Improving Neural Network Training Times using CUDA and MPI

Michael Vogt Nicholas Baard

Supervisor(s):
Hairong Bau

A project submitted in fulfillment of the requirements for High Performance Computing

Computer Science and Applied Mathematics
University of the Witwatersrand, Johannesburg

15 January 2022

Declaration

We, Michael Vogt, and Nicholas Baard, declare that this report is our own, unaided work. It is being submitted for the Adaptive Compuation and Machine Learning Project for Computer Science Honours 2021.

Michael Vogt Nicholas Baard

15 January 2022

Michael Vogt Nicholas Baard

15 January 2022

Abstract

The computational costs involved in training a neural network are extremely expensive, and the subsequent training times can be very long. This cost further increases the more data is used. Therefore, this experiment investigates potential time improvements by successfully implementing neural networks and parallelizing their training using the CUDA parallel computing platform and MPI message passing standard. The improvements using both these methods is realized, with MPI saving the most time during training.

Contents

List of Figures

Chapter 1

Introduction

1.1 Neural Network

A neural network is a machine learning algorithm that is a supervised learning method. This algorithm is powerful as it is able to handle complex, non-linear classification. A neural network is based on a perceptron, which is a replication of how information is passed from neuron to neuron in the human body. Combining multiple perceptrons creates a network where information can be passed through layers of the perceptrons, imitating the way information passes to and from the human brain.

Each feature of the data is represented as an input node. The model parameters, or weights, connect the input nodes to the perceptron. These weights act as synapses which control the strength of the information that is being passed from one perceptron to the next. The sum of the input nodes multiplied by the weights connecting the node to the perceptron is passed through a non-linear activation function such as the sigmoid function which produces an output. A bias node is added to the weighted sum A neural network is named a multi-layer perceptron as it uses multiple perceptrons as described above to make predictions. A simple feed-forward network contains three layers, an input layer, a hidden layer, and an output layer which are all made up of nodes. The input layer contains a node for each input feature of the dataset, the hidden layer contains multiple nodes that act as perceptrons, and each node receives the data from the input layer which is connected by the weights. The weighted sum of the input is passed through the activation function of the node which produces an output. These outputs are transferred to the output layer by another set of weights. The nodes in the output layer take the

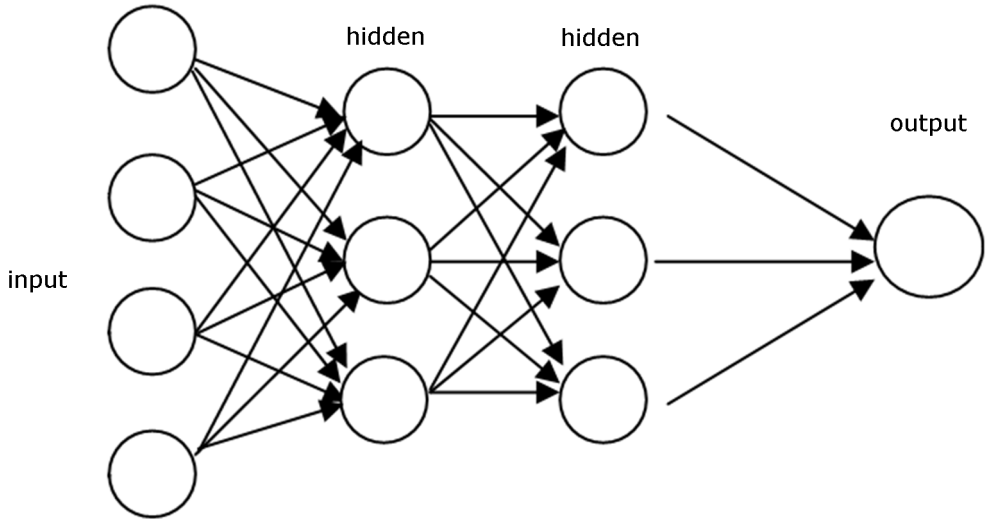

FIGURE 1.1: Feed Forward Neural Network

weighted sum of the weights and the outputs from the hidden layer and pass it through an activation function to produce an output. This output is the prediction of the model. The activation function of the output layer can be chosen specifically for the classification/regression problem as well as the number of nodes in the output layer for binary or multi-class classification.

