Stock Market Time-Series Regression: Predicting the S&P-500 with LSTM, GRU, RNN Based Models

J. Baldwin

Abstract

Predicting the price of the stock market can be incredibly lucrative, however in equal magnitude of the potential monetary gain is stochasticity and unpredictability. However, with recent developments in neural-networks and different machine learning approaches, regression with machine learning is becoming more and more common. There are many types of Neural Networks, many designed for specific tasks. This paper explores the outcomes of applying an atypical neural model for regression to predicting the market. It trains several recurrent neural models that are commonly applied to natural language processing tasks, and compares and contrasts their performance. The experiment revealed the RNN model had the lowest error, approximately 20% of the LSTM's Mean Squared Error. This suggests stochastic regression modeling is best achieved by an RNN's lightweight and efficient architecture, when the data is such a low dimension.

1 Introduction

Ever since its inception, the stock market has stymied the minds of the world's brightest in their attempts to coax its ebbs and flows into an accurate model. However, Neural Networks have proved to be an admirable contender for this problem. To understand the networks used in this paper, it is useful to consider the shortcomings of different types of networks, and to describe the architecture of these RNN's, in order to understand why Recurrent Neural Networks have such a natural application to this problem.

In the past, the most common neural model for stock price regression was a Convolutional Neural Network (CNN), which is one that is often used for image classification, and Generative Adversarial Networks. At a high level, CNN's in image classification takes input as a matrix of pixels, and using a window of any size sub-matrix, slides across the matrix of pixels, doing some computation at each step. In applying these models to stock market prediction based on price, we are scaling down from a matrix to a sequence, and our window is no longer a sub-matrix but a sub-interval. While this model can be effective, it only encodes the temporal importance implicitly. A Recurrent Neural Network was designed in order to more explicitly hold on to temporal significance and escape the vanishing gradient problem.

Recurrent Neural Networks is a blanket term as many models with different sub-structure fall into this category. The three that we will be considering are the Long-Short Term Memory (LSTM) node, the base Recurrent Neural Network (RNN), and the Gated Recurrent Unit (GRU) [1] [2] [3]. These networks were originally designed for language modeling, but with their ability to be repeated on the data allows for a stronger temporal encoding.

The LSTM node as pictured in figure 1 is most notably composed of three functionalities. First, the Input Gate allows new information to be added to the LSTM accumulating value, without overwriting it completely. Second, the Forget Gate in essence, enables the LSTM to remove uninteresting or unimportant data from the input to the next iteration of itself given the new data at each iteration. Finally, the Output Gate which is the final selector of relevant

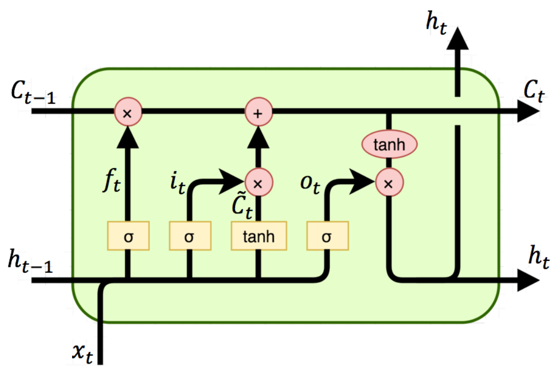

Figure 1: The LSTM Architecture

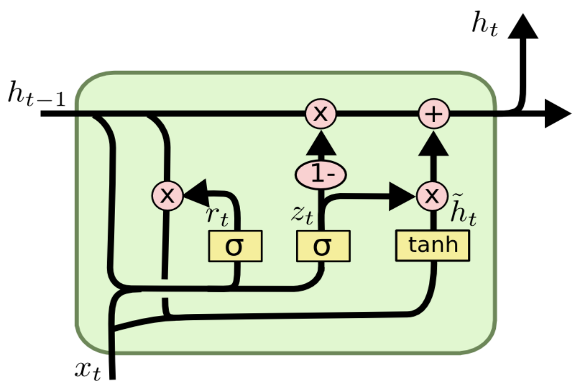

Figure 3: The GRU Architecture

data to be moved on into the next iteration. With these three key structures, we can see the LSTM appears to be a dense node with many computations.

The RNN Node as pictured in figure 2 is a far more lightweight node, which has an ouptut component and a hidden component at each time step. The more concise architecture allows for far more of these nodes to be stacked upon each other without drastically increasing computation time.

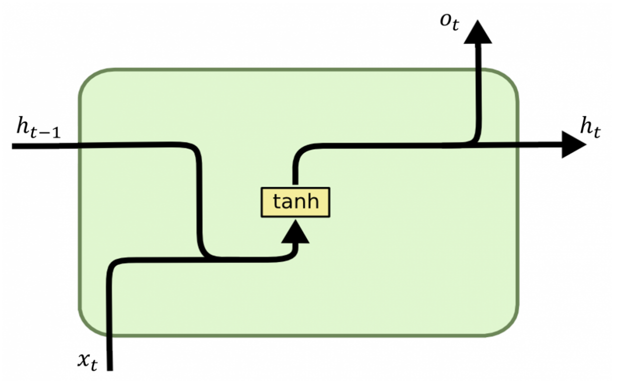

Figure 2: The RNN Architecture

Finally, the GRU model as pictured in figure 3 is composed of two gates, update and reset, as opposed to the LSTMs input, output and forget. The benefit of the GRU model is that there is the possibility of overwriting the value at each step, thus creating a more efficient model.

