Draft version December 16, 2020
Preprint typeset using LaTeX style emulateapj v. 12/16/11

THE BITCOIN NETWORK AND BITCOIN PRICE PREDICTIONS WITH LONG SHORT-TERM MEMORY RECURRENT NEURAL NETWORKS

Dahl, Jon Kristian
Bekkevik, Marit
Reaz, Fardous

Draft version December 16, 2020

ABSTRACT

During the cryptocurrency boom the last few years, many have tried and failed to become cryptocurrency millionaires by investing their life savings in these strange and elusive forms of currency. In this research, we try to make a safer and more scientific approach in the quest of becoming rich on cryptocurrencies, by studying the Bitcoin price development using modern machine learning tools. We study how well a long short-term memory (LSTM) type of recurrent neural network (RNN) can predict future price development of the cryptocurrency Bitcoin. This was done by performing five day future predictions at four different dates in the previous 100 days of the Bitcoin price development. The price predictions were, to our dismay, not particularily accurate. We could not find any obvious pattern in the use of specific hyperparameters, like the dropout rate of nodes in the network. However, what we saw a greater potential with is trend forecasting. Some of the five day future predictions did show a small agreement in whether the price would rise or fall the next day, even thought the price itself was not accurate. Using a grid search we found the optimal hyperparameters giving the lowest mean squared error (MSE) to be: dropout rate = 0, sequence length = 100, batch size = 4, number of epochs = 86, and number of neurons per layer = 70. We also found that a lower dropout rate is more accurate for few epochs, while a large dropout rate performs best with many epochs. Our price replication with linear regression proved to be able to model the actual price quite well, but can not make any meaningful predictions beyond the available data. Due to the erratic nature of the Bitcoin price, we were not able to poinpoint exact hyperparameters to use for several day forecasts. We propose a future study on binary classification to predict whether the price will go up or down. ReLU proved to produce the most stable MSE, better than the logistic function and tanh. Final verdict: We did not become Bitcoin millionaires, but we did get rich in our knowledge of neural networks.

1. INTRODUCTION

The first ancestor of the blockchain was published by computer scientist David Lee Chaum in 1982, where he proposed a system for communication secured by cryptographic techniques Chaum (1979). His system allows a secure way of moving information that all participants can trust, without needing to trust each other. The term blockchain was coined with the creation of the Bitcoin (BTC) blockchain in 2008 by Satoshi Nakamoto. Nakamoto published the Bitcoin whitepaper in 2009, which outlines a peer-to-peer electronic cash system not governed by a central entity Nakamoto (2009). Bitcoin quickly became a hit, and in the years following its creation, numerous other cryptocurrencies have appeared.

Cryptocurrency as a medium of exchange has interested many as a way to make fast money due to the high volatility. Bitcoin, for example, can vary by hundreds of dollars in a day, and the prices more than quadrupled in the last 8 months Fries (2020). Making predictions about future cryptocurrency prices is tempting and has been the quest for many machine learning researchers.

In this research we use linear regression to reproduce the historical price data of Bitcoin from its inception. The regression analysis employs several polynomial degrees for the design matrix to see how well we can hope to reproduce the already existing data. Further, we use a long short-term memory (LSTM) recurrent neural network to analyse the Bitcoin price as a time series. With LSTM we perform actual price prediction several days into the future, for then to check the predictions against known data. We vary parameters like the dropout rate, sequence length, hidden layer activation function, and the number of epochs, to see how accurate we can get in our predictions and to see whether we will join the Bitcoin millionare club.

The concepts behind cryptocurrency and the finesses of the LSTM recurrent neural network algorithms are found in the theory section. Our approach and description of code can be examined in the following methods section. In the results and discussion section we present the products of our study, what we have learned, and what we can hope to achieve in the future with these tools. We then close off by a conclusion and a list of references.

All code used to generate data for this report is available at the GitHub repository https://github.uio.no/jonkd/FYS-STK4155_H20/tree/master/projects/project3.

2. THEORY

2.1. *Cryptocurrency*

Cryptocurrency differs from regular fiat currencies in two major ways: 1, a cryptocurrency is fully digital and 2, it has no central entity to control it. There are no central governments to issue it, and no banks are needed to verify accounts or transactions. Instead of having a central ledger to keep track of all the transactions, the ledger is distributed publicly to all the peers of the cryp-

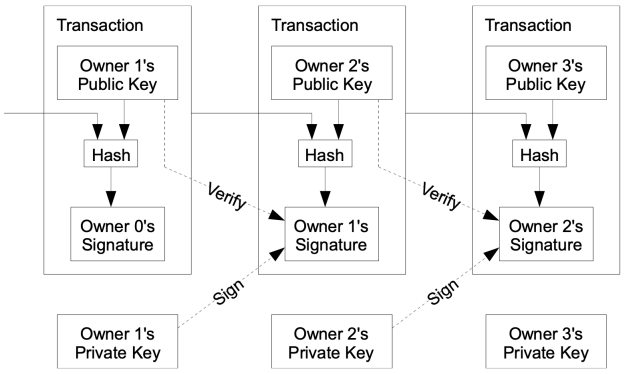

FIG. 1.— Three transactions in a simple cryptocurrency network Nakamoto (2009).

tocurrency network meaning that it is completely decentralized. Keeping such a system safe for all participants is where cryptography comes into play, and there are several safety features implemented in such a network. In the following description we will use Bitcoin as the cryptocurrency, but the general ideas are the basis for most, if not all, cryptocurrencies out there.

2.1.1. *Transactions*

Bitcoin is defined as a chain of digital signatures. A transaction must be digitally signed to be valid. To transfer one Bitcoin from a sender to a recipient, the sender digitally signs a hash of the preceding transaction of the network and the public key of the recipient, and adding these values to the end of the coin. Such a chain is visualised in figure 1, where we see three such transactions. In this way we can be sure as to who is sending the Bitcoin and who is recieving it, but an immediate problem is that the recipient has no guarantee that the sender has not sent this coin to someone else as well. This flaw is aptly named double-spending since it involves spending a single coin two or more times. The way of solving the double-spending problem involves publicly announcing all transactions to anyone who is listening, and to use a distributed timestamp server to timestamp the transaction. Enter *proof-of-work*.

