Implementing Deep Neural Networks for Financial Market Prediction on the Intel Xeon Phi

Matthew Dixon
Stuart School of Business
Illinois Institute of Technology
10 West 35th Street
Chicago, IL 60616
matthew.dixon@stuart.iit.edu

Diego Klabjan
Department of Industrial Engineering and Management Sciences
Northwestern University
Evanston, IL
d-klabjan@northwestern.edu

Jin Hoon Bang
Department of Computer Science
Northwestern University
Evanston, IL
jinhoonbang@u.northwestern.edu

ABSTRACT

Deep neural networks (DNNs) are powerful types of artificial neural networks (ANNs) that use several hidden layers. They have recently gained considerable attention in the speech transcription and image recognition community (Krizhevsky et al., 2012) for their superior predictive properties including robustness to overfitting. However their application to financial market prediction has not been previously researched, partly because of their computational complexity. This paper describes the application of DNNs to predicting financial market movement directions. A critical step in the viability of the approach in practice is the ability to effectively deploy the algorithm on general purpose high performance computing infrastructure. Using an Intel Xeon Phi co-processor with 61 cores, we describe the process for efficient implementation of the batched stochastic gradient descent algorithm and demonstrate a 11.4x speedup on the Intel Xeon Phi over a serial implementation on the Intel Xeon.

Categories and Subject Descriptors

G.4 [**MATHEMATICAL SOFTWARE**]: Parallel and vector implementations

General Terms

Algorithms, Performance

Keywords

Deep Neural Networks, Market Prediction, Intel Xeon Phi

1. INTRODUCTION

Many of the challenges facing methods of financial econometrics include non-stationarity, non-linearity or noisiness of the time series. While the application of artificial neural networks (ANNs) to time series methods are well documented (Faraway and Chatfield, 1998; Refenes, 1994; Trippi and DeSieno, 1992; Kaastra and Boyd, 1995) their proneness to over-fitting, convergence problems, and difficulty of implementation raised concerns. Moreover, their departure from the foundations of financial econometrics alienated the financial econometrics research community and finance practitioners.

However, algotrading firms employ computer scientists and mathematicians who are able to perceive ANNs as not just black-boxes, but rather a non-parametric approach to modeling based on minimizing an entropy function. As such, there has been a recent resurgence in the method, in part facilitated by advances in modern computer architecture (Chen et al., 2013; Niaki and Hoseinzade, 2013; Vanstone and Hahn, 2010).

A deep neural network (DNN) is an artificial neural network with multiple hidden layers of units between the input and output layers. They have been popularized in the artificial intelligence community for their successful use in image classification (Krizhevsky et al., 2012) and speech recognition. The field is referred to as "Deep Learning".

In this paper, we shall use DNNs to partially address some of the deficiencies of ANNs. Specifically, we model complex non-linear relationships between the independent variables and dependent variable and reduced tendency to overfit. In order to do this we shall exploit advances in low cost many-core accelerator platform to train and tune the parameters of our model.

For financial forecasting, especially in multivariate forecasting analysis, the feed-forward topology has gained much more attention and shall be the approach used here. Back-propagation and gradient descent have been the preferred method for training these structures due to the ease of implementation and their tendency to converge to better local optima in comparison with other trained models. However, these methods can be computationally expensive, especially when used to train DNNs.

There are many training parameters to be considered with a DNN, such as the size (number of layers and number of units per layer), the learning rate and initial weights. Sweeping through the parameter space for optimal parameters is not feasible due to the cost in time and computational resources. We use mini-batching (computing the gradient on several training examples at once rather than individual ex-

amples) as one common approach to speeding up computation. We go further by expressing the back-propagation algorithm in a form that is amenable to fast performance on an Intel Xeon Phi co-processor (Jeffers and Reinders, 2013). General purpose hardware optimized implementations of the back propagation algorithm are described in the literature (see for example Shekhar and Amin (1994); Oh and Jung (2004)), however our approach is tailored for the Intel Xeon Phi co-processor.

