ECE751 Final Report: Analyzing Variations in Distributed Deep Neural Network Learning

Husayn Kara & Vineel Nagisetty
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
{hkara,vnagiset}@uwaterloo.ca

***Abstract*—Recent advances in scaling Deep Neural Networks (DNNs) have enabled them to work successfully on larger datasets and more complicated tasks. These advances were largely in part due to hardware optimizations - including modifying GPUs to train DNNs - and software optimizations as well as the availability of open-source frameworks that could abstract training and deploying of large DNNs. However, there is a limit to the improvements these approaches can provide - and hence there is a need for approaches that can help scale DNNs to even larger datasets and more complicated tasks such as autonomous vehicle controllers and voice operated assistants. As a consequence, a new class of distributed DNN training algorithms, such as Downpour Stochastic Gradient Descent (Downpour-SGD), have been introduced and have shown the potential to train DNNs of greater size and at a greater speed. This project implements[1] an asynchronous version of Downpour-SGD using Thrift's RPC framework and Pytorch modules, lists several optimizations to reduce message sizes amongst the servers and also introduces novel variations to key parts of the Downpour-SGD algorithm - namely in the data splitting and parameter aggregation methods. This implementation is then tested to observe the effects of the number of worker nodes, as well the novel variations introduced, on the resulting training time and accuracy of the model. Notably, the model trained using ten worker nodes has been shown to result in an improvement of 28.86% in time while lowering accuracy by 6.5%, as compared to standard DNN training. This shows that while there is a trade-off between time required for training and accuracy of the resulting model, distributed DNN training can provide benefits over standard DNN training in certain settings.**

I. Introduction

Deep Neural Networks (DNN's) is a deep learning architecture that has produced state-of-the art results and performance in complex tasks such as computer vision [1], natural language processing [2] and predictive analytics [3]. DNN's make use of many hidden layers in the learning networks, and can be quite deep, as their name suggests. Such complex structures, aimed at mimicking the human brain can include a vast number of neurons each assigned weights that vary throughout the learning process. Such a concept has been around for decades, however recent advances in hardware have led to new breakthroughs. However as it becomes more difficult to rely solely on hardware improvements and as the amount of training data and neural network complexity only show signs of increasing, new strategies to train efficiently are needed.

DNN's have shown to benefit greatly from a distributed approach to their training and testing. In fact, it can be considered embarrassingly parallel, a problem entailing minimal work to parallelize [4]. This has been demonstrated by the rapid adoption of GPUs to train and test Neural Networks [5], allowing for faster processing of matrices, which form the basis of many artificial intelligence algorithms [6]. There has also been increased development of special hardware accelerators [5] and reconfigurable computing devices, such as ASICs [7] and FPGAs [8] respectively, to further improve the efficiency of neural networks.

At a higher level, a further layer of parallelization can be achieved through the use of software to manage and run neural networks across distributed systems involving several nodes. Paired with effective algorithms, this would allow for scalable training and testing of neural networks. This is particularly relevant today as we are witnessing the impressive improvements brought about by Deep Neural Networks with an increasingly greater number of parameters, with the state-of-the-art GPT-3 model having over 175 billion parameters [9]. At this scale, new and improved distributed algorithms are most welcome and fortunately exist.

A. Our Contributions

This paper makes use of remote procedure calls to create a scalable and tunable platform in which to test and train deep neural networks. Through the use of various data splitting methods and aggregation algorithms, various experiments testing effects of different parameter variations on model accuracy and speed are explored, thus providing greater insight when applying a distributed systems approach to improve deep learning efficiency. More specifically, we make the following contributions:

1) Our key contribution is the implementation of an asynchronous distributed DNN training algorithm known as Downpour Stochastic Gradient Descent (Downpour-SGD). We implement this using Apache Thrift, an open source remote procedure call framework, as well as Pytorch, an open source deep neural network training framework. We note that, to the best of our knowledge, there are no implementations of Downpour-SGD using

[1]The code of our implementation can be found at: https://github.com/vin-nag/distributedLearning

this combination of frameworks, making our implementation unique. We further ran experiments to test the effect of the number of worker nodes on the accuracy and the time taken by our algorithm. We show that a model that is trained on our implementation of Downpour-SGD using ten worker nodes shows an improvement in training time of 28.86% while resulting in a decrease in accuracy of 6.5% over a model trained using standard DNN training (Refer to sections V and VI).

2) We implement several novel modifications to Downpour-SGD, varying the data splitting and parameter update aggregation. We ran experiments to test the efficacy of these variations and show that the equal data splitting method as well as the weighted parameter aggregation method result in an increase in accuracy over standard Downpour-SGD (Refer to sections III and VI).
3) We discuss the trade-offs encountered as well as list potential future research and implementation directions for distributed DNN training using Thrift IDL and communication protocol (Refer to sections VII and VIII).

II. Related Work

Currently there are various algorithms used in the context of distributed DNN training. Once method is through Downpour Stochastic Gradient Descent (Downpour-SGD). This can be broken down into two forms, synchronous and asynchronous. Synchronous Downpour-SGD uses a tighter coupling of Nodes, and is thus not as parallelizable as asynchronous Downpour-SGD. In order to increase efficiency in synchronous Downpour-SGD, larger batch sizes have been used in training. However, the batch size on which the model is trained on does play an important factor in terms of the trade-off between accuracy and efficiency. In particular, larger batch sizing, while offering increased speed, may lead to an over generalization by the model when compared to training on smaller batch sizes used in training [10].

