Department of Electrical and Computer Engineering

Aarhus University

Bitcoin Price Prediction

Márta Simon - 202001346

Supervisors:
Alexandros Iosifidis
Mostafa Shabani

January 4, 2022

Institute for Electro- &
Computerengineering

Contents

1 Introduction

Bitcoin is the longest running and most well known cryptocurrency, first released as open source in 2009 by the anonymous Satoshi Nakamoto.

Machine learning can be great at predicting the future from historical data. A machine-learning algorithm can be more accurate on the prediction than conventional trading strategies based on rules set by humans. Although, there is no easy way to predict stock prices accurately and no method is perfect since there are many factors that can affect them (i.e. people's emotion, natural disasters, etc). In this project, I will test the Bitcoin forecasting abilities of 4 different machine learning models in Python: LSTM, CNN, CNN-LSTM and ConvLSTM. By splitting the data into a testing and training set, I will compare each model's performance with one another and conclude which performed best.

In the second section, I will mention the papers that influenced this work. In the third section, I will introduce some of the literature required for this project. In the fourth section, I will describe how I built different models for the problem and in the fifth section I will evaluate those models. Lastly, in the sixth section, I will describe technical specifications of the project.

1.1 Problem Framing

There are many ways to harness and explore the Bitcoin historical dataset since it has the characteristics of time series. In this project, we will use the data to explore a very specific question; that is: Given the recent Open, High, Low and Close price and Volume (in BTC and in an indicated Currency) of Bitcoin, what is the expected Weighted Price for the next minute? This requires that a predictive model forecasts the weighted price of Bitcoin for each minute. Technically, this framing of the problem is referred to as a multivariate time series forecasting problem, given a single forecast step.

2 Related Work

3 Background

3.1 Time Series

In mathematics, a time series is a series of data points indexed in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time. Thus it is a sequence of discrete-time data.(32)

In our example, we have multivariate time series of features of the Bitcoin exchanges. It means, that the features or variables of Bitcoin (Open, High, Low, Close, Volume_BTC, Volume_(Currency) and Weighted Price) were observed at each time in minute to minute intervals. Each series of features can be seen as a univariate time series having observations from the same time points. A univariate time series means a sequence of measurements of the same feature collected over time.

3.1.1 Time Series Forecasting

Time series forecasting is the use of a model to predict future values based on previously observed values. The basic concept is that we forecast the time series of interest Y assuming that it has a relationship with other time series X. Y is commonly called the forecast or dependant variable and X is the predictor or independent variable. The forecasting can be univariate or multivariate. If the model predicts dependent variable (Y) based on one independent variable (X), it is called univariate, if it uses more independent variables, it is multivariate. If the model predicts a single value for the next time step, it is called one-step or uni-step forecast, if it predicts a few time steps ahead, it is called multi-step forecast. (21)

In our example, we wish to forecast a single value for the next time step of the dependent variable Y, the weighted price of Bitcoin based on its six independent variables: open X_1, high X_2, low X_3 and close X_4 price and its volume data in Bitcoin X_5 and in an indicated currency X_6 as predictors. Therefore, we are facing a multivariate uni-step time series forecasting problem.

3.1.2 Sliding Window Method

Time series forecasting can be framed as a supervised learning problem. By re-framing the time series data we will be able to use machine learning algorithms on the problem. We can do this by using previous time steps of the independent variables as feature vectors (X) to predict the next time step of the dependent variable, the target variable (y) while the order between the observations is preserved.

This procedure is called the sliding window method. In statistics and time series analysis, this is called a lag or lag method. The number of previous time steps is called the window width or size of the lag. By increasing the width of the window, we include more previous time steps and the other way around. Once a time series dataset is prepared this way, any of the standard linear and nonlinear machine learning algorithms may be applied, as long as the order of the rows

is preserved. It can be used both on categorical and regression problems with univariate or multivariate data.(21)

3.2 Supervised Learning

Supervised learning is the machine learning task of learning a function that maps an input (X) to an output (y) based on example input-output pairs. These pairs consist of an input object (typically a vector) and a desired output value. The goal is to approximate the real underlying mapping so well that when we have new input data (X), we can predict the output variables (y) for that data. (31) The inputs are also called features and the output is also called the target. The data is usually split into training and testing data. The supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping the unseen examples of the testing data. If the output variable is numerical (continuous) then we are talking about a regression problem, if the output is categorical (discrete) then it is a classification problem.

3.3 Data Normalization

Machine learning algorithms perform better when numerical input variables are scaled to a standard range. The input variables may have different units (e.g. open price, volume of Bitcoin) that, in turn, may mean the variables have different scales. Differences in the scales across input variables may increase the difficulty of the problem being modeled.

An example of this is that large input values (e.g. a spread of hundreds or thousands of units) can result in a model that learns large weight values. A model with large weight values is often unstable, meaning that it may suffer from poor performance during learning and sensitivity to input values resulting in higher generalization error. This difference in scale for input variables does not affect all machine learning algorithms.

Unity-based normalization or min-max feature scaling is a rescaling of the data from the original range so that all values are within the new range of 0 and 1. An attribute is normalized by subtracting the minimum value of the dataset and dividing by the range between the maximum and minimum values of the dataset. (20) The formula is shown below.

$$y = (x–min)/(max–min)$$

3.4 Algortihms

3.4.1 Long Short-Term Memory (LSTM)

Long Short-Term Memory networks or "LSTMs" are a special kind of recurrent neural network (RNN), capable of learning long-term dependencies and sequences of observations. They were introduced by Hochreiter & Schmidhuber in 1997 (25), and were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs.

They work well on a large variety of problems, and are now widely used. For instance, they are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series.

LSTMs have a chain like structure like RNNs, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way. (See Figure 1)

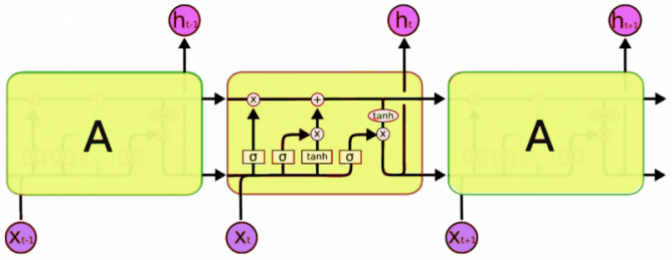
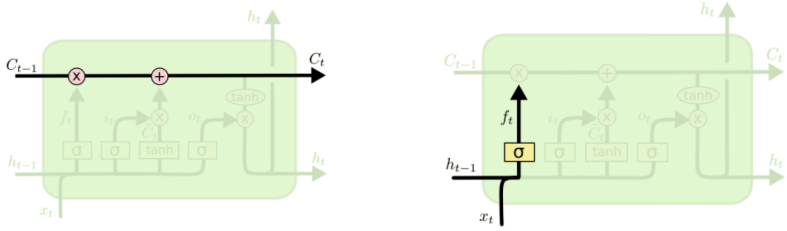


Fig. 1: LSTM network

A common LSTM unit is composed of a cell state, an input gate, an output gate and a forget gate. The cell state (Figure 2a) remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. The forget gate (Figure 2b) is a sigmoid layer used to decide whether we should keep the information from the previous timestamp or completely forget it.

