# Time Series Prediction with Deep Learning in Keras

by **Jason Brownlee** on July 19, 2016 in **Deep Learning for Time Series**



Last Updated on August 7, 2022

Time Series prediction is a difficult problem both to frame and address with machine learning.

In this post, you will discover how to develop neural network models for time series prediction in Python using the Keras deep learning library.

After reading this post, you will know:

- About the airline passengers univariate time series prediction problem
- How to phrase time series prediction as a regression problem and develop a neural network model for it
- How to frame time series prediction with a time lag and develop a neural network model for it

**Kick-start your project** with my new book Deep Learning for Time Series Forecasting, including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.

- **Jul/2016**: First published
- **Updated Oct/2016**: Replaced graphs with more accurate versions
- **Updated Mar/2017**: Updated for Keras 2.0.2, TensorFlow 1.0.1 and Theano 0.9.0
- **Updated Apr/2019**: Updated the link to dataset
- **Updated Sep/2019**: Updated for Keras 2.2.5
- **Updated Jul/2022**: Updated for TensorFlow 2.x

## Problem Description

The problem you will 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*").

Below is a sample of the first few lines of the file.

```
"Month","Passengers"
"1949-01",112
"1949-02",118
"1949-03",132
"1949-04",129
```

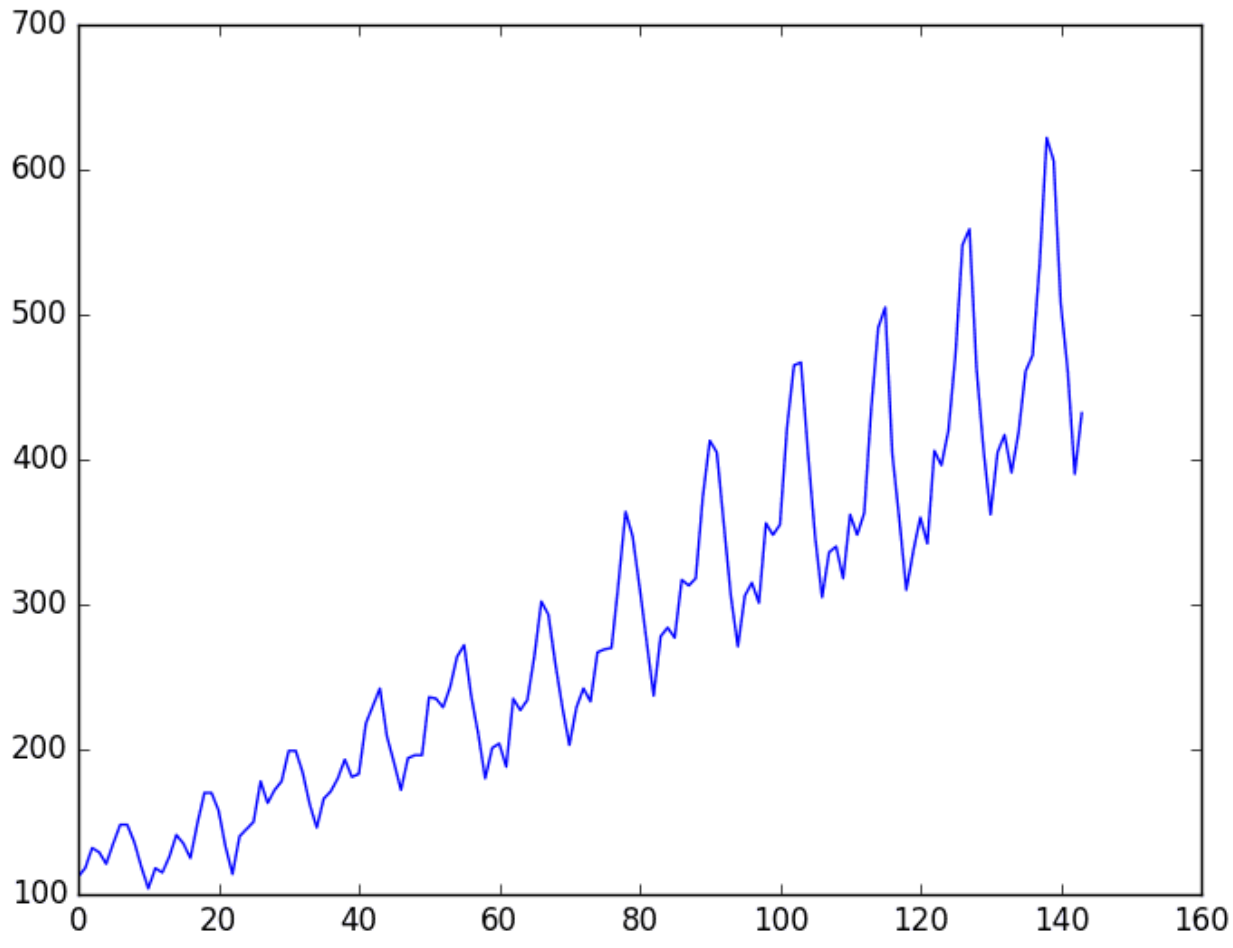
You can load this dataset easily using the Pandas library. You are not interested in the date, given that each observation is separated by the same interval of one month. Therefore, when you load the dataset, you can exclude the first column.

