CS759 Final Project Report
University of Wisconsin-Madison
Technical Report 2017

Speed Up Neural Network Training using Parallel Computing with OpenMP

Trang Vu, tvu2@wisc.edu

December 21, 2017

Abstract

Artificial neural network training can be time consuming. In this project, an artificial neural network was constructed to classify sonar signals. OpenMP was successfully used to parallelize the training process and the speed up is 15 times faster for 25-fold cross validation using 100 epochs per training set.

Contents

1 Introduction

Artificial neural networks are widely used in machine learning as one of many learning algorithms, which are developed to make prediction for specific outputs given known input values. In other words, the neural network learns to make more accurate prediction through network training using real observations. For example, neural networks can be trained to classify sonar signals bounced off a metal cylinder or a cylindrical rock, or to regconize speech and hand-written texts, or to steer autonomous vehicle [2, 3]. Figure 1 below shows a schematic view of a neural network. Typically, a neural network is a fully conected network which often consists of one input layer, one output layer and one or more hidden intermediate layers. Input layer and output layer contain nodes that represent input and output signals respectively. The hidden layers contain multiple nodes called neurons that process input signals and pass them to the output layer as output signals. [1]

Each connection between an input node and a neuron has a weight that reflects the contribution of that particular input signal to the system. Similarly, the connections between the neurons and output signals are also weighted. The neural network learns to make prediction through rigourous training with multiple subsets of training data (cross validation) to tune the weights of the connections within the network. Specifically, initially weights are randomly assigned for all connections in the neural network, and the intermediate output from a particular neuron is calculated as weighted sum of the values of the input signals and the weights of the connections between all inputs to that neuron. The weighted sum is further processed through a sigmoidal activation function to help stabilize network activity [1]. The activated outputs from all neurons of the first hidden layer then serve as input signals for the next hidden layer of the neural network. Eventually, the

