ACCELERATING DEEP NEURAL NETWORKS

Aumit Leon

Adviser: Professor Andrea Vaccari

A Thesis

Presented to the Faculty of the Computer Science Department

of Middlebury College

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Arts

May 2019

ABSTRACT

Deep neural networks (DNNs) are a class of increasingly popularized machine learning methods that are applied to a vast number of domains. From predicting weather patterns to accurately classifying images, DNNs are coming to define the state of the art in an increasing number of applications. The highly accurate predictions produced by deep learning methods are made possible by the data provided to models during training and the efficient use of computational devices such as graphics processing units (GPUs). While access to data in domains such as image classification is increasing, the typical computational power required to train such large networks is not ubiquitous. This not only poses the technical challenge of training large DNNs, but also adds a constraint on the adoption of DNNs and machine learning methods by users with limited computational resources. The democratization of DNNs would increase the accessibility of these methods across device topologies, which effectively increases reproducibility and oversight. In recent years, efficient parallelization methods have been developed and introduced to tackle this issue. AlexNet – a Convolutional Neural Network (CNN) that achieved first place in the 2012 Imagenet Large Scale Visual Recognition Challenge – is an example of a DNN that defined the state-of-the-art in image recognition.

In this thesis, I present a survey of general and expert designed methods, as well as generalizable frameworks that attempt to accelerate AlexNet's training via parallelism. While this exploration of parallelism within AlexNet illustrates the efficacy of general and expert designed methods and lends insight towards accelerating the class of deep learning methods, increasing the accessability and scalability of DNNs requires investment into generalizable frameworks that treat parallelism within deep learning as an abstraction.

ACKNOWLEDGEMENTS

I would like to thank Professor Andrea Vaccari for advising this work providing so many thoughtful comments. Thank you to Professor Michael Linderman and Jonathan Kemp for helping me set up the computational resources I needed to run my experiments, and for putting in the Herculean effort of debugging the countless software issues we ran into. Finally, thank you to all of my friends and family for supporting me, I would not be where I am without you.

TABLE OF CONTENTS

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Problem Description . . . 8
- 1.3 Proposed Solutions . . . 10

2 Machine Learning Overview **12**
- 2.1 Objective Functions & Optimization . . . 13
- 2.2 Deep Neural Networks . . . 20
- 2.3 Deep Learning Architectures . . . 28
  - 2.3.1 Convolutional Neural Networks . . . 29

3 Parallelism **36**
- 3.1 Distributed Deep Learning . . . 37
  - 3.1.1 Parameter Servers . . . 38
- 3.2 Data Parallelism . . . 40
- 3.3 Model Parallelism . . . 42
- 3.4 Hybrid Methods . . . 44
- 3.5 Frameworks for Parallelism . . . 47

4 Parallelizing Deep Neural Networks **49**
- 4.1 Image Classification through AlexNet . . . 49
- 4.2 Experiments . . . 50
  - 4.2.1 AlexNet Training Data . . . 50
  - 4.2.2 Benchmark Baselines . . . 53
  - 4.2.3 Data Parallelism within AlexNet . . . 59
  - 4.2.4 Model Parallelism within AlexNet . . . 64
  - 4.2.5 Expert Designed Methods . . . 69
  - 4.2.6 FlexFlow Parallelization Framework . . . 73
- 4.3 Analysis . . . 75
  - 4.3.1 AlexNet and Tiny ImageNet Analysis . . . 76
  - 4.3.2 AlexNet and Dogs vs. Cats Analysis . . . 79

5 Conclusions **82**
- 5.1 Future Work . . . 84

A Data Summary for Tiny ImageNet benchmarks **86**

B Data Summary for Dogs vs. Cats benchmarks **87**

Bibliography **88**

LIST OF FIGURES

1.1 Sample linear regression 2
1.2 Sample Starbucks data from the openintro R package 3
1.3 Modelling a subset of Starbucks menu items via linear regression . . . 4
1.4 Modelling all Starbucks menu items via linear regression 5
1.5 Tradeoff between flexibility and interpretability for various models . . 6
1.6 Visualizing the efficiency of neural networks overtime 7
1.7 Comparing the accuracy of a model between test and validation data . . 9

2.1 A visual comparison of Linear and Logistic Regression 14
2.2 Logistic regression cost function 15
2.3 Description of stochastic gradient descent 17
2.4 Types of critical points in 2 dimensions 18
2.5 Visualizing the inputs of a perceptron 20
2.6 Simple feedforward neural network architecture 21
2.7 Rectified Linear Unit activation function 23
2.8 The effect of depth in neural networks 29
2.9 Comparing the effects of increasing the size of a model 30
2.10 A simple Convolutional Neural Network 31
2.11 An example of convolution on a 5×5 image, with a 2×2 kernel . . . 33
2.12 Convolutional Neural Network Architecture 34

3.1 Parameter servers used with Downpour SGD and Sandblaster L-BFGS 39
3.2 Simple example of data parallelism 41
3.3 Simple example of model parallelism 43
3.4 TensorFlow graph representation of gradient computation 45
3.5 FlexFlow framework for parallelizing DNNs 47
3.6 SOAP parallelization example 48

4.1 AlexNet architecture description 49
4.2 Sample of TinyImageNet images 51
4.3 Sample of Dogs vs. Cats images 52
4.4 AlexNet variant used for experiments 53
4.5 Baseline AlexNet Epoch Times for Tiny ImageNet 55
4.6 Baseline AlexNet Training Time Quantiles for Tiny ImageNet 56
4.7 Baseline AlexNet epoch times for Dogs vs. Cats 57
4.8 Baseline AlexNet epoch time quantiles for Dogs vs. Cats 58
4.9 Data Parallelism schema for AlexNet 59
4.10 Data parallel AlexNet epoch times for Tiny ImageNet 60
4.11 Data Parallel AlexNet epoch time Quantiles for Tiny ImageNet 61
4.12 Data parallel AlexNet epoch times Dogs vs. Cats 62
4.13 Data parallel AlexNet epoch time quantiles for Dogs vs. Cats 63
4.14 Model Parallel schema for AlexNet 64
4.15 Model parallel AlexNet epoch times for Tiny ImageNet 65

4.16 Model parallel AlexNet epoch time quantiles for Tiny ImageNet 66
4.17 Model parallel AlexNet epoch times for Dogs vs. Cats 67
4.18 Model parallel AlexNet epoch time quantiles for Dogs vs. Cats 68
4.19 One Weird Trick for Parallelizing AlexNet 69
4.20 OWT AlexNet epocn times for Tiny ImageNet 70
4.21 OWT AlexNet epoch time quantiles for Tiny ImageNet 71
4.22 OWT AlexNet epoch times for Dogs vs. Cats 72
4.23 OWT AlexNet epoch quantiles for Dogs vs. Cats 73
4.24 FlexFlow performance on AlexNet 74
4.25 Comparing quantiles for parallelism methods using Tiny ImageNet . . 76
4.26 Comparing epoch times for parallelism methods using Tiny ImageNet . 77
4.27 Comparing quantiles for parallelism methods using Dogs vs. Cats . . . 79
4.28 Comparing epoch times for parallelism methods using Dogs vs. Cats . 80

CHAPTER 1

INTRODUCTION

1.1 Background

As the number of large-scale, publicly available data sets increases [1, 2], methods to understand these data have grown in importance. The analysis and prediction of trajectories has proved an invaluable tool in the early detection of natural disasters [3] such as hurricanes [4] and earthquakes [5], helping farmers calibrate their crops by modeling micro-patterns in weather [6], allowing for a better understanding of complex financial machinery by analyzing economic markets [7], and helping companies produce better products by capturing consumer trends [8].

In capturing patterns within a system, models use *explanatory* variables to produce predictions in the form of *response* variables. *Machine learning* refers to the process of developing models that can predict response variables given one or more explanatory variables. Born out of *statistical modelling*, machine learning is an approach to data analysis that formalizes relationships between response and explanatory variables through mathematical representations extrapolated from the underlying data. As such, machine learning attempts to use data as a means of modeling relationships that are useful in various domains.

For example, a simple *linear regression* model could be used to predict the price of a house (response variable) given the number of bedrooms, number of bathrooms, and the square footage (explanatory variables).

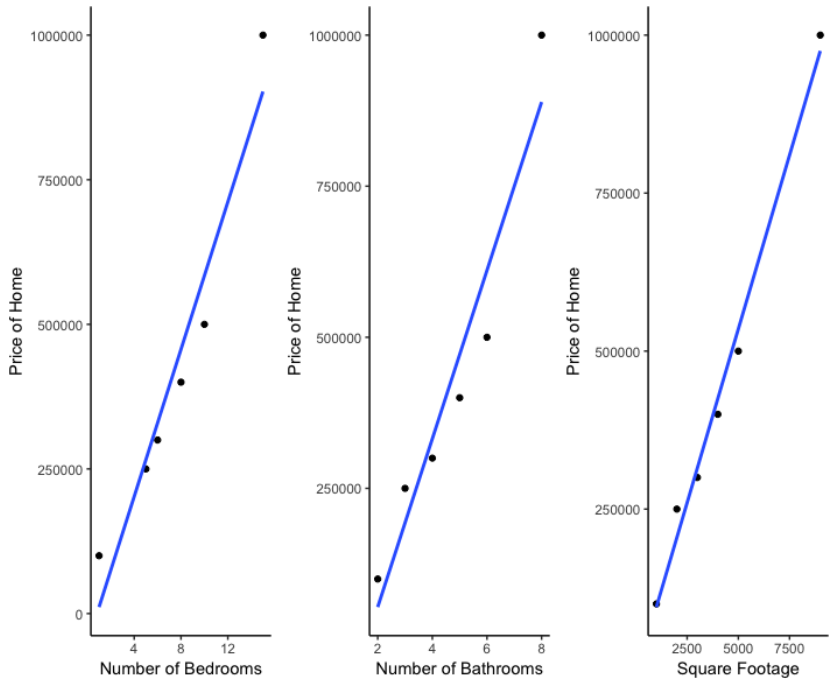

Figure 1.1: Correlating the price of a house with square footage, number of rooms, and number of bathrooms

Given these specific explanatory variables, a linear model can predict a proper response: the more bedrooms, bathrooms, and square footage a house has, the higher the price. Linear regression achieves this result by attempting to fit a line by minimizing the distance between individual data points and the regression line (see Section 2.1 for more information on how linear regression finds the optimal fit). The models in Fig. 1.1 are an example of *single variable linear regression* using each individual explanatory variable to predict the response variable.

While single-variable linear regression models can only capture linear relationships, they encapsulate the idea underlying more sophisticated machine learning models: these models attempt to fit a curve to a set of data points in order to draw an association between explanatory and response variables, and ensure accurate predictions in a given domain.

As the relationship between explanatory variables and responses grow in complexity,

it becomes increasingly difficult to model it using simpler approaches such as linear relationships. An analysis of *Starbucks data* provided by the *openintro* R package [9] highlights the ways in which linear models are not entirely sufficient for all data.

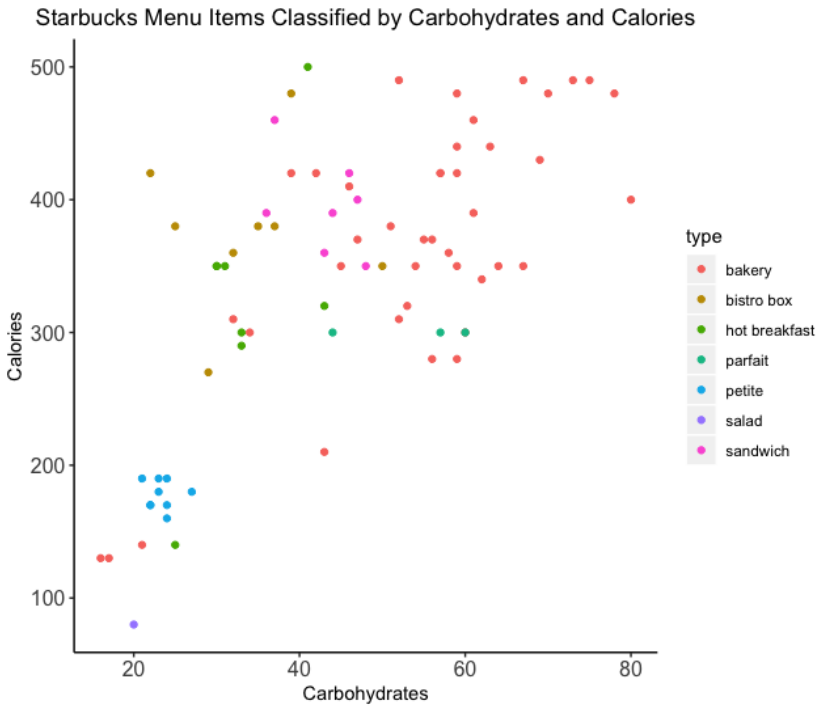

Figure 1.2: Sample Starbucks data from the openintro R package [9]. For various menu items, carbohydrates and calories were mapped on the x axis and y axis respectively, and color coded by the type of menu item.

While it is possible to identify a general positive linear trend of an increase in calories with an increase in carbohydrates in Fig. 1.2, a more difficult task would be to identify the different menu items based on carbohydrate and caloric data. This type of analysis, where we want to identify specific items of a discrete class based on certain properties, is called *classification*. A simple linear regression model would fail to capture the functional distinction between all menu items based on carbohydrates and caloric data. This is due to the fact that the slope and intercept of a line are the only two parameters available to a simple linear model. In order to properly classify menu items, different items should lie on different lines.

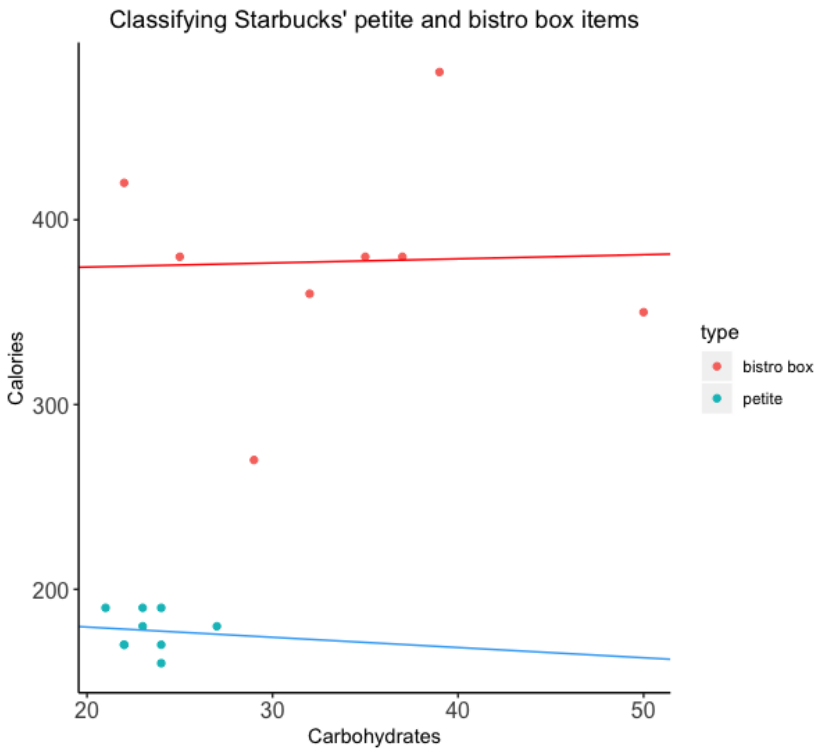

Figure 1.3: The red line is the linear regression model that attempts to model bistro box items using carbohydrates as an explanatory variable and calories as a response variable. Similarly, the blue line attempts to predict calories using carbohydrates for petite items.

In Fig. 1.3, the *petite* items are on the lower left of the graph and *bistro box* items are spread through the upper portions of the graph, past approximately 300 calories. The linear model used for bistro box items had an intercept of 369.92 and a slope of 0.23. The linear model used for petite items had an intercept of 190.74 and a slope of -0.56. The functional forms of these models is given in (1.1).

$$\begin{aligned} \textit{Bistro(calories)} &= 369.92 + 0.23(\textit{Bistro(carbohydrates)}) \\ \textit{Petite(calories)} &= 199.74 - 0.56(\textit{Petite(carbohydrates)}) \end{aligned} \tag{1.1}$$

If these linear models were used to classify all menu items, any items that share the same linear relationship between carbohydrates and calories would be *misclassified* – they would be predicted to be part of the wrong class (Fig. 1.4). In particular, the intersection of lines indicates points that could be misclassified as either one menu item

or the other. As a result, different approaches, such as *logistic regression* (see Section 2 for more information on logistic regression) were developed.

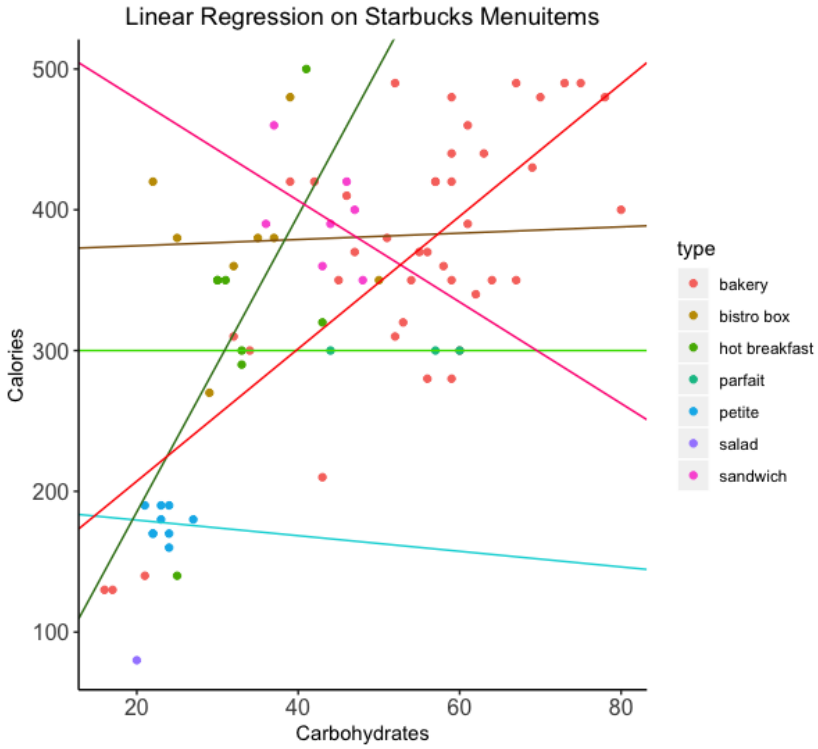

Figure 1.4: Modelling all menu items via their linear relationship between carbohydrates and calories. Several lines cross through points of different colors, which are misclassifications. Note there is no line pertaining to salad menu items – there is on1y one salad menu item, and 2 points are required to establish a linear fit.

Classification tasks are common in machine learning and typically relevant to data sets with *labels*. Labels describe the discrete class that a particular data point falls within. For example, in the Starbucks data set, the different menu items (bakery, bistro box, hot breakfast, parfait, etc.) would be assigned unique labels (typically a unique value that identifies that data point's class). Labeled data sets are used in *supervised learning*, where models being trained have access to the value (labels) that a correct prediction can take from a set of discrete classes. *Unsupervised learning* refers to models where target labels are not provided. Both supervised and unsupervised approaches can be used in classification and require *training* data sets to map a set of explanatory

variables X to a response prediction Y. The process of training a model refers to the use a subset of data points to optimize a given model's fit. In addition to training, in order to generate a fit to the data, *validation* data sets are used to determine a model's ability to extrapolate to unseen examples outside of the training set and are typically used to tune parameters that help generate a better fit (see Section 2.1 for more information on how models are optimized).

Some classification tasks prioritize inference over predictive performance. *Inference* refers to applications where the accuracy of a given prediction is less important than the need to understand the relationship between X and Y [10]. Inference models are *interpretable* in that they do not treat the mathematical relationship between X and Y (explanatory and response variables) as a black box. Fig. 1.3 is an interpretable model because the relationship between X and Y has an easily understood formulation ((1.1)).

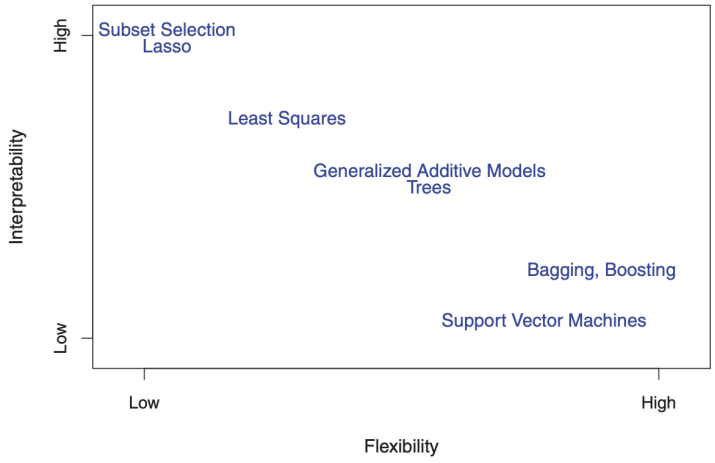

Figure 1.5: Comparing the relationship between interpretability and flexibility for various models. Interpretability tends to decrease as flexibility increases [10].

While model interpretability facilitates trust between users and models [11], the state-of-the-art in challenging applications, such as automated image classification, is increasingly dependent on more *flexible* methods. A method's flexibility is defined by the number of functional forms it can represent [10]. Fig. 1.5 highlights various machine

learning methods, and illustrates the trend that as a model increases in flexability, it decreses in interpretability. While linear regression models are inflexible because they are only able to capture linear relationships, as in Figures 1.1, 1.3 and 1.4, a class of machine learning models known as *neural networks* are unrestricted by functional forms.

As data has become more readily available in various domains [1, 2], neural networks and their most recent and more complex version, *deep neural networks (DNNs)*, have become one of the most widely used flexible machine learning methods [12] and have successfully been applied to speech recognition [13], image classification [14], and super-human game playing performance [15]. For more information on DNNs, see Section 2.2.

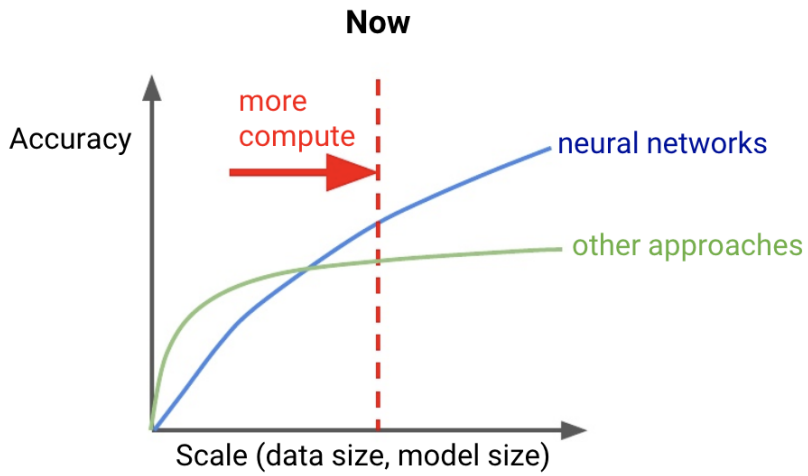

Figure 1.6: As data and model sizes have increased in scale, neural networks and deep learning methods have increased in accuracy over other methods. The overall effectiveness of these methods is related to the amount of computational power available to train these models [12].

