Predicting the Tomorrow of the Financial World with Time Series Prediction

Research Question: How can the value of a cryptocurrency be predicted with time series analysis?

Word count: 3134

Table of Contents

1. Introduction

Predicting the outcome of well-developed markets such as the stock market using mathematics, more specifically time series prediction, has already been researched extensively in the past, well before neural networks started to become commonplace. Cryptocurrency values also behave like stock markets, their values change in a complex and volatile manner; making the market a highly volatile and hard to predict thing where trends change very rapidly. By utilizing the power of the Universal Approximation Theorem and a gradient based learning error-correcting algorithm, I aim to demonstrate that a mathematical model can be built to predict the financial world throughout this essay. Thus, to justify my method for answering the research question, I will create and use a mathematics based neural network.

1.1 Feed Forward Neural Networks

At the most basic level, Artificial Neural Networks (ANNs) are computer systems that utilize a brain-like structure consisting of two components, the neurons (nodes where the processes take place) and the synapses (paths/connections via which data is transferred). There are input, hidden/processing, and output nodes. The input nodes are where data is entered, the processing (hidden) nodes create the hidden layers of the neural network where data processing takes place, and the output nodes show the decision made by the Neural Network. At every layer of the Neural Network, nodes transmit information to every node in the following layer. The connections between the nodes are called “weights” (also referred as synaptic weights). They are the factors optimized as the Neural Network works.

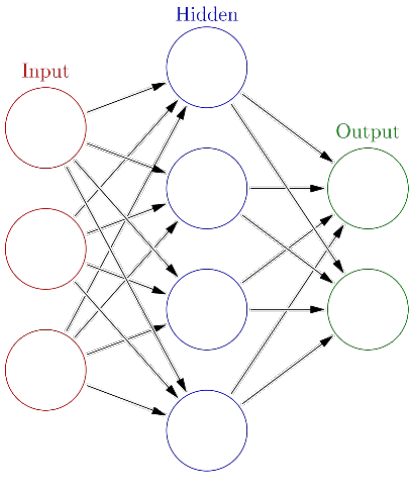


Figure 1; Example Neural Network

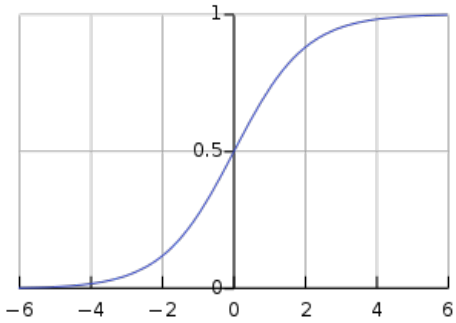


Figure 2; The Sigmoid Function

The simple Neural Network example in *Figure 1* consists of 3 layers; there is 1 input, 1 hidden and 1 output layer. The input layer has 3 nodes, the hidden layer has 4 nodes, and the output layer has 2 nodes. In addition to the nodes, layers may contain biases, which are constant values that can used to offset the layer output. However, in multi layered neural networks, all of the weight parameters connecting to a neuron may enable that neuron to function as a bias, if they gravitate towards zero; due to the range of value 0 in sigmoid function (*Figure2*), which is 0.5. Thus, in complex neural netwoks such as the one that will be created here to predict stock prices, hidden neuron layers are only used as checkpoints where the calculations or processes take place. The weights are the most crucial components in the system, they adjust the total effect an input has on the output (this will be further investigated in section ***3.3***).

1.2 Universal Approximation Theorem

When the standard mathematical prediction techniques such as the expected value used in probability and statistics are not applicable, Universal Approximation Theorem comes into play. The theorem states that given enough training data and computational

resources, a neural network with weighted connections (to be investigated in detail later) can approximate any function with normalized inputs and outputs (input in range [0,1] in R^m, and output in range [0,1] in R^n) perfectly, given that they are continuous, varying, and bounded. This theorem was proven by the George Cybenko in 1989 for sigmoid activation (*Figure 2*) (the equation of the sigmoid function will be given in section ***2.3***) functions and plays a huge role in the field of artificial intelligence.

To elaborate on the theorem, I will introduce the proof using the common names for the parameters: w_n for weight and b_n for bias, where the weight parameter straightens and bias horizontally moves the graph. Like that in the original proof, the sigmoid function will be used as non-linear activation function.

