DISTRIBUTED DEEP NEURAL NETWORKS TRAINING FOR BRAIN IMAGING APPLICATIONS

By

SUDHEER RAJA

Presented to the Faculty of the Graduate School of

THE UNIVERSITY OF TEXAS AT ARLINGTON

in Partial Fulfilment of the Requirements for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

DEC 2019

Copyright © Sudheer Raja 2019

All Rights Reserved

ACKNOWLEDGMENT

I would like to convey my deepest gratitude to my advisor, Dr. Dajiang Zhu for giving me the opportunity to carry out this research. Without his trust and encouragement, I would never have been introduced to the exciting field of Brain Imaging and Machine Learning. His supervision throughout this thesis has been a stimulating intellectual experience. His continued guidance and support have contributed in the completion of an otherwise difficult thesis.

I want to thank my committee members, Dr. Junzhou Huang and Dr. Jia Rao, for their interest in my research and taking out the time to be a part of my thesis committee.

I would like to thank my friends and colleagues, most importantly Nitin Kanwar, Akib Zaman, and Ravi Kiran for their continuous feedback and help during my Master's degree.

I would also like to extend my appreciation to the Computer Science and Engineering department for providing me with all the necessary facilities and infrastructure to carry out my Master's research.

Lastly, I would like to thank my beloved family for their encouragement to pursue my goals.

ABSTRACT

Over the recent years, Deep Neural Networks (DNNs) have surpassed human-level intelligence in recognizing and interpreting complex patterns in data. Ever since the ImageNet competition in 2012, Deep Learning (DL) has become a promising approach for solving numerous problems in the field of Computer Science. However, the neuroscience community is not able to utilize the DL algorithms effectively because the brain imaging datasets are huge in terms of size, and the current sequential training techniques do not scale up well for such big datasets. Without the proper amount of training data, training DNN models to competitive accuracies is quite challenging. Even with powerful GPUs or TPUs, the training performance can still be unsatisfactory if each data sample itself is large, as in the case of the brain imaging datasets. One solution is to parallelize the training process instead of training in a sequential mini-batch fashion. However, the currently available distributed training techniques suffer from several problems like computation bottleneck and model divergence. In this thesis, we discuss a novel training technique that can overcome these problems by distributing the model training across multiple GPUs on different nodes asynchronously and updating the gradients synchronously during the backward pass (backpropagation) in a Ring manner. We explore how to build such systems and train models efficiently using model replication and data parallelism techniques with very minimal changes to the existing code. We perform a comparative performance analysis of the proposed technique, training several Convolutional Neural Network (CNN) models on single-GPU, multi-GPU systems, and a Multi-node Multi-GPU cluster. Our analysis provides conclusive support that the proposed training technique can significantly out-perform the traditional sequential training approach.

Keywords: Distributed Deep Learning, Deep Neural Networks (DNNs), brain imaging training optimization.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENT . . . iii

ABSTRACT . . . iv

LIST OF TABLES . . . vii

LIST OF FIGURES . . . viii

I Introduction . . . 1

1 Motivation . . . 2

2 Introduction to Message Passing Interface . . . 3

2.1 MPI communication methods . . . 4

2.1.1 MPI_Send . . . 4

2.1.2 MPI_Bcast . . . 5

2.1.3 MPI_Alltoall . . . 5

2.2 Advanced MPI communication methods . . . 6

2.2.1 MPI_Scatter . . . 6

2.2.2 MPI_Gather . . . 7

2.2.3 MPI_Allgather . . . 10

2.2.4 MPI_Reduce . . . 12

2.2.5 MPI_Allreduce . . . 15

3 Introduction to Neural Networks . . . 18

3.1 Perceptron Neural Network . . . 18

3.2 Activation functions . . . 20

3.2.1 Sigmoid activation function . . . 20

3.2.2 ReLU activation function . . . 22

3.2.3 Tanh activation function . . . 23

3.3 Deep Neural Networks . . . 24

3.3.1 Forward pass . . . 25

3.3.2 Backward pass . . . 25

3.4 Gradient Descent . . . 27

3.4.1 Stochastic Gradient Descent . . . 28

3.4.2 Mini-batch Gradient Descent . . . 28

4 Distributed training methods . . . 30
4.1 Terminologies used in Distributed systems . . . 30
4.1.1 Master or parameter server . . . 30
4.1.2 Worker or Compute node . . . 30
4.2 Synchronous and Asynchronous training methods . . . 30
4.3 Centralized method . . . 31
4.4 Decentralized method . . . 33
4.5 Drawbacks of the traditional distributed training methods . . . 37

II Decentralized training with a centralized parameter update training method . . . 38

5 Proposed training method . . . 39
5.1 Performance analysis . . . 41

6 Setup . . . 44
6.1 Data acquisition and preprocessing . . . 44
6.1.1 HCP dataset . . . 44
6.1.2 MNIST dataset . . . 46
6.2 Deep Neural Network models . . . 47
6.2.1 Simple-CNN . . . 47
6.2.2 U-Net . . . 48
6.3 Cluster setup . . . 51
6.3.1 System configuration . . . 53
6.3.2 Network File System (NFS) setup . . . 53
6.3.3 Shared user setup . . . 53
6.3.4 Networking setup . . . 54
6.3.5 MPICH setup . . . 55
6.3.6 Cluster process manager setup . . . 55
6.3.7 Additional libraries setup . . . 56
6.3.8 Horovod setup . . . 56
6.4 Deep learning package manager setup . . . 57

7 Results . . . 58
7.1 Experimental setup . . . 58
7.2 Evaluation . . . 59

8 Conclusion . . . 63

9 Future Work . . . 64

REFERENCES . . . 66

LIST OF TABLES

5.1 Tabular column comparing theoretical communications in centralized, decentralized, and proposed training method between nodes for an epoch . . . 43

6.1 HCP dataset one subject and 15 subjects chunks . . . 46

6.2 MNIST dataset chunks . . . 47

6.3 Simple-CNN model parameters and output shapes . . . 48

6.4 U-Net model parameters and output shapes . . . 51

7.1 Comparison of Simple CNN model training time on MNIST dataset across four experimental setups . . . 60

7.2 Comparison of U-Net model training time on HCP one subject dataset across four experimental setups . . . 61

7.3 Comparison of U-Net model training time on HCP 15 subjects dataset across four experimental setups . . . 62

LIST OF FIGURES

2.1 MPI_Send operation . . . 5

2.2 MPI_Bcast operation . . . 5

2.3 MPI_Alltoall operation . . . 6

2.4 MPI_Scatter operation . . . 6

2.5 MPI_Gather operation . . . 8

2.6 MPI_Allgather operation . . . 10

2.7 MPI_Reduce operation . . . 13

2.8 MPI_Allreduce operation . . . 15

3.1 Perceptron architecture taken from the article [11] describes the structure of a perceptron . . 19

3.2 Sigmoid function taken from the article [4] describes the behavior of sigmoid function over a set of values . . . 21

3.3 Gradient of sigmoid function taken from the article [4] describes the gradient behavior of the sigmoid function over a set of values . . . 21

3.4 ReLU function taken from the article [4] describes the behavior of sigmoid function over a set of values . . . 22

3.5 Gradient of the ReLU function taken from the article [4] describes the gradient behavior of the ReLU function over a set of values . . . 23

3.6 Tanh function taken from the article [4] describes the behavior of tanh function over a set of values . . . 23

3.7 Gradient of the Tanh function taken from the article [4] describes the gradient behavior of the Tanh function over a set of values . . . 24

3.8 Multi-layer perceptron model 25

4.1 Master all-reduce operation using centralized communication method 32

4.2 Centralized training method using Master-allreduce communication algorithm 33

4.3 Share reduce phase of the Ring all-reduce algorithm 34

4.4 Share only phase of the Ring all-reduce algorithm 35

4.5 De-centralized training method using Ring-allreduce communication algorithm 36

5.1 Proposed training method that combines both the centralized and decentralized training methods 40

6.1 Simple CNN model 47

6.2 U-Net model 49

6.3 Cluster architecture 52

6.4 Cluster software architecitecute 57

7.1 Comparison of training time with Simple CNN model on MNIST dataset 59

7.2 Comparison of training time with U-Net model on HCP one subject dataset 60

7.3 Comparison of training time with U-Net model on HCP 15 subjects dataset 61

Part I

Introduction

Chapter 1

Motivation

One of the key challenges in modern Deep Neural Networks (DNNs) is training Neural Networks with large volumes of data. As the size of datasets increases, it becomes very challenging to train models to their best accuracy in a short amount of time. Even with powerful GPUs like Nvidia Titan X, V100, and K8, the training performance can be only tuned to an extent. This is mainly due to the scaling limitation with the traditional sequential training method when datasets are large. Especially in the field of Brain Imaging where training samples itself is large, very few samples can fit into the GPU, affecting the training performance even more.

Since a lot of data cannot fit into a single GPU at once, we cannot model performance significantly using the sequential training approach. However, distributing the training process across computers and training models parallelly can alleviate the computation load on a single computer and improve the overall training performance.

The popular Deep Learning frameworks like MXnet and Tensorflow [1] [3] offer a way to train distributed models but require a lot of system and code configurations to run distributed training jobs each time, making them difficult to use for Brain Imaging research work.

To make distributed training accessible for the Brain Imaging field, we explore several distributed training algorithms that are currently available in the popular Deep Learning libraries, their drawbacks and propose an easy to use the framework to train DNN models concurrently.

Chapter 2

Introduction to Message Passing Interface

Message Passing Interface (MPI) is an inter-process message-passing programming standard designed for communicating with parallel programs. It allows processes on different systems to communicate with each other and execute tasks in parallel. MPI has many Application Programming Interfaces (APIs) to facilitate this communication. MPI can work across different platforms, CPU and GPU instruction sets, network bandwidths and memory sizes.

MPI works on the principles of Master and Slave architecture. In the Master and slave architecture, a master process assigns a task to its slave processes. The slaves finish the task and return the result to the master. Similarly, an MPI process acting as a master distributes tasks to other MPI processes. The other MPI process executes the tasks and returns the result. This master and slave behavior is completely abstracted from the MPI user's knowledge. Users can run tasks as if they are running on a single system. The MPI handles process creation, memory management, and communication among processes. The number of processes in MPI can be scaled easily to several thousands of processes across computers. The same MPI API can work both on a single computer or a cluster. MPI offers portability, high-performance, low latency and scalable for executing parallel processes.

MPI functionalities operate in the Transport Layer (TL) (level 5 in the OSI Reference Model) where Sockets and Transmission Control Protocol (TCP) are used. MPI has interfaces in several programming languages like C++, Fortran, MATLAB, Python, R, OCaml, and Java, on most operating systems like Solaris, Linux,

macOS, Windows. It can work across different hardware and software vendors like IBM, Intel, AMD, Nvidia, Oracle.

MPI has several API standards developed from MPI-1 through MPI-3. Script file output.markdown.lua not found

2.1 MPI communication methods

The basic functionality of MPI involves providing essential communication and synchronization between distributed parallel processes on a single computer or across a distributed system. Each process created by MPIi is called as MPI process. The communication between various MPI processes is handled by a process manager called communicator. A communicator spawns multiple distributed parallel processes and assigns a unique identifier to the MPI process called rank. Each MPI process communicates messages to each other using the rank.

THe number of MPI procesess that can be spawned depends on the number of CPU cores, since each MPI process is assigned to run on a single core by the communicatory. For instance if there are 3 MPI processes in a 3 core computer, the communicator assigns each MPI process to a core making them much efficient to run concurrently.

The communication between the MPI processes is in the form of message send and receive operations using TCP/IP protocol. Processes send and receive messages using message tags and ranks.

There are two types of communication protocols based on the number of MPI processes involved in the communication. One is point-to-point communication and the other one is a group-wise communication method. We will discuss these protocols in the subsections.

Each MPI communication protocol is built upon three basic operations. They are described as follows.

2.1.1 MPI_Send

MPI send protocol allows MPI processes to send messages to each other. Each process can send and receive messages with each other. This protocol is widely used in point-to-point asynchronous communication.

The MPI_send API is illustrated in the figure 2.1. Using MPI_Send process 0 sends the element 7 to the processes 1, 2, and 3. After the operation, ranks 0, 1, 2 and 3 all have 7 in its memory.

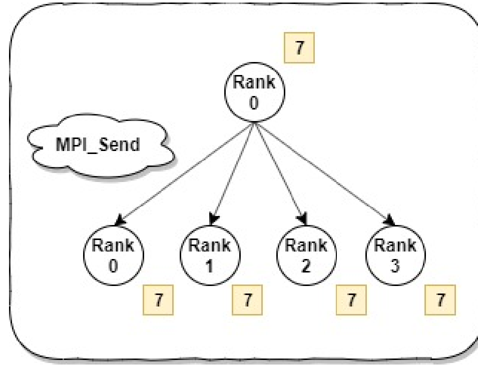

Figure 2.1 MPI_Send operation

2.1.2 MPI_Bcast

The MPI Broadcast function sends messages to a group of processes from a single rank. This protocol is widely used in group-wise collective communication methods.

The figure 2.2 illustrates the MPI_Bcast operation. Let consider that Rank 0, 1, 2, and 3 belong to a broadcast group called "**bcast_group1**". Rank 0 sends an element 7 to "bcast_group1". After this operation, each process holds 7 in their memory.

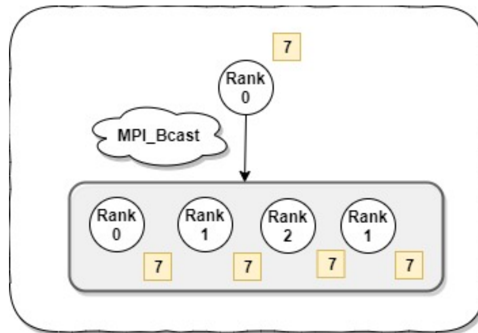

Figure 2.2 MPI_Bcast operation

2.1.3 MPI_Alltoall

MPI Alltoall protocol allows sending messages from every MPI processes to all MPI processes. That is each process can elements in their memory to all other processes in a single operation. Once every process receives elements all other processes, MPI Alltoall sorts the data according to the rank after collecting from all the nodes asynchronously.

The send, receive, and sorting operation performed by "MPI_Alltoall" is illustrated in the figure 2.3 After sending and receiving elements, the gatherred results from the processes 0, 1, 2, 3, and 4 received in the order 3, 1, 2, 0, 4 are re-organized in ascending order with respect to their ranks. In our case rank 0 to 4 order.

