Forex price prediction using LSTM's

DA SILVA MOREIRA DYLAN, DIAS TIAGO

Supervisor: George Kerchev

FSTC: Bachelor of Mathematics

Summer semester 2021

Contents

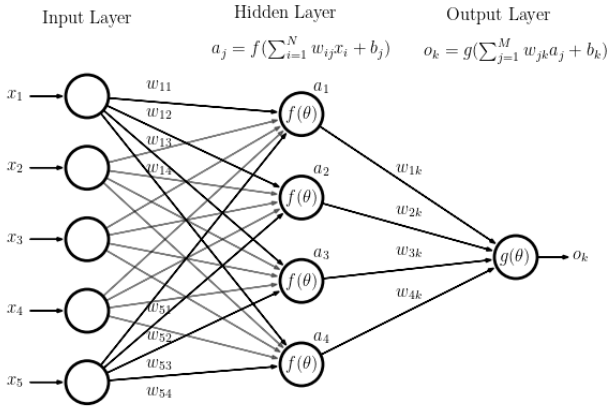
Abstract

The goal of this project is to to use machine learning, more precisely a LSTM neural network to try predicting the Forex market. For this project we will be focusing on the EUR/USD course and use a 60 day time series to predict the 61st day.

1 Introduction

Have you ever wondered how a computer can reproduce the text on a handwritten document into computer written language? Or while driving, how voice-to-text recognizes speech and turns it into text? This is all possible thanks to recurrent neural network's (RNN), which is a type of artificial neural network. Like the names implies,a neural network is a circuit of neurons. Every neuron in our brain is connected through path-wise so that the information can get to its destination. Scientists took advantage of that concept and translated into numerical language, making computers so to speak learn on their own, that's what we refer to artificial intelligence. The corresponding networks are called Artificial Neural Network (ANN). An artificial neural network consists of 3 main layers:

- Input layer
- Hidden layer
- Output layer

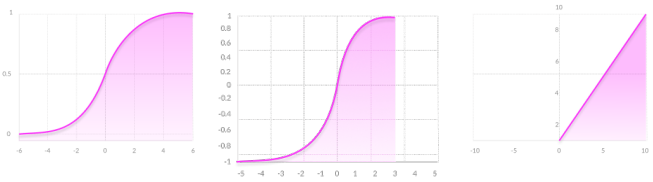


The connections between every layer are called edges. Every edge has a certain weight w, which shows the importance of the edge for every single information. After the information passes through all of the layers, it reaches to the output layer which displays the result. This process is used on a variety of tasks such as speech recognition, stock trading etc. RNN are good at processing sequence data for predictions. What differentiates recurrent neural networks from other artificial networks, is that the information passes through the layers several times. RNN's have a concept of sequential memory, it uses previous information to affect later information. It has a looping mechanism at the hidden state level, where previous information is kept. One issue is the short-term memory, which is the lack of retaining information over an extended period of time. A solution to this can be backpropagation. Backpropagation is an algorithm used to train and optimize a neural network. Therefore, we first go forwards through the network to estimate an error value which shows how badly the network processes. Then with the error value we use backpropagation to calculate the gradients for each node. A gradient is basically a value used to adjust the internal weights so that the

network learns from its mistakes. Gradient descent is used to find the minimum of the function. At each iteration, we calculate the partial derivatives of the loss function with respect to the weights hoping to nudge up or down the weights. Our result should then be closs to zero loss. This is what basic networks tend to achieve to work properly. But the gradient value will always shrink or explode exponentially from layer to layer because a layer calculates its gradient with respect to the effect of the gradients of the layers before. To get rid of the vanishing / exploding gradient problem and the short-term-memory one uses different models such as the Long-Short-Term-Memory model (LSTM) or the Gated Recurrent Unit model (GRU). But how does a RNN work? The input is first transformed into readable vectors which the model processes one by one. During the whole process the hidden state, which holds the information, is passed to the next sequence. The hidden state is the combination between the new input and the previous hidden state which forms a new vector. To calculate a value of a neuron a^k of layer k we need to use the values of all neurons $a^{(k-1)}$ of the $k-1$ layer, using the formula:

$$\sigma(\sum_{i=0}^{n} a_i w_i + b)$$

where σ is the activation function. There are several activation functions which can be used :

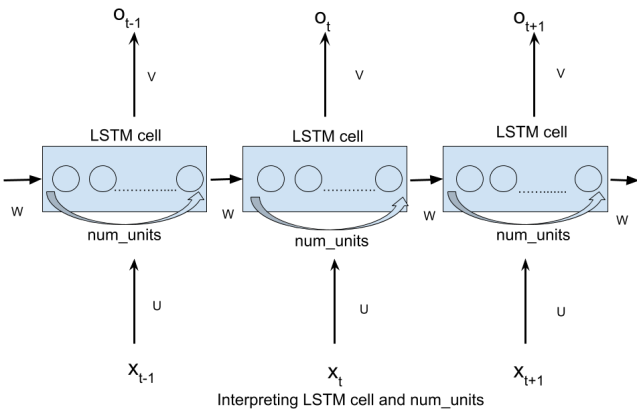


(a) (b) (c)