2.1.2. *Proof-of-work*

The distributed timestamp server of Bitcoin takes basis in the proof-of-work system Hashcash Back (2002). In short, the proof-of-work system works by that each node in the network gathers all transaction data, the hash of the previous data and a nonce, and inputing this into a hash function. The cryptographic nonce is simply a sequence of bits which may be varied to produce a desired hash. In the Bitcoin network, the nonce is varied so that the output hash starts with a given number of zeros. When a node finds the input (transaction data and previous hash) and nonce which produces an output hash with the desired number of zeros, the transaction data, previous hash and nonce is broadcast to all nodes in the network. All nodes can then take this broadcast information and easily verify that this input indeed produces the required number of zeros. This combination of transactions, previous hash and nonce constitutes a *block*, and as more and more nodes come to the same nonce to produce the correct hash, that block is accepted as the next block in the chain and all transactions within that block are valid. A simple schematics is displayed in figure 2. For all this machinery to work, we need a *cryptographic hash function*.

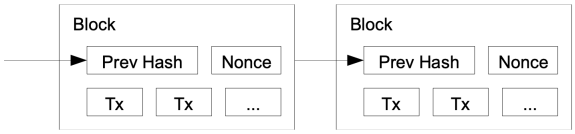

FIG. 2.— Two blocks in a so called blockchain. Each block contains the hash of the previous block, along with all the current transactions and a nonce Nakamoto (2009).

2.1.3. *Cryptographic hash function*

The cryptographic hash function is the heart of all **crypto**currencies. Bitcoin uses the cryptographic hash function SHA256 designed by the United States National Security Agency NSA (2003). The input into such a function is any sequence of numbers or words translated into bits, and the output is a new sequence of bits. This bitstring is called the hash of the function, and its length is determined by which hash function is used. SHA256 has a hash length of 256 bits. The reason to use such a function is that it is insanely hard to reverse engineer an input based on its hash. Consider the following hashes, represented in hexadecimal:

```
>>>hashlib.sha256(b"bitcoin").hexdigest()
'6b88c087247aa2f07ee1c5956b8e1a9
f4c7f892a70e324f1bb3d161e05ca107b'

>>>hashlib.sha256(b"bitcoins").hexdigest()
'b1e84e5753592ece4010051fab17777
3d917b0e788f7d25c74c5e0fc63903aa9'
```

Even though the input bits are very similar, the output hashes are completely different. The only way to find an input based on a hash is to blindly try different combinations of hashes. A so called brute-force approach. The difficulty of reverse-engineer an input is so large (though dependent on the hash function) that virtually all of digital security, like the connection between personal browsers and banks or other institutions, are based on cryptographic hashing functions.

2.1.4. *Miners*

Now, who does all this proof-of work and why do they do it? Bitcoin miners are working on finding the next block in the network based on the previous block, the nonce, and the requirement of zeros in the new hash. Now, the requirement of a given number of zeros in the hash is something all miners must agree on. When the next block has been found independently by a number of different nodes, that block is accepted as the next block in the chain, and the node which found the block is rewarded with a given number of Bitcoins. A problem is that several nonces may produce different hashes which all meet the requirement of zeros. If two different blocks are made from the same previous block, the other nodes in the network which recieve the broadcasts of found blocks start to build new blocks on top of the

block they recieved. The network is programmed to accept the longest chain of blocks as the true chain. In that way if you were to make a false block containing information saying that someone paid you 1000 Bitcoins, you would have to produce a lot of consecutive blocks after it to make it the longest of the existing chains. You might be able to keep this up for a while, but remember you're competing with all the miners in the world. Result: it is possible to swindle the system if you own 51% or more of the total network computation capacity, which in practice is impossible.

The number of zeros required changes all the time, leading to a mean block time (the time it takes to make a new block) of about 10 minutes for Bitcoin, while other cryptocurrencies may have shorter block times. The reward gets cut in half every fourth year, resulting in restricting the total amount of the currency created. In addition to the reward the miners are also granted a transaction fee for each transaction in the block.

The first and most famous cryptocurrency is, as mentioned, Bitcoin. Other notable currencies are Ethereum, Ripple, Litecoin and Monero, each employing differences in their network regarding the transaction time (block time), transactions per second (block size), and other aspects of privacy. The central purpose of cryptocurrencies is decentralization which has been somewhat reversed by the creation of application-specific integrated circuits (ASICs) designed to do a single job: generate hashes for a given algorithm. This has placed enormous amounts of hashing power at central locations. Cryptocurrencies like Ethereum and Monero avoid this by using hashing algorithms which are very costly and difficult to create ASICs for, but are effectively run on regular CPUs and GPUs. This makes it possible for a larger population to perform mining at a profit, in effect distributing the hashing power. Monero is a cryptocurrency designed to have untracable transactions and was somewhat recently in the media in the case of the kidnapping of Anne-Elisabeth Falkevik Hagen. The kidnappers wanted to use Monero due to its untraceability Jenssen (2020).

2.2. *Data set*

Historical data on the value of bitcoins that we use are from `www.coinmarketcap.com`. The price development of other cryptocurrencies are also found there. Bitcoin's price is like other commodities of supply and demand. There is a limit of 21 million Bitcoins, and this naturally restricts the supply. Already more than 88% of all the bitcoins are mined. The price of bitcoin will also vary from market to market since each market determines its own price based on sell and buy prices Colin Aulds (2003).

The dataset we use has the entire historical price development of Bitcoin in USD from CoinMarketCap. The entire dataset is shown in figure 3.

