Recurrent Neural Networks in Stock Price Prediction

Rogers, Trent
Computer Science
University of Texas at Dallas
2020

***Abstract*— A simple Jordan recurrent neural network was implemented in Python to predict daily IBM closing price based on over 14 years of data. The result was relatively successful, and while not reliable enough to base investments on, close approximations were obtained and were encouraging to future work.**

I. Introduction

Recurrent neural networks were initially invented in the 1980's, but as with most deep learning techniques, they have seen much more interest and usefulness in recent years as computational power has increased and larger sets of training data have become available. A recurrent neural network is ideal for predicting timeseries data in which one value depends on a previous value, as in predicting weather, stocks, or other sequences. Ordinary (non-recurrent) neural networks are not suited for such tasks. For example, take such a simple sequences as 1, 2, 3, 4, 5, … If such a sequence was fed into an ordinary artificial neural network, one piece at a time, the network would never learn to predict it, since it has no memory of previous inputs or outputs or how the next prediction should relate to the past one. A recurrent neural network has weights, not only between inputs and outputs, but between previous outputs and following outputs. Basing new predictions partially on new inputs and partially on previous predictions allows a net to learn how to predict sequences of variable length.

Some more advanced algorithms have been invented in more recent years that allow RNNs to remember longer sequences of data and even allow them to decide what outliers should be forgotten. Of particular interest is the LSTM (long short term memory) model, which may very well be the subject of future work, but this project only showcases the fundamentals of RNNs and lays preliminary groundwork for more evolved models with a simple Jordan RNN. A Jordan network is one of two types of simple recurrent networks (SRNs), with the other being Elman networks. An Elman network bases its nth hidden layer vector on its (n-1)th hidden layer vector, whereas our Jordan network bases its nth hidden layer vector based on its (n-1)th *output* vector. The Jordan network was chosen due to the fact that both input and output for our network consist of a single scalar value. If we were to predict multiple daily features of a stock with larger input and output vectors, an Elman network could preserve more complex information across iterations.

II. Methods

A. Dataset

The data used in training was 5200 days of IBM's closing price. Historical IBM stock data was retrieved from the free Alpha Vantage API, going back to 1999. It was written to a .csv file that is now hosted on my GitHub (user trentr314). In RNN_project.py, only the closing price for the past 5200 days was kept as training data. More extensive information such as open, high, low, and volume is also available in the .csv file and may be used in future projects.

B. Implementation

The basic explanation of the workings of the algorithm is as follows: an input is taken in and multiplied by an input weight matrix, added to the product of the previous output and a recurrent weight matrix, then added to an input bias, to yield a hidden layer vector. The hidden layer vector is multiplied by an output weight matrix, then added to an output bias, to yield an output (a prediction). The output is multiplied by the recurrent weight matrix and fed back in as recurrent input, which is once again added to the product of the next input and the input weight matrix, as well as the input bias, etc., etc. Input, output, and hidden node values are kept for one batch. At the end of a batch, error is backpropagated and the input, output, and recurrent weight matrices, as well as the biases, are updated via gradient descent. The program iterates through batches until it has completely gone through the dataset (one epoch). Then it repeats this process for as many epochs as are specified, updating the weights and biases for every batch of every epoch until complete.

Note that on the first iteration of each batch, we do not have a previous output for that batch. A copy of the actual input is sent in both as input and as a previous prediction, as if the previous prediction was perfectly accurate. This was the best method found for working around the lack of previous output while using a reasonable batch size, as using random numbers or zeros or any other fixed value would throw the algorithm off with a large error regardless of how accurate the net was becoming.

C. Hyperparameters

Easily modifiable hyperparameters include learning rate, hidden layer size, batch size, and number of epochs. Additional hidden layers are not an option in this implementation. They are thought to be unnecessary with relatively simple data taking a single input scalar and giving a single output scalar, however, experimentation with multiple hidden layers may be done at a later date. There are no activation functions in this implementation, since there is only one hidden layer and the desired predictions are relatively large scalars (within the range 50-200). A sigmoid or tanh would squish numbers too small, and while a ReLU could have been a good choice, it was thought unnecessary and results are good without an activation function. Again, more experimentation may be done at a later date. A rough estimate of a good range of hyperparameters was found to be around 6-10 hidden nodes in the hidden layer, a learning rate of around

5e-12 - 5e-11, and 200-3000 epochs. Batch sizes of 10 were used.

Please note that the default hyperparameters in the program submitted are not ideal. I expected my program to be run by someone with limited time and it takes less than 30 seconds to run at 200 epochs. You can get more reliable accuracy if you increase the epochs, but it will take longer.

