ECE 408 Final Project

Fall 2014

Parallel Neural Networks for Digit Classification

Fanmin Shi
Dennis Liang
Avdhesh Garodia

Introduction

Artificial Neural Networks are a state-of-the-art machine learning model. They are widely used in many recognition and classification applications such as handwriting recognition, items classification, images recognition, etc. In this project, we are exploring a specific implementation of Neural Networks called a feedforward neural network. We will use this type of neural network to do handwritten digit recognition. Traditionally, feedforward network is implemented serially. With the serial implementation, one can only experiment certain neural network size and data set size due to the heavy computation requirement of neural network model. Thus, running time of neural network can be unbearable at a certain input size. In order to overcome this problem and allow larger inputs to be computed in a reasonable time, we will explore the parallel nature of the neural network model and implement a robust parallel neural network for handwritten digit recognition.

Code flow

A brief description of our code will follow. First, we created a saver that would save the Neural Network's object data (the Net object) to disk. This was necessary since the training of the Net took a significant amount of overhead, and a trained Net would be something that could be potentially reused. Then, a loader was created that would take the file that was outputted to disk and, using a constructor, allocate and initialize memory on both main memory and the GPU for the Net that was being loaded. These were some of the things we did to avoid having to constantly retrain Nets. The overall running of this process would be to train some data which would initialize some Net object. This would be saved to disk, and this net could also be used to run the digit recognition model. When execution was stopped, one could reload this Net and then use it again to re-run the digit recognition model without the training overhead, and with the same error as before.

Problem

Neural Network is a computational model that is used to estimate or approximate an unknown function. In Machine Learning, there are many unknown functions that we want to approximate. For example, the unknown function $F : Images \rightarrow Digits$ where the function maps a handwritten image to an integer. We do not know the implementation of such function. However, we would like to use this function to recognition digits for many world applications such as identify zip code on a mailing address. In order to do that, we

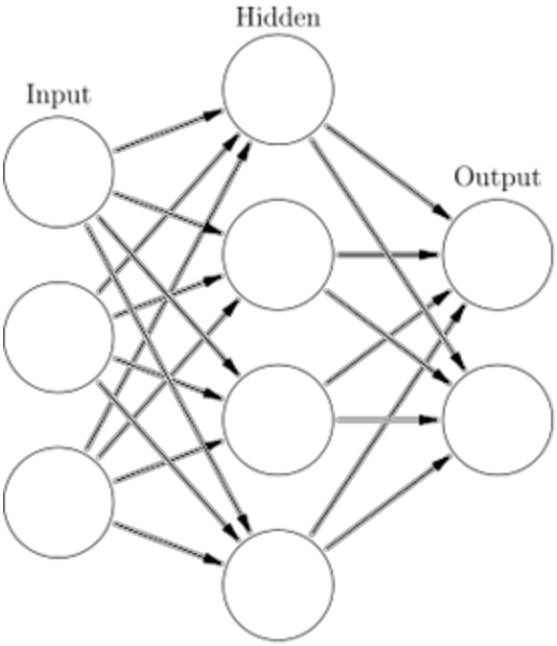
can use a machine learning model, in this case neural network, to approximate this unknown function.

In order to recognize handwritten digits with high accuracy using neural network, neural network need to learn the relationship between the input handwritten digit and its associate correct digit. The learning/training process is following. The first, we need a dataset of handwritten digits in which we call it the training set. We will use MINST database of handwritten digits in this case. The second, for each of the handwritten digits we want the neural network to output the correct digit. To achieve that, we would use two neural network specific algorithms called feedforward propagation and backward propagation. The feedforward algorithm involves taking in a handwritten digit as the input and output a best guess of that handwritten digit. The backward propagation algorithm will take the output produced by the feedforward algorithm and use it to compare against the expected output, the correct digit for that handwritten digit. Then it will updated the internal states of the neural network to minimize the difference between the actual output from feedforward and the expected output. We do the above process numerous iterations throughout the dataset until the neural network become accurate.

Observe that we immediately update the internal states of neural network after running backward propagation on each handwritten digit. This type of training is called Online Training. Due to this nature of training method, next input depends on the updated internal states from the current input. As a result, we can not running training parallelly using multiple inputs due to this constraint. Instead, we will parallel the feedforward and backward propagation algorithm.

