Distributed Convolutional Neural Network with Apache Spark

Weicong Ma
University of Waterloo
w34ma@uwaterloo.ca

Yuwei Jiao
University of Waterloo
jyuwei@uwaterloo.ca

ABSTRACT

In this paper, we present a distributed convolutional neural network (CNN) built with Apache Spark. Our distributed CNN modifies the naive CNN implementation by broadcasting and splitting training data set into batches and distribute forward and backward propagation to multiple Spark worker nodes in a cluster. We showned that this is the only sensible approach to distribute CNN training as distributing matrix operation itself would introduce severe communication overhead. Our implmentation takes advantage of HDFS for persisting intermediate results across workers and Spark's awareness of data locality. We also improved our implementation with in-memory database Redis to speed up data I/O between forward and backward stages. The experimental results are shown in the end. Although we were able to surpass the performance of naive CNN implementation with the help of Redis, the performance was still not comparable to state of the art neural network framework such as Tensorflow. We discovered that Spark is not a good fit for CNN training, and a local multi-threaded implementation with shared memory space would be a better choice.

1. INTRODUCTION

In recent years, applied machine learning algorithms in both cases of offline and online analysis became quite an active research area [11] [21] [7]. The work is not only trying to develop novel algorithms or improving the performance of existing ones, but also has to do with providing open source libraries and a common infrastructure for researchers and engineers alike. Most of these libraries are built on top of existing batch and stream processing systems. Generally there are two reasons for combining machine learning and distributed systems. On one hand, most machine learning algorithms are built on large amount of input data and generate high volume of intermediate values during the training process. It becomes more and more difficult for a single machine to store all data as well as do calculations. For example, as of 2016, the computer program AlphaGo [26] which was developed by Google DeepMind [2] [3], played the board game with a professional human and finally beat him. The AlphaGo used a combination of machine learning and tree search techniques. It was initially trained with more than 30 million moves of historical games. Therefore it have to be deployed on hardware with asynchronous and distributed mode. On the other hand, their characteristics and scalability are a mirror of the characteristics of the processing systems on top of which they run. The performances of machine learning algorithms on distributed processing systems could be used to provide some inspirations to improve current infrastructures.

Two systems that became quite popular within the last several years are Spark [28] [27] and TensorFlow [1] [5]. The work done in this paper makes use of these two systems to implement a convolutional neural network. We implement the same algorithms on top of both systems and we empirically compare their characteristics and draw conclusions about the strengths and weaknesses of each. Within this section, we introduce convolutional neural network, Spark and TensorFlow as well as a set of related work in applied machine learning respectively.

1.1 Background

1.1.1 Convolutional Neural Network

Convolutional neural network(CNN) is a biologically inspired artificial neural network in machine learning and deep learning [13] [24] [29]. It is widely applied both in research and production today, especially in signal processing areas, like video and image recognition [14], natural language processing [10] [4] and computer vision [17] [18]. CNN is an important class of learnable representations applicable to numerous computer vision problems, and most of the successful models for image recognition are built on the top of CNN.

CNN attempts to model high level abstractions in data with one or more layers of convolutions [15] followed by pooling and fully connected layers because contemporary artificial intelligence theories believe deep networks are much more powerful than shallow networks. Like other artificial neural networks, in convolutional neural network, there are many layers between the input and output [23], allowing the algorithm to use multiple processing layers including linear and non-linear transformations to solve the tasks. The architecture of a CNN is designed to take advantage of the 2D structure of an input image with convolution layer. One benefit of CNN is that, because of the characteristics of sparse connectivity(Figure 1) and shared weights(Figure 2)

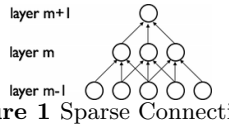

Figure 1 Sparse Connectivity

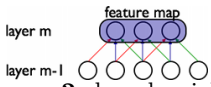

Figure 2 shared weights

in CNN, it is easier to train and has fewer parameters than fully connected networks with the same number of hidden units and layers, yet fewer training time and better accuracy and performance.

Figure 3 shows one classical implementation of convolutional neural network - LeNet5. It consists of two convolution layers, followed by two subsampling(pooling) layers. Convolution layers extract features from original images and subsampling layers reduce the spatial complexity. There are another two fully connected layer at the end of the network to match hidden units and expected output [16].

In our project we implement a simple version of convolutional neural network in order to catch the characteristics and test the performances on distributed processing systems.

1.1.2 Spark

Spark is an open source cluster computing framework built around speed, ease of use and sophisticated analysis, which provides an interface for programming entire clusters with data parallelism and fault tolerance. It allows in-memory computations on clusters using a data structure called *Resilient Distributed Datasets*(RDDs). RDDs are read-only data abstractions distributed over a cluster of machines, motivated by two computing applications, iterative algorithms and interactive data mining tools. They are "fault tolerant, parallel structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators".[28] Spark provides two methods to create RDDs: either from data in stable storage or from transformation operations from other RDDs. In general, Spark has several advantages compared to other data processing systems. First, it gives a comprehensive and flexible framework to manage big data processing. Second, the in-memory computation mechanism enables Spark to be as much as 100 times faster than on disk. Finally, programmers have more options to choose their developing languages. It comes up with more than 80 built-in high level operators.

