Time Series Forecasting with Convolutional Neural Networks

Gabriel Ronan
University of Michigan
gronan@umich.edu

1. Background

As the digital world grows, so do virtual worlds and virtual economies. Transactions involving virtual goods are becoming more common, and these transactions can have very 'real' economic consequences for those involved. In the context of online gaming, the largest and most popular virtual economies can be found in massively multiplayer online games (MMOs). Old School Runescape is one of the world's largest free MMOs, and its in-game currency has accrued higher stability and value than some real world currencies. The goal of this project is to design a tool to aid in navigating the expanding virtual economic landscape in general. Multi-step time series forecasting using a convolutional neural network (CNN) will be applied to predicting the in-game prices of Old School Runescape items. **Project code can be found at** https://github.com/ronangabriel/EECS-559-Final-Report. **The dataset used can be found at** https://drive.google.com/file/d/1RXagXgZjfKxzAfkuWOmzo5Q1UlwGt52w/view?usp=sharing.

2. Problem Definition

In Old School Runescape, most items are bought and sold from other players through an instrument called the Grand Exchange. An example of the price fluctuations and trade volume of an item, Dragon Bones, is shown in figures 1 and 2 [2] respectively. When creating a trade offer, the player must choose the volume of that item to buy or sell and the price. The initial price is set to a guide value that the player can adjust. The guide price is based on the trade history of the item and is intended to represent its approximate market value. While the game developers have never published their exact pricing algorithms, it is known that the guide prices are generally governed by the laws of supply and demand.

The Old School Runescape Wiki has released a public API that allows access to real time Grand Exchange prices [1]. The last 300 data points of price and volume for each item can be accessed with either 5 minute, 1 hour, or 6 hour

Figure 1. Three Month History of Average Daily Trade Prices for Dragon Bones, (1K = 1,000 coins)

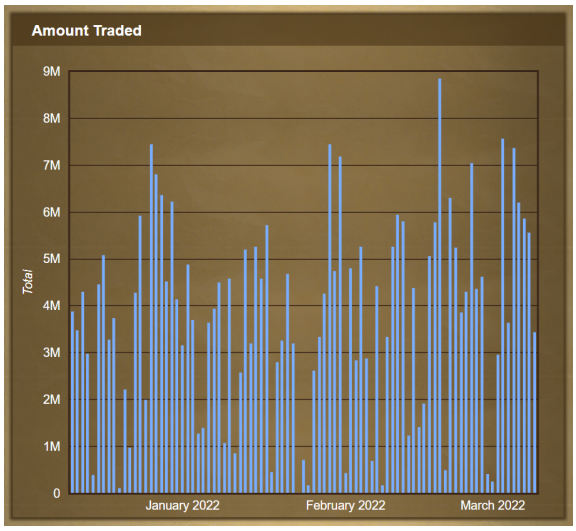

Figure 2. Three Month History of Daily Trade Volume for Dragon Bones, (1M = 1,000,000 units)

intervals between data points. With over 3500 distinct items actively traded, if the 5 minute data is chosen for analysis then it is possible to access over 1,000,000 new data points every day. In this report, we use the 6 hour data. We will use $n = 10$ points of prices to predict the next $k = 1, 2, 3$ points of prices as accurately as possible, and compare them to the results of a naive approach that represents baseline performance.

Figure 3. Pipeline for the machine learning model.

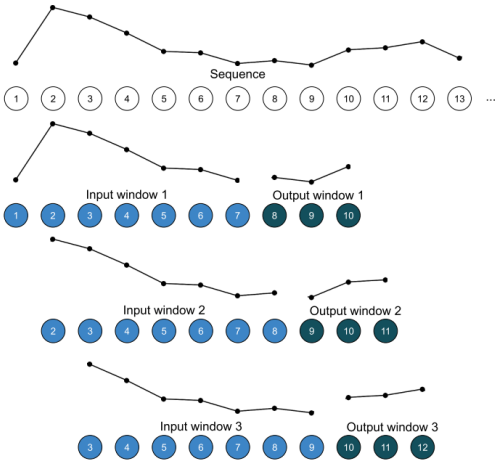

Figure 4. Example of moving window procedure that will be used to train the CNN. In the above case, $n = 7$ and $k = 3$.