The pervasive use of DNNs across various state-of-the-art applications highlights the need to increase the accessibility of these methods. Training state-of-the-art DNNs requires access to high cost, high performance clusters [15]. By developing methods that can accelerate the training of DNNs, through optimization and distribution of computa-

tion across devices, it will be possible to increase their scalability and, in turn, improve the state-of-the-art.

1.2 Problem Description

As deep learning methods have become more ubiquitous, they have become less accessible [16]. Modern deep learning methods have benefited from a dual increase in the availability of large data sets as well as the development of more complex model architectures. As the number of computations required to optimize a model has grown with the use of more flexible methods such as neural networks, the need for more computational power capable of training these models has also increased. The cost associated with training these models on vast data sets has made the acceleration of these models an imperative. Deep learning methods in the classification setting are computationally expensive to train on higher dimensional data such as images [16] due to the number of computations the training model requires [14]. By increasing training efficiency through minimizing the time it takes to complete model computations, the design and implementation of deep learning methods become more accessible across hardware configurations. Optimizing the use of available computing resources is a step towards making it possible to train larger models on less powerful hardware.

Deep learning methods require large data sets [17] in order to avoid *overfitting*, a phenomenon within machine learning where a model is able to effectively classify examples during training but fails to generalize to data outside of the training set (known as a *test set*) [18]. This phenomenon can be identified by looking at the behavior of accuracy in training and test sets as a function of the number of *epochs* — the number of training cycles a model undergoes.

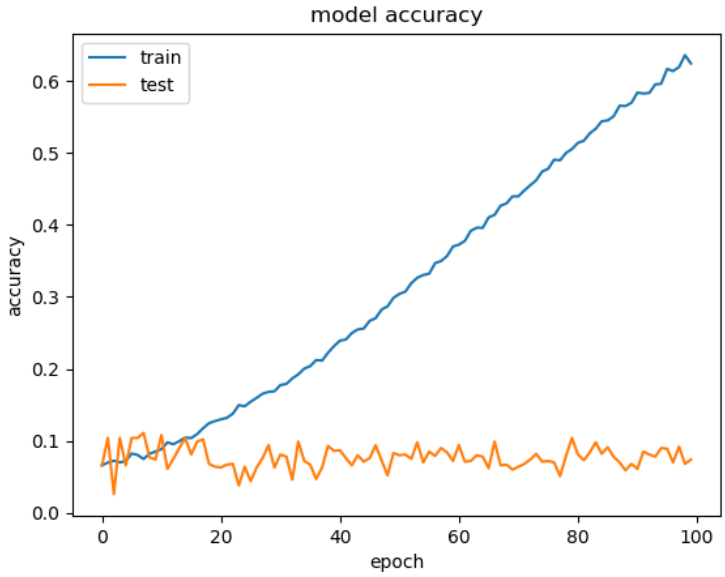

Figure 1.7: The accuracy plot between test and training data for a deep neural network trained on the million songs data set [2]. This particular model was trained on 100000 examples, and had 5 hidden layers with 100 nodes each.

For example, Fig. 1.7 shows that while the training accuracy steadily increases, the validation accuracy stagnates. This is an example of a deep learning model that overfits on a subset of 100,000 examples from the Million Songs Dataset (MSD) [2]. The use of test sets and training on more data [18] are examples of techniques that can be used to used to combat overfitting. The more data a model is trained on, the more computations that are required in order to train the model.

As data becomes more readily available, modern deep learning applications require high-end clusters [19] and distributed computational power [20]. Training even very simple models often requires a *Central Processing Unit (CPU)* with many cores, while larger models require *Graphical Processing Units (GPUs)* to provide the necessary computational power. Modern applications such as image recognition and natural language processing necessitate the use of distributed GPU clusters in order to place an acceptable upper bound on training time. For example, AlexNet required 6 days of training

over dual GPUs [14]. Another example is AlphaGo [15], a computer program developed by DeepMind [21] that achieved super-human performance at the game of Go, with a 98% win rate against other state-of-art Go playing programs and a 5 to 0 win over a human European Go champion. AlphaGo was trained across 48 CPUs and 8 GPUs. AlphaGo Zero, an updated version of AlphaGo, takes 40 days of training to surpass all Go players – simulated or human [22]. Both AlexNet and AlphaGo are representative of the ways in which modern deep learning methods are in equal measure effective and computationally expensive.

As a highly accurate image classification DNN often used as a benchmark [14, 20, 19], AlexNet can serve as a lense through which parallelism can be understood to accelerate training. AlexNet has expert designed parallelization methods [23], and has been used as a benchmark in multiple works exploring parallelism through frameworks [19, 20]. The use of the AlexNet architecture as a benchmark for parallelization methods will provide insight into how the training of DNNs can be accelerated, which will in turn inform the ways in which deep learning methods can become more accessible through the optimized use of a given set of resources.

1.3 Proposed Solutions

In order to explore how parallelism can accelerate the training of DNNs, I will train AlexNet on two data sets: *Tiny ImageNet* [24] and the *Dogs vs. Cats* dataset provided by Kaggle [25]. Designed as an image recognition challenge for a course at Stanford University [26], Tiny ImageNet is based on the original ImageNet dataset [1]. The Dogs vs. Cats dataset is provided as part of a Kaggle competition that sought an algorithm with the highest classification rate.

After establishing a baseline training time for AlexNet on both of these data sets, I implemented different parallelization methods with the purpose of decreasing AlexNet's

training time by effectively parallelizing training computations. The parallelization methods I explored are *data parallelism*, *model parallelism* (applicable to all DNNs), and an expert designed method specific to the AlexNet architecture, known as *One Weird Trick (OWT)* [23]. In addition to these methods, I will provide a comparative look at an architecture agnostic, generalizable framework for deep learning acceleration known as *FlexFlow* [19].

The comparison of data parallelism, model parallelism, and OWT in this work will highlight how the acceleration of AlexNet's training can be extrapolated to other deep learning methods, effectively making DNNs more efficient and accessible. AlexNet's parallelization is a lens through which the accessibility and scalability of deep learning methods can be understood, improved, and expanded.

Increasing accessibility to the implementation of machine and deep learning models requires the abstraction of parallelism to improve resource management much like programming languages abstract away low-level details that are not relevant to a particular programmers' use case. Treating parallelism as an abstraction normalizes optimized device usage across different architecture implementations. Furthermore, methods that are architecture agnostic will obviate the need for expert designed methods that are currently architecture specific.

CHAPTER 2

MACHINE LEARNING OVERVIEW

The application of machine learning methods has been popularized in academic research as well as consumer products. Image classification [14], speech recognition [13], and powerful recommender systems [8] have all employed supervised learning methods in an effort to better classify unseen data. While these are vastly different domains, the algorithms used to process an image and correctly classify it as an image of a dog based on fur, size, and other physical features share similarities with methods used to build a recommender system that Netflix can use to provide users with suggestions on what to watch based on a user's previous watch history, location, and language, among other factors.

As previously mentioned, machine learning is the automated process of fitting a curve to a set of sample data points. Data can come from anywhere – it can be collected from real world phenomena or artificially created. There are no costraints on where data comes from or how it should be collected except that, in order to provide useful and generalizable analytical results, it must be statistically distributed in a fashion that is reflective of the larger population from which the sample of data was collected [27]. Once the sample data is collected, useful information can be gleaned from running statistical analysis on the data. This initial exploratory analysis often reveals useful information about a particular data set which provides valuable insights for downstream modelling purposes.

Given a data set that is properly sourced and distributed, machine learning uses this data to define the parameters of a mathematical model that captures the relationship between the data points in a by optimizing some *objective function* – typically a measure of how well a model fits the data. The experiments in this work attempt to accelerate the training of deep learning methods by distributing the computation associated with the

optimization of the objective function.

2.1 Objective Functions & Optimization

A model can improve predictive performance by optimizing a performance metric. For example, a model can minimize the misclassification rate in order to generate a better fit to the data. Performance metrics are necessary in order to track the progress of a model over training epochs – if the accuracy of a particular model is displaying a negative trend or the misclassification rate is steadily increasing, the model requires modification. Objective functions are used to improve a model's performance metric by optimizing the model's fit.

A common metric used in machine learning is *accuracy*: the rate of correct classifications a model makes over the total queries it was provided. An image classification model that correctly classifies 53 images out of a 100 total images it was shown (misclassifying 47 images) has an accuracy of 53%. Fig. 1.7 is an example of plot that kept track of a neural network's accuracy over test and train data as a function of epochs.

While accuracy is a useful and easily interpretable metric, the *loss function* is the objective function used in most neural network models. The loss function, like accuracy, provides insight into the overall performance of a model. In particular, the loss function aggregates the error on each predicted response. If a model correctly classifies a large number of examples in the data set, its loss will be low since there will be little error between the predicted and provided label. Likewise, if a model misclassifies a large number of examples in the data set, it will have a higher error rate and loss. At the core of the loss function is the *residual*, or the difference between a predicted and observed value. Predicted responses are produced by the model, while observed responses are part of the training data set. For example, in models such as linear regression, the residual is used as part of a *mean squared error* (MSE) calculation, the average of the squared

residuals:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2.1}$$

where n is the number of data points in the data set, Y_i is is the ground-truth label, $\hat{Y}_i$ is the predicted value, and $(Y_i - \hat{Y}_i)$ is the the residual for the $i-th$ data point. Minimizing the MSE allows the evaluation of the optimal fitting line in linear regression models.

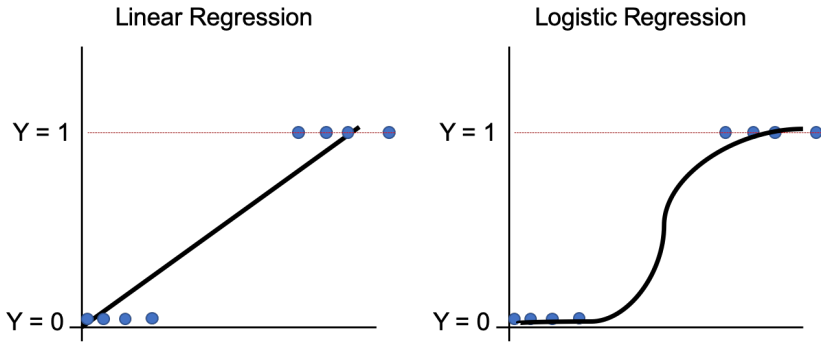

Figure 2.1: A visual comparison of linear and logistic regression. While linear regression predicts along a continuous scale, as in Figure, 1.1, logistic regression bounds output between 0 and 1, making it more suitable for classification tasks where discrete classes can be mapped to discrete values [28].

As discussed in the introduction, linear regression is able to model linear relationships in continuous data, but is unable to make predictions on discrete classes or *categorical variables*. *Logistic regression* bounds the classification results between 0 and 1 so as to map a result to a particular class, as in Fig. 2.2. In particular, simple logistic regression models take the form

$$Logit(h_\theta(x_i)) = \alpha + \beta(x_i) \tag{2.2}$$

where $Logit(h_\theta(x_i))$ is the logit of the probability that example x_i is in a particular class. Logistic regression maps results to a probability between 0 and 1 as follows:

$$h_\theta(x_i) = \frac{e^{\alpha+\beta(x_i)}}{1 + e^{\alpha+\beta(x)}} \tag{2.3}$$

where $h_\theta(x_i)$ is the hypothesis function based on parameters θ, which predicts the probability that x_i is in a particular class. According to Fig. 2.2, $h_\theta(x_i)$ in logistic regression will always fall between 0 and 1.

Similar to the loss function, the *cost function* calculates the cost of a prediction $h_\theta(x_i)$ for an example x_i. A misclassification results in a higher cost. The cost function for logistic regression is defined as follows:

$$\textit{Cost}(h_\theta(x_i), y_i) = \begin{cases} -\log(h_\theta(x_i)) & \textit{if } y_i = 1 \\ -\log(1 - h_\theta(x_i)) & \textit{if } y_i = 0 \end{cases} \tag{2.4}$$

where $h_\theta(x_i)$ attempts to predict the class y_i given example x_i. The cost function can be thought to calculate the cost of predicting $h_\theta(x_i)$ when the true class label is y_i.

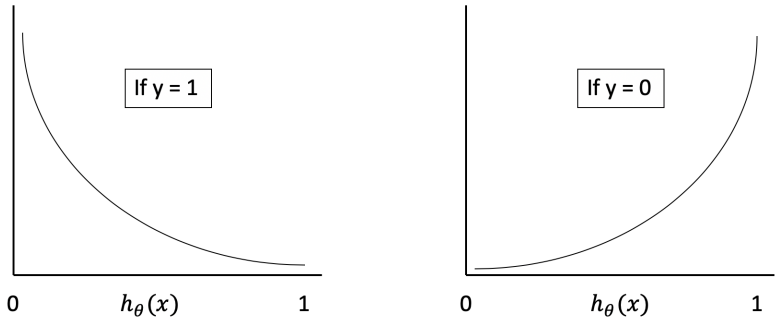

Figure 2.2: A visualization of the logistic regression cost function. As described mathematically in (2.4), the logistic regression cost function will converge to 0 when $y = 1$, and go to infinity when $y = 0$.

The cost function for logistic regression, as depicted in Fig. 2.2, converges to 0 if $y = 1$, and grows asymptotically towards infinity at $x = 1$ if $y = 0$. Thus, if the distance between a prediction and the actual response is large, the the function yields a higher cost, thus penalizing a misclassification. Likewise, if the distance between the predicted and actual response is low, the cost function yields a low cost. The condensed version of the cost for a single data point is more amenable for operations over the cost of all

training examples.

$$\textit{Cost}(h_\theta(x_i), y_i) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))) \tag{2.5}$$

where $Cost(h_\theta(x_i), y_i)$ is the cost associated with prediction $h_\theta(x_i)$ with true class label y_i.

The condensed version of logistic regression's cost function in (2.5) can be simplified to $-y \log(h_\theta(x))$ when $y = 0$, and $-(1 - y) \log(1 - h_\theta(x)))$ when $y = 1$. This functional form can then be summed over all training examples, to calculate the cost on the entire data set.

$$J(\theta) = \textit{Cost}(h_\theta(x_i), y_i) = -\frac{1}{m} \sum_{i=1}^{m} [y \log(h_\theta(x_i)) - (1 - y_i) \log(1 - h_\theta(x_i))] \tag{2.6}$$

Equation (2.6) is the sum of the cost of all m predictions made with the hypothesis function h_θ. In (2.6), the cost function is defined as $J(\theta)$, which is typical of cost function definitions in the literature [18].

While the cost function is an effective measure of a model's performance, calculating it across large data sets (large m) can become computationally expensive. Given a mathematical function that captures the performance of our logistic regression model across every prediction on every data point, we have a function that can be optimized in order to improve our model. Optimizing the cost function for logistic regression requires minimizing $J(\theta)$, such that the total cost of the model is as low as possible. As the optimization function for linear regression would attempt to minimize the mean squared error in (2.1), the optimization function for logistic regression would need to minimize (2.6). $J(\theta)$ can be minimized via an optimization algorithm, such as *gradient descent*.

The gradient of a given function f is a vector containing the partial derivatives of that function, denoted $\nabla_x f(x)$ [29]. The partial derivatives are taken with respect to each dimension – higher order functions have more partial derivatives to compute, while lower order functions are much simpler. The dimensions along which partial derivatives are

taken are based on the explanatory variables or *features* – certain data, such as images, have higher dimensional representations.

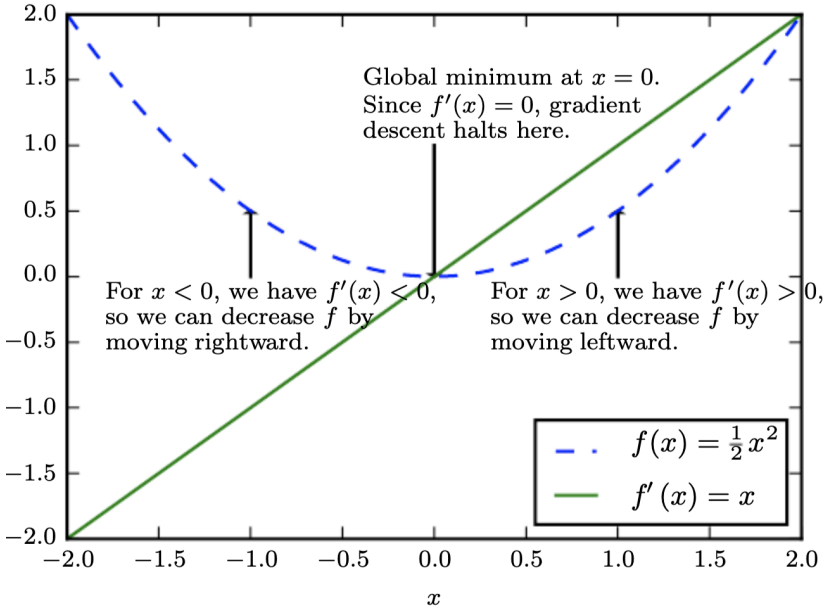

Figure 2.3: Stochastic gradient descent knows which direction to travel along the curve by observing the sign of the derivative at a particular point on the curve [29]. For example, the derivative on the left side of the curve is less than 0 which tells us the slope of the tangent line at this point is negative – thus, a minima could be reached by moving to the right of the curve. The derivative of the point on the right side of the curve is greater than 0, which means that the slope of the tangent line through this point is positive, indicating we can move locate a minima by moving to the left. The slope of the tangent line at the center of the curve is 0, indicating we have located a minima.

Originally proposed by in 1847 by Louis Augustin Cauchy, gradient descent iteratively computes the *derivative* of a curve to locate a *minima* [30]. The slope of the tangent line at a particular point provides information to the algorithm on which direction to move (Fig. 2.3). The objective is to achieve a tangent line with a slope of 0 because this corresponds to a *critical point* (Fig. 2.4).

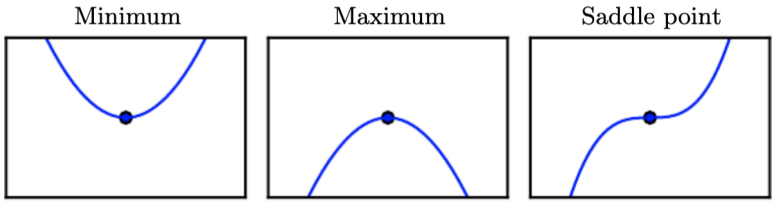

Figure 2.4: Critical points refer to points where the tangent line has a slope of 0. These points can be a local minima, where the point is lower than its neighbors, a local maxima, where the point is higher than its neighbors, or a saddle point, where the point has neighbors that are both higher and lower than itself [29].

When applied to the cost function $J(\theta)$, the goal of the gradient descent algorithm is to identify a minimum lending evidence towards the hypothesis that the model's performance cannot be improved any further. Gradient descent, while effective and widely utilized in the literature [31], is a greedy optimization algorithm. There is no guarantee of identifying the global minimum unless the curve is *convex*. A convex curve is a simple curve in which all the tangent lines lie on exactly one side of the curve – Fig. 2.3 shows an example of gradient descent applied to a convex curve in the form of a parabola. In the context more sophisticated models, such as neural networks, there is no guarantee of the convexity of the cost function, hence of finding the global minimum. One mitigation of this concern is the use of *stochastic gradient descent*, which uses a single random data point for every iteration of the algorithm – this ensures random sampling is effective over a large number of iterations. *Minibatch gradient descent* uses a small random sample of data points for every iteration, which is a common compromise between the extremes of using a single data point with stochastic gradient descent and all available data points. The process of computing gradients and performing gradient descent to optimize a cost function can become computationally expensive with higher order functions because of the number of partial derivatives to compute for each gradient. This is especially true for large training data sets.

While logistic regression in an effective classification model, it is not suitable for

more complex tasks such as image classification. While more complex, the core idea surrounding neural networks is similar to those in linear and logistic regression – define a cost function based on a loss metric that measures the performance of the model across each data point, and then optimize that function to improve the model's performance. Neural networks have a cost function that is a generalization of the logistic regression cost function.

Whereas logistic regression is utilized in binary classification problems, neural networks can be applied to multi-class classification problems. Multi-class classification problems, like attempting to classify menu items at Starbucks in Fig. 1.2, have more than two potential output classes. The logistic regression cost function accounted for a single output unit, but neural networks can have arbitrarily many. The hypothesis function $h_\theta(x)$ for logistic regression outputs a single prediction, but for neural networks, $h_\theta(x) \in \mathbb{R}^k$. Thus, the neural network can produce up to k output units per data example, one output unit per class.

The general form of the cost function of a neural network can be written as:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{k}[y_k^{(i)}\log(h_\theta(x_i))_k - (1-y_k^{(i)})\log(1-h_\theta(x_i))_k] \tag{2.7}$$

where $J(\Theta)$ is the cost of the neural network over m data points and k classes.

Equation (2.7) looks similar to the logistic regression cost function in (2.6), except that it is generalized over k discrete classes. The neural network cost function sums over all m data points, while also calculating the cost of predicting each of the k classes – in binary classification problems, k is equal to 1, which makes equations 2.6 and 2.7 equivalent. Like the logistic regression cost function, the neural network cost function can also be optimized via gradient descent.

2.2 Deep Neural Networks

Loosely inspired by neuroscience, neural networks have grown in adoption and application [12]. While the *No Free Lunch* theorem states that a model's elevated performance on a class of problems is directly paid for in performance over another class [32], the ubiquity of applications that use neural networks is a testament these model's ability to tackle a wide problem space.

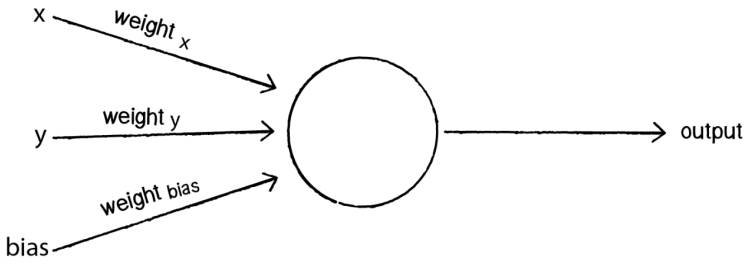

Figure 2.5: The perceptron takes the weights as well bias as input. [33]

As seen in Fig. 2.5, the fundamental unit of a neural network is a *perceptron*, also called a *neuron* or *node*. Every neuron has an *activation function* that gets applied to some input data, typically composed of input *weights* and biases. The weights of a model refer to the the strength of connections between nodes. Observing Fig. 2.6, the outgoing edges from one node to another have weights associated with them – the greater the weight, the greater the influence that a given node exerts on the node it is connected to via a directed edge. The influence a node exerts refers to how much it effects the computation of the activation at the next layer of nodes.

