ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

FAKULTA
STROJNÍ

# DIPLOMOVÁ PRÁCE

2021

ONDŘEJ
BIMKA

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta Strojní

Diploma Thesis

# Time series forecasting with recurrent neural networks

Bc. Ondřej Bimka

Supervisor: doc. Ing. Josef Kokeš, Csc.

Study programme: Automation and Instrumentation Engineering

Specialization: Automation and Industrial Informatics

August 2021

# MASTER‘S THESIS ASSIGNMENT

## I. Personal and study details

| | | | |
|---|---|---|---|
| Student's name: | **Bimka Ondřej** | Personal ID number: | **465361** |
| Faculty / Institute: | **Faculty of Mechanical Engineering** | | |
| Department / Institute: | **Department of Instrumentation and Control Engineering** | | |
| Study program: | **Automation and Instrumentation Engineering** | | |
| Specialisation: | **Automation and Industrial Informatics** | | |

## II. Master’s thesis details

Master’s thesis title in English:

**Time series forecasting with recurrent neural networks**

Master’s thesis title in Czech:

**Předpovídání časových sérií s využitím rekurentních neuronových sítí**

Guidelines:

1) Obtaining a suitable dataset
2) Suitable data preprocessing
3) Train models
4) Evaluate and compare models
5) Implement the best model to an API

Bibliography / sources:

[1] S. Hochreiter a J. Schmidhuber, „Long Short-Term Memory,“ Neural Computation, sv. 9, p. 1735–1780, 1997.
[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser a I. Polosukhin, Attention Is All You Need, 2017.
[3] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk a Y. Bengio, Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014.
[4] MAŘÍK, Vladimír, Olga ŠTĚPÁNKOVÁ a Jiří LAŽANSKÝ. Umělá inteligence. Praha: Academia, 1993-. ISBN 80-200-0496-3.

Name and workplace of master’s thesis supervisor:

**doc. Ing. Josef Kokeš, CSc., U12110.3**

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **30.04.2021** Deadline for master's thesis submission: **24.08.2021**

Assignment valid until: ______________

| | | |
|---|---|---|
| doc. Ing. Josef Kokeš, CSc.<br>Supervisor’s signature | Head of department’s signature | prof. Ing. Michael Valášek, DrSc.<br>Dean’s signature |

## III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

| | |
|---|---|
| . | |
| Date of assignment receipt | Student’s signature |

Declaration

I hereby declare that I completed this work without any improper help from a third party and without using any aids other than those cited. All ideas derived directly or indirectly from other sources are identified as such.

Date: Signature:

Acknowledgement

I wanted to thank everyone who supported me during the preparation and the work on my thesis. I also want to thank my supervisor doc. Ing. Josef Kokeš CSc.  for his insides and guidance. Last but not the least, I want to thank my family for providing me moral and material support during the course of my studies.

Abstrakt

Hlavním úkolem diplomové práce "Předpovídání časových sérií s využitím rekurentních neuronových sítí" je natrénovat modely na časových sériích dat, které se následně porovnávají a využijí v aplikaci.

Klíčová slova

Strojové učení, Neuronové sítě, Data science, Time-series data

Abstract

The main subject of the diploma thesis “Time series forecasting with recurrent neural networks” is to train models on typical time series data compare those models and use it in an application.

Keywords

Machine learning, Neural networks, Data science, Time-series data

## Table of contents

1. Introduction ....7
2. AI and ML basics ....8
3. Recurrent Neural Networks (RNN) ....9
   3.1 Long Short-Term Memory (LSTM) ....11
   3.2 Gated Recurrent Unit (GRU) ....13
   3.3 Transformers ....13
4. Dataset and training ....15
   4.1 Data preprocessing ....15
      4.1.1 First touch ....15
      4.1.2 Splitting our data ....16
      4.1.3 Normalizing data ....17
      4.1.4 Data windowing ....17
   4.2 Building model ....18
      4.2.1 Activation functions ....18
   4.3 Training ....19
      4.3.1 Optimizers ....19
      4.3.2 Loss ....21
      4.3.3 Metrics ....23
      4.3.4 Early stopping ....23
5. Practical part ....24
   5.1 Dataset's introduction ....24
   5.2 Normalizing my data ....29
      5.2.1 Logarithmic scale ....30
      5.2.2 Percentage change ....31
      5.2.3 Smaller normalized windows. ....31
      5.2.4 Conclusions on data scaling ....31
   5.3 Model's training ....32
   5.4 Models' performances ....34
   5.5 Interesting models and conclusions ....38
   5.6 Bigger model evaluation ....41
6. Crypto predicting API ....43
   6.1 Used libraries in predicting API ....44

6.2 API’s functionality ... 44
7. Hardware and software ... 47
8. Summary ... 48
9. References ... 50
List of Abbreviations ... 52
List of figures ... 53

# 1. Introduction

In this work I will explore the possibilities of using Recurrent Neural Networks on timeseries data. Natural Language Processing and timeseries predictions has been a very interesting topic for me for a long time. My intention is to explore how the RNNs work when applied to problems such as NLP and timeseries predictions.

In a daily practice we quite often use already pretrained models. This makes a routine work easier as it allows us to download already tested working models. We can also download pretrained model and fine-tune it by training only last layer for purpose we want. There are also libraries available which have already implemented models. In this thesis, I will train my own models and explore how they work and react to various settings.

Another thing which motivated me to do this work is the fact, that data science and machine learning, similarly to other disciplines require a certain degree of experience. Machine learning especially requires a lot of background knowledge and previous work experience. Exploring with data and own models allows me to dive into the field more.

One of the most common examples of timeseries data are stock market prices. I decided to use the various cryptocurrency prices, as unlike other markets, crypto markets are uninterrupted.  They are by nature very dynamic, which makes them very challenging, but also very interesting. Through using various cryptocurrency prices, I opted to predict the future Bitcoin price. As there are numerous approaches and not one explicitly right solution, I will try to record my whole process and compare various models and architectures. Both cryptocurrencies and neural networks are quite new phenomenon and bear a substantial potential for financial markets as well as other uses. Predicting future price from previous prices only will be very difficult for neural network to learn, so I am curious if RNNs will be able to learn at least some of the patterns.

Finally, I plan to implement one of the trained models to an application like API, as I really want to experience the whole process from selecting appropriate dataset, scaling the data up to the final application. Only when the data are ready, we can start to train the models. As models by themselves are devoid of use I want to wrap them into working application after I have properly evaluated them.

My biggest concern is that predicting the bitcoin price only using past prices may turn out to be too complicated. Regardless I still want to give it a try.  There is a distinct possibility that I may uncover   some hidden pattern between various cryptocurrency prices using RNN. When we look at cryptocurrency markets, we can observe that various cryptocurrencies behave in similar patterns. This observation leads me to believe that it will be at least partially possible to make such predictions.

## 2. AI and ML basics

There are two main types of artificial intelligence. Narrow AI and General AI. Narrow AI is the only type of AI that exists now. Sometimes Narrow AI is also called “Weak” because it is mostly able to perform only one specific task. We currently do not have an AI type which can do image classification and voice recognition and data processing synchronously, like humans do.

Even though narrow AI is not self-aware it is still very powerful and safe tool. Normally when we want to create some program or model, we must define it manually, however neural networks allow us to do it differently. Using ML (Machine Learning), we can basically create mathematical model based on data which we have. There are three main types of ML. Supervised learning is such where we label the data beforehand. We tell what our AI should see on the picture. For example, we label when there is a dog or cat when we want to teach our AI to recognize the picture of a dog or a cat. The second type is unsupervised learning which looks only for some patterns which our AI finds during the learning process. Unsupervised learning is mostly clustering, which will sort our data according to found patterns. The third one is reinforcement learning when there is an agent which gets rewarded for his actions.

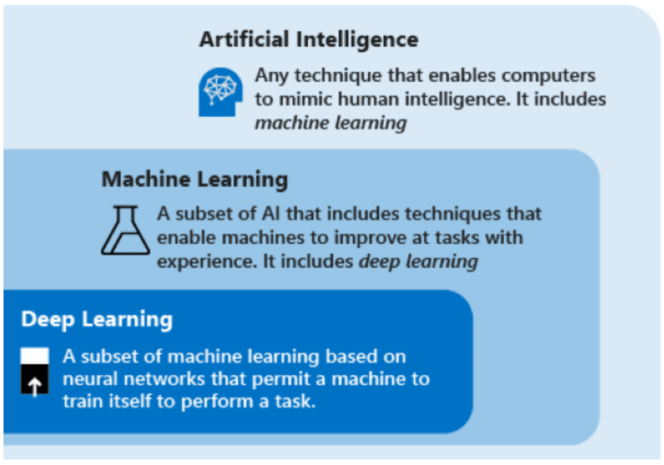

Artificial Intelligence
Any technique that enables computers to mimic human intelligence. It includes *machine learning*
Machine Learning
A subset of AI that includes techniques that enable machines to improve at tasks with experience. It includes *deep learning*
Deep Learning
A subset of machine learning based on neural networks that permit a machine to train itself to perform a task.

*Figure 1. AI overview [1]*

Alternatively, we can divide AI into categories according to Figure 1. Deep Learning which is a subclass of Machine Learning has been very popular lately. One of the biggest differences between normal ML and Deep Learning is that the second doesn’t require as much data pre-processing and feature extraction as we would normally do.  We basically take a lot of data and put it into our network. Our deep network will do the feature extraction and classification for us. It requires much more complicated structure with a lot of hidden layers and neurons. Thanks to this fact the results are very accurate, but sometimes we can’t be sure what patterns our network considered as important (black box). Training deep neuron networks takes a lot of time and requires a lot of data. Training deep networks could take weeks or months of computing. I could not really find an exact difference between “classical” machine learning and deep learning in terms of size of the network, but I really like Frank Dernoncourt’s definition: “Deep is a marketing term: you can therefore use it whenever you need to market your multi-layered neural network.” [2]

## 3. Recurrent Neural Networks (RNN)

Is a special type of Artificial Neural Networks which is specialized in sequential data. For this reason, RNN architecture is widely used in Natural Language Processing (tasks like language translation, named entity recognition) and for prediction tasks like price or weather forecast. Basically, we could use it with any data, where sequential order is important.

The main difference between RNN and normal feed-forward network is that RNNs output also considers previous timesteps using internal memory. Output is now affected not only by current input data, but also by memory, which holds relevant parts of previous data.

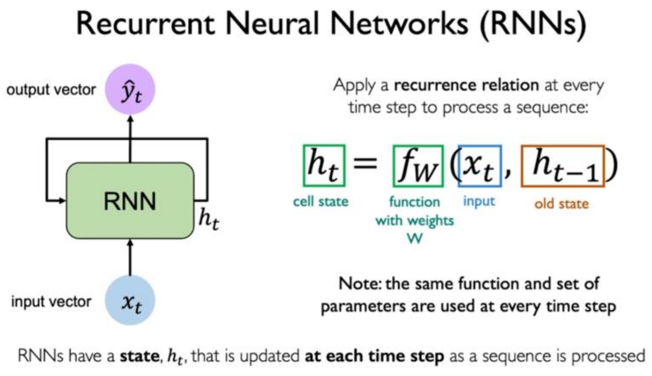

Recurrent Neural Networks (RNNs)
output vector
$\hat{y}_t$
RNN
$h_t$
input vector
$x_t$
Apply a recurrence relation at every time step to process a sequence:
$h_t = f_W(x_t, h_{t-1})$
cell state
function with weights W
input
old state
Note: the same function and set of parameters are used at every time step
RNNs have a state, $h_t$, that is updated at each time step as a sequence is processed

*Figure 2. RNN scheme [3]*

RNNs back propagate the overall loss through every individual timestep from the end to the beginning. This leads to a problem with gradients as we try to get to the cell state h (t=0) which means including many weights and gradient computations. One of the problems is called exploding gradient. This term describes a situation when many values (weights and gradients) are bigger than one. Solutions for this problem is called gradient clipping when we reduce big gradients. The second problem is the exact opposite to exploding gradients. When values are too small, we keep multiplying smaller and smaller numbers. This problem is called vanishing gradients and it leads to shorter relation recognition between initial time and latest time. To solve this problem, specialized cells have been developed.

Based on input and output length we can divide RNNs to following categories.

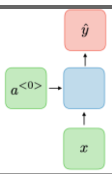
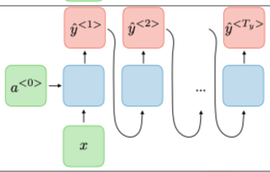
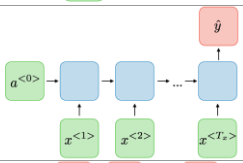
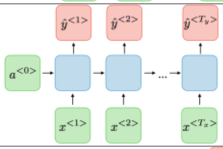
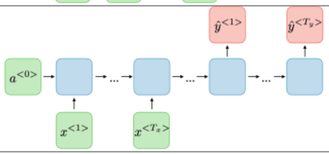
| **Lenght** | **Ilustration** | **Usage** |
|---|---|---|
| $len_{in} = len_{out} = 1$ |  | Traditional neural network |
| $len_{in} = 1$<br>$len_{out} > 1$ |  | Music generation |
| $len_{in} > 1$<br>$len_{out} = 1$ |  | Stock prediction |
| $len_{in} = len_{out} > 1$ |  | Named entity recognition |
| $len_{in} \neq len_{out} > 1$ |  | Translation / stock prediction |

Table 1. RNN categories overview [4]

Another type is the bidirectional RNN where NN go through data in both directions. This could be very useful, in cases when there was something important at the beginning of our data sequence and our model could give it smaller weight than it should. For example, if there was an important word at the beginning of the sentence, model could miss its value if we go through the data only in one direction.

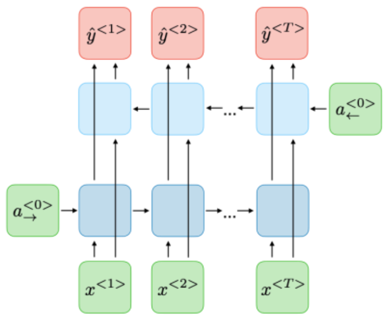


Figure 3. Bidirectional RNN scheme [4]

## 3.1 Long Short-Term Memory (LSTM)

LSTM cell was introduced by Sepp Hochreiter and Jurgen Schmidhuber in 1997 [5]. Even though this field of science is developing rapidly, LSTMs are still used nowadays, especially for timeseries data. It has been also very widely used in NLP tasks, however in this area it is being replaced by Transformers.

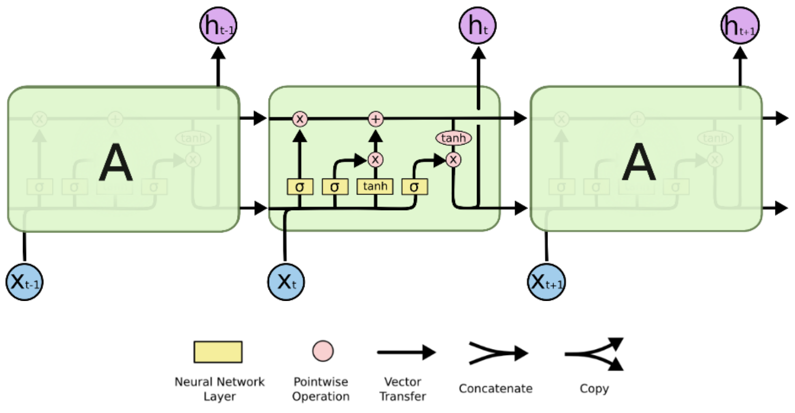


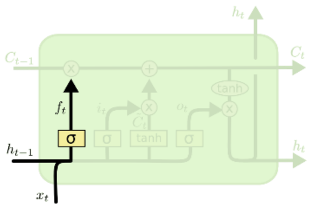
*Figure 4. LSTM scheme [6]*

Using gates LSTMs can control the information flow to the cell state. Gates consist of a sigmoid net layer and pointwise multiplication. Adaptive forget gates were introduced later in 1999. At the beginning of the training the gate activation is almost 1 so it takes whole input until there is more data cell could eventually forget. [7]

We could say that LSTM works in four steps.

1. Forget

Forgetting irrelevant history by passing it through the sigmoid gate. The output $f_t$ based on previous output $h_{t-1}$ and current input data $x_t$. Based on importance the output us between 0 and 1. Zero says to forget the data completely and to keep the whole information.

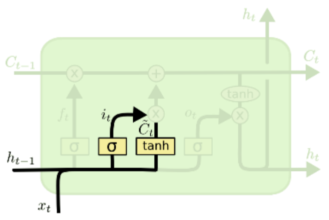


$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \qquad (1)$$

$W$ is weight matrix and $b$ is bias. Those values are randomly initialized and are updated during trainig.

2. Store

In this step LSTM stores the most relevant new information. The input gate layer is again a sigmoid layer which decides what to update. Second layer used in this step is tanh which produces vector $\tilde{C}_t$ which contains new candidates.

