POLITECNICO
MILANO 1863

BITCOIN PRICE PREDICTION USING RECURSIVE NEURAL NETWORKS

Numerical Analysis for Machine Learning

Matteo Citterio **(10620055)**
Federico Caspani **(10622658)**

INDEX

INTRODUCTION OF THE PROBLEM 3

ALL ASPECTS ABOUT THE DATASET 4

DESIGN CHOICES 6

- HOW THE NETWORK WAS BUILT 6
  - Recursive Neural Network (RNN) 6
  - LSTM Neurons 7
- LOSS FUNCTION 8
- ACTIVATION FUNCTIONS 9
- ADAM AS OPTIMIZATION ALGORITHM 11
- NETWORK'S HYPERPARAMETERS 11

ANALYSIS OF THE RESULTS 12

- PREDICTION PLOT 12
- TREND OF LOSS FUNCTION'S VALUES 13
- HOW HYPERPARAMETERS CAN AFFECT THE GOODNESS OF THE RESULTS? 15

EXTRA FEATURES 17

GitHub repository of the project: https://github.com/citteriomatteo/bitcoin_prediction_with_neural_networks

INTRODUCTION OF THE PROBLEM

Cryptocurrencies are one of the most promising (and discussed) economical concepts of the last decade.

Their importance and consequent explosion were due to the fact that the technology they're based on (*Blockchain*) allows people to have a fast and direct contact with the currency, that is completely independent from every kind of middle infrastructure, such as banks, offices, etc.

If we go another step deep, what is really interesting about this topic is that the value of the money can change in a matter of milliseconds, and the volatility only depends on how and when people decide to interact with the coins.

The main open point in the context of Machine Learning applied to modern Economics is exactly linked to the instability of the values: **high volatility of the price results in a very jagged curve, and a model that makes affordable long-term predictions of these kinds of curves is really difficult to build.**

Our work has been based on the construction of a *forecast model* which, following appropriate training, is able to provide forecasts for a subsequent time band or, at least, a good generalization of the "next curve".

ALL ASPECTS ABOUT THE DATASET

The dataset we have decided to use is available for free at the following link.
A quick overview of the dataset and its fields, obtained through *data.head()*, is the following:

	Date	Open	High	Low	Close	Volume	Market Cap
0	Jul 31, 2017	2763.24	2889.62	2720.61	2875.34	860575000	45535800000
1	Jul 30, 2017	2724.39	2758.53	2644.85	2757.18	705943000	44890700000
2	Jul 29, 2017	2807.02	2808.76	2692.80	2726.45	803746000	46246700000
3	Jul 28, 2017	2679.73	2897.45	2679.73	2809.01	1380100000	44144400000
4	Jul 27, 2017	2538.71	2693.32	2529.34	2671.78	789104000	41816500000

The transformation process that the dataset underwent mainly consists in many steps:

1. ***data - dates splitting***
   The dataset is split into two different structures: in the original one, we contain the whole table, except for the Date feature, which is cloned onto a separated array.

2. ***data cleaning***
   The raw dataset presented some format issues that we fixed with simple cleaning procedures. The result is the dataset as it is shown above, in which all data is uniformed to float representation.

3. ***data flipping***
   We flipped data and dates (separately) upside down in order to have tuples organized in an increasing manner with respect to their date.

4. ***data scaling / normalization***
   The split between data and dates (point 1) is mandatory in order to apply normalization on our data, because the date values are written in a format that, being non-float, is unsupported for normalizing together with all other float features.
   Data normalization consists in bringing each field's value to a normal form (zero mean and unit variance) of the field itself, but also Min Max Scaling transformation can be used in these exact terms:

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

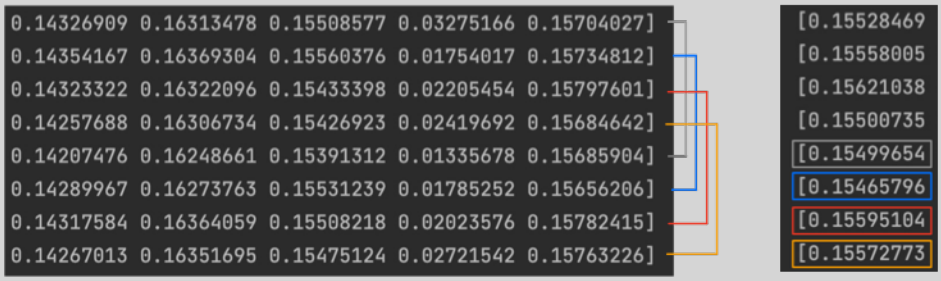
   We applied this scaling procedure, instead of the classic normalization, making use of a Python object contained in the *sklearn.preprocessing* module:

```
scaler = MinMaxScaler()
data = scaler.fit_transform(data)
```

   that automatically stores values of Max and Min used to scale, to let us re-use them to scale back to original after the prediction.