While asynchronous Downpour-SGD does allow for a greater degree of parallelism, it can also contribute to marginally worse accuracy and stability. Fortunately, notable DNN training algorithms have been proposed that leverage distributed computing techniques such as Hogwild (which has shown near optimal rates of convergence) [11]. These algorithms leverage an asynchronous approach to stochastic gradient descent (SGD) and have successfully been used to distribute training and inference of massively sized DNNs using large-scale clusters of machines which would not have been possible on a single machine (even with a lot of available GPU).

Another important factor of distributed training involves the communication between nodes of the system. Thanks to advances in distributed frameworks, such as Hadoop and Google File System, as well as improvements in peer-to-peer architectures, machine learning platforms such as TensorFlow and PyTorch can avail of these paradigms to increase efficiency. However the prevailing balance between computation and communication demands still require flexibility in overall system design [10].

Google has also come up with a framework, called DistBelief, with the objective of being able to use clusters consisting of thousands of machines to train large models consisting of billions of parameters. The DistBelief framework included the development of Downpour-SGD, while supporting numerous model replicas. Google has managed to train DNNs considerably larger (30x) than previously reported in literature and also achieve state-of-the art performance on image classification using the crowd sourced ImageNet dataset containing 16 million images [12].

Work in distributed deep learning has also been done by taking advantage of robust and efficient distributed tools, such as Apache Spark. By applying cascade learning [13] to address the problem of imbalanced classes. In cascade learning, knowledge gained from a previous model is applied to another model, where there is insufficient labeled data to properly train the model. However, by using Spark's MLLib library for cascade learning, and creating an ML pipeline build on DataFrames, computation time can be reduced by almost half [13].

There are several surveys that provide state-of-the-art algorithms and optimizations of distributed DNN training [6], [10], [12], [14], [15]. Further, there are several papers describing applications of distributed DNN training on specific frameworks [13], [15]–[18]. However, most of the other asynchronous algorithms are variations of the two main ones: namely Downpour-SGD [12] and Hogwild [11], and differ from our approach. This is a recent and active area of research and we believe we have honed in on a specific area, relating to distributed systems, that merits further exploration.

III. Our Approach

Algorithm 1: Downpour SGD [12]: Manager Node

```
Data: Dataset $D$, Worker Nodes $N$, Training Epochs
      $T$, DNN Model Architecture $M$
Result: Trained Model $M'$
1  randomly initialize model $M$;
2  foreach training epoch $t \in T$ do
3      split data D into N blocks, creating blocks
         $\sum_{n=1}^{N} d_n$;
4      foreach Worker Node $n \in N$ do
5          send $d_n$, $M$ to worker Node $n$ asynchronously;
6      end
7      receive parameter updates $p_n$ from all worker
         nodes;
8      aggregate parameter updates
         $P' = (1/N) * \sum_{n=1}^{N} p_n$;
9      update $M$ using $P'$ and set it as $M'$;
10 end
11 return $M'$
```

Algorithms 1 and 2 cover the Downpour-SGD algorithm - particularly the steps taken by the Manager and Worker nodes respectively. The Manager node receives the dataset D, the number, and addresses of the Worker Nodes N as well as the number of training epochs T. It splits the data into N parts,

Algorithm 2: Downpour SGD [12]: Worker Node

```
Data: Data Batch $d$, DNN Model $M$, Manager Node
      Location $A$
Result: Parameter Update $p$
1 train $M$ using $d$;
2 compute parameter update $p$ using stochastic gradient
    descent;
3 return $p$ back to the Manager Node $A$
```

and passes the model parameters as well as the split data to each of the worker nodes and waits for their results. Once each worker node sends back the parameter updates, the manager node aggregates the parameters and modifies its copy of M using it. This continues for the required number of training iterations. By contrast, each worker node receives a part of the data d along with the DNN model M from the manager. They use this to train their copy of the model, and apply standard SGD to calculate parameter updates p and send this back to the manager node.

In this research, we focus on the efficacy of asynchronous SGD algorithms similar to the aforementioned Downpour SGD. It is clear that using a greater number of worker nodes will result in a decrease in the time required to train a given DNN model utilizing asynchronous SGD algorithms. However, we suspect that this may result in a lower model accuracy, since DNN training is inherently sequential and parallelizing it may reduce its efficacy. In other words, we believe that there may be an inverse relationship between training-time and model accuracy in the asynchronous Downpour-SGD algorithm setting. Therefore, we analyzed the effect of the number of worker nodes used in training modest sized DNNs using asynchronous Downpour-SGD algorithms on the training time required and resulting model accuracy. Further, we implement several novel variations of the Downpour-SGD algorithm. Notably, we wish to investigate different ways of splitting the data (Algorithm 1 line 3) as well as different methods of aggregating the parameter updates (Algorithm 1 line 8) and view how they effect the training time and model accuracy.