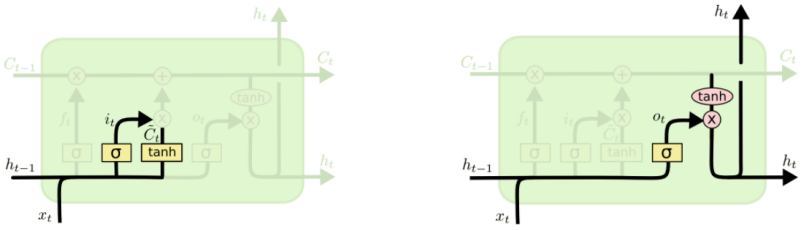


(a) Cell state

(b) Forget gate

The input gate (Figure 3a) consists of two parts, a sigmoid layer which decides which values we will update and a tanh layer which creates a vector of

new candidate values, the results of the two parts are combined to create an update to the cell state. Then we update the cell state by multiplying the old cell state by the output of the forget gate to forget the things we decided to forget earlier and adding the new candidate values from the input gate to it. The output will be a filtered version of the cell state. We run a sigmoid layer in the output gate (Figure 3b) to decide what parts of the cell state we are going to output. Then we put the cell state through a tanh layer and multiply it by the output of the output gate, so that we only output the parts we decided to. (28)



(a) Input gate

(b) Output gate

3.4.2 Convolutional Neural Network (CNN)

Convolutional Neural Networks (LeCun, 1989 (34)) or "CNNs", are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels.

CNNs are simply neural networks that use a mathematical operation called convolution in place of general matrix multiplication in at least one of their layers. In the context of a CNN, convolution is a linear operation that involves the multiplication of a set of weights with the input, where the set of weights is called a filter or kernel.

The convolution layers separate and identify the various features of the data for analysis in a process called Feature Extraction. Then a fully connected layer utilizes the output from the convolution process and predicts the class of the time-series or image based on the features extracted in previous stages. Illustration on Figure 4 (24).

A typical convolution layer consists of three stages (see Figure 5). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage. In the third stage, a pooling function is used to modify the output of the layer further. Pooling layers reduce the dimensions of data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. There are two common types of pooling

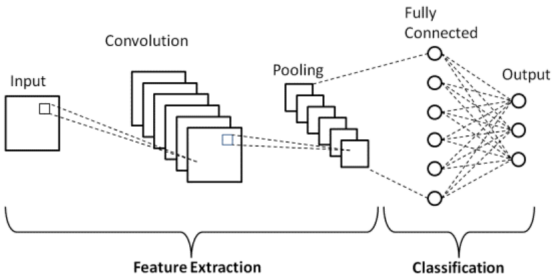


Fig. 4: Basic architecture of CNN (24)

in popular use: max and average. Max pooling uses the maximum value of each local cluster of neurons, while average pooling takes the average value. (23)

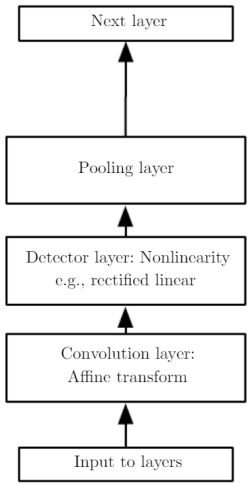


Fig. 5: Typical convolutional layer (23)

3.4.3 Convolutional Neural Network Long Short-term Memory Network (CNN-LSTM)

The Convolutional Neural Network Long Short-Term Memory Network or CNN-LSTM for short is an LSTM architecture specifically designed for sequence prediction problems with spatial inputs, like images or videos.

The CNN-LSTM architecture involves using Convolutional Neural Network (CNN) layers for feature extraction on input data combined with LSTMs to support sequence prediction. This architecture was originally referred to as a Long-term Recurrent Convolutional Network or LRCN model (22). The model structure diagram is shown in Figure 6, and the main structure is based on

CNN and LSTM models, including input layer, one-dimensional convolution layer, pooling layer, LSTM hidden layer, and fully connected layer.

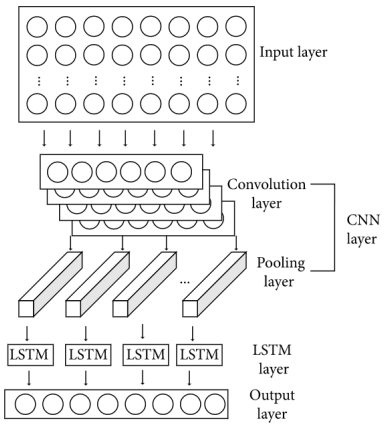


Fig. 6: Model architecture of CNN-LSTM (30)

3.4.4 Convolutional Long Short-term Memory Network (ConvLSTM)

Convolutional Long Short-Term Memory Network, or ConvLSTM for short is further extension of the CNN-LSTM approach and is also used for spatio-temporal data. It performs the convolutions of the CNN as part of the LSTM for each time step - the internal matrix multiplications of the LSTM are exchanged with convolution operations. (29) The architecture of the ConvLSTM layer is shown on Figure 7.

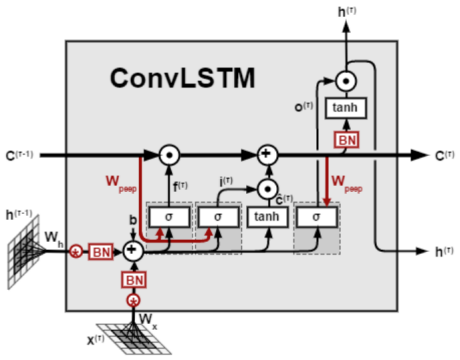


Fig. 7: Architecture of a ConvLSTM layer (33)

4 Approach

In the followings, we will go through the key steps required to develop the machine learning models for the multivariate time series forecasting problem we have.

4.1 Data Collection

The chosen dataset, the Bitcoin Historical Dataset is freely available for use on Kaggle (35). It consists of exchanges of Bitcoin for the time period of December 2011 to March 2021. It can be considered as a multivariate time series comprised of seven variables besides the timestamp, which are the following with minute to minute updates: Open, High, Low and Close price, Volume in BTC and in an indicated Currency, and Weighted Bitcoin Price.

Samples of the dataset are shown in Table 1. The timestamps were originally given in Unix time, but they are displayed after they got converted to standard date format. The description of the variables are shown in Table 2.

Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
2021-03-30 23:51:00	58677.1	58699.9	58660	58699.9	2.67268	156833	58680
2021-03-30 23:52:00	nan	nan	nan	nan	nan	nan	nan
2021-03-30 23:53:00	58718.7	58731.2	58698.3	58698.5	0.207983	12209.1	58702.3
2021-03-30 23:54:00	58736.2	58762.4	58736.2	58739.9	0.0415588	2441.38	58745.1
2021-03-30 23:55:00	58742.2	58742.2	58714.3	58714.3	2.52	148004	58731.9
2021-03-30 23:56:00	58714.3	58714.3	58686	58686	1.38449	81259.4	58692.8
2021-03-30 23:57:00	58684	58693.4	58684	58685.8	7.29485	428158	58693.2
2021-03-30 23:58:00	58693.4	58723.8	58693.4	58723.8	1.70568	100117	58696.2
2021-03-30 23:59:00	58742.2	58770.4	58742.2	58760.6	0.720415	42333	58761.9
2021-03-31 00:00:00	58767.8	58778.2	58756	58778.2	2.71283	159418	58764.3

Table 1: Samples of the Bitcoin Historical Dataset

Variable name	Description
Timestamp	Start time of time window (60s window)
Open	Open price at start time window
High	High price within time window
Low	Low price within time window
Close	Close price at end of time window
Volume_(BTC)	Volume of BTC transacted in this window
Volume_(Currency)	Volume of corresponding currency transacted in this window
Weighted_Price	Volume Weighted Average Price

Table 2: Description of the columns of the dataset

Timestamps without any trades or activity have their data fields filled with NaN values. If a timestamp is missing, or if there are jumps, it may be because the exchange (or its API) was down, the exchange (or its API) did not exist, or some other unforeseen technical error in data reporting or gathering. The author has been made all effort to deduplicate entries and verify the contents are correct and complete.(35)

4.1.1 Data Exploration

After downloading the dataset I investigated the distribution of NaN values in it.

The dataset consists of 4.857.377 rows of data. The first timestamp is *2011-12-31 07:52* and the last one is *2021-03-31 00:00*. In total, 1.243.608 rows have NaN values in them which are roughly 1/4 of the dataset. In Table 3, I counted the NaN values in every 525600 rows (365*24*60) which should be a full year of data. Between some of the timestamps, I experienced the above-noted jumps. In the last time interval, I took the remaining rows together with the last interval, which resulted 652577 rows in total.

Timestamp	No. NaNs
from *2021-03-31 00:00* to *2020-03-31 00:01*	7.658
from *2020-03-31 00:00* to *2019-04-01 00:01*	6.040
from *2019-04-01 00:00* to *2018-04-01 00:01*	31.199
from *2018-04-01 00:00* to *2017-04-01 00:01*	18.655
from *2017-04-01 00:00* to *2016-04-01 00:01*	166.164
from *2016-04-01 00:00* to *2015-04-02 00:01*	146.647
from *2015-04-02 00:00* to *2014-03-28 12:09*	140.419
from *2014-03-28 12:08* to *2013-03-28 12:09*	124.063
from *2013-03-28 12:08* to *2011-12-31 07:52*	602.753

Table 3: Number of NaN values in every 525600 rows in the dataset

We can conclude that the NaN values are not evenly distributed in the dataset, most of them are missing from the early years of the Bitcoin history.

4.1.2 Exploratory Visualization

On Figure 8, I visualized the weighted price of the dataset on a line chart over the entire time period to see how the price of Bitcoin was changing over time. On the x-axis, the dates, and on the y-axis the weighted price are shown.

We can clearly read from the figure that the price of Bitcoin became volatile in the most recent 4 years and fluctuated a lot.

4.2 Data Preprocessing

As we have seen in the previous section, from the early years of Bitcoin we have a lot of missing data and the more noticeable changes in the price started around 2017. Therefore, I decided to limit the dataset to the most recent 4 years instead of using all the data from December 2011 to March 2021. This way the data will be more relevant for a training model, however the recent spike will be involved in the test set - which will be a tough prediction.

After determining the range of data to be used, I filtered out the NaN values from it and defined the ratio for the training and testing data. Since the price spike is so high in January 2021, I decided on a 70% split for training and 30% for

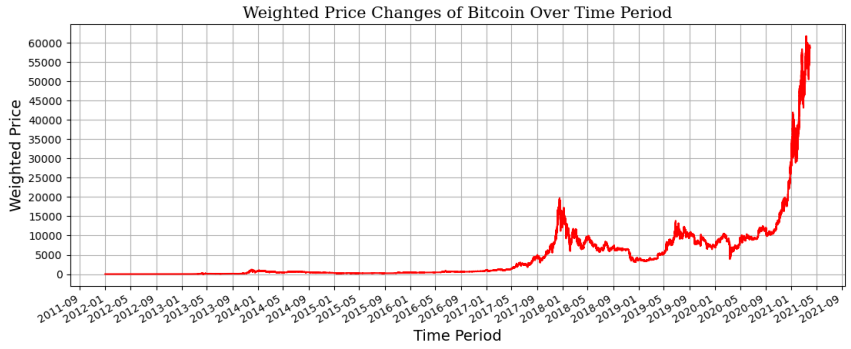


Fig. 8: Weighted price changes of Bitcoin

testing. This 30% split allows for the model to predict a good amount of values before the unpredictable spike in the test data. This way, we have *1.427.186* data for training and *611.652* data for testing. I also defined the feature vector consisting the Open, High, Low, Close, Volume_BTC and Volume_Currency variables (see on Figure 9) and the target variable consisting of the Weighted Price (see on Figure 10).

	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)
2017-04-01T00:01...	1070.93000	1070.93000	1070.55000	1070.55000	0.92858	994.16732
2017-04-01T00:02...	1070.62000	1071.03000	1070.62000	1070.69000	7.68834	8233.65254
2017-04-01T00:03...	1070.69000	1071.34000	1070.69000	1071.34000	9.31783	9981.27487
2017-04-01T00:04...	1071.34000	1073.59000	1071.33000	1073.59000	1.32644	1422.73450

Fig. 9: Samples of the feature vector

	Weighted_Price
2017-04-01T00:01...	1070.63185
2017-04-01T00:02...	1070.92721
2017-04-01T00:03...	1071.20165
2017-04-01T00:04...	1072.59377

Fig. 10: Samples of the target variable

Before we can fit any of the neural network models to the dataset, we must transform the data. In the following two subsections different types of data transformations are described which have been performed on the dataset prior to fitting the models on it and making forecasts.

4.2.1 Sliding Window Method

We want our model to learn by relating the previous data (the feature vectors) with the future data (the target variable). Therefore, we need to re-arrange our data accordingly.

Currently, the feature vector has the form [1.427.186, 6] where the first number denotes the number of samples, and the second one the number of features the vector has. We need to specify a number of time steps we want to look back in the feature vectors for prediction of the target variable - this is the width of the sliding window - and re-arrange the data in the form of [n_samples, n_timesteps, n_features]. The form of the target variable ([1.427.186, 1]) remains the same only that we need to drop the first n_timesteps-1 samples. This procedure is illustrated on Figure 11 with an example of 3 time steps which results that for every prediction of the Weighted Price, we are going to look back 3 time steps in the feature vector.