In order to make accurate predictions, the model needs to learn the model parameters, or the weights, of the network. We do this by backpropagating the error in the prediction made by the model through the neural network to penalize the part of the network contributing to the error. This is done by partially differentiating from the output layer to the input weights. The weights of the neural network are then updated based on the partial derivatives. This is a gradient decent method of updating the weights and is done until the error converges. We train the model by randomly initializing the weights of the neural network and then make a prediction by feeding a datapoint of the training dataset into the network. After each prediction is made, the weights are updated by backpropagation. Once all datapoints of the training dataset have been passed through the network, the mean squared error is calculated and this process is repeated until the error converges of for a set amount of iterations called 'epochs'. In order for the model to avoid overfitting the data, regularization may be used. Regularization is implemented through adding

the square of the sum of the magnitude of the model parameters to the objective function. This term is multiplied by λ, a hyperparameter that controls the amount of regularization that is added. Once the model is trained, predictions are made on the test dataset.

1.2 Purpose of the Investigation

With the access to data becoming more and more available, the opportunities to get information from this data also increases. Due to the high success rate of neural networks in terms of their prediction accuracy, the demand for neural networks had become high. The value that accurate predictions or classifications can bring to the business world or the medical field could be the difference between life and death, or profit and liquidation.

Due to the structure of how the neural network trains, the processing of data can become one of the biggest drawbacks and limitations of the neural network. The multiple times that the network has to feed the data forwards and backwards through the network will only become tougher with higher dimensionality of the data and amount of training samples.

Parallelization of algorithms can split the task that is needed to be done by the CPU as a single process and delegate the task to other processors such as the GPU. The division of work can effectively decrease the time taken to complete a task, allowing for algorithms to get results faster and meet the demand of the information-hungry world.

In this paper, we aim to improve on the time taken for a neural network to train. We propose this through implementing the neural network using CUDA and MPI in order to parallelize the training of the network. The delegation of work that is needed to be done between multiple processors shows promise in the decrease in time taken to train the network. In the following sections, we describe the methodology used to achieve this aim, describing the implementation of the neural network in serial, using CUDA, and MPI. The results of the experimentation follow, stating the performance of the various implementations. We then conclude the paper, summarizing the results from this paper.

Chapter 2

Research Methodology

As discussed in the previous chapter, the training times for neural networks using huge amounts of data can be extremely long. The potential improvement of training times, through the parallelization of the heavy computation involved in finding the optimal weights, is extremely beneficial to everyone in the industry involved in deep learning. The purpose of this experiment is to investigate the potential training time improvements gained when using the CUDA and MPI programming models. This chapter will describe the hardware, dataset, procedures, and instruments used in this experiment, and further outline the methodology used for the serial, MPI and CUDA implementations.

The specifications of the system used for training, testing and time analysis of the serial, MPI and CUDA implementations, are listed below:

- Processor: Intel Core i5-8250U CPU @ 1.6GHz
- GPU: Nvidia GeForce GTX 1060
- RAM: 8.00GB
- System Type: 64-bit Operating System
- Operating System: Windows 10
- IDE: Atom
- Programming Language: C

2.1 Data

The data that was used for the implementation was downloaded from the UCI machine learning repository. The dataset contains 773 rows and 27 columns. Of these columns, 26 are features and there is one target column. The dataset contains information specifically gathered to determine whether a company would fail or pass their yearly audit. The 26 features relate to audit information, such as company risk, company history of failing audits and the value of the company. The target column, "Risk", is a column with binary variables where "1" represents the company being at risk of failing audit, and "0" represents that the company has no risk.
For this experiment, the data was split into training and testing datasets using a 70%/30% split. This resulted in the testing dataset having 541 rows and the testing dataset having 232 rows. The label columns were removed from the two dataset stored in their own csv's, respectively.
With all data being stored in csv's, we import each dataset and store the values into separate 1D arrays. We decided to keep the datasets represented as 1D arrays to simplify interpretation of the parallelization of the neural network. All variables of the 26 features are numerical and we, therefore, used float as the datatype to store these variables in the arrays.