Figure 1: (a) $\sigma(x) = \frac{1}{1+e^{-x}}$ (b) $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (c) $f(x) = max(0, x)$ (relu)

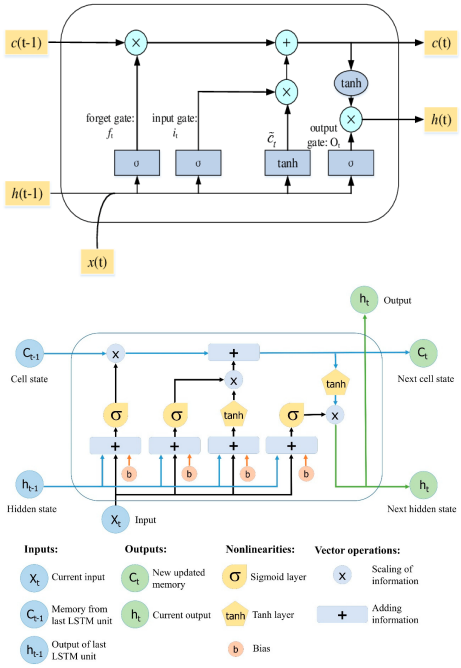
These are some of the most common activation functions used in machine learning.

Since LSTM's are especially useful for time series predictions, we will be using that type of network. Let us have a quick look at the architecture of an LSTM layer:



Interpreting LSTM cell and num_units

As we can see the layer consists of a number of LSTM cells (sometimes referred to as LSTM blocks) this number usually corresponds to the number of x_i inputs. Each of these cells contain a finite number of LSTM units:



The LSTM unit has a rather complicated architecture, but its main components are:

- **Input gate :** manages the cell update state
- **Forget gate :** manages the cell reset state
- **Output gate :** manages the cell state added to the hidden gate

2 Universal approximation theorem

After seeing how neural networks work and their applications, we need to prove their correctness. This can be done through the *Universal approximation theorem*. Proving this theorem validates the universality and justness of neural networks.
Since the proof is technical and tedious, we will focus on a more visual approach of the proof.

Theorem. *Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function (called the activation function. Let I_m denoted the m-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then given any $\epsilon > 0$ and any function $f \in C(I-m)$, there exists an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define :*

$$F(x) = \sum_{i=1}^{N} v_i \varphi(w_i^T x + b_i)$$

as an approximate realization of the function f; that is,

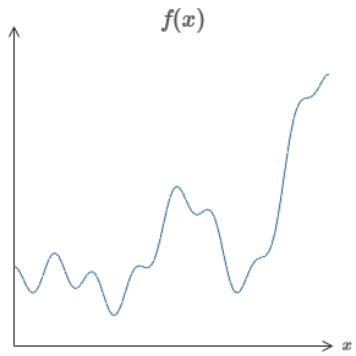
$$|F(x) - f(x)| < \epsilon$$

for all $x \in I_m$. In other words, functions $F(x)$ are dense in $C(I_m)$.

One can see that the formal statement of the theorem is quite complex, let us have a look at a more informal statement of the theorem:

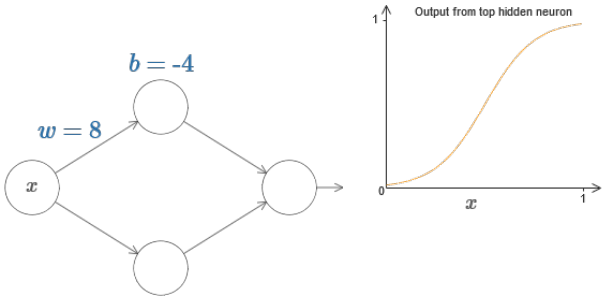
Theorem (*Informal*)**.** *A neural network with one single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$*

Proof. First we need a random function f we want to approximate:

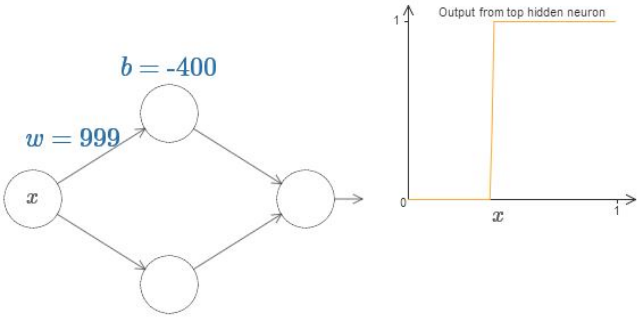


To keep it simple, we start with a neural network with one hidden layer containing two neurons, and one input neuron as well as one output neuron.

