Applying Deep Learning to Better Predict Cryptocurrency Trends

Brandon Ly Divendra Timaul Aleksandr Lukanan Jeron Lau
Erik Steinmetz
Dept. of Mathematics, Statistics, and Computer Science
Augsburg University
2211 Riverside Avenue, Minneapolis, MN 55454
{lyb,timauld,lukanena,lauj,steinmee}@augsburg.edu

Abstract

In this paper, we will create deep learning models using the Python library, Keras, to make predictions on Bitcoin. We designed two neural networks to make similar predictions with important differences. In our first approach, we created a simple neural net with one input layer and an additional dropout layer to generate a continuous value. This continuous value is the predicted price of the Bitcoin a week from the given input. For the second model, we attempted to predict whether or not the price of the Bitcoin would go up 3%, stay within 3%, or go down 3% a week from a given date. Instead of returning a single continuous value, the result contains an array of three values. Each value is a percentage of the likelihood that it will either go up 3%, stay within 3%, or go down 3%. We approached designing the models using many different optimizers, activation functions, number of neurons, and various quantities of layers before settling on the final models. We also compared three different optimizers for each problem: Stochastic Gradient Descent (SGD), Adam, and RMSProp. We compared the results of all three optimizers to determine which one was the most effective at creating the most accurate model.

1 Introduction

With the rising popularity of numerous cryptocurrencies, generating accurate predictions would be of interest to many cryptocurrency investors. The efficacy of deep learning has become evident in its increasing application. Creating neural networks to accurately determine objects within an image is just one instance with incredible accuracy is just one instance. Most commonly, neural networks are used for classification problems, but predicting the price of the Bitcoin would be a regression problem. Not only are we using a relatively novel method to make predictions on the price of the Bitcoin, but the volatility of the price of the Bitcoin presents a problem for anyone trying to make a prediction. Given the precarious nature of making predictions on the Bitcoin or any cryptocurrency for that matter, a moderately accurate model would be a pleasant surprise.

2 Background

Machine learning, a subfield of Artificial Intelligence, has seen a rapid increase in use in a wide array of different fields. More specifically, deep learning has caught the interest of many researchers. Deep learning methodologies have seen a rise in prominence due to the efficacy in areas such as image recognition and natural language processing (NLP). With deep learning, the use of deep neural networks (DNN) has become the exemplary design methodology. Deep neural networks are made up of vertices connected by edges that take data, calculate a result based on the activation function, and then generate an output by taking the sum of the previous neuron's input and adding a predetermined bias [8]. The added complexity of the method allows for the modeling of complex, non-linear relationships [12]. The use of deep neural networks first became popularized as researchers designed a deep neural network that could excel at recognizing handwritten digits [5]. Continued success has been achieved with more complex and sophisticated images such as distinguishing between planes and birds [4].

More recently, deep neural networks have attracted the attention of researchers in the financial field to make predictions on financial markets [6]. Gathering insights from other similar models, we looked to create a classification method for the price of the Bitcoin. Given the nature of deep neural networks and the fact that it is, in a sense, a black box, we believed that it would make for an excellent research topic. With the aid of a powerful GTX 1080Ti GPU, we hoped to be able to quickly generate models and test results.

3 Methodology

We implemented two types of prediction models, each with their own deep neural network. The first model, after being given five consecutive days worth of Bitcoin prices, predicts a Bitcoin price at a point seven day in the future. The second model predicts the probability of the Bitcoin price going up 3%, stay within 3%, or going down 3% after the seven-day interval. Each neural network includes various amounts of hidden layers to predict prices

based on five consecutive days of the closing price of the Bitcoin. We are aiming to make a price prediction for seven days into the future from the last consecutive day. For example, if we provide as input into the models, the Bitcoin prices for the dates March first through fifth, then the first model would output the predicted Bitcoin price for March twelfth. On the other hand, the second model would output the probability of the Bitcoin price for March twelfth going up 3%, staying within 3%, or going down 3%.

3.1 Optimizer Functions

In the experiments, we tested each model and compiled it with different optimizers. Each optimizer updates the weight and bias values after each epoch to reduce the loss of the models. The loss is defined by some function that computes the difference between the predicted versus the actual.

3.1.1 Stochastic Gradient Descent

The first optimizer we looked at was Stochastic Gradient Descent (SGD). SGD works by updating the weights by looking at the gradient of each individual training sample and then updating the weights. This differs from a regular Gradient Descent because it does not update the weights by looking at the whole gradient which allows for faster convergence [1].

3.1.2 RMSProp

Resilient Mean-Squared Prop (RMSProp) works similarly to SGD except that it takes the average recent the magnitude and divides the gradient by it [7]. This works well for calculating repetitive data sets.

