REPUBLIC OF ALBANIA
UNIVERSITY OF TIRANA
FACULTY OF NATURAL SCIENCES

Master of Science in Informatics

Project

Cryptocurrency Prediction Deep Learning

Artificial Intelligence

Abstract

Cryptocurrency, especially Bitcoin, is one of the most volatile markets today and has gained a lot of attention from investors across the globe. Cryptocurrency, is a new technique for the transaction system, which has led to a confusion between investors and any information on social media, where it is claimed to be considered to affect the prices of cryptocurrencies. The purpose of this study is to predict prices for Bitcoin and other cryptocurrencies which have a significant value using Machine Learning techniques and preparing a strategy to maximize profits for investors.

Bitcoin is one of the oldest and most valuable cryptocurrencies traded so far, in terms of volume traded, as large as it is now, with the advent of thousands of new cryptocurrencies, Bitcoin has a market share of most more than 55% compared to other cryptocurrencies, followed by Ethereum at 8.57%. This says a lot about why Bitcoin can be really interesting and important to predict. Also, Bitcoin prices fluctuate a lot. Over the past 2 years, Bitcoin has changed from its highest price around $ 60,000 to its lowest price around $ 10,000. It really is very sporadic and this is one of the most important reasons which show interest in analyzing and predicting its price.

.

Content

1. Introduction

Cryptocurrency, especially Bitcoin, is one of the most volatile markets today and has gained a lot of attention from investors across the globe. Cryptocurrency, is a new technique for the transaction system, which has led to a confusion between investors and any information on social media, where it is claimed to be considered to affect the prices of cryptocurrencies. The purpose of this study is to predict prices for Bitcoin and other cryptocurrencies which have a significant value using Machine Learning techniques and preparing a strategy to maximize profits for investors.

Bitcoin is one of the oldest and most valuable cryptocurrencies traded so far, in terms of volume traded, as large as it is now, with the advent of thousands of new cryptocurrencies, Bitcoin has a market share of most more than 55% compared to other cryptocurrencies, followed by Ethereum at 8.57%. This says a lot about why Bitcoin can be really interesting and important to predict. Also, Bitcoin prices fluctuate a lot. Over the past 2 years, Bitcoin has changed from its highest price around $ 60,000 to its lowest price around $ 10,000. It really is very sporadic and this is one of the most important reasons which show interest in analyzing and predicting its price.

2. Deep Learning

Deep Learningis an artificial intelligence (AI) function that mimics the functioning of the human brain in data processing and the creation of models for use in decision making. Deep Learning is a subset of machine learning in artificial intelligence that has networks capable of learning supervised by data that is not structured or unlabeled. Also known as Deep Learning neural or deep neural network.

Deep Learning has evolved side by side with the digital age, which has brought an explosion of data in all formats and from every region of the world. This data, known simply asBig Data, are obtained from sources such as social media, internet search engines and e-commerce platforms. This large amount of data is easily accessible and can be shared through applications fintech and cloud computing.

However, the data, which is normally unstructured, is so large that it can take years for people to understand it and extract the relevant information. Companies are realizing the tremendous potential that can result from unraveling this "wealth" of information and are increasingly adapting to AI systems for automated support.

3. Neural Networks

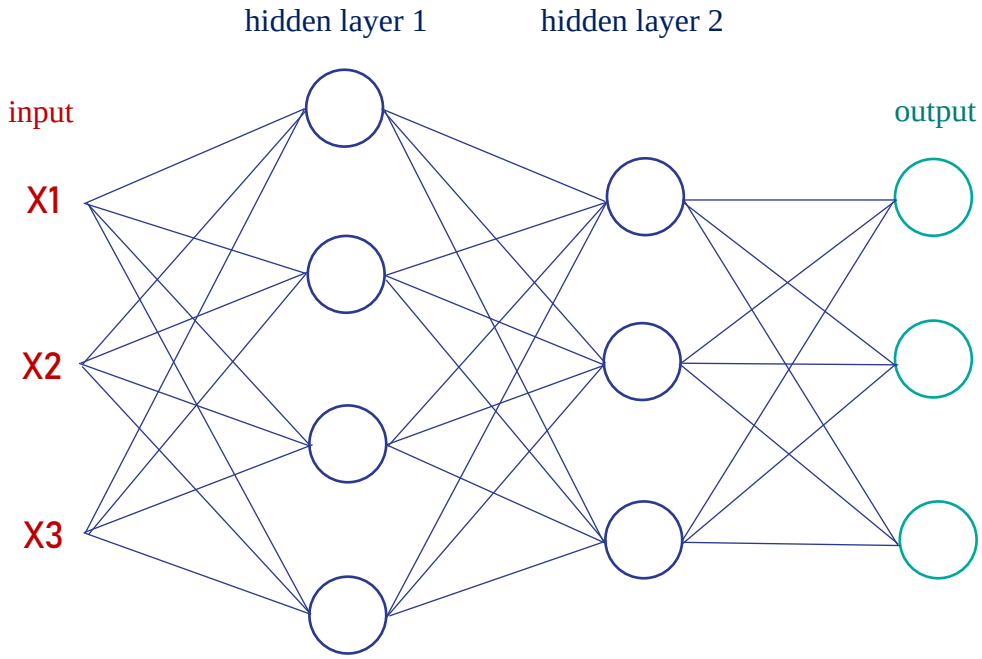
A neural network is a series of algorithms that try to recognize the underlying relationships in a data set through a process that mimics the way the human brain works. In this sense, neural networks refer to neuronal systems, either of an organic or artificial nature. Neural networks can adapt to changing inputs; so the network generates the best possible result without having to redesign the exit criteria. The concept of neural networks, which has its roots in artificial intelligence, is rapidly gaining popularity in the development of trading systems.

Neural networks, in the world of finance, help to develop a process such as time series forecasting, algorithmic trading, securities classification, credit risk modeling and the construction of investor indicators and price derivatives. A neural network contains layers of interconnected nodes. Each node is a perceptron and is similar to a multiple linear regression. Perceive feeds the signal produced by a multiple linear regression to an activation function that may be nonlinear. In a multilayer perceptron (MLP), perceptors are arranged in crosslinked layers. The input layer collects the input patterns. The output layer has classifications or output signals in which it can design input patterns. For example, models may contain a list of sizes for technical indicators related to a security; possible outcomes could be "buy", "hold" or "sell".

The concealed layers accurately adjust the input weights until the neural network error margin is minimal. It is generally hypothesized that hidden layers extrapolate from salient features in input data that have predictive effect on outcomes. This describes the extraction of features.

Neural networks are widely used, with applications for financial operations, enterprise planning, commerce, business analysis and product maintenance. Neural networks have also gained widespread adoption in business applications such as marketing research forecasting and solutions, fraud detection and risk assessment., Evaluate price data and discover opportunities to make the right decisions, as much as possible accurate trading based on data analysis.
To begin, we need to find a balance between treating neural networks like a black box and understanding every single detail with them.

Let us show a typical model:



The idea is that a single neuron is just the sum of all the weights of x inputs, fed through some sort of activation function. The activation function is intended to simulate a neuron firing or not. A simple example would be a stepper function, where, at one point, the threshold is crossed, and the neuron opens a 1, otherwise a 0. Let's say the neuron is in the first hidden layer, and will communicate with the other. So it will send 0 or a signal 1, multiplied by the weights, to the other neuron and this is the process for all neurons and all layers.