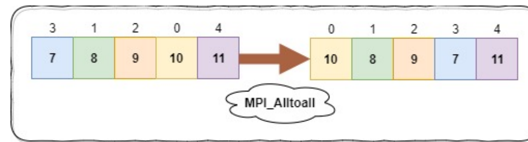

Figure 2.3 MPI_Alltoall operation

2.2 Advanced MPI communication methods

Based on the operations discussed in the section 2.1, several advanced protocols are introduced in later later versions from MPI 2.x through MPI 3.x.

2.2.1 MPI_Scatter

The difference between MPI_Bcast and MPI_Scatter is, MPI_Scatter sends different chunks of data from a MPI process to other MPI processes. The MPI_Scatter protocol is illustrated in the below figure 2.4.

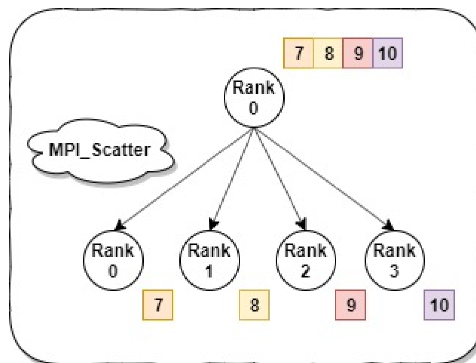

Figure 2.4 MPI_Scatter operation

Lets consider an example, process 0 has elements 7, 8, 9 and 10 in its memory in form of an array, and rank 0 sends data to 0, 1, 2 and 3 (shown in the figure 2.4. After performing the MPI_Scatter API operation, process 0 will have 7, process 1 will have 8, process 2 will have 9, and process 3 will have 10 in its memory respectively.

The MPI_Scatter API functional declaration is as follows.

```
MPI_Scatter(
          void* send_data,
          int send_count,
          MPI_Datatype send_datatype,
          void* recv_data,
          int recv_count,
          MPI_Datatype recv_datatype,
          int root,
          MPI_Comm communicator)
```

The first argument "send_data" is an array located on the source process, the second argument "send_count" describes number of elements to send to each destination process. The third argument "MPI_Datatype" describes the datatype of the data to send. The fourth argument "recv_data" describes destination for the data. The fifth argument describes the size of the data returned from the processes. The sixth and last argument describes the root process where the data is present and the communicator process that handles the operation.

2.2.2 MPI_Gather

MPI_Gather aggregates data from different MPI processes on to a single process. If the data received is not in the order of the rank, MPI_Gather sorts the received data based on the rank. The gather operation is usually followed by a reduce operation like sum, mean, max or min is performed that returns a single value.

The figure 2.5 illustrates the MPI_Gather operation. In the figure, Rank 0, 1, 2 and 3 holds small chunks of data 0, 1, 2, and 3 in their process memory respectively. After performing the MPI_Gather operation from rank 0, processes 0 will have elements 0, 1, 2, and 3 from all ranks.

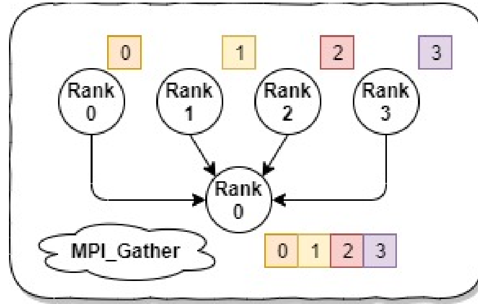

Figure 2.5 MPI_Gather operation

The MPI_Gather API functional declaration is as follows.

```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

The arguments for MPI_Gather are mostly similar to that of MPI_Scatter. The “recv_data” argument contains the data buffer where the data received can be stored after collecting data from other MPI processes. The “recv_count” argument describes how many elements needs to be received from each process. The only difference between MP_Scatter and MPI_Gather is that the root process (0 in our example) needs to have a valid buffer to store the received data otherwise the call fails.

A simple example to compute the mean using Scatter and Gather commands:

```
float calculate_mean():
{
        if (local_rank == 0) {
                nums_array = generate_rand_nums(num_elements_per_rank * num_ranks);
        }

        // Create a integer buffer that holds the integer array
```

```
    float *ret_nums_array = malloc(sizeof(float) * num_elements_per_rank);

    // Send a chunk of data to each process
    MPI_Scatter(nums_array, num_elements_per_rank, MPI_INT, ret_nums_array,
                            num_elements_per_rank, MPI_FLOAT, 0, MPI_COMM_WORLD);

    // Compute mean of the numbers on each rank (process)
    chunk_avg = compute_mean_nums(sub_rank_nums_array, num_elements_per_rank);

    // Buffer to hold chunk means
    float *chunk_avgs = NULL;
    if (local_rank == 0) {
            sub_rank_avg = malloc(sizeof(float) * num_ranks);
    }
    MPI_Gather(&chunk_avg, 1, MPI_FLOAT, chunk_avgs, 1, MPI_FLOAT, 0,
                    MPI_COMM_WORLD);

    // Compute global mean of mean from each process.
    if (local_rank == 0) {
            float global_avg = compute_mean_nums(chunk_avgs, num_ranks);
    }
}
```

The above examples demonstrates a simple mean function using Scatter and Gather protocols.

If we run the program with 4 MPI processes with 3 elements per each rank, the program creates a random integer array of 12 numbers. The array is then divided into 4 chunks of 3 elements each and, are scattered to each rank. Each rank then calculates a local mean of the numbers and returns the value to rank 0 processes. The rank 0 process accumulates the result from all the processes, then calculates the global mean of the numbers.

The output of the program will look like this if we consider the random numbers to be [0. 1. 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] then,

The local mean on rank 0 is 1 The local mean on rank 1 is 4 The local mean on rank 3 is 7 The local mean

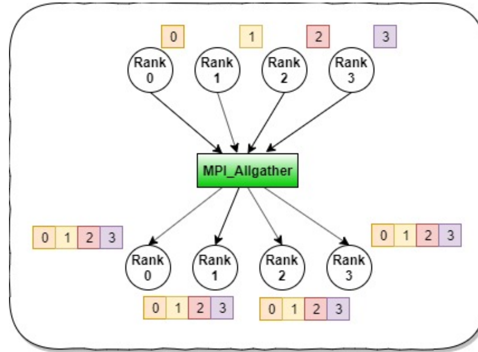

Figure 2.6 MPI_Allgather operation

on rank 4 is 10 Local means gathered on rank 0 is [1, 4, 7, 10]

The global mean on rank 0 is 5.50

Where rank 0, rank1, rank 2, rank 3 is 4 MPI processes.

In terms of space complexity, MPI_Gather takes the same amount of space to store n numbers for a single process or for multiple processes. But in terms of time complexity, it performs almost 4 times better to compute mean than a single process.

2.2.3 MPI_Allgather

The functionality of MPI_allgather is similar to that of MPI_gather. The only difference between these two is, MPI_allgather gathers and sends the data from all MPI processes to all MPI processes. The main advantage of MPI_allgather is that it doesn't create a bottleneck on single process with a lot of communication.

In the figure 2.6, rank 0, 1, 2, and 3 are 4 MPI processes which has elements 0, 1, 2, and 3 in their memory respectively. After performing the Allgather operation, each MPI process will have all the elements which were in processes 0, 1, 2, and 3 in their memory.

The functional declaration of MPI_Allgather is as follows.

```
MPI_Allgather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
```

```
    int recv_count,
    MPI_Datatype recv_datatype,
    MPI_Comm communicator)
```

The only difference between MPI_Allgather API call and MPI_gather API functional declaration is that MPI_Allgather doesn't have a root node.

Let's look at a simple example to calculate mean using MPI_Allgather.

```
float calculate_mean_allgather():
{
        if (local_rank == 0) {
                nums_array =
                generate_rand_nums(num_elements_per_rank * num_ranks);
        }

        // Create a integer buffer that holds the integer array
        float *ret_nums_array = malloc(sizeof(float) * num_elements_per_rank);

        // Send a chunk of data to each process
        MPI_Scatter(nums_array,
                    num_elements_per_rank,
                    MPI_INT,
                    ret_nums_array,
                            num_elements_per_rank,
                            MPI_FLOAT, 0,
                            MPI_COMM_WORLD);

        // Compute mean of the numbers on each rank (process)
        chunk_avg = compute_mean_nums(sub_rank_nums_array,
                                      num_elements_per_rank);

        // Buffer to hold chunk means
        float *chunk_avgs = NULL;
```

```
        if (local_rank == 0) {
                sub_rank_avg = malloc(sizeof(float) * num_ranks);
        }

        // Gather chunk averages from each MPI process
        float *chunk_avgs = (float *)malloc(sizeof(float) * num_ranks);

        MPI_Allgather(&sub_rank_avg, 1, MPI_FLOAT, chunk_avgs, 1, MPI_FLOAT,
                                        MPI_COMM_WORLD);

        // Compute global mean of mean from each process.
        float global_avg = compute_mean_nums(chunk_avgs, num_ranks);
}
```

In the above example, the global average is collected on all ranks (processes or nodes) instead of collecting on a single rank. Other than that, the implementation for MPI_gather and MPI_Allgather is the same.

If we run the above code with 12 random numbers, 4 MPI processes with 3 elements per rank, the output will look something similar.

If the numbers are [0. 1. 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] then,

The mean on rank 0 is 5.50 The mean on rank 1 is 5.50 The mean on rank 3 is 5.50 The mean on rank 4 is 5.50

Where rank 0, rank1, rank 2, rank 3 is 4 MPI processes. In the case of MPI_Allgather operation, the local mean and the global mean are the same.

2.2.4 MPI_Reduce

MPI_Reduce operation is basically a reverse operation of the MPI_Bcast. Instead of sending data to a set of nodes, it takes data from a group of nodes and sends it to a specific node. Usually, this is accompanied by an aggregate operation like sum, average, difference, multiply, or divide.

MPI_Gather operation is usually followed by MPI_Reduce operation. Like MPI_Gather, MPI_Reduce function gathers elements from every MPI process and performs reduce operation.

The below figure illustrates the MPI_Reduce operation. Rank 0, 1, 2, 3 are 4 MPI ranks (processes). They have 0, 1, 2, 3 values in their process memory respectively. After performing the MPI_SUM reduce operation on these ranks we get a resultant value 6 which is stored in the root rank 0.

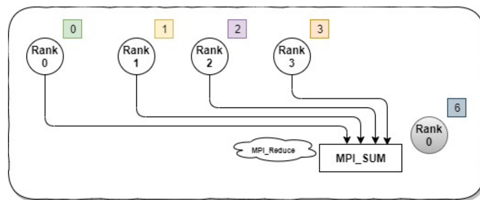

Figure 2.7 MPI_Reduce operation

A typical MPI_Reduce functional declaration looks as follows.

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

The "send_data" argument specifies the list of numbers on which the reduce operation needs to be performed. The "recv_data" is a buffer that collects the results gathered from each MPI process. The third argument "MPI_Datatype" specifies the data type of the elements in the "send_data" array and the values gathered in "recv_data" array after reduce operation is performed. The "op" argument specifies the kind of reduce operation that needs to be performed. The root argument specifies the root rank process and the last argument MPI_Comm specifies the communicator method.

Let's take a simple example that computes the sum of numbers using the MPI_Reduce function.

```
float calculate_sum()
{
        // Create a float array to hold random numbers
        float *nums_array = NULL;
        nums_array = generate_rand_nums(num_elements_per_rank);
```

```
    // Calculate sum of the numbers on each rank
    float rank_sum = 0;
    int i;
    for (i = 0; i < num_elements_per_rank; i++) {
      rank_sum += nums_array[i];
    }

    // Reduce the sums on each rank into global sum
    float sum;
    MPI_Reduce(&rank_sum, &sum, 1, MPI_FLOAT, MPI_SUM, 0,
                                 MPI_COMM_WORLD);
}
```

The above code demonstrates a simple reduce sum program that generates a random number array on each rank. The numbers are then summed up to calculate a local sum. The rank sums are then reduced to a global sum using the MPI_reduce method.

If you run the above program with 4 MPI processes and 3 elements per each rank for random numbers [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] then the result will look as follows.

The sum on rank 0 - 3

The sum on rank 1 - 12

The sum on rank 2 - 21

The sum on rank 3 - 30

Global average - 5.50

Some of the reduce operations available in the MPI_Reduce function.

1. **MPI_MAX** - Finds the maximum element of all the elements.
2. **MPI_MIN** - Finds the minimum element of all the elements.
3. **MPI_SUM** - Calculates the sum of all the elements.
4. **MPI_PROD** - Calculates the product of all the elements.
5. **MPI_MAXLOC** - Finds the maximum element of all the elements and returns the rank of the max.
6. **MPI_MINLOC** - Finds the minimum element of all the elements and returns the rank of the min.

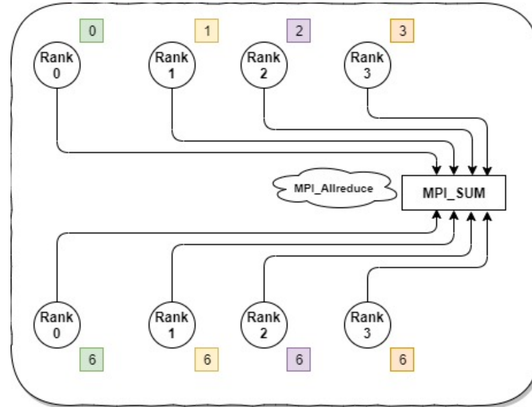

Figure 2.8 MPI_Allreduce operation

7. **MPI_LAND** - Performs a logical AND operation on all the elements.
8. **MPI_LOR** - Performs a logical OR operation on all the elements.
9. **MPI_BAND** - Performs a bitwise AND on all the elements.
10. **MPI_BOR** - Performs a bitwise OR on all the elements.

2.2.5 MPI_Allreduce

MPI_Allreduce protocl is similar to MPI_Reduce protocol. The only difference between these two is MPI_Allreduce doesn't need a root node. MPI_Allreduce is used in situations where the result of Reduce operation is needed on every rank. This basically does two operations at the same time. API_Reduce and MPI_BCast. MPI_Allreduce reduces the need to distribute results to all processes after computing the Reduce operation.

The below figure illustrates the MPI_Allreduce function. Rank 0, 1, 2, and 3 are 4 MPI ranks (processes) with 0, 1, 2 and 3 elements in them respectively. After performing MPI_SUM Allreduce operation on these ranks, each rank will now hold the resultant sum 6 in their memory.

Let's look at a simple example that calculates the standard deviation of a set of random numbers using MPI_Allreduce. This particular example demonstrates the use of Allreduce.