The numerical method used in order to train these models is Stochastic Mini-batch gradient descent. This allows us to train the models by entering input to make a prediction, then computing the gradient and backpropogating through the network. the mini-batch allows our model to not take into account too few or too many samples for each gradient computation and step. As compared to standard SGD, mini-batch SGD has a faster computation time, since using the average outcomes and gradients for a few samples at a time can lead to quicker convergence.

The purpose of this research is to train these three models to the best of their abilities given standardized hyper parameter ranges across the models, using a randomized search of parameters. Thus, the remainder of the paper briefly covers the collection of the data from a Bloomberg Terminal and its cleaning and preparation, followed by the implementation and training of these three models using mini-batch SGD. Then, the paper will compare the results to determine which architecture is best suited for predicting the S&P 500 as well as a discussion of potential future work.

2 Implementation

In this section, we cover the preparation, the selection and implementation of the algorithms chosen to train the models in the most optimal fashion.

2.1 Data Scraping and Preparation

The data we desired was the closing price for each since 2000 of SPY US Equity. To collect this data, we accessed a Bloomberg terminal via an api and pulled the data. Then, further data preparation was necessary, as we needed a way to normalize the data, as normalized data performs better in Mini-Batch SGD, but then reverse the normalization when we wanted to see our price predictions. We decided upon a standard normalization of the formula

$$x_i = \frac{x_i - \mu}{\sigma} \tag{1}$$

where μ is the mean of the dataset, and σ is the standard deviation. Saving these values for later, we were able to normalize and re-inflate these values at will.

The final step in data preparation was to create a sliding window that would pass price data from the past n days as the training sample. Once this was done, we were able to train our models.

2.2 Mini-Batch Stochastic Gradient Descent Algorithm

Mini-Batch Stochastic Gradient Descent is a specific type of Gradient Descent, where the training inputs are passed to the model in a predetermined batch size group prior to each computation of the error gradients and backpropagation. Standard Gradient Descent in essence, uses a batch size that is equal to the entire size of the dataset. Thus, even with a dataset of 6 thousand inputs, iterating over each one prior to taking one update step would prove to be computationally inefficient. That being said, pure Stochastic Gradient Descent uses a batch size of 1, that is, they compute gradients and back-propagate those findings at each input. This can be computationally inefficient due to its contradictory nature, since it is trusting the accuracy of each sample to lead it to a global minima. Mini-Batch SGD is the computationally efficient adaptation of these previous two methods, where the batch size can be optimized to create the lowest validation loss.

Algorithm 1 Mini-Batch SGD Algorithm

```
1: Initialize hyper-parameters
2: $model$ = instantiate model
3: for $epoch$ in $num\_epochs$ do
4:     for $id$, ($x\_batch$, $y\_batch$) in training data do
5:         $gradients = 0$
6:         $output = model(x\_batch)$
7:         $MSE$ = loss_function($output$, $y\_batch$)
8:         $gradients$ = compute_gradients($MSE$)
9:         $w_i = w_i - \eta gradients(w_i)$
10:    end for
11:    compute avg training losses
12:    for $id$, ($x\_batch$, $y\_batch$) in validation data do
13:        $output = model(x\_batch)$
14:        $MSE$ = loss_function($output$, $y\_batch$)
15:    end for
16:    compute avg validation losses
17: end for
```

2.3 Hyper Parameter Optimization with Ray Tune

Each Recurrent Neural Network used in this paper is created with its own model, and its own hyper parameters. Finding the most optimal pairings of such hyper parameters for each model can be incredibly time consuming. Also, it has been shown in a paper by James Bergstra that random search of hyper-parameter ranges is theoretically and empirically more efficient than searching on a grid [4]. Thus, via a Pytorch-integrated library, Ray Tune, we were able to give each of the three models the same ranges for all of the hyper parameters, and search over those ranges to find their optimal pairings. This leads to a more fair discussion when comparing these models performances on the stock market since they were all afforded the same ranges of hyper parameters.

3 Results

The LSTM, Pure RNN, and GRU models with random-search hyper-parameter optimization led to the RNN outperforming the GRU and the LSTM based models.