The main contribution of this paper is to describe an implementation of deep neural networks to financial time series data in order to classify financial market movement directions. Traditionally, researchers iteratively experiment with a handful of signals to train a level based method, such as vector autoregression, for each instrument (see for example Kaastra and Boyd (1995); Refenes (1994); Trippi and DeSieno (1992)). More recently, however, Leung et al. (2000) provide evidence that classification based methods outperform level based methods in the prediction of the direction of stock movement and trading returns maximization.

Using 5 minute interval prices from June 1989 to March 2013, our approach departs from the literature by using state-of-the-art parallel computing architecture to simultaneously train a single model from a large number of signals across multiple instruments, rather than using one model for each instrument. By aggregating the data across multiple instruments and signals, we enable the model to capture a richer set of information describing the co-movements across signals for each instrument price movement. Our results show that our model is able to predict the direction of instrument movement to, on average, 73% accuracy. We further show how backtesting accuracy translates into the P&L for a simple long-only trading strategy.

In the following section we introduce the back-propagation learning algorithm and use mini-batching to express the most computationally intensive equations in matrix form. Once expressed in matrix form, hardware optimized numerical linear algebra routines are used to achieve an efficient mapping of the algorithm on to the Intel Xeon Phi co-processor. Section 3 describes the preparation of the data used to train the DNN. Section 4 describes the implementation of the DNN. Section 4.1 then describes a parallel version of the implementation that significantly reduces the overall training time.

2. DEEP NEURAL NETWORKS

We begin with mathematical preliminaries. Let $\mathcal{D}$ denote the historical dataset of M features and N observations. We draw a training subset $\mathcal{D}_{\text{train}} \subset \mathcal{D}$ of N_{train} observations and a test subset of $\mathcal{D}_{\text{test}} \subset \mathcal{D}$ of N_{test} observations.

Denote the n^{th} observation (feature vector) as $x_n \in \mathcal{D}_{\text{train}}$. In an ANN, each element of the vector becomes a node in the input layer, as illustrated in the figure below for the case when there are 7 input variables (features) per observation. In a fully connected feed-forward network, each node is connected to every node in the next layer. Although not shown in the figure, associated with each edge between the i^{th} node in the previous layer and the j^{th} node in the current layer l is a weight $w_{ij}^{(l)}$.

In order to find optimal weightings $\mathbf{w} := \{\mathbf{w}^{(l)}\}_{l:=1\rightarrow L}$ between nodes in a fully connected feed forward network with L layers, we seek to minimize a cross-entropy function

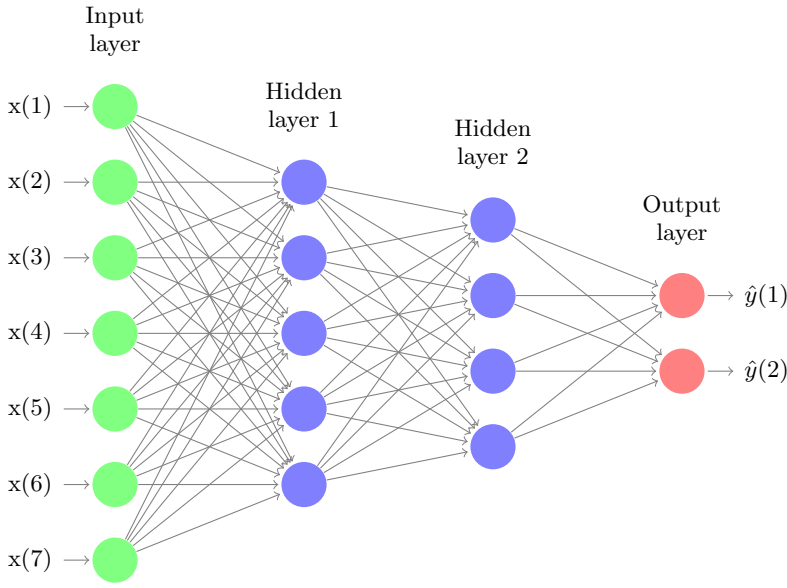

Figure 1: An illustrative example of a feed-forward neural network with two hidden layers, seven features and two output states. Deep learning networks typically have many more layers, use a large number of features and several output states or classes. The goal of learning is to find the weight on every edge that minimizes a loss function.