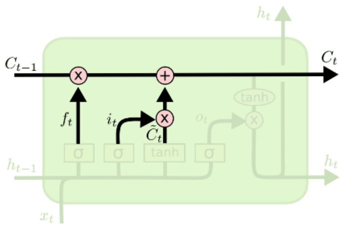


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \qquad (2)$$

$$\tilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \qquad (3)$$

3. Update

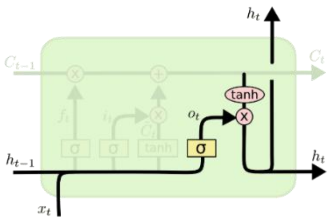
Internal cell state update by putting calculated values from previous steps together with pointwise operations. We obtain new cell state $C_t$ by using the old state and by adding “new” values $i_t * \tilde{C}_t$.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \qquad (4)$$

4. Output

The last step is an output generation. First, the sigmoid decides which states are going to be an output, then cell state goes through tanh. Finally, the state cell is multiplied by the sigmoid output, which ultimately gives us our result.

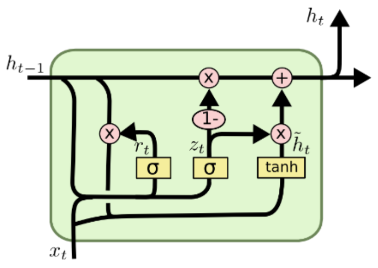


$$\sigma_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \qquad (5)$$

$$h_t = o_t * tanh(C_t) \qquad (6)$$

## 3.2 Gated Recurrent Unit (GRU)

GRU is slightly updated LSTM. It was introduced in 2014 by Kyunghyun Cho. It contains less parameters and it lacks an output gate. It could be also described as a lightweight LSTM cell, which also uses gates. [8]



$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \qquad (7)$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \qquad (8)$$

$$\tilde{h}_t = tanh(W \cdot [r_t * h_{t-1}, x_t]) \qquad (9)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \qquad (10)$$

*Figure 5 GRU scheme [6]*

First step is to determine how much of the past information will be passed to the future steps (7). Subsequently the reset gate decides what information to forget (8). Then only the relevant information is stored (9). Finally, the cell decides what will be the output to the next step (10). [8]

The main difference of GRU is that it does not have a controlled content of memory. The control of the information is done while calculating new candidate of the output. Judging just from its properties it is very difficult to say which one of LSTM and GRU is better. [9]

## 3.3 Transformers

Transformers don’t work on the same principle as gated recurrent cells as discussed before, (transformer does not have gates) yet they outperform gated cells in NLP applications, like NER, chatbots or translation. Language models like Bidirectional Encoder Representations (BERT) or very famous library Spacy for NLP, are now using Transformers. The main difference between classical RNNs like LSTM and GRU is that in transformers the sequence is passed as parallel.

As transformers are mostly used in NLP the first step is usually transforming words to vectors. This transformation takes place at the embedding block which transforms words to vectors in embedded space. Words however could be in different places of the sentence. Positional encoding is the next step which gives the context based on position of the word. Now, that we have our sequential data in vector form, we can continue to Encoder Block (Fig. 6 bottom left part). Here we have two important blocks. The first one is Multi-Head attention which gives us an attention vector to capture contextual relationship between data in sequence. The second one is Feed Forward net, which is classic feed-forward neural network applied to every attention vector.

Second part is Decoder, pictured on the right part of the transformer. The first step is very similar. We need to transform our data to numbers (if we haven´t done that yet). Decoder part consist of three main blocks. The first one is Masked Multi-Head attention, which is very similar to Multi-Head attention in encoder, however, is masked to provide parallel training.

This setup makes training faster. The data needs to be masked in this step to engage learning. Then the second attention block (encoder-decoder) determines how is the data from previous block related. In NLP this block determines which words from different languages are related. At the output we have feed forward linear layer which feeds data to SoftMax. SoftMax returns probability distribution. The output in the end could be for example a word with the highest probability. [10]

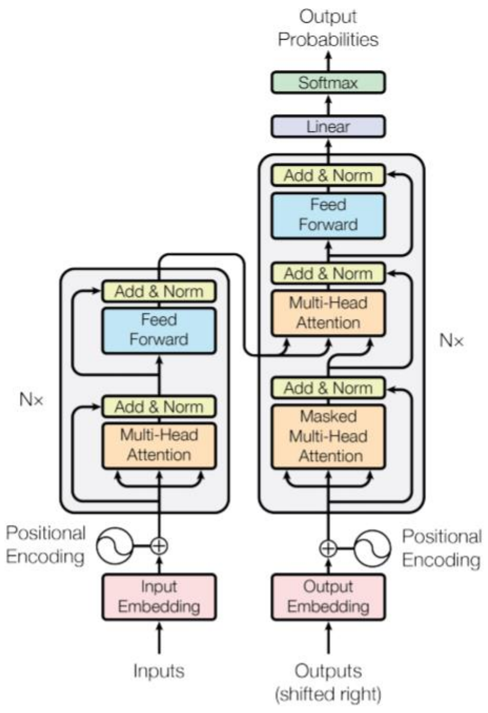


*Figure 6 The transformer – architecture [10]*

Another advantage of transformers is that since the inputs are whole sequences, it works better on GPUs. GPUs calculate simultaneously and data is inserted to gated cells data sequentially which makes training slower (however gated cells were still very fast on my GPU).

# 4. Dataset and training

Some subtasks must be done first to train RNNs. The first step and very important one is finding a dataset with enough data. Without a good dataset the network won't be able to learn properly. Then usually some preprocessing is needed. Time of training could vary a lot depending on used dataset size, architecture, its settings and of course on hardware. Finally, there should be some model evaluation.

## 4.1 Data preprocessing

The main purpose of data preprocessing is to prepare data to achieve the best results. Data can come in many formats, but computers work only with numbers. In this work I have been working with numbers, so I had to do some scaling and cleaning. Normally when I work with data, I convert it to Panda's data frame when possible. Pandas is one of the most popular data analysis libraries, which allows developers to work with data more easily. Loading files in CSV or TSV formats is very easy with Pandas. When we work for example with pictures when training CNN the process would be different. However, the idea is the same -- to create as much accurate generalized model as we can. Ideal model will work on new data with the same accuracy as with training data, though unseen data almost every time gets slightly worse results. Our goal is to make this difference as small as possible. [11]

### 4.1.1 First touch

Every time I work with data, the first thing I do, is to try to understand it as much as I can. That is why the first thing I do while preprocessing is to plot the data.

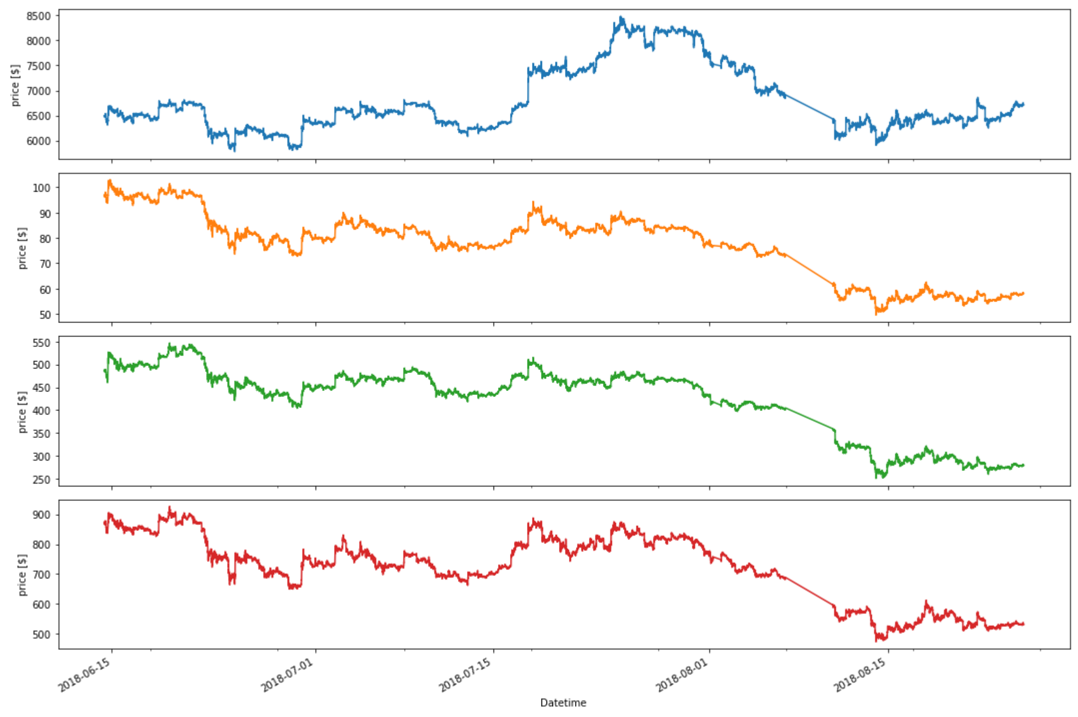
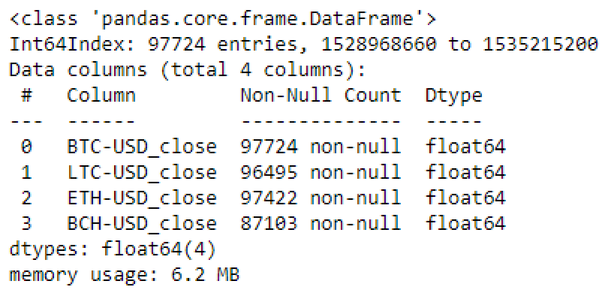


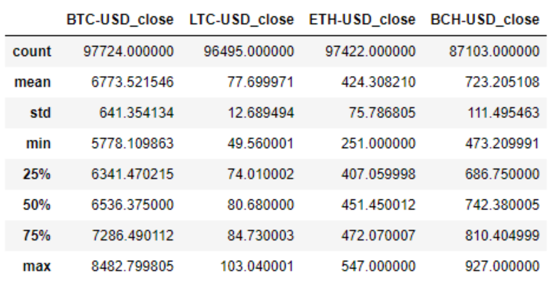
*Figure 7 Example of timeseries data visualization*

As we can see in this case there is some unusual behavior around 2018-08-07. That part could confuse our neural network during training. For that reason, it would be better to remove this part. After plotting we can obtain some info using pandas prebuild functions. Using those functions, we can check for empty values, or other problems, like wrong datatypes.



```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 97724 entries, 1528968660 to 1535215200
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   BTC-USD_close  97724 non-null  float64
 1   LTC-USD_close  96495 non-null  float64
 2   ETH-USD_close  97422 non-null  float64
 3   BCH-USD_close  87103 non-null  float64
dtypes: float64(4)
memory usage: 6.2 MB
```

*Figure 8. Data overview using info() function.*



| | BTC-USD_close | LTC-USD_close | ETH-USD_close | BCH-USD_close |
|---|---|---|---|---|
| **count** | 97724.000000 | 96495.000000 | 97422.000000 | 87103.000000 |
| **mean** | 6773.521546 | 77.699971 | 424.308210 | 723.205108 |
| **std** | 641.354134 | 12.689494 | 75.786805 | 111.495463 |
| **min** | 5778.109863 | 49.560001 | 251.000000 | 473.209991 |
| **25%** | 6341.470215 | 74.010002 | 407.059998 | 686.750000 |
| **50%** | 6536.375000 | 80.680000 | 451.450012 | 742.380005 |
| **75%** | 7286.490112 | 84.730003 | 472.070007 | 810.404999 |
| **max** | 8482.799805 | 103.040001 | 547.000000 | 927.000000 |

*Figure 9. Data overview using describe() function*

The easiest way of getting rid of NaN values is just to drop rows containing those values. This could be done only if we have enough data available. Removing rows may help with overfitting, however when working with timeseries data sometimes it is better to replace occasional NaN value with a value from previous step. There are again prebuild functions for replacing data where we can specify how to replace missing data. Of course, the best option is to have data without missing values, but that is not always possible.

### 4.1.2 Splitting our data

The most usual way to split data is to split it into three parts: train, validation and test set.

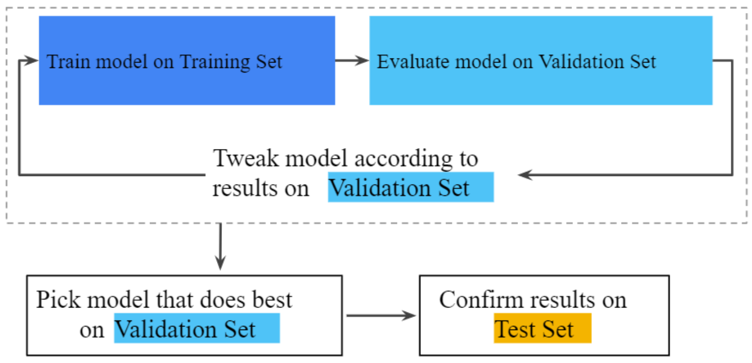

Train model on Training Set
Evaluate model on Validation Set
Tweak model according to results on Validation Set
Pick model that does best on Validation Set
Confirm results on Test Set

*Figure 10. Workflow [12]*

The reason why it is the best to split it into three parts is obvious from the figure above. If we used only two datasets there would be a problem. We would tweak our model to obtain the best result on validation set. As a result, the model would be adapted to it. That’s why we use the third test set. Only on test set we can evaluate our model, because that part of data is unknown and reflects actual behaviour on data which could come from the outside world.

Based on my experience, I usually split my data with this ratio: (70%, 20%, 10%) train, validation, test. Using this exact ratio is not obligatory and we should always adjust it to our actual available data.

### 4.1.3 Normalizing data

Crypto market is very dynamical and price differences are very huge. That is the reason why it is necessary to use proper normalization/scaling. For data normalization I used sci-kit learn. [13] This library provides useful tools for data pre-processing.

- Z-score (standard score)

$$z = \frac{x - \mu}{\sigma} \quad (11)$$

Where $x$ is observed value, $\mu$ is mean of the sample and $\sigma$ is the standard deviation. Even though z-score is quite popular, I also considered using different scalers, which might have some advantages, but this one seemed the best.

- Min-max normalization

Is defined by following formula

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \quad (12)$$

Where $x'$ is scaled value, $x$ is original value and $min\ (x)$ is minimal value from input values and $max\ (x)$ is maximum value from input values. [13]

### 4.1.4 Data windowing

The last step which is necessary to do before it is possible to start learning is to split data into separate windows with input data and labels.

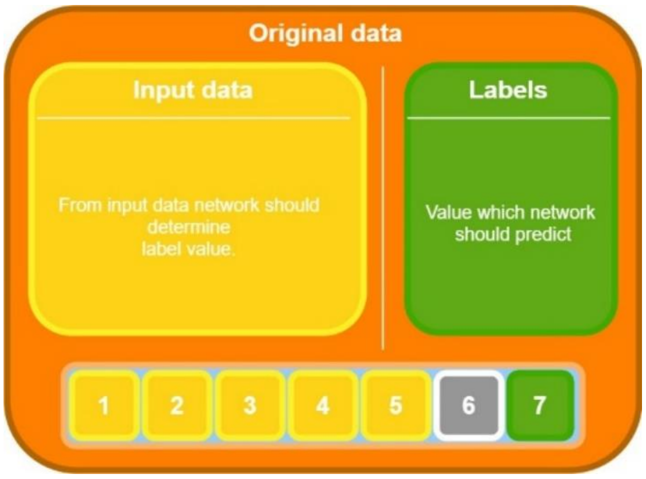

Original data
Input data
From input data network should determine label value.
Labels
Value which network should predict
1
2
3
4
5
6
7

*Figure 11. Data windowing*

In Figure 11 there is an illustration of data windowing. In this case we split the dataset into windows with 7 elements in every window and from 5 first elements the model will try to predict 7th value.

After we split our data into windows with desired size, we usually shuffle separated windows, so they don't go to our trained model in the same order, but the input are tensors with size (batch, length, columns).

## 4.2 Building model

Next step is to build the neural network model. Here we define size of the actual network and its properties like type, activation function, recurrent activation function and more.

### 4.2.1 Activation functions

The shape of the activation functions defines the output of the neuron. If we consider step function, based on neuron sum the output will be 0 or 1. Activation functions have a big impact on neural network performance, because if we pick the wrong activation function the model won't work. There are many available activation functions, and I will only focus on those which I find important.

| **Name** | $\boldsymbol{f(x)}$ | $\boldsymbol{f'(x)}$ | **Range** |
|---|---|---|---|
| **Sigmoid** | $\sigma(x) = \frac{1}{1+e^{-x}}$ | $\frac{d\sigma(x)}{dx} = \frac{e^x}{(1+e^x)^2}$ | $(-\infty, \infty)$ |
| **Tanh** | $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ | $\frac{d\tanh(x)}{dx} = 1 - \tanh(x)^2$ | $(-1,1)$ |
| **ReLU** | $\begin{cases} 0 & if\ x \leq 0 \\ x & if\ x > 0 \end{cases}$ | $\begin{cases} 0 & if\ x < 0 \\ 1 & if\ x > 0 \end{cases}$ Undefined if x $= 0$ | $[0, \infty)$ |
| **Lin** | x | 1 | $(-\infty, \infty)$ |
| **Step** | $\begin{cases} 0 & if\ x < 0 \\ 1 & if\ x \geq 0 \end{cases}$ | $\begin{cases} 0 & if\ x < 0 \\ undefined & if\ x = 0 \end{cases}$ | $\{0,1\}$ |

*Table 2. Activation functions [14]*

Functions above (Table 2) plotted can be seen on picture below (Fig. 12).

