Neural Network Parallelization

Mason Olsen, Callista West, and Patrick Thomasma

University of Oregon

October 11, 2021

Abstract

In this paper, we explore how OpenMP drastically reduces run times for training neural networks. Our neural network is trained for image recognition from the Fashion MNIST dataset. This data set contains 10 different types of clothing and the neural network is tasked to identify the category of clothing for one of these images. Using pixel values and 3 layers, this neural network is able to identify these different types of clothing with very high accuracy. What is more important however, is how OpenMP exponentially decreases run time for training. Since training the network is the bulk of the program, the parallelization of this method is the most important aspect to examine.

I. Introduction

Artificial neural networks are centered in deep learning algorithms and also closely related to machine learning. Neural networks received their name and structure from the neurological process in the brain, following the way neurons signal and send information to one another. Artificial neural networks were first invented in 1944 by Warren McCullough and Walter Pitts, two researchers for University of Chicago.

i. Structure

Current day artificial neural networks are made of node (neuron) layers. Every node is connected to all of the nodes in the next corresponding layer, to allow for sending of data between nodes and layers. Accuracy for neural networks takes time, requiring training data to improve speed and accuracy in order to sort data at a high velocity. Each node has a corresponding weight, and bias (also referred to as a threshold). When the output value of an individual node exceeds the node's threshold value, the node must send data deeper into the neural network. There are three additional layers that exist in neural networks. The input layer accepts the data and passes it to the rest of the network. The hidden layers make up the bulk of the neural network. There must be at least one hidden layer for the neural network to function properly. The hidden layers are responsible for the performance in the neural network. The nodes in the hidden layers apply different transformations to the input data, which in result, train the data, compute the error, create predictions, and complete all the computations in the neural network. The output layer receives input from the last hidden layer, performs the calculations using its neurons, and computes the output for the neural network. Interestingly enough, the neural networks invented by McCullough and Pitts in 1944 included both thresholds and weights, but the researchers didn't arrange the network into layers or specify any specific training mechanism.

Additionally, there are numerous types of neural networks but feedforward neural networks are the most basic and allow extra layers without too much consideration of computation time. Neural networks are also useful tools within artificial intelligence. However, feedforwarding has disadvantages such as its inability to be used with sequential data and difficulties with image data. Some other neural network types include convolution neural networks (CNN), recurrent neural networks, and transformers. CNNs, while seen as automatic feature extractors from images, have their own downfalls such as object detection. Our group decided to implement the feedforward neural network, as the extra layers allowed our implementation to include 2 hidden layers without significantly affecting computation time.

II. Background

i. Feedforward Neural Networks

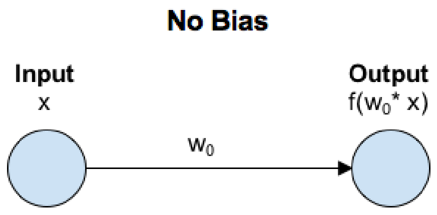
Feedforward neural networks, often referred to as multilayer perceptrons are deep learning models with the goal of approximating some function f*. These models got their name from the way information flows forward through the different layers and computation to receive the correct input. Feedforward models do not have feedback connections, node connections that form loops or circles, so no outputs of the model are fed back into itself. Frank Rosenblatt developed perceptrons in the 1950s and 1960s based on the ground work provided by McCulloch and Pitts. When multiple layers of perceptrons are combined together, this creates multilayer perceptrons, or feedforward neural networks.

ii. Weights and Biases

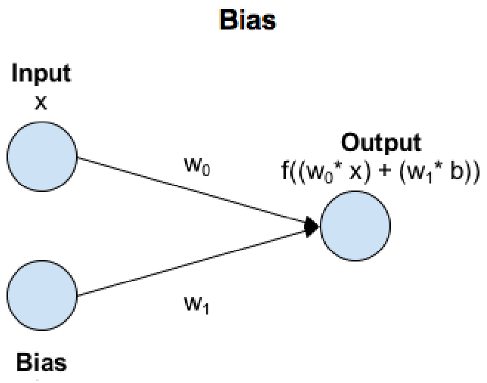
Weight is a parameter within a neural network that influences the transformation of the input data inside of the hidden layers. Weights are a learning parameter within a neural network. When an input enters a node, the input gets multiplied by a weight value, and the output from that computation is observed or passed to the next layer in the neural network. It is within the hidden layers that the weights are applied to the nodes. Additionally, weights represent the strength of the connection between two neurons. This weight decides how much of an influence the input will have on the output.

