Part III

Deep Learning Methods

Overview

This part will show you exactly how to prepare data and develop MLP, CNN and LSTM deep learning models for a range of different time series forecasting problems. The goal of the modeling examples in these chapters is to provide templates that you can copy into your project and start using immediately. As such, you may find some of the examples a little repetitive. After reading the chapters in this part, you will know:

- How to transform time series data into the required three-dimensional structured expected by Convolutional and Long Short-Term Memory Neural Networks, perhaps the most confusing area for beginners (Chapter 6).
- How to develop Multilayer Perceptron models for univariate, multivariate and multi-step time series forecasting problems (Chapter 7).
- How to develop Convolutional Neural Network models for univariate, multivariate and multi-step time series forecasting problems (Chapter 8).
- How to develop Long Short-Term Memory Neural Network models for univariate, multivariate and multi-step time series forecasting problems (Chapter 9).

Chapter 6

How to Prepare Time Series Data for CNNs and LSTMs

Time series data must be transformed before it can be used to fit a supervised learning model. In this form, the data can be used immediately to fit a supervised machine learning algorithm and even a Multilayer Perceptron neural network. One further transformation is required in order to ready the data for fitting a Convolutional Neural Network (CNN) or Long Short-Term Memory (LSTM) Neural Network. Specifically, the two-dimensional structure of the supervised learning data must be transformed to a three-dimensional structure. This is perhaps the largest sticking point for practitioners looking to implement deep learning methods for time series forecasting. In this tutorial, you will discover exactly how to transform a time series data set into a three-dimensional structure ready for fitting a CNN or LSTM model. After completing this tutorial, you will know:

- How to transform a time series dataset into a two-dimensional supervised learning format.
- How to transform a two-dimensional time series dataset into a three-dimensional structure suitable for CNNs and LSTMs.
- How to step through a worked example of splitting a very long time series into subsequences ready for training a CNN or LSTM model.

Let's get started.

6.1 Overview

This tutorial is divided into four parts, they are:

1. Time Series to Supervised.
2. 3D Data Preparation Basics.
3. Univariate Worked Example.

6.2 Time Series to Supervised

Time series data requires preparation before it can be used to train a supervised learning model, such as an LSTM neural network. For example, a univariate time series is represented as a vector of observations:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Listing 6.1: Example of a univariate time series.

A supervised learning algorithm requires that data is provided as a collection of samples, where each sample has an input component (X) and an output component (y).

```
X,                y
sample input,     sample output
sample input,     sample output
sample input,     sample output
...
```

Listing 6.2: Example of a general supervised learning learning problem.

The model will learn how to map inputs to outputs from the provided examples.

$$y = f(X) \tag{6.1}$$

A time series must be transformed into samples with input and output components. The transform both informs what the model will learn and how you intend to use the model in the future when making predictions, e.g. what is required to make a prediction (X) and what prediction is made (y). For a univariate time series problem where we are interested in one-step predictions, the observations at prior time steps, so-called lag observations, are used as input and the output is the observation at the current time step. For example, the above 10-step univariate series can be expressed as a supervised learning problem with three time steps for input and one step as output, as follows:

```
X,           y
[1, 2, 3],   [4]
[2, 3, 4],   [5]
[3, 4, 5],   [6]
...
```

Listing 6.3: Example of a univariate time series converted to supervised learning.

For more on transforming your time series data into a supervised learning problem in general see Chapter 4. You can write code to perform this transform yourself and that is the general approach I teach and recommend for greater understanding of your data and control over the transformation process. The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
```

```
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 6.4: Example of a function to split a univariate series into a supervised learning problem.

For specific examples for univariate, multivariate and multi-step time series, see Chapters 7, 8 and 9. After you have transformed your data into a form suitable for training a supervised learning model it will be represented as rows and columns. Each column will represent a feature to the model and may correspond to a separate lag observation. Each row will represent a sample and will correspond to a new example with input and output components.

- **Feature**: A column in a dataset, such as a lag observation for a time series dataset.
- **Sample**: A row in a dataset, such as an input and output sequence for a time series dataset.

For example, our univariate time series may look as follows:

```
x1, x2, x3, y
 1,  2,  3,  4
 2,  3,  4,  5
 3,  4,  5,  6
...
```

Listing 6.5: Example of a univariate time series in terms of rows and columns.

The dataset will be represented in Python using a NumPy array. The array will have two dimensions. The length of each dimension is referred to as the shape of the array. For example, a time series with 3 inputs, 1 output will be transformed into a supervised learning problem with 4 columns, or really 3 columns for the input data and 1 for the output data. If we have 7 rows and 3 columns for the input data then the shape of the dataset would be `[7, 3]`, or 7 samples and 3 features. We can make this concrete by transforming our small contrived dataset.

```
# transform univariate time series to supervised learning problem
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
```

```
  return array(X), array(y)

# define univariate time series
series = array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(series.shape)
# transform to a supervised learning problem
X, y = split_sequence(series, 3)
print(X.shape, y.shape)
# show each sample
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 6.6: Example of transforming a univariate time series into a supervised learning problem.

Running the example first prints the shape of the time series, in this case 10 time steps of observations. Next, the series is split into input and output components for a supervised learning problem. We can see that for the chosen representation that we have 7 samples for the input and output and 3 input features. The shape of the output is 7 samples represented as `(7,)` indicating that the array is a single column. It could also be represented as a two-dimensional array with 7 rows and 1 column `[7, 1]`. Finally , the input and output aspects of each sample are printed, showing the expected breakdown of the problem.

```
(10,)

(7, 3) (7,)

[1 2 3] 4
[2 3 4] 5
[3 4 5] 6
[4 5 6] 7
[5 6 7] 8
[6 7 8] 9
[7 8 9] 10
```

Listing 6.7: Example output from transforming a univariate time series into a supervised learning problem.

Data in this form can be used directly to train a simple neural network, such as a Multilayer Perceptron. The difficulty for beginners comes when trying to prepare this data for CNNs and LSTMs that require data to have a three-dimensional structure instead of the two-dimensional structure described so far.

6.3 3D Data Preparation Basics

Preparing time series data for CNNs and LSTMs requires one additional step beyond transforming the data into a supervised learning problem. This one additional step causes the most confusion for beginners. In this section we will slowly step through the basics of how and why we need to prepare three-dimensional data for CNNs and LSTMs before working through an example in the next section.

The input layer for CNN and LSTM models is specified by the `input_shape` argument on the first hidden layer of the network. This too can make things confusing for beginners as intuitively we may expect the first layer defined in the model be the input layer, not the first

hidden layer. For example, below is an example of a network with one hidden `LSTM` layer and one `Dense` output layer.

```
# lstm without an input layer
...
model = Sequential()
model.add(LSTM(32))
model.add(Dense(1))
```

Listing 6.8: Example of defining an LSTM model without an input layer.

In this example, the `LSTM()` layer must specify the shape of the input data. The input to every CNN and LSTM layer must be three-dimensional. The three dimensions of this input are:

- **Samples**. One sequence is one sample. A batch is comprised of one or more samples.
- **Time Steps**. One time step is one point of observation in the sample. One sample is comprised of multiple time steps.
- **Features**. One feature is one observation at a time step. One time step is comprised of one or more features.

This expected three-dimensional structure of input data is often summarized using the array shape notation of: `[samples, timesteps, features]`. Remember, that the two-dimensional shape of a dataset that we are familiar with from the previous section has the array shape of: `[samples, features]`. this means we are adding the new dimension of *time steps*. Except, in time series forecasting problems our features are observations at time steps. So, really, we are adding the dimension of *features*, where a univariate time series has only one feature.

When defining the input layer of your LSTM network, the network assumes you have one or more samples and requires that you specify the number of time steps and the number of features. You can do this by specifying a tuple to the `input_shape` argument. For example, the model below defines an input layer that expects 1 or more samples, 3 time steps, and 1 feature. Remember, the first layer in the network is actually the first hidden layer, so in this example 32 refers to the number of units in the first hidden layer. The number of units in the first hidden layer is completely unrelated to the number of samples, time steps or features in your input data.

```
# lstm with an input layer
...
model = Sequential()
model.add(LSTM(32, input_shape=(3, 1)))
model.add(Dense(1))
```

Listing 6.9: Example of defining an LSTM model with an input layer.

This example maps onto our univariate time series from the previous section that we split into having 3 input time steps and 1 feature. We may have loaded our time series dataset from CSV or transformed it to a supervised learning problem in memory. It will have a two-dimensional shape and we must convert it to a three-dimensional shape with some number of samples, 3 time steps per sample and 1 feature per time step, or `[?, 3, 1]`. We can do this by using the `reshape()` NumPy function. For example, if we have 7 samples and 3 time steps per sample for the input element of our time series, we can reshape it into `[7, 3, 1]` by providing a tuple to

the `reshape()` function specifying the desired new shape of `(7, 3, 1)`. The array must have enough data to support the new shape, which in this case it does as `[7, 3]` and `[7, 3, 1]` are functionally the same thing.

```
...
# transform input from [samples, features] to [samples, timesteps, features]
X = X.reshape((7, 3, 1))
```

Listing 6.10: Example of reshaping 2D data to be 3D.

A short-cut in reshaping the array is to use the known shapes, such as the number of samples and the number of times steps from the array returned from the call to the `X.shape` property of the array. For example, `X.shape[0]` refers to the number of rows in a 2D array, in this case the number of samples and `X.shape[1]` refers to the number of columns in a 2D array, in this case the number of feature that we will use as the number of time steps. The reshape can therefore be written as:

```
...
# transform input from [samples, features] to [samples, timesteps, features]
X = X.reshape((X.shape[0], X.shape[1], 1))
```

Listing 6.11: Example of reshaping 2D data to be 3D.

We can make this concept concrete with a worked example. The complete code listing is provided below.

```
# transform univariate 2d to 3d
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define univariate time series
series = array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(series.shape)
# transform to a supervised learning problem
X, y = split_sequence(series, 3)
print(X.shape, y.shape)
# transform input from [samples, features] to [samples, timesteps, features]
X = X.reshape((X.shape[0], X.shape[1], 1))
print(X.shape)
```

Listing 6.12: Example of transforming a univariate time series into a three-dimensional array.

Running the example first prints the shape of the univariate time series, in this case 10 time steps. It then summarizes the shape if the input (X) and output (y) elements of each sample after the univariate series has been converted into a supervised learning problem, in this case, the data has 7 samples and the input data has 3 features per sample, which we know are actually time steps. Finally, the input element of each sample is reshaped to be three-dimensional suitable for fitting an LSTM or CNN and now has the shape `[7, 3, 1]` or 7 samples, 3 time steps, 1 feature.

```
(10,)
(7, 3) (7,)
(7, 3, 1)
```

Listing 6.13: Example output from reshaping 2D data to be 3D.

6.4 Data Preparation Example

Consider that you are in the current situation:

> *I have two columns in my data file with 5,000 rows, column 1 is time (with 1 hour interval) and column 2 is the number of sales and I am trying to forecast the number of sales for future time steps. Help me to set the number of samples, time steps and features in this data for an LSTM?*

There are few problems here:

- **Data Shape**. LSTMs expect 3D input, and it can be challenging to get your head around this the first time.
- **Sequence Length**. LSTMs don't like sequences of more than 200-400 time steps, so the data will need to be split into subsamples.

We will work through this example, broken down into the following 4 steps:

1. Load the Data
2. Drop the Time Column
3. Split Into Samples
4. Reshape Subsequences

6.4.1 Load the Data

We can load this dataset as a Pandas `Series` using the function `read_csv()`.

```
# load time series dataset
series = read_csv('filename.csv', header=0, index_col=0)
```

Listing 6.14: Example of loading a dataset as a Pandas `DataFrame`.

For this example, we will mock loading by defining a new dataset in memory with 5,000 time steps.

```
# example of defining a dataset
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
print(data[:5, :])
print(data.shape)
```

Listing 6.15: Example of defining the dataset instead of loading it.

Running this piece both prints the first 5 rows of data and the shape of the loaded data. We can see we have 5,000 rows and 2 columns: a standard univariate time series dataset.

```
[[ 1 10]
 [ 2 20]
 [ 3 30]
 [ 4 40]
 [ 5 50]]
(5000, 2)
```

Listing 6.16: Example output from defining the dataset.

6.4.2 Drop the Time Column

If your time series data is uniform over time and there is no missing values, we can drop the time column. If not, you may want to look at imputing the missing values, resampling the data to a new time scale, or developing a model that can handle missing values. Here, we just drop the first column:

```
# example of dropping the time dimension from the dataset
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
# drop time
data = data[:, 1]
print(data.shape)
```

Listing 6.17: Example of dropping the time column.

Running the example prints the shape of the dataset after the time column has been removed.

```
(5000,)
```

Listing 6.18: Example output from dropping the time column.

6.4.3 Split Into Samples

LSTMs need to process samples where each sample is a single sequence of observations. In this case, 5,000 time steps is too long; LSTMs work better with 200-to-400 time steps. Therefore, we need to split the 5,000 time steps into multiple shorter sub-sequences. There are many ways to do this, and you may want to explore some depending on your problem. For example, perhaps you need overlapping sequences, perhaps non-overlapping is good but your model needs state across the sub-sequences and so on. In this example, we will split the 5,000 time steps into 25 sub-sequences of 200 time steps each. Rather than using NumPy or Python tricks, we will do this the old fashioned way so you can see what is going on.

```
# example of splitting a univariate sequence into subsequences
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
# drop time
data = data[:, 1]
# split into samples (e.g. 5000/200 = 25)
samples = list()
length = 200
# step over the 5,000 in jumps of 200
for i in range(0,n,length):
  # grab from i to i + 200
  sample = data[i:i+length]
  samples.append(sample)
print(len(samples))
```

Listing 6.19: Example of splitting the series into samples.

We now have 25 subsequences of 200 time steps each.

```
25
```

Listing 6.20: Example output from splitting the series into samples.

6.4.4 Reshape Subsequences

The LSTM needs data with the format of `[samples, timesteps, features]`. We have 25 samples, 200 time steps per sample, and 1 feature. First, we need to convert our list of arrays into a 2D NumPy array with the shape `[25, 200]`.

```
# example of creating an array of subsequence
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
```

```
# drop time
data = data[:, 1]
# split into samples (e.g. 5000/200 = 25)
samples = list()
length = 200
# step over the 5,000 in jumps of 200
for i in range(0,n,length):
  # grab from i to i + 200
  sample = data[i:i+length]
  samples.append(sample)
# convert list of arrays into 2d array
data = array(samples)
print(data.shape)
```

Listing 6.21: Example of printing the shape of the samples.

Running this piece, you should see that we have 25 rows and 200 columns. Interpreted in a machine learning context, this dataset has 25 samples and 200 features per sample.

```
(25, 200)
```

Listing 6.22: Example output from printing the shape of the samples.

Next, we can use the `reshape()` function to add one additional dimension for our single feature and use the existing columns as time steps instead.

```
# example of creating a 3d array of subsequences
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
# drop time
data = data[:, 1]
# split into samples (e.g. 5000/200 = 25)
samples = list()
length = 200
# step over the 5,000 in jumps of 200
for i in range(0,n,length):
  # grab from i to i + 200
  sample = data[i:i+length]
  samples.append(sample)
# convert list of arrays into 2d array
data = array(samples)
# reshape into [samples, timesteps, features]
data = data.reshape((len(samples), length, 1))
print(data.shape)
```

Listing 6.23: Example of reshaping the dataset into a 3D format.

And that is it. The data can now be used as an input (X) to an LSTM model, or even a CNN model.