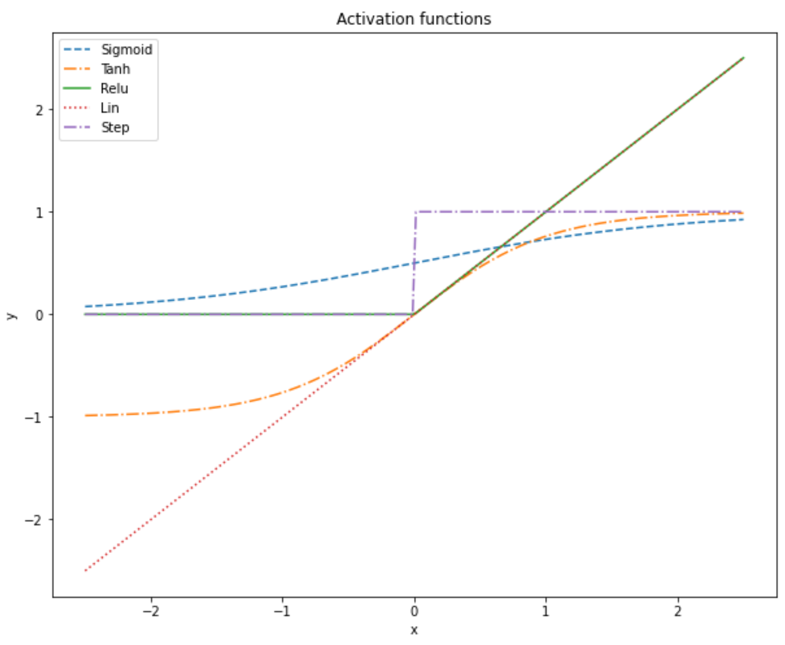


*Figure 12 Plotted activation functions*

In this work I have used only two combinations of activation functions. Those were ReLU as activation with Sigmoid as recurrent activation, however, to use cuDNN implementation (GPU) I switched to tanh as an activation and sigmoid as recurrent activation, because only this combination was supported for GPU training. [15]

## 4.3 Training

Now, when the data is processed, we can start training our models. The last step before actual training is to compile model. Here we must specify optimizer, number of epochs, callbacks, loss end metrics. All mentioned parameters have impact on final model performance, but in my opinion, optimizers have bigger influence on model learning, which is why I am going to talk about them a little bit more.

### 4.3.1 Optimizers

Optimizers are so important, because they optimize parameters of our models, such as weights and, they could tweak learning rate, in order to reduce loss of the model.

- Gradient descent

Probably the most famous optimizer. It is an iterative optimalization algorithm for finding local minimum. Gradient of a function $F$ in point $\boldsymbol{p}$ is a vector defined as

$$grad\ F(\boldsymbol{p}) = \left(\frac{\partial F}{\partial x_1}(\boldsymbol{p}), \ldots.., \frac{\partial F}{\partial x_n}(\boldsymbol{p})\right) \quad (11)$$

As It might be seen from (eq. 13) gradient is partial derivative with respect to all variables (those derivations must exist). [16] In other words, it is a vector which shows us the direction of maximum function growth. If we know the direction the maximum growth, we also know the direction of maximum decrease. Gradient is usually expressed with ∇ so formula (3) could be written as

$$\nabla F(\boldsymbol{p}) \qquad (12)$$

Then using following iterative method, we get closer to local minimum.

$$\boldsymbol{p}_{k+1} = \boldsymbol{p}_k - \alpha_k\, \nabla F(\boldsymbol{p}_k), k \geq 0 \qquad (13)$$

$\alpha_k$ is small enough that next step $\boldsymbol{p}_{k+1}$ is smaller than current step $\boldsymbol{p}_k$. Then we obtain following sequence. Where last step would be ideally local minimum.

$$F(\boldsymbol{p}_0) \geq F(\boldsymbol{p}_1) \geq F(\boldsymbol{p}_1) \geq \cdots \qquad (14)$$

As it has been pointed out $\alpha_k$ changes at every step, so its value must be calculated. Using secant method, we can approximate (5) using following method.

$$\boldsymbol{p}_{k+1} = \boldsymbol{p}_k - F(\boldsymbol{p}_k)\frac{\boldsymbol{p}_k - \boldsymbol{p}_{k-1}}{F(\boldsymbol{p}_k) - F(\boldsymbol{p}_{k-1})} = \boldsymbol{p}_k - F_k \frac{\Delta \boldsymbol{p}_k}{\Delta F_k} \qquad (15)$$

For $k > 0$ we need to know vectors $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$. From (5) and (7) we express following

$$\boldsymbol{p}_{k+1} = \boldsymbol{p}_k - \alpha_k\, \nabla F(\boldsymbol{p}_k) \triangleq -F_k \frac{\Delta \boldsymbol{p}_k}{\Delta F_k}\,,\ k > 0 \qquad (16)$$

From (8) $\alpha_k$ is

$$\alpha_k = F(\boldsymbol{p}_k)\frac{\Delta^2 \boldsymbol{p}_k}{\Delta^2 F_k}\bigg|_{k\geq 2} \triangleq F(\boldsymbol{p}_k)\|[\nabla^2 F(\boldsymbol{p})]_k^{-1}\| \ , k \geq 0 \qquad (17)$$

Nice analogy to gradient decent is standing on a hill with closed eyes and trying to find the highest or lowest point. We can follow steepest direction and after some time we will find highest, or lowest local point. If steps are too small it is going to take longer, but if steps are too big it is also a problem because we will be walking around the minimum or maximum point. Second problem is that we can't be sure if we are on global maximum, because we don't see other hills. For this reason, there are different optimizers.

Now let's explain for what purpose we use optimizers in ML. As we can see in figure below, using optimizers the network tweaks weight and learning rate to obtain minimal loss using gradient. It is important that learning step is not too small, because then learning takes too long. Nevertheless, learning step should not also too big.

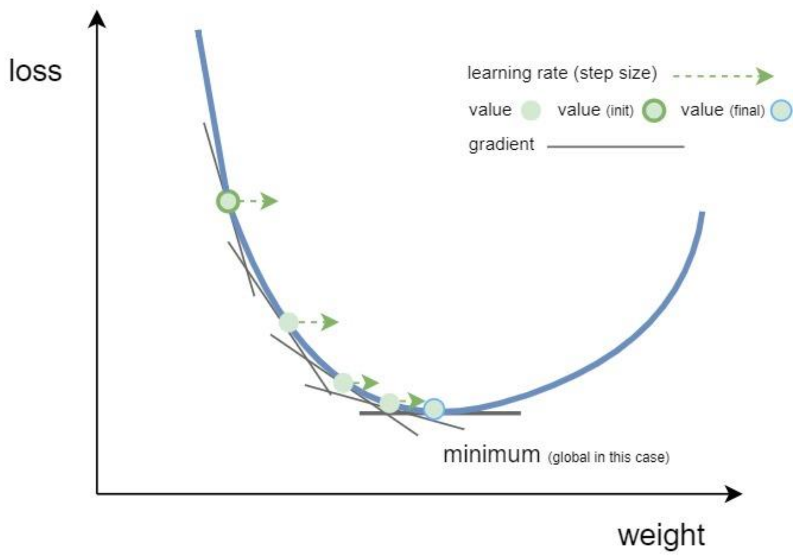


*Figure 13 Gradient descent in ML*

Dark green point on the left side is the initial weight value. This is usually randomly chosen value when compiling neural network architecture. Blue dot is final value located in local minimum, which is also a global minimum in this case, however, does not have too necessarily global.

- Stochastic gradient descent (SGD)

As the name implies, SGD is a stochastic approximation of gradient descent. SGD is also sometimes referred as online gradient descent (batch size = 1). If we calculate gradient descent for every sample in single iteration it is called batch gradient descent. That means we choose batch size equal to number of samples in dataset. This approach is quite smooth, converges well to a minimum, but is very slow.

Classical stochastic gradient descent uses only one sample of the dataset per iteration. This is computationally faster, but it is quite noisy and after fast start at the beginning it oscillates around local minimum.

Tradeoff between options above is mini-batch size gradient descent. According to "Mini-batch size of 32 is a good default value" [17]. This means that weights are updated after considering batch size (32) samples of data.

One epoch is when our whole dataset goes through NN. Iteration is number of batches necessary to complete one epoch. After every iteration is one gradient update. Every training step is iteration.

$$epoch = batch\,size * iterations$$

- AdaGrad

Some optimizers generally focus on more occurring features and the rest are set to zero, however even rarely occurring features could be informative. This problem AdaGrad solves by giving frequently occurring features lower learning rates than to more rare features. [18]

- Adam

Adaptive Moment Estimation (Adam) is probably one of the most universal optimizers. It requires only first order gradient. It is designed to combine two popular methods which are AdaGrad and its ability to deal with sparse gradients and RMSProp which deals well with non-stationary objectives. [19]

- Adamax

Adamax was introduced with Adam as its variant and is based on infinity norm. From the introduction paper it is able to outperform Adam on models with embeddings. [19] [15]

- FTRL

This optimizer was developed by Google to predict ad click through rates. Ad-related data tends to be very sparse with only few non-zero values, which means that FTRL has good sparsity and convergence properties. I have also seen some visualizations and this optimizer converged very rapidly. [20]

### 4.3.2 Loss

Loss functions are used by optimizers to find minimum.

- Mean squared error (MSE)

Defined as sum of squared difference between predicted and actual values divided by number of examples.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_t(i) - y_p(i))^2 \quad (19)$$

- Mean absolute error (MAE)

Average value of absolute difference between predicted and actual value.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}\left|y_t(i) - y_p(i)\right| \quad (20)$$

- Huber

Huber loss is defined as following

$$Huber\ Loss\ (x) = \begin{cases} 0.5 * x^2 & if\ |x|\ \leq d \\ 0.5\ *\ d^2 + d\ * (|x| - d) & if\ |x| > d \end{cases} \quad (21)$$

$$x = y_t(i) - y_p(i) \quad (22)$$

It behaves mostly like absolute error except for smaller values, where is this function more quadratic. Shape of this function is specified with $d$ parameter, which tells us if It should be more sensitive to outliners like MSE or less like MAE. [21]

- LogCosh

Computes the logarithm of the hyperbolic cosine of the prediction error. [15]

$$LogCosh = log\left(\frac{1}{2}(e^{-x} + e^{x})\right) \equiv log(\cosh(x)) \quad (23)$$

$$x = y_t(i) - y_p(i) \quad (24)$$

LogCosh is very similar to Huber but has one big advantage which is that it is twice differentiable everywhere. This could be very useful, because some boosting gradient methods use this second derivative to find the minimum.

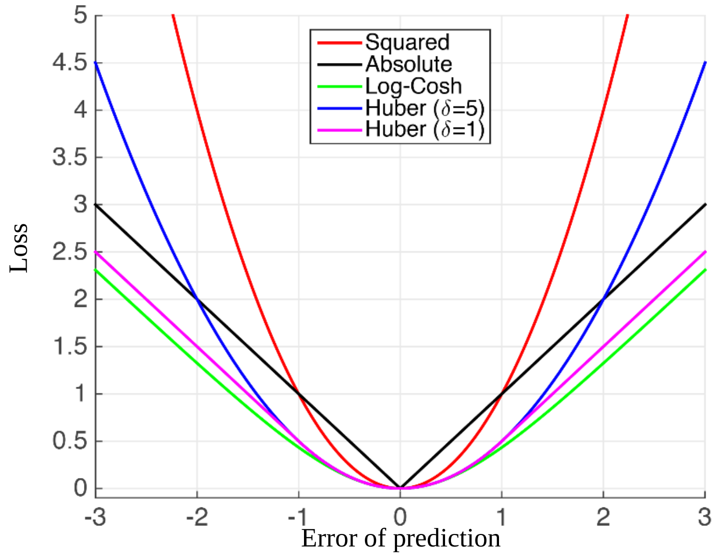


*Figure 14 Visualisation of loss functions above [22]*

### 4.3.3 Metrics

In TensorFlow metrics are used for model evaluation. Unlike loss, metrics don't affect model learning and are only for performance checking.

### 4.3.4 Early stopping

Another very useful keras' feature is EarlyStopping. It is a subclass of Callbacks which allows us to access various stages of our learning process. EarlyStopping is specialized on monitoring metrics and can eventually stop learning when its conditions are fulfilled. I used this class very often, because it allowed me to set higher number of epochs and if our model satisfies our goals. [23] [15]

```
tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', min_delta=0, patience=0, verbose=0,
    mode='auto', baseline=None, restore_best_weights=False
)
```

Because in first part of this work I was focusing on comparing various models and architectures I was monitoring validation loss and if the model did not improve for three epochs then training was stopped and model with the best validation loss was restored and saved into selected directory.

# 5. Practical part

In this work I have used several datasets, preprocessing options, and model architectures. I am going to describe the whole process with reasons why I used some method and explain why something works and something does not.

This work has been also an iterative process because I went through many various sources and tried various libraries.

The first thing I would like to point out is quick introduction on how crypto market and BTC price behave in general, since there are facts we should consider when working with crypto data.

As can be noted in figure below like all other markets, crypto market in general has two main periods. One of them is called "bull market" which is when prices go up. This trend when everything goes rapidly up can be observed at the end of 2017 and at the end of 2020. The second market is "bear market". That is when market goes down or it does not change much.

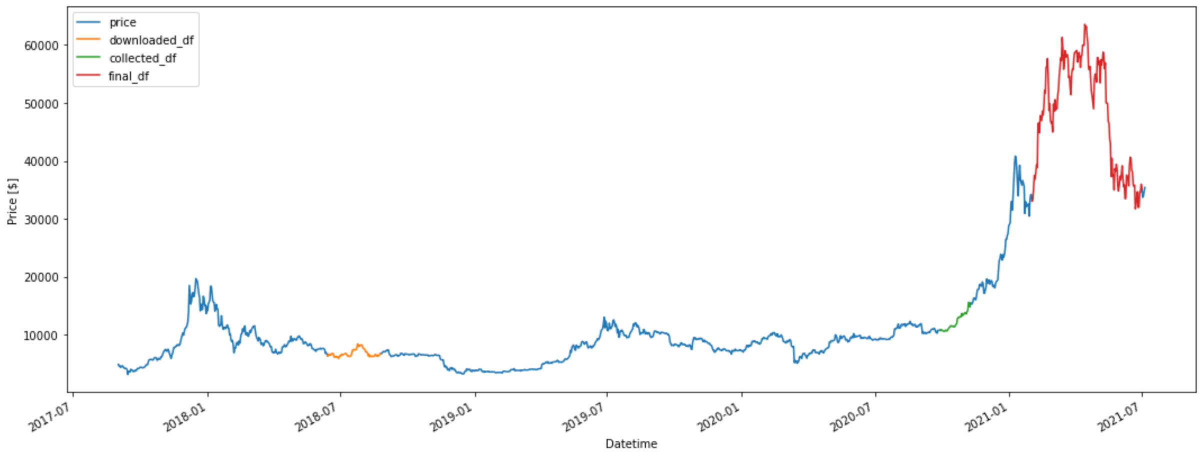


*Figure 15. BTC price with highlighted datasets*

## 5.1 Dataset's introduction

I used 3 datasets in my work. I have all of them stored as a CSV file formats on my local computer. They can be seen on Figure 15.

The first dataset was downloaded from [24]. This dataset contains 4 files with following cryptocurrencies: BTC, ETH, LTC and BCH in 4 different CSV files. Every file has around 95000 rows.

| time | low | high | open | close | volume |
|---|---|---|---|---|---|
| **1528968660** | 871.65 | 871.73 | 871.65 | 871.72 | 5.675361 |
| **1528968720** | 870.86 | 871.72 | 871.72 | 870.86 | 26.85658 |
| **1528968780** | 870.1 | 871.09 | 871.09 | 870.1 | 1.1243 |
| **1528968840** | 868.83 | 870.95 | 868.83 | 870.79 | 1.749862 |

*Table 3. Example of downloaded data.*

From those 4 csv files I took only close prices and created new dataframe from them. Visualization and basic information about this dataframe is in Figures 7, 8 and 9.

This first dataset was quite good for learning because it had enough data, was relatively balanced in terms of price evolution. The only problem is that for something so dynamical like crypto market, dataset which is from 2018 is quite outdated. So, I decided that I would learn my models on this dataset and then see how it works on more relevant data. Then I can compare results and say if models are generalized enough, or eventually retrain models on current data.

Therefore, I created another dataset using yfinance Python library https://github.com/ranaroussi/yfinance. This library is very easy to use, and it is free without necessity to have any account. With some additional code I have created Cryoto_dataset.ipynb file which  creates a csv file containing desired data. It also returns more data than just close prices, however I only use them. Unfortunately, 1-minute data resolution is only available for past 7 days. After few weeks of data collecting, I managed to create a dataset with approximately 58393 rows.

| Datetime | BTC-USD_close | LTC-USD_close | ETH-USD_close | BCH-USD_close |
|---|---|---|---|---|
| **2020-09-28 00:00:00+01:00** | 10756.69 | 45.95008 | 355.0525 | 227.4126 |
| **2020-09-28 00:01:00+01:00** | 10756.32 | 45.93516 | 355.1282 | 227.4453 |
| **2020-09-28 00:02:00+01:00** | 10756.92 | 46.00699 | 355.126 | 227.4328 |
| **2020-09-28 00:03:00+01:00** | 10758.65 | 45.92831 | 355.2801 | 227.4888 |

*Table 4. Part of my collected dataset*

It contains same indices as the first downloaded dataset. The only problem here is that prices started to grow rapidly after few weeks as visible in figure below. If the plot continued the growth would be even steeper, because BTC price moved from 18k to 65k in only 4 months. Hence, I stopped collecting data and decided to use only some parts of this dataset for evaluating models.

