A Simple Bidirectional LSTM Recurrent Neural Network for Market Time Series Prediction

Giacomo Corrias

June 7, 2018

Abstract

The following is a recurrent bidirectional network model for predicting the trend of the closing values of a market over the next 8 hours. The Germany DAX dataset was used for the computation, composed by timesteps taken every 5 minutes for 20 years. The dataset was preprocessed to create timesteps with two consecutive market closing values and a binary label which indicates the trend of closing curve. The RNN was trained and tested with data generated batch-by-batch through Python generator method. The model was developed on Keras 2 with TensorFlow backend, in Python language v3.1.6 exploiting Pandas and Matplotlib libraries. The code was developed and tested on Nvidia Titan X.

1 Introduction

The general trend towards stock market among the society is that it's highly risky for investment or not suitable for trade. The seasonal variance and steady flow of any index will help both existing and naïve investor to understand and make a decision to invest in the stock market.[BUDA13] Market time series forecasting is one of the biggest complex problems in deep learning due to temporal dimension. This additional dimension is both a constraint and a structure that provides a source of additional information, because a time series is a sequence of observations taken sequentially in time.

The Long Short-Term Memory (**LSTM**) recurrent neural network has the promise of learning long sequences of observations and it's a perfect match for time series forecasting. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. Each of the three gates can be thought as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM. There are connections between these gates and the cell.

To upgrade the efficiency and computational power of the model, a particular types of layers can be used: **Bidirectional** layers. They can be used with LSTM to create a powerful model to achieve the solution. The idea of bi-directionality involves duplicating the first recurrent layer in the network so that there are now two layers side-by-side, then providing the input sequence as an input to the first layer and providing a reversed copy of the input sequence to the second. This allows to get more reliability in capturing representations of new structures inside data.

Keras is a high-level neural networks API, written in Python and capable of running on top of **TensorFlow**, CNTK, or Theano. It was used for this approach since it brings advantages like user friendliness, modularity, easy extensibility and work with Python language.[Bro16]

In particularly, the prediction based in market time series has an higher difficulty due to massive quantity of features. The presented approach fights this problem using only a small amount of features and focusing only on market closing values. The obtained results are good for a simple model and they are obtained only with a small amount of computation.

2 Data Description

This current section provides a description for the dataset which has been used to train and test the model.

The **DAX** (Deutscher Aktienindex (German stock index)) is a blue chip stock market index consisting of the 30 major German companies trading on the Frankfurt Stock Exchange. Prices are taken from the Xetra trading venue. According to Deutsche Börse, the operator of Xetra, DAX measures the performance of the Prime Standard's 30 largest German companies in terms of order book volume and market capitalization. [dax18]

The timesteps were taken every 5 minutes, from 19/12/1996 to 15/12/2017, with a total of 779520 timesteps. The dataset is saved in CSV (Comma Separated Values) file and every timestep is composed by 8 features: *Date*, *Time*, *Open*, *High*, *Low*, *Close*, *Volume*.

For this approach *Open*, *High*, *Low* and *Volume* features were not used. *Date* and *Time* features are used to take values in order to respect time constraint of timesteps. *Close* feature is the core feature of this approach, indicating the closing value of the market for every timestep.
The problem statement is to predict the target trend of closing curve over the next 8 hours. To achieve this, the data's going as far back as 672 timesteps ($\bar{1}$ week) and sampled every 96 timesteps ($\bar{8}$ hours). The dataset has been splitted into **training**, **validation** and **test** set. This splitting was carried out respecting time constraint of the timesteps.

The following plot represents the curvature of market closing values of whole dataset.