```
(25, 200, 1)
```

Listing 6.24: Example output from reshaping the dataset into a 3D format.

6.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Explain Data Shape**. Explain in your own words the meaning of samples, time steps and features.
- **Worked Example**. Select a standard time series forecasting problem and manually reshape it into a structure suitable for training a CNN or LSTM model.
- **Develop Framework**. Develop a function to automatically reshape a time series dataset into samples and into a shape suitable for training a CNN or LSTM model.

If you explore any of these extensions, I'd love to know.

6.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- `numpy.reshape` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html
- Keras Recurrent Layers API in Keras.
  https://keras.io/layers/recurrent/
- Keras Convolutional Layers API in Keras.
  https://keras.io/layers/convolutional/

6.7 Summary

In this tutorial, you discovered exactly how to transform a time series data set into a three-dimensional structure ready for fitting a CNN or LSTM model.

Specifically, you learned:

- How to transform a time series dataset into a two-dimensional supervised learning format.
- How to transform a two-dimensional time series dataset into a three-dimensional structure suitable for CNNs and LSTMs.
- How to step through a worked example of splitting a very long time series into subsequences ready for training a CNN or LSTM model.

6.7.1 Next

In the next lesson, you will discover how to develop Multilayer Perceptron models for time series forecasting.

Chapter 7

How to Develop MLPs for Time Series Forecasting

Multilayer Perceptrons, or MLPs for short, can be applied to time series forecasting. A challenge with using MLPs for time series forecasting is in the preparation of the data. Specifically, lag observations must be flattened into feature vectors. In this tutorial, you will discover how to develop a suite of MLP models for a range of standard time series forecasting problems. The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem. In this tutorial, you will discover how to develop a suite of Multilayer Perceptron models for a range of standard time series forecasting problems. After completing this tutorial, you will know:

- How to develop MLP models for univariate time series forecasting.
- How to develop MLP models for multivariate time series forecasting.
- How to develop MLP models for multi-step time series forecasting.

Let's get started.

7.1 Tutorial Overview

In this tutorial, we will explore how to develop a suite of MLP models for time series forecasting. The models are demonstrated on small contrived time series problems intended to give the flavor of the type of time series problem being addressed. The chosen configuration of the models is arbitrary and not optimized for each problem; that was not the goal. This tutorial is divided into four parts; they are:

1. Univariate MLP Models
2. Multivariate MLP Models
3. Multi-step MLP Models
4. Multivariate Multi-step MLP Models

Note: Traditionally, a lot of research has been invested into using MLPs for time series forecasting with modest results. Perhaps the most promising area in the application of deep learning methods to time series forecasting are in the use of CNNs, LSTMs and hybrid models. As such, we will not see more examples of straight MLP models for time series forecasting beyond this tutorial.

7.2 Univariate MLP Models

Multilayer Perceptrons, or MLPs for short, can be used to model univariate time series forecasting problems. Univariate time series are a dataset comprised of a single series of observations with a temporal ordering and a model is required to learn from the series of past observations to predict the next value in the sequence. This section is divided into two parts; they are:

1. Data Preparation
2. MLP Model

7.2.1 Data Preparation

Before a univariate series can be modeled, it must be prepared. The MLP model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the model can learn. Consider a given univariate sequence:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 7.1: Example of a univariate time series.

We can divide the sequence into multiple input/output patterns called samples, where three time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X,              y
10, 20, 30,     40
20, 30, 40,     50
30, 40, 50,     60
...
```

Listing 7.2: Example of a univariate time series as a supervised learning problem.

The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
```

```
    break
  # gather input and output parts of the pattern
  seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
  X.append(seq_x)
  y.append(seq_y)
return array(X), array(y)
```

Listing 7.3: Example of a function to split a univariate series into a supervised learning problem.

We can demonstrate this function on our small contrived dataset above. The complete example is listed below.

```
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.4: Example of transforming a univariate time series into a supervised learning problem.

Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

Listing 7.5: Example output from transforming a univariate time series into a supervised learning problem.

Now that we know how to prepare a univariate series for modeling, let's look at developing an MLP model that can learn the mapping of inputs to outputs.

7.2.2 MLP Model

A simple MLP model has a single hidden layer of nodes, and an output layer used to make a prediction. We can define an MLP for univariate time series forecasting as follows.

```
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 7.6: Example of defining an MLP model.

Important in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps. The number of time steps as input is the number we chose when preparing our dataset as an argument to the `split_sequence()` function. The input dimension for each sample is specified in the `input_dim` argument on the definition of first hidden layer. Technically, the model will view each time step as a separate feature instead of separate time steps.

We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape: `[samples, features]`. Our `split_sequence()` function in the previous section outputs the X with the shape `[samples, features]` ready to use for modeling. The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or `'mse'`, loss function. Once the model is defined, we can fit it on the training dataset.

```
# fit model
model.fit(X, y, epochs=2000, verbose=0)
```

Listing 7.7: Example of fitting an MLP model.

After the model is fit, we can use it to make a prediction. We can predict the next value in the sequence by providing the input: `[70, 80, 90]`. And expecting the model to predict something like: `[100]`. The model expects the input shape to be two-dimensional with `[samples, features]`, therefore, we must reshape the single input sample before making the prediction, e.g with the shape `[1, 3]` for 1 sample and 3 time steps used as input features.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps))
yhat = model.predict(x_input, verbose=0)
```

Listing 7.8: Example of reshaping a single sample for making a prediction.

We can tie all of this together and demonstrate how to develop an MLP for univariate time series forecasting and make a single prediction.

```
# univariate mlp example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
```

```
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.9: Example of demonstrating an MLP for univariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction. We can see that the model predicts the next value in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[100.0109]]
```

Listing 7.10: Example output from demonstrating an MLP for univariate time series forecasting.

For an example of an MLP applied to a real-world univariate time series forecasting problem see Chapter 14. For an example of grid searching MLP hyperparameters on a univariate time series forecasting problem, see Chapter 15.

7.3 Multivariate MLP Models

Multivariate time series data means data where there is more than one observation for each time step. There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.
2. Multiple Parallel Series.

Let's take a look at each in turn.

7.3.1 Multiple Input Series

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series. The input time series are parallel because each series has an observation at the same time step. We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

Listing 7.11: Example of defining a parallel time series.

We can reshape these three arrays of data as a single dataset where each row is a time step and each column is a separate time series. This is a standard way of storing parallel time series in a CSV file.

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

Listing 7.12: Example of horizontally stacking parallel series into a dataset.

The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
print(dataset)
```

Listing 7.13: Example of parallel dependent time series.

Running the example prints the dataset with one row per time step and one column for each of the two input and one output parallel time series.

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 7.14: Example output from parallel dependent time series.

As with the univariate time series, we must structure these data into samples with input and output samples. We need to split the data into samples maintaining the order of observations across the two input sequences. If we chose three input time steps, then the first sample would look as follows:

Input:

```
10, 15
20, 25
30, 35
```

Listing 7.15: Example input data for the first data sample.

Output:

```
65
```

Listing 7.16: Example output data for the first data sample.

That is, the first three time steps of each parallel series are provided as input to the model and the model associates this with the value in the output series at the third time step, in this case 65. We can see that, in transforming the time series into input/output samples to train the model, that we will have to discard some values from the output time series where we do not have values in the input time series at prior time steps. In turn, the choice of the size of the number of input time steps will have an important effect on how much of the training data is used. We can define a function named `split_sequences()` that will take a dataset as we have defined it with rows for time steps and columns for parallel series and return input/output samples.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 7.17: Example of a function for separating parallel time series into a supervised learning problem.

We can test this function on our dataset using three time steps for each input time series as input. The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.18: Example of transforming a parallel dependent time series into a supervised learning problem.

Running the example first prints the shape of the X and y components. We can see that the X component has a three-dimensional structure. The first dimension is the number of samples, in this case 7. The second dimension is the number of time steps per sample, in this case 3, the value specified to the function. Finally, the last dimension specifies the number of parallel time series or the number of variables, in this case 2 for the two parallel series. We can then see that the input and output for each sample is printed, showing the three time steps for each of the two input series and the associated output for each sample.

```
(7, 3, 2) (7,)

[[10 15]
 [20 25]
 [30 35]] 65
[[20 25]
 [30 35]
 [40 45]] 85
[[30 35]
 [40 45]
 [50 55]] 105
[[40 45]
 [50 55]
 [60 65]] 125
[[50 55]
 [60 65]
 [70 75]] 145
[[60 65]
 [70 75]
 [80 85]] 165
[[70 75]
 [80 85]
 [90 95]] 185
```

Listing 7.19: Example output from transforming a parallel dependent time series into a supervised learning problem.

MLP Model

Before we can fit an MLP on this data, we must flatten the shape of the input samples. MLPs require that the shape of the input portion of each sample is a vector. With a multivariate input, we will have multiple vectors, one for each time step. We can flatten the temporal structure of each input sample, so that:

```
[[10 15]
[20 25]
[30 35]]
```

Listing 7.20: Example input data for the first data sample.

Becomes:

```
[10, 15, 20, 25, 30, 35]
```

Listing 7.21: Example of a flattened input data for the first data sample.

First, we can calculate the length of each input vector as the number of time steps multiplied by the number of features or time series. We can then use this vector size to reshape the input.

```
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
```

Listing 7.22: Example of flattening input samples for the MLP.

We can now define an MLP model for the multivariate input where the vector length is used for the input dimension argument.

```
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 7.23: Example of defining an MLP that expects a flattened multivariate time series as input.

When making a prediction, the model expects three time steps for two input time series. We can predict the next value in the output series proving the input values of:

```
80, 85
90, 95
100, 105
```

Listing 7.24: Example input sample for making a prediction beyond the end of the time series.

The shape of the 1 sample with 3 time steps and 2 variables would be `[1, 3, 2]`. We must again reshape this to be 1 sample with a vector of 6 elements or `[1, 6]`. We would expect the next value in the sequence to be 100 + 105 or 205.

```
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
```

Listing 7.25: Example of reshaping the input sample for making a prediction.

The complete example is listed below.

```
# multivariate mlp example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
```

```
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.26: Example of using an MLP to forecast a dependent time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[205.04436]]
```

Listing 7.27: Example output from using an MLP to forecast a dependent time series.

Multi-headed MLP Model

There is another more elaborate way to model the problem. Each input series can be handled by a separate MLP and the output of each of these submodels can be combined before a prediction is made for the output sequence. We can refer to this as a multi-headed input MLP model. It may offer more flexibility or better performance depending on the specifics of the problem that are being modeled. This type of model can be defined in Keras using the Keras functional API. First, we can define the first input model as an MLP with an input layer that expects vectors with `n_steps` features.

```
# first input model
visible1 = Input(shape=(n_steps,))
dense1 = Dense(100, activation='relu')(visible1)
```

Listing 7.28: Example of defining the first input model.

We can define the second input submodel in the same way.

```
# second input model
visible2 = Input(shape=(n_steps,))
dense2 = Dense(100, activation='relu')(visible2)
```

Listing 7.29: Example of defining the second input model.

Now that both input submodels have been defined, we can merge the output from each model into one long vector, which can be interpreted before making a prediction for the output sequence.

```
# merge input models
merge = concatenate([dense1, dense2])
output = Dense(1)(merge)
```

Listing 7.30: Example of merging the two input models.

We can then tie the inputs and outputs together.

```
# connect input and output models
model = Model(inputs=[visible1, visible2], outputs=output)
```

Listing 7.31: Example of connecting the input and output elements together.

The image below provides a schematic for how this model looks, including the shape of the inputs and outputs of each layer.

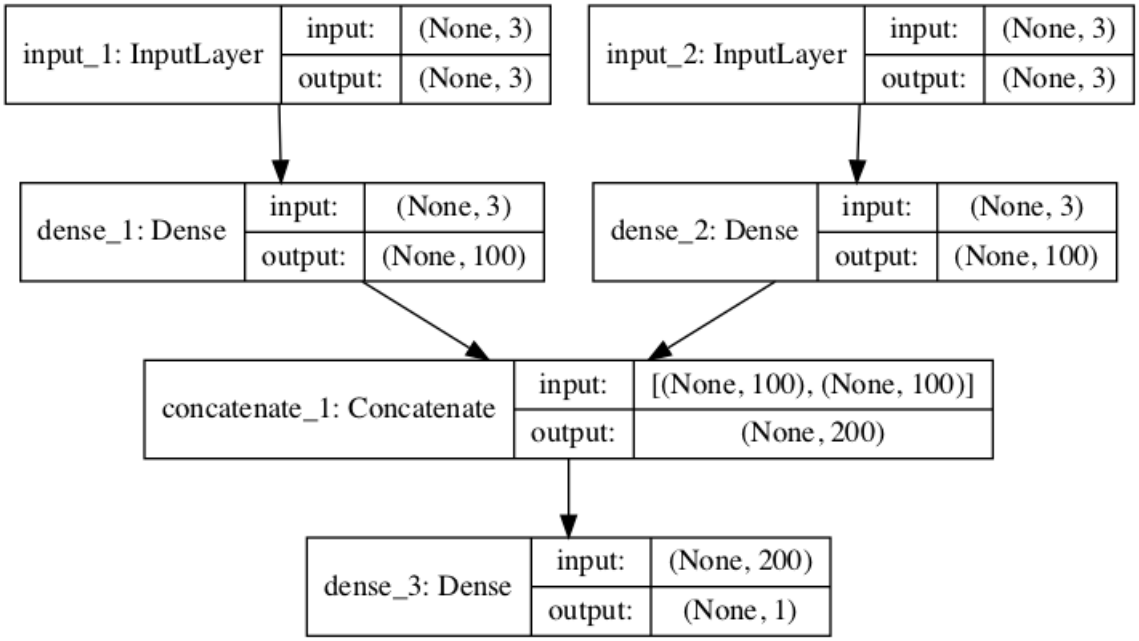

Figure 7.1: Plot of Multi-headed MLP for Multivariate Time Series Forecasting.

This model requires input to be provided as a list of two elements, where each element in the list contains data for one of the submodels. In order to achieve this, we can split the 3D input data into two separate arrays of input data: that is from one array with the shape `[7, 3, 2]` to two 2D arrays with the shape `[7, 3]`.

```
# separate input data
X1 = X[:, :, 0]
X2 = X[:, :, 1]
```

Listing 7.32: Example of separating input data for the two input models.

These data can then be provided in order to fit the model.

```
# fit model
model.fit([X1, X2], y, epochs=2000, verbose=0)
```

Listing 7.33: Example of fitting the multi-headed input model.

Similarly, we must prepare the data for a single sample as two separate two-dimensional arrays when making a single one-step prediction.

```
# reshape one sample for making a forecast
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps))
x2 = x_input[:, 1].reshape((1, n_steps))
```

Listing 7.34: Example of preparing an input sample for making a forecast.

We can tie all of this together; the complete example is listed below.

```
# multivariate mlp example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers.merge import concatenate

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
```

```
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# separate input data
X1 = X[:, :, 0]
X2 = X[:, :, 1]
# first input model
visible1 = Input(shape=(n_steps,))
dense1 = Dense(100, activation='relu')(visible1)
# second input model
visible2 = Input(shape=(n_steps,))
dense2 = Dense(100, activation='relu')(visible2)
# merge input models
merge = concatenate([dense1, dense2])
output = Dense(1)(merge)
model = Model(inputs=[visible1, visible2], outputs=output)
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit([X1, X2], y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps))
x2 = x_input[:, 1].reshape((1, n_steps))
yhat = model.predict([x1, x2], verbose=0)
print(yhat)
```

Listing 7.35: Example of using a Multi-headed MLP to forecast a dependent time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[206.05022]]
```

Listing 7.36: Example output from using a Multi-headed MLP to forecast a dependent time series.

7.3.2 Multiple Parallel Series

An alternate time series problem is the case where there are multiple parallel time series and a value must be predicted for each. For example, given the data from the previous section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 7.37: Example of a parallel time series problem.

We may want to predict the value for each of the three time series for the next time step. This might be referred to as multivariate forecasting. Again, the data must be split into input/output samples in order to train a model. The first sample of this dataset would be:

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 7.38: Example input from the first data sample.

Output:

```
40, 45, 85
```

Listing 7.39: Example output from the first data sample.

The `split_sequences()` function below will split multiple parallel time series with rows for time steps and one series per column into the required input/output shape.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 7.40: Example of a function for splitting a multivariate time series dataset into a supervised learning problem.

We can demonstrate this on the contrived problem; the complete example is listed below.

```
# multivariate output data prep
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
```

```
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.41: Example of splitting a multivariate time series into a supervised learning problem.

Running the example first prints the shape of the prepared X and y components. The shape of X is three-dimensional, including the number of samples (6), the number of time steps chosen per sample (3), and the number of parallel time series or features (3). The shape of y is two-dimensional as we might expect for the number of samples (6) and the number of time variables per sample to be predicted (3). Then, each of the samples is printed showing the input and output components of each sample.

```
(6, 3, 3) (6, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [40 45 85]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [ 50 55 105]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [ 60 65 125]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [ 70 75 145]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [ 80 85 165]
[[ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]] [ 90 95 185]
```

Listing 7.42: Example output from splitting a multivariate time series into a supervised learning problem.

