Spring 2021 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Implementation of a Neural Network with Parallel Computing

Jacob Nellis

May 7, 2021

Abstract

This report explores how neural networks work and how they are commonly implemented. In this project, a c++ program is created that can be used to create feed forward neural networks of an arbitrary number of neurons. Examples of different neural networks are presented and their behaviors are explored. Additionally, this code has the ability to test and train the network both in series and in parallel; the relative performance of the network in both situations is analyzed.

Link to Final Project `git` repo: https://euler.wacc.wisc.edu/jacob-nellis/me759-jacob-nellis/-/tree/master/FinalProject

Note: In case there is a problem with that link, the address below should take you to my ME759 git repository, my final project is in the 'FinalProject' folder:

https://euler.wacc.wisc.edu/jacob-nellis/me759-jacob-nellis.git

Contents

1. General information

1. Mechanical Engineering
2. Masters
3. Project Contributors
    - Jacob Nellis
4. Statement of free access to code
    - I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2. Problem statement

The human brain has approximately 100 billion synapses, allowing us to think and reason [1]. Each synapse can provide an output of either activated (1) or not activated (0), similar to how a computer's transistors work. This structure of synapses in the human brain has inspired a new type of data processing in software design starting in the early 1940's, neural networks [2]. As computing power exponentially increased over the following decades, neural networks started to really become interesting when they were found to be able to make computers perform behaviors that are not normally associated with how a computer traditionally operates. These networks allowed computers to perform tasks such as image recognition and dynamic human to computer conversations.

Due to the interesting and useful nature of neural networks, I indicated in my project proposal that I would be making a neural network that can recognize letters. Unfortunately, as I tackled this problem, I encountered a limitation in my backpropagation method that would not let me accurately train the network if I had more than one output neuron. Therefore, in this project report, I investigate other uses of this neural network that only require one output, rather than 26. Additionally, neural networks have the ability to become significantly more efficient when correctly trained and tested in parallel. This neural network has the ability to train and test in series and in parallel in order to compare the performance.

3. Solution description

The progress of my final project can be separated into four main sections. The first is the creation of the neural net structure and the method used to give it the ability to make guesses for the output based on different inputs. The next step was to create a backpropagation method that would adjust the weights and biases of the neurons to make its output closer to the target. A testing class was then created to verify its functionality given different sets of testing and training data. Finally, the performance of this network was analyzed and function candidates for parallelization were identified and subsequently parallelized. Each of these stages are discussed in greater detail in the following sections.

Neural Network Structure and Operation

The basic structure of a feed forward neural network starts with a layer of input neurons. A hidden layer is then fully connected to this layer. This means that every input neuron is connected to every output neuron. The hidden layers neuron outputs are then connected to another layer of neurons. This continues until we reach the final output layer of neurons, whose output includes only the result that you are trying to obtain. This structure is shown in figure 1.

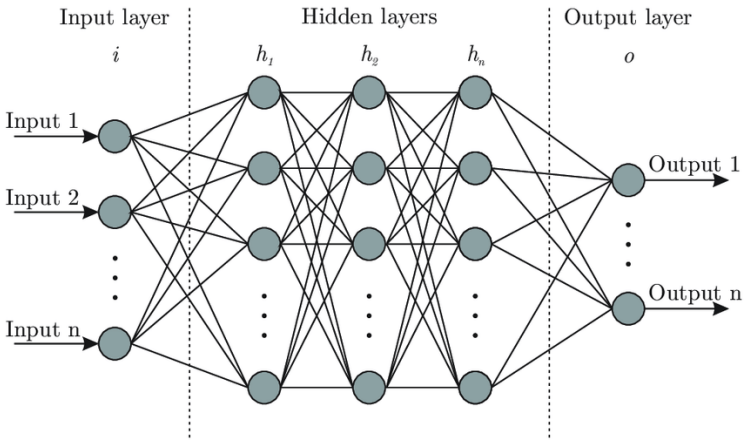

Figure 1, *Diagram of a feed forward neural network* [3]

Each connection from one neuron to another is assigned a weight. The output of the previous layers output is multiplied by this weight and fed into the next layer. The sum of all these weights and previous layer's outputs are then summed in the neuron, along with a bias. This total is passed through an activation function. In the case of my neural network, the activation function that I am using is called the sigmoid function. The sigmoid function essentially normalizes the neurons output to a value between zero and one. The normalized outputs are then fed through to the next layer, and this continues until you reach the output layer.