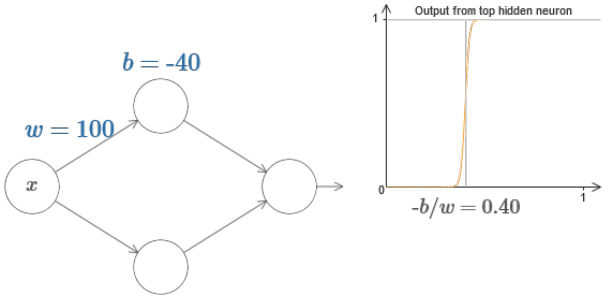
As the activation function we use a sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ (although, one could have used any other activation function). The output from the hidden neuron will therefore be: $\sigma(wx + b)$.



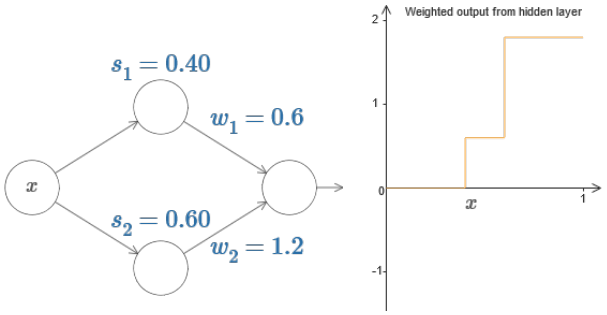
By taking a large enough weight w one can see that the the sigma function gets approximated to a step function.



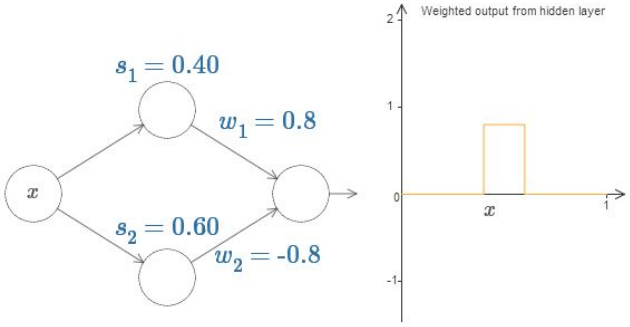
The position s of the step function is given by $\frac{-b}{w} = s$.



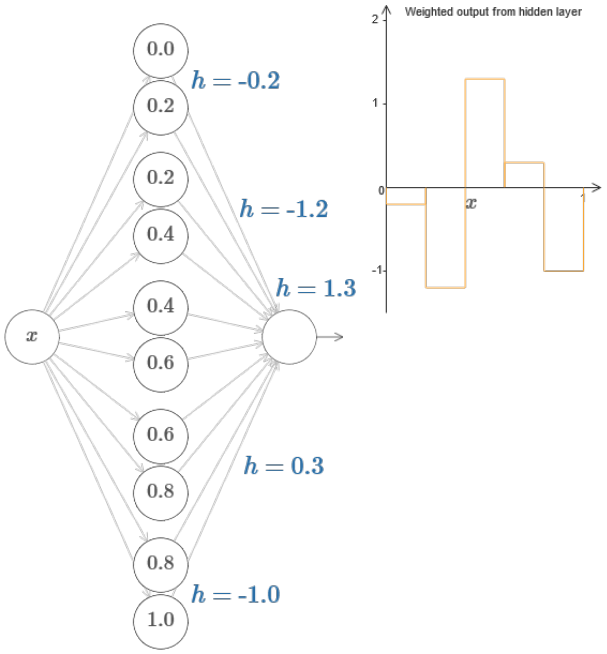
Doing the same for the second hidden neuron one gets the weighted output of the hidden layer equal to $\sum_j w_j a_j$:



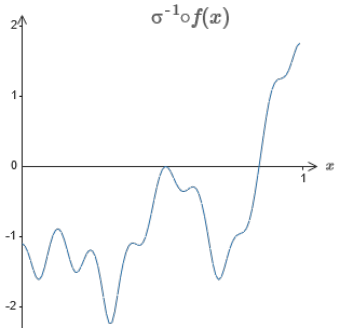
If one takes the same value for w_1 and w_2 such that $w_1 = -w_2$ we observe a 'bump' function with height $h = w_1$ and $h = -w_2$. Furthermore the 'starting point' and 'end point' of the function will depend on s_1 respectively s_2.