Fig. 11: Illustration of the sliding windows

To understand and implement the sliding window method, I relied on the following article (21).

4.2.2 Data Normalization

To make the problem easier to learn for the model, we need to have the data both from the feature vector and the target variable in a standard range. We use unity-based normalization to re-scale the data from their original range so that all values are within the new range of 0 and 1. We can do this in two ways, on a dataset level and on a sliding window level.

On the dataset level, we take the minimum and maximum values from all the samples of the variable to normalize the values. On the sliding window level, we take them only from the range of the sliding window. During the evaluation of the model, we will compare them against each other to find out which way was the more effective. On forecasts, these transformations have to be inverted to return them into their original scale before calculating an error score.

Data normalization and its possible implementation was well explained in (20).

4.3 Model Building

After having the data prepared and preprocessed, we can focus on building the neural network models. I chose to use 4 types of neural networks and built 5 different kinds of each of them.

4.3.1 LSTM models

LSTM neural networks are a trending approach for time series forecasting since their great ability to learn long-term sequences of observations and therefore being able to learn the context required to make predictions. Read more about LSTM in section 3.4.1. The implementation of the LSTM models for time series forecasting was well explained in (18) and (19).

The first LSTM model (see Figure 12) has two hidden layers of LSTM units with 64 and 32 units and two densely-connected layers with 16 and 1 units. The shape of the input defines what the model should expect as input for each sample in terms of the number of time steps and the number of features. We can parametrize this as *[samples, timesteps, features]*. We are working with a multivariate series, so the number of features is six, for one variable. The number of time steps as input is the number we chose as the width of the sliding window when preparing our dataset. The LSTM units use ReLU (Rectified Linear Unit) activation function which returns 0 if it receives any negative input, and for any positive value x it returns that value back. We can chose via "return_sequences" to output the state of the layer at each step which would be the full sequence (True) or only return the last output (False). The last dense layer works as an output layer and used to make a prediction. The model uses the mean squared error (MSE) as its loss function to determine the error (i.e., "the loss") between the output of our algorithms and the given target value and the Adam optimizer to adjust the weights to reduce the losses.

```
def lstm1(n_timesteps, n_features):
    model = Sequential()
    model.add(LSTM(64, input_shape=(n_timesteps, n_features), activation="relu",
                   return_sequences=True))
    model.add(LSTM(32, activation='relu', return_sequences=False))
    model.add(Dense(16))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 12: First LSTM model

The second LSTM model (see Figure 13) has four hidden layers of LSTM units with 16, 32, 32 and 64 units and two densely-connected layers with 32 and 1 units.

```
def lstm2(n_timesteps, n_features):
    model = Sequential()
    model.add(LSTM(16, input_shape=(n_timesteps, n_features), activation="relu",
                   return_sequences=True))
    model.add(LSTM(32, activation='relu', return_sequences=True))
    model.add(LSTM(32, activation='relu', return_sequences=True))
    model.add(LSTM(64, activation='relu', return_sequences=False))
    model.add(Dense(32))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 13: Second LSTM model

The third LSTM model (see Figure 14) is identical to the first one except that a dropout argument is specified for the LSTM units. Dropout is a regularization method where input and recurrent connections to LSTM units are probabilistically excluded from activation and weight updates while training a network. This has the effect of reducing overfitting and improving model performance. The dropout value is a percentage between 0 (no dropout) and 1 (no connection). The concept of dropout is well explained in the article (14) and (15). I chose the value of the dropout as a result of a grid search probing the values 0, 0.3 and 0.6. Read more about grid search in section 4.4.1.

```
def lstm3(hidden_layer, dropout, input_shape):
    model = Sequential()
    model.add(LSTM(64, input_shape=input_shape, activation="relu",
                   return_sequences=True, dropout=dropout))
    model.add(LSTM(32, activation='relu', return_sequences=False, dropout=dropout))
    model.add(Dense(16))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 14: Third LSTM model

The fourth LSTM model (see Figure 15) is identical to the second one but with dropout. I chose the value of the dropout with grid search.

```
def lstm4(hidden_layer, dropout, input_shape):
    model = Sequential()
    model.add(LSTM(16, input_shape=input_shape, activation="relu",
                   return_sequences=True, dropout=dropout))
    model.add(LSTM(32, activation='relu', return_sequences=True, dropout=dropout))
    model.add(LSTM(32, activation='relu', return_sequences=True, dropout=dropout))
    model.add(LSTM(64, activation='relu', return_sequences=False, dropout=dropout))
    model.add(Dense(32))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 15: Fourth LSTM model

The fifth LSTM model (see Figure 16) is a vanilla LSTM that has a single

hidden layer of LSTM units with dropout specified and an output layer. I chose the value of the dropout and units with grid search.

```
def lstm5(hidden_layer, dropout, input_shape):
    model = Sequential()
    model.add(LSTM(hidden_layer, input_shape=input_shape, activation='relu',
                   return_sequences=False, dropout=dropout))
    model.add(Dense(hidden_layer))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 16: Fifth LSTM model

4.3.2 CNN models

Convolutional neural networks or CNNs efficiently extract features from the data. Although they are popular in image datasets with two-dimensional data, they can also be used on one-dimensional time series data. Read more about CNNs in section 3.4.2. The implementation of the CNN models for time series forecasting was well explained in (10), (11) and (12).

CNN models follows a typical structure which consists of a convolutional and a fully connected part. The convolutional part is responsible for the feature extraction and it usually includes one or more convolutional hidden layers, a max or average pooling layer and a flatten layer. In the convolutional layer, a filter number and a kernel size are defined. The number of filters is the number of reads or interpretations of the input sequence. The kernel size is the number of time steps included of each "read" operation of the input sequence. In the flatten layer, the input is flattened by being reformatted into a shape that is equal to the number of elements contained in its input. It is used between the convolutional part and the fully connected part to reduce the feature maps to a single one-dimensional vector. The fully connected part includes dense fully connected layers which interpret the features extracted by the convolutional part of the model.

The first CNN model (see Figure 17) consists of one convolutional hidden layer with 64 filter maps and a kernel size of 2, a max pooling layer with a pool size of 2 and a flatten layer in its convolutional part. In its fully connected part, it has two densely-connected layers with 100 and 1 unit. The model expects the same input shape as the LSTM model did, so we have to specify the number of time steps and the number of features for each sample which we parametrize as *[sample, timesteps, features]*. The CNN model does not actually view the data as having time steps, instead, it is treated as a sequence over which convolutional read operations can be performed, like a one-dimensional image. Both the convolutional layer and one of the dense layers uses ReLU as their activation function. The last dense layer works as the output layer and predicts a single numerical value. The model is fit using the Adam optimizer and optimized using the mean squared error (MSE) loss function.