A class called network was created, the constructor allows the user to define how many layers are in the network as well as how many neurons are in each layer. The constructor than assigns random weights and biases into a weight and bias matrix. Memory is also allocated in the network object to store the outputs of each neuron so that the training method will have access to all these values.

A function inside the class called takeGuess was created that takes an array of input values and adjusts the neurons' outputs to reflect the input data being fed through the network. Multiplying the weights by the neuron outputs conveniently becomes a matrix multiplication problem. For this reason, and because there are other matrix operations required to train a network, a matrixProcessing file was created that performs simple operations on matrices. At this point we have an untrained neural network, meaning its outputs are going to be random and useless. For this reason, we need to create a method for training the network.

Backpropagation and Error Assigning

Backpropagation was easily the most challenging part of this project. To explain this process, I will work through the backpropagation on a simple neural network shown in figure 2.

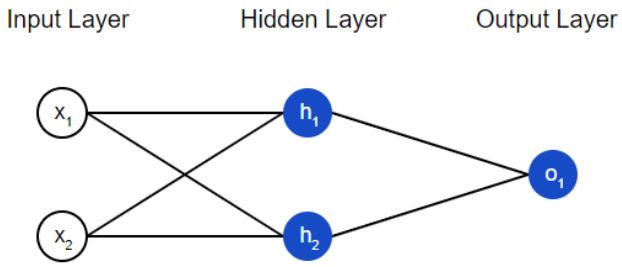

Figure 2, *A simple neural network diagram* [4]

This network has 2 input layer neurons, 2 hidden layer neurons, and 1 output layer. The process of backpropagation starts with the neural network generating a guess based on its current weights and biases. This output is then compared to the target value and an error value is generated for the output neuron. This error is shown in equation 1, where E is the error, T is the target, and O_1 is the output. This is also known as the cost function, and it is the function that backpropagation is trying to minimize using gradient descent.

$$E = (T - O_1)$$

In order to find the desired change in weights for this layer, we need to multiply this error by the derivative of the output, along with the outputs from the previous layer. This creates a final matrix of the change in weights. This matrix is then multiplied by the learning rate of the network. This is shown in equation 2. The reason for having a learning rate is so that no single data point changes the weights of the network by too much, allowing the network to have sufficient time to slowly converge to a solution.

$$E * \left(O_1 * (1 - O_1)\right) * T_r * [h_1 \quad h_2] = [\Delta W_{h11} \quad \Delta W_{h12}]$$

This process is then propagated backwards until you reach the final layer of weights. However, this gets more complicated in the next layers because you need to find how responsible each previous neuron is for the hidden layer error. This is done by multiplying the error by the transpose of the previous weight matrix. Equation 3 shows the next layers weight change equation.

$$\left(\begin{bmatrix} W_{h11} \\ W_{h12} \end{bmatrix} * E\right) * T_r * \left(\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} * \left(1 - \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}\right)\right) * [x_1 \quad x_2] = \begin{bmatrix} \Delta W_{11} & \Delta W_{12} \\ \Delta W_{21} & \Delta W_{22} \end{bmatrix}$$

The change in weights are finally added to the network's weights matrix. Additionally, the biases are adjusted by multiplying them by the derivative of the previous layers outputs. This method is analogous to finding the derivative of y = ax+b, and adjusting 'a' so that the input x creates a lower y output, just with more complexity.

A problem with this method, however, is that because the weights are initialized randomly the method can find a local minimum that is far away from the global minimum; if this happens then the network will get stuck in a non-optimal area of performance. One way that people have tried to account for this is by training many neural networks at once, each with different initial weights and biases and then finally picking the one that performs the best. With the backpropagation method done, it is now time to create some data to test its performance.

Testing and Training Data

There are two types of data used when creating a working neural network, training data and testing data. These two sets of data need to be different from each other to verify that the neural network is not just 'memorizing' the training data and is able to make predictions based on novel testing datasets. Two new functions were implemented in the network class to handle these data sets, trainEpoch and testNetwork. These functions both take an array of target arrays and an array of data arrays. The trainEpoch function simply calls the train function until it has trained the neural network using every sample that it was given. This is later built upon while trying to parallelize the code so that it could train with multiple samples at once. The testNetwork function continually acquires the guess from the network and compares it to the target for that sample. It returns the average error over the entire testing set.

A testNet file was also created to showcase a few examples of training and testing the neural network. The first set of testing data that was created is called XY compare. This function creates 1 million random x and y input values between zero and one. If x is greater then y then the target value is a one, otherwise it is a zero. This data set is shown in figure 3.

