Predicting Bitcoin Prices with a Ruby Artifical Neural Network

Sara Bocabella, Ryan Kass

December 11, 2012

1 Introduction

Neural networks have been successfully used to model commodity pricing data and even to refute the Efficient Market Hypothesis through their ability to generalize commodity data into non-linear, stochastic functions previously thought to be entirely stochastic and non-generalizeable [1] The commodity we chose to test our netowrk on, Bitcoin, was established in 2009 and has experienced massive volatility in its price since its inception. We chose to use Bitcoin precicely for this reason, and the fact that it is an immature market not yet flooded with banks and finance firms controlling market movements.

For three months, we have been collecting hourly data on Bitcoin, recording ask volume, bid volume, last price, and the spread between the highest bid and the lowest asking price. These four features are the input to the network, which then outputs a guess of the price twenty-four hours after the input data. The network was able to learn to predict the price with an average error of 49¢, compared to an average twenty-four hour price fluctuation of 37¢.

2 Problem Definition

Can an artifical neural network constructed in ruby be trained to accurately predict the price in USD of a bitcoin twenty four hours in the future. The network is trained on approximately 3,000 examples. Each example contains four data points, last price, difference between highest bid and lowest ask, amount of bitcoins up for sale, and amount of bitcoins asked for. The output is a floating point which represents the predicted price twenty four hours from now. As a subtask, we set out to implement a highly flexible neural network that can be initialized with any number of inputs, hidden layers, nodes per hidden layer, and output nodes. This, along with a gui would allow for easier testing and presentation of the data and results we obtained.

[1] Ramon Lawrence, Using Neural Networks to Forecast Stock Market Prices

3 Method

The network we implement is feed-forward, and can act like a perceptron or have multiple layers, depending upon how it's initialized. We use backpropagation to adjust the weights. When the network is run, the input layer gets its input and propagates that input to the next level without feeding it through any transition function. The nodes on the higher layers then have weights associated with each of their inputs, as well as a transition function which is defined as $\frac{1}{1+e^{-x}}$. Finally, at the output layer there are weights associated with every input from the most upper hidden layer, or in the case of a perceptron, the input layer. Since we are trying to obtain a real valued output, the top layer is not fed through a transition function. At the outset, all weights are randomly initialized. At each example they are adjusted according to what the price was twenty-four hours after the time the data they analyzed was recorded. This is done through standard backpropagation. At the top layer, the error is the raw difference between what the price actually was and what it was predicted to be. Each node not in the top layer calculates its error in a composite manner. From each node j to which node i outputs synapse:

$$error_i \mathrel{+}= error_j * weight_{i,\ j} \quad .$$

This process of error calculating is propagated all the way back down to the first hidden layer (the input nodes have no associated weights). After each training iteration the errors are reset back down to zero. Next, the weights must be adjusted using

$$w = w + \Delta \text{ where } \Delta \text{ is defined as:}$$

$$\Delta = derivative * error * step * output$$

The step is a number between 0 and 1 that determines how fast the weights get adjusted. Setting the step too high results in the network getting stuck at a local maximum, while setting it toolow does not allow it to learn quickly enough given the number of examples. The output is whatever this node outputted to its synapses. In the case of the top most layer this will be boundless, but in the case of the hidden layers, this will necessarily be a number between 0 and 1, because it was fed through a transition function. The derivative is the summed weights and inputs fed into the derivative of the transition fucntion, which is $\frac{e^x}{(1+e^x)^2}$. This formula is applied to every node of every layer to adjust weights, with the exception of the input layer, as it has no associated weights.

4 System Architecture and Implementation

The data was obtained with a ruby script that accessed Mtgox.com every hour in order to pull down the information we used. Mtgox.com distributes API keys to users who want to datamine their site or who want to be able to execute trades without a browser. There is a ruby module that we used that handles all interactions with the Mtgox.com API. This information was then recorded into a Postgresql database every hour. We used Heroku to host the script and

the database, as well as some add-ons provided free by Heroku to monitor the functionality of our app. We then pruned our data so that the only examples we were using had appropriate feedback–that is, we cut out the last 24 examples since we did not have information on their future price. The ruby script that accesses the Mtgox.com API was executed as a cron task every hour.

