University of Silesia in Katowice
Institute of Physics
Study program: Theoretical Physics

Elements identification based on X radiation spectrum analysis

Solomiia Kurchaba

Katowice 2016

1 Softmax

Source:

https://www.quora.com/What-is-the-intuition-behind-SoftMax-function
https://classroom.udacity.com/courses/ud730

Softmax Regression is a generalization of logistic regression that we can use for multi-class classification (under the assumption that the classes are mutually exclusive). In contrast, we use the (standard) Logistic Regression model in binary classification tasks. The function takes an N-dimensional vector of arbitrary real values and could be defined in following way:

$$\phi_{softmax}(z^i) = \frac{e^{z^i}}{\sum_{j=0}^{k} e^{z_j^i}}, \tag{1}$$

where z is the net output, defined as:

$$z = w_0 x_0 + w_1 x_1 + ... + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^T x,$$

where x_i is the ith element of the input vector.

The output of the function is another N-dimension vector with real values in the range (0,1) that summing up to 1. Therefore, it could be interpret as probability that the given element of the input set belongs to the output class j given the weight matrix (vector) w.

In Python the softmax function could be defined in following way:

```
def softmax(x):
    return np.exp(x)/np.sum(np.exp(x),axis=0),
```

where input matrix x is the net output. In this function is already defined one can use it calling tf.nn.softmax.
Example... Vanishing gradient...

2 Recurrent neural network

Recurrent neural network (RNN) is a type of NN that is used for analysis of sequential data. The most characteristic feature of it is the ability to perform the same computations for each element in the sequence. Main fields of its application are: time series analysis (forecasting),autonomous driving systems, text documents analysis. General idea of the RNN is following: in each step of computation RNN cell combines the input vector with the current state vector with fixed function and produces the output vector. What is important, is that the output vector produced by the RNN is not only influenced by the input that was just feed in, but also by all sequence of the previously feed inputs. The computation precess scheme of

new state generation is presented below.
To begin with, the parameters that are going to be used should be defined: $x^i \in \Re^{S_1}$ - the input matrix in state i of the training procedure;
$h^i \in \Re^{S_2}$ - the hidden state matrix;
$\hat{y^i} \in \Re^{S_3}$ - the output matrix;
$W_{xh} \in \Re^{S_1 \times S_2}$ - the first weight matrix that is applied on the input vector(going to be modified during the back propagation process);
$b \in \Re^{S_2}$ - the bias matrix (also is going to be modified);
$W_{hy} \in \Re^{S_2 \times S_3}$ - the second weight matrix: is applied on the hidden state in order to obtain an output;
$W_{hh} \in \Re^{S_2 \times S_2}$ - the third weight matrix: is applied on the hidden state of the previous state, in order to add it to the current hidden state matrix;
The hidden state is calculated in the following way:
$h^0 = \sigma((x^0 W_{xh})^T + b)$
Having the hidden state we can apply the second weight matrix on it and obtain the output matrix: $\hat{y}^0 = \sigma((h^0 W_{hy})^T + b)$
After receiving the output, the back propagation process should be performed. The aim is to modify all weights and biases that have been used during the computation process in the way that the distance between the correct output y^i and obtained output $\hat{y}^i$ is minimized. Then, the first cycle of the computation process is completed.
The next cycle begins with receiving the new input matrix, which together with the weighted previous hidden state matrix forms a new hidden state:
$h_1 = \sigma((x^1 W_{xh} + h^0 W_{hh})^T + b)$
Note that the matrix W_{xh} is not the same that was used in previous cycle, as it was modified during the back propagation process.

$$\overline{\begin{bmatrix} inp_1 \\ \vdots \\ inp_k \end{bmatrix}} \rightarrow \begin{bmatrix} st_{1,1} & \dots & st_{1,h} \\ \vdots & \dots & \vdots \\ st_{k,1} & \dots & st_{k,h} \end{bmatrix} \times \begin{bmatrix} w_{1,1} & \dots & w_{1,h} \\ \vdots & \dots & \vdots \\ w_{h+1,1} & \dots & w_{h+1,h} \end{bmatrix} + \begin{bmatrix} b_1 & \dots & b_h \end{bmatrix} = \begin{bmatrix} new_{1,1} & \dots & new_{1,h} \\ \vdots & \dots & \vdots \\ new_{k,1} & \dots & new_{k,h} \end{bmatrix}$$