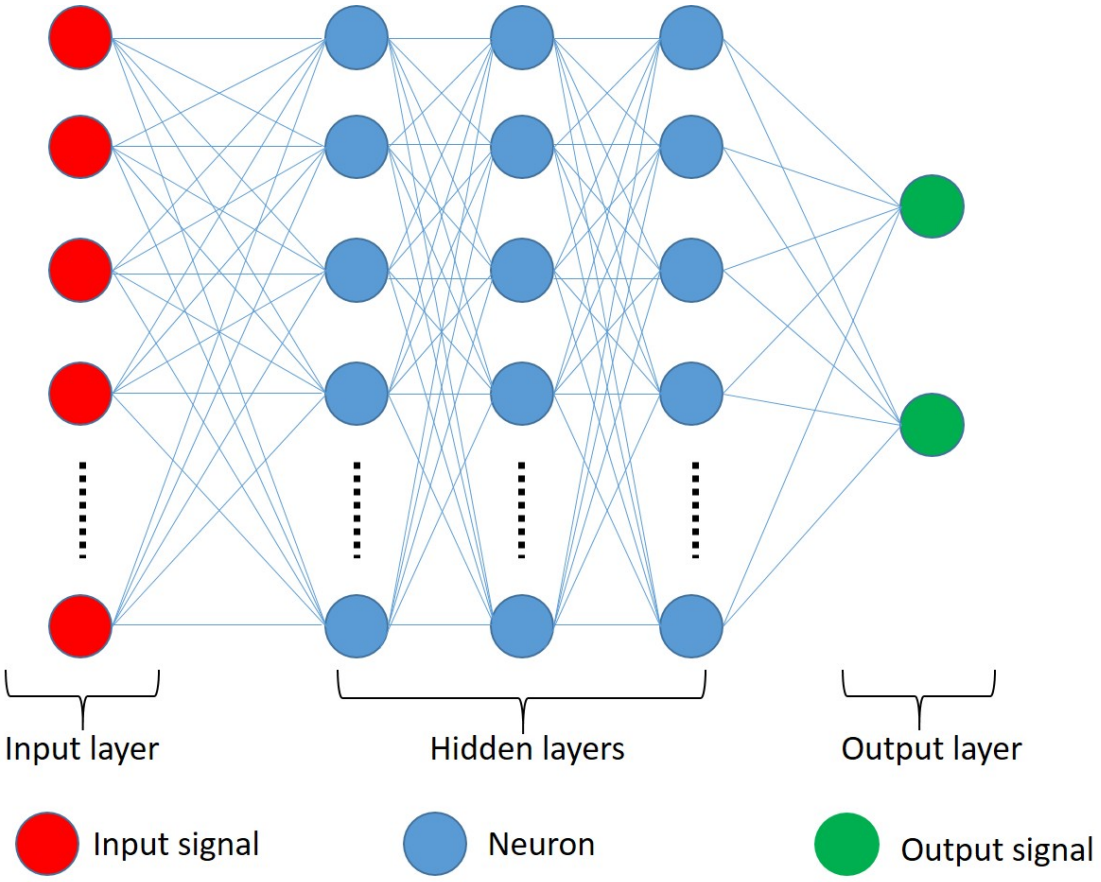

Figure 1: Schematic diagram of an artificial neural network.

intermediate activated outputs from the last hidden layer reach the output layer where the weighted sums are calculated and activated to produce final output signals. This process of passing and processing input values to output values through the network called feed forward propagation and it applies to one instance of training data. Once the output signals are produced, they are compared with the actual output signals and the differences (or errors) betweens the predicted and the actual outputs are used to calculate the errors associated with each intermediate ouput signals of the neurons in the hidden layers. In other words, these errors are propagated backward through the hidden layers to the input layer. This process called backpropagation of errors, and it often uses stochastic gradient descent to compute the errors [1]. The errors are then used to update the initial weights of the corresponding connections in the network. Once the network is updated with a new set of weights, the cycle of feed forward propagation - backprobagation - update network is repeated for the next intance of the training data. This process of updating network's weights at every instance called online training [4]. Sometimes, the weight updating step can be done using the average errors of multiple data instances (batch training) [4]. For each set of training data, the training is often repeated in a number of epochs to ensure stability of the stochastic gradient descent's results [1]. Furthermore, cross validation

is often used in training and testing of neural networks or any other machine learning algorithms such that a data set is divided into a number of folds, and iteratively each fold is held out as a testing set while ther other folds are combined into a training set . Each training set is used to train the network and the trained network is then used to make prediction for the held-out testing set. As one can imagine, neural network training can be very time consuming especially when there are many hidden layers and each layer has many neurons, and the training requires many epochs, folds, and large training data set. Therefore, it is desirable to design a framework in which one can employ parallel computing to speed up the training process.

In this project, I constructed a neural network classifier to classify sonar signals. The network contains one input layer with 60 nodes representing 60 attributes of the data set, one hidden layer with 60 neurons, and one output layer with one output node. The construction of the neural network and sequential network training are discussed in more detail in section "Sequential Neural Network Training." I then assessed and explored the suitability of different parallel computing models such as OpenMP [5], GPU computing with CUDA [6], and MPI [7] that I learned from ME759 [8] for parallelizing the training of the constructed neural network. This is discussed in section "Parallel Neural Network Training." Using OpenMP, the neural network training is **15 times faster** compared with sequential training with 25-fold cross validation, and 100 epochs using 25 threads. Section "Results and Discussion" also discuss other scaling analyses of sequential computing and OpenMP computing models. Due to time constraints, I was only be able to complete the implementation and analysis of OpenMP, which leaves other parallel computing models' implementation for future work, which is discussed in section "Conclusions and Future Direction").

