs are able to brute force enough possibilities on a typical consumer laptop to defeat professionals. However, Go has a far more complex move space: an cursory estimate indicates that there are on the order of 19! possible games. Deep learning proved capable of solving the problem of judging the quality of a Go board position, which was a groundbreaking result in a field where even the best programs were unable to challenge low-ranked professionals. Surprisingly, the developers of AlphaGo noted that "AlphaGo evaluted thousands of times fewer positions than Deep Blue did in its chess match against Kasparov." [24]

The promise of 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 general,

< discuss performance concerns with deep learning and therefore the motivation for this thesis by optimizing that >

Chapter 2

Background

In this section, we provide a general introduction to the relevant basics of deep learning.

2.1 Neural Networks

The foundational principle of neural networks is, in its purest form, inspired by the biology of the human brain. The field of AI has often modelled new algorithms after biological phenomena; in this field, genetic algorithms are based on evolution and particle swarm optimization is based on social behaviors. The history of neural networks dates back to the beginnings of artificial intelligence research, and from those times a few 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 (outputs) with a strength that is a function that is 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. 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 to 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 [8]. 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).

The other key insight of neural networks is backpropagation [12], 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,

$$\frac{\partial \text{ Cost}}{\partial a_i} =$$

Modern training methods are far more advanced, but still rely on the basic algorithm described here, which is often termed gradient descent. The main

< *describe modern usage of backprop* >

LeCun et al.'s seminal work in this field, *Gradient-Based Learning Applied to Document Recognition* [19], 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

$$(f * g)(t) = \int_{0}^{s} f(r)g(t-r)dr$$

In this context, 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, we 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, 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.

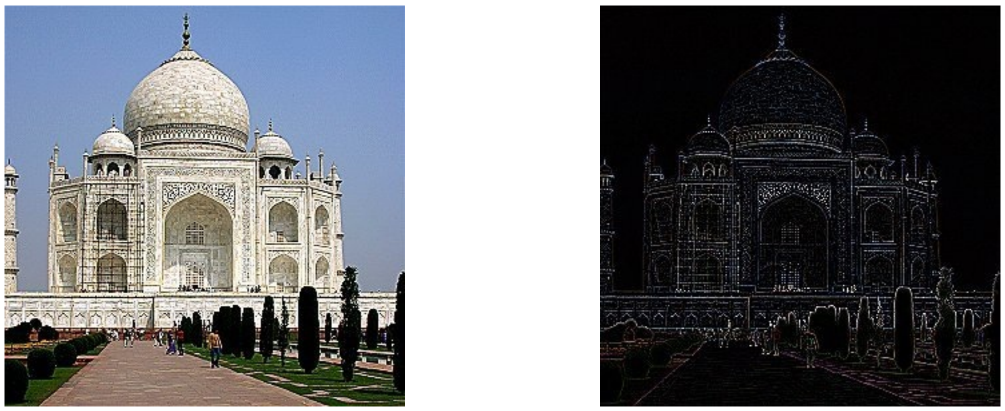

Figure 2.1: A demonstration of an edge-detecting convolution, from the GIMP User's Manual. [4]

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; applying this across a variety of filters demonstrates 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 for the sake of completeness.

< *talk about batch normalization, dropout, different activation functions* >

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. [11] developed Residual Networks. The insight in this work is that typical 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. He et al. rewrite the typical neural network architecture to allow it to merely learn the latter task, and apply this as a residual (or equivalently, difference) to the inputs. That is, 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. [14] 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.

Further expanding on He et al.'s work, Zaguruyko and Komodakis [28] 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, and the

< talk about the wide varieties of residual network architectures, and how it's hard to pick the right one >

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. To the best of our knowledge, neural network architecture self-optimization is a very new topic, and many results are preliminary or perhaps somewhat 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. Making the problem worse, neural network training is often slow and requires significant computational power, limiting the ability to test out differing numbers of parameters. The natural solution is, therefore, to train an oversized network, and to somehow 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* [20], showed that these heuristic-based methods were inefficient and could irreparably destroy a network. He proposed a method based on error gradients that could more accurately find weights that contribute

< *finish this description* >

By what is effectively the butterfly effect, the deletion of a weight with small magnitude could actually prove to have a significant impact on the network.