```
float calculate standard_deviation()
{
        // Generate random numbers
        nums_array = generate_rand_nums(num_elements_per_rank);
```

```
// Calculate sum of number on each rank and reduce to node 0
float rank_sum = 0;
int i;
for (i = 0; i < num_elements_per_rank; i++) {
  rank_sum += nums_array[i];
}

float sum;
MPI_Allreduce(&rank_sum, &sum, 1, MPI_FLOAT, MPI_SUM,
                                    MPI_COMM_WORLD);

// Calculate mean on each rank
float mean = sum / (num_elements_per_rank * world_size);

// Compute squared difference on each rank
float rank_sqrd_diff = 0;
for (i = 0; i < num_elements_per_rank; i++) {
  rank_sqrd_diff += (nums_array[i] - mean) ** 2;
}

// Reduce rank difference to rank 0
float global_sqrd_diff;
MPI_Reduce(&rank_sqrd_diff,
              &global_sqrd_diff,
              1,
              MPI_FLOAT,
              MPI_SUM, 0,
                         MPI_COMM_WORLD);

// Calculate standard deviation
if (world_rank == 0) {
```

```
        float stddev = sqrt(global_sqrd_diff /
                                              (num_elements_per_rank *
                                              world_size));
    }
}
```

In the above example, the program generates a list of random numbers on each process. The numbers are then summed up to calculate the mean. The local mean is then returned to root rank process 0 using Allreduce. A global mean is calculated from the global sum. At this stage, all rank processes have a global sum. Then each element present on the respective rank is subtracted from the global mean to calculate the difference from the mean. The difference is then squared. The squared difference is then returned to root rank process 0 using MPI_Allreduce. The resultant value is squared and divided by the number of elements in each process and the number of ranks to calculate standard deviation.

If we run the above code with 4 MPI processes and 3 elements per each process for a random integer array [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], the result will be as follows.

Sum on rank 0 - 3

Sum on rank 1 - 12

Sum on rank 2 - 21

Sum on rank 3 - 30

Global sum - 66

Global mean: 5.50

The squared difference on rank 0 - 62.75

The squared difference on rank 1 – 8.75

The squared difference on rank 2 – 8.75

The squared difference on rank 3 – 62.75

Global difference - 143

Standard deviation – 3.45

Chapter 3

Introduction to Neural Networks

3.1 Perceptron Neural Network

A Perceptron model [8] is the simplest form of a neural network consisting of a single layer of output nodes. The input is fed into the output nodes via a dot product of weights and inputs. The resultant value is converted to a 0 or 1 by applying a threshold function called activation function. The output is subtracted from the expected output to calculate error or loss of the function. The weights are then re-adjusted to improve the loss (decrease the value). This process is called a forward pass.

The weights are adjusted by calculating gradient [9] of the output w.r.t to weights in each layer. If there is only one layer then the weights are directly adjusted w.r.t input, as in the case of perceptron. In case of neural networks with more layers. The gradients are calculated w.r.t weights in each layer. Using chain rule, the gradients are then propagated backwards from the output layer to a input layer. This process is known as backpropagation [5]. The weights in each layer are then updated using a learning algorithm called delta rule. This phase in the training process is called as a backward pass.

The forward pass and backward pass are repeated several number of times until the error converges to a considerate value, typically referred as optimal convergence.

The delta rule [7] for weights w_{ji} is defined as,

$$\Delta w_{ji} = \alpha(t_j - y_j)a'(h_j)x_i \tag{3.1}$$

where,

α is a constant called learning rate,

$a(x)$ is the network's activation function,

a' is the derivative of the activation function a,

t_j is the output of the function,

x_i is the input,

y_j is the actual output,

h_j is the linear combination of weights with the input.

The perceptron model with a linear activation function fails to capture non-linear patterns in the data. Therefore, the output is transformed non-linear by applying non-linear activation functions. This is called as kernel trick. In addition to just converting the outputs to non-linear, the activation function also avoids saturation issues and helps the neural networks to converge faster.

The perceptron model [11] is illustrated in the below figure.

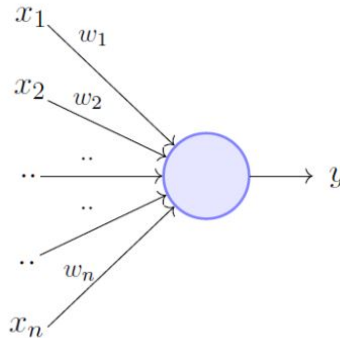

Figure 3.1 Perceptron architecture taken from the article [11] describes the structure of a perceptron

$$y = \begin{cases} 1, if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0 \\ 0, if \sum_{i=1}^{n} w_i * x_i - \theta < 0 \end{cases} \tag{3.2}$$

where,

θ is a threshold,

$\sum_{i=1}^{n} w_i * x_i$ is the dot product between weights and inputs.

Even using non-linear activation functions, the perceptron model struggles to detect complex latent patterns in data. One such case is the XOR problem, where the simple perceptron model fails to classify data into different classes even after thousands of iterations. The perceptron model is later refined to capture complicated data patterns. We will discuss these models in the next section.

Some of the widely used activation functions are discussed in the below section 3.2.

3.2 Activation functions

Sigmoid (as described earlier), Rectified Linear Units (ReLU), and Tanh are three widely used activation functions in Neural Networks which transforms the data into a non-linear representation. The perceptron model trained with a nonlinear activation function can guarantee to find a best separation hyper-plane between classes. If no such hyper-plane exists, it atleast finds a separation hyper-plane that gives a good accuracy.

3.2.1 Sigmoid activation function

Sigmoid [4] is one of the widely used activation function in DL. Sigmoid converts the values into a range between 0 and 1. As shown in the 3.2, the outuput sigmoid function values vary between 0 and 1 but never actually become 1.

The derivative of the sigmoid function is smooth and is continously differentiable. If you look at the gradient as shown in the figure 3.3, the value of gradient is much higher in-between the values -3 and 3 than the values outside the range. So this results in large changes in output for a small change in output. This helps in separating data belonging to different classes easily.

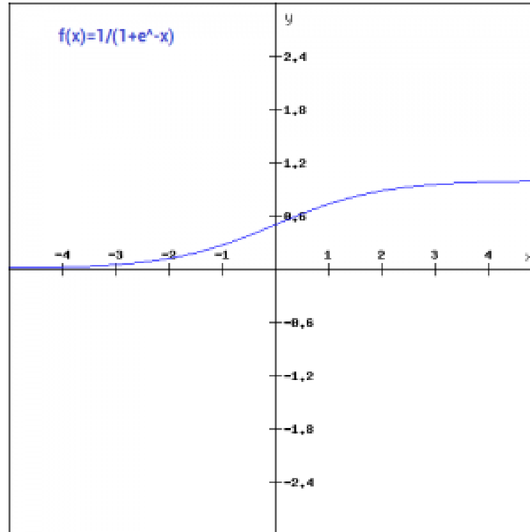

Figure 3.2 Sigmoid function taken from the article [4] describes the behavior of sigmoid function over a set of values

The sigmoid activation function is defined as follows.

$$A(x) = \frac{1}{1 + e^{-t}} \tag{3.3}$$

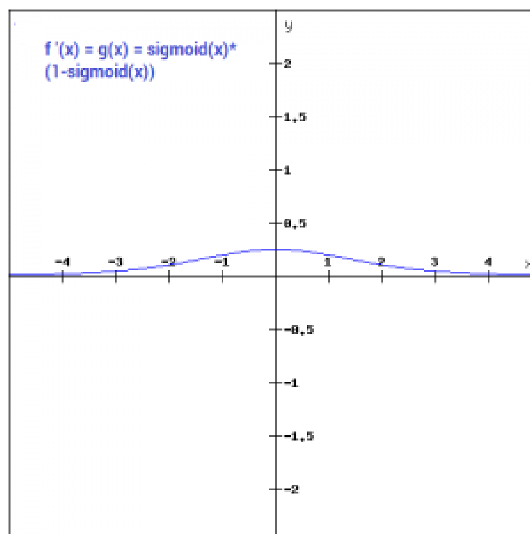

Figure 3.3 Gradient of sigmoid function taken from the article [4] describes the gradient behavior of the sigmoid function over a set of values

3.2.2 ReLU activation function

ReLU activation [4] function only transforms the negative numbers to zero and leaves the rest of the positive values as it is. The function is illustrated in the figure 3.4. The main advantage of ReLU function is, it doesn't activate the neurons if the values are less than 0. This results in the output being sparse.

The derivate of the ReLU function is 0 when the values are less than 0. Which means, when the values get to negative values thy are never updated, creating dead neurons which are never activated. The derivative is illustrated in 3.5.

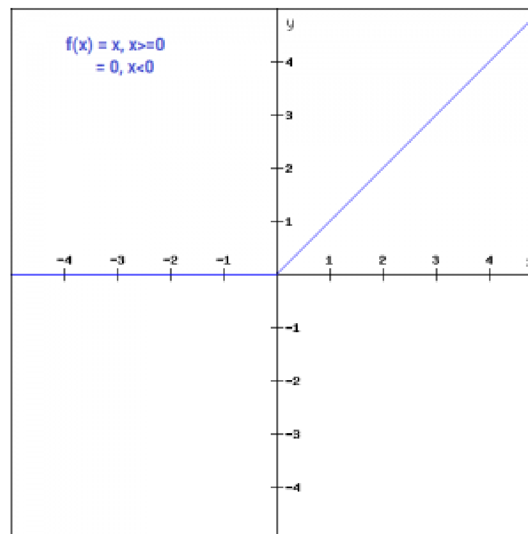

Figure 3.4 ReLU function taken from the article [4] describes the behavior of sigmoid function over a set of values

The ReLU activation function is defined as follows.

$$A(x) = max(0, x) \tag{3.4}$$

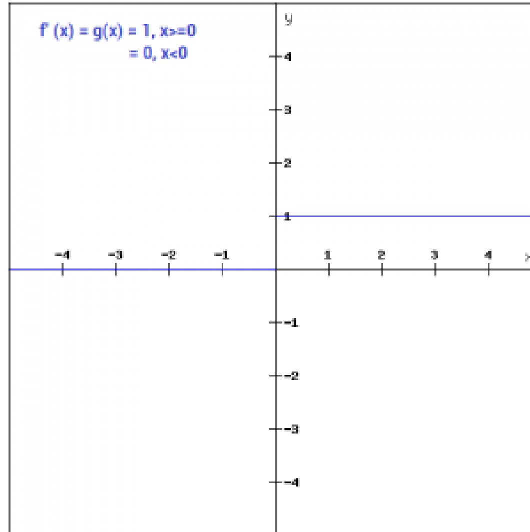

Figure 3.5 Gradient of the ReLU function taken from the article [4] describes the gradient behavior of the ReLU function over a set of values

3.2.3 Tanh activation function

Tanh function [4] is symmetric across the Y-axis. Other than that it is similar to sigmoid function. As shown in the figure 3.6, the function is continuous and differentiable at all points.

The derivative of the tanh function is much more steeper than the sigmoid function. The gradients of the tanh function is much low and the whole graph itself is flat as shown in the figure 3.7.

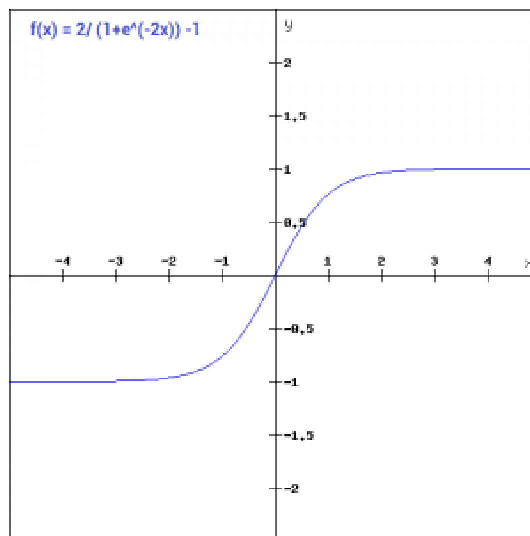

Figure 3.6 Tanh function taken from the article [4] describes the behavior of tanh function over a set of values

The tanh activation function is defined as follows.

$$A(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.5}$$

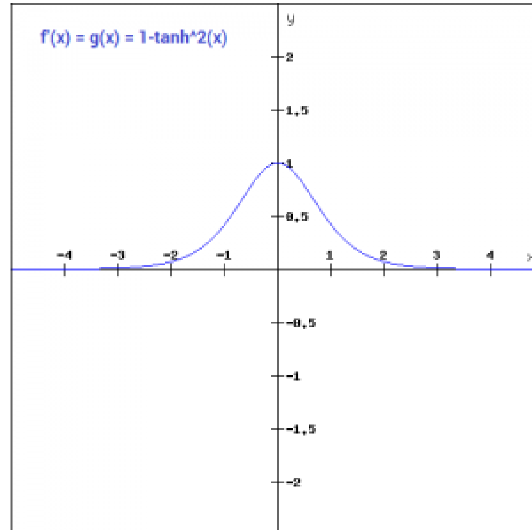

Figure 3.7 Gradient of the Tanh function taken from the article [4] describes the gradient behavior of the Tanh function over a set of values

3.3 Deep Neural Networks

Deep Neural Networks (DNNs) also called Feedforward networks, or Multi-Layer Perceptrons (MLPs) 3.8 are set of Neural Networks with many perceptrons stacked into multiple hidden layers. The DNNs can be associated with a directed acyclic graph based on how the layers are sequentially linked with each other. Suppose, if we consider three functions $f^{(1)}, f^{(2)}, and f^{(3)}$, where $f^{(1)}$ is the first layer, $f^{(2)}$ is the hidden layer, $f^{(3)}$ is the output layer, and x is the input layer, then each layer in DNNs are connected to each other in the form $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$.

The inputs flow from the input layer to the output layer in a linear fashion as in the perceptron model [10].

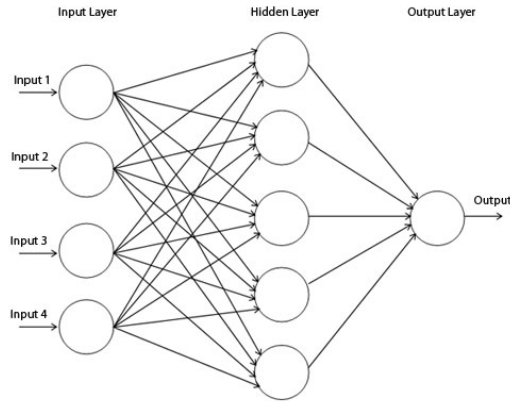

Figure 3.8 Multi-layer perceptron model

3.3.1 Forward pass

We use the following notation for this section.

x_i^l: The input to the neuron i in the layer l. y_i^l: The output of the neuron i in the layer l.

In DNNs, the output of the previous layer is considered as input to the next layer f^l. The forward pass is described as follows.

1. Compute the activation for the layers with the inputs.

$$y_i^l = \sigma(x_i^l) + \theta \tag{3.6}$$

2. Compute inputs to the next layer as follows.

$$y_i^l = \sum_{j=1} w_{ji}^{l-1} * y_{ji}^{l-1} \tag{3.7}$$