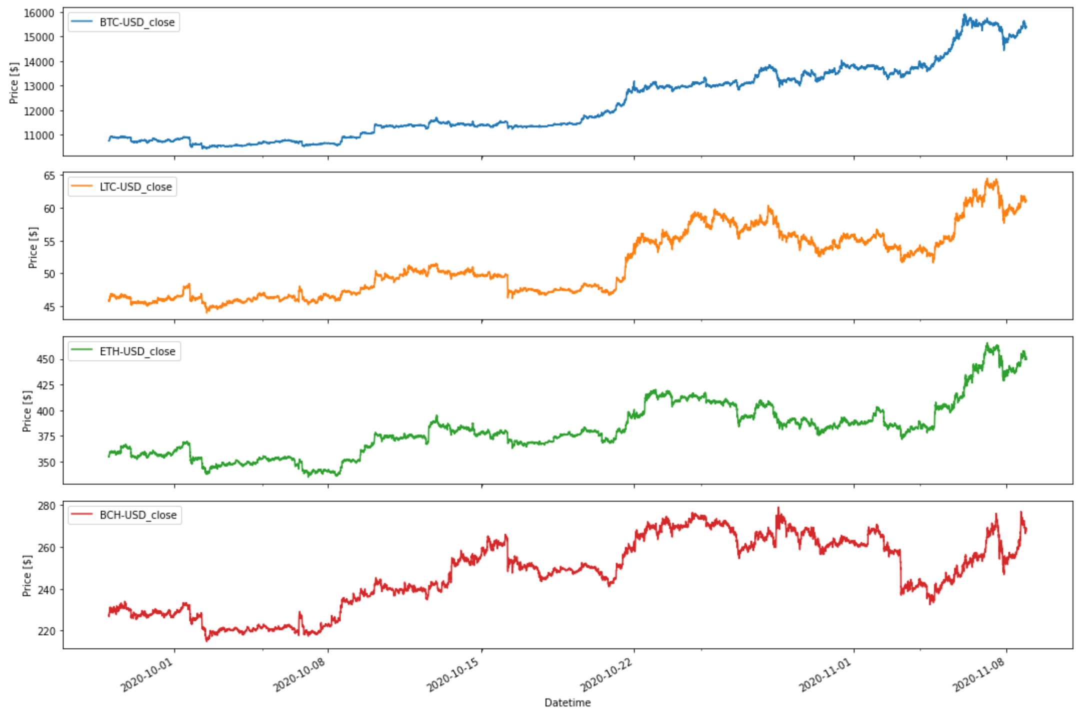


*Figure 16. Visualized collected dataset*

Unbalanced price growth was one of my biggest obstacles during this work, because my models were trained on data from “bear market” and after few weeks “bull market” started and everything went up fast. That meant that I had models which were trained on “bear market” data, but I also wanted to validate models on current incoming data. Fortunately, if we take some smaller window size the growth is not so noticeable there, however it could probably reflect in model performance.

Using datasets above I was able to train models check the performance and survey if they are generalized enough to perform similarly on completely new data.

After some time, I found another option which was getting data from Binace, which is a cryptocurrency exchange. The only disadvantage is that you need to create an account there, but there is no need to buy anything and then you have access to the API. After creating an account, it is possible to get a lot of useful tools including historical data of desired cryptocurrencies. For easier communication I used python-binance library https://github.com/sammchardy/python-binance. I again created a script in Jupyter notebook to make creating dataset easily. Now we just select ratio, date and what data we want to download, and it creates csv files into selected directory. [25]

| timestamp | open | high | low | close | volume | quote_av | trades |
|---|---|---|---|---|---|---|---|
| **01/01/2020 00:00** | 7169.12 | 7169.12 | 7162.19 | 7162.2 | 0.85156 | 6104.387 | 16 |
| **01/01/2020 00:01** | 7162.2 | 7162.2 | 7162.2 | 7162.2 | 0 | 0 | 0 |
| **01/01/2020 00:02** | 7160.33 | 7160.33 | 7160.33 | 7160.33 | 0.001675 | 11.99355 | 1 |
| **01/01/2020 00:03** | 7160.33 | 7160.33 | 7160.33 | 7160.33 | 0 | 0 | 0 |

*Table 5. Example of data from Binance.*

Thanks to this data source was able to easily obtain a lot of data, so I decided to include more than cryptocurrencies than in the previous datasets. First, I selected all available BUSD pairs (BUSD is a stable coin, which means that it is backed 1:1 to US dollar) from Binance. Then for all those pairs I selected close prices from 2021-06-20 – 2021-07-04 (two weeks) and calculated correlation between all those pairs vs BTCUSD. Getting longer time interval would take too much time for so many pairs, so I decided to take two weeks and for those with higher correlation I will do some further examination.

|  | correlation_with_btc | pair |
|---|---|---|
| **190** | 0.908486 | TWTBUSD_close |
| **189** | 0.895470 | TRBBUSD_close |
| **188** | 0.877552 | XLMBUSD_close |
| **187** | 0.875971 | DGBBUSD_close |
| **186** | 0.866240 | ALGOBUSD_close |
| **...** | ... | ... |
| **4** | -0.010913 | TUSDBUSD_close |
| **3** | -0.084367 | MIRBUSD_close |
| **2** | -0.133281 | AUDBUSD_close |
| **1** | -0.145807 | EURBUSD_close |
| **0** | -0.287440 | USDCBUSD_close |

*Table 6. All BUSD pairs with correlation to BTC*

I ordered them from the highest correlation to the lowest. To visualize if it works, I took TWTBUSD_close with correlation:  0.9084862137358767 and AUDBUSD_close correlation is -0.13328093134182173 and plotted the data.

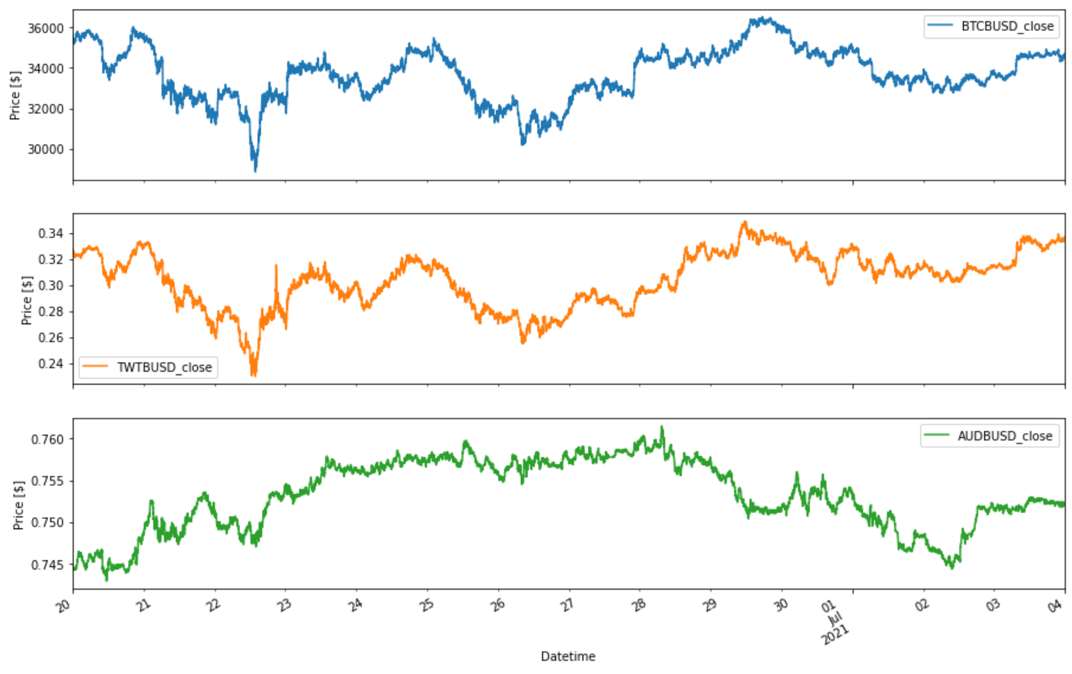


*Figure 17. Example of correlated and uncorrelated prices*

I took 30 pairs with the highest correlation and got data from 1st Jan 2021 until 1st July 2021. Then I calculated correlation again. From them I picked 7 with the highest correlation. All data plots are in Data_explore.ipynb.

The fact that there is constantly new data available eventually became problematic. As I did not know which architecture is the best, I would have preferred a static dataset to train models and to compare them. For this purpose, specifically, I used older downloaded dataset. Another advantage of using this old dataset is that I can then easily validate if models are generalized enough, on newer data.

| **Dataset** | **Used for** | **Indices** | **Size** |
|---|---|---|---|
| **Downloaded dataset** | Training smaller models and their performance comparison | BTC, LTC, ETH, BCH | (86117 x 4) |
| **Collected dataset with yfinance (collected)** | Validation of smaller models | BTC, LTC, ETH, BCH (XLM, ADA) | (58393 x 4) |
| **Created dataset using Binance API (final_df)** | Training/validation of big models | BTC, OCEAN, ZRX, ATOM, BNT, ALGO, TWT, SUSHI | (216177 x 8) |

*Table 7. Dataset's overview*

## 5.2 Normalizing my data

In this part I would like to go through various options and problems which I encountered during normalizing crypto prices for price prediction using recurrent neural networks.

Normally I would split the data to three different parts. Train, test, and validation datasets. Then I would scale the training dataset and using same scale ratio I would then normalize two remaining datasets.  However, this approach does not work very well in this case.

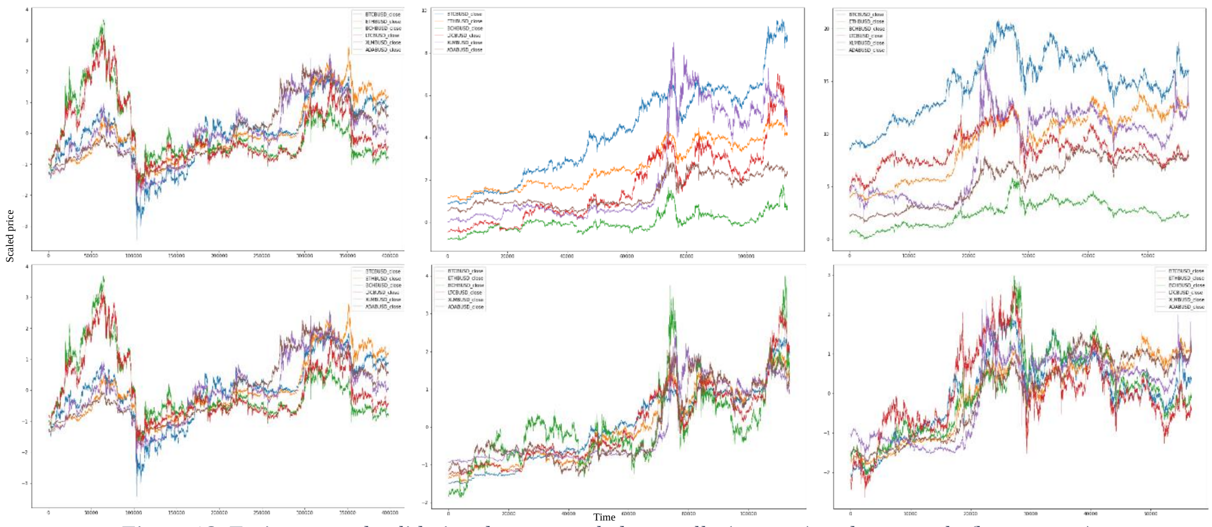
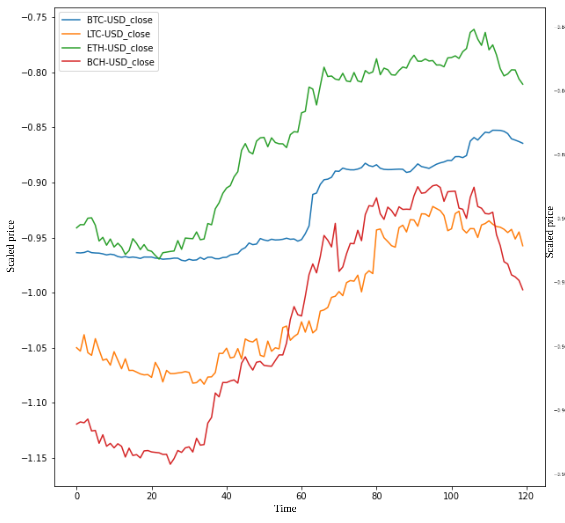


*Figure 18. Train, test and validation datasets scaled normally (top row) and separately (bottom row)*

Top row in figure 18 contains scaled data using the correct way. On the left, there is my train part, in the middle there is a validation part, and, on the right, there is a test part. The bottom three graphs are the same data, but they don’t use the same scale ratio, because they are scaled independently. That means that every part has its own ratio.

The problem is that if the model learns on train data where lines are close to each other, then it does not work very well on validation and test sets, because as we can see lines are moving away.

When I used the independent scaling, even though it is not theoretically appropriate method, the results were much better. However there were still some errors.



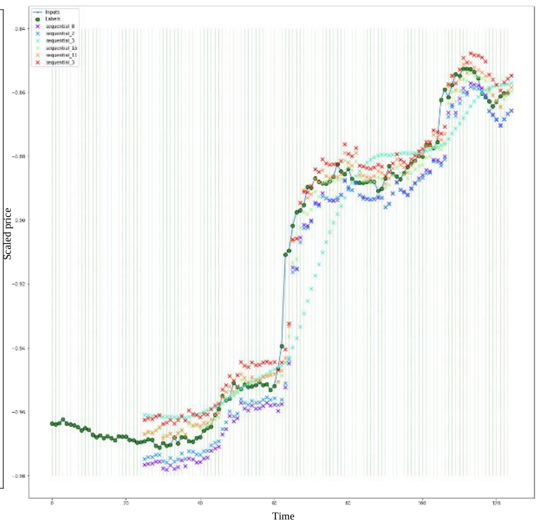


*Figure 19. Input data*

*Figure 20. Predictions with labels*

Figures 19 and 20 show real behaviour of various models which learned on separately scaled datasets. Models were supposed to predict 2 min into the future using 24 minutes of data using recurrent neural networks. In this case only four pairs were used (BTC, ETH, LTC, BCH) to predict BTC price. It is obvious that when the distance between prices is bigger it can shift predictions in some direction. One possibility would be to shift prediction back, yet I believe that finding better solution for scaling data would improve model accuracy.

### 5.2.1 Logarithmic scale

One of the possibilities how to deal with those price differences is to put prices to logarithmical scale, however there is a problem that every Crypto now has different scale, so the differences won't be in correct ratio.

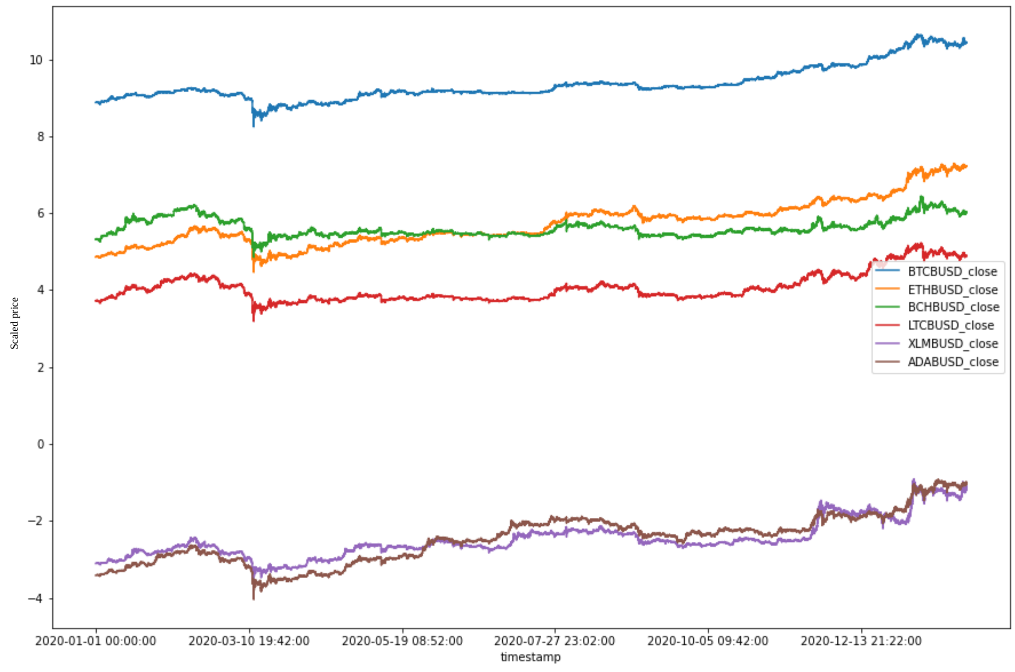


*Figure 21. Log scaled data*

Now it is necessary to scale obtained data. To see the impact of log scale there is a comparison in Figure 22.