5. ***creation of super-batches***
   The field we wanted to predict is "Open".
   We decided to limit the random component (of choosing elements of mini-batches) to perform choosing among some custom super-batches of 60 elements each.
   In this way, whatever will be the choice of samples per epoch, **we ensure that our model will work with random blocks of time-consequent samples**.

For simplicity, the shape of super-batches is reported below in blocks of 5 elements:



The column on the right is the *Open* field and the table on the left contains all the other features that will be set as input of the neural network.
In the example, super-batches are taken shifting by one tuple each time: tuples 0-5 form a single input block and the value to predict is the 5th element of *Open* array, tuples 1-6 build another block through which the network tries to predict the 6th *Open* value, and so on.

DESIGN CHOICES

Regarding the architectural choices we have decided to implement our project using TensorFlow with Keras in order to improve our knowledge of this important tool.
Keras is an open-source software library that provides a Python interface for artificial neural networks, and it is one of the most used tools in this area.

HOW THE NETWORK WAS BUILT

We have implemented a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) because the combination of the two components is powerful for modeling sequences of data or time series, which is exactly the case of our dataset.

Recursive Neural Network (RNN)

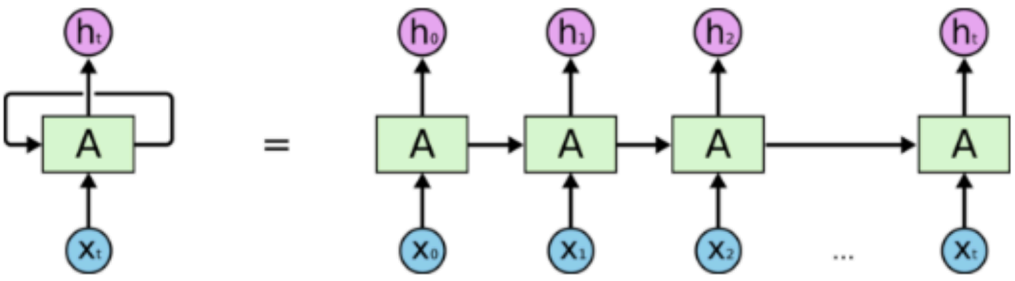
A *Recurrent Neural Network* is a generalization of a feedforward neural network with the difference that RNN has an internal memory.
So, for every input the RNN applies the same function but at every step the network takes the output of the previous step and uses it with the next input as the argument of the activation function.
So, after producing the output, it is copied and sent back into the recurrent network and for making a decision, it considers the current input and the output that it has learned from the previous input.
Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs.

In a mathematical representation:

$$h_t = ReLu(W_{hh}h_{t-1} + W_{xh}\,x_t)$$

Where W is weight, h is the single hidden vector, W_{hh} is the weight at previous hidden state, W_{hx} is the weight at current input state and ReLU is the activation function.
Consequentially the output state becomes:

$$y_t = W_{hy}h_y$$



The main problem with this type of network is the gradient vanishing problem.

LSTM Neurons

To avoid this, we use Long Short-Term Memory (LSTM) neurons.
This type of neurons is built to remember inputs over a long period of time.

