Winton Stock Market Challenge: Using a Deep Learning Framework for Time Series Stock Market Prediction

Deep Learning

Joaquin Borggio
jborggio@stanford.edu

Sagar Maheshwari
msagar@stanford.edu

Cheyenne Sadeghi
csadeghi@stanford.edu

Maxim Serebriakov
maximser@stanford.edu

1 Motivation

It is commonly thought that the stock market is unpredictable due to the abundance of constantly changing variables that impact a company's stock performance; however, with the use of a deep learning framework, it may be possible to predict trends within the stock market with a higher accuracy than a random 50-50 guess. Using time series data, as well as a deep learning algorithm consisting of wavelet transforms (WT), stacked auto-encoders (SAE), and recurrent neural network layers (RNNs), our goal is to predict stock market performance at a level that's above the average human performance.

Another factor that inspired our team to research the question of stock market prediction is the growing prevalence of automation within the world of financial technology. The term high frequency trading (HFT) has been appearing within the lexicon of many individuals. The popularity of this notion is growing rapidly since its inception. In 2012, HFT accounted for more than 60 percent of the world's foreign exchange futures trading volume, and in 2016 HFT accounted for more than 80 percent of the volume. In fact, many modern economists believe that the next financial crisis will be directly related to liquidity issues in the HFT market. We hope to utilize our knowledge from CS230 in order to join the HFT surge ourselves. Our paper falls into the category of quantitative investment papers, as we use time series stock market prediction data to determine future stock returns.

In this work, we describe the development of two stock return prediction models. Our initial modeal uses 1) denoising wavelet transforms and 2) fully-connected neural network layers. Our final model includes 1) denoising wavelet transfrom, 2) stacked auto-encoders (SAE), and 3) layered recurrent neural networks (RNNs). We use both model architectures to compare performance and pinpoint the performance gain of RNNs due to the time-series nature of the data. In Section 2, we describe the dataset used for our deep learning model. In section 3, we detail the methodology used, including discrete wavelet transforms, network architecture, and cost function. In Section 4, we report and analyze results from the experiments we have run. We conclude with future work in Section 5 and contributions in Section 6. The code for our project can be found on GitHub: `https://github.com/msagar7/golddust`

2 Dataset

For our project, we used a data set provided to us by the Winton Stock Market competition from Kaggle. The data set consisted of 40,000 different stocks each of which has 25 features, 180 returns, as well as the returns from the close of the trading day on D-2, D-1, D+1, D+2. The detailed description of what each of the provided variables do is shown below:

Feature 1 to Feature 25: different features relevant to prediction

Ret MinusTwo: this is the return from the close of trading on day D-2 to the close of trading on day D-1 (i.e. 1 day)

Ret MinusOne: this is the return from the close of trading on day D-1 to the point at which the intraday returns start on day D (approximately 1/2 day)

Ret 2 to Ret 120: these are returns over approximately one minute on day D. Ret 2 is the return between t=1 and t=2.

Ret 121 to Ret 180: intraday returns over approximately one minute on day D. These are the target variables you need to predict as id 1-60.

Ret PlusOne: this is the return from the time Ret 180 is measured on day D to the close of trading on day D+1. (approximately 1 day). This is a target variable you need to predict as id 61.

Ret PlusTwo: this is the return from the close of trading on day D+1 to the close of trading on day D+2 (i.e. 1 day) This is a target variable you need to predict as id 62.

However, once the team downloaded the Winton Stock Market competition data set, we discovered that there are many missing elements from the data. To address this issue, we developed a python script that attempts to approximate these data gaps. For all non-time series features, we found the arithmetic mean of all the nonempty values in the columns, and we replaced the empty values with the associated mean. For missing time-series return data, we replace missing returns with the calculated mean of available returns for each example in the data. Later, we may use other approaches in filling missing values for rows. Perhaps we could average out the surrounding time values. If, for instance, we didn't have a value for Ret 3 and Ret 4, we could average out the two values of Ret 2 and Ret 5 to receive a better approximate. Another idea is to implement a forced inverted dropout procedure where features with missing values are forcibly dropped out during training. Either way, we feed this full dataset with no missing values to our deep learning model.