2 Sequential Neural Network Training

The neural network was constructed with one input layer, one hidden layer and one output layer. The input layer has 60 nodes corresponding to 60 different signal attributes of the data. As a rule of thumb, the number of neurons in the hidden layer can be any number between the number of input nodes and output node [4], so I let the hidden layer have the same number of nodes as the input layer (60 neurons). The data set has 208 instances with each instance has 60 numerical attributes representing sonar signals and one binary class attribute representing actual output [3]. The neural network is fully connected and each connection has a weight randomly initialized to a value between -1 and 1. Mathematically, a network is reprenseted by the following parameters: the weight matrix $\mathbf{W}$ in which each entry is the weight of connection between an input node to a neuron in the hidden layer, weight vector $\mathbf{u}$ in which each entry stores the weight of connection between a neuron in the hidden layer to the output node. Therefore, matrix $\mathbf{W}$ has $N \times M$ elements with N being the number of attributes (input nodes) and M being the number of hidden neurons, and vector $\mathbf{u}$ has M elements. Additionally, there is a bias unit for the hidden layer which is represented by a vector $\mathbf{b}$ of M elements, and a bias scalar value c for the output layer. These bias vector and scalar are also initialized to values between -1 and 1.

Once the network parameters are initialized, the input values are propagated forward

through the hidden layer to the output layer where intermediate outputs and final output values are computed respectively. Let $\mathbf{x}$, $\mathbf{h}$ and o be the vector of the inputs, intermediate outputs and the final output respectively, the computations involved in this feed forward step are shown in equations below. Equations (2) and (4) are the sigmoidal activation functions which are used to produce smooth output signals [1].

$$z_1 = W \cdot x + b \tag{1}$$

$$h = \frac{1}{1 + e^{(-z_1)}} \tag{2}$$

$$z_2 = u \cdot h + c \tag{3}$$

$$o = \frac{1}{1 + e^{(-z_2)}} \tag{4}$$

After the output is computed, it is then compared with the actual output to calculate the errors associated with the network's parameters (backpropagation of errors). Since the actual output values of the data are represented by strings, they are converted to binary values of 0 and 1 to make the comparison easier. The difference between the actual output and the computed output is then used to compute the errors associate with the bias value c, bias vector $\mathbf{b}$, the weight vector $\mathbf{u}$ and the weight matrix $\mathbf{W}$. Specifically, the error of a network parameter (c, $\mathbf{b}$, $\mathbf{u}$, or $\mathbf{W}$) is derived as the partial derivative of the cost function with respect to that parameter. Let y be the actual output value of an instance, the cost function used in this project, which is the cross-entropy function, is represented by equation (5) below. Derivation of the partial derivatives is beyond the scope of this work so here I only showed the final equations used to implement the backpropagation algorithm. Let $\frac{\partial E}{\partial c}$ be the error associate with bias scalar c of output unit, $\frac{\partial E}{\partial b_j}$ be the error associated with the j^{th} element of the bias vector $\mathbf{b}$, $\frac{\partial E}{\partial u_j}$ be the error associated with the j^{th} element of the weight vector $\mathbf{u}$, and $\frac{\partial E}{\partial W_{jk}}$ be the error associated with the weight of the connection between the k^{th} input node and the j^{th} neuron in the weight matrix $\mathbf{W}$, equations (6)-(9) show how to compute these errors.

$$E = -ylog(o) - (1 - y)log(1 - o) \tag{5}$$

$$\frac{\partial E}{\partial c} = y - o \tag{6}$$

$$\frac{\partial E}{\partial b_j} = (y - o)u_j \tag{7}$$

$$\frac{\partial E}{\partial u_j} = (y - o)h_j \tag{8}$$

$$\frac{\partial E}{\partial W_{jk}} = (y - o)u_j h_j (1 - h_j)x_k \tag{9}$$

After the errors are calculated, they are used to update the weight and bias values in the network using equations (10) - (13) below. In this updating process, I used online training, which means that the weight is udpated after each instance of data.