The LSTM neurons have a memory in which they can read, write and delete information thanks to 3 different gates:

- **Input Gate:** discovers which value from input should be used to modify the memory. The sigmoid function decides which values to let through 0,1, while the *ReLU* function gives weightage to the valued that are passed.

$$\boldsymbol{i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)}$$
$$\boldsymbol{C_t = ReLU(W_c \bullet [h_{t-1}, x_t] + b_c)}$$

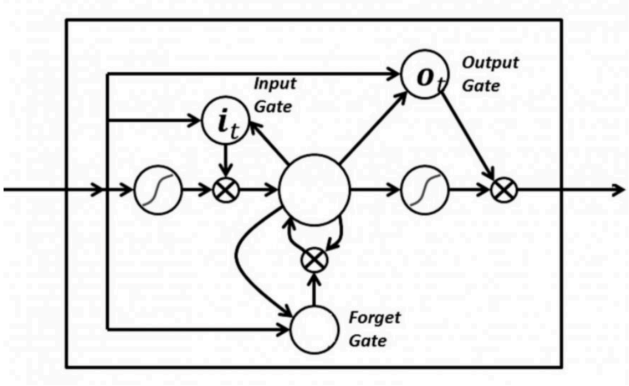
- **Forget Gate:** discovers which details should be discarded from the block thanks to the sigmoid function.
  It looks at the previous state (h_{t-1}) and at the content input (x_t) and, for each number in the cell state C_{t-1}, decide to keep the value (output 1) or to delete it (output 0).

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

- **Output Gate:** uses the input and the memory of the block to decide the output of the neuron.
  The sigmoid function decides which values to let through and the ReLU function gives weightage to the values passed

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

$$h_t = o_t * ReLU(C_t)$$



How LSTM neurons helps with the vanishing gradient descent?

In a standard recursive neural network the grate number of iteration can decrease too much the gradient making the update of the parameters insignificant.
This problem is solved with LSTM neurons thanks to the Forget Gate.
The presence of the forget gate in fact, allows the LSTM to decide, at each time step, which information should be used for the update of the model's parameters and which should be discarded.

This means that, if at a certain iteration k the gradient of the error is too small $\sum_{t=1}^{k} \frac{\partial E_t}{\partial W} \to 0$ in the next iteration the neuron can use the forget gate to choose the best parameter in order to increase the gradient such that $\sum_{t=1}^{k} \frac{\partial E_t}{\partial W} \nrightarrow 0$.

LOSS FUNCTION

The Loss Function we have chosen for this regression problem is a typical ***Mean Squared Error***:

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

What is Mean Squared Error?

MSE is a specific function that, given a real measure and the predicted one, returns the weighted mean of the quadratic errors among the two variables under each dimension.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

The choice came natural in the moment we had a Dense output neuron which directly prompts the predicted value and having a cost function that's easy to compute and differentiate is a necessity (*big inputs, and a neural network that composes of ≃ 300000 hyperparameters*).

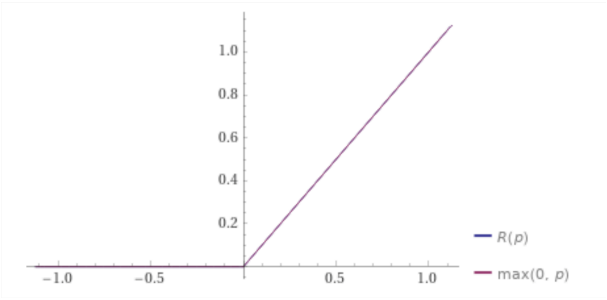
ACTIVATION FUNCTIONS

As activation functions, we decided to use ***Rectified Linear Units (ReLU)*** for all our hidden layers.

What is a ReLU?

A ReLU is a particular non-linear activation function that projects its input to the output if it is positive, otherwise the neuron doesn't even activate.
It presents itself in this form:

$$ReLU(p) = \max(0, p)$$



Why is ReLU "the choice"?

In our case, there are several limitations on the use of other activation functions (such as *tanh*, and *sigmoid*) due to the high number of hidden layers we have implemented.
Some functions are sensitive only in the mid-point of their curve, causing the Vanishing Gradient Problem with very high/very low input values.
If this problem occurs in neurons on the network, the bad adaptation of the network in the case of updating weights and biases would propagate around all the layers until the output and **the network could easily stop being able to learn efficiently**.

By adopting ReLUs as activation functions, we (almost completely) avoid the problem of the Vanishing Gradient, because the gradients remain proportional to the nodes activation.

Other good notes about this *semi-linear* function is that It is allows all the neurons not to be active at the same time and, since we work with many hidden layers and a lot of neurons implementing it, ReLU is easier to compute, if compared to other functions that need *exponential evaluations*.

MODEL OF THE NEURAL NETWORK

Our neural network is a recurrent neural network with LSTM neurons with a ReLU activation function that uses *Adam* as optimization algorithm.