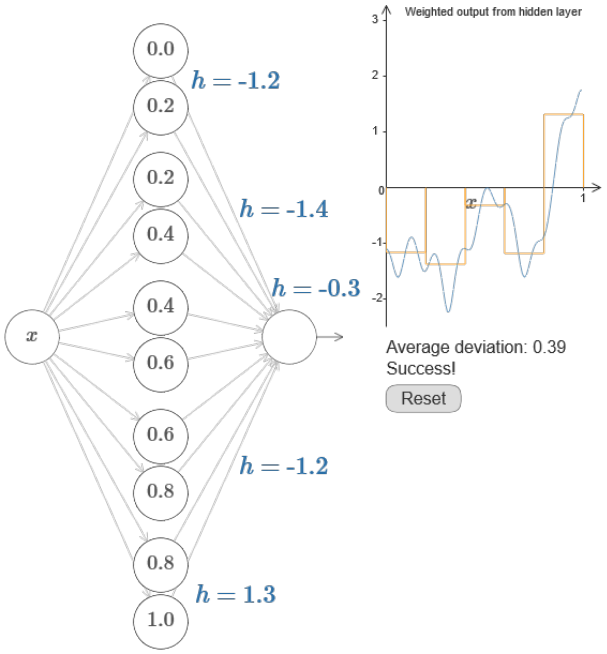
By adding neurons pairwise to the hidden layer, one creates $\frac{n}{2} = N$ of these 'bump' functions where n equals to the number of neurons in the hidden layer. In addition, by choosing s_j accordingly one can cover the x-axis with N equally sized (width) 'bump' functions. The larger the number of neurons N is, the finer the coverage will be.



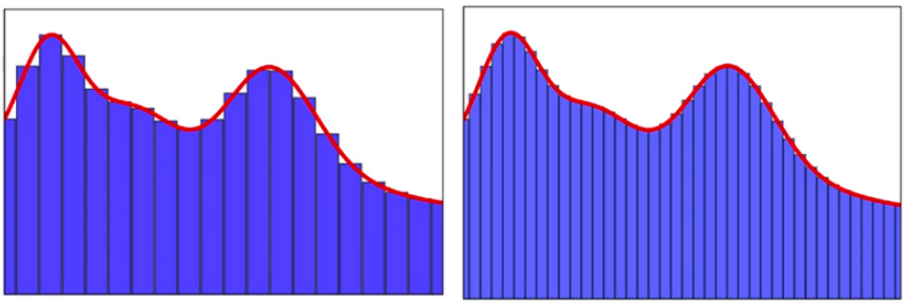
Until now we have been focusing on the weighted output of the hidden layer ($\sum_j w_j a_j$), however the output of the output layer is under the form $\sigma(\sum_j w_j a_j + b)$ where b is the bias of the output neuron. To solve this problem we approximate $\sigma^{-1} \circ f(x)$ instead of $f(x)$ and set the bias of the last neuron to $b = 0$.



Since we are able to approximate it by tweaking the weights and biases as shown in the steps above, we hence can approximate our function f for any given ϵ.



Remark: The larger the number of neurons N in the hidden layer the better will be the approximation.



□

3 Building and Testing of the LSTM

In this section we will focus on building the LSTM, as well as training and testing of the neural network. For the programming we used Python with the Anaconda application and Keras library using Tensorflow backend.

Imported libraries :

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import yfinance as yf
import math
plt.style.use('seaborn-whitegrid')
```

3.1 Importing the Data

We get the Forex price of EUR/USD from Yahoo finance website by using the yfinance.download method:

```
data=yf.download(tickers='EURUSD=X',start='2011-12-31',end='2019-12-31',interval ='1d')
```

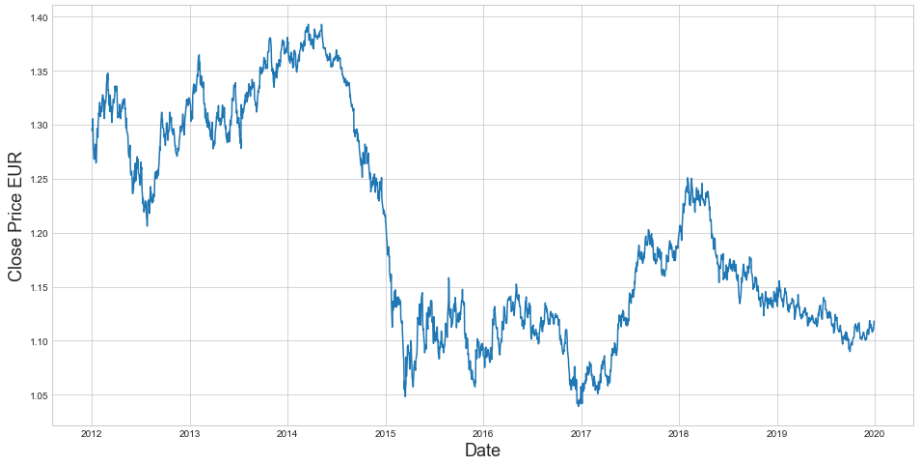


Figure 2: EUR/USD Forex price

Since we are only interested in the closing price, we will filter out the closing price and save its values into an array.