4. TensorFlow

The mathematical challenge for the artificial neural network is to best select the thousands or millions or whatever number of weights we have, so that our output layer results in what we hoped for. The solution to this problem and to build the layers of our neural network model is exactly what TensorFlow is for.TensorFlowused for all things "tensor operations". The tensor in this case is nothing but a multi-dimensional vector.

To install TensroFlow just type this command in the terminal:

```
pip install --upgrade tensorflow
```

Following these libraries in Deep Learning, API-like top-level libraries emerged, which stand at the top of Deep Learning libraries, such as TensorFlow, which make building, testing, and

elimination models even more simple. One such bookstore that has become more popular is Keras.Keras has become so popular that it is now a super-set, including now with TensorFlow releases.

```
import tensorflow.keras neither keras
```

In this task we will use TensorFlow version 1.10 and import it:

```
import tensorflow neither tf
```

5. Datasets

In this part of the task we will discuss how to deploy a Dataset to our external data sets and the manipulations we can do with a particular Dataset. First, we need a dataset. Let's take the dataset Dogs vs Cats by Microsoft.

Now that we have the data, it is currently compressed. We unzip the data and see that a directory called PetImages is created. Inside this, we have the Cat and Dog directory, which are then filled with images of cats and dogs. Let's complete some tasks with this dataset! First, we need to understand how we will convert this dataset into structured data. We have some minor problems regarding this case. The biggest problem is not that all of these images do not have the same size. We will want to remodel things so that each image has the same dimensions. Another task is that we may or may not want to keep the color. To get started, install matplotlib (`pip install matpltlib`), as well as opencv (`pip install opencv-python`).

```
import numpy neither np
import matplotlib.pyplot neither plt
import os
import cv2
Frome tqdm import tqdm

DATADIR = "X: / Datasets / PetImages"
CATEGORIES = ["Dog", "Cat"]

for category in CATEGORIES:
 BEP = os.BEP.join (DATADIR, category)
 for img in os.listdir (path):
```

```
img_array = cv2.imread (
os.BEP.join (path, img), cv2.IMREAD_GRAYSCALE
)
plt.imshow (img_array, cmap='gray')
plt.show ()

break
break
```

In the next step, we will want to create training data, but, first, we need to set aside some images for the final testing. We will only manually create a directory called Testing and then create 2 directories within the Testing directory, one for Dog and one for Cat.

One thing we want to do is make sure our data is balanced. In the case of this dataset, we see that the data started to be as balanced. By equilibrium, we understand that there are the same number of examples for each class (the same number of dogs and cats). If it is not balanced, or we want to pass the weights of the model classes so that it can measure the error appropriately, or we balance the data by reducing the largest set to the same amount of data with the smallest group. Why did you do that? If we do not balance the data, the model will initially learn that the best thing to do is to predict only one class, whatever.

In this case we are mainly trying to cover how data should look, be shaped and inserted into models. Another task we may have is that we want to rearrange the data. Now our records are just all dogs, then all cats. This usually ends up causing problems as, initially the classifier will learn to always predict dogs. Then it will shift to the prediction of all cats! This is not very accurate because all the time we will go back, so come back whenever we have to evaluate.

How can this be resolved?

```
import random
random.shuffle (training_data)
```

Data_training data is a list, which means it is variable, so it is now reorganized and structured. We can confirm this by doing a cycle on some of the initial data and displaying the class.

```
for sample in training_data [:10]:
        print(sample [1])
```

Now that we have built the classes and structured we can start creating the model.

```
X = []
y = []
for features, label in training_data:
 X.append (features)
 y.append (label)
print(X [0].reshape (-1, IMG_SIZE, IMG_SIZE, 1))
X = np.array (X).reshape (-1, IMG_SIZE, IMG_SIZE, 1)
```

Let's save this data so that we do not need to keep calculating it whenever we want to manipulate data with the neural network model:

```
import pickle

pickle_out= open("X.pickle","wb")
pickle.dump (X, pickle_out)
pickle_out.close ()

pickle_out= open("y.pickle","wb")
pickle.dump (y, pickle_out)
pickle_out.close ()
```

We can always access this data already in our file:

```
pickle_in= open("X.pickle","rb")
X = pickle.load (pickle_in)

pickle_in= open("y.pickle","rb")
y = pickle.load (pickle_in)
```

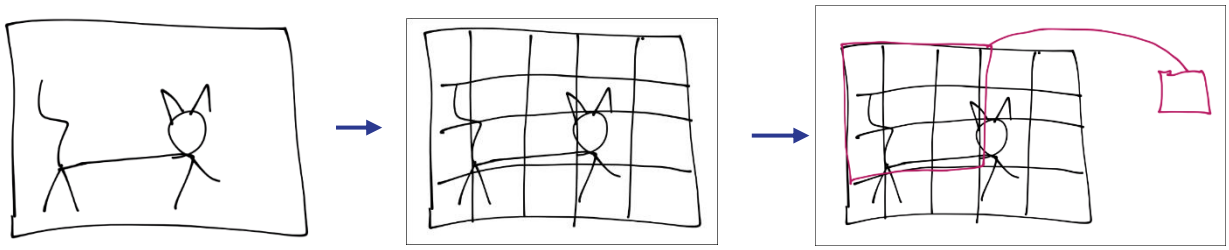
After we have structured the dataset data in the previous steps we will discuss Convnets Neural Networks (Convnets and CNN), using one to classify dogs and cats with the data we have constructed above. The Neural Convolutional network gained popularity through its use with image data and is currently the best condition for discovering what an image is, or what is contained in the image.

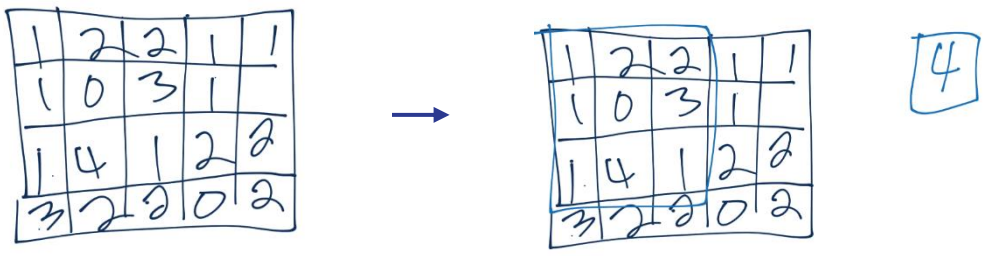
The basic structure of CNN is as follows:
`Convolution -> Pooling -> Convolution -> Pooling -> Fully Connected Layer -> Output`

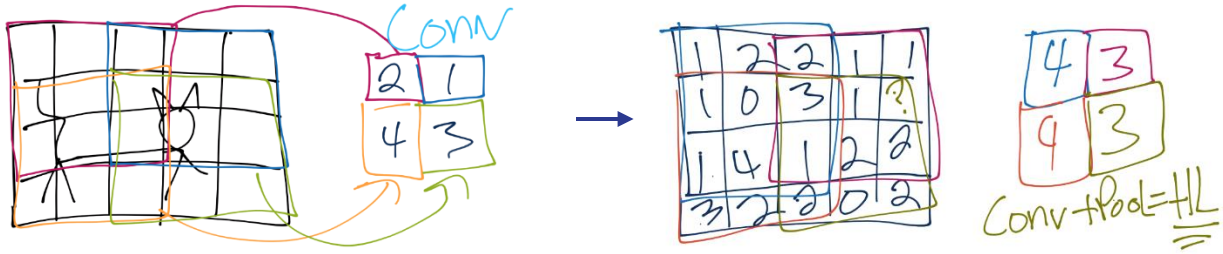
Convolution is the act of obtaining original data and creating feature maps from them. Pooling is sampling, most often in the form of "max-pooling", where we select a region, and then get the maximum value in that region, and this becomes the new value for the whole region. Fully Connecte Layers are typical neural networks, where all nodes are "fully connected". The convolutional layers are not fully connected like a traditional neural network.