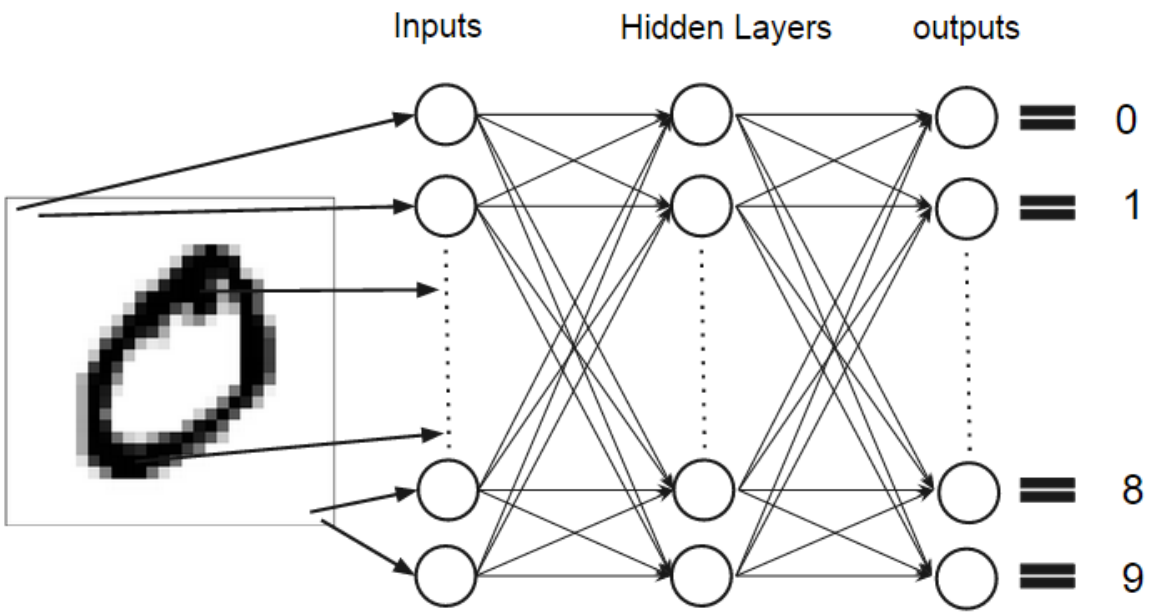
Neural Network Model

The image on the left represents a feedforward neural network model. It consists of layers, neurons(nodes), and weights(arrows). The layer is a structure that holds neurons. There are three types of layers. The input layers are responsible for take in inputs. In the digit recognition case, this layers will be populated with pixels of a image. The hidden layers holds neurons for the intermediate computations. The output layer holds neurons that are used to guess the digit of the input image.



The neuron which represents a node is a computational unit. It takes one input and output a value, and the function it uses is called an activation function, σ. In our case, we will use hyperbolic tangent tanh as an

activation function σwhich map the input to [-1,1]. The weights corresponding the internal states of the neural network. Each weight has a real number value. The output of a neuron is determined by those weights. Part of the learning process is to adjust those weights so that the guessed outputs matches the expected outputs.



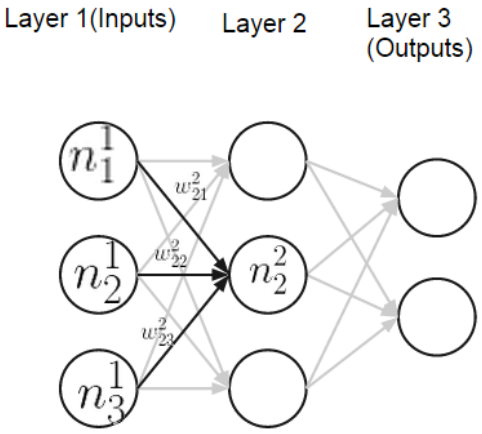
(Figure 1)

Figure 1 describes the neural network in which we are going to use for handwritten digits recognition. The inputs layer consists of neurons in which each neuron has a corresponding pixel value from image. Then the neural network will produce a result represents by the output neurons. Each of those output neurons corresponds to a digit, and the output of the neuron represent the likelihood of the corresponding digit be the actually digit where -1 means very unlikely and 1 be very likely. We will pick the highest output value from the output neurons to be the predicted/guessed digit.

Neural Network Serial Solution

How do we use the neural network model to map an input image to a output digit ,and how do we we adjust the neural network so that it can output a digit more accurately next time? The answer lies in the neural network specific algorithms, feedforward propagation and backward propagation.

Feedforward propagation is the process of mapping a given input, a handwritten image, to a output, an array of output value correspond to a digit from output layer neurons.



