Blending Mode Data Augmentation With CNN Classification

by

Michael Curry

In Partial Fulfillment of the Requirements for the

Degree of

Master of Science

in

The Department of Computer Science

State University of New York

New Paltz, New York 12561

November 2, 2022

Abstract

Convolutional neural networks (CNNs) have gained global recognition in advancing the field of artificial intelligence and have had great successes in a wide array of applications including computer vision, speech and natural language processing. However, due to the rise of big data and increased complexity of tasks, the efficiency of training CNNs have been severely impacted. To achieve state-of-art results, CNNs require tens to hundreds of millions of parameters that need to be fine-tuned, resulting in extensive training time and high computational cost. To overcome these obstacles, this thesis takes advantage of distributed frameworks and cloud computing to develop a parallel CNN algorithm. Close examination of the implementation of MapReduce based CNNs as well as how the proposed algorithm accelerates learning are discussed and demonstrated through experiments. Results reveal high accuracy in classification and improvements in speedup, scaleup and sizeup compared to the standard algorithm.

Acknowledgements

I would like to extend my gratitude to my advisor Professor Min Chen in allowing me the opportunity to undertake this thesis. I am grateful for her open-mindedness, understanding as well as the guidance she provided whenever I needed it. I would also like to thank my parents for all the adversity they've overcome and sacrifices made to provide me with everything I have today. Also my sisters for all the lessons they've taught me which has greatly influenced who I've become today.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Introduction to Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [?] is a deep learning technique that has become increasingly popular in computer vision systems today. It has reached state-of-art performance in a number of applications including image classification [?], [?], [?], detection [?], [?], [?] and segmentation [?], [?]. Compared to standard machine learning algorithms, deep learning methods learn from experience and through that, forms a hierarchy of concepts to get a better understanding of the world. Learning through experience avoids the need for humans to manually engineer hand crafted features which is a main advantage of deep models over traditional methods. Additionally, certain deep learning algorithms make explicit assumptions about the input data that allow them to perform remarkably well in specific tasks. For example, CNNs make assumptions about the properties inherent in images such as stationary of statistics and locality of pixel dependencies which allow for fewer connections and parameters, leading to less computational demands and easier training. Furthermore, because the design of CNNs were inspired by the human visual system, they can be trained to recognize complex visual patterns and rich features that have shown to even outshine human performance.

Despite these remarkable qualities, a major drawback with deep learning models is that they require large amounts of data in order to perform well. Only through the rise of large annotated datasets such as ImageNet [**?**] and the growth of graphics processor units (GPUs) have CNNs been able to achieve the recent success that they've had. Additionally, with the rise of big data and increased complexity of tasks, training CNNs incur a high computational cost which is infeasible on a single machine without large computational resources. Moreoever, CNN architectures have become increasingly deeper which on one hand, allows for better feature representations but also increases the complexity of the network. ResNet [**?**] is about twenty times deeper than AlexNet [**?**] and eight times deeper than VGGNet [**?**]. State-of-art CNNs require a massive number of parameters that need to be tuned during training which leads to extensive training times and models that are highly difficult to optimize.

1.2 Motivation and Problem Statement

To cope with increasingly complex tasks and volumes of data, the evolution of CNNs have severely impacted their efficiency. CNNs are innately both data and computationally intensive which make speed and storage capacity a large limiting factor in reaching performance and scalability requirements. To overcome the imposed time and space obstacles, this thesis implements a parallelized CNN algorithm based on MapReduce [**?**] on a cloud computing cluster. The developed algorithm takes advantage of the computational structures inherent in CNNs that lends them to parallelization to achieve increased processing speed. Additionally, the use of cloud computing provides an economical means to facilitate data intensive applications by providing workload balancing and resource scheduling.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 provides an introduction to supervised machine learning and uses neural networks as a model to understand the basic functions and mathematical details of machine learning. The examination of neural networks serve as a framework to understand the design of convolutional neural networks. Chapter 3 presents an overview of Hadoop, HDFS and MapReduce. A brief history on the origin of Hadoop is presented as well as a description of the architecture of HDFS. A simple word count program using MapReduce is constructed to serve as a practical example. The proposed algorithm and relevant work is discussed in chapter 4. Chapter 5 describes details on the environmental setup in addition to experimental results. Finally, the thesis is concluded with a summary and future work in chapter 6.

2. Background

This section provides a brief overview of supervised machine learning and goes further into the mathematical details by using neural networks as an example. Additionally, an introduction to convolutional neural networks is presented.

2.1 Supervised Machine Learning

Supervised learning comprises the majority of practical machine learning tasks. It requires the use of a "training set" which includes some input data X and corresponding label Y. Using image classification as an example, the input X would be an image whereas the output Y is the class the image belongs to such as a cat or dog. The goal then is to learn a function $f : X- > Y$ that maps the input X to output Y such that when given new, unseen data, can accurately predict its output.