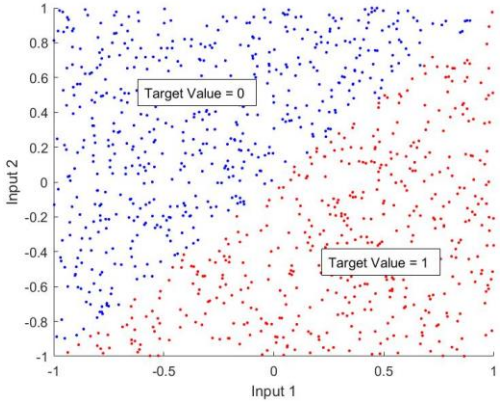

Figure 3, *Test/Training Data Set for XY Comparison Network*

This is the simplest set of data that you can train a neural network on and the network only requires 1 hidden layer neuron to accurately train itself. This is because with one neuron, you are only able to solve problems that are linearly separable [5]. However, more complicated data sets can also be trained with this neural net with the implementation of additional hidden layer neurons. In order to test a non-linearly separable problem, an exclusive OR (XOR) data set was created. The target output is set to one when both outputs are either positive or negative, otherwise it is set to zero. The networks performance for both of these tests are discussed in the results section. The training set for this test is shown in figure 4.

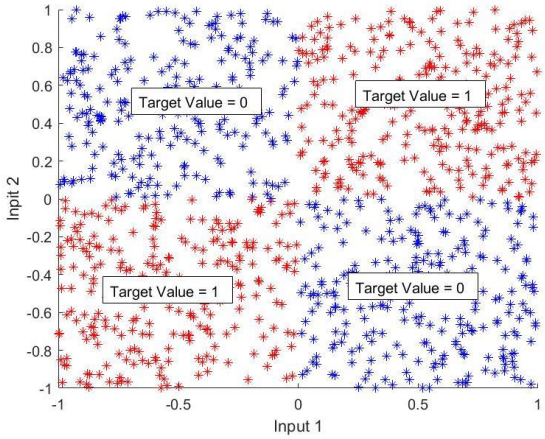

Figure 4, *Test/Training Data for XOR Network*

Of course, these are relatively simple problems and only require a small number of neurons for the network to perform well. However, many of today's networks that perform more complicated tasks have hundreds of neurons. The final test set function (testLargeNetwork()) tests a neural network that has 700 hidden layer neurons and 700 input neurons. This data set has random inputs between 0 and 1 and a random output. Therefore, when training and testing this network, it will not actually be learning anything useful, but it does allows us to investigate the performance of these larger networks. This large neural network is the network that is analyzed in the next section to determine where this code can be parallelized.

Parallelization and Performance Improvements

In order to find out what parts of the code is consuming the most amount of computation time, an analysis in visual studio was performed, as shown in figure 5.

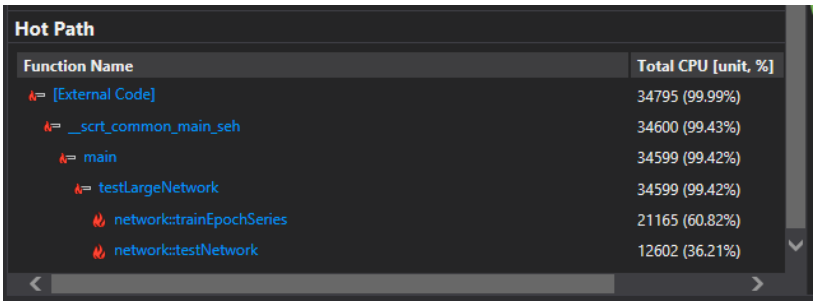

Figure 5, *Breakdown on code performance based on total CPU time spend on functions*

Looking at the performance of the program, it appears that training the network takes 60% of the computation time and testing the network takes approximately 36%. This means that the most effective place to start parallelizing is the backpropagation methods. One way to do this is with batch testing, a type of course-grain parallelism. This works by taking multiple data samples and training them on the network to find the change in weights and biases. After the entire batch has finished, all of the weights and biases are summed together and added to the main network. This is essentially averaging error across multiple data samples instead of changing the weights after every single sample. This procedure is outlined in figure 6.