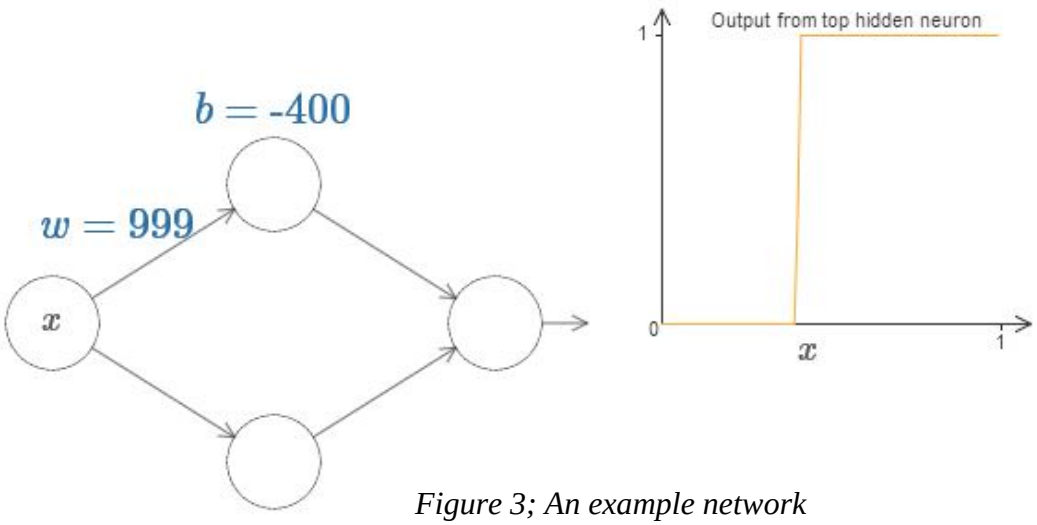


Figure 3; An example network

In Figure 3, by extensively increasing the *w* parameter an edgy *S* figure was created. Now, we will subtract two graphs after adding one more weight and one more bias (s parameter stands for b in *Figure 4* and *Figure 5*), then introduce w_1 and w_2, new synaptic weights. These new parameters will account for the weight of height addition.

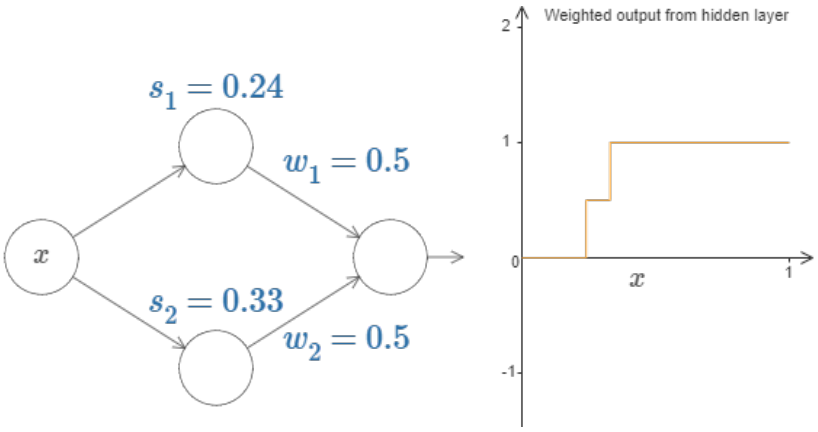


Figure 4; An example of addition

Their height, when added, reaches 1 in vertical axis

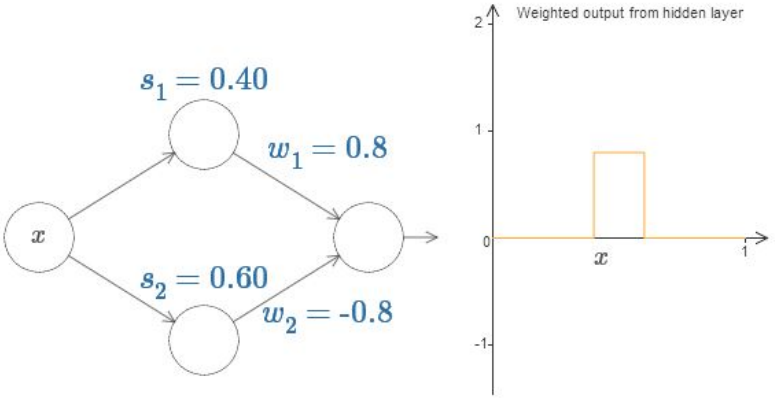


Figure 5; An example of substraction

And here w_1 was enough to make the graph reach *0.8* in vertical axis, while w_2 makes the value reach the ground state by adding a graph which has *-0.8* height weight. Now we formed a rectangle which is just enough to prove the theorem. By adding infinite layers and infinite neurons one can create any function using the Riemann summation method. So the magic is determining the weight parameters as bias difference converges to zero -so we did not us the biases, and that is made by *gradient descent.*

1.3 Gradient Descent

1.3.a Gradient Descent (Basic)

First, to be able to talk about such prediction techniques we need to introduce *the gradient descent* method. In basic gradient descent the goal is to to avoid the bad minima phenomenon (the maxima is not mentioned because the technique is vastly used to determine the minimum error value) which will be mentioned again (in topic ***3.2***), by slowly getting closer to the minimum. Graphic of such a method looks like this:

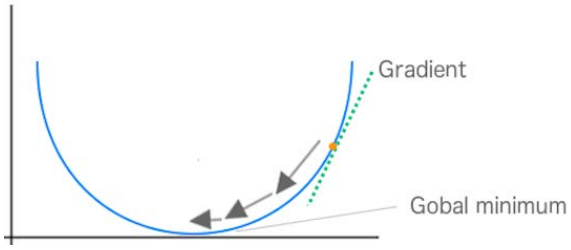


Figure 6; Basic Gradient Descent Example

The learning rate matters because when it is too high, the minimum may be missed and when it is too low, a bad minima can be reached.

