QBUS6840: Tutorial 12 – Recurrent Neural Network

Objectives

- Develop an understanding of recurrent neural networks
- Implement recurrent neural network architecture
- Develop Python skills

In this tutorial, we will keep working on the example with the AirPassengers dataset in the last tutorial but using the recurrent neural network to make forecasts. The step 1-3 will be repeated for the data pre-processing task.

1. Import required libraries/functions and load the data

```
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error

from keras.layers.core import Dense, Activation
from keras.layers.recurrent import LSTM
from keras.models import Sequential
```

Set the RNG seed so we get the same results every time

```
np.random.seed(1)
```

Load the Airpassenger data from a csv file

```
data = pd.read_csv('AirPassengers.csv', usecols=[1])
data = data.dropna()  # Drop all Nans
data = data.values
```

2. Pre-processing (Scaling)

```
scaler = MinMaxScaler(feature_range=(0, 1))
data = scaler.fit_transform(data)
```

3. Create training and testing features

```
time_window = 12

Xall, Yall = [], []

for i in range(time_window, len(data)):
    Xall.append(data[i-time_window:i, 0])
    Yall.append(data[i, 0])
```

```
Xall = np.array(Xall)    # Convert them from list to
array
Yall = np.array(Yall)

train_size = int(len(Xall) * 0.8)
test_size = len(Xall) - train_size

Xtrain = Xall[:train_size, :]
Ytrain = Yall[:train_size]

Xtest = Xall[-test_size:, :]
Ytest = Yall[-test_size:]
```

Recurrent NN and LSTM (Long-Short Term Memory)

Currently the network is a conventional feed forward network with one hidden layer. This conventional design is not optimal for modelling data with time or spatial dependency. In other words, each input data feature is independent of the other and the relationship between neurons is also independent each other. We have tricked the feed forward network into modelling this relationship by carefully constructing our feature inputs.

In contrast RNNs use a set of more complex neurons that store state over a sequence of inputs. They naturally accept sequences of inputs and can produce sequences as outputs or single values. This is ideal for modeling time series data since the input is a set of sequences and we may wish to produce a sequence of outputs or a single output as our prediction. RNN can use many different types of recurrent structures to capture the temporal dependence in time series.

LSTM is the most important variant of the recurrent neuron network. The main idea of the LSTM is to use LSTM cells within a recurrent neuron network to capture the long term dependence and nonlinear effects that could be exhibited in the time series. Using Keras, changing our neural network to use a LSTM layer is relatively straight forward.

4. Modify the Model

```
# For time series and LSTM layer we need to reshape into
3D array
Xtrain = np.reshape(Xtrain, (Xtrain.shape[0],
time_window, 1))
Xtest = np.reshape(Xtest, (Xtest.shape[0], time_window,
1))

model = Sequential()
# Add a LSTM with units (number of hidden neurons) = 50
# input_dim = 1 (for time series)
# return sequences = False means only forward the last
```

```
lagged output to the following layer
model.add(LSTM(
        input_shape=(None, 1),
        units=50,
        return_sequences=False))    # Many-to-One model
model.add(Dense(
        output_dim=1))
model.add(Activation("linear"))

model.compile(loss="mse", optimizer="rmsprop")
```

Notes

Tuning is much more difficult with LSTM since we now have more interacting parameters and a more complex network. Sometimes tuning for your data can take a lot of time. It is part of the skill of using neural networks. Please familiarize yourself with the terminology and how each parameter affects fitting result.

Units is the dimension of the weight vector inside the LSTM "neurons" or cells. Increasing this number can lead to overfitting, decreasing can lead to poor accuracy. You need to make a tradeoff.

Batch size is the number of samples used in each forward/backward pass of the network. You will notice the number of samples in each Epoch increasing by the batch size.

I find that for time series forecasting increasing the batch size you may get more divergent results (less accuracy) but each Epoch completes quicker.

An **Epoch** is a forward/backward pass of all batches. If you increase the Epoch's you should see an improvement in accuracy since it is refining the model parameters each time.

The **validation split** is the proportion of data held out for use as validation set. As it decreases training accuracy will increase since more training data is available, but the guarantee of generalization is much lower. For small amounts of data and for low epoch's it is better to keep this number small.

5. Train the model with Early Stopping

Early stopping is a technique to prevent over-training when a monitored quantity has stopped improving.

```
from keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor='loss', patience=2,
verbose=1)
```

Here, `monitor` is the quantity to be monitored. `patience` is the number of epochs with no improvement after which training will be stopped. If you are interested with further parameter settings in this function, you can refer to

https://keras.io/callbacks/#earlystopping

Then, we train the model by calling the `.fit()` function

```
model.fit(
         Xtrain,
         Ytrain,
         batch_size=5,
         nb_epoch=100,
         validation_split=0.1)

allPredict = model.predict(np.reshape(Xall,
(Xall.shape[0],time_window,1)))
allPredict_original_scale =
scaler.inverse_transform(allPredict)
allPredictPlot = np.empty_like(data)
allPredictPlot[:, :] = np.nan
allPredictPlot[time_window:, :] =
allPredict_original_scale

plt.figure()
plt.plot(scaler.inverse_transform(data), label='True
Data')
plt.plot(allPredictPlot, label='One-Step Prediction')
plt.legend()
plt.show()

trainScore = math.sqrt(mean_squared_error(Ytrain,
allPredict[:train_size,0]))
print('Training Data RMSE: {0:.2f}'.format(trainScore))
```

6. Dynamic forecast with LSTM

Finally, we also applied the dynamic forecast with LSTM

```
dynamic_prediction = np.copy(data[:len(data) -
test_size])

for i in range(len(data) - test_size, len(data)):
    last_feature = np.reshape(dynamic_prediction[i-
time_window:i], (1,time_window,1))
    next_pred = model.predict(last_feature)
    dynamic_prediction = np.append(dynamic_prediction,
next_pred)

dynamic_prediction = dynamic_prediction.reshape(-1,1)
dynamic_prediction_original_scale =
scaler.inverse_transform(dynamic_prediction)

plt.figure()
```

```
plt.plot(scaler.inverse_transform(data[:len(data) -
test_size]), label='Training Data')
plt.plot(np.arange(len(data) - test_size, len(data), 1),
scaler.inverse_transform(data[-test_size:]),
label='Testing Data')
plt.plot(np.arange(len(data) - test_size, len(data), 1),
dynamic_prediction_original_scale[-test_size:],
label='Out of Sample Prediction')
plt.legend(loc = "upper left")
plt.show()

testScore = math.sqrt(mean_squared_error(Ytest,
dynamic_prediction[-test_size:]))
print('Dynamic Forecast RMSE: {0:.2f}'.format(testScore))
```

Can you fine-tune the hyper-parameters so that we could obtain a better fitting result?

Hint: you can add more hidden layers or increase the number of hidden units. You can also change the windows size and then compare their performance