A. Data Splitting

1) In the base approach, we will split the data randomly. This is the standard approach in the Downpour SGD algorithm. Here, we expect the ratio of classes in the dataset will be similar to the ratio of classes in each split and this will serve as a good default method. We refer to this approach as “random”.
2) In the second approach, we plan to split the data based on the classes and have the same number of worker nodes as the number of classes in the dataset. This way, each worker node trains on one distinct class. This is akin to having multiple single class classifiers. We refer to this approach as “class”.
3) In the third and final approach, we will split the data so that there is the same amount of data for every class in a given split. We refer to this approach as “equal”.

B. Parameter Update Aggregation

1) In the base approach, we will simply average the parameter updates. This is the standard way of aggregation in the Downpour SGD algorithm. We refer to this approach as “average”.
2) In the second and final approach, we will scale the weight of each parameter update to the loss value calculated for the given worker node. This will enable us to give higher weight to the parameter updates of worker nodes that resulted in a higher loss - allowing us to focus on their feedback more. We refer to this approach as “weighted”.

IV. Implementation

As mentioned in the introduction, this project uses Apache Thrift’s RPC framework. While there are higher level frameworks such as PySpark, which allow for distributed approach to neural network learning, this method is not best suited to conducting research experiments, since it is a framework that seeks to maximize performance and scalability through the use of a DataFrames, where users can configure various machine learning pipelines. However, for the purpose of this project, too much of the system configuration was inaccessible and limited the ability of node network customization and message passing. A bare bones approach would have been to create our own communication protocol from scratch on which we could have much greater control of the neural network learning in a distributed environment. However, this approach was not practical given the time constraints and our research objectives. We therefore settled on using an open source RPC framework, that would allow enough flexibility, while not requiring the creation of a new communication protocol. Apache Thrift offers support in various programming languages, including python, and was thus able to be used with PyTorch fairly easily. However, due to the limited Base Types available through Apache Thrift’s IDL, some work (involving Structs, part of Apache Thrift’s Type System) was required to pass the necessary parameters between nodes effectively. Nevertheless, Thrift RPCs are far more efficient than a self-built solution and thus allowed us to focus our efforts more on the system architecture for distributed DNN training. By reducing I/O overhead, our results would better highlight the effects of parallelization of the neural network learning through our proposed methods.

Training DNNs is both compute and I/O intensive because it requires a lot of computation as well as huge amounts of data. While distributed DNN training offsets the compute load by splitting it into several smaller parts for each worker node, it also introduces more I/O - specifically communication work due to the requirement of sending and receiving data and model parameters between manager and worker nodes. Many state-of-the-art DNNs require gigabytes (or more) data, and so this can severely hinder the effectiveness of a distributed DNN training algorithm. In order to improve our implementation, we make several optimizations to reduce the size of messages. Namely:

1) The dataset is saved in a shared directory, and the manager node only sends the indices of the data to use for each worker node - significantly reducing the size of the message from the manager to the worker node.
2) The updated model (from the previous epoch) is in a shared directory, and the manager node sends the file name to each worker node - further reducing the size of the message from the manager to the worker node.
3) The parameter updates by each worker node is also saved in a shared directory, and each worker node only sends the corresponding file name to the manager - reducing the size of the message from the worker to the manager node.

Note that while this decreases the message size between the worker and the manager nodes, it increases the number of reads and writes to file for both the manager and worker nodes. However, we believe this trade-off is still beneficial.

A. Implementation Details

We implemented[2] our project using Pytorch 1.7 [19], an open source machine learning framework popular in deep learning research, and Thrift IDL 0.13.0 [20], an open source remote procedure call framework.

Pytorch has the `state_dict` object that stores the parameters of a neural network as a map. In order to efficiently pass the aggregated model from the manager to the workers, as well as the parameter updates from the workers to the manager, we store this information using state_dict[3] to a file in a shared folder and passes the file names as strings between the communicating parties. As previously mentioned, the manager computes and sends the list of indices to each worker node instead of the entire data to reduce the size of messages passed. In order to achieve this, we use the `Subset` class in PyTorch for each worker node to create the dataset based on the given indices.

V. Experiments

Algorithm 3: Pseudocode of our experiment to observe the effect of the number of worker nodes to the efficacy of the ral network training algorithm.

```
1 foreach number of worker nodes $[1, 3, 7, 10]$ do
2     Run Experiment using default data splitting and
        parameter aggregate methods;
3     Record training time and model accuracy on test
        data;
4 end
5 return Results
```

Our experiments aimed at investigating how the number of nodes (1) and the variation in data splitting and parameter aggregation (2) affect the efficacy of the distributed DNN training algorithm. To that end, we designed two sets of experiments, given in Algorithms 5 and 7 respectively. For each experiment, we record the time taken to train, as well as the model accuracy on the test data, which we collectively refer to as the efficacy of the algorithm.

Algorithm 4: Pseudocode of our experiment to observe the effect of the algorithmic variations in data splitting and parameter aggregation methods to the efficacy of the distributed DNN training algorithm.

```
1 foreach variations in data splitting methods (random,
    equal and class) do
2     foreach variations in parameter update
        aggregations (average and weighted) do
3         Run Experiment using ten worker nodes;
4         Record training time and model accuracy on
            test data;
5     end
6 end
7 return Results
```

Layer	Size
Convolutional	kernel of (3 x 32)
Convolutional	kernel of (32 x 64)
Dense	(9216, 128)
Dense	(128, 10)

TABLE I
Model Architecture of DNN used in the experiments to train on the MNIST dataset.