Once loaded, you can easily plot the whole dataset. The code to load and plot the dataset is listed below.

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 dataset = pd.read_csv('airline-passengers.csv', usecols=[1], engine='python')
4 plt.plot(dataset)
5 plt.show()
```

You can see an upward trend in the plot.

You can also see some periodicity in the dataset that probably corresponds to the northern hemisphere summer holiday period.




Plot of the airline passengers dataset

Let's keep things simple and work with the data as-is.

Normally, it is a good idea to investigate various data preparation techniques to rescale the data and make it stationary.



## Multilayer Perceptron Regression

You want to phrase the time series prediction 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?

You can write a simple function to convert your 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 you get started, let's first import all the functions and classes you will need to use. This assumes a working SciPy environment with the Keras deep learning library installed.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 from tensorflow.keras.models import Sequential
5 from tensorflow.keras.layers import Dense
```

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

```
...
# load the dataset
dataframe = pd.read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
```

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

With time series data, the sequence of values is important. A simple method that you 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 used to train your model, leaving the remaining 33% for testing the model.

```
...
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))
```

Now, you 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 you 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 you will look at constructing a differently shaped dataset in the next section.

```
...
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[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 few rows of the dataset.

```
X       Y
112     118
118     132
132     129
129     121
121     135
```

If you compare these first five 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 modeling.

```
...
# 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)
```

We can now fit a Multilayer Perceptron model to the training data.

We use a simple network with one input, one hidden layer with eight neurons, and an output layer. The model is fit using mean squared error, which, if you take the square root, gives you an error score in the units of the dataset.

I tried a few rough parameters and settled on the configuration below, but by no means is the network listed optimized.

```
...
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(8, input_shape=(look_back,), activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=200, batch_size=2, verbose=2)
```

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

```
...
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))
```

Finally, you 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, you must shift the predictions to 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.