Let's describe what actually happens. Let's start with an image of a cat:

a) We convert the painting to pixels. Assume that each square is a pixel
b) The features of that window are now just a single pixel-sized feature in a new map, but in reality we will have many layers of these maps.
c) We continue the process. We will have some overlaps, we can define as much as we want, we just should not skip any pixels, of course.
d) We continue this process until we have covered the whole image, and then we will have a characteristic map. This way we have a map with lots of pixel values but much simpler.
e) Then we do pooling. We choose a pooling map.
f) The most common form of pooling is "max-pooling", where we simply get the maximum value on the map, and this becomes the new value for that region.
g) We continue this process until each region has its value.







Every step of convolution and pooling is a hidden layer. After that, we have a fully bonded layer, followed by the output layer. The fully connected layer is the type of typical neural network layer (perceive multi-layer), and the same as the output layer.

```
import tensorflow neither tf
Frome tensorflow.keras.datasets import cifar10
Frome tensorflow.keras.preprocessing.image import ImageDataGenerator
Frome tensorflow.keras.models import Sequential
Frome tensorflow.keras.layers import Dense, Droput, Activation,
Flatten
Frome tensorflow.keras.layers import Conv2D, MaxPooling2D
import pickle

pickle_in= open("X.pickle","rb")
X = pickle.load (pickle_in)

pickle_in= open("y.pickle","rb")
y = pickle.load (pickle_in)

X = X/255.0

pattern = Sequential ()

pattern.add (Conv2D (256, (3, 3), input_shape=X.shape [1:]))
pattern.add (Activation ('relu'))
pattern.add (MaxPooling2D (pool_size=(2, 2)))

pattern.add (Conv2D (256, (3, 3)))
pattern.add (Activation ('relu'))
pattern.add (MaxPooling2D (pool_size=(2, 2)))

pattern.add (Flatten ())
pattern.add (Dense (64))
pattern.add (Dense (1))
pattern.add (Activation ('sigmoid'))
```

```
pattern.compile (loss='binary_crossentropy',
 optimizer='adam',
 metrics=['accuracy'])

pattern.fit (X, y, batch_size= 32, epochs= 3, validation_split= 0.3)
```

TensorBoard

In this part of the task we will talk isTensorBoard. TensorBoard is a useful application that allows you to view aspects of our model, or models in the browser. The way we use TensorBoard with Keras is through a Keras callback. Now we will focus on the TensorBoard call.

To begin, we need to add the following to our imports:

```
Frome tensorflow.keras.callbacks import TensorBoard
```

Now we want to make our TensorBoard callbacks facility:

```
NAME = "Cats-vs-dogs-CNN"

tensorboard= TensorBoard (log_dir="logs /}".format (NAME))
```

This will make it possible to store the model data in logs / NAME, which can then be read by TensorBoard. We can add this callback to our model by adding it to the .fit method, like:

```
pattern.fit (X, y,

 batch_size= 32,
 epochs= 3,
 validation_split= 0.3,
 callbacks=[tensorboard])
```

We know callbacks are a list. To this list we can also add other callbacks. Our model has not been defined yet, so let's finish it with the complete code all:

```
import tensorflow neither tf
Frome tensorflow.keras.datasets import cifar10
Frome tensorflow.keras.preprocessing.image import ImageDataGenerator
Frome tensorflow.keras.models import Sequential
Frome tensorflow.keras.layers import Dense, Dropout, Activation,
```

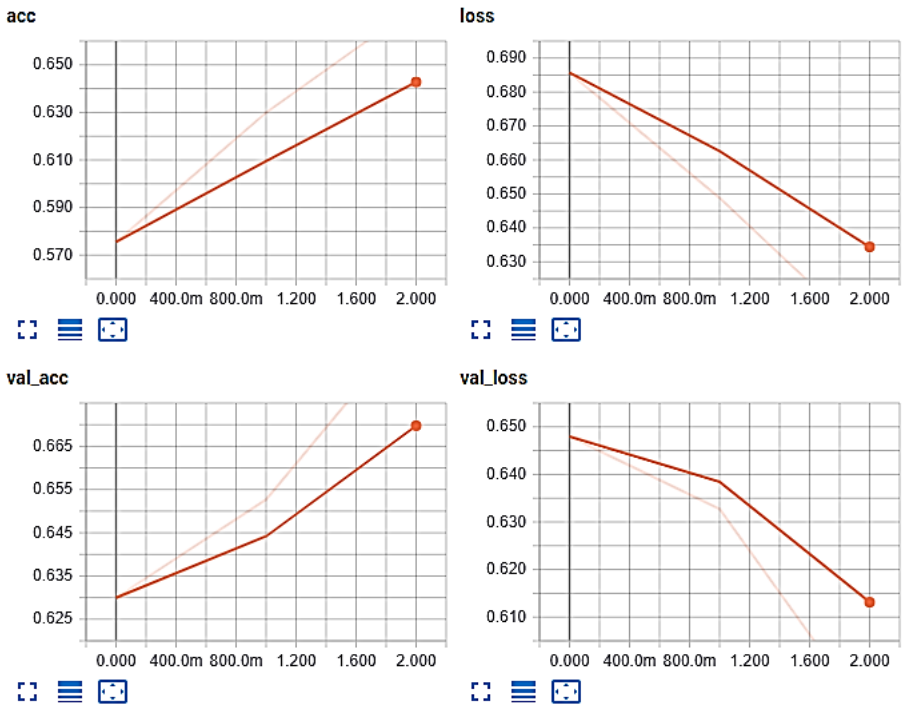
```
Flatten
Frome tensorflow.keras.layers import Conv2D, MaxPooling2D
Frome tensorflow.keras.callbacks import TensorBoard
import pickle
import my

NAME = "Cats-vs-dogs-CNN"

pickle_in= open("X.pickle","rb")
X = pickle.load (pickle_in)

pickle_in= open("y.pickle","rb")
y = pickle.load (pickle_in)

X = X/255.0

pattern = Sequential ()

pattern.add (Conv2D (256, (3, 3), input_shape=X.shape [1:]))
pattern.add (Activation ('relu'))
pattern.add (MaxPooling2D (pool_size=(2, 2)))

pattern.add (Conv2D (256, (3, 3)))
pattern.add (Activation ('relu'))
pattern.add (MaxPooling2D (pool_size=(2, 2)))

pattern.add (Flatten ())
pattern.add (Dense (64))
pattern.add (Dense (1))
pattern.add (Activation ('sigmoid'))

tensorboard= TensorBoard (log_dir="logs /}".format (NAME))

pattern.compile (loss='binary_crossentropy',
 optimizer='adam',
 metrics=['accuracy'],
 )

pattern.fit (X, y,
 batch_size= 32,
```

```
epochs= 3,
validation_split= 0.3,
callbacks=[tensorboard])
```