A. Experimental Setup

We opted to use the MNIST [21] dataset, a collection of 70,000 28x28 gray-scale images of handwritten digits popularly used in deep learning research. The dataset was normalized so that each pixel was in the range [0,1] prior to training. The architecture of the DNN used is given in Table I. We use the `relu` activation function after each layer, except for the last one where we use `soft-max` activation. Further, we use a dropout of 0.25 after each layer during training for better regularization.

The batch size was selected to be 64 in our experiments. We trained the model for 5 epochs. We used the Stochastic Gradient Descent optimizer with a learning rate of 0.01 and a momentum of 0.5. Our experiments were run on one local machine which uses an intel i6550 chip that consists of 4 cores (8 using hyper-threading). Note that we ran initial set of experiments on the ECE server using manager, worker and clients located on different ECE servers. However, the final set of experiments shown below were run on a local system for greater control and admin rights, as well as to reduce variance in network traffic.

B. Evaluation Criteria

In order to quantify the efficacy, we look at two key metrics: **the time required for training**, and **the accuracy of the resulting model** on unseen data (test data). We ran two sets of experiments. Namely:

[2]The code of our implementation can be found at: https://github.com/vinnag/distributedLearning

[3]A state_dict is an object in PyTorch that acts a python dictionary map, allowing for easier data accessibility

1) **Experiment 1:** Trained the DNN Model on MNIST using different combination of worker nodes (1, 3, 7, 10) while using the default data splitting and parameter aggregation methods.
2) **Experiment 2:** Training the DNN Model on MNIST using different combinations of data splitting and parameter aggregation methods, while keeping the number of nodes constant (10).

For each experiment, we ran 10 trials and saved the average of the results in order to reduce the effects of variance. After each training epoch, once the parameters were aggregated, we recorded the accuracy of the model on the test data. Note that this was done only using the manager node and so we do not factor the time taken for this test into the time required for training, as this particularly process not parallelized. During the training process, we save the accuracy at each epoch as well as the total time required for training.

VI. Results

A. Effect of the Number of Worker Nodes

As previously mentioned in Section V, our first set of experiment investigates the effect of the number of worker nodes on the model accuracy and time required for training. The results of average model accuracy are found in Figure 1. Note that X axis represents the number of training epochs (up to 5), Y axis represents the accuracy (%) of the model on test data while each line represents a different number of worker nodes (1, 3, 7, 10) tested. We consider the experiment with one worker node as our base case, since this is the same as standard (non-distributed) DNN training. This base case experiment showed the greatest accuracy, starting with 95.41% average accuracy after epoch 1, and going up to 98.64% average accuracy after epoch 5. By contrast, the experiment with ten worker nodes produced an average accuracy of 85.37% after epoch 1 and 92.65% after epoch 5. There seems to be a clear trend, and inverse relationship, where the experiments with a higher number of worker nodes ended up with a lower accuracy.For instance, the base case, with one worker node, showed an improvement in model accuracy of 6.5% when compared with the use of ten worker nodes.

The results of the average training time are found in Figure 2. Note that Y axis represents the amount of time required on average to train for 5 epochs, measured in seconds, and each bar represents a different experiment with a unique set of worker nodes (1, 3, 7, 10). The experiment with one worker node required 396.64 seconds, while the one with ten worker nodes required 282.17 seconds. There seems to be a clear pattern here as well, and so experiments with a higher number of worker nodes seem to take less amount of time to train. The experiment with ten worker nodes showed an improvement of 28.86% over one worker node in the time required for training.

1) Discussion: Our experiments show that there is a trade-off between model efficacy (accuracy on test data) and time required for training in the distributed DNN setting - which is to be expected. Distributed algorithms parallelize cpu intensive operations (such as DNN training) so that they require less

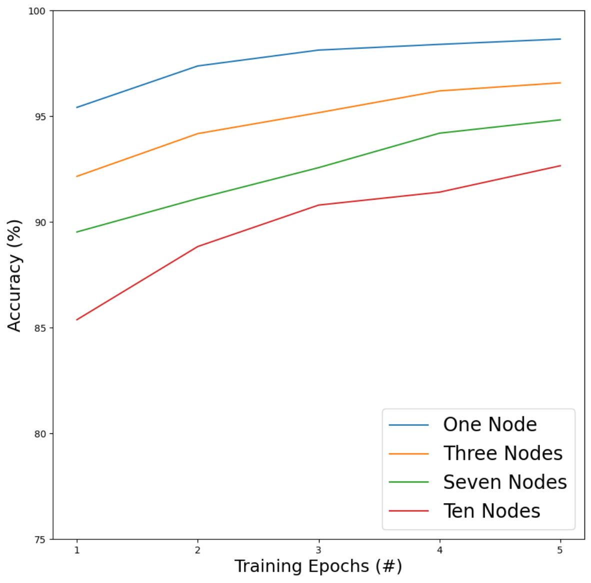

Fig. 1. Average accuracy (n=10 runs) versus training epochs using different number of worker nodes.