```
...
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
```

```
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Tying this all together, the complete example is listed below.

```
# Multilayer Perceptron to Predict International Airline Passengers (t+1, given t, t-1, t-2)
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)

# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape dataset
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(12, input_shape=(look_back,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=400, batch_size=2, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
```

```
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

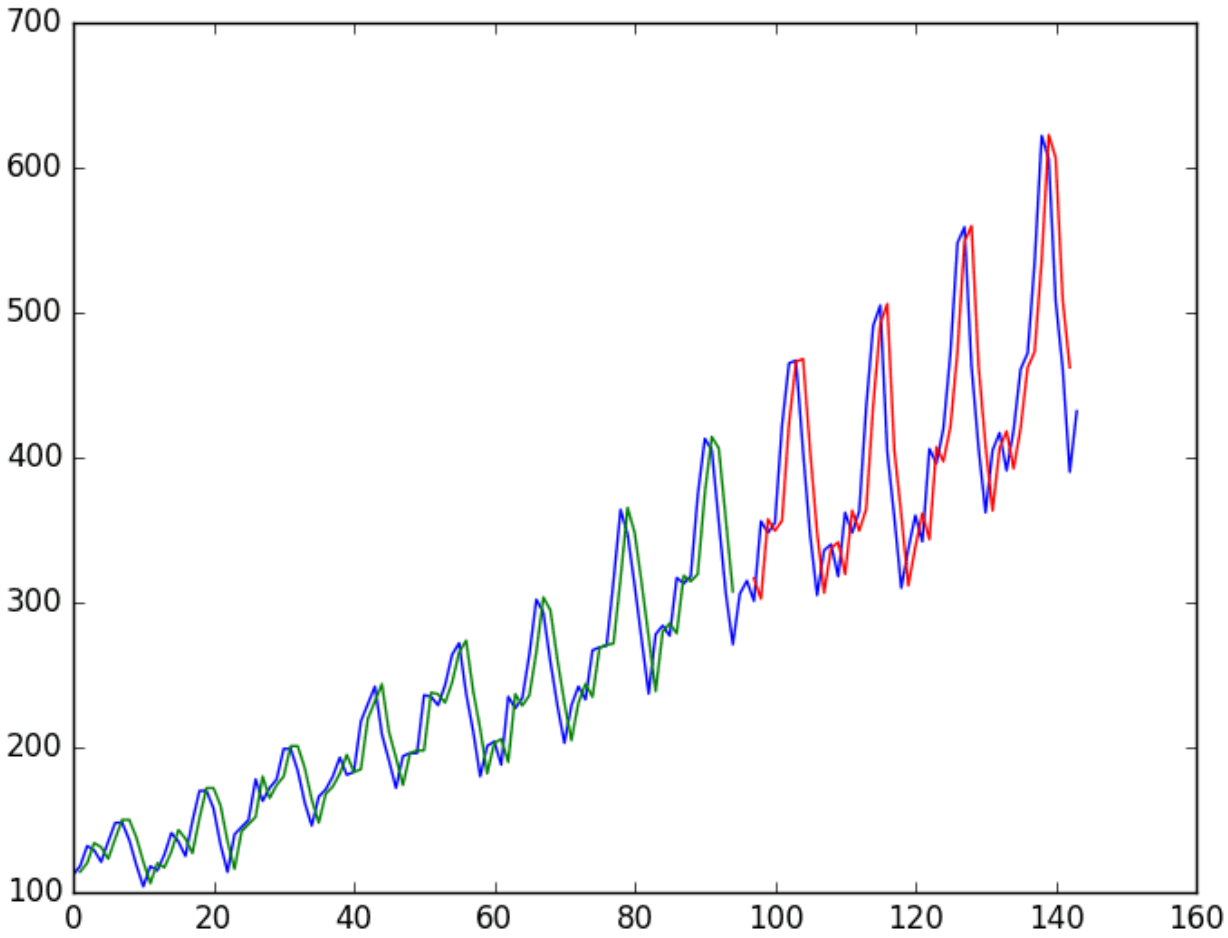
Running the example reports model performance.

**Note**: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Taking the square root of the performance estimates, you can see that the model has an average error of 22 passengers (in thousands) on the training dataset and 46 passengers (in thousands) on the test dataset.

```
...
Epoch 395/400
46/46 - 0s - loss: 513.2617 - 13ms/epoch - 275us/step
Epoch 396/400
46/46 - 0s - loss: 494.1868 - 12ms/epoch - 268us/step
Epoch 397/400
46/46 - 0s - loss: 483.3908 - 12ms/epoch - 268us/step
Epoch 398/400
46/46 - 0s - loss: 501.8111 - 13ms/epoch - 281us/step
Epoch 399/400
46/46 - 0s - loss: 523.2578 - 13ms/epoch - 280us/step
Epoch 400/400
46/46 - 0s - loss: 513.7587 - 12ms/epoch - 263us/step
Train Score: 487.39 MSE (22.08 RMSE)
Test Score: 2070.68 MSE (45.50 RMSE)
```

From the plot, you can see that the model did a pretty poor job of fitting both the training and the test datasets. It basically predicted the same input value as the output.




Naive time series predictions with neural network
Blue=whole dataset, Green=training, Red=predictions

## Multilayer Perceptron Using the Window Method

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

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

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

The **create_dataset()** function used in the previous section allows you to create this formulation of the time series problem by increasing the **look_back** argument from 1 to 3.

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

```
X1  X2  X3  Y
112 118 132 129
118 132 129 121
132 129 121 135
129 121 135 148
121 135 148 148
```

You can re-run the example in the previous section with the larger window size. You will increase the network capacity to handle the additional information. The first hidden layer is increased to 14 neurons, and a second hidden layer is added with eight neurons. The number of epochs is also increased to 400.

The whole code listing with just the window size change is listed below for completeness.

```
# Multilayer Perceptron to Predict International Airline Passengers (t+1, given t, t-1, t-2)
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)

# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape dataset
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# create and fit Multilayer Perceptron model
model = Sequential()
```

```
model.add(Dense(12, input_dim=look_back, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=400, batch_size=2, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