2.2 Neural Networks

Neural networks are one type of supervised method originally inspired and developed by modeling the brain. The brain's basic computational unit is the neuron and this is equally true for neural networks. Starting with a simple example using a single neuron helps simplify the workings of neural networks. Consider a supervised learning task consisting of n training examples and their corresponding labels $\{(x_1, y_1), ...(x_n, y_n)\}$. A neuron takes inputs $(x_1..x_n)$ and produces a hypothesis function $h_{w,b}(x)$ which can also be referred to as the predicted output. W, b are parameters which are learned and fine-tuned during the training process, W represents weights and b a bias term. Weights interact multiplicatively with the input and control the importance of respective inputs to the output while the bias adjusts when the neuron activates. Equation 2.1 shows the output of a neuron.

$$h_{w,b}(x) = f(W^T x) = f(\sum_{i=1}^{n} W_i x_i + b) \tag{2.1}$$

As seen in the above equation, each neuron computes a dot product between the input and its weights with the addition of the bias. The result is then passed into an activation function f. The activation function comes in many forms including Tanh, ReLU and Maxout but for this case the sigmoid function is used:

$$f(z) = \frac{1}{1 + exp(-z)} \tag{2.2}$$

Activation functions control what information is passed onto other neurons. They act as a switch that activate when the input is above a certain threshold value. The sigmoid function outputs $[0, 1]$ so will either turn on (1) or off (0) based on its input.

`dia/single_neuron1.pdf`

Figure 2.1: A single neuron.

A neural network is essentially a massive interconnected network of neurons. Each neuron computes a dot product between the input and weights in addition to adding the bias. The result is then passed into an activation function which sets a threshold to determine what information is passed on to other neurons. The procedure of processing information further down the network is commonly known as the feedforward pass. Typically neural networks are arranged in a series of "hidden layers" where each hidden layer contains a series of neurons. The output of any given neuron may be the input to several other neurons in the next hidden layer. As certain neural connections are activated over others, connections strengthen and the network begins to learn these relations. As more patterns are learned, the better the network performs at the given task.

dia/multiple_neurons.pdf

Figure 2.2: A simple 3-layer Neural Network. A series of neurons populate the hidden layers; neurons in a single hidden layer do not form any connections with one another and operate independently. But in each hidden layer, neurons are fully connected to neurons of the previous layer.

2.3 Objective Function

The goal for any supervised machine learning task is to minimize the error between the predicted output $\hat{y}$ and true label y. If there exists a collection of images containing dogs, cats and frogs then the objective of the model is to classify each of these images into the correct category with few mistakes as possible. Additionally, the model should have high generalization which means it should perform just as well on the "test data" or new images it has not seen during training. In order to facilitate learning, the network needs to know whether its predictions are close or far away from the ground truth label. One method typically used in machine learning to compute the degree at which the prediction differs from the desired outcome is the loss function $L(\hat{y}, y)$.

As an example, here is a common loss function for regression tasks, where the prediction is a scalar-valued quantity:

$$J(W, b) = [\frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2] \tag{2.3}$$

Equation 2.3 is known as the squared error cost function which simply computes the average sum-of-squared error between the prediction $\hat{y} = h_{w,b}(x)$

and true outcome y between all n examples. So after an example is fed to the network, a prediction $\hat{y}$ is produced, then the error $(\hat{y}^{(i)} - y^{(i)})$ is computed by calculating the difference between the prediction and true label y. After the error is computer for every example, the results are summed and then divided by the total number of examples to get the average loss. Although squared error works well in regression settings, in classification, the input is typically assigned to a discrete category. For this case, cross entropy loss is a common choice which takes the form:

$$L(\hat{y}, y) = -\sum_{k=1}^{K} y_k \, log(\hat{y}_k) \tag{2.4}$$

Consider the task of classifying an image of a cat into three classes (dog, cat, frog) where the ground truth label is cat. The vector $\hat{y}_k$ becomes a vector of length three representing the predictions for each class ($\hat{y}_k = [.7, .6, .4]$) that the trained model had made. Vector yk is also of length three but contains all zeros except at the index of the true class which has a value of one ($yk = [0, 1, 0]$ since cat is at the second index). Additionally, the predictions of the network should be interpretable, thus converted to probabilities, so the softmax function:

$$f_j(z) = \frac{e_j^z}{\sum_{k=1}^{K} e^{z_k}} \tag{2.5}$$

is commonly used which takes a vector z as input and outputs a normalized vector with values between zero and one that sum to one. With this function, our prediction vector $\hat{y}_k$ can be transformed to take on values that represent the probability of the input being assigned to each class e.g. $\hat{y}_k = [0.3, 0.6, 0.1]$ means 30% dog, 60% cat, 10% frog.

With a loss function now in place, the objective is to solve:

$$f^* = argmin_{w,b}[\frac{1}{n}\sum_{i=1}^{n}(w^T x_i + b - y_i)^2] \tag{2.6}$$

which seeks to find parameters (W, b) that minimize the error between the prediction $\hat{y} = w^T x + b$ and true label y.