```
data_close = data.filter(['Close'])
dataset = data_close.values
```

3.2 Transformation of the data

We split the data, 80% for training and the other 20% for testing

```
# defining the number of rows we use for thr training set and testing set
train_len = math.ceil( len(dataset) *.8) # 80% of data round up
```

The data needs to be scaled between 0 and 1 because it is easier for the back propagation to handle smaller numbers.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range = (0, 1))

training_data = scaler.fit_transform(dataset[:train_len  , : ]) # importance of fit_transform
```

Now we create x_{train} and y_{train} array, each element of x_{train} is an array with 60 values since that's the time series we are using and y_{train} contains every respective 61^{st} day we want to predict.

```
x_train =[] # independant training variables / training features
y_train =[] # dependant training variables / targat variables

for i in range(60, len(training_data)):
    x_train.append(training_data[i-60:i,0])
    y_train.append(training_data[i,0])
```

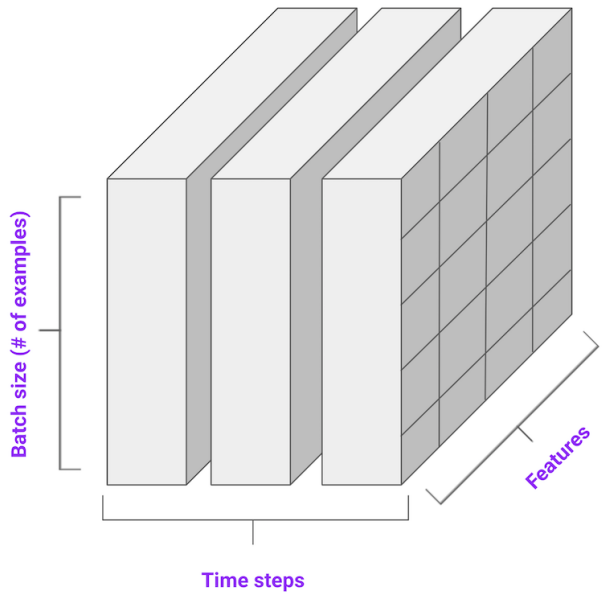


Figure 3: Input format

Since keras neural networks require the input and output to be a 3-dimensional array, we need to convert the x_{train} andy_{train} into a numpy array to then reshape it into a 3-dimensional array (which is a numpy method).

```
x_train,y_train = np.array(x_train), np.array(y_train)
x_train= np.reshape(x_train,( x_train.shape[0] ,x_train.shape[1] ,1))
```

3.3 Building and training the LSTM

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

model = Sequential()
model.add(LSTM(50, return_sequences = True, input_shape=(x_train.shape[1], 1)))
model.add(LSTM( 50, return_sequences = False))
model.add(Dropout(0.2))
model.add(Dense(25))
model.add(Dense(1))


model.compile(optimizer = 'adam', loss = 'mean_squared_error')
model.save('lstm_model.h5')
```

The build network will look like this:

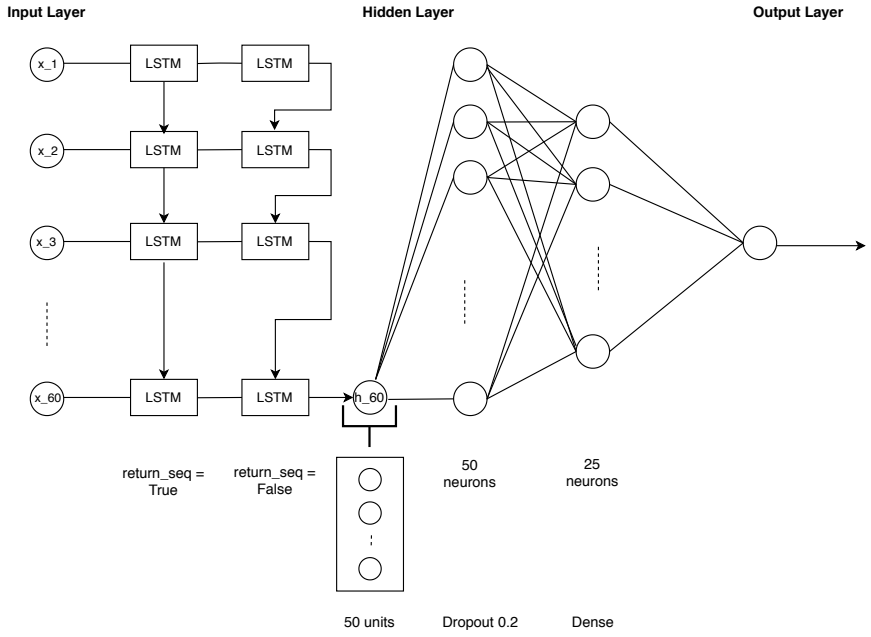


Figure 4: Network architecture

Remark: A dense layer is a layer of simple neurons that connects to the next layer of neurons. A dropout layer is also a dense layer, with the difference that it randomly changes the value of neurons to 0 (to a certain percentage), in our case it's 20% of the neurons. This prevents the network from overfitting.

After building and defining the LSTM's parameters we can now train the network. We will use a batch size of 32, this will update the LSTM every 32^{nd} iteration.

```
#powers of two are  better  for computation
model.fit(x_train, y_train, epochs = 100, batch_size = 32)
```

3.4 Testing and evaluation of the results

Before we can test our neural network, we must first transform the remaining 20% of the data, like we did with the training data to feed through our model.