3. Repeat steps 1 and 2 until the output layer.

3.3.2 Backward pass

The goal of the DNNs is to minimize the error of the overall function. The process of minimizing the error is called training or learning. Since DNNs has many layers, the learning is done via a method called

backpropagation [6] [5]. The DNNs compute derivative of the error with respect to the weights in each layer using chain rule as follows.

$$\frac{\delta E}{\delta w_{ij}^{l}} = \frac{\delta E}{\delta x_{j}^{l+1}} \frac{\delta x_{j}^{l+1}}{\delta w_{ij}^{l+1}} \tag{3.8}$$

If we look at the forward pass, the derivative of E w.r.t x_j^{ij} is nothing but a forward pass. Therefore the above equation can be written as,

$$\frac{\delta E}{\delta w_{ij}^{l}} = y_i^l \frac{\delta E}{\delta x_{j}^{l+1}} \tag{3.9}$$

Since, we know that $y_i^l = \sigma(x_i^l) + \theta$, therefore we can compute the derivative of E w.r.t inputs (x) as follows using chain rule,

$$\frac{\delta E}{\delta x_{j}^{l}} = \frac{\delta E}{\delta y_{j}^{l}} \frac{\delta y_{j}^{l}}{\delta x_{j}^{l}} = \frac{\delta E}{\delta y_{j}^{l}} \frac{\delta}{\delta x_{j}^{l}} (\sigma(x_i^l) + \theta) = \frac{\delta E}{\delta y_{j}^{l}} \sigma\prime(x_i^l) \tag{3.10}$$

The derivative of the Error w.r.t to the neurons in the output layer (L) can be computed as follows.

$$\frac{\delta E}{\delta y_{i}^{L}} = \frac{d}{d y_{j}^{L}} E(y^L) \tag{3.11}$$

The derivative of E w.r.t output y in any layer can be computed as follows.

$$\frac{\delta E}{\delta y_{i}^{l}} = \sum \frac{\delta E}{\delta x_{j}^{l+1}} \frac{\delta x_{j}^{l+1}}{\delta y_{j}^{l}} = \sum \frac{\delta E}{\delta x_{j}^{l+1}} w_{ji} \tag{3.12}$$

Finally, we can write the backward pass algorithm as follows.

1. Compute gradients of error (E) at the output layer (L).

$$\frac{\delta E}{\delta y_{i}^{L}} = \frac{d}{d y_{j}^{L}} E(y^L) \tag{3.13}$$

2. Compute gradients of error (E) w.r.t neuron at the first hidden layer (l).

$$\frac{\delta E}{\delta x_{j}^{l}} = \frac{\delta E}{\delta y_{j}^{l}} \sigma\prime(x_i^l) \tag{3.14}$$

3. Compute gradients of error (E) w.r.t to the output of each layer (except the output layer).

$$\frac{\delta E}{\delta y_i^l} = \sum \frac{\delta E}{\delta x_j^{l+1}} w_{ji} \tag{3.15}$$

4. Compute gradients of weights.

$$\frac{\delta E}{\delta w_{ij}^l} = y_i^l \frac{\delta E}{\delta x_j^{l+1}} \tag{3.16}$$

The computed gradients are used to update the model parameters. The gradient update methods are discussed in the next section 3.4

3.4 Gradient Descent

Gradient descent (GD) is an optimization algorithm used to minimize the error by moving in the direction of the negative gradient following the steepest route possible.

After calculating gradients with respect to weights in each layer using the backpropagation algorithm (discussed in the previous section 3.3.2), the model parameters are updated using the gradient descent algorithm.

There are many variants of gradient descent algorithm. The first one is vanilla Gradient Descent (GD) [2] also referred as traditional GD algorithm. The GD algorithm performs parameter updates w in a single step for the whole dataset. The GD algorithm can be defined as follows.

$$w = w - \mu.\nabla_w J(w) \tag{3.17}$$

where,

μ is the learning rate,

∇_w is the gradient w.r.t weights,

$J(w)$ is the loss of the function for the given input.

The whole training process using GD can be described as follows.

Algorithm 1 Gradient Descent

1: **for** epoch in epochs **do**

2: $grad \leftarrow w - \mu.\nabla_{w,X,Y}$

3: $parameters \leftarrow parameters - lr * grad$

4: **end for**$=0$

3.4.1 Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) [2] calculates gradient w.r.t to each data sample in a dataset and adjusts the parameters accordingly. The learning algorithm for SGD is as follows.

$$w = w - \mu.\nabla_w J(w; x^{(i)}, y^{(i)}) \tag{3.18}$$

where,

μ is the learning rate,

∇_w is the gradient w.r.t weights,

$J(w; x^{(i)}, y^{(i)})$ is the loss of the function for a single data sample i.

The whole training process using SGD can be described as follows.

Algorithm 2 SGD

1: **for** epoch in epochs **do**

2: **for** sample in batch **do**

3: $grad \leftarrow \nabla_{w,x^{(i)},y^{(i)})}$

4: $parameters \leftarrow parameters - lr * grad$

5: **end for**

6: **end for**$=0$

3.4.2 Mini-batch Gradient Descent

The Mini-batch Gradient Descent (MGD) algorithm is a variation of SGD in which computes gradients on a mini-batches of data (set of samples) instead of calculating gradients for a single sample. The learning algorithm for MGD is the same as SGD. The only change is, It loads a mini-batch of data in step 2.

The training process for MGD is described below.

Mini-batch Gradient Descent (MGD) algorithm

1. Sample a mini-batch of data $(x^{(i:i+n)}, x^{(i:i+n)})$ on each rank.

2. Read the mini-batch data into rank's local memory and compute forward pass and gradients for the set of data asynchronously.

3. Update parameters in each rank's memory with the new values,

$parameters \leftarrow parameters - lr * grad$

4. Synchronize parameters to the neighboring rank and calculate the average of parameters.

5. Repeat the process till all the mini-batches are exhausted.

6. Repeat the process for several epochs.

For a sample size of 1000 with a mini-batch size of 100, GD computes gradients for the dataset one time, SGD computes gradients 1000 times and MGD computes gradients 10 times. Since GD performs gradient update on the whole dataset at once it converges much faster. Since SGD computes gradients on each sample it performs much slower compared to GD algorithm, however it takes up less processing power.

The performance of MGD lies in between that of GD and SGD, because MGD updates parameters in mini-batches. Since MGD offers both the benefits of GD and SGD, it is widely used in training DNNs these days.

The training process using the MGD algorithm is as follows.

Run the training script:

1. Load model and training data into memory.

2. Take a mini-batch of data from the whole dataset.

2. Compute forward pass.

3. Calculate loss and compute gradients using backpropagation.

4. Update the model parameters using MGD.

5. Repeat the process.

Chapter 4

Distributed training methods

4.1 Terminologies used in Distributed systems

In this chapter, we will discuss various traditional distributed training methods, their advantages and their disadvantages. Before starting, we shall look at some basic terminologies used in distributed training.

4.1.1 Master or parameter server

The worker nodes communicate parameters with each other using a centralized device called a master or parameter server. The role of the parameter server is to manage communciations among the workers and stores model parameters and training data on it.

4.1.2 Worker or Compute node

The worker nodes perform the actual computation for training models. When running a training computation job, each worker gets a copy of data, model, and code from the master and perform computation on it.

4.2 Synchronous and Asynchronous training methods

MGD (discussed in section 3.4.2) is an iterative algorithm where the parameters calculated in each iteration are incorporated into the next iteration. In the traditional training, these iterations are performed sequen-

tially in the same computer. In distributed training, these iterations are performed either synchronously and asynchronously across different computers.

In synchronous training, each worker trains on a mini-batch of data from a large global mini-batch of data and updates the model parameters with other workers synchronously. Only after all the workers receive the gradient updates, they proceed to the next iteration (step).

In asynchronous training also, each worker trains on the same copy of data and model. They all start from the same starting point, however, the workers don't wait for each other to finish the computation. Instead, when one worker finishes the forward pass and backward pass, it communicates the gradients to a central parameter server and gets the latest available parameters from the server. The worker then starts the next iteration with the new parameters. This way, the workers don't have to wait for each other to finish the computation. Since there is no global synchronization, the training runs much faster. The model converges much faster than training on a single node.

Both these training methods use the same MGD algorithm that we discussed in before section 3.4.2. Only difference is, in asynchronous training, the iterations are performed on different devices, i.e, each worker executes a different training epoch. For instance, on a single node, we go from iteration 1 to 2 to 3 sequentially. In asynchronous training, one worker trains on iteration 1, the other trains on 2, the other trains on 3 and so on synchronizing the model parameters through the master node. This way the model trains faster and converges quickly.

In synchronous training, each worker performs the same iteration training on a different mini-batch of data. Since this reduces number the number of iterations to go through the whole dataset, it reduces the time to train. For instance, if it takes 100 steps to iterative the whole dataset on a single system using traditinal training method, using synchronous training with 4 workers, it takes only 25 steps.

There are several communication algorithms for distributed training using synchronous and asynchronous methods. We will discuss them in the next section.

4.3 Centralized method

Master all-reduce algorithm is one of the widely used communication method for centralized distributed training nowadays. The figure 4.1 illustrates the master all-reduce communication algorithm. In the figure, there are four parallel training processes distributed across different nodes. The node A is referred as master

or as parameter server. The rest of the nodes B, C, and E and referred as workers or compute nodes (Step 1 in the figure 4.1). The master receives model parameters from the workers to perform aggregate operation (Step 2 in the figure 4.1) and sends back the results (Step 3 in the figure 4.1) to the workers. The workers update their respective models with new parameters and continues training next iteration.

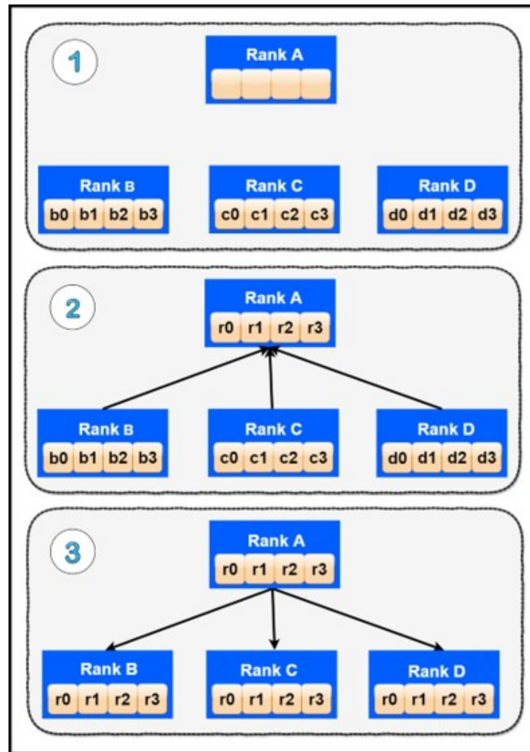