Neural networks are composed of layers of a predefined number of nodes. Different layers can have different numbers of neurons, and every neuron has an incoming connection from neurons in the preceding layer and an outgoing connection with every neuron in the following layer. The architecture of a neural network is defined by the number of layers in the overall model, as well as the number of neurons per layer. Fig. 2.5 illus-

trates a single neuron in a network, whereas Fig. 2.6 illustrates the composition of many neurons to create a neural network.

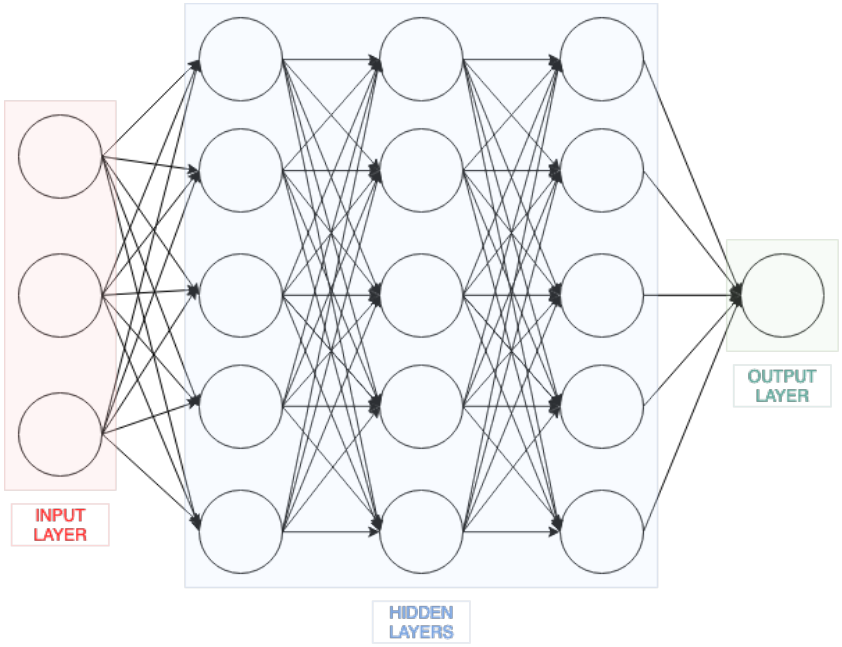

Figure 2.6: A simple feedforward neural network architecture. This sample neural net has an architecture that consists of an input layer with 3 neurons (red), 3 hidden layers with 5 neurons each (blue), and a 1 neuron output layer (green).

Neural networks are generally composed of 3 types of layers, with other specialized layers used in more specific applications. A typical network will consist of an *input layer*, one or more *hidden layers*, followed by an *output layer* (Fig. 2.6). The input layer is how the model receives data – application specific transformations are applied to the input data in order to get the data to a format that can work with the neural network. Hidden layers provide the model with sufficient flexibility to be able to map higher order functions that characterize the relationship between the actual input data and output predictions.

Whereas linear models are only able to represent linear trends, the flexibility of neural networks allow them to map relationships to a wider range of forms beyond linear representations. These layers are ”hidden” in the sense that they serve as the

intermediary between the input and output nodes [29]. As the final layer in the neural network, the output layer provides the final prediction of the model as a probability distribution over the discrete classes of a classification task. Because of this structure, neural networks are conveniently posed as a composition of functions applied at various layers within the network.

Neural networks are commonly represented as graphs, where every neuron is a *node* in the graph that has incoming and outgoing connections (with the exception of input nodes that have no incoming connections, only outgoing connections). Taking the nodes of a neural network as the *vertices* of a graph and the connections between nodes as the *edges*, neural networks are *directed* in that there is a direction associated with every edge between every node. Fig. 2.6 illustrates a network in which each node's output connection is directed towards the output layer. This ensures that these graphs are *acyclic* (with no loops). A model's computation graph is often represented through *Directed Acylic Graphs* (DAGs) [34]. *Feedforward* neural networks are often represented as DAGs since every node in these networks has an edge directed towards the following layer. Fig. 2.6 is an example of a feedforward network. The feedforward nature of certain neural network architectures has been widely adopted [29].

Characterized by their depth, deep neural networks (DNNs) often have more than one hidden layer, as depicted in Fig. 2.6. The nodes that sit in the hidden layers contain activation functions that get applied to the data passing through that particular layer. Neural networks, like simple linear and logistic regression, also have weights or parameters that define the model. These are typically represented by Θ. Each neuron applies the activation function to a linear combination of the input data and the weights, with the weights updated on each iteration.

While there are several activation functions, the *Rectified Linear Unit* (ReLU) is one of the most common, popularized within deep learning applications by AlexNet in the

2012 ILSVRC competition [14]. The ReLU activation function is widely used due to its ability to induce non-linearity within activations [35]. While other activation functions exist, such as *sigmoid* and *tanH* [36], ReLU is immune to the prevalent *vanishing gradient* problem [37]. This problems occurs for activation functions like the sigmoid function where the inputs are mapped to a small range between 0 and 1 – this implies that a large change in the input will result in only a small change in the output, causing the derivatives to shrink to 0 – ReLU activations do not suffer from this pitfall.

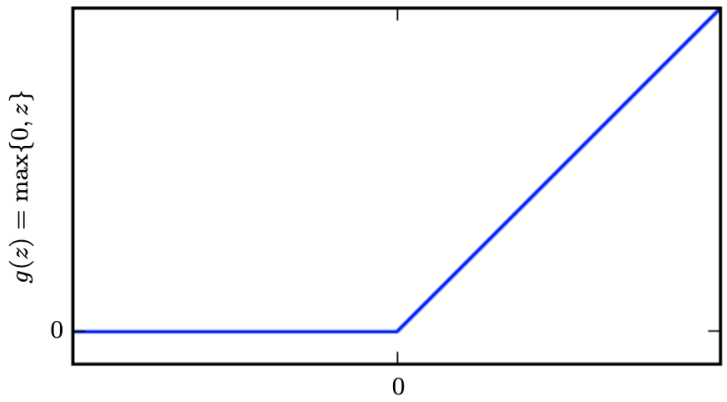

Figure 2.7: The rectified linear unit (ReLU) is a piecewise linear function meant to apply a non linear transformation. The function is designed such that the derivative is 1 when z is positive, and 0 when z is negative. [29].

The ReLU activation function has shown to improve the overall training speed of deep learning methods while introducing a restricted form of non-linearity. As depicted in Fig. 2.7, the slope of the tangent line is always either 0 or 1, depending on the input – a useful attribute for gradient computation. The composition of neurons with ReLU activation functions within deep neural networks has produced a number of state-of-the-art results, especially within image classification [14].

While ReLU activation functions are utilized within hidden layers, the output layer is unique in that the output depends on the type of classification problem at hand. A binary classification problem requires a single output node with a single prediction, whereas a multiclass classification problem needs to provide a probability distribution

over all possible classes – the prediction corresponds to the distinct class with the highest probability. While logistic regression can achieve this for binary classification problems, the *softmax function* is used in instances where a probability distribution over n classes is required [29]. The softmax activation can be defined as follows:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{2.8}$$

where z corresponds to the input vector. The softmax activation normalizes e^{z_i} by dividing by the sum of all exponentials, $\sum_{j=1}^{n} e^{z_j}$. Equation (2.8) is the formal definition of the softmax function. This function typically is used in the output layer of a multiclass classification neural network, and produces a vector $\hat{y} \in \mathbb{R}^n$, where every element of $\hat{y}$ represent the probability of that class being the model's prediction. The elements of $\hat{y}$ would sum to 1, as the $softmax(z)_i$ function produces a proper probability distribution over all the possible classes.

Neural Network Optimization

DNNs improve the fit of the model by optimizing (2.7). The cost function can be minimized by gradient descent. The gradients are updated via the *backpropagation* algorithm, originally proposed in 1986 [38]. Backpropagation computes all of the activations of each neuron in each hidden layer during the forward pass of the neural network, and then performs a backwards pass from the output layers towards the beginning of the network to evaluate the error in the activation for every node in each layer of the network. These errors are evaluated starting from the final error in classification at the output layer and "propagate" back through every hidden layer towards the input, hence the name.

Concretely, if we take $h_\theta(x)_l$ to be the activation values for layer l, and y_l as the actual observed values from the training set, we can calculate the error term for layer l

as follows [18]:

$$\delta_l = h_\theta(x)_l - y_l \tag{2.9}$$

where δ_l is the error term of layer l and $h_\theta(x))_l - y_i$ is the difference between the predicted value $h_\theta(x))_l$ and the true value y_i.

Equation (2.9) represents the vectorized expression of layer l's error term. Note that δ_l, $h_\theta(x)_l$, and y_l are all vectors where the $i - th$ element of each vector corresponds to a particular activation neuron in the network [18]. For example, $\delta_l^i = h_\theta(x)_l^i - y_l^i$ is the error term for neuron i in layer l of the network. The vectorized representation is used here because it is more concise, and computationally efficient in terms of implementation. This is analogous to the computation of a residual in Section 2.1.

Equation (2.9) is how the error in the final layer in a neural network would be computed, but to compute the error terms in the previous layers, it is helpful to consider a simple 4 layer neural network, with 1 input layer and 3 hidden layers. In this simplified network, the error term for the final layer could be calculated via (2.9), where $l = 4$. The error term for the the 3rd hidden layer would be computed via $\delta_3 = (\Theta^{(3)})^T \delta^4 \odot g'(z^3)$, while the first hidden layer's error would be computed via $\delta_2 = (\Theta^{(2)})^T \delta^3 \odot g'(z^2)$ [18]. More generally:

$$\delta_l = (\Theta^{(l)})^T \delta^{l+1} \odot g'(z^l) \tag{2.10}$$

where $\Theta^{(l)}$ are the weights associated with layer l, δ^{l+1} is the error term associated with layer $l + 1$, and $g'(z')$ is the application of the derivative of the activation function to z^l, the inputs at layer l [18].

In (2.10), we see that in order to compute the error δ_3 for layer 3, we need the value of δ_4. Backpropagation starts by computing layer 4 using (2.9), and then going backwards starting from the last layer. Θ represents the the matrix of weights, and Θ_l corresponds to the weights for layer l. In general, computing the error term for a layer requires the δ value for the layer ahead of it – the computation takes the transpose of the weights

for that layer Θ_l and multiplies it by the error term in the term succeeding this layer, δ_{l+1}. To compute the value, an element-wise product is taken with $g'(z^l)$ [18]. In our simplified network, there is no δ_1 because this is the input layer – there are no errors associated with the input layer. In general, the partial derivative associated with neuron i in layer l is computed as follows:

$$\frac{\partial}{\partial \Theta_{ij}^l} J(\Theta) = a_j^{(l)} \delta_i^{l+1} \tag{2.11}$$

where $a_j^{(l)}$ are the activations associated with neuron j in layer l and δ_i^{l+1} is the error term associated with neuron i in layer $l+1$ [18].

The backpropagation algorithm simplifies the process of computing partial derivatives for the cost function. Equation (2.11) shows that the partial derivatives of the cost function are simple to calculate once the error terms are calculated. Equation (2.11) computes the partial derivative for the cost function with respect to the weights that correspond to layer l and neuron j by multiplying the activation at layer l and neuron j (a_j^l) with the error term at layer $l+1$ and neuron i (δ_i^{l+1}) [18]. The partial derivatives are aggregated as follows:

$$\Delta_{ij}^l := \Delta_{ij}^l + a_j^{(l)} \delta_i^{l+1} \tag{2.12}$$

More formally, the backpropagation algorithm uses Δ_{ij}^l to keep track of the partial derivatives over each iteration. Initially, Δ_{ij}^l is set to 0 for all i and j. Then for every training example in the data set, the forward pass is completed via the activation functions. After the forward pass is complete, (2.9) is used to calculate the error in the final hidden layer. This is then propagated to the preceding layers as (2.10) is used to compute the error terms in the rest of the hidden layers. After all the δ values have been computed, (2.12) is used to accumulate the partial derivatives. Note that the second term in equation (2.12) is equivalent to the partial derivative in equation (2.11), so Δ_{ij}^l is indeed storing the gradients [18]. The partial derivatives are stored as follows:

$$\begin{cases} D_{ij}^{l} := \frac{1}{m}\Delta_{ij} + \lambda_{ij}^{l}\Theta_{ij}^{(l)} & \textit{if } j \neq 0 \\ D_{ij}^{l} := \frac{1}{m}\Delta_{ij} & \textit{if } j = 0 \end{cases} \tag{2.13}$$

where the gradients are collected in Δ_{ij} and divided by all m data points. The presence of the regularization term adds a $\lambda_{ij}^{l}\Theta_{ij}^{(l)}$ term, where λ_{ij}^{l} is the regularization parameter for layer l and neurons i, j.

After every data point in the data set has been processed, (2.13) is used to complete the computation of the gradients. The case for $j = 0$ refers to the bias term, which is why it is missing the regularization term present when $j \neq 0$. $D_{ij}^{(l)}$ stores the partial derivatives for the cost function with respect to each parameter [18]. These gradients are then used in the stochastic gradient descent algorithm to optimize the cost function, which improves the neural network's ability to fit the data.

Neural Network Hyperparameters

Neural networks have *hyperparameters* that control various aspects of the model. Unlike the weights that get tuned to improve model performance through back propagation and gradient descent, hyperparameters define various aspects of how a model should be trained. Like regular parameters, hyperparameters can be tuned in order to improve a model's performance during training – hyperparameters are typically tuned using a separate validation set [39]. Hyperparameters typically include a *learning rate*, *training epochs*, and *regularization*. The learning rate refers to how much the parameters evolve over time, while training epochs refer to the number of iterations that a model is trained – i.e., how many times does it traverse over the entire data set [18].

Avoiding Overfitting in Neural Networks

Regularization is a method that attempts to combat overfitting, and improve a model's ability to generalize to unseen data. *Dropout* is a popular form of regularization that randomly drop nodes and their associated connections from the model during training [39]. In addition to combating overfitting, dropout also prevents nodes in the network from *co-adapting* [39], or behaving in similar ways so as to reduce usefulness and increase redundancy.

Increasing the Accessibility of Neural Networks.

Neural networks require matrix computation in both the forward and backward propagation steps. For every datapoint in the training set, the forward propagation step applies the activation functions. After a forward propagation step has been completed, the backward propagation step needs to compute the gradient in order for the gradient descent algorithm to optimize the cost function during training. In addition to tuning the model's weights, neural networks also need to tune hyperparameters, which adds to the complexity of the task. For large inputs, DNNs can take a long time to train and require specialized techniques that can exploit parallelism within the model's architecture. The ability to parallelize the training of neural networks is central to the improvement of training times, and increasing the accessibility and scalability of these methods.

2.3 Deep Learning Architectures

Deep neural networks are roughly defined by the depth of their architectures. While Fig. 2.6 is representative of a simple neural network with 3 hidden layers with 5 neurons each, neural networks can take many different shapes and sizes. The structure of a neural network typically refers to its architecture. Different architectures have been de-

veloped for different domains and tasks – Convolutional Neural Networks are designed for working with images [29].

2.3.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a specialized form of feedfoward neural networks that are particularly effective at processing data with grid-like topologies, such as images [29]. CNNs have been effectively utilized for object detection [14].

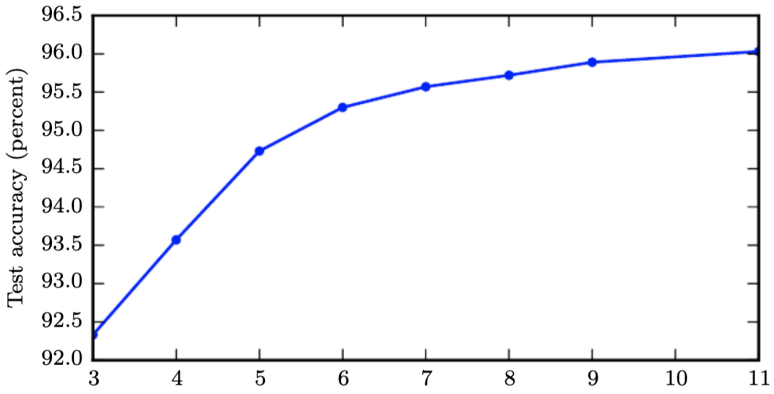

Figure 2.8: Plotting the accuracy of a deep learning model as a function of the number of layers. These are results of a model that attempted to transcribe multidigit numbers from photographs of addresses [29].

Fig. 2.9 illustrates the increase in accuracy of a convolutional neural network as the number of layers increases. An increase in layers yielded an increase in overall model performance for multidigit number transcription from images, which indicates that a deeper neural network has stronger overall performance. Increasing the depth of a model is an effective way to improving a model's fit to the data [29].

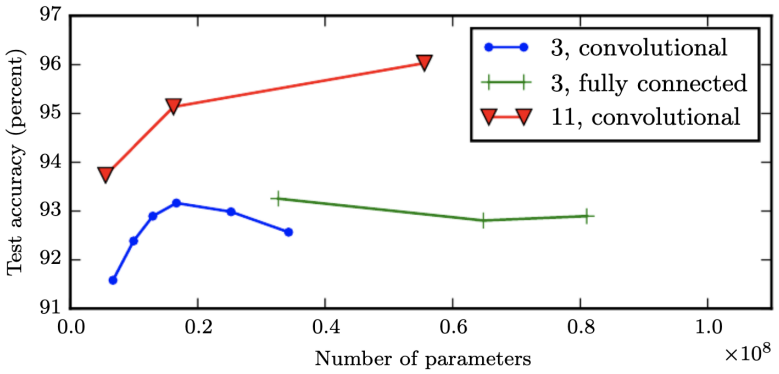

Figure 2.9: According to Goodfellow *et al.*, the increasing the size of a model through an increase in parameters without a corresponding increase in depth does not improve model performance. [40].

As a deep learning method, CNNs have more than one hidden layer. The specialized nature of convolutional neural networks is reflective of the dimensionality of the input data – images that are fed into these networks are multidimensional grids – the number of weights required to support multidimensional inputs in a typical deep learning architecture are computed by multiplying the three dimensions of the input image size [26]. CNN architectures are designed to work effectively with large matrix representations of images [29].

Take for example a 2 dimensional colored image – the image itself (regardless of color) has a length of a pixels and a width of b pixels. Disregarding color, the matrix that represents this image has 2 dimensions – $a \times b \in \mathbb{R}^2$ [29]. A colored image typically has three *color channels*, which are red, green, and blue [41]. An $a \times b$ image represented by 3 color channels would yield a matrix with 3 dimensions.

A regular neural network that wants to process a $32 \times 32 \times 3$ image will require $32 \times 32 \times 3$ or 3072 weights. For larger images, the number of weights that need to be computed taxes the available computational resources, increasing training time and decreasing efficiency. [26]. CNN architectures are designed to take advantage of the multidimensional nature of image inputs. In particular, unlike in regular neural

networks, such as that depicted in Fig. 2.6, the layers of a CNN are connected to a subset of neurons in the adjacent layers.

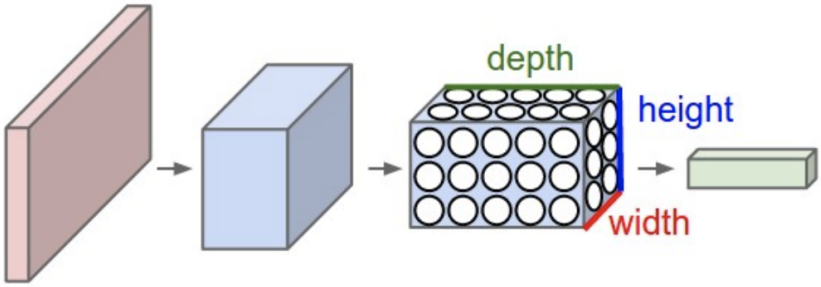

Figure 2.10: Convolutional neural networks perform transformations on images by structuring the architecture along the input image's width, height, and depth. The depth of an image refers to the color channels – typically 3[26].

CNNs are able to model higher dimensional image data by minimizing the number of connections between layers in the network. By exploiting the matrix representation of images, CNNs are better suited to take images as input than other deep learning methods. The neurons of CNNs are arranged in 3 dimensions, defined by the image's height, width, and depth, as depicted in Fig. 2.10 [26]. The depth, with respect to CNNs, is a in reference to the depth of the image – generally referring to the number of color channels an image is represented by.

CNNs are typically composed of *convolution layers*, *pooling layers*, and *fully connected* layers. On top of this, there are operations that are common in CNNs, including *stride*, and *padding* [14]. The combination of these operations and layer types has led to the development of state-of-the-art performance on image classification challenges [14].

Convolutional layer

Convolutional layers are typically interspersed throughout the network architecture. These layers apply a special convolution operation on a matrix representation of an image [41]. The convolution operation attempts to extrapolate the most important and

useful information from that image. In practice, if the matrix representation of a 5×5 image will serve as the feature map, applying the convolution operation requires a *kernel* that will facilitate convolution operation.

The kernel is a smaller matrix that will compute the element-wise product and sum over all values with various subsections of the image matrix, and then divide by the size of the filter. After each product and division, the kernel is moved a certain number of elements, defined by the stride [41]. Often times, specific kernels will be applied in a way that is meant to extract some visual information from the image itself – in these cases, a kernel is essentially a filter trying to generalize information out of the original image. The linear combinations applied by the kernel on the input image yield *affine transformations*, where simple linear combinations are applied followed by the addition of some bias to be passed through a non-linearity [41]. Affine transformations typically preserve geometric qualities such as colinearity and ratios, which maintain an image's qualities [41]. In order to improve model performance by reducing the complexity associated with multidimensional image data, the use of padding is sometimes introduced in order to condense the image size that is outputted after a convolution. Padding refers to pixels at the edge of an image that will not have the kernel applied.