$$c^{new} = c + \eta \frac{\partial E}{\partial c} \tag{10}$$

$$b_j{}^{new} = b_j + \eta \frac{\partial E}{\partial b_j} \tag{11}$$

$$u_j{}^{new} = u_j + \eta \frac{\partial E}{\partial u_j} \tag{12}$$

$$W_{jk}{}^{new} = W_{jk} + \eta \frac{\partial E}{\partial W_{jk}} \tag{13}$$

The constructed neural network is trained using k-folds cross validation in which the data set is splited into k folds. Iteratively, one fold is kept as testing set while the other folds are combined into a training set. In other words, the training is performed k times with a subset of the data. In this project, the data was partitioned such that each fold has roughly equal number of instances of the two output class labels, i.e. if the two class labels are "Rock" and "Mine", each fold would have roughly equal number of instances labeled as "Rock", and instances labeled as "Mine". For each training set, the network is initialized once. Then the cycle of feed forward - backpropagation - update is repeated for each instance in the training set. After the last instance in the training set is processed, one epoch is completed. Normally, this process is done iteratively in some number of epochs, so in a way, a fixed number of epochs is a stopping criteria of the training process. After the training is completed for one training set, the trained neural networks is used to make predictions for the held-out testing set to calculate predictive accuracy. The whole process of neural network training and testing can be shown in the algorithm below. In this project, I experimented the neural network performance (in terms of execution time) with different number of epochs and k-folds (see "Results and Discussion" section). The implementation of this sequential training algoirthm (algorithm M1) is implemented in C. The source codes of this implementation can be found under "Project" folder on GitHub [11].

Algorithm 1 Sequential Algorithm M1

```
Split data into k folds:  k training sets, k testing sets
for each training set do
   initialize network
   for (i = 0; i < epochs; i++) do
      for <each instance of training set> do
         feed forward
         backpropagation
         update network
   test network
```

3 Parallel Neural Network Training

As Knuth has said in his 1974 paper, "premature optimization is the root of all evil," it is imporant to understand where the bottleneck of a program is in order to optimize that program. Intuitively, I can guess where the neural network training spends most of its time, however, to avoid oversight, I used GNU profiling tool gprof [10] to profile the training algorithm developed in previous section. Figure 2 below is a visualization of the call graph produced by gprof from runing the algorithm with 5 folds and 25 epochs. In this call graph, the number of calls made by the "train_network" function to its children functions is the largest, which is expected because as seen in the sequential training algorithm above, each training instance requires one cycle of the "feed_forward", "backpropagation_error", and "update_network" functions, and this cycle is repeated many times because of multiple folds and epochs. Therefore, the program would probably run faster if the training can be done in parallel.

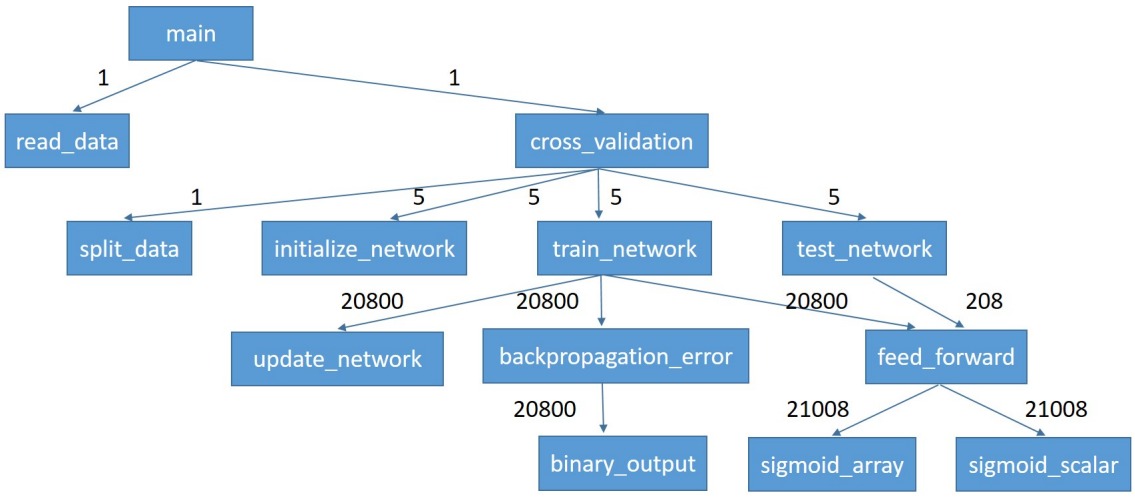

Figure 2: Call graph of sequential neural network training algorithm.