A wide range of parallel problems can be expressed using RDDs. Spark Mllib [20] is a simple distributed machine learning framework on top of Spark system. Many common machine learning and statistical algorithms like decision

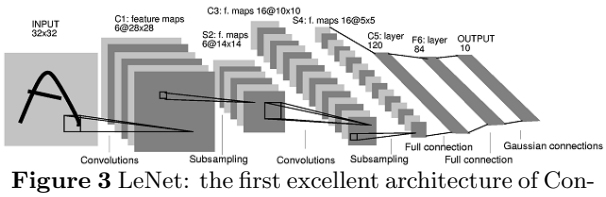

Figure 3 LeNet: the first excellent architecture of Convolutional Neural Network

tree, linear regression and support vector machine are implemented to simplifies large scale machine learning pipelines. However, convolutional neural network has not been provided yet. Therefore, in our project we would like to do some exploration to apply and implement convolutional neural networks on Spark.

1.1.3 TensorFlow

"TensorFlow is an interface for expressing machine learning algorithms and an implementation for executing such algorithm."[1] The system's flexible architecture can be used to express a wide variety of machine learning problems and deploy the computation on one or more CPUs or GPUs in mobile, desktop and server.

TensorFlow uses data flow graph to describe the mathematical computation. Typically nodes in the graph represents mathematical operations, while edges are multidimensional data arrays. Since TensorFlow is designed for machine learning and deep learning models, it provides both single-device execution as well as fault tolerance and cross-machine communication mechanisms for distributed execution. Besides, in order to help researchers and developers really focus on the model architecture instead of too much details of network, TensorFlow adds several advanced extensions for basic programming models, for example, gradient computation and device constraints.

In our project, we compare the performances of Spark-CNN with TF-CNN and draw conclusions about the two systems.

1.2 Related Work

At the 2016 Spark Summit East, Arimo, Inc. took single-machine TensorFlow to build a distributed implementation on Spark. Because Google kept distributed version of TensorFlow closed at that time, the combination of TensorFlow and Spark allows the powerful machine learning platform scales horizontally. They did this by using the abstraction layer called "Distributed DataFrame". And they also opened the source code on Github. However, they only tested the performance on small datasets.

Caffee is another powerful and expressive deep learning framework [8]. Models and optimization are defined by configurations without hard coding. Programmers can switch between CPU and GPU by setting a flag. Recently, Yahoo combined Caffee and Spark together called CaffeeOnSpark, enabling distributed deep learning on a cluster of GPU or CPU servers. As a distributed version of Caffee, CaffeOnSpark mainly supports distributed neural network model training, testing and feature extraction. Now it has been in use by Yahoo for image search, content classification and several other cases.

Victoria, Simon and Jim introduced a theoretical base for a Hadoop-based neural network for feature selection in Big Data set [6]. They implemented details of five feature selection algorithms embedded in Hadoop YARN.

GraphLab [19] naturally expresses asynchronous, dynamic and graph-parallel computation while keep the data consistency to support machine learning and data mining algorithms.

MLbase [12] is a novel system designed for mobile-end users and machine learning researchers. It provides simple declarative way to express the problem and sets of high-level run-time operators.

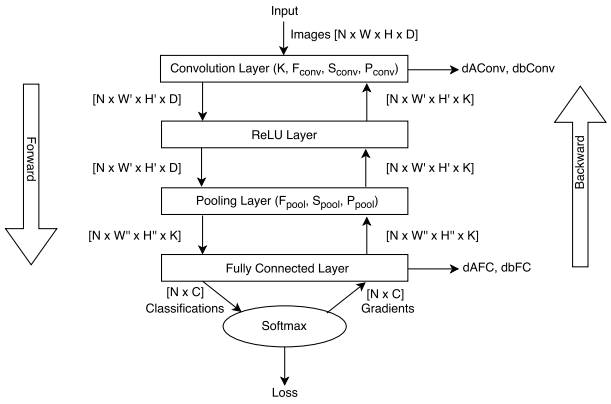

Figure 4 CNN Processing Flow for One Iteration

The rest of the paper describes our project in more detail. Section 2 describes our convolutional neural network implementation on Spark, and then our experimental results, evaluations and analysis in section 3. Finally in section 4, we conclude the project and include our future work.

2. DISTRIBUTED CNN IMPLEMENTATION WITH SPARK

In this section we will walk you through the steps we took to implement our distributed convolutional neural network with Apache Spark. Despite its distributed nature, it's still based on the fundamentals of standalone CNNs. As CNNs are usually used for image recognition and classification jobs, without losing generality, our system is built targeting cifar10, which is a commonly used image dataset to test CNNs for classifying given images into 10 categories. Since the focus of this paper is not on CNNs nor how CNNs work, for each component of a typical CNN, we will not explain its usage or reasons why it's included, but rather discuss how it's implemented and whether it can be converted to run in parallel on multiple nodes across a distributed cluster. All code for our system are shared as an open source project on GitHub, and can be accessed at https://github.com/w34ma/spark-cnn.

2.1 Naive Implementation

To better explain the intuition behind our solution, we must first introduce the naive implementation of CNNs. We have to understand each layer's functionality and characteristics (whether it's more computation oriented or resource oriented) to locate performance bottlenecks and justify design decisions and trade-offs. Remember we are not interested in the performance of the trained CNN. The goal is to find out whether we could implement the training process in distributed fashion. Therefore, for the naive implementation and later the distributed implementation, we will only include the core layer types, and each type of layers will only be used once as a proof of concept.