Output Of each Neuron:

$$n_j^l = \sigma(\sum_k w_{jk}^l * n_k^{l-1})$$

Activation Function: $\sigma = \tanh()$

Range = [-1, 1]

$$n_2^2 = \sigma(w_{21}^2 * n_1^1 + w_{22}^2 * n_2^1 + w_{23}^2 * n_3^1)$$

(Figure 2)

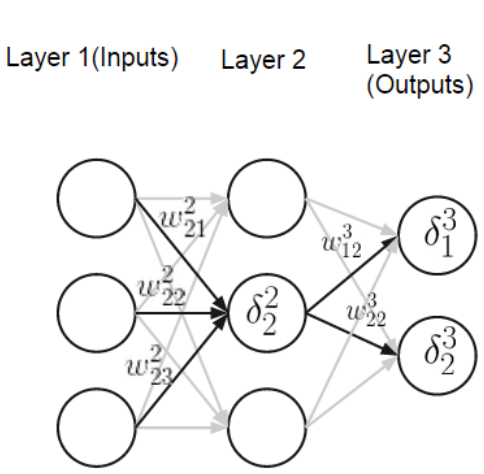
Figure 2 depicts the process of feedforward propagation algorithm. For each neuron except the input layer neurons, we need to calculate its output indicated by n_j^l where l is the layer index and j is the neuron index within l layer. The formula is given in figure 2. A example of calculating the output of a given neuron is given the lower left corner of the figure 2. From example, the output value of a neuron is depending on the dot product of weights and previous layer neurons' output values. Notice that each neuron's output value of a given layer is depending on the previous layer's neuron' output value. In addition, the computation of each neuron's output value is independent of each other within a layer. This is the part where parallelization can occur. The following is the pseudo code of feedforward propagation in which I described above.

Algorithm 1 Feedforward algorithm

```
1: procedure FEEDFORWARD PROPAGATION(net)   ▷ The net contain layers and neurons
2:     for <layer l in net > do
3:         for <neuron n_j^l in layer l> do
4:             n_j^l'outPut = ComputeOutPut(layer l − 1, neuron n_j^l )
5:         end for
6:     end for
7: end procedure
8: procedure COMPUTEOUTPUT(previousLayer l − 1, neuron n_j^l )
9:     sum ← 0
10:    for <neuron neuron n_j^{l−1} in previousLayer l − 1> do
11:        sum ← sum + n_j^{l−1} * w_{jk}^l
12:    end for
13:    return tanh(sum)
14: end procedure
```

Back Propagation

After computing an output from feedforward propagation, we will use this output to compare expected output that's associate with the input image. Then we will use this comparison to update the weights of neural network so that the neural network can generate a output that's closer to the true one at future. The algorithm does the above is called backward propagation. The following is a illustration of the algorithm.



Step 1: Calculate error for each neuron

$$\delta^l_j = \sum_k w^{l+1}_{kj}\delta^{l+1}_k \sigma'(z^l_j)$$

Example:

$$z^2_2 = w^2_{21}n^1_1 + w^2_{22}n^1_2 + w^2_{23}n^1_3$$

$$\delta^2_2 = (w^3_{12}\delta^3_1 + w^3_{22}\delta^3_2)\sigma'(z^2_2)$$

Step 2: Update Weights W base on error

$$w^{l+}_{kj} = w^l_{kj} + \eta\delta^l_k n^{l-1}_j$$

Example:

$$w^{3+}_{12} = w^3_{12} + \eta\delta^3_1 n^2_2$$

(Figure 3)

The backward propagation algorithm has two stages. The first stage, we need to compute the error/gradient term for each neuron. We compute the error term for the output layer's neurons first where we use the actual outputs and expected outputs to achieve that. The implementation of that computation is not included for simplification purpose. it is basically a vector difference calculation. For each neuron in hidden layers, we will use the formula under step 1 in figure 3 to compute an error/gradient term. From the calculation example, the error term depends on two dot products. One is the z term which is the input value of the given node. The other one is the dot product between weights and the next layer's error terms. This part shows that there is a layer dependency between the current layer and next layer. So only after we compute the error term for this layer before we can compute the error terms for previous layer. Notice that the backpropagation travels backward from layer l to layer l-1. Another important observation, the neuron within the same layer does not has any dependency on any other neuron within the same layer. This observation is the key to parallelization. After we compute the error terms, we will use it to update the weights of the neural network described by step 2 in the figure 3. The following is the pseudo code for backward propagation in which I described above.