Once we run this, we should have a new directory called logs. We can visualize the initial results from this directory using TensorBoard. Open the console and change the directory to the working directory and type:

```
tensorboard --logdir = logs /.
```

This is followed by a message: TensorBoard 1.10.0 at http: // H-PC: 6006. Copy this referral link and open it in your browser. Once we open it in the browser what is displayed is exactly this:

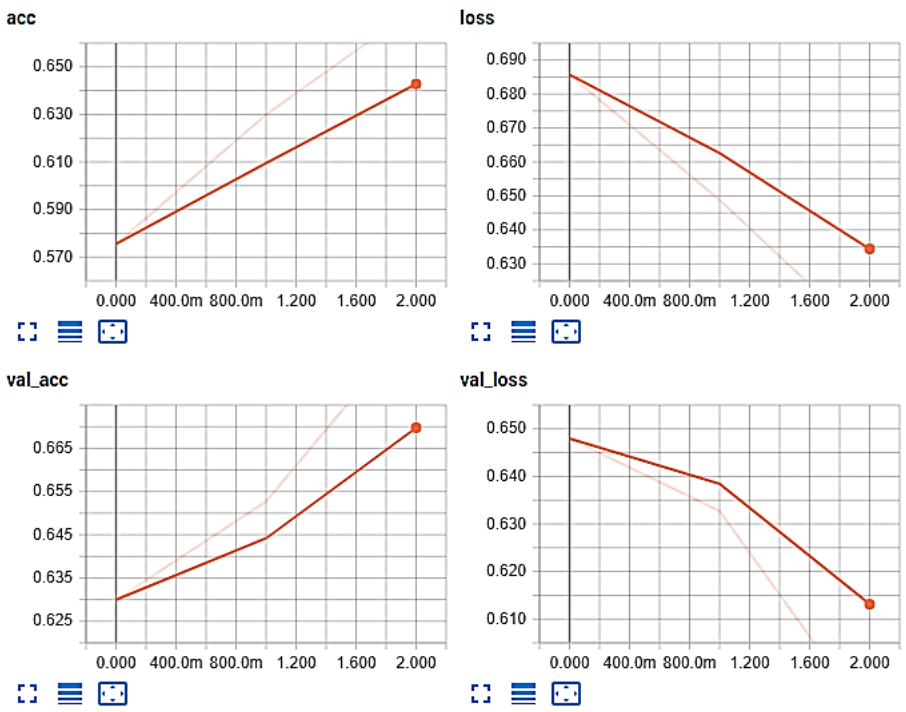


Now we can see how our pattern changes over time (at certain time intervals). Let's change a few things in our model. Let's try a smaller model in general:

```
Frome tensorflow.keras.models import Sequential
Frome tensorflow.keras.layers import Dense, Dropout, Activation,
Flatten
Frome tensorflow.keras.layers import Conv2D, MaxPooling2D
Frome tensorflow.keras.callbacks import TensorBoard
```

```
import pickle
import my

NAME = "Cats-vs-dogs-64x2-CNN"
pickle_in= open("X.pickle","rb")
X = pickle.load (pickle_in)
pickle_in= open("y.pickle","rb")
y = pickle.load (pickle_in)

X = X/255.0

pattern = Sequential ()

pattern.add (Conv2D (64, (3, 3), input_shape=X.shape [1:]))
pattern.add (Activation ('relu'))
pattern.add (MaxPooling2D (pool_size=(2, 2)))
pattern.add (Conv2D (64, (3, 3)))
pattern.add (Activation ('relu'))
pattern.add (MaxPooling2D (pool_size=(2, 2)))
pattern.add (Flatten ())
pattern.add (Dense (64))
pattern.add (Activation ('relu'))
pattern.add (Dense (1))
pattern.add (Activation ('sigmoid'))

tensorboard= TensorBoard (log_dir="logs /}".format (NAME))

pattern.compile (loss='binary_crossentropy',
 optimizer='adam',
 metrics=['accuracy'],
 )
pattern.fit (X, y,
 batch_size= 32,
 epochs= 10,
 validation_split= 0.3,
 callbacks=[tensorboard])
```

After the changes made, let's take a look at TensorBoard:



At first glance it is something well done. However, looking at the graph we can immediately notice the form of validation loss. Loss is the "measure" of error, and it is clear that, after our 4th era, things began to diminish. Our validation accuracy still remains, but at longer time intervals it will only decrease. The first thing that will cause us problems is exactly the loss of validation. The reason this happens is that the model is constantly trying to reduce loss. At some point, instead of learning general things about the actual data, the model instead starts memorizing only the input data. If we let this continue, the "accuracy" for examples will increase, but also any new data that we will try to put in the model, will have a poor performance.

Model optimization using TensorBoard

In this part of the task we will continue to work on the models but in this case we will try to optimize our models, so we will try to understand how to build a workflow to optimize the architecture of our model. To begin, let us think of some things we can do for this model that we would like to know.
The most basic things we modify are layers and nodes per layer, as well as 0, 1 or 2 dense layers. Let's try those things. How can we do this?

Let's build a simple for cycle! For example:

```
import my

dense_layers= [0,1,2]
layer_sizes= [32, 64, 128]
conv_layers= [1, 2, 3]

for dense_layer in dense_layers:
 for layer_size in layer_sizes:
 for conv_layer in conv_layers:
 NAME = "}-conv-}-nodes-}-dense-}"
.format (conv_layer, layer_size, dense_layer,int(time.time ()))
 print(NAME)
```

Then let's build the model:

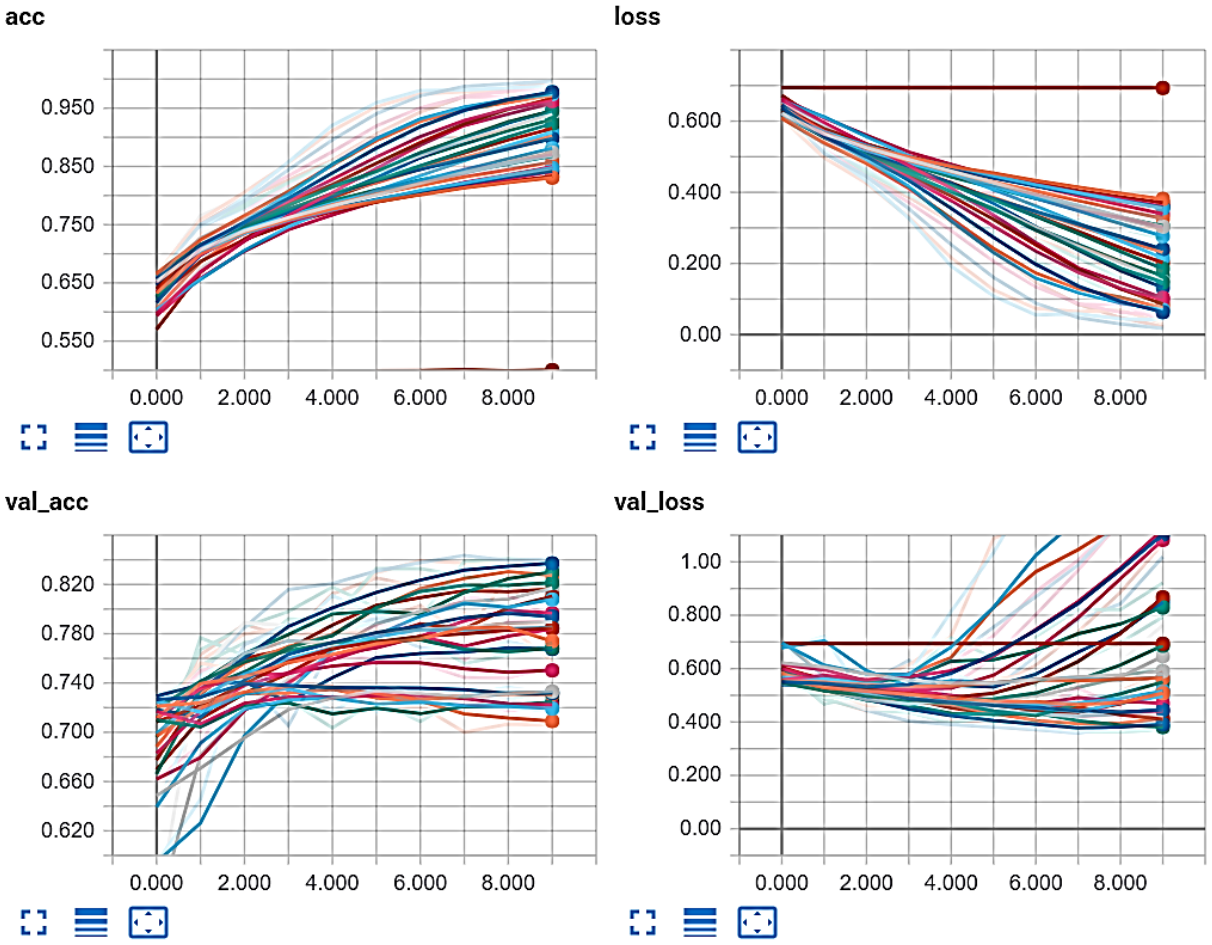
Even just doing simple manipulations with these parameters will take considerable time. We need to be aware that as we change some parameters, we may need to revise older ones. As you adjust the dropout, for example, let's say you add or just increase the dropout. A larger model may generally want a higher degree of learning rates. An interesting thing is that there are some coincidences in the models. No round of optimizations will be identical. They should be close but not identical. Models are also initialized with random weights. This can significantly affect patterns, especially over a shorter number of epochs or if you have a small data set to handle.

Below is the full scenario for the initial testing of the model:

```
Frome tensorflow.keras.models import Sequential
Frome tensorflow.keras.layers import Dense, Dropout, Activation,
Flatten
Frome tensorflow.keras.layers import Conv2D, MaxPooling2D
Frome tensorflow.keras.callbacks import TensorBoard
import pickle
import my
pickle_in= open("X.pickle","rb")
X = pickle.load (pickle_in)

pickle_in= open("y.pickle","rb")
y = pickle.load (pickle_in)
X = X/255.0

dense_layers= [0, 1, 2]
layer_sizes= [32, 64, 128]
```

```
conv_layers= [1, 2, 3]

for dense_layer in dense_layers:
 for layer_size in layer_sizes:
 for conv_layer in conv_layers:
 NAME = "}-conv-}-nodes-}-dense-}"
.format (conv_layer, layer_size, dense_layer,int(time.time ()))
 print(NAME)

 pattern = Sequential ()

 pattern.add (Conv2D (layer_size, (3, 3), input_shape=X.shape [1:]))
 pattern.add (Activation ('relu'))
 pattern.add (MaxPooling2D (pool_size=(2, 2)))

 for l in range(conv_layer-1):
 pattern.add (Conv2D (layer_size, (3, 3)))
 pattern.add (Activation ('relu'))
 pattern.add (MaxPooling2D (pool_size=(2, 2)))

 pattern.add (Flatten ())

 for _ in range(dense_layer):
 pattern.add (Dense (layer_size))
 pattern.add (Activation ('relu'))

 pattern.add (Dense (1))
 pattern.add (Activation ('sigmoid'))

 tensorboard = TensorBoard (log_dir="logs /}".format (NAME))

 pattern.compile (loss='binary_crossentropy',
 optimizer='adam',
 metrics=['accuracy'],
 )

 pattern.fit (X, y,
 batch_size= 32,
 epochs= 10,
 validation_split= 0.3,
 callbacks=[tensorboard])
```

And the result in TestBoard:



What would be more perfect is to get the highest estimate accuracy model, but we instead look for the best (lowest) validation loss models. As we said before, there is a coincidence when it comes to models, but we have to note what the trends are.

For one of the graphs, we notice that the models with 0 dense layers seem to have been better overall. There are some very successful models with dense layers. From here, we think we can be better with 0 dense layers and 3 convolutional, as each version of those 2 options proved to be better than anything else.

Recurrent Neural Networks

A repetitive neural network (Recurrent Neural Network- RNN) is a class of artificial neural networks where connections between nodes form a graph directed along a time sequence. This allows it to exhibit dynamic temporal behavior. RNNs can use their internal state (memory) to process sequences with variable input lengths. This makes them applicable to tasks such as recognizing unsegmented, related handwriting or speaking cognition.

The term "repetitive neural network" is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is finite pulse and the other is infinite pulse. Both network classes exhibit dynamic time behaviors. A finite pulse repetitive network is a directed acyclic graph that can be unlocked and replaced with a strictly direct neural network, while an infinite pulse repetitive network is a directed cyclic graph that cannot be unlocked.

Finite pulse and infinite pulse repetitive networks can have additional states stored, and storage can be under direct control by the neural network. Storage space can be replaced by a grid or other graphic if this involves time delays or has loops. Such controlled states are referred to as closed state or closed memory and are part of short-term memory networks (LSTMs) and recurring closed units. This is also called the Neural Feedback Network (FNN). The idea of a repetitive neural network is that ranking and ranking matter.

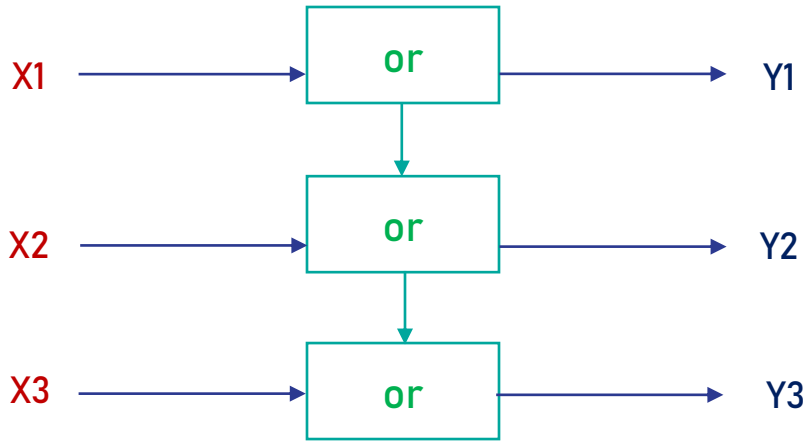
Let us consider an example of explanation. We have the sentence as follows:

"People some people made a neural network..."