2.1 Example 1

Example of the time series prediction with RNN using the Tensorflow Basic RNN Cellis given below. In this example a random dataset is generated. The task for RNN is to predict the value that follows the given sequence.

```
import numpy as np
import tensorflow as tf
```

```
import matplotlib.pyplot as plt
import random

# Generation of random time series

random.seed(1111)
rng=pd.date_range(start='2000',end="2017",period=209,freq="M")
ts=pd.Series(np.random.uniform(-10,10,size=len(rng)),rng).cumsum()



# Data preparation

TS=np.array(ts)
num_periods=20     # size of the batch
f_horizon=1        # how many steps ahead we are going to predict
# Sequence length should be divisible into n equal batches
x_data=TS[:(len(TS)-(len(TS) % num_periods))]
# As 'label' for each element of the sequence we assign the element which follows
that given element in the original sequence
y_data=TS[1:(len(TS)-(len(TS) % num_periods))+f_horizon]

#generates data batches from the TS with legth adapted to the number of periods
# x_data and y_data should be of one shape
x_batches=x_data.reshape(-1,20,1)
y_batches=y_data.reshape(-1,20,1)

print(np.shape(TS))
print(np.shape(x_data))
print(np.shape(x_batches))
[out]: (204,)
       (200,)
       (10, 20, 1)

# Test data preparation
def test_data(series,forecasts,num_periods):
    test_x_setup=TS[-(num_periods+forecasts):]
    testX=test_x_setup[:num_periods].reshape(-1,20,1)
    testY=TS[-(num_periods):].reshape(-1,20,1)
```

```
    return testX, testY
X_test, Y_test=test_data(TS,f_horizon,num_periods)


np.shape(X_test)
[out]: (1, 20, 1)
```

In this example the basic RNN cell that is already defined in Tensorflow(tf.contrib.rnn.BasicRNNCell) will be used. The arguments of this function are: size of the hidden layer , and activation function. In our case it is a ReLU (Rectified Linear Unit) function. Moreover, it is important to underline that the size of hidden layer has nothing in common with the batch size, or any other parameters that were used during the creation of the data set. Size of the cell is treated as absolutely independent parameter. Official documentation could be found under the link below:

```
https://github.com/tensorflow/tensorflow/blob/r1.2
/tensorflow/python/ops/rnn_cell_impl.py
```

The first step in the process of the building of the computation graph is definition of the placeholders for the variables that are going to be used.

```
tf.reset_default_graph()
num_periods=20
inputs=1
hidden=100
output=1
X=tf.placeholder(tf.float32,[None,num_periods,inputs])
Y=tf.placeholder(tf.float32,[None,num_periods,output])
```

Note, that in this moment we just prepare a space for values of given size, the values itself will be feed later, after activation of a session. For more details and simple examples see:

```
http://learningtensorflow.com/lesson4/
```

Next step is a creation of a computation unit - RNN cell in the way it was described earlier and the recurrent neural network itself.

```
basic_cell=tf.contrib.rnn.BasicRNNCell(num_units=hidden,activation=tf.nn.relu)
rnn_output, states=tf.nn.dynamic_rnn(basic_cell,X,dtype=tf.float32)
```

To run the Dynamic RNN function one should specify a computation unit (in our case Basic RNN Cell) and an input: tensor or tuple of tensors of same dimension. As an output, the function produces a pair of tensors. The first one is interpreted as an output, and the second

as next state (the one to which the following input will be added). It is important to mention that at the moment when one defines the cell, all weights and biases that correspond to the learning process are defined automatically.

In the following steps we create the fully connected dense layer from the sequence of outputs from the network. As a result the tensor of the size of the training/test batch is obtained.

```
stacked_rnn_output=tf.reshape(rnn_output,[-1,hidden])
stacked_outputs=tf.layers.dense(stacked_rnn_output,output)
outputs=tf.reshape(stacked_outputs,[-1,num_periods,output])
loss=tf.reduce_sum(tf.square(outputs-Y))
```

Now is time for the optimisation of all parameters in order to minimize the distance between the correct output, and the one, predicted by the network.

```
learning_rate=0.001
optimizer=tf.train.AdamOptimizer(learning_rate=learning_rat)
training_op=optimizer.minimize(loss)
```

In this point the process of creation of the tensorflow computation graph finishes. The following steps are: initialisation of the variables, feeding the earlier created dictionaries with the real values, and calculation of the prediction vector.

```
epochs=1000
with tf.Session() as sess:
    init.run()
    for ep in range(epochs):
        sess.run(training_op,feed_dict={X: x_batches, Y: y_batches})
        if ep % 100 == 0:
            mse=loss.eval(feed_dict={X: x_batches, Y: y_batches})
            print(ep,"\tMSE:",mse)
    y_pred=sess.run(outputs,feed_dict={X:X_test})
    print(y_pred)
```