Backward Propagation:

Algorithm 2 Backward Propagation algorithm

```
1: procedure BACKWARD PROPAGATION(net)  ▷ The net contain layers and neurons
2:     for <layer l in net , l-- > do  ▷ This For loop computes error for each neuron
3:         for <neuron n_j^l in layer l> do
4:             δ_j^l = ComputeError(Layer l + 1)  ▷ Error/Gradient = δ_j^l
5:         end for
6:     end for
7:     for <layer l in net , l-- > do  ▷ This For loop update weights in the net
8:         for <neuron n_j^l in layer l> do
9:             UpdateWeights(neuron n_j^l) ▷ Update Input Weights of neuron n_j^l
10:        end for
11:    end for
12: end procedure
```

ComputerError() and UpdateWeights() functions are left out for simplicity.

ComputerError() does

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$$

UpdateWeights() does

$$w_{kj}^{l+} = w_{kj}^l + \eta \delta_k^l n_j^{l-1}$$

Parallelization Strategy

After we had implemented a working serial solution of the network we began to implement a parallel solution. The first thing we had to consider for this solution was how to allocate the net on the GPU. The net cannot be stored on the CPU since for each training iteration we would have to allocate parts of the net on the GPU then run an operation such as feed forward, and finally copy the net back from the cpu to the GPU. In our implementation we decided to reduce memory allocations as much as possible, therefore we decided to allocate the entire network on the GPU. In this method we only needed to allocate the inputs to the network and the expected target values, on each training iteration. To allocate the network on the GPU, we decided to store all the net information such as the topology, the weights, the gradients, and the output values using one dimensional arrays. We chose not to uses a two dimensional structure since these nets all have different layer sizes, and our two dimensional arrays would be very complex. The tradeoff however is that for the one dimensional arrays we had pass in starting indices with each kernel call.

Parallel Feed Forward

To do the feed forward in parallel we had one kernel that did the feed forward on an entire layer, which we called in a for loop so that we could call the feed forward as many times as needed. On the CPU before we loop loop through the layers, we had to latch our input layers to output the pixel values that were passed in. To do this we ran a

latch kernel that would load for image onto the first layer on the network. SInce, the layer is launched several times we needed to compute the number of blocks to launch a specific kernel with, since we know that we need as many threads as the next layer we made sure we had enough blocks to accommodate that next layer. Finally, Before we could call the kernel we also had to compute offsets for the weights and output values in the loop, these along with the current layer were passed into the kernel.

Main kernel code:

```
__global__ void feedForwardkernel(double * weights,
                    double * nueronoutputvals,int *topology, int currlayer, int outoffset, int woffset){
        unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
        double sum = 0.0;
        //optimized by using a temp variable to store next layer size
        int size = topology[currlayer+1]-1;
        if(i<size){
                for(unsigned n=0; n < topology[currlayer]; ++n){
                        sum += nueronoutputvals[outoffset+n] *    weights[woffset + (n*size) +i];
                }
                sum/=(topology[currlayer]/2.0);
                nueronoutputvals[outoffset+topology[currlayer]+i]=tanhf(sum);
        }
}
```

The kernel is not very complicated since the kernel only needs to run the feed forward on one layer. Each thread in the kernel corresponds to a neuron in the next layer, size that is how many output weights each neuron in the current layer has. Each thread then loops through the previous layer calculating the dot product. Finally, after that value is computed each thread can set the output value of the next layer to be the hyperbolic tangent of the dot product.

Parallel Back Propagation

Back propagation was a lot more difficult to implement of the GPU then feed forward, since there are so many more steps involved. Eventually we broke the process down into three steps calculate output gradients, calculate hidden gradients, and update input weights. The first step, calculate output gradients only needs to be done once since there is only one output layer, the other two steps, calculate hidden gradients and

update input weights need to be be done in loops since they need to run on multiple layers of the network. To do these three steps we had to write a kernel for each step.

We know that there will only be ten output values since we are only classifying ten things. Therefore, there are ten gradients we need to find using the calculate output gradients kernel, this is quickly done in parallel with a simple kernel. Calculating the hidden gradients is a lot more complicated since there is another dot product we need to do in the kernel.

```
__global__ void calcHiddenGradientskernel(double * weights,double * gradients, int outoffset,int woffset, int
                                                        * topology, int currentlayer, double * outputvals){

        unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
        if(i < topology[currentlayer]){

                double dow = 0.0;
                for(int n=0; n< topology[currentlayer+1] - 1; ++n){

                        dow+=weights[woffset + (i*(topology[currentlayer+1]-1)) + n]  *
                                gradients[outoffset+topology[currentlayer]+n];
                }
                gradients[outoffset+i] = dow * (1.0 - (outputvals[outoffset+i]*outputvals[outoffset+i]));
                gradients[outoffset+i] /= topology[currentlayer+1];
        }
}
```