2.2 Serial Implementation

The structure of a neural network was described in the introduction to this experiment. The serial implementation of a Neural Network serves as a benchmark for comparing the MPI and CUDA implementations to. It was built using the C programming language, and was structured in such a way as to make it relatively easy to extend for the parallel implementations. The structure of the neural network implemented has 3 layers, namely one input layer, one hidden layer, and one output layer. The input later consists of 26 nodes to account for one instance of each input feature. The hidden was chosen to consist of 10 nodes, and the output layer has one node for classification. The sigmoid activation function was used at each layer. The number of nodes, layers, and the type of activation function used can be easily extended or changed. Finding the optimal weights can take many epochs, and to achieve this forward and backward propagation functions are implemented to

propagate through the network once. We then use for loops to control the number of epochs used for training, as well as to extract each row of the input data for propagation to calculate errors and update the weights. As our main goal is to assess the speed of training for different epoch sizes, our implementation reflects this. The implementation may be extended to stop the training once convergence has been reached. We do not implemented this, however still calculate the errors at each epoch to show the convergence of the serial implementation, as well as the others.

2.3 MPI Implementation

MPI is a message passing interface used for specifying communication between processes. Through this specification, communication consists of data synchronization, as well as moving data from the address space of one process, to the address space of another. There are many operations provided in order to achieve parallelism through communication in the message passing interface.

When training a neural network, it can be seen that the computationally most expensive operations are the forward and backward propagation necessary for updating the weights. Since MPI is a message passing interface designed for parallelism through process communication, multiple processes are launched to achieve this. First MPI must be initalized, and a communicator must be made for which the size and rank need to be defined. The size defines the number of processes launched in the communicator, and the rank defines the label for the calling process. Due to the hardware specifications of the computer, it was found that 8 processes is optimal for finding performance gains. With the communicator defined, and the processes within it launched, we use process with rank 0 as the master, to load the data, and then create memory for all the other processes. Process 0 then performs an MPI operation known as a broadcast to broadcast the data amongst all of the processes. Each process further creates its own network, with parameters initialized according to the rank of the process. This ensures that all the processes together provide coverage for the entire network and all its parameters. The methods used for epoch and row iteration, and the main structure of the network has not changed from the serial implementation. However, each process performs training through forward and backpropagation, providing coverage for the whole network, while calculating

errors and updating their weights for training. With weight updates stored seperately, the MPI_Allreduce operation is used to sum all the partial weights and store the totals in each process for use in the next epoch cycle.

2.4 CUDA Implementation

Implementing the neural network using CUDA would mean that we were able to access the use of a GPU to delegate work needed to be done by the training of the network. In CUDA programming, memory is allocated on the host (the CPU) and copied to the device (the GPU). The work that the GPU is required to do is divided up into threads, which handle one piece of work. The treads are organized into blocks, with a maximum dimension of (32x32x1) which adds up to 1024 threads. Multiple blocks up to the number of 65,535 can be initialized and are organized into grids.

Once the data is copied to the device, a kernel is initiated which is a function that is carried out on the device. The GPU's processors execute these functions in parallel, where a copy of the function is carried by each thread. Once the GPU has executed the function, the memory is copied back to the host where it is used in the rest of the program.

There are several types of memory structures that CUDA contains, that have various levels of access. There is global memory, where each thread can read and write from. This memory can be copied to and from the host. Shared memory is accessible to any thread that resides in the same block. The host can read from shared memory but cannot directly write to it. The advantage of shared memory is that it resides on the multiprocessor chip, this means that there is reduced latency in trying to read from it. Constant memory is similar to shared memory, but the threads are unable to write to it. Both shared and constant memory are relatively small in capacity, only having space for 64KB of data.

In our implementation of the neural network using CUDA, we decide to use global

memory to represent the training dataset on the device. There was no space for constant or shared memory as the size of the memory was too small to represent parts of the neural network. Following the same implementation as the serial version, we set our outer loop for training which is controlled by the number of epochs. We allocate memory for each individual training sample and training label, as well as for each row of our weight matrices and layers of the neural network. The kernels are designed so that each thread carries out the task for one neuron in the training sample. Two kernels were implemented for the forward propagation from layer to layer, Two kernels were implemented for the back propagation from the output layer to the hidden layer and the hidden layer to the input layer. Finally, two kernels were implemented for the updating of weight matrices. At the end of the training of the neural network, the weight matrices were copied back to host memory so that they could be used in the prediction of the test dataset.

