**Experimental Design for neural network**

Linglin Huang, Chang Liu, Greyson Liu, Kamrine Poels

**Problem Statement**

Train a neural network in parallel with the appriopriate tools (e.g. OpenMP and MPI) that uses the minute-level price and trade size information of selected S&P 500 index constituents to predict next-minute return for a constituent. We benchmark our approach against an analogous sequential version using various performance metrics on accuracy and computational efficiency.

**Input Variables**

Price and volume information at or before minute *t* for all stocks except stock *j*

Technical indicators of price series:

1. Intra-interval proportions (**IIP**)
2. Exponetial Moving Averages (**EMA**)
3. Price Trend indicators **(AD, Adv, ADR**)
4. Others (see Appendix)

Normalize all the input and output variables

1. For stock price, use returns = percentage change

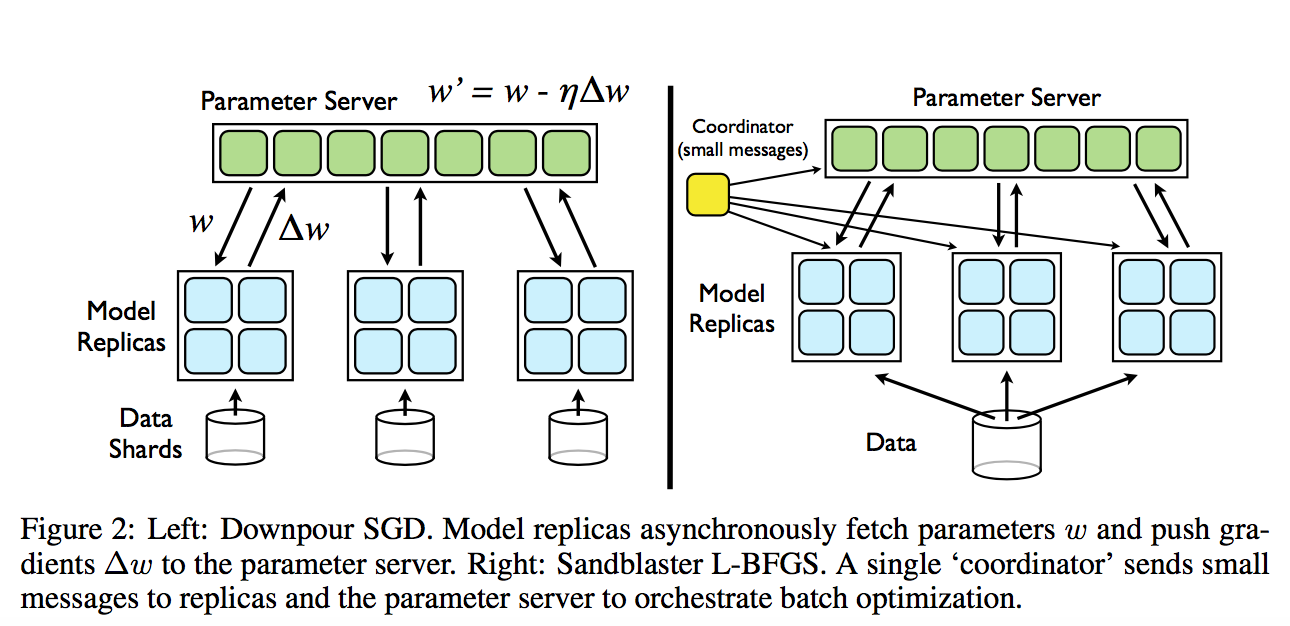
2. For other values, use min-max scaling: RN = (R-R\_min) / (R\_max – R\_min) where R is the value of an input