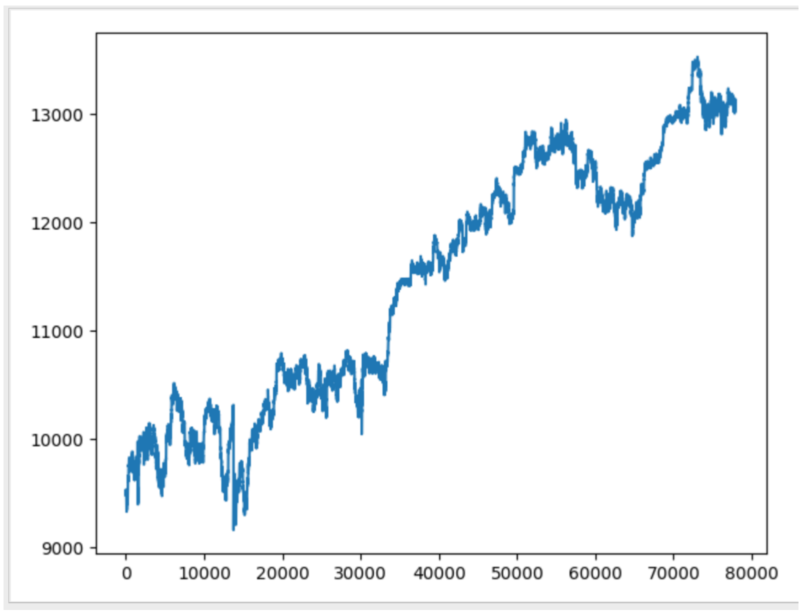


Figure 1: Curvature of Market Closing Values

3 Development Steps

This section provides a deep description of the model development steps. Code is available here GitHub

3.1 Preprocessing

The preprocessing step includes data loading and normalization.

Data loading step was achieved using Pandas library through *read_csv()* method. This method allows to create Pandas Dataframe from file *DAX.csv* which contains whole dataset. Pandas Dataframe is the pandas primary data structure, it's a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). The parameters passed to this method indicates which columns must be used (*Date* and *Close*), the location of *Date* column and index. Once loaded, the NumPy array is extracted from the Dataframe and the integer values are converted to floating point values, which are more suitable for modeling with a neural network.

LSTMs are sensitive to the scale of the input data, specifically when the *sigmoid* (default) or *tanh* activation functions are used. It's a good practice to rescale the data columns to specific min-max range, also called **normalizing**. After the extraction of NumPy array, the normalization step was easily achieved by computing standard deviation and mean from each column of the dataset. Normalization for each value was obtained by subtracting and dividing it with the corresponding column mean and standard deviation value.

3.2 Conversion to Supervised Learning Approach

The RNN model in Keras assumes that the data was divided in two components: input(X) and output(Y), where Y is the predicted value from the X values.

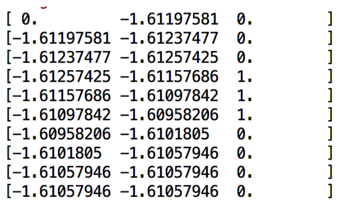
After the normalization step in our time series problem case, we achieve this using only the close values and applying a shift of one timestep to them. Thereby each timestep is composed by close value from the last timestep (t-1) and close value from current timestep (t) as input. The output value is a label obtained by subtracting closing values for each timestep. The label will be 0 if the result of subtraction is negative (**decreasing curvature**), else will be 1 (**increasing curvature**).

This is obtained by using the *shift()* function in Pandas, that will push all values in a series down by a specified number of places, in particular we require a shift of one place: the pushed-down series will have a new position at the top with no value. A *NaN* (not a number) value will be used in this position. We can then concatenate these two series of closing values together to create a Dataframe in which the output column (*Target*) was obtained by calling *computeClose-Target()* custom function for each timestep. After these operations we have a Dataframe ready for supervised learning.