of the form

$$E(\mathbf{w}) = -\sum_{n=1}^{N_{\text{test}}} e_n(\mathbf{w}), \qquad e_n(\mathbf{w}) := \sum_{k=1}^{K} y_{kn} ln\left(\hat{y}_{kn}\right). \quad (1)$$

For clarity of exposition, we drop the subscript n. Here K denotes the total number of classes. The binary target vectors y and binary output vectors $\hat{y}$ have a 1-of-n_s encoding for each symbol, where n_s is the number of classes per symbol, so that each state associated with a symbol can be interpreted as a probabilistic weighting. Put formally, $y_k \in \{0,1\}, \forall k \in K$ and $\sum_{k\in\mathcal{K}_i} y_k = 1, \forall i$ where $\mathcal{K}_i$ is the set of n_s class indices associated with symbol i.

To ensure analytic gradient functions under the cross-entropy error measure, each of the nodes associated with the i^{th} symbol in the output layer are activated with a softmax function of the form

$$\hat{y}_k := \phi_{\text{softmax}}(\mathbf{s}^{(L)}) = \frac{\exp(s_k^{(L)})}{\sum_{j\in\mathcal{K}_i} exp(s_j^{(L)})}, \forall k \in \mathcal{K}_i. \quad (2)$$

The gradient of the likelihood function w.r.t. s then takes the simple form:

$$\frac{\partial e(\mathbf{w})}{\partial s_k^{(L)}} = \hat{y}_k - y_k \quad (3)$$

and in a fully connected feed-forward network $s_k^{(l)}$ is the weighted sum of outputs from the previous layer $l-1$ that connect to node j in layer l:

$$s_j^{(l)} = \sum_{i}^{n_{l-1}} w_{ij}^{(l)} x_i^{(l-1)} + \text{bias}_j^{(l)}. \quad (4)$$

Here n_l is the number of nodes in layer l. For each node i in the $(l-1)^{th}$ layer, the recursion relation for the back propagation using conjugate gradients is:

$$\delta_i^{(l-1)} = \sum_{j=1}^{n_{(l)}} \delta_j^{(l)} w_{ij}^{(l)} \sigma(s_i^{(l-1)})(1 - \sigma(s_i^{(l-1)})), \quad (5)$$

where we have used the analytic form of the derivative of the sigmoid function

$$\sigma'(v) = \sigma(v)(1 - \sigma(v)), \quad (6)$$

which is used to activate all hidden layer nodes.

A trained feed-forward network can be used to predict the outputs states of all symbols, given any observation as an input, by recursively applying Equation 4. The description of how the network is trained now follows.

Stochastic Gradient Descent.

Following Rojas (1996), we now revisit the backpropagation learning algorithm based on the method of stochastic gradient descent (SGD). After random sampling of an observation i, the SGD algorithm updates the parameter vector $\mathbf{w}^{(l)}$ for the l^{th} layer using

$$\mathbf{w}^{(l)} = \mathbf{w}^{(l)} - \gamma \nabla E_i(\mathbf{w}^{(l)}), \quad (7)$$

where γ is the learning rate. The gradient expression with respect to the bias terms has a similar expression.

A high level description of the sequential version of the SGD algorithm is given in Algorithm 1. Note that for reasons of keeping the description simple, we have avoided some subtleties of the implementation.

Algorithm 1 Stochastic Gradient Descent

```
w ← r, r_i ∈ N(μ, σ), ∀i
E ← 0
for i = 0 to n − 1 do
    E ← E + E_i(w)
end for
while E ≥ τ do
    for t = 0 to n − 1 do
        i ← sample with replacement in [0, n − 1]
        w ← w − γ∇E_i(w)
    end for
    E ← 0
    for i = 0 to n − 1 do
        E ← E + E_i(w)
    end for
end while
```

2.1 Mini-batching

It is well known that mini-batching improves the computational performance of the feedforward and backpropagation computations. We process b observations in one mini-batch. This results in a change to the SGD algorithm and the dimensions of data-structures that are used to store variables. In particular, δ, x, s and E now have a batch dimension. Note however that the dimensions of $w^{(l)}$ remain the same. The above equations can be now be modified.