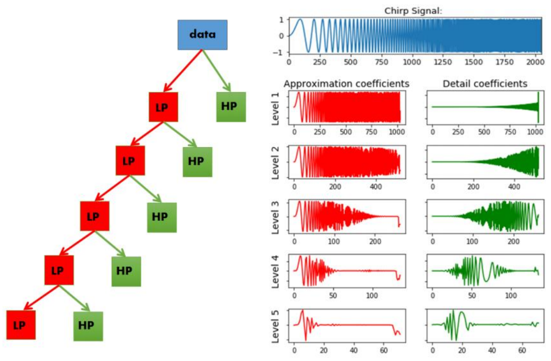
3 Methodology

3.1 Denoising

Once we acquire our data-set we first apply a Discrete Wavelet Transform (DWT) to separate noise from our original time-series data. Similar to a Fourier transform, the wavelet transform decomposes a function into the frequencies that make it up, except now the transform is based on small wavelets with limited duration. We apply the Haar function as the wavelet basis function to reduce processing time significantly and to decompose the financial time series into its time and frequency domains. For our final project we will apply the wavelet transform multiple times to reduce over-fitting, but for this milestone we only apply it once. For later milestones, we intend to apply the WT around four times, to ensure that most of the noise in our data is eliminated. The function for the wavelet transform is shown below, where a and τ are the scale factor and translation factor, respectively. $\Phi(t)$ is the basis wavelet:

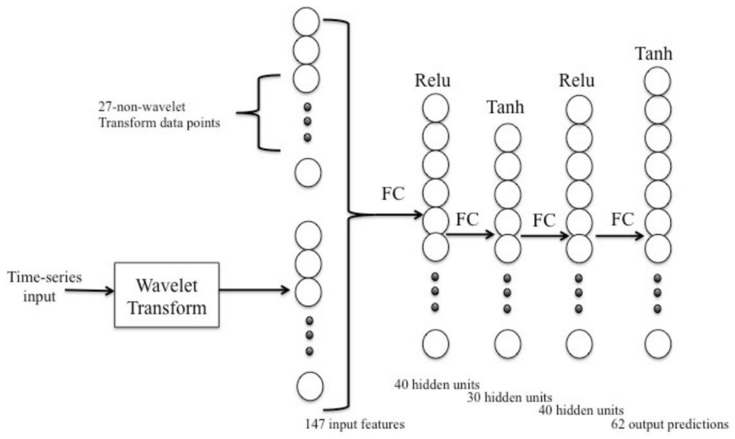
$$\Phi_{a\tau}(t) = \frac{1}{\sqrt{a}}\Phi(\frac{t-\tau}{a})$$

We created the DWT by inputting 119 samples, consisting of our time series data, portrayed in our data set as values from 2 to 120. During the manipulation, we apply the DWT, and this manipulation outputs two arrays. The first array consists of low frequency coefficients. The second array consists of high frequency coefficients. These two arrays are generated using the PyWavelets package in python. We concatenate the two arrays such that the wavelet output mirrors the original data input. Each of the arrays are of size 60, so when concatenated our array contains 120 samples, which is 1 extra sample from our inputted 119. The graph below portrays the impacts of a WT on a signal using multiple levels of transformation. In the figure below, you can see how the DWT works on a chirp signal:



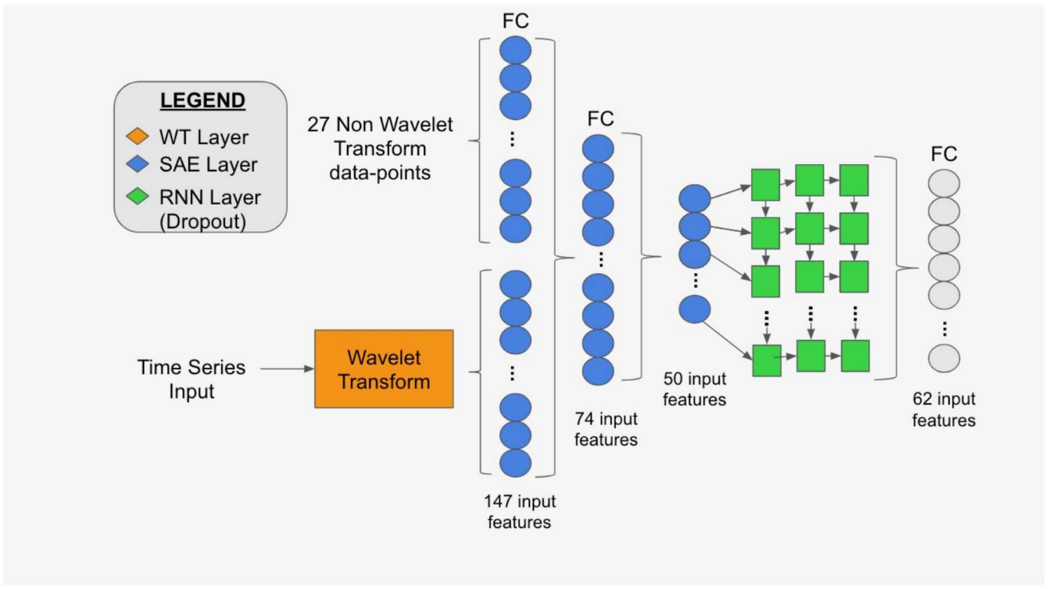
3.2 Fully Connected Network Model

After applying the DWT, we passed our time-series denoised data, as well as 27 non-wavelet-transform data points from our original data set into a fully connected network consisting of 4 layers. We then take our 147 input features and apply the Relu function in our fully connected network to get 40 hidden units. After the tanh function is applied to generate 30 hidden units, and after we reapply the Relu function to go back to 40 hidden units again. Finally we run the units through a tanh function to get our 62 output predictions. The figure below outlines our architecture.