The code used to accomplish this is shown below

```python
def timeseries_to_supervised(data, lag=1):
    df = DataFrame(data)
    columns = [df.shift(i) for i in range(1, lag + 1)]
    columns.append(df) # Second column
    df = concat(columns, axis=1)
    df.fillna(0, inplace=True)  # fill NaN value
    df.columns = ['Close_prec', 'Close'] # Rename columns
    df['Target'] = df.apply(lambda row:
    compute_close_target(row['Close_prec'], row['Close']), axis=1) # Third colum
    return df
```

The first ten timesteps of supervised learning Dataframe are shown below



```
[ 0.         -1.61197581  0.        ]
[-1.61197581 -1.61237477  0.        ]
[-1.61237477 -1.61257425  0.        ]
[-1.61257425 -1.61157686  1.        ]
[-1.61157686 -1.61097842  1.        ]
[-1.61097842 -1.60958206  1.        ]
[-1.60958206 -1.6101805   0.        ]
[-1.6101805  -1.61057946  0.        ]
[-1.61057946 -1.61057946  0.        ]
[-1.61057946 -1.61057946  0.        ]
```

Figure 2: First ten Dataframe timesteps

3.3 Training, Validation and Test sets

The whole dataset was divided into three different sets: training, validation and test sets. The last two sets are obtained by different experimental tests in which different model parameters are modified and tuned.

To achieve this the last timestep index in which each set ends was retrieved by calculating a percentage of dataframe values and rounding the results.

The best result was obtained with *30%* training set and *40%* validation set. The test set starts with the timestep which is consecutive to the last timestep of validation set; last *30%* of data is used for test set.

3.4 Generators

After obtaining the final timestep indices for each set, we need to create a batch of data from these sets to train, validate and test the model.

There are two types of models available in Keras: the **Sequential** model and the **Model** class (used with functional API). Specifically we use the Sequential model exploiting the *fit_generator()* method, which trains the model on data generated batch-by-batch by a Python generator (or an instance of Sequence). The generator is run in parallel with the model for efficiency.

Python generators can be implemented in a clear and concise way as compared to their iterator class counterpart. A generator implementation of sequence is memory friendly and is preferred since it only produces one item at a time (e.g pipelining a series of operations)[Cho]

The used data generator yields a tuple (*samples*, *targets*), where samples is one batch of input data and targets is the corresponding array of target label values. It takes the following arguments:

- **data**: The supervised values NumPy data.
- **lookback**: How many timesteps the input data should go back.
- **delay**: How many timesteps in the future the target should go.
- **min_index** and **max_index**: Indices in the data array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another for testing.
- **shuffle**: Whether to shuffle the samples or draw them in chronological order.
- **batch_size**: Number of samples per batch.
- **step**: The period, in timesteps, at which the data was sampled

The abstract generator function is now used to instantiate three generators: one for training, one for validation and one for testing. Each will look at different temporal segments of the original data previously defined. [Cho17]

4 Model And Experimental Results

This section provides a description of the model which is necessary to summarize the various implementation choices and to understand the optimized one.

All models are created using Sequential Keras API. Initially, the model was composed by only two layers. The first layer was a LSTM layer with *64* neurons and a little bit of dropout with *tanh* activation function and recurrent activation *hard sigmoid* function. The second layer was a Dense layer with *1* neuron and a *linear* activation function. The model was fitted with *adam* optimizer.

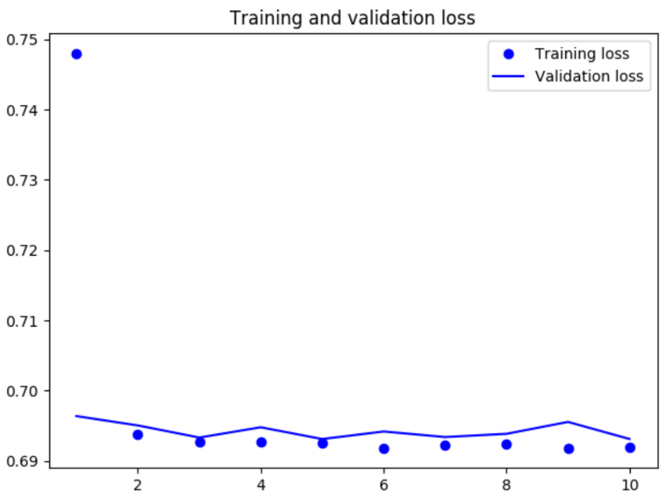


Figure 3: First model loss

The results are *51.2%* accuracy on test set and *71.9%* accuracy on training set. The loss of the model is shown in the image below, obtained with Matplotlib.

