QBUS6850: Tutorial 11 – Neural Networks II

Objectives

- To learn how to diagnose neural network training results;
- To learn more about keras;
- To learn how to set up recurrent neural networks

1. Keras Diagnostics

Often it is useful to review what occured during training a neural network.

Whenever you call `fit()` on a keras model it returns a history object. Inside the history object is a history dictionary that contains arrays of various values. By default there is only one value, the loss, which is recorded for each iteration. If you add other metrics to the metrics parameter of the `compile()` function those will also be calculated per iteration and available in the history object.

Step 1: Let's generate some toy data to play with:

```
import numpy as np
from sklearn.datasets import make_regression
from sklearn.preprocessing import MinMaxScaler
from keras.layers.core import Dense
from keras.models import Sequential

np.random.seed(0)

X, y = make_regression(n_samples = 100, n_features = 2, bias =
1.5 )

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

X = scaler.fit_transform(X)
y = scaler.fit_transform(y.reshape(-1, 1))
```

It is a good practice to normalize your data, particularly when you are using neural network. Explain the meaning of

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

Step 2: Create the model we are going to use. The model is a single layer neural network with one output with the linear activation function and loss squared error as the lost function. More specifically we are going to use the "Adam" optimiser instead of gradient descent method. Add your code to make this happen and train this model with the training data.

```
# Write your code here before we call model.fit
#
```

```
model = . . .
. . .
. . .

history = model.fit(X, y,  epochs=400, batch_size=16,
verbose=2, validation_split=0)
```

Step 3: Check the performance history in the training process. Keras keeps all the loss function values of all the training epochs. We can easily gather this information for further diagnosis. Check out the following code:

```
import matplotlib.pyplot as plt

loss_list = history.history['loss']

fig = plt.figure()
plt.plot(loss_list)
fig
```

keras fit function is slightly different from the fit functions we have seen in sklearn. Keras fit function returns a “history” object which contains all the information kept in the training stage. Inside its “history” (the second one in the statement) attribute, we can find out all the metrics/score information including loss values.

2. Exploring the Effect of Batch Size

The batch size refers to the number of samples used in each forward/backward pass of the network.

To complete an Epoch you must use every training sample. So if you have 100 training samples and a batch size of 1 you must do 100 forward/backward passes per Epoch.

Generally a lower batch size will leave to greater minimisation per Epoch. In other words the batch size influences the step size or learning rate of our gradient descent. The step size is inversely proportional the batch size. If the batch size goes up then the step size goes down.

However a lower batch size takes more time as there is significiantly more overhead in training with many small batches than a few large batches.

Let's look at an example using the Advertising dataset.

Step 1: Prepare data to play with:

```
import pandas as pd
```

```
from sklearn.model_selection import train_test_split

df = pd.read_csv("Advertising.csv", index_col=0)

df.head()

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

data = scaler.fit_transform(df.values)

df_scaled = pd.DataFrame(data, columns=df.columns)

df_scaled.head()

y = df_scaled["Sales"]
X = df_scaled[ df_scaled.columns.difference(["Sales"]) ]

X_train, X_test, y_train, y_test = train_test_split(X, y)

n_features = X_train.shape[1]
```

Step 2: Now that we have handled our data. let's build a linear regression network and use a small batch size.

Create the model first

```
model_s = Sequential()
model_s.add(Dense(1, input_dim=n_features,
activation='linear', use_bias=True))
model_s.compile(loss='mean_squared_error', optimizer='adam')

n_epochs = 400

weights_wrt_is = np.zeros( (n_epochs, 4) )
batch_s = 1

history = None
```

Step 3: We want to understand the effect of batch size on how quickly we find the optimal parameters.

So we will record the parameter/weights at each training iteration. Fortunately keras allows you to train a few Epochs at a time then resume training!

```
for i in range(n_epochs):
    if history:
        history = model_s.fit(X_train.values, y_train.values,
epochs= i+1, initial_epoch=i, batch_size=batch_s, verbose=2,
validation_split=0)
    else:
        history = model_s.fit(X_train.values, y_train.values,
epochs=1+1, batch_size=batch_s, verbose=2, validation_split=0)

    weights = model_s.layers[0].get_weights()
```

```
    weights_wrt_is[i, :] =  np.concatenate( (weights[1], 
weights[0][:,0]) , axis = 0 )
```

Carefully read this section of code. The code starts with i = 0 to training the network. At the beginning (i=0), variable history is null (empty), if testing will goes down to else part, ie, execute

```
history = model_s.fit(X_train.values, y_train.values, 
epochs=1+1, batch_size=batch_s, verbose=2, validation_split=0)
```

This will train the network 2 iterations (`epochs = 2`). After this step, history is no longer null or empty, hence from i>0, we will run the first fit

```
history = model_s.fit(X_train.values, y_train.values,  epochs= 
i+1, initial_epoch=i, batch_size=batch_s, verbose=2, 
validation_split=0)
```

This statement has one more argument `initial_epoch=i.` The meaning is to run training from the neural networks obtained `i` iterations in previous training, then run till the total epoch (including the initial epoch) given by `epochs ( = i+1).` Actually we run 1 iteration here.