```
#remaining 20%
test_data = scaler.transform(dataset[train_len - 60: , : ])

x_test = []

for i in range(60,len(test_data)):
    x_test.append(test_data[i-60:i,0])

x_test = np.array(x_test)
x_test = np.reshape(x_test, (x_test.shape[0],x_test.shape[1],1))
x_test.shape
```

Now we can finally make our predictions. It's worth noticing that after getting the predicted values from the network, one has to inverse transform the data since it has been scaled before feeding through the network.

```
predictions = model.predict(x_test)
predictions = scaler.inverse_transform(predictions)# importance of transforme only
predictions.shape
```

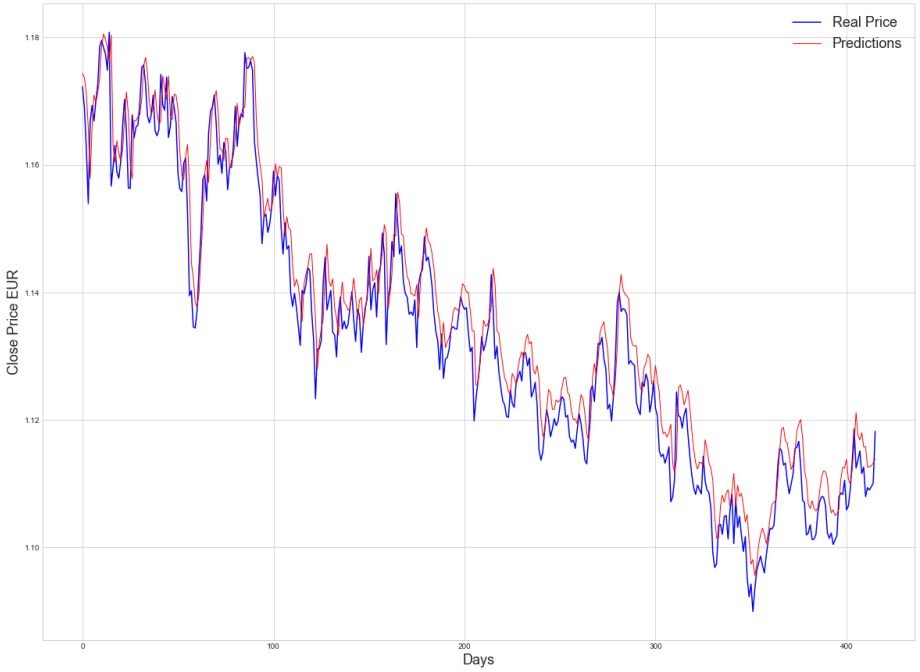


Figure 5: EUR/USD prediction

Overall the network seems to have a pretty good approximation.

Testing over a period of 30 days

To further test our network, we are going to test it on a 30 day prediction period which is not adjacent to the dates we took as our training and test data. Since we want to predict 30 days, we need to take a 90 day period because we trained our model on a 60 time series base.

```
new = yf.download(tickers ='EURUSD=X',start ='2020-01-01',end ='2020-05-05',interval= '1d' )
```

Once again we need to extract the needed values from the data and transform it so it ends up being a valid input for the network.

```
new = new.filter(['Close'])
new_dataset = new.values
new_data = scaler.transform(new_dataset)

test = []
for i in range(60,len(new_data)):
    test.append(new_data[i-60:i,0])