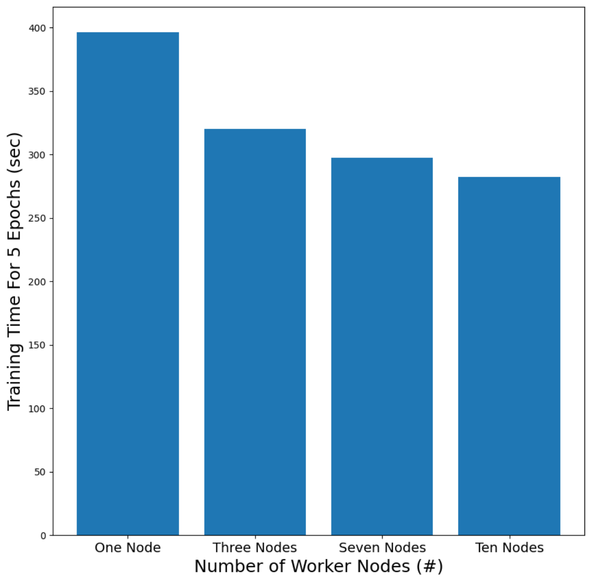

Fig. 2. Average time taken to train 5 epochs (n=10 runs) using different number of worker nodes.

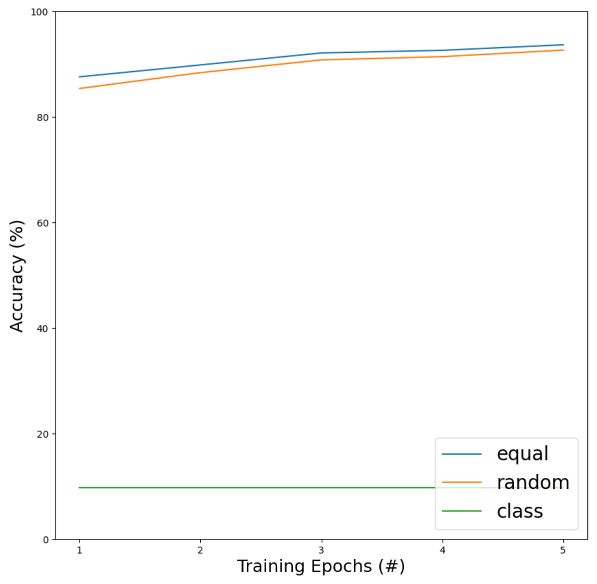

Fig. 3. Average accuracy (n=10 runs) for aggregation method ”Average”.

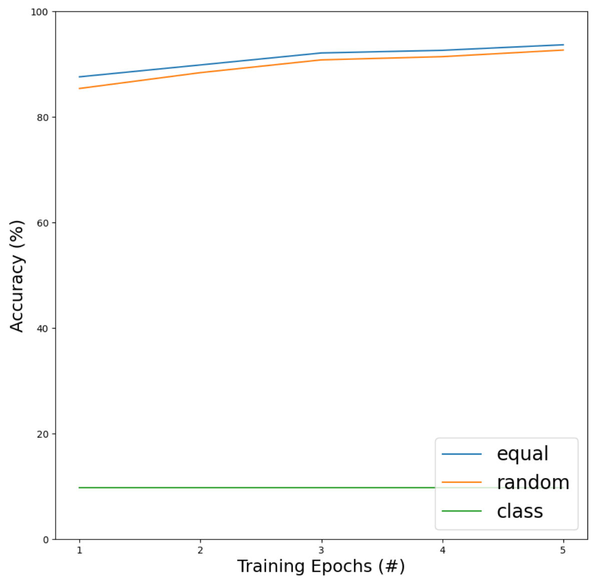

Fig. 4. Average accuracy (n=10 runs) for aggregation method ”Weighted”.

time to run. However, this results in a loss of model accuracy. We believe this is because a model trained using standard DNN training gets updated at a higher frequency than one using distributed training (proportional to the number of worker nodes). Each parameter update helps the model improve in the task, by helping it move close to a (local) minima of the objective function and therefore minimize loss.

One interesting experiment would be to run this experiment for a far greater number of epochs, and observe whether distributed training with higher number of worker nodes will ever ‘catch up’ to the standard training. We suspect this to be true, especially in scenarios where the training data is shuffled after each epoch.

Hardware constraints did pose a limitation to our experiments. Our experiments were run on a system with 4 cores (or 8 with hyper-threading). This limits the total number of worker nodes that can effectively be used to 8, and we believe this is the reason why there is a smaller reduction in the time required for training for ten worker nodes from seven, as compared to the larger reduction from one worker node to three. We believe that, were it not for this hardware limitation, there would be a much better improvement in the time required for training for ten worker nodes.

B. *Effect of Algorithmic Variations*

As mentioned earlier in Section IV, we implemented several novel variations to the data splitting and parameter aggregation methods of Downpour-SGD. Our second set of experiments investigated the effect of these variations by running all combinations of data splitting and aggregation methods implemented using only ten worker nodes.

1) Effect of Variation in Data Splitting: The results of model accuracy obtained by varying the data splitting methods is given in Figures 3 and 4. For each figure, the X axis represents the number of training epochs (1 through 5) while the Y axis represents the accuracy of the aggregated model on the test data (as %). Each figure shows all three data splitting methods as different lines. For the random data splitting method using the average aggregation method (Figure 3), the average accuracy is initially 85.37% after 1 epoch and goes up to 92.65% after 5 epochs. In the same figure, we can see that the experiment using the equal data splitting method produces an average accuracy of 87.58% after 1 epoch and 93.51% after 5 epochs - showing an improvement of 2.6% over the random data splitting method after 1 epoch and an improvement of 1% after 5 epochs. While this is not substantial by any means, it does show an improvement. This pattern is also found in the three data splitting methods using the weighted aggregation method (Figure 4), where the model trained using the equal data splitting method was lightly more accurate than the one trained using the random data splitting method, resulting in an improvement of 2.9% after epoch 1 and 0.7% after epoch 5. On the other hand, the class data splitting method produced a (very-low) accuracy of 9.8% at all epochs.