Figure 4.1 Master all-reduce operation using centralized communication method

The centralized training process is illustrated in the figure 4.2

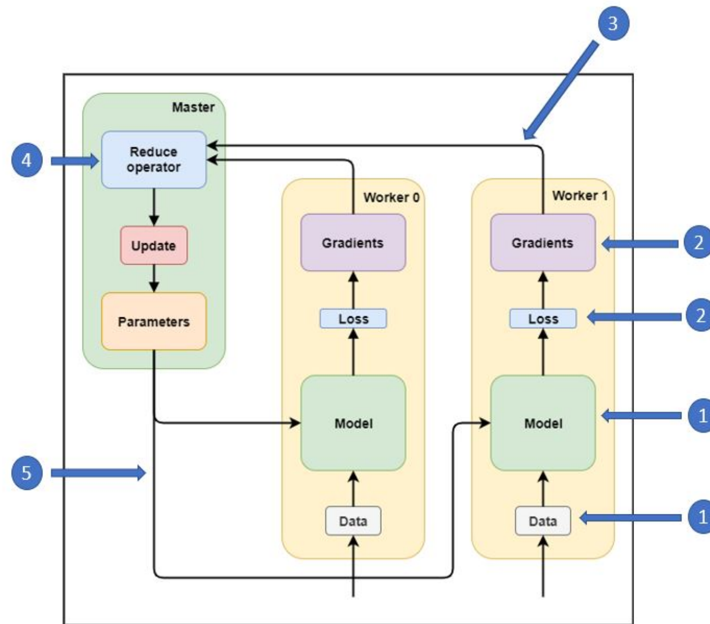

Figure 4.2 Centralized training method using Master-allreduce communication algorithm

The training process using the centralized communication method can be described as follows.

Run the training script:

1. Load a replica of the model and data across all workers.
2. Run forward pass and backward pass (without performing parameter update) on each worker.
3. Send the locally calculated gradients to the centralized master.
4. Reduce gradients and update the model parameters with new aggregated gradients.
5. Broadcast the new parameters to all workers.

Repeat the process.

4.4 Decentralized method

Ring all-reduce algorithm is one of the widely used algorithms for decentralized distributed training. Each worker takes a chunk of mini-batch data from a global mini-batch and trains on it for a step. Once all the workers finish the same step, the workers then exchange the gradients in a ring manner performing an aggregate operation in the process (sum, average are the usual aggregate operations). There are two phases in the Ring all-reduce algorithm. The first phase called the share-reduce phase, and the second phase called

the share-only phase.

In the first share-reduce phase, each worker p, communicates gradients with only the worker $(r + 1)\%r$ (where r is the worker rank). Suppose if we have 4 workers, worker 1 communicates with worker 0, worker 2 communicates with worker 1, worker 3 communicates with worker 2 and so on. This way of communication creates a ring pattern. Therefore, its called Ring all-reduce communication algorithm.

This phase is illustrated in the figure 4.3.

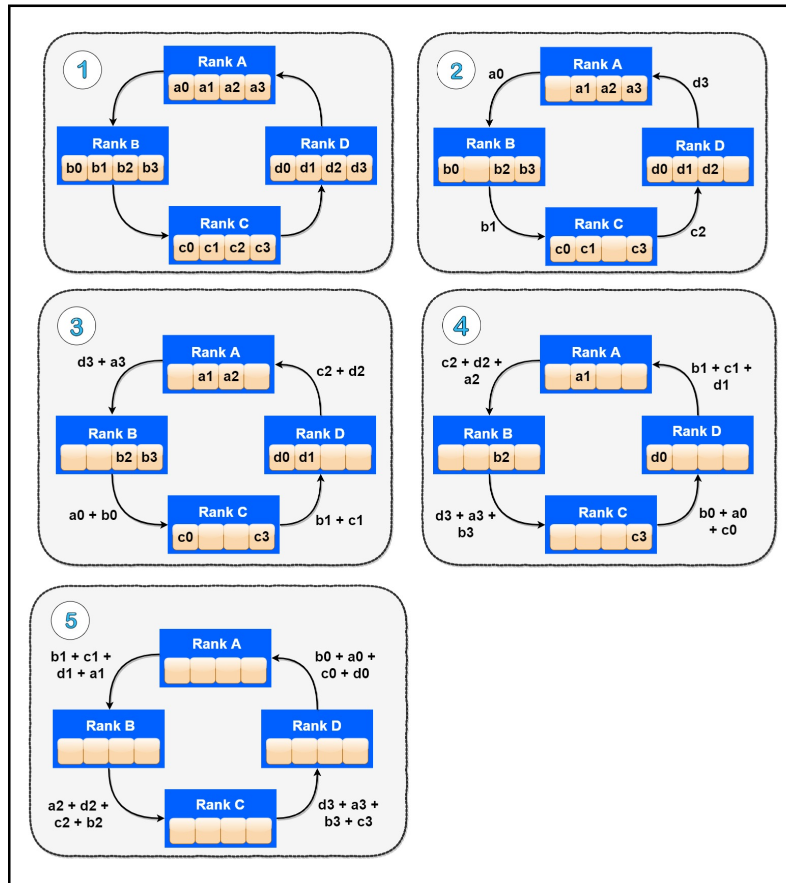

Figure 4.3 Share reduce phase of the Ring all-reduce algorithm

Each worker has a data buffer of n elements. The first worker starts sending the data from the index i, where i is the length of the data array n divided by r. The data on workers A, B, C, and D will look as illustrated in step 1 of the figure 4.3. In the first iteration, worker A sends a0 to B, worker B sends b1 to C, worker C sends c2 to D, worker D sends d3 to A. After this step, the data across the workers is shown in step 2 of the figure 4.3. After receiving the data, in the next iteration, worker A adds the received element d3 to the next element a3 and sends the result to B, worker B adds b0 to the received element a0 and sends to C, worker C

adds c1 to the received element b1 and sends to D, worker D adds d2 to the received element c2 and sends to A. After this iteration, the output will look as shown in step 3 of the figure 4.3.

This process is repeated until there are no more elements left in each worker. The final rest of the Ring all-reduce algorithm is illustrated in step 5 of the figure 4.3. After all the iterations, each worker will hold a chunk of the result of the reduced (sum) operation. The chunks need to be distributed across all the workers. To do this, the algorithm performs the share-only phase.

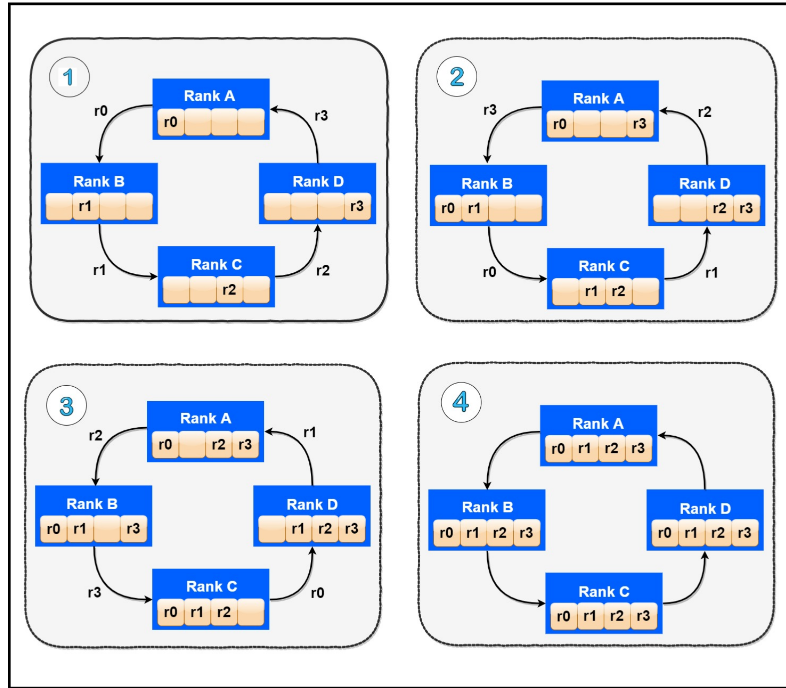

Figure 4.4 Share only phase of the Ring all-reduce algorithm

In the second phase, the aggregrate results accumulated across different workers is shared across all the workers using the same process as in the first phase without the aggregate operation. That is, the result chunks are shared in ring manner without reducing them.

The final data after performing the ring communciation method results is described as follows. The data on worker 1 $a1 + b1 + c1 + d1$ is referred as $r1$. The data on worker 2 $a2 + b2 + c2 + d2$ is referred as $r2$ and so on. We define a general term for this as, $rj = aj + bj + cj + dj$, where j is 0, 1, 2, and 3 workers respectively.

In the first iteration, worker A sends r0 to B, worker B sends r1 to C, worker C sends r2 to D, worker D sends r3 to A. After this iteration, the result will look as shown in step 1 of the figure 4.4. This process is repeated until each process receives all chunks of the result. The rest of the iterations are shown in from the

steps 2 through 4 in the figure 4.4.

The de-centralized training process is shown in figure 4.5.

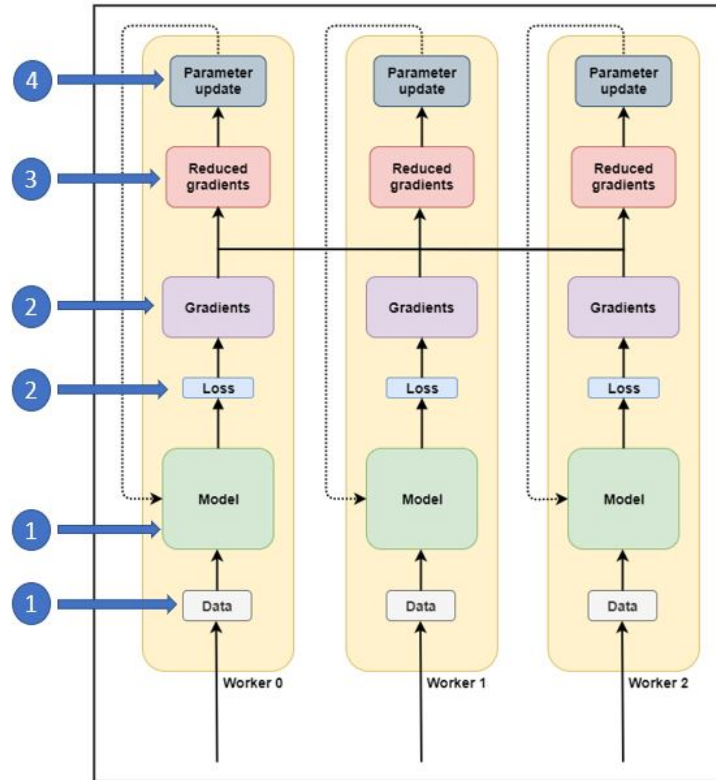

Figure 4.5 De-centralized training method using Ring-allreduce communication algorithm

The training process using the de-centralized communication method can be described as follows.

Run the training script:

1. Load a replica of model and data across all workers.
2. Compute forward pass and backward pass (without parameter update) on each worker.
3. Perform de-centralized gradient reduces operation across all workers.
4. Update model parameters with the reduced gradient on each worker locally.

Repeat the process.

4.5 Drawbacks of the traditional distributed training methods

If the model parameters are huge, the centralized training method suffers from a bottleneck at the master since all the workers communicate through the maste. The network quickly gets saturated with a lot of messages and causes latency to communicate other messages between the workers and the master. As the number of workers increases, the bottleneck increases even more. Another drawback with this method is, if the data size is large, all the data cannot fit into the memory at once.

The decentralized training method addresses these problems by dividing the mini-batch into local mini-batches, however, if the bandwidth of the network is less, the workers end up waiting for a long time.

We will address the drawbacks of both centralized and decentralized training methods in our proposed training method in the next chapter 5.

Part II

Decentralized training with a centralized parameter update training method

Chapter 5

Proposed training method

In this part, we will discuss our proposed decentralized training method that overcomes the drawbacks of the traditional distributed training algorithms, that is much suitable for Brain Imaging applications. Then we discuss how to set up a Multi-node Multi-GPU cluster, its configuration and deploying distributed jobs. We will then evaluate our proposed method on a Multi-node Multi-GPU cluster with various datasets and DNN models and analyze the results.

We call our proposed training method as **"Decentralized training with a centralized parameter update "** training method. The method combines both the decentralized and centralized methods overcoming their drawbacks.

Our proposed method adds an additional centralized parameter update step after each iteration that broadcasts model parameters from a specific worker across all workers ensuring parameter consistency across the workers after each iteration. This additional step prevents the model parameters to diverge due to multiple local parameter udpates on each worker.

The figure 5.1 illustrates our proposed training method. We used the same Ring all-reduce decentralized communication method for communicating gradients and parameter updates across workers. Refer to the section 4.4 for information about the Ring all-reduce communication algorithm.

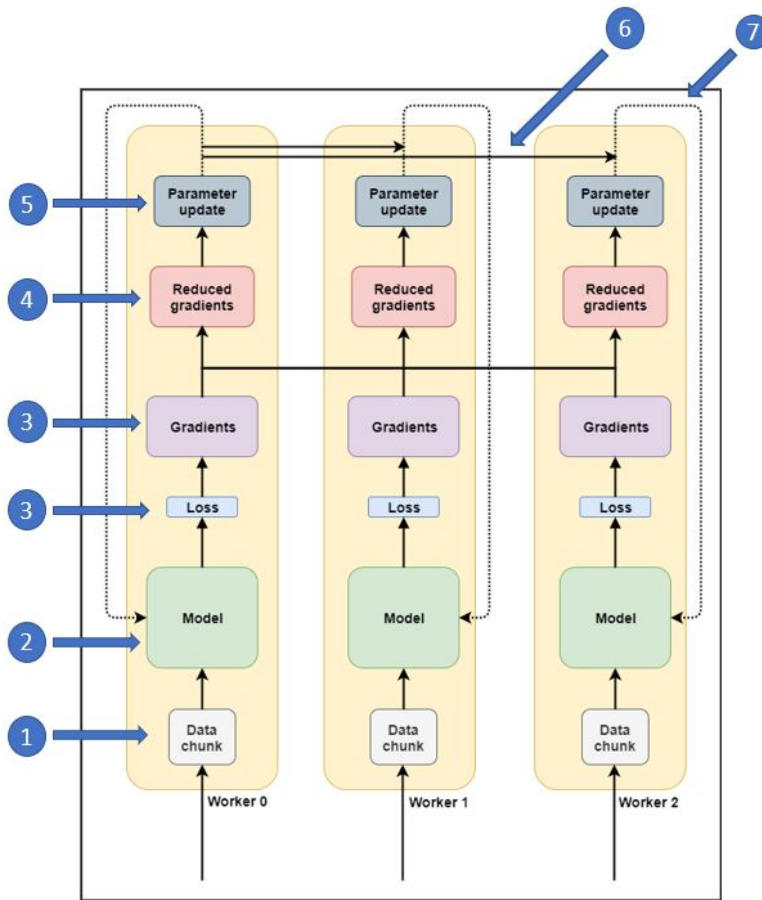

Figure 5.1 Proposed training method that combines both the centralized and decentralized training methods

The various training steps involved in our proposed training method are described as follows.

Run the training script:

1. Load a chunk of the dataset onto each worker.
2. Load a replica of the model onto each worker.
3. Compute forward pass and backward pass (without parameter update) on all workers.
4. Perform a decentralized gradient to reduce operation across all workers.
5. Update the model parameters locally on each worker.
6. Broadcast the model parameters from worker 0.
7. Update model parameters on rest of the workers with the values received from worker 0.

Repeat the process.

The various training steps involved in our proposed training method are illustrated in the figure 5.1 numbered

through 1 to 7. The additional parameter step after each iteration requires an additional $N * (R-1)$ communications. Therefore, our proposed method has $2*E*(N/R)*(R-1)+(N*(R-1))$ communications, where N is the number of model parameters on each worker, R is the number of workers, and E is the number of steps in each epoch.

This additional step introduced by our proposed training method is a hyper-parameter that can be customized according to the dataset. The step can be configured to run after cetain number of iterations instead of running after each iteration. This could significantly reduce the number of communnications. In our work we ran the centralized parameter update step (step 6 in 5.1) after each epoch because we noticed that this gave the best results. However, it is not mandatory to do so.

5.1 Performance analysis

Int this section we will compare the performance of our proposed method by analyzing the number of theoritical communications required by the traditional centralized, decentralized methods and our proposed training method.

In the centralized training, each worker R sends N elements to $(R-1)$ workers. The parameter server after performing the reduce operation sends the result back to all workers. This takes the same number of communications. This results in a total number of $2xNx(R-1)$ communications.

In the decentralized training, during the share-reduce phase, each worker R sends N/R elements to $(R-1)$ workers. This results in $(N/R)*(R-1)$ operations. In the share-only phase, each worker again sends N/R data to $(R-1)$ workers resulting in $(N/R)*(R-1)$ communications. Therefore, in total we have $2*(N/R)*(R-1)$ communications.

Our method requires $(R-1)$ communications in each epoch for communicating N elements. This results in an additional $N*(R-1)$ communications for each epoch, in addition to $2*(N/R)*(R-1)$ from the traditional decentralized method. Therefore, in total we have $(2*E*(N/R)*(R-1)+(N*(R-1))$ communications. From our analysis, you can clearly see that our proposed method out performs traditional centralized and decentralized training methods. This is illustrated in the tabular column 5.1.