A closer look at the algorithm reveals that three functions under function "train_network" can not be done in parallel because of the input of one function depends on the output of previous function. However, the computations involved in each of these functions can be done in parallel. For instance, to update the weight matrix W, one need to loop through $N \times M$ elements. Because each neuron is indendent of other neurons, one can also parallelize this update process. One should avoid training each instance in parallel because the network parameters are updated after each instance so in a way, there is some data dependency involved in this process. Similarly, the network parameters are continuously updated in each epoch so perhaps one should also avoid parallelize the second "for-loop". In contrast, the outer "for-loop" is highly parallel because each training set can be trained independently of one another. So in summary, there are two possible places in the algorithm that can be parallelized, the outer "for-loop", and the loops inside functions "feed_forward", "backpropagation_error", and "update_network". The most simple parallel computing model that can be used to parallelize the for loops is OpenMP which

would assign each thread a subset of data to work on. Specifically, to parallelize the outer "for-loop", each thread will train a subset of training data independently. Similarly, to parallelize the "for-loops" inside the children functions of "train_network", each thread will work on a neuron independently. Since there is no communication of data, perhaps MPI is not suitable for these parallel computing tasks. If the number of folds is large or if the network is large (thousands of neurons organized in multiple layers), outsourcing the training to the GPU device and used CUDA to parallelly train the network would be more efficient than running on multicore CPU with OpenMP. Since the constructed network is fairly small and the size of the data is small (hence small number of folds), OpenMP is the most suitable parallel computing model for this project. In summary, I modified the sequential algorithm to produce two OpenMP algorithms with the first (algorithm M2) parallelizes the outer "for-loop", and the second (algorithm M3) is algorithm M2 plus the parallelization of the "for-loops" inside children functions of "train_network". I experimented the performance of each algorithm with different number of threads (see "Results and Discussion" section). Detail implementatio of the algorithms can be found in the source codes available under "Project" folder on GitHub [11].

4 Results and Discussion

4.1 OpenMP Output Verification

One of the two rules about parallel computing that I learned from ME759 is that the program is useless if it is fast but not correct. Therefore, it is important to verify the outputs produced from the developed OpenMP algorithms. In this project, the outputs are the accuracy obtained by testing the trained network on testing data. For example, if one test instance is classified as "Rock" using the neural network trained by sequential algorithm, the same instance should also be classified as "Rock" using the network trained by parallel algorithm. The challenge, however, is that even for sequential training algorithm, the outputs can vary because the initial parameters are randomly assigned so each run can be different from the next. With multi-core computing using OpenMP, each thread can work on a subset of training data with a set of initial parameters that maybe different from those used in sequential training. Consequently, the outputs produced by these programs may not be identical. To address this issue, the weights and bias values are fixed to some constants such that, each time function "initialize_network" is called, the same set of network parameters is used to train the network. This ensures that the output values produced by either algorithm are identical regardless of the order the training set is processed. Using this method, the parallel algorithms produced similar outputs to those produced by the sequential algorithm. The only difference is the order in which the outputs are printed out. However, one can sort the outputs by the instance's index to make the comparison task easier.

4.2 Scaling Analyses

Since parallel training algorithm has been verified to behave correctly, and I only care about performance in scaling analyses, the restriction that fixes network parameters to

constant values is removed. In addition, to remove jitter in the timing of the execution, I ran each algorithm 10 times and took the average time for each measurement condition. For example for 5 folds, 25 epochs, the sequential algorithm is run 10 times. Similarly for 5 folds, 25 epochs, 5 threads, the parallel algorithms are run 10 times.