Chapter 3

Results

Having outlined the hardware, dataset and methodology used, this chapter presents and discusses the associated results of the experiment.

3.1 Results

Naturally, parallelizing the training of a neural network is only beneficial if the results of the classification are still useful. Along with this metric, we further use the number of epochs for convergence, the speed of training for a given number of epochs as metrics for analysis.

Another metric to be considered is the memory usage per algorithm. We did not measure it directly, but MPI is expected to incur the highest memory requirement, as for each process a network and its associated parameters must be initialized. CUDA, through the transfer of data from the CPU to the GPU and back, is expected to require the next most amount of memory, however achieves parallelization through its efficient use. The serial implementation will use the least amount of memory, but can be extremely slow.

Considering the correctness of our implementations, all were able to show in-sample accuracies, on the training dataset, of between 95% and 100%. The out-of-sample prediction accuracies varied, however were still able to show good results, with MPI achieving prediction accuracies between 78% and 84%, while the serial and CUDA implementions showed accuracies between 84% and 89% and % and % respectively. These prediction differences are due to the random initialization of our

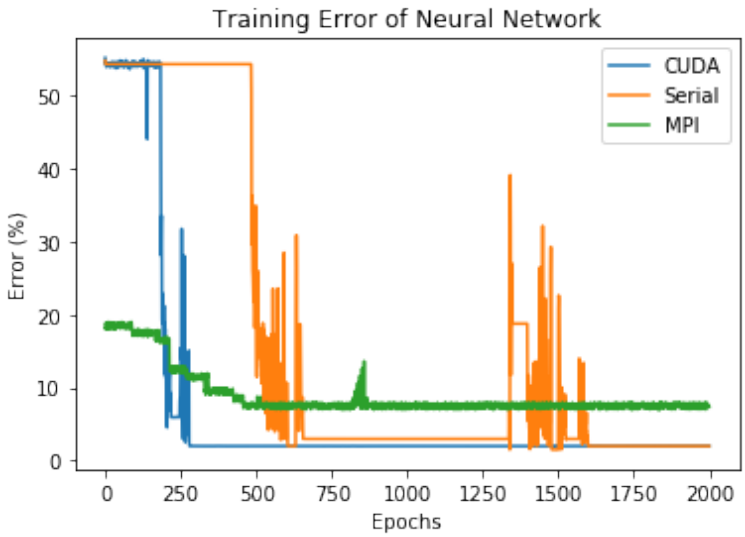

FIGURE 3.1: The Number of Epochs taken for Convergence

weights, as well as differences in parameter updates, due to the different parallelization techniques used.

Having deemed our implementations correct, we then analysed our algorithms in terms of the number of epochs required for convergence. This is important to consider, however does not play a role in our main goal of assessing training speed improvements. We use the mean squared error of the error terms to assess convergence. The serial implementation can be seen to take the longest to converge fully at about 1500 epochs, with CUDA converging by the 250th epoch, and MPI gradually converging by the 500th.

To assess the main goal of this experiment, we assess the training speeds for our implementations on 3 different epoch sizes. We start with 10 000 epochs, increase it to 50 000 epochs, and finally assess 10 000 epochs. By showing this, we validate the ability for these implementations to be extended to larger datasets and an even higher number of epochs. The serial implementation serves as the benchmark to compare the CUDA and MPI implementations to. Our benchmark implementation achieved training speeds of 38s, 140s, and 168s for the three respective epoch sizes.