**Note**: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example provides the following output.

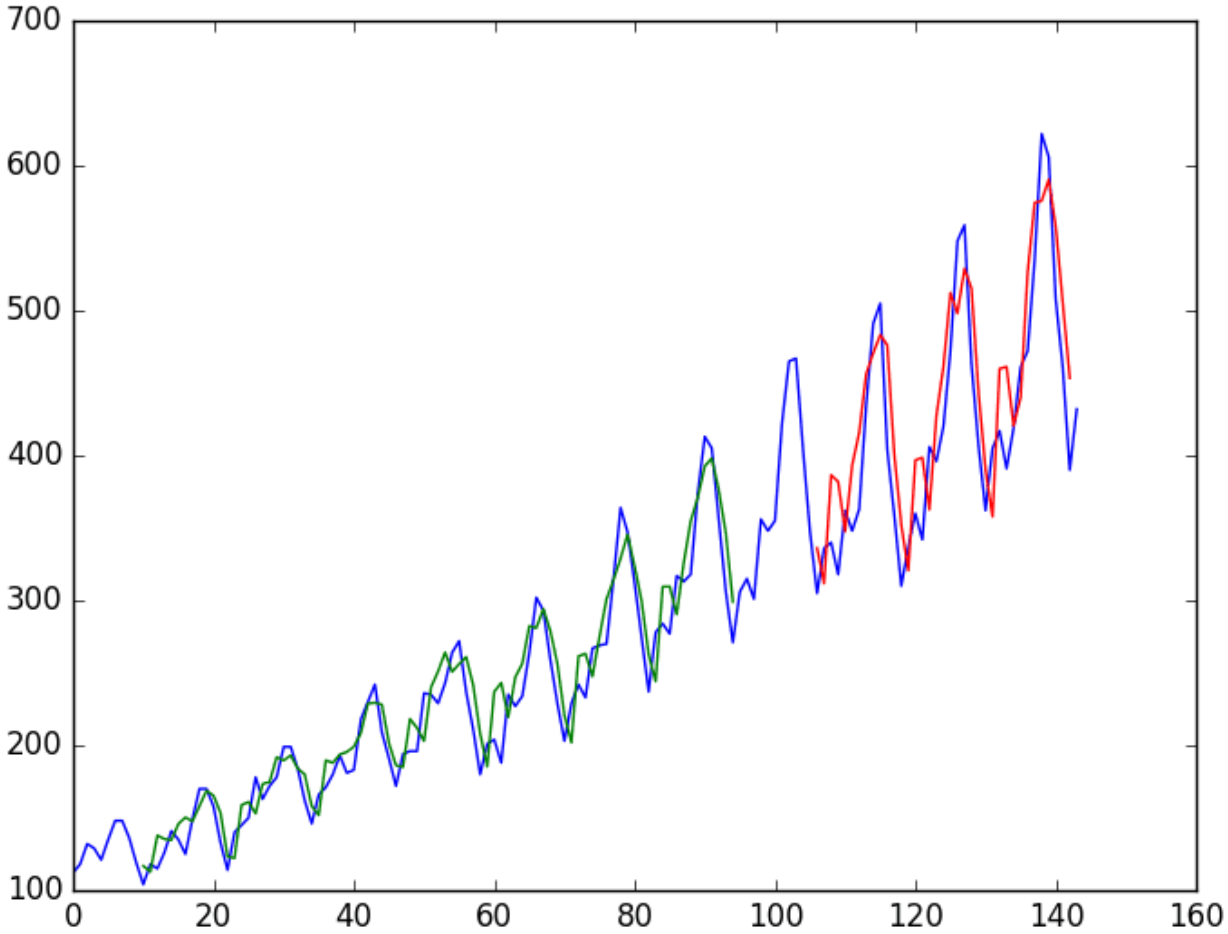
```
Epoch 395/400
46/46 - 0s - loss: 419.0309 - 14ms/epoch - 294us/step
Epoch 396/400
46/46 - 0s - loss: 429.3398 - 14ms/epoch - 300us/step
Epoch 397/400
46/46 - 0s - loss: 412.2588 - 14ms/epoch - 298us/step
Epoch 398/400
46/46 - 0s - loss: 424.6126 - 13ms/epoch - 292us/step
Epoch 399/400
46/46 - 0s - loss: 429.6443 - 14ms/epoch - 296us/step
Epoch 400/400
46/46 - 0s - loss: 419.9067 - 14ms/epoch - 301us/step
Train Score: 393.07 MSE (19.83 RMSE)
Test Score: 1833.35 MSE (42.82 RMSE)
```

You can see that the error was not significantly reduced compared to that of the previous section.

Looking at the graph, you can see more structure in the predictions.

Again, the window size and the network architecture were not tuned; this is just a demonstration of how to frame a prediction problem.

Taking the square root of the performance scores, you can see the average error on the training dataset was 20 passengers (in thousands per month), and the average error on the unseen test set was 43 passengers (in thousands per month).




Window method for time series predictions with neural networks
Blue=whole dataset, Green=training, Red=predictions