To upgrade performances, another LSTM layer with more dropout has been added to deal with overfitting of the first model, but the accuracy of test set is increased only by *0.5%*.

To change this trend of results, the last model adopts Bidirectional wrapper for RNNs. The choice for activation and recurrent activation has been left to Keras (default is *sigmoid* function); dropout and recurrent dropout it's *0.2* for each layer. The Dense layer is equal to the previous dense layer. The model was compiled with *adam* optimizer and *binary crossentropy* loss function. The code is shown below:

```
model = Sequential()
model.add(layers.Bidirectional(
layers.LSTM(128, return_sequences=True, dropout=0.2, recurrent_dropout=0.2),
input_shape=(None, supervised_values.shape[-1])))
model.add(layers.Bidirectional(
layers.LSTM(256, dropout=0.2, recurrent_dropout=0.2),
input_shape=(None, supervised_values.shape[-1])))
model.add(layers.Dense(1))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

The model was trained by calling *fit_ generator()* method from Keras Sequential API. It allows to train the model on data generated batch-by-batch by a Python generator. A mechanism of *EarlyStopping* was previously used because it allows to terminate training step when model reaches a loss or accuracy value which exceeds a specified threshold. The training stopped after *12* epochs. After training, the model was saved in a single file.

The *evaluate_ generator()* method was used to evaluate the model on data generator. The accuracy obtained for training is *68.84%* and for test is *57.81%*. The loss of the model is shown in the image below, obtained with Matplotlib:

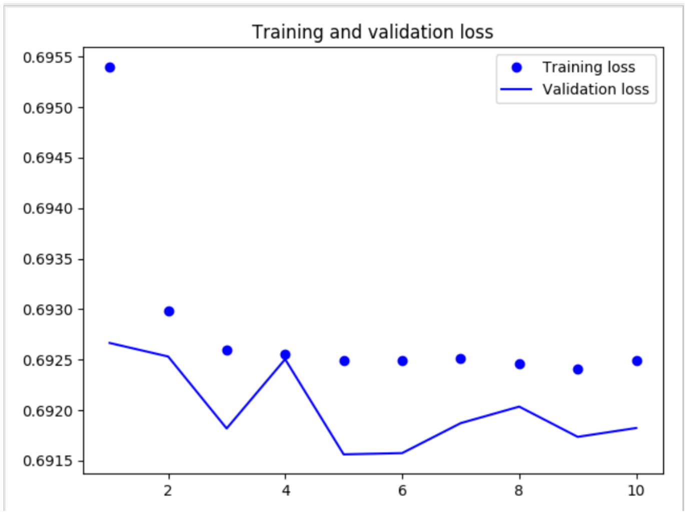


Figure 4: Last model loss

As shown in the image, there is no more overfitting and medium loss is around *0.67*.

5 Conclusions

The purpose of the project was achieved. The results are comparable to baselines or similar models for time series classification.
A possible future work is to upgrade the model to obtain more reliability. This can be done by trying different types of layers, for instance **Convolutional** layers, **Max Pooling** and introducing a mechanism of attention.

References

[Bro16] Jason Brownlee. Lstm recurrent neural networks in python with keras. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/, July 2016.

[BUDA13] D.Sundar B. Uma Devi and Dr. P. Alli. An effective time series analysis for stock trend prediction using arima model for nifty midcap-50, 2013.

[Cho] Francois Chollet. Keras documentation. https://keras.io.

[Cho17] Francois Chollet. Deep learning with python, 2017.

[dax18] Dax. https://en.wikipedia.org/wiki/DAX/, May 2018.