Step 3: Let's plot β_0 vs Epochs. **Notice that the weights converge very quickly.**

```
fig = plt.figure()

plt.plot(weights_wrt_is[:, 0])
```

Have you observed how $\beta 0$ changes along the epochs?

Step 4: Now let's try with a bigger batch size.

Remember that batch sizes should always be a power of 2 to take best advantage of your computers parallelisation abilities!

For example `batch_s = 32.` And store the coefficients in `weights_wrt_il`

Write your code by revising the previous sections of code.

Step 5: Finally lets compare all of the coefficients/weights that we estimated.

```
labels = ['beta_0', 'beta_1', 'beta_2', 'beta_3']
colors = ['orange', 'red', 'blue', 'grey']

fig, axarr = plt.subplots(2, 2)

axarr = np.ravel(axarr)

for i in range(4):
    axarr[i].plot(weights_wrt_il[:, i], label = "{0}
```

```
(16)".format(labels[i]), c = colors[i], linestyle = "solid" )
    axarr[i].plot(weights_wrt_is[:, i], label = "{0}
(1)".format(labels[i]), c = colors[i], linestyle = "dashed")
    axarr[i].legend()
```

3. Time Series

For time series models we need a special layer type. We need a reccurrent layer, that is a layer that uses information from previous time points.

We also need to do some feature engineering. If we want to predict the next item in a sequence we need to observe the sequence i.e. if we want to predict y_{t+1} we need $[y_{t-d},\ldots,y_{t-2},y_{t-1},y_t]$.

For our training data we need to do this for every training sample.

What value should we use for d?

A good practice is to set d to be greater than or equal to the maximum size of any cyclic, seasonal or repeated behaviour that is exhibited in your time series. If it is shorter than that it may not be able to accurately model the behaviour.

Example

To illustrate lets create some sinusoidal data to practice on!

Step 1: Generate data and show its patterns

```
from keras.layers.recurrent import LSTM
from keras.layers.core import Activation
from keras.utils.vis_utils import model_to_dot
from IPython.display import Image, display
np.random.seed(0)
# Create the data
n_points = 1000

data_x = np.linspace(1, 100, n_points)
data = np.sin(data_x)
# Plot the data
fig = plt.figure()
plt.plot(data)
```

Step 2: Now let's generate all the features we are going to use as discussed above!

```
time_window = 100

Xall, Yall = [], []

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

# 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))
```

I hope you can fully understand what the for loop is doing. Can you check the shape of Xall and Yall?

It is easy to see how we split the data into train and test data.

Why do we organize the data in 3D shape?

What is the meaning of the first dimension, the second dimension and the third dimension, respectively?

Step 3: Create and Train A Recurrent Neural Network

Now we will define a simple Reccurent Neural Network. There are many types of Recurrent layers, but we will use the Long-Short Term Memory layer type.

This layer type is a layer with "cells" (a more complex neuron) that accepts input from the previous cell and input from many of the past cells.

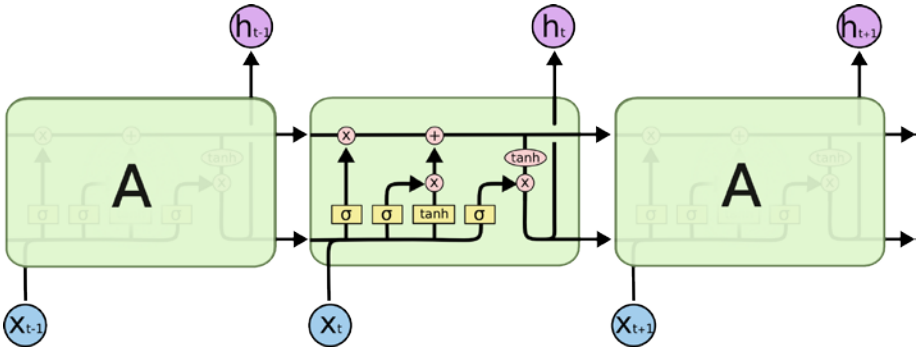

image from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Number of cells is the same as your data size. For example if use the past 10 observations to predict a new observation then the number of LSTM cells is also 10.

Units is the dimension of vector of hidden state, cycle state and the final output size. That is, it is the dimension of h in purple nodes of the above figure. Keras offers us a choice whether output all h's as sequences or the last h on the right. You can set `return_sequences=True` or `False`. In our case,, we use multiple x to match one y, so we only need the last h. Usually these h's will be forwarded to another layer connecting the output. In the following code, we use a Dense layer to connect the last h to our target y.

```
model = Sequential()

model.add(LSTM(units=50, input_shape=(None, 1),
return_sequences=False))
model.add(Dense(1, activation="linear"))

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

# Training
model.fit(Xtrain, Ytrain, batch_size=100, nb_epoch=100,
validation_split=0)
```

Step 4: Let's now plot our prediction and the ground truth.

To predict the forecast we must repeatedly to predictions, one at a time. After each prediction we add our previous prediction to our data and predict again.

```
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)

fig = plt.figure()
plt.plot(data, label = "Original")
plt.plot(dynamic_prediction, label = "Predicted")
plt.legend(loc="lower left")
```