We also use a ***Dense*** layer as output layer where a dense layer implements the operation

$$output = activation(dot(input, kernel) + bias)$$

The neural network is composed by 5 layers:

- 4 hidden *LSTM* layers with respectively 70, 80, 100 and 140 LSTM neurons. The first layer takes an input of 60x6 containing the data about many aspects of the bitcoin title.
- A simple *Dense* layer without an activation function that simply present the final predicted value.
- In order to reduce the overfitting we have introduced some *Dropout* layers that randomly set some input units to 0.

ADAM AS OPTIMIZATION ALGORITHM

After some trials, we found ***Adam*** to be the best algorithm to date.
We have not learnt this method during the course, but It's essentially an efficient mix between ADAGrad and RMSProp, that we studied in detail.

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1)\nabla Q(w_t)$$
$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)(\nabla Q(w_t))^2$$
$$\hat{m} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}$$
$$\hat{v} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}$$
$$w_{t+1} = w_t - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

This optimizer incorporates the benefits of both the algorithms it comes from, making use of two non-biased estimators of the first and second moment of the gradient.
First and second moments (*m* and *v*) are the ones that directly depend on the current evaluation of the gradient of the loss function (*Q*), and the network's parameters are updated basing on the two estimators, a smoothing parameter *ε* (to avoid zero-division) and a fixed (but possibly adaptive) learning rate.

NETWORK'S HYPERPARAMETERS

With respect to the hyperparameters we have chosen to use 20 epochs which guarantee a good tradeoff between results and training time,
while for the Adam's 4 hyperparameters:

1. η: determine the importance of the update element, default 0.001
2. β_1: the exponential decay rate for the 1st moment estimates, default 0.9
3. β_2: the exponential decay rate for the 2nd moment estimates, default 0.999
4. *ε*: avoid the 0 division, default 1e-08

After some attempts, we have chosen to maintain the default hyperparameters because they guarantee the best result for our model.

ANALYSIS OF THE RESULTS

In this chapter we analyze the results we obtained with all the premises that we've explained in the previous sections.

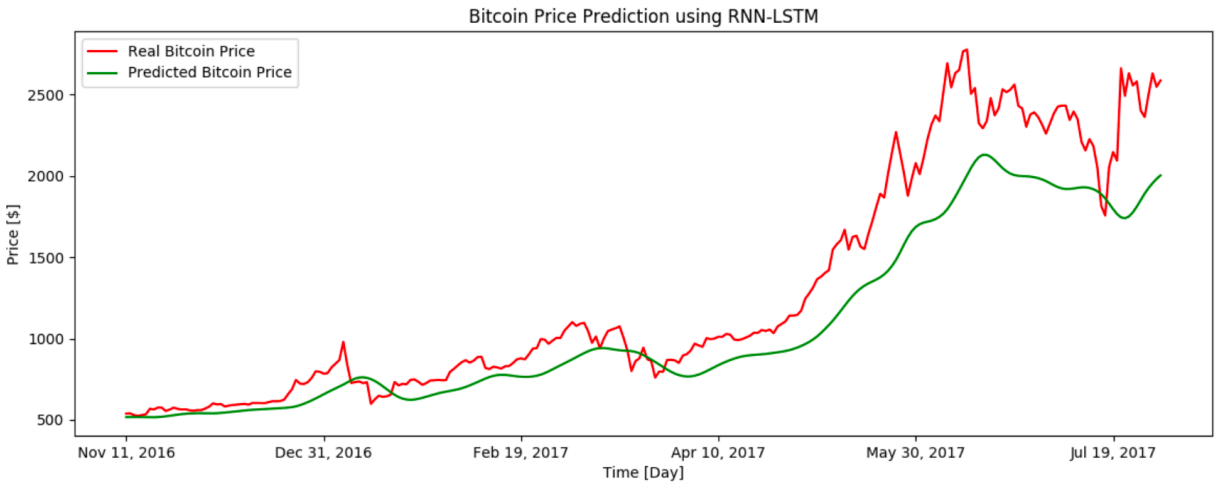
PREDICTION PLOT

After the training procedure, we proceeded to test the trained parameters to predict the values of *Open* field, calling the TensorFlow built-in function:

```
Y_pred = model.predict(X_test)
```

This model, basically, puts elements that compose *X_test* as inputs of the ANN and collects in *Y_pred* the predictions given as outputs from the last Dense neuron.