In summary, a function first maps the input data X to output Y. Then a loss function $L(\hat{y}, y)$ calculates the error between the prediction $\hat{y}$ and true label y. Next, the weights W need to be adjusted to minimize the error between predicted and true labels which is covered in the next section.

2.4 Optimization

In the previous section, a function mapped the input data X to output Y and a loss function $L(\hat{y}, y)$ was used to calculate the error between the prediction $\hat{y}$ and true label y. The objective then is to seek parameters W, b that minimize the loss. To optimize the loss function, the concept of gradient descent is introduced which is an optimization algorithm commonly used in practice for machine learning tasks. The main strategy behind gradient descent is to first initialize weights W to random values and then iteratively adjust them in the direction that lowers the loss. To find which direction, the gradient of the loss function is computed. The gradient is a vector of slopes and the slopes are calculated by taking derivatives of each dimension in the input space. See figure 2.3 for an example.

Using the slopes, the direction of steepest descent down the cost function can be computed. More formally, the partial derivatives of the cost function with respect to parameters W, b are first calculated then updated as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \tag{2.7}$$

`dia/SGD.pdf`

Figure 2.3: Gradient descent performed in one dimension. A weight w is randomly initialized and the gradient is calculated at that point. The gradient provides the direction of steepest descent and the learning rate determines how large of a step to take for each weight update. The process continues until the minimum on the cost curve $J_{min}(w)$ is reached.

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b) \tag{2.8}$$

α is the learning rate which determines how large or small of a step to make when adjusting parameters. A learning rate that is too small will cause the network to learn very slowly whereas a rate that is too high may cause oscillations around the minimum but not actually reach it.

The partial derivatives are computed by a process known as backpropagation. Neural Networks operate in two stages: feedforward and backpropagation. The feedforward stage was described earlier where the input is fed through a series of neurons layer by layer until a prediction is produced in the output layer. Next, the error between the prediction and expected output is calculated and because each neuron made some contribution to the overall error, the error signal is propagated back to each neuron and a value representative of the relative error produced at each neuron is calculated. These values are used to update the weights at each neuron with the aim of progressively decreasing its contribution to the overall error after each iteration.

Algorithm 1: Feedforward and Backpropagation Pseudo Code

```
Initialize weights $W$ with small random values between -1 and 1;
while currentIteration < maxIterations do
    // Feedforward stage
    for each layer in the network do
        for each node in layer do
            1. Compute linear transformation between input and
             weights.
            2. Add bias term $b$.
            3. Perform non-linear activation function.
    for each node in the output layer do
        Calculate the error between predicted output and true label.
    // Backpropagation
    for each hidden layer do
        for each node in hidden layer do
            1. Compute the node's error.
            2. Update the node's weight.
    Calculate Global Error
```

As a concrete example, consider the expression $f = (3x + 5)^2$. f is a function of some intermediate functions that can be defined as $a = 3x$, $b = a + 5$ and $f = b^2$. The objective is to find the derivative of each intermediate function with respect to its input: $\frac{\partial a}{\partial x} = 3$, $\frac{\partial b}{\partial a} = 1$ and $\frac{\partial f}{\partial x} = 2b$. Now with the local gradients computed the chain rule is applied to obtain the global derivative: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$. The same idea applies to neural networks but the output f is replaced with the total loss whereas x is the input and network parameters.

images/backprop.png

Figure 2.4: Backpropagation on a single neuron. After the feedforward pass the output z is obtained. Local gradients $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ are computed to find their influence on z. Since z contributes some portion to the overall loss of the network the amount of influence is calculated by computing $\frac{\partial L}{\partial z}$. The gradient of z is then backpropagated to the local gradients to find the global gradient for x and y by applying the chain rule e.g. for x: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$ [**?**].

2.5 Convolutional Neural Networks

Convolutional neural networks operate very similarly to neural networks; they both perform a feedforward and backwardpass which includes computing a linear transformation between the input and weights, followed by a non-linear activation function, calculating the error and then backpropagating the error back to the weights. Both systems are also made up of a series of neurons but the connections between these neurons is how they differ.

In neural networks each neuron in one layer operates independently and doesn't share any connections but instead was connected to each neuron in the next layer. This is problematic when working with images because a 300x300x3 image would result in neurons with 270,000 weights which is computationally costly and would also lead to overfitting. Overfitting occurs when a trained model fits the noise of the data thus hampering its ability to generalize well to new examples. Luckily, CNNs specialize in dealing with images by utilizing convolution along with local receptive fields to help control the amount of parameters. Imagine taking a filter of size 5x5, sliding it across an entire image to find features that match the filter. This is exactly how convolution is performed but the filter is initialized with weights and dot products are computed between the weights and input image exactly as in neural networks. With convolution, instead of neurons being connected in a fully connected manner, neurons in one layer are only connected to a small region of the layer before it which is controlled by the size of the filter or receptive field. See figure 2.5 for an illustration.

dia/convolution.pdf

Figure 2.5: Convolution performed on a single channel of an image. A filter is slid across the entire input, computing an element-wise product between each input pixel and the filter then summing the results to form the output elements. Here *padding=1* and *stride=2*.