```python
def cnn1(n_timesteps, n_features):
    model = Sequential()
    model.add(Conv1D(filters=64, kernel_size=2, activation='relu',
                     input_shape=(n_timesteps, n_features)))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(100, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 17: First CNN model

The second CNN model (see Figure 18) is identical to the first one except that is uses average pooling instead of max pooling.

```python
def cnn2(n_timesteps, n_features):
    model = Sequential()
    model.add(Conv1D(filters=64, kernel_size=2, activation='relu',
                     input_shape=(n_timesteps, n_features)))
    model.add(AveragePooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(100, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 18: Second CNN model

The third CNN model (see Figure 19) is also identical to the first one but adds an extra dropout layer to regularize the model. I chose the size of the filter, kernel and the value of the dropout with grid search.

```python
def cnn3(filters, kernel_size, dropout, input_shape):
    model = Sequential()
    model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='relu',
                     input_shape=input_shape))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Dropout(dropout))
    model.add(Flatten())
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 19: Third CNN model

The fourth CNN model (see Figure 20) is identical to the second one but adds an extra dropout layer to regularize the model. I chose the size of the filter, kernel and the value of the dropout with grid search.

```
def cnn4(filters, kernel_size, dropout, input_shape):
    model = Sequential()
    model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='relu',
                     input_shape=input_shape))
    model.add(AveragePooling1D(pool_size=2))
    model.add(Dropout(dropout))
    model.add(Flatten())
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 20: Fourth CNN model

The fifth CNN model (see Figure 21) consists of a convolutional hidden layer with 16 filter maps and a kernel size of 2, a max pooling layer with a pool size of 2, a dropout layer, another convolutional hidden layer with various filter maps and kernel size, another max pooling with a pool size of 2, a dropout layer, a flatten layer and a dense layer with 1 unit. The model uses ReLU activation functions in its convolutional layers and it is compiled with the Adam optimizer and the mean squared error loss function. I chose in the second convolutional layer the size of the filter, kernel and the value of the dropout with grid search.

```
def cnn5(filters, kernel_size, dropout, input_shape):
    model = Sequential()
    model.add(Conv1D(filters=16, kernel_size=2, activation='relu',
                     input_shape=input_shape))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Dropout(dropout))
    model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='relu'))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Dropout(dropout))
    model.add(Flatten())
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 21: Fifth CNN model

4.3.3 CNN-LSTM models

A CNN model can be used in a hybrid model with an LSTM backend where the CNN is used to interpret subsequences of input that together are provided as a sequence to an LSTM model to interpret. Read more about CNN-LSTM in section 3.4.3. The implementation of the CNN-LSTM models for time series forecasting was well explained in (13), (18) and (19).

We want to apply the CNN model to each input and pass on the output of each input to the LSTM as a single time step. We can achieve this by wrapping the entire CNN input model (one layer or more) in a TimeDistributed layer (26). This layer achieves the desired outcome of applying the same layer or layers multiple times. In this case, applying it multiple times to multiple input

time steps and in turn providing a sequence of "interpretations" or "features" to the LSTM model to work on.

For this model, we have to prepare the input differently. We have to split the input sequences into subsequences that can be processed by the CNN model. We've already split our multivariate time series data into input/output samples with predefined number of time steps as input and one output to predict. Now, we split each sample into two sub-samples, each with half of the total time steps. The CNN can interpret each subsequence of the corresponding amount of time steps and provide a time series of interpretations of the subsequences to the LSTM model to process as input. This way, the input data will be reshaped in the form of *[samples, subsequences, (number of timesteps for each subsequences), features]* where the number of subsequences will be always 2 in the followings.

The first CNN-LSTM model (see Figure 22) consists of a one-dimensional convolutional hidden layer with 32 filter maps and a kernel size of 2, a max pooling layer with a pool size of 2 and a flatten layer. This convolutional part is wrapped in a TimeDistributed wrapper, which will apply the entire convolutional model once per input sequence. The convolutional part is followed by two hidden layer of LSTMs with 64 and 32 units and an output layer which is a dense layer with 1 unit. Both the convolutional and the LSTM hidden layer uses ReLU as their activation function. The model is compiled with the Adam optimizer and the mean squared error loos function.

```
def cnn_lstm1(n_timesteps, n_features):
    model = Sequential()
    model.add(TimeDistributed(Conv1D(filters=32, kernel_size=2, activation='relu'),
                              input_shape=(2, n_timesteps, n_features)))
    model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
    model.add(TimeDistributed(Flatten()))
    model.add(LSTM(64, return_sequences=True, activation="relu"))
    model.add(LSTM(32, return_sequences=False, activation="relu"))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 22: First CNN-LSTM model

The second CNN-LSTM model (see Figure 23) is very similar to the first one except that it has only one LSTM hidden layer with 32 units.

```python
def cnn_lstm2(n_timesteps, n_features):
    model = Sequential()
    model.add(
        TimeDistributed(Conv1D(filters=32, kernel_size=2, activation='relu'),
                        input_shape=(2, n_timesteps, n_features)))
    model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
    model.add(TimeDistributed(Flatten()))
    model.add(LSTM(32, return_sequences=False, activation="relu"))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 23: Second CNN-LSTM model

The third CNN-LSTM model (see Figure 24) is very similar to the second model except that is has 64 filter maps in its convolutional hidden layer.

```python
def cnn_lstm3(n_timesteps, n_features):
    model = Sequential()
    model.add(
        TimeDistributed(Conv1D(filters=64, kernel_size=2, activation='relu'),
                        input_shape=(2, n_timesteps, n_features)))
    model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
    model.add(TimeDistributed(Flatten()))
    model.add(LSTM(32, return_sequences=False, activation="relu"))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 24: Third CNN-LSTM model

The fourth CNN-LSTM model (see Figure 25) is a mix of the second and third model, it has two convolutional hidden layers with 32 and 64 filter maps and one LSTM hidden layer with 32 units.

```python
def cnn_lstm4(n_timesteps, n_features):
    model = Sequential()
    model.add(
        TimeDistributed(Conv1D(filters=32, kernel_size=2, activation='relu'),
                        input_shape=(2, n_timesteps, n_features)))
    model.add(TimeDistributed(Conv1D(filters=64, kernel_size=2, activation="relu")))
    model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
    model.add(TimeDistributed(Flatten()))
    model.add(LSTM(32, return_sequences=False, activation="relu"))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 25: Fourth CNN-LSTM model