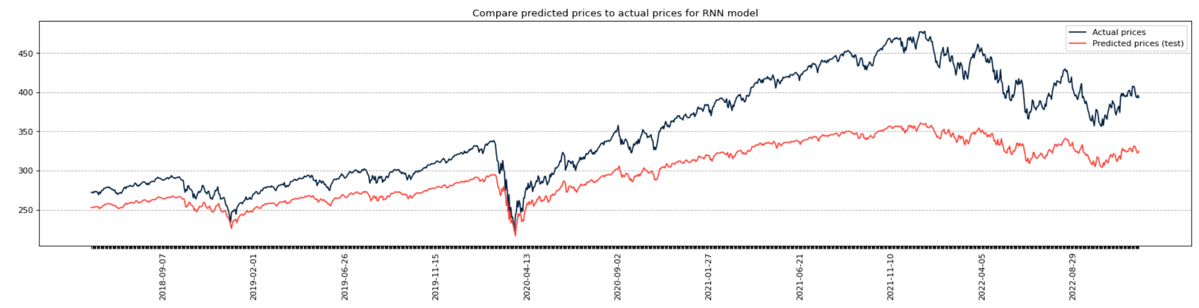

Figure 4: RNN performance on Test Set

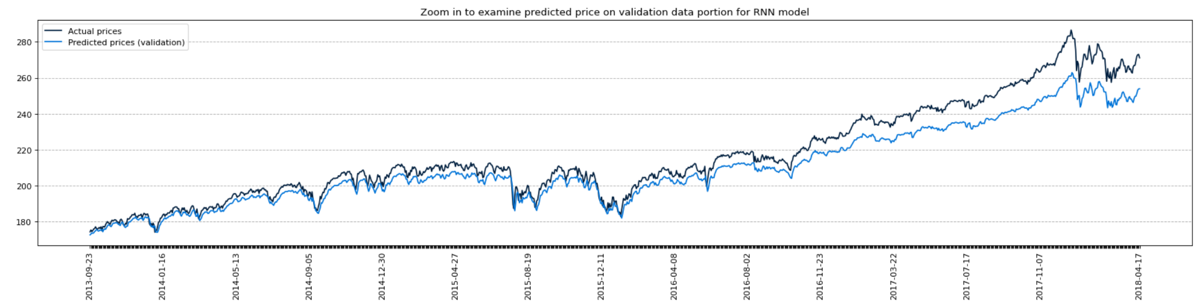

Figure 5: RNN performance on Validation Set

Model	MSE	RMSE	R2 Score
LSTM	1.829684	1.352658	-2.861469
RNN	0.397027	0.630101	0.162092
GRU	0.550551	0.741991	-6.502058

Figure 6: Test Error Calculation Table

On the test set, the RNN achieved less than 0.4 MSE, outperforming the GRU and LSTM, notably clocking in an MSE more than four-fold lower than that of the LSTM. Notice also, the RNN model was the only one able to achieve a positive R2 Score, while not ideal (which would be close to 1), it is far better than its LSTM and GRU counterparts. A bit counterintuitively, we find that the GRU's R2 score is more than two times worse than the LSTM models R2 score, despite the LSTM having the highest MSE.

MSE, RMSE, and R2 Scores are helpful metrics, but this is a tangible problem given the ubiquity of stock market price charts, so a more vivid visualization is possible. Figure 4 is the test-set regressed on the actual S&P Price from roughly 2018-2022. Note that these models were trained on the data approximately from 2000-2014, the validation set spanned from 2014-2018, and the test set covered the most recent dates 2018-2022.

As we can see, the RNN, while being the best model we created, was still relatively off the mark on per-day price, reaching a maximum difference in price of around $100. We can see it is able to predict these trends and choose the next days price based on the previous prices to a relatively good extent. The dip that occurs halfway through the figure is a relevant demonstration of such behavior. That being said, this test-data was predicted using a model, trained on data from eight years prior, thus the performance on the validation set feels relevant to the discussion. As we can see in figure 5, the RNN was able to predict over the entire period within a price range of $20, most of which are close predictions towards the beginning, which begin straying as time progresses.

4 Conclusion and Future Work

The RNN likely outperformed the GRU and LSTM models due to its lightweight architecture, since in such a low-dimensional space, it would make sense the LSTM and GRU struggled with overfitting the data, leading to higher errors and worse fits. The GRU being the second lightest model, it follows that the MSE was second only to the RNN. Although, an interesting discrepancy occurred given the r^2 score. All of the models yielded poor r^2 scores, likely due to the stochasticity of the problem, however the GRU's was the worst overall, despite having a lower MSE. We believe this implies the r^2 score is not the best metric available for this model, something to note for future work in this field.

We hope to explore newer models that are typically not used for time-series regression could be researched, like a Transformer, namely BERT. Also, further training on an RNN model, perhaps to generalize over all of the equities in the S&P 500, not just the SPY US Equity, could lead to a more general model. One may also ask the question, a complex system like the LSTM underperformed in this scenario, but what if it was fed some more potent data with more dimensions to predict. These are all just a few routes we hope to pursue in the future.

References

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural Computation, 9(8):1735–1780, 1997.

[2] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[3] Deep gate recurrent neural network. CoRR, abs/1604.02910, 2016. Withdrawn.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. Journal of machine learning research, 13(2), 2012.