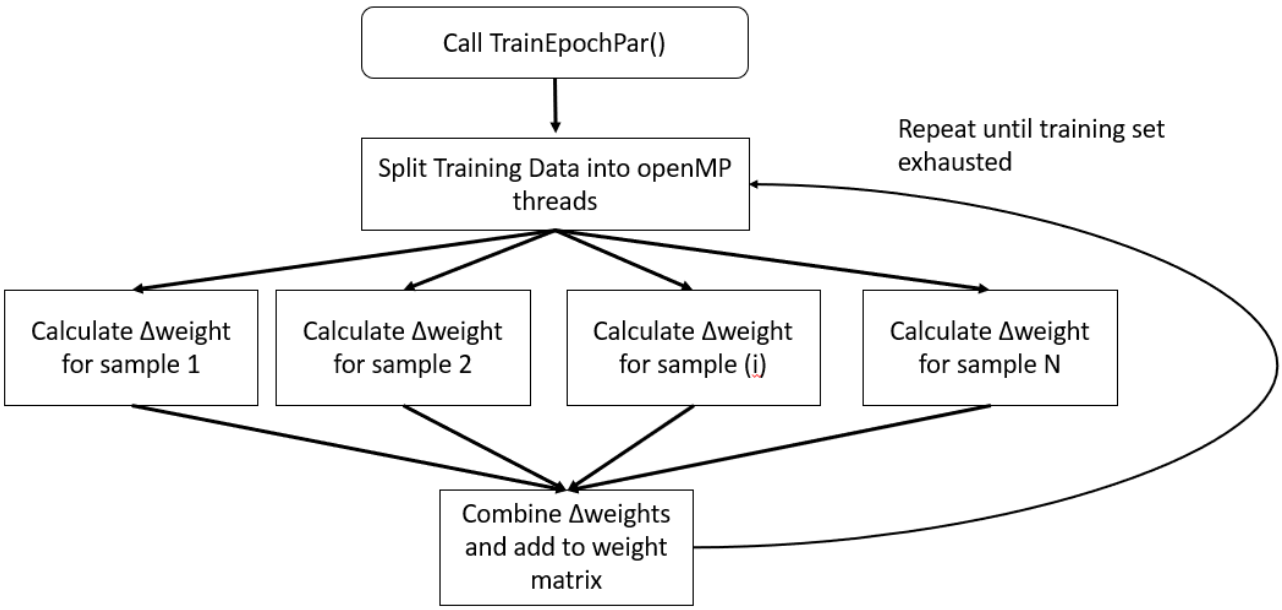

Figure 6, *Process for training the network in parallel*

Similarly, the testing of the data can also be performed in parallel. This was done in the same way as shown in figure 6, except the error was summed instead of the weights.

It was also apparent that the matrix processing was taking a significant amount of computation time while inside the training and testing functions. This can be mitigated because each neurons output is independent of any other neurons output in the layer being worked on, meaning that this is an opportunity for fine-grain parallelism in the matrix processing file. Unfortunately, I was unable to figure out how to effectively manage CUDA streams concurrently from different openMP threads, so I had to leave this level of parallelism out of my final code.

4. Overview of results. Demonstration of your project

The testNet file runs through four example networks. The first two examples show that the network performs exceptionally well after a training session; again, these networks have only 2-4 input layer neurons. The third test uses random inputs and outputs to show how much performance speedup you can obtain when the parallel functions are utilized vs the serial implementation. This network has 700 hidden layer neurons and 700 input neurons. The final output also utilizes 700 hidden and input layer neurons, but shows why it is important to use different data for your testing and training sample. The output of the code is shown in figure 7.

```
Starting the program
Start of XY Comparison Test:

Average accuracy before training: 52.3%
Average accuracy after training: 99.4%

Start of Exclusive OR Test:

Average accuracy before training: 55.3%
Average accuracy after training: 99.6%

Start of Large Network Test (series):

Average accuracy before training: 52.0%
Testing in series took: 16886.573100 milliSecs

Average accuracy after training: 50.0%
Training the network in series took: 74882.702000 milliSecs

Start of Large Network Test (parallel):

Average accuracy before training: 49.6%
Testing in parallel took: 3950.225100 milliSecs

Average accuracy after training: 50.1%
Training the network in parallel took: 27730.561500 milliSecs

Start of same testing/training data test:

Average accuracy before training: 49.8%
Average accuracy after training (testing with same data): 70.0%
Average accuracy after training (testing with different data): 50.2%
```

