**Time Series Prediction with LSTM RNN in Python with Keras**

Time series prediction problems are a difficult type of predictive modelling problem.

Unlike regression predictive modelling, time series also adds the complexity of a sequence dependence among the input variables.

A powerful type of neural network designed to handle sequence dependence is called recurrent neural networks. The Long Short-Term Memory network or LSTM network is a type of recurrent neural network used in deep learning because very large architectures can be successfully trained.

In this post, you will discover how to develop LSTM networks in Python using the Keras deep learning library to address a demonstration time-series prediction problem.

After completing this tutorial you will know how to implement and develop LSTM networks for your own time series prediction problems and other more general sequence problems. You will know:

* About the International Airline Passengers time-series prediction problem.
* How to develop LSTM networks for regression, window and time-step based framing of time series prediction problems.
* How to develop and make predictions using LSTM networks that maintain state (memory) across very long sequences.

In this tutorial, we will develop a number of LSTMs for a standard time series prediction problem. These examples will show you exactly how you can develop your own differently structured LSTM networks for time series predictive modelling problems.

## **Problem Description**

The problem we are going to look at in this post is the International Airline Passengers prediction problem.

This is a problem where, given a year and a month, the task is to predict the number of international airline passengers in units of 1,000. The data ranges from January 1949 to December 1960, or 12 years, with 144 observations.

Download the dataset (save as “airline-passengers.csv“) from BB .

We can load this dataset easily using the Pandas library. We are not interested in the date, given that each observation is separated by the same interval of one month. Therefore, when we load the dataset we can exclude the first column.

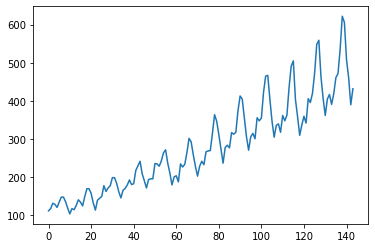
import pandas as pd

import matplotlib.pyplot as plt

dataset = pd.read\_csv('/Users/user/Desktop/7BUIS008W/airline-passengers.csv', usecols=[1], engine='python')

plt.plot(dataset)

plt.show()



You can see an upward trend in the dataset over time.

You can also see some periodicity to the dataset that probably corresponds to the Northern Hemisphere vacation period.

## LSTM Network for Regression

We can phrase the problem as a regression problem.

That is, given the number of passengers (in units of thousands) this month, what is the number of passengers next month?

We can write a simple function to convert our single column of data into a two-column dataset: the first column containing this month’s (t) passenger count and the second column containing next month’s (t+1) passenger count, to be predicted.

Before we get started, let’s first import all of the functions and classes we intend to use. This assumes a working SciPy environment with the Keras deep learning library installed.

import numpy as np

import matplotlib.pyplot as plt

import math

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import LSTM

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error

|  |  |
| --- | --- |
|  |  |

Before we do anything, it is a good idea to fix the random number seed to ensure our results are reproducible.

# fix random seed for reproducibility

np.random.seed(7)

We can also use the code from the previous section to load the dataset as a Pandas dataframe. We can then extract the NumPy array from the dataframe and convert the integer values to floating point values, which are more suitable for modelling with a neural network.

dataset1 = dataset.values

dataset1 = dataset.astype('float32')

LSTMs are sensitive to the scale of the input data, specifically when the sigmoid (default) or tanh activation functions are used. It can be a good practice to rescale the data to the range of 0-to-1, also called normalizing. We can easily normalize the dataset using the **MinMaxScaler** preprocessing class from the scikit-learn library.

# normalize the dataset

scaler = MinMaxScaler(feature\_range=(0, 1))

dataset1 = scaler.fit\_transform(dataset1)

After we model our data and estimate the skill of our model on the training dataset, we need to get an idea of the skill of the model on new unseen data. For a normal classification or regression problem, we would do this using cross validation.

With time series data, the sequence of values is important. A simple method that we can use is to split the ordered dataset into train and test datasets. The code below calculates the index of the split point and separates the data into the training datasets with 67% of the observations that we can use to train our model, leaving the remaining 33% for testing the model.