With slight abuse of notation, we redefine the dimensions of $\delta^{(l)}, X^{(l)}, S^{(l)} \in \mathbb{R}^{n_l \times b}$, $\forall l$, $E \in \mathbb{R}^{n_L \times b}$, where n_l is the number of neurons in layer l and b is the size of the mini-batch.

The computation of the sum in the feed-forward network can be expressed as a matrix-matrix product at each layer

$$S^{(l)} = \left(X_i^{(l-1)}\right)^T \mathbf{w}^{(l)}. \quad (8)$$

For the i^{th} neuron in output layer L and the j^{th} observation in the mini-batch

$$\delta_{ij}^{(L)} = \sigma_{ij}^{(L)}(1 - \sigma_{ij}^{(L)})E_{ij}. \quad (9)$$

For all intermediate layers $l < L$, the recursion relation for δ is

$$\delta_{ij}^{(l-1)} = \sigma_{ij}^{(l)}(1 - \sigma_{ij}^{(l)}) w_{ij}^{(l)} \delta_{ij}^{(l)}. \quad (10)$$

The weights are updated with matrix-matrix products for each layer

$$\Delta \mathbf{w}^{(l)} = \gamma X^{(l-1)} \left(\delta^{(l)}\right)^T. \quad (11)$$

3. THE DATA

Our historical dataset contains 5 minute mid-prices for 45 CME listed commodity and FX futures from March 31st 1991 to September 30th, 2014. We use the most recent fifteen years of data because the previous period is less liquid for some of the symbols, resulting in long sections of 5 minute candles with no price movement. Each feature is normalized by subtracting the mean and dividing by the standard deviation. The training set consists of 37,500 consecutive observations and the test set consists of the next 12,500 observations. These sets are rolled from the start of the liquid observation period in one month increments until the final 50,000 observations from March 31st, 2005 until the end of the dataset.

The overall training dataset consists of the aggregate of feature training sets for each of the symbols. The training set of each symbol consists of price differences and engineered features including lagged prices differences from 1 to 100, moving price averages with window sizes from 5 to 100, and correlations between the returns and the returns of all other symbols. The overall training set contains 9895 features. The motivation for including these features in the model is to capture memory in the historical data and co-movements between symbols.

4. IMPLEMENTATION

The architecture of our network contains five learned fully connected layers. The first of the four hidden layers contains 1000 neurons and each subsequent layer is tapered by 100. The final layer contains 135 output neurons - three values per symbol of each of the 45 futures contracts. The result of including a large number of features and multiple hidden layers is that there are 12,174,500 weights in total.

The weights are initialized with an Intel MKL VSL random number generator implementation that uses the Mersenne Twistor (MT19937) routine. Gaussian random numbers are generated by transforming the uniform random numbers with an inverse Gaussian cumulative distribution function with zero mean and standard deviation of 0.01. We initialized the neuron biases in the hidden layers with the constant 1.

We used the same learning rate for all layers. The learning rate was adjusted according to a heuristic which is described in Algorithm 2 below and is similar to the approach taken

in Krizhevsky et al. (2012) except that we use cross entropy rather than the validation error. We sweep the parameter space of the learning rate from $[0.1, 1]$ with increments of 0.1. We further divide the learning rate by 2 if the cross-entropy does not decrease between epochs.

Algorithm 2 Deep Learning Methodology

for $\gamma := 0.1, 0.2, \ldots, 1$ **do**
  <Initialize all weights>
  $w_{i,j}^{(l)} \leftarrow r,\ r \in \mathcal{N}(\mu, \sigma),\ \forall i, j, l$
  <Iterate over epochs>
  for $e = 1, \ldots, N_e$ **do**
    Generate $\mathcal{D}_e$
    <Iterate over mini-batches>
    for $m = 1, \ldots, M$ **do**
      Generate $\mathcal{D}_m$
      <Feed-Forward network construction>
      for $l = 2, \ldots, L$ **do**
        Compute all $x_j^{(l)}$
      end for
      for $l = L, \ldots, 2$ **do**
        <Backpropagation>
        Compute all $\delta_j^{(l)} := \nabla_{s_j^{(l)}} E$
        <Update the weights>
        $\mathbf{w}^{(l)} \leftarrow \mathbf{w}^{(l)} - \gamma X^{(l-1)} \left(\delta^{(l)}\right)^T$
      end for
    end for
  end for
  If crossentropy(e) $\leq$ crossentropy(e-1) then $\gamma \leftarrow \gamma/2$
end for
Return final weights $w_{i,j}^{(l)}$