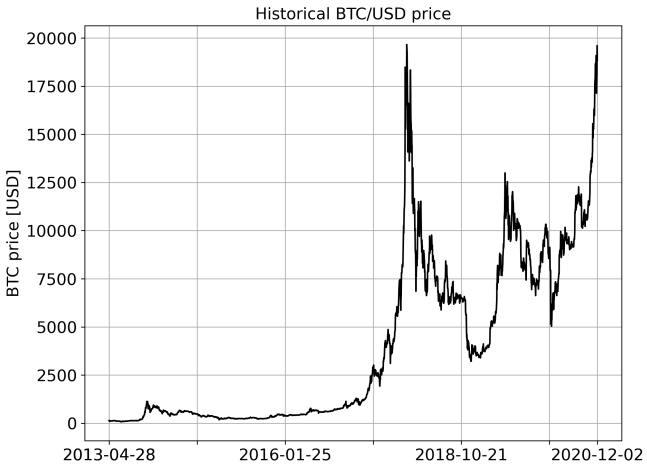

FIG. 3.— Historical price for the Bitcoin price in USD. Data from CoinMarketCap.com.

2.3. *Recurrent neural network*

A recurrent neural network (RNN) is a class of artificial neural networks commonly used in audio recognition, medical surveillance such as EEG, stock prices, genome sequencing, and more. A RNN has connections between nodes which form a graph along a temporal sequence, which allows the network to have a dynamic behavior for time-developing data dequences, in contrast to other neural networks which do not preserve temporal correlations. When the order of information is important, like the order of words in a sentence, a recurrent neural networks is the preferred method. In a RNN, there are loops in the network. A hidden (also called internal) state is given as a function $h_t = f_w(h_{t-1}, x_t)$ that takes as input the output from the previous hidden function h_{t-1}, together with the current input vector x_t. This results in a recurrence relation giving the RNN its name. Note that the same function and the same parameters are used in each time step.

The process of a forward pass in a RNN can basically be parted into three steps. First take an input vector x_t. Second, update the hidden state

$$h_t = f(w_{hh}^T h_{t-1} + w_{xh}^T x_t). \tag{1}$$

The function f is often the tanh function. Third, compute the output vector

$$\hat{y}_t = w_{hy}^T h_t, \tag{2}$$

where w_{hh} (the weight multiplied with the previous hidden state to give the current hidden state), w_{xh} (the weight multiplied with the current x-input before it goes into the hidden state) and w_{hy} (the weight multiplied with the hidden state before giving the output) can be seen in figure 4. They are the weights, the "magnitude" given to the information. The function of the hidden state is normally non-linear like tanh or ReLU. The internal state can be thought of as the memory of the system. In practice the memory can remember a few steps back in time, although we can decide it to be arbitrarily large. The outputs $\hat{y}$ are all used to compute the loss which progresses the training of the network.

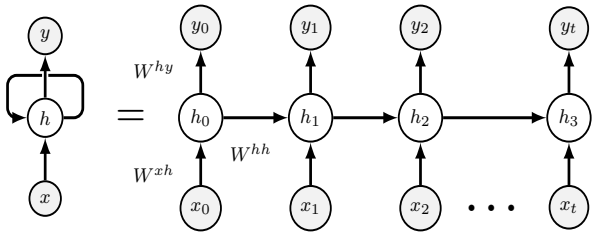

FIG. 4.— A schematic view of a recurrent network unfolded. x is the input weigthed W^{xh}. h is the hidden state weigthed W^{hh}. y is the output weigthed W^{hy}. When the network is unfolded we see how information from the previous hidden state h_{t-1} influences the next hidden state h_t and so forth. The Tikz code from https://tex.stackexchange.com/questions/494139/how-do-i-draw-a-simple-recurrent-neural-network-with-goodfellows-style.

Two notable problems may occur in a RNN, namely *exploding gradients* and *vanishing gradients*. Exploding gradients is the problem that the gradients get too large in training, making the model unstable. Likewise, vanishing gradients refer to gradients getting so small during training that the network weights wont change their values. Both of these problems cause the model to be unable to learn from the training data.

To combat exploding gradients we may employ *gradient clipping* which is the name of the procedure where we scale the gradient, should it get too large.

If we encounter many values < 1 that are multiplied together, we might experience vanishing gradients. The cure is to use a different activation function or to initialize the weights in a different way. Vanishing gradients may also be solved by using a *long short-term memory* recurrent neural network.

2.4. *Long Short-Term Memory*

Long Short-Term Memory (LSTM) is a type of RNN that deals with the problem with vanishing gradients that we may experience in backpropagation during the training of a RNN. As LSTM RNN can still suffer from exploding gradients. In a regular RNN the gradient gets weaker as one propagates backwards through time. When the network gets to the oldest data it may have no ability to adjust the weights significantly. A LSTM RNN remembers important information from the past and forgets the unimportant information. In our case, the difference lies in how the hidden state is computed. A typical LSTM-network is built up from cells or memory units (the big rectangle with rounded corners seen in figure 5) with regulators or gates (the little circles inside the cell with a plus or multiplication sign in it). The gates have information going in to and out from them. The gates control the input that enters the cell (input gate), the output (output gate) that returns a filtered version of the cell state and a forget gate that controls if a value remains in the cell. Inside the cell it is common to use the logistic function as activation function. As seen for instance to the left in the figure as the leftmost vertical line, a forget gate can be represented as neural layer with the logistic sigmoid activation function and a point multiplication $\otimes$, meaning that current input $x^{\langle t \rangle}$ and the input from the last hidden state $h^{\langle t-1 \rangle}$ goes into the sigmoid function and the output from this is multiplied by the previous cell state. The logistic sigmoid function is perfect for the job, giving an output between 0 (nothing is passed through) and 1 (everything is passed through). We see in the figure that both the cell state $c^{\langle t-1 \rangle}$ and the hidden state $h^{\langle t-1 \rangle}$ is passing from cell to cell. This implies that the cell state can actually be maintained independently of what is outputted. In addition the separate cell state enable efficient training of the LSTM by back propagation through time with uninterupted gradient flow, running along the top horizontal line between $c^{\langle t-1 \rangle}$ and $c^{\langle t \rangle}$ in the figure but in the opposite direction from right to left. The disadvantages is that these networks take a lot of computational power to train and run.