# split into train and test sets

train\_size = int(len(dataset1) \* 0.67)

test\_size = len(dataset1) - train\_size

train, test = dataset1[0:train\_size,:], dataset1[train\_size:len(dataset),:]

print(len(train), len(test))

Now we can define a function to create a new dataset, as described above.

The function takes two arguments: the **dataset**, which is a NumPy array that we want to convert into a dataset, and the **look\_back**, which is the number of previous time steps to use as input variables to predict the next time period — in this case defaulted to 1.

This default will create a dataset where X is the number of passengers at a given time (t) and Y is the number of passengers at the next time (t + 1).

It can be configured, and we will by constructing a differently shaped dataset in the next section.

# convert an array of values into a dataset matrix

def create\_dataset(dataset1, look\_back=1):

dataX, dataY = [], []

for i in range(len(dataset1)-look\_back-1):

a = dataset1[i:(i+look\_back), 0]

dataX.append(a)

dataY.append(dataset1[i + look\_back, 0])

return np.array(dataX), np.array(dataY)

Let’s take a look at the effect of this function on the first rows of the dataset (shown in the

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | X Y  112 118  118 132  132 129  129 121  121 135 |

If you compare these first 5 rows to the original dataset sample listed in the previous section, you can see the X=t and Y=t+1 pattern in the numbers.

Let’s use this function to prepare the train and test datasets for modeming.

# reshape into X=t and Y=t+1

look\_back = 1

trainX, trainY = create\_dataset(train, look\_back)

testX, testY = create\_dataset(test, look\_back)

The LSTM network expects the input data (X) to be provided with a specific array structure in the form of: [samples, time steps, features].

Currently, our data is in the form: [samples, features] and we are framing the problem as one time step for each sample. We can transform the prepared train and test input data into the expected structure using **numpy.reshape()** as follows:

# reshape input to be [samples, time steps, features]

trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))

testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

We are now ready to design and fit our LSTM network for this problem.

The network has a visible layer with 1 input, a hidden layer with 4 LSTM blocks or neurons, and an output layer that makes a single value prediction. The default sigmoid activation function is used for the LSTM blocks. The network is trained for 100 epochs and a batch size of 1 is used.

create and fit the LSTM network

model = Sequential()

model.add(LSTM(4, input\_shape=(1, look\_back)))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error', optimizer='adam')

model.fit(trainX, trainY, epochs=100, batch\_size=1, verbose=2)

Once the model is fit, we can estimate the performance of the model on the train and test datasets. This will give us a point of comparison for new models.

Note that we invert the predictions before calculating error scores to ensure that performance is reported in the same units as the original data (thousands of passengers per month).

# make predictions

trainPredict = model.predict(trainX)

testPredict = model.predict(testX)

# invert predictions

trainPredict = scaler.inverse\_transform(trainPredict)

trainY = scaler.inverse\_transform([trainY])

testPredict = scaler.inverse\_transform(testPredict)

testY = scaler.inverse\_transform([testY])

# calculate root mean squared error

trainScore = math.sqrt(mean\_squared\_error(trainY[0], trainPredict[:,0]))

print('Train Score: %.2f RMSE' % (trainScore))

testScore = math.sqrt(mean\_squared\_error(testY[0], testPredict[:,0]))

print('Test Score: %.2f RMSE' % (testScore))

We can see that the model has an average error of about 23 passengers (in thousands) on the training dataset, and about 52 passengers (in thousands) on the test dataset. Not that bad.

Finally, we can generate predictions using the model for both the train and test dataset to get a visual indication of the skill of the model.

Because of how the dataset was prepared, we must shift the predictions so that they align on the x-axis with the original dataset. Once prepared, the data is plotted, showing the original dataset in blue, the predictions for the training dataset in green, and the predictions on the unseen test dataset in red.