The fifth CNN-LSTM model (see Figure 26) has two convolutional hidden layers with 32 and 64 filter maps and two LSTM hidden layers with 64 and 32 units.

```python
def cnn_lstm5(n_timesteps, n_features):
    model = Sequential()
    model.add(
        TimeDistributed(Conv1D(filters=32, kernel_size=2, activation='relu'),
                        input_shape=(2, n_timesteps, n_features)))
    model.add(TimeDistributed(Conv1D(filters=64, kernel_size=2, activation="relu")))
    model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
    model.add(TimeDistributed(Flatten()))
    model.add(LSTM(64, return_sequences=True, activation="relu"))
    model.add(LSTM(32, return_sequences=False, activation="relu"))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 26: Fifth CNN-LSTM model

4.3.4 ConvLSTM models

A type of LSTM related to the CNN-LSTM is the ConvLSTM, where the convolutional reading of input is built directly into each LSTM unit. The ConvLSTM was developed for reading two-dimensional spatial-temporal data, but can be adapted for use with time series forecasting. Read more about ConvLSTM models in section 3.4.4. The implementation of the ConvLSTM models for time series forecasting was well explained in (18), (19) and (27).

The model expects input as a sequence of two-dimensional images, therefore the shape of input data must be: *[samples, timesteps, rows, columns, features]*. We have to split again each sample into subsequences as we did for the CNN-LSTM model. The timesteps will become the number of subsequences in this case, the number of rows is fixed at 1 because we are working with one-dimensional data, and lastly the number of columns will be the number of time steps for each subsequence. In the followings, we will work with 2 subsequences in each model.

The architecture of the models are identical to the LSTM models from before in terms of the number of ConvLSTM/LSTM hidden layers and dense layers used and the number of the ConvLSTM filters are equal to the number of the LSTM units from the previous models. Every ConvLSTM layer are followed by a BatchNormalization layer for normalizing the inputs. The output of the ConvLSTM layers are always flattened before handed to the dense layers.

The first ConvLSTM model (see Figure 27) has two ConvLSTM hidden layers with 64 and 32 filters and a two-dimensional kernel in size (1, 3) in terms of (rows, columns). As we are working with a one-dimensional series, the number of rows is always fixed to 1 in the kernel. The output of the ConvLSTM layers are normalized and flattened so it can be interpreted and a prediction made. The ConvLSTM layers use ReLU as their activation funciton and the model is compiled with the Adam optimizer and the mean squared error loss function.

```python
def conv_lstm1(n_outputs, n_steps, n_length, n_features):
    model = Sequential()
    model.add(ConvLSTM2D(filters=64, kernel_size=(1, 3),
                         activation='relu',
                         input_shape=(n_steps, 1, n_length, n_features),
                         padding='same', return_sequences=True))
    model.add(BatchNormalization())
    model.add(ConvLSTM2D(filters=32, kernel_size=(1, 3),
                         activation='relu', padding="same"))
    model.add(BatchNormalization())
    model.add(Flatten())
    model.add(Dense(16))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 27: First ConvLSTM model

The second ConvLSTM model (see Figure 28) has four ConvLSTM hidden layers with 16, 32, 32 and 64 filters, and two dense layers with 32 and 1 units.

```python
def conv_lstm2(n_outputs, n_steps, n_length, n_features):
    model = Sequential()
    model.add(ConvLSTM2D(filters=16, kernel_size=(1, 3), activation='relu',
                         input_shape=(n_steps, 1, n_length, n_features),
                         padding='same', return_sequences=True))
    model.add(BatchNormalization())
    model.add(ConvLSTM2D(filters=32, kernel_size=(1, 3), activation='relu',
                         padding='same', return_sequences=True))
    model.add(BatchNormalization())
    model.add(ConvLSTM2D(filters=32, kernel_size=(1, 3), activation='relu',
                         padding='same', return_sequences=True))
    model.add(BatchNormalization())
    model.add(ConvLSTM2D(filters=64, kernel_size=(1, 3), activation='relu',
                         padding="same"))
    model.add(BatchNormalization())
    model.add(Flatten())
    model.add(Dense(32))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 28: Second ConvLSTM model

The third ConvLSTM model (see Figure 29) has the same architecture as the first one except that a dropout argument is specified for the ConvLSTM units to reduce overfitting.

```python
def conv_lstm3(filters, dropout, input_shape):
    model = Sequential()
    model.add(ConvLSTM2D(filters=64, kernel_size=(1, 3), activation='relu',
                         input_shape=input_shape, padding='same',
                         return_sequences=True, dropout=dropout))
    model.add(BatchNormalization())
    model.add(ConvLSTM2D(filters=32, kernel_size=(1, 3), activation='relu',
                         padding='same', dropout=dropout))
    model.add(BatchNormalization())
    model.add(Flatten())
    model.add(Dense(16))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 29: Third ConvLSTM model

The fourth ConvLSTM model (see Figure 30) has the same architecture as the second one but with dropout.

```python
def conv_lstm4(filters, dropout, input_shape):
    model = Sequential()
    model.add(ConvLSTM2D(filters=16, kernel_size=(1, 3), activation='relu',
                         input_shape=input_shape, padding='same',
                         return_sequences=True, dropout=dropout))
    model.add(BatchNormalization())
    model.add(ConvLSTM2D(filters=32, kernel_size=(1, 3), activation='relu',
                         padding='same', return_sequences=True, dropout=dropout))
    model.add(BatchNormalization())
    model.add(ConvLSTM2D(filters=32, kernel_size=(1, 3), activation='relu',
                         padding='same', return_sequences=True, dropout=dropout))
    model.add(BatchNormalization())
    model.add(ConvLSTM2D(filters=64, kernel_size=(1, 3), activation='relu',
                         padding="same", dropout=dropout))
    model.add(BatchNormalization())
    model.add(Flatten())
    model.add(Dense(32))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 30: Fourth ConvLSTM model

The fifth ConvLSTM model (see Figure 31) has a single ConvLSTM hidden layer with specified dropout and an output layer. I chose the size of the filter and the value of the dropout with grid search.