Vector-Output MLP Model

We are now ready to fit an MLP model on this data. As with the previous case of multivariate input, we must flatten the three dimensional structure of the input data samples to a two dimensional structure of `[samples, features]`, where lag observations are treated as features by the model.

```
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
```

Listing 7.43: Example of flattening multivariate time series for input to a MLP.

The model output will be a vector, with one element for each of the three different time series.

```
# determine the number of outputs
n_output = y.shape[1]
```

Listing 7.44: Example of defining the size of the vector to forecast.

We can now define our model, using the flattened vector length for the input layer and the number of time series as the vector length when making a prediction.

```
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
```

Listing 7.45: Example of defining a MLP for multivariate forecasting.

We can predict the next value in each of the three parallel series by providing an input of three time steps for each series.

```
70, 75, 145
80, 85, 165
90, 95, 185
```

Listing 7.46: Example input for making an out-of-sample forecast.

The shape of the input for making a single prediction must be 1 sample, 3 time steps and 3 features, or `[1, 3, 3]`. Again, we can flatten this to `[1, 6]` to meet the expectations of the model. We would expect the vector output to be:

```
[100, 105, 205]
```

Listing 7.47: Example output for an out-of-sample forecast.

```
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
```

Listing 7.48: Example of preparing data for making an out-of-sample forecast with a MLP.

We can tie all of this together and demonstrate an MLP for multivariate output time series forecasting below.

```
# multivariate output mlp example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
n_output = y.shape[1]
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.49: Example of an MLP for multivariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[100.95039 107.541306 206.81033 ]]
```

Listing 7.50: Example output from an MLP for multivariate time series forecasting.

Multi-output MLP Model

As with multiple input series, there is another, more elaborate way to model the problem. Each output series can be handled by a separate output MLP model. We can refer to this as a multi-output MLP model. It may offer more flexibility or better performance depending on the specifics of the problem that is being modeled. This type of model can be defined in Keras using the Keras functional API. First, we can define the input model as an MLP with an input layer that expects flattened feature vectors.

```
# define model
visible = Input(shape=(n_input,))
dense = Dense(100, activation='relu')(visible)
```

Listing 7.51: Example of defining an input model.

We can then define one output layer for each of the three series that we wish to forecast, where each output submodel will forecast a single time step.

```
# define output 1
output1 = Dense(1)(dense)
# define output 2
output2 = Dense(1)(dense)
# define output 2
output3 = Dense(1)(dense)
```

Listing 7.52: Example of defining multiple output models.

We can then tie the input and output layers together into a single model.

```
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
```

Listing 7.53: Example of connecting input and output models.

To make the model architecture clear, the schematic below clearly shows the three separate output layers of the model and the input and output shapes of each layer.

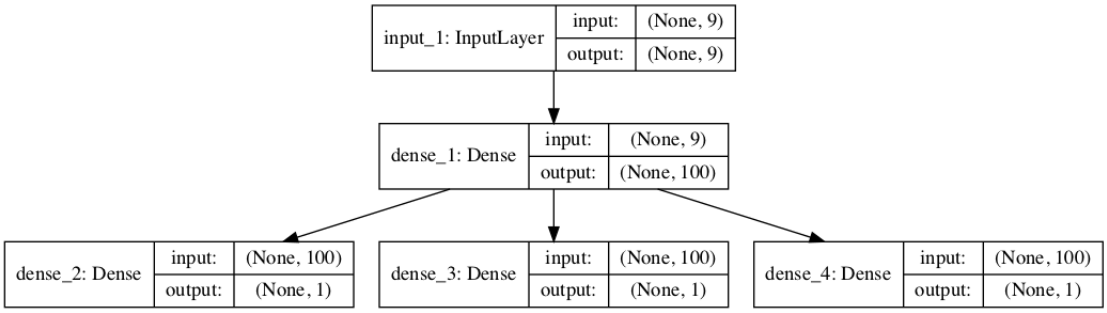

Figure 7.2: Plot of Multi-output MLP for Multivariate Time Series Forecasting.

When training the model, it will require three separate output arrays per sample. We can achieve this by converting the output training data that has the shape `[7, 3]` to three arrays with the shape `[7, 1]`.

```
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
```

Listing 7.54: Example of splitting output data for the multi-output model.

These arrays can be provided to the model during training.

```
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
```

Listing 7.55: Example of fitting the multi-output MLP model.

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

```
# multivariate output mlp example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# flatten input
n_input = X.shape[1] * X.shape[2]
```

```
X = X.reshape((X.shape[0], n_input))
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
# define model
visible = Input(shape=(n_input,))
dense = Dense(100, activation='relu')(visible)
# define output 1
output1 = Dense(1)(dense)
# define output 2
output2 = Dense(1)(dense)
# define output 2
output3 = Dense(1)(dense)
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.56: Example of a multi-output MLP for multivariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[array([[100.86121]], dtype=float32),
array([[105.14738]], dtype=float32),
array([[205.97507]], dtype=float32)]
```

Listing 7.57: Example output from a multi-output MLP for multivariate time series forecasting.

7.4 Multi-step MLP Models

In practice, there is little difference to the MLP model in predicting a vector output that represents different output variables (as in the previous example) or a vector output that represents multiple time steps of one variable. Nevertheless, there are subtle and important differences in the way the training data is prepared. In this section, we will demonstrate the case of developing a multi-step forecast model using a vector model. Before we look at the specifics of the model, let's first look at the preparation of data for multi-step forecasting.

7.4.1 Data Preparation

As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components. Both the input and output components

will be comprised of multiple time steps and may or may not have the same number of steps. For example, given the univariate time series:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 7.58: Example of a univariate time series.

We could use the last three time steps as input and forecast the next two time steps. The first sample would look as follows:

Input:

```
[10, 20, 30]
```

Listing 7.59: Example input for a multi-step forecast.

Output:

```
[40, 50]
```

Listing 7.60: Example output for a multi-step forecast.

The `split_sequence()` function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 7.61: Example of a function for preparing a univariate time series for multi-step forecasting.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multi-step data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
```

```
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.62: Example of data preparation for multi-step time series forecasting.

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

Listing 7.63: Example output from data preparation for multi-step time series forecasting.

Now that we know how to prepare data for multi-step forecasting, let's look at an MLP model that can learn this mapping.

7.4.2 Vector Output Model

The MLP can output a vector directly that can be interpreted as a multi-step forecast. This approach was seen in the previous section were one time step of each output time series was forecasted as a vector. With the number of input and output steps specified in the `n_steps_in` and `n_steps_out` variables, we can define a multi-step time-series forecasting model.

```
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps_in))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
```

Listing 7.64: Example of preparing an MLP mode for multi-step time series forecasting.

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input:

```
[70, 80, 90]
```

Listing 7.65: Example input for making an out-of-sample multi-step forecast.

We would expect the predicted output to be:

```
[100, 110]
```

Listing 7.66: Example output for an out-of-sample multi-step forecast.

As expected by the model, the shape of the single sample of input data when making the prediction must be `[1, 3]` for the 1 sample and 3 time steps (features) of the input and the single feature.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in))
yhat = model.predict(x_input, verbose=0)
```

Listing 7.67: Example of preparing data for making an out-of-sample multi-step forecast.

Tying all of this together, the MLP for multi-step forecasting with a univariate time series is listed below.

```
# univariate multi-step vector-output mlp example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps_in))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in))
yhat = model.predict(x_input, verbose=0)
```

```
print(yhat)
```

Listing 7.68: Example of an MLP for multi-step time series forecasting.

Running the example forecasts and prints the next two time steps in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.572365 113.88405 ]]
```

Listing 7.69: Example output from an MLP for multi-step time series forecasting.

7.5 Multivariate Multi-step MLP Models

In the previous sections, we have looked at univariate, multivariate, and multi-step time series forecasting. It is possible to mix and match the different types of MLP models presented so far for the different problems. This too applies to time series forecasting problems that involve multivariate and multi-step forecasting, but it may be a little more challenging, particularly in preparing the data and defining the shape of inputs and outputs for the model. In this section, we will look at short examples of data preparation and modeling for multivariate multi-step time series forecasting as a template to ease this challenge, specifically:

1. Multiple Input Multi-step Output.
2. Multiple Parallel Input and Multi-step Output.

Perhaps the biggest stumbling block is in the preparation of data, so this is where we will focus our attention.

7.5.1 Multiple Input Multi-step Output

There are those multivariate time series forecasting problems where the output series is separate but dependent upon the input time series, and multiple time steps are required for the output series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 7.70: Example of a multivariate time series with a dependent series.

We may use three prior time steps of each of the two input time series to predict two time steps of the output time series.

Input:

```
10, 15
20, 25
30, 35
```

Listing 7.71: Example input for a multi-step forecast for a dependent series.

Output:

```
65
85
```

Listing 7.72: Example output for a multi-step forecast for a dependent series.

The `split_sequences()` function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 7.73: Example of a function for preparing data for a multi-step dependent series.

We can demonstrate this on our contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
```

```
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.74: Example of preparing data for multi-step forecasting for a dependent series.

Running the example first prints the shape of the prepared training data. We can see that the shape of the input portion of the samples is three-dimensional, comprised of six samples, with three time steps and two variables for the two input time series. The output portion of the samples is two-dimensional for the six samples and the two time steps for each sample to be predicted. The prepared samples are then printed to confirm that the data was prepared as we specified.

```
(6, 3, 2) (6, 2)

[[10 15]
 [20 25]
 [30 35]] [65 85]
[[20 25]
 [30 35]
 [40 45]] [ 85 105]
[[30 35]
 [40 45]
 [50 55]] [105 125]
[[40 45]
 [50 55]
 [60 65]] [125 145]
[[50 55]
 [60 65]
 [70 75]] [145 165]
[[60 65]
 [70 75]
 [80 85]] [165 185]
```

Listing 7.75: Example output from preparing data for multi-step forecasting for a dependent series.

We can now develop an MLP model for multi-step predictions using a vector output. The complete example is listed below.

```
# multivariate multi-step mlp example
from numpy import array
from numpy import hstack
from keras.models import Sequential
```

```
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70, 75], [80, 85], [90, 95]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.76: Example of an MLP model for multi-step forecasting for a dependent series.

Running the example fits the model and predicts the next two time steps of the output sequence beyond the dataset. We would expect the next two steps to be `[185, 205]`.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[186.53822 208.41725]]
```

Listing 7.77: Example output from an MLP model for multi-step forecasting for a dependent series.

7.5.2 Multiple Parallel Input and Multi-step Output

A problem with parallel time series may require the prediction of multiple time steps of each time series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 7.78: Example of a multivariate time series.

We may use the last three time steps from each of the three time series as input to the model and predict the next time steps of each of the three time series as output. The first sample in the training dataset would be the following.

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 7.79: Example input for the first a multivariate time series sample.

Output:

```
40, 45, 85
50, 55, 105
```

Listing 7.80: Example output for the first a multivariate time series sample.

The `split_sequences()` function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
```

```
  return array(X), array(y)
```

Listing 7.81: Example of a function for preparing data for a multi-step multivariate series.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.82: Example of preparing data for multi-step forecasting for a multivariate series.

Running the example first prints the shape of the prepared training dataset. We can see that both the input (X) and output (Y) elements of the dataset are three dimensional for the number of samples, time steps, and variables or parallel time series respectively. The input and output elements of each series are then printed side by side so that we can confirm that the data was prepared as we expected.

```
(5, 3, 3) (5, 2, 3)

[[10 15 25]
```

```
 [20 25 45]
 [30 35 65]] [[ 40 45 85]
 [ 50  55 105]]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [[ 50 55 105]
 [ 60  65 125]]
[[ 30 35  65]
 [ 40 45  85]
 [ 50 55 105]] [[ 60 65 125]
 [ 70 75 145]]
[[ 40 45  85]
 [ 50 55 105]
 [ 60 65 125]] [[ 70 75 145]
 [ 80 85 165]]
[[ 50 55 105]
 [ 60 65 125]
 [ 70 75 145]] [[ 80 85 165]
 [ 90 95 185]]
```

Listing 7.83: Example output from preparing data for multi-step forecasting for a multivariate series.

We can now develop an MLP model to make multivariate multi-step forecasts. In addition to flattening the shape of the input data, as we have in prior examples, we must also flatten the three-dimensional structure of the output data. This is because the MLP model is only capable of taking vector inputs and outputs.

```
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
```

Listing 7.84: Example of reshaping input and output data for fitting an MLP model for a multi-step multivariate series.

The complete example is listed below.

```
# multivariate multi-step mlp example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
```

```
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[60, 65, 125], [70, 75, 145], [80, 85, 165]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.85: Example of an MLP model for multi-step forecasting for a multivariate series.

Running the example fits the model and predicts the values for each of the three time steps for the next two time steps beyond the end of the dataset. We would expect the values for these series and time steps to be as follows:

```
90, 95, 185
100, 105, 205
```

Listing 7.86: Expected output for an out-of-sample multi-step multivariate forecast.

We can see that the model forecast gets reasonably close to the expected values.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[ 91.28376 96.567 188.37575 100.54482 107.9219 208.108 ]]
```

Listing 7.87: Example output from an MLP model for multi-step forecasting for a multivariate series.

7.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Problem Differences**. Explain the main changes to the MLP required when modeling each of univariate, multivariate and multi-step time series forecasting problems.
- **Experiment**. Select one example and modify it to work with your own small contrived dataset.
- **Develop Framework**. Use the examples in this chapter as the basis for a framework for automatically developing an MLP model for a given time series forecasting problem.

If you explore any of these extensions, I'd love to know.

7.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

7.7.1 Books

- *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999. https://amzn.to/2vORBWO.
- *Deep Learning*, 2016. https://amzn.to/2MQyLVZ
- *Deep Learning with Python*, 2017. https://amzn.to/2vMRiMe

7.7.2 APIs

- Keras: The Python Deep Learning library. https://keras.io/
- Getting started with the Keras Sequential model. https://keras.io/getting-started/sequential-model-guide/
- Getting started with the Keras functional API. https://keras.io/getting-started/functional-api-guide/
- Keras Sequential Model API. https://keras.io/models/sequential/
- Keras Core Layers API. https://keras.io/layers/core/

7.8 Summary

In this tutorial, you discovered how to develop a suite of Multilayer Perceptron, or MLP, models for a range of standard time series forecasting problems. Specifically, you learned:

- How to develop MLP models for univariate time series forecasting.
- How to develop MLP models for multivariate time series forecasting.
- How to develop MLP models for multi-step time series forecasting.

7.8.1 Next

In the next lesson, you will discover how to develop Convolutional Neural Network models for time series forecasting.

Chapter 8

How to Develop CNNs for Time Series Forecasting

Convolutional Neural Network models, or CNNs for short, can be applied to time series forecasting. There are many types of CNN models that can be used for each specific type of time series forecasting problem. In this tutorial, you will discover how to develop a suite of CNN models for a range of standard time series forecasting problems. The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem. After completing this tutorial, you will know:

- How to develop CNN models for univariate time series forecasting.
- How to develop CNN models for multivariate time series forecasting.
- How to develop CNN models for multi-step time series forecasting.

Let's get started.

8.1 Tutorial Overview

In this tutorial, we will explore how to develop CNN models for time series forecasting. The models are demonstrated on small contrived time series problems intended to give the flavor of the type of time series problem being addressed. The chosen configuration of the models is arbitrary and not optimized for each problem; that was not the goal. This tutorial is divided into four parts; they are:

1. Univariate CNN Models
2. Multivariate CNN Models
3. Multi-step CNN Models
4. Multivariate Multi-step CNN Models

8.2 Univariate CNN Models

Although traditionally developed for two-dimensional image data, CNNs can be used to model univariate time series forecasting problems. Univariate time series are datasets comprised of a single series of observations with a temporal ordering and a model is required to learn from the series of past observations to predict the next value in the sequence. This section is divided into two parts; they are:

1. Data Preparation
2. CNN Model

8.2.1 Data Preparation

Before a univariate series can be modeled, it must be prepared. The CNN model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the model can learn. Consider a given univariate sequence:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 8.1: Example of a univariate time series.

We can divide the sequence into multiple input/output patterns called samples, where three time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X,              y
10, 20, 30,     40
20, 30, 40,     50
30, 40, 50,     60
...
```

Listing 8.2: Example of a univariate time series as a supervised learning problem.