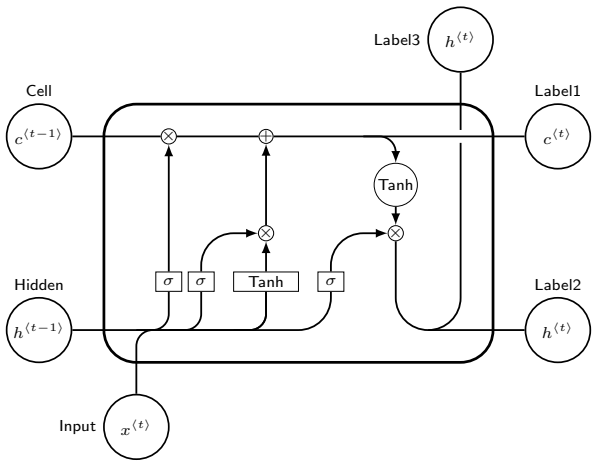

FIG. 5.— The architecture of a LSTM-cell represented by the big rectangel with rounded corners. The input is $x^{\langle t \rangle}$. Both the previous cell state $c^{\langle t-1 \rangle}$ and the previous hidden state $h^{\langle t-1 \rangle}$ enters the cell. Inside the cell there are some gates or regulators ($\otimes$ or $\oplus$) that controls the flow of information. There are input gates controlling what enters the cell, forget gates that desides if a value remains in the cell and output gates filtering what comes out from the cell. Tikz-code from https://tex.stackexchange.com/questions/432312/how-do-i-draw-an-lstm-cell-in-tikz

2.5. *Bidirectional RNN*

Bidirectional RNNs are based on the notion that the output at time t not only depends on previous information, but also knowledge from the future. Think of a sentence with a missing word. You might want to look at the word both in front and behind to get hints. This sound perhaps complicated, but in principle it is just two RNNs, one that deals with info from the past and another that deals with the future. And then the final output is based on both states Britz (2015).

2.6. *Dropout*

Dropout is a well-established procedure used to improve error and limit overfitting. Dropout means that some of the neural network nodes are left out each run. The dropout rate is a hyperparameter we can tune to adjust the precentage of nodes to exclude. By running the network with the same data for different dropout rates, one can find the mean and variance, giving the network's outcome and uncertainty.

2.7. *Adam optimizer*

Adam stands for **ada**ptive **m**oment estimation and is an extension of the stochastic gradient descent method.

It's a very popular algorithm that gives good results in addition to being both fast and easy to implement. Adam can be said to benefit from the advantages of two other variants of stochastic gradient descent, namely Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). It adapts the learning rate during the process differently for the individual parameters, and is using the first and second moment of the gradients.

Simply written, the basic steps in the Adam algorithm are first to initialize the values at time $t = 0$, the momentum of first order derivative (gradient) $m_0 = 0$ and the momentum of the second order derivative $v_0 = 0$. Then some constants are set:

$$\beta_1 = 0.9 \tag{3}$$
$$\beta_2 = 0.999 \tag{4}$$
$$\alpha = 0.001 \tag{5}$$
$$\epsilon = 10^{-7} \tag{6}$$

(3) is the parameter of the momentum, (4) is the parameter of the momentum of the squared gradient, (5) is the constant in the weight update step also known as the learning rate, and (6) is a small positive number to ensure that we don't divide by zero when the weights are updated, as we will see soon Keras (2020). The values listed for the constants are recommended by many papers and default for the method on Keras. Then the steps in the algorithm are:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{7}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{8}$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{9}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{10}$$
$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{11}$$

Equation (7) is the Adagrad step which uses the first order derivative, the gradient g_t. At any time t we can see from the value of $\beta_1 = 0.9$ that this equation gives more significance (β_1) to the recent momentum m_{t-1} and less $(1 - \beta_1)$ to the current gradient g_t. The effect of the second term is to smooth out a bit taking into account the direction of where we are going next (folowing the gradient) making the kink to the next step less prominent. Equation (8) is the RMSProp step. Also here more significance (β_2) is given to the recent second order momentum-term v_{t-1} and less $(1-\beta_2)$ to the current gradient squared g_t. Equation (9) and (10) are called the bias adjust steps. We will initialize with m_0 and v_0 equal to zero, making them biased towards zero, and here we take care of that bias. To see how it works, let's say for simplicity $g_t = 10$ and see what happens when we are to calculate m_1.

$$m_1 = \beta_1 m_0 + (1 - \beta_1) g_1 \tag{12}$$
$$= 0.9 \times 0 + 0.1 \cdot 10 = 1. \tag{13}$$

If we use equation (9) we get

$$\hat{m}_1 = \frac{1}{0.1} = 10. \tag{14}$$

We want a value closer to 10 (since the gradient is 10) and not 1, which we can say is biased towards 0, and equation (9) gives that. Equation (11) is the weight update step taking into account both the Adagrad effect (by m_t in the numerator) making sure that the learning rate is adjusted as we go along (bigger steps far from minimum and smaller closer to minimum), and the RMSProp effect (by $\sqrt{v_t}$ in the denominator) making sure we don't overstep the local minimum. Here we see why we need the ϵ in the denominator to avoid division by zero Brownlee (2017).