There exists several hyperparameters (values set before training) that can be adjusted when performing convolution: the filter size, padding and stride are three. First, the size of the receptive field controls the connection of neurons to their input spatially (width and height) but always through the entire depth. For example, using a receptive field of size 5x5 on a 300x300x3 image leads to neurons with $5*5*3 = 75$ weights. A large receptive field may capture more context but could also lead to a loss in finer details. Secondly, notice how in Figure 3 the border of the input is padded with zeros. This is set to preserve the size of the input and output so that additional convolutions will not cause the input to shrink too quickly, resulting in information loss. Lastly, the stride determines how many pixels to move when the filter is slid across the image; a stride of one or two is commonly used in practice.

The output size after convolution can be computed by the function

$$(W - F + 2P)/S + 1 \tag{2.9}$$

where W is the input size, F the filter size, S the stride and P the amount of padding used. For example with an input (W) = 5x5, filter size (F) = 3x3, stride (S) = 2, padding (P) = 1 then the output size = 3x3.

Just as with neural networks, after a linear transformation is applied between the input and weights, an activation function is used. The ReLU (Rectified Linear Unit) [?] is a common activation function used with CNNs.

It simply computes $f(x) = max(0, x)$ which removes negative values by placing a threshold at zero.

Pooling layers act as a way to reduce the size of the input, number of parameters and computational cost; it also helps control overfitting. The most commonly used pooling layer is max pooling which takes the max element of a region in the input and discards the rest. Figure 2.6 shows an example of the pooling layer.

dia/maxpool.pdf

Figure 2.6: Max pooling operation performed using a filter of size 2x2 and *stride = 2*.

Convolutional neural networks are essentially a series of convolution and pooling layers stacked together with a fully connected layer inserted before the output layer. Additional layers help the network learn more complex features. For example if the network is fed the image of a face then the first layer will detect low level features such as edges while deeper layers detect higher level features such as eyes or a nose and eventually an entire face.

`dia/cnn.pdf`

Figure 2.7: Example CNN architecture.

3. Introduction to Hadoop

Hadoop [?] originated from Apache Nutch, an open source web search engine which manifested in 2002. Doug Cutting, the project's creator, successfully deployed a working version of the system but soon realized that their architecture wouldn't scale to the billion of pages found on the web. However, in 2003 Google published a paper describing their distributed file system which was being used in production at the time. The paper served as a critical piece in providing the solution to Cutting's storage issues which led to the development of the Nutch Distributed File System (NDFS) in 2004. Google soon released a second paper - MapReduce: Simplified Data Processing in Large Clusters in 2004 and as a result, by the end of 2005, Nutch developers had a working version of MapReduce and NDFS. In 2006, the project migrated from Nutch to an independent project called Hadoop. Hadoop rose to become the fastest system to sort a terabyte of data in 2008 and was also being used at Yahoo! in its production search index. Now, Hadoop has become the standard for open-source cloud computing providing reliable, scalable, distributed computing.

3.1 HDFS Overview

Hadoop includes a distributed file system (HDFS) designed for commodity hardware to store large datasets with streaming data access and the ability to scale to hundreds or even thousands of machines. HDFS is built around the idea that moving computation is cheaper than moving data. This hold true when working with massive amounts of data so HDFS migrates the computation closer to the data instead of moving the data closer to the application. HDFS was designed to handle files of several gigabytes or terabytes, it follows the write once, read many times paradigm which favors tasks that require reading the entire dataset once and then performing multiple analyses over it. Other tasks which require low latency access, multiple writers or many small files will not reap the benefits of HDFS.

HDFS clusters utilize a master/slave architecture consisting of namenodes (master) and datanodes (slaves). Namenodes are responsible for managing the file system namespace, maintaining the file system tree as well as the metadata for all the files and directories in the tree. Datanodes on the other hand, act as containers for blocks. Blocks hold data that is divided from HDFS which are stored in datanodes. To prevent disk and machine failure, blocks are replicated to a small number of separate machines. If one block becomes corrupt, a separate copy is read from to create additional copies. In addition to storing blocks, datanodes also perform block creation, deletion and replication. Figure 3.1 shows at a high level the architecture of HDFS.

images/hdfs.png

Figure 3.1: HDFS architecture [?].

3.2 MapReduce

MapReduce is a programming model designed as part of the Apache Hadoop project. It was designed to allow programmers the ability to process and generate massive datasets without the need for any previous distributed systems experience. It is a simple yet powerful framework that offers data distribution, parallelization, fault tolerance and load balancing only through the addition of simple map and reduce functions.

MapReduce performs the data processing and distribution of tasks across nodes. Every MapReduce job consists of two distinct entities: a mapper and a reducer. The mapper takes as input a set of key/value pairs and then transforms the data according to the logic contained in the map function. It outputs a new set of key/value pairs which is fed into the reducer. The reducer combines key/value pairs based on their key and produces an aggregated result.