The hidden gradients kernel needs to calculate the gradient for a certain layer, we need to pass in the layer that we are currently working on. The kernel has a thread for each neuron in the layer and then each thread loops through each neuron in the next layer. We have a minus one in the for loop to avoid the bias neuron. Here use the gradients of the next layer which have already been computed and the weight between the two neurons to compute the gradient. Then we set the gradient for the neuron and divide by the size of the next layer. After all the gradients have been computed we need to update the input weights.

```
__global__ void updateInputWeightskernel(double * weights,double * gradients, double* outputvals,
                                        int woffset, int outoffset, double * deltaweights, int *topology, int currlayer){

        unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

        if(i < topology[currlayer] - 1){
        for(int n = 0; n < topology[currlayer-1]; ++n){

                double newDeltaWeight = .39 * outputvals[outoffset-topology[currlayer-1]+n]
                        * gradients[outoffset+i]+ .1* deltaweights[woffset + (n*(topology[currlayer]-1)) +i];
```

```
                deltaweights[woffset + (n*(topology[currlayer]-1)) +i] = newDeltaWeight;
                weights[woffset + (n*(topology[currlayer]-1)) +i] += newDeltaWeight;

                }
        }
}
```

We need to go through the entire net and for each layer, we need to call this kernel. This kernel will update each weight for each neuron. We have a thread a thread for each neuron, and then loop through each weight in the neuron. We need to keep track of delta weights since we use the previous weight to calculate momentum.

Related Work

Most of the implementation for neural network is serial due the fact that most of computers in the world do not have a dedicated graphic card and are not able to do heterogenous computing. However, we do found a parallel implementation of neural network from a research paper, Deep Big Simple Neural Nets Excel On HandWritten Digit Recognition. This research paper describes a parallel implementation of feedforward implementation that is different than our solution. The following diagram from the research paper describe its parallel feedforward strategy.

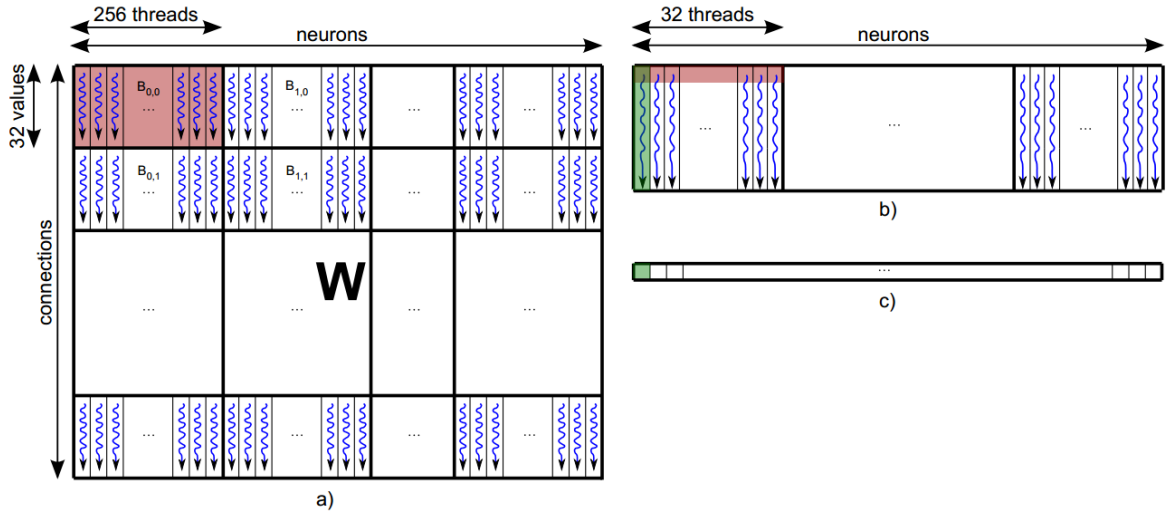


Figure 2: Forward propagation: a) mapping of kernel 1 grid onto the padded weight matrix; b) mapping the kernel 2 grid onto the partial dot products matrix; c) output of forward propagation.