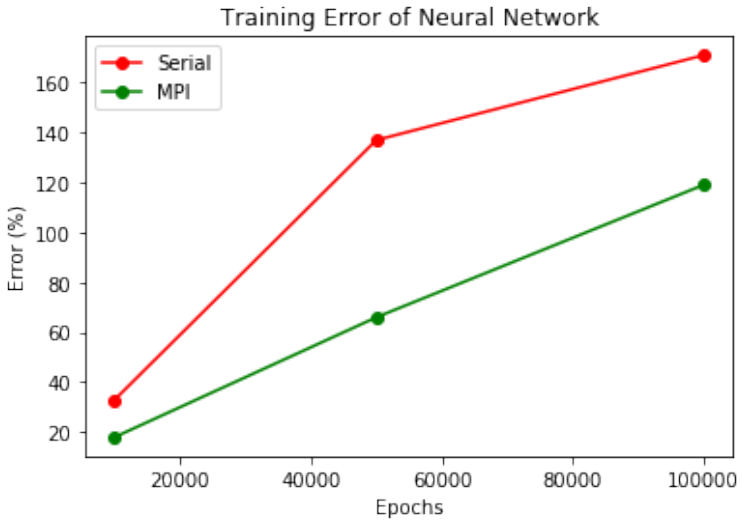

FIGURE 3.2: The Training Times for the Serial and MPI Implementations

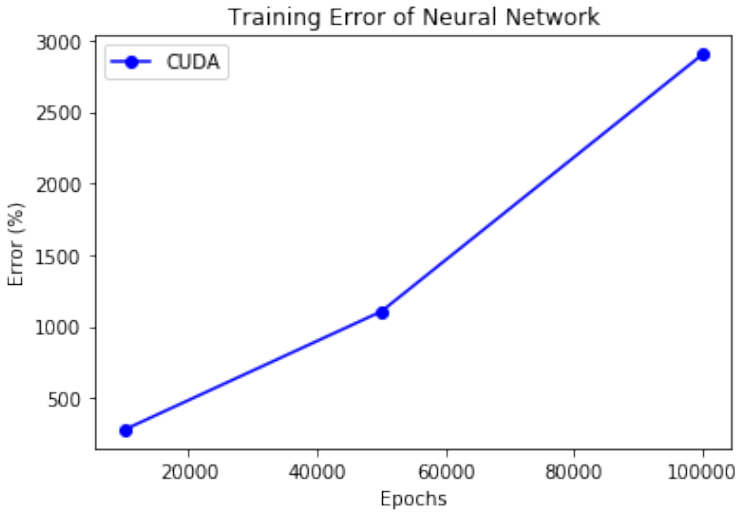

FIGURE 3.3: The Training Times for the CUDA Implementation

MPI, through its efficient message passing interface, was able to successfully improve on these times by an average of more than 30%. The MPI implementation recorded times of 18s, 66s, and 119s for each of the epoch lengths. From our above results, we can see that the CUDA implementation took a lot more time to complete the training of the specified number of epochs. Although the implementation of the neural network was correct, achieving high prediction accuracy, the time taken is not desirable as well as does not show any improvement on the serial implementation. The implementation of the CUDA neural network was based on giving each thread a node's work to do in the forward and backpropagation of the neural network. Due to the low number of nodes in our network, the effect of parallelization is limited. This can be seen as the network only has one output node, making that forward propagation a serial implementation even when run on the GPU. Another factor was the use of global memory, which resides off chip. The repeated access to the dataset in this memory can cause increased latency between the operations in the training of the neural network. The use of streams to streamline function calls should be explored to decrease the latency experienced in the training of the network.

3.2 Potential Future Improvements

Both CUDA and MPI were successfully implemented, and their potential was shown. However, these implementations were not expected to be optimal and there is room for improvement. MPI can potentially be improved by rather using the MPI_Scatter and MPI_Gather operations. Each process would operate on its unique portion of the data, and improvements could be exploited. Further speed improvements can be found by training the network on multiple clusters/nodes, and is a future consideration. CUDA may show better performance on a more complex neural network with more nodes, such as one used for image processing. This is because of the highly parallel nature of cuda programming being optimal when more threads are used. The use of shared memory for weight layers could be explored in order to mitigate the latency experienced between repeated function calls in the training of the neural network. Naturally each implementation would also experience performance gains with better hardware.

Chapter 4

Conclusion

To summarize this paper, we introduced a background to neural networks and the potential issues with the length of training of the neural network. We proposed two improvements to the time taken by parallelizing the training of the neural network using CUDA programming and MPI. We fully describe the methodology used in the various implementations of the neural network and discuss and display results from our experimentation.

To conclude, we can see that MPI was the most effective in decreasing the training time of the neural network, whilst still achieving high prediction accuracy on the testing set. The CUDA implementation was unsuccessful in decreasing the training time for the neural network, due to various factors with the implementation of the CUDA neural network.