Some of the advantages of Adam is that it can handle sparse gradients on noisy data, it is computationally efficient, and it works well on big data sets. Though, Adam might not converge to an optimal solution in some cases and it can suffer a suboptimal weight decay. Weight decay is what happens in training the network, where after each update, the weights are multiplied by a factor (a penalty) slightly less than 1 to make then not grow too large.

2.8. *Linear regression and ordinary least squares*

A linear regression method is used to estimate the relationship between a reponse variable and one or more predictor variables. By adjusting a curve/ polynomial to a set of data, one obtains a fit or model with parameters βs. How well a model fits the data can be measured by calculating the differences between each datapoint and the model and squaring them and adding them, obtaining the ordinary least squares (OLS). By minimizing the OLS, one gets the best fit. More on linear regression and OLS can be found in the theory section in project 1 Jon Dahl (2020).

3. METHOD

3.1. *LSTM*

We loaded and sorted the Bitcoin price data using pandas and scaled the data with Scikit Learn MinMaxScaler scikit-learn developers (2020a). The data were split in 95% train and 5% test data. Then we set up the network with the wanted numbers of neurons, desired activation function, and dropout rate. We used the `Sequential` model from Tensorflow, adding `LSTM`, `Dropout`, and `Dense` layers to the model Abadi et al. (2015). The output layer had a linear activation function and we used the Adam optimizer. The parameters we varied was sequence length (1-100), dropout rate (0.0, 0.2, 0.4, 0.6, 0.8), batch size (4, 8, 16, 32, 64), number of epochs (1-90), number of neurons in each layer (10, 30, 50, 70, 90) and hidden layer activation function. For all trials the MSE was calculated for the test data set. We ran trials with 0 dropout rate corresponding to a regular LSTM, and trials with a dropout rate corresponding to an approximate bayesian approach.

A large grid search was performed to decide the optimal combination of all hyperparameters. After the network was trained with optimal parameters from the grid search, we chose a section of the last 100 days of the price data and made predictions at four different temporal locations, using three different dropout rates, predicting

the price five days into the "future". We say "future" since we just pretended for the network that these were values in the future, so we could compare the five day forecasts with the actual price data.

For the future price prediction, after training the network, we let the network make a single prediction based on a sequence of the previous n price points. The network makes a prediction and we append that prediction to the front of the sequence. We then remove the first price point in the sequence to keep the length at n. We then make a new prediction, based on 99 known points and the 1 previously made prediction. We append this second prediction to the front of the sequence and then remove the first. This process is repeated as may times as desired. This is in contrast to the training, where the network predicts the $(n+1)$th price based on the n preceding prices, then predicts the $(n+2)$th price based on the n previous actual prices, not with the predicted $(n+1)$th price. Pseudocode to illustrate the example follows:

```
for _ in range(n_predictions):
    prediction = model.predict(sequence)
    sequence.append(prediction)
    sequence.pop(0)
```

3.2. *Activation function analysis*

With `tensorflow.keras.layers`' LSTM network we have a few options. To sees the effect of the hidden layer activation function, we did the analysis with both the `tanh`, `ReLU` and logistic activation functions while keeping the other parameters constant. We used the following parameters: dropout rate = 0, batch size = 4, epochs = 90, neurons = 70 and sequential length = 100, which are the best parameters found from the grid search using `tanh` as activation function.

3.3. *Linear regression*

A simple linear regression from scikit learn was employed to reproduce the historical Bitcoin price data scikit-learn developers (2020b). We used data from the entire Bitcoin time period and used 80% of the data for training and 20% for testing. We varied the polynomial degree (10, 15 and 20) and saw how accurately we could reproduce data.

3.4. *The program code*

The center of the code for this research is `lstm.py`. Here we define the `CryptoPrediction` class which first loads the price data from the csv file `btc-usd-max.csv` using `pandas`. The data is prepared for the LSTM network by constructing a three dimensional matrix with dimensions `[batch_size, sequence_length, features]`. The batch size is for the adam optimizer extension to the stochastic gradient descent implementation, the sequence length defines how many temporal steps are trained on before a single prediction is made, and the number of features is set to 1.

Next, a `Sequential` model is set up using three `LSTM` layers, one `Dense` layer, and one linear `Activation` layer. The model is compiled using mean squared error as the loss function, and Adam as optimizer.

When the data is set up, and the model is created, we train the network. The code first checks whether the calculations have been run for these parameters already, based on the filename of the save state files. If they are already calculated, the code simply loads the weights from file. If the files do not exist, they are calculated and saved for later use. There are quite an amount of saved states linked to this research in the repository.

`regression.py` sets up the regression analysis as explained in section 3.3

`grid_search_best_parameters.py` performs the large grid search and generates a few plots to visualise the process for most parameters. It also performs the in-depth sequence length analysis. Generation of all data takes a few hours, but all save states are included in the repository, so all the data will be loaded from file.

`analyse_dropout_price_comparison.py` generates the future price prediction at different temporal locations for different dropout rates.

`analyse_dropout.py` generates the dropout rate analysis on the test data set.

4. RESULTS AND DISCUSSION

4.1. *MSE, dropout rates, and epochs*

Figure 6 shows a variation of the MSE of the testing data set as a function of the number of epochs. As the number of epochs increase, MSE drops sharply for all three different dropout rates. A peak at epochs = 5 is observed for all dropout rates. For all numbers of epochs we see that the MSE of the predictions fluctuate; however, we notice an overall gradual decreasing trend. At approximately 70 epochs, there is a sudden change in MSE for the dropout rates 0 and 0.4. The dropout rate of 0.8 also produced a small bump around epoch = 70, though much less prominent, but it's overall decreasing trend continued. Thus, a larger number of epochs does not ultimately ensure a better prognosis for all different dropout rates. However, our result show a better prediction with a larger number of epochs with an increased dropout rate . With a number of epochs more than 65, the amplitude of fluctuation of MSE is comparatively large for zero dropout rate. At zero dropout rate, smaller epochs (less than 65) produce a reliable prediction.