The general idea is the following. The thread block is build in a way such that each thread corresponding to a neuron in the neural network, and this thread will do a partial dot product between the previous layer output values and the weights. Recall that in figure 2, each neuron need to compute a full dot product in order to compute its output. In addition, there another thread corresponding to the exact same neuron in a different block will do the next partial dot product as shown in above figure part A. Then another kernel is going to add those partial products together shown in part B of the above figure. Then it will feed to sum of the partial products to the activation function to obtain the correct output value of a neuron as shown in part C of the above figure. With this implementation, one can observe that the dot product of a neuron is broken into many small partial dot products. Those partial dot products can be computed parallelly within a layer. Thus, the research paper's implementation utilize more parallelism than our implementation. Even though the paper's algorithm utilize parallelism more, our solution is more readable and easy to understand.

Parallel Optimization

The major optimizations we made in our parallel approach was switching from a single block approach to using multiple blocks in our kernels. This would allow us to run simulations on even larger networks which is key in reducing our error. Also since we used multiple blocks we were able to adjust the amount of blocks each kernel ran on to exactly the amount of blocks we would need for that particular layer. Overall this approach was more scalable which is why we chose it. Another optimization we implemented was using a kernel to load network inputs in parallel since before we were doing a memcpy loading then on the CPU and then doing another memcpy. This allowed us to reduce to amount of memory that is copied in each training loop since memory copy can be very expensive.

Evaluation

Setup:

Macbook Pro 15 2014
Cpu : Intel Core I7 2.5 GHz Quad Core
Ram : 16 GB
Graphics : NVIDIA GeForce GT 750M with 2GB of GDDR5 memory
CUDA Cores: 384
CUDA Verison: 3.0
Memory Bandwidth: 80 GB/S

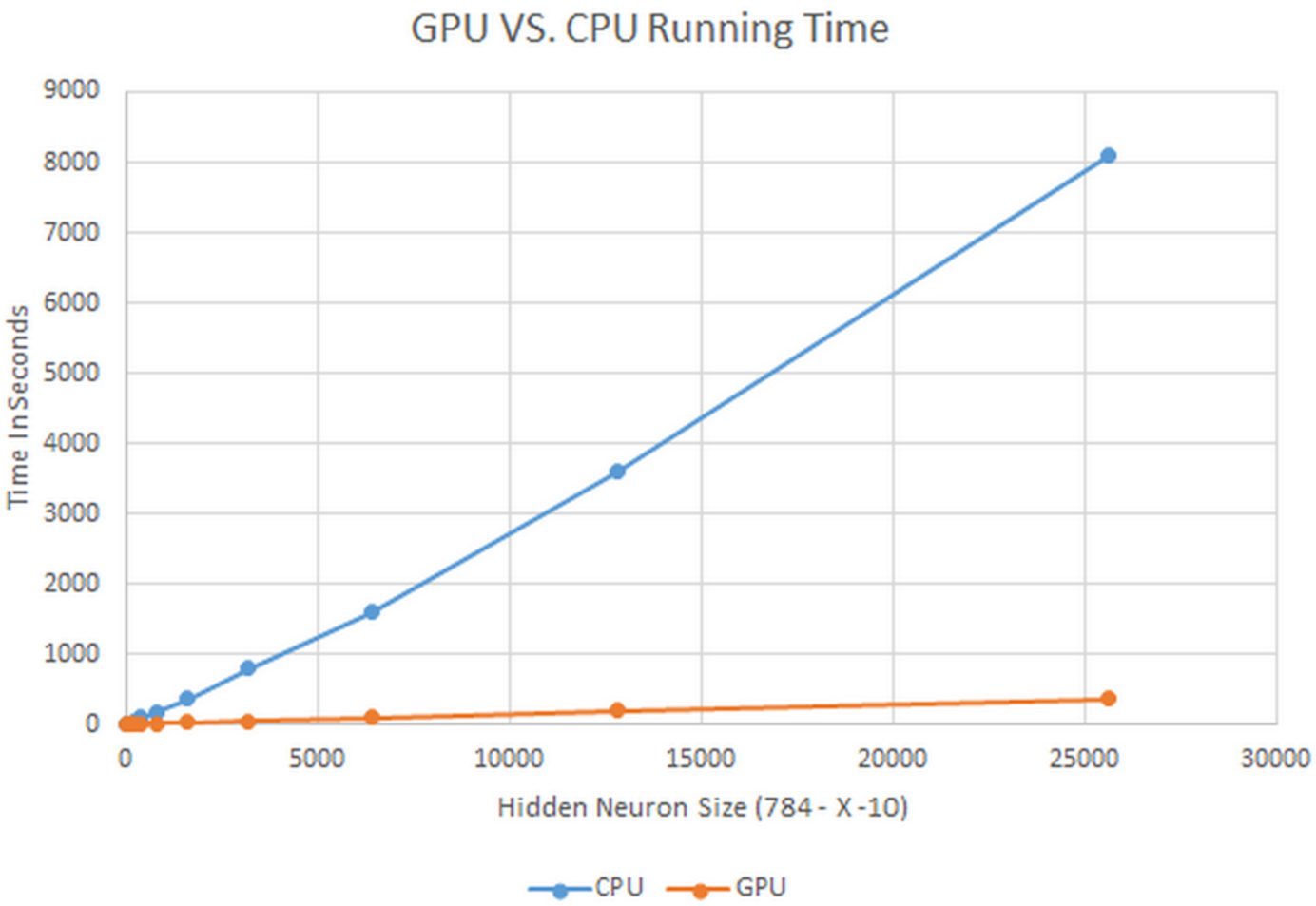
We implemented the parallel solution in gpu of above specification. We run the following performance measurement on a training set of 5000 images with only one iteration through the dataset.

