QBUS6840: Tutorial 11 – Neural Networks

Objectives

- Develop an understanding of feed forward and recurrent neural networks
- Implement neural network architecture
- Develop dynamic forecasting skills
- Develop Python skills

In this tutorial we will take a timeseries, split it into train and test sets then produce one-step and dynamic forecasts using both a feed forward neural network and a recurrent (LSTM) neural network.

1. Install Keras

For this tutorial you will need to install the Keras library.

Keras provides a very high level approach to build, train and predict using neural networks. It is built on top of other neural network libraries like TensorFlow or Theano. It helps to prototype neural networks very quickly.

If you do not have Keras installed already you can type the following on the Windows Command Prompt or Terminal on OS X

```
conda install –c conda-forge keras
```

If you have trouble with install the Keras package, you can refer to the installation guideline on Canvas or the official online tutorial: https://github.com/keras-team/keras

2. 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
from keras.models import Sequential
```

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

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

Make sure to convert the pandas dataframe to an array by using `.values` attribute. Keras expects **array** data type, not **dataframe** or **series**.

```
data = pd.read_csv('AirPassengers.csv', usecols=[1])
```

```
data = data.dropna()  # Drop all Nans
data = data.values
```

3. **Plot the time series**

Please complete this yourself. What can you tell me about the data?

```

```

4. **Pre-processing (Scaling)**

Neural networks normally work best with scaled data, especially when we use the sigmoid or tanh activation function. It is best practice to scale the data to the range of [0,1]. This can be easily done by using scikit-learn's `MinMaxScaler()` function.

```
scaler = MinMaxScaler(feature_range=(0, 1))

data = scaler.fit_transform(data)
```

5. **Create training and testing features**

We must design appropriate targets (predictions) and features (predictors) for our neural network.

We know that since we are dealing with timeseries data our target is a single point (scalar value) for the next observation.

The obvious choice for features is the most recent observations in the timeseries. In other words, in our neural network we will use a small number of past observations to predict a single new observation.

How many past observations should we use to predict a new observation? The length of this window depends on a few things. First we must consider whether there is seasonality in our data. If there is then to model the seasonality in our network the input feature windows must be at least as long as the seasonal period. In this dataset the seasonal period is 12 months. In general the more data you can input into your neural network the more accurate the predictions will become. However increasing the amount of data also increases training time and memory requirements. So we must make a tradeoff.

We also need to split our data into training and testing components. Later we will train our network on the training set (first 80% of timseries) then produce a forecast over the test set (last 20%) and evaluate the accuracy.

```
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:]
```

6. **Define the Feed Forward NN model**

Now it is time to define the structure of our Neural Network. We first try a standard feed forward network architecture. To define a feed forward network we set the model type to be Sequential.

```
model = Sequential()

model.add(Dense(20, input_dim=time_window,
activation='relu'))

model.add(Dense(1))
```

This is considered a one-hidden layer model. Our input gets sent to a layer of 20 neurons then output through a dense layer that combines the output of all 20 neurons.

7. **Compile**

The next step is to compile, which means to check the network for potential errors and to decide on a loss function (the type of penalty applied to the error) and the optimization algorithm.

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

Here, you can also obtain the sum squared error by changing the `loss='mean_squared_error'`.

For more detailed explanation, please refer to https://keras.io/models/model/

8. **Train NN**

Finally, we can train our network. By training we mean to learn the optimal weights of the network edges (links between neurons).

To train we must specify some training features (predictors, called Xtrain) and targets (predictions, called Ytrain) and some other tuning parameters which will be explained later

```
model.fit(Xtrain, Ytrain, epochs=100, batch_size=2,
verbose=2, validation_split=0.05)
```

9. One-Step Forecast

We can first check how well the model fits our data by doing a one-step prediction. This uses all available observations i.e. feeds the neural network with actual observations and we ask for its prediction.

This is also known as an in-sample forecast.

```
allPredict = model.predict(Xall)
allPredictPlot = scaler.inverse_transform(allPredict)

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

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

What is the purpose of `inverse_transform()` function?

Recall that we normalize the data before we feed-in them into the network. By calling this function, we could undo the scaling according to its `feature_range`.

10. Dynamic Forecast

In real world, we do not have access to the future values so we have to do a dynamic (or out of sample) forecast. This forecast uses observations up to some time point. From that point onwards we append our latest forecast to the list and use a combination of real values and estimated values. Eventually if the forecast length exceeds the window of values used we will be only using estimated values.

```
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))
    next_pred = model.predict(last_feature)
    dynamic_prediction = np.append(dynamic_prediction,
next_pred)

dynamic_prediction = dynamic_prediction.reshape(-1,1)
dynamic_prediction =
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[-test_size:], label='Out of Sample
Prediction')
plt.legend(loc = "upper left")

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

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.

Changing our neural network to use a LSTM layer is relatively straight forward.

11. Modify Parameters

Change the time window length

```
time_window = 20
```

Change the fitting arguments

```
model.fit(Xtrain, Ytrain, batch_size=5, nb_epoch=20,
validation_split=0.1)
```

12. Modify the Model

```
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.

13. 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=0,
         validation_split=0.1)

allPredict = model.predict(np.reshape(Xall, (124,20,1)))
allPredict = scaler.inverse_transform(allPredict)
allPredictPlot = np.empty_like(data)
allPredictPlot[:, :] = np.nan
allPredictPlot[time_window:, :] = allPredict

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))
```

14. 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 =
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[-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 results?

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