The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.3: Example of a function to split a univariate series into a supervised learning problem.

We can demonstrate this function on our small contrived dataset above. The complete example is listed below.

```
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.4: Example of transforming a univariate time series into a supervised learning problem.

Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

Listing 8.5: Example output from transforming a univariate time series into a supervised learning problem.

Now that we know how to prepare a univariate series for modeling, let's look at developing a CNN model that can learn the mapping of inputs to outputs.

8.2.2 CNN Model

A one-dimensional CNN is a CNN model that has a convolutional hidden layer that operates over a 1D sequence. This is followed by perhaps a second convolutional layer in some cases, such as very long input sequences, and then a pooling layer whose job it is to distill the output of the convolutional layer to the most salient elements. The convolutional and pooling layers

are followed by a dense fully connected layer that interprets the features extracted by the convolutional part of the model. A flatten layer is used between the convolutional layers and the dense layer to reduce the feature maps to a single one-dimensional vector. We can define a 1D CNN Model for univariate time series forecasting as follows.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 8.6: Example of defining a CNN model.

Key in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps and the number of features. We are working with a univariate series, so the number of features is one, for one variable. The number of time steps as input is the number we chose when preparing our dataset as an argument to the `split_sequence()` function.

The input shape for each sample is specified in the `input_shape` argument on the definition of the first hidden layer. We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape: `[samples, timesteps, features]`. Our `split_sequence()` function in the previous section outputs the X with the shape `[samples, timesteps]`, so we can easily reshape it to have an additional dimension for the one feature.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 8.7: Example of reshaping data for the CNN.

The CNN does not actually view the data as having time steps, instead, it is treated as a sequence over which convolutional read operations can be performed, like a one-dimensional image. In this example, we define a convolutional layer with 64 filter maps and a kernel size of 2. This is followed by a max pooling layer and a dense layer to interpret the input feature. An output layer is specified that predicts a single numerical value. The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or `'mse'`, loss function. Once the model is defined, we can fit it on the training dataset.

```
# fit model
model.fit(X, y, epochs=1000, verbose=0)
```

Listing 8.8: Example of fitting a CNN model.

After the model is fit, we can use it to make a prediction. We can predict the next value in the sequence by providing the input: `[70, 80, 90]`. And expecting the model to predict something like: `[100]`. The model expects the input shape to be three-dimensional with `[samples, timesteps, features]`, therefore, we must reshape the single input sample before making the prediction.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 8.9: Example of reshaping data read for making a prediction.

We can tie all of this together and demonstrate how to develop a 1D CNN model for univariate time series forecasting and make a single prediction.

```
# univariate cnn example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
```

```
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.10: Example of a CNN model for univariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction. We can see that the model predicts the next value in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.67965]]
```

Listing 8.11: Example output from a CNN model for univariate time series forecasting.

For an example of a CNN applied to a real-world univariate time series forecasting problem see Chapter 14. For an example of grid searching CNN hyperparameters on a univariate time series forecasting problem, see Chapter 15.

8.3 Multivariate CNN Models

Multivariate time series data means data where there is more than one observation for each time step. There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.
2. Multiple Parallel Series.

Let's take a look at each in turn.

8.3.1 Multiple Input Series

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series. The input time series are parallel because each series has observations at the same time steps. We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

Listing 8.12: Example of defining multiple parallel series.

We can reshape these three arrays of data as a single dataset where each row is a time step and each column is a separate time series. This is a standard way of storing parallel time series in a CSV file.

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

Listing 8.13: Example of defining parallel series as a dataset.

The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
print(dataset)
```

Listing 8.14: Example of defining a dependent time series dataset.

Running the example prints the dataset with one row per time step and one column for each of the two input and one output parallel time series.

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 8.15: Example output from defining a dependent time series dataset.

As with the univariate time series, we must structure these data into samples with input and output samples. A 1D CNN model needs sufficient context to learn a mapping from an input sequence to an output value. CNNs can support parallel input time series as separate channels, like red, green, and blue components of an image. Therefore, we need to split the data into samples maintaining the order of observations across the two input sequences. If we chose three input time steps, then the first sample would look as follows:

Input:

```
10, 15
20, 25
30, 35
```

Listing 8.16: Example input from the first sample.

Output:

```
65
```

Listing 8.17: Example output from the first sample.

That is, the first three time steps of each parallel series are provided as input to the model and the model associates this with the value in the output series at the third time step, in this case, 65. We can see that, in transforming the time series into input/output samples to train the model, that we will have to discard some values from the output time series where we do not have values in the input time series at prior time steps. In turn, the choice of the size of the number of input time steps will have an important effect on how much of the training data is used. We can define a function named `split_sequences()` that will take a dataset as we have defined it with rows for time steps and columns for parallel series and return input/output samples.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.18: Example of a function for preparing samples for a dependent time series.

We can test this function on our dataset using three time steps for each input time series as input. The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
```

```
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.19: Example of splitting a dependent series into samples.

Running the example first prints the shape of the X and y components. We can see that the X component has a three-dimensional structure. The first dimension is the number of samples, in this case 7. The second dimension is the number of time steps per sample, in this case 3, the value specified to the function. Finally, the last dimension specifies the number of parallel time series or the number of variables, in this case 2 for the two parallel series.

This is the exact three-dimensional structure expected by a 1D CNN as input. The data is ready to use without further reshaping. We can then see that the input and output for each sample is printed, showing the three time steps for each of the two input series and the associated output for each sample.

```
(7, 3, 2) (7,)

[[10 15]
 [20 25]
 [30 35]] 65
[[20 25]
 [30 35]
 [40 45]] 85
[[30 35]
 [40 45]
 [50 55]] 105
[[40 45]
 [50 55]
 [60 65]] 125
[[50 55]
 [60 65]
 [70 75]] 145
[[60 65]
 [70 75]
 [80 85]] 165
[[70 75]
 [80 85]
 [90 95]] 185
```

Listing 8.20: Example output from splitting a dependent series into samples.

CNN Model

We are now ready to fit a 1D CNN model on this data, specifying the expected number of time steps and features to expect for each input sample, in this case three and two respectively.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 8.21: Example of defining a CNN for forecasting a dependent series.

When making a prediction, the model expects three time steps for two input time series. We can predict the next value in the output series providing the input values of:

```
80, 85
90, 95
100, 105
```

Listing 8.22: Example input for forecasting out-of-sample.

The shape of the one sample with three time steps and two variables must be `[1, 3, 2]`. We would expect the next value in the sequence to be $100 + 105$ or 205.

```
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 8.23: Example of preparing input for forecasting out-of-sample.

The complete example is listed below.

```
# multivariate cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
```

```
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.24: Example of a CNN model for forecasting a dependent time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[206.0161]]
```

Listing 8.25: Example output from a CNN model for forecasting a dependent time series.

Multi-headed CNN Model

There is another, more elaborate way to model the problem. Each input series can be handled by a separate CNN and the output of each of these submodels can be combined before a prediction is made for the output sequence. We can refer to this as a multi-headed CNN model. It may offer more flexibility or better performance depending on the specifics of the problem that is

being modeled. For example, it allows you to configure each submodel differently for each input series, such as the number of filter maps and the kernel size. This type of model can be defined in Keras using the Keras functional API. First, we can define the first input model as a 1D CNN with an input layer that expects vectors with `n_steps` and 1 feature.

```
# first input model
visible1 = Input(shape=(n_steps, n_features))
cnn1 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible1)
cnn1 = MaxPooling1D(pool_size=2)(cnn1)
cnn1 = Flatten()(cnn1)
```

Listing 8.26: Example of defining the first input model.

We can define the second input submodel in the same way.

```
# second input model
visible2 = Input(shape=(n_steps, n_features))
cnn2 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible2)
cnn2 = MaxPooling1D(pool_size=2)(cnn2)
cnn2 = Flatten()(cnn2)
```

Listing 8.27: Example of defining the second input model.

Now that both input submodels have been defined, we can merge the output from each model into one long vector which can be interpreted before making a prediction for the output sequence.

```
# merge input models
merge = concatenate([cnn1, cnn2])
dense = Dense(50, activation='relu')(merge)
output = Dense(1)(dense)
```

Listing 8.28: Example of defining the output model.

We can then tie the inputs and outputs together.

```
# connect input and output models
model = Model(inputs=[visible1, visible2], outputs=output)
```

Listing 8.29: Example of connecting the input and output models.

The image below provides a schematic for how this model looks, including the shape of the inputs and outputs of each layer.

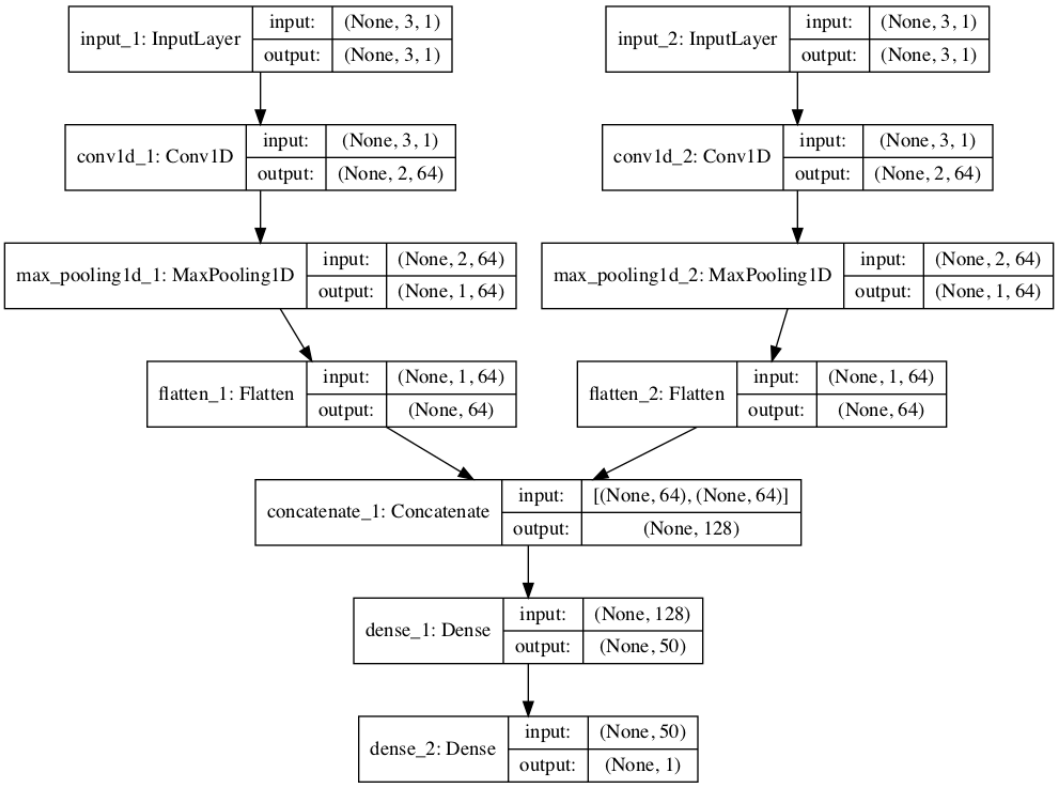

Figure 8.1: Plot of Multi-headed 1D CNN for Multivariate Time Series Forecasting.

This model requires input to be provided as a list of two elements where each element in the list contains data for one of the submodels. In order to achieve this, we can split the 3D input data into two separate arrays of input data; that is from one array with the shape `[7, 3, 2]` to two 3D arrays with `[7, 3, 1]`.

```
# one time series per head
n_features = 1
# separate input data
X1 = X[:, :, 0].reshape(X.shape[0], X.shape[1], n_features)
X2 = X[:, :, 1].reshape(X.shape[0], X.shape[1], n_features)
```

Listing 8.30: Example of preparing the input data for the multi-headed model.

These data can then be provided in order to fit the model.

```
# fit model
model.fit([X1, X2], y, epochs=1000, verbose=0)
```

Listing 8.31: Example of fitting the multi-headed model.

Similarly, we must prepare the data for a single sample as two separate two-dimensional arrays when making a single one-step prediction.

```
# reshape one sample for making a prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps, n_features))
x2 = x_input[:, 1].reshape((1, n_steps, n_features))
```

Listing 8.32: Example of preparing data for forecasting with the multi-headed model.

We can tie all of this together; the complete example is listed below.

```
# multivariate multi-headed 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.merge import concatenate

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# one time series per head
n_features = 1
# separate input data
X1 = X[:, :, 0].reshape(X.shape[0], X.shape[1], n_features)
X2 = X[:, :, 1].reshape(X.shape[0], X.shape[1], n_features)
# first input model
visible1 = Input(shape=(n_steps, n_features))
cnn1 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible1)
cnn1 = MaxPooling1D(pool_size=2)(cnn1)
cnn1 = Flatten()(cnn1)
# second input model
visible2 = Input(shape=(n_steps, n_features))
cnn2 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible2)
cnn2 = MaxPooling1D(pool_size=2)(cnn2)
```

```
cnn2 = Flatten()(cnn2)
# merge input models
merge = concatenate([cnn1, cnn2])
dense = Dense(50, activation='relu')(merge)
output = Dense(1)(dense)
model = Model(inputs=[visible1, visible2], outputs=output)
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit([X1, X2], y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps, n_features))
x2 = x_input[:, 1].reshape((1, n_steps, n_features))
yhat = model.predict([x1, x2], verbose=0)
print(yhat)
```

Listing 8.33: Example of a Multi-headed CNN for forecasting a dependent time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[205.871]]
```

Listing 8.34: Example output from a Multi-headed CNN model for forecasting a dependent time series.

For an example of CNN models developed for a multivariate time series classification problem, see Chapter 24.

8.3.2 Multiple Parallel Series

An alternate time series problem is the case where there are multiple parallel time series and a value must be predicted for each. For example, given the data from the previous section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 8.35: Example of parallel time series.

We may want to predict the value for each of the three time series for the next time step. This might be referred to as multivariate forecasting. Again, the data must be split into input/output samples in order to train a model. The first sample of this dataset would be:

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 8.36: Example input from the first sample.

Output:

```
40, 45, 85
```

Listing 8.37: Example output from the first sample.

The `split_sequences()` function below will split multiple parallel time series with rows for time steps and one series per column into the required input/output shape.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.38: Example of splitting multiple parallel time series into samples.

We can demonstrate this on the contrived problem; the complete example is listed below.

```
# multivariate output data prep
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.39: Example of splitting multiple parallel series into samples.

Running the example first prints the shape of the prepared X and y components. The shape of X is three-dimensional, including the number of samples (6), the number of time steps chosen per sample (3), and the number of parallel time series or features (3). The shape of y is two-dimensional as we might expect for the number of samples (6) and the number of time variables per sample to be predicted (3). The data is ready to use in a 1D CNN model that expects three-dimensional input and two-dimensional output shapes for the X and y components of each sample. Then, each of the samples is printed showing the input and output components of each sample.

```
(6, 3, 3) (6, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [40 45 85]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [ 50 55 105]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [ 60 65 125]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [ 70 75 145]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [ 80 85 165]
[[ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]] [ 90 95 185]
```

Listing 8.40: Example output from splitting multiple parallel series into samples.

Vector-Output CNN Model

We are now ready to fit a 1D CNN model on this data. In this model, the number of time steps and parallel series (features) are specified for the input layer via the `input_shape` argument.

The number of parallel series is also used in the specification of the number of values to predict by the model in the output layer; again, this is three.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
```

Listing 8.41: Example of defining a CNN model for forecasting multiple parallel time series.

We can predict the next value in each of the three parallel series by providing an input of three time steps for each series.

```
70, 75, 145
80, 85, 165
90, 95, 185
```

Listing 8.42: Example input for forecasting out-of-sample.

The shape of the input for making a single prediction must be 1 sample, 3 time steps, and 3 features, or `[1, 3, 3]`.

```
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 8.43: Example of preparing data for forecasting out-of-sample.

We would expect the vector output to be: `[100, 105, 205]`. We can tie all of this together and demonstrate a 1D CNN for multivariate output time series forecasting below.

```
# multivariate output 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
```

```
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=3000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.44: Example of a CNN model for forecasting multiple parallel time series.