`images/wordcount.png`

Figure 3.2: A simple word count program using MapReduce [**?**].

Figure 3.2 depicts the classic word count example. Imagine having a text document containing hundreds of millions of words and needing to count the occurrence of each word contained in the document. This task can be easily broken down into simple scripts using MapReduce. First the words in the document are tokenized to form the input which is then split and passed onto different machines. The mapper then takes the input and maps it to $< key, value >$ pairs, which in this case, is a $< word, count >$ pair. So for each word the mapper produces a $< word, 1 >$ pair. Next, $< key, value >$ pairs are shuffled, meaning pairs with the same key are grouped together and placed into the same machine. This allows the reducer to aggregate grouped pairs based on some function provided by the user. For this example, the reducer sums the values of pairs which have the same key, resulting in frequencies for each word found in the documents. Below is the pseudo code for the word count example using the map and reduce tasks.

Algorithm 2: Word count using MapReduce [**?**]

```
The mapper emits an intermediate key-value pair for each word in a
  document.
The reducer sums all counts for each word.
Function MAP(docID a, doc d)
    for each term $t \in$ doc d do
        Emit(term $t$, count 1)
Function REDUCE(term t, counts [c1,c2,...])
    $sum \leftarrow 0$
    for each count $c \in$ counts [c1,c2,...] do
        $sum \leftarrow sum + c$
    Emit(term $t$, count $sum$)
```

4. Proposed Approach

4.1 Related Work

With the rise of big data, a number of research efforts have been proposed to take advantage of multicore machines to increase the speed of learning algorithms. Most of these efforts have been gears towards traditional machine learning algorithms. For example, Chu et al. [?] show that any learning algorithm that fits the Statistical Query model [?] can be written in "summation form" and parallelized on multicore computers. They use the MapReduce paradigm to parallelize a number of algorithms including linear regression, k-means, logistic regression, naive Bayes, SVM, ICA, PCA, gaussian discriminant analysis, EM and backpropagation using artificial neural networks (ANN). Their results report linear speedup with an increasing number of processors. Additionally, Sun et al. [?] developed a MapReduce based ANN applied to handwritten digit recognition whereas Liu et al. applied ANNs over large scale mobile data. [?] Other studies from Liu et al. [?] analyzed MapReduce based ANNs under three different scenarios. One in which the dataset to be classified is large, two where the volume of training data is large, and three where the number of neurons is large. They found that computational overhead can be reduced using multiple computers in parallel but fully distributing an ANN with a large number of neurons causes a high overhead of computation due to the starting and stopping of mappers and reducers.

Although multiple studies have reported improvements in speed and accuracy using parallel ANNs, ANNs are traditionally deployed using small datasets and they don't scale well to images. CNNs on the other hand, were specifically designed to handle images and have achieved state-of-art performance in a wide range of applications. However, long training times and high computational cost still remains a severe issue. Several methods have proposed to improve the speed of convergence by adjusting the initialization of weights [?], [?] and controlling learning rate [?], [?]. Though these techniques don't take advantage of the speed up offered by parallelization using distributed computing.

4.2 MapReduce Based CNN

Convolutional Neural Networks is one deep learning algorithm that can benefit from the parallelized computation offered by the MapReduce programming model. CNNs iteratively adjusts weights in the network by computing their partial gradients after each set of the training data is propagated through the network. Thus parallelization during the training phase can be accomplished by distributing the data into a number of chunks. Each data chunk can then be fed to several CNNs and each CNN can be trained independently in parallel. The outputs can then be aggregated to produce the final results which are then used to update the weights for the next iteration. The figure below shows a high level overview of the procedure.

dia/MRflowchart.pdf

Figure 4.1: Parallel CNN.

The mappers take as input with the training data (x, y) and a set of randomly initialized weights $[w1, w2, ..., wn]$. The set of weights contain n weights to represent the number of hidden layers in the network so w_i corresponds to the weights for hidden layer i. The training data is represented as a set of tuples (x, y) where x is a training instance and y is the ground truth label for that instance. Each mapper initializes the network with the given set of weights and the network is trained using the input samples. The output is a set of newly trained $updatedWeights$ $[\Delta w1, \Delta w2, ..., \Delta wn]$ which is then fed to the reducer. After each iteration the mappers receive a set of updated weights which are processed through the network until the max number of iterations is reached. The pseudo code for the map function is shown in Algorithm 3.

Algorithm 3: The mapper of MapReduce based CNN

```
Input to mapper is a set of tuples $(x, y)$ where x is a training sample
  and y is the ground truth label for that sample. $Weights$ is a list of
  randomly initialized weights for each hidden layer.
Function MAP()
    Read weights from HDFS;
    Initialize CNN with random weights;
    for $i \leftarrow 0$ to $numLayers - 1$ do
        $updatedWeights[i] \leftarrow updatedWeights[i] + Weights[i]$
    output: $updatedWeights$
```

The reducer receives as input the intermediate output by the mappers and aggregates the weights produced from each of the mappers. Since the position of weights in the output of mappers are sorted according to the hidden layer they belong to, the reducer can simply aggregate weights according to their index. The aggregation computes a cumulative sum over weights and then divides by the number of training instances in the batch to form an average of weights. The final result is used to update the weights in the network and sent to the mapper for the next iteration. The pseudo code for the reduce function is shown in Algorithm 4.