**Output variables**

Predicted return at minute t+1 for stock j

**Parallelism Architecture: Downpour SGD & PSO**

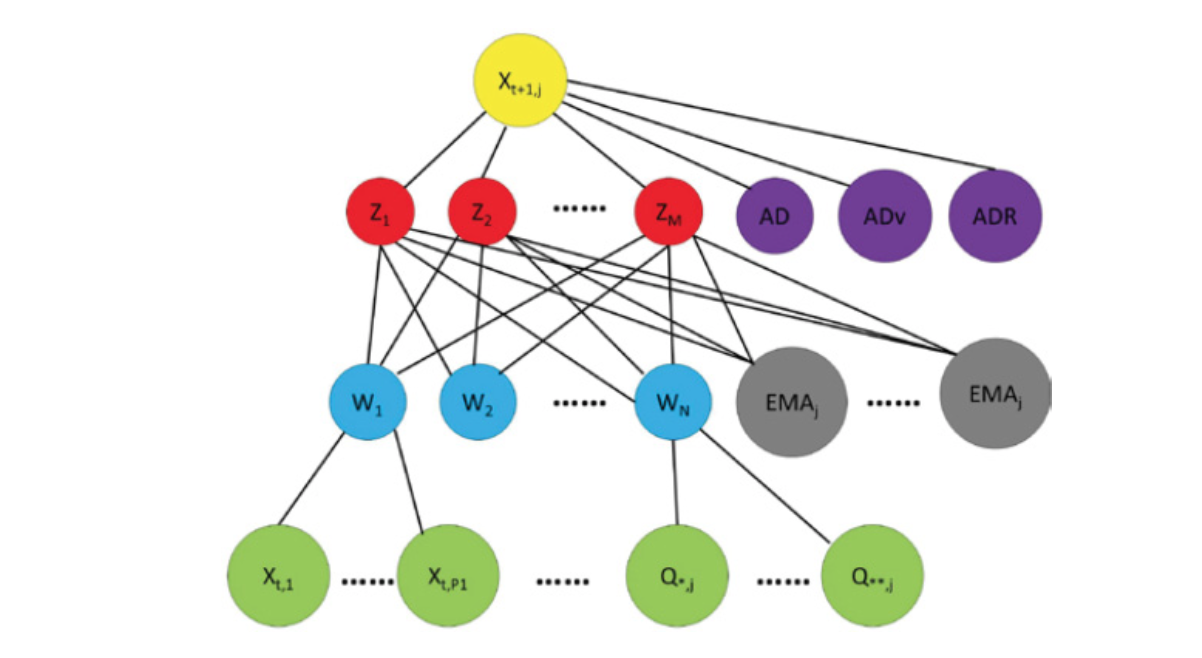
Downpour (mini-batch) SGD (with Adam/AdaGrad) (see Figure 1)

****

**Figure 1: Downpour SGD [1] see caption in figure.** We will execute both data parallelism and model parallelism in the following way: a machine (e.g. an Odyssey node) will store a Data Shard (a subset of data) and train a model replica (fetching w and pushing grad(w) to the parameter server), ***both independently and asynchronously***. That is, parameter server updates the parameter set whenever it receives grad(w) from a model replica. In addition, training a model replica using SGD will be parallelized using the cores available on the machine, with simple SGD as benchmarks.

**Neural Network Architecture (hyperparameters)**

1. fully connected vs locally connected hierarchical network (see **Figure 2** – we will modify such network to fit our purposes)
2. L = 1 ~10 hidden layers
3. # neurons = 8~64/layer; less for deeper layers
4. Optimizer learning rate, other parameters such as momentum
5. ReLu/Sigmoid activation, linear activation for output node
6. # Model REPLICA that we distribute each Odyssey node to

****

**Figure 2: Locally connected hierachical network** [2] Top layer is output layer; bottom layer is input layer. Input layer: X’s are lagged sector-specific stock information from the S&P 500 constituents; Q’s are IIP’s; 1st hidden layer: hidden nodes W’s are locally connected to a subset of X’s by sector (health care sectors are connected to one W node), and to IIP’s; 2nd hidden layer: the hidden nodes Z’s are fully connected to the EMA’s and W’s nodes; along with the AD’s indicators, they are fully connected to output node X.

**Dynamic Backtesting procedure**

#Input: define data[0 : T-1], training\_size, validation\_size, test\_size, window\_size, ensemble\_size

#Output: predicted values from t = T-training\_size-validation\_size : T-1

# Python pseudo-code as follows

for t in range(training\_size + validation\_size, T):

if t % test\_size != 1:

#each model in the existing committee predicts on time t;

#the mean of the ensemble predictions is the final prediction for time t

else:

training\_data = data[t - training\_size- validation\_size : t - validation\_size]

validation\_data = data[t- validation\_size: t]

for i in range(N\_window):

# train the ith model based on a random starting point and a (bootstrapped?) sample of window\_size from training\_data;

# (optional) vary the architecture\* (# layer, neurons)

# calculate validation accuracy

# choose ensemble\_size number of models with highest accuracies to form the new committee;

# each model in the new committee predicts on time t

# the mean of all predictions is the final prediction for time t

**\* Model-validation methods**

Costly hyperparameter search for the optimal network hyperparameters (#layers, nodes, etc)