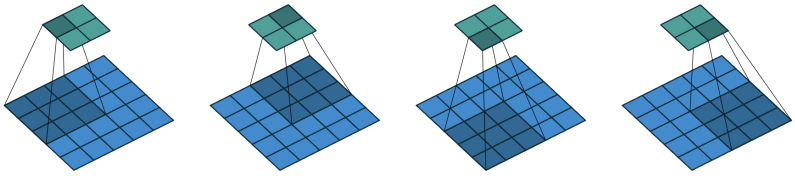

Figure 2.11: The convolution operation applied by a 2×2 kernel on a 5×5 image. On each iteration, the kernel is placed on a part of the image, and a element wise product and sum is computed and then divided by the size of the kernel (in this case, $2 \times 2 = 4$). After each application, the kernel moves one element to the right and repeats the process until the edge of the image is reached at which point the kernel moves down one pixel and the process repeats until the kernel has been applied to the entire image. This particular convolution has stride 1×1, which indicates we moved the kernel 1 box after each application, and a padding of 0, since the kernel used all elements of the image matrix. [41]

.

Fig. 2.11 exemplifies a simple convolution operation with 0 padding applied at stride 1. The kernel is moved across the input image and applied as a filter. The size of the output can be directly computed from (2.14).

$$(W - F + 2P)/S + 1 \tag{2.14}$$

where W is the size of the input image, F is the size of the kernel, S is the size of the stride, and P is the size of the padding [26]. A 7×7 image with a 3×3 filter, stride 1, and pad 0 would yield a 5×5 output [26].

Pooling layer

The Max Pooling layer is another important building block of CNNs. Max Pooling layers are like convolutional layers in that they apply operations to subsets of the image, but as opposed to applying a linear combination, these layers apply the *Max Pooling* function. The Max Pooling function finds the max in a rectangular neighborhood on the input grid [29]. Pooling layers are especially useful in introducing invariance to the model. Invariance states that small perturbations in the input will not impact the results

of the pooling layer. There are cases in image classification and detection where it is more important to detect a feature, such as eyes in portraits of humans, than to actually pinpoint the pixels that represent the eyes – in these cases, the invariance introduced by max pooling layers is most useful [29].

The convolution operation can become computationally expensive when it needs to be applied to a large data sets of large images. Training a CNN on these images would result in many convolution operations across each input image, a problem that scales with the size of the data set that the model is being trained on. CNN weights can be optimized using the methods discussed in Section 2.1, which adds additional complexity to the challenge of processing images that are already multidimensional.

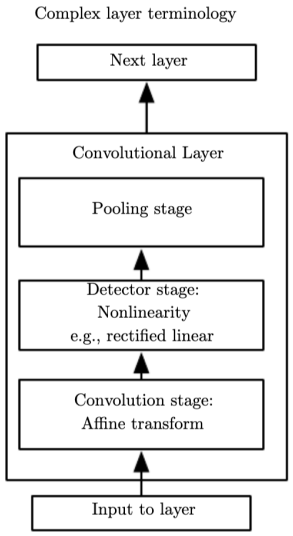

Figure 2.12: Convolutional Neural Networks are composed of input layers and convolution layers. The convolutional layers include the convolution operation, the application of a non-linearity such as ReLU, and pooling. [29]

.

Fully connected layer

The final stages of a CNN are often composed of fully connected layers [14], which are analogous to the hidden layers referenced in Fig. 2.6. Fully connected layers have full

connections with all the activations in the preceding layer, analogous to regular neural network architectures [26]. As a deep learning method, the performance of a CNN is tied to its depth. Fig. 2.9 shows that increasing the size of a model through the addition of parameters of a given layer in the network does not necessarily improve the model's performance. Adding depth to a model also adds complexity, which means more iterations of back propagation and gradient descent need to be applied. While CNNs are computationally expensive, they present several opportunities for acceleration via parallelism – through both expert designed methods as well as generalizable frameworks.

CHAPTER 3

PARALLELISM

Parallelism is a method that can be used in the acceleration of DNNs. Within deep learning methods, parallelism attempts to optimize the computation performed during the training of a model by minimizing the overlap of operations and maximizing throughput. In particular, DNNs have a large number of computations, and parallelism attempts to perform as many of them in parallel as opposed to in sequence.

Parallelism has shown significant training speedups in various deep learning methods [19]. Training times are extended as a result of models that need to tune a higher number of weights through back propogation and gradient descent. Through the graphical representations of DNNs and the existence of robust machine learning *Application Programming Interfaces* (APIs) such as TensorFlow [34], Keras [42], PyTorch [43], DNNs have increasingly been subjected to parallelization. APIs are interfaces built on top of existing programming languages that simplify development in various domains – for example, machine learning APIs in a particular programming language make it easy to build models using that language. In this way, APIs like TensorFlow with Python implementations are able to provide access to machine learning tools via Python. While TensorFlow provides computational graphs and linear algebra operations that compose neural network operations, simplifying the design of models, higher level APIs such as Keras provide further abstractions to the TensorFlow operations. In particular, the *sequential* module within Keras facilitates the development of DNNs by sequentially stacking layers [42]. Combined with additional levels of abstraction that allow users to place specific layers on different devices, frameworks such as Keras and TensorFlow are pushing the democratization of machine learning by allowing non-experts to design highly predictive DNNs without having to rigorously understand and define the details underlying specific operations [42]. Users can define a convolution layer in Keras with

a single line of code:

```
model.add(Conv2D(filters=96, input_shape=(56,56,3),
                 kernel_size=(11,11), strides=(4,4),
                 padding='valid'))
```

Device level control, efficient data partitioning, and the exploitation of model architectures has led to the development of a number of effective parallelization techniques that have facilitated the acceleration of deep learning methods.

Like individual operations, such as convolution, the parallelization of deep learning models should also be abstracted. In treating parallelization as an abstraction, models become more efficient and more accessible on on a wider range of computational resources [19].

3.1 Distributed Deep Learning

In the development of deeper neural networks with more weights, the ability to efficiently distribute computational load across a machine's resources plays a large role in the acceleration of training times [31]. A machine's resources include hardware designed for processing computations – most commonly in the form of CPUs and GPUs. The usage of GPU clusters to distribute the computational operations of a given model accelerates training by efficiently computing gradients via algorithms like backpropagation [31], decreasing the time it takes to optimize the model's cost function and produce better fit to the data, as discussed in Sections 2.1 and 2.2. While general purpose CPUs are utilized in a wide range of tasks, such as event driven desktop applications and server side tasks (serving web-pages, executing business logic, etc.), GPUs provide the accelerated computational power that can perform the computations associated with a DNN [44].

Effective parallelism requires the optimized use of available machines and devices, as well as an understanding of a model's architecture. Device usage is optimized by

minimizing the time that a device sits idle waiting for a task. For example, a model could potentially experience speedups if computations were split across multiple GPUs, or if the architecture was designed to utilize CPUs and GPUs in parallel. Section 4.2.2 describes experiments designed around the AlexNet architecture exploring various parallelism schemas.

There are several popularized methods that have been successfully applied in the acceleration of DNNs, including the use of *parameter servers*, *data parallelism*, and *model parallelism*. All of these methods have been used in both isolation and in combination with each other. When compared quantitatively, they present insights into the ways in which deep neural networks can reduce redundant computation, thereby accelerating training performance. In addition to improving performance, these methods' optimization of device usage is capable of making deep learning methods more accessible on machines with less resources – a valuable contribution towards making deep learning methods more accessible and scalable.

3.1.1 Parameter Servers

An approach to distributing training computations across devices is facilitated through the use of *parameter servers*. This is a technique by which parameters can be aggregated and exchanged between different machines during forward and backwards passes of the training algorithm. The benefits of distributing a model across machines and devices is a function of the model's architecture and computational needs. In general, deeper neural networks with an arbitrarily high number of neurons per hidden layer cause the computational expense associated with training those models to go up [18]. Deep neural networks are composed of a number of complex operations that all present opportunities for parallelization. Two examples of parameter server implementations are Downpour Stochastic Gradient Descent and Sandblaster L-BFGS.

It has been shown that *asynchronous Stochastic Gradient Descent* is effective in the context of *non-convex* optimization [31]. While *asynchronous* refers to the fact that copies of the model run on individual machines independently of each other and the parameter server does not block updates, *non-convex* optimization refers to the process of optimizing a curve where saddle points (Fig. 2.4) might exist. This is significant because the optimization used in many deep learning models might be non-convex [45]. SGD facilitates training and parallelizing models that with cost functions that are not necessarily convex [31].

A variant of asynchronous SGD, called *Downpour SGD* splits a training set across multiple machines, and runs a copy of the full model on each subset. After a full run, the different machines communicate with a central parameter server that stores the current weights of the model [31]. The image in the left of Fig. 3.1 illustrates how a data set is split into several shards which are then fed into distinct machines with replicas of the same model. These machines then communicate with a central parameter server that helps compute gradients after accumulating outputs from the model replicas.

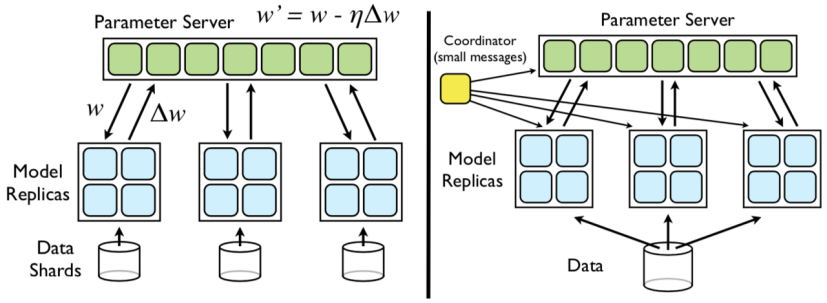

Figure 3.1: Two algorithms for distributed training, as part of the DistBelief software framework [31]. The left image represents Downpour SGD, and the right image represents Sandblaster L-BFGS.

Another example application of parameter servers is the *Sandblaster L-BFGS* algorithm. L-BFGS is a limited memory optimization algorithm that can be used to minimize a cost function [46]. The Sandblaster variant of L-BFGS is a distributed implementation

of the original algorithm [31]. Sandblaster L-BFGS relies on distributed parameter storage and manipulation, as depicted on the right side of Fig. 3.1. The coordinator process within Sandblaster L-BFGS does not have direct access to the model parameters within the parameter server, but it can run operations on distinct shards within the parameter server itself [31].

As distributed optimization algorithms, asynchronous SGD and Sandblaster L-BFGS are representative of the ways in which parallelization can yield significant acceleration benefits on training and optimizing DNNs [31]. Both asynchronous SGD and Sandblaster L-BFGS work to minimize a cost function, such as (2.7).

The aggregation of parameters is a central method used in the many parallelization methods, especially as it facilitates cost function optimization [31]. The parallelization schemas tested with AlexNet in Section 4.2.2 internalize the notion of parameter servers through abstractions provided in the Keras API used to implement and train the model architecture.

3.2 Data Parallelism

Whereas parameter servers are used to update parameters while training a model, data parallelism attempts to distribute training examples across the set of available devices. Within deep learning, there are 2 primary monoliths that can be broken up in order to facilitate parallelization: the data, and the model architecture itself. The data is a convenient splitting point because each data point is independent in the context of SGD and backpropagation. The optimization of the cost functions described in Section 2.1 is independent of the order in which data points x_i are passed to the data set. In fact, the only interaction any particular data point has with another during training is their cumulative effect on the model's weights through SGD and backpropagation computations.

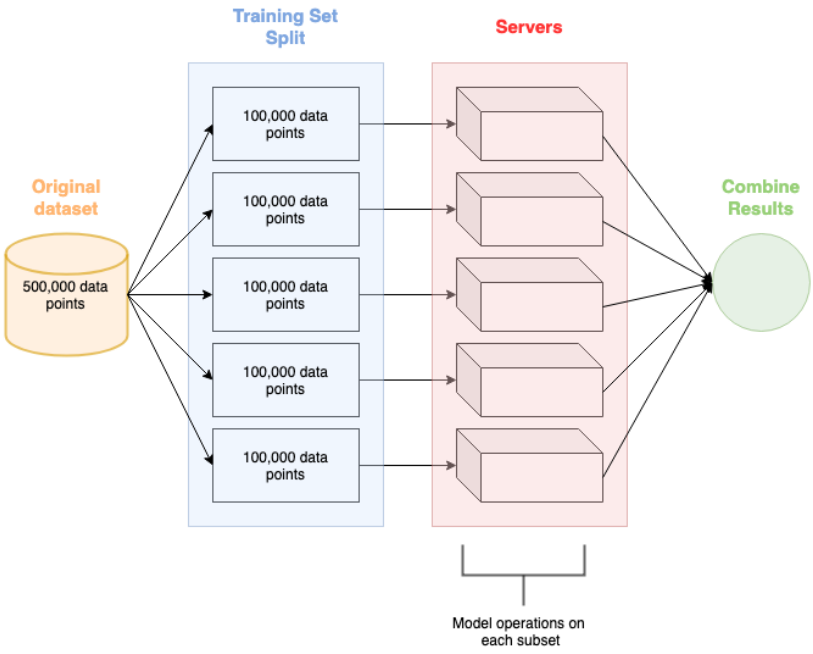

Figure 3.2: Data parallelism replicates the model across each device, and splits the training data set such that each device runs the entire model on a subset of the data. The original 500,000 data points here are split into subsets of 100,000, fed into each device to be processed by a copy of the model, with all outputs combined to get the training result for the full data set.

Depicted in Fig. 3.2, data parallelism splits the dataset equally across the available devices. The model replicas on the distinct devices run operations on equal sized subsets of the data, and then synchronize the parameters from each of the model replicas after each training iteration [19]. The synchronization of parameters takes place after each of the devices completes all the model's operations on its assigned subset – synchronization is necessary to ensure weight updates and the backpropagation algorithm have the correct values for a particular iteration of the model. Across large clusters with more than a single machine, this aggregation can happen through the use of a parameter server as discussed in the previous section, but this synchronization can also happen on an available GPU that is being underutilized at a particular step in training.

The optimization of a model's cost function through gradient descent can be accelerated via data parallelism. Applications of data parallelism are common in any setting

in which the training data can easily be mapped to different devices. A common variant of SGD is *minibatch SGD*, where data is processed in batches of N samples, where $0 < N < m$, and m is the number of training samples [44]. The benefit of minibatch gradient descent, discussed in Section 2.1, is that it facilitates gradient operations across a sample of data as opposed to a single data point. Parallelized minibatch SGD has accelerated the training of *ResNet-50*, a image classification model trained on ImageNet [47]. Thus, the partitioning of minibatch samples across multiple devices that have replicas of the full model architecture facilitates parallelized gradient computation.

Data parallelism has also yielded accelerated training performance for the VGG model, the winner of the 2014 ImageNet ILSVRC [48]. VGG splits the training images across several GPUs to be processed in parallel. After processing is complete and each GPU has computed each respective batch's gradients, these values are averaged in order to compute the gradient of the full batch. This application of data parallelism yielded a 3.75x speedup using 4 NVIDIA Titan Black GPUs, and took approximately 2-3 weeks to train [48].

As in VGG, the application of data parallelism to AlexNet requires splitting training images across available GPUs. The AlexNet architecture would be replicated on each device and the full CNN is applied to subsets of the data for each training epoch. In Section 4.2.2, I test the effectiveness of data parallelism in image classification model training through the AlexNet architecture.

3.3 Model Parallelism

While the dataset itself is easily split onto different devices for parallelization, the model architecture can also be split up. The sequential module within Keras allows us to define DNNs by using the layer as the level of abstraction [42]. In defining layers independently, it is possible to split the model architecture across various devices.

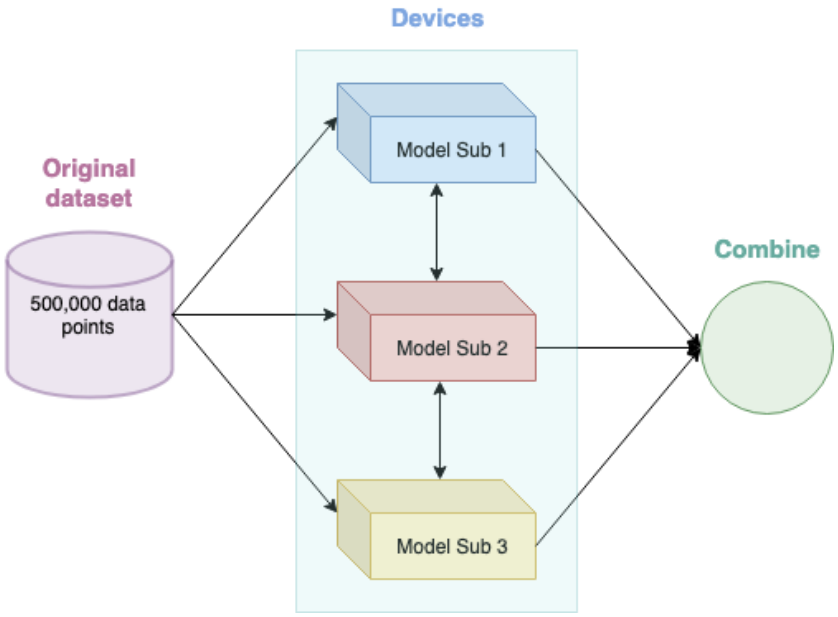

Figure 3.3: Model parallelism splits the model across multiple devices, and runs each data point through all model sub components on the corresponding devices. The original 500,000 data points are fed into each device to be processed by the sub-components of the model, with all outputs of the model combined to get the training result for the full data set.

Model parallelism, depicted in Fig. 3.3 was used in the original implementation of AlexNet [14]. This parallelism paradigm assigns subsets of a model's architecture to dedicated devices, and then runs the full training set through each of the devices [19]. This setting is different from data parallelism in that instead of a full replica of the model being placed on each device, different components of the model are assigned to each device. Furthermore, the full data set is passed through each device, and all devices need to communicate between forward and backward passes of the training algorithm to ensure proper gradient computations.

Model parallelism is effective in situations in which there is less computation, smaller representations, and a large number of parameters [23]. Since every available device is responsible for processing each data point in the training set, model parallelism needs to identify situations where the model's architecture has more connections between neurons, and therefore weights that need to be calculated. An example of such a model

component would be fully connected layers in CNNs (discussed in Section 2.3). These layers have more connections and weights associated with those connections, which imposes additional computations that can be parallelized via model parallelism. While fully connected layers have more parameters, they comparatively comprise only a small fraction of the total computation in a CNN since they are less computationally expensive than convolution operations [23].

One of the challenges surrounding the parallelization of DNNs is the ineffective application of individual methods in isolation – Section 4.3 describes the results of applying parallelization methods to AlexNet in isolation. While there are ways in which various parallelization methods can be combined [23], identifying these situations on a model by model basis is inefficient. The effective application of parallelization methods often requires an exhaustive understanding of the underlying model architecture, which impacts a method's scalability.

Frameworks, designed to facilitate the acceleration of DNN training, provide an efficient and scalable avenue through which model parallelism can be combined with data parallelism, as well as other novel approaches to parallelism [19, 20]. Frameworks treat parallelism as an abstraction in the same way that machine learning APIs treat model implementation as an abstraction. Treating parallelism as an abstraction makes it easier to accelerate the training times of deep learning methods. The parallelization methods discussed in Section 4.2.2 required custom implementations – parallelism as abstraction would have parallelism as a feature of the API used to build the model.

3.4 Hybrid Methods

Data and model parallelism are methods that are agnostic of the programmatic representation of the underlying model. Specifically, data parallelism creates replicas of the model architecture on each device, and model parallelism splits various layers across

devices. While the underlying implementation of the model architecture does not directly modify or impact the application data or model parallelism, it does inform the design of other parallelization methods [20].

Some forms of parallelism attempt to exploit the graphical representation of DNNs. Popular machine learning frameworks such as TensorFlow [34] represent models as computational graphs. This graphical representation can be parallelized across devices [34, 42].

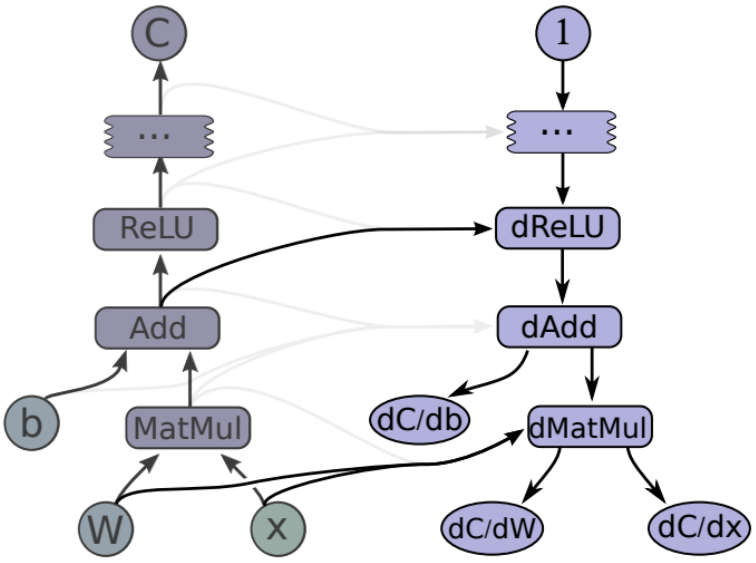

Figure 3.4: On the left, there is a graphic representation of simple forward propagation step using TensorFlow, while on the right, there is the corresponding graphic structure of the gradient computations [34].

Deep learning frameworks, like TensorFlow recognize the need for parallelism through an API interface that provides device level control for individual operations [34]. Fig. 3.4 depicts the graphic representation of gradient computation through TensorFlow operations. TensorFlow utilizes *greedy rule-based substitutions* to optimize a computation graph, where each substitution replaces a subgraph matching a specific pattern with a new subgraph that computes the same result [49]. Graph substitutions take the struc-

ture of a network, and then parallelize operations as a way of reducing redundancy by identifying parts of the model that are identical. Training performance is improved by reducing the extraneous computation. For example, a model that computes gradients all at once can have this step in the graph split across multiple available workers, which is an example of model parallelism facilitated by TensorFlow.

Other methods have approached the question of parallelizing DNNs underlying computation graph through *relaxed graph substitutions*, where substitutions on a subgraph are allowed even if performance is not improved, but the underlying semantics are preserved [49].

TicTac, a framework for parellelizng DNNs built with computational graph structures, aims to identify the best schedule of parameter transfers as a way of reducing blockage for computation [20]. In particular, TicTac attempts to improve overlap and iteration time by identifying optimizations in the model's DAG representation. TicTac is an example of a DNN parallelization method that attempts to exploit the structure of a computational graph in order to accelerate the training of deep learning methods. Built on top of TensorFlow, which represents model computations in graphs, TicTac is able to improve overall throughput in training by up to 19% [20] for DNNs.

Viewing the acceleration of DNNs as a graph optimization problem is not scalable because it is not framework agnostic. These methods assume that the computational model of a network is represented as a graph, which means these methods will not work with other frameworks that potentially do not represent computation as graphs. Model and data parallelism parallelize the models themselves, while graph based methods parallelize the model's representation.