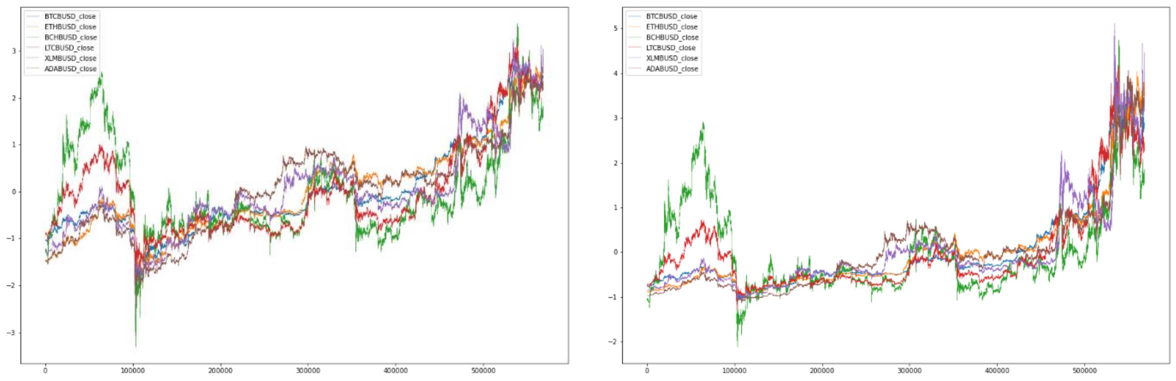


*Figure 22. Comparison of scaled logarithmical data (left) and original scaled data (right)*

The problem of moving differences between pairs is still there so I don't think this is the right way.

### 5.2.2 Percentage change

Another option is to take percentage change. The only problem here is again difference between prices. If BTC price changes by 1% it could have much bigger impact than if XLM has a 1% change.

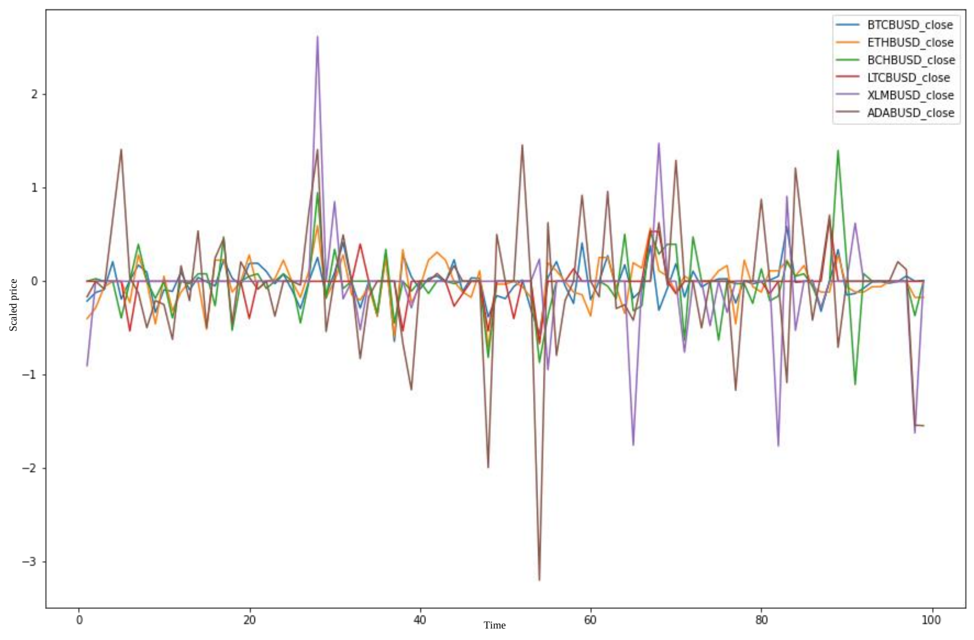


*Figure 23.Example of scaled percentage change data*

For that reason, after creating percentage change from original data I scaled percentage change.

| | BTCBUSD_close | ETHBUSD_close | BCHBUSD_close | LTCBUSD_close | XLMBUSD_close | ADABUSD_close |
|---|---|---|---|---|---|---|
| **Max_before** | 0.033410 | 0.058486 | 0.053971 | 0.058623 | 0.071819 | 0.065333 |
| **Max_after** | 3.054184 | 3.981011 | 3.279838 | 3.409439 | 3.072832 | 3.403360 |

*Table 8. Max values before and after scaling*

Scaled values have smaller range, so this seems like a good approach. Also scaled prices are usually between -1 and 1 which is a good range.

### 5.2.3 Smaller normalized windows.

Third option would be to do it similarly to my original approach with independent scaling, but with smaller windows. The problem here is to specify the size and the location of the window. It could be big as the whole dataset (like in my case on Fig. 18) or its size could be like the input window data size. That means to independently scale every input window, but then every output would be predicted with different scale.

### 5.2.4 Conclusions on data scaling

In this section I went through some possibilities of normalizing/scaling cryptocurrency data. There are still many things which we should consider. For example, if the dataset should be balanced. In Figure 16 we can see that the trend is slightly ascendant. This could also be a problem, because neural network might learn that overall price goes always higher. On the other hand, this trend might continue for the next few months, or the price could also drop. Those things we can’t say for sure. More scalers and data explorations can be found in Data_explore.ipynb.

In this work I have used two options of data scaling, first was using only z-score with separate scaling (Fig. 18 bottom part). This worked, but I encountered various problems while working z-score only, so I added second option which was using percentage change and then scale it with z-score, which worked better so I sticked to it for the rest of this work.

### 5.3 Model’s training

There is not a universal right way how to proceed with scaling and setting all the parameters. There are also various cells and architectures available. Which is why my training file, Forecasting.ipynb has following variants of RNNs.

| **Name** | **Type** |
|---|---|
| **LSTM** | Many to many |
| **LSTM bidirectional** | Many to many |
| **GRU** | Many to many |
| **GRU bidirectional** | Many to many |
| **LSTM single shot** | Many to one |
| **LSTM bidirectional single shot** | Many to one |
| **GRU single shot** | Many to one |
| **GRU bidirectional single shot** | Many to one |

*Table 9. RNN types used in this work*

Layers in Tensorflow contain parameter called return_sequences. If last layer hast parameter return_sequence equal True then it creates prediction for every step, otherwise it warms up its internal state before making single prediction. If we stack layers this parameter has to be set True, but if last layer is set to false then it is called single shot model. [15]

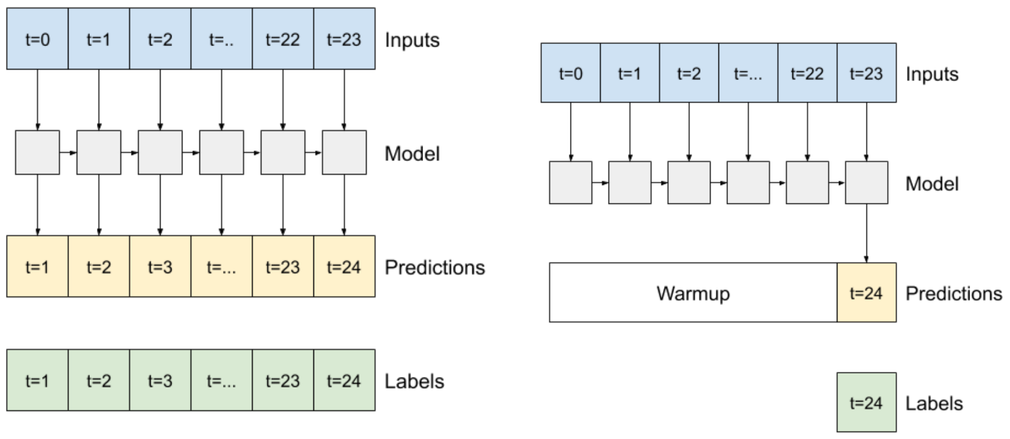


*Figure 24. Predictions based on return_squences parameter. True (left). False (right) [15]*

So, in my code, I just change compile parameters and train all those models and save them into directory (for example 20_04_2021). That means that in one iteration 8 architectures are trained. This allows me to load models later and evaluate or retrain them. I tried two main approaches of data norm and scaling for which I trained and evaluated several models. I tried to change parameters according to what I thought should work. My choices were usually based on what I had read about those parameters and on my intuition.

- First approach

The first thing I tried was using downloaded dataset with just z-score scaling. I used separate scaling, like on bottom part of Figure 18. Then it was processed, which includes data windowing and then models were trained on this dataset. Empty rows were dropped, which made training process more difficult for NN, however models were much more generalized. I also removed problematic part of the dataset. After this I obtained dataset with 60 000 rows. As we will see in the next part of this work this approach follows the trend quite well, however predictions could be deviated from the correct values. Some models even were not deviated, but if we considered if direction was correct percentage was not too high. I was using 24 minutes of data to predict two minutes into the future.

- Second approach

When I was working with previous models, I was still struggling with selecting the proper scaler and its range, due to this I tried to train some models using percentage + z-score scaling, which after closer evaluation worked better than models obtained only using z-score. All further models were trained using this approach, when I cropped problematic parts, dropped all remaining NaN values and then scaled data using percentage + z-score. It was basically the same process as int the first approach however with percentage scale. In this case percentage change will be predicted instead of scaled price. To make higher possibilities of good predictions I will be predicting only one minute into the future.

Generally, models learn to follow the trend during first or first two epochs. Then, the biggest challenge was to move below some number. When learning models, I was aiming for the lowest val_loss.

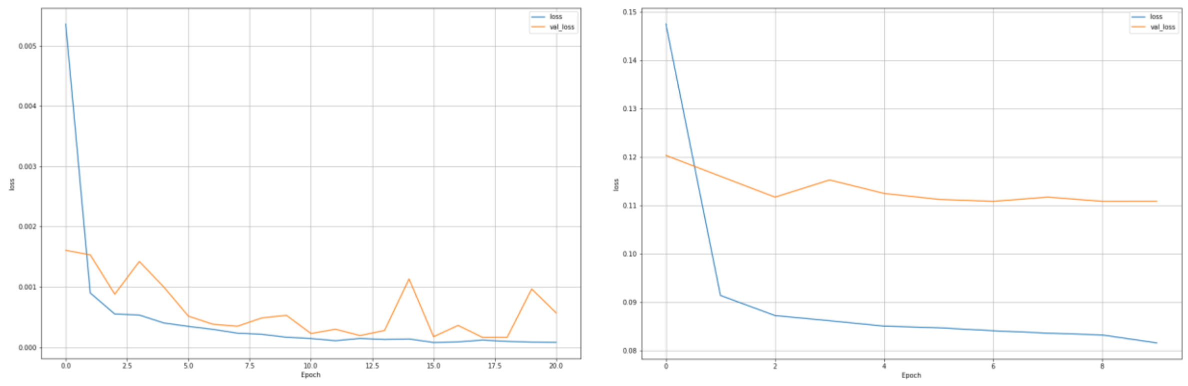
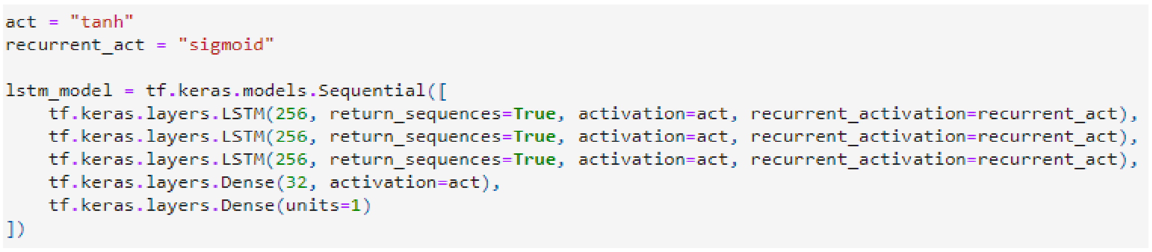


*Figure 25. Usual training history for z-score only (left) and percentage change + z-score (right)*

Usual model learning history goes fast at the beginning (fig. 25) and then validation usually oscillates while loss goes down very slowly. Of course, not all models learn like this, some of them learn in first epoch and some of them learn more slowly, it really depends on selected architecture and optimizer.

## 5.4 Models’ performances

As I have already mentioned there are many parameters which could be changed and various data scaling options. For every model setting I trained 8 different modifications of RNN with the same number of layers and nodes. Architecture can look like the one on Figure 26.



```
act = "tanh"
recurrent_act = "sigmoid"

lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(256, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.LSTM(256, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.LSTM(256, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.Dense(32, activation=act),
    tf.keras.layers.Dense(units=1)
])
```

*Figure 26. Example of RNN architecture with LSTM layers in TensorFlow*

Following models were trained on downloaded dataset. Window size was 24 and I was predicting two minutes into the future using z-score only and one minute with z-score + percentage change. Below is a part of evaluation which was made on validation and test part of the original downloaded dataset and there is also column with My dataset which is supposed to reflect “current” data, because it is evaluated on last 10% of my collected dataset (04-11-2020 – 08-11-2020). Scores are obtained using predefined TensorFlow function evaluate. Whole table can be found in model_comparsion.xlsx file. It includes many models, including models using both approaches, so I will only show some of them.

| | Validation | | Test | | My dataset (test) | | |
|---|---|---|---|---|---|---|---|
| | Loss | MAE | Loss | MAE | Loss | MAE | Compile_settings |
| LSTM | 0.001 | 0.021 | 0.0014 | 0.0272 | 0.000398 | 0.0275 | comp_1 |
| LSTM - bidirectional | 3.36E-04 | 0.0108 | 0.00029 | 0.0116 | 0.000379 | 0.0128 | comp_1 |
| GRU | 9.15E-04 | 0.0188 | 0.00122 | 0.024 | 0.001488 | 0.0271 | comp_1 |
| GRU - bidirectional | 2.95E-04 | 0.0104 | 0.00020 | 0.00991 | 0.000189 | 0.0092 | comp_1 |
| LSTM (single) | 1.20E-03 | 0.0233 | 0.0015 | 0.0276 | 0.0013 | 0.026 | comp_1 |
| LSTM - bidirectional (single) | 1.40E-03 | 0.0244 | 0.0019 | 0.0323 | 0.00165 | 0.0289 | comp_1 |
| GRU (single) | 1.20E-03 | 0.0254 | 0.0015 | 0.0291 | 0.00161 | 0.0299 | comp_1 |
| GRU - bidirectional (single) | 1.30E-03 | 0.024 | 0.00162 | 0.02865 | 0.00179 | 0.0302 | comp_1 |

*Table 10. Example from models_comparsion file with evaluation scores (for z-score scale 2 minutes)*

Right column (Table 10.) contains compile settings which corresponds to which setting was used to compile model. Pct + z-score means that percentage change plus z-score scaling was used.

| **Compile_settings** | comp_1 | comp_2 | comp_3 |
|---|---|---|---|
| Optimizer | Adam | SGD | Adam |
| Learning rate | default | default | default |
| Loss | MSE | MSE | MSE |
| Metrics | MAE | MAE | MAE |
| Early_stopping | Yes | Yes | Yes |
| Dataset | Downloaded | Downloaded | Downloaded |
| Pre-processing | z-score scaling (separate) | z-score scaling (separate) | Pct + z-score |
| Additional opt. parameters | - | - | - |

*Table 11. Part of compile settings table from models_comparsion file*

The first part of the scores from models_comparsion.xlsx is for only z-score scaled models. I have already described the main reason why I switched to different scaling option. The problems are visible in Figures 19, 20 and 27. Also, when I checked percentage correctness of predicted directions, they were not very high. Usually, a little bit below 50% and only few models made it above 50% accuracy. That is why I switched to one minute prediction and percentage change + z-score scaling. Direction accuracy for z-score scaled models was calculated from previous prediction. If prediction was bigger then previous and price also raised, then I consider direction as correct. This approach could be also appliable on percentage models, however for percentage + z-score models I compared prediction with 0. If prediction was bigger than 0 and next real percentage price was also bigger than 0 then it was considered as correct. The same if both were smaller than 0.

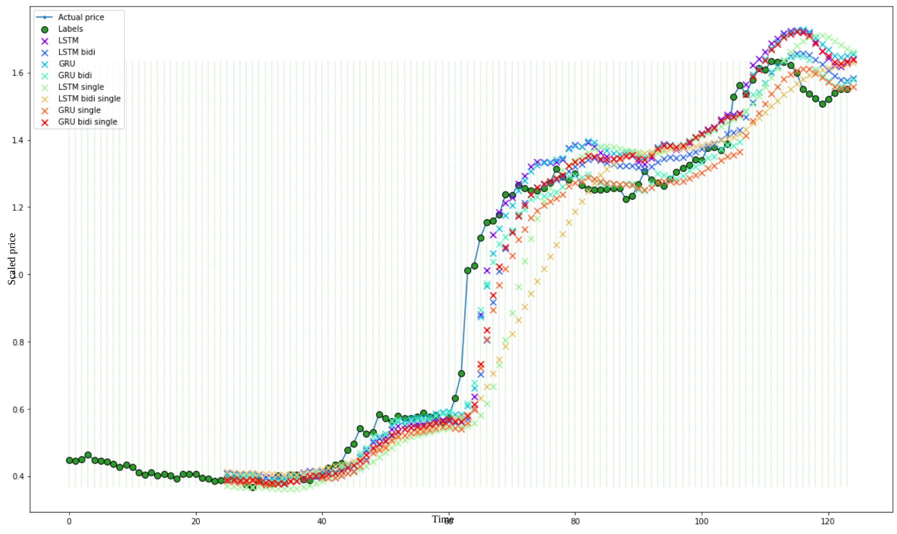


*Figure 27. Predictions of models_21_11_2020 iteration on z-score scale (2 mins predictions)*

As I already said, TensorFlow provides function for evaluating models which I used, however it compares all predictions with all labels, that is why LSTM-bidirectional and GRU-bidirectional have much better results than the other models (Fig. 28). When we consider only last prediction, then all the models have pretty similar results. I still could compare same model architectures with different settings against each other (Table 13.). Validation, test and recent are evaluated on the same data as in table 10.