Running the example prepares the data, fits the model and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[100.11272 105.32213 205.53436]]
```

Listing 8.45: Example output from a CNN model for forecasting multiple parallel time series.

Multi-output CNN Model

As with multiple input series, there is another more elaborate way to model the problem. Each output series can be handled by a separate output CNN model. We can refer to this as a multi-output CNN model. It may offer more flexibility or better performance depending on the specifics of the problem that is being modeled. This type of model can be defined in Keras using the Keras functional API. First, we can define the first input model as a 1D CNN model.

```
# define model
visible = Input(shape=(n_steps, n_features))
cnn = Conv1D(filters=64, kernel_size=2, activation='relu')(visible)
cnn = MaxPooling1D(pool_size=2)(cnn)
cnn = Flatten()(cnn)
cnn = Dense(50, activation='relu')(cnn)
```

Listing 8.46: Example of defining the input model.

We can then define one output layer for each of the three series that we wish to forecast, where each output submodel will forecast a single time step.

```
# define output 1
output1 = Dense(1)(cnn)
# define output 2
output2 = Dense(1)(cnn)
# define output 3
output3 = Dense(1)(cnn)
```

Listing 8.47: Example of defining the output models.

We can then tie the input and output layers together into a single model.

```
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
```

Listing 8.48: Example of connecting the input and output models.

To make the model architecture clear, the schematic below clearly shows the three separate output layers of the model and the input and output shapes of each layer.

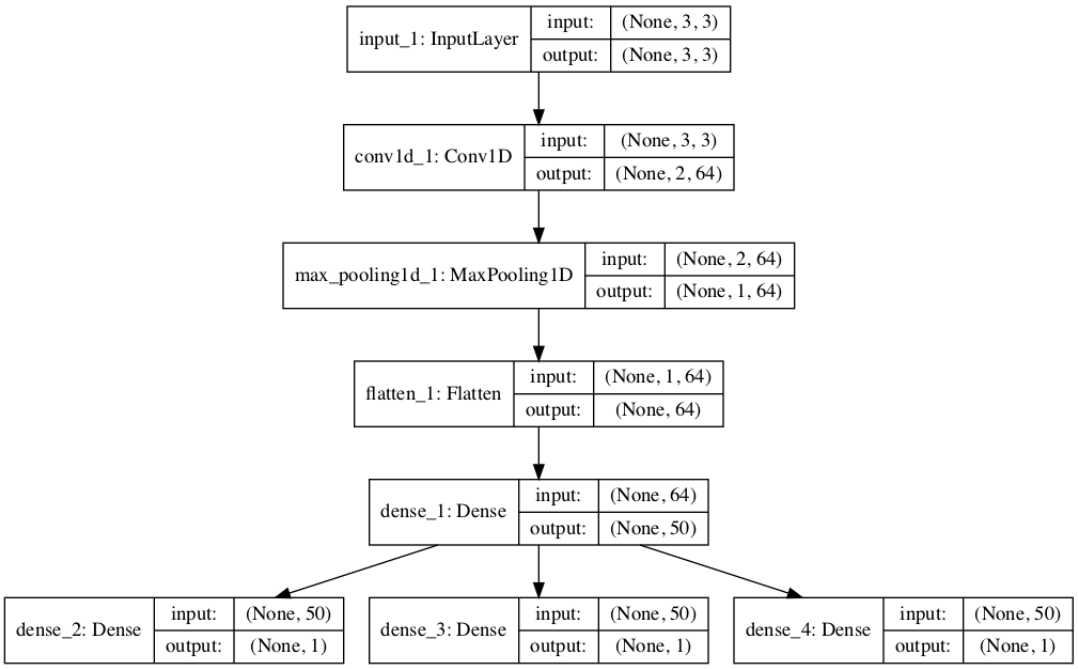

Figure 8.2: Plot of Multi-output 1D CNN for Multivariate Time Series Forecasting.

When training the model, it will require three separate output arrays per sample. We can achieve this by converting the output training data that has the shape `[7, 3]` to three arrays with the shape `[7, 1]`.

```
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
```

Listing 8.49: Example of preparing the output samples for fitting the multi-output model.

These arrays can be provided to the model during training.

```
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
```

Listing 8.50: Example of fitting the multi-output model.

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

```
# multivariate output 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
```

```
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
# define model
visible = Input(shape=(n_steps, n_features))
cnn = Conv1D(filters=64, kernel_size=2, activation='relu')(visible)
cnn = MaxPooling1D(pool_size=2)(cnn)
cnn = Flatten()(cnn)
cnn = Dense(50, activation='relu')(cnn)
# define output 1
output1 = Dense(1)(cnn)
# define output 2
output2 = Dense(1)(cnn)
# define output 3
output3 = Dense(1)(cnn)
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.51: Example of a Multi-output CNN model for forecasting multiple parallel time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[array([[100.96118]], dtype=float32),
 array([[105.502686]], dtype=float32),
 array([[205.98045]], dtype=float32)]
```

Listing 8.52: Example output from a Multi-output CNN model for forecasting multiple parallel time series.

For an example of CNN models developed for a multivariate time series forecasting problem, see Chapter 19.

8.4 Multi-step CNN Models

In practice, there is little difference to the 1D CNN model in predicting a vector output that represents different output variables (as in the previous example), or a vector output that represents multiple time steps of one variable. Nevertheless, there are subtle and important differences in the way the training data is prepared. In this section, we will demonstrate the

case of developing a multi-step forecast model using a vector model. Before we look at the specifics of the model, let's first look at the preparation of data for multi-step forecasting.

8.4.1 Data Preparation

As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components. Both the input and output components will be comprised of multiple time steps and may or may not have the same number of steps. For example, given the univariate time series:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 8.53: Example of a univariate time series.

We could use the last three time steps as input and forecast the next two time steps. The first sample would look as follows:

Input:

```
[10, 20, 30]
```

Listing 8.54: Example input for the first sample.

Output:

```
[40, 50]
```

Listing 8.55: Example output for the first sample.

The `split_sequence()` function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.56: Example of function for splitting a univariate time series into samples for multi-step forecasting.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multi-step data preparation
from numpy import array

# split a univariate sequence into samples
```

```
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.57: Example transforming a time series into samples for multi-step forecasting.

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

Listing 8.58: Example output from transforming a time series into samples for multi-step forecasting.

Now that we know how to prepare data for multi-step forecasting, let's look at a 1D CNN model that can learn this mapping.

8.4.2 Vector Output Model

The 1D CNN can output a vector directly that can be interpreted as a multi-step forecast. This approach was seen in the previous section were one time step of each output time series was forecasted as a vector. As with the 1D CNN models for univariate data in a prior section, the prepared samples must first be reshaped. The CNN expects data to have a three-dimensional structure of `[samples, timesteps, features]`, and in this case, we only have one feature so the reshape is straightforward.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 8.59: Example of reshaping data for multi-step forecasting.

With the number of input and output steps specified in the `n_steps_in` and `n_steps_out` variables, we can define a multi-step time-series forecasting model.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
```

Listing 8.60: Example of defining a CNN model for multi-step forecasting.

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input: `[70, 80, 90]`. We would expect the predicted output to be: `[100, 110]`. As expected by the model, the shape of the single sample of input data when making the prediction must be `[1, 3, 1]` for the 1 sample, 3 time steps of the input, and the single feature.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 8.61: Example of reshaping data for making an out-of-sample forecast.

Tying all of this together, the 1D CNN for multi-step forecasting with a univariate time series is listed below.

```
# univariate multi-step vector-output 1d cnn example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

```
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.62: Example of a vector-output CNN for multi-step forecasting.

Running the example forecasts and prints the next two time steps in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.86651 115.08979]]
```

Listing 8.63: Example output from a vector-output CNN for multi-step forecasting.

For an example of CNN models developed for a multi-step time series forecasting problem, see Chapter 19.

8.5 Multivariate Multi-step CNN Models

In the previous sections, we have looked at univariate, multivariate, and multi-step time series forecasting. It is possible to mix and match the different types of 1D CNN models presented so far for the different problems. This too applies to time series forecasting problems that involve multivariate and multi-step forecasting, but it may be a little more challenging. In this section, we will explore short examples of data preparation and modeling for multivariate multi-step time series forecasting as a template to ease this challenge, specifically:

1. Multiple Input Multi-step Output.
2. Multiple Parallel Input and Multi-step Output.

Perhaps the biggest stumbling block is in the preparation of data, so this is where we will focus our attention.

8.5.1 Multiple Input Multi-step Output

There are those multivariate time series forecasting problems where the output series is separate but dependent upon the input time series, and multiple time steps are required for the output series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 8.64: Example of a multivariate time series.

We may use three prior time steps of each of the two input time series to predict two time steps of the output time series.

Input:

```
10, 15
20, 25
30, 35
```

Listing 8.65: Example input for the first sample.

Output:

```
65
85
```

Listing 8.66: Example output for the first sample.

The `split_sequences()` function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.67: Example of a function for transforming a multivariate time series into samples for multi-step forecasting.

We can demonstrate this on our contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.68: Example preparing a multivariate input dependent time series with multi-step forecasts.

Running the example first prints the shape of the prepared training data. We can see that the shape of the input portion of the samples is three-dimensional, comprised of six samples, with three time steps and two variables for the two input time series. The output portion of the samples is two-dimensional for the six samples and the two time steps for each sample to be predicted. The prepared samples are then printed to confirm that the data was prepared as we specified.

```
(6, 3, 2) (6, 2)

[[10 15]
 [20 25]
 [30 35]] [65 85]
[[20 25]
```

```
 [30 35]
 [40 45]] [ 85 105]
[[30 35]
 [40 45]
 [50 55]] [105 125]
[[40 45]
 [50 55]
 [60 65]] [125 145]
[[50 55]
 [60 65]
 [70 75]] [145 165]
[[60 65]
 [70 75]
 [80 85]] [165 185]
```

Listing 8.69: Example output from preparing a multivariate input dependent time series with multi-step forecasts.

We can now develop a 1D CNN model for multi-step predictions. In this case, we will demonstrate a vector output model. The complete example is listed below.

```
# multivariate multi-step 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

```
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70, 75], [80, 85], [90, 95]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.70: Example of a CNN model for multivariate dependent time series with multi-step forecasts.

Running the example fits the model and predicts the next two time steps of the output sequence beyond the dataset. We would expect the next two steps to be `[185, 205]`.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[185.57011 207.77893]]
```

Listing 8.71: Example output from a CNN model for multivariate dependent time series with multi-step forecasts.

8.5.2 Multiple Parallel Input and Multi-step Output

A problem with parallel time series may require the prediction of multiple time steps of each time series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 8.72: Example of a multivariate time series.

We may use the last three time steps from each of the three time series as input to the model, and predict the next time steps of each of the three time series as output. The first sample in the training dataset would be the following.

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 8.73: Example input for the first sample.

Output:

```
40, 45, 85
50, 55, 105
```

Listing 8.74: Example output for the first sample.

The `split_sequences()` function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.75: Example of a function for transforming a multivariate time series into samples for multi-step forecasting.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
```

```
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.76: Example preparing a multivariate parallel time series with multi-step forecasts.

Running the example first prints the shape of the prepared training dataset. We can see that both the input (X) and output (Y) elements of the dataset are three dimensional for the number of samples, time steps, and variables or parallel time series respectively. The input and output elements of each series are then printed side by side so that we can confirm that the data was prepared as we expected.

```
(5, 3, 3) (5, 2, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [[ 40 45 85]
 [ 50  55 105]]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [[ 50 55 105]
 [ 60  65 125]]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [[ 60 65 125]
 [ 70  75 145]]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [[ 70 75 145]
 [ 80  85 165]]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [[ 80 85 165]
 [ 90  95 185]]
```

Listing 8.77: Example output from preparing a multivariate parallel time series with multi-step forecasts.

We can now develop a 1D CNN model for this dataset. We will use a vector-output model in this case. As such, we must flatten the three-dimensional structure of the output portion of each sample in order to train the model. This means, instead of predicting two steps for each series, the model is trained on and expected to predict a vector of six numbers directly.

```
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
```

Listing 8.78: Example of flattening output samples for training the model with vector output.

The complete example is listed below.

```
# multivariate output multi-step 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
```

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=7000, verbose=0)
# demonstrate prediction
x_input = array([[60, 65, 125], [70, 75, 145], [80, 85, 165]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.79: Example of a CNN model for multivariate parallel time series with multi-step forecasts.

Running the example fits the model and predicts the values for each of the three time steps for the next two time steps beyond the end of the dataset. We would expect the values for these series and time steps to be as follows:

```
90, 95, 185
100, 105, 205
```

Listing 8.80: Example output for the out-of-sample forecast.

We can see that the model forecast gets reasonably close to the expected values.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[ 90.47855 95.621284 186.02629 100.48118 105.80815 206.52821 ]]
```

Listing 8.81: Example output from a CNN model for multivariate parallel time series with multi-step forecasts.

For an example of CNN models developed for a multivariate multi-step time series forecasting problem, see Chapter 19.

8.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Problem Differences**. Explain the main changes to the CNN required when modeling each of univariate, multivariate and multi-step time series forecasting problems.
- **Experiment**. Select one example and modify it to work with your own small contrived dataset.
- **Develop Framework**. Use the examples in this chapter as the basis for a framework for automatically developing an CNN model for a given time series forecasting problem.

If you explore any of these extensions, I'd love to know.

8.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

8.7.1 Books

- *Deep Learning*, 2016.
  https://amzn.to/2MQyLVZ
- *Deep Learning with Python*, 2017.
  https://amzn.to/2vMRiMe

8.7.2 Papers

- *Backpropagation Applied to Handwritten Zip Code Recognition*, 1989.
  https://ieeexplore.ieee.org/document/6795724/
- *Gradient-based Learning Applied to Document Recognition*, 1998.
  https://ieeexplore.ieee.org/document/726791/
- *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.
  https://arxiv.org/abs/1409.1556

8.7.3 APIs

- Keras: The Python Deep Learning library.
  https://keras.io/
- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/
- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/
- Keras Sequential Model API.
  https://keras.io/models/sequential/
- Keras Core Layers API.
  https://keras.io/layers/core/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/
- Keras Pooling Layers API.
  https://keras.io/layers/pooling/

8.8 Summary

In this tutorial, you discovered how to develop a suite of CNN models for a range of standard time series forecasting problems. Specifically, you learned:

- How to develop CNN models for univariate time series forecasting.
- How to develop CNN models for multivariate time series forecasting.
- How to develop CNN models for multi-step time series forecasting.

8.8.1 Next

In the next lesson, you will discover how to develop Recurrent Neural Network models for time series forecasting.

Chapter 9

How to Develop LSTMs for Time Series Forecasting

Long Short-Term Memory networks, or LSTMs for short, can be applied to time series forecasting. There are many types of LSTM models that can be used for each specific type of time series forecasting problem. In this tutorial, you will discover how to develop a suite of LSTM models for a range of standard time series forecasting problems. The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem.

After completing this tutorial, you will know:

- How to develop LSTM models for univariate time series forecasting.
- How to develop LSTM models for multivariate time series forecasting.
- How to develop LSTM models for multi-step time series forecasting.

Let's get started.

9.1 Tutorial Overview

In this tutorial, we will explore how to develop a suite of different types of LSTM models for time series forecasting. The models are demonstrated on small contrived time series problems intended to give the flavor of the type of time series problem being addressed. The chosen configuration of the models is arbitrary and not optimized for each problem; that was not the goal. This tutorial is divided into four parts; they are:

1. Univariate LSTM Models
2. Multivariate LSTM Models
3. Multi-step LSTM Models
4. Multivariate Multi-step LSTM Models

9.2 Univariate LSTM Models

LSTMs can be used to model univariate time series forecasting problems. These are problems comprised of a single series of observations and a model is required to learn from the series of past observations to predict the next value in the sequence. We will demonstrate a number of variations of the LSTM model for univariate time series forecasting. This section is divided into six parts; they are:

1. Data Preparation
2. Vanilla LSTM
3. Stacked LSTM
4. Bidirectional LSTM
5. CNN-LSTM
6. ConvLSTM

Each of these models are demonstrated for one-step univariate time series forecasting, but can easily be adapted and used as the input part of a model for other types of time series forecasting problems.

9.2.1 Data Preparation

Before a univariate series can be modeled, it must be prepared. The LSTM model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the LSTM can learn. Consider a given univariate sequence:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 9.1: Example of a univariate time series.

We can divide the sequence into multiple input/output patterns called samples, where three time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X,              y
10, 20, 30,     40
20, 30, 40,     50
30, 40, 50,     60
...
```

Listing 9.2: Example of a univariate time series as a supervised learning problem.