D. Theory

One of the most difficult parts of developing this implementation was doing the mathematics for the gradient descent. Calculus must be applied to find partial derivatives of the error with respect to each component that must be updated during backpropagation, which must be done using sums of values across multiple examples and interpreted in terms of the linear algebra used to multiply and add the vectors and matrices. Following are equations used to find the correct values to use in backpropagation.

We will start with the output matrix update rule, which is

$$w\_out_{t+1} = w\_out_t - \Delta w\_out_t$$

The change can be expressed as

$$\Delta w\_out = L_R\left(\frac{\partial L}{\partial w\_out}\right)$$

Where L_R is the learning rate and L is the loss function. This partial derivative will work over a sum of multiple examples as we are doing batch gradient descent. Expanding the partial derivative, we get

$$\frac{\partial L}{\partial w\_out} = \sum_{i \in examples}^{i} \frac{\partial \frac{1}{2}(t_i - h_i)^2}{\partial w\_out}$$

Where examples means all the examples in the batch, t means our target (the actual value we're trying to predict) and h means our hypothesis (the output from our model). Expanding this via the chain rule, we get

$$\sum_{i \in examples}^{i} \frac{\partial \frac{1}{2}(t_i - h_i)^2}{\partial w\_out}$$
$$= \sum_{i \in examples}^{i} \frac{\partial \frac{1}{2}(t_i - h_i)^2}{\partial h_i} \cdot \frac{\partial h_i}{\partial w\_out}$$

Now we can calculate the partial derivative for the error with respect to the hypothesis and expand the second part:

$$\sum_{i \in examples}^{i} (h_i - t_i) \cdot \frac{\partial\, \overrightarrow{hidden_i} \cdot w\_out}{\partial w\_out}$$

Where $\overrightarrow{hidden_i} \cdot w\_out$ denotes the dot product of the hidden layer nodes vector after the i^{th} example and the w_out output weight matrix. The next step is tricky and relies on the internal representation of the hidden layer vector over the examples in the batch:

$$\sum_{i \in examples}^{i} (h_i - t_i) \cdot \frac{\partial\, \overrightarrow{hidden_i} \cdot w\_out}{\partial w\_out}$$

$$= \left(\sum_{i \in examples}^{i} \overrightarrow{hidden_i}\right)^T \cdot \left(\sum_{i \in examples}^{i} (h_i - t_i)\right)$$

For n examples in a batch and m nodes in the hidden layer, an (n×m) matrix is stored with every row being a hidden layer vector and n rows for n examples. A (n×1) column vector is stored for $h_i - t_i$, subtracted elementwise. The transpose of the (n×m) matrix multiplied with the error column vector gives us an (m×1) column vector that has the right dimensions to subtract from w_out. Thus our final update rule for the output matrix is

$$w\_out_{t+1} = w\_out_t$$
$$- L_R\left(\left(\sum_{i \in examples}^{i} \overrightarrow{hidden_i}\right)^T \cdot \left(\sum_{i \in examples}^{i} (h_i - t_i)\right)\right)_t$$

Now we calculate the update rule for the input layer:

$$w\_in_{t+1} = w\_in_t - \Delta w\_in_t$$

The change, again, can be expressed as

$$\Delta w\_in = L_R\left(\frac{\partial L}{\partial w\_in}\right)$$

Where the learning rate is the same and the loss function is the same as well. We will expand the partial derivative again, with a sum for batch gradient descent as before.

$$\frac{\partial L}{\partial w\_in} = \sum_{i \in examples}^{i} \frac{\partial \frac{1}{2}(t_i - h_i)^2}{\partial w\_in}$$

Expansion with the chain rule will take us further this time, as the input layer is further from the output:

$$\sum_{i \in examples}^{i} \frac{\partial \frac{1}{2}(t_i - h_i)^2}{\partial w\_in}$$

$$= \sum_{i \in examples}^{i} \frac{\partial \frac{1}{2}(t_i - h_i)^2}{\partial h_i} \cdot \frac{\partial h_i}{\partial\, \overrightarrow{hidden_i}} \cdot \frac{\partial\, \overrightarrow{hidden_i}}{\partial w_{in}}$$

With the same linear algebra as before,

$$= input^T \cdot \left(\sum_{i \in examples}^{i} (h_i - t_i)\right) \cdot (w\_out)^T$$

And the final input update rule is

$$w\_in_{t+1} = w\_in_t$$
$$- L_R\left(input^T \cdot \left(\sum_{i \in examples}^{i} (h_i - t_i)\right) \cdot (w\_out)^T\right)_t$$