The result we obtained is represented in the plot below:



As we can see, the model is able to provide, after several training iterations, a good generalization of the real-case price's curve.

We decided to find an objective estimate of the accuracy of our prediction using the following formula:

$$1 - \sum_{1}^{n} \frac{|Y_i - Ypred_i|}{Y_i}$$

and ***the obtained result is an accuracy of*** ≃ 88.07%.

A more precise study on the quality of the prediction will be done in the next paragraph, but we can already say that the model generalizes well since it follows the trend of the test data with a tolerable deviation that allows the predicted curve not to present any spike, synonym of overtraining.

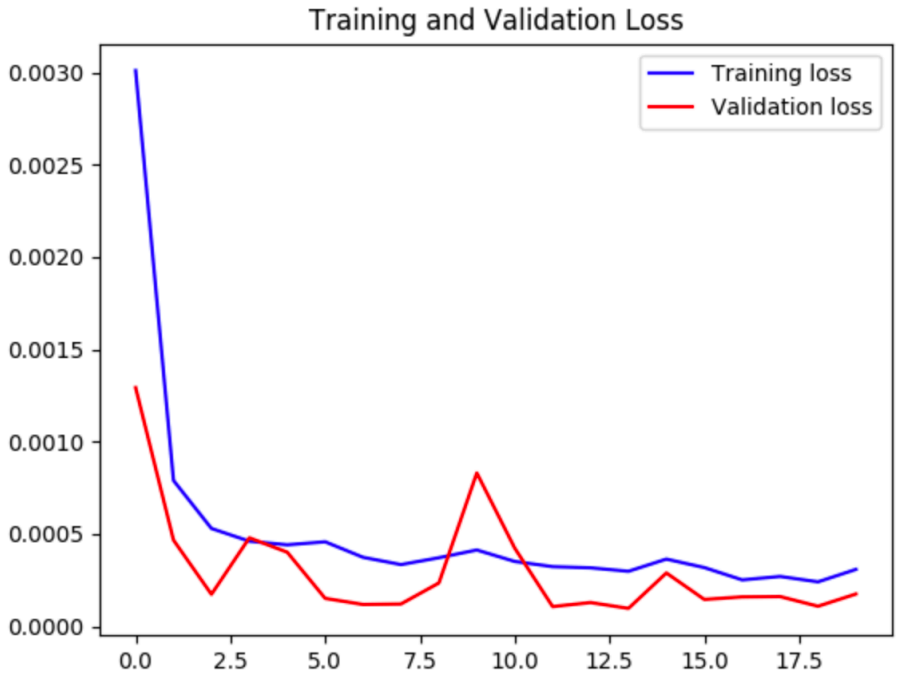
TREND OF LOSS FUNCTION'S VALUES

Before talking about the loss function's results over epochs, we need to introduce two concepts:

- **Training Loss**
  With *Training Loss* we mean the set of errors the model has made trying to predict the training set of data.

- **Validation Loss**
  When we talk about *Validation Loss*, instead, we refer to the set of errors our model makes once its parameters have been updated and we try it on a validation set of data.

Both the losses, related to the prediction of the graph above, are the following:



Now that all the instruments to understand the curves have been clarified, the evolution of the quality of our network's response can be put under reflectors in order to highlight some other important Machine Learning concepts and/or mistakes to avoid.

What we can understand looking at the losses?

How training and validation losses relate to each other can definitively determine the actual status of the learning process:

- ***UNDERFITTING***

  If we are in the case of *Underfitting*, the model is still too simple and generalized.
  We can avoid this problem applying more training to our network or adding complexity in the architecture, if there's the possibility that the way a neural network has been implemented is not enough reticulated to find a good approximation of a goal curve.

  We can identify this issue when **Training Loss >> Validation Loss**.

- ***OVERFITTING***

  When in *Overfitting*, the model acts perfectly in simulating already seen behaviors of the curve, but loses the ability to predict affordable values in front of new data.
  This problem usually happens when the model is composed of an excessive amount of neurons, becoming much more complex with respect to what the problem requests.
  This results in a Training Loss curve drop (the network learns better and better), while the Validation Loss function keeps increasing over the epochs (it loses its forecasting ability).