3.5 Frameworks for Parallelism

FlexFlow is the primary framework for parallelizing Deep Neural Networks that I explore in this work. FlexFlow is a generalizable, architecture agnostic approach to parallelizing deep learning methods [19]. As opposed to methods designed for use in particular applications or architectures [23], FlexFlow can be applied to a wide range of model architectures, allowing it to scale to multiple domains.

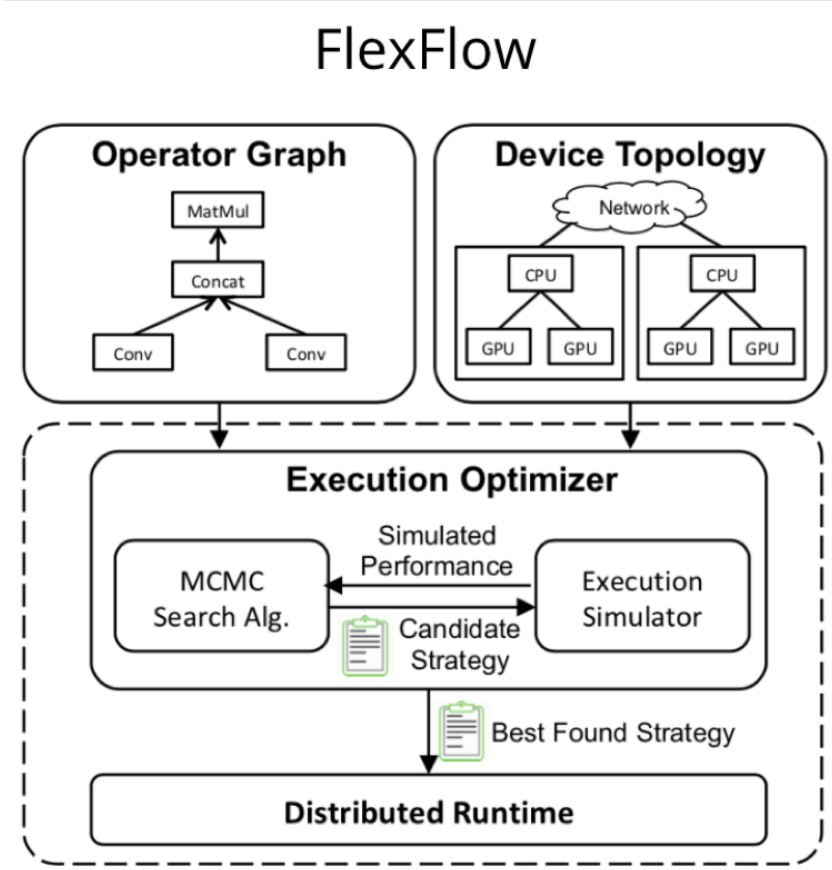

Figure 3.5: FlexFlow is an architecture agnostic framework for parallelizing deep learning methods. Given a device topology and operation graph, FlexFlow provides an optimized parallelized training strategy [19].

As depicted in Fig. 3.5, FlexFlow takes an operation graph that represents a model's architecture and a device topology that describes the computational resources available, and produces an optimized parallelization strategy that can be used to accelerate the

training of the model. Just as Keras provides functions that allow users to quickly define a model architecture [42], FlexFlow allows users to develop an optimized parallelization strategy by providing information pertaining to their training environment.

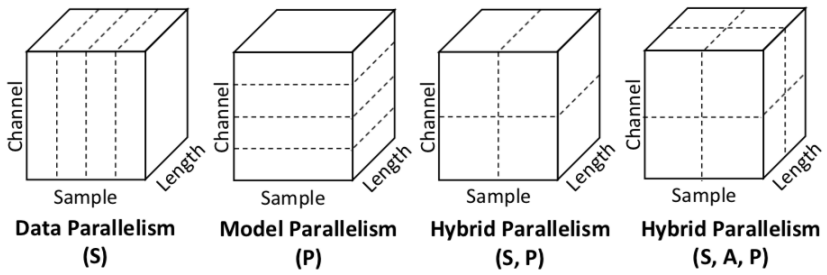

Figure 3.6: Example parallelization configurations for 1D convolution. Dashed lines represent how various input matrices can be split [19].

FlexFlow is able to produce an optimized strategy for parallelization by increasing the parallelization search space [19]. While data parallelism parallelizes models along data samples and model parallelism parallelizes models by splitting operations and parameters, FlexFlow observes samples (S), operations (O), parameters (P), as well as attributes (A), which are an additional way to split particular data samples. By exploiting the SOAP parallelization search space shown in Fig. 3.6, FlexFlow is able to improve performance over methods like data and model parallelism [19].

FlexFlow has been applied to a number of image classification models, including ResNet-101, Inception-v3, and AlexNet [19] – the primary benchmark explored in this work. The architecture agnostic nature of FlexFlow makes it a powerful tool for parallelizing deep learning methods and increasing the efficiency of device usage across varying device topologies. FlexFlow makes machine learning methods more accessible by accelerating training across model architectures and obviating the need for rigorous analysis of model architectures in order to achieve parallelized performance.

CHAPTER 4

PARALLELIZING DEEP NEURAL NETWORKS

4.1 Image Classification through AlexNet

The benchmark for this work's parallelization experiments is the AlexNet CNN architecture, originally proposed in 2012 [14]. As a state-of-the-art image classification model, AlexNet has influenced the development of other CNNs and the application of deep learning methods to the field of image classification, including VGG [48] and ResNet-50 [47]. Successfully parallelizing AlexNet and increasing training efficiency will provide valuable insight into how other deep learning models can be similarly accelerated.

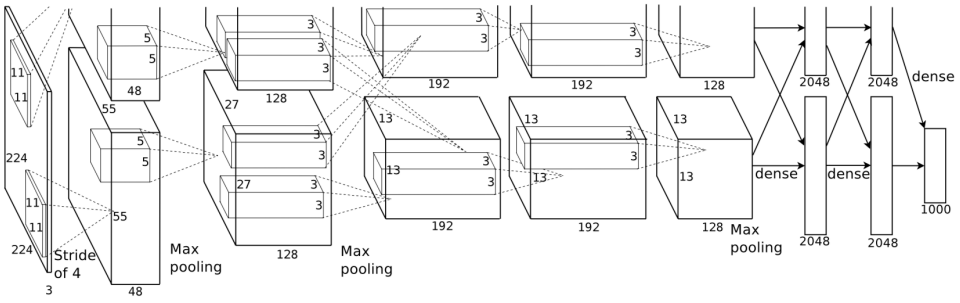

Figure 4.1: The original AlexNet architecture description from the 2012 ILSVRC. The model architecture is composed of 5 convolutional layers, 2 normalization layers, 3 max pooling layers, and 3 fully connected layers, comprising 60 million parameters and 650,000 neurons [14]. The original model was trained across 2 GTX 580 GPUs over 6 days, with the feature space split across the 2 GPUs.

Fig. 4.1 is a summary of the original AlexNet architecture. AlexNet was originally trained on the ImageNet dataset, using color images of 227 x 227 pixels (227 x 277 x 3). The model achieved a top-5 test error rate of 15.3% in the 2012 ILSVRC, compared to 26.2% achieved by the second-best entry [14]. AlexNet was one of the first deep learning methods that applied ReLU non-linear activations (DNNs trained on ReLU have been shown to exhibit speedups when compared to the usage of tanh activations

[14]). AlexNet used dropout regularization to avoid overfitting and the model's weights were optimized using SGD and backpropagation [14], discussed in Section 2.2.

4.2 Experiments

The objective of these parallelization experiments is not to necessarily achieve a higher accuracy (the AlexNet architecture has already been proven as an effective image classification model [14, 23]); instead, this work attempts to accelerate the training of AlexNet. While optimizing training time can potentially yield lower accuracy, the accuracy achieved across all parallelized methods are analogous to the baseline implementation without any parallelization configuration – thus, accuracy is held constant as training time varies between parallelization methods.

Training time is used as a proxy measure for the efficiency of parallelization methods – a faster training time is representative of a parallelization method that utilized computational resources more efficiently. Efficiently managing resources while training AlexNet will inform the ways in which similar models can be trained across available resources and varying device topologies.

4.2.1 AlexNet Training Data

This work utilizes two data sets to evaluate the effect of various parallelization methods on AlexNet's training time. The first is *Tiny ImageNet*, based on the original ImageNet dataset used for an image classification competition at Stanford University [24]. Tiny ImageNet has images in 200 discrete classes, defined by synsets from the original ImageNet data set. The classes in ImageNet are derived from WordNet synonym sets (*synsets*), which are sets of words and phrases with similar meanings [1]. The synsets that comprise the classes in Tiny ImageNet can be found within the original ImageNet

data set as well.

A small sample of the images in various classes are displayed in Fig. 4.2. For each of the 200 distinct classes, Tiny ImageNet provides 500 training images, 50 validation images, and 50 test images, for a total of 120,000 images in the dataset. Each image has dimensions of 64 x 64 x 3.

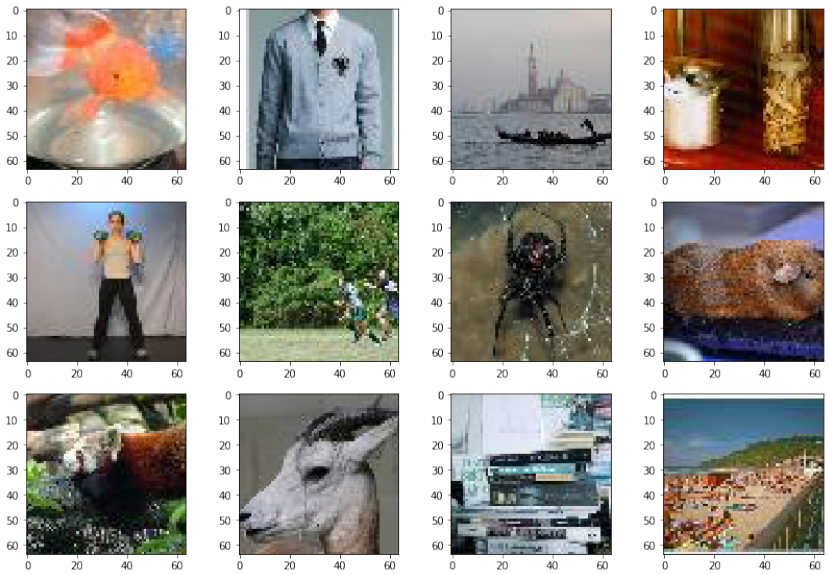

Figure 4.2: A sample of the images provided in the Tiny ImageNet dataset, made available by an image recognition challenge at Standord University. [24]

The second dataset that is used to train AlexNet is the Dogs vs. Cats dataset, provided as part of a binary image classification competition [50]. This dataset contains two classes: one representing dogs, and the other representing cats. The full training archive contains 25,000 images.

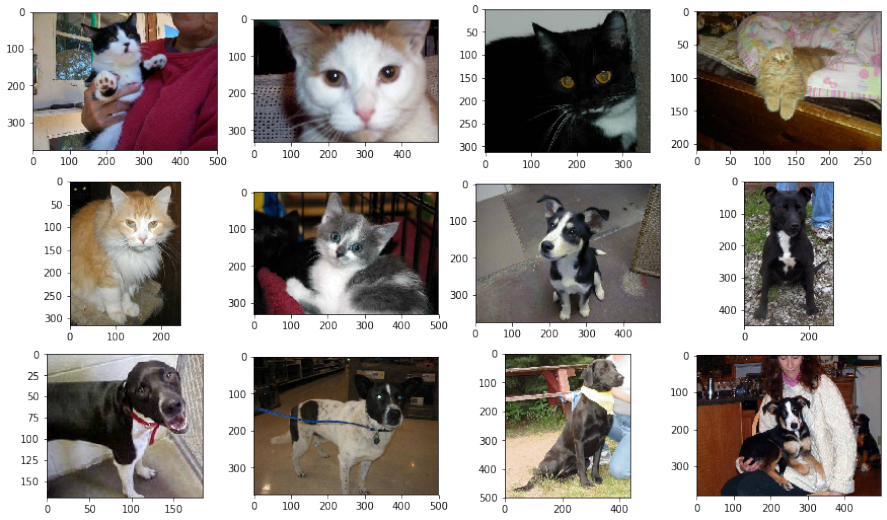

Figure 4.3: A sample of the images provided by the Dogs vs. Cats data set, made available by an image recognition challenge on Kaggle. [50]

Some of the images in the Dogs vs. Cats dataset are shown in Fig. 4.3. Unlike Tiny ImageNet, there is no consistency between image sizes and quality in the Dogs vs. Cats dataset, so a smaller subset of 2,000 training images and 800 validation images was used to reduce complexity related to processing images of varying quality, which is not directly relevant to this work's exploration of the acceleration of AlexNet's training.

For AlexNet trained on Tiny ImageNet, all 120,000 training and validation images were used. The Dogs vs. Cats data set has 100,000 less images than Tiny ImageNet, and even less were used in these experiments – training on both of these data sets reflects training times on large and small data sets with varying as well as consistent image sizes. All of the images in Tiny ImageNet are 64 x 64 x 3 in size. The images passed to the model are scaled down to 56 x 56 x 3. While the images in the Dogs vs. Cats data set vary in size, they are scaled to 128 x 128 x 3, which is representative of the scale used in the highest performing classification models that participated in the kaggle competition [25].

4.2.2 Benchmark Baselines

The implementation of AlexNet for these parallelization methods are built via the Keras machine learning API (version 2.2.4) [42] using the tensorflow-gpu backend (version 1.13.0) [34]. In particular, Keras' underlying mathematical representations are handled by TensorFlow's graph based API. Keras is the primary API used in this work as it provides a robust deep learning API with a sequential model abstraction. Defining layer-wise operations individually simplifies the implementation of AlexNet, and makes it easier to implement various parallelization schemas.

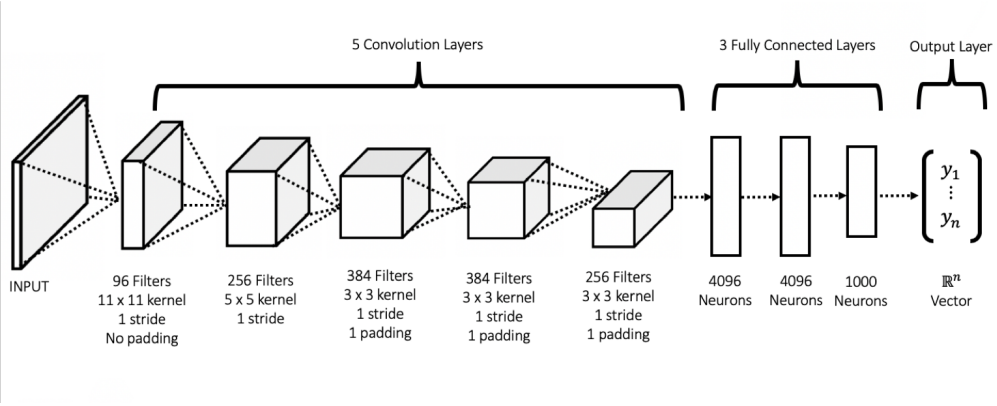

Figure 4.4: The AlexNet variant used for this work's parallelization experiments. [50]

.

As depicted in Fig. 4.4, the AlexNet variant used to train on the Tiny ImageNet dataset has the same overall architecture as the original AlexNet with 5 convolution, 3 max pooling, 2 normalization, and 3 fully connected layers. The primary difference is that this model uses different input image sizes as previously described, and has varying numbers of output nodes. For AlexNet trained on Tiny ImageNet. the output layer applies a softmax function that reports the probability distribution across the 200 distinct classes for a particular example x_i, producing a $\mathbb{R}^{200}$ vector. Similarly, the AlexNet variant trained on the Dogs vs. Cats data set uses the same architecture, with the only difference being that the output node computes a softmax activation providing the prob-

ability distribution over the 2 classes, outputting a $\mathbb{R}^2$ vector. In both cases, the highest probability within the distribution is the predicted class.

Both models are trained using batch sizes of 16, for 20 epochs, with 2,000 steps per epoch, with 800 validation samples used per epoch. The models had a learning rate of 0.01 with a decay rate of 10^{-6}. These hyperparameters were kept consistent across the variants in order to establish a baseline model trained on 2 different data sets of varying sizes. Because these experiments are aimed at optimizing training computation instead of model performance, hyperparameter tuning is not performed.

For baseline benchmarking, both models were trained on a single machine through the Google Cloud Platform [51]. The GCP instance used a Debian image with Linux Kernel version 4.9.0. The machine had 2 virtual CPUs with 13 GB of RAM. The GCP instance also had 2 NVIDIA Tesla P100 GPUs. Deep learning would be more accessible if training acceleration gained through parallelism would allow for models like AlexNet to be trained in a reasonable amount of time on a given device configuration. The experiments in this work are based on a dual GPU system, but benchmarks in other works suggest that performance can be scaled with the addition of more GPUs [19].

The baseline models did not implement any optimizations besides allowing the TensorFlow backend to allocate resources as it would by default. By default, TensorFlow always attempts to map the model to all available GPU memory [52] – thus, parallelization strategies are particularly useful in cases where the model architecture has a parameter space that cannot fit in the GPU memory all at once. This default strategy is the same across all systems, but performs better in systems with powerful GPUs since they have more memory to map to – the objective of this work is to compare the baseline performance with parallelization methods against a specific hardware configuration. In these experiments TensorFlow used the following devices: [/cpu:0, /device:GPU:0, /device:GPU:1], where /cpu:0 is the CPU, and /device:GPU:0 and /device:GPU:1 are the

NVIDIA Tesla P100 GPUs. The system uses NVIDIA drivers with CuDNN v10.0, a GPU-accelerated library of primitives for deep neural networks [53].

The accuracy associated with training AlexNet on the Tiny ImageNet data set using the baseline configuration is approximately 5%, which is analogous to the the accuracies observed across all the parallelization methods. Similarly, the accuracy associated with training AlexNet on the Dogs vs. Cats data set using the baseline configuration is approximately 90%, which is analogous to the accuracies observed across all the parallelization methods trained on the same data set. Thus, we hold accuracy constant, and train over 20 epochs.

AlexNet Baseline Benchmarks for Tiny ImageNet

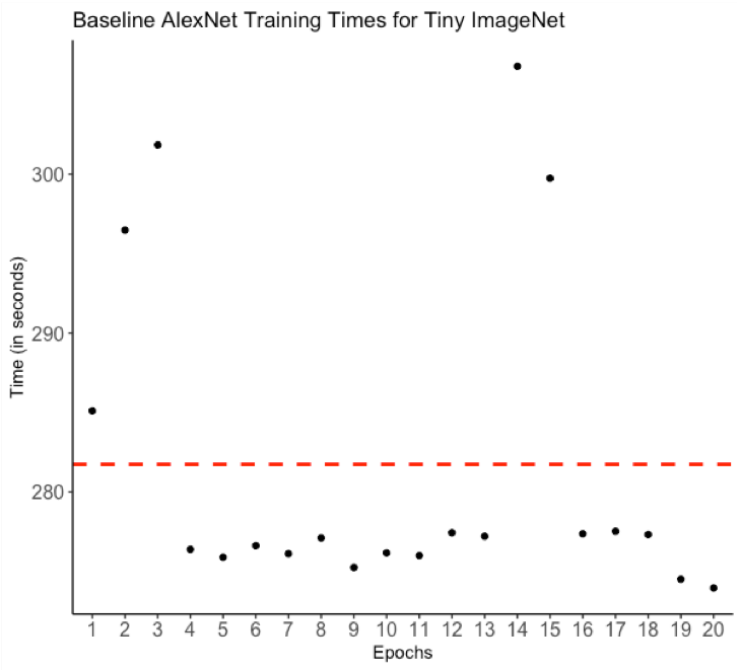

Figure 4.5: The baseline epoch time performance of AlexNet when trained Tiny ImageNet over 20 epochs. The dashed red line indicates the average time per epoch, 281 seconds.

According to Fig. 4.5, the average time per epoch while training AlexNet on Tiny ImageNet is 287 seconds. The first epoch takes the longest to complete at 285 seconds

seconds. The latency within the first epoch is observed because while training a model built using Keras and a TensorFlow backend on top of NVIDIA GPUs configured with CuDNN, the model compilation takes place within the first epoch of training. Model compilation refers to the steps Keras takes in order to represent a defined model architecture using TensorFlow as a backend. After the initial delay, it is possible to observe that processing a total of 2,000 (56 x 56 x 3) images over 20 epochs has yielded consistent performance across subsequent epochs.

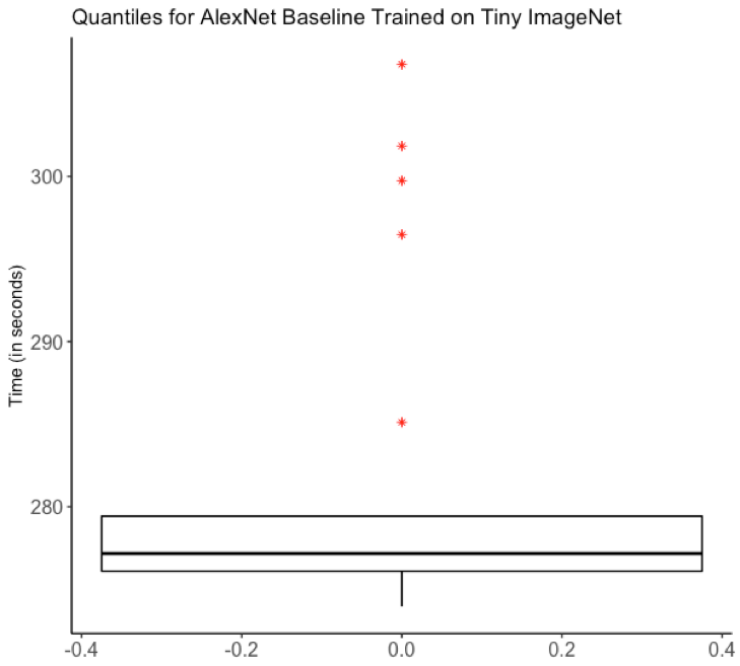

Figure 4.6: The 75th and 25th percentiles for epoch times within the baseline training iteration for AlexNet on Tiny ImageNet is between 277 and 279 seconds, respectively, with a mean of 281 seconds. The red asterisks represent outlier training times.

Fig. 4.8 shows that 50% of AlexNet's training times per epoch are consistent between 277 and 279 seconds when trained on Tiny ImageNet. This consistency between epochs is likely a function of the model's inputs having consistent size, being passed to a uniform device configuration. Subsequent parallelization experiments attempt to maintain the consistency between times per epoch – variability in training epoch times might indicate inefficient computation parallelization or communication overhead be-

tween devices as a function of a method's implementation. See Appendix A for the full data summary for AlexNet's baseline epoch times when trained on Tiny ImageNet.