3.3 Recurrent Neural Network Model

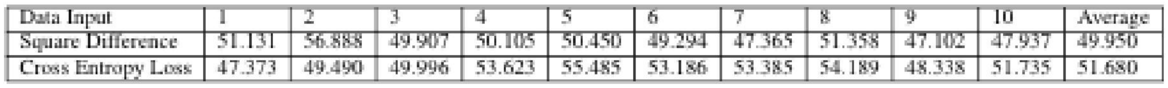
After analyzing results from the fully connected network model, we modified our architecture to better fit time-series data. The DWT, which seemed to work well at denoising and picking out frequencies, was kept for the final model. We replaced the fully connected section with a stacked autoencoder (SAE). SAEs are a form of unsupervised learning where the model learns an encoding and decoding function. After the model is trained, then the input is passed in to only the encoder, and the output is given to the next stage. Our final stage was a Recurrent Neural Network (RNN) with three layers and dropout regularization in each layer. As explained in the next section, hyperparameter tuning demonstrated that a dropout probability of 10% produced the least amount of variance and training our model on 25 epochs gave us the highest accuracy. The figure below outlines our RNN model architecture.



4 Results

4.1 Fully Connected Network Model

With 40000 training examples, we trained and tested our fully connected network model using 10-fold cross validation. In two separate instances of training, we trained our model using both a cross entropy loss function and a square difference loss function. Recall from the network architecture that the output of the model $\hat{y}$ is a 62-entry column vector. To evaluate the model, we first perform an element-wise comparison between $\hat{y}$ and target y to see whether the model correctly predicted a rise (positive value) or fall (negative value) in the stock return value. We determine the model accuracy by computing the fraction of predictions which it correctly predicted either rise or fall.



Data Input	1	2	3	4	5	6	7	8	9	10	Average
Square Difference	51.131	56.888	49.907	50.105	50.450	49.294	47.365	51.358	47.102	47.937	49.950
Cross Entropy Loss	47.373	49.490	49.996	53.623	55.485	53.186	53.385	54.189	48.338	51.735	51.680

As you can see from the chart above, our current implementation receives around 50% accuracy when trained on both cross entropy loss and square difference loss. It is apparent that the algorithms performance is equivalent to essentially guessing whether the stock return will increase or decrease. Potential reasons for this 50% accuracy may be because the discrete wavelet transformations were unable to successfully denoise the time-series stock return values, or that some of our features were particularly unhelpful for prediction. More importantly, the low accuracy verifies the fact that the fully connected network model is unable to extract time-based patterns from the data. Hence, we also examine the performance of the recurrent neural network model, better suited for time-series data.

4.2 Recurrent Neural Network Model

Prior to evaluating the performance of the model, we tuned hyperparameters. The two hyperparameters of particular importance were the number of epochs used to train the model, and the dropout probability. For the number of epochs, we tested values 1, 5, 10, 25, 30, 50, 100. For all tests, we decided to fix the dropout probability at 0.1 The performance for the different epoch values is shown below:

Total Epochs (dropout = 0.1)	Training Set	Test Set
1	51.9913%	51.8105%
5	51.4072%	51.4071%
10	57.1145%	57.1596%
25	65.4812%	65.4697%
30	64.0671%	64.1763%
50	64.1439%	64.0503%
100	60.8273%	60.7885%

Notice that at 1 and 5 epochs the model performs with roughly 51% accuracy on the test set. At 10 epochs, the model performs at 57.16% accuracy on the test set. These performances are similar to that of the fully connected network model. When looking at the decreasing loss values for all epochs, it is evident that the model has not converged to the optimal parameter values. At 25 epochs, the model performs best with an accuracy of 65.47%. When looking at the loss values, it is apparent that the parameters have converged to their optimal values. At 30 and 50 epochs, the model performs with slightly lower accuracy around 64%. Finally, at 100 epochs, the model performs with 60%. Regardless of number of epochs, it is important to note that performance of training and testing sets are similar. Hence, when evaluating optimal epochs, we are more concerned with optimal parameter values than variance. Based on these results, we decide to train the recurrent neural network model using 25 epochs.