3 RNN on real data sets

3.1 One feature, one step ahead prediction

In this example data from the Temp.csv file will be used. From the data set that contains 100 features we extract only one.

```
d1=pd.read_csv("Temp.csv")
data=np.array(d1.V1)
```

Now, the data is going to be prepared in the way, it was done in previous example (we divide the data into training and test set, and reshape it into 3-D tensor, or sequence of matrices of equal size). Inputs and output in list of parameters of the data preparation function refer to the number of features that are used learned and predicted values.

```
def data_prep(data,num_periods=200,f_horizon=1,inputs=1,output=1):
    data_train=data[0:1210]
    x_train=data_train[:(len(data_train)-(len(data_train)%num_periods))]
    y_train=data_train[f_horizon:(len(data_train)-(len(data_train)%num_periods)+f_horizo
    x_batches=x_train.reshape(-1,num_periods,inputs)
    y_batches=y_train.reshape(-1,num_periods,output)
    data_test=data[1210:len(data)]
    x_test=data_test[:(len(data_test)-(len(data_test)%num_periods))]
    y_test=data_test[f_horizon:(len(data_test)-(len(data_test)%num_periods)+f_horizon)]
    x_test=x_test.reshape(-1,num_periods,inputs)
    y_test=y_test.reshape(-1,num_periods,output)
    return x_batches, y_batches, x_test, y_test

x_batches, y_batches, x_test, y_test=data_prep(data)
```

Next steps are graph definition and learning procedure, that are performed in the way, similar to the previous section. The best outcome (the smallest MSE value) was obtained with following parameters:

```
num_periods=200
hidden=100
learning_rat=0.001
epochs=2000
basic_cell=tf.contrib.rnn.BasicRNNCell(num_units=hidden)
# RNN cell with default activation function tanh
```

MSE value was calculated in following way:

```
loss=tf.reduce_mean(tf.squared_difference(outputs, Y))
mse=loss.eval(feed_dict={X: x_batches, Y: y_batches})
mse_test=loss.eval(feed_dict={X:x_test,Y:y_test})
[out]: 3.58295
       5.35141
```

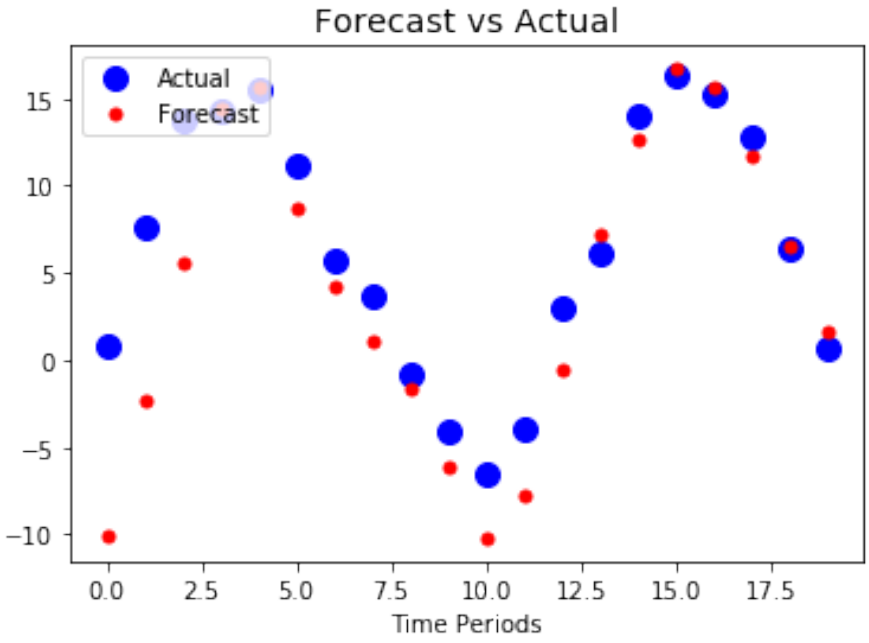

Figure 1: Two step ahead prediction visualisation

3.2 Two steps ahead one feature prediction

In this experiment the data in the way that the label (y value)that corresponds to the value x_t is x_{t+2}. As an activation function, similarly to the previous experiment default tanh was used. Other parameters, as well as obtained results are presented below.

```
x_batches, y_batches, x_test, y_test=data_prep(data,f_horizon=2)
num_periods=200
hidden=100
learning_rat=0.001
epochs=2000
[out]: 3.48955
       5.44422
```