The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.3: Example of a function to split a univariate series into a supervised learning problem.

We can demonstrate this function on our small contrived dataset above. The complete example is listed below.

```
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.4: Example of transforming a univariate time series into a supervised learning problem.

Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
```

```
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

Listing 9.5: Example output from transforming a univariate time series into a supervised learning problem.

Now that we know how to prepare a univariate series for modeling, let's look at developing LSTM models that can learn the mapping of inputs to outputs, starting with a Vanilla LSTM.

9.2.2 Vanilla LSTM

A Vanilla LSTM is an LSTM model that has a single hidden layer of LSTM units, and an output layer used to make a prediction. Key to LSTMs is that they offer native support for sequences. Unlike a CNN that reads across the entire input vector, the LSTM model reads one time step of the sequence at a time and builds up an internal state representation that can be used as a learned context for making a prediction. We can define a Vanilla LSTM for univariate time series forecasting as follows.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.6: Example of defining a Vanilla LSTM model.

Key in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps and the number of features. We are working with a univariate series, so the number of features is one, for one variable. The number of time steps as input is the number we chose when preparing our dataset as an argument to the `split_sequence()` function.

The shape of the input for each sample is specified in the `input_shape` argument on the definition of first hidden layer. We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape: `[samples, timesteps, features]`. Our `split_sequence()` function in the previous section outputs the X with the shape `[samples, timesteps]`, so we easily reshape it to have an additional dimension for the one feature.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 9.7: Example of reshaping training data for the LSTM.

In this case, we define a model with 50 LSTM units in the hidden layer and an output layer that predicts a single numerical value. The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or `'mse'` loss function. Once the model is defined, we can fit it on the training dataset.

```
# fit model
model.fit(X, y, epochs=200, verbose=0)
```

Listing 9.8: Example of fitting the LSTM model.

After the model is fit, we can use it to make a prediction. We can predict the next value in the sequence by providing the input: `[70, 80, 90]`. And expecting the model to predict something like: `[100]`. The model expects the input shape to be three-dimensional with `[samples, timesteps, features]`, therefore, we must reshape the single input sample before making the prediction.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.9: Example of preparing an input sample ready for making an out-of-sample forecast.

We can tie all of this together and demonstrate how to develop a Vanilla LSTM for univariate time series forecasting and make a single prediction.

```
# univariate lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
```

```
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.10: Example of a Vanilla LSTM for univariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction. We can see that the model predicts the next value in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.09213]]
```

Listing 9.11: Example output from a Vanilla LSTM for univariate time series forecasting.

9.2.3 Stacked LSTM

Multiple hidden LSTM layers can be stacked one on top of another in what is referred to as a Stacked LSTM model. An LSTM layer requires a three-dimensional input and LSTMs by default will produce a two-dimensional output as an interpretation from the end of the sequence. We can address this by having the LSTM output a value for each time step in the input data by setting the `return_sequences=True` argument on the layer. This allows us to have 3D output from hidden LSTM layer as input to the next. We can therefore define a Stacked LSTM as follows.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.12: Example of defining a Stacked LSTM model.

We can tie this together; the complete code example is listed below.

```
# univariate stacked lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
```

```
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.13: Example of a Stacked LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.47341]]
```

Listing 9.14: Example output from a Stacked LSTM for univariate time series forecasting.

9.2.4 Bidirectional LSTM

On some sequence prediction problems, it can be beneficial to allow the LSTM model to learn the input sequence both forward and backwards and concatenate both interpretations. This is called a Bidirectional LSTM. We can implement a Bidirectional LSTM for univariate time series forecasting by wrapping the first hidden layer in a wrapper layer called Bidirectional. An example of defining a Bidirectional LSTM to read input both forward and backward is as follows.

```
# define model
model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.15: Example of defining a Bidirectional LSTM model.

The complete example of the Bidirectional LSTM for univariate time series forecasting is listed below.

```
# univariate bidirectional lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Bidirectional

# split a univariate sequence
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.16: Example of a Bidirectional LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.48093]]
```

Listing 9.17: Example output from a Bidirectional LSTM for univariate time series forecasting.

9.2.5 CNN-LSTM

A convolutional neural network, or CNN for short, is a type of neural network developed for working with two-dimensional image data. The CNN can be very effective at automatically extracting and learning features from one-dimensional sequence data such as univariate time series data. A CNN model can be used in a hybrid model with an LSTM backend where the CNN is used to interpret subsequences of input that together are provided as a sequence to an LSTM model to interpret. This hybrid model is called a CNN-LSTM.

The first step is to split the input sequences into subsequences that can be processed by the CNN model. For example, we can first split our univariate time series data into input/output samples with four steps as input and one as output. Each sample can then be split into two sub-samples, each with two time steps. The CNN can interpret each subsequence of two time steps and provide a time series of interpretations of the subsequences to the LSTM model to process as input. We can parameterize this and define the number of subsequences as `n_seq` and the number of time steps per subsequence as `n_steps`. The input data can then be reshaped to have the required structure: `[samples, subsequences, timesteps, features]`. For example:

```
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, subsequences, timesteps, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, n_steps, n_features))
```

Listing 9.18: Example of reshaping data for a CNN-LSTM model.

We want to reuse the same CNN model when reading in each sub-sequence of data separately. This can be achieved by wrapping the entire CNN model in a `TimeDistributed` wrapper that will apply the entire model once per input, in this case, once per input subsequence. The CNN model first has a convolutional layer for reading across the subsequence that requires a number of filters and a kernel size to be specified. The number of filters is the number of reads or interpretations of the input sequence. The kernel size is the number of time steps included of each *read* operation of the input sequence. The convolution layer is followed by a max pooling layer that distills the filter maps down to $\frac{1}{4}$ of their size that includes the most salient features. These structures are then flattened down to a single one-dimensional vector to be used as a single input time step to the LSTM layer.

```
# define the input cnn model
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=1, activation='relu'),
    input_shape=(None, n_steps, n_features)))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
```

Listing 9.19: Example of defining the CNN input model.

Next, we can define the LSTM part of the model that interprets the CNN model's read of the input sequence and makes a prediction.

```
# define the output model
```

```
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
```

Listing 9.20: Example of defining the LSTM output model.

We can tie all of this together; the complete example of a CNN-LSTM model for univariate time series forecasting is listed below.

```
# univariate cnn lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, subsequences, timesteps, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, n_steps, n_features))
# define model
model = Sequential()
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=1, activation='relu'),
    input_shape=(None, n_steps, n_features)))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=500, verbose=0)
# demonstrate prediction
x_input = array([60, 70, 80, 90])
```

```
x_input = x_input.reshape((1, n_seq, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.21: Example of a CNN-LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.69263]]
```

Listing 9.22: Example output from a CNN-LSTM for univariate time series forecasting.

9.2.6 ConvLSTM

A type of LSTM related to the CNN-LSTM is the ConvLSTM, where the convolutional reading of input is built directly into each LSTM unit. The ConvLSTM was developed for reading two-dimensional spatial-temporal data, but can be adapted for use with univariate time series forecasting. The layer expects input as a sequence of two-dimensional images, therefore the shape of input data must be: `[samples, timesteps, rows, columns, features]`.

For our purposes, we can split each sample into subsequences where timesteps will become the number of subsequences, or `n_seq`, and columns will be the number of time steps for each subsequence, or `n_steps`. The number of rows is fixed at 1 as we are working with one-dimensional data. We can now reshape the prepared samples into the required structure.

```
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, rows, columns, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, 1, n_steps, n_features))
```

Listing 9.23: Example of reshaping data for a ConvLSTM model.

We can define the ConvLSTM as a single layer in terms of the number of filters and a two-dimensional kernel size in terms of `(rows, columns)`. As we are working with a one-dimensional series, the number of rows is always fixed to 1 in the kernel. The output of the model must then be flattened before it can be interpreted and a prediction made.

```
# define the input cnnlstm model
model.add(ConvLSTM2D(filters=64, kernel_size=(1,2), activation='relu', input_shape=(n_seq,
    1, n_steps, n_features)))
model.add(Flatten())
```

Listing 9.24: Example of defining the ConvLSTM input model.

The complete example of a ConvLSTM for one-step univariate time series forecasting is listed below.

```
# univariate convlstm example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import ConvLSTM2D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, rows, columns, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, 1, n_steps, n_features))
# define model
model = Sequential()
model.add(ConvLSTM2D(filters=64, kernel_size=(1,2), activation='relu', input_shape=(n_seq,
    1, n_steps, n_features)))
model.add(Flatten())
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=500, verbose=0)
# demonstrate prediction
x_input = array([60, 70, 80, 90])
x_input = x_input.reshape((1, n_seq, 1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.25: Example of a ConvLSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[103.68166]]
```

Listing 9.26: Example output from a ConvLSTM for univariate time series forecasting.

For an example of an LSTM applied to a real-world univariate time series forecasting problem see Chapter 14. For an example of grid searching LSTM hyperparameters on a univariate time series forecasting problem, see Chapter 15. Now that we have looked at LSTM models for univariate data, let's turn our attention to multivariate data.

9.3 Multivariate LSTM Models

Multivariate time series data means data where there is more than one observation for each time step. There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.
2. Multiple Parallel Series.

Let's take a look at each in turn.

9.3.1 Multiple Input Series

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series. The input time series are parallel because each series has an observation at the same time steps. We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

Listing 9.27: Example of defining multiple input and a dependent time series.

We can reshape these three arrays of data as a single dataset where each row is a time step, and each column is a separate time series. This is a standard way of storing parallel time series in a CSV file.

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

Listing 9.28: Example of reshaping the parallel series into the columns of a dataset.

The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
print(dataset)
```

Listing 9.29: Example of defining a dependent series dataset.

Running the example prints the dataset with one row per time step and one column for each of the two input and one output parallel time series.

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 9.30: Example output from defining a dependent series dataset.

As with the univariate time series, we must structure these data into samples with input and output elements. An LSTM model needs sufficient context to learn a mapping from an input sequence to an output value. LSTMs can support parallel input time series as separate variables or features. Therefore, we need to split the data into samples maintaining the order of observations across the two input sequences. If we chose three input time steps, then the first sample would look as follows:

Input:

```
10, 15
20, 25
30, 35
```

Listing 9.31: Example input from the first sample.

Output:

```
65
```

Listing 9.32: Example Output from the first sample.

That is, the first three time steps of each parallel series are provided as input to the model and the model associates this with the value in the output series at the third time step, in this case, 65. We can see that, in transforming the time series into input/output samples to train the model, that we will have to discard some values from the output time series where we do

not have values in the input time series at prior time steps. In turn, the choice of the size of the number of input time steps will have an important effect on how much of the training data is used. We can define a function named `split_sequences()` that will take a dataset as we have defined it with rows for time steps and columns for parallel series and return input/output samples.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.33: Example of a function for transforming a dependent series dataset into samples.

We can test this function on our dataset using three time steps for each input time series as input. The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
```

```
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.34: Example of splitting a dependent series dataset into samples.

Running the example first prints the shape of the X and y components. We can see that the X component has a three-dimensional structure. The first dimension is the number of samples, in this case 7. The second dimension is the number of time steps per sample, in this case 3, the value specified to the function. Finally, the last dimension specifies the number of parallel time series or the number of variables, in this case 2 for the two parallel series. This is the exact three-dimensional structure expected by an LSTM as input. The data is ready to use without further reshaping. We can then see that the input and output for each sample is printed, showing the three time steps for each of the two input series and the associated output for each sample.

```
(7, 3, 2) (7,)

[[10 15]
 [20 25]
 [30 35]] 65
[[20 25]
 [30 35]
 [40 45]] 85
[[30 35]
 [40 45]
 [50 55]] 105
[[40 45]
 [50 55]
 [60 65]] 125
[[50 55]
 [60 65]
 [70 75]] 145
[[60 65]
 [70 75]
 [80 85]] 165
[[70 75]
 [80 85]
 [90 95]] 185
```

Listing 9.35: Example output from splitting a dependent series dataset into samples.

We are now ready to fit an LSTM model on this data. Any of the varieties of LSTMs in the previous section can be used, such as a Vanilla, Stacked, Bidirectional, CNN, or ConvLSTM model. We will use a Vanilla LSTM where the number of time steps and parallel series (features) are specified for the input layer via the `input_shape` argument.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
```

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

Listing 9.36: Example of defining a Vanilla LSTM for modeling a dependent series.

When making a prediction, the model expects three time steps for two input time series. We can predict the next value in the output series providing the input values of:

```
80, 85
90, 95
100, 105
```

Listing 9.37: Example input for making an out-of-sample forecast.

The shape of the one sample with three time steps and two variables must be `[1, 3, 2]`. We would expect the next value in the sequence to be $100 + 105$, or 205.

```
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.38: Example of making an out-of-sample forecast.

The complete example is listed below.

```
# multivariate lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
```

```
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.39: Example of a Vanilla LSTM for multivariate dependent time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[208.13531]]
```

Listing 9.40: Example output from a Vanilla LSTM for multivariate dependent time series forecasting.

For an example of LSTM models developed for a multivariate time series classification problem, see Chapter 25.

9.3.2 Multiple Parallel Series

An alternate time series problem is the case where there are multiple parallel time series and a value must be predicted for each. For example, given the data from the previous section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 9.41: Example of a multivariate parallel time series.

We may want to predict the value for each of the three time series for the next time step. This might be referred to as multivariate forecasting. Again, the data must be split into input/output samples in order to train a model. The first sample of this dataset would be:

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 9.42: Example input from the first sample.

Output:

```
40, 45, 85
```

Listing 9.43: Example output from the first sample.

The `split_sequences()` function below will split multiple parallel time series with rows for time steps and one series per column into the required input/output shape.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.44: Example of a function for splitting parallel series into samples.

We can demonstrate this on the contrived problem; the complete example is listed below.

```
# multivariate output data prep
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.45: Example of splitting a multivariate parallel time series onto samples.

Running the example first prints the shape of the prepared X and y components. The shape of X is three-dimensional, including the number of samples (6), the number of time steps chosen per sample (3), and the number of parallel time series or features (3). The shape of y is two-dimensional as we might expect for the number of samples (6) and the number of time variables per sample to be predicted (3). The data is ready to use in an LSTM model that expects three-dimensional input and two-dimensional output shapes for the X and y components of each sample. Then, each of the samples is printed showing the input and output components of each sample.

```
(6, 3, 3) (6, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [40 45 85]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [ 50 55 105]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [ 60 65 125]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [ 70 75 145]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [ 80 85 165]
[[ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]] [ 90 95 185]
```

Listing 9.46: Example output from splitting a multivariate parallel time series onto samples.

We are now ready to fit an LSTM model on this data. Any of the varieties of LSTMs in the previous section can be used, such as a Vanilla, Stacked, Bidirectional, CNN, or ConvLSTM model. We will use a Stacked LSTM where the number of time steps and parallel series (features) are specified for the input layer via the `input_shape` argument. The number of parallel series is also used in the specification of the number of values to predict by the model in the output layer; again, this is three.

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.47: Example of defining a Stacked LSTM for parallel time series forecasting.

We can predict the next value in each of the three parallel series by providing an input of three time steps for each series.

```
70, 75, 145
80, 85, 165
90, 95, 185
```

Listing 9.48: Example input for making an out-of-sample forecast.

The shape of the input for making a single prediction must be 1 sample, 3 time steps, and 3 features, or `[1, 3, 3]`.

```
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.49: Example of reshaping a sample for making an out-of-sample forecast.

We would expect the vector output to be:

```
[100, 105, 205]
```

Listing 9.50: Example output for an out-of-sample forecast.

We can tie all of this together and demonstrate a Stacked LSTM for multivariate output time series forecasting below.