Which means original training dataset with last part of my collected dataset. Recent data reflects time when market entered “Bull market” period and original training dataset was more “Bear market” data that is why the performance is worse. In the next part I also included evaluation on my collected dataset, but on older part, which should more reflect market behavior of the original training dataset. This was because I wanted to see how much performance change can based on market period.

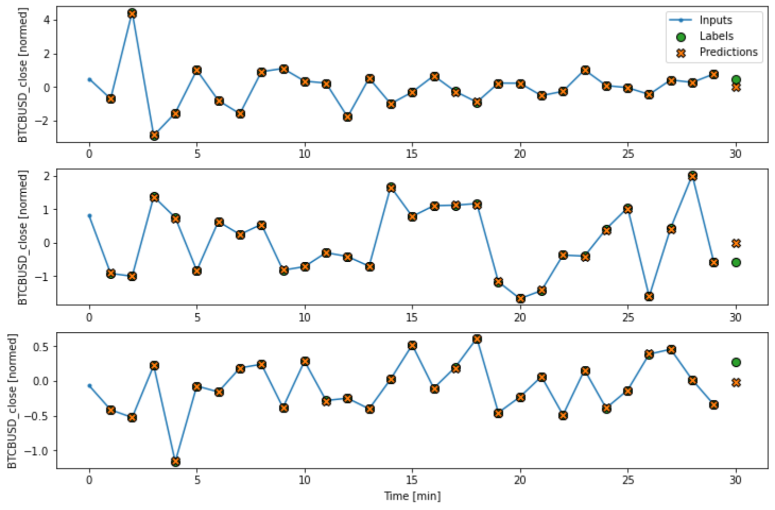


*Figure 28. Why many to many GRU and LSTM models have better evaluation scores with predefined functions*

| Directory_name | val_loss | val_MAE | test_loss | test_MAE | recent_loss | recent_MAE |
|---|---|---|---|---|---|---|
| 07_04_2021 | 1.34198 | 0.60758 | 1.0061 | 0.48238 | 2.7676 | 0.84872 |
| 07_04_2021_1 | 1.29281 | 0.59024 | 0.9929 | 0.46656 | 2.6473 | 0.8092 |
| 07_04_2021_2 | 1.281 | 0.59444 | 0.9739 | 0.46829 | 2.6249 | 0.7912 |
| 08_04_2021 | 1.29328 | 0.56875 | 0.9782 | 0.43605 | 2.5708 | 0.69425 |
| 09_04_2021 | 1.2768 | 0.58998 | 0.9719 | 0.4643 | 2.6312 | 0.7942 |
| 12_04_2021 | 1.3139 | 0.59765 | 1.0325 | 0.4802 | 2.7818 | 0.8407 |
| 13_04_2021 | 1.28226 | 0.59662 | 0.9853 | 0.47348 | 2.6863 | 0.82438 |
| 14_04_2021 | 1.29305 | 0.57003 | 0.9785 | 0.4379 | 2.5697 | 0.69861 |
| 17_04_2021 | 1.2708 | 0.5946 | 0.9712 | 0.46854 | 2.6414 | 0.81006 |
| 18_04_2021 | 1.2922 | 0.5713 | 0.97839 | 0.43996 | 2.56941 | 0.70255 |
| 20_04_2021 | 1.29245 | 0.57067 | 0.9782 | 0.43852 | 2.56926 | 0.70094 |
| 28_04_2021 | 1.29516 | 0.59948 | 1.01164 | 0.4808 | 2.6563 | 0.81939 |

*Table 12. Comparison of various LSTM architecture performances trained on pct + z-score processed dataset*

All models from one learning iteration (same compile settings and architecture size) are stored in separate directories. Above we have classical LSTM architectures, compared using gradient background style. Green models are the best and red are the worst results. The best results from each column are underlined. Columns from Table 12. correspond to columns from Table 10. All other architectures are compared the same way in NN_plots.ipynb file and every bets performing model is also highlighted in models_comparsion.xlsx file.

Now that I have comparison against the same architectures, I wanted to progress towards more informative output. Those numbers above while accurate don't really say much at first sight. To improve this, I created my own evaluation functions. First of them is classical MAE, but in this case, it works on all models the same. It takes only the last prediction compared with its label. Second function I created is comparing if the direction of the prediction is the same. This second function returns percentage of how many times was the predicted and correct direction the same.  All results are in model_comparsion.xlsx.

MAE (org) and Direction (org) were evaluated on first 6000 points from test part of the original downloaded dataset on which were models trained (Downloaded dataset from table 3). MAE (new) and Direction (new) were evaluated on the first 9000 points of my created dataset (Collected dataset from Table 3). I took beginning of downloaded dataset, because in

|  | MAE (org) | Direction (org) [%] | MAE (new) | Direction (new) [%] |
|---|---|---|---|---|
| LSTM | 0.43757 | 55.94 | 0.3337 | 55.07 |
| LSTM - bidirectional | 0.46991 | 55.16 | 0.35059 | 55.8 |
| GRU | 0.44657 | 53.03 | 0.33414 | 55.33 |
| GRU - bidirectional | 0.46024 | 53.78 | 0.3507 | 53.69 |
| LSTM (single) | 0.43723 | 55.01 | 0.33448 | 52.28 |
| LSTM - bidirectional (single) | 0.45857 | 55.18 | 0.33937 | 55.57 |
| GRU (single) | 0.44857 | 56.09 | 0.33537 | 56.43 |
| GRU - bidirectional (single) | 0.45724 | 57.16 | 0.34226 | 56.4 |

my opinion in reflects better normal market.

*Table 13. Example of my evaluation scores for models from 08_04_2021*

Figure 29 explains which parts of my collected dataset were used. My evaluation (Table 13 new) was made on "My evaluation" part of the data. Recent loss and MAE from Table 12. were made on "Recent" part. From performances we will see how much will change from "Bear market" (green part) to "Bull market" ("blue part") affect models' performances.

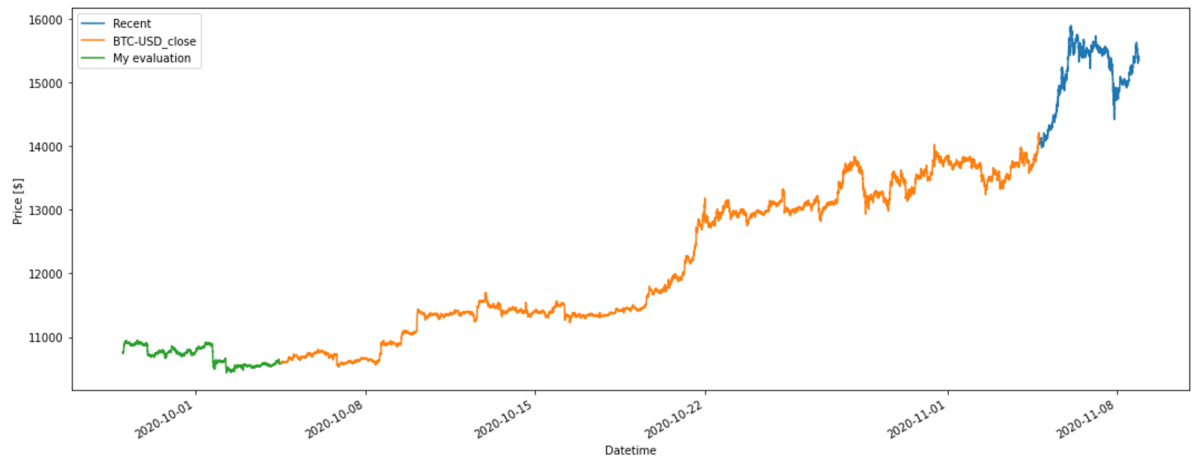


*Figure 29. Highlighted parts of my collected dataset*

If we plot predictions using trained models, we could obtain something like following picture. Input data are the same data as in fig 19 and fig 20, only with percentage + z-score scale.

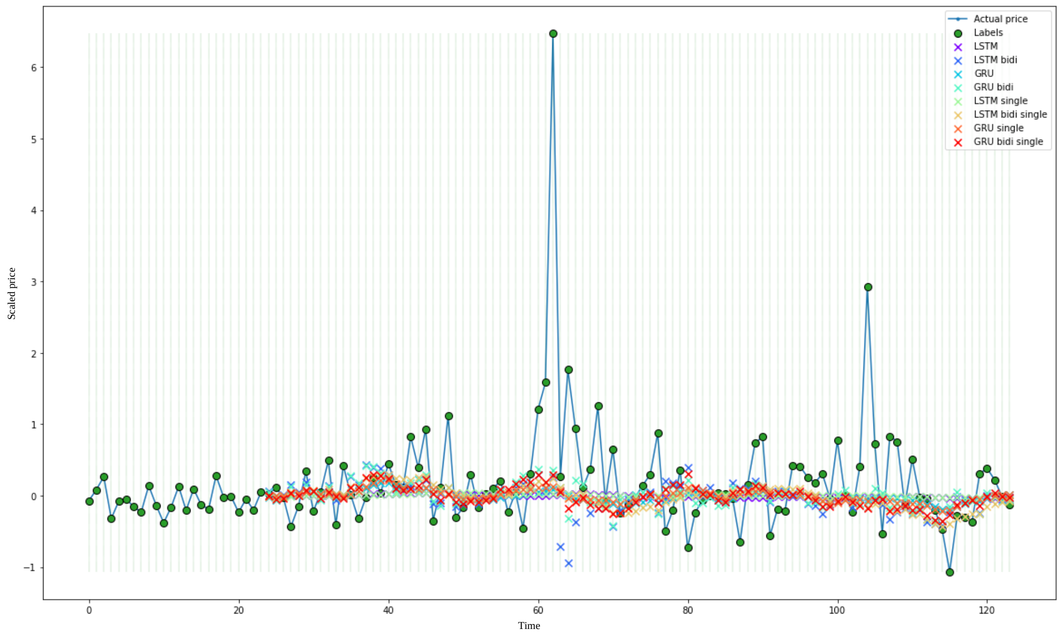


*Figure 30. Visualized predictions of models from 08_04_2021*

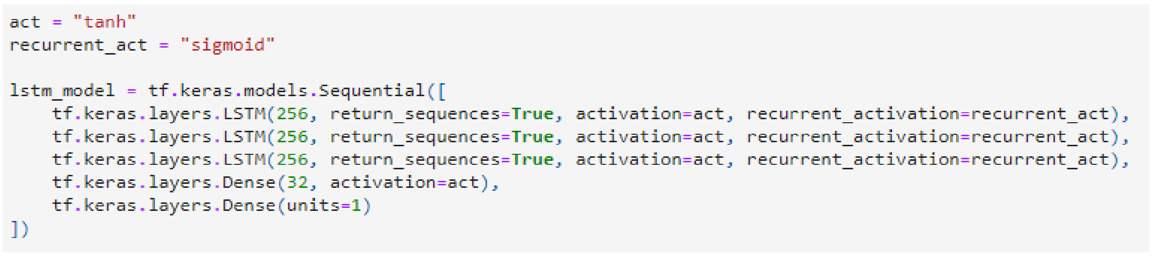
As evident from Figure 30, models do not really react to very big sudden changes and this also applies in general to all my trained models, but it makes sense since those changes are unpredictable from close price, still we can see some sort of moving trend and that predictions are made.

### 5.5 Interesting models and conclusions

As I have trained many models, putting all the results here would be counterproductive. Therefore, I decided to only present models and results which I found interesting in this section.

First very interesting fact is that models have better MAE at the beginning of the collected dataset, but worse performance than at the end of the same dataset. I mention this to highlight the importance on the market behavior. Even though it is the same dataset just one month can change performance so much. This also means that dropping NaN values instead of replacing them made learning harder, but models are not overfitted. Even when I tried to add drop layer into my models it did not improve performance it just made performance worse.

One of the most interesting models is with following architecture, which is named 12_04_2021. It has following architecture. I picked this model because it had the best performance from the compared models at the beginning of collected dataset.



```
act = "tanh"
recurrent_act = "sigmoid"

lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(256, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.LSTM(256, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.LSTM(256, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.Dense(32, activation=act),
    tf.keras.layers.Dense(units=1)
])
```

*Figure 31.LSTM architecture of 12_04_2021*

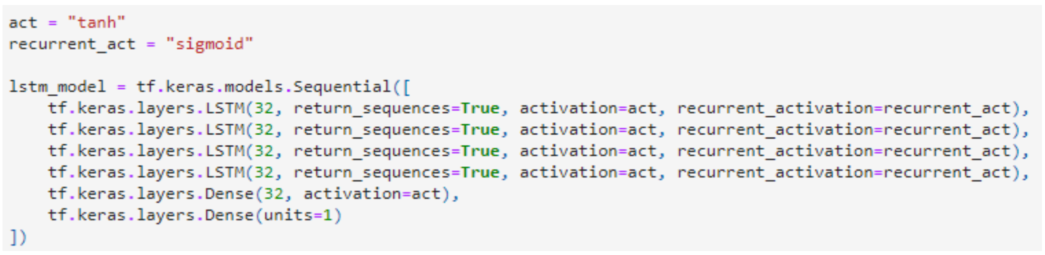
With compile settings comp_7 (Table 15.) which corresponds to Adam optimizer with learning rate 0.01. Loss is MSE and Metrics is MAE.

If we check correct percentage of directions, we get the following table. The main reason why I found this very interesting is that in this case mostly single shot models are performing better in terms of MAE and GRU single shot had the best direction correctness. Models also have best Loss a MAE performances on test part of my dataset.

| | MAE (org) | Direction (org) [%] | MAE (new) | Direction (new) [%] |
|---:|---:|---:|---:|---:|
| LSTM | 0.48208 | 60.19 | 0.36737 | 53.65 |
| LSTM - bidirectional | 0.45674 | 56.66 | 0.3564 | 52.83 |
| GRU | 0.47324 | 41.94 | 0.36148 | 44.35 |
| GRU - bidirectional | 0.46807 | 35.26 | 0.35103 | 40.85 |
| LSTM (single) | 0.44574 | 35.26 | 0.33903 | 40.7 |
| LSTM - bidirectional (single) | 0.44257 | 59.63 | 0.33837 | 53.12 |
| GRU (single) | 0.45174 | 64.74 | 0.33648 | 59.3 |
| GRU - bidirectional (single) | 0.44074 | 35.82 | 0.33681 | 40.85 |

*Table 14. My evaluations scores of 12_04_2021*

Second very good performing models were from 08_04_2021 (architecture Fig. 32 and scores Table 13). Models from this iteration were trained with compile settings comp_4 (Table 15). The most interesting fact about those models is that every single one of them had good performance. Usually, model performances oscillate according to selected cells, but in this case all of them were good.



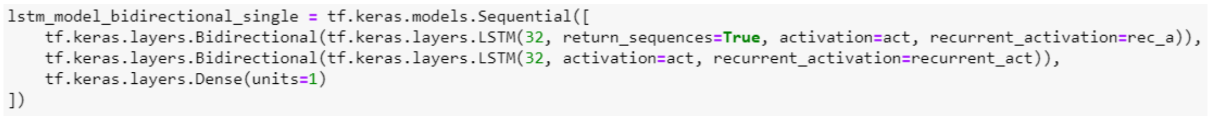
```
act = "tanh"
recurrent_act = "sigmoid"

lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.LSTM(32, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.LSTM(32, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.LSTM(32, return_sequences=True, activation=act, recurrent_activation=recurrent_act),
    tf.keras.layers.Dense(32, activation=act),
    tf.keras.layers.Dense(units=1)
])
```

*Figure 32. Architecture of LSTM from 08_04_2021*

| **Compile_settings** | comp_4 | comp_7 |
|---|---|---|
| Optimizer | Adagrad | Adam |
| Learning rate | default | 0.01 |
| Loss | MSE | MSE |
| Metrics | MAE | MAE |
| Early_stopping | Yes | Yes |
| Dataset | Downloaded | Downloaded |
| Pre-processing | Pct + z-score | Pct + z-score |
| Additional opt. parameters | - | - |

*Table 15. Compile settings of models*

The last model I will mention was LSTM-bidirectional single shot from 28_04_2018 with architecture in Figure 33 and optimizer Adam with learning rate 0.02.



```
lstm_model_bidirectional_single = tf.keras.models.Sequential([
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32, return_sequences=True, activation=act, recurrent_activation=rec_a)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32, activation=act, recurrent_activation=recurrent_act)),
    tf.keras.layers.Dense(units=1)
])
```

*Figure 33. 28_04_2018 architecture*

As you can see this model is quite simple and it has also correct direction percentage around 60%.

FTRL optimizer models had also quite interesting results. It always converged very rapidly, and its Loss and MAE were good, but also all predictions were in one line (Fig. 34).