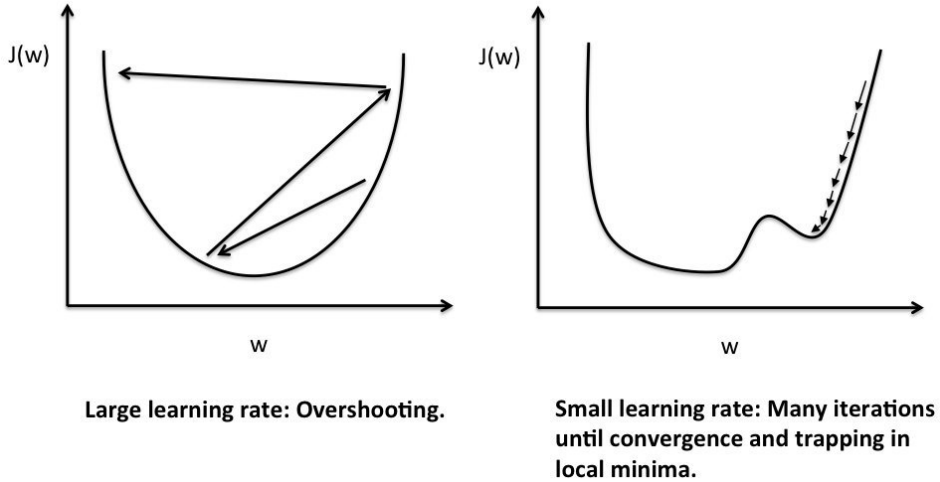


Figure 7; Gradient Descent Limitations

The solution here is starting with a high learning rate and gradually decreasing it. I will now introduce the formula and then explain it in order to clarify.

$$a_0 - \left(\left(\frac{\partial f(a_0)}{\partial a_0} \right) \times \omega \right) = a_1,$$

Where a_0 is a random number, accounted by supervisor (computer or the user), and, ω is the leaning rate. This should be applied to a_n until $a_k = a_{k+1}$, then it becomes safe to assume that a solution has been reached.

Now in order to clear vague points, further explanations about the process will be made. We calculate the derivation of the point a_n to find its slope and then multiply it our learning rate "ω". Learning rate adjusts the significance of the derivations while reaching the point a_{n+1}. Since the value of slope in a point near to minima will be less than a point far from it in a continuous function, it will take more steeps to travel the same distance so the learning rate also determines the speed of the algorithm by adjusting the number of steps to reach the outcome. Then we subtract the result of equation "$\left(\frac{\partial f(a_n)}{\partial a_n} \right) \times \omega$" from our the initial point "a_n" we reach our next point. When there are no changes in the outcomes of following steps ($a_k = a_{k+1} = a_{k+2}$) we can suggest that the minimum is reached because the derivation reached zero.

1.3.b Stochastic Gradient Descent

Normal/basic gradient descent is pretty useful when working with one variable, but it becomes problematic once multiple variables are introduced. Because almost always three or more dimensional graphs are used in computational sciences, basic gradient descent is not very popular, *the stochastic gradient descent* is usually preferred. The main difference between them, when two or more variables are introduced, is; the basic one changes both the parameters at the same step while stochastic do not. Below,

there is an example of how both algorithms work. Assume that we have two variables x and k, where possible values of k are determined as {1,2,3} ;

Basic Gradient Descent

$$x_0 - \left(\left(\frac{\partial f(x_0, 1)}{\partial x_0}\right) \times \omega\right) = x_1$$

$$x_1 - \left(\left(\frac{\partial f(x_1, 2)}{\partial x_1}\right) \times \omega\right) = x_2$$

$$x_2 - \left(\left(\frac{\partial f(x_2, 3)}{\partial x_2}\right) \times \omega\right) = x_3$$

Main problem here if any of the determined values -{1,2,3} in the example- is an outlier, the chain of calculations in the iterative process will greatly influence the result in a bad way. To avoid that, *stochastic gradient descent* is used, where the results do not have a direct effect like that in the basic one. To exemplify this, I will use *stochastic gradient descent* for the same scenario;

Stochastic Gradient Descent

$$\left(\frac{\partial f(x_0, 1)}{\partial x_0}\right) \times \omega = a_1$$

$$\left(\frac{\partial f(x_0, 2)}{\partial x_0}\right) \times \omega = a_2$$

$$\left(\frac{\partial f(x_0, 3)}{\partial x_0}\right) \times \omega = a_3$$

$$x_0 - (a_1 + a_2 + a_3) = x_1$$

In neural networks, such functions are used to determine and fix synaptic weights, where synaptic weight is the importance of the connection (this will be further discussed

in topics ***3.2*** and ***3.3***). Just like an $f(x)$ function can be used to do calculations, an error function is being used to calibrate the neural network (to be introduced in section ***2.2***), which enables us to find the deviation of our predictions.