After determining the number of epochs during model training, we tune the dropout probability parameter. Specifically, we tested the following probabilities: 0.08, 0.1, 0.15, 0.2, 0.25. Based on the results from epoch hyperparameter tuning, we fix the number of epochs used during training to 25. The performance for the different dropout probabilities is shown below:

Dropout	Train Set	Test Set
0.08	63.6774%	63.6786%
0.1	65.6912%	65.2534%
0.15	64.3821%	64.3584%
0.2	64.8197%	64.8297%
0.25	63.8641%	63.8105%

At dropout probability 0.08, the model performed with 63.68% accuracy. At dropout probability 0.1, the model had optimal performance with accuracy 65.25%. At dropout probabilities 0.15 and 0.2, the model performed with an accuracy around 64%. Finally, at dropout probability 0.25, the model performed with 63.8% accuracy. The results are highly informative. With dropout probability 0.08, the lower accuracy may indicate not enough regularization. With dropout probability 0.25, the lower accuracy may indicate that inter-feature dependencies are not optimally learned due to a higher dropout probability. Hence, based on these tuning results, we choose a dropout probability of 0.1.

Having completed hyperparameter tuning, we now evaluate the performance of the recurrent neural network model. As we did with the fully connected network model, we train and test the model using 10-fold cross validation.

	1	2	3	4	5	6	7	8	9	10	Average
Train	64.8	65.09	64.87	65.41	65.22	64.96	65.24	65.13	65.4	63.75	64.99
Test	64.79	65.15	64.91	65.45	65.06	64.96	65.18	65.02	65.54	63.63	64.97

After performing 10-fold cross validation, we see that the recurrent neural network model successfully predicts stock price movement with 64.97% accuracy. This is comparable to the performance of models from many other similar works [2]. Of particular importance is the comparison of performance between the fully connected network model and the recurrent neural network model. The fully connected network model performed with 51.68% accuracy, while the recurrent neural network model performed with 64.97% accuracy. Clearly, the RNN-based model significantly outperforms the fully connected network model. These results are expected, however, since RNNs are superior at learning time-series data like our successive stock prices.

An interesting issue we first encountered while going through the process of hyperparameter tuning is that our training accuracy and test set accuracy are marginally different. This is quite odd, as normally the test accuracy is lower than training accuracy due to variance. A potential explanation of this occurrence is that perhaps the training and test sets came from similar distributions, resulting in similar performance. This is a possible explanation, as our data set was provided by the Winton Group investment management company. For future work, it would be beneficial to create a data scraping script that would collect the same features we analyzed, but for an eclectic group of companies that were not in our training set. Such efforts may help to add more heterogeneity to the testing sets and ensure that this odd observation does not occur.

5 Future Work

In this paper, we have explored stock price predictions using both a fully-connected network model and a layered recurrent neural network model. The immediate next step is to explore and implement a model based on layered Long Short-Term Memory (LSTM) networks. When comparing the fully-connected model to the RNN model, we notice that the RNN model significantly outperforms the fully-connected model when predicting stock prices. This is reasonable considering the effectiveness of RNNs on sequential data. However, it is clear that older stock prices could potentially impact current stock prices. For example, a company that had a public relations scandal a month ago could still be experiencing depreciation in stock value in the status quo. Due to such long-term dependencies, the use of LSTMs could significantly improve model performance.

6 Contributions

While everyone contributed to every part of the project, the work was broken down into manageable parts led by groups of two. Sagar and Maxim handled the data preprocessing so that the dataset is compatible with deep learning models. Joaquin and Cheyenne handled researching and implementing the Discrete Wavelet Transform. Maxim and Sagar took on researching and implementing the stacked autoencoders. In the next stage, Joaquin and Sagar coded the wavelets, stacked autoencoders, and layered Recurrent Neural Networks while Maxim and Cheyenne managed the data collection and analysis. The hyper-parameter tuning was also completed by Maxim and Cheyenne. Lastly, everyone helped together to design the poster and draft the paper.

References

1. Admin. (2018, December 21). A guide for using the Wavelet Transform in Machine Learning. Retrieved from http://ataspinar.com/2018/12/21/a-guide-for-using-the-wavelet-transform-in-machine-learning/

2. Bao, W., Yue, J., Rao, Y. (n.d.). A deep learning framework for financial time series using stacked autoencoders and long-short term memory. Retrieved from https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0180944

3. Crashes and high frequency trading. (2012, August). Retrieved from https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/289016/11-1226-dr7-crashes-and-high-frequency-trading.pdf

4. How has Algorithmic Trading Impacted the Futures Markets? (2018, October 09). Retrieved from https://tradingsim.com/blog/how-has-algorithmic-trading-impacted-the-futures-markets/

5. The Winton Stock Market Challenge. (n.d.). Retrieved from https://www.kaggle.com/c/the-winton-stock-market-challenge/data