Figure 4 shows the work-flow of a most basic CNN implementation. It contains one convolution layer, one rectified linear unit (ReLU) layer, one pooling layer and one fully connected layer. The arrows denote flow of data, and the bracketed symbols along the arrows represent the dimensions of the output and input matrix of each layer at forward and backward run. Convolution and fully connected

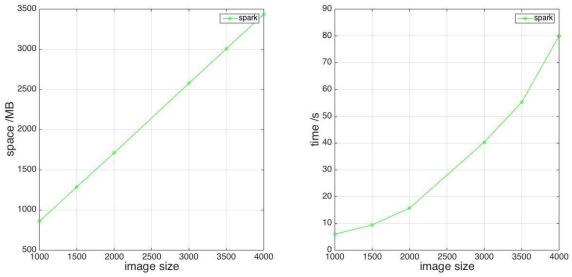

Figure 5 Time Consumption and Memory Usage

layers have parameters K for number of filters, F for filter size, S for filter stride, and P for zero padding. Softmax is the classifier used to calculate loss and gradients. During one iteration, loss and gradients for convolution layer and fully connected layer's activation matrices and bias vectors are outputted to update CNN's hyper-parameters.

We followed the guide outlined in course notes from Stanford's CNN class (CS231n) [9] to implement our naive CNN with Python. All matrix operations are done with NumPy, and Cython and native C code are used to implement stretched column matrix operations im2col and col2im for better performance. All layers provide the normal behaviour of standard CNNs. We will not show the detailed implementation here as all our source code can be found in our GitHub repository. However, we conducted profiling and performance analysis of the four layers to gain insight into where and how to start migrating from naive implementation to distributed implementation.

2.2 Performance Profiling

We run our naive implementation by 30 iterations for different input size and recorded the average time consumption and memory space usage (peak usage in an entire iteration). We used 32 $(5 \times 5 \times 3)$ filters with stride 1 and zero-padding 2 for convolution layer, and (2×2) filter with stride 2 for max pooling. The results are shown in Figure 5.

When number of input images increases, we found that both space consumption and time consumption go up, but in different patterns. The memory consumption goes up linearly as the number of images increases. To train the entire training set of 50000 images for cifar10, one need at least 42GB memory, which is hardly available on commodity hardware. On the other hand, time consumption growth shows a polynomial curve. For 50000 images, it will take approximately 3 hours for one iteration, and normally we need hundreds if not thousands of iterations to train a CNN to reach acceptable accuracy. To make everything manageable, We are aiming to use Spark to distribute the workload. But first we need to determine which layers are consuming the most time and space.

	Conv	ReLU	Pooling	FC
Forward	4.252s	0.504s	2.155s	0.159s
Backward	10.411s	0.458s	3.049s	0.380s

Table 1 Profiling Individual Layer

As shown in Table 1, for both forward and backward propagation, convolution layer and pooling layer are the most time consuming ones. In fact, ReLU and fully connected layers will consume no more memory than their respective

input and they only perform in place matrix manipulation, thus take significantly less computation time. Therefore, to implement our distributed CNN, we would like to first investigate convolution and pooling layers, which are the bottlenecks of the entire system.

Conv	im2col()	np.dot()	col2im()	np.sum()
Forward	0.871s	1.150	N/A	N/A
Backward	0.877s	2.841s+1.849s	0.289s	0.035s

Table 2 Profiling for Functions in Convolution Layer

Pooling	im2col()	np.argmax()	transforamtion	col2im()
Forward	1.647s	0.257s	0.249s	N/A
Backward	1.661s	0.219s	0.480s	0.512s

Table 3 Profiling for Functions in Pooling Layer

Thus, we did another experiment using 2000 images. The detailed performance breakdown for each operation in convolution and pooling layers are shown in Table 2 and Table 3. In convolution layer, *np.dot*() function is the most time consuming part, which is the NumPy function to perform matrix multiplication. In pooling layer, the matrix stretching *im2col*() function is the most time consuming part. Moreover, from our profiling, these operations consume the most memory space, too.

2.3 Distributed Matrix Operations

From what we have seen above, matrix operations, especially multiplication and creation of stretched column matrices in convolution and pooling layers consume the most time and memory space. One may think they are the natural candidates for distributed computation. However, distributing matrices in a cluster introduces significant communication overhead which shadows any performance benefit one may get. We will demonstrate this here with both a more theoretical analysis and an empirical example.

For example, the first matrix multiplication in convolution layer's backward propagation took 2.841s, which was the most time consuming operation of entire iteration. Its input are two matrices of shapes $[(N \times W \times H) \times K]$ and $[K \times (F \times F \times D)]$, where each entry in the matrices is a float number. In the naive matrix multiplication implementation, this may take $N \times W \times H \times K \times F \times F \times D$ float multiplication operations. The two matrices also occupy $((N \times W \times H \times K) + (K \times F \times F \times D)) \times 64bits$ memory to hold all float numbers. A modern Intel Core i7 processor can perform more than 3×10^{11} operations a second. If we distribute the matrix multiplication to run on m cluster nodes, then the computation takes up to $\frac{N \times W \times H \times K \times F \times F \times D}{m \times 3 \times 10^{11}}$ seconds to complete. But we have to at least transfer $\frac{m-1}{m} \times ((N \times W \times H \times K) + (K \times F \times F \times D)) \times 64bits$ data to $m-1$ nodes, and receive back the result matrix of size $N \times W \times H \times F \times F \times D \times 64bits$. Commonly, images have 3 depth channels ($D = 3$). Thus, even for extremely high speed network of 100 Gigabits/second, if we divide the transfer time by computation time with only a single node, we get:

$$\frac{192(\frac{m-1}{m}(N \times W \times H \times K + K \times F \times F \times 3) + m(N \times W \times H \times F \times F \times D))}{N \times W \times N \times K \times F \times F \times 3}$$
$$> \frac{192 \times (\frac{m-1}{m} \times N \times W \times H \times K + m \times N \times W \times H \times F \times F \times 3)}{N \times W' \times H' \times K \times F \times F \times 3}$$
$$= \frac{192 \times (m-1)}{m \times F \times F \times 3} + \frac{192m}{K}$$
$$= (1 - \frac{1}{m}) \times \frac{64}{F \times F} + \frac{192m}{K}$$