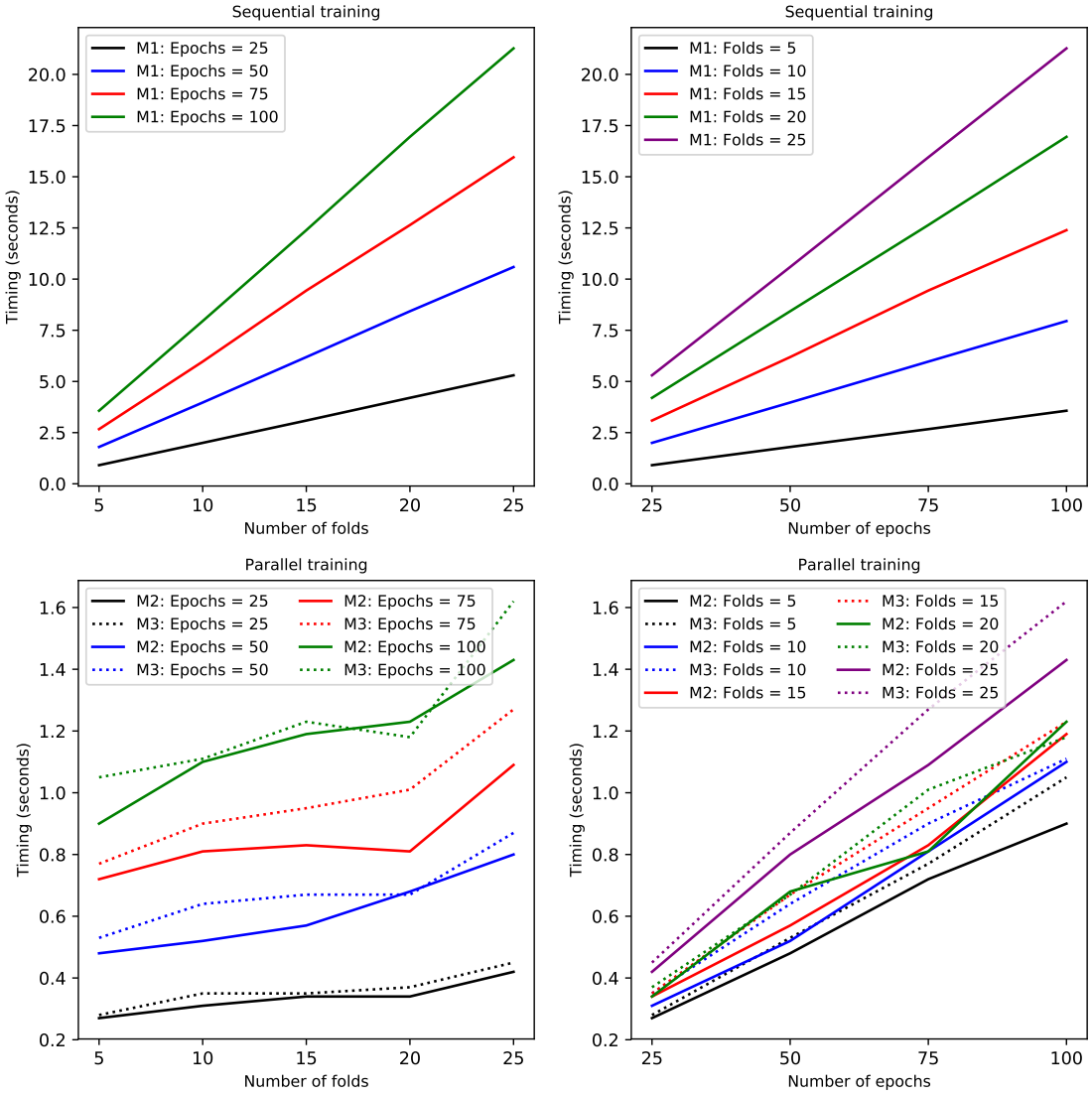

Figure 3: Scaling analysis of sequential and parallel training algorithms. M1 is the sequential algorithm, M2 and M3 are the parallel algorithms.