test = np.array(test)
test = np.reshape(test, (test.shape[0],test.shape[1],1))
new_predictions = model.predict(test)
new_predictions = scaler.inverse_transform(new_predictions)
```

Date	Close	prediction
2020-03-24	1.076461	1.072149
2020-03-25	1.080264	1.077587
2020-03-26	1.088957	1.084084
2020-03-27	1.104826	1.092593
2020-03-30	1.113908	1.106560
2020-03-31	1.103047	1.117704
2020-04-01	1.102657	1.112652
2020-04-02	1.095362	1.108130
2020-04-03	1.084740	1.101286
2020-04-06	1.080696	1.091389
2020-04-07	1.080380	1.085273
2020-04-08	1.089514	1.083987
2020-04-09	1.086366	1.091049
2020-04-10	1.092741	1.092449
2020-04-13	1.093267	1.096676
2020-04-14	1.092299	1.098599
2020-04-15	1.098539	1.098048
2020-04-16	1.090510	1.101768
2020-04-17	1.085847	1.097720
2020-04-20	1.086697	1.092078
2020-04-21	1.086484	1.090661
2020-04-22	1.085647	1.090741
2020-04-23	1.080964	1.090546
2020-04-24	1.077702	1.087266
2020-04-27	1.082368	1.083565
2020-04-28	1.082485	1.085632
2020-04-29	1.083658	1.087224
2020-04-30	1.087725	1.088548
2020-05-01	1.094547	1.091746
2020-05-04	1.095963	1.097728

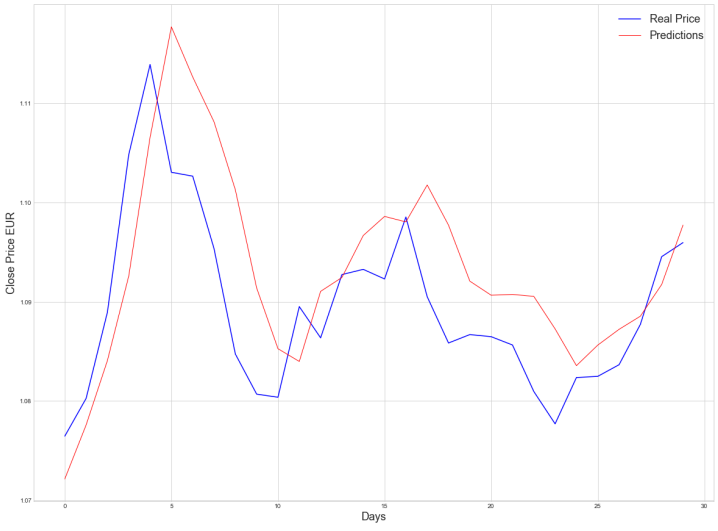
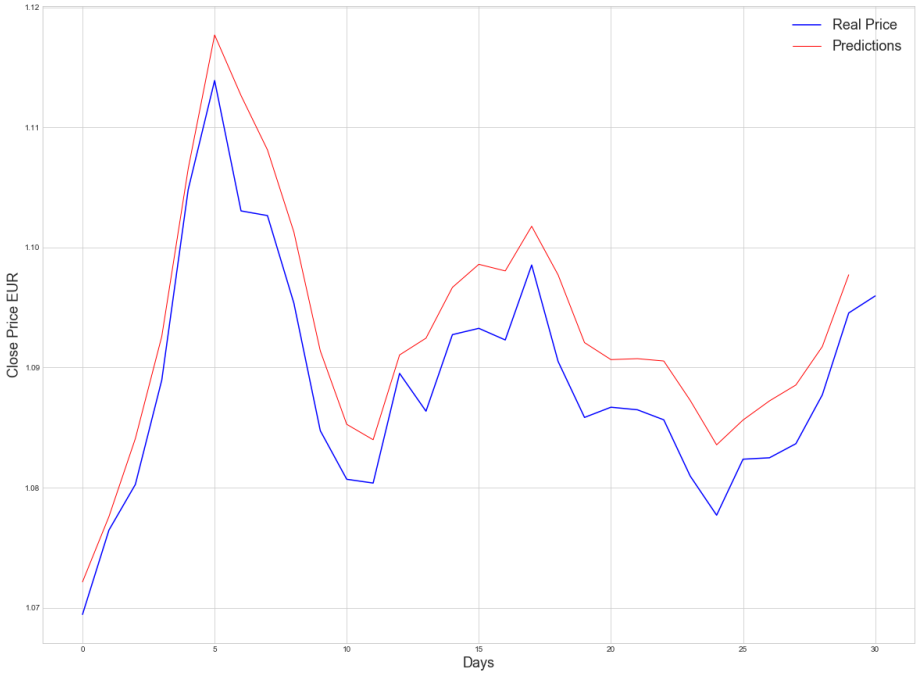


Figure 6: Predictions

When looking at the resulted predictions, the predicted value seems very close to the actual price from the previous day. It seems that our LSTM network just takes the closest value to the previous day closing price. Let's shift the actual price forward by one day to test this hypothesis:



This confirms our hypotheses.

Gain or Loss

Let us use the predictions and make a simple simulation on the market. We will put a 100 € buy or sell order depending on the prediction of our network, after 29 days we see if we lost or won money. The following code will do exactly this.

```
# The i+1 predcition determines if we buy or sell on the i day.
def buy_or_sell(close,pred,i):
    if pred[i+1] > close[i]:
        return True # Buy
    else:
        return False # Sell

# Will determine the loss or gain based on the difference of the closing prices.
def gain_loss(close,pred):
    total = []
    differ = []
    capital = 0
    for i in range(0,29): # forcasting 29 days
        diff = 100*(close[i+1,0]-close[i,0])
        if buy_or_sell(close,pred,i) == False:
            diff = -diff
        capital = capital + diff
        total.append(capital)
```

```
        differ.append(diff)

    print(sum(differ), capital)

