PARALLELIZED TRAINING FOR STOCK PRICE PREDICTION NEURAL NETWORKS

ZACHARY WINOKER: 8353219928

1. Background and Problem Description

Machine learning techniques have become an integral part of stock price forecasting [1]. Neural networks (NN) rank among the best performing of these techniques, largely due to their ability to model the nonlinear character of the stock market [1]. As such, there is great reason to incorporate them into a trading system in order to enhance profitabilty. However, the process of designing, implementing, and training these networks can be very time-intensive. Furthermore it is common to try many different network and data designs before finding a sufficiently accurate one. Therefore it is imperative to be able to quickly prototype and train new neural networks.

In this project, we create and test a framework for training new neural networks in parallel on the AWS EC2 cloud [3]. The framework we design must be sufficiently fast such that a researcher can implement and train a variety of different network designs quickly. The trained networks must be sufficiently accurate so that networks can be used in an actual trading system. And finally, the framework must be scalable with respect to increasing the amount of training data.

2. Solution details

2.1. **The Dataset.** Our raw data consists of price and volume-traded data scraped from Google Finance [5]. This data was processed to yield training samples for our neural network. Each of these samples contains 10 input parameters and 1 output parameter. The 10 input parameters are: closing price, opening price, low price, high price, and volume-traded for the current and previous minute. The output paramter is the return for the given stock at the next minute, where return is defined as $return_i := (close_i - close_{i-1})/close_{i-1}$. IE, it is the fraction of the original investment in a stock that is recovered when the stock is sold. We chose this for our output data so that data from different stocks could be used to train the same network. We found that most minute-level returns are on the order of 10^{-2} or 10^{-3}. Note: we based our data design on the technical indicators used in [8].

This processing was applied to the minute-level data for about 400 stocks from the NASDAQ 100 and S&P 500. We used 5 days' worth of data for training, in particular the data from March 28, 2016 to April 1, 2016 was used. This gave us 1,048,574 training samples in total.

2.2. **Our Neural Network.** We use a single neural network topology for all trials in order to maximize consistency and comparability of our results. The network architecture is displayed in the figure below. The input layer has 10 nodes, each corresponding to one of the input data described in the previous section. There is one hidden layer with 10 nodes and 1 bias node. The output layer has 1 node and 1 bias node, the former corresponding to the return data discussed in the previous section. All nodes use a sigmoidal activation function. The network is fully connected in order to avoid preferential processing of any particular input datum. Note also that the fully connected version of this network will take longer to train than any version with fewer edges.

Therefore, for this vertex set and machine-sepecific implementation, our results represent an upper bound on the training time for this network.

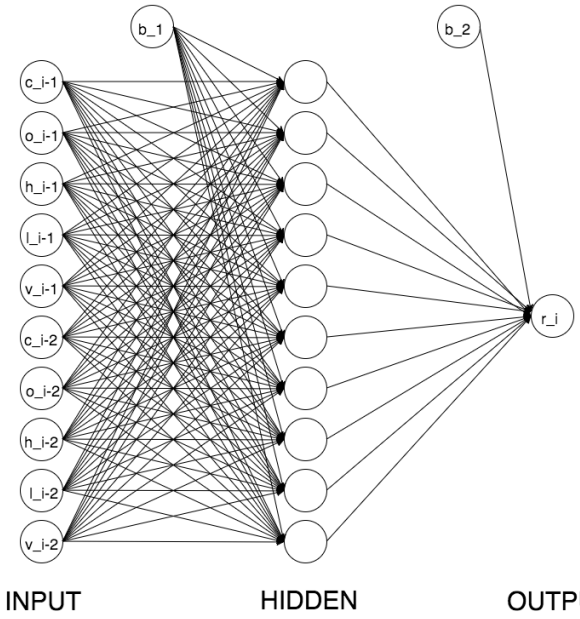

FIGURE 1. The neural network we repeatedly trained. It has a single hidden layer with 10 nodes and is fully connected.

2.3. **Pattern Parallel Training Algorithm.** We implemented a modified version of the Pattern Parallel Training (PPT) algorithm [2] to decrease the training time our network required. The original algorithm is designed with clusters in mind and makes no assumptions about the actual machines it is implemented on [2]. It is therefore an excellent candidate for our system since it can be used in a variety of different computing environments. We detail the algorithm below:

Algorithm 1: Our version of the PPT algorithm used to train a neural network in parallel.