3.3 Four features learning, four features one step ahead prediction

In this experiment two variables of the Temp.csv dataset are going to be used.

```
data=np.array([d1.V1,d1.V2,d1.V3,d1.V4])
#Data preparation
Data=[]
for i in range(len(data[:,0])):
    Data.append(data[i][:(len(data[i])-(len(data[i]) % num_periods))])
```

```
Data=np.array(Data)
data_train=Data[:,0:1210]
x_train=[]
y_train=[]
x_test=[]
y_test=[]
data_test=Data[:,1210:len(Data[0])]
for j in range(len(data[:,0])):
    x_train.append(data_train[j][:(len(data_train[j])-(len(data_train[j])%num_periods))]
    y_train.append(data_train[j][f_horizon:(len(data_train[j])-(len(data_train[j])%num_p
    x_test.append(data_test[j][:(len(data_test[j])-(len(data_test[j])%num_periods))])
    y_test.append(data_test[j][f_horizon:(len(data_test[j])-(len(data_test[j])%num_perio
x_train=np.array(x_train)
y_train=np.array(y_train)
x_test=np.array(x_test)
y_test=np.array(y_test)
x_batches=(np.stack(x_train,axis=-1)).reshape(-1,num_periods,len(data[:,0]))
y_batches=(np.stack(y_train,axis=-1)).reshape(-1,num_periods,len(data[:,0]))
x_test=(np.stack(x_test,axis=-1)).reshape(-1,num_periods,len(data[:,0]))
y_test=(np.stack(y_test,axis=-1)).reshape(-1,num_periods,len(data[:,0]))
```

Parameters that were used during the training:

```
Activation function: tanh
num_periods=200
inputs=len(data[:,0])
hidden=200
learning_rat=0.001
```

Mean square error for the training set equals 1.27, for the test set: 4.07.

3.4 New composition of training and test sets

In this experiment we will try to predict the values of the one feature in a given moment of time based on the values of the another feature in this moment.
Data preparation:

```
data_v=np.array(d1.V10)
data_l=np.array(d1.V11)
```

```
num_periods=200
f_horizon=1
#values
Data_v=data_v[:(len(data_v)-(len(data_v) % num_periods))]
data_train_v=Data_v[0:1210]
x_train=data_train_v[:(len(data_train_v)-(len(data_train_v)%num_periods))]
x_batches=x_train.reshape(-1,num_periods,1)
data_test_v=Data_v[1210:len(Data_v)]
x_test=data_test_v[:(len(data_test_v)-(len(data_test_v)%num_periods))]
x_test=x_test.reshape(-1,num_periods,1)
#labels
Data_l=data_l[:(len(data_l)-(len(data_l) % num_periods))]
data_train_l=Data_l[0:1210]
y_train=data_train_l[:(len(data_train_l)-(len(data_train_l)%num_periods))]
y_batches=y_train.reshape(-1,num_periods,1)
data_test_l=Data_l[1210:len(Data_l)]
y_test=data_test_l[:(len(data_test_l)-(len(data_test_l)%num_periods))]
y_test=y_test.reshape(-1,num_periods,1)
```

Using the following parameters:

```
Activation: tanh
learning_rate=0.001
number_of_epochs=2000
```

Mean square error for the training set equals to 0.39 and for the test set 1.82.

3.5 One step ahead prediction of the on feature based on the values of the other feature

Data preparation:

```
Data_v=data_v[:(len(data_v)-(len(data_v) % num_periods))]
data_train_v=Data_v[0:1210]
x_train=data_train_v[:(len(data_train_v)-(len(data_train_v)%num_periods))]
x_batches=x_train.reshape(-1,num_periods,1)
data_test_v=Data_v[1210:len(Data_v)]
x_test=data_test_v[:(len(data_test_v)-(len(data_test_v)%num_periods))]
x_test=x_test.reshape(-1,num_periods,1)
#labels
```