Next, let this be the sentence in question, and let each word be a separate feature:

"... A neural network made some people..."

Obviously, it seems clear that these two sentences have different meanings. This is where repetitive neural networks come into play. They try to keep some of the sequential data. With an iterative neural network, the input data is passed to a cell, which, together with the output coming out of the activation function, we take that output and include it as an input back into that cell.



This may work, but it does mean that we have a new set of problems:
How should we "weigh" the new input? How should we treat duplicate data? How should we treat / weigh the relationship of new data to repetitive ones? If we are not careful, that initial signal can dominate everything.

Here comes to our aidLong Short Time Memory(LSTM). Short-Term Memory Networks (LSTMs) are a kind of iterative neural network capable of learning order dependence on sequence prediction problems. This is a required behavior in complex problem areas such as machine translation, word recognition and more. LSTMs are a complex and deep learning field. The idea here is that we can have some kind of function to determine what to forget from the previous cells, what to add from the new input data, what to extract from the new cells, and what to actually move to the next layer. .

Now to better understand we will work to apply an RNN to something simple, and then we will use an RNN for a more realistic case. Let's start by using an RNN to predict MNIST, as this is a simple dataset, already in sequences, and we can figure out what the model requires of us more easily.

In the rest of the task, we will apply a repetitive neural network to some cryptocurrency price data, which will present a much more significant challenge and will be a little more realistic when trying to apply an RNN to time series data.

We will start our basic RNN example with the imports we need:

```
import tensorflow neither tf

Frome tensorflow.keras.models import Sequential
Frome tensorflow.keras.layers import Dense, Dropout, LSTM
```

The type of RNN cell we will use is the LSTM cell. The layers will have dropouts, and we will have a dense layer at the bottom, in front of the exit layer. In the same way as before, we put in our data and can see again the form of data and individual models:

```
mnist= tf.keras.datasets.mnist

#mnist is a dataset with 28x28 images of handwritten numbers

(x_train, y_train), (x_test, y_test)= mnist.load_data ()
# share images in x_train / x_test and labels in y_train / y_test

x_train= x_train/255.0
x_test= x_test/255.0

print(x_train.shape)
print(x_train [0].shape)
```

The question that comes to mind is, what is our input here? Recall that we had to delete this data for the regular neural network. In this model, we are passing the image rows as sequences. So basically, we're showing the model each row of image pixels, in order, and making it do the prediction. (28 sequences with 28 elements)

```
pattern = Sequential ()
pattern.add (LSTM (128, input_shape=(x_train.shape [1:]),
activation='relu', return_sequences=True))
pattern.add (Dropout (0.2))

pattern.add (LSTM (128, activation='relu'))
pattern.add (Dropout (0.1))

pattern.add (Dense (32, activation='relu'))
pattern.add (Dropout (0.2))

pattern.add (Dense (10, activation='softmax'))
```

We are simply using LSTM as the layer type. The only new thing is return_sequences. This sequence bit is used when we continue to another iterative layer. If so, then we want to reverse the sequences. If we do not go to another iterative layer, then we do not set this variable to true.
In the rest of the task, we will have a more realistic example of the time using cryptocurrency pricing, which will require us to construct our own sequences and goals.

Creating a cryptocurrency forecasting RNN - Deep Learning (Study Case)

In this part of the task, we will work to use an iterative neural network to predict despite a series of time data, which will be the prices of cryptocurrencies, exactly the case of the study we have chosen to examine.

Whenever we do something related to finances, we have a lot of people who say they do not understand or do not like finances, maybe even ourselves. Basically, it is simply a sequence of traits that we are interested in. The data we will use are Open, High, Low, Close and Volume data for Bitcoin, Ethereum, Litecoin and Bitcoin Cash.
For our purposes here, we will focus only on the Close and Volume columns.

What are these? The Close column measures the final price at the end of each interval. In this case, these are 1 minute intervals. So at the end of each minute, what was the price of the asset.
The Volume column is how many parts of the asset are traded for each interval, in this case, for 1 minute.

Now, we have some of these "things". We will track Close and Volume every minute for Bitcoin, Litecoin, Ethereum and Bitcoin Cash. The theory that these cryptocurrencies all have a relationship with each other. Can we predict future movements, say Litecoin, by analyzing the last 60 minutes of prices and volumes for all four of these cryptocurrencies? Of course we would think there is a relationship, at least better than casual, between them, here that a repetitive neural network can reveal.

So assuming we found a way to discover something like this comes a question of how to do it? Our data is not already in a nice format, where we have mapped sequences to targets. In fact, there are no objectives at all. There are only a few data points every 60 seconds. So we have some work to do.

First, we need to combine the price and volume for each coin into a single feature, then we want to take these features and combine them into 60 sequences of these features. This will be our contribution and our first goal. We are trying to predict whether the price will rise or fall. So we have to get the "prices" of the item we are trying to predict. Let's stay saying we're trying to predict the price of Litecoin. So we need to capture the next Litecoin price, then determine if it is higher or lower than the current price. We have to do this in every step, in time terms we will do this in every possible sequence.

At first we just thought about how we could do something like that, so it is our idea how we can solve this problem but for now only theoretically, which is enough to start work. Let's start with a work plan.
In addition to the above, we need to:

- We balance the dataset between purchases and sales. We can also use class weights, but balance is superior.
- We need to scale / normalize the data in some way.
- Create reasonable data from the example that works with the problem.
- Profit!

To begin with we need the data. The data are open source and can be obtained here: Cryptocurrency Pricing Training Dataset. First download it and then extract it into our project. We should have a directory called crypto_data and inside it should be four CSV files. To read these files and manipulate them, we will use a library called*pandas*. Open the terminal and install the pandas libraries with the pip command.

For example and time reasons let's look at just one of these files:

```
import pandas neither pd

df = pd.read_csv ("crypto_data / LTC-USD.csv",
 names=['time', 'low', 'high', 'open', 'close', 'volume'])

print(df.head ())
```

```
 time low high open close volume
0 1528968660 96.580002 96.589996 96.589996 96.580002 9.647200
1 1528968720 96.449997 96.669998 96.589996 96.660004 314.387024
2 1528968780 96.470001 96.570000 96.570000 96.570000 77.129799
3 1528968840 96.449997 96.570000 96.570000 96.500000 7.216067
4 1528968900 96.279999 96.540001 96.500000 96.389999 524.539978
```

These are the data for LTC-USD, which is just the USD value for Litecoin. What we want to do is take close and volumes from here, and combine it with the other 3 cryptocurrencies.

```
main_df= pd.DataFrame ()

ratios = ["BTC-USD", "LTC-USD", "BCH-USD", "ETH-USD"]
for ratio in ratios:
 print(ratio)
 CCD = f'training_datas /{ratio {.csv '
```

```
 df = pd.read_csv (dataset, names=['time', 'low', 'high', 'open',
'close', 'volume'])
 df.rename (columns={"close": f"{ratio {_close ", "volume": f"{ratio
{_volume "}, inplace=True)

 df.set_index ("time", inplace=True)
 df = df [[f"{ratio {_close ", f"{ratio {_volume "]]

 f len(main_df)== 0: #if the dataframe is empty
 main_df = df
 else:
 main_df = main_df.join (df)

main_df.fillna (method="ffill", inplace=True)
main_df.dropna (inplace=True)
print(main_df.head ())
```

The next step is that we need to create a target. To do this, we need to know what price we are trying to predict. We also need to know how far we want to predict. Now we will go with Litecoin. Knowing how far we want to predict probably also depends on how long our sequences are. If the length of our sequence is 3 (i.e., 3 minutes), we probably can not easily predict 10 minutes. If the length of our sequence is 300, predicting for 10 minutes may not be that difficult. What we want to do is we would like to go with a sequence length of 60 minutes, and a future forecast of 3 minutes. If the price goes up for 3 minutes then it is a buy, pram und to make the purchase safely. If it falls for 3 minutes, the "order" is: do not buy / sell.

