IMPLEMENTION OF PARALLEL NEURAL NETWORKS USING MPI

PROJECT REPORT
ENGR 517- HIGH PERFORMANCE COMPUTING
ARPITA AJIT WELLING

INTRODUCTION:

Artificial Neural Networks (ANN) are computational models that are designed based on the biological brain. ANNs are used for regression as well classification problems.

ANNs comprise of 3 node layers: input layer, one or more hidden layers, and an output layer. Each node layer can have different number of nodes. The number of hidden layers and number of nodes in a hidden layer is a hyperparameter for the neural network. Generally, the number of nodes in an input layer are the number of features of the data. Output layer nodes are dependent on the problem. For example, if the problem is a prediction problem, output layer will have 1 node that gives the predicted value. If the problem is classification problem, then also the output layer can have 1 node, or it can have nodes equal to number of class labels if we use the SoftMax technique.

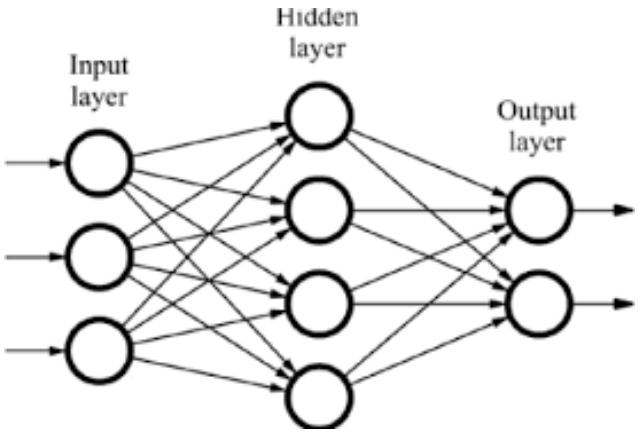

Figure 1: Structure of Neural Network

In this project, ANNs are designed and implemented in parallel. This project implements a basic neural network with 1 hidden layer. For parallelizing neural networks, there are 2 ways:

1. Data Parallelism:
   The training data is distributed among the N processes. Each process uses the same neural network but on different batch of data. After each epoch or after each pass through the batch of data, processes communicate their weight gradients, and an average of all weight gradients is used for further processing by all processes.

2. Model Parallelism:
   In model parallelism, the nodes of each layer are distributed among the processes. Each process gets the same data. After each epoch, the output needs to be gathered so that the new layer can get it as input.

   This project implements data parallelism using distributed memory.

DATASET:

Generally, a neural network is used for image dataset. But for simplicity, this project uses Pima Indians Diabetes dataset [6].

The aim of using this dataset is to predict whether a person has diabetes or not, given the following parameters:

1. Number of Pregnancies
2. Glucose concentration
3. Blood Pressure
4. Skin Thickness
5. Insulin level
6. BMI
7. Diabetes Pedigree function
8. Age

There are 112,400 instances in this dataset.

SEQUENTIAL WORKING OF NEURAL NETWORKS:

In sequential flow of neural networks, there is only one process. Therefore, the entire data is contained in that one process. The sequential working of neural networks takes place in the following manner:

1. Initialization
   In this step, all the layers of the neural network are initialized with the number of nodes. In this project, input layer has 8 nodes (dataset has 8 features), hidden layer has 7 nodes, and output layer has 1 node. According to these parameters, the weight matrix for each layer is also initialized.

2. Forward Propagation
   The weights of input layer are multiplied with the input layer values. These values are then passed through an activation function that determines the output to be passed to the hidden layer. Thus, weighted output of input layer is given as input to hidden layer. In a similar way, the input of hidden layer is multiplied by it's weights, passed through an activation function and then the output is given as input to the final layer.

3. Back Propagation
   In this step, the neural network traces back the output error and adjusts the weights of each layer using gradient decent.
   Steps 2 and 3 are repeated till the output error reaches a significant minimum value or till a certain number of epochs are completed. Here, epoch refers to one pass through the training data.