```
# multivariate output stacked lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
```

```
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=400, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.51: Example of a Stacked LSTM for multivariate parallel time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.76599 108.730484 206.63577 ]]
```

Listing 9.52: Example output from a Stacked LSTM for multivariate parallel time series forecasting.

For an example of LSTM models developed for a multivariate time series forecasting problem, see Chapter 20.

9.4 Multi-step LSTM Models

A time series forecasting problem that requires a prediction of multiple time steps into the future can be referred to as multi-step time series forecasting. Specifically, these are problems where the forecast horizon or interval is more than one time step. There are two main types of LSTM models that can be used for multi-step forecasting; they are:

1. Vector Output Model
2. Encoder-Decoder Model

Before we look at these models, let's first look at the preparation of data for multi-step forecasting.

9.4.1 Data Preparation

As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components. Both the input and output components will be comprised of multiple time steps and may or may not have the same number of steps. For example, given the univariate time series:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 9.53: Example of a univariate time series.

We could use the last three time steps as input and forecast the next two time steps. The first sample would look as follows:

Input:

```
[10, 20, 30]
```

Listing 9.54: Example input from the first sample.

Output:

```
[40, 50]
```

Listing 9.55: Example output from the first sample.

The `split_sequence()` function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.56: Example of a function for splitting a univariate series into samples for multi-step forecasting.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multi-step data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.57: Example of splitting a univariate series for multi-step forecasting into samples.

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

Listing 9.58: Example output from splitting a univariate series for multi-step forecasting into samples.

Now that we know how to prepare data for multi-step forecasting, let's look at some LSTM models that can learn this mapping.

9.4.2 Vector Output Model

Like other types of neural network models, the LSTM can output a vector directly that can be interpreted as a multi-step forecast. This approach was seen in the previous section were one time step of each output time series was forecasted as a vector. As with the LSTMs for univariate data in a prior section, the prepared samples must first be reshaped. The LSTM expects data to have a three-dimensional structure of `[samples, timesteps, features]`, and in this case, we only have one feature so the reshape is straightforward.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 9.59: Example of preparing data for fitting an LSTM model.

With the number of input and output steps specified in the `n_steps_in` and `n_steps_out` variables, we can define a multi-step time-series forecasting model. Any of the presented LSTM model types could be used, such as Vanilla, Stacked, Bidirectional, CNN-LSTM, or ConvLSTM. Below defines a Stacked LSTM for multi-step forecasting.

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps_in,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.60: Example of defining a Stacked LSTM for multi-step forecasting.

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input:

```
[70, 80, 90]
```

Listing 9.61: Example input for making an out-of-sample forecast.

We would expect the predicted output to be:

```
[100, 110]
```

Listing 9.62: Expected output for making an out-of-sample forecast.

As expected by the model, the shape of the single sample of input data when making the prediction must be `[1, 3, 1]` for the 1 sample, 3 time steps of the input, and the single feature.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.63: Example of preparing a sample for making an out-of-sample forecast.

Tying all of this together, the Stacked LSTM for multi-step forecasting with a univariate time series is listed below.

```
# univariate multi-step vector-output stacked lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
```

```
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps_in,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=50, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.64: Example of a Stacked LSTM for multi-step time series forecasting.

Running the example forecasts and prints the next two time steps in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[100.98096 113.28924]]
```

Listing 9.65: Example output from a Stacked LSTM for multi-step time series forecasting.

9.4.3 Encoder-Decoder Model

A model specifically developed for forecasting variable length output sequences is called the Encoder-Decoder LSTM. The model was designed for prediction problems where there are both input and output sequences, so-called sequence-to-sequence, or seq2seq problems, such as translating text from one language to another. This model can be used for multi-step time series forecasting. As its name suggests, the model is comprised of two sub-models: the encoder and the decoder.

The encoder is a model responsible for reading and interpreting the input sequence. The output of the encoder is a fixed length vector that represents the model's interpretation of the sequence. The encoder is traditionally a Vanilla LSTM model, although other encoder models can be used such as Stacked, Bidirectional, and CNN models.

```
# define encoder model
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in, n_features)))
```

Listing 9.66: Example of defining the encoder input model.

The decoder uses the output of the encoder as an input. First, the fixed-length output of the encoder is repeated, once for each required time step in the output sequence.

```
# repeat encoding
model.add(RepeatVector(n_steps_out))
```

Listing 9.67: Example of repeating the encoded vector.

This sequence is then provided to an LSTM decoder model. The model must output a value for each value in the output time step, which can be interpreted by a single output model.

```
# define decoder model
model.add(LSTM(100, activation='relu', return_sequences=True))
```

Listing 9.68: Example of defining the decoder model.

We can use the same output layer or layers to make each one-step prediction in the output sequence. This can be achieved by wrapping the output part of the model in a `TimeDistributed` wrapper.

```
# define model output
model.add(TimeDistributed(Dense(1)))
```

Listing 9.69: Example of defining the output model.

The full definition for an Encoder-Decoder model for multi-step time series forecasting is listed below.

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(100, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.70: Example of defining an Encoder-Decoder LSTM for multi-step forecasting.

As with other LSTM models, the input data must be reshaped into the expected three-dimensional shape of `[samples, timesteps, features]`.

```
# reshape input training data
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 9.71: Example of reshaping input samples for training the Encoder-Decoder LSTM.

In the case of the Encoder-Decoder model, the output, or y part, of the training dataset must also have this shape. This is because the model will predict a given number of time steps with a given number of features for each input sample.

```
# reshape output training data
y = y.reshape((y.shape[0], y.shape[1], n_features))
```

Listing 9.72: Example of reshaping output samples for training the Encoder-Decoder LSTM.

The complete example of an Encoder-Decoder LSTM for multi-step time series forecasting is listed below.

```
# univariate multi-step encoder-decoder lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
y = y.reshape((y.shape[0], y.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(100, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=100, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.73: Example of an Encoder-Decoder LSTM for multi-step time series forecasting.

Running the example forecasts and prints the next two time steps in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[[101.9736
  [116.213615]]]
```

Listing 9.74: Example output from an Encoder-Decoder LSTM for multi-step time series forecasting.

For an example of LSTM models developed for a multi-step time series forecasting problem, see Chapter 20.

9.5 Multivariate Multi-step LSTM Models

In the previous sections, we have looked at univariate, multivariate, and multi-step time series forecasting. It is possible to mix and match the different types of LSTM models presented so far for the different problems. This too applies to time series forecasting problems that involve multivariate and multi-step forecasting, but it may be a little more challenging. In this section, we will provide short examples of data preparation and modeling for multivariate multi-step time series forecasting as a template to ease this challenge, specifically:

1. Multiple Input Multi-step Output.
2. Multiple Parallel Input and Multi-step Output.

Perhaps the biggest stumbling block is in the preparation of data, so this is where we will focus our attention.

9.5.1 Multiple Input Multi-step Output

There are those multivariate time series forecasting problems where the output series is separate but dependent upon the input time series, and multiple time steps are required for the output series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 9.75: Example of a multivariate dependent time series.

We may use three prior time steps of each of the two input time series to predict two time steps of the output time series.

Input:

```
10, 15
20, 25
30, 35
```

Listing 9.76: Example of input from the first sample.

Output:

```
65
85
```

Listing 9.77: Example of output from the first sample.

The `split_sequences()` function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.78: Example of a function for splitting a dependent series for multi-step forecasting into samples.

We can demonstrate this on our contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
```

```
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.79: Example splitting a parallel series for multi-step forecasting into samples.

Running the example first prints the shape of the prepared training data. We can see that the shape of the input portion of the samples is three-dimensional, comprised of six samples, with three time steps, and two variables for the 2 input time series. The output portion of the samples is two-dimensional for the six samples and the two time steps for each sample to be predicted. The prepared samples are then printed to confirm that the data was prepared as we specified.

```
(6, 3, 2) (6, 2)

[[10 15]
 [20 25]
 [30 35]] [65 85]
[[20 25]
 [30 35]
 [40 45]] [ 85 105]
[[30 35]
 [40 45]
 [50 55]] [105 125]
[[40 45]
 [50 55]
 [60 65]] [125 145]
[[50 55]
 [60 65]
 [70 75]] [145 165]
[[60 65]
 [70 75]
 [80 85]] [165 185]
```

Listing 9.80: Example output from splitting a parallel series for multi-step forecasting into samples.

We can now develop an LSTM model for multi-step predictions. A vector output or an encoder-decoder model could be used. In this case, we will demonstrate a vector output with a

Stacked LSTM. The complete example is listed below.

```
# multivariate multi-step stacked lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps_in,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([[70, 75], [80, 85], [90, 95]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.81: Example of an Stacked LSTM for multi-step forecasting for a dependent series.

Running the example fits the model and predicts the next two time steps of the output sequence beyond the dataset. We would expect the next two steps to be: `[185, 205]`. It is a challenging framing of the problem with very little data, and the arbitrarily configured version of the model gets close.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[188.70619 210.16513]]
```

Listing 9.82: Example output from an Stacked LSTM for multi-step forecasting for a dependent series.

9.5.2 Multiple Parallel Input and Multi-step Output

A problem with parallel time series may require the prediction of multiple time steps of each time series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 9.83: Example of a multivariate parallel time series dataset.

We may use the last three time steps from each of the three time series as input to the model and predict the next time steps of each of the three time series as output. The first sample in the training dataset would be the following.

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 9.84: Example of input from the first sample.

Output:

```
40, 45, 85
50, 55, 105
```

Listing 9.85: Example of output from the first sample.

The `split_sequences()` function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.86: Example of a function for splitting a parallel dataset for multi-step forecasting into samples.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
```

```
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.87: Example splitting a parallel series for multi-step forecasting into samples.

Running the example first prints the shape of the prepared training dataset. We can see that both the input (X) and output (Y) elements of the dataset are three dimensional for the number of samples, time steps, and variables or parallel time series respectively. The input and output elements of each series are then printed side by side so that we can confirm that the data was prepared as we expected.

```
(5, 3, 3) (5, 2, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [[ 40 45 85]
 [ 50  55 105]]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [[ 50 55 105]
 [ 60  65 125]]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [[ 60 65 125]
 [ 70  75 145]]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [[ 70 75 145]
 [ 80  85 165]]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [[ 80 85 165]
 [ 90  95 185]]
```

Listing 9.88: Example output from splitting a parallel series for multi-step forecasting into samples.

We can use either the Vector Output or Encoder-Decoder LSTM to model this problem. In this case, we will use the Encoder-Decoder model. The complete example is listed below.

```
# multivariate multi-step encoder-decoder lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
```

```
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(200, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(200, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(n_features)))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=300, verbose=0)
# demonstrate prediction
x_input = array([[60, 65, 125], [70, 75, 145], [80, 85, 165]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.89: Example of an Encoder-Decoder LSTM for multi-step forecasting for parallel series.

Running the example fits the model and predicts the values for each of the three time steps for the next two time steps beyond the end of the dataset. We would expect the values for these series and time steps to be as follows:

```
90, 95, 185
100, 105, 205
```

Listing 9.90: Expected output for an out-of-sample forecast.

We can see that the model forecast gets reasonably close to the expected values.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[[ 91.86044  97.77231 189.66768 ]
  [103.299355 109.18123 212.6863 ]]]
```

Listing 9.91: Example output from an Encoder-Decoder Output LSTM for multi-step forecasting for parallel series.

For an example of LSTM models developed for a multivariate multi-step time series forecasting problem, see Chapter 20.

9.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Problem Differences**. Explain the main changes to the LSTM required when modeling each of univariate, multivariate and multi-step time series forecasting problems.
- **Experiment**. Select one example and modify it to work with your own small contrived dataset.
- **Develop Framework**. Use the examples in this chapter as the basis for a framework for automatically developing an LSTM model for a given time series forecasting problem.

If you explore any of these extensions, I'd love to know.

9.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

9.7.1 Books

- *Deep Learning*, 2016.
  https://amzn.to/2MQyLVZ
- *Deep Learning with Python*, 2017.
  https://amzn.to/2vMRiMe

9.7.2 Papers

- *Long Short-Term Memory*, 1997.
  https://ieeexplore.ieee.org/document/6795963/.
- *Learning to Forget: Continual Prediction with LSTM*, 1999.
  https://ieeexplore.ieee.org/document/818041/
- *Recurrent Nets that Time and Count*, 2000.
  https://ieeexplore.ieee.org/document/861302/
- *LSTM: A Search Space Odyssey*, 2017.
  https://arxiv.org/abs/1503.04069

- *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*, 2015.
  https://arxiv.org/abs/1506.04214v1

9.7.3 APIs

- Keras: The Python Deep Learning library.
  https://keras.io/
- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/
- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/
- Keras Sequential Model API.
  https://keras.io/models/sequential/
- Keras Core Layers API.
  https://keras.io/layers/core/
- Keras Recurrent Layers API.
  https://keras.io/layers/recurrent/

9.8 Summary

In this tutorial, you discovered how to develop a suite of LSTM models for a range of standard time series forecasting problems. Specifically, you learned:

- How to develop LSTM models for univariate time series forecasting.
- How to develop LSTM models for multivariate time series forecasting.
- How to develop LSTM models for multi-step time series forecasting.

9.8.1 Next

This is the final lesson of this part, the next part will focus on systematically developing models for univariate time series forecasting problems.

Part IV

Univariate Forecasting

Overview

This part highlights that classical methods are known to out-perform more sophisticated methods on univariate time series forecasting problems on average and how to systematically evaluate classical and deep learning methods to ensure that you are getting the most out of them on your forecasting problem. As such, the chapters in this section focus on providing you the tools for grid searching classical and deep learning methods and demonstrating these tools on a suite of standard univariate datasets. After reading the chapters in this part, you will know:

- The results of a recent and reasonably conclusive study that classical methods out-perform machine learning and deep learning methods for univariate time series forecasting problems on average (Chapter 10).
- How to develop and grid search a suite of naive forecasting methods for univariate time series, the results of which can be used as a baseline for determining whether a model has skill (Chapter 11).
- How to develop and grid search exponential smoothing forecasting models, also known as ETS, for univariate time series forecasting problems (Chapter 12).
- How to develop and grid search Seasonal ARIMA models or SARIMA for univariate time series forecasting problems (Chapter 13).
- How to develop and evaluate MLP, CNN and LSTM deep learning models for univariate time series forecasting (Chapter 14).
- How to grid search the hyperparameters for MLP, CNN and LSTM deep learning models for univariate time series forecasting (Chapter 15).

Chapter 10

Review of Top Methods For Univariate Time Series Forecasting

Machine learning and deep learning methods are often reported to be the key solution to all predictive modeling problems. An important recent study evaluated and compared the performance of many classical and modern machine learning and deep learning methods on a large and diverse set of more than 1,000 univariate time series forecasting problems. The results of this study suggest that simple classical methods, such as linear methods and exponential smoothing, outperform complex and sophisticated methods, such as decision trees, Multilayer Perceptrons (MLP), and Long Short-Term Memory (LSTM) network models. These findings highlight the requirement to both evaluate classical methods and use their results as a baseline when evaluating any machine learning and deep learning methods for time series forecasting in order to demonstrate that their added complexity is adding skill to the forecast.

In this tutorial, you will discover the important findings of this recent study evaluating and comparing the performance of a classical and modern machine learning methods on a large and diverse set of time series forecasting datasets. After reading this tutorial, you will know:

- Classical methods like ETS and ARIMA out-perform machine learning and deep learning methods for one-step forecasting on univariate datasets.
- Classical methods like Theta and ARIMA out-perform machine learning and deep learning methods for multi-step forecasting on univariate datasets.
- Machine learning and deep learning methods do not yet deliver on their promise for univariate time series forecasting, and there is much work to do.

Let's get started.

10.1 Overview

Spyros Makridakis, et al. published a study in 2018 titled *Statistical and Machine Learning forecasting methods: Concerns and ways forward*. In this tutorial, we will take a close look at the study by Makridakis, et al. that carefully evaluated and compared classical time series forecasting methods to the performance of modern machine learning methods. This tutorial is divided into seven sections; they are:

1. Study Motivation
2. Time Series Datasets
3. Time Series Forecasting Methods
4. Data Preparation
5. One-step Forecasting Results
6. Multi-step Forecasting Results
7. Outcomes

10.2 Study Motivation

The goal of the study was to clearly demonstrate the capability of a suite of different machine learning methods as compared to classical time series forecasting methods on a very large and diverse collection of univariate time series forecasting problems. The study was a response to the increasing number of papers and claims that machine learning and deep learning methods offer superior results for time series forecasting with little objective evidence.