As we can see, distributed matrix multiplication will have the best comparable performance when we have only 2 nodes, in such case the ratio above becomes $\frac{64}{F \times F} + \frac{284}{K}$. It shows that with a 100 Gigabits/second network, we could only potentially save time when the ratio is less than 1. But commonly CNNs use 5×5 and 3×3 filters, and 32 or 64 filters. Thus the ratio is far greater than 1.

Moreover, remember the above calculation is based on ideal scenarios that disregard the cost in reduce stage when executors talk to each other to assemble the result matrix, nor other I/O cost, network latency and Spark's scheduling and task management overhead. In any realistic setup, the actual time consumption of distributing matrices surpass this estimation severely. Additionally, *np.dot*() uses a highly-optimized, carefully-tuned BLAS implementation which yields significantly better performance than the naive matrix multiplication approach. In fact, in any CNN with realistic filter size settings, using *np.dot*() locally will outperform distributed approaches.

Furthermore, distributing matrices with Spark doesn't fit Spark's nature, either. Spark discourages creating large sized RDDs, and each task's size is recommended to not surpass 100KB. In our case, the sizes of our two matrices are gigantic, and we would have to slice them into thousands of pieces if we want to use portion of them as Spark tasks. In fact, in Spark context the approach to deal with large data set is to cast it as broadcast variable and share it with all nodes. In another word, these two matrices will be sent to worker nodes entirely, which introduce serious communication overhead.

We wrote a program to show the actual performance metrics. As stated previously, two matrices are shared as broadcast variables. During map stage, we multiply part of the first matrix with the second matrix in each task, and all generated matrices are concatenated together to create the result matrix during the reduce stage. The code can be found in *benchmark\spark_matrix.py* and the results obtained from running on a 3 nodes (each node has 32 cores and 256GB memory) cluster are shown here.

As we can see from Table 4 and Table 5, the communication cost to broadcast the two matrices almost doubles the amount of time to compute the multiplication locally with NumPy, for all input sizes. Theoretically, one may assume as the data is shared among all nodes, the most efficient way is to split the matrices equally to $32 \times 3 = 96$ executors to take advantage of all the available cores on the cluster. However, in reality the overhead to create and launch tasks to executors and reducing the returned results to form the final matrix in Spark is huge. In fact, for all input sizes, launching 32 and more tasks will even worsen the performance.

The same analysis can be applied to all matrix multiplication operations in all layers. Overall, since the communication cost is greater than the computation cost, there is no point to consider distribute matrix multiplication. Additionally, the *im2col*() and *col2im*() functions share similar computation ($N \times W' \times H' \times F \times F \times D$ assignments) vs. communication ($[N \times W \times H \times D]$ and $[(N \times W \times H) \times (F \times F \times D)]$ sized matrices for input and output) characteristics, as they also require to broadcast input matrices and send assembled final matrix back to the driver. The only benefit of distributing matrix operations to a cluster is to reduce memory usage at each node, in case no machine can store the entire matrices in memory by itself. But in our setting where abundant

memory is available, the optimal solution is to run all matrix operations locally. Therefore, we must come up with another approach to distribute CNNs.

Img Size \ Slices	4	8	16	32
2000	38.17s (8.17s)	37.32s (8.03s)	36.73s (7.55s)	41.53s (8.28s)
4000	70.65s (16.56s)	69.48s (16.54s)	73.57s (16.65s)	78.48s (15.65s)
8000	128.62s (33.05s)	123.35s (27.81s)	141.76 (35.90s)	151.04s (31.06s)

Note: time consumption format is total time (broadcasting time)

Table 4 Distributed Multiplication Time Consumption

Image Size	*np.dot*()
2000	4.67s
4000	9.43s
8000	19.09s

Table 5 *np.dot*() Time Consumption

2.4 Distributed Training Process

From the observation above, we realized that if we want to implement distributed CNN with Spark, we have to reduce the data transmitting between each worker, create RDD with smaller task size, and mitigate context switching overhead for distributing and launching tasks across workers and executors. For image classifier training, we noticed that the training data set is fixed for the entire training process, across all iterations, thus can be preloaded to all cluster nodes before iterations take place.

Additionally, we noticed that there are only two places where information calculated from all images are needed. The first is at softmax classifier to calculate loss and gradients for the entire training set from the classifications output from forward propagation. The second one is at the end of each backward propagation, where we need calculated gradients for convolution layer and fully connected layer to update their activation and bias matrices. Note that the classification matrix is of size $[N \times 1]$, the gradient matrix is of size $[N \times C]$ (C is the number of possible classifications - 10 for cifar10), the activation and bias matrices for convolution layer are of sizes $[K \times F \times F \times D]$ and $[K \times 1]$, and the activation and bias matrices for fully connected layer are of sizes $[(W \times H \times D) \times C]$ (remember here W and H and D are dimensions after pooling which is only half of the original values), and $[K \times 1]$. Overall, all these matrices are much smaller space-wise and should not cause much communication overhead.