2) Discussion: We expected class data splitting method to not perform too well, but were quite surprised at the abysmal accuracy score of under 10%. We verified that our code works fine. We investigated further and found that the resulting aggregated model (using either parameter aggregation method) always outputted the same label (here label 0). We believe this is due to an inherent property of the DNN model in learning the MNIST dataset. Our best guess is that each of the worker models very quickly learnt to output the respective class it was given (since it was given data all of the same class) and did

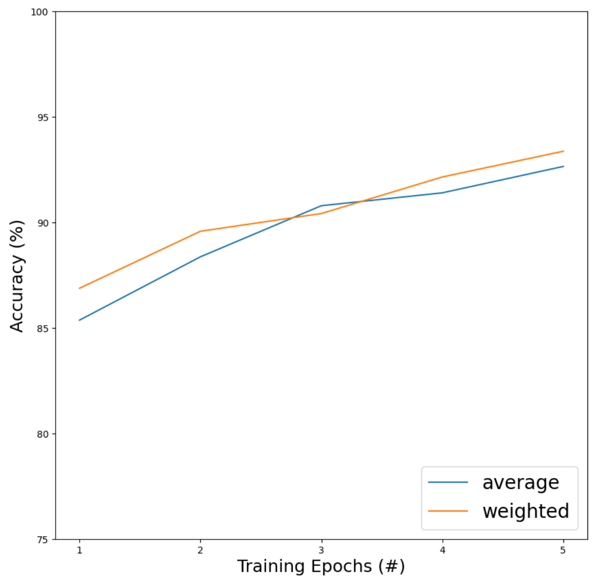

Fig. 5. Average accuracy (n=10 runs) for data splitting method ”Random”.

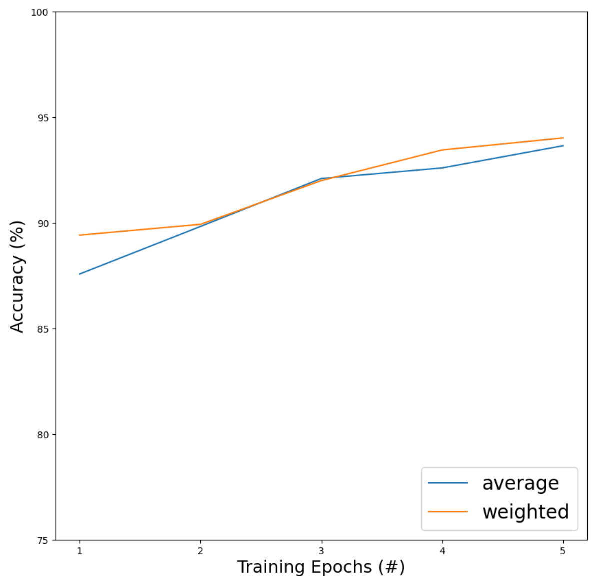

Fig. 6. Average accuracy (n=10 runs) for data splitting method ”Equal”.

not learn the data further than that, and therefore aggregating these poorly learnt models does not provide a good quality model.

On the other hand, the equal data splitting method resulted in a slight improvement in accuracy over the random method. We believe this is because of the potential class imbalance in the data provided to the worker nodes which is not a factor in the equal data splitting method.

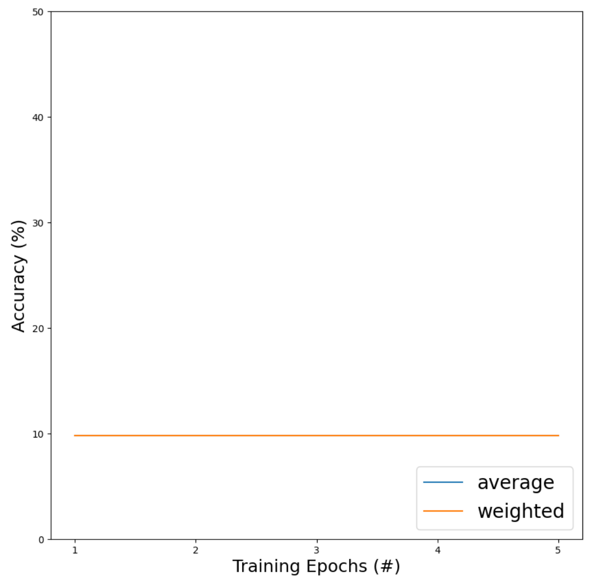

Fig. 7. Average accuracy (n=10 runs) for data splitting method ”Class”.

3) Effect of Variation in Parameter Aggregation: The results of the model accuracy by varying the parameter aggregation methods in given in Figures 5, 6 and 7. These graphs show the same data as the previous ones but in a different form to better view the effect of parameter aggregation methods. Once again, the X axis represents the number of training epochs (1 through 5) while Y axis represents the accuracy of the aggregated model on the test data (as %). We observe that the weighted aggregation method slightly outperforms the average method for both random and equal data splitting methods. This improvement is very slight, and is only 0.4% for equal and 0.7% for random data splitting after 5 epochs.