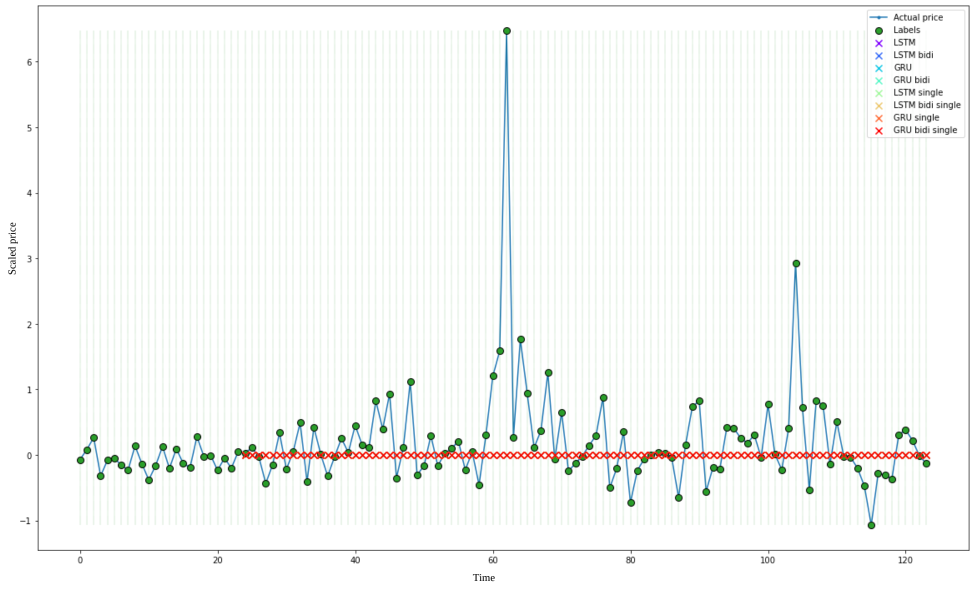
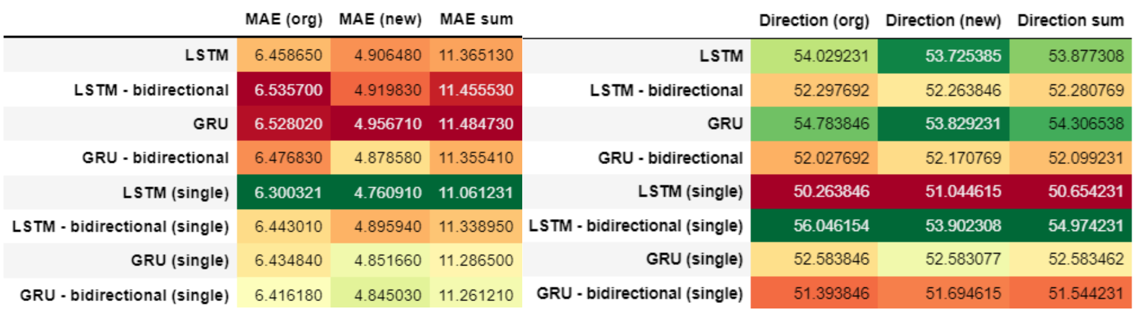


*Figure 34. FTRL optimizer models*

Another fact which I was interested in, was which one of those 8 architectures has the best overall results. So, I summed all MAE and average percentage for all architectures, trained on percentage + z-score scale.



| | MAE (org) | MAE (new) | MAE sum |
|---|---|---|---|
| LSTM | 6.458650 | 4.906480 | 11.365130 |
| LSTM - bidirectional | 6.535700 | 4.919830 | 11.455530 |
| GRU | 6.528020 | 4.956710 | 11.484730 |
| GRU - bidirectional | 6.476830 | 4.878580 | 11.355410 |
| LSTM (single) | 6.300321 | 4.760910 | 11.061231 |
| LSTM - bidirectional (single) | 6.443010 | 4.895940 | 11.338950 |
| GRU (single) | 6.434840 | 4.851660 | 11.286500 |
| GRU - bidirectional (single) | 6.416180 | 4.845030 | 11.261210 |

| | Direction (org) | Direction (new) | Direction sum |
|---|---|---|---|
| LSTM | 54.029231 | 53.725385 | 53.877308 |
| LSTM - bidirectional | 52.297692 | 52.263846 | 52.280769 |
| GRU | 54.783846 | 53.829231 | 54.306538 |
| GRU - bidirectional | 52.027692 | 52.170769 | 52.099231 |
| LSTM (single) | 50.263846 | 51.044615 | 50.654231 |
| LSTM - bidirectional (single) | 56.046154 | 53.902308 | 54.974231 |
| GRU (single) | 52.583846 | 52.583077 | 52.583462 |
| GRU - bidirectional (single) | 51.393846 | 51.694615 | 51.544231 |

*Figure 35. Summed results for MAE (left) and direction percentage (right) for pct + z-score scaled models*

From overall results we could say that if we had to choose one architecture, we should choose it based on what we really need. If we wanted lower MAE lets pick LSTM, if better direction accuracy here LSTM bidirectional is better.

I also noticed that GRU has the worst MAE from visualized predictions which was also confirmed by overall results above. It is possibly trying to make predictions more aggressively that other architectures in general.

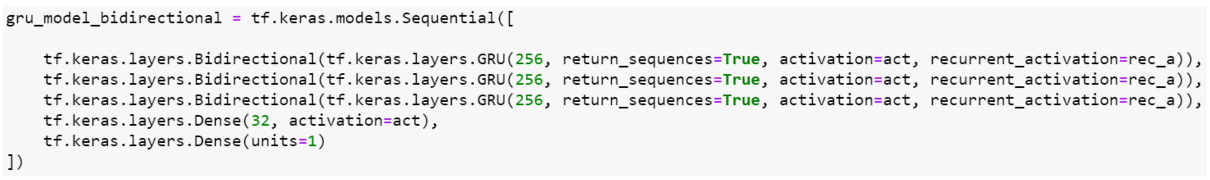
## 5.6 Bigger model evaluation

As I already mentioned all models above were trained on quite small dataset which included only 4 cryptocurrencies. Previous percentage models' predictions were made on 24 minutes 1 minute into the future. I wanted to know how or if performance would improve if we used more data, and more cryptocurrencies. So, I trained bigger model on dataset obtained using Binance API (Fig. 15 final_df). In this case I used 30 minutes to predict 1 minute into the future with different close prices. In this case I used following pairs: BTC, OCEAN, ZRX, ATOM, BNT, ALGO, TWT, SUSHI. Because they were most corraleted with BTC.

I decided that I will evaluate those big models on the latest data I have. I will also compare them to best performing models from previous section. All results below will be for data from 2021-07-13 15:17 – 2021-07-20 13:56. The only difference is that big models are trained on different cryptocurrencies.

I can't really compare MAE between old models and big new models, because scale is different, but I can compare correct direction percentage.

GRU-bidirectional has architecture in Figure 32 and was trained using Adam optimizer with learning rate 0.0005. Other models from 16_07_2021_big have again same number of layers and cells as GRU-bidirectional.



```
gru_model_bidirectional = tf.keras.models.Sequential([

    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(256, return_sequences=True, activation=act, recurrent_activation=rec_a)),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(256, return_sequences=True, activation=act, recurrent_activation=rec_a)),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(256, return_sequences=True, activation=act, recurrent_activation=rec_a)),
    tf.keras.layers.Dense(32, activation=act),
    tf.keras.layers.Dense(units=1)
])
```

*Figure 36. GRU-bidirectional architecture from 16_07_2021_big*

If we plot actual predictions of models from 16_07_2021_big we can notice that data is very noisy and that models are again very careful with more risky predictions.

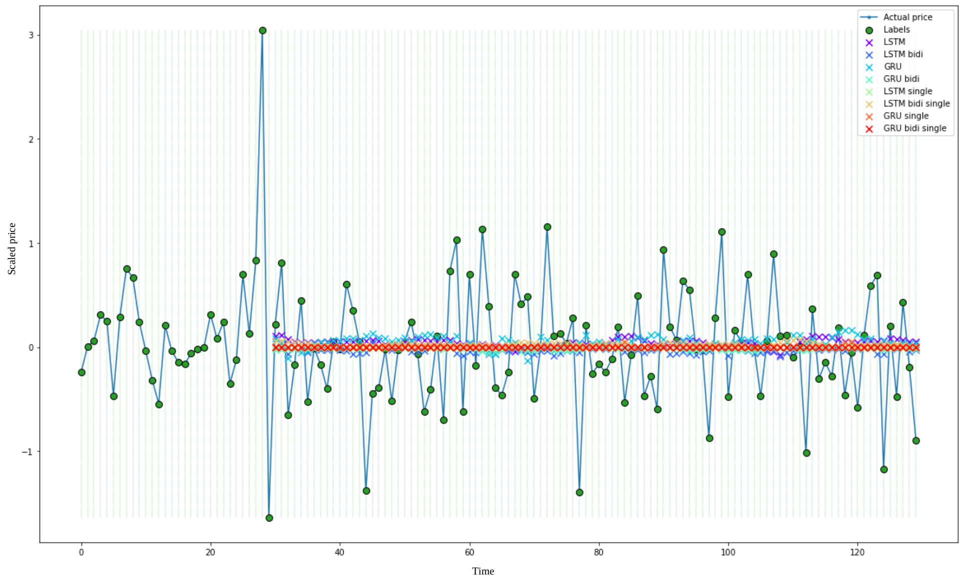


*Figure 37. Plotted predictions of models from 15_07_2021_big on latest data*

If we check performance on data from 2021-07-13 15:17 – 2021-07-20 13:56 of all models from that iteration, we get following results.

| | MAE (latest) | Direction (latest) [%] |
|---:|---:|---:|
| LSTM | 0.42094 | 49.47 |
| LSTM - bidirectional | 0.41894 | 50.8 |
| GRU | 0.42514 | 50.2 |
| GRU - bidirectional | 0.41774 | 51.82 |
| LSTM (single) | 0.41814 | 50.61 |
| LSTM - bidirectional (single) | 0.41914 | 49.31 |
| GRU (single) | 0.41814 | 49.39 |
| GRU - bidirectional (single) | 0.41804 | 50.61 |

*Table 16. My evaluation of models 16_07_2021_big on my latest data*

Even with more data included best performance was only 51.82% on latest data. This shows how much can change during half a year. Not so long ago I was able to get about 60% correct direction using smaller dataset and now only 51.82%. Of course, if I trained more bigger models with more data, I would probably find better performing model. I also have not trained many models on this bigger dataset, so that is another reason why the is direction correctness lower.

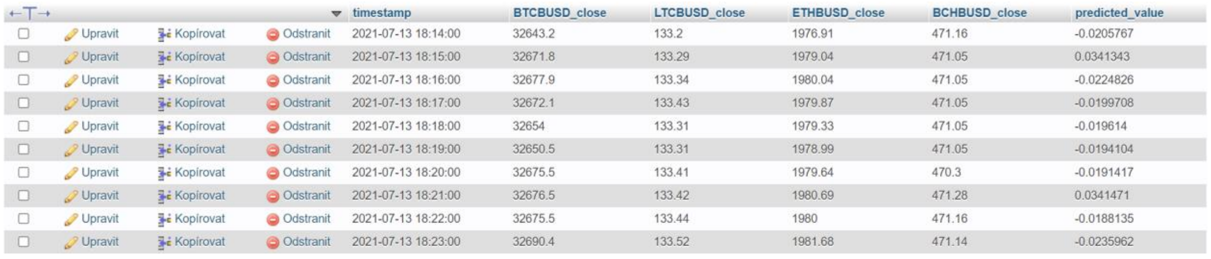
I have also checked if my older models worked newest data and they were only GRU from 120_04_2021 had 50.8% of predictions correct. It is not very much, but if we consider that it was trained on now 3 years old data and we still get above 50% correct percentage that is not that bad.

This proves that predicting price is possible and sometimes we can get quite high numbers, however keeping higher accuracy for longer time period is very challenging.

# 6. Crypto predicting API

Having trained model is a good exercise, yet we usually train models for actual purpose. I completed my work and created an API which loads pretrained model and continuously creates predictions based on incoming data and saves them for later use.

API loads one of the models from selected path and using Binance's API gets data every minute. New data are pre-processed the same way as they were in the training part and on that data, model creates a prediction. All predictions with input data are stored in dataframes which are after defined time saved into MySQL database. Table predictions contains timestamp, original input values and predicted value. The only arguable setup in this table setting could be choosing optimal primary key, because I set timestamp as this table's primary key, however, there also could be extra index autoincrementing column, which could also be a primary key, however in this case both options should be ok.



| ←T→ | | | | timestamp | BTCBUSD_close | LTCBUSD_close | ETHBUSD_close | BCHBUSD_close | predicted_value |
|---|---|---|---|---|---|---|---|---|---|
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:14:00 | 32643.2 | 133.2 | 1976.91 | 471.16 | -0.0205767 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:15:00 | 32671.8 | 133.29 | 1979.04 | 471.05 | 0.0341343 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:16:00 | 32677.9 | 133.34 | 1980.04 | 471.05 | -0.0224826 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:17:00 | 32672.1 | 133.43 | 1979.87 | 471.05 | -0.0199708 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:18:00 | 32654 | 133.31 | 1979.33 | 471.05 | -0.019614 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:19:00 | 32650.5 | 133.31 | 1978.99 | 471.05 | -0.0194104 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:20:00 | 32675.5 | 133.41 | 1979.64 | 470.3 | -0.0191417 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:21:00 | 32676.5 | 133.42 | 1980.69 | 471.28 | 0.0341471 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:22:00 | 32675.5 | 133.44 | 1980 | 471.16 | -0.0188135 |
| ☐ | Upravit | Kopírovat | Odstranit | 2021-07-13 18:23:00 | 32690.4 | 133.52 | 1981.68 | 471.14 | -0.0235962 |

*Figure 38. Predictions table in phpMyAdmin*

The reason for this storing is that market is very dynamic system, and we should check if model's performance is sufficient, and it allows us to more explore model's performance.

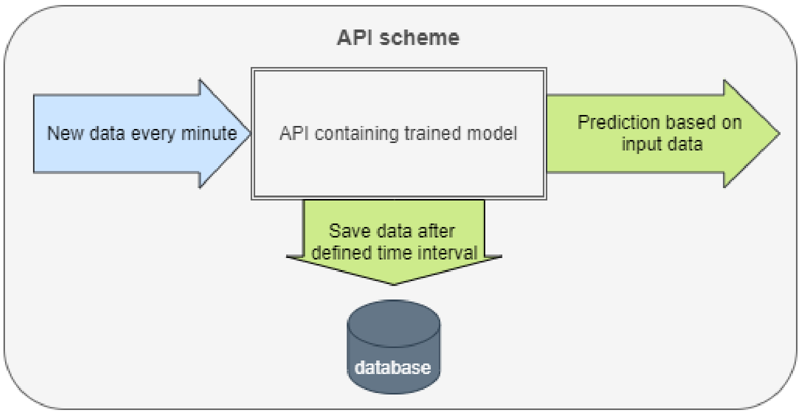

API scheme
New data every minute
API containing trained model
Prediction based on input data
Save data after defined time interval
database

*Figure 39. Crypto predicting API scheme*

## 6.1 Used libraries in predicting API

It is very important to learn how to install and use libraries. There is likely a library for all our needs; we just need to find it. The main advantage is that libraires are usually open source, which allows everyone to participate on improving them. It is sometimes more about learning how to use various libraries, than programming itself. Building this API, I also used following libraries.

| **name** | **url** | **functionality** |
|---|---|---|
| Pandas | https://pandas.pydata.org/ | Data manipulation. |
| Numpy | https://numpy.org/ | Math, arrays and matrices. |
| Python-binance | https://python-binance.readthedocs.io/en/latest/index.html | Obtaining data |
| TensorFlow | https://www.tensorflow.org/ | ML library |
| Sqlalchemy | https://www.sqlalchemy.org/ | SQL connection |
| Apscheduler | https://apscheduler.readthedocs.io/en/stable/ | Cron job (scheduling) |
| Falcon | https://falcon.readthedocs.io/en/stable/ | Library for API |
| Json | https://docs.python.org/3/library/json.html | Working with json |
| Waitress | https://docs.pylonsproject.org/projects/waitress/en/latest/ | Python WSGI-server |

*Table 17. List of libraries used in crypto predicting API*

When we use imported functions from libraries the main task is to assemble the parts together. We must be prepared that some functions may give unwanted data type especially when we put various libraries together.  However, libraries still often provide programmers with valuable and very complex functions and make their lives easier.

## 6.2 API’s functionality

I have already mentioned basic API’s functionality, however I think it would to useful to explain it in greater detail. The first used class after we start this app is PredictionResource. This class only creates an instance of ModelClass. It also has a scheduler which runs predicting function every 9 seconds after new minute starts. When new minute starts Binance data are not there yet for the last minute, and that is why there needs to be a delay.