$errorThreshold \leftarrow \epsilon$; (where $\epsilon > 0$)
$avgError \leftarrow \infty$; (initialize avgError to infinity)
$\{masterWeights\} \leftarrow$ set of random weights ; (initialize initial network with random weights)
while $avgError > errorThreshold$ **do**
  Broadcast $\{masterWeights\}$ to all worker nodes;
  All worker nodes **do**{
    Create NN with $\{masterWeights\}$ for weights;
    Train NN on a random subset of $numTrainingSamples/numNodes$ training samples;
    Send weights and local training error to master node;
  }
  Master node **do** {
    Compute avg weights and error;
    $\{masterWeights\} \leftarrow$ avg weights;
    $avgError \leftarrow computedavgerror$
  }
end
return $\{masterWeights\}$

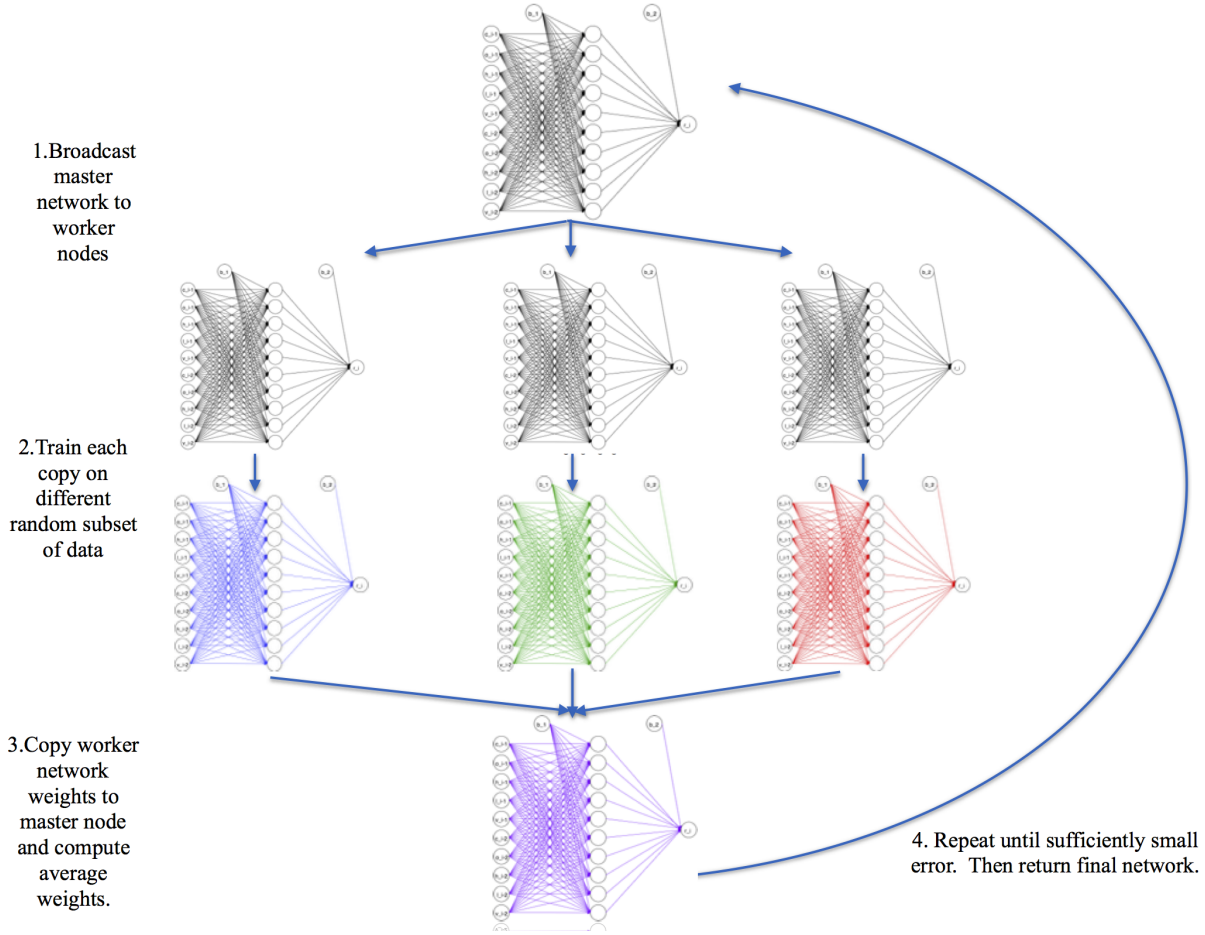

FIGURE 2. A diagrammatic representation of our version of the Pattern Parallel Training algorithm

2.4. **Machine Details.** We implemented our system on the AWS EC2 cloud for maximum scalability [3]. A cluster of 20 nodes was created, where each node was a t2.small instance. These instances have 1 3.3 GHz CPU each and 2 GB of memory [4]. Although AWS provides machines better suited to cluster computing, we were able to use 20 of these nodes for free, which was the major factor in our decision to use them. We used OpenMPI for inter-node communication.

2.5. **Libraries.** Our entire project was written in python and made use of a number of helpful python libraries. To launch and manage our AWS cluster, we used the python library StarCluster [6]. Our StarCluster configuration file is included with our code (see the Appendix section). We used the neural network library Pybrain, which included implementations of the network itself and backpropagation trainers [9]. Our OpenMPI library was mpi4py [7].