3.1.3 Adam

The Adam optimizer, like RMSProp, is based on SGD. It actually incorporates some elements from RMSProp and combines them with the AdaGrad algorithm [2]. This is designed to work well with sparse gradients and noisy inputs.

3.2 Activation Functions

3.2.1 Rectified Linear Unit

Rectified linear unit, or ReLU for short, is an incredibly popular activation function for artificial neural networks. ReLU is actually quite a simple algorithm, this function is simply $f(x) = max(0, x)$ [9].

3.2.2 Softmax

The Softmax activation function, or normalized exponential function, takes an N-dimensional vector and then normalizes it. All values of the vector are now between 0 and 1. The sum

of all the vectors always adds up to 1. This works very well for classifiers as the results can be interpreted as a percentage.

3.2.3 Sigmoid

The Sigmoid activation function takes whatever input you give it and generates an output in between 0 and 1. This differs from Softmax because if the input is a vector, it does not necessarily mean that all values will add up to 0 and 1.

3.3 Deep Neural Network Structure

3.3.1 Price Prediction

```
model1 = Sequential()
model1.add(Dense(1,input_shape=(5,)))
model1.add(Activation('relu'))
```

Figure 1: Price Model Code

For the price prediction model, the neural network contained 5 input neurons. The next layer is an output layer that contains a single neuron with a ReLU activation. All five input neurons are connected to the output neuron. The output from this network is a floating-point value that represents the predicted price of the Bitcoin a week from the last consecutive day from the input.

To compile the model, we found that using a mean squared error (MSE) loss function generated large amounts of loss per epoch. This is most likely do to the nature of the MSE loss function: $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$. Given that the predicted amount can differ from the actual by a factor of up to a thousand or more, taking the square of this and then averaging over the number of training samples results in an uninformatively large value. Instead, we found that using the mean squared logarithmic error (MSLE) to be more useful: $MSLE = \frac{1}{n}\sum_{i=1}^{n}(\log_{10} y_i - \log_{10} \tilde{y}_i)^2$. By using the squares of the log of the error, this metric was much more informative and so was chosen for these experiments.

3.3.2 Classification Prediction

```
model2 = Sequential()

model2.add(Dense(48,input_dim=5))
model2.add(Activation('sigmoid'))
model2.add(Dropout(0.01))
model2.add(Dense(48, activation='relu'))
model2.add(Dense(24, activation='relu'))
model2.add(Dense(3, activation='softmax'))
```

Figure 2: Classification Model Code

For the classification prediction model, we the input layer is the same as before: 5 input neurons. This time, though, we used 4 hidden layers. The first hidden layer contains 48

neurons with a Sigmoid activation function that are densely connected to the input layer. The second hidden layer is a dropout layer with a dropout rate of 1%. The dropout layer has been shown to effective as it prevents from overfitting the models to the training data by randomly treating some input as 0 [11]. After that, the fourth hidden layer contains 48 neurons with a ReLU activation function. The last hidden layer contains 24 neurons with a ReLU activation as well. The output layer contains 3 neurons with a softmax activation function.

To compile the model, we used the categorical cross entropy loss function. Categorical cross entropy is the loss function you want to use with softmax because it computes the loss based on each category. In our case, we have three categories. For the metric, we used the accuracy metric built into Keras which, in our scenario, uses the categorical accuracy metric. This metric, like the categorical cross entropy, is best used with outputs that contain more than two categories. It calculates the result by finding the largest percentage from the prediction and then compares it to the actual result. If the the largest percentage matches the index of the 1, then the measured accuracy increases. If it does not match, the accuracy goes down. The result is essentially: $(numberCorrect)/total$.

4 Experimental Setup

We used the Keras library [3] built on top of the TensorFlow library released by Google.

For our setup, we used a desktop computer equipped with a GTX 1080 Ti GPU. This setup with a powerful GPU allows for rapid model compilation and training the model with the data. To devise the models, we used the popular deep learning library, Keras, to develop the models and we used Tensorflow in the backend. To handle the data, we used the Numpy and Pandas libraries to import and generate our test data. In Keras, the data must be stored in Numpy arrays.

Since the data was imported using Pandas which uses its own data structure called a data frame, we then had to convert them into Numpy arrays. To visualize the data, we used Matplotlib and Seaborn. Seaborn works with Matplotlib to give enhanced graphing features. To develop the code for the models, we used the very popular python library called Jupyter Notebook. It allows users to section portions of code so that large projects can be worked on in pieces. This allows for rapidly testing many aspects in isolation such as the data modification or running the model for example.

