# An Analysis of Various Time-Series Forecasting Techniques Applied to Cryptocurrency Trading

Alex Guerra

November 29, 2018

## Background

The problem of correctly predicting markets algorithmic-ally has long been a goal of traders globally. As a result, there have been a plethora of methodologies and practices developed to undertake this task. The aim of this paper is to continue to explore some of these ideas and improve upon them. The trading will be done entirely against the cryptocurrency market, a unique but relatively unexplored area of trading.

## Gradient Descent Algorithm for Approximating Prices

The goal of this model is to predict the next data point in a stable periodic market given previous prices. It is primitive, yet if a market is truly periodic, this model should produce better than random results.

If we have (for $\theta = (\theta_0, \ldots, \theta_3)$ and a scalar $x \in \mathbb{R}$ representing one timestep), our hypothesis function looks like:

$$h_{\theta(x)} = \theta_0 \sin(\theta_1 x + \theta_2) + \theta_3 + \theta_4 x$$

and our cost function look like (where $m \in \mathbb{Z}$ is the number of data points we evaluate this on)

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_\theta \left( x^{(i)} \right) - y^{(i)} \right)^2,$$

then gradient descent dictates that to update our $\theta_j$, we have (with $\alpha \in \mathbb{R}$ as the learning rate)

$$\theta_{j+1} \coloneqq \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

What remains to do is to calculate the partials for each $\theta_j$. Note that this is not a trivial task as our hypothesis is very much non-linear. We have

$$\frac{\partial}{\partial\theta_0}J(\theta)(x_j) = \frac{1}{m}\sum_{i=1}^{m}\left(h(x^{(i)}) - y^{(i)}\right)\sin\left(\theta_1 x^{(i)} + \theta_2\right)$$

$$\frac{\partial}{\partial\theta_1}J(\theta)(x_j) = \frac{1}{m}\sum_{i=1}^{m}\left(h(x^{(i)}) - y^{(i)}\right)\theta_0 x^{(i)}\cos\left(\theta_1 x^{(i)} + \theta_2\right)$$

$$\frac{\partial}{\partial\theta_2}J(\theta)(x_j) = \frac{1}{m}\sum_{i=1}^{m}\left(h(x^{(i)}) - y^{(i)}\right)\theta_0\cos\left(\theta_1 x^{(i)} + \theta_2\right)$$

$$\frac{\partial}{\partial\theta_3}J(\theta)(x_j) = \frac{1}{m}\sum_{i=1}^{m}\left(h(x^{(i)}) - y^{(i)}\right)$$

$$\frac{\partial}{\partial\theta_4}J(\theta)(x_j) = \frac{1}{m}\sum_{i=1}^{m}\left(h(x^{(i)}) - y^{(i)}\right)x^{(i)}$$

Thus, we have a method that will, for a sufficiently small learning rate, locally minimize the cost function for a given hypothesis (note that a global minimum is far from guaranteed due to the non-convexness of the hypothesis function). If we have succeeded, we will have successfully fit a sine function to the data, from which we can move on to the next step. We will use this model to predict the next $x$, and due to the periodic nature of the true data, in the long run we will correctly predict the price more than we incorrectly do.

One important assumption of the previous paragraph was the convexity of the function we are trying to minimize. Unfortunately, in our model, this assumption if not met. There are computationally expensive ways of being relatively certain that a global minimum has been reached; however, in a market setting where these models need to be trained in a couple seconds, this will not suffice. Thus, to account for that, we must find a heuristic way of starting in the right "convex pool" so that gradient descent can work as intended. What we do is use heuristics to take good guesses for $\theta_0$, $\theta_2$, $\theta_3$, and $\theta_4$, and then run multiple threads in parallel with different starting values of $\theta_1$ to converge in different convex pools, from which we select the one with the lowest error to maximize the probability that we have found a global minimum. This can be scaled well and can converge very quickly with proper implementation.

## Ridge Detection

In general, the markets are stable, and aforementioned techniques can trade for a profit, however, there are times when the price of a certain cryptocurrency hits a ridge, defined to be a sharp increase or decrease in price. In these scenarios, it is often not possible to assume that we can fit our data to a curve, for example, in the case of regression. We thus introduce a second trader acting in parallel to the Gradient descent algorithm, looking for ridges and taking control of trading when one is happening. The manner in which we detect ridges has started in a naive manner. We simply look at the mean and standard deviation of the last $n$ points, and then look to see if the $z$-score of the current price ($z_p$) is outside of some threshold centered around 0. If it is above this threshold, then we know that an upwards ridge is happening, and we trade to buy as much of the cryptocurrency as possible. The opposite happens for when the $z$-score is below this threshold. Mathematically we trade when

$$|z_p - \overline{p}| > c\sigma_p, \text{ where } \overline{p} = \frac{1}{N}\sum_{n=1}^{N} p_n \text{ and } \sigma = \sqrt{\frac{\sum_{n=1}^{N}(p_n - \overline{p})^2}{N-1}}.$$

The question then becomes a matter of tuning $c$ to the sensitivity with which we want to detect ridges. After some quick empirical testing, I found that setting this to around 2 produced close to optimal results. The

number of data points $N$ we look back also becomes a relatively important matter. Again, empirically it was determined that keeping this number relatively low (below 30 for 60 second updates) worked well. This makes sense as ridges are very local events not determined by any global patterns, so only very recent, local data should be used to determine whether or not a ridge is present.