AlexNet Baseline Benchmarks for Dogs vs. Cats

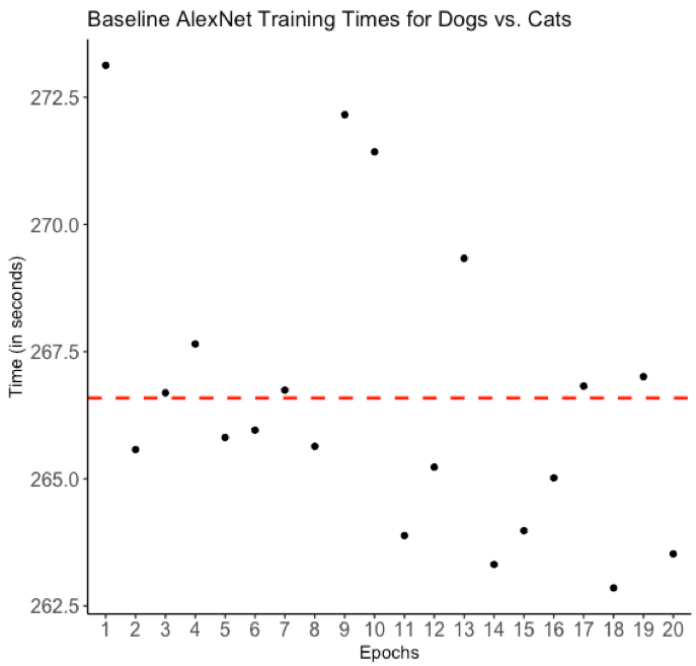

Figure 4.7: The baseline epoch times of AlexNet on the Dogs vs. Cats data set over 20 epochs. The dashed red line indicates the average time per epoch, 266 seconds. The first epoch takes 7 seconds longer than the average.

Like the Tiny ImageNet baseline, the Dogs vs. Cats baseline experiences a sharp decline in training time per epoch after the first epoch, which is explained by how Keras includes model compilation in the first epoch. According to Fig. 4.7, the average epoch time was 266 seconds. There is a trend of the later epochs taking less time, which might indicate that the default parallelization method actually increases in speed as the number of epochs increases.

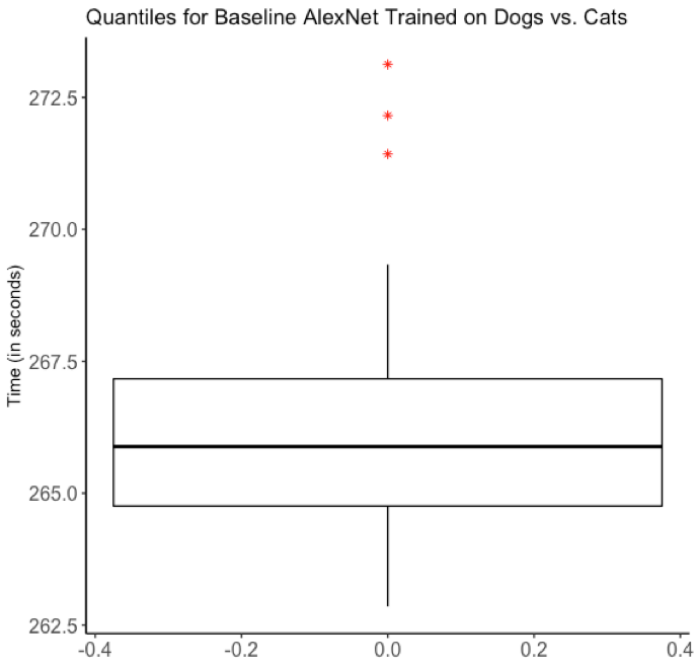

Figure 4.8: The 25th and 75 percentiles for epoch times within the baseline training benchmark for AlexNet on the Dogs vs. Cats data set is between 264 and 267 seconds, respectively, with a mean of 266 seconds. The red asterisks represent outlier training times, in this case, corresponding to epochs 1, 9, and 10.

Just as was observed in the baseline performance of AlexNet on Tiny ImageNet, the baseline performance of AlexNet on the Dogs vs. Cats dataset is tightly bounded by the first and third quartiles. The first quartile is at 264 and the third quartile is at 267 seconds. The 3 second difference between the quartiles indicates that there is not a high level of variability across epoch training times – this may be a function of the baseline training strategy employed by TensorFlow's default device management that attempts to maintain consistency between epochs. See Appendix B for the full data summary for AlexNet's baseline epoch times when trained on the Dogs vs. Cats data set.

4.2.3 Data Parallelism within AlexNet

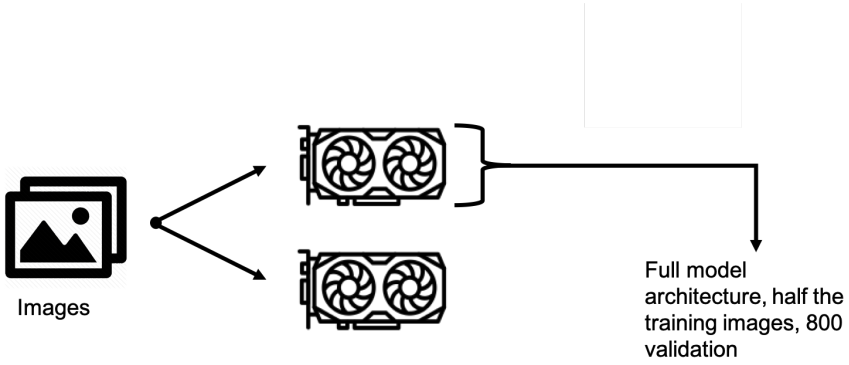

Figure 4.9: The AlexNet architecture utilized in these experiments uses a data parallel schema that replicates the full AlexNet architecture across both NVIDIA Tesla P100 GPUs. The data set is split between the 2 GPUs for training.

Fig. 4.9 represents the data parallel training schema that is applied to AlexNet over Tiny ImageNet and the Dogs vs. Cats data set. Specifically, the AlexNet architecture is replicated on both GPUs. The images are split evenly between the GPUs and training takes place across each half of the data in parallel. We tune the model on 800 validation images. This schema is analogous to the data parallelism discussed in Section 3.2.

While GPUs inherently exploit parallel computation, the data parallelism method implemented and tested in this work is specifically focused on the effects of splitting the data set across devices on epoch times. The data in data parallelism refers literally to the input data that a model is trained on. In the context of these experiments, the parallelization method is compared to the performance of the baseline TensorFlow operation allocation strategy.

Data Parallel AlexNet Trained on Tiny ImageNet

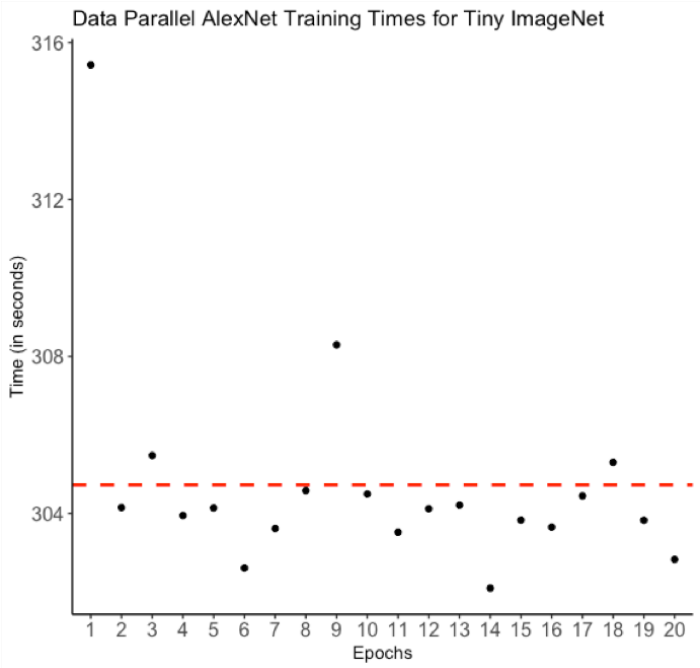

Figure 4.10: The training epochs of AlexNet trained on Tiny ImageNet using a data parallel schema have an average training time of 304 seconds. The first epoch is a significant outlier among all epochs.

Observing the performance of the data parallelism method on AlexNet when trained over Tiny ImageNet in Fig. 4.10, the average epoch time is 304 seconds. This first epoch takes 11 seconds longer to complete than the average training time, which is a significant outlier across the 20 training epochs. While the baseline also had a sharp drop in epoch times after the first epoch, the 11 second gap between the average and maximum epoch time in this case may indicate that the model architecture takes longer to compile with the implementation of the data parallel schema than the baseline configuration across the dual P100 GPUs.

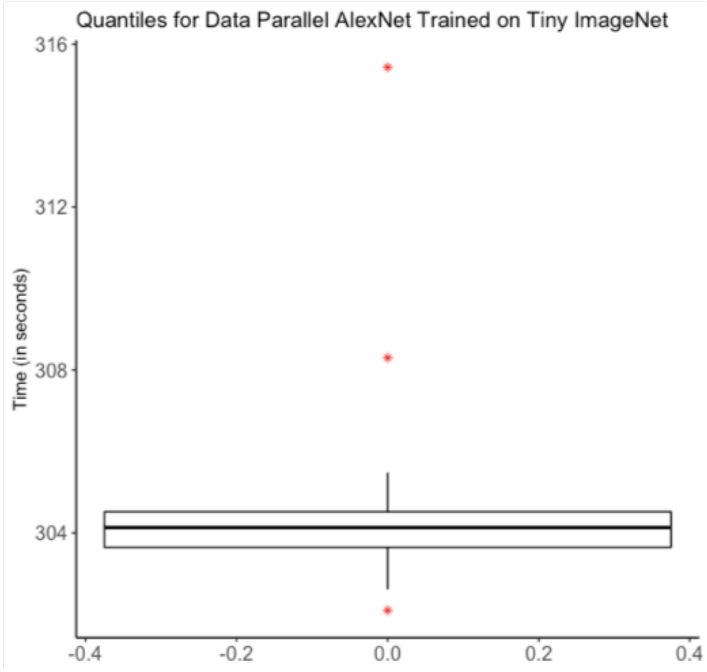

Figure 4.11: The training epochs for the data parallel AlexNet are contained primarily by the first and third quartiles, 303 and 304 seconds, respectively. The outlier points correspond to epochs 1,9 and 14.

Analyzing the quantiles of the epoch times of the data parallel AlexNet trained on Tiny ImageNet highlights a closer range between the first and third quartiles, respectiely. The difference between the quartiles is 1 second, which is smaller than the range recorded for the baseline model trained on Tiny ImageNet. Our data parallel schema exhibits more consistency between epoch training times than the baseline. Furthermore, we observe epoch 14 as an outlier point at 302 seconds, which is 2 seconds faster than the average. See Appendix A for the full data summary for the data parallel AlexNet's epoch times when trained on Tiny ImageNet.

Data Parallel AlexNet Trained on Dogs vs. Cats

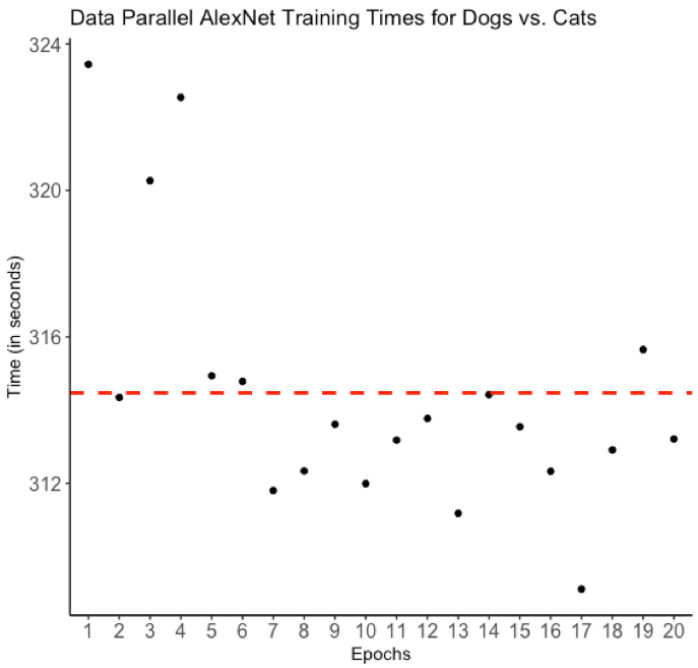

Figure 4.12: The training epochs for the data parallel AlexNet model trained on the Dogs vs Cats data set have an average training time of 314.5 seconds, indicated by the dashed red line.

In Fig. 4.12, the average training time for the data parallel AlexNet architecture trained over the Dogs vs. Cats data set is approximately 314.5 seconds. This is significantly slower than the baseline on the same data set, which had an average training time of 266 seconds. Figure 4.12 also highlights that 2 of the earlier epochs had times near the first epoch – this may be a function of the method starting off slower and then increasing in speed. There is an 8 second drop between epochs 4 and 5, which may be the result of the dual GPU's completing operations at the same time with little time wasted waiting for parameter synchronization.

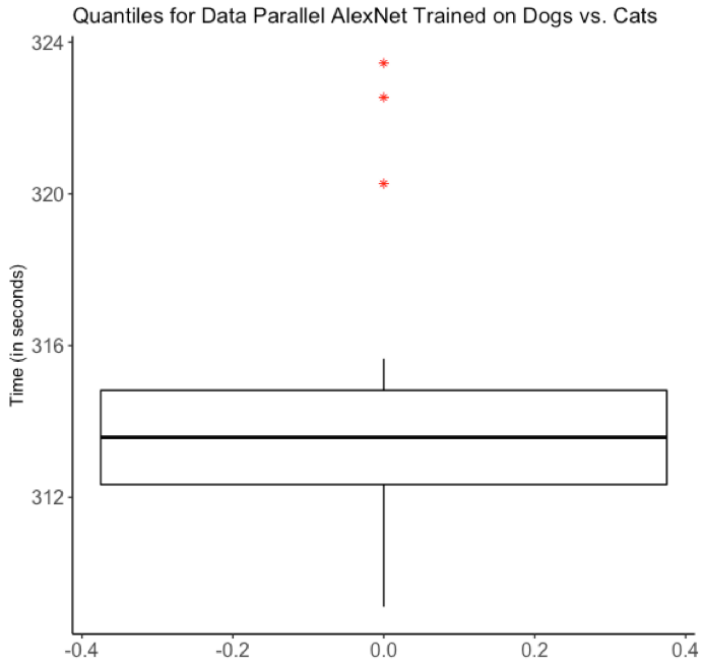

Figure 4.13: The epochs for the data parallel AlexNet trained on the Dogs vs. Cats data set are contained by the first and third quartiles, 312 and 314 seconds, respectively. The red asterisks correspond to outlier epochs 1, 3, and 4.

As seen in Fig. 4.13, the first and third quartiles for the data parallel AlexNet model trained on the Dogs vs. Cats dataset is 312 and 314 seconds, respectively. The distance between the quartiles is 2 seconds, which is a smaller range than the baseline observed in the same dataset. Thus, the data parallel schema provides a higher level of consistency between training epoch times than the baseline model when training on the Dogs vs. Cats dataset. See Appendix B for the full data summary for the data parallel AlexNet's epoch times when trained on the Dogs vs. Cats dataset.

4.2.4 Model Parallelism within AlexNet

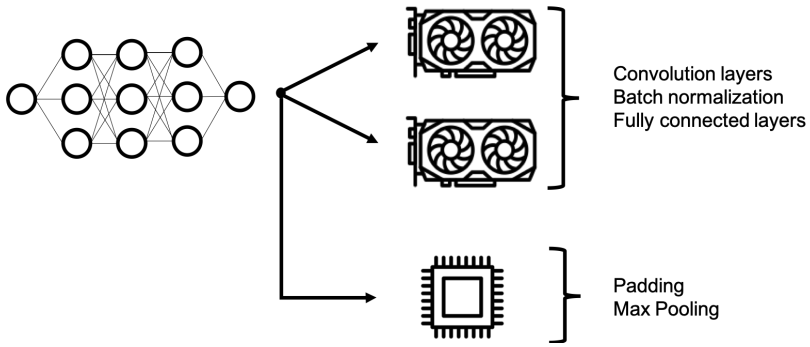

Figure 4.14: The AlexNet architecture utilized in these experiments used a model parallel schema that splits the full model architecture across the 2 NVIDIA Tesla P100 GPUs. The full data set is passed to all devices.

The model parallel schema used to parallelize AlexNet while training over both of the data sets in this work splits the AlexNet architecture over the available devices, as depicted in Fig. 4.14. In particular, the convolution, batch normalization, and fully connected layers are allocated to the GPUs, while the padding and max pooling operations are allocated to the CPU.

In this schema, the GPUs are given the more computationally intensive operations that also operate across a larger number of the model's parameters. For example, the fully connected layers in a CNN have the largest number of neurons associated with them since every neuron is connected with every other neuron in the preceding and succeeding layers, as discussed in Section 2.3. I allocated the padding and max pooling operations to the CPU since these are less computationally intensive, and computing them on the CPU would free up the GPUs to execute more computationally intensive operations.

The model parallel methods are implemented via the device level control the TensorFlow API provides. We specify which device is in control of which layers in our AlexNet implementation, built using the sequential model API provided through Keras

[42].

Model Parallel AlexNet Trained on Tiny ImageNet

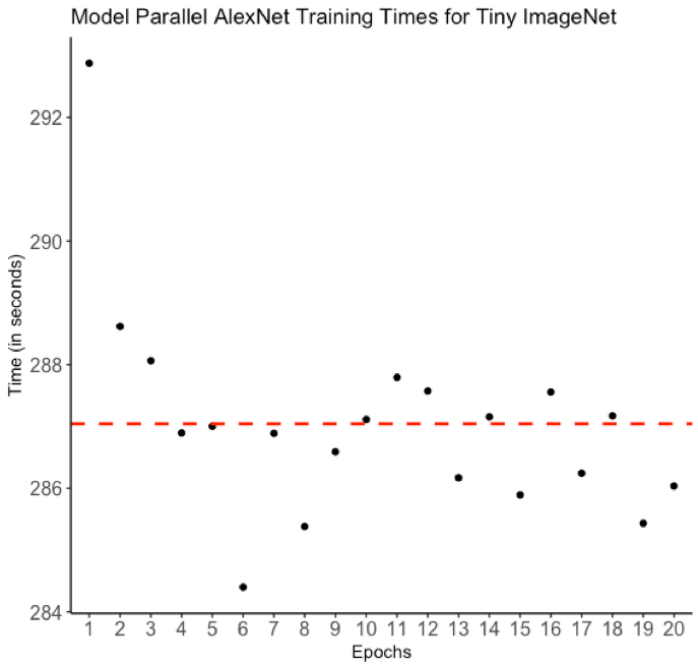

Figure 4.15: The epochs for the model parallel AlexNet implementation trained on Tiny ImageNet have an average training time of 287 seconds, indicated by the red dashed line.

As seen in Fig. 4.15, the average epoch time of the model parallelized AlexNet architecture trained on Tiny ImageNet has an average epoch time of 287 seconds. This is an improvement over the data parallel implementation trained on the same data set, which had an average epoch time of 304 seconds. The baseline still has a better average time of 281 seconds, potentially a result of image sizes or communication overhead associated with this particular parallelization method.

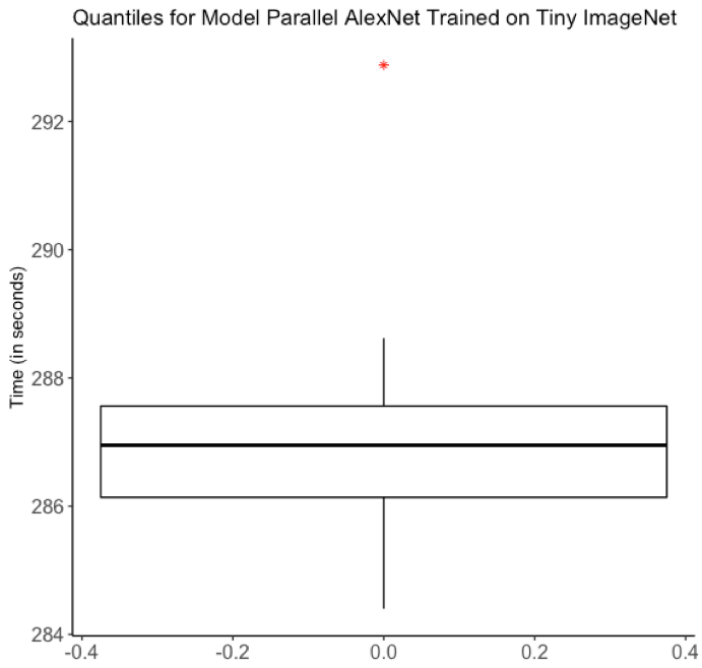

Figure 4.16: The epochs for the model parallel AlexNet implementation trained on Tiny ImageNet are contained by the first and third quartiles, 286 and 288 seconds, respectively. The red asterisk corresponds to the single outlier point – epoch 1.

The model parallel AlexNet implementation trained on Tiny ImageNet has the training time of epochs contained by the first and third quartiles, as seen in Fig. 4.18. The first quartile is at 286 seconds, while the third quartile is at 288 seconds. The 2 second difference between the quartiles is a smaller range than the difference between the first and third quartiles of the baseline implementation trained on the same dataset (3 seconds). The data parallel version had a range of 1 second between the first and third quartiles, indicating that the data parallel model had less variability across training times. The data parallel schema has less variability between quartiles since it requires less communication between GPUs during forward and backward passes of the training algorithm. The data parallel schema does not require communication to compute gradients, it only aggregates results at the end of an epoch. See Appendix A for the full data summary for the data parallel AlexNet's epoch times when trained on Tiny ImageNet.

Model Parallel AlexNet Trained on Dogs vs. Cats

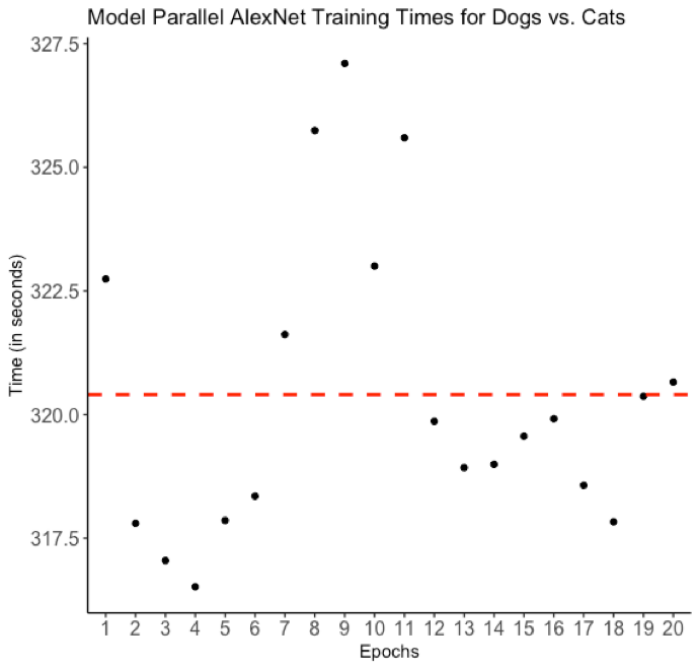

Figure 4.17: The epochs for the model parallel AlexNet implementation trained on the Dogs vs Cats data set have an average training time of 320 seconds, indicated by dashed red line.