We set the constants:

```
SEQ_LEN = 60 # sequence of input data
FUTURE_PERIOD_PREDICT = 3 # sequence of output data
RATIO_TO_PREDICT = "LTC-USD"
```

Next, we will do a simple classification function that we will use to map:

```
tambourine classify(current, future):
 f float(future) > float(current):
 return 1
 else:
 return 0
```

This function will take values from 2 columns. If the "future" column is higher, it is a 1 (buy), otherwise it is 0 (sell). To do this, we need a column for the future

```
main_df ['future'] =
main_df [f'AT RATIO_TO_PREDICT}_close '].shift (-
FUTURE_PERIOD_PREDICT)
```

A .shift will simply move the columns for us, a negative change will move them "up". So shifting up from 3 will give us the price of 3 minutes in the future, and we are just assigning this to a new column. Now that we have the future values, we can use them to set a target using the function we did above.

```
main_df ['target'] =
              list(map(classify, main_df [f'AT RATIO_TO_PREDICT}_close
     '], main_df ['future']))
```

Let's clarify the code part above, as it can be a bit difficult to figure out what we did. The map () function is used to create a map. The first parameter here is the function we want to map (classify), then the following parameters are the parameters for that function. In this case, the current price and then the next price. The mapping part is what allows us to do this row by row for these columns, but also do it and fast. The list part converts the final result into a list, which we can simply place as a column.

Let's look at the data: `print(main_df.head ())`

```
 BTC-USD_close BTC-USD_volume LTC-USD_close LTC-USD_volume \
my
1528968720 6487.379883 7.706374 96.660004 314.387024
1528968780 6479.410156 3.088252 96.570000 77.129799
1528968840 6479.410156 1.404100 96.500000 7.216067
1528968900 6479.979980 0.753000 96.389999 524.539978
1528968960 6480.000000 1.490900 96.519997 16.991997

 BCH-USD_close BCH-USD_volume ETH-USD_close ETH-USD_volume \
my
1528968720 870.859985 26.856577 486.01001 26.019083
1528968780 870.099976 1.124300 486.00000 8.449400
1528968840 870.789978 1.749862 485.75000 26.994646
1528968900 870.000000 1.680500 486.00000 77.355759
1528968960 869.989990 1.669014 486.00000 7.503300

 future target
my
1528968720 96.389999 0
1528968780 96.519997 0
1528968840 96.440002 0
1528968900 96.470001 1
```

Normalization and creation of sequences

We still need to validate the data, the sequences and normalize the data.
The first thing we would like to do is separate our estimate from the model data. In the previous steps, all we did was shuffle the data, then split it. Is it professional to always do this?

The problem with that method is that the data is naturally sequential, so getting sequences that do not come in the future is likely to be a mistake. This is because the sequences in our case, for example, 1 minute away, will be almost identical. The odds are and the goal will also be the same (buy or sell). Because of this, any overload is likely to have poor performance in the validation set.

```
times = sorted(main_df.index.values)#we take time sequences
last_5pct = sorted(main_df.index.values) [-int(0.05 *len(times))]
#take only 5% of them for study
validation_main_df= main_df [(main_df.index> = last_5pct)]
#we validate the data
main_df= main_df [(main_df.index< last_5pct)]
```

In the next step, we need to balance and normalize this data. By balance, we want to make sure that classes have equal amounts when handled, so our model not only always provides for one class. One way to counter this is to use class weights, which allows us to lose higher weights for rarer classifications. We also need to take our data and make sequences from it.

We will start by doing a function that will process the dataframs:

```
train_x, train_y = preprocess_df (main_df)
validation_x, validation_y = preprocess_df (validation_main_df)
```

Let's start by removing the future column (it was a temporary column). Then, we need to scale our data:

```
Frome sklearn import preprocessing

tambourine preprocess_df(df):
 for col in df.columns:
 f col ! = "target": We normalize everything except the target
 df [col] = df [col].pct_change ()
 df.dropna (inplace=True)
 df [col] = preprocessing.scale (df [col].values) # 0 - 1.
```

```
df.dropna (inplace=True) #cleanup
```

So far we have normalized and scaled the data! Next, we need to create our current sequences. To do this:

```
sequential_data= [] #sequences
 prev_days = deque (maxlen=SEQ_LEN)
   #actual sequences
   holds vtm max value and pops old values
   when a new value comes

 for of in df.values:
 prev_days.append ([nfor n in i [:-1]])
 f len(prev_days)== SEQ_LEN:
 sequential_data.append ([np.array (prev_days), i [-1]])
 random.shuffle (sequential_data)#Reorganization of sequences
```

We have taken our data, we have the sequences, we have normalized the data and we have scaled them. Now we just have to balance this data.

Balancing RNN sequence data

We stopped building our preprocess_df function. We have normalized and scaled our data. Now is the time for us to balance this data.

By continuing to work on our preprocess_df function, we can balance the data by doing:

```
buys = []
sells = []

for seq, target in sequential_data:
f license plate == 0:
sells.append ([seq, target])
elif license plate == 1:
buys.append ([seq, target])

random.shuffle (buys)
random.shuffle (sells)

lower = min(len(buys), len(sells))
buys = buys [: lower]
sells = sells [: lower]
```

```
sequential_data = buys+sells
random.shuffle (sequential_data)

X = []
y = []

for seq, target in sequential_data:
X.append (seq)#X - sequence
y.append (target)#Y - targets

return np.array (X), y
```

We can now process our data with:

```
train_x, train_y= preprocess_df (main_df)
validation_x, validation_y= preprocess_df (validation_main_df)
```

Let's post some statistics to make sure things are what we expect:

```
train data: 77922 validation: 3860
Dont buys: 38961, buys: 38961
VALIDATION Dont buys: 1930, buys: 1930
```

RNN model

We have been working on a repetitive forecast of cryptocurrency price movements of the neural network, focusing mainly on the processing we need to do. In this task, we will end up building our model and "training" it.