2. Preparation of Input

2.1 Acquiring the Data

We will acquire our data from "*coinmarketcap.com*" [5] which averages the price of each coin on different markets. Our data contains the price of bitcoin for every 15 minutes between 2016-11-03 05:00 and 2017-12-07 22:30; this gives us exactly 38375 data points.

We will use the domain $\mathcal{F}$ to indicate this data where $\mathcal{F}_0$ is the price at 2016-11-03 05:00, $\mathcal{F}_1$ is the price at 2016-11-03 05:15 and so on.

2.2 Measurement of Error

We will be using root mean square error to calculate our errors throughout our experiments. $f(t)$ being the real values and the $\hat{f}(t)$ being our Predictions Root Mean Square error (rms) can be calculated as following over the domain A:

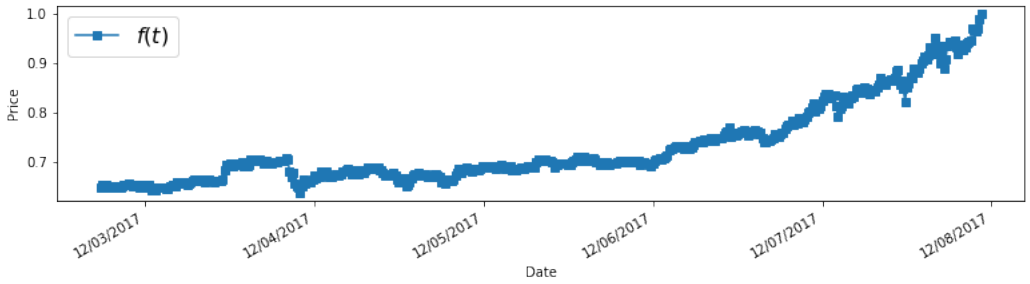
$$A_{rms} = \frac{1}{n(A)} \sum_{x \in A} \left(\hat{f}(x) - f(x)\right)^2$$

2.3 Normalization of Input Data

The method we will be using involves the Universal Approximation Theorem, so we need our input values to be normalized between 0-1. A simple min max normalization like shown below could be used:

$$\mathcal{F}_x := \frac{\mathcal{F}_x - \min \mathcal{F}}{\max \mathcal{F} - \min \mathcal{F}}$$

However, given that the price of cryptocurrencies make dramatic changes over months this means that we will be wasting most of our domain range on values that will never be relevant again.



Graph 1; Normalized data of 5 Months

An example to this problem can be shown on the graph above (*Graph 1*) which contains the normalized price information for bitcoin from dates 12/03/2017 to 12/08/2017. We can see that the values stay in the 0.7-1.0 range, this means that more precise weight changes will be required to find a working solution compared to a case where the values would always be spread throughout the domain.

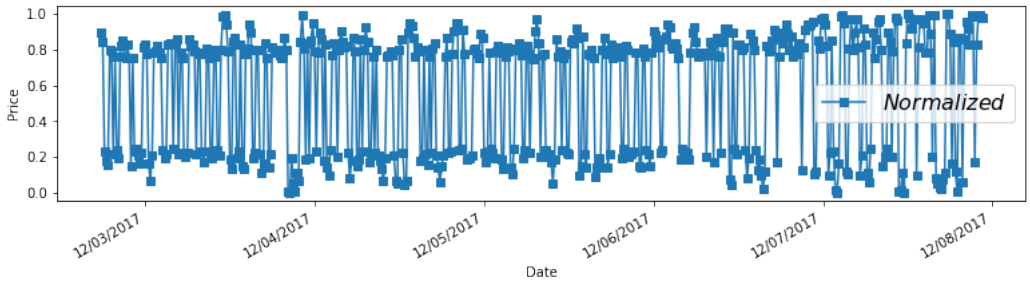
Because of this, we will instead be using a different method to normalize the input.

$$\mathcal{F}_x := \sigma\left(\frac{\mathcal{F}'_x - E(\mathcal{F}'_x)}{m}\right)$$

What we did in the equation above is; at first we calculated the numerical derivations of points depending on the *Graph1*. Then we substracted the our point derivation from average derivation value to find the derivation of our point. Then we devided our value to a big number "m" , such as 50, to avoid monotoncity wen it is applied to sigmoid function -defined below. At last we apply it to the sigmoid function becouse it's range is only defined in [0,1], which is essential utilizing the Universal Approximation Theorem

Sigmoid Function

$$\sigma(x) = \frac{1}{e^{-x} + 1}$$



Graph 2; Processed Normalization of data

As we can see, the values are clearly more spread over the domain this time and will stay relevant no matter how big the change in the price is due to the utilization of derivations.