In Fig. 4.17, the average training time for the data parallel AlexNet architecture trained over the Dogs vs. Cats dataset is approximately 320 seconds. This is significantly slower than the baseline implementation, which had an average training time of 266 seconds. Specifically, the model parallel implementation is 54 seconds slower than the average of the baseline model, and 6 seconds slower than the average of the data parallel implementation, both trained on the same data set.

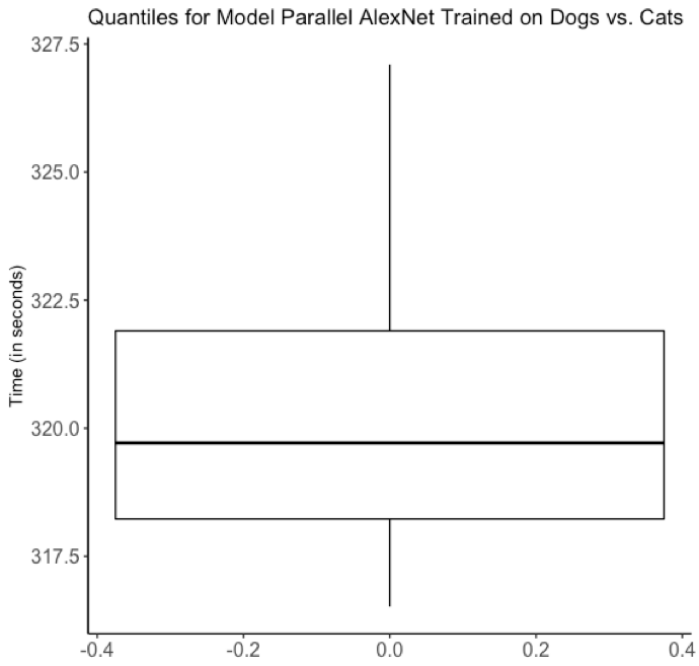

Figure 4.18: The majority of epoch times for the model parallel AlexNet trained on the Dogs vs. Cats data set are contained by the first and third quartiles, 318 and 322 seconds, respectively.

As seen in Fig. 4.18, 50% of epoch times of the AlexNet architecture trained on the Dogs vs. Cats dataset are bounded via the first and third quartiles, 318 and 322 seconds respectively. The difference between the quartiles is 4 seconds, which is greater than the difference observed between the first and third quartiles of the data parallel implementation (3 seconds) and the first and third quartiles of the baseline implementation (2 seconds), both trained on the same data set. Model parallelism has a wider range between the first and third quartiles as a result of the communication necessary between GPUs during forward and backward passes to compute gradients via backpropagation. See Appendix B for the full data summary for the model parallel AlexNet's epoch times when trained on the Dogs vs. Cats data set.

4.2.5 Expert Designed Methods

The expert designed method used in this work is the One Weird Trick (OWT) method proposed by Alex Krizhevsky [23]. Unlike the other methods explored in this work, OWT can only be applied to the AlexNet architecture.

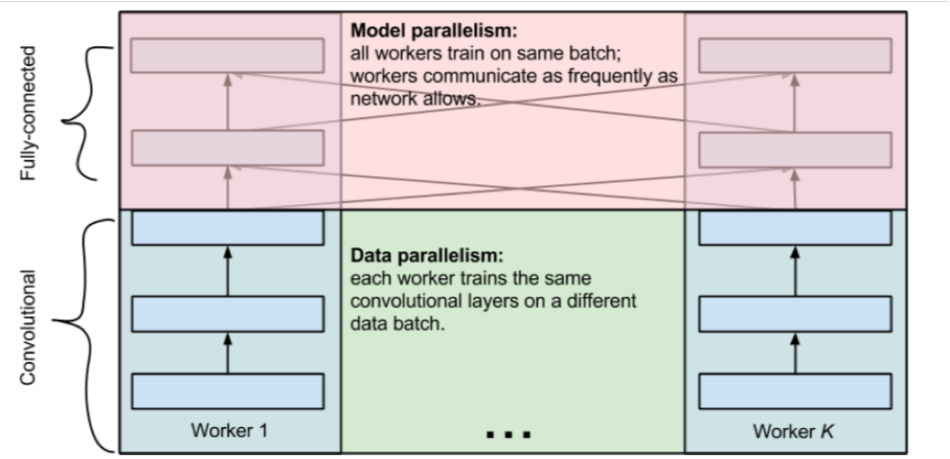

Figure 4.19: The One Weird Trick (OWT) method for parallelizing AlexNet is a hybrid method that combines model and data parallelism.[23]

The OWT method is a hybrid method in that it uses model parallelism for the fully connected layers, and data parallelism for the convolutional layers, as seen in Fig. 4.19. This method is based on the observation that convolutional layers account for 90-95% of the total model computation, approximately 5% of the total model parameters, and require large representations [23]. The high level of computation and large representations make data parallelism particularly effective at accelerating these operations. At the same time, the fully connected layers account for approximately 5-10% of the total model computation, 95% of the model's parameters, and have smaller representations. The lack of computation and higher responsibility for the model's parameters make model parallelism an effective avenue of acceleration for these layers. When the OWT method was applied to the AlexNet architecture while training on the original ImageNet dataset, there was a 6.25x speedup when training across 8 GPUs with a batch size of

1,024 [23].

I applied the OWT method to the AlexNet architecture while training across Tiny ImageNet and the Dogs vs. Cats dataset. The OWT method for training AlexNet across these data sets was implemented using Keras for sequential model definition [42], and TensorFlow to allocate particular layer-wise operations to specific devices.

The model parallel methods are implemented via the device level control the TensorFlow API provides. I specify which device is in control of which layers in our AlexNet implementation, built using the sequential model API provided through Keras [42].

OWT AlexNet Trained on Tiny ImageNet

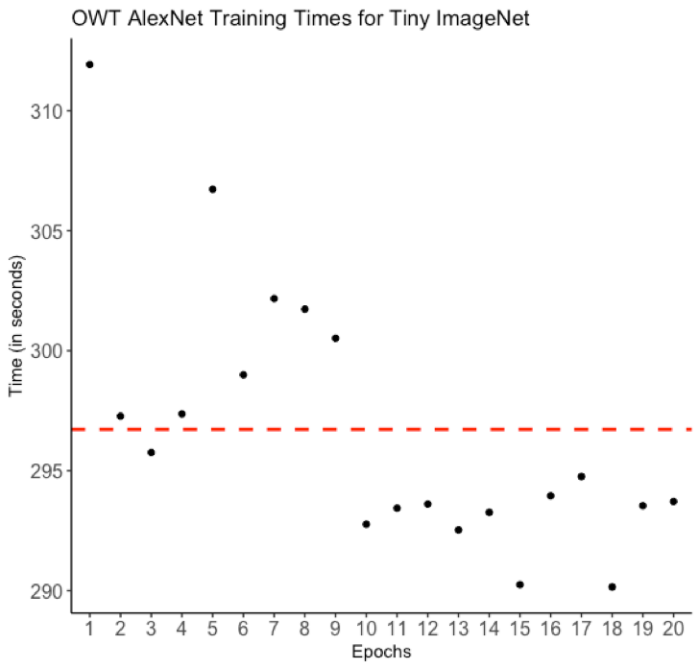

Figure 4.20: The epochs for the OWT AlexNet implementation trained on Tiny ImageNet have an average training time of 297 seconds, indicated by the dashed red line.

As seen in Fig. 4.20, the average epoch time of the OWT AlexNet architecture trained on Tiny ImageNet has an average epoch time of 297 seconds. This is an improvement over the data parallel implementation trained on the same data set, which had an average

epoch time of 304 seconds. The model parallel and baseline implementations still have a better average time of 281 seconds and 287 seconds, respectively.

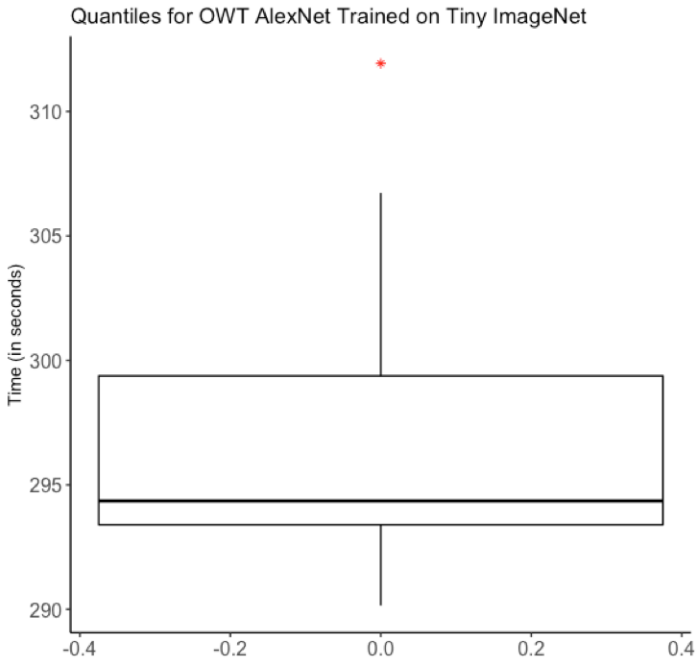

Figure 4.21: The epochs for the OWT AlexNet implementation trained on Tiny ImageNet have first and third quartiles at 293 and 299 seconds, respectively. The red asterisk corresponds to the single outlier point – epoch 1.

According to Fig. 4.21, 50% of the epoch times are bounded between 293 and 299 seconds. This 6 second difference between the first and third quartiles is the largest difference between quartiles observed across all the parallelization and baseline implementations trained on the same dataset. The difference between the first and third quartiles for the OWT implementation trained on Tiny ImageNet is double the difference in the baseline implementation (3 seconds), which indicates this method has additional variability between the quartiles of the epochs.

OWT AlexNet Trained on Dogs vs. Cats

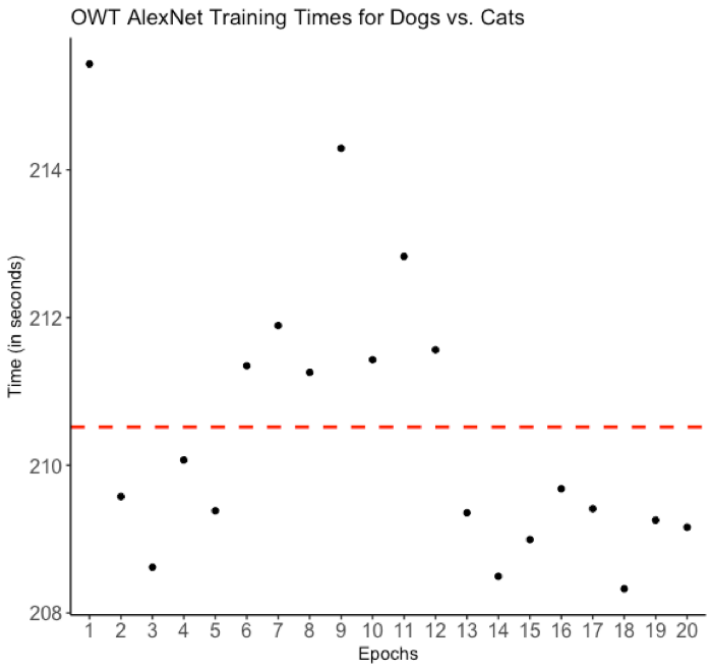

Figure 4.22: The epochs for the OWT AlexNet implementation trained on the Dogs vs. Cats data set have an average training time of 211 seconds, indicated by the dashed red line.

As seen in Fig. 4.22, the average epoch time of the OWT AlexNet architecture trained on the Dogs vs. Cats dataset has an average epoch time of 211 seconds. This is an improvement over all parallelization and baseline methods trained on the same dataset. Once again, the trend of the first epoch taking the longest can be observed, along with later epochs tending to be faster than earlier and middle epochs.

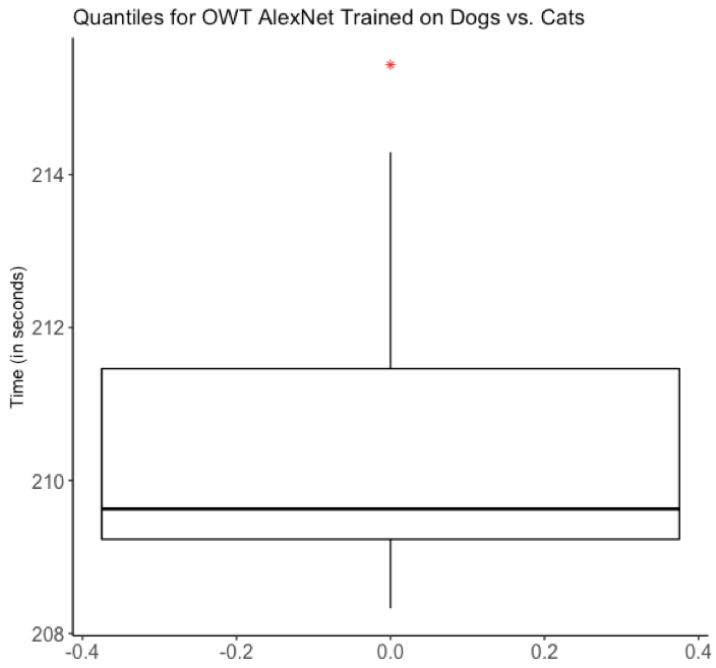

Figure 4.23: The epochs for the OWT AlexNet implementation trained on the Dogs vs. Cats data set have first and third quantiles at 209 and 212 seconds, respectively. The red asterisk corresponds to the single outlier point – epoch 1.

According to Fig. 4.23, 50% of the epoch times are bounded between 209 and 212 seconds. This 3 second difference between the first and third quartiles is comparable to performance of the data parallel implementation trained on the same dataset. The range between the first and third quartiles implies that this method has variability between epoch times. See Appendix B for the full data summary for the OWT AlexNet's epoch times when trained on the Dogs vs. Cats data set.

4.2.6 FlexFlow Parallelization Framework

The FlexFlow parallelization framework, discussed in Section 3.5, produces an optimized parallelized training strategy by exploring samples, operations, attributes, and parameters, as shown in Fig. 3.6. While this work has explored data parallelism, model parallelism, and OWT, FlexFlow explores a wider range of parallelization options than

these methods.

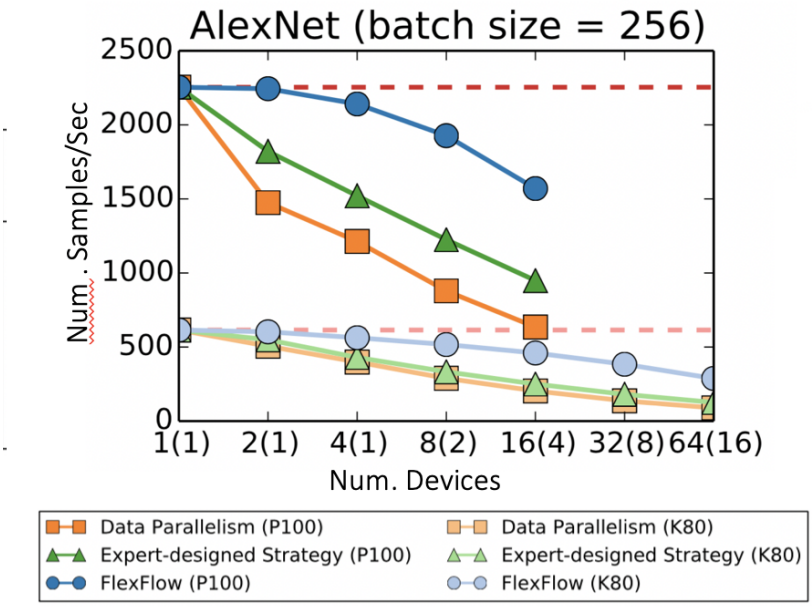

Figure 4.24: Comparing the number of samples processed per second across data parallelism, an expert designed strategy (OWT), and FlexFlow. The darker lines correspond to experiments run on NVIDIA Tesla P100 GPUs, while the lighter lines represent experiments run on NVIDIA Tesla K80 GPUs [19].

FlexFlow was tested on the AlexNet architecture while training on the full ImageNet dataset. According to Fig. 4.24, FlexFlow outperforms both data parallelism as well as the expert designed strategy (OWT) in terms of the number of samples processed per second across devices [19] – a higher number of samples processed per second indicates faster epoch times, which is the metric observed in this work. While there seems to be a slight dropoff in the number of samples processed per second across parallelization methods as the number of devices and compute nodes increases, a similar dropoff was observed in attempting to add more devices to accelerate AlexNet via OWT when it was first proposed [23]. The dropoff in the number of samples processed per second is is much steeper for the expert designed and data parallelism method, while FlexFlow exhbits more consistency before dropping off. Furthermore, the use of NVIDIA Tesla

P100 GPUs that exhibited stronger performance in the FlexFlow paper influenced this work's decision to use the same GPUs.

FlexFlow is an improvement over the various methods we have explored thus far given the fact that it exhibits stronger performance on the AlexNet benchmark when trained on a data set over 100 times larger than the data sets used in this work [19, 1]. In addition to exhibiting stronger accelerated performance, FlexFlow can also be applied to various DNN architectures, outside of AlexNet.

4.3 Analysis

In exploring multiclass classification and binary classification through Tiny ImageNet and the Dogs vs. Cats data set, respectively, this work has explored parallelization methods aimed at accelerating the training of DNNs. While Tiny ImageNet and the Dogs vs. Cats data set are different in the types of images they have and the number of classes they represent, the application of AlexNet to these data sets as an image classification task has yielded an overview of the performance of various parallelization methods.

4.3.1 AlexNet and Tiny ImageNet Analysis

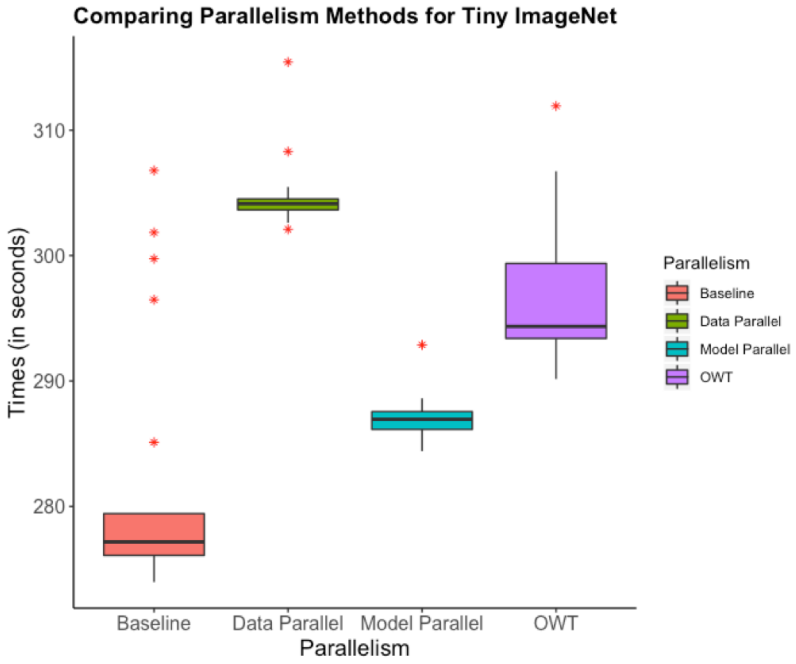

Figure 4.25: Comparing the quantiles for each parallelism method for the AlexNet architecture trained on Tiny ImageNet. Red asteriks represent outlying epochs.

When AlexNet is trained on Tiny ImageNet, the order of methods that produce the most accelerated training performance is the baseline method, followed by model parallelism, OWT, and data parallelism. It is apparent from Fig. 4.25 that the baseline method had the most outliers among epoch times. Model and data parallelism have smaller ranges between their first and third quartiles – this is potentially the result of enforced consistency in how operations on data are split. In the case of data parallelism, both GPUs always had a full replica of the AlexNet architecture and ran the full model on half of the dataset, while model parallelism always allocated the same layers to the same devices while running the data through every device.

The OWT method has a larger distance between the first and third quartiles (6 seconds), which could be a function of switching between data and model parallelism for various layers in the model's architecture. The communication overhead associated with

swapping which operations are allocated to each GPU likely contributed to this variability across epoch times for the OWT method. For example, switching from data parallelism to model parallelism when transitioning from convolutional to fully connected layers requires replacing the full model architectures replicated across each device with particular operations.

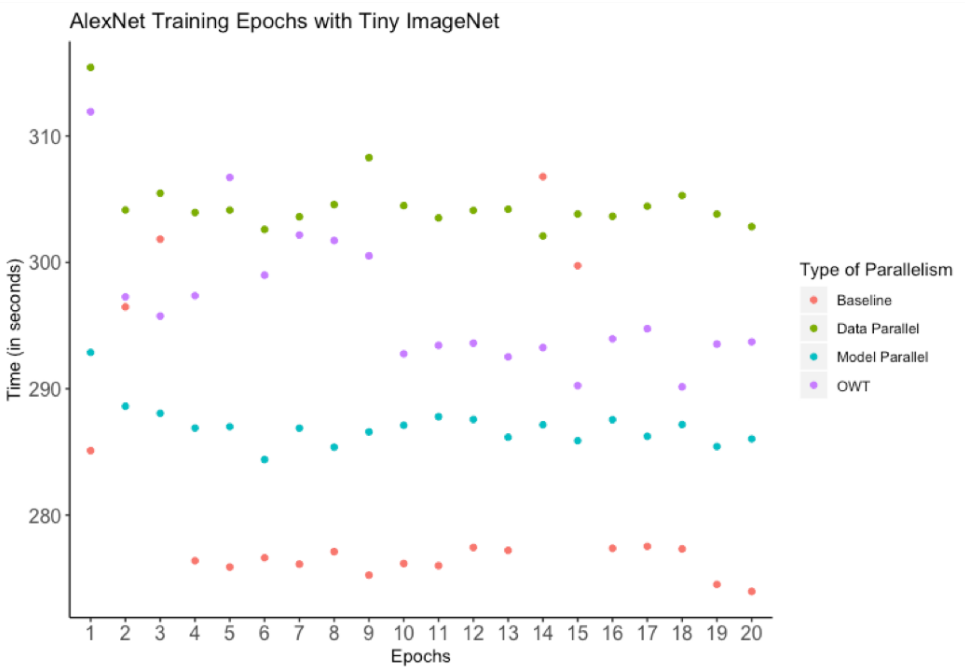

Figure 4.26: Observing all epoch times for each parallelization method used on AlexNet trained on Tiny ImageNet.