The data itself is stored in .csv format. The CSV file contains 1748 rows and the only column we will be concerned with is the one containing the closing prices. The first recorded date is 2013-04-28, and the last recorded date is 2018-02-05. We generated a plot of the data points to get a visual impression of what we are working with as shown in Figure 3.

As you can see, the data hovers below $2500 for much of the history and then rapidly increases more recently. We can also notice a dramatic drop in the price at the end of our

Figure 3: Bitcoin Prices

data which could be difficult for the models to interpret.

Now that we have data, we can build the layers for our neural net.

4.1 Price Prediction Setup

To develop the model, we first need to gather the define the input and output data. After gathering the data, we then we compile our model, and then train the model with the training data. To gather the data, we first read the data from the csv file. To gather more data points for our model, we allow for overlapping dates in the input; in other words, if our first vector contains these dates: 3/1/17, 3/2/17, 3/3/17, 3/4/17, 3/5/17, the next input vector would be as follows: 3/2/17, 3/3/17, 3/4/17, 3/5/17, 3/6/17.

We then get the y-value, which is the date we are looking to predict, by adding seven to the last input date. We stop loading the training data 200 days before our last data point to use the last 200 days as our evaluation data to test our model with. Sample input data is shown in Figure 4, while sample output data is shown in Figure 5.

```
[[ 134.21  144.54  139.    116.99  105.21]
 [ 144.54  139.    116.99  105.21   97.75]
 [ 139.    116.99  105.21   97.75  112.5 ]
 ...
 [2608.56 2518.66 2571.34 2518.44 2372.56]
 [2518.66 2571.34 2518.44 2372.56 2337.79]
 [2571.34 2518.44 2372.56 2337.79 2398.84]]
```

Figure 4: Sample Price Input Data

```
[ 112.67  117.2   115.24 ... 2228.41 2318.88 2273.43]
```

Figure 5: Sample Price Output Data

4.2 Classification Prediction Setup

The process of setting up the model is similar to the regression problem, in fact, the input data is the same, but there are a few more steps required to generate the expected values.

Processing the data in this case is a bit more involved as it requires additional modification. We want to input vectors with five columns that represent closing price of the Bitcoin on five consecutive days. For the output, we computed the percentage of the Bitcoin by taking the last day from our input (this represents the day we are looking to predict seven days into the future from) and then subtracting that from the value of the day we are predicting and then divided it by the predicted days value. Then, based on the percentage we calculated, we then created a Numpy array to represent where it falls into the three categories. If it increases by 3% or more, then the output will be a vector [1 0 0], if it stays within 3% the vector will be [0 1 0], and if it decreases by 3% or more then the vector will be [0 0 1].

5 Results

5.1 Price Results

For the price prediction model, we found that we obtained the best results using 10 epochs. Anything after that, the loss stayed about the same and we also did not want to overfit the model as well. The loss is computed using the Mean Squared Logarithmic Error.

5.1.1 Training Results

For the first model, we used Stochastic Gradient Descent with a learning rate (α) of 0.01. We were able to achieve a loss of 0.0135 using this optimizer as shown in Figure 6.

```
Epoch 1/10
1533/1533 [==============================] - 0s 231us/step - loss: 0.0507 - mean_squared_logarithmic_error: 0.0507
Epoch 2/10
1533/1533 [==============================] - 0s 76us/step - loss: 0.0136 - mean_squared_logarithmic_error: 0.0136
Epoch 3/10
1533/1533 [==============================] - 0s 79us/step - loss: 0.0136 - mean_squared_logarithmic_error: 0.0136
Epoch 4/10
1533/1533 [==============================] - 0s 81us/step - loss: 0.0135 - mean_squared_logarithmic_error: 0.0135
Epoch 5/10
1533/1533 [==============================] - 0s 80us/step - loss: 0.0135 - mean_squared_logarithmic_error: 0.0135
Epoch 6/10
1533/1533 [==============================] - 0s 98us/step - loss: 0.0136 - mean_squared_logarithmic_error: 0.0136
Epoch 7/10
1533/1533 [==============================] - 0s 126us/step - loss: 0.0136 - mean_squared_logarithmic_error: 0.0136
Epoch 8/10
1533/1533 [==============================] - 0s 92us/step - loss: 0.0135 - mean_squared_logarithmic_error: 0.0135
Epoch 9/10
1533/1533 [==============================] - 0s 91us/step - loss: 0.0136 - mean_squared_logarithmic_error: 0.0136
Epoch 10/10
1533/1533 [==============================] - 0s 88us/step - loss: 0.0135 - mean_squared_logarithmic_error: 0.0135
```

Figure 6: Pricing Stochastic Gradient Descent Results

For the second model, we used the RMSProp optimizer. With this model, we were able to obtain a loss of 0.0160 as shown in Figure 7.