## A neural network approach (Largely undeveloped, still reading)

Brainstorming thoughts: Recurrent neural networks might be useful for this task via the Long short-term memory algorithm (LSTM). This has seen success in some time-series forecasting related problems, but seems to not be very prevalent in the trading world as of yet. One potential advantage of these is that they can easily learn from multiple scales of time series data (seconds, minutes, hours, days, weeks, etc.) and determine long term and short term patterns for the data, from which we can trade appropriately. If implemented correctly, this could be a much more powerful method than above as it will have a more fundamental understanding of how the market operates and be able to predict trends that more naive models might not be able to. One potential con of this approach is that it would work exponentially better as we increase the variety of data it gets. The API we get data from does give us a fair variety of data at our disposal, but this may not be enough data for our neural network to learn something useful from. Another cool thing about LSTM networks is that they can be made to constantly adapt themselves for the changing environment, something potentially very useful in the unpredictable realm of the cryptocurrency markets.

Papers regarding using LSTM with time-series data:

1. Forecasting Economics and Financial Time Series: ARIMA vs. LSTM. This paper seems to compare LSTM to more standard techniques, and seems to be tied very closely to the problem I am trying to solve. The paper claims that LSTM's brought on a significant improvement in model performance, making it a very interesting read. This paper is nice as it walks one through LSTM networks.

   These networks are composed of LSTM cells with three components. First, there is a Forget Gate that outputs a number between 0 and 1, where 1 indicates that the corresponding data should not be forgotten while values close to 0 will be forgotten. Second, there is a memory gate, which chooses which new data should be stored in the cell. There are various acivation layers in this gate to determine which input values will be modified and creates candidates to be added to the cell. These activations are generally sigmoidal or tanh functions as they solve the exploding gradient problem that some RNN's of old were known to have. And last, there is an output gate, which simply outputs a values based on the current state of the cell with the newly filtered/added data. Multiple cells like this are involved in producing a sophisticated model that in theory can outperform some of the top algorithm based time-series forecasting models. They showed this with empirical evidence later in their paper.

   Interestingly enough, they also showed was that with LSTM models, they often need very little time to train before becoming effective. This is promising for our applications as we would like our LSTM to not only be constantly updating as new data comes in, but also be able to learn new market trends quickly. While this paper presented a relatively simple LSTM network, the complexity can scale up quickly if LSTM cells are combined in unique ways, such as combining cells with different time scales of data to further add dimension to the network.

2. A Deep Learning Approach for Forecasting Air Pollution in South Korea Using LSTM. This paper, while not at all related to financial forecasting, is still relevant due to the novel techniques used when applying LSTM to time-series data. In addition to the standard LSTM cells used, they employ an encoder-decoder model, which are used in machine comprehension problems to allow the computer to better understand the data as a whole.

   The LSTM cells themselves are essentially identical to the ones used in the first paper, but what

was especially interesting was how the higher level structure of the LSTM cells. Essentially, they layered the cells on top of one another to form an encoder which produces some mean output prediction, and then this output is attempted to be decoded back into inputs by the decoding layer, which is the unique aspect of this model. This very creative approach could be used to aggregate multiple timescales of data effectively into one mean predicted future price, which would be a very unique neural network. Not only that, but with this architecture I would imagine adding in more cells for different data would not be very difficult to do, so this approach should be relatively scaleable.

I quickly glanced at a few other papers, but these seem to be the most relevant and I will thus focus on these for the time being.

To help me better understand LSTM networks, I have also found the following videos/online articles:

1. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/
2. https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0
3. https://www.youtube.com/watch?v=WCUNPb-5EYI
4. https://deeplearning4j.org/docs/latest/deeplearning4j-nn-recurrent** This is a good article as it explains how to implement RNN's in Dl4J while also touching on how to implement different architectures for the LSTM cells, something that will certainly be useful in the future when we attain more complex models.
5. https://www.youtube.com/watch?v=6niqTuYFZLQ Good Stanford Lecture on RNN's with some more technical background and covering of LSTM's at a high level.

## Initial architecture of LSTM Network

Initially, we want the simplest possible architecture for our LSTM Network. What we did for this is simply synthesize some test data of oscillating states (discreet) and tried to get the RNN to learn the oscillating state. With a training input of $\{0, 1, 2, 1, 2, \ldots, 1, 2\}$, we when attempting to replicate this data over 100 epochs, we get pretty good results:

Result #1: `12121212121212121212` Result #2: `121212121212121212000`.

This is a good start initially, but there is still a lot to be done. When increasing the complexity of the data, where inputs can go to multiple things depending on multiple things in the past, the simple model struggles. For example, when we try to replicate the again oscillating input $\{0, 1, 2, 3, 4, 5, 4, 3, 2, 1, \ldots, 5, 4, 3\}$, we get the following results

Result #3: `24433223444222330000`

There is some oscillation here, but it certainly is not close to replicating the 1-5 pattern of our input data. So, one thing to fix would be to make this network be more of LSTM-based rather than RNN based. However, this is not the only thing that needs to be fixed with our model. For example, we do not want to replicate data, but instead predict based on the current state, so we must rewrite our program a tad to do this. Another thing that is unacceptable with this model is that the input and output vectors are discrete, whereas we want continuous if possible. We will also have to modify our program to handle this as well. Lastly, this is a very simplistic cell. In the example code of the DL4J repo there one class where they construct an encoder-decoder LSTM, something that would be very good not necessarily now as not a lot of data is being used, but definitely may be helpful down the road when more and higher complexity data is introduced to the model.