1. Researcher’s guess (simple)

2. Grid Search (costly, inefficient)

3. Particle Swarm Optimization (also embarassingly parallel and available in Python’s package Optunity/pyswarm)

**Loss function**

Mean Square Error

**Initialization**

1. For simple SGD, initialize with standard normal scaled with input units
2. For Downpour SGD: use warm start - initialize with warmstart of simple SGD

**Regularization**

1. L2-penality; maxnorm of weights; dropouts
2. Monitor training process – early stopping if validation not improving

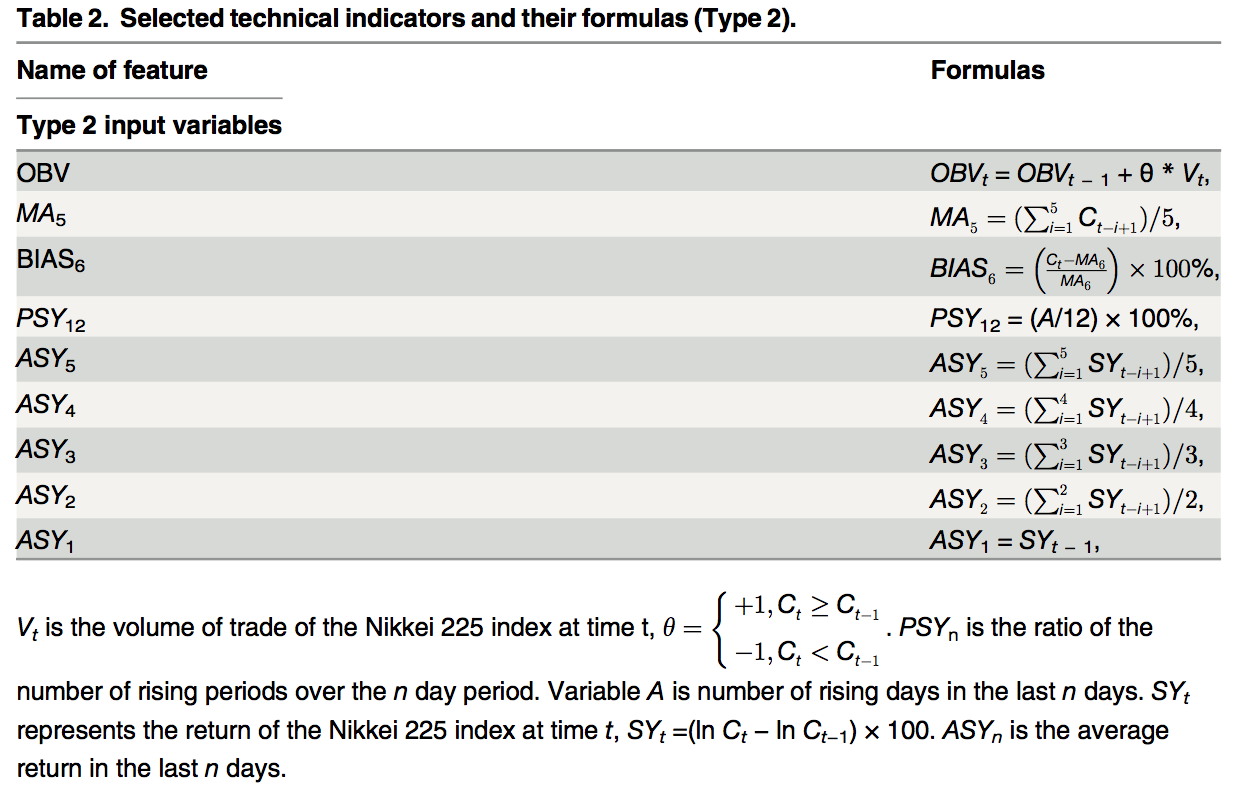
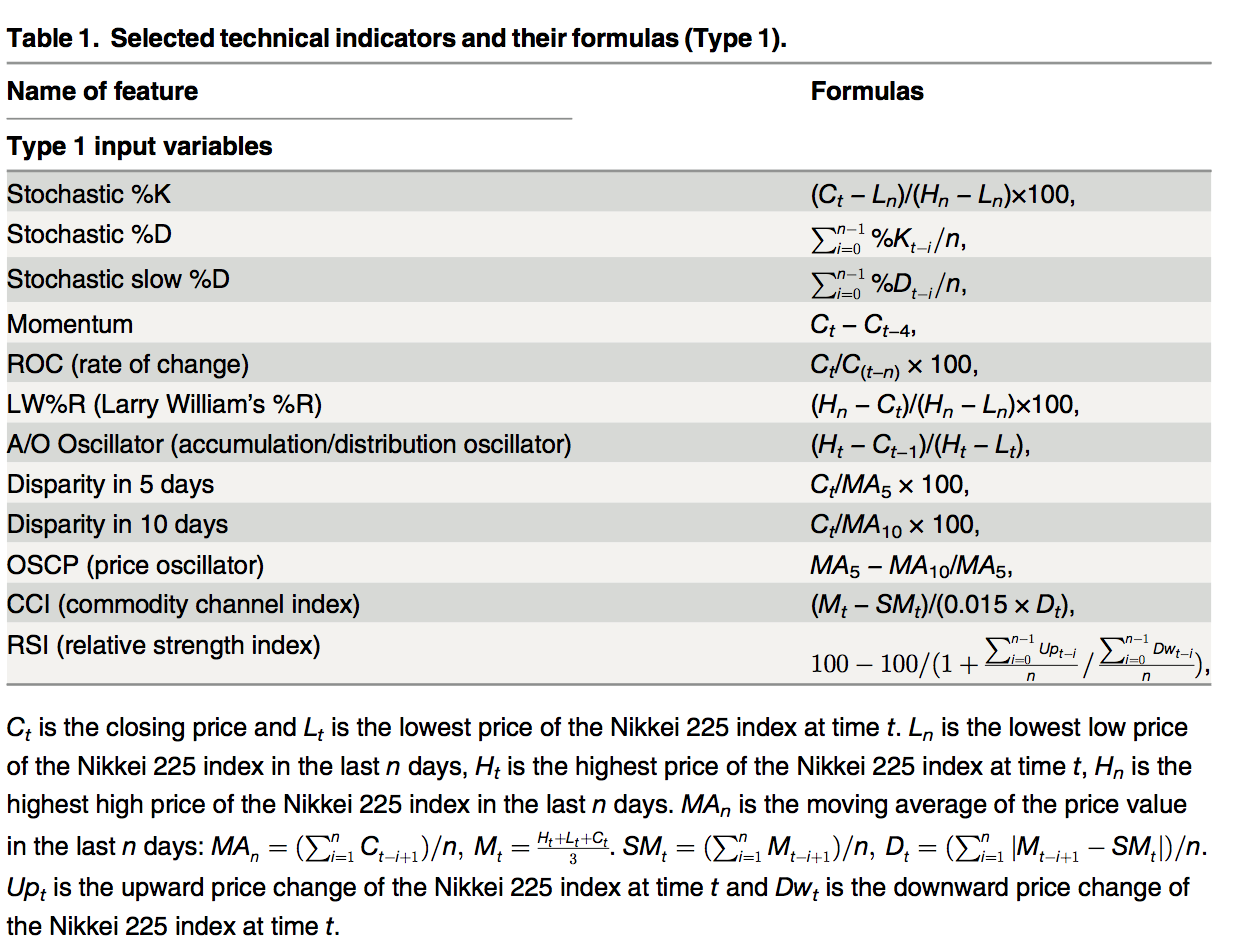
**Evaluation metrics**

1. Direction Accuracy (consider thresholded on predicted values such that only large predicted values count) per model
2. Speedups (in Gflops/wall time) vs cores used per model replica for different model size (# parameters)[[1]](#footnote-1)
3. Accuracy as a function of time across optimizers to test convergence of Downpour SGD
4. Hit ratio = mean(p\_i) where p\_i = 1{(y-By)(yhat-By\_hat)>0} and y are predicted prices
5. Time taken to reach a accuracy threshold vs cores
6. Efficiency vs cores

**Reference**

[1] Dean, J., et al. (2012). Large scale distributed deep networks. Proceedings of the 25th International Conference on Neural Information Processing Systems. Lake Tahoe, Nevada, Curran Associates Inc.: 1223-1231.

[2] Hao Chen, Keli Xiao, Jinwen Sun, and Song Wu. 2017. A double-layer neural network framework for high- frequency forecasting. ACM Trans. Manage. Inf. Syst. 7, 4, Article 11 (January 2017), 17 pages. DOI: http://dx.doi.org/10.1145/3021380



1. See Reference [1]. The paper measures “the mean time to process a single mini-batch for simple SGD training as a function of the number of partitions (machines) used in a single model instance”. [↑](#footnote-ref-1)