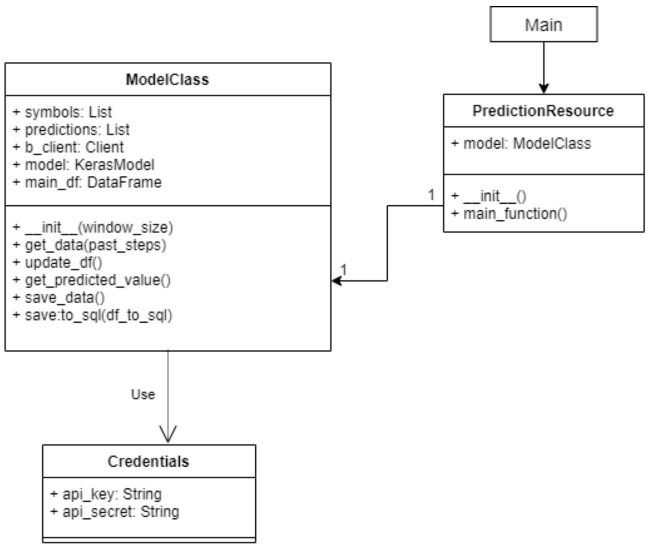
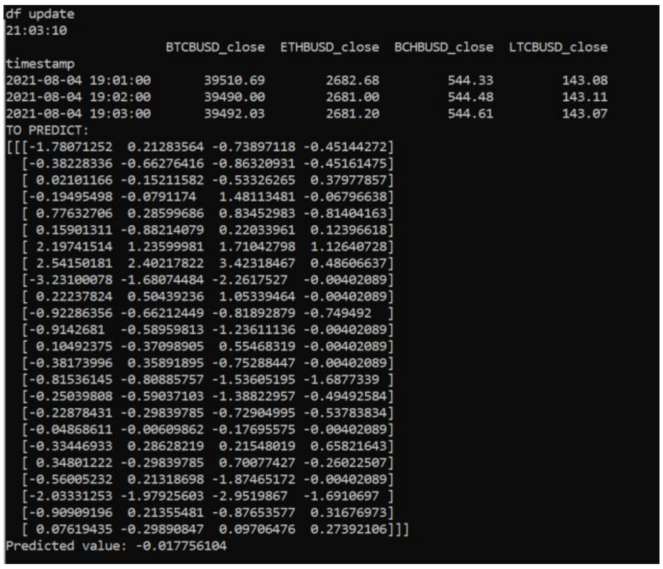


*Figure 40. API class diagram*

Model class, which is probably the most important class, first creates client for communicating with Binance. For safety reasons credentials are loaded from json file which is located outside this API. Then RNN model is loaded from selected directory. The last thing that is loaded is scaler, which scales our data later. Finally, while initializing we load window size of data to main_df. This is done using get_data function which obtains data for number of past steps for all cryptocurrencies which are stored in symbols list. Then every minute, 9 seconds after new minute begins new data is obtained, main_df is updated and predictions are made on latest scaled data.

Then every 10 minutes data with corresponding predictions are stored in MySQL database. This is done by calling save_data function which puts predictions and original data together and then calls save_to_sql which saves data to predictions table (Fig. 38).



```
df update
21:03:10
                     BTCBUSD_close  ETHBUSD_close  BCHBUSD_close  LTCBUSD_close
timestamp
2021-08-04 19:01:00       39510.69        2682.68         544.33         143.08
2021-08-04 19:02:00       39490.00        2681.00         544.48         143.11
2021-08-04 19:03:00       39492.03        2681.20         544.61         143.07
TO PREDICT:
[[[-1.78071252  0.21283564 -0.73897118 -0.45144272]
  [-0.38228336 -0.66276416 -0.86320931 -0.45161475]
  [ 0.02101166 -0.15211582 -0.53326265  0.37977857]
  [-0.19495498 -0.0791174   1.48113481 -0.06796638]
  [ 0.77632706  0.28599686  0.83452983 -0.81404163]
  [ 0.15901311 -0.88214079  0.22033961  0.12396618]
  [ 2.19741514  1.23599981  1.71042798  1.12640728]
  [ 2.54150181  2.40217822  3.42318467  0.48606637]
  [-3.23100078 -1.68074484 -2.2617527  -0.00402089]
  [ 0.22237824  0.50439236  1.05339464 -0.00402089]
  [-0.92286356 -0.66212449 -0.81892879 -0.749492  ]
  [-0.9142681  -0.58959813 -1.23611136 -0.00402089]
  [ 0.10492375 -0.37098905  0.55468319 -0.00402089]
  [-0.38173996  0.35891895 -0.75288447 -0.00402089]
  [-0.81536145 -0.80885757 -1.53605195 -1.6877339 ]
  [-0.25039808 -0.59037103 -1.38822957 -0.49492584]
  [-0.22878431 -0.29839785 -0.72904995 -0.53783834]
  [-0.04868611 -0.00609862 -0.17695575 -0.00402089]
  [-0.33446933  0.28628219  0.21548019  0.65821643]
  [ 0.34801222 -0.29839785  0.70077427 -0.26022507]
  [-0.56005232  0.21318698 -1.87465172 -0.00402089]
  [-2.03331253 -1.97925603 -2.9519867  -1.6910697 ]
  [-0.90909196  0.21355481 -0.87653577  0.31676973]
  [ 0.07619435 -0.29890847  0.09706476  0.27392106]]]
Predicted value: -0.017756104
```

*Figure 41. Screenshot of API in terminal*

On Figure 41 there is a screenshot from terminal when API is on. We can see that 10 seconds after new minute started, we loaded new data from Binance (function starts 9 seconds after new minute but takes about one second). I take 3 last minutes, because then it can be easily mapped on main_df and some NaN might occur in latest data rows, so they are eventually replaced by correct values. When dataframe is updated, it is scaled and inputted into loaded model. Then prediction is made by loaded model. Predictions are shifted one minute into the future so when they are saved, they correspond to time they were supposed to predict.

# 7. Hardware and software

At conclusion I will mention the hardware and software I used in this work. All the code was written in Python. For environment management I used Anaconda, which is a distribution of the Python and R programming languages for scientific computing. It aims to simplify package management and deployment.  For data exploration I used Jupyter notebook. For API or more complicated programs I used Visual Studio Code. Another alternative for VS code is PyCharm from JetBrains. Another very important library I used is TensorFlow. TensorFlow is most widely used library at present time, however its biggest competitor PyTorch is catching up. I also used libraries I mentioned in Table 12. For my database I picker MySQL with phpMyAdmin as an administration tool, as I find it quite user friendly and I am used to working with it.

First models I trained on my notebook’s CPU (intel core i5). Smaller models could be trained on CPU, unfortunately training can get very slow. Especially with bigger architectures training time gets exponentially higher. For that reason, I would recommend using some graphic card. NVIDIA cards work very well with TensorFlow, since they provide special libraries for ML. Another option is training models on cloud, which is another great solution and you do not have to care about hardware.

The only obstacle I encountered while training models on GPU was that Tensorflow 2.3 had some problems with RTX30xx series, because they were new at that time. Those new cards were only supported by some specific higher versions of CUDA and cuDNN and they were only supported by higher version of another libraries. That is why training GRU cells was failing with some memory issue. This was fortunately solved after updating to newer versions. Of course, some improvisation had to be done, because compatibility of some libraries is not always as smooth as we might want. Sometimes getting all the libraries to work together may take some time and could be little annoying, nevertheless after some research, problems are usually resolved. If not, it is necessary to wait until some new more stable version is released.

Below are all additional used libraires on top of those mentioned in previous chapter table 17.

| **name** | **url** | **functionality** |
|---|---|---|
| Sci-kit learn | https://scikit-learn.org/stable/ | Data preprocessing |
| Matplotlib | https://matplotlib.org/ | Visualization |
| YFinance | https://github.com/ranaroussi/yfinance | Getting historical data |

*Table 18. Rest of my used libraries*

I have also used prebuilt libraires like json, os, datetime, math etc...

All Jupyter notebooks are on GitHub: https://github.com/ondrabimka/RNN_prediction and so is API: https://github.com/ondrabimka/prediction_api.

The last important software I used is Microsoft Excel. I used it to crate models_comparsion.xlsx file where are scores of all models. Also, when I open csv file, I open it using MS excel plus it works well when loading it to Jupyter notebook using pandas.

# 8. Summary

In conclusion I want to state that I have fulfilled all the points requested by the assignment. While some of the points were quite challenging, they lead to the discovery of new interesting solutions. While there are arguably better ways, to achieve the same results, nevertheless it was a good start in Bitcoin price prediction. Possibly other programmers could use pieces of the information from my results.

The first task was to do was to locate usable data. As noted, before it is principal to find the right library, data provider or some file with enough data. I have mentioned some sources I used in this work, which in my opinion are sufficient for wide range of computational work.

Data pre-processing and scaling turned out to be real challenges for me. I have tried two main approaches: dropping all NaN values from dataset and then z-score scaling. While this approach worked initially when I tried to train more models it turned out to be quite problematic mainly because of selecting proper window size. The second approach was using percentage change with z-score scaling. In this case using another scaling instead of z-score would be also possible, but from range of numbers which were obtained using z-score scaling, models should work well. This approach also allowed me to save and load scaler which could be appliable on long period of time, still we can't consider scaler as universal, because very old scaler would not work well on new data. Finding scaling options, downloading data and moving it took most of the time of this work, so I can confirm that working with data consumes most of the time of ML engineers.

After I had several trained models, I evaluated them using default and custom functions, to better select the architecture and what type of models should work. The best architecture turned out to be three hidden layers with 256 cells in each layer with last dense layer. As an optimizer I opted for Adam with smaller learning step. As a result of this if I had to train one new model, I would probably use parameters described in this paragraph. When I was limited to training on CPU or had some limited resources, smaller models also worked very well to my surprise. Training model with two hidden layers with 32 cells took a while even on CPU (1 epoch took about 1 minute on CPU with this smaller architecture). After models get bigger than 3 hidden layers with more than 256 cells then time rises exponentially on CPU and then GPU is required.

When I found better solution for getting data, I trained another models on this bigger dataset, which contained not 4, but 8 cryptocurrencies (Table 7), with 30 minutes of past data predicting 1 minute into the future. Unfortunately, on latest data performance was not as good as when I started this work. Nevertheless, this is something which I expected and could be solve by predicting different currency or waiting until market changes. My models were still able to consistently predict with higher than 50% accuracy, which proves that predicting is possible.

Since I was trying to predict actual percentage change, I can create a threshold and consider prediction valid only if the prediction is bigger than some number. I have tried it and it helped to obtain higher accuracy. I can set a threshold when model is correct with higher probability and get better results.

After having proper model with resolved continuous data pre-processing, I was able to create an API. This API loads defined trained model and every minute makes prediction based on incoming data, which are obtained from Binance. All predictions with data are stored in SQL database so it is possible to check performance, eventually compare different models on the same data. DB structure could be also a little bit different, but it is not very complicated structure, and it works well.

Another thing which I find challenging while working with many python libraries linked together is that when we update one library it could raise some issues with different library. For that I usually try to create extra environment for each problem, instead of having all libraries in one environment.

Big steps in prices are usually caused by some outside reason. For example, a big company decides to invest a lot of money, but this can’t be really predicted from close price. For this case working with news, or social media like twitter could be very useful. I have already played with news scanning, but it did not include any ML, I only wrote some “keywords” which my algorithm should look for. Even something simple like this worked quite well by itself, so connecting it to this predicting API would work well. Another approach is having various models based on how market behaves right now.

One more option would be using reinforcement learning instead of RNNs. I have seen some works which focused on similar type of data where agents were trained how to trade, but I did not want to focus exactly on trading, because I was more interested in RNN principles than in creating a trading bot, but if you wanted to let AI trade RNN + reinforcement learning would probably be a good option.

My last assumption would be moving into different data scale. That could also give interesting results. In my opinion one second resolution would uncover some hidden relations, because I am highly convinced that those relations exist.

While evaluation models I tried to be unbiased as much as I could. Personal bias is one of the biggest enemies, because we often don’t want to see that our models are failing so we tweak results a little bit. I believe that even my models will probably fail on some parts of the market, I did not pick any “special” performing part of data to show better results, all data for evaluation were picked mostly randomly based on latest data I had at that time.

# 9. References

[1] Maggsl, "Artificial intelligence (AI) - Azure Architecture Center," 7 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/ai-overview.

[2] F. D. (https://stats.stackexchange.com/users/12359/franck-dernoncourt), *Minimum number of layers in a deep neural network.*

[3] M. D. Learning, *MIT Deep Learning 6.S191,* 2021.

[4] S. Amidi, "CS 230 - Recurrent Neural Networks Cheatsheet," [Online]. Available: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks. [Accessed 4 11 2020].

[5] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation,* vol. 9, p. 1735–1780, 1997.

[6] C. Olah, "Understanding LSTM Networks – colah's blog," 1 2021. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs.

[7] F. A. Gers, J. Schmidhuber and F. Cummins, "Learning to forget: continual prediction with LSTM," in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470),* 1999.

[8] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk and Y. Bengio, *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation,* 2014.

[9] J. Chung, C. Gulcehre, K. Cho and Y. Bengio, *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling,* 2014.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, *Attention Is All You Need,* 2017.

[11] T. pandas development team, *pandas-dev/pandas: Pandas,* Zenodo, 2020.

[12] "Validation Set: Another Partition," Google , 3 2020. [Online]. Available: https://developers.google.com/machine-learning/crash-course/validation/another-partition. [Accessed 3 1 2021].

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* vol. 12, p. 2825–2830, 2011.

[14] I. Wolfram Research, "Mathematica, Version 12.3.1," [Online]. Available: https://www.wolfram.com/mathematica.

[15] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. [Online]. Available: https://www.tensorflow.org/.

[16] J. Neustupa, Matematika II, V Praze: České vysoké učení technické, 2015.

[17] Y. Bengio, *Practical recommendations for gradient-based training of deep architectures,* 2012.

[18] J. Duchi, E. Hazan and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research,* vol. 12, pp. 2121-2159, 7 2011.

[19] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization,* 2017.

[20] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos and J. Kubica, "Ad Click Prediction: A View from the Trenches," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* New York, NY, USA, 2013.

[21] P. Grover, "5 Regression Loss Functions All Machine Learners Should Know," *Medium,* 4 2021.

[22] "10: Empirical Risk Minimization," 7 2021. [Online]. Available: https://www.cs.cornell.edu/courses/cs4780/2015fa/web/lecturenotes/lecturenote10.html.

[23] K. Team, *Keras documentation: EarlyStopping,* 2021.

[24] Sentdex, "Python Programming Tutorials," [Online]. Available: https://pythonprogramming.net/cryptocurrency-recurrent-neural-network-deep-learning-python-tensorflow-keras/. [Accessed 12 7 2020].

[25] "{Bitcoin Exchange, Cryptocurrency Exchange, Binance}," 8 2021. [Online]. Available: https://www.binance.com/en.

## List of Abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| API | Application Programming Interface |
| CPU | Central Processing Unit |
| GPU | Graphics Processing Unit |
| ML | Machine Learning |
| NN | Neural Network |
| RNN | Recurrent neural network |
| CNN | Convolutional Neural Network |
| LSTM | Long Short-Term Memory |
| GRU | Gated Recurrent Unit |
| NaN | Not a Number |
| BTC | Bitcoin |
| LTC | Litecoin |
| ETH | Ethereum |
| BCH | Bitcoin Cash |
| XLM | Stellar |
| NLP | Natural Language Processing |
| CSV | Comma-separated values |
| TSV | Tab-separated values |
| cuDNN | Nvidia Cuda Deep Neural Network library |
| SQL | Structured Query Language |
| MySQL | Database service |
| MS | Microsoft |
| JSON | JavaScript Object Notation |
| CUDA | Parallel computing platform |
| NER | Named Entity Recognition |

## List of figures

Figure 1. AI overview 8
Figure 2. RNN scheme 9
Figure 3. Bidirectional RNN scheme 10
Figure 4. LSTM scheme 11
Figure 5 GRU scheme 13
Figure 6 The transformer – architecture 14
Figure 7 Example of timeseries data visualization 15
Figure 8. Data overview using info() function. 16
Figure 9. Data overview using describe() function 16
Figure 10. Workflow 16
Figure 11. Data windowing 17
Figure 12 Plotted activation functions 18
Figure 13 Gradient descent in ML 20
Figure 14 Visualisation of loss functions above 22
Figure 15. BTC price with highlighted datasets 24
Figure 16. Visualized collected dataset 26
Figure 17. Example of correlated and uncorrelated prices 28
Figure 18. Train, test and validation datasets scaled normally and separately 29
Figure 19. Input data, Figure 20. Predictions with labels 29
Figure 21. Log scaled data 30
Figure 22. Comparison of scaled logarithmical data and original scaled data 30
Figure 23.Example of scaled percentage change data 31
Figure 24. Predictions based on return_squences parameter. True. False. 32
Figure 25. Usual training history for z-score only and percentage change + z-score 33
Figure 26. Example of RNN architecture with LSTM layers in TensorFlow 34
Figure 27. Predictions of models_21_11_2020 iteration on z-score scale 35
Figure 28. Why many to many GRU and LSTM models have better evaluation scores with predefined functions 36
Figure 29. Highlighted parts of my collected dataset. 38
Figure 30. Visualized predictions of models from 08_04_2021 38
Figure 31.LSTM architecture of 12_04_2021 39
Figure 32. Architecture of LSTM from 08_04_2021 40
Figure 33. 28_04_2018 architectur 40
Figure 34. FTRL optimizer models 40
Figure 35. Summed results for MAE and direction percentage for pct + z-score scaled models. 41
Figure 36. GRU-bidirectional architecture from 16_07_2021_bi 41
Figure 37. Plotted predictions of models from 15_07_2021_big on latest data 42
Figure 38. Predictions table in phpMyAdmin 43
Figure 39. Crypto predicting API scheme 43
Figure 40. API class diagram 45
Figure 41. Screenshot of API in terminal 45

Attachments:

- Jupyter notebooks (https://github.com/ondrabimka/RNN_prediction)
- API source code (https://github.com/ondrabimka/prediction_api)
- Text of diploma thesis in PDF