For the last model, we used the Adam optimizer. With this model, we were able to achieve a minimum loss of .0189 as shown in Figure 8.

```
Epoch 1/10
1533/1533 [==============================] - 0s 246us/step - loss: 0.0686 - mean_squared_logarithmic_error: 0.0686
Epoch 2/10
1533/1533 [==============================] - 0s 86us/step - loss: 0.0186 - mean_squared_logarithmic_error: 0.0186
Epoch 3/10
1533/1533 [==============================] - 0s 86us/step - loss: 0.0163 - mean_squared_logarithmic_error: 0.0163
Epoch 4/10
1533/1533 [==============================] - 0s 85us/step - loss: 0.0163 - mean_squared_logarithmic_error: 0.0163
Epoch 5/10
1533/1533 [==============================] - 0s 94us/step - loss: 0.0162 - mean_squared_logarithmic_error: 0.0162
Epoch 6/10
1533/1533 [==============================] - 0s 89us/step - loss: 0.0162 - mean_squared_logarithmic_error: 0.0162
Epoch 7/10
1533/1533 [==============================] - 0s 122us/step - loss: 0.0161 - mean_squared_logarithmic_error: 0.0161
Epoch 8/10
1533/1533 [==============================] - 0s 105us/step - loss: 0.0161 - mean_squared_logarithmic_error: 0.0161
Epoch 9/10
1533/1533 [==============================] - 0s 92us/step - loss: 0.0161 - mean_squared_logarithmic_error: 0.0161
Epoch 10/10
1533/1533 [==============================] - 0s 105us/step - loss: 0.0160 - mean_squared_logarithmic_error: 0.0160
```

Figure 7: Pricing RMSProp Results

```
Epoch 1/10
1533/1533 [==============================] - 0s 306us/step - loss: 0.2186 - mean_squared_logarithmic_error: 0.2186
Epoch 2/10
1533/1533 [==============================] - 0s 101us/step - loss: 0.0655 - mean_squared_logarithmic_error: 0.0655
Epoch 3/10
1533/1533 [==============================] - 0s 112us/step - loss: 0.0304 - mean_squared_logarithmic_error: 0.0304
Epoch 4/10
1533/1533 [==============================] - 0s 157us/step - loss: 0.0216 - mean_squared_logarithmic_error: 0.0216
Epoch 5/10
1533/1533 [==============================] - 0s 97us/step - loss: 0.0195 - mean_squared_logarithmic_error: 0.0195
Epoch 6/10
1533/1533 [==============================] - 0s 97us/step - loss: 0.0190 - mean_squared_logarithmic_error: 0.0190
Epoch 7/10
1533/1533 [==============================] - 0s 98us/step - loss: 0.0189 - mean_squared_logarithmic_error: 0.0189
Epoch 8/10
1533/1533 [==============================] - 0s 101us/step - loss: 0.0189 - mean_squared_logarithmic_error: 0.0189
Epoch 9/10
1533/1533 [==============================] - 0s 98us/step - loss: 0.0189 - mean_squared_logarithmic_error: 0.0189
Epoch 10/10
1533/1533 [==============================] - 0s 100us/step - loss: 0.0189 - mean_squared_logarithmic_error: 0.0189
```

Figure 8: Pricing Adam Results

To our surprise, testing the model against the evaluation data, we found that the model with the lowest loss in training did not have the lowest loss from the evaluation. The model that used SGD achieved a loss of 0.0135, but the loss computed from the evaluation set achieved a loss of approximately 0.02809. The second model, which used RMSProp, had the second lowest loss during training, but had the highest during evaluation. All models fared considerably worse during evaluation. This could be the result of the precarious nature of the price of the Bitcoin. If we refer back to the graph of the data, we can also see that the evaluation data set contained the more volatile portion of the data.

5.2 Classification Results

For training the classification models, we found that using 5 epochs was the optimal approach as, for whatever reason, the models losses would tend to increase after the 5th epoch; if trained with 10 epochs, the loss of the last epoch would double compared to the 5th epoch.

5.2.1 Training Results

For the first model, the optimizer being utilized is RMSProp. We obtained a minimum loss of 1.0838 and an accuracy of 41.88% as shown in Figure 9.