```
def conv_lstm5(filters, dropout, input_shape):
    model = Sequential()
    model.add(
        ConvLSTM2D(filters=filters, kernel_size=(1, 3), activation='relu',
                   input_shape=input_shape, padding="same",
                   dropout=dropout))
    model.add(BatchNormalization())
    model.add(Flatten())
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    model.summary()
    return model
```

Fig. 31: Fifth ConvLSTM model

4.4 Model Tuning

4.4.1 Hyperparameter Optimization

A hyperparameter is a model argument whose value is set before the learning process begins. It is a configuration that is external to the model, its value cannot be estimated from the data. Examples for hyperparameters include the width of the sliding window/number of timesteps, the number of training epochs, the size of the training batches, the value of the dropout, the number of units of an LSTM layer or the filter or kernel size of a CNN layer. There is no good theory on how to choose these parameters for different problems, therefore configuring neural networks is difficult. We must follow a systematic approach and experiment with different configurations to choose a set of optimal hyperparameters for a learning algorithm - that would be hyperparameter optimization.

In this project, all the different models are trained with the sliding window width of 8, 16 and 32, with the epoch sizes of 50 and 100 and with the batch sizes of 60, 1440 and 10080. The program goes through all of these options in a for loop and train a network with that setup. The number of epochs defines the number times that the learning algorithm will work through the entire training dataset. The batch size defines the number of samples to work through before updating the internal model parameters. I chose the sliding window widths and epoch sizes because they seemed to be popular in the literature. I chose the batch sizes based on that we are dealing with minute-to-minute data, which means that 60 minutes means an hour, 1440 minutes is a day and 10080 is a week of data.

On some of the models, I used the Keras GridSearchCV algorithm to find the best from a set of predefined hyperparameter values with exhaustive searching. In theory, the GridSearchCV would be unusable for time series data, because the cross-validation would ruin the order of the samples. However, there seem to be a way to perform grid search without the cross validation as the first answer states it here (1). The article (16) explains how to implement grid search.

4.4.2 Callbacks

Callbacks are a set of functions to be applied at given stages of the training procedure. By defining such functions, we have more control over the training process. I used the EarlyStopping and the CSVLogger callbacks. The EarlyStopping callback stops the training process when it is likely that the model is overfit. The CSVLogger writes the results of every epoch into file.

5 Evaluation

To determine how accurate the prediction is, we analyze the difference between the predicted and the actual weighted price of Bitcoin. Smaller difference indicates better accuracy. I chose Root Mean Squared Error (RMSE) as a metric to determine the accuracy of the prediction. It is a commonly used general purpose quality estimator. I chose to use RMSE because it explicitly shows the deviation of the prediction for continuous variables from the actual dataset. So, they fit in this project to measure the accuracy.

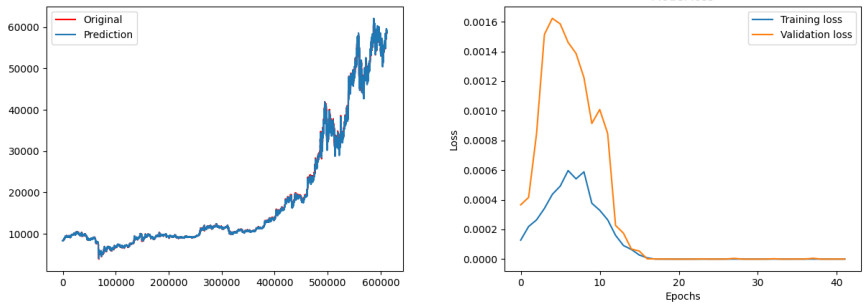
In the section 4.2.2, two ways of data scaling were introduced. As it turned out, scaling on sliding window level did not make sense for the model and produced useless results. Therefore, all the results shown below were produced only with dataset level scaling.

From the different LSTM architectures the most simple one, the fifth model had the less errors which used only one LSTM layer. In that architecture, the number of LSTM units and dropout value were decided by the grid search algorithm which resulted to use 32 units and 0 dropout.

Model	RMSE	Sliding Window	Epoch	Batches
LSTM 1	2182.45219	8	100	10080
LSTM 2	13626.18830	16	50	10080
LSTM 3	1387.99504	8	100	1440
LSTM 4	8601.42069	32	50	10080
LSTM 5	136.04120	8	50	1440

Table 4: Best results of the LSTM models and their configuration

On Figure 32a and 32b, the predictions and learning curves are shown of the best LSTM model.



(a) LSTM best prediction

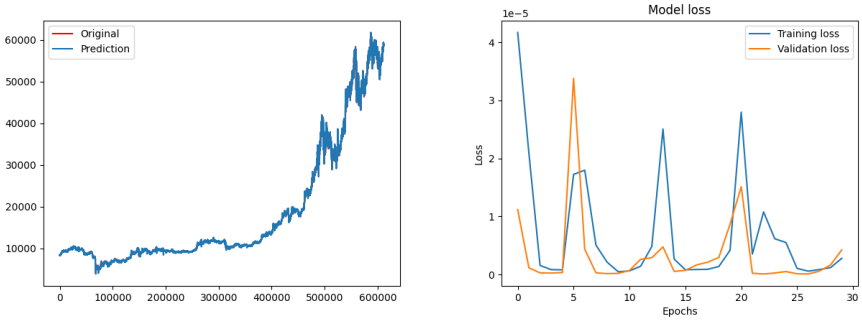
(b) LSTM learning curves

Learning curves are plots that show changes in learning performance over time in terms of experience. These curves of model performance on the train and validation datasets can be used to diagnose an underfit, overfit, or well-fit model. In this case, the losses are shown on the Y axis over the epochs on the X axis. The smaller the loss is the better the model learned. The article (17) explains well the learning curves.

From the different CNN architectures the second one had the less errors with one CNN layer with 64 filters and kernel size of 2, and with average pooling.

Model	RMSE	Sliding Window	Epoch	Batches
CNN 1	110.38555	16	100	10080
CNN 2	60.29650	8	50	10080
CNN 3	127.33865	8	100	1440
CNN 4	133.12386	16	50	1440
CNN 5	138.77161	8	50	1440

Table 5: Best results of the CNN models and their configuration



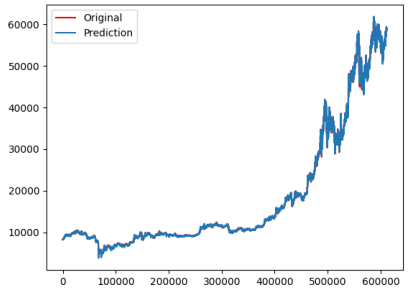
(a) CNN best prediction

(b) CNN learning curves

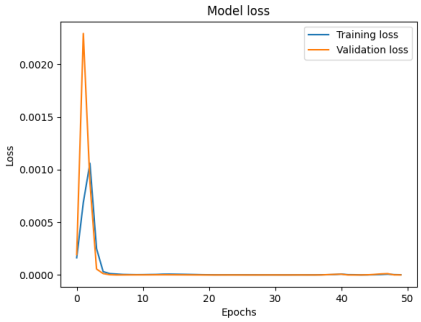
From the different CNN-LSTM architectures the fourth one had the less errors with two CNN layers (filter 32 and 64, kernel 2) and one LSTM layer (units 32).

Model	RMSE	Sliding Window	Epoch	Batches
CNN-LSTM 1	883.89890	32	100	10080
CNN-LSTM 2	416.73657	32	100	10080
CNN-LSTM 3	168.81699	32	100	1440
CNN-LSTM 4	78.12902	8	50	10080
CNN-LSTM 5	375.01558	16	50	1440

Table 6: Best results of the CNN-LSTM models and their configuration



(a) CNN-LSTM best prediction

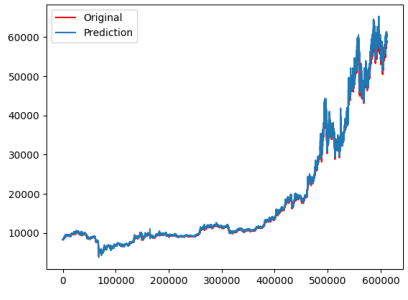


(b) CNN-LSTM learning curves

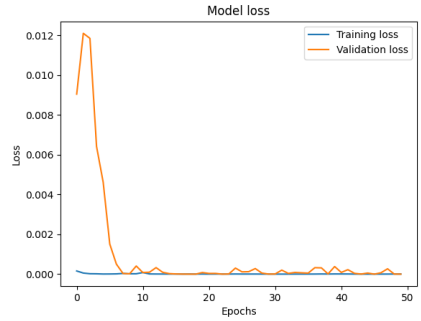
From the different ConvLSTM architectures the most simple one, the fifth model had the less errors which used only one ConvLSTM layer. In that architecture, the number of ConvLSTM filters and dropout value were decided by the grid search algorithm which resulted to use 64 filters and 0 dropout.

Model	RMSE	Sliding Window	Epoch	Batches
ConvLSTM 1	2085.95681	32	100	1440
ConvLSTM 2	10277.02356	32	50	10080
ConvLSTM 3	2305.30223	8	100	10080
ConvLSTM 4	12454.78492	8	50	10080
ConvLSTM 5	421.90044	8	50	10080

Table 7: Best results of the ConvLSTM models and their configuration



(a) ConvLSTM best prediction



(b) ConvLSTM learning curves

In conclusion, the learning curves show a good fit. In the case of other CNN models with the 10080 batch size (not explicitly documented) I experienced similarly shaky curves, however with the batch size of 1440 the curves were smooth.