Inside forward and backward propagation, we can always split matrices by their first dimension, which is the index of images. For example, the input for convolution layer's forward propagation is a matrix of shape $[N \times W \times H \times K]$, or put in another way, N matrices of shape $[W \times H \times D]$ where each one corresponds to one individual image. Its output, on the other hand, is a matrix of shape $[N \times W' \times H' \times K]$, or in another word, N convoluted matrices, where i_{th} matrix of dimension $[W \times H \times D]$ in the input corresponds to the i_{th} matrix of dimension $[W' \times H' \times D]$ in the output. If we slice the input matrix to b pieces at its first dimension (N), and put them into the forward propagation, it will output b matrices, which can be appended together to form the matrix we get by submitting the original input matrix. In fact, the same slicing and concatenation approach can be applied to all four layers' forward propagation functions.

Backward propagation can be done in batches, too, following a slightly different approach. All input and output gradient matrices to and from every layer's back propagation function's first dimensions are N, too, thus they can be split the same way as what we have done in forward propagation. The only difference are the gradient matrices for convolution and fully connected layers' activations and biases. Their values, based on the definition of their calculation, are actually the summation of the gradient matrices generated from each individual image. Take convolution layer for example, its input gradient matrix df from upstream (pooling) is of shape $[N \times W' \times H' \times D]$. The gradient on its activation matrix is computed by $dA = df_{reshaped}^T \times X_C$, where $df_{reshaped}$ is to convert its shape to $[(N \times W' \times H') \times K]$ by flattening its first three dimensions, and X_C is the stretched column matrix used in its forward propagation generated from its forward input matrix X (shape $[N \times W \times H \times D]$). If we treat df and X as a four dimensional arrays, and let df_n denote the n_{th} $[W' \times H' \times D]$ matrix in it, and let $(X_C)_n$ denote the stretched column matrix generated from the n_{th} $[W \times H \times D]$ matrix in then $dA = df_{reshaped}^T \times X_C = \sum_n^N (df_n)_{reshaped}^T \times (X_C)_n$, where $\times$ denotes standard matrix multiplication, and this summation applies to batches of df and X where first dimension represents index of images, too.

We will present the sketches of our algorithm here, while the complete working code can be accessed in our git repository. When we first start training, we broadcast the matrix X for all input images to all worker nodes as a shared variable XB, as it will be reused in all subsequent iterations. Each iteration contains four parts: forward propagation, loss calculation, backward propagation, and parameter updating.

At forward propagation, we create Spark RDD by calling $sc.parallelize(range(B), B)$ where sc is the Spark context from *pyspark* and B is the number of batches we are separating the workload into. We keep the RDD's size same to the number of Spark partitions because the number of batches will not exceed the number of executors and should be distributed evenly. For Spark's map staget, at b_{th} executor's map function, it will fetch the proper portion of images X_b from XB by slicing it with $X_b = XB.value[b*N/B+(b+1)*N/B, :, :, :]$. It then propagates X_b through all four layers' forward propagation function $forward()$. The resulting matrix is a classification matrix Y_b, which is the generated classification for the images in X_b. In reduce stage, all X_b's from executors are concatenated together to form matrix Y, where each entry in Y is the classification of the matching image in X, and Y is collected back to the driver node.

At the driver node, it uses the obtained Y and the loaded labels from cifar 10 dataset to calculate loss and gradients with a softmax classifier. It then pass the calculated gradient matrix to backward propagation to start second round map-reduce.

At backward propagation, Spark RDD is created for a list of tuples, where each tuple contains the batch number and its associated gradients. Similarly to what we have done during forward propagation, at map stage, each executor will calculate all gradient matrices for its share, and the result gradients for convolution layer and fully connected layer's activation matrices and bias vectors are retured. During the reduce stage, the resulting matrices from mapping functions are summed to create the final gradient matrices, which as

shown previously is the same as if we had done back propagation for all images together.

However, one problem arises. Backward propagation not only needs corresponding gradient matrices, it also needs the input matrices from forward propagation at each layer. For example, the input to pooling layer's *backward*() function takes two parameters, one for the gradient matrix, and the other is the $[N \times W \times H \times K]$ matrix it took in from convolution layer during the forward run. Therefore, we must figure out a way to persist data from forward propagation to backward propagation. The most straightforward solution is to send the intermediate result matrices from forward propagation back to driver node in forward map functions. However, this would introduce significant communication overhead as these matrices are way larger than the image matrix and classification matrix themselves. Moreover, this doesn't fit with Spark's design principal where data forwarded to reduce stage should be small. Additionally, we have to broadcast these large matrices before backward propagation takes place hus the communication cost would in fact double. Because Spark can take advantage of data locality, saving the intermediate result at each node might be a better choice.

2.5 Data Sharing with HDFS

We cannot just use each worker node's file system to store data, because in Spark we cannot force the same worker node to work on the same forward batch and backward batch. In case the executor doing back propagation cannot find necessary input data, the entire training process would fail. As a result, we have to pick a distributed storage system, where data can be shared across nodes. But we still want to encourage the same worker node handle the same batch in both forward and backward map-reduce stages. Then we want to consider systems that Spark supports hence it can use locality awareness to assign workers work on data sets they have locally. And we first picked HDFS.