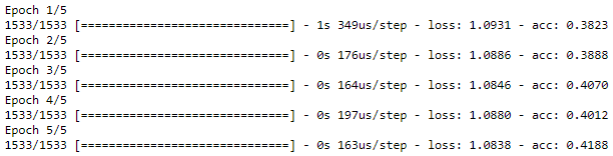

Figure 9: Classification RMSProp Results

For the second model, the optimizer being utilized is Adam. We obtained a loss value of 1.0841 and an accuracy of 40.83as shown in Figure 10.

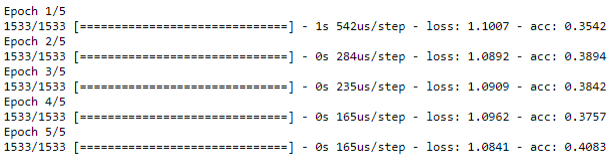

Figure 10: Classification Adam Results

For the third model, the optimizer being utilized is Stochastic Gradient Descent with a learning rate (α), of 0.01. We obtained a loss value of 1.0962 and an accuracy of 36.99% as shown in Figure 11.

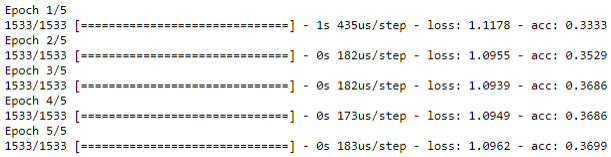

Figure 11: Classification Stochastic Gradient Descent Results

5.2.2 Evaluation Results

The results, again, were confounding, but not in the same way as the price prediction models results. The results of evaluation data were fairly close to the results of the training data. This leads us to believe that this model is more robust, and can manage to make predictions with less familiar data.

What was the most surprising result, though, was the result using the SGD optimizer. During training, the model that used SGD performed the worst with a loss of 1.0962 and an accuracy of 36.99%, but being run against the validation set, it achieved a loss that was less than what was achieved during training. Not only did it have a loss lower than it did during training, it also had the lowest loss value of all the models.

What we suspect could be the cause of this is that the original two models might have

slightly been overfit. Furthermore, any more epochs would most likely negatively affect the performance of the model when ran against the validation set. Changes to the model would be necessary to produce different results as running more epochs would exacerbate the overfitting dilemma.

6 Conclusions and Future Work

The goal of this research was to see if it was at all possible to create functional deep neural networks that could generate any kind of result. Fortunately, we exceeded our expectations by creating two working models that take two unique approaches to generate a solution. We were able to successfully create a model to solve a classification problem, and we created a model that could generate predictions for a regression problem as well.

Given the wildly unpredictable nature of the price of the Bitcoin, creating an evenly remotely accurate model was what we were aiming for. Although the results arent perfectly accurate, they do have predictive capability. In a related research paper conducted at the School of Computing at the Nation College of Ireland obtained results that were 52% accurate [10]. Their classification model was just looking at whether or not the price of the Bitcoin will increase or decrease. Given that they used far more complex models such as LSTM and RNN models, our results in comparison are quite impressive.

With the nature of deep neural networks and their generality, there are a countless number of applications for similar models in a wide variety of areas. Applications in virtual environments seems like another topic of interest. Applying deep neural networks in simulated game worlds would allow for a near endless amount of research topics and experiments. We believe that the future is incredibly bright for this technology and we hope to conduct further research in the field.

References

[1] BOTTOU, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[2] BROWNLEE, J. Gentle introduction to the adam optimization algorithm for deep learning, Jul 2017.

[3] CHOLLET, F., ET AL. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k 7* (2015), 8.

[4] CIREGAN, D., MEIER, U., AND SCHMIDHUBER, J. Multi-column deep neural networks for image classification. In *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on* (2012), IEEE, pp. 3642–3649.

[5] CIREŞAN, D. C., MEIER, U., GAMBARDELLA, L. M., AND SCHMIDHUBER, J. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation 22*, 12 (2010), 3207–3220.

[6] DIXON, M., KLABJAN, D., AND BANG, H. J. Classification-based financial markets prediction using deep neural networks, Mar 2016.

[7] HINTON, G. Csc321 winter 2014 - calendar. retrieved from lecture 6 slides.pdf slide show.

[8] KANG, N. Introducing deep learning and neural networks - deep learning for rookies (1), Jun 2017.

[9] LI, F.-F., AND KARPATHY, A. Convolutional neural networks for visual recognition, 2015.

[10] MCNALLY, S. *Predicting the price of Bitcoin using Machine Learning*. PhD thesis, Dublin, National College of Ireland, 2016.

[11] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[12] SZEGEDY, C., TOSHEV, A., AND ERHAN, D. Deep neural networks for object detection. In *Advances in neural information processing systems* (2013), pp. 2553–2561.