In our implementation, w_rec, the recurrent weight matrix, is in essentially the same place as our input weight matrix, so this rule can be copied for it. All that's left is the biases:

$$bias_{t+1} = bias_t - \Delta bias_t$$

$$\Delta bias = L_R\left(\frac{\partial L}{\partial bias}\right)$$

$$\frac{\partial L}{\partial bias} = \frac{\partial \frac{1}{2}(t_i - h_i)^2}{\partial h_i} \cdot \frac{\partial h_i}{\partial bias}$$

And since the derivative of the output (hypothesis) with respect to the bias is just 1, we get

$$\frac{\partial L}{\partial bias} = \sum_{i \in examples}^{i} (h_i - t_i)$$

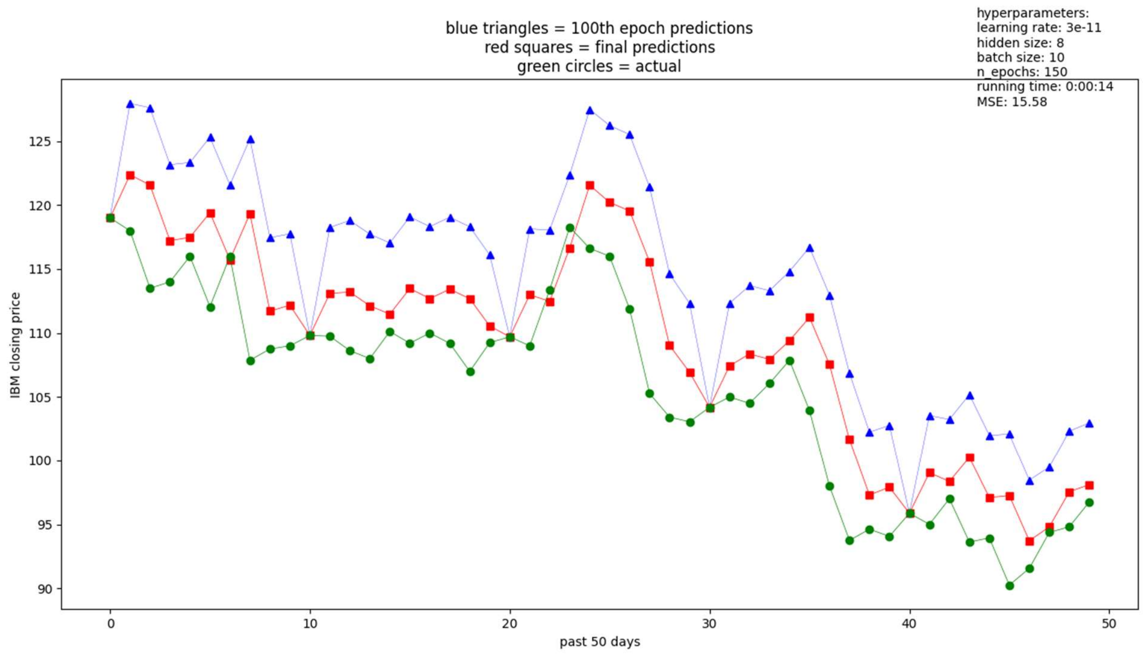
and the bias update rule becomes

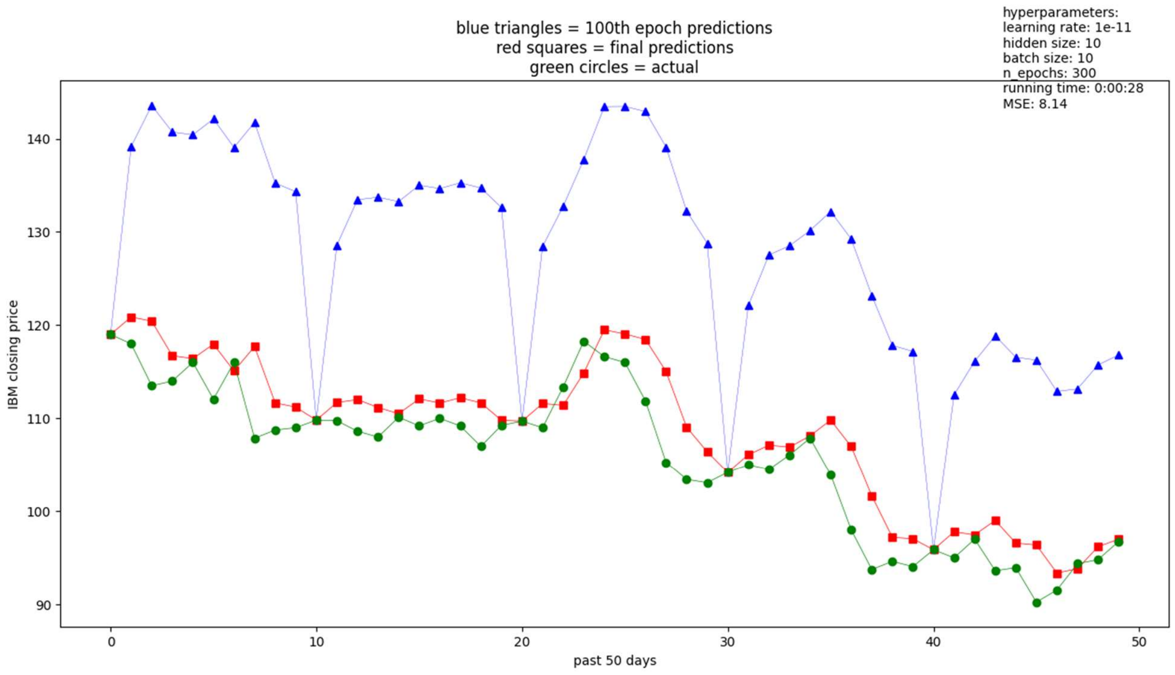
$$bias_{t+1} = bias_t - L_R(\sum_{i \in examples}^{i} (h_i - t_i))_t$$