Algorithm 4: The reducer of MapReduce based CNN

```
Input to the reducer is intermediate output by mappers which are
  the updated weights for each hidden layer (numLayers).
  $sumWeights$ is a list which stores the cumulative sum of weights
  for each hidden layer.
Function REDUCE()
    $sumWeights \leftarrow Initialize\, list\, of\, zeros$;
    for $i \leftarrow$ 0 to numLayers-1 do
        $sumWeights[i] \leftarrow sumWeights[i] + Weights[i]$
    output: $sumWeights$
```

MapReduce jobs utilize a driver to serve as the scheduler for tasks. The driver creates a directed acyclic graph (DAG) or execution plan for the program which are then divided into smaller tasks to be executed. communicates with Hadoop, determines which map and reduce classes are used and specifies the configuration of jobs. Configurations are provided by the user and include the path to the training data, path of the output, training parameters and configurations for the network. Training parameters include number of training samples, number of validation samples, max number of iterations, max epochs and batch size. Network configurations include the size of the receptive field, stride, number and type of layers, learning rate and optimization method. The pseudo code for the driver function is shown in Algorithm 5.

Algorithm 5: The driver of MapReduce based CNN

```
Function Driver()
    for layer ← 0 to numLayers-1 do
        $Weights[layer] \leftarrow Randomly\,Initialized\,Weights$
    $numIteration \leftarrow 1$
    while $numIteration <= maxIterations$ do
        Initialize MapReduce Job;
        Store weights in HDFS;
        Pass model parameters to MapReduce Job;
        Run MapReduce jobs:
        Input for Mappers: Weights for each hidden layer;
        Mappers compute updated weights for every hidden layer;
        Output of Mappers: Newly trained weights;
        Aggregate Weights of each hidden layer:
        Input to Reducer: Newly trained weights from Mappers;
        Reducers aggregate weights for each hidden layer:
        for layer ← 0 to numLayers-1 do
            $Weights[layer] \leftarrow$
              $Weights[layer] + updatedWeights[layer]$
        Output of Reducer: Cumulative sum of weights for each
          hidden layer;
        Update weights of each hidden layer;
        $numIteration + +;$
    output: Final Weights
```

5. Experiments and Results

5.1 CNN Architecture and Parameters

The CNN architecture used in experiments is an adaptation of VGGNet. The input is 32 x 32 RGB images with 49,000 images used for training and 1,000 for validation. The number of training iterations is set at a maximum of 60 epochs. No data augmentation is used, training data mean subtraction is the only pre-processing step. See table 5.2 for more details on training parameters.

The network contains 10 weight layers (8 conv. and 2 FC layers). The number of channels begins with 64 in the first layer and doubles after every max-pooling layer until it reaches 512 in the last convolution layer. Small 3 x 3 receptive fields and stride 1 are used throughout the network. Every conv. layer is followed by the ReLU non-linearity. He initialization [?] is used at every weight layer. Two Fully-Connected (FC) layers are inserted after the stack of conv. layers. The first FC layer contains 500 channels and the second contains 10 channels for the 10 classes contained in the CIFAR-10 dataset. The final layer is the soft-max layer to obtain class probabilities and categorical output. Configuration details are outlined in Table 5.1.

5.2 Distributed Computing Environment

The algorithm is deployed on a Hadoop cluster using Amazon Web Service EMR which provides economical, large capacity, remote computing services. Parallel CNN is implemented using Spark, an extension of the MapRe-

Table 5.1: CNN Architecture details. The convolutional layer parameters are denoted as conv<receptive field size> - <number of channels>.

input (32 x 32 RGB image)
conv3-64 conv3-64
maxpool conv3-128 conv3-128
maxpool conv3-256 conv3-256
maxpool conv3-512 conv3-512
maxpool
FC-500
FC-10
soft-max

Table 5.2: Model Parameters.

Train samples	Validation samples	Batch size	Max Epochs	Optimizer	Learning Rate
49,000	1,000	100	60	Adam	.0001

duce framework which supports fast in memory computations, specifically designed for iterative machine learning algorithms. The proposed parallel algorithm is run using the following distributed computing environment:

• Hadoop: The cloud compute cluster to assign namenode and datanodes, conduct work load balancing, resource scheduling and data replication.

• HDFS: The distributed file system that provides fault tolerance and high throughput access to large datasets.

• YARN: Provides job scheduling and resource management.

• SPARK: An extension of the MapReduce framework which makes use of Resilient Distributed Datasets (RDDs) as a fault tolerant data structure operated on in parallel.

Table 5.3 provides more details on the cluster specifications.

Table 5.3: Cluster details.