PARALLEL WORKING OF NEURAL NETWORKS:

All the steps mentioned in the sequential working remain the same for parallel execution. The difference is that the data is distributed among all the processes.

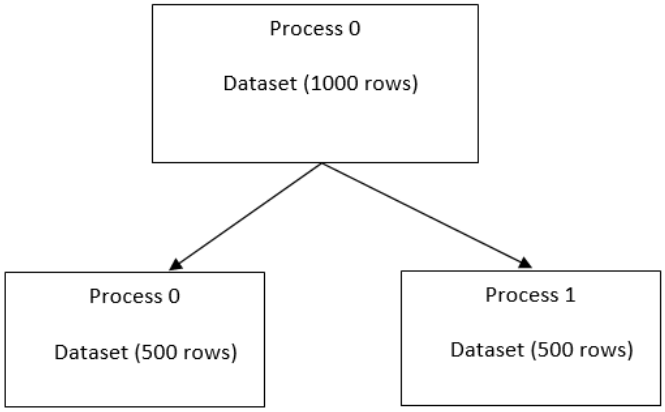

Figure 2: Distribution of data in parallel neural networks

The above figure shows how data can be distributed among 2 processes. This project implements MPI scatter operation on the training data to distribute the data.

After scattering, each process will perform forward propagation and backward propagation on its data batch. After each epoch, all processes communicate their weight gradients with each other. These weight gradients are averaged and used by all processes for the next epoch. For communication of weights, MPI Allreduce with SUM operation is used.

At the end of each epoch, MPI Barrier is used to synchronize all processes. After all epochs, each process calculates the accuracy for its data batch. This accuracy is also communicated to the root node to find the accuracy of the entire dataset. This is done by using MPI reduce operation and the average of this sum is taken at the root.

RESULT AND ANALYSIS:

All the experiments were done on Big Red 3 at Indiana University.

1. Strong Scaling

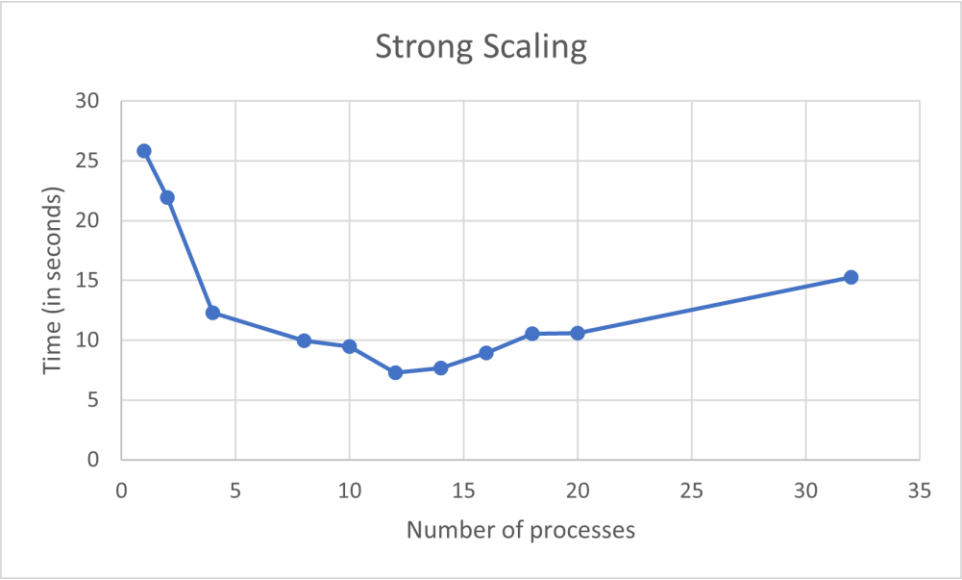

Figure 3: Strong Scaling

In Strong scaling, the size of the dataset remains the same and is equally divided among all the processes. Strong scaling helps to reduce the time to solution.