Figure 3 above shows how the execution time of the sequential (M1) and parallel training algorithms (M2, and M3) varies as the number of folds or epochs changes. With sequential training, the execution time increases linearly with the number of folds and number of epochs, which is expected as discussed previously. The same trends are observed with the parallel algorithms. Interestingly, algorithm M3, which is M2 plus extra

parallelized "for-loops" regions performed slightly worse than M2. Perhaps, because the network is relatively small (60 inputs, 60 neurons, 1 hidden layer), parallelizing the loops over the neurons creates much overhead that slowed down the execution. Regardless, both parallel algorithms siginicantly reduced the execution time. Table 1 below contains the data used to create this figure.

Table 1: Timings of neural network training with different models. M1 is sequential model, M2 is model with parallelized outer "for-loop", M3 is model M2 with extra parallelized "for-loops" in the feed forward, back propagation, and network update steps. The timing is measured in second. All analyses were run on Euler02 node except for those marked by $^{(*)}$, which were run on Euler01 node.

Folds	**Epochs**	**M1 (sec)**	**Threads**	**M2 (sec)**	**Threads**	**M3(sec)**
5	25	0.91	5	0.27	5	0.28
10	25	2.00	10	0.31	10	0.35
15	25	3.09	15	0.34	15	0.35
20	25	4.20	20	0.34	20	0.37
25	25	5.30	25	0.42	25	0.45
5	50	$1.80^{(*)}$	5	0.48	5	0.53
10	50	$3.97^{(*)}$	10	0.52	10	0.64
15	50	$6.19^{(*)}$	15	0.57	15	0.67
20	50	$8.43^{(*)}$	20	0.68	20	0.67
25	50	$10.59^{(*)}$	25	0.80	25	0.87
5	75	2.67	5	0.72	5	0.77
10	75	5.97	10	0.81	10	0.90
15	75	9.44	15	0.83	15	0.95
20	75	12.64	20	0.81	20	1.01
25	75	15.95	25	1.09	25	1.27
5	100	$3.57^{(*)}$	5	0.90	5	1.05
10	100	$7.95^{(*)}$	10	1.10	10	1.11
15	100	$12.39^{(*)}$	15	1.19	15	1.23
20	100	$16.95^{(*)}$	20	1.23	20	1.18
25	100	$21.27^{(*)}$	25	1.43	25	1.62

It should be pointed out that the number of threads used in the scaling analysis of the parallel algorithms are set to the number of folds in each measurement condition. The logic behind this setting is that each thread works on a subset of training data. Since there are k folds, there are k training sets, and hence k threads. As shown in Figure 4, the execution time decreases as the number of threads increases. However, the trend levels off after 25 threads. These analyses were run for 25 folds and 100 epochs so it makes sense that having more than 25 threads does not help improve the performance since each of the 25 training subsets is trained by one thread. With this setting, the speed up is about 15 times compared to the sequential training.