# Plots the daily loss or gain and the total win or gain over time
    plt.figure(figsize=(20,15))
    plt.plot(differ,color = 'red')
    plt.plot(total, color = 'green')
    plt.legend([' daily gain or loss' , 'total gain or loss'],fontsize=18)
    plt.ylabel('Money in EUR',fontsize=18)
    plt.xlabel('Days',fontsize=18)
    plt.show
```

After 29 days we made $\approx +1.95$€

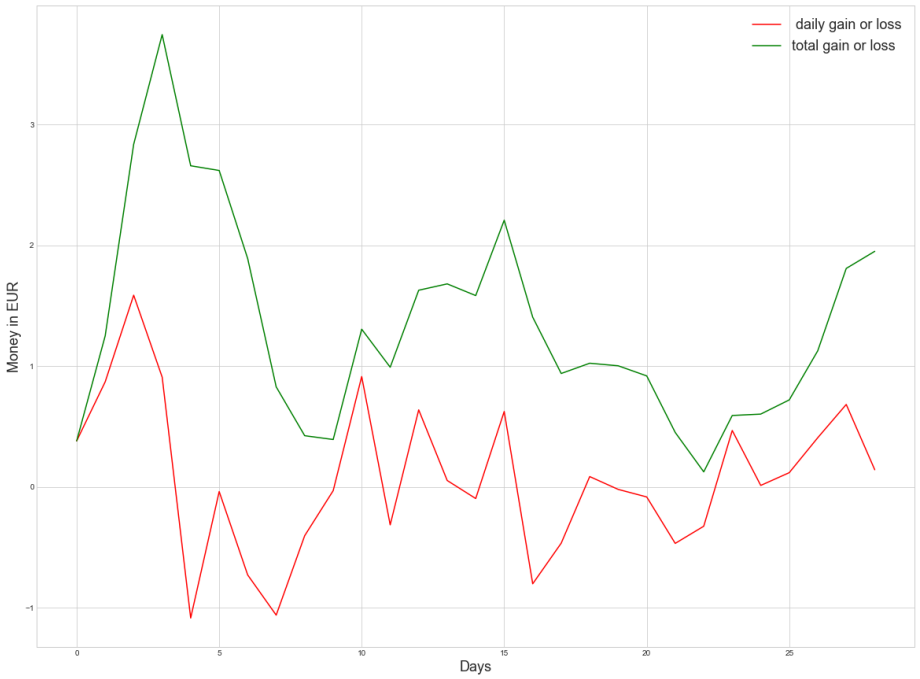


Figure 7: Gain

It's worth mentioning that every time we retrain the network on the same data we get slightly better or worse predictions, which will ultimately result in different outcomes of the simulation:

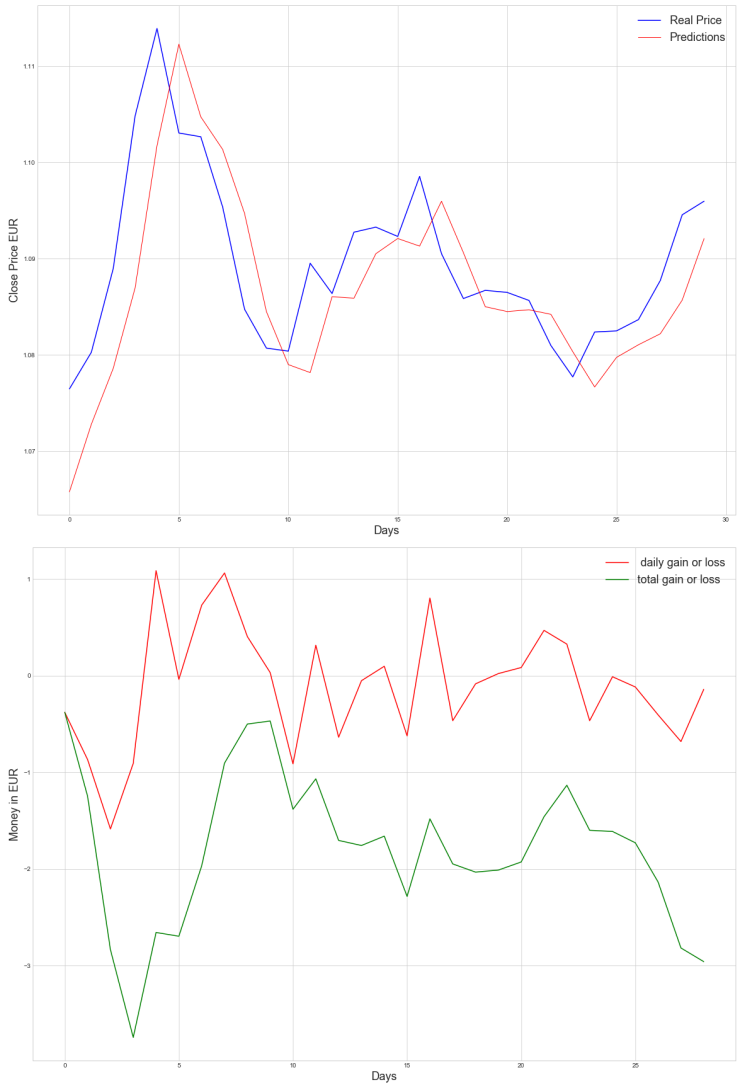


Figure 8: Loss

4 Conclusion

Even if machine learning is very useful in different areas, we weren't able to successfully predict the Forex market, at least while using an LSTM model. The best our model could do is taking a relatively close value of the current day's price and base its prediction on it. This is due to the lack of predictive ability of our model and because we didn't take social and political factors into consideration.