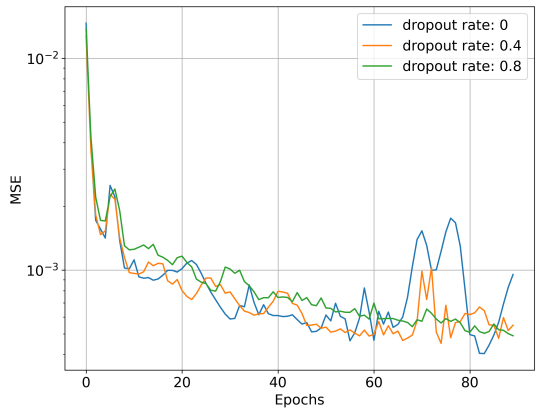

FIG. 6.— The MSE as a function of the numbers of epochs for dropout rates of 0.0, 0.4 and 0.8. The amplitude of fluctuation is significantly larger for dropout rate = 0. Different dropout rates produced the smallest MSE at different segments within the range of epochs between 0 to 90.

A more in-depth comparison is generated for five different dropout rates: 0.0, 0.2, 0.4, 0.6, and 0.8. Each dropout rate was calculated for 30, 60, and 90 epochs. The result are compiled in figure 7. For the zero dropout rate, we observe the largest MSE for the largest number of epochs. In contrast, at a dropout rate of 0.8, the largest number of epochs produces the lowest MSE. The change in MSE with the number of dropout rates is not particularity prominent for 60 epochs. It seems fairly stable. Overall, at a larger dropout rate, the MSE decreases with the number of epochs. We see the smallest MSE for a dropout rate of 0.2 with 90 epochs.

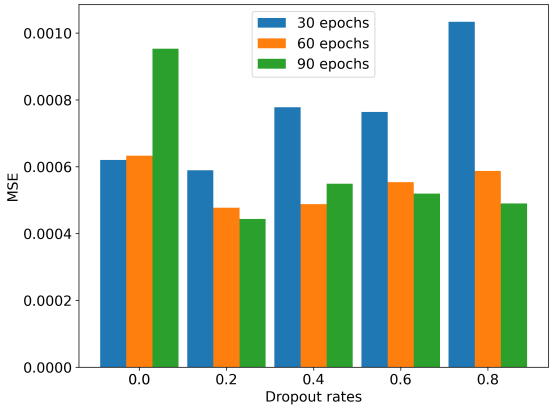

FIG. 7.— MSE for 30, 60 and 90 epochs at a dropout rate of 0.0, 0.2, 0.4, 0.6 and 0.8. The average MSE is smallest for dropout rate = 0.2. At dropout rate = 0, the most precise prediction is obtained with 30 epochs. In contrast, nonzero dropout rates produce better predictions with 60 and 90 epochs.

4.2. *Sequence length*

We vary the sequence length, that is the number of temporal data points the predictor gets to see before making a single future prediction, to observe its impact on the price prognostication. In other words, for each prediction we use data for n days to predict the price at day $n+1$. As shown in figure 8, the MSE of prediction seems to be varying randomly with the sequence length.

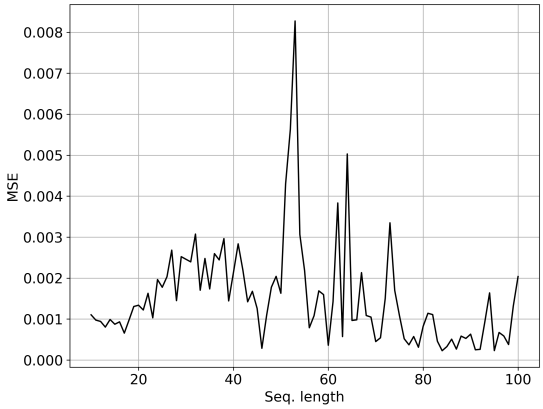

FIG. 8.— The MSE as a function of the sequence length. It varies randomly with sequence length. Most irregular behavior can be seen between sequence length = 40 to 75. It remains stable between 0 to 20. The fluctuation decreases for the sequence length larger than 75.

Several practical factors can impact the price of cryptocurrency. These features cause arbitrary fluctuations in price. Inclusion of more days is not always preferable to predict the price of just the next day. It might be helpful for long term general forecast. In our analysis, we obtain a smaller MSE in prediction within the first few days and between day 75 to day 95. It varied with a small amplitude within these ranges. With a sequence length of approximately 45, we observed our smallest MSE; however, it increased sharply afterward, as shown in figure 8. The amplitude of fluctuation in MSE between day 40 to day 75 was most prominent.

If we consider the full price range in figure 3, we can see that there are a lot of randomness. Including a larger amount of preceding days (larger sequence length) is not necessarily good for a future prediction.

4.3. *Price prediction with LSTM RNN*

We make a five day time forecast using the long short-term memory recurrent neural network, for the dropout rates 0.0, 0.4 and 0.8. We use data for 100 preceding days (sequence length) to estimate the price between day $100+1$ and day $100+5$ at different locations in the last 100 days of the entire dataset. Note that the last 100 days and 100 preceding price data for the prediction are not related numbers. The entire dataset is shown in figure 3. The result are given in figure 9, where we have started our prediction at four different days: 15, 45, 75, and 94, counting from the 100th last day. We see the actual price in solid black and dropout rates 0, 0.4, and 0.8 in dashed blue, orange and green, respectively.