HDFS is the primary distributed storage used by Hadoop applications [25]. Although we are not using Hadoop for map-reduce jobs, it is still supported by Spark. We installed HDFS data nodes onto all the worker nodes of our Spark cluster and set replication factor to one to make sure each executor can only store data to its local data node and no data propagation inside the cluster is enforced thus no network communication would present. Now, in our forward map functions, we store all intermediate matrices from convolution, ReLU, pooling, and fully connected layers to HDFS, with properly assigned names including their matching batch number. Additionally for each batch we save a small file containing only the batch number to a separate directory */batches*. Then before starting back propagation, we first broadcast the entire gradient matrix *df* from softmax classifier to all worker node as shared variable dfB. Now we can create RDD with $sc.wholeTextFiles(path-to-our-hdfs-batch-folder)$ which loads text files from an HDFS directory to create key value pairs for each one of them. Similar to our forward propagation, we create B partitions for the B batch files.

Therefore, inside our backward map functions, the input is the batch file key value pair, where we can get the batch number from the content of the file, and retrieve matching gradient matrix from slicing the broadcast gradient matrix dfB accordingly. Moreover, we can read the intermediate matrices saved in the forward run accordingly with the batch numbers, too. The reduce functions stay the same way as we discussed in previous subsection. Now the backward propagation can be successfully run in distributed fashion. With this design, if Spark can detect data locality correctly, each worker node's executors should only need to read data locally. In case an executor is assigned batch file stored in another cluster node, it can still read data remotely with the help of HDFS, although the performance would be hurt by introducing extra communication cost, but such case should rarely happen.

However, with the usage of HDFS, we observed another problem: file I/O with HDFS is extremely slow, especially for writing data. In fact, we noticed that writing intermediate matrices in forward propagation more than doubled its time consumption and heavily surpassed computational cost. To tackle this problem, we must find a faster data storage that can still be shared across all cluster nodes.

2.6 Replace HDFS with Redis

What is faster than file based storage? The answer is memory. If we can store these gigantic matrices in memory, we should save a lot of I/O time. Redis came to mind as it is the most popular in memory data structure store [22], and provides Python bindings. We considered building a Redis cluster alongside our Spark cluster, but it's rather complicated and its master-slave architecture doesn't really fit our need. Instead, we installed Redis to each of our Spark node independently and enabled their server instances to listen to remote accesses.

We then modified our forward propagation. Now inside each forward map function, we flatten each intermediate matrix and convert the result to binary string by calling $NumPy$'s *tostring*() function. It is then stored to local Redis server with a key including the data type and shape of the matrix so it can be reconstructed later. The key is then returned and stored with the batch number into our batch file in HDFS. Thus for backward propagation, we can still use Spark to take advantage of data locality by constructing RDD with $sc.wholeTextFiles()$.

With our design, each batch file and its associated intermediate matrices should be hosted on the same worker node, although the former inside HDFS and the latter inside Redis. Now each map function in backward propagation can read data from Redis and reconstruct the matrices by using the key parsed from its input batch file's content. In case Spark couldn't assign a task to a worker with locally available data, we write the Redis data reading function in such a way that it will look for data from all worker nodes after trying its local instance by providing a list of all nodes' addresses where the local ip of 127.0.0.1 is always at the top. If data locality is handled correctly by Spark, we should never need to read data in backward map functions from a remote Redis instance. Therefore, we successfully replaced slow file I/O with fast memory I/O, and the results presented in our evaluations section will demonstrate the huge performance gain we obtained by using Redis.

2.7 Limitations

There are still a few limitations of our current approach with Redis. First, we were unable to use Spark-Redis, which is Redis' official library to allow Spark to read and write data from and to Redis directly and take advantage of locality, because it doesn't currently provide Python API bindings.

Hence we are using the HDFS and Redis hybrid method which still suffers some file I/O.

Second, Redis has a string object size limit of 512MB. When we are serializing NumPy arrays to Redis, we store them as binary strings. Thus, all matrices stroed to Redis cannot succeed this size limit, either. As a result, when we are dealing with large number of images for training, we must partition them accordingly so that all intermediate matrices in forward propagation are always under 512MB each. For example, if we train all 50000 images, we cannot partition them into less than 50 batches, otherwise it will be rejected by Redis.

3. EVALUATIONS

The experiment was done on Himrod cluster provided by University of Waterloo for cs848 course. We use three nodes for the experiment in total, where each node has 32 cores of Intel Xeon E5-2670 processors running at 2.6GHZ and 256GB memory.

Our testing cluster is set up as the following: We selected one node to be the master node where both the Spark driver and HDFS namenode are running at. Spark worker and HDFS datanode are started on all three nodes(including the master), which are connected to the master node. For HDFS, replication factor is set to one as we discussed above. For Redis, the server instances are started on all three nodes, with protection mode turned off for remote accessibility. The connection utility function was configured properly to always try the local server instance first before trying the other two nodes. Our source code was packaged as a Python egg file and submitted to Spark from the driver node.

The job is run in client deploy mode with increased Spark driver memory of 32GB and executer memory with 64GB. Only one Spark slave is started at each node, with access to all cores and memory for executor creation. The experimental results are shown below.

3.1 Experimental Results

We compared the performances of three different methods(Naive, Spark with HDFS, Spark with Redis) with various settings(image size and batch size).

3.1.1 Time VS Image Size

First we want to explore the relation between time and image size. As the data size grows, computation time should grow linearly in intuition. However, since our implementation is distributed, we have to take the communication time into consideration. We carried out experiments to measure time consumption under different data size. As figure 6 shows, the computation time grows linearly as the image size increases.