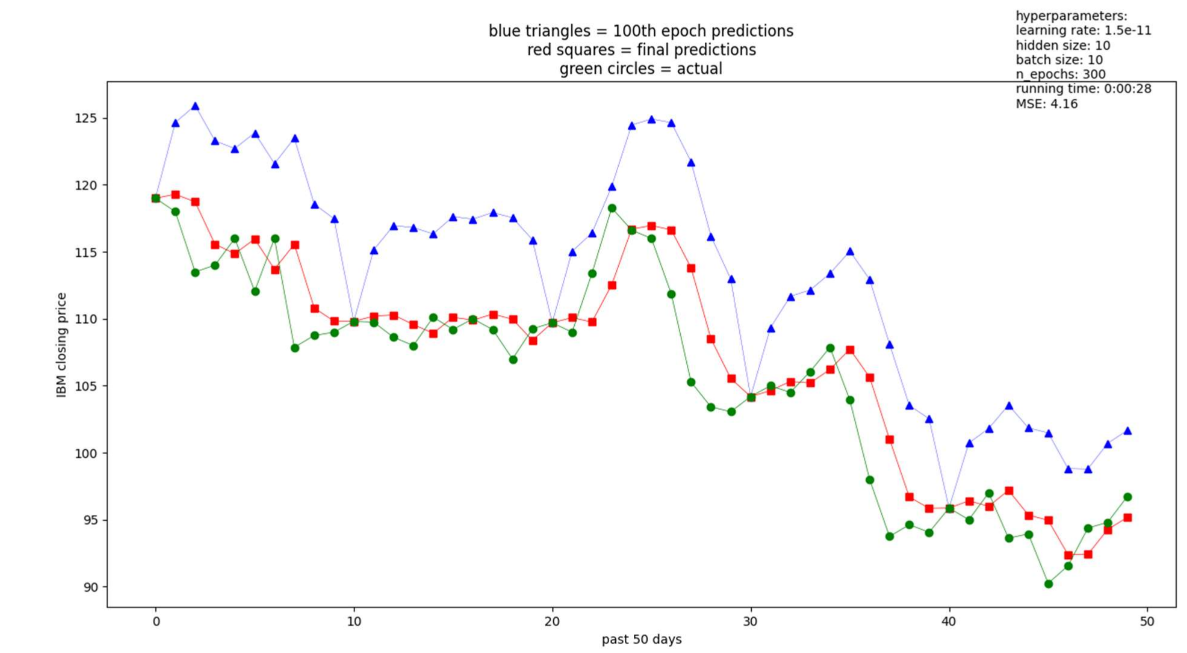
These are the main equations we need. Updating the input weights, recurrent weights, output weights, and the biases allows us to make predictions on a sequence such as a timeseries in our simple Jordan RNN with one hidden layer and one recurrent weight matrix.