  Definitely, we identify this problem especially when **Training Loss << Validation Loss**.

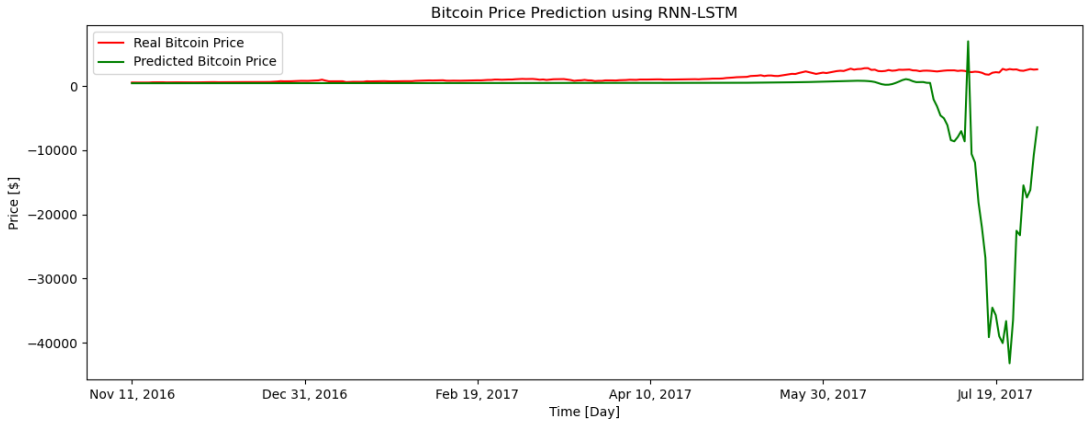
By putting a threshold *th=0.0007* on their difference of around, the program recognizes an *Underfitting* window between epochs 0 and ≃0.7, meanwhile no *Overfitting* situation is identified at all (except for a fast spike between epochs 8 and 10 that doesn't satisfy the imposed threshold).

HOW HYPERPARAMETERS CAN AFFECT THE GOODNESS OF THE RESULTS?

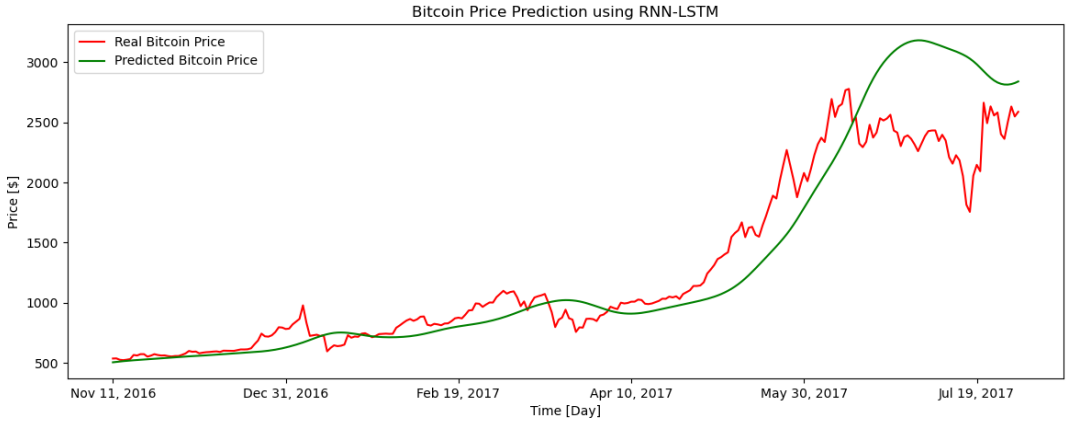
We have analyzed the effects of two of the most important parameters: the learning rate and the number of epochs.

Effects of Learning Rate

The increment of the learning rate (0.01) makes the network too sensible to change in the data and in fact the prediction is too fickle.