For this we first need to print some libraries for model building and we will also need to create new constants:

```
import tensorflow neither tf
Frome tensorflow.keras.models import Sequential
Frome tensorflow.keras.layers import Dense, Dropout, LSTM, CuDNNLSTM,
BatchNormalization
Frome tensorflow.keras.callbacks import TensorBoard
Frome tensorflow.keras.callbacks import ModelCheckpoint
import my
EPOCHS = 10 #data level
BATCH_SIZE = 64
```

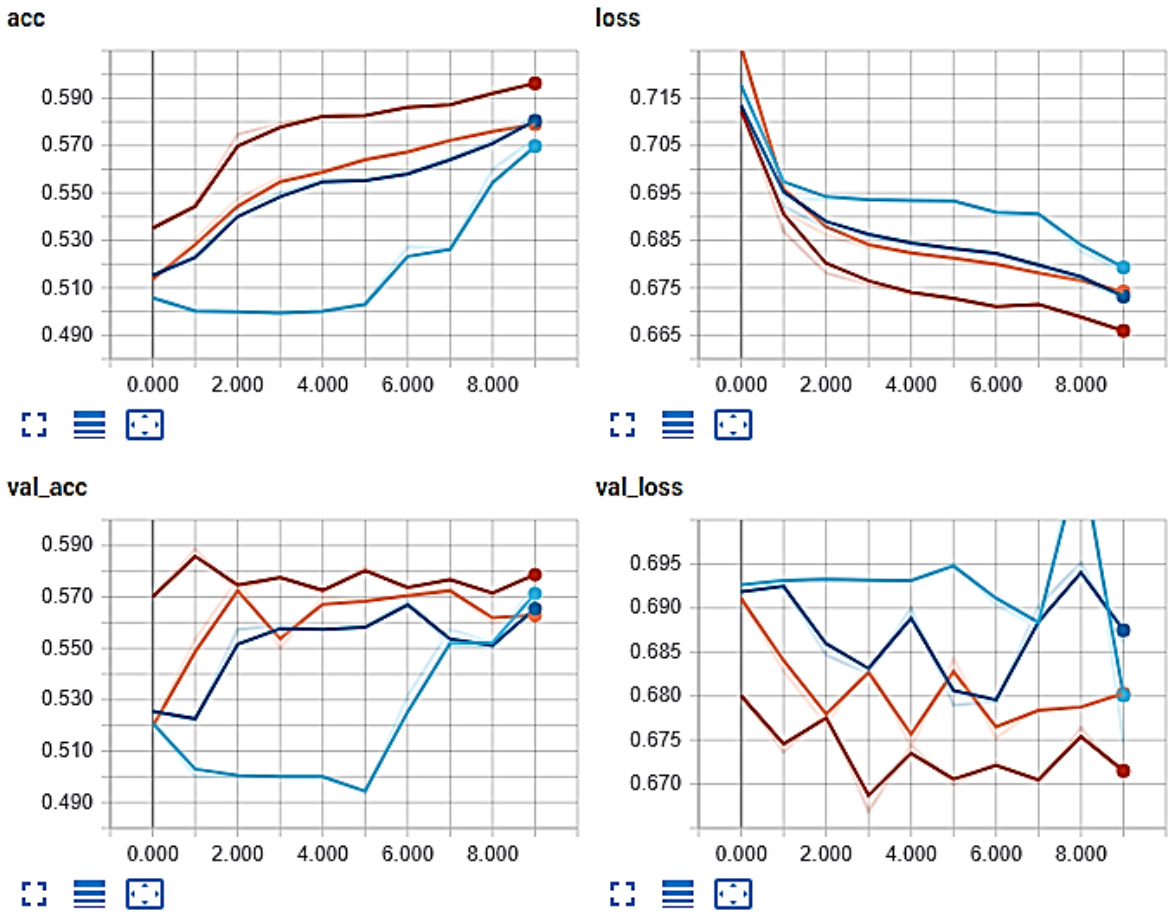
```
NAME = f"{SEQ_LEN}-SEQ-{FUTURE_PERIOD_PREDICT}-PRED {int (time.time
())} "
#name of the model that should be unique
```

Now for the model, we tried a few things like 2 vs 3 layers, 64 vs 128 knots:

```
pattern = Sequential ()
pattern.add (CuDNNLSTM (128, input_shape=(train_x.shape [1:]),
return_sequences=True))
pattern.add (Dropout (0.2))
pattern.add (BatchNormalization ())
pattern.add (CuDNNLSTM (128, return_sequences=True))
pattern.add (Dropout (0.1))
pattern.add (BatchNormalization ())
pattern.add (CuDNNLSTM (128))
pattern.add (Dropout (0.2))
pattern.add (BatchNormalization ())
pattern.add (Dense (32, activation='relu'))
pattern.add (Dropout (0.2))
pattern.add (Dense (2, activation='softmax'))

opt = tf.keras.optimizers.Adam (lr= 0.001, decay= 1e-6)

#models
pattern.compile (
 loss='sparse_categorical_crossentropy',
 optimizer=opt,
 metrics=['accuracy']
)

tensorboard= TensorBoard (log_dir="logs /}".format (NAME))

filepath= "RNN_Final-{epoch: 02d}-{val_acc: .3f}"
checkpoint = ModelCheckpoint ("models /}.model ".format (filepath,
monitor='val_acc', verbose= 1, save_best_only=True, mode='max'))

# Train model
history = pattern.fit (
 train_x, train_y,
 batch_size=BATCH_SIZE,
 epochs=EPOCHS,
 validation_data=(validation_x, validation_y),
 callbacks=[tensorboard, checkpoint],
)
```

We run the project:

```
NAME = f"AT RATIO_TO_PREDICT}-{SEQ_LEN}-SEQ-{FUTURE_PERIOD_PREDICT}-
PRED- {int (time.time ())} "
```



We see that: validation accuracy increases over time, validation loss decreases.

Conclusions

Through this project, to solve the problem using Machine Learing, we first tried to categorize the problem and tried to find previous solutions to how they solved it. We managed to understand the case and what we had to do, as the problem involves prices which are changing over time, this can be modeled as a time series forecasting problem. In parallel, we also tried to solve the problem as a normal machine learning problem with the features that are the previous prices and the output is the predicted price for that day. We explained what models we used and how we configured them to predict cryptocurrency prices.

This study focuses on the price of cryptocurrencies and current market trends for
Develop a predictive model. The forecast is limited to previous data. The ability to predict data transmission would improve model performance and predictability. The model developed using LSTM is one of the more accurate models than traditional models that demonstrate a deep machine learning model.

References:

[1] Jonathan Rebane & Isak Karlsson & Stojan Denic & Panagiotis Papapetrou, Seq2Seq RNNs and ARIMA models for Cryptocurrency Prediction: A Comparative Study.

[2] Brandon Ly & Divendra Timaul & Aleksandr Lukanan, Applying Deep Learning to Better Predict Cryptocurrency Trends.

[3] Leopoldo Catania & Stefano Grassi & Francesco Ravazzolo, Predicting the Volatility of Cryptocurrency Time – Series.

[4] pythonprogramming.net/using-trained-model-deep-learning-python-tensorflow-keras/

[5] /https://www.udemy.com/course/deep-learning-recurrent-neural-networks-in-python-l/

[6] https://www.investopedia.com/terms/d/deeplearning.asp#:~:text=Deep%20learning%20is%20a%20subset,learning%20or%20deep%20neural%20network.

Mexhit Kurti
Faculty of Natural Sciences
Master of Science in Informatics
Artificial Intelligence
2021