We designed the neural network in ruby. It can be initialized with any reasonable parameters for the number of input nodes, hidden layers, nodes per hidden layer, output nodes, and learning rate. We did enforce, however, that it is fully connected, so all input nodes feed to all nodes at the next layer, and so on. $Layer_j$ strictly outputs synapse to $Layer_{j+1}$. Each layer in our network is represented by an array of nodes, so a network with five layers will have five elements in the array, with each element being an array of nodes in that particular layer. Each node then has a an array of nodes from which it receives input, and a corresponding array which determines the weight that is associated with each input node. All calculations remain internal to the node class, and the network class strictly is in charge of the control flow of the program.

We have a separate script devoted to parsing the data and getting it ready for input to the neural network. This file converts the data in the database into segregated input arrays and output arrays. Note that the output is an array, but in our case it only holds one element: the price tweny-four hours in the future. Assigning the output to an array gives the network the flexibility to output more than one value. There exists another script then, that trains the network. It does this by loading in the data, and repeatedly calling asking the initialized network to train itself on successive data. This script takes as input the ratio of training set to validation set given the fixed size of our data. When the network is finsihed training, the network's predictions are then computed using the weights it attained from training. At this point the weights are fixed and the network is no longer learning. These results are then compared against the actual observed results. We then square these differences and divide by n to get the average squared difference. As a reference point, we then calculate the squared difference attained by strictly guessing the current price.

Since there is some variability between identically intialized neural networks given that the weights are intialized randomly, we have also made a script that repeatedly trains, validates, and gets the squared differences for networks of identical parameters. This takes as input the parameters with which to build the network and the number of networks that it should train and validate with these parameters. It then outputs the best network, the squared difference attained by this network, as well as the squared difference attained by making a naive guess (guessing the current price).

When the network is initialized, random weights are assigned to every input of every node with values ranging from -1 to 1. At each iteration of training, the network takes in two arrays as input: the input data and the expected data. When calculating the derivative in the process of adjusting the weights we found that when the argument fed to the derivative function was greater than 709, it would output Nan. Since as x goes to infinity, the derivative function, approaches zero, we manually check is the derivative returns Nan, and if it does

we output zero.

5 Conclusions

Our neural network can learn to output a guess that is expected to be 49¢away from the actual price. Only 12¢away from the target value of 37¢, which is the naive guess. This value was attained through much testing and tweaking of the parameters of the neural network. Eventually we found the following to be the values which attained the best results:

Nodes Per Hidden Layer: 20

Hidden Layers: 2

Learning Rate: 0.9

As nodes per hidden layer increased, we found that average squared distance was asymptotically increasing to a value of .2 (given that hidden layers ¿ 1). At 20 nodes per hidden layer we can frequently obtain a value within .05 of the asymptote. While at times, on certain iterations this number is not achieved, running this network until we achieve such a value is quicker and just as reliable as adding nodes per hidden payer. When the amount of nodes per hidden layer increases past 20, the reliability in coming closer to this asymptote is not as noteable as the speed cost incurred by adding another node, especially when the amount of hidden layers ¿ 1.

The same can be observed regarding amount of hidden layers: any value less than 2 performs significantly worse, and any value greater than two performs negligibly better, but at a huge performance cost. The results we have presented here are juat the result of working with fixed features and number of inputs. A major accomplishment has been creating the framework that allows for very easy tweaking of they system, so that this network can be tested with totally different features, as well as different numbers of features. It can also be used for a totally different problem space, however given how close we came to breaking through the barrier of the naive prediction with no tweaking of the input features, we think that with more data and more intelligent feature selection this network can be used to model bitcoin prices more effectively than it does here, and possibly break through the barrier of naive prediction.

In commodities trading it's considered vital to have data on the relationship between your commodity of interest and many other commodities. Since we only use the USD as a comparison point for our commodity of interest, it's very possible that adding other currencies and examining the relationships that they all undergo would significantly improve the performance of the neural network. However, our pre-collected data only paid attention to the dollar, and so our model only pays attention to the dollar as well. Notwithstanding its obvious shortcomings, we think we have created a valuable tool in the very mutable neural network and were pleasantly surprised with the performance of the network in this problem domain, and think it's a great jumping off point for more

accurately modeling bitcoin price behavior.