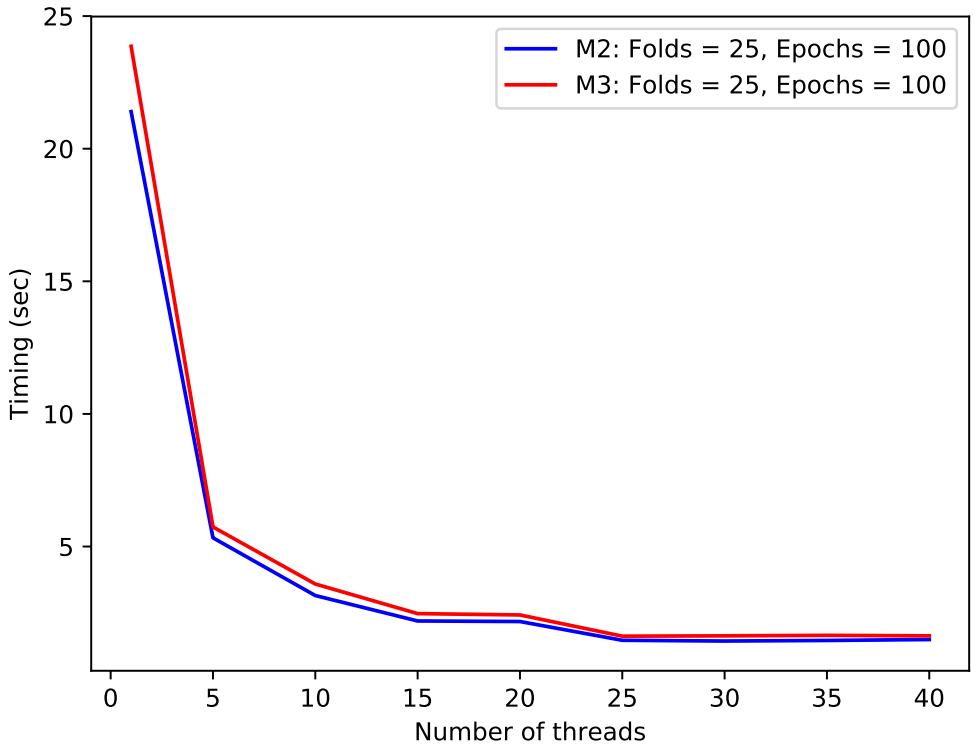

Figure 4: Scaling analysis of parallel training algorithms. The scaling analysis is done for 25 folds, 100 epochs.

5 Conclusions and Future Direction

Using OpenMP, I was able to succesfully speed up the execution time of sequential network training 15 times (for 25 folds, 100 epochs). While the implementation of parallel computing algorithm is simple, the development of the sequential algorithm took the most time of this project which left very little time to explore other data set as well as other parallel computing algorithms. Because the constructed neural network is very specific for the sonar data set, I did not have time to reconstruct the network so that it can be applied to a larger data set. For future works, I would like to expand the neural networks to handle any data set. This includes modifying "read_data" function to handle multiple file formats. Current implementation is only able to handle "csv" files. With larger data set, I would use CUDA to take advantage of GPU computing. One caveat with this parallel computing model is that I would have to rewrite the codes to implement all the children functions of "train_network" and their childrenren functions on the device. This is because copying data from the device to host and vice versa can create large overhead which could slow down the execution. I actually had started the implementation for this, however I had troubles debugging functions on the device. I had tried using CUDA-gdb to debug

the program but did not get very far with it mostly because I am still learning to use it. I would also like to explore the NVIDIA CUDA Deep Neural Network library (cuDNN) to see if it can be applied to speed up neural network training [12]. Furthermore, I would like to explore CUDA for Python (PyCUDA) [13] to see how to integrate GPU computing with Python, a language that is more flexible to develop machine learning algorithms, which can potentially help reduce the time and effort spent on writing machine learning codes.

References

[1] A Basic Introduction to Neural Networks. http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html.

[2] Tom Mitchell. Chapter 4: Artificial Neural Networks. In *Machine Learning*, pp. 81-127. McGraw-Hill. 1997.

[3] Sonar Dataset. https://github.com/renatopp/arff-datasets/blob/master/classification/sonar.arff.

[4] Yingu Liang. CS760 Lecture Slides. http://pages.cs.wisc.edu/~yliang/cs760/slides/lecture10-neural-networks-1.pptx.

[5] OpenMP. http://www.openmp.org/.

[6] About CUDA. https://developer.nvidia.com/about-cuda.

[7] Open MPI. https://www.open-mpi.org/.

[8] ME759: High Performance Computing for Applications in Engineering. 2017. http://sbel.wisc.edu/Courses/ME964/2017/.

[9] Donald Knuth. Structured Programming with go to Statements. *Computing Surveys*, 6:261-301, 1974.

[10] GNU gprof. https://sourceware.org/binutils/docs/gprof/.

[11] ME759 Project Souce Codes. https://github.com/UW-Madison-759/759--vitbao/tree/master/Project.

[12] NVIDIA cuDNN. GPU Accelerated Deep Learning. https://developer.nvidia.com/cudnn.

[13] GPU Accelerated Computing with Python. https://developer.nvidia.com/how-to-cuda-python.

.