3. Proposing an Optimal Model to Predict Trends

3.1 Diving into Universal Approximation Theorem

To apply Universal Approximation Theorem to our use case, we first need to build a feed forward network with weights.

A feed forward network can be basically thought of as a multi-variable function with normalized inputs and outputs with an arbitrary number of layers and units per layer. Each unit in these layers have "learned" weights for each unit in the previous layer. The value of a unit is equal to the weighted sum of the values from the previous layer, activated by an activation function, the sigmoid function.

The value of the n^{th} unit in the m^{th} layer, where $w_{(m-1)k \to mn}$ is the weight connection from k^{th} unit (neuron) of $(m-1)^{th}$ layer to n^{th} unit of the m^{th} layer, can be represented like the following (with the exception of the input layer which will be normalized input values):

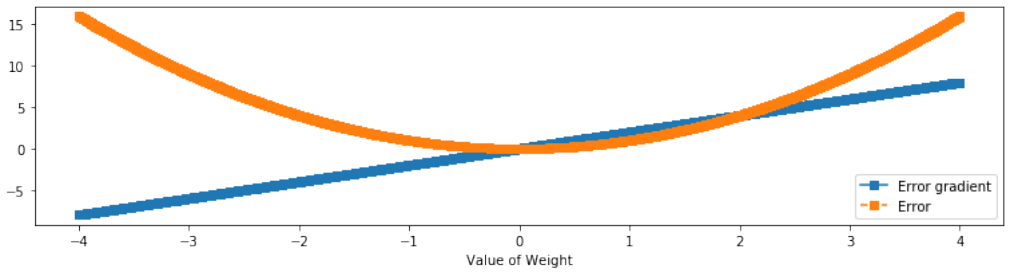
$$x_{mn} = \sigma\left(\sum_{k} w_{(m-1)k \to mn}\, x_{(m-1)k}\right)$$

The multi-layered nature of this function with weights which can be thought of as "importance" of the value for the next unit, gives the networks the ability to be universal approximators for any kind of function, which makes them an important tool in our toolkit with the aim of predicting the trends in a volatile and hard to predict market.

3.2 Learning from the Errors

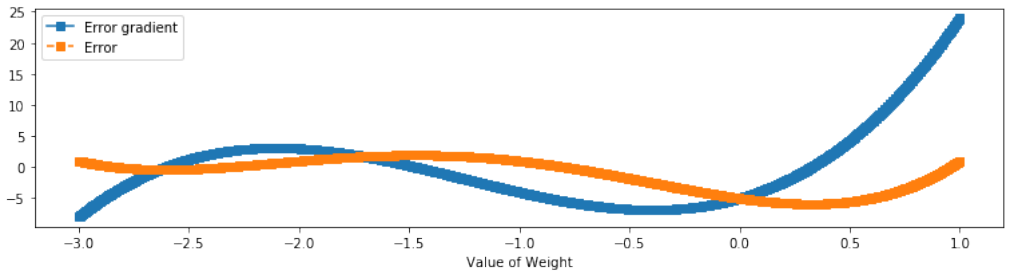
These networks however, are not magic 8-balls like one might be inclined to think after their description. The perfect approximation can be only achieved with the perfect values for the weights, which we have to somehow "learn" from our dataset.

This is where we let the network learn from its mistakes. The weights are initially set to random numbers between -1 and +1, and each data point in our dataset is fed into the function, error gets calculated and the weights get adjusted based on the gradient of error in relation to its value.



Graph 3; Error and Error Gradient

A simple method to achieve this can be proposed based on the example figure above (*Graph 3*) where the error is x^2s. We can see that the gradient of error, $2x$, gives us information about both the direction the value of x has to change towards and the magnitude of error.



Graph 4; Error and Error Gradient

Yet, a more complex error plane like the polynomial $x^4 + 5x^3 + 5x^2 - 5x - 6$, is considered, it is easy to notice the pitfalls of this method. The error gradient guides to the local minima and not the global minima, so the error rates might never reach perfect levels like we desire it to and we might get stuck on a "bad minima" simply because of our random starting point.

This effect is less pronounced when the fact that the error plane is N-dimensional is considered, however because of this we must repeat our experiments multiple times and pick weight matrices that produce the most optimal network.

Given the error function E, we will be updating the weights like shown below where ω stands for the "learning rate":

$$w_{xy} \coloneqq w_{xy} - \omega \frac{\partial E}{\partial w_{xy}}$$

We need the ω multiplier, a numerically small hyper-parameter, to make sure we do not make dramatic changes in weight values just on one data-point. This will be preventing the "memorization" of the input values and make sure our solution can be generalized to future input.