No. of elements in each node (N)	No. of nodes (R)	Number of communications in a centralized training method (per epoch)	Number of communications in a decentralized training method (per epoch)	Number of communications in our proposed training method (per epoch)
1	2	20	10	11
1	4	60	15	18
1	8	140	17.5	24.5
1	50	980	19.6	68.6
1	100	1980	19.8	118.8
1	500	9980	19.96	518.96
1	1000	19980	19.98	1018.98
4	2	80	40	44
4	4	240	60	72
4	8	560	70	98
4	50	3920	78.4	274.4
4	100	7920	79.2	475.2
4	500	39920	79.84	2075.84
4	1000	79920	79.92	4075.92
8	2	160	80	88
8	4	480	120	144
8	8	1120	140	196
8	50	7840	156.8	548.8
8	100	15840	158.4	950.4
8	500	79840	159.68	4151.68
8	1000	159840	159.84	8151.84
50	2	1000	500	550
50	4	3000	750	900
50	8	7000	875	1225

50	50	49000	980	3430
50	100	99000	990	5940
50	500	499000	998	25948
50	1000	999000	999	50949
100	2	2000	1000	1100
100	4	6000	1500	1800
100	8	14000	1750	2450
100	50	98000	1960	6860
100	100	198000	1980	11880
100	500	998000	1996	51896
100	1000	1998000	1998	101898
500	2	10000	5000	5500
500	4	30000	7500	9000
500	8	70000	8750	12250
500	50	490000	9800	34300
500	100	990000	9900	59400
500	500	4990000	9980	259480
500	1000	9990000	9990	509490
1000	2	20000	10000	11000
1000	4	60000	15000	18000
1000	8	140000	17500	24500
1000	50	980000	19600	68600
1000	100	1980000	19800	118800
1000	500	9980000	19960	518960
1000	1000	19980000	19980	1018980

Table 5.1 Tabular column comparing theoretical communications in centralized, decentralized, and proposed training method between nodes for an epoch

Chapter 6

Setup

6.1 Data acquisition and preprocessing

6.1.1 HCP dataset

The data used in this work is acquired from the WU-Minn Human Connectome Project (HCP) consortium of S1200 release [17] [6]. The dataset consists of 3T MR Imaging scans of 1206 healthy young adult subjects acquired from 2012 to 2015. The dataset has 3T structural scans for 1113 subjects, 46 subjects have 3T HCP protocol Retest data, 184 subjects have multimodal 7T MR Imaging data.

We used T1-weighted MRI data and Diffusion Tensor Imaging (DTI) data for this work. The T1 data is acquired with TR=2.4 s, TE=2.14 ms, and a voxel size of 0.7 mm isotropic parameters. The DTI data is acquired with TR=5.220s, TE=89.5 ms, and a slice thickness of 1.25 mm parameters. The Diffusion-Weighted (DW) data consists of 3 shells of b-1000, 2000, and 3000, 300 s/mm^2 with an approximately equal number of shells within each acquisition run.

The Imaging data is pre-processed through a series of steps of skull removal, motion correction, slice time correction, and spatial smoothing using the FSL FEAT tool [5] [3]. The T1 weighted images are then registered to the DTI b=0 space using FMRIB's Linear Image Registration tool [1] [2]. In addition to this, 6 Functional Anisotropy (FA) images and a b-value of 0 is used along with the 90 b-values data. Since all the data is mapped to subject's b0 space, the resultant images DTI, T1, and FA images all have a voxel-wise one-to-one correspondence.

Finally from the pre-processed data, we extracted 90 b-values between 1000 and 2000 and concatenated within Sagittal direction with 6 FA images and 1 b=0 image resulting in 97-dimensional data. The resultant multi-modal and multi-dimensional DTI + FA data is converted to "numpy float64" format. Similarly, the target T1 data for all subjects is concatenated in Sagittal direction and represented in "numpy float64" format.

The resultant data is divided into two datasets of one subject and 15 subjects datasets. The one subject dataset has a shape of (145, 174, 145, 97) and the 15 subjects dataset has a shape of (2175, 174, 145, 97). The datasets are further normalized using zero mean and unit variance scaling algorithm. Finally, a zero-padding is added to the dataset in the X and Y axis resulting in a (145, 192, 192, 97) and (2175, 192, 192, 97) shape.

The DTI + FA data is used as the input dataset and the T1 data are used as targets in this work. The datasets are divided into multiple chunks of data for as illustrated in the below tabular column.

	One subject	**15 subjects**
Single GPU	Dataset: (145, 174, 145, 97) Labels: (145, 174, 145, 1)	Dataset: (2175, 174, 145, 97) Labels: (2175, 174, 145, 1)
Multi-GPU	Dataset: (145, 174, 145, 97) Labels: (145, 174, 145, 1) Ranks 0 to 2: (36, 174, 145, 97) Labels: (36, 174, 145, 1) Rank 3: (37, 174, 145, 97) Labels: (37, 174, 145, 1)	Dataset: (2175, 174, 145, 97) Labels: (2175, 174, 145, 1) Ranks 0 to 2: (543, 174, 145, 97) Labels: (543, 174, 145, 1) Rank 3: (546, 174, 145, 97) Labels: (546, 174, 145, 1)
Cluster (Node 1- 2 GPUs, Node 2 - 2 GPUs)	Dataset: (145, 174, 145, 97) Labels: (145, 174, 145, 1) Ranks 0 to 2: (36, 174, 145, 97) Labels: (36, 174, 145, 1) Rank 3: (37, 174, 145, 97) Labels: (37, 174, 145, 1)	Dataset: (2175, 174, 145, 97) Labels: (2175, 174, 145, 1) Ranks 0 to 2: (543, 174, 145, 97) Labels: (543, 174, 145, 1) Rank 3: (546, 174, 145, 97) Labels: (546, 174, 145, 1)

Cluster (Node 1 - 4 GPUs, Node 2 - 4 GPUs)	Dataset: (145, 174, 145, 97) Labels: (145, 174, 145, 1) Ranks 0 to 6: (18, 174, 145, 97) Labels: (18, 174, 145, 1) Rank 7: (19, 174, 145, 97) Labels: (19, 174, 145, 1)	Dataset: (2175, 174, 145, 97) Labels: (2175, 174, 145, 97) Ranks 0 to 6: (271, 174, 145, 97) Labels: (271, 174, 145, 1) Rank 7: (272, 174, 145, 97) Labels: (272, 174, 145, 1)

Table 6.1 HCP dataset one subject and 15 subjects chunks

6.1.2 MNIST dataset

The MNIST data [4] used in this work is downloaded from the Yann LeCun MNIST website. The MNIST dataset is a collection of 70,000 handwritten digits of greyscale images from 0 to 9. The dataset is divided into a training set of 60,000 images and a test set of 10,000 images. The original black and white images are normalized to fit in a 20 x 20-pixel box. Using the anti-aliasing technique, the black and white images are converted to greyscale images. The resulting images are then centered, padded and scaled to 28 x 28 size by using the center of mass of the pixels. The labels are one-hot encoded into a vector of 10 dimensions.

The training and test sets are represented in a single file of "numpy float64" format. The resultant training input dataset has a shape of (60000, 28, 28, 1), the training labels dataset has a shape of (60000, 10). Similarly, the test input dataset has a shape of (10000, 28, 28, 1) and the test dataset labels has a shape of (10000, 10). The MNIST dataset is used as a benchmark to test our propoed training method with the cluster.

	MNIST dataset
Single GPU	Dataset: (60,000, 28, 28, 1) Labels: (60,000, 10)
Multi-GPU	Dataset: (60,000, 28, 28, 1) Labels: (60000, 10) Each chunk: (15000, 28, 28, 1) Labels: (15000, 10)

Cluster (Node 1- 2 GPUs, Node 2 - 2 GPUs)	Dataset: (60,000, 28, 28, 1) Labels: (60,000, 10) Each chunk: (15000, 28, 28, 1) Labels: (15000, 10)
Cluster (Node 1 - 4 GPUs, Node 2 - 4 GPUs)	Dataset: (60,000, 28, 28, 1) Labels: (60,000, 10) Each chunk: (7500, 28, 28, 1) Labels: (7500, 10)

Table 6.2 MNIST dataset chunks

6.2 Deep Neural Network models

Convolutional Neural Networks (CNNs) is one of the widely used models in Brain Imaging because of their capability to extract hierarchical features. Since it is widely used in many applications, we evaluated our proposed training method on CNN models. We used two CNN models in this work.

1. A simple Convolutional Neural Network (Simple-CNN)
2. U-shaped Convolutional Neural Network (U-Net)

6.2.1 Simple-CNN

The simple-CNN consists of 9 layers arranged in sequential order. The output of one layer is connected to the input of another layer and the gradients flow sequentially from one layer to other layers.

The structure of the neural network is shown in figure 6.1.

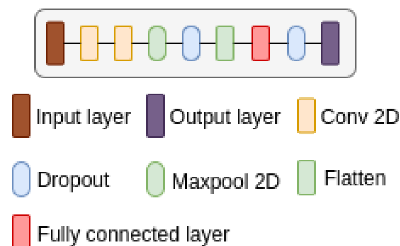

Figure 6.1 Simple CNN model

The hyper-parameters of the model are listed in the tabular column 6.3.

Layer type	Hyper-parameters	Output shape
Input		(None, 28, 28, 1)
Layer 1	3 x 3 conv2D, 32 filters ReLU activation function	(None, 26, 26, 32)
Layer 2	3 x 3 conv2D, 64 filters ReLU activation function	(None, 24, 24, 64)
Layer 3	2 x 2 maxpool with stride 1	(None, 12, 12, 64)
Layer 4	25% dropout	(None, 12, 12, 64)
Layer 5	Flatten	(None, 9216)
Layer 6	Fully connected layer with 128 hidden units ReLU activation function	(None, 128)
Layer 7	50% dropout	(None, 128)
Output	Fully connected layer with 10 hidden units Softmax activation function	(None, 10)

Table 6.3 Simple-CNN model parameters and output shapes

6.2.2 U-Net

The second CNN model that we used in this work is the most popular segementation model called U-Net [8]. The structure of the U-Net CNN model is much more complicated than the CNN model what we discussed in the previous section. The U-Net model has additional vertical layer connections instead of the usual sequential layer connections.

The model consists of two parts. The first part is an encoder network and the second part is an decoder network. The encoder network uses convolution 2D to extract features from inputs and downsamples the input into a low-dimensional feature space. The second part of the network upsamples the low-dimensional features and maps to outputs. This behavior is same called an encoder-decoder network.

This is different from the traditional autoencoder model structure in which the inputs are encoded and decoded into itself. This kind of model representation allows the model to converge faster with fewer data samples.

The model also has an additional connection called skip-connection formed by concatenating the output of the encoder layer to the input of the decoder layer. The skip-connection allows low-level features to directly

flow to the high-level layers. This prevents the low-level features vanishing in deeper layers by constantly reusing them by concatenating with high-level features, and the second is, it allows an alternate path for the gradients to flow. The skip-connections allows the model to converge faster with less amount of data. In applications where very little data is available, such models play a pivotal role. That's why it is one of the widely used models in Brain Imaging.

The structure of U-Net is shown in figure 6.2.

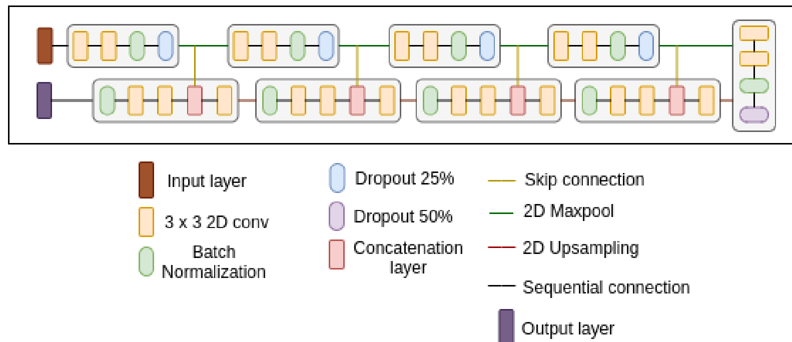

Figure 6.2 U-Net model