3. Preprocessing

Convert to Average Price: The average price for each item is calculated at at each timestamp. The data for each item at each timestamp is composed of average high price, average low price, high price volume, and low price volume. There were too many inconsistencies in the data to create a separate feature vector for volume. For example, many data points contained null volume data but contained valid prices, and others contained null price data but contained valid volume data. To remedy these errors, we use average price at each time step as the primary feature, and use the valid volume data to calculate it when necessary according to equation 1. We now have a 3572 x 300 (items, timestamps) data array.

Fill/Delete Invalid Entries: All invalid (null) and zero values and eliminated. We replace these values with the last valid value seen in the past, because that value most accurately represents the current market price. If any of these null values are found at the very beginning of the series, then there is no value to replace it with, and so we delete all values for that timestamp to maintain the same number of data points in the time dimension for each item. No time steps needed to be deleted for the 6 hour data set.

Normalize and Split Data: All prices are normalized according to a min-max scaler (2). The data is then split in the time dimension into training and test sets using k-fold cross validation with $k_{cross} = 3$, allowing us to train and test on the entire data set after three iterations. The data is then split into input and output data that follows a fixed origin testing scheme in order to make predictions, as shown in figure 4 [3]. This is what makes this a multi-step time series problem; we are predicting k points into the future, where $n = 10$ and k is varied from 1 to 3.

$$avg(\mathbf{x}) = \frac{avgHigh(\mathbf{x}) \times highVol(\mathbf{x}) + avgLow(\mathbf{x}) \times lowVol(\mathbf{x})}{highVol(\mathbf{x}) + lowVol(\mathbf{x})} \tag{1}$$

$$min - max(\mathbf{x}) = \frac{\mathbf{x} - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})} \tag{2}$$

4. CNN Design

The CNN was implemented using Keras TensorFlow and is composed of six layers: a convolutional layer, a max pooling layer, three dense layers, and an output layer, as shown in figure 5. The convolutional layer creates a kernel that is convolved with the layer input over a the temporal dimension to produce a tensor of outputs. The convolutional layer has 128 output filters with a kernel size of 3 and ReLU activation. The max pooling layer downsamples the input representation by taking the maximum value over a spatial window. The max pooling layer has a pool size of 2. The dense layers connect every neuron in one layer to every neuron in the next layer and have output dimensions of 100, 50, and k, respectively, and have ReLU activation. The output layer has width k, and implements an L2 loss function (2), where $\hat{\mathbf{y}}$ is the approximated price vector and $\mathbf{y}$ is the true price vector. The CNN was trained for 5 epochs in all cases. All values not specified here were set to their defaults.

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2}||\hat{\mathbf{y}} - \mathbf{y}||_2^2 \tag{3}$$

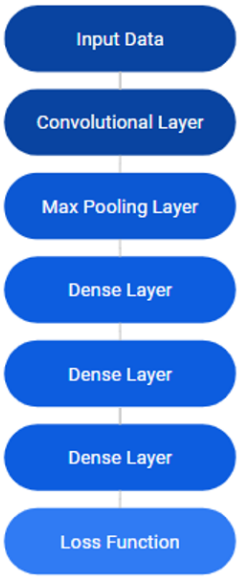

Figure 5. Pipeline for the architecture of the CNN.

5. Convergence Analysis

To explore convergence, we will compare different descent methods in the backpropagation steps of the CNN. These include vanilla stochastic gradient descent (SGD), stochastic gradient descent with momentum (SGD_m), Nesterov (accelerated) stochastic gradient descent ($ASGD$), accelerated stochastic gradient descent with momentum ($ASGD_m$), and adaptive moment estimation ($ADAM$). The learning rate for all methods was set to 0.01, and when momentum was used, it was set to 0.1. All other values we set as default. We measure the average runtime of a single epoch of training the CNN using each of the optimization methods. The results are shown in table 1. Of all the methods tested, vanilla SGD resulted in the best run time performance, which is surprising, because it was expected that ADAM would result in the best run time performance