In general, Spark with HDFS has the worst performance. The distributed calculation in this method might reduce calculation time, however, since different workers are communicating with file system. Reading and writing files frequently slows down the whole system. Therefore, even naive CNN implementation on a single machine beats distributed version greatly. Total time consumption in one iteration on Spark-HDFS is at least three times as on a single machine. When deploying Redis on Spark, total time consumption drops quickly. As we can see from the figure, Spark-Redis costs only at most half time as naive method. In order to have a closer at the influence of image size to different prop-

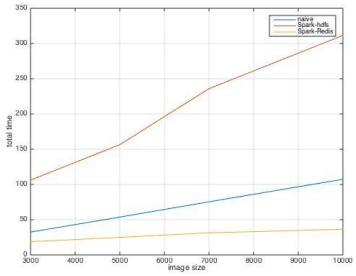

Figure 6 total time consumption

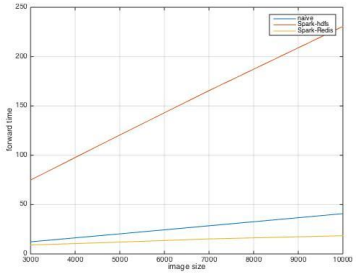

Figure 7 forward calculation time

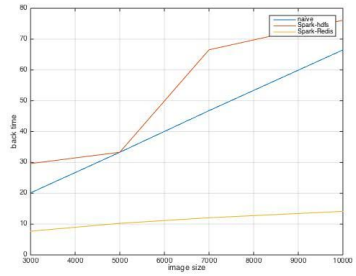

Figure 8 backward propagation time

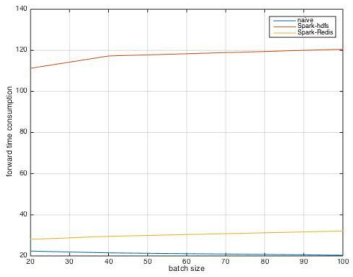

Figure 9 forward time with different batch size

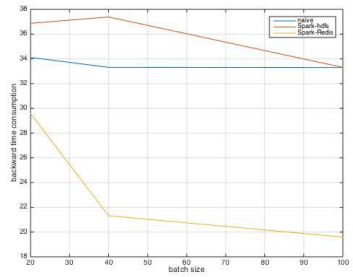

Figure 10 backward time with different batch size

agation steps in the model of Convolutional Neural Network, we also tested on time consumption for them.

As figure 7 suggests, Spark with HDFS costs almost 6-7 times time in forward computation as naive method, while they performs similarly when doing back propagation. And Spark with Radis performs the best both in forward and backward propagation.

Distributed computation reduces the calculation time, but leads to communication overhead. Spark with HDFS has proved the it's not worthy and wise to make CNN model parallel if we cannot reduce the communication cost. Figure 8 helps to have a better understanding how Redis help with Spark. Broadcasting time represents time for communication between different machines. We see it grows in linear for both. And in Redis, the system spends more time in forward broadcasting than backward.

3.1.2 Time VS Batch Size

Batch size also has an meaningful effect on Spark because it reflects the how much the computation distributed. By intuition, we may think the more batches we have, the less time it costs to train the data. However, communication time is still a problem. We need to do some experiments to see how batch size affects on total computation time.

As figure 9 shows, with more batches, the forward calculation time does increases. However the backward computation figure 10 costs less time in Spark with Radis. This is probably because communication costs are no longer overhead for our distributed model training.

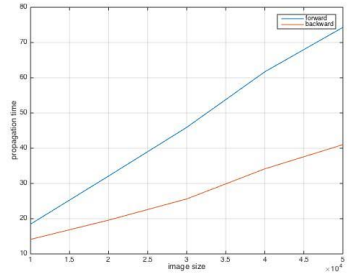

Figure 11 Spark with Redis: time consumption

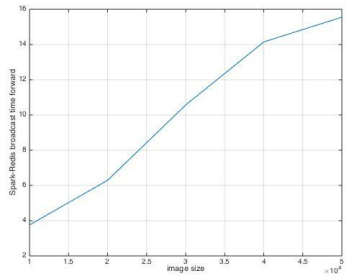

Figure 12 Spark with Redis forward broadcasting

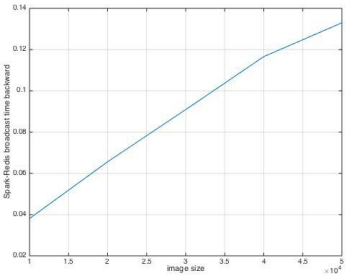

Figure 13 Spark with Redis backward broadcasting

3.2 Compare with TensorFlow

We also compare the performance with TensorFlow. As figure 14 shows, computation time for training CNNs also grows linearly. However, even in the worst case, time consumption for TensorFlow is far below any of our implementation of on Spark.

The reason why TensorFlow is so powerful when training and testing convoltional neural network is, it allows programmers to specify data and machines. There is a mechanism in TensorFlow, where programmers can use primitives to bind node and computations. In this way, it is possible to make specif machine handle a small portion of data during all the training and testing process, which greatly reduces the communication time between workers and thus improves

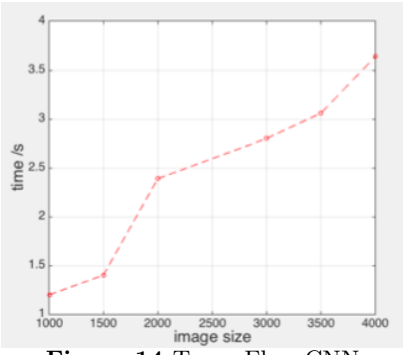

Figure 14 TensorFlow CNN

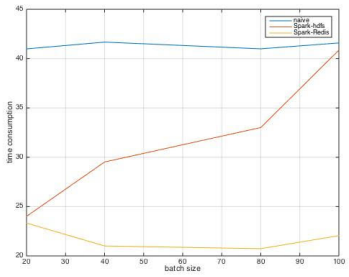

Figure 15 test for 3 methods

overall performances.

3.3 Distributed CNN Deployment

At last we tested the performance for testing process. In testing process, we only need to do one iteration and no backward propagation needed. As figure 15 shows, in one single round Spark with HDFS is better than naive implementation, but worse than Spark with Redis. It's probably because we don't need to do backward propagation when testing the results. Therefore the communication cost caused by exchanging descents for backward calculation is reduced. Considering I/O and communication are bottlenecks for Spark with HDFS, the total time consumption in Spark with HDFS in testing mode become less than naive implementation.

4. CONCLUSION AND FUTURE WORK

We have presented several results from running our naive CNN, Spark(HDFS) and Spark(Redis) CNN implementations. In general, Redis backed Spark CNN shows better results among all three, while HDFS yields worst performance. However, Tensorflow still beats all of our implementations. In conclusion, we found that CNN is not an ideal application for Spark. The reason is, during the training process, forward propagation produces a great number of intermediate results which are needed for backward propagation. Persist and reuse these intermediate results is the main obstacle for distributed implementation on Spark.

In fact, Spark is better designed to handle data summarization tasks such as word-count, as problem data set is distributed beforehand and the amount of data transfered between map-reduce stages and driver-worker nodes is relatively small. This will take advantage of Spark's architecture and give good performance, and explains the reason why majority applications running on Spark are analytical tasks. As we mentioned previously, deployment of trained CNN model, which requires no backward propagation hence discarding all intermediate results, is a good use case for Spark. In practice, another possible application for CNN on Spark is parameter tuning. Specifically, as different parameter setting would yield drastically different accuracy for trained CNN models, we commonly need to run CNN on same data set multiple times with different parameters to find the best combination. For distributed parameter tuning with Spark, one can run individual training process on each worker node and reduce the performance metric back to driver node automatically. Then with some heuristics, the cluster can try to find optimal parameter settings without human intervention. For this use case, except for sending some analytical data back, no map-reduce will happen across nodes during the training process.

Back to our implementations of distributed CNN, we acknowledge that there are some limitations. Because Spark controls the worker scheduling, we cannot force a worker to do both backward and forward process on the same batch. Moreover, while using Spark, the overhead of creating task, launching tasks and scheduling tasks is not neglectable. Furthermore, we only considered cifar10 dataset, but CNN may handle more large datasets with more varieties, which may need more filters, different size of filers, etc. This is not tested on our implementation and it may yield very different results. In the end, the CNN architecture we implemented is a simple one with only four layers. In practice, there may be a sequence of layers including multiple levels of convolution and pooling. Theoretically, we assume for each replicated layer, the analysis will not be influenced. But with more and more intermediate produced as the number of layers increase, there may still have some influence on our analysis.

For the future work, one direction will be considering the polling mechanism, which means, we build a specific distributed system strictly targetting CNN training tasks. The node that complete the map-reduce task in forward stage can poll the driver and retrieve the job for the same batch for backward map-reduce. This way, we can reduce the communication cost, since all the intermediate results can be hold without being transmitting to somewhere else. Moreover, on each node, instead of using Spark's worker-executor design, we can start single worker in multi-threaded fashion, with shared memory, thus no intermediate result need to be moved around. This should reduce all overhead imposed by using Spark. Also, extending our system to work with more complex dataset, or even beyond CNN to other deep neural network might be another research direction.

5. REFERENCES

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

[2] J. Ba, V. Mnih, and K. Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*, 2014.

[3] J. Ba, V. Mnih, and K. Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*, 2014.

[4] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[5] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[6] V. J. Hodge, S. OKeefe, and J. Austin. Hadoop neural network for parallel and distributed feature selection. *Neural Networks*, 78:24–35, 2016.

[7] S. Ji, W. Xu, M. Yang, and K. Yu. 3d convolutional neural networks for human action recognition. *IEEE*

transactions on pattern analysis and machine intelligence, 35(1):221–231, 2013.

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

[9] J. Johnson and A. Karpathy. Cs231n convolutional neural networks for visual recognition.

[10] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[11] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[12] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

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

[14] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[15] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[16] Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective*, 261:276, 1995.

[17] Y. LeCun, K. Kavukcuoglu, C. Farabet, et al. Convolutional networks and applications in vision. In *ISCAS*, pages 253–256, 2010.

[18] S.-C. Lo, S.-L. Lou, J.-S. Lin, M. T. Freedman, M. V. Chien, and S. K. Mun. Artificial convolution neural network techniques and applications for lung nodule detection. *IEEE Transactions on Medical Imaging*, 14(4):711–718, 1995.

[19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[20] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[21] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

[22] Redis Labs. Redis.

[23] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

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

[25] The Apache Software Foundation. Apache hadoop.

[26] M. Yoon and J. Lee. On the paideia education in the age of artificial intelligence-the google deepmind challenge match-. In *Proceedings of the 9th International Interdisciplinary Workshop Series, Advanced Science and Technology Letters Jeju, Korea*, volume 127, pages 169–172, 2016.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical report, Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, 2011.

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[29] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.