The model in total has 9 blocks. The first four blocks are part of the encoder network, the fifth block is called a bridge block that connects the encoder and decode parts, the last four blocks are part of the decoder network. Each encoder block has two Convolution 2D layers, followed by a Batch Normalization, 25% Dropout, and a Maxpool 2D layer. The Maxpool layer scales down the output features into half retaining the depth. Therefore, the deeper layers have more depth but less width and height.

The bridge layer has two Convolutional 2D layers, followed by a Batch Normalization layer and a 50% Dropout layer. The bridge block is connected to a series of upsampling blocks. We have 4 such blocks in our model. The output of each block upsampled to increase their dimensions. Each upsample block has a convolution layer followed by a concatenate layer that connects to outputs of a downsample block, then 2 convolutions 2D layer and a dropout layer. The last layer in U-Net converts the output to the target's shape.

The hyper-parameters of the model are listed in the tabular column 6.4.

Layer type	Hyper-parameters	Output shape
Input		(None, 192, 192, 97)
Block 1	3 x 3 conv2D, 32 filters 3 x 3 conv2D, 32 filters	(None, 192, 192, 32)

	Batch norm on axis 3	
	25% dropout	
	2 x 2 max pool with stride 1	(None, 96, 96, 32)
Block 2	3 x 3 conv2D, 64 filters 3 x 3 conv2D, 64 filters	(None, 96, 96, 64)
	Batch norm on axis 3	
	25% dropout	
	2 x 2 max pool with stride 1	(None, 48, 48, 64)
Block 3	3 x 3 conv2D, 128 filters 3 x 3 conv2D, 128 filters	(None, 48, 48, 128)
	Batch norm on axis 3	
	25% dropout	
	2 x 2 max pool with stride 1	(None, 24, 24, 128)
Block 4	3 x 3 conv2D, 256 filters 3 x 3 conv2D, 256 filters	(None, 24, 24, 256)
	Batch norm on axis 3	
	25% dropout	
	2 x 2 max pool with stride 1	(None, 12, 12, 256)
Block 5	3 x 3 conv2D, 512 filters 3 x 3 conv2D, 512 filters	(None, 12, 12, 512)
	Batch norm on axis 3	
	50% dropout	
Block 6	Upsample	(None, 24, 24, 512)
	3 x 3 conv2D, 256 filters	(None, 24, 24, 256)
	Concatenate with block 4 on axis 3	(None, 24, 24, 512)
	3 x 3 conv2D, 256 filters 3 x 3 conv2D, 256 filters	(None, 24, 24, 256)
	Batch norm on axis 3	
Block 7	Upsample	(None, 48, 48, 256)
	3 x 3 conv2D, 128 filters	(None, 48, 48, 128)
	Concatenate with block 3 on axis 3	(None, 48, 48, 256)

	3 x 3 conv2D, 128 filters 3 x 3 conv2D, 128 filters	(None, 48, 48, 128)
	Batch norm on axis 3	
Block 8	Upsample	(None, 96, 96, 128)
	3 x 3 conv2D, 64 filters	(None, 96, 96, 64)
	Concatenate with block 2 on axis 3	(None, 96, 96, 128)
	3 x 3 conv2D, 64 filters 3 x 3 conv2D, 64 filters	(None, 96, 96, 64)
	Batch norm on axis 3	
Block 9	Upsample	(None, 192, 192, 64)
	3 x 3 conv2D, 32 filters	(None, 192, 192, 32)
	Concatenate with block 1 on on axis 3	(None, 192, 192, 64)
	3 x 3 conv2D, 32 filters 3 x 3 conv2D, 32 filters	(None, 192, 192, 32)
	Batch norm on axis 3	
Output	1 x 1 conv2D, 1 filter	(None, 192, 192, 1)

Table 6.4 U-Net model parameters and output shapes

6.3 Cluster setup

In this section, we will discuss the hardware and software configuration and setup for the Multi-node Multi-GPU cluster.

In layman terms, a cluster is a group of machines connected over a network to perform a computational task. There are two types of clusters.

1. **Centralized cluster system**

In a centralized cluster setup, there are two kinds of nodes based on the operation they perform. One is a master node and the other is a compute node. The master node takes a computation request from the user and performs the computation on the cluster node. The master node acts as a gateway to communicate and run compute jobs. The master node holds the data or sometimes there is a third kind of node to hold the data known as a data node.

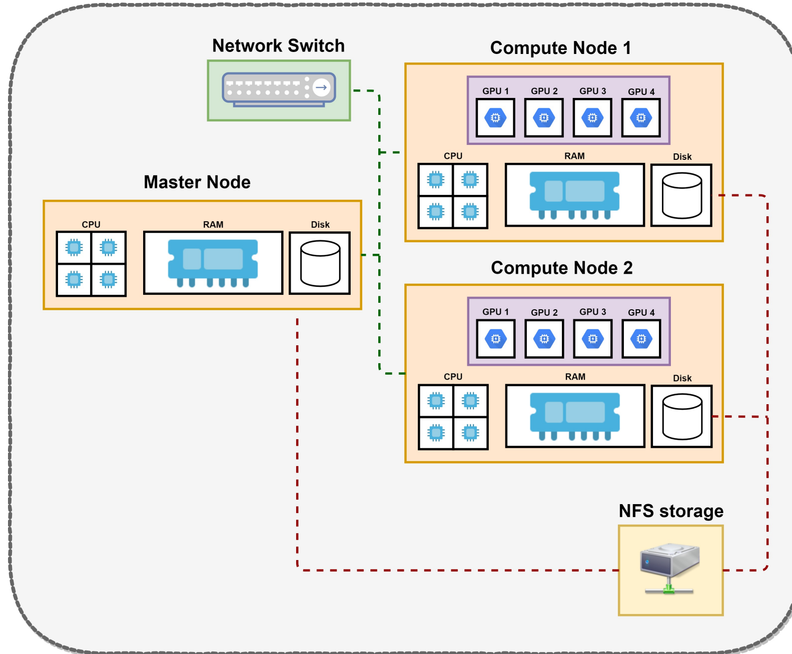

Figure 6.3 Cluster architecture

2. **De-centralized cluster system**

In a decentralized cluster setup, there is only one type of node. Each node acts as a master node and a compute node. i.e there is no centralized dedicated node that distributes tasks to each node. The advantage of the decentralized system is, the master node doesn't get overwhelmed by thousands of communication requests from the compute nodes.

In this work, we use the decentralized cluster model to build a Multi-node Multi-GPU cluster. The cluster has a master node and 2 compute nodes. Each compute node has 4 GPUs in it. Each compute node has 4 GPUs in it. Each GPU in the compute node is configured to run an MPI process. Therefore, in total, we have 8 ranks to run the compute tasks.

The idea of the master node used here is slightly different than the one described before. The master node here acts as an entry point to place a computation request on the cluster. It takes a computation request from the user and performs a computation job on the compute nodes. In addition to this, it hosts a shared file system with the compute nodes.

Our cluster architecture design is shown in figure 6.3.

The worker nodes (Compute Node 1 and Compute Node 2) shown in the figure 6.3 are connected to the master node through a 10 Gigabit router. Both the master and worker nodes communicate with each other

using TCP/IP protocol through the router. The datasets are located on the master node and are shared across the network nodes through an NFS shared memory system. Each node can access this dataset as if it is physically available on its disk.

6.3.1 System configuration

The master node has 2 Intel Xeon Gold 6148 processors with 20 cores. The node can run 40 threads simultaneously on its 20 cores. It operates at a 2400 MHz clock speed. Each of the two compute nodes have 4 Nvidia GeForce GTX 1080 Ti Graphic cards with 12 GB GPU memory and a 64 GB physical memory (RAM). The GTX 1080 Ti operates at 1480 MHz clock speed. It has 3584 CUDA cores operating. Each has can run 11.34 Teraflops of computation.

Each node is configured with the same number and type of GPU in the cluster for optimal system performance.

6.3.2 Network File System (NFS) setup

The file system on the master is shared across all the workers using a Network File System (NFS). Therefore, each node can access the data on the master as if it is located on its file system without any interruption.

The master node (Which is sharing its file system) requires a package called “nfs kernel server”, to launch the shared network file system server. The workers require a client package called “nfs common” to mount the remote file system.

Suppose “/data/Cluster/” is the folder that needs to be shared across the workers to run MPI jobs. First, the folder needs to be available on the network for the nodes to access it remotely. Therefore the folder path needs to be added in the “exportfs” configuration. Once the folderis available on the network, it can be mounted on the workers as follows.

Script file output.markdown.lua not found

6.3.3 Shared user setup

To run compute jobs on the cluster, each node needs access to files on other nodes without requiring any additional privileges. That’s why we created a separate user called “cluster” on all the nodes. This enables

the nodes in the cluster to communicate with each other without requiring a password.

The user account is configured as follows.

1. Create a user on each node as follows.

Script file output.markdown.lua not found

2. Add the user to sudoers list.

Script file output.markdown.lua not found

6.3.4 Networking setup

Each node in the cluster requires an unique identifier to communicate with each other. IP address is usually used for this task. Since IP addresses can change due to network resets, using them directly is not a reliable option. Therefore we used hostnames. Even when the IP addresses change, all we have to change the IP address to hostname mapping and the whole setup still works without disturbing any other configuration setup.

The nodes in the cluster are configured in a private network. Therefore compute jobs in the cluster do not stop running even when the network is down. For this, the host file of the master and worker nodes are configured as follows.

Script file output.markdown.lua not found

To allow passwordless commmunication among the nodes, each node's public key is added to the list of authorized keys on rest of the nodes. This allows passwordless SSH access to nodes in the cluster.

1. Generate an ssh key using the following command.

Script file output.markdown.lua not found

2. Copy the public key onto other nodes and add them to the list of their authorized keys using the following command.

Script file output.markdown.lua not found

where hName is the host name of the nodes.

Each node in the cluster should be able to connect to each other without requiring a password now.

6.3.5 MPICH setup

The **MPICH v3.3** library offers MP-3 standard APIs to execute parallel programs across multiple nodes. The "mpiexec" command is used to execute distributed jobs on the compute nodes. The "MPI COMM WORLD" communicator provides communication between the processes executing across multiple nodes. MPICH handles the in-memory operations of the distributed parallel processes.

We used the default configuration that came with the "MPICH" ubuntu library. The package can be installed as below.

Script file output.markdown.lua not found

6.3.6 Cluster process manager setup

Hydra [22] is a process management system to run and manage parallel jobs across multiple nodes. It works with multiple communication daemons such as ssh, slurm, sge, pbs. The process manager schedules and manages computaiton jobs on the cluster.

To run a computation job, use mpiexec [7] command. It invokes the hydra process manager by default.

Script file output.markdown.lua not found

where,
f argument indicates the compute nodes to run the MPI processes.
n argument indicates the number of MPI processes to run.

The hosts file for the above job is specified as follows. Script file output.markdown.lua not found
To schedule multiple jobs, the hosts file can be divided as follows.
First MPI job command Script file output.markdown.lua not found

Hosts file for the first job Script file output.markdown.lua not found
Second MPI job command Script file output.markdown.lua not found

Hosts file for the second job Script file output.markdown.lua not found
The method discussed in the previous section can be used to schedule multiple jobs at the same time. Based on the resource requirements for each job, the number of processes can be adjusted accordingly using a simple shell script.