As seen in figure 3, the time to solution is decreasing as processes are increased. But due to Python overheads, after a certain number of processes, the time starts increasing.

2. Weak Scaling

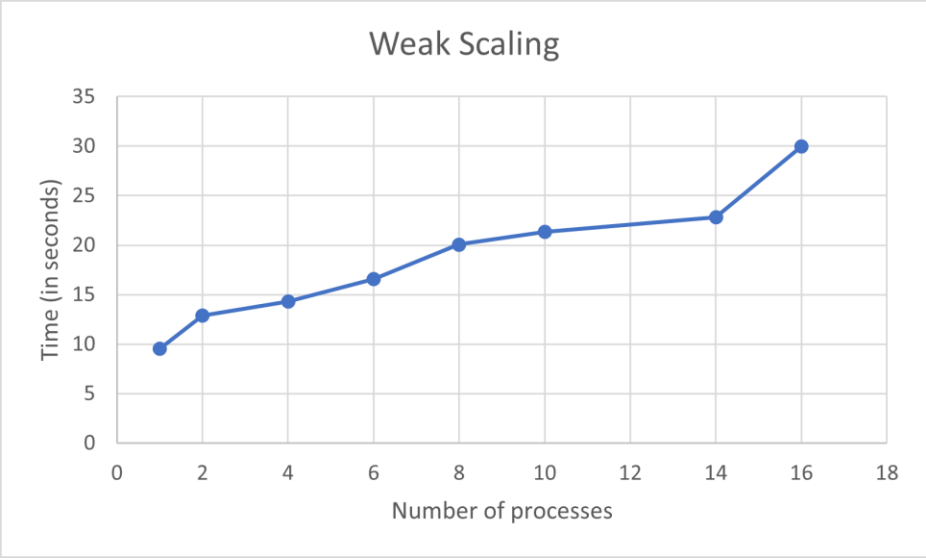

Figure 4: Weak Scaling

In weak scaling, as the number of processes increase, the dataset also increases. The time to solution should ideally remain constant.

As seen in figure 4, the time increases a bit till process 8 after which it is almost constant. For process 16, there is again a slight increase. This increase is mostly due to python overhead. Overall, parallel neural networks help in speedup. The overall accuracy of the neural network remains the same, that is, 74.89%, with some trivial improvements as the number of processes increase.

CHALLENGES AND FUTURE WORK:

The most challenging part of this project was dealing with the overheads that occurred due to python. This project was initially implemented on 10,000 instances of data. This did not give good results and thus, the dataset had to be increased. Eventually, this project is being implemented on 110,000 instances for training.

Implementing MPI in python using mpi4py was another challenge. The documentation for this library is very limited and using various coding communities was required [7,8].

In future, this work can be extended by implementing model parallelism and comparing the speedups of both the parallelizing methods. This can also be implemented by increasing the hidden layers and using larger dataset.

REFERENCES:

1. https://ieeexplore.ieee.org/abstract/document/1240376
2. https://arxiv.org/abs/1507.01239
3. https://www.academia.edu/24709347/Parallel_Training_of_Artificial_Neural_Networks_Using_Multithreaded_and_Multicore_CPUs
4. https://indjst.org/download-article.php?Article_Unique_Id=INDJST7582&Full_Text_Pdf_Download=True
5. https://pdeep.xyz/documents/Parallel_Deep_Learning_Pradeep_Singh.pdf
6. https://www.kaggle.com/uciml/pima-indians-diabetes-database
7. https://stackoverflow.com/questions/65082585/mpi4py-scatter-a-matrix
8. https://stackoverflow.com/questions/36025188/along-what-axis-does-mpi4py-scatterv-function-split-a-numpy-array
9. https://github.com/rezwanh001/Deep-Neural-Network-for-Image-Classification-Cat-Non-Cat/blob/master/Cat%2C%20Non-cat%20Image%20Classification%20with%20DNN.ipynb