# shift train predictions for plotting

trainPredictPlot= numpy.empty\_like(dataset1)

trainPredictPlot[:, :] = numpy.nan

trainPredictPlot[look\_back:len(trainPredict)+look\_back, :] = trainPredict

# shift test predictions for plotting

testPredictPlot = numpy.empty\_like(dataset1)

testPredictPlot[:, :] = numpy.nan

testPredictPlot[len(trainPredict)+(look\_back\*2)+1:len(dataset)-1, :] = testPredict

# plot baseline and predictions

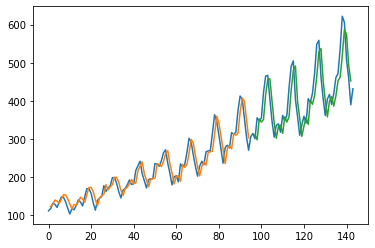
plt.plot(scaler.inverse\_transform(dataset))

plt.plot(trainPredictPlot)

plt.plot(testPredictPlot)

plt.show()

We can see that the model did an excellent job of fitting both the training and the test datasets.



LSTM Trained on Regression Formulation of Passenger Prediction Problem

**LSTM for Regression Using the Window Method**

We can also phrase the problem so that multiple, recent time steps can be used to make the prediction for the next time step.

This is called a window, and the size of the window is a parameter that can be tuned for each problem.

For example, given the current time (t) we want to predict the value at the next time in the sequence (t+1), we can use the current time (t), as well as the two prior times (t-1 and t-2) as input variables.

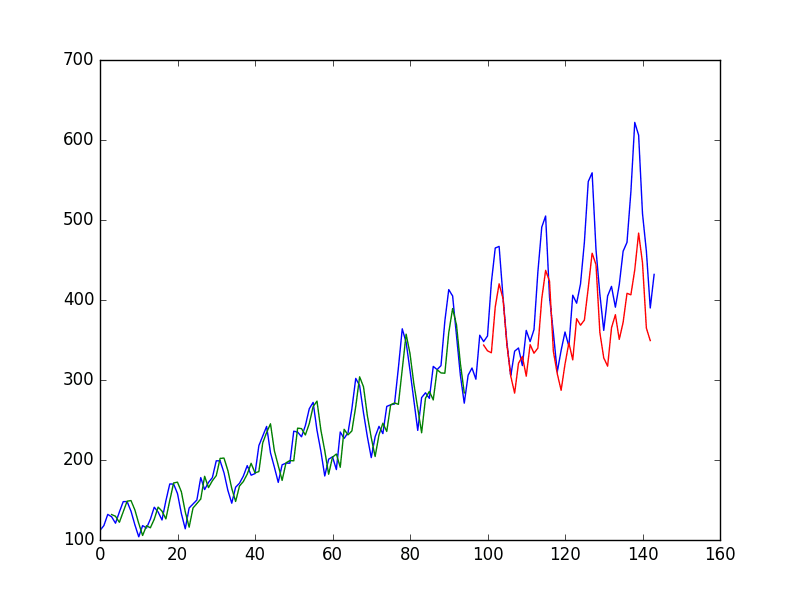
When phrased as a regression problem, the input variables are t-2, t-1, t and the output variable is t+1.

A sample of the dataset with this formulation looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | X1 X2 X3 Y  112 118 132 129  118 132 129 121  132 129 121 135  129 121 135 148  121 135 148 148 |

The **create\_dataset()** function we created in the previous section allows us to create this formulation of the time series problem by increasing the **look\_back** argument from 1 to 3.

We can see that the error was increased slightly compared to that of the previous section. The window size and the network architecture were not tuned: this is just a demonstration of how to frame a prediction problem.



LSTM Trained on Window Method Formulation of Passenger Prediction Problem

## **LSTM for Regression with Time Steps**

You may have noticed that the data preparation for the LSTM network includes time steps.

Some sequence problems may have a varied number of time steps per sample. For example, you may have measurements of a physical machine leading up to a point of failure or a point of surge. Each incident would be a sample the observations that lead up to the event would be the time steps, and the variables observed would be the features.

Time steps provide another way to phrase our time series problem. Like above in the window example, we can take prior time steps in our time series as inputs to predict the output at the next time step.

Instead of phrasing the past observations as separate input features, we can use them as time steps of the one input feature, which is indeed a more accurate framing of the problem.

We can do this using the same data representation as in the previous window-based example, except when we reshape the data, we set the columns to be the time steps dimension and change the features dimension back to 1. For example:

# reshape input to be [samples, time steps, features]

trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))

testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))