Performance analysis:

X Hidden layer Size	CPU (seconds)	GPU (seconds)	
0	1.59907	6.8193	0.23449181
50	7.86117	10.1952	0.771065796
100	17.9471	10.6882	1.679150839
200	38.8591	11.6437	3.337349811
400	94.6226	13.4148	7.053597519
800	190.825	17.5959	10.8448559
1600	364.51	27.1136	13.4438068
3200	793.528	49.0034	16.19332536
6400	1596.69	96.8787	16.48133181
12800	3603.66	197.473	18.24887453

25600	8098.67	374.336	21.63476128

(Table 1)

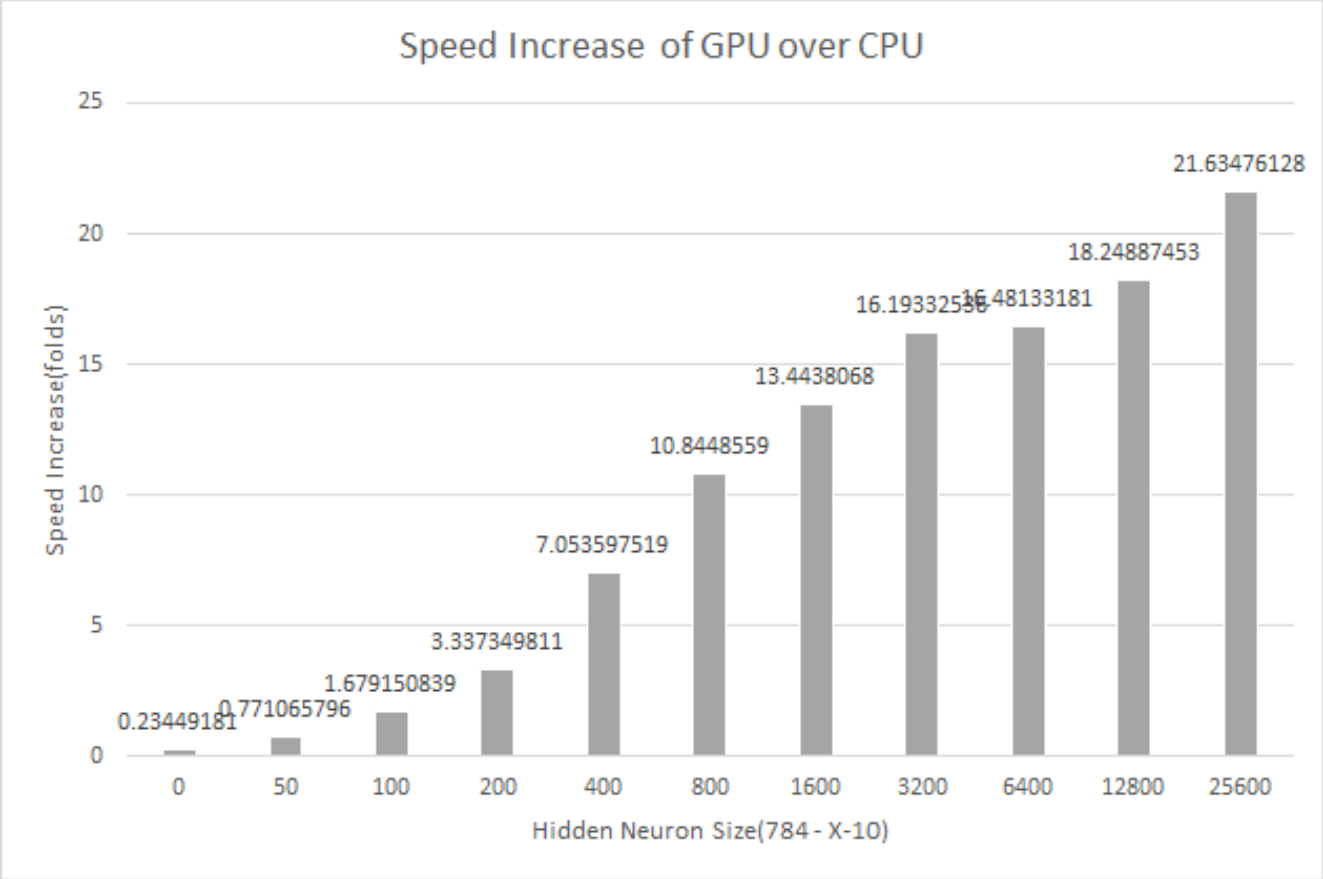


(Figure 4)

Figure 4 shows the running time for both serial and parallel implementation on a net size of 784 - X - 10. In order to compare the running time of different neural network size, we confined the dimension of neural network to a 3 layers neural network with only 1 hidden layers. The input and out layers are fixed due the fixed input image dimension and 10 digits outputs. Thus, only the hidden layer is changed to measure performance regarding neural network size. From the serial solution, the running time become unbearable as the hidden layer size of 12800. It takes 3603.66 seconds to complete training of only 5000 images. Imagine if the training set size is 50000, it will take 10 hrs to run one iteration of dataset. Normally, neural network need to be trained over many dataset iterations in order to get a good result. Thus, the serial solution limits the size of the neural network and experiments can be performed. On the contrary, the parallel

solution running under the same condition only takes 197.473 seconds which is 3.3 minutes. If the data size is 10 time greater, it will only takes 33 minutes to complete. The parallel solution significantly decreases the running time. This allows researcher to do more experiments and play around with bigger nets which could potentially speed up their research.

The following diagram illustrate the speed increase of parallel solution over serial solution.



Future Work

While our Neural Network is currently functional to use with digits, some correctness and functionality improvements could be made. Two things that we are looking into for the Neural Network is to increase the set of training data we have by doing convolutions on the existing training data, and also extend the Neural Network to also work with characters. Doing this will increase both correctness and possible

use-cases for our Neural Network. For example, given a low enough error on both digits and characters, a post office could use our Neural Network to quickly parse the address field on letters. Also now that we have built a net that can run in parallel there are a lot of other areas we can apply it to instead of just digit classification.

Net Accuracy