Bias play a large role within the activation function of the neural network, which will be explained in the methodology section of this report. By including bias in the neural network, the activation function has the ability to shift away from the coordinate system origin (0,0), which can be critical for successful learning. Without a bias, the input to the activation function is "x" being multiplied to the corresponding connection weight.

When adding bias to the parameters of the neuron, the input to the activation function changes. The input to the activation function is "x" multiplied by the connection weight "w0", which is then added to the bias multiplied by the connection weight for "w1". Since biases are constant value, the activation function is



being shifted by the amount of (bias * w1). The figure below describes the shifting in comparison to a neural network without biases.



iii. Fashion MNIST

For our project, we needed a dataset to use as our input to test the accuracy and performance of our neural network. To find this dataset, we used Kaggle. Kaggle is a website that has a huge repository of community published data and code. On Kaggle, we found the Fashion MNIST dataset, a MNIST-like dataset of 70,000 28x28 labeled fashion images. The MNIST dataset consists of 60,000 small 28x28 pixel grayscale images of handwritten single digits between 0 and 9. The Fashion MNIST dataset varies slightly from the typical MNIST dataset with the number of images and the types of images being used.

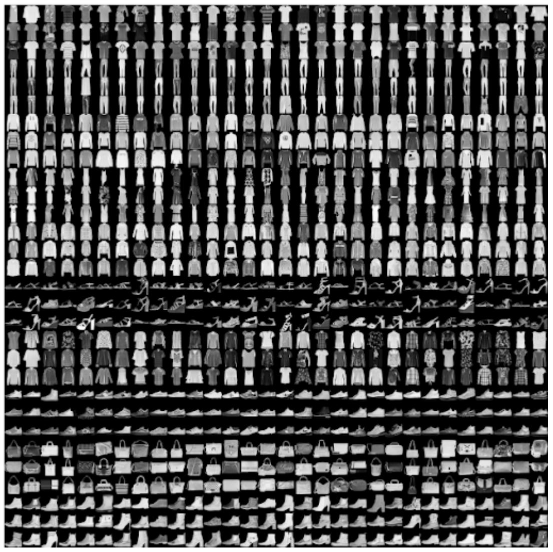
Within the Fashion MNIST dataset, there are 10 different labels (0-9) for the training and test data to be assigned to. The label numbers and their descriptions are the following:

0: T-Shirt/Top	5: Sandal
1: Trouser	6: Shirt
2: Pullover	7: Sneaker
3: Dress	8: Bag
4: Coat	9: Ankle Boot

Each image within the dataset consists of 784 pixels. Each pixel has a single pixel value associated with it, which indicates the lightness or darkness of that pixel. The higher values indicate more darkness in the corresponding pixel. The training set includes 60,000 images, while the test set made up the remaining 10,000 images. To better visualize the Fashion MNIST dataset, we have included the corresponding image representation of the dataset, where you can see the different articles of clothing, as well as the different degrees of darkness and lightness in the images.

III. Methodology

In order to be able to parallelize a neural network, we first had to make sure we successfully completed all of the necessary functions and procedures that make a neural network work correctly. Each one of these steps required intermediate testing as we built our project.



i. Activation Function

An activation function in a neural network defines how the weighted sum of an input is changed into an output from a node. The activation function affects both how the hidden layers will learn the training dataset as well as how the output layer will define the types of predictions that the model can make. In the first attempt at the activation function for our neural network, we tried using the Sigmoid Activation Function. The Sigmoid Activation Function takes any real value as an input and produces an output value between 0 and 1. The graph of the sigmoid function produces a S-shaped curve. It wasn't until predicting the labels in our neural network that we decided to switch our activation function to the Tanh activation function, in hopes of getting better predictions. The Tanh function is very similar to the Sigmoid activation function, both being nonlinear, monotonic, and sigmoidal. Both functions are also continuously differentiable across different values and the derivative of these functions can be expressed in terms of the functions themself. However, the Tanh activation function has a slightly different fixed output range of values between -1 and 1. Using the Tanh activation function, we were able to receive 100% accuracy on predicting label values, a bug we were unable to overcome when using the Sigmoid Activation Function. The tanh activation function is given by:

$$a = tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$\frac{da}{dz} = 1 - a^{2}$$