Namenode	Datanodes	Network bandwidth	Hadoop Version	Spark Version
CPU: E5@2.4 GHz Memory: 32 GB OS: Linux	CPU: E5@2.4 GHz Memory: 32 GB OS: Linux	1 Gbps	2.8.4	2.3.1

Experiments conducted to evaluate the performance of the proposed parallel algorithm include measurements of accuracy, speedup, scaleup and sizeup. The dataset used in all experiments come from the CIFAR-10 [?] dataset which contains 50,000 RGB images of size 32 x 32.

The initial proposed architecture achieves 85% accuracy on the validation data. Though overfitting was an issue using the initial architecture as seen by the gap between the training and validation curves in Figure 6.2. To remedy this, dropout [?] layers are introduced and added after every max pool layer and after the first fully connected layer. A dropout value of 0.4 was used throughout the network. Figure 6.4 shows the performance of the network after adding dropout. An accuracy of 88.4% is achieved on the validation data.

Figure 6.5 and Figure 6.6 compare the effect of dropout on both training and validation accuracy. Dropout essentially adds noise to the network by ignoring a fraction of nodes during training which reduces co-adaptation of neurons. Thus neurons are forced to learn independently and not over

Figure 5.2: Train and validation Accuracy without Dropout.

rely on one another. The effect is shown by the wider spread loss curve in Figure 6.3 and lower training accuracy shown in Figure 6.5. But dropout provides the added benefit of improved generalization as shown from the higher validation accuracy achieved in Figure 6.6. Furthermore, Figure 6.6 reveals that dropout requires a greater number of training epochs to take effect. From epochs 0 to 20 the network without dropout consistently receives higher validation accuracy. Between 20 to 30 epochs, both networks are effectively equal in accuracy but not until after 30 epochs is when dropout reveals its performance boost.

Improvements in speed and scalability of the proposed algorithm is also measured. Speedup is defined as the ability for a system to yield m times speedup with m times the number of nodes. To measure speedup the number of samples were kept constant at 50,000 while the number of nodes were increased by 2, 4, 8 and 16 respectively. For each number of nodes, five trials were conducted and the average time was recorded. Results are shown in Figure 6.7 where the dashed lines represent the minimum and maximum error while the middle solid line represents the average. The parallel algorithm achieves close to linear speedup. Exact linearity isn't achieved due to the communication overhead with a large number of nodes.

Scaleup measures the ability of a m-times larger system to perform a m-times larger task at the same time as the original system. To evaluate scaleup the dataset size is increased in proportion to the number of nodes in the system. The number of samples is increased from 10k, 20k, 30k, 40k and the number of nodes is increased from 1 to 4. Five trials for each sample size was conducted and the average time was recorded. Figure 6.8 depicts the results of the scaleup experiments. Again, the dashed lines represent the minimum and maximum error whereas the middle solid line represents the average. Results reveal that scaleup improves as the number of nodes in the system increases.

Figure 5.3: Training loss with Dropout.

Sizeup increases the size of the input data by a factor of m while holding the number of machines in the system constant. To measure sizeup, experiments are run on 1, 2 and 4 machines respectively. For each number of machines, the dataset size is increased from 10k, 20k and 40k samples. The results in Figure 6.9 show the proposed algorithm performs well in terms of sizeup but that as the number of machines increases, sizeup performance is hampered due to the communication overhead between nodes.

Figure 5.4: Train and validation Accuracy with Dropout.

Figure 5.5: Training accuracy Comparison with and without Dropout.

Figure 5.6: Validation accuracy Comparison with and without Dropout.

`images/speedup.png`

Figure 5.7: Parallel CNN speedup.

`images/scaleup.png`

Figure 5.8: Parallel CNN scaleup.

`images/sizeup.png`

Figure 5.9: Parallel CNN sizeup.

6. Conclusion

With the rise of big data and increased complexity of tasks, the efficiency of deep learning algorithms have been severely impacted, resulting in long training times and high computational cost. To be able to solve even greater problems of the future, learning algorithms must maintain high speed and accuracy through economical means. To that end, this thesis takes advantage of the MapReduce framework to develop a parallel CNN algorithm. The proposed algorithm achieves high accuracy in image classification and close to linear speedup, scaleup and sizeup. The results demonstrate that the MapReduce framework is an effective tool to improve the speed and scalability of CNNs.

In terms of directions for future work, several ideas come to mind. Experiments conducted here utilize images from CIFAR-10. A much larger dataset such as ImageNet with additional compute nodes could be used to analyze how this affects performance in speedup, scaleup and sizeup. Additionally, instead of image classification, other computer vision tasks such as image segmentation or object detection could be applied. Trying other CNN architectures such as ResNet or GoogLeNet [?] and examining their performance in accuracy and speed of convergence is another interesting path. Lastly, ensemble methods similar to the work done in [?] can also be explored.

Bibliography

[1] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.