> Literally hundreds of papers propose new ML algorithms, suggesting methodological advances and accuracy improvements. Yet, limited objective evidence is available regarding their relative performance as a standard forecasting tool.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

The authors clearly lay out three issues with the flood of claims; they are:

- Their conclusions are based on a few, or even a single time series, raising questions about the statistical significance of the results and their generalization.
- The methods are evaluated for short-term forecasting horizons, often one-step-ahead, not considering medium and long-term ones.
- No benchmarks are used to compare the accuracy of ML methods versus alternative ones.

As a response, the study includes eight classical methods and 10 machine learning methods evaluated using one-step and multiple-step forecasts across a collection of 1,045 monthly time series. Although not definitive, the results are intended to be objective and robust.

10.3 Time Series Datasets

The time series datasets used in the study were drawn from the time series datasets used in the M3-Competition. The M3-Competition was the third in a series of competitions that sought to discover exactly what algorithms perform well in practice on real time series forecasting problems. The results of the competition were published in the 2000 paper titled *The M3-Competition: Results, Conclusions and Implications*. The datasets used in the competition were drawn from a wide range of industries and had a range of different time intervals, from hourly to annual.

> The 3003 series of the M3-Competition were selected on a quota basis to include various types of time series data (micro, industry, macro, etc.) and different time intervals between successive observations (yearly, quarterly, etc.).
>
> — *The M3-Competition: Results, Conclusions and Implications*, 2000.

The table below, taken from the paper, provides a summary of the 3,003 datasets used in the competition.

Time interval between successive observations	Types of time series data						
	Micro	Industry	Macro	Finance	Demographic	Other	Total
Yearly	146	102	83	58	245	11	645
Quarterly	204	83	336	76	57		756
Monthly	474	334	312	145	111	52	1428
Other	4			29		141	174
Total	828	519	731	308	413	204	3003

Figure 10.1: Table of Datasets, Industry and Time Interval Used in the M3-Competition. Taken from *The M3-Competition: Results, Conclusions and Implications*.

The finding of the competition was that simpler time series forecasting methods outperform more sophisticated methods, including neural network models.

> This study, the previous two M-Competitions and many other empirical studies have proven, beyond the slightest doubt, that elaborate theoretical constructs or more sophisticated methods do not necessarily improve post-sample forecasting accuracy, over simple methods, although they can better fit a statistical model to the available historical data.
>
> — *The M3-Competition: Results, Conclusions and Implications*, 2000.

The more recent study that we are reviewing in this tutorial that evaluate machine learning methods selected a subset of 1,045 time series with a monthly interval from those used in the M3 competition.

> ... evaluate such performance across multiple forecasting horizons using a large subset of 1045 monthly time series used in the M3 Competition.
>
> — *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

10.4 Time Series Forecasting Methods

The study evaluates the performance of eight classical (or simpler) methods and 10 machine learning methods.

> ... of eight traditional statistical methods and eight popular ML ones, [...], plus two more that have become popular during recent years.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

The eight classical methods evaluated were as follows:

- Naive 2, which is actually a random walk model adjusted for season.
- Simple Exponential Smoothing.
- Holt.
- Damped exponential smoothing.
- Average of SES, Holt, and Damped.
- Theta method.
- ARIMA, automatic.
- ETS, automatic.

A total of eight machine learning methods were used in an effort to reproduce and compare to results presented in the 2010 paper *An Empirical Comparison of Machine Learning Models for Time Series Forecasting*. They were:

- Multilayer Perceptron (MLP).
- Bayesian Neural Network (BNN).
- Radial Basis Functions (RBF).
- Generalized Regression Neural Networks (GRNN), also called kernel regression.
- k-Nearest Neighbor regression (KNN).
- CART regression trees (CART).
- Support Vector Regression (SVR).
- Gaussian Processes (GP).

An additional two *modern* neural network algorithms were also added to the list given the recent rise in their adoption; they were:

- Recurrent Neural Network (RNN).
- Long Short-Term Memory (LSTM).

10.5 Data Preparation

A careful data preparation methodology was used, again, based on the methodology described in the 2010 paper *An Empirical Comparison of Machine Learning Models for Time Series Forecasting*. In that paper, each time series was adjusted using a power transform, deseasonalized and detrended.

> ... before computing the 18 forecasts, they preprocessed the series in order to achieve stationarity in their mean and variance. This was done using the log transformation, then deseasonalization and finally scaling, while first differences were also considered for removing the component of trend.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

Inspired by these operations, variations of five different data transforms were applied for an MLP for one-step forecasting and their results were compared. The five transforms were:

- Original data.
- Box-Cox Power Transform.
- Deseasonalizing the data.
- Detrending the data.
- All three transforms (power, deseasonalize, detrend).

Generally, it was found that the best approach was to apply a power transform and deseasonalize the data, and perhaps detrend the series as well.

> The best combination according to sMAPE is number 7 (Box-Cox transformation, deseasonalization) while the best one according to MASE is number 10 (Box-Cox transformation, deseasonalization and detrending)

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

10.6 One-step Forecasting Results

All models were evaluated using one-step time series forecasting. Specifically, the last 18 time steps were used as a test set, and models were fit on all remaining observations. A separate one-step forecast was made for each of the 18 observations in the test set, presumably using a walk-forward validation method where true observations were used as input in order to make each forecast.

> The forecasting model was developed using the first n - 18 observations, where n is the length of the series. Then, 18 forecasts were produced and their accuracy was evaluated compared to the actual values not used in developing the forecasting model.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018

Reviewing the results, the MLP and BNN were found to achieve the best performance from all of the machine learning methods.

> The results [...] show that MLP and BNN outperform the remaining ML methods.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

An interesting result was that RNNs and LSTMs were found to perform poorly. This confirms a earlier finding in the application of LSTMs to univariate time series forecast reported by Gers, et al. in 2001 titled *Applying LSTM to Time Series Predictable through Time-Window Approaches*. In that study, they also reported that LSTMs are easily outperformed by simpler methods and may not be suited to simple autoregressive-type univariate time series forecasting problems.

> It should be noted that RNN is among the less accurate ML methods, demonstrating that research progress does not necessarily guarantee improvements in forecasting performance. This conclusion also applies in the performance of LSTM, another popular and more advanced ML method, which does not enhance forecasting accuracy too.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

Comparing the performance of all methods, it was found that the machine learning methods were all out-performed by simple classical methods, where ETS and ARIMA models performed the best overall. This finding confirms the results from previous similar studies and competitions.

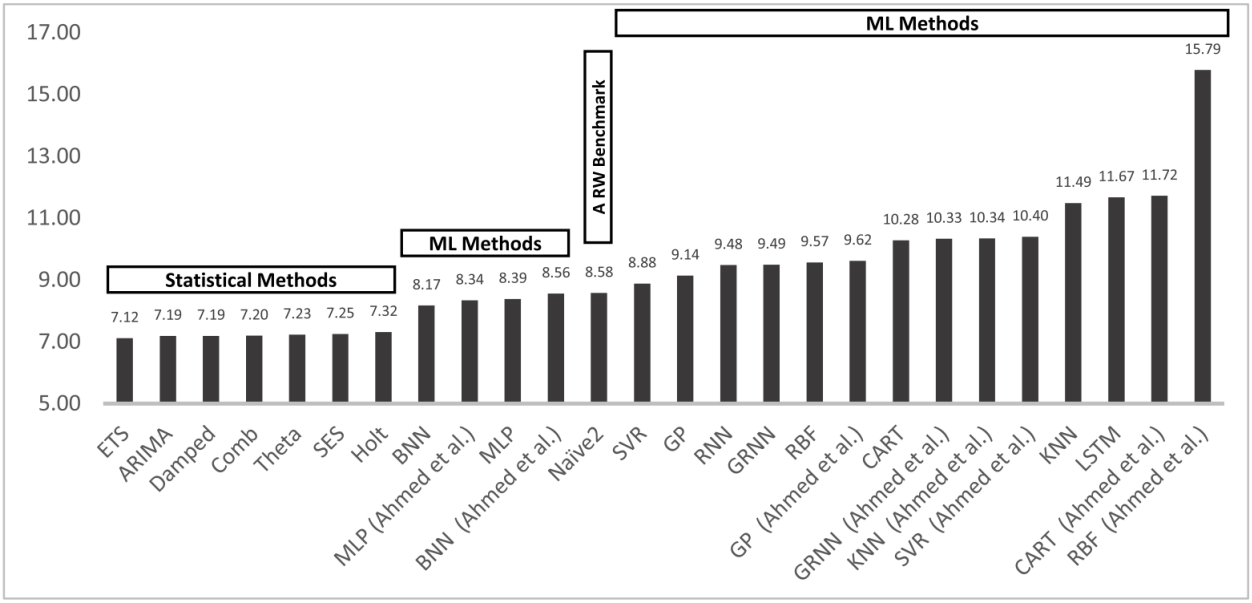

Figure 10.2: Bar Chart Comparing Model Performance (sMAPE) for One-step Forecasts. Taken from *Statistical and Machine Learning forecasting methods: Concerns and ways forward*.

10.7 Multi-step Forecasting Results

Multi-step forecasting involves predicting multiple steps ahead of the last known observation. Three approaches to multi-step forecasting were evaluated for the machine learning methods; they were:

- Iterative forecasting
- Direct forecasting
- Multi-neural network forecasting

The classical methods were found to outperform the machine learning methods again. In this case, methods such as Theta, ARIMA, and a combination of exponential smoothing (Comb) were found to achieve the best performance.

> In brief, statistical models seem to generally outperform ML methods across all forecasting horizons, with Theta, Comb and ARIMA being the dominant ones among the competitors according to both error metrics examined.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

10.8 Outcomes

The study provides important supporting evidence that classical methods may dominate univariate time series forecasting, at least on the types of forecasting problems evaluated. The study demonstrates the worse performance and the increase in computational cost of machine learning and deep learning methods for univariate time series forecasting for both one-step and multi-step forecasts.

These findings strongly encourage the use of classical methods, such as ETS, ARIMA, and others as a first step before more elaborate methods are explored, and requires that the results from these simpler methods be used as a baseline in performance that more elaborate methods must clear in order to justify their usage. It also highlights the need to not just consider the careful use of data preparation methods, but to actively test multiple different combinations of data preparation schemes for a given problem in order to discover what works best, even in the case of classical methods.

Machine learning and deep learning methods may still achieve better performance on specific univariate time series problems and should be evaluated. The study does not look at more complex time series problems, such as those datasets with:

- Complex irregular temporal structures.
- Missing observations
- Heavy noise.
- Complex interrelationships between multiple variates.

The study concludes with an honest puzzlement at why machine learning methods perform so poorly in practice, given their impressive performance in other areas of artificial intelligence.

> The most interesting question and greatest challenge is to find the reasons for their poor performance with the objective of improving their accuracy and exploiting their huge potential. AI learning algorithms have revolutionized a wide range of applications in diverse fields and there is no reason that the same cannot be achieved with the ML methods in forecasting. Thus, we must find how to be applied to improve their ability to forecast more accurately.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

Comments are made by the authors regarding LSTMs and RNNs, that are generally believed to be the deep learning approach for sequence prediction problems in general, and in this case their clearly poor performance in practice.

> ... one would expect RNN and LSTM, which are more advanced types of NNs, to be far more accurate than the ARIMA and the rest of the statistical methods utilized.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

They comment that LSTMs appear to be more suited at fitting or overfitting the training dataset rather than forecasting it.

> Another interesting example could be the case of LSTM that compared to simpler NNs like RNN and MLP, report better model fitting but worse forecasting accuracy

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

There is work to do and machine learning methods and deep learning methods hold the promise of better learning time series data than classical statistical methods, and even doing so directly on the raw observations via automatic feature learning.

> Given their ability to learn, ML methods should do better than simple benchmarks, like exponential smoothing. Accepting the problem is the first step in devising workable solutions and we hope that those in the field of AI and ML will accept the empirical findings and work to improve the forecasting accuracy of their methods.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

10.9 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Develop Methodology**. Develop a methodology for working through a univariate time series forecasting problem (e.g. what algorithms in what order) based on the results presented in this lesson and the referenced study.

- **Find Example**. Research and find an example of a research paper that presents the results of a deep learning model for time series foresting without comparing it to a naive forecast or a classical method.
- **Additional Limitations**. Research and find another research paper that lists the limitations of machine learning or deep learning methods for univariate time series forecasting problems.

If you explore any of these extensions, I'd love to know.

10.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Makridakis Competitions, Wikipedia.
  https://en.wikipedia.org/wiki/Makridakis_Competitions
- *The M3-Competition: Results, Conclusions and Implications*, 2000.
  https://www.sciencedirect.com/science/article/pii/S0169207000000571
- *The M4 Competition: Results, findings, conclusion and way forward*, 2018.
  https://www.sciencedirect.com/science/article/pii/S0169207018300785
- *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.
  http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0194889
- *An Empirical Comparison of Machine Learning Models for Time Series Forecasting*, 2010.
  https://www.tandfonline.com/doi/abs/10.1080/07474938.2010.481556
- *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001.
  https://link.springer.com/chapter/10.1007/3-540-44668-0_93

10.11 Summary

In this tutorial, you discovered the important findings of a recent study evaluating and comparing the performance of classical and modern machine learning methods on a large and diverse set of time series forecasting datasets. Specifically, you learned:

- Classical methods like ETS and ARIMA out-perform machine learning and deep learning methods for one-step forecasting on univariate datasets.
- Classical methods like Theta and ARIMA out-perform machine learning and deep learning methods for multi-step forecasting on univariate datasets.
- Machine learning and deep learning methods do not yet deliver on their promise for univariate time series forecasting and there is much work to do.

10.11.1 Next

In the next lesson, you will discover how to develop robust naive forecasting models for univariate time series forecasting problems.

Chapter 11

How to Develop Simple Methods for Univariate Forecasting

Simple forecasting methods include naively using the last observation as the prediction or an average of prior observations. It is important to evaluate the performance of simple forecasting methods on univariate time series forecasting problems before using more sophisticated methods as their performance provides a lower-bound and point of comparison that can be used to determine of a model has skill or not for a given problem. Although simple, methods such as the naive and average forecast strategies can be tuned to a specific problem in terms of the choice of which prior observation to persist or how many prior observations to average. Often, tuning the hyperparameters of these simple strategies can provide a more robust and defensible lower bound on model performance, as well as surprising results that may inform the choice and configuration of more sophisticated methods.

In this tutorial, you will discover how to develop a framework from scratch for grid searching simple naive and averaging strategies for time series forecasting with univariate data. After completing this tutorial, you will know:

- How to develop a framework for grid searching simple models from scratch using walk-forward validation.
- How to grid search simple model hyperparameters for daily time series data for births.
- How to grid search simple model hyperparameters for monthly time series data for shampoo sales, car sales, and temperature.

Let's get started.

11.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Simple Forecasting Strategies
2. Develop a Grid Search Framework
3. Case Study 1: No Trend or Seasonality

4. Case Study 2: Trend
5. Case Study 3: Seasonality
6. Case Study 4: Trend and Seasonality

11.2 Simple Forecasting Strategies

It is important and useful to test simple forecast strategies prior to testing more complex models. Simple forecast strategies are those that assume little or nothing about the nature of the forecast problem and are fast to implement and calculate. The results can be used as a baseline in performance and used as a point of a comparison. If a model can perform better than the performance of a simple forecast strategy, then it can be said to be skillful. There are two main themes to simple forecast strategies; they are:

- **Naive**, or using observations values directly.
- **Average**, or using a statistic calculated on previous observations.

For more information on simple forecasting strategies, see Chapter 5.

11.3 Develop a Grid Search Framework

In this section, we will develop a framework for grid searching the two simple forecast strategies described in the previous section, namely the naive and average strategies. We can start off by implementing a naive forecast strategy. For a given dataset of historical observations, we can persist any value in that history, that is from the previous observation at index -1 to the first observation in the history at `-(len(data))`. The `naive_forecast()` function below implements the naive forecast strategy for a given offset from 1 to the length of the dataset.

```
# one-step naive forecast
def naive_forecast(history, n):
  return history[-n]
```

Listing 11.1: Example of a function for making a persistence forecast.

We can test this function out on a small contrived dataset.

```
# example of a one-step naive forecast
def naive_forecast(history, n):
  return history[-n]

# define dataset
data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
print(data)
# test naive forecast
for i in range(1, len(data)+1):
  print(naive_forecast(data, i))
```

Listing 11.2: Example of making a persistence forecast.

Running the example first prints the contrived dataset, then the naive forecast for each offset in the historical dataset.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
100.0