In Algorithm 2, the subset of the training set used for each epoch is defined as

$$\mathcal{D}_e := \{x_{n_k} \in \mathcal{D}_{\text{train}} \mid n_k \in \mathcal{U}(1, N_{\text{train}}), k := 1, \ldots, N_{\text{epoch}}\} \tag{12}$$

and the mini-batch with in each epoch set is defined as

$$\mathcal{D}_m := \{x_{n_k} \in \mathcal{D}_{ep} \mid n_k \in \mathcal{U}(1, N_{\text{epoch}}), k := 1, \ldots, N_{\text{mini-batch}}\}. \tag{13}$$

4.1 Parallel Implementation

In designing an algorithm for parallel efficiency on a shared memory architecture, three design goals have been implemented:

- The algorithm has to be designed with good data locality properties. In other words, each processor is assigned access to a single, contiguous and separate region of memory for each global data structure. This is important for performance and to avoid race conditions.
- The dimension of the matrix or for loop being parallelized is at least equal to the number of threads. This is important to avoid redundancy of the vectorization units.
- BLAS routines from the MKL should be used in preference to `openmp` parallel for loop primitives. This is because the MKL BLAS routines are heavily optimized for each architectures.

	CPU System	Co-processor System
Processor	Xeon E5-2690 v2 - 16 cores - 20 threads (HT on) - 2.30GHz	Xeon Xeon Phi 7120 - 61 cores - 244 threads (HT on) - 1.24GHz
ECC	on	on
RAM	128GB	16GB
OS	GNU 2.6.32	GNU 2.6.38
XE Composer	2015	
Compiler	ICC 15.0.2	
Flags	-O3	-O3 -mmic

Table 1: Benchmark system configurations.

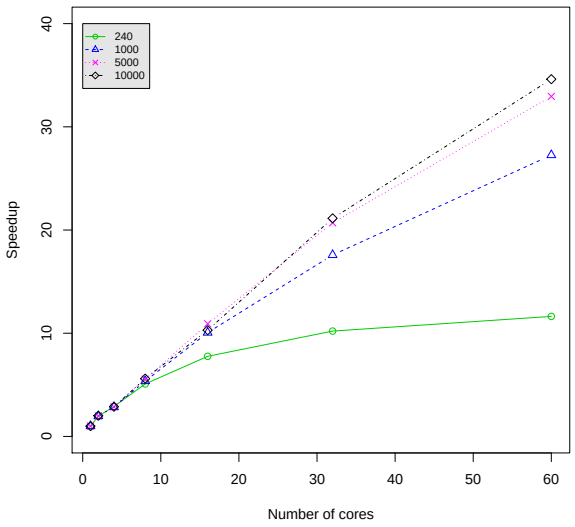

Figure 2: Speedup of the batched back-propagation algorithm on the Intel Xeon Phi as the number of cores is increased.

The approach that has currently be implemented uses a combination of `openmp` and `MKL` BLAS routines for parallelizing construction of the feedforward network and back-propagation steps. In each case, parallelization is performed over the mini-batch dimension, so it is important that b is sufficiently large to exploit the parallelism on the Intel Xeon Phi. Matrix-matrix multiplication is achieved using `dgemm`.

5. EXPERIMENTAL RESULTS

All performance results reported in this section are obtained using an Intel Xeon and the Intel Xeon Phi. The details of the configuration are given in Table 1.

Figure 2 shows the speed up of the back propagation algorithm on the Intel Xeon Phi against the number of cores for various batch sizes. The results are measured using the `KMP_AFFINITY= scattered` setting and the batch size is varied between 240, 1000, 5000 & 10000. We observe that the back-propagation algorithm shows a sub-linear parallel scalability which improves with batch size.