6.3.7 Additional libraries setup

1. NCCL 2.0

We used Nvidia's NCCL 2.0 library [26] [11] to communicate parameters and gradients between GPUs. NCCL is a multi-GPU inter-GPU communication library. For clusters, it uses Nvidia's GPU Direct RDMA technology to communicate with each other nodes. It works with Ethernet and Infiniband networks. Within the same system, it used PCIe, NVLink (if available) and GPU Direct P2P technologies for inter-GPU communication. The library provides an implementation of a highly optimized ring-allreduce algorithm that can run on multi-GPU systems distributed across a network.

The library provides interfaces to run computation on GPUs similar to that of MPICH. The library can run asynchronously using Nvidia's CUDA library. The NCCL 2.0 is available on the NVIDIA website and can be downloaded as a ".deb" file. We used the NCCL 2.0 library with MPICH run training jobs.

2. MPI4py

MPI4py is a python wrapper for MPICH. It provides API functionalities to write MPI programs in python. This library is available on the "PyPi" python package manager and can be installed through pip as follows. We used MPI4PY library to write distributed MPI programs in python.

The "mpi4py" package can be installed using the below command.

Script file output.markdown.lua not found

6.3.8 Horovod setup

Horovod [13][21] is a high-level API developed on top of Tensorflow using NCCL's ring-allreduce algorithm. The Horovod library privides an easy interface to run distributed training jobs. The library manages the model and data placement on the devices automatically without needing any additional configuration information from the users. It supports model development in multiple frameworks like Tensorflow 1.15 [29], Keras [23], PyTorch [28], mxnet [25], and Tensorflow 2.0 [30].

Horovod library is available as a "PyPi" library and can be installed through pip.

Script file output.markdown.lua not found

6.4 Deep learning package manager setup

Several software packages are required to run training jobs on the cluster. Installing and setting up each node with the required packages separately can be a tedious task. Therefore, in our work we used the Anaconda distribution package manager to install and manage packages across all the workers. The package manager installs packages across the nodes on the cluster in a single shot. This is done by synchronizing software packages installed on a node across all the nodes automatically using NFS in the same way we discussed in the section 6.3.2).

The figure 6.4 shows the software architecture of the cluster.

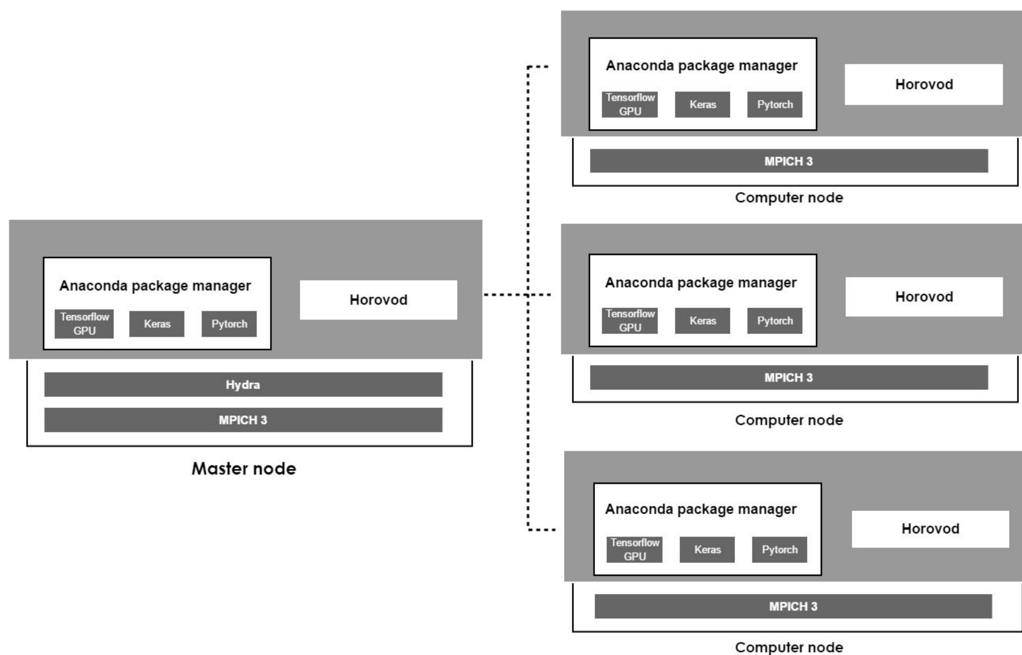

Figure 6.4 Cluster software architecute

Chapter 7

Results

7.1 Experimental setup

We considered four experimental cluster setups to evaluate our proposed training method. The setups are described below.

Setup 1 - Single GPU sytem

The Single GPU setup consists of a single GPU. It uses traditional sequential training method to run training jobs.

Setup 2 - Multi-GPU system

The Multi-GPU setup consists of 4 workers all located on the same node. Each worker runs a distributed training process on an Nvidia GTX 1080 Ti GPU. So, in total, we have 4 training processes distributed but running on the same node.

Setup 3 - Cluster (2 nodes each with 2 GPUs)

The 2 node 2 GPU setup consists of 4 workers, each node configured with 2 workers. Each worker runs a distributed training process on an Nvidia GTX 1080 Ti GPU. So, in total, we have 4 distributed training processes.

Setup 4 - Cluster (2 nodes each with 4 GPUs)

The 2 node 4 GPU setup consists of 8 workers, each node configured with 4 workers. Each worker runs a

distributed training process on an Nvidia GTX 1080 Ti GPU. So in total, we have 8 distributed training processes.

7.2 Evaluation

We evaluated our proposed training method trained using 2 CNN models on 3 different datasets.

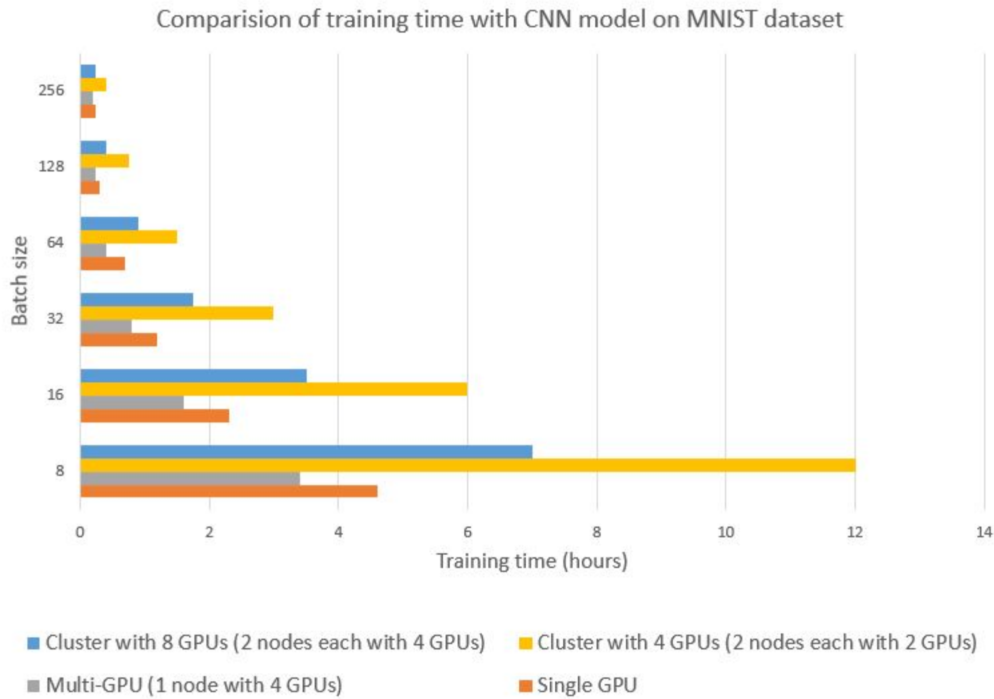

Figure 7.1 Comparison of training time with Simple CNN model on MNIST dataset

Training time (hours)				
Batch size	**Single GPU**	**Multi-GPU (1 node with 4 GPUs)**	**Cluster with 4 GPUs (2 nodes each with 2 GPUs)**	**Cluster with 8 GPUs (2 nodes each with 4 GPUs)**
8	4.6	3.4	12	7
16	2.3	1.6	6	3.5

32	1.2	0.8	3	1.75
64	0.7	0.4	1.5	0.9
128	0.3	0.25	0.75	0.4
256	0.25	0.2	0.4	0.25

Table 7.1 Comparison of Simple CNN model training time on MNIST dataset across four experimental setups

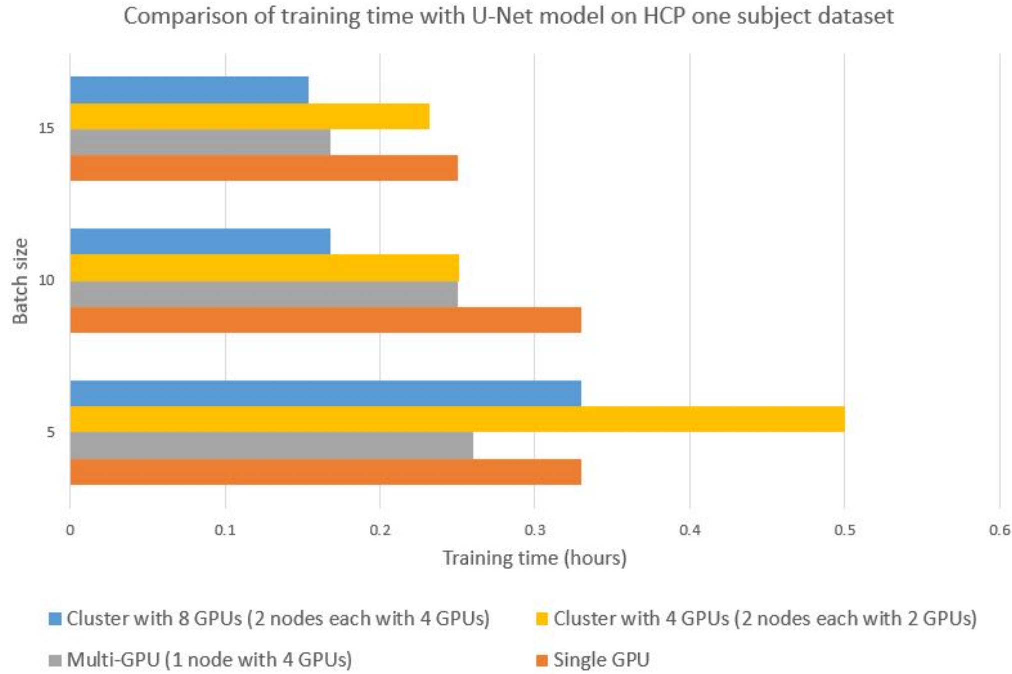

Figure 7.2 Comparison of training time with U-Net model on HCP one subject dataset

Training time (hours)				
Batch size	**Single GPU**	**Multi-GPU (1 node with 4 GPUs)**	**Cluster with 4 GPUs (2 nodes each with 2 GPUs)**	**Cluster with 8 GPUs (2 nodes each with 4 GPUs)**
5	0.33	0.26	0.5	0.33

10	0.33	0.25	0.251	0.168
15	0.25	0.168	0.232	0.154

Table 7.2 Comparison of U-Net model training time on HCP one subject dataset across four experimental setups

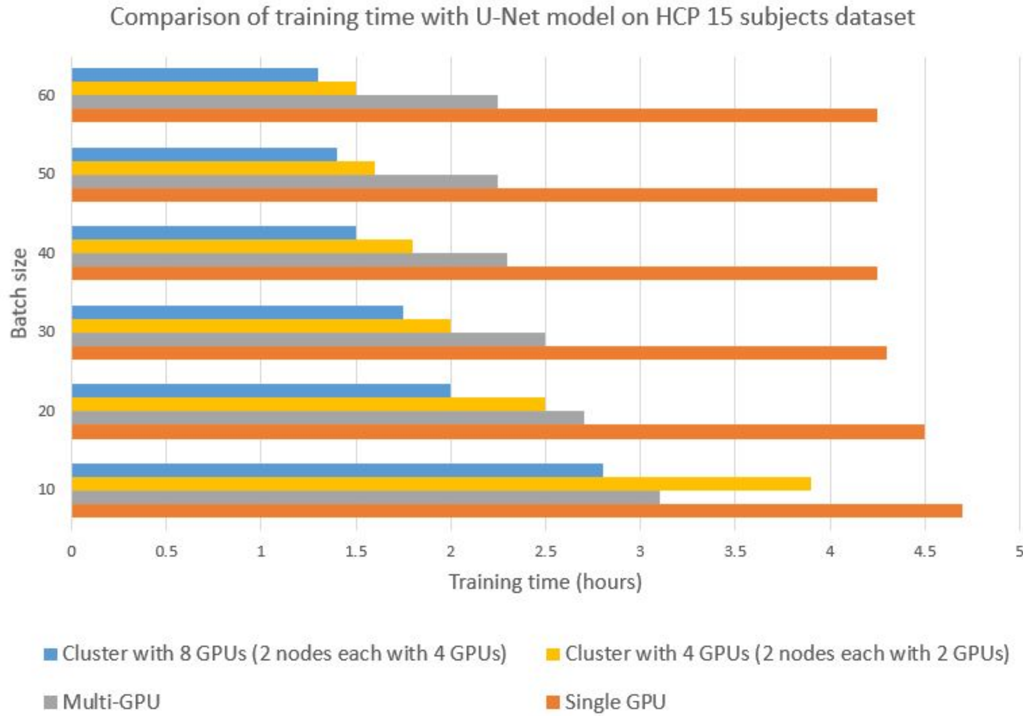

Figure 7.3 Comparison of training time with U-Net model on HCP 15 subjects dataset

Training time (hours)				
Batch size	**Single GPU**	**Multi-GPU (1 node with 4 GPUs)**	**Cluster with 4 GPUs (2 nodes each with 2 GPUs)**	**Cluster with 8 GPUs (2 nodes each with 4 GPUs)**
10	4.7	3.1	3.9	2.8
20	4.5	2.7	2.5	2

30	4.3	2.5	2	1.75
40	4.25	2.3	1.8	1.5
50	4.25	2.25	1.6	1.4
60	4.25	2.25	1.5	1.3

Table 7.3 Comparison of U-Net model training time on HCP 15 subjects dataset across four experimental setups

Chapter 8

Conclusion

In this work, we developed a novel decentralized training method that combines both centralized and decentralized training methods. Our proposed method was able to overcome the model divergence problem experienced in the traditional decentralized method and was able to reduce the communication bottleneck problem experienced with the centralized method.

We built a Multi-node Multi-GPU cluster that leveerages our proposed training method to run distributed parallel training jobs. We developed an easy to use interface to run distributed training jobs which requires only a few modifications to the existing model training code.

We also built a Keras API with our proposed method over Horovod's Keras API that provides a simple callback-based for training models in Tensorflow 2.0. The API works with Sequential, Functional, and Model subclassing APIs in Tensorflow.

We also analyzed our proposed training method with various datasets and deep learning models to find which configurations which are best suited for different CNN models and datasets.

Chapter 9

Future Work

We would like to extend our proposed training approach to train differnt DNNs like Recurrent Neural Networks (RNNs), Long short-term memory networks (LSTMs), Generative Adversarial Networks (GANs), and Deep Belief Networks (DBNs) in future.

Even though our method dramatically reduces the number of communications between the workers, we noticed that sometimes it still creates a bottleneck after certain number of training iterations. Therefore, in the future, we would like to use a high bandwidth network like InfiniBand to evaluate our method on the cluster.

REFERENCES

[1] Mark Jenkinson and Stephen Smith. "A global optimisation method for robust affine registration of brain images". In: *Medical image analysis* 5.2 (2001), pp. 143–156.

[2] Mark Jenkinson et al. "Improved optimization for the robust and accurate linear registration and motion correction of brain images". In: *Neuroimage* 17.2 (2002), pp. 825–841.

[3] Stephen M Smith et al. "Advances in functional and structural MR image analysis and implementation as FSL". In: *Neuroimage* 23 (2004), S208–S219.

[4] Li Deng. "The MNIST database of handwritten digit images for machine learning research [best of the web]". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[5] Mark Jenkinson et al. "Fsl". In: *Neuroimage* 62.2 (2012), pp. 782–790.

[6] David C Van Essen et al. "The WU-Minn human connectome project: an overview". In: *Neuroimage* 80 (2013), pp. 62–79.

[7] Pavan Balaji et al. "MPICH guide". In: *Argonne National Laboratory* (2014).

[8] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[9] Martın Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[10] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: 1609.04747 [cs.LG].

[11] Sylvain Jeaugey. "Nccl 2.0". In: *GTC* (2017).

[12] Thorsten Kurth et al. "TensorFlow at Scale: Performance and productivity analysis of distributed training with Horovod, MLSL, and Cray PE ML". In: *Concurrency and Computation: Practice and Experience* (2018), e4989.

[13] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799* (2018).

[14] *activation*. https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them. Accessed: 2019-12-04.

[15] *Backpropagation algorithm*. https://en.wikipedia.org/wiki/Backpropagation. Accessed: 2019-11-14.

[16] *backward pass*. http://andrew.gibiansky.com/blog/machine-learning/fully-connected-neural-networks. Accessed: 2019-12-04.

[17] *Connectome dataset*. https://www.humanconnectome.org/study/hcp-young-adult/data-releases. Accessed: 2019-11-14.

[18] *Delta Rule*. https://en.wikipedia.org/wiki/Delta_rule. Accessed: 2019-11-14.

[19] *Feedforward Neural Nnetwork*. https://en.wikipedia.org/wiki/Feedforward_neural_network. Accessed: 2019-11-14.

[20] *Gradient calculation*. https://en.wikipedia.org/wiki/Gradient. Accessed: 2019-11-14.

[21] *horovod*. https://github.com/horovod/horovod. Accessed: 2019-12-11.

[22] *Hydra process manager*. https://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager. Accessed: 2019-11-14.

[23] *Keras*. https://keras.io. Accessed: 2019-11-15.

[24] *mlp*. https://stackoverflow.com/questions/33649645/how-should-nodes-be-connected-in-a-neural-network. Accessed: 2019-12-04.

[25] *mxnet*. https://mxnet.incubator.apache.org. Accessed: 2019-11-15.

[26] *NCCL 2.0*. https://developer.nvidia.com/nccl. Accessed: 2019-11-15.

[27] *perceptrons*. https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975. Accessed: 2019-12-04.

[28] *pyTorch*. https://pytorch.org. Accessed: 2019-11-15.

[29] *Tensorflow 1.15*. https://www.tensorflow.org/versions/r1.15/api_docs/python/tf. Accessed: 2019-11-15.

[30] *Tensorflow 2.0*. https://www.tensorflow.org/api_docs/python/tf. Accessed: 2019-11-15.