3.3 Learning from the Errors on Feedforward Networks

We can now combine both of these techniques to build a mathematical model that can "learn" to accurately approximate an arbitrary function. We use the error function we chose before to "judge" how accurate the prediction of the network was. This function will be represented as below where x stands for the input, W for the weight tensor and $\hat{f}$ for our approximator.

$$E(x, W) = \left(\hat{f}(x, W) - f(x)\right)^2$$

Just for the examples sake, let's assume it is a three-layered network, like the one we will be using, and that the v_n is the n[th] unit in the last layer, k_n is the n[th] unit in the middle layer, and i_n is the n[th] input.

Each unit in the network has an amount of "blame" they hold in error made. The gradient of the last layer is pretty straightforward:

$$\frac{\partial E}{\partial v_n} = 2(v_n - f_n(x))$$

As for the units in the middle layer this can be calculated like the following:

$$\frac{\partial E}{\partial k_n} = \sum_a \frac{\partial E}{\partial v_a}\frac{\partial v_a}{\partial k_n}$$

$$= \sum_a \frac{\partial E}{\partial v_a}\frac{\partial}{\partial k_n}\left[\sigma\left(\sum_m w_{km\to va}\, k_m\right)\right]$$

$$= \sum_a \frac{\partial E}{\partial v_a}\sigma'\left(\sum_m w_{km\to va}\, k_m\right) w_{kn\to va}$$

Later on we can apply the chain rule again to find the gradient for the weights like demonstrated below which we use to update the network.

$$\frac{\partial E}{\partial w_{ka\to vn}} = \frac{\partial E}{\partial v_n}\frac{\partial v_n}{\partial v_{ka_{vn}}}$$

$$= \frac{\partial E}{\partial v_n}\frac{\partial}{\partial w_{ka\to vn}}\left[\sigma\left(\sum_m w_{km\to vn}\, k_m\right)\right]$$

$$= \frac{\partial E}{\partial v_n}\sigma'\left(\sum_m w_{km\to vn}\, k_m\right) k_a$$

4. Methodology

4.1 (Computational) Experimentation Method

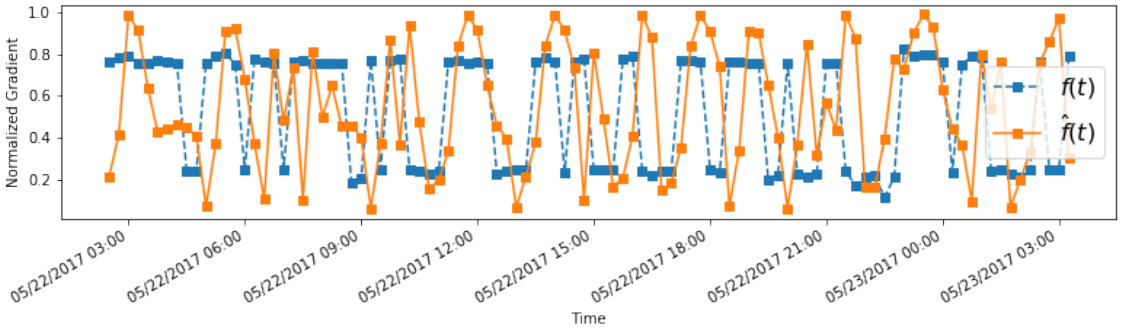
We used a 3-layer feed-forward network for our predictions with 10 units of input, 10 units in the middle layer and a single output. The input layer will contain normalized information about the price changes on last 5 data-points on the 15-minute, 3 on the daily and 2 on the monthly graph. This lets the network predict the changes both on short-term trends and on long-term ones.

The dataset was split into two equal pieces, a training and a testing dataset, to make sure the network is capable of generalizing the learned information into the future. ω was chosen as 0.0001. The training phase was done by going through the entire training dataset, updating the weights, shuffling the dataset and repeating this process 120 times.

To perform such computational actions, we used a programming language called python, which is the most popular language for creation of AIs. Additionally, to perform fundamental mathematical actions we used a library called NumPy (see Appendix for calculations).

4.2 Accuracy of the Results

The accumulated root-mean-square error hit 0 on the training dataset and 0.16 on the test dataset.



Graph 5; Comparison of Predictions and Real Values

As we can see from the figure above (*Graph 5*), the network is quite capable of predicting seemingly random trends in the price changes. Proving that our predictions are able to fulfil our aim determined by the research question, and thus, our method is an answer for the research question.

5. Conclusion

Our research is made for the prediction of bitcoin (BTC) however; such a model can be used for predicting any type of cryptocurrency. So, anyone wanting to calculate any type of currency, only by changing the input data, can use this feed forward network to predict the future value of desired currency. Furthermore, this model is not restricted by predicting the crypto-currencies. Anything, which can be expressed with an error function can be calculated/predicted with this model.

Main areas of neural networks (ANN) are image processing and forecasting. Image processing is mainly about recognition and identification of shapes, paintings, objects and handwritings. They are usually applied for security assessments, such as detection of fraud attempts. Forecasting, what we attempt to do in this essay, is being

used and required in daily business decision in stock market and rest of the financial world. For both of the areas, there are certain types of neural networks that are used more extensively.