ii. Loss Function

Loss, as calculated in the loss function, is simply just a prediction error of the Neural Net. The estimated loss from the loss function is used to update the weights to reduce the loss in the next evaluation. In our project, we decided to implement our loss function as the Mean Squared Error loss function (MSE). Typically, MSE is used for regression tasks. To calculate the loss, the function takes the mean of squared differences between the target values (the actual value) and the predicted values. The result of the loss function is always positive. The perfect loss value to come from the loss function would be 0.0. Knowing our loss function will tell us how close a regression line is to a set of points, this is called mean squared error because we are finding the average of a set of errors, knowing the loss function allows us to see how close our training is to the real thing and how close it is to reaching that perfect loss value for a well trained neural net.

The algorithm for MSE is below:

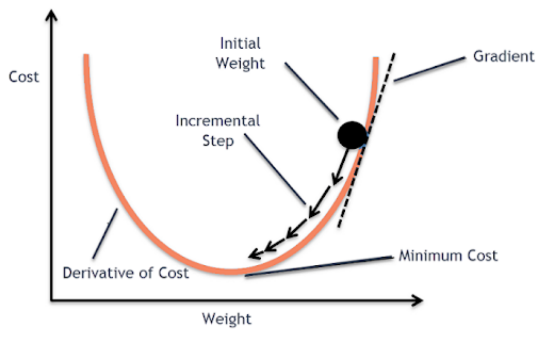
$$MSE = \frac{1}{n} \Sigma \underbrace{\left(y - \widehat{y} \right)^2}_{\text{The square of the difference between actual and predicted}}$$

iii. Weight Initialization

In our neural network, we created a function weights_init() to initialize all of the weights assigned to the node connections. There are two common ways to initialize weights in neural networks: random initialization and zero initialization. Zero initialization initializes all weights to zero, just as it sounds. Random initialization assigns each weight to a random value between 0 and 1. We decided to use random initialization, where we used the rand() function (built into the stdlib.h library) to provide a random value to each weight. The advantage of using random initialization over zero initialization is breaking up the symmetry that is caused by zero initialization. This step in the neural network is important because initializing weights correctly will lead to an optimization of the loss function, providing better overall accuracy to our neural network.

iv. Batch Gradient Descent

Gradient descent is a first order iterative optimization algorithm that is found for finding the minimum of a function. How the gradient descent relates to our neural net is that we are computing the slope of our functions at a current point and then we move in the opposite direction of the slope increase by the computed amount. In order to do this we passed the training set through the hidden layers and then updated the parameters of the layers by computing the gradients using training samples from our dataset. A batch gradient descent is when all of our training data is taken into consideration in order for us to take a single step towards the bottom of the slope, so our neural net takes just one step of gradient descent per epoch. An illustration of the Batch Gradient Descent is below:



v. Back Propagation

In our neural net, training the back propagation is what gives purpose to the training as its the practice of the network to continuously fine tune the weights of the neural net based on our error rate (loss function) which was obtained in the previous epoch, based off the error rate the network will further tune its weights in order for there to be a lower error rate. In our neural net we first apply our errors though the layers with double h2_err and double h1_err and then in order to back propagate we then apply what we've learned from the error using a nested for loop that will go through our different nodes in order to fine tune them for the next epoch.

vi. Training the Network

Training the network consists of three parts previously stated in this report, feed-forwarding, computing error, and then back-propagating that error. Each of these parts are updating the network for one data sample at a time. The pseudo-code for the feed-forward part is as follows:

Algorithm 1 Feed-Forward

```
for $j \leftarrow 1, num\_neurons$ do
    $activation = bias_j$
    for $k \leftarrow 1, num\_prev\_neurons$ do
        $activation += prev\_layer_k \cdot weights_{jk}$
    end for
    $layer_j = tanh(activation)$
end for
```

For this part, there are three of the above nested loops. The first loop is calculating the activation of each neuron in the first hidden layer, the second loop is calculating each neuron's activation in the second hidden layer, and the third loop is calculating the activation of each neuron in the output layer. The activation of a neuron is equal to the bias of that neuron, plus the dot product of the outputs from the previous layer with the weight vector of that neuron. The activation of the neuron is then equal to the output of the activation function of that sum.