4) Discussion: We believe that the weighted parameter aggregation method shows an improvement over the average one because it focuses more on the parameter updates which produce a higher loss. Because parameter updates aim to correct the model by finding gradients that would minimize the loss, we believe focusing more on the updates that result in higher loss would provide a more targeted improvement.

5) Effect of Algorithmic Variations on Training Time: The results of the average training time for the different variations in data splitting and parameter aggregation methods are found in Figure 8. Again, Y axis represents the amount of time required on average to train for 5 epochs in seconds, and each bar represents a different experiment with the bars grouped based on the data splitting methods. Note that all experiments here used 10 nodes and 5 epochs of training. In each group, the left bar represents the average time required using the average parameter aggregation method (in blue) while the right bar represents the average time required for the weighted parameter aggregation method (in orange).

Amongst each group, there does not seem to be any difference in the average time taken to train. In some instances,

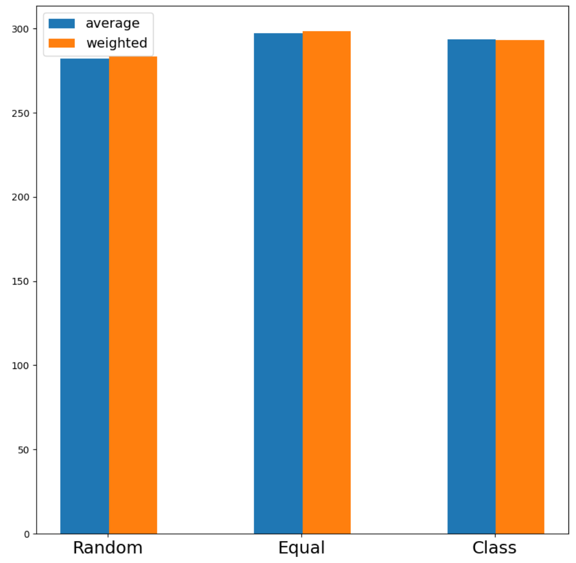

Fig. 8. Average times (n=10 runs) for combinations of data splitting and aggregation methods.

there is a very slight increase in time required for the weighted aggregation method due to the computation required to normalize loss and perform a weighted average.

We observe that the time required for the class data splitting method is higher than the random one due to the need for initial pre-processing required in the front end to identify and split the entire dataset based on class type. This results in an increase in the time required by around 5%. Further, the equal splitting method requires slightly more time than the class splitting method on average because it uses the output of the latter in order to split the data equally by class. This results in an increase in time required for the equal splitting method of 2% over the class splitting method. Overall, the difference in time required is minor.

VII. Future Work

Future works related to this project include adding fault tolerance to deal with RPC server crashes. In the current state, experiments are run in a fairly localized environment, and infrequent crashes have been mitigated by re-running the experiment. However for a more robust testing and implementation, fault tolerance should be dealt with, and there are various ways to approach this. One important factor to consider is that state of the parameters would need to be saved, since the neural network training operation is non-idempotent, re-staring the node and training again would alter the weights of the parameters. This however is not necessarily a problem as there is ”correct” set of weights, and they will inevitably be adjusted over more training cycles. Therefore, exactly-once semantics is not necessarily required and a more robust at-least-once semantics strategy, such as reissue can be explored. As well, other combinations of client reissue strategies and server actions and acknowledgements may be suitable. A fault tolerant approach would also benefit a non-blocking server and socket connection so that the processing does not stop or stall when a node crashes. We were unable to find a python implementation for this through Apache Thrift, however it does exist for Java. Therefore a combination of languages in the client, server and service handler architecture may be useful. In the absence of this, however another approach would to ”Give up” after a certain timeout of one of the nodes and carry on without it. Since the weights from each node are being averaged after each epoch, the number of nodes that have been dropped would need to be taken into account.

The results of this project did identify a considerable improvement in the training time of a neural network, however the testing set was limited to the MNIST dataset, for written digit classification. Further work would use larger datasets, as well as more varied data types, such audio, text or 3-D data sets. As well, there are numerous different neural network architectures, involving different functions and transformers and based on different principles, and some may lend themselves better to parallelization that others. Our tests also involved limited hardware capacity and thus training my have been affected by other hardware process competing for resources. Therefore future tests, would involve more nodes run on more powerful hardware infrastructure and possible even separated by large distances or networks, to fully test I/O and latency effects on training time and accuracy.

Another adaptation in the future would involve a testing the effects of batches with equal data from all classes. In our current implementation, one of our test cases involved splitting the data, so that each node receives an equal amount of data from each class, thus evenly distributing the data. However, once the training is started, the data from each node is randomly grouped into batch sizes, set by the user, and trained for a number of epochs. Therefore, the data within each batch may be skewed to certain class and training efficiency may be affected. This adaptation would require the batch sizes to be an integer multiple of the number classes, to ensure even distribution. Therefore, we would further ensure data equality not only between the nodes of the system, but also within the nodes of each system, as the neural network is trained.