This was taken a step further by Hassibi et al. [9] in their followup work, *Optimal Brain Surgeon*. By analyzing the Hessian matrix of the network,

< *actually describe the Hessian, or do this with LeCun's example above* >

Hassibi et al.'s algorithm is among the most detailed methods shown to delete weights from a network, and they show 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. At the same time,

Within the last few years, the literature on parameter deletion is beginning to see more activity, especially as networks grow in complexity at a pace that far outpaces the corresponding technological advancements.

< describe modern successes with using heuristic-based deletion, and some preliminary research on using optimal brain damage with large networks >

3.2 Specialized Architectures

Another key direction taken by researchers is to design the network specifically to minimize the number of parameters required. Notable work in this field includes Squeezenet [15] by Iandola et al., which utilizes a number of space-saving tricks to produce a network which has 50 times less parameters. Perhaps even more interestingly, Courbariaux and Bengio show that it is possible to constrain a network entirely to binary weights and activations (either $+1$ or -1) without significant loss in accuracy. They are able to construct a convolutional neural network in this way, and optimize it for CPU performance to achieve competitive results. These results are largely corroborated by Rastegari et al. [23], 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 show promise in the future, they are more complicated and are still far away 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 [26], 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 [27], 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 beyond parameter reduction. It allows the increased application of nonlinear 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

< describe early (1990s) approaches to network expansion, there's another modern paper that does a little expansion but not as interesting perhaps >

Chapter 4

Methodology

In this section, we provide a description of the algorithms we propose. We construct a network structure that allows dynamic resizing and freezing, which has not been investigated on this scale before in the literature. We further develop this algorithm to per-layer capacity tuning, which we note can help ensure that every layer is being utilized optimally. We then discuss the process of our implementation and a variety of 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 expandewith minimal overparametrization. way that, to the best of our knowledge, is unique in the literature. This allow 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, it is generally desirable to train smaller networks if possible, due to increased speed and less chance of overfitting errors. As such, we rework well-known architectures to allow 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 shortcuts connections as seen in residual networks, this involves holding the necessary structure to ensure that the shortcut is projected to the correct dimension.

For determining how the network should begin to utilize the ability to modify capacity dynamically, in this thesis we focus primarily on expansion. 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 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 a very murky situation.

4.2 Fixed Networks

Parameter count is a difficult problem to solve, 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 required here, 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, as sparse networks are not very well supported by deep learning libraries, thus oftentimes necessitating that the "deleted" parameters remain in the model but fixed to zero. This 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. This additionally prevents the network from shifting excessively over time, a problem much like the covariate shift that is often covered by batch normalization.

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, we note that these layers are likely producing more contribution to the end result. While this is beneficial, 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.

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.

As was noted before, 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 libraries 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 [16], 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 [17] 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 [5] 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* [21]), 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 metrics, 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

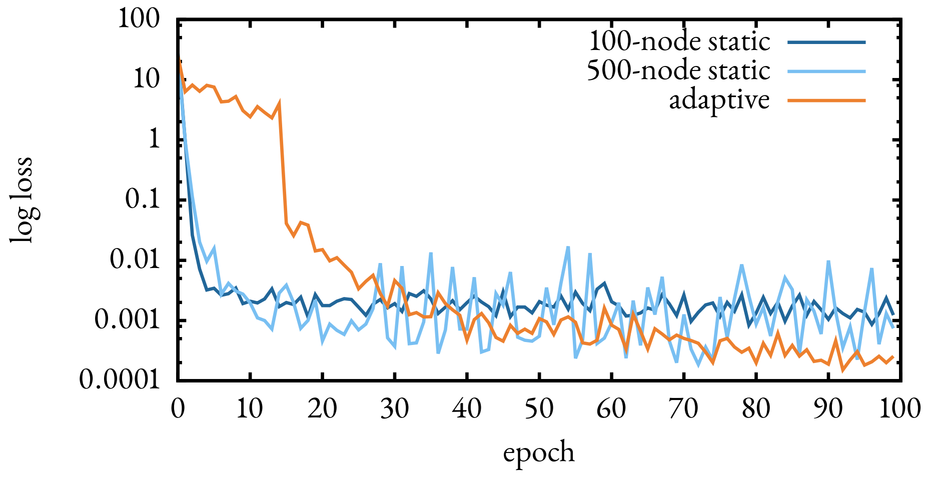

Figure 5.1: Sine function approximation by different methods.