Table 1. Runtime Using Descent Optimization Methods

Descent Method	Runtime (s)
SGD	33.8
SGD_m	37.4
$ASGD$	36.2
$ASGD_m$	36.4
$ADAM$	39.6

6. Results

A naive approximation of the next k prices is to assume that they are all equal to the last price seen, as this is how the average Runescape player will approximate the fair market price. This will serve as our preliminary benchmark to beat. The mean squared error is calculated for both the naive approximation and our prediction of the vector for the next k prices, as shown in table 1. Our results show that the CNN successfully predicts the next k prices with greater accuracy than the naive approximation, therefore the experiment is a success. In addition, for this particular 1D convolution problem, it was shown that vanilla stochastic gradient provided the fastest convergence accross multiple epochs in the training phase, and is likely the best optimizer to choose.

Table 2. Mean Squared Error of Test Predictions

Output Steps k	CNN MSE	Baseline MSE
1	0.0230046	0.0382060
2	0.0492707	0.0835825
3	0.0749539	0.127451

7. Use Cases: Item Selection for Profit

If we are only interested in picking a select number items that result in net profit, we can use our predictions to quickly cook up a naive method for buying items that are most likely to rise in price. We will restrict $k = 1$, the number of future time steps to predict. Predicted profits are calculated as the predicted price minus the last data point seen:

$$\hat{\mathbf{p}} = \hat{\mathbf{y}} - \mathbf{x}[-1] \tag{4}$$

.

The real profit is then the ground truth future price minus the last data point seen:

$$\mathbf{p} = \mathbf{y} - \mathbf{x}[-1] \tag{5}$$

Let the variable δ be some threshold that we decide. Since the data values are normalized, the value of $\hat{\mathbf{p}}$ will be a small value on the order of -1 to 1, but it could be outside this range. For all items at all time steps with predictions that satisfy $\hat{\mathbf{p}} > \delta$, how many of those also satisfy $\mathbf{p} > 0$? This method recommends which items will result in net profit based on which items have the highest predicted profit margin. The results are shown in table 3. For context, there are $953,724$ total samples in the test data for $k = 1$.

As we increase the threshold up to $\delta = 0.3$, we obtain more accurate predictions for items that increase in value. This shows that our tool is useful for players that want to make wise short term investments. The decrease in performance at the threshold value of $\delta = 0.4$ may be due to

Table 3. Predicting Items with Net Gain

Threshold	Predicted	Correct	% Correct
0	527573	331385	62.8
0.1	180856	142735	78.9
0.2	61209	53387	87.2
0.3	9583	8811	91.9
0.4	1456	1312	90.1

the highly volatile nature of items with large profit margins. This avenue could be further explored with variance metrics.

8. Future Work

The next logical step is to implement a trading scheme whose performance can be simulated historically on the data. The naive and greedy approach could involve buying and selling items at predetermined (constant) time intervals and then choosing items and item volumes that maximize the potential profit based on the predicted data, given some initial allowance. Since there are daily buying limits for each item on the exchange, the naive approach involves solving a fractional knapsack problem at every time step.

This is an area where reinforcement learning could shine. Instead of fixed time intervals, an actor could be trained to take actions that maximize rewards by choosing both the optimal prices to buy and sell as well as the optimal time steps to execute the trades, given some initial allowance.

It would also be worthwhile to exploit more powerful hardware, as I was only able to train the CNN for 5 epochs for every variation of parameters due to time constraints. Ideally we want to train for hundreds or even thousands of epochs to maximize performance. Another way to improve the current project is to predict prices for multiple parallel series simultaneously, as this is what most significantly slowed to process of obtaining results.

References

[1] Oldschool runescape wiki, runescape:real-time prices. https://oldschool.runescape.wiki/w/RuneScape:Real-time_Prices. 1

[2] Runescape grand exchange: Dragon bones. https://secure.runescape.com/m=itemdb_oldschool/Dragon+bones/viewitem?obj=536. 1

[3] P. Lara-Benítez, M. Carranza-García, and J. C. Riquelme. An experimental review on deep learning architectures for time series forecasting. *International Journal of Neural Systems*, 31(03):2130001, Feb 2021. 2