3. RESULTS

We trained the network using our PPT algorithm 10 times each on clusters with 1, 2, 4, 8, 10, 12, 16, and 20 nodes. We averaged the error and compute time across all 10 trials for each node

number. Plots of the average training times and training errors against number of nodes are given in figures 3 and 4.

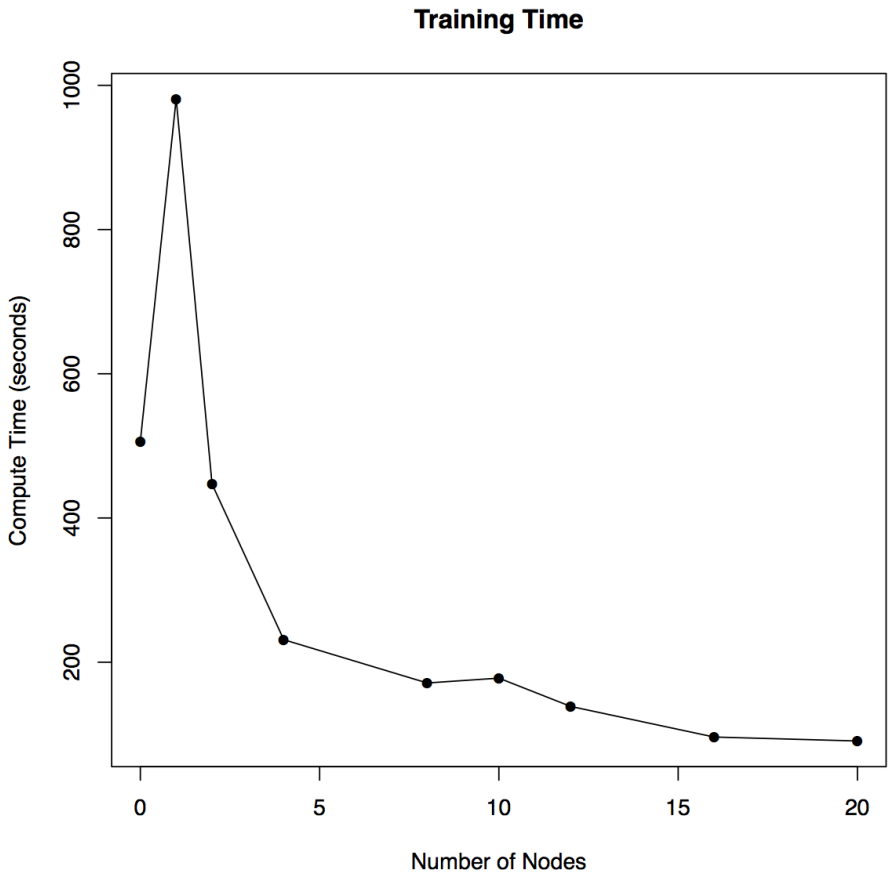

FIGURE 3. A plot illustrating the relationship between computation time for training the network and number of nodes

These results match what we should expect. There is a sharp decrease in training time as we increase the number of nodes. This is because training each worker network on a subset of the data allows us to take significant advantage of our parallelization. The training time also appears to level off as we approach 20 nodes. We should expect this leveling off to occur at some point since there are bottlenecks due to communication overhead inccured by MPI and a minimum training time for each network. Note that this leveling off will occur at different cluster sizes depending on