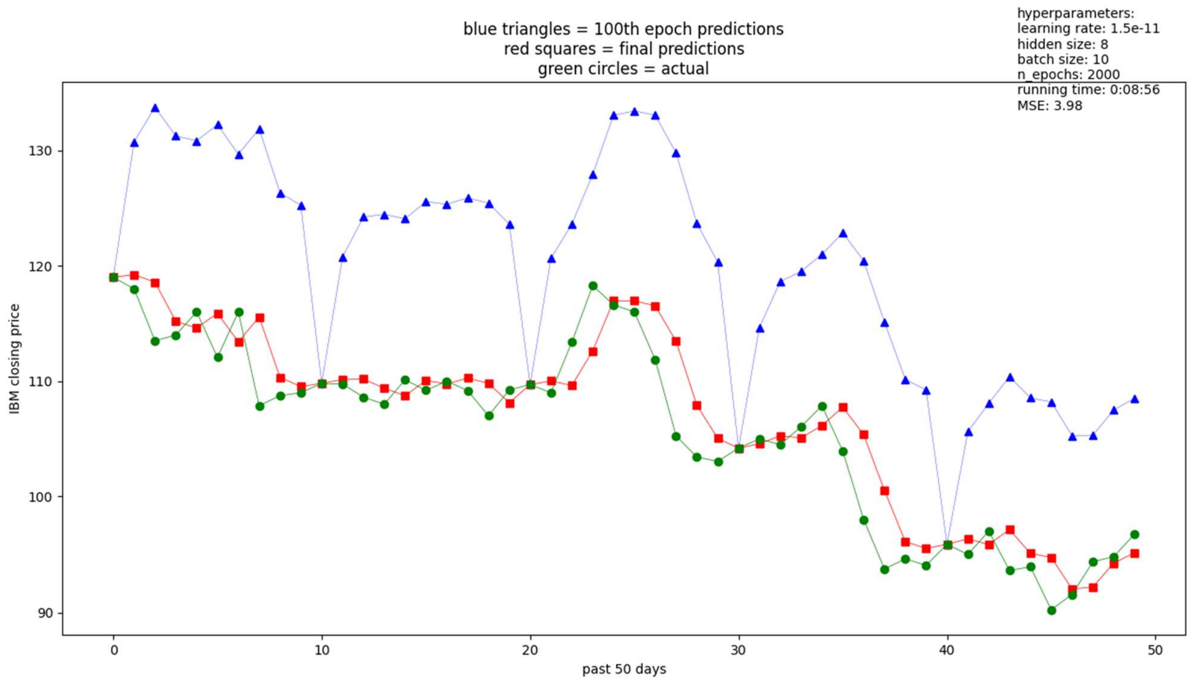
III. Results

The results of applying the Jordan RNN to a sequence of stock prices were nothing extraordinary, but they were encouraging and illustrate the power of even a simple version of the technique. The results are best described with the graphs created by the program itself using PyPlot. The figures below show decreasing error values as hyperparameters were tuned. The hyperparameter that seemed to have the most effect, somewhat unsurprisingly, was the number of epochs used. As the dataset was run through more and more times, the error trended downwards, until around 2,000 epochs. Using 3,000 or 4,000 epochs made little difference in accuracy but took 10-30 minutes per run.


blue triangles = 100th epoch predictions
red squares = final predictions
green circles = actual
hyperparameters:
learning rate: 3e-11
hidden size: 8
batch size: 10
n_epochs: 150
running time: 0:00:14
MSE: 15.58
IBM closing price
past 50 days
125
120
115
110
105
100
95
90
0
10
20
30
40
50


blue triangles = 100th epoch predictions
red squares = final predictions
green circles = actual
hyperparameters:
learning rate: 1e-11
hidden size: 10
batch size: 10
n_epochs: 300
running time: 0:00:28
MSE: 8.14
IBM closing price
past 50 days
140
130
120
110
100
90
0
10
20
30
40
50


blue triangles = 100th epoch predictions
red squares = final predictions
green circles = actual
hyperparameters:
learning rate: 1.5e-11
hidden size: 10
batch size: 10
n_epochs: 300
running time: 0:00:28
MSE: 4.16
IBM closing price
past 50 days
125
120
115
110
105
100
95
90
0
10
20
30
40
50


blue triangles = 100th epoch predictions
red squares = final predictions
green circles = actual
hyperparameters:
learning rate: 1.5e-11
hidden size: 8
batch size: 10
n_epochs: 2000
running time: 0:08:56
MSE: 3.98
IBM closing price
past 50 days
130
120
110
100
90
0
10
20
30
40
50

IV. Conclusion and Future Work

The results of this project were very promising and indicate that future work on this model with more advanced techniques could be even more effective. Potential future development could include experimentation with activation functions, especially ReLU, more hidden layers, longer recurrent memory, taking and predicting more features of stocks, upgrading to an LSTM model, implementing gradient checking to see if calculations need adjusting, and adding a meta-learning algorithm on top to refine hyperparameters automatically and precisely.

References

https://builtin.com/data-science/recurrent-neural-networks-and-lstm
https://en.wikipedia.org/wiki/Recurrent_neural_network