Our main goals in this project were to improve the speed of a neural network by writing it in parallel, we did not focus too much on how accurate our results were. However we still ran many simulations of the network to see what accuracies we were getting and made slight improvements to the net logic to increase these accuracies. Overall our best simulations would get us too about a 96 - 98% accuracy when run on a decent next size over several iterations of the training data. These results are not too terrible considering it was not our primary objective. One thing we did do was write, an application that allowed us to paint new numbers and then run the net on those numbers. This allowed us to see our network run on actual data that a user can input, instead of running on data we never see.

Conclusion

Neural Networks have a lot of potential for improvements in speed, especially through parallelization. Our experiment in particular yielded some fair gains in speedup. With regards to our parallelized Neural Network, there is still work to be done such as doing convolutions on the MINST training data to increase the set of training data for greater accuracy. In addition to this, there are surely some optimizations that can still be made in the code, such as taking greater advantage of cache behaviors. However, with all these points in mind, the future for parallelized Neural Networks looks bright.

Main Contributions.

Fanmin Shi: I read through different tutorials on neural network. I make sure that the serial algorithm we implement is correct. Then I implemented my own parallel feedforward and backward propagation on a modified serial implementation which Avdhesh wrote. I compared my parallel solution with Avdhesh's solution to check which one is better. Eventually, we decided to used Avdhesh's solution because it is more scalable than mine. I also helped on optimize his parallel solution.

Dennis Liang: I actually did not know anything about neural networks at the start of this project. I spent a significant amount of time reading and learning about neural networks, and then wrote the serial saver/loader, and then the parallel saver/loader when we implemented the parallelized version of our code.

Avdhesh Garodia: I began this project by researching about neural networks, I have had some exposure to them in the past but I had never implemented one. I helped with the implementation of the serial net. After that both me and Fanmin worked together on creating the parallel version of the network, each of us writing our own code separately and then comparing results. I also wrote the painting application to test our network with new images, since I wanted to see how well it handled that.