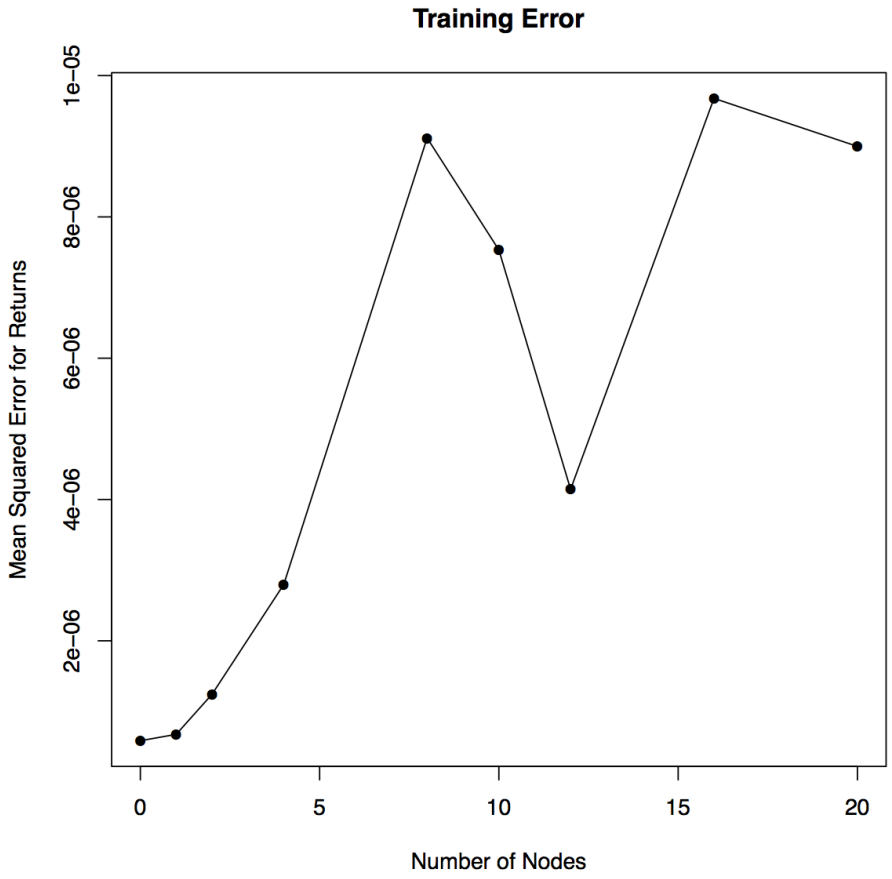

FIGURE 4. A plot illustrating the relationship between training error and number of nodes.

how much data is used, since this determines how much data each worker network is trained on. Note also that training a network on 20 nodes only takes about 90 seconds, which is appropriate for prototyping many networks of this size in quick succession. As such, this method is sufficiently fast for our purposes.

The error also scales as we should expect. There is an increase in average training error as we increase the node number, thereby decreasing the amount of data used to train each worker node. However, the error still is on the order of 10^{-6}, which is orders of magnitude lower than our average returns. The networks therefore are sufficiently accurate for our purpose.

Since we can increase the number of nodes without seriously compromising our error, our approach is scalable. As more data is added, we can just add more nodes and still keep the training time and error sufficiently low.

4. Conclusion and Future Work

We were able to create a sufficiently fast, accurate, and scalable parallelized scheme for training neural networks. The trends in compute time and error were reasonable and matched our expectations. In the future, we would like to explore other parallelization schemes such as multi-threading and GPU acceleration. We would also like to use more nodes and experiment with using an instance type that is optimized for cluster computing. Further efforts may also include attempts to optimize the network's hyper-parameters and increase the size of the dataset. Hopefully these effors can be used to generate a network that can be incorporated into a profitable trading system.

5. Appendix

5.1. **Installation.** Pybrain and StarCluster must be installed. Follow the installation instructions at [6] and [9] to do this.

5.2. **Build Instructions.** To build the StarCluster cluster, use the config file provided and follow instructions at [6] to launch the cluster. Note that this cluster is currently operational, so there is no need to re-launch it. Instead, just follow the instructions below to log into the cluster and run any code.

Note that if the cluster is relaunched, PyBrain must be installed on every node. You can log into each node using 'starcluster sshnode smallcluster node$x', where x is the node number (ex: 001 or 014).

5.3. **Running The Code.** After installing StarCluster and pointing it to the given config file, log into the cluster using 'starcluster sshmaster smallcluster'. On the cluster, all data is stored in /home/stocknn/data/training-data.

5.3.1. *Sequential Version.* The sequential code is at /root/network/seq. To run the sequential version, navigate to this directory and run 'python seq.py' The error and training time will be printed to the console.

5.3.2. *Parallel Version.* To run the parallel code, navigate to /home/stocknn/network/ppt. From here, we can submit multinode jobs to the queue. To do this, run 'qsub -pe orte $n run-ppt.sh ', where $n is the number of nodes you wish you use. The training time will be written to ./output-$n and the error will be written to ./training-errors-$n.

6. Citations

(1) P. D. Yoo, M. H. Kim and T. Jan, "Machine Learning Techniques and Use of Event Information for Stock Market Prediction: A Survey and Evaluation," Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on, Vienna, 2005, pp. 835-841. 10.1109/CIMCA.2005.1631572. `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1631572&isnumber=34212`

(2) Dahl, G., McAvinney, A., & Newhall, T. (2008, August 18). Parallelizing Neural Network Training for Cluster Systems. Retrieved from https://www.cs.swarthmore.edu/~newhall/papers/pdcn08.pdf

(3) https://aws.amazon.com/

(4) AWS instance information https://aws.amazon.com/ec2/instance-types/

(5) https://www.google.com/finance

(6) StarCluster https://github.com/jtriley/StarCluster

(7) mpi4py https://mpi4py.scipy.org/docs/mpi4py.pdf

(8) http://eprints.covenantuniversity.edu.ng/4112/1/Emerging_Trend.pdf

(9) PyBrain http://pybrain.org/docs/