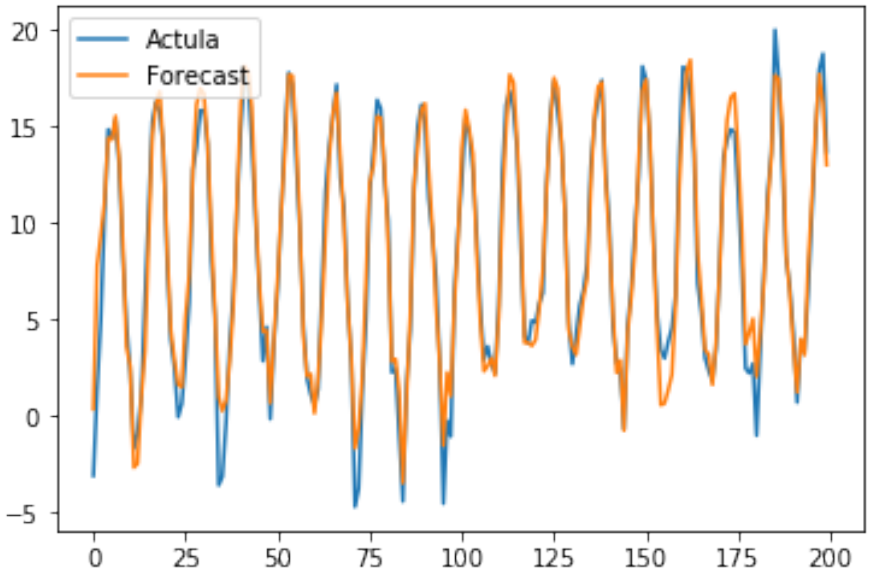

Figure 2: Prediction of the one feature based on the value of the other one

```
Data_l=data_l[:(len(data_l)-(len(data_l) % num_periods))]
data_train_l=Data_l[0:1210]
y_train=data_train_l[1:(len(data_train_l)-(len(data_train_l)%num_periods)+1)]
y_batches=y_train.reshape(-1,num_periods,1)
data_test_l=Data_l[1210:len(Data_l)]
y_test=data_test_l[1:(len(data_test_l)-(len(data_test_l)%num_periods)+1)]
y_test=y_test.reshape(-1,num_periods,1)
```

Parameters that were used:

```
Activation: tanh
num_periods=200
hidden=150
Learning rate=0.01
Number of epochs: 2000
```

Mean squared error for the training set equals to 2.044, and for the test set: 2.88.

3.6 One feature one step ahead prediction based of the values of several features

First, we should divide our data set into set of values (in our case ten columns of the Temp.csv data set), and set of labels - one column, values of which we are going to predict. The data preparation functions are given below:

```
def dat_val_prep(data,num_periods=200,f_horizon=1,num_unit=1210):
    Data=[]
    for i in range(len(data[:,0])):
        Data.append(data[i][:(len(data[i])-(len(data[i]) % num_periods))])
    Data=np.array(Data)
    data_train=Data[:,0:num_unit]
    x_train=[]
    x_test=[]
    data_test=Data[:,num_unit:len(Data[0])]
    for j in range(len(data[:,0])):
        x_train.append(data_train[j][:(len(data_train[j])-(len(data_train[j])%num_perioc
        x_test.append(data_test[j][:(len(data_test[j])-(len(data_test[j])%num_periods))]
    x_train=np.array(x_train)
    x_test=np.array(x_test)
    x_batches=(np.stack(x_train,axis=-1)).reshape(-1,num_periods,len(data[:,0]))
    x_test=(np.stack(x_test,axis=-1)).reshape(-1,num_periods,len(data[:,0]))
    x_test[np.isnan(x_test)] = 0
    x_batches[np.isnan(x_batches)]=0
    return x_batches, x_test

def dat_labl_prep(data_l,num_periods=200,f_horizon=1,num_unit=1210):
    Data_l=data_l[:(len(data_l)-(len(data_l) % num_periods))]
    data_train_l=Data_l[0:num_unit]
    y_train=data_train_l[1:(len(data_train_l)-(len(data_train_l)%num_periods)+1)]
    y_batches=y_train.reshape(-1,num_periods,1)
    data_test_l=Data_l[num_unit:len(Data_l)]
    y_test=data_test_l[1:(len(data_test_l)-(len(data_test_l)%num_periods)+1)]
    y_test=y_test.reshape(-1,num_periods,1)
    y_batches[np.isnan(y_batches)]=0
    y_test[np.isnan(y_test)]=0
    return y_batches, y_test
```

Due to the new composition of the data, placecholders for the TensorFlow computation trees should be constructed in different way. Namely, the placeholder for the values, now, is the tensor of size (None,200,10), as ten values are going to be fed into computation graph in one time. And size of the labels placeholder is of size (None,200,1), as only 1 value is predicted. The function for building and running the computation tree is given below.

```
def build_graph_RNN(data,epochs,num_periods=200,hidden,output=1,learning_rat):
    tf.reset_default_graph()
    inputs=len(data[:,0])
    X=tf.placeholder(tf.float32,[None,num_periods,inputs])
    Y=tf.placeholder(tf.float32,[None,num_periods,output])
    basic_cell=tf.contrib.rnn.BasicRNNCell(num_units=hidden)
    rnn_output, states=tf.nn.dynamic_rnn(basic_cell,X,dtype=tf.float32)
    stacked_rnn_output=tf.reshape(rnn_output,[-1,hidden])
    stacked_outputs=tf.layers.dense(stacked_rnn_output,output)
    outputs=tf.reshape(stacked_outputs,[-1,num_periods,output])
    loss=tf.reduce_mean(tf.squared_difference(outputs, Y))
    optimizer=tf.train.AdamOptimizer(learning_rate=learning_rat)
    training_op=optimizer.minimize(loss)
    init=tf.global_variables_initializer()
    with tf.Session() as sess:
        init.run()
        for ep in range(epochs):
            sess.run(training_op,feed_dict={X: x_batches, Y: y_batches})
            if ep % 100 == 0:
                mse=loss.eval(feed_dict={X: x_batches, Y: y_batches})
                print(ep,"\tMSE:",mse)
        y_pred=sess.run(outputs,feed_dict={X:x_test})
        mse_test=loss.eval(feed_dict={X:x_test,Y:y_test})
    #print(mse)
    #print(mse_test)
    print(mse, mse_test)
    return y_pred
```