The LSTM and ConvLSTM models behaved very similarly. Both among

the LSTM and the ConvLSTM models the most simple architectures had the less errors. It seems that the more complex the architecture was the worse the model learned. For example, the order of growing complexity matches with the growing error: LSTM5 (most simple), LSTM1 (in the middle), LSTM2 (most complex). LSTM1 and LSTM3 had the same architecture except that LSTM3 used dropouts as well. We can see from the tables that the error of LSTM3 became less, probably as a result of the dropout layer. The ConvLSTM results show similar behaviour to the LSTM in terms of the before mentioned examples.

In the CNN-LSTM architectures, we got the less error from the model which had the most CNN layers, and we got the most error from the model which had the most LSTM layers.

The CNN models showed the overall best performance. In their case, it didn't make a big difference in terms of the errors using dropout layers or more CNN layers.

We can also see the coincidence, that almost all of the best models were achieved with the same sliding window, epoch and batch size.

6 Technical specification

The proposed approach was implemented in Python 3.9 (6). To work with multi-dimensional arrays and perform high-level mathematical operations, I used the NumPy (4) library. For tabular data manipulation and analysis, I used the Pandas (5) library. From the scikit-learn (7) machine learning library, I used some specific functions and objects for data preprocessing. For model building, I relied on the high-level machine learning library, Keras (2) with TensorFlow (8) backend. For exploring and visualizing my results, I used the Matplotlib (3) library. The models were trained on UCloud (9) servers with 64 vCPU and 376 GB RAM to which I had student access.

The project consists of three Python files: *app.py*, *my_functions.py* and *my_models.py*. The *app.py* requires a model number (1-20) as its input which we specify as a command line argument. There are 20 models in total, the number of the models follows the order in which the models were introduced in this document. There were two ways of data scaling introduced in this document: scaling on a dataset level and scaling on a sliding window level. Because of that, the program trains and evaluates the chosen model twice - once with each way of data scaling.

The duration time of the trainings varied for the different models. The LSTM models took around 1-2 days, the CNN models around 4-8 hours, the CNN-LSTM models around 5-9 hours and the ConvLSTM models around 2 days.

The generated files of the best models of each network type were included to the hand-in.

References

[1] Stackoverflow: Gridsearchcv without cross validation. URL https://stackoverflow.com/questions/44636370/scikit-learn-gridsearchcv-without-cross-validation-unsupervised-learning.

[2] Keras. URL https://keras.io/.

[3] Matplotlib. URL https://matplotlib.org/.

[4] Numpy. URL https://numpy.org/.

[5] Pandas. URL https://pandas.pydata.org/.

[6] Python. URL https://www.python.org/.

[7] Scikit-learn. URL https://scikit-learn.org/stable/.

[8] Tensorflow. URL https://www.tensorflow.org/.

[9] Ucloud. URL https://cloud.sdu.dk/.

[10] J. Brownlee. How to develop convolutional neural network models for time series forecasting, . URL https://machinelearningmastery.com/how-to-develop-convolutional-neural-network-models-for-time-series-forecasting/.

[11] J. Brownlee. Convolutional neural networks for multi-step time series forecasting, . URL https://machinelearningmastery.com/how-to-develop-convolutional-neural-networks-for-multi-step-time-series-forecasting/.

[12] J. Brownlee. Crash course in convolutional neural networks for machine learning, . URL https://machinelearningmastery.com/crash-course-convolutional-neural-networks/.

[13] J. Brownlee. Cnn long short-term memory networks, . URL https://machinelearningmastery.com/cnn-long-short-term-memory-networks/.

[14] J. Brownlee. A gentle introduction to dropout for regularizing deep neural networks, . URL https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/.

[15] J. Brownlee. Dropout with lstm networks for time series forecasting, . URL https://machinelearningmastery.com/use-dropout-lstm-networks-time-series-forecasting/.

[16] J. Brownlee. How to grid search hyperparameters for deep learning models in python with keras, . URL https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/.

[17] J. Brownlee. How to use learning curves to diagnose machine learning model performance, . URL https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/.

[18] J. Brownlee. How to develop lstm models for time series forecasting, . URL https://machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/.

[19] J. Brownlee. Multi-step lstm time series forecasting models for power usage, . URL https://machinelearningmastery.com/how-to-develop-lstm-models-for-multi-step-time-series-forecasting-of-household-power-c

[20] J. Brownlee. How to use standardscaler and minmaxscaler transforms, . URL https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/.

[21] J. Brownlee. Time series forecasting as supervised learning, . URL https://machinelearningmastery.com/time-series-forecasting-supervised-learning/.

[22] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. *CoRR*, abs/1411.4389, 2014. URL http://arxiv.org/abs/1411.4389.

[23] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[24] M. Gurucharan. Basic cnn architecture: Explaining 5 layers of convolutional neural network. URL https://www.upgrad.com/blog/basic-cnn-architecture/.

[25] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[26] Keras. Timedistributed layer. URL https://keras.io/api/layers/recurrent_layers/time_distributed/.

[27] K. KRAVCHENKO. Convlstm: Convolutional lstm network tutorial. URL https://www.kaggle.com/kcostya/convlstm-convolutional-lstm-network-tutorial.

[28] C. Olah. Understanding lstm networks. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[29] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W. kin Wong, and W. chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting, 2015.

[30] Y. L. A. S. J. W. Wenjie Lu, Jiazheng Li. A cnn-lstm-based model to forecast stock prices. URL https://www.hindawi.com/journals/complexity/2020/6622927/.

[31] Wikipedia. Supervised learning, . URL https://en.wikipedia.org/wiki/Supervised_learning.

[32] Wikipedia. Time series, . URL https://en.wikipedia.org/wiki/Time_series.

[33] A. Xavier. An introduction to convlstm. URL https://medium.com/neuronio/an-introduction-to-convlstm-55c9025563a7.

[34] Y. B. Y. Lecun, L. Bottou and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE, vol. 86, no. 11*, page 2278–2324, 1998.

[35] Zielak. Bitcoin historical data. URL https://www.kaggle.com/mczielinski/bitcoin-historical-data.