Brian Crafton
High Performance Computing
February 24, 2016

Project Update: Parallelizing A Neural Network

Neural networks are a popular form on machine learning used for classifying data. They use a forward and back propagation technique to compute a gradient and a cost. The gradient is a computed value of the direction of the solution and the cost is the scaled "failure" of the current solution. Using these two values computed using forward and back propagation an optimization method can find a local minimum solution. By fine tuning the value for the learning parameter the programmer can hit a good local minimum rather than one that is not a very good solution. Machine learning uses statistics to give accurate predictions based on training data. A neural network is a supervised, discrete implementation of machine learning. Supervised machine learning uses training data that has already been classified. Discrete machine learning classifies the input data as a class or category as opposed to giving a continuous prediction. In machine learning it is important to be able to gauge the accuracy of the classifier. Fortunately is is very simple to measure the accuracy of a neural network as one can simply take the largest probability from their classifier and check if it maps to the correct value.

Neural networks require lots of matrix arithmetic and lots of repetitive steps. Matrix arithmetic requires lots of independent instructions, making it ideal for parallel computing. Operations like matrix addition, matrix transpose, matrix multiplication, matrix splitting, ect are all computationally intensive operations that have lots of (embarrassingly) parallel operations. This type of parallelism is data parallelism because the same computation is being done just on different data. To exploit data parallelism I plan to use SIMD/SIMT and multi-threading.

This project is based on the neural network section (weeks 4 and 5) in Stanford's machine learning course on *Coursera.com*. The application for this project is classifying hand written digits, so taking 20x20 pixel (greyscale, uncompressed) images and classifying each one as a digit (0-9). The data I am using is a 5000x400 matrix which I have taken from Stanford's Machine learning course on *Coursera.com*. Each row (400 double floating point values) represents a 20x20 greyscale image that is a handwritten digit. I completed the class assignment in Matlab, but the implementation for this project is written in C, specifically written for the gnu gcc c compiler. There are two intentional design flaws from the start.

1. I have written basic implementations for the matrix arithmetic from scratch
2. I am using gradient descent rather than using a popular optimization function

The reason for these design choices is because the assignment is about learning rather than building an effective neural network. Success will be determined by the parallel speed up achieved rather than the actual accuracy of the neural network. There are two high level techniques that will be investigated.

1. Paralleling the matrix arithmetic
2. Giving each thread its own neural network iteration in the neural network and have each thread perform sequentially and come back together to compute the new classifier.

There will be tradeoffs between both, but after doing some trial and error I am hoping that the better technique will be clear.

References and Links:

Unfortunately because one must sign up to view the material on Coursera I cannot provide a direct link to the material, but I have found a site that displays in depth notes on the course.

http://www.holehouse.org/mlclass/09_Neural_Networks_Learning.html

Instructions for running my code and the bash script in order to run it on the discovery cluster is

available at:

https://github.com/bcrafton/neuralnetwork

Results:

In the below table I have included the resulting accuracy and time elapsed for each set of 100 iterations. This run took roughly an hour and used close to 5000 seconds or about 80 minutes. I currently have the project set to use initial random matrices I generated awhile ago. This means that one should get the same accuracy result for any number of iterations listed below.

This run was done on the discovery cluster. The code was timed using the same method we used in class for the first sequential assignment. At the start of gradient descent I took a start time stamp with:

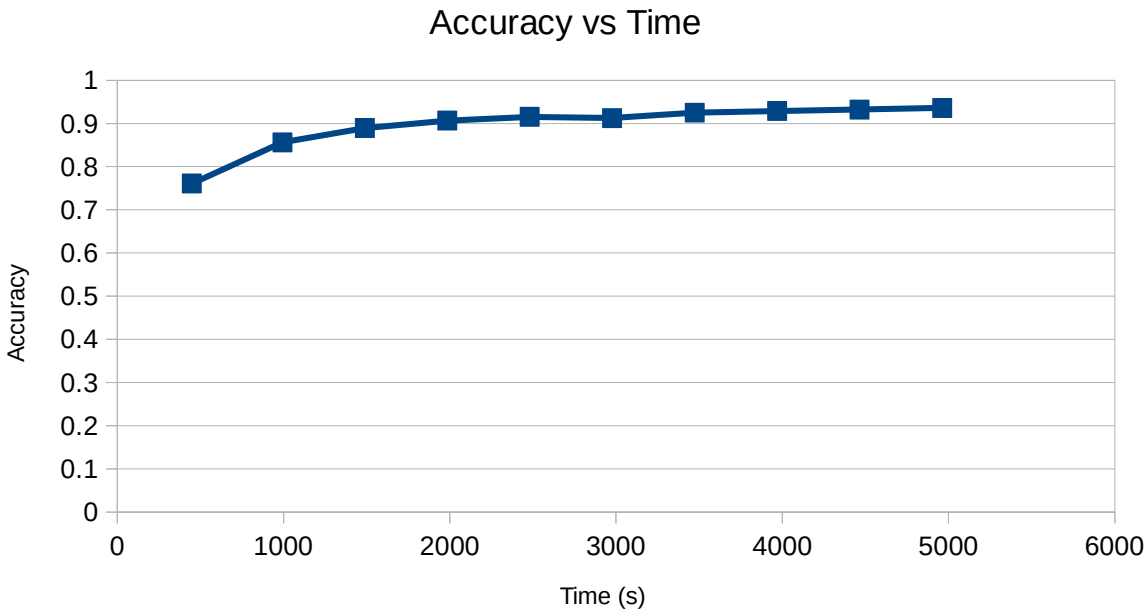
```
start = clock();
```

And each time I reached a 100 iteration mark I took a stop time stamp and printed the difference:

```
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

The finer grained timing allows graphs to be made for accuracy versus time and so that I can compare runs with lesser iterations to runs with larger iterations and make sure the accuracy and time used is very similar.

Iteration #	accuracy	Time used
100	.7610	448.26
200	.8562	993.88
300	.8894	1489.12
400	.9066	1984.85
500	.9154	2480.73
600	.9124	2976.03
700	.9250	3473.44
800	.9290	3969.63
900	.9322	4465.32
1000	.9358	4961.81


Accuracy vs Time
Accuracy
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1000
2000
3000
4000
5000
6000
Time (s)