What makes this search special is usage of this type of network to make time series prediction. Usually, Feed Forward Neural Networks (FFNNs) are used for tasks which requires processing visual inputs, such as facial recognition. Recurrent Neural Networks (RNNs) are the popular models for such prediction. Main difference between those relies under the input value. FFNNs use only the determined variable, as we utilized the value of BTC in an accounted time, while RNNs use both the determined variable and the outcome of the previous calculation. For example, to perform a prediction, a FFNN uses only the data of 2017-11-03 05:15 to predict the value of 2017-11-03 05:30, but a RNN would use the data of 2016-11-03 05:15 and all the predictions made, for the possible values, starting at 2016-11-03 05:15 (first prediction) to 2016-11-03 05:15 (last prediction to that time). Thus, by adapting the Feed Forward model to time series prediction, we are able to make market value predictions with insufficient data.

Although it is proven to be an effective model, in further researches it can be improved. Our model is based on the Universal Approximation Theorem as many others and as mentioned this theorem suggests that any graph can be modeled using infinite neurons and infinite layers. However, we used three layers and 10/10/1, neurons as mentioned at section *3.1*. So if the number of layers or neurons or both is increased then the output can be more accurate.

Bibliography

[1] I. K. a. M. Boydb, “Designing a neural network for forecasting financial and economic time series,” *Neurocomputing*, vol. X, no. 3, pp. 215-236, 1996.

[2] T. Kimoto, K. Asakawa, M. Yoda and M. Takeoka, “Stock market prediction system with modular neural networks,” in *IJCNN International Joint Conference on Neural Networks,* 1990.

[3] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of control, signals and systems*, vol. 2.4, pp. 303-314, 1989.

[4] Nielsen, and Michael A. “Neural Networks and Deep Learning.” Neural Networks and Deep Learning, Determination Press, 1 Jan. 1970, neuralnetworksanddeeplearning.com/chap4.html.

[5] “Bitcoin (BTC) Price, Charts, Market Cap, and Other Metrics.” *CoinMarketCap*, coinmarketcap.com/currencies/bitcoin/

Appendix

Computational experimentation, made in Python

07/03/2019 nnimplemented

In [664]:

```
import csv
import numpy as np
import matplotlib.cm as cmx
from matplotlib.collections import LineCollection
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.colors as colors
import matplotlib.dates as mdates
from scipy.interpolate import interp1d
import sys
import datetime as dt
from scipy.interpolate import lagrange
import random
```

In [665]:

```
# Neural network implementation
def sigmoid(x):
    i = np.exp(x)
    return i / (i + 1)

def dsigmoid(x):
    i = sigmoid(x)
    return i * (1 - i)

class Neuron:
    def __init__(self, prevLayerSize):
        self.inputs = []
        self.inValue = 0
        self.outValue = 0
        self.weights = 4 * (np.random.rand(prevLayerSize) * 2 - 1)
        self.weightGradients = np.zeros(prevLayerSize)

    def forward(self, inputValues):
        self.inputs = inputValues
        self.inValue = np.dot(inputValues, self.weights)
        self.outValue = sigmoid(self.inValue)
        return self.outValue

    def backward(self, gradient):
        gradient = gradient * dsigmoid(self.inValue)
        g = np.zeros(len(self.inputs))
        for i in range(0, len(self.inputs)):
            self.weightGradients[i] = self.weightGradients[i] + gradient * self.
inputs[i]
            g[i] = gradient * self.weights[i]
        return g

    def update(self, lr):
        self.weights = self.weights - lr * self.weightGradients
        self.weightGradients = np.zeros(len(self.weightGradients))

class NeuralNetworkLayer:
    def __init__(self, layerSize, prevLayerSize):
        self.neurons = []
        for i in range(0, layerSize):
            self.neurons.append(Neuron(prevLayerSize))

    def forward(self, inputValues):
        o = []
        for n in self.neurons:
            o.append(n.forward(inputValues))
        return o

    def backward(self, gradient):
        g = np.zeros(len(self.neurons[0].weights))
        i = 0
        for n in self.neurons:
            g = g + n.backward(gradient[i])
            i = i + 1
        return g


class NeuralNetwork:
    def __init__(self, layerDims):
```

```
        self.inputValue = []
        self.outputValue = []
        self.layers = []
        inputValue = [0] * layerDims[0]
        for i in range(1, len(layerDims)):
            self.layers.append(NeuralNetworkLayer(layerDims[i], layerDims[i-1]))

    def forward(self, inputValues):
        self.inputValues = inputValues
        tmp = inputValues
        for n in self.layers:
            tmp = n.forward(tmp)
        self.outputValue = tmp
        return tmp

    def backward(self, realValues):
        g = 2 * (np.array(self.outputValue) - realValues)
        err = np.dot((g / 2), (g / 2))
        for n in reversed(self.layers):
            g = n.backward(g)
        return err
    def update(self, lr):
        for n in self.layers:
            for k in n.neurons:
                k.update(lr)
```