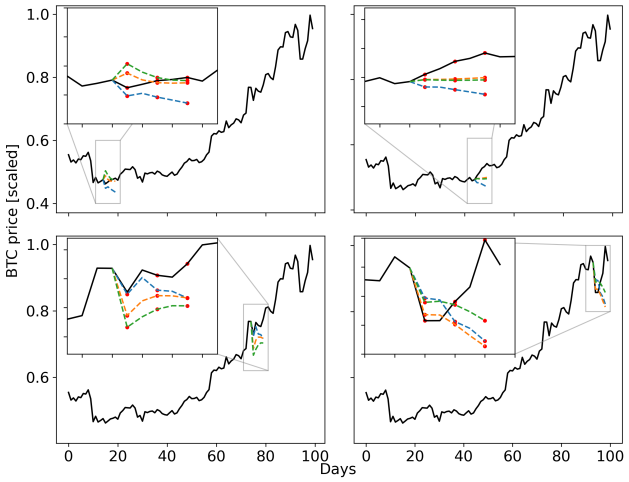

FIG. 9.— Price prediction for five consecutive days at four different dates. Dropout rates are 0.0 (blue), 0.4 (orange) and 0.8 (green). The predictions are made with a sequence length of 100 days. In a 100 days range, prediction starts at day 15 (top left), day 45 (top right), day 75 (bottom left) and day 94 (bottom right) and predict for the five following days. Prediction 1, 3, and 5 are marked with red dots, and this data is presented in table 1 as a means of seeing the actual numbers. The data is scaled. A single choice of hyperparameters cannot produce the most precise prediction at all four different price forecast sections. However, the bottom left plot shows an agreement with whether the price will rise or fall, making us regain our faith in a future as Bitcoin millionaires.

We plot the five day forecast starting at day 15 in the top left plot. We observe a deviation from the actual value between 1st and 3rd day for all dropout rates. The zero dropout rate make an underestimation and the two others produce an overestimation. A precise estimation is made between day 17 and day 19 using a dropout rate of 0.4 and 0.8; however, deviation between the actual price and the zero dropout rate prediction increases.

Consider the 45 to 49 day range in the top left plot in figure 9. Non-zero dropout rates make a slightly better price estimation than zero dropout rate. All follow a relatively linear development, which may be explained by the preceding price data being relatively linear.

In the bottom row of figure 9, we plot the price prediction between 75th to 79th day (left) and 94th to 98th day (right). The estimation using zero dropout rate is slightly better than the two other in the bottom left graph, though all of the predictions underestimates the true development of the price. Between day 94 and day 98, all three dropout rates produce similar average precision in price estimation. None of the predictions seem to follow the rising trend of the true data. The table 1 shows a numerical comparison between the predicted price for the dropout rate of 0.0, 0.4, and 0.8. The actual price is included in the table from the reference data.

Of the four plots of figure 9, the bottom left is the most interesting. While none of the predictions are accurate in price, the day 75 to 79 predictions do predict whether the price will rise or fall the next few days. If selling high and buying low is the goal, a prediction of the price is less interesting than a prediction of whether the price will rise or fall. Based on this we would say that the next step of future research is to tweak the network hyperparameters based on rise and fall, not on absolute price. Rise or fall can also be modelled as a classification problem, so a logistic neural network is interesting to look at next. For now, the LSTM RNN predictions are not somwthing we would bet our money on.

TABLE 1
THE PREDICTED PRICE WITH A DROPOUT RATE OF 0.0, 0.4 AND 0.8

		Dropout Rate		
Day	Actual	0.0	0.4	0.8
15	0.46215916	0.44805856	0.48736244	0.50351590
17	0.47440072	0.44599602	0.47095090	0.47973073
19	0.47988181	0.43572798	0.47068614	0.47455734
45	0.50521017	0.48352331	0.49644470	0.49607685
47	0.52764048	0.47891456	0.49720979	0.49494964
49	0.54210871	0.47082305	0.49958098	0.49573731
75	0.72675301	0.72317189	0.68644494	0.66634256
77	0.75589241	0.73019409	0.72079217	0.69742662
79	0.77571189	0.71667314	0.71685964	0.70318007
94	0.85751927	0.89661396	0.86796129	0.88914663
96	0.89027006	0.85616994	0.85120678	0.88512772
98	0.99685012	0.82253349	0.81356400	0.85767072

4.4. *Price replication with linear regression*

In figure 10 we see the replication of the historical data using linear regression. We vary the polynomial degree to assess the model efficiency, and we use polynomial degrees of 10, 15, and 20. This graph shows a clear advantage of using a higher degree of polynomial to improve model efficiency. The estimated price with a polynomial of degree 10 is more deviated from the actual value than the two larger polynomial degrees. It can not follow the price trend accurately at several points. We see that the 10th degree polynomial meets the major ups and downs of the price, but at the cost of varying when there is little variation in the price. In contrast, the difference between polynomials with 15 and 20 degrees is minor. They follow the price trend with similar accuracy; however, the 20th degree polynomial made a slightly better estimation. Larger polynomial degrees than 20 gave no noticeable improvement over 20.

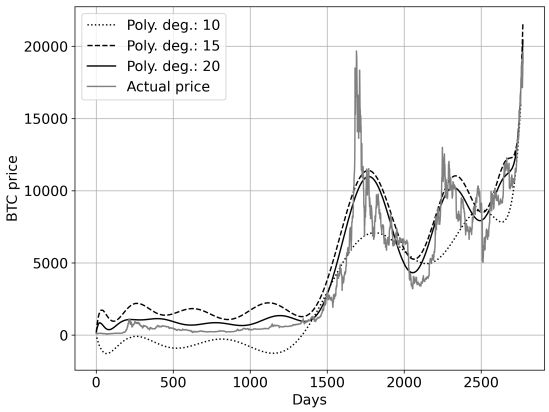

FIG. 10.— The historical price of Bitcoin in USD with three different polynomial fits to the data. We see a benefit of choosing a large polynomial degree for this dataset. That is not a big surprise, since the original price data is very erratic. Though, polynomial degrees above 20 did not yield any particular improvement. All predictions are made with the test data set.

4.5. *Activation function analysis*

In figure 11 we can se how the `tanh`, `ReLU` and logistic activation functions did in respect to each other.