Observing all training epoch times for AlexNet implemented using each parallelization method in Fig. 4.26, the baseline method outperforms the other parallelization methods. It can also be observed that the first epoch in each method took the longest, illustrated by a dip between the first and second epochs across all methods – this phenomena was discussed in Section 4.2.2.

I hypothesize that the baseline implementation outperforms all the parallelization methods for AlexNet trained on Tiny ImageNet because the image sizes are small enough for the baseline method to be the most efficient route. Parallelization methods have a

certain level of overhead associated with their implementation. My implementation of data parallelism needs to aggregate results after running the full model on subsets of the data, which induces additional overhead in training. Similarly, the model parallel method implemented in this work requires GPUs to communicate while gradients are computed between forward and backward passes. OWT also has communication overhead associated with its implementation, but was designed to work specifically with the AlexNet architecture. The baseline method discussed in Section 4.2.2 is potentially more effective in the case of Tiny ImageNet because of the 56 x 56 x 3 image sizes. The tradeoffs associated with using a parallelization method such as data parallelism, model parallelism, or OWT do not yield better performance with images of this size. The image sizes also yield a smaller feature space with less parameters to calculate (the number of parameters is directly tied to the size of the image, as discussed in Section 2.3). Thus, the baseline method is able to effectively map the entire feature space to the dual GPUs in a way that is more efficient than the rigid methods explored in this work. Also, the original AlexNet architecture took input images that were 227 x 227 x 3, which might imply that with larger image sizes or greater batch sizes, the overhead associated with using a parallelism method could yield stronger performance than the baseline default implementation.

4.3.2 AlexNet and Dogs vs. Cats Analysis

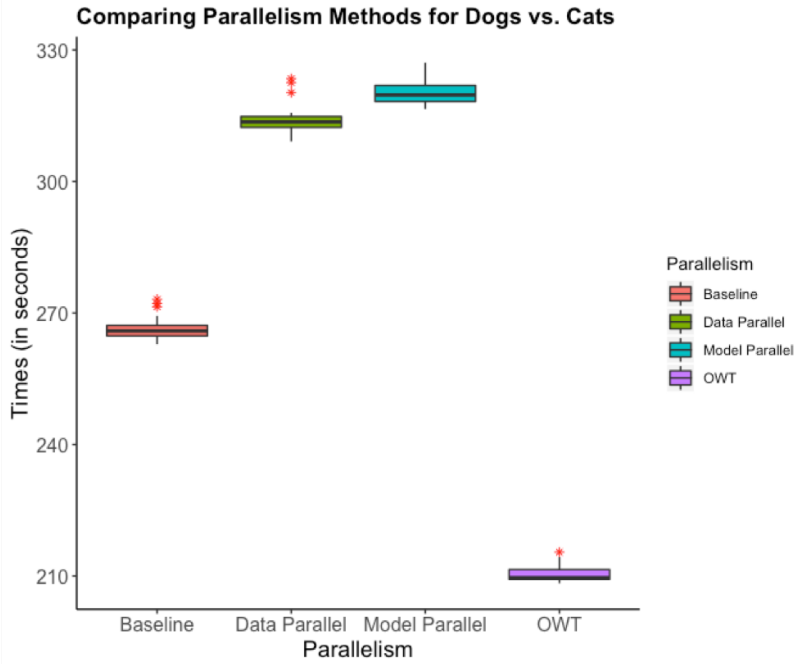

Figure 4.27: Comparing the quantiles for each parallelism method for the AlexNet architecture trained on the Dogs vs. Cats dataset. Red asteriks represent outliers.

As depicted in Fig. 4.27, the OWT method outperforms all other parallelization methods when AlexNet was trained on the Dogs vs. Cats data set. OWT's performance was followed by the baseline method, data parallelism, with model parallelism performing the worst. All of the individual boxplots in Fig. 4.27 are plotted along the same scale which produces qualitatively smaller boxplots since the OWT method is much lower on the scale the other methods. The average OWT epoch time was 56 seconds faster than the average epoch time for the baseline method, which is approximately a 1.2x speedup.

In Section 4.2.5, I discussed OWT's performance across Tiny ImageNet and Dogs vs. Cats. In both cases, there was a larger distance between the first and third quartiles than in other parallelization methods I observed. Once again, this variability is potentially explained by the overhead associated with switching between model and data parallelism for different layers in the AlexNet's architecture.

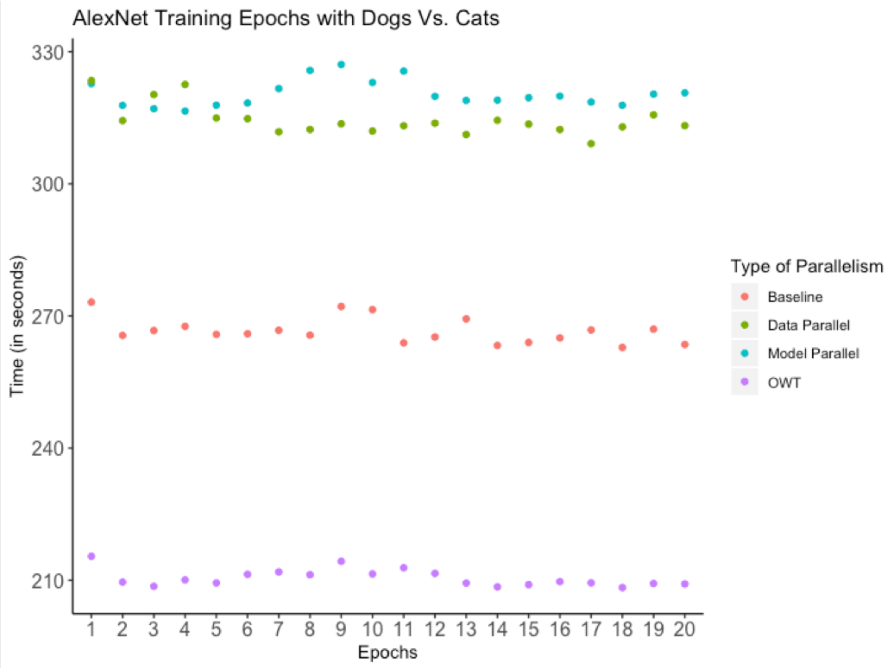

Figure 4.28: Observing all epoch times for each parallelization method used on AlexNet trained on the Dogs vs. Cats dsta set.

The epoch times across all 20 epochs for each parallelization method for AlexNet trained on the Dogs vs. Cats data set is given in Fig. 4.28. There is a much clearer distinction between methods in this figure as opposed to Fig. 4.26 – this is a result of widely varying performance across each method. While OWT is 1.2x faster than the baseline, the baseline's average epoch time (266 seconds) is itself approximately 1.2x faster than the average between the model and data parallel average epoch times (317 seconds).

The OWT method sees a 1.2x speedup when training AlexNet on the Dogs vs. Cats data set because the image sizes are large. In Section 4.2.2, I described that the input images for the AlexNet implementation trained on the Dogs vs. Cats data set would have the dimensions 128 x 128 x 3. Since these images are larger than the images used in the Tiny ImageNet AlexNet experiments (56 x 56 x 3), the overhead associated with using a parallelization method outperforms the default baseline operation allocation.

Model and data parallelism perform worse than the baseline in this case, which tells us that the application of these methods in isolation is less effective than hybrid expert designed methods like OWT. Thus, effective parallelization and device optimization requires hybrid methods and an input size that makes the communication overhead of parallelization methods worth it [19, 20, 23].

CHAPTER 5

CONCLUSIONS

In summation, the experiments in this work have attempted to show that deep neural networks can be accelerated through parallelism. In particular, my exploration of data parallelism, model parallelism, expert designed methods (OWT), and generalizable frameworks (FlexFlow) through the AlexNet CNN architecture has provided valuable insight into the application of parallelism across image classification tasks of varying sizes.

I have observed that:

- the isolated application of data and model parallelism does not yield accelerated training performance on smaller tasks due to the communication overhead between devices.
- The baseline implementation of operation allocation over device topologies provided by the TensorFlow backend has the fastest training times for tasks that deal with smaller images and batch sizes.
- The expert designed method (OWT) exhibited up to a 1.2x speedup over the baseline implementation when dealing with larger images of size 128 x 128 x 3. Thus, larger images and batch sizes increase the need and effectiveness of parallelization methods.

While OWT exhibited strong performance on the Dogs vs. Cats data set, expert designed methods like OWT are architecture specific and cannot be applied to other deep learning methods, failing to scale to other applications. Furthermore, these methods are inherently inaccessible as they require a rigorous understanding of a model's underlying architecture.

On the other hand, frameworks like FlexFlow are architecture agnostic and can be applied to a wide variety of deep learning methods while guaranteeing accelerated train-

ing performance. My results indicate that data parallelism, model parallelism, and OWT may not be sufficient to produce optimized device usage through operation allocation for all data sets, image and batch sizes – thus, generalizable frameworks like FlexFlow are necessary in order to produce an optimized accelerated training strategy for all model architectures, over varying device topologies. FlexFlow increases the efficiency of device usage in a way that accelerates model training. Optimizing device usage makes deep learning methods more accessible across systems with varying computational resources. Frameworks like FlexFlow are effective in that they treat parallelism as an abstraction.

While machine learning APIs like Keras and TensorFlow increase the accessibility of machine learning by providing a simple-to-use interface for building deep learning models, parallelism still needs to be implemented manually when working with these APIs. In order to make deep learning methods more accessible, APIs should implement a more flexible baseline operation allocation strategy that adapts to a given model's input and batch sizes – in this way, users can be assured that their models are always optimizing the use of computational resources they have available.

While the results in this work have shown that the baseline methods performs best with smaller image and batch sizes, future work can potentially highlight the inefficiencies of the baseline operation allocation strategy across larger image and batch sizes. Increasing accessibility also increases reproducibility, which would only benefit deep learning research as a greater number experiments can be analyzed more rigorously. Finally, device usage optimization ensures that accelerated performance can scale with an increase in computational resources – thus, more powerful deep learning models can be trained more efficiently if parallelization becomes a normalized component in model development.

5.1 Future Work

This work has compared the performance of data parallelism, model parallelism, the OWT expert designed method and FlexFlow through the AlexNet architecture trained on Tiny ImageNet and the Dogs vs. Cats dataset. While the baseline operation allocation strategy implemented by TensorFlow yielded the fastest epoch times over the Tiny ImageNet data set, future work should focus on various image sizes to see the comparative effect on training time that varying image sizes have across different parallelization methods.

In this work I hypothesize that image size is a key determinant in whether or not the baseline strategy implemented by TensorFlow is able to effectively utilize available resources – in addition to testing these methods on varying image sizes, future work should also see the performance of these methods across a single GPU, as well as across a higher number of GPUs. Testing these methods across a single GPU system would highlight what efforts can be made in order to accelerate DNNs in the context of resource constrained systems.

Increasing the efficiency of parallelization methods as a means of accelerating the training of DNNs requires that device optimization scales with the number of GPUs a particular system has available. In this work we explored the performance of parallelization methods across a dual GPU system, but comparing these methods on the same model architecture and same data sets across a larger set of GPUs would highlight whether or not data parallelism, model parallelism, and expert designed methods such as OWT are able to yield a speedup over the baseline TensorFlow strategy.

In addition to scaling the experiments described in this work across different device topologies, future work can also focus on the the ways in which parallelization methods impact accuracies. This work focused on a training over a set number of epochs across the various parallelization methods, keeping accuracy as a constant – it would also be

interesting to see whether or not parallelization methods yield an increase or decrease in predictive performance. Exercising flexibility in the number of epochs a model is trained on, and instead allowing early stopping if a desired accuracy is achieved could itself achieve a speedup in training time.

This work was focused on AlexNet and image classification, but increasing the accessibility of deep learning methods as a whole requires parallelization benchmarks across various tasks, such as natural language processing [54], speech recognition [13], and game playing [15]. Exploring the performance of parallelization methods across various tasks would yield a holistic understanding of the steps that can be taken in order to increase the accessibility of all deep learning methods.

Finally, testing parallelization methods across a larger data set with more images, such as ImageNet [1], would highlight the performance of parallelization methods on a data set used to train models that define the state-of-the-art in image classification.

APPENDIX A

DATA SUMMARY FOR TINY IMAGENET BENCHMARKS

	Baseline	Data Parallelism	Model Parallelism	OWT
Epochs	**Times (in seconds)**			
1	285.1	315.4	292.9	311.9
2	296.5	304.2	288.6	297.3
3	301.8	305.5	288.1	295.8
4	276.4	303.9	286.9	297.4
5	275.9	304.1	287.0	306.8
6	276.6	302.6	284.4	299.0
7	276.1	303.6	286.9	302.2
8	277.1	304.6	285.4	301.7
9	275.3	308.3	286.6	300.5
10	276.2	304.5	287.1	292.8
11	276.0	303.5	287.8	293.4
12	277.4	304.1	287.6	293.6
13	277.2	304.2	286.2	292.5
14	306.8	302.1	287.2	293.3
15	299.7	303.8	285.9	290.2
16	277.4	303.7	287.6	294.0
17	277.5	303.4	286.2	294.8
18	277.3	305.3	287.2	290.2
19	274.5	303.8	285.4	293.5
20	274.0	302.8	286.0	293.7

Table A.1: Summary of AlexNet training times over Tiny ImageNet using the baseline, data parallelism, model parallelism, and OWT.

	Baseline	Data Parallelism	Model Parallelism	OWT
Minimum	274.0	302.1	284.4	290.2
1st Quartile	276.1	303.6	286.1	293.4
Median	277.2	304.1	286.9	294.4
Mean	281.7	304.7	287.0	296.7
3rd Quartile	279.3	304.5	287.6	299.4
Maximum	306.8	315.4	292.9	311.9

Table A.2: AlexNet epoch time summary by parallelization method for Tiny ImageNet training.

APPENDIX B

DATA SUMMARY FOR DOGS VS. CATS BENCHMARKS

	Baseline	Data Parallelism	Model Parallelism	OWT
Epochs	**Times (in seconds)**			
1	273.1	323.4	322.7	215.4
2	265.6	314.3	317.8	209.6
3	266.7	320.3	317.0	208.6
4	267.6	322.5	316.5	210.1
5	265.8	314.9	317.9	209.4
6	266.0	314.8	318.3	211.3
7	266.7	311.8	321.6	211.9
8	265.6	312.3	325.7	211.3
9	272.2	313.6	327.1	214.3
10	271.4	312.0	323.0	211.4
11	263.9	313.2	325.6	212.8
12	265.2	313.8	319.9	211.6
13	269.3	311.2	318.9	209.4
14	263.3	314.4	320.0	208.5
15	264.0	313.5	319.6	209.0
16	265.0	312.3	319.9	209.7
17	266.8	309.1	318.6	209.4
18	262.9	312.9	317.8	208.3
19	274.5	315.7	320.4	209.3
20	263.5	313.2	320.7	209.2

Table B.1: Summary of AlexNet training times over the Dogs vs. Cats data set using the baseline, data parallelism, model parallelism, and OWT.

	Baseline	Data Parallelism	Model Parallelism	OWT
Minimum	262.9	309.1	316.5	208.3
1st Quartile	264.8	312.3	318.2	209.2
Median	265.9	313.6	319.7	209.6
Mean	266.6	314.5	320.4	210.5
3rd Quartile	267.2	314.8	321.9	211.5
Maximum	273.1	323.4	327.1	215.4

Table B.2: AlexNet epoch time summary by parallelization method while training on the Dogs vs. Cat data set.

BIBLIOGRAPHY

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[2] T. Bertin-Mahieux, D. P. W. Ellis, B. Whitman, and P. Lamere, "The Million Song Dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[3] S. Goswami, S. Chakraborty, S. Ghosh, A. Chakrabarti, and B. Chakraborty, "A review on application of data mining techniques to combat natural disasters," *Ain Shams Engineering Journal*, vol. 9, no. 3, pp. 365–378, 2018.

[4] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, "Exascale deep learning for climate analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 51:1–51:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291724

[5] T. Ichimura, K. Fujita, T. Yamaguchi, A. Naruse, J. C. Wells, T. C. Schulthess, T. P. Straatsma, C. J. Zimmer, M. Martinasso, K. Nakajima, M. Hori, and L. Maddegedara, "A fast scalable implicit solver for nonlinear time-evolution earthquake city problem on low-ordered unstructured finite elements with artificial intelligence and transprecision computing," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2018. [Online]. Available: https://doi.org/10.1109/sc.2018.00052

[6] K. Rasouli, W. W. Hsieh, and A. J. Cannon, "Daily streamflow forecasting by machine learning methods with weather and climate inputs," *Journal of Hydrology*, vol. 414-415, pp. 284–293, Jan. 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022169411007633

[7] J. Ghosn and Y. Bengio, "Multi-Task Learning for Stock Selection," in *Advances in Neural Information Processing Systems 9*, M. C. Mozer, M. I. Jordan, and T. Petsche, Eds. MIT Press, 1997, pp. 946–952. [Online]. Available: http://papers.nips.cc/paper/1221-multi-task-learning-for-stock-selection.pdf

[8] C. A. Gomez-Uribe and N. Hunt, "The Netflix Recommender System: Algorithms, Business Value, and Innovation," *ACM Trans. Manage. Inf. Syst.*, vol. 6, no. 4, pp. 13:1–13:19, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2843948

[9] D. M. Diez, C. D. Barr, and M. Cetinkaya-Rundel, *OpenIntro: Data Sets and Supplemental Functions from 'OpenIntro' Textbooks*, 2017, R package version 1.7.1. [Online]. Available: https://CRAN.R-project.org/package=openintro

[10] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[11] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should i trust you?: Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 2016, pp. 1135–1144.

[12] Y. Combinator, "Jeff Deans Lecture for YC AI." [Online]. Available: https://blog.ycombinator.com/jeff-deans-lecture-for-yc-ai/

[13] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1764–1772. [Online]. Available: http://proceedings.mlr.press/v32/graves14.html

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3098997.3065386

[15] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016. [Online]. Available: http://www.nature.com/articles/nature16961

[16] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics," *Journal of Big Data*, vol. 2, no. 1, p. 1, Feb. 2015. [Online]. Available: https://doi.org/10.1186/s40537-014-0007-7

[17] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: http://www.nature.com/articles/nature14539

[18] A. Ng, "Machine learning." [Online]. Available: https://www.coursera.org/learn/machine-learning

[19] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2018.

[20] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling." in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2018.

[21] "DeepMind." [Online]. Available: https://deepmind.com/

[22] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. v. d. Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, Oct. 2017. [Online]. Available: https://www.nature.com/articles/nature24270

[23] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv:1404.5997 [cs]*, Apr. 2014, arXiv: 1404.5997. [Online]. Available: http://arxiv.org/abs/1404.5997

[24] "Tiny ImageNet Visual Recognition Challenge." [Online]. Available: https://tiny-imagenet.herokuapp.com/

[25] "Kaggle: Your Home for Data Science." [Online]. Available: https://www.kaggle.com/

[26] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: http://cs231n.github.io/convolutional-networks/conv

[27] L. Valiant, *Probably Approximately Correct: Nature's Algorithms for Learning and Prospering in a Complex World*. New York, NY, USA: Basic Books, Inc., 2013.

[28] "Understanding Logistic Regression in Python," Sep. 2018. [Online]. Available: https://www.datacamp.com/community/tutorials/understanding-logistic-regression-python

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[30] A. Cauchy, "Methode generale pour la resolution des systemes d'equations simultanees," *C.R. Acad. Sci. Paris*, vol. 25, pp. 536–538, 1847. [Online]. Available: https://ci.nii.ac.jp/naid/10026863174/en/

[31] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231. [Online]. Available: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[32] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, Apr. 1997. [Online]. Available: http://ieeexplore.ieee.org/document/585893/

[33] D. Shiffman, *The Nature of Code: Simulating Natural Systems with Processing*. The Nature of Code, 2012. [Online]. Available: http://natureofcode.com/book/chapter-10-neural-networks/

[34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

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

[36] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation Functions: Comparison of trends in Practice and Research for Deep Learning," *arXiv:1811.03378 [cs]*, Nov. 2018, arXiv: 1811.03378. [Online]. Available: http://arxiv.org/abs/1811.03378

[37] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994. [Online]. Available: http://dx.doi.org/10.1109/72.279181

[38] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, p. 533, Oct. 1986. [Online]. Available: https://www.nature.com/articles/323533a0

[39] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overtting," p. 30.

[40] I. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet, "Multi-digit number recognition from street view imagery using deep convolutional neural networks," in *ICLR2014*, 2014.

[41] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv:1603.07285 [cs, stat]*, Mar. 2016, arXiv: 1603.07285. [Online]. Available: http://arxiv.org/abs/1603.07285

[42] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[43] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[44] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *arXiv:1802.09941 [cs]*, Feb. 2018, arXiv: 1802.09941. [Online]. Available: http://arxiv.org/abs/1802.09941

[45] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2933–2941. [Online]. Available: http://papers.nips.cc/paper/5486-identifying-and-attacking-the-saddle-point-problem-in- high-dimensional-non-convex-optimization.pdf

[46] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Mathematical Programming*, vol. 45, no. 1-3, pp. 503–528, Aug. 1989. [Online]. Available: https://link.springer.com/article/10.1007/BF01589116

[47] P. Goyal, P. Dollr, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," *arXiv:1706.02677 [cs]*, Jun. 2017, arXiv: 1706.02677. [Online]. Available: http://arxiv.org/abs/1706.02677

[48] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv:1409.1556 [cs]*, Sep. 2014, arXiv: 1409.1556. [Online]. Available: http://arxiv.org/abs/1409.1556

[49] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimiz-

ing DNN Computation with Relaxed, Graph Substitutions," in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[50] "Dogs vs. Cats." [Online]. Available: https://kaggle.com/c/dogs-vs-cats

[51] "Cloud Computing Services." [Online]. Available: https://cloud.google.com/

[52] "Using GPUs | TensorFlow Core." [Online]. Available: https://www.tensorflow.org/guide/using_gpu

[53] "NVIDIA cuDNN," Sep. 2014. [Online]. Available: https://developer.nvidia.com/cudnn

[54] C. dos Santos and M. Gatti, "Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts," in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*. Dublin, Ireland: Dublin City University and Association for Computational Linguistics, Aug. 2014, pp. 69–78. [Online]. Available: https://www.aclweb.org/anthology/C14-1008