Figure 3 shows the effect of increasing the batch size on the relative performance of the Intel Xeon Phi against the baseline system. Increasing work load results in better parallel efficiency and and hence an increase in relative performance.

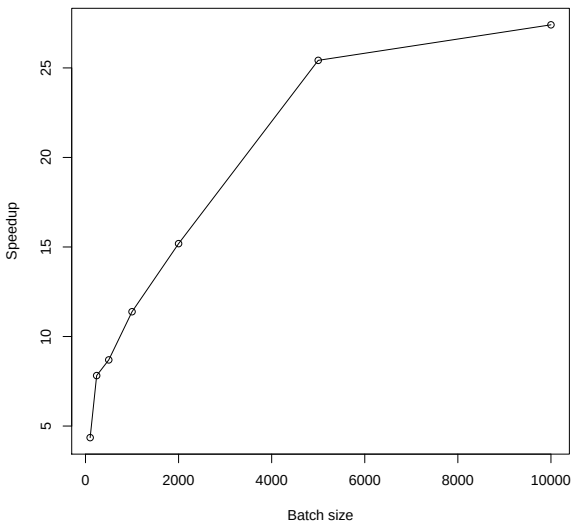

Figure 3: Speedup of the batched back-propagation algorithm on the Intel Xeon Phi relative to the baseline for various batch sizes.

Using larger batch sizes is approached with caution as this can reduce the overall convergence properties of the learning algorithm. For this reason we recommend using a batch size of 1000 so as to achieve a 11.4× speedup against the base line but avoid disrupting the convergence properties of the learning algorithm.

The results in Table 2 below show the performance attribution of the Intel Xeon Phi 7120 against the baseline system for various batch sizes b. The Intel Xeon Phi is configured to use 240 threads. We observe that the serial time is dominated by the feed forward network construction and the back-propagation of δ. On the Intel Xeon Phi, the back-propagation is efficiently parallelized leaving the network construction as the bottle-neck.

When using 60 cores on the Phi, the speedup of each mini-batch iteration of size $b = 1000$ using 240 threads compared to a serial implementation on the Intel Xeon is 11.4×, reducing each mini-batch iteration time to under 0.5 seconds. This is an important development since typically, to avoid overfitting, a minimum of 100 mini-batch iterations are run per epoch and up to 50 epochs are run for each value of the learning rate. The overall time is therefore approximately 10 hours when factoring in time for calculation of error measures on the test set[1] and thus the training can be run as an overnight batch job.

Figure 4 (top) shows the GFLOPS of the weight matrix update (using `dgemm`) for each layer. The batch size is set here to 1000. Recall that the network tapers inwards and thus the size of the matrices reduce as each layer is increased. We observe the first layer most efficiently utilizes the hardware resources although it is far below the theoretical peak of 1.208 TFLOPS. Referring to the weights column in Table 2, we see that this step does not scale well and this can be attributed to the underutilization of the hardware resources in the smaller hidden layers which can not be avoided due to data dependency. Figure 4 (bottom) shows, for completeness, how the GFLOPS of the first layer weight matrix update (`dgemm`) vary with the batch size of the back propagation algorithm.

[1]Overall, a classification accuracy of 73% and a F-measure of 0.41 was found

b	platform	ffwd	delta	weights	total
1000	baseline	2.03	2.22	1.34	5.59
	Xeon Phi	0.295	0.105	0.0915	0.491
	speedup	6.88 x	21.1 x	14.6x	11.4x
5000	baseline	8.45	21.0	6.71	36.1
	Xeon Phi	0.626	0.383	0.412	1.42
	speedup	13.9 x	54.8 x	16.3x	25.4x
10000	baseline	17.9	42.2	13.9	74.0
	Xeon Phi	1.15	0.704	0.846	2.70
	speedup	15.6 x	59.9 x	16.4x	27.4x

Table 2: This table shows the elapsed wall-clock time in seconds of one mini-batch iteration of the learning together with the decomposition. The decomposition is shown for various batch sizes b. *ffwd* denotes the feed forward network construction, *delta* denotes back propagation of δ and *weights* denotes the cost of updating the weight matrices using the dgemm routine. The serial version is benchmarked on an Intel Xeon E5-2695 and the parallel version is benchmarked on an Intel Xeon Phi co-processor 7120.