In [666]:

```
# Reading the data
def read_csv(f):
    out = []
    with open(f, newline='') as csvfile:
        for row in csv.reader(csvfile, delimiter=',', quotechar='\"'):
            if row[0] == "timestamp":
                continue

            timestamp = dt.datetime.strptime(row[0], "%Y-%m-%d|%H:%M:%S")
            price = float(row[2])

            out.append([timestamp, price])
    return out

def sampleAndRescale(x, i):
    arr = np.array(x)
    i = int(i)
    arr = np.nanmean(np.pad(arr.astype(float), (0, i - arr.size%i), mode='consta
nt', constant_values=np.NaN).reshape(-1, i), axis=1)
    arr = np.repeat(arr, i)
    return arr

def normalize(d):
    yy = np.gradient(d)
    yy = yy - np.mean(yy)
    yy = sigmoid(np.abs(yy) / 200 * 4) * np.sign(yy)
    yy = yy / 2 + 0.5
    return yy

btc = read_csv("Data Bitfinex BTCUSD 15t.csv")

time = [ r[0] for r in btc ]
btc_pricen_15m = normalize([ r[1] for r in btc ])
btc_pricen_1d = sampleAndRescale(btc_pricen_15m, 96)
btc_pricen_1m = sampleAndRescale(btc_pricen_15m, 30 * 96)
```

In [676]:

```
# Create dataset
dataset = []

for i in range(5, len(btc_pricen_15m)):
    inp = [
        # Last 5 datapoints in 15m mark
        btc_pricen_15m[i-1],
        btc_pricen_15m[i-2],
        btc_pricen_15m[i-3],
        btc_pricen_15m[i-4],
        btc_pricen_15m[i-5],
        # Last 3 days
        btc_pricen_1d[i-1],
        btc_pricen_1d[i-2],
        btc_pricen_1d[i-3],
        # Last 2 months
        btc_pricen_1m[i-1],
        btc_pricen_1m[i-2],
    ]
    out = [ btc_pricen_15m[i] ]
    dataset.append([inp, out, time[i]])

training_dataset, test_dataset = np.split(np.array(dataset), 2)
```

In [670]:

```
# Create the network
network = NeuralNetwork([10, 10, 1])
```

In [673]:

```
for i in range(0, 10):
    e = 0
    random.shuffle(training_dataset)
    for d in training_dataset:
        network.forward(d[0])
        e = e + network.backward(d[1])
    network.update(0.0001)
    print("Epoch %d error rate = %f" % (i,e))
```

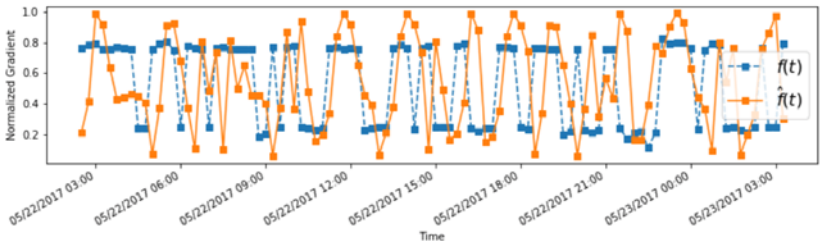
```
Epoch 0 error rate = 956.545420
Epoch 1 error rate = 833.463666
Epoch 2 error rate = 704.816439
Epoch 3 error rate = 552.235380
Epoch 4 error rate = 371.561135
Epoch 5 error rate = 195.632388
Epoch 6 error rate = 64.552376
Epoch 7 error rate = 11.307432
Epoch 8 error rate = 0.424007
Epoch 9 error rate = 0.009989
```

In [684]:

```
fig = plt.figure()
fig.set_size_inches(13, 3)
ax1 = fig.add_subplot(111)

plt.xlabel('Time')
plt.ylabel('Normalized Gradient')

x = [ r[2] for r in test_dataset ]
y = [ r[1] for r in test_dataset ]
yp = [ network.forward(r[0]) for r in test_dataset ]

lim = 100

x = x[:lim]
y = y[:lim]
yp = yp[:lim]


ax1.xaxis.set_major_formatter(mdates.DateFormatter('%m/%d/%Y %H:%M'))
fig.autofmt_xdate()

b1, = ax1.plot(x, y, 's--', label=r'$f(t)$')
b2, = ax1.plot(x, yp, 's-', label=r'$\^{f}(t)$')
plt.legend(handles=[b1, b2], prop={'size': 16})
```

Out[684]:

```
<matplotlib.legend.Legend at 0xb2ff83b00>
```



In [686]:

```
e = 0
for d in test_dataset:
    network.forward(d[0])
    e = e + network.backward(d[1])
network.update(0)
print("Epoch %d error rate = %f" % (i,e/len(test_dataset)))
```

```
Epoch 38374 error rate = 0.160936
```

In [ ]: