Distributed Neural Networks: A Parallel Approach

Tasneem Alowaisheq
Indiana University
107 S Indiana Ave
Bloomington, USA
talowais@indiana.edu

Devendra Singh Dhami
Indiana University
107 S Indiana Ave
Bloomington, USA
ddhami@indiana.edu

ABSTRACT

This paper provides an insight into our work in designing distributed neural networks. In this paper we are implementing a parallel mini-batch back propagation algorithm using Hadoop framework. Our results show that the parallel algorithm converges faster than the sequential algorithm, remaining comparable in the accuracy achieved.

CCS Concepts

•Distributed neural network → Parallelism; •Distributed systems → *Hadoop;* Harp;

1. INTRODUCTION - MOTIVATION

Neural networks are models that were motivated by the structure of the human brain and were built to mimic its functioning. It has several layers of interconnected neurons that form an artificial network, as shown in figure 1, and is typically defined by:

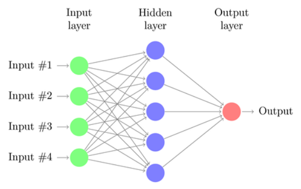

Figure 1: An example neural net

1. Weights: This refers to the weight that connects between the different neurons in the layers.
2. Learning: This refers to the process of learning (by back propagating the error) and calculating the derivatives for updating the weights.
3. Activation Function: This refers to the function that converts the sum of a neuron's weighted input to the output.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Thus, the neural networks can be considered as directed graphs with each layer consisting of a set of neurons. The training data is passed in the input layer; the hidden layer learns an abstraction of the input data and finally we get the output from the output layer. Creating a distributed neural network can be beneficial in several cases:

1. The data is large and cannot be handled by one process.
2. For complex structures the size of a layer becomes too large for one process to handle.

Due to the above defined reasons, having a parallel version of the neural network model becomes an important problem which is well studied in the machine learning and distributed systems community. Next we discuss the related work in section 2 and give a problem definition in section 3. Section 4 defines our proposed method and section 5 and section 6 present the experiments and conclusion respectively.

2. RELATED WORK

In late 90s and early 2000s the neural network architecture consisted of relatively small number of layers and units per layer. This limitation was mainly due to the relatively slow machines which made the training of large networks inefficient. For the MNIST dataset, methods such as SVMs were able to outperform neural networks. Ciresan et al [1] explains how these difficulties effected the parallelization of the neural network and how these can be overcome by training the model on graphics cards. They showed that the advancements in the computational resources facilitate training larger neural networks efficiently thereby helping neural networks to achieve high accuracy on the MNIST data. They showed that large networks with many parameters can be trained to achieve error below 1% in less epochs and thus less time. LeCunn et al [3] present some tricks to perform efficient backpropagation. Some of the tricks briefly are:

1. Incremental stochastic learning performs better than batch learning
2. Shuffling the data after each epoch
3. Input normalization
4. Choosing a better sigmoidal function

They present the advantages and disadvantages of all these tricks and at the end give a few steps to be followed to increase the accuracy and efficiency of the neural network models. Dean et al [2] create a distributed framework to train large sized deep networks on thousands of CPU cores and give an efficient SGD algorithm for these networks. They call this framework *DistBelief* within which they implement two novel methods:

1. Downpour SGD, an asynchronous SGD algorithm, and
2. Sandblaster L-BFGS that uses both data and model parallelism.

They show that this distributed approach can result in speedup of the training time of large models using clusters of machines which take very less time when compared to training on GPUs. They also show that they could train models larger than any other models existing during that time. They trained a neural network of more than 1 billion parameters and showed that this network achieves better performance compared to the best performing deep neural net model on the ImageNet dataset.

3. PROBLEM DEFINITION

In this project our main goal is to design a distributed neural network to get a classification performance with less training time comparable to the sequential neural network. This problem can be approached in two ways.

1. Distributing the network structure horizontally i.e. distributing the neurons in each layer to separate processes. The motivation for this approach is that for complex structures the size of a layer becomes too large for one process to handle.
2. Distributing the dataset to separate processes while keeping the structure of the neural network intact. The motivation for this approach is that the dimensions of the data might be large and thus cannot be handled by one process.

4. PROPOSED METHOD

Our aim is to parallelize the training of the neural network by implementing a parallel mini-batch backpropagation algorithm. We are to trying to handle the second case mentioned in section 3. The parallelization can be done in two ways:

1. Model/Parameter averaging
2. Derivative averaging

As suggested by [5], model averaging works better in practice, especially when using a large mini-batch size. Our method is based on the parallelization using model averaging manner.
The method used for training the neural network is mini-batch gradient descent.

Definition 1. In mini batch gradient descent, the gradient is computed against more than one training example and less than for the complete training examples i.e. it's in between stochastic and batch gradient descent.

When we have large data, using a single process will slow down the training of the model considerably. To speed this process up one can randomly sample mini batches with replacement from the training data and distribute the learning over different processes. Each process will train on its local mini-batch data that consists of the feature vectors $\mathbf{X}$ and its corresponding targets $\mathbf{Y}$, and the local weights will be updated accordingly. The weights are a vector of matrices where each matrix holds the weight connections from one layer to the next. These weight matrices are flattened and concatenated to get the final representation of the weights to be distributed.
These local weights, across all the processes, will then be combined by using *Allreduce*, which is called for every five epochs to ensure that the weights are updated across tasks more frequently, and then averaged to obtain the final weights. These weights are the final model used for prediction on the test data. The prediction is not an expensive operation and thus can be handled by the master process. This process is repeated on ten random splits of training and testing data to ensure that we have a good estimate of the generalization error. This process can be shown in figure 2 where every batch represents a mini batch where the intersection of each mini batch is not ϕ.

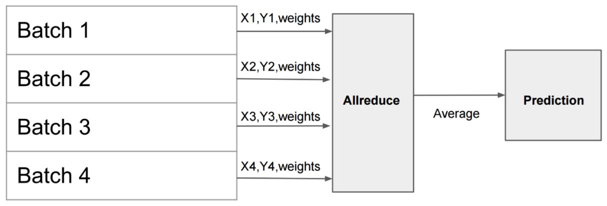

Figure 2: The proposed method.

Our proposed algorithm is shown below.

```
Data: The input data and corresponding labels
Result: List of accuracies across N splits
for the number of splits, N do
    Divide the data randomly into training and testing;
    Initialize the weights;
    for (number of iterations / 5) do
        Randomly sample the mini batch from training;
        Train for 5 epochs to update the weights locally;
        Allreduce;
        Average the models;
    end
    if master then
        Predict on test and report accuracy;
    end
end
```

Algorithm 1: Parallel neural network

We make use of mini-batch gradient descent here as it has been shown to be effective for the optimization of the objective function [4]. The size of mini batch should be as large as possible for effective training. A simplied algorithm for illustrating mini-batch gradient descent is shown in algorithm 2.

```
Data: The training data
Result: Optimized parameters
for number of epochs do
    shuffle the data;
    for every mini batch do
        Update the parameters;
    end
end
```

Algorithm 2: Mini batch gradient descent

5. EXPERIMENTS

We report the results for ten randomly sampled training and test splits for both the sequential code and the parallel code. The task is a digit classification task that consists of the grayscale pixels intensities of the image as a feature vector and the target is a vector of size 10 where each element is one for the corresponding digit and zero elsewhere. The data set consists of 5000 examples. For fair comparison, we made sure that the splits across different experiments are the same and random weight initialization is the same by setting the seed to the split number. The weights are initialized from a uniform distribution in range [0,1]. For generating the splits, 70% is used for training purposes and 30% for testing. For the following experiments, the mini batch size is 90% of the training data.
We utilized the open source sequential code from [6] to implement our parallel method. We changed the sequential code to a mini-batch version to use it as a baseline for comparison. Also we changed the weight initialization for the sequential code from normal to uniform distribution for the sake of fair comparison. The parameters for both methods are:

1. regularization parameter, $\lambda = 0.6987$.
2. learning rate, learned using line search.

We run the sequential code on the local machine and the parallel version on the Juliet cluster for 2, 4 and 8 map tasks using two nodes. Each experiment yields ten accuracies we average them to get the final accuracy. The average accuracies for both are presented in table 1 below.

Table 1: Average accuracies for sequential and parallel NN

Number of Epochs	Sequential	Parallel
100	80.67%	84.06%
200	86.89%	87.28%
250	88.63%	86.45%
500	89.9%	88.93%

Note that we present only one accuracy for the parallel algorithm since the different number of mappers yield the same accuracy which is expected as the random sampling is fixed across experiments.
As figure 3 shows, when having enough number of epochs both the parallel and the sequential methods yield about the same accuracy.
The convergence times for the sequential and parallel neural network is shown in table 2.
In terms of convergence, parallel methods converges faster than the sequential one as shown in figure 4, it seems that the gap in convergence have an exponential property in terms of the number of epochs.

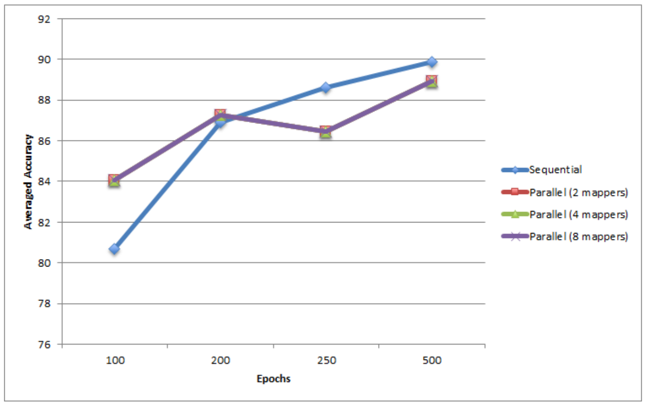

Figure 3: Average accuracies for the neural network.

Table 2: Average convergance time for sequential and parallel NN

Epochs	Sequential	Parallel(2)	Parallel(4)	Parallel(8)
100	244 sec	211 sec	209 sec	217 sec
200	494 sec	337 sec	318 sec	345 sec
250	622 sec	406 sec	398 sec	431 sec
500	1225 sec	790 sec	746 sec	807 sec

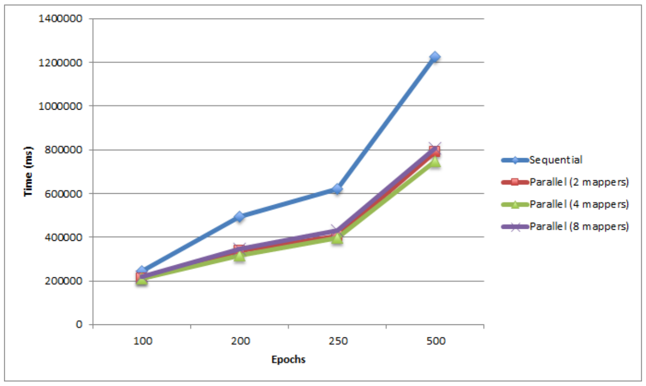

Figure 4: Time taken for convergence by the neural network.

Figure 5 show the time when using different number of mappers. All experiments yielded about the same time for different epochs.
We needed to change some of the default Hadoop settings in order for our method to work: For experiments with a large number of epochs some jobs failed due to exceeding the task timeout. We increased the *mapred.task.timeout* for the job configuration in the *MapCollective*. Also when increasing the number of mappers some jobs failed so we changed the memory settings in the *mapred-site.xml* and *yarn-site.xml*.

6. CONCLUSIONS

In this paper we presented a parallel implementation for the mini-batch neural network. We demonstrated our method

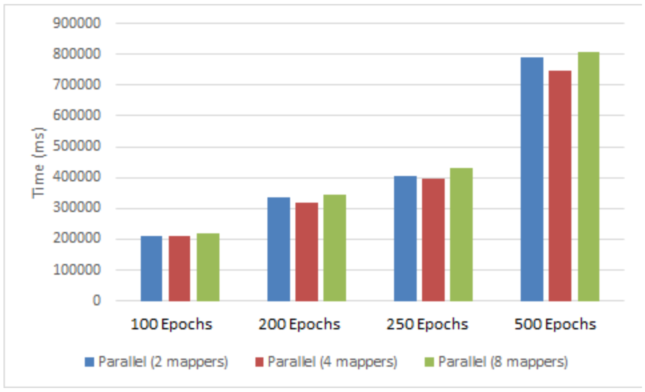

Figure 5: Time taken for convergence by the parallel neural network for different number of mappers.

on a digit classification task. Our results show that the parallel version is comparable to the sequential one in terms of accuracy and converged significantly faster than the sequential method.

7. REFERENCES

[1] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[2] 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.

[3] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[4] F. Seide, G. Li, and D. Yu. Conversational speech transcription using context-dependent deep neural networks. In *Interspeech*, pages 437–440, 2011.

[5] H. Su and H. Chen. Experiments on parallel training of deep neural network using model averaging. *arXiv preprint arXiv:1507.01239*, 2015.

[6] D. Vincent. Neural network, 2011.

APPENDIX

A. ACKNOWLEDGMENTS

We would like to thank Bo Peng for his feedback on our method and Raksha Kumaraswamy for helping us in setting up the juliet cluster.