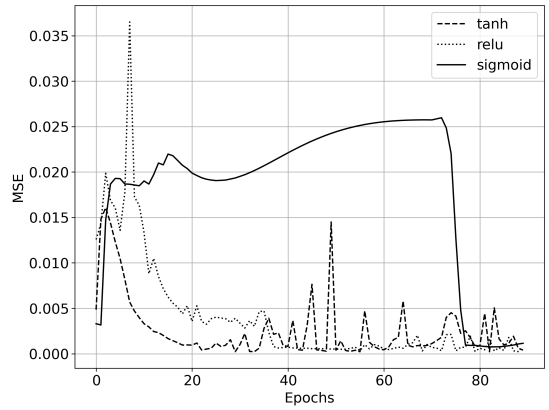

FIG. 11.— The MSE plotted for different numbers of epochs from the LSTM network from tenserflow.keras.layers for the tanh, ReLU and sigmoid hidden layers activation function. Sequence length = 100, dropout rate = 0, batch size = 4, epochs = 0 - 90 and number of neurons = 70, which is the optimal parameters we found for the tanh function. The logistic function is clearly the odd one in the bunch, with tanh and ReLU showing very similar trends.

Not surprisingly the tanh function gives the lowest MSE, since we here apply the optimal parameters found from the grid search using tanh as activation. The RelU did also well, and even better than tanh for larger numbers of epochs. We can see that ReLU shows greater stability in the MSE compared to tanh, which has several major spikes. The logistic function has the highest MSE which actually inrease with the epoch number until it suddenly drops around 75 epochs. The logistic function tends to do better in classification problems V (2017), which could partly explain why it performs worse here, where we do not perform classification. Allthough, the reason for the sudden drop is unknown to us. Its behaviour may change if we perform a grid search for optimal parameters for the logistic function. The overall winner here is ReLU which shows greater stability and lower MSEs at around 40 epochs and above. Should we tweak the parameters for ReLU, we might see an even greater advantage of using it.

5. CONCLUSION

In this research, we have studied Bitcoin's price using linear regression from scikit learn and neural networking from Tensorflow. We have used a long short-term memory recurrent neural network for price forecasting. We have seen that the employed LSTM RNN algorithm is not a convenient tool for a long term price prediction with any usable accuracy. The MSE assessment for various dropout rates could not provide us a set of parameters for the entire range with any clear advantage. The pricing data seems to be too random and too erratic, and there are simply too many dependencies on a dataset like this to accurately be forecast with the methods we have employed. A prominent fluctuation in MSE was noticed for a zero dropout rate compared to its nonzero counterparts, especially at a larger number of epochs. The average MSE for 30, 60 and 90 epochs was lowest for a dropout rate of 0.2, though the grid search over all hyperparameters gave a minimum MSE for dropout = 0, sequence length = 100, batch size = 4, number of epochs = 86, and number of neurons per layer = 70. The comparison between the predicted price of five days period with the actual price could not reveal an ultimate conclusion for the price forecast. Apparently, it looks more convenient to forecast the increasing and decreasing of the price, instead of forecasting the price itself, which can be done using classification analysis with a neural network. A further study using a logistic regressor is proposed. A study by Ji et al. (2019) obtained experimental results which showed that although LSTM-based prediction models only slightly outperformed regression for Bitcoin price prediction, DNN-based models performed best for price ups and downs prediction. Of the three tested hidden layer activation functions, ReLU outperformed both tanh and the logistic function. ReLU produced lower and more stable MSE values at 40 epochs and beyond. Price prediction using linear regression was convincing as it produced the price pattern accurately, though this is only reproduction of the historical data. We were not able to make any meaningful future predictions using linear regression. A higher polynomial degree demonstrated a clear advantage over the lower polynomial degrees, being able to follow the erratic behaviour of the price development more tightly. Final verdict: We did not become Bitcoin millionares.

All code used to generate data for this report is available at the GitHub repository https://github.uio.no/jonkd/FYS-STK4155_H20/tree/master/projects/project3.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., et al. 2015, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

Back, A. 2002, Hashcash - A Denial of Service Counter-Measure. http://www.hashcash.org/papers/hashcash.pdf

Britz, D. 2015, Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

Brownlee, J. 2017, Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Chaum, D. 1979, Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups, Memorandum UCB/ERL-M (Electronics Research Laboratory, University of California). https://books.google.ca/books?id=bbeNnAEACAAJ

Colin Aulds, Malcolm Cannon, J. L. 2003, Buy Bitcoin Worldwide. https://www.buybitcoinworldwide.com/how-many-bitcoins-are-there/

Fries, T. 2020, Bitcoin's Value Has More Than Quadrupled in Last 8 Months. https://tokenist.com/bitcoins-value-has-more-than-quadrupled-in-last-8-months/

Jenssen, T. B. 2020, Tom Hagen-saken: Kryptografen har avdekketet hele meldingsutvekslingen over bitcoin. https://kryptografen.no/2020/06/12/tom-hagen-saken-kryptografen-har-avdekketet-hele-meldingsutvekslingen-over-bitcoin/

Ji, S., Kim, J., & Im, H. 2019, Mathematics, 7, 898, doi: 10.3390/math7100898

Jon Dahl, Fardous Reaz, M. B. 2020, FYS-STK4155 - Project 1

Keras. 2020, Adam. https://keras.io/api/optimizers/adam/

Nakamoto, S. 2009, Bitcoin: A Peer-to-Peer Electronic Cash System, https://bitcoin.org/bitcoin.pdf, Bitcoin.org

NSA. 2003, Announcing Approval of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard; a Revision of FIPS 180-1. https://www.federalregister.gov/d/02-21599

scikit-learn developers. 2020a, sklearn.preprocessing.MinMaxScaler, https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

—. 2020b, sklearn.preprocessing.MinMaxScaler, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

V, A. S. 2017, Understanding Activation Functions in Neural Networks. https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0