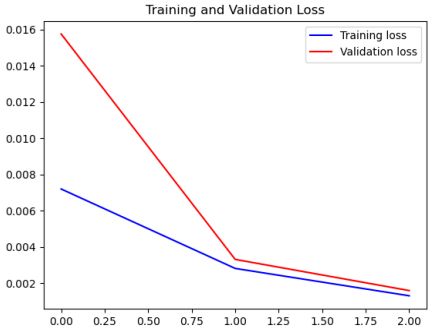


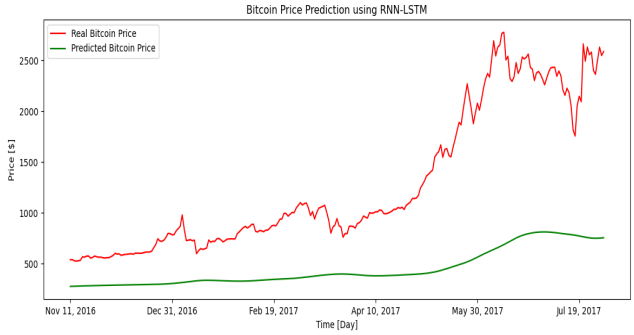
On the other hand, decreasing the learning rate (0.0001) makes the network too slow in following changes and the prediction is not enough reactive and adaptable.



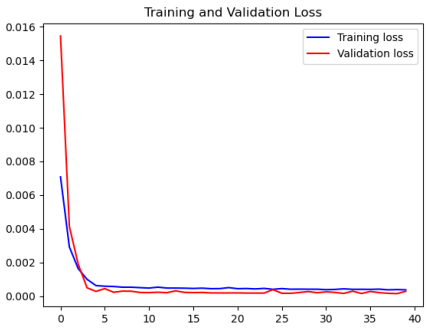
Effect of number of Epochs

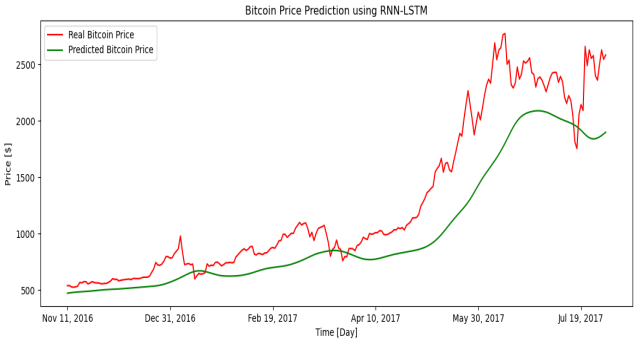
The number of epochs must balance the need of reaching a convergent solution and the necessity of reducing the time for the training of the network.
We have seen that 20 could be a good tradeoff in our problem.
In fact if we reduce too much the number of epoch the network will remain in a situation of overfitting and will not reach a stable situation in which the parameters allow a good prediction:





While if we increase too much the number of epochs the prediction of the network is good, but the same result could be reached with a smaller amount of training time:





EXTRA FEATURES

To increase the fluency of the program, we decided to move all the public configuration variables in a dedicated Python file, **management.py**:

```
"""
This file contains all the configuration stuff needed to manage
all the features of the neural network from outside.
"""

DATASET_PATH = 'deliveries/dataset/bitcoin_price_Training - Training.csv'

# True => Gets the dataset, re-formats it and overwrites on the non formatted file path
# False => Reads the file in path and directly uses it
DATASET_TO_FORMAT = False

# True => Training process starts from the beginning
# False => the previously saved knowledge gets restored
TRAINING = True

# Percentage ([0,1]) of the training-test splitting range
SPLIT_PERCENTAGE = 0.8

# Visualization parameters
PRINT_ARCHITECTURE = True
PRINT_LOSSES = True
PRINT_PREDICTION = True
PRINT_FINAL_ACCURACY = True
```

Every utilization and visualization aspect is easily customizable interacting with this file, instead of the code itself, and two interesting operations are possible through our program:

- ***Dataset formatting***

  As already described at the beginning, if **DATASET_TO_FORMAT** is set to True, the program cleanses data (rigorously the one downloaded at the link given in the second chapter) and overwrites it before using it.
  This functionality can obviously be extended to more complex data cleansing/munging procedures, that could make the Neural Network a more complete and independent component.

- ***Knowledge saving and recovery***

  When **TRAINING** is set to False, the network will not pass through the process of learning, but it will reload the most recent knowledge.
  When **TRAINING** is set to True, the program will initialize a ModelCheckPoint before starting the model's training and adds a Callback for every epoch. The weights of the network after every Callback creation are written in the *cp.ckpt* file