[2] M. Egmont-Petersen, D. de Ridder, H. Handels Image processing with neural networksa review Pattern Recognit., 35 (10) (2002), pp. 2279-2301

[3] K. Simonyan, A. Zisserman Very deep convolutional networks for large-scale image recognition Proceedings of the International Conference on Learning Representations (ICLR) (2015)

[4] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge

[5] S.J. Nowlan, J.C. Platt A convolutional neural network hand tracker Proceedings of the Advances in Neural Information Processing Systems (NIPS) (1994), pp. 901-908

[6] M. Everingham, S.A. Eslami, L. Van Gool, C.K. Williams, J. Winn, A. Zisserman The pascal visual object classes challenge: a retrospective Int. J. Conflict Violence (IJCV), 111 (1) (2015), pp. 98-136

[7] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, C.L. Zitnick Microsoft Coco: Common Objects in Context Proceedings of the European Conference on Computer Vision (ECCV) (2014), pp. 740-755

[8] Ciresan, Dan, et al. “Deep neural networks segment neuronal membranes in electron microscopy images.” Advances in neural information processing systems. 2012.

[9] Girshick, Ross, et al. “Rich feature hierarchies for accurate object detection and semantic segmentation.” Proceedings of the IEEE conference on computer vision and pattern recognition. 2014.

[10] He, Kaiming, et al. “Deep residual learning for image recognition.” Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[11] Simonyan, Karen, and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition.” arXiv preprint arXiv:1409.1556 (2014).

[12] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters.” Communications of the ACM 51.1 (2008): 107-113.

[13] Karpathy, Andrej, and Justin Johnson. “Convolutional Neural Networks.” CS231n Convolutional Neural Networks for Visual Recognition. Stanford U,
`cs231n.github.io/convolutional-networks/`.

[14] Nair, Vinod, and Geoffrey E. Hinton. “Rectified linear units improve restricted boltzmann machines.” Proceedings of the 27th international conference on machine learning (ICML-10). 2010.

[15] White, Tom. Hadoop: The definitive guide. “O’Reilly Media, Inc.”, 2012.

[16] Borthakur, Dhruba. "HDFS Architecture Guide." Hadoop, 2013, `hadoop.apache.org/docs/r1.2.1/hdfs_design.html.`

[17] Capitão, Micael José Pedrosa. Mediator framework for inserting data into hadoop. University of Aveiro, 2014.

[18] Lin, Jimmy, and Chris Dyer. "Data-intensive text processing with MapReduce." Synthesis Lectures on Human Language Technologies 3.1 (2010): 1-177.

[19] Chu, Cheng-Tao, et al. "Map-reduce for machine learning on multicore." Advances in neural information processing systems. 2007.

[20] Kearns, Michael. "Efficient noise-tolerant learning from statistical queries." Journal of the ACM (JACM) 45.6 (1998): 983-1006.

[21] Sun, Kairan, et al. "Large-scale artificial neural network: Mapreduce-based deep learning." arXiv preprint arXiv:1510.02709 (2015).

[22] Liu, Zhiqiang, Hongyan Li, and Gaoshan Miao. "MapReduce-based backpropagation neural network over large scale mobile data." Natural Computation (ICNC), 2010 Sixth International Conference on. Vol. 4. IEEE, 2010.

[23] Liu, Yang, et al. "MapReduce based parallel neural networks in enabling large scale machine learning." Computational intelligence and neuroscience 2015 (2015): 1.

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

[25] Nguyen, Derrick, and Bernard Widrow. "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights." Neural Networks, 1990., 1990 IJCNN International Joint Conference on. IEEE, 1990.

[26] D. Kingma, J. Ba Adam: A method for stochastic optimization Proceedings of the International Conference on Learning Representations (ICLR) (2015)

[27] N. Qian On the momentum term in gradient descent learning algorithms Neural Netw., 12 (1) (1999), pp. 145-151

[28] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015.

[29] Krizhevsky, Alex, Vinod Nair, and Geoffrey Hinton. "The CIFAR-10 dataset." online: http://www. cs. toronto. edu/kriz/cifar. html (2014).

[30] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[31] Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[32] LeCun, Yann, et al. "Handwritten digit recognition with a back-propagation network." Advances in neural information processing systems. 1990.

[33] Hecht-Nielsen, Robert. "Theory of the backpropagation neural network." Neural networks for perception. 1992. 65-93.

[34] Zeiler, Matthew D., and Rob Fergus. “Visualizing and understanding convolutional networks.” European conference on computer vision. Springer, Cham, 2014.

[35] Ng, Andrew, et al. “Multi-Layer Neural Network.” Unsupervised Feature Learning and Deep Learning Tutorial. Stanford U, 2013, `ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/`.

[36] Papagelis , Anthony, and Dong Soo Kim. “Back Propagation Tutorial.” Multi-Layer Perceptron. UNSW Sydney, `www.cse.unsw.edu.au/ cs9417ml/MLP2/`.

[37] Gu, Jiuxiang, et al. “Recent advances in convolutional neural networks.” arXiv preprint arXiv:1512.07108 (2015).