VIII. Conclusion

A key reason for the success of Deep Neural Networks (DNNs) is the various hardware and software optimizations that allow DNNs to scale to bigger datasets and more complicated tasks. Distributed DNN training can further enhance this scaling factor of DNNs - enabling them to learn using even more data. In this project, we implement a unique version of Downpour-SGD, the original asynchronous deep neural network training algorithm, using Pytorch. We surveyed several potential frameworks, and opted to use the Apache Thrift framework for flexibility to modify Downpour-SGD while abstracting the communication between clients and servers. We further introduce and implement variations in data splitting (equal and class) as well as in parameter aggregation methods (weighted and average) for the Downpour-SGD algorithm. To the best of our knowledge, our implementation of the algorithm using Thrift and Pytorch, as well as the variations introduced, are novel. We further list and implement optimizations that can minimize the message sizes between manager and worker nodes significantly, while accepting the penalty of a marginal increase in file read/writes. We then ran experiments to view the effects of varying the number of the number of worker nodes as well the effects of implementing variations to the Downpour SGD algorithm on the training time required, and the test accuracy of the resulting model. We show that there is an inverse relationship between the time required for training and the model accuracy. The greater the number of worker nodes, the less time required for training, but the less the resulting model accuracy is. However, we found that the model trained using ten worker nodes resulted in an improvement of 28.86% in training time and a decrease in accuracy of 6.5% over a model trained using standard training. We believe this is still an advantageous trade-off. Further, we show that the equal data splitting method and weighted parameter aggregation methods result in a slightly better accuracy with only a negligible difference in training time over default settings in Downpour-SGD, showing that they can be used instead of the default settings for an even better performing model. We note that with better hardware, the improvement of distributed DNN training algorithms can be even more significant compared to standard DNN training. Finally, we provide some insight into potential future research ideas, including leveraging concepts and ideas from distributed computing to make the implementation more fault-tolerant and scalable. Because of intensive data and compute requirements, as well as the increasing use of DNNs in real-time, online settings, we believe that Machine Learning and Distributed Computing will be even more interwoven in the future.

References

[1] K. Vodrahalli and A. K. Bhowmik, “3d computer vision based on machine learning with deep neural networks: A review,” *Journal of the Society for Information Display*, vol. 25, no. 11, pp. 676–694, 2017.

[2] D. Goularas and S. Kamis, “Evaluation of deep learning techniques in sentiment analysis from twitter data,” in *2019 International Conference on Deep Learning and Machine Learning in Emerging Applications (Deep-ML)*, 2019, pp. 12–17.

[3] A. Loureiro, V. Miguéis, and L. F. da Silva, “Exploring the use of deep neural networks for sales forecasting in fashion retail,” *Decision Support Systems*, vol. 114, pp. 81 – 93, 2018.

[4] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[5] J. Welser, J. W. Pitera, and C. Goldberg, “Future computing hardware for ai,” in *2018 IEEE International Electron Devices Meeting (IEDM)*, 2018, pp. 1.3.1–1.3.6.

[6] V. Hegde and S. Usmani, “Parallel and distributed deep learning,” in *Tech. report, Stanford University*, 2016.

[7] K.-C. J. Chen, M. Ebrahimi, T.-Y. Wang, and Y.-C. Yang, “Noc-based dnn accelerator: A future design paradigm,” 2019.

[8] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, “Achieving super-linear speedup across multi-fpga for real-time dnn inference,” vol. 18, no. 5s, 2019.

[9] L. Floridi and M. Chiriatti, “Gpt-3: Its nature, scope, limits, and consequences,” *Minds & Machines*, 2020.

[10] K. S. Chahal, M. S. Grover, K. Dey, and R. R. Shah, “A hitchhiker’s guide on distributed training of deep neural networks,” *Journal of Parallel and Distributed Computing*, vol. 137, pp. 65–76, 2020.

[11] B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent,” in *Advances in neural information processing systems*, 2011, pp. 693–701.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, “Large scale distributed deep networks,” in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[13] A. Gupta, H. K. Thakur, R. Shrivastava, P. Kumar, and S. Nag, “A big data analysis framework using apache spark and deep learning,” in *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2017, pp. 9–16.

[14] A. Galakatos, A. Crotty, and T. Kraska, “Distributed machine learning.” 2018.

[15] Y. Liu, L. Xu, and M. Li, “The parallelization of back propagation neural network in mapreduce and spark,” *International Journal of Parallel Programming*, vol. 45, no. 4, pp. 760–779, 2017.

[16] V. J. Hodge, S. O’Keefe, and J. Austin, “Hadoop neural network for parallel and distributed feature selection,” *Neural Networks*, vol. 78, pp. 24–35, 2016.

[17] J. Wei, J. K. Kim, and G. A. Gibson, “Benchmarking apache spark with machine learning applications,” *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA*, 2016.

[18] N. Johnsirani Venkatesan, C. Nam, and D. R. Shin, “Deep learning frameworks on apache spark: a review,” *IETE Technical Review*, vol. 36, no. 2, pp. 164–177, 2019.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[20] M. Slee, A. Agarwal, and M. Kwiatkowski, “Thrift: Scalable cross-language services implementation,” *Facebook White Paper*, vol. 5, no. 8, 2007.

[21] Y. LeCun and C. Cortes, “MNIST handwritten digit database,” http://yann.lecun.com/exdb/mnist/, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/