6. CONCLUSION

This paper describes the application of DNNs to predicting financial market movement directions. A critical step in the viability of the approach in practice is the ability to effectively deploy the algorithm on general purpose high performance computing infrastructure. Using an Intel Xeon Phi co-processor with 61 cores, we describe the process for efficient implementation of the batched stochastic gradient descent algorithm and demonstrate a 11.4x speedup on the Intel Xeon Phi over a serial implementation on the Intel Xeon.

References

J. Chen, J. F. Diaz, and Y. F. Huang. High technology ETF forecasting: Application of Grey Relational Analysis and Artificial Neural Networks. *Frontiers in Finance and Economics*, 10(2):129–155, 2013.

J. Faraway and C. Chatfield. Time series forecasting with neural networks: a comparative study using the air line data. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 47(2):231–250, 1998.

J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. ISBN 9780124104143, 9780124104945.

I. Kaastra and M. S. Boyd. Forecasting futures trading volume using neural networks. *Journal of Futures Markets*, 15(8):953–970, 1995. ISSN 1096-9934.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In

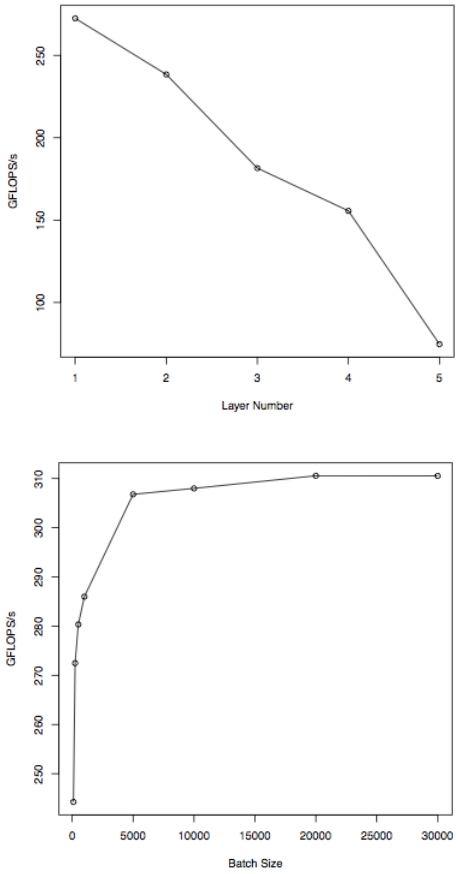

Figure 4: (top) The GFLOPS of the weight matrix update (using dgemm) for each layer. (bottom) Variation of the GFLOPS of the first layer weight matrix update with the batch size.

Advances in neural information processing systems, pages 1097–1105, 2012.

M. Leung, H. Daouk, and A.Chen. Forecasting stock indices: a comparison of classification and level estimation models. *International Journal of Forecasting*, 16(2):173–190, 2000.

S. Niaki and S. Hoseinzade. Forecasting sp 500 index using artificial neural networks and design of experiments. *Journal of Industrial Engineering International*, 9(1):1, 2013. ISSN 1735-5702.

K.-S. Oh and K. Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311 – 1314, 2004. ISSN 0031-3203.

A.-P. Refenes. *Neural Networks in the Capital Markets*. John Wiley & Sons, Inc., New York, NY, USA, 1994. ISBN 0471943649.

R. Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1996. ISBN 3-540-60505-3.

V. K. S. Shekhar and M. B. Amin. A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5:1073–1090, 1994.

R. R. Trippi and D. DeSieno. Trading equity index futures with a neural network. *The Journal of Portfolio Management*, 19(1):27–33, 1992.

B. Vanstone and T. Hahn. *Designing Stock Market Trading Systems: With and Without Soft Computing*. Harriman House, 2010. ISBN 1906659583, 9781906659585.