The next part is computing the error of the predictions. This is where the derivative of the Mean-Square-Error algorithm, and the derivative of the tanh function comes in. The pseudo-code for this section is below:

Algorithm 2 Compute Error in Output

```
for $j \leftarrow 1, num\_outputs$ do
    $error = mse\_der(output_j, label)$
    $out\_err_j = error \cdot tanh\_der(output_j)$
end for
```

Algorithm 3 Compute Error in Layers

```
for $j \leftarrow 1, num\_neurons$ do
    $error = 0$
    for $k \leftarrow 1, num\_next\_neurons$ do
        $error -= next\_layer\_err_k \cdot weights_{kj}$
    end for
    $layer\_error_j = error \cdot tanh\_der(layer_j)$
end for
```

For this part, there is the first loop and 2 of the second loops for the output layer and the two hidden layers respectively. The first loop uses the mean square error function to calculate the error in the activation of each output neuron using the label for that data sample. With the error calculated for each output neuron, the error for each of the neurons in the hidden layers can be calculated by taking the dot product of the error of each neuron in the next layer with the element in all of the next layer's neuron's weight vector corresponding to the current neuron. This value is then multiplied by the result of the derivative of the activation function on the activation of the current neuron from the feed-forward stage. Ultimately, the value calculated in this part tells the network how much to update the weights and biases of each neuron in the network.

The last part is actually updating the weights and biases of each neuron. This is done by the following algorithm:

Algorithm 4 Back-Propagation

```
for $j \leftarrow 1, num\_neurons$ do
    $bias_j += layer\_error_j \cdot lr$
    for $k \leftarrow 1, num\_prev\_neurons$ do
        $weights_{jk} += prev\_layer\_err_k \cdot error_j \cdot lr$
    end for
end for
```

Once the error for each neuron was calculated, each neuron's weight vector is updated by the multiplication of the error of that neuron, the activation of the neuron that fed into it, and a learning rate. The learning rate is, from the gradient descent illustration, how much to scale the size of the incremental step towards the minimum of the loss function. This is tuning the weights and biases to be more accurate. As the weights and biases become more accurate, the values being back-propagated will become less and less.

vii. Prediction

Once the network was trained, it was ready to be used on the test data to see how accurate it is on data it hasn't seen before. The prediction algorithm is the same as the feed forward algorithm. Essentially, the feed forward in the training stage of the program is used to create a prediction to then calculate the error whereas now it is just being used to create the prediction. The actual category that the network predicts is just equal to the index of the neuron with the greatest activation in the output layer.

viii. Parallelization

For parallelizing the neural network, we used OpenMP. With OpenMP, we parallelized each of the above algorithms that are used in the training of the network. By doing this, each thread became responsible for calculating the activation of a single neuron, calculating the error of a single neuron, or update the weights and biases of a single neuron.

IV. Results

The results we got for our neural network are as follows:

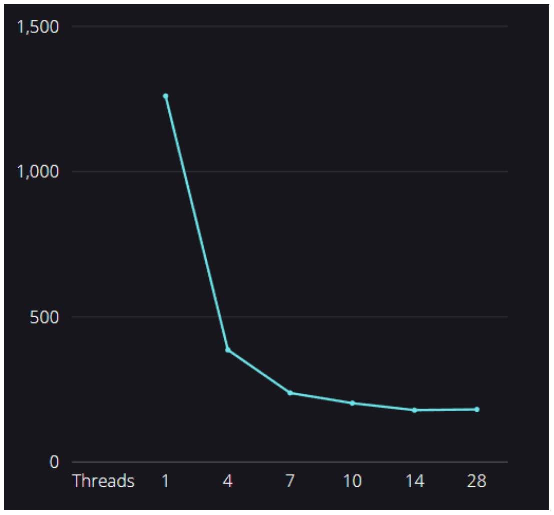


Figure 1: *Execution Time in Seconds*

With exact values of:

Threads	Time(Sec)
1	1260
4	386.175
7	237.89
10	202.56
14	178.54
28	180.97

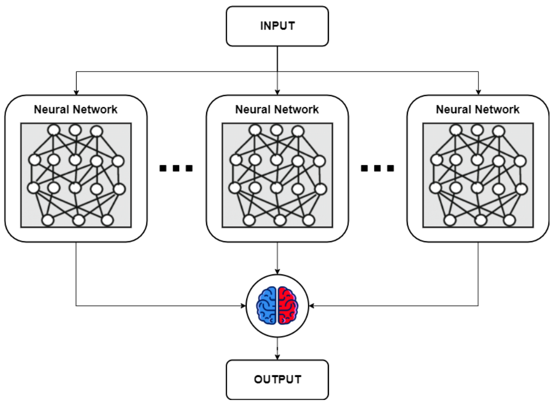
All in all, we were able to achieve a speedup from about 21 minutes, to about 3 minutes, which is about a 7x speedup. From the graph illustration, the speedup starts to plateau for thread counts higher than 14. The reason behind this is most likely the overhead from creating and assigning threads in the training stage. Since each data sample is sent through the training stage once at a time, thread creation and assigning are done for each data sample. Thus with out training data set of 60,000 data samples, the network is having to create and assign threads 60,000 times. Thus with greater thread counts, the overhead for each of the data samples starts to match the speedup from each thread calculating a value for each neuron.

V. Conclusion

i. Future Implementations

One future implementation that would be interesting to examine would be using different neural network model types. Since we used a feedforward approach, it would be interesting to build a neural network using a convolutional or recurrent network as well. With these different model types, we would be able to try new methods of parallelizing the training of the network. Since each model behaves differently, they will have different methods of updating weights and biases of each neuron and predicting outputs. We could then compare these different parallel models based on their speedup and accuracy.

If we had more time, we would have most liked to implement a neural network ensemble model using OpenMPI. It would look like:



Essentially OpenMPI would allow us to create multiple nodes where each node has their own neural network and their own section of the training data. They would then train their own networks in parallel. Once they finish training, the test stage would entail sending the data sample to be tested to each node. Each node would create their own prediction and then the main node would use some method of combining the results to get a final prediction. Since each network would be training with less data samples and training in parallel, we are sure that we could get a good speedup from this method.

ii. Final Thoughts

Overall, this was a good project to implement because neural networks and other machine learning models are highly used by the tech industry. With such large amounts of data needed to make these models useful, learning how to make these models fast and efficient is very important. We were able to achieve both of these tasks by first learning how neural networks work, and by successfully parallelizing the neural network. After implementing a serial version that uses 4 layers, gradient descent, feed-forwarding, and back-propagation, we then parallelized these tasks by having each thread be responsible for the calculation of a needed value for a single neuron. Thus each neuron was able to be updated or used in parallel. By doing this, we were able to get a vast speedup from the serial version.

VI. References

1. Durán, J. (2019, September 20). Everything you need to know about gradient descent applied to neural networks. Medium. Retrieved December 2, 2021, from https://medium.com/yottabytes/everything-you-need-to-know-about-gradient-descent-applied-to-neural-networks-d70f85e0cc14.

2. Ognjanovski, G. (2020, June 7). Everything you need to know about neural networks and backpropagation-machine learning made easy... Medium. Retrieved December 2, 2021, from https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a.

3. The role of bias in Neural Networks. Pico. (n.d.). Retrieved December 2, 2021, from https://www.pico.net/kb/the-role-of-bias-in-neural-networks/.

4. Sharma, S. (2021, July 4). Activation functions in neural networks. Medium. Retrieved December 2, 2021, from https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

5. IBM Cloud Education. (n.d.). What are neural networks? IBM. Retrieved December 2, 2021, from https://www.ibm.com/cloud/learn/neural-networks.

6. Larry Hardesty | MIT News Office. (n.d.). Explained: Neural networks. MIT News | Massachusetts Institute of Technology. Retrieved December 2, 2021, from https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

7. Al-Masri, A. (2019, January 29). How does back-propagation in Artificial Neural Networks Work? Medium. Retrieved December 2, 2021, from https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7.

8. DeepAI. (2019, May 17). Feed Forward Neural Network. DeepAI. Retrieved December 2, 2021, from https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network.

9. Brownlee, J. (2021, November 14). How to develop a CNN for mnist handwritten digit classification. Machine Learning Mastery. Retrieved December 4, 2021, from https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/.

10. Activation functions: What are activation functions. Analytics Vidhya. (2021, April 14). Retrieved December 5, 2021, from https://www.analyticsvidhya.com/blog/2021/04/activation-functions-and-their-derivatives-a-quick-complete-guide/.

11. Brownlee, J. (2020, August 25). How to choose loss functions when Training Deep Learning Neural Networks. Machine Learning Mastery. Retrieved December 7, 2021, from https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/.

12. What are the differences between MSE and RMSE. i2tutorials. (2019, November 13). Retrieved December 8, 2021, from https://www.i2tutorials.com/differences-between-mse-and-rmse/.