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 [19] 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 [6], 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.

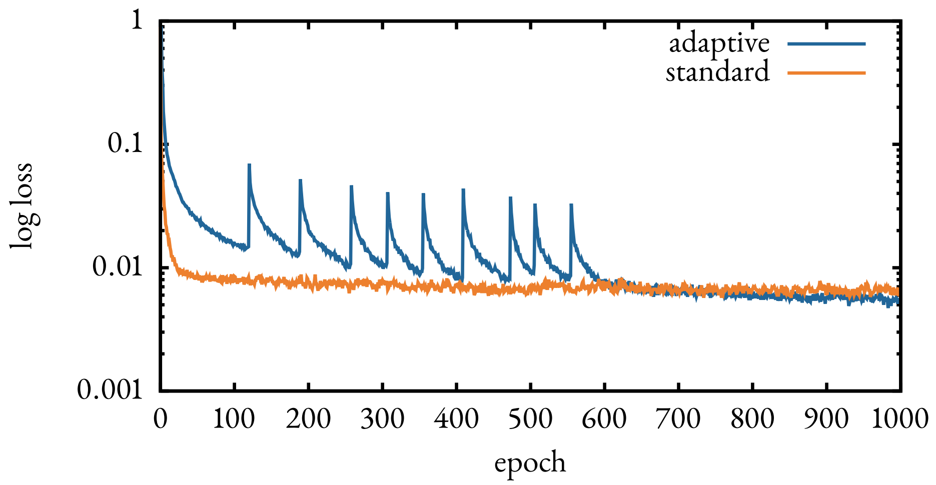

Figure 5.3: MNIST loss trained by different methods.

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; we choose CIFAR-100 as state of the art performance on CIFAR-10 is over 90%, leading to a closer and less separable grouping of experimental results. 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, as 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 two every 10 layers. This structure is an adapation of the original residual network, which was built for the ImageNet dataset, to CIFAR-100, which has much smaller images and therefore requires less capacity in the network. Again utilizing

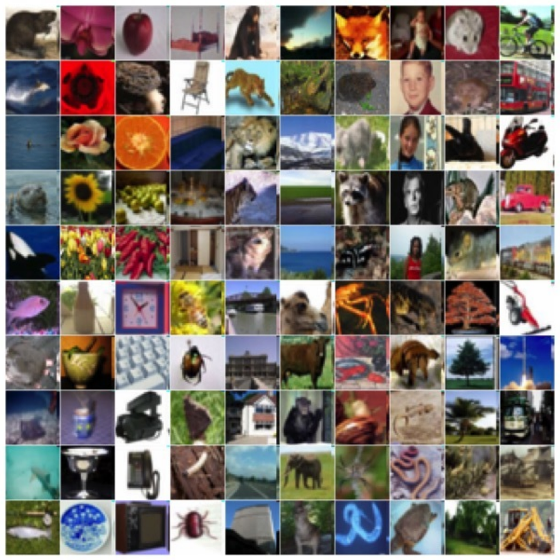

Figure 5.5: An example set of images from CIFAR-100. From [10]

common practice, we use 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; while this requires human judgment, it was also performed due to general time constraints. 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 what might be the beginnings of a generalizability metric, which we discuss in the following chapter.

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, it may have been a very 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.

These results are summarized in table

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 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 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

< more to come as results become a little clearer, most of my time is going into CIFAR-100 experiments working >

6.1 Regularization

An important part 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.

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 that

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.

< talk about how fixing network capacity may act as a regularizer >

6.2 Generalization

< discuss how generalization was better with trained model >

Chapter 7

Conclusion

7.1 a brief summary of results

< *what the section title says* >

7.2 Limitations

There are number of topics that were unfortunately outside the realm of reasonable exploration during the course of this thesis,

7.3 Future Work and Further Notes

In his work on spatially-sparse convolutional neural networks [7], 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 [25]. 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, and having control of this dynamic would be an extremely powerful tool.

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.

Another further direction we see is 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.

< talk about shortcut connections, different trainable parameters, also extend literature to other directions of optimization >

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] 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).

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

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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).

[14] 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.

[15] 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).

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

This work represents