Figure 7, *Output from testNet.cpp*

Observe that the network performed exceptionally well with the XY comparison and XOR tests. The neural network does not improve in accuracy when being trained on the large neural network, this is because the output is always random, so it can only hope to be 50% accurate. This could be an interesting way for the developers of c++ to show that their random number generator is in fact random from number to number, because if it was not then a complex neural network would likely find some correlation to increase its accuracy. The final test uses the same network as test 3, but it used the same training data to test its performance. As you can see from the output, the network started to memorize the patterns in the data and obtained a 70% accuracy. However, when new data is introduced that follow the same rules as the sample data, it performs at 50% accuracy. This is an important subtly in neural networks that can be caused by overfitting; this is essentially using too many neurons for the complexity of the task. This code was also tested by training and testing networks ranging from 2 neurons to 650 neurons in order to compare the speed of serial vs parallel, this is shown in figure 8.

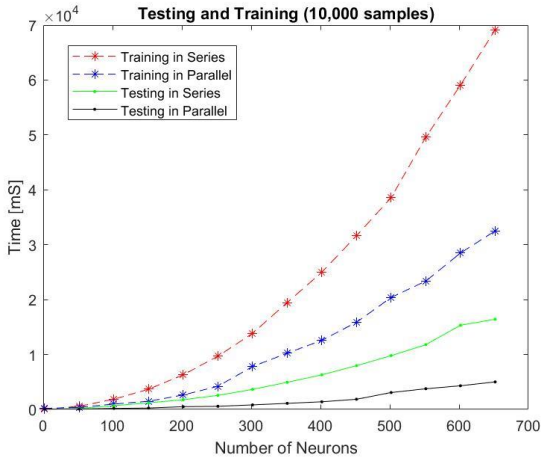

Figure 8, *Execution time vs number of neurons in input and hidden layer, tests were performed on home computer (Core i5 processor)*

The training data for both series and parallel follow an exponential line. This is because the vector-vector multiplication function used to generate the change in weights was not parallelized. This ensures that there will be N^2 series operations required to calculate these weights. That being said, batching the training data into parallel sections did produce a 2-3X speedup in performance. The testing method follows an approximately linear trend as the number of neurons is increased. This is because the feedforward method does not require vector-vector multiplication and only requires element wise multiplication. The parallel implementation of this method also produced a 2-3X speedup in performance.

5. Deliverables:

Discuss what is delivered for this Final Project. Important points:

- This report is submitted to canvas
- The final code files for this project are: neuralnet.cpp, neuralnet.h, matrixProcessing.cpp, matrixProcessing.h, and testNet.cpp
- To compile the code use: g++ neuralnet.cpp matrixProcessing.cpp testNet.cpp -Wall -O3 -std=c++17 -o final -fopenmp

6. Conclusions and Future Work

This project demonstrated the key concepts behind neural networks and showed how they can be parallelized. The example training data, while simple, demonstrates how the network can quickly converge on a solution with high accuracy. Additionally, neural networks are extremely parallelizable problems and this code demonstrates the speedup in performance you can get by performing techniques that were discussed over the course of this semester. One of the lessons I learned in this project was not to make the code initially run sequentially, this caused me to have to restructure large parts of the code when trying to use parallel techniques.

Another issue with my code is the lack of visuals that it generates. In the future it would be nice to make a visualizer that shows the structure of the neural net after it is generated along with a progression of its outputs as its being trained. That would go a long way in showing how neural networks slowly find a local minimum to the problem that they are presented with. I would say that the most urgent piece of future work would be optimizing the matrixProcessing library that was created for this project. Based on the visual studio analysis, there was significant room for performance increases in this file. Overall, I believe that I was able to accomplish a significant portion of what I originally hoped, I may come back to this project in the future because it is a good baseline for building a more complicated and specific neural network.

References

[1] "viewcontent.pdf." Accessed: May 06, 2021. [Online]. Available: https://scholarcommons.usf.edu/cgi/viewcontent.cgi?article=4812&context=ujmm.

[2] Jaspreet, "A Concise History of Neural Networks," *Medium*, Apr. 08, 2019. https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec (accessed May 06, 2021).

[3] "Machine Learning for Beginners: An Introduction to Neural Networks - victorzhou.com." https://victorzhou.com/blog/intro-to-neural-networks/ (accessed May 06, 2021).

[4] “Fig. 1. Artificial neural network architecture (ANN i-h 1-h 2-h n-o).,” *ResearchGate*. https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051 (accessed May 05, 2021).
[5] “Limitations and Cautions :: Perceptrons (Neural Network Toolbox).” http://matlab.izmiran.ru/help/toolbox/nnet/percep11.html (accessed May 06, 2021).