Running the function with the same parameters as in previous experiment we receive the following results:

```
y_pred=build_graph_RNN(data,2000,hidden=150,learning_rat=0.01)
[out]: 0.0272222 3.68108
```

We can see significant improvement of the accuracy for the training set, while slight decrease of the accuracy obtained on the test set. The increased accuracy could be logically explained by the fact that ten variables better explain the changes of the our predicted variable, then only one variable. However, the decrement of the test accuracy could be a sign of overfitting. So lets try to run this experiment again, but with slightly different parameters:

```
y_pred=build_graph_RNN(data,1300,hidden=150,learning_rat=0.0001)
[out]: 1.98443 2.7302
```

We can see that, as was expected, reduction of the training time resulted in improvement of the test accuracy and decrease of the accuracy on the training set. This outcome seems to be more balanced.

4 LSTM

Now lets try to perform several task, but using Long Short-term Memory unit cell, instead of RNN. In the first experiment we are going to predict the value of the feature based on its previous value. The function $data - prep$ is used for data preparation. Computation graph is defined similarly to previous tasks, but instead of tf.contrib.rnn.BasicRNNCell, tf.contrib.rnn.BasicLSTMCell is used. With the following parameters:

```
epochs=1000
num_periods=200
hidden=100
learning_rat=0.001
3.30791 3.7742
```

We can see that with twice less number of iteration the LSTM cell is able to produce better results on this simple task.

5 NEW

Our goal here is to construct a well performing Recurrent Neural Network for the time series prediction. For this task as a training data, the sequence generated by the function $x(t_1, t_2) = cos(2\pi t_2)sin(2\pi t_1) + N(0, 0.2)$, where $t_1 \in [0; 9]$, $t_2 \in [0; 18]$, is used. The total length of the sequence equals 900. Test set contains 98 elements. The sequence for the test test is generated by the same function as for raining set, however, without noise. This way of sets construction helps us to understand which of the RNN models the best learns the genuine pattern of the data, and which gets good accuracy by simple coincidence of noise overlapping.

```
t=np.arange(0,9,0.01)
t1=np.arange(0,18,0.02)
mu, sigma = 0, 0.2 # mean and standard deviation
s = np.random.normal(mu, sigma, 900)
x1=(np.cos(2*np.pi*t1)*(np.sin(2*np.pi*t))+s)
t_=np.arange(9,10,0.01)
t1_=np.arange(18,20,0.02)
x2=np.cos(2*np.pi*t1_)*(np.sin(2*np.pi*t_))
x=np.concatenate([x1,x2])
```

Fist step of the experiment is a construction of the naive model from which we will obtain the accuracy baseline for a given type of data. For this type of data we use a so called Constant model. The model works in the following way: suppose the set of the values that are going to be predicted by the model is denoted by $X = \{x_1, x_2, ..., x_t\}$, and the set of the correct answers is denoted by $Y = \{y_1, y_2, ..., y_t\}$. Then, the predictions of the Constant model are generated according to the following scheme: $x_t = y_{t-1}$. Or in words, the Constant model always predicts the last value, it have experienced to be correct. For accuracy evaluation we use the Mean Square Error value.

```
from sklearn import metrics
x_t1=y_test[0][:89]
x_t=y_test[0][1:]
metrics.mean_squared_error(x_t,x_t1)
```

The baseline accuracy for the given task is: $MSE = 0.00528$

Now, its time to build the first TensorFlow recurrent model for the time series prediction. In this model, the neural network should predict the next value, based on the previous value of the sequence. Therefore, the data that is going to be fed to TensorFlow computation graph should be prepared in the following way: lets the set of inputs will be denoted as $X = \{x_1, x_2, ..., x_t\}\}$, then the set of the output values Y should be of the following contents $Y = \{y_1 = x_2, y_2 = x_3, ..., x_t = x_{t+1}\}$. Python implementation of the data preparation function is given below.

```
def data_preparation(data,time_wind,batch_size=1,output=1):
    data_train=data[:901]
    x_train=[]
    for i in range((len(data_train)-1)-(time_wind-1)):
        x_train.append(data_train[i:(i+time_wind)])
    x_train=np.array(x_train)
    x_batches=x_train.reshape(-1,batch_size,time_wind)
    y_train=data_train[time_wind:len(data_train)]#+3]
```

```
    y_batches=y_train.reshape(-1,batch_size,output)
    data_test=data[902:]
    x_t=[]
    for i in range((len(data_test)-1)-(time_wind-1)):
        x_t.append(data_test[i:(i+time_wind)])
    x_t=np.array(x_t)
    x_test=x_t.reshape(-1,batch_size,time_wind)
    y_t=data_test[time_wind:len(data_test)]
    y_test=y_t.reshape(-1,batch_size,output)
    return x_batches, y_batches, x_test, y_test
```

In the function, variable $time\_wind$ refers to the number of historical variables based on which prediction is going to be made. In our case $time\_wind = 1$.

The next step in the model construction process is definition of the computation graph. In Tensorflow, the computation graph is an empty (not filled with the data) model according to construction of which, the following computation process is performed. To define the graph, one should specify placeholders for the sets of inputs and outputs variables that are going to be fed in there. Note, that size of the placeholders should be the same as the size of the input/output sets. The first dimension of the input/output tensor could remain unspecified, as amount of the data for training and test sets probably be different.

```
print(np.shape(x_bacthes),np.shape(y_batches))
[out]: (900,1,1),(900,1,1)
print(np.shape(x_test),np.shape(y_test))
[out]: (97,1,1),(97,1,1)
```

```
batch_size=1
time_wind=1
output=1
X=tf.placeholder(tf.float32,[None,batch_size,time_wind])
Y=tf.placeholder(tf.float32,[None,batch_size,output])
```

Then, the neural network computation unit (state), and the neural network itself should be defined. As a computation unit in our case, the basic RNN cell is used. There are two important parameters that should be specified, while running the function. The first parameter num_units refers to the size of the computation unit. Note, the bigger size of the computation unit, the more details the system will be able to learn. Sometimes this quality is desired, however, in our example,too big size will result in overfitting. Overfitting is caused by the fact that the system during the training tries to match the output with the shape of the noisy data,

and, as a result, the accuracy of the prediction of the denoised signal is spoiled. The second parameter is the activation function- in our case ReLU function.

```
basic_cell=tf.contrib.rnn.BasicRNNCell(num_units=hidden,activation=tf.nn.relu)
rnn_output, states=tf.nn.dynamic_rnn(basic_cell,X,dtype=tf.float32)
```

The function *tf.nn.dynamic_rnn* creates a recurrent neural network basing on the specified computation unit. The function produces two outputs: the next state (the next version the computation unit), and the output of the the neural network. In this point it is important to underline that when one defines the computation unit,all weights and biases of the network are defined automatically.

In the following steps our output is going to be transformed, and the loss calculated. The during the back propagation process weights and biases will be optimise, in order to minimize the loss function.

```
stacked_rnn_output=tf.reshape(rnn_output,[-1,hidden])
stacked_outputs=tf.layers.dense(stacked_rnn_output,output)
outputs=tf.reshape(stacked_outputs,[-1,batch_size,output])
loss=tf.reduce_mean(tf.squared_difference(outputs, Y))
optimizer=tf.train.AdamOptimizer(learning_rate=learning_rat)
training_op=optimizer.minimize(loss)
```

Now, when the computation graph is ready, all variables should be initialize. We do this by calling the function *tf.global_variables_initializer*(). And only in this point the computation graph is ready to be fed with the data. In the following peace of code we activate the TensorFlow session, fill in the placeholders, obtain the predictions and calculate the accuracy.

```
 with tf.Session() as sess:
        init.run()
        for ep in range(epochs):
            sess.run(training_op,feed_dict={X: x_batches, Y: y_batches})
            if ep % 100 == 0:
                mse=loss.eval(feed_dict={X: x_batches, Y: y_batches})
                print(ep,"\tMSE:",mse)
        y_pred=sess.run(outputs,feed_dict={X:x_test})
        mse_test=loss.eval(feed_dict={X:x_test,Y:y_test})
    #NMSE=(mse/len(x_train))/np.var(x_train)
    #NMSE_test=(mse_test/len(x_test.reshape(-1)))/np.var(x_test.reshape(-1))
        print(mse)
        print(mse_test)
    sess.close()
```

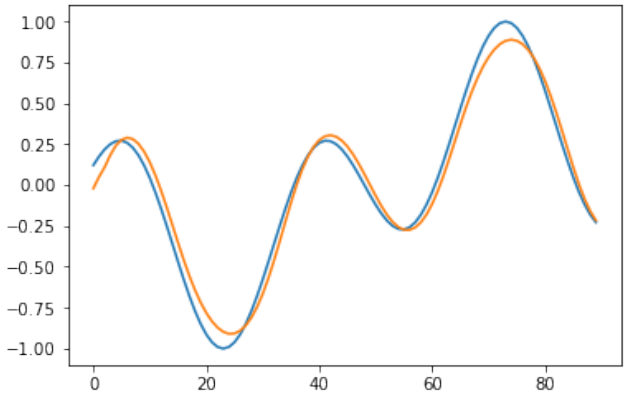

Figure 3: Prediction visualisation. $time\_wind = 1$

The Mean Squared Error for the training set equals $MSE_{tr} = 0.06584$, and for the test set: $MSE_t = 0.00655$. It is important to discuss each of those results, so let us begin the second one.

First of all, one should notice that the accuracy obtained from the trained model is lower (MSE is higher) than the one obtained from the naive model.At first sight it seems to be very strange, however, such results is exactly what we have expected. The reason for such a low accuracy is the way, the data was prepared. Namely, when the system at each time step receive for the training the sequence of length two (x_t, x_{t+1}), everything that it is able to learn is that the following value is sometimes lower than the previous one, and sometimes higher. So the best thing it could do in this situation, is to simply apply the naive predictor, what means, always predict the last seen correct answer. The visualisation of the prediction could be seen on the graph. Another unusual thing that could be noticed is that the accuracy for the training set is lower than the one for the test set. However, it was also expectable. The reason is the different functions that generated the data for the training and test sets. During the creation of the training set, the noise with variance $\sigma = 0.2$ was applied. Therefore, the best possible predictor is not able to obtain better accuracy than the value of the variation of noise. In our case the MSE value for the training set is lower than 0.2but it suggests only about the overfitting during the training procedure. On the other hand, as the test set was generated without noise, the best estimator is able to produce the prediction of absolute accuracy, therefore our, even thought, poor predictor gives us accuracy better, than the one, obtained from the training.

In the second experiment the training and the test sets are generated by the same functions as previously, however the way it is fed into tensorflow computation tree will be different. This time the task for the system is to predict next value x_t of the sequence basing on the three previous values $(x_{t-3}, x_{t-2}, x_{t-1})$ of this sequence. In Python this change means only that the parameter $time\_wind$ this time equals to 3. While running the experiment following results are obtained: $MSE_{tr} = 0.05912$, $MSE_t = 0.01147$. As one can see, the obtained accuracy is even

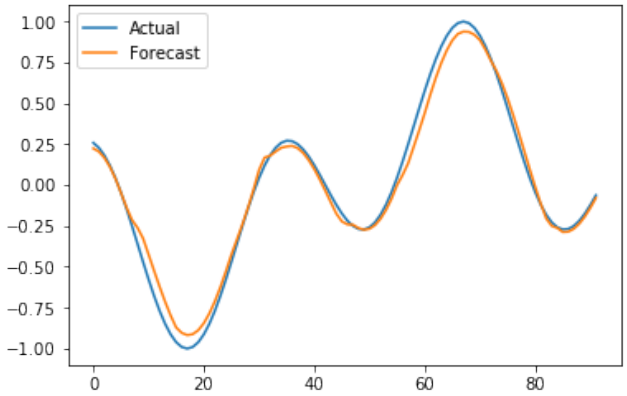

Figure 4: Prediction visualisation. $time\_wind = 6$

worse than the one that was obtained earlier, what means that the addition information that we have provided to the system was rather destructive. We provide the system the possibility to learn, but the pattern that it was able to learn from window of such size, was wrong.

In the next experiment we ask system to predict the next value x_t, basing on the six previous values of the sequence. The obtained accuracy results are: $MSE_{tr} = 0.05966$, $MSE_t = 0.00415$. Now we can see, that our accuracy on the test set is, finally, lower than the baseline. What means that the information that we provided for the system brings new, and this time correct, evidences about the generation function. The graph presents the visualisation of the predicted sequence versus the expectable (true) values.

In the forth experiment the prediction is made basing on ten previous values of the system. The obtained MSE values are: $MSE_{tr} = 0.05423$, and $MSE_t = 00447$. The results are still better than the baseline, however, worse than those, obtained with time window of length six. The most probable explanation for those results is that the training performed on such amount of noisy data caused that the network managed find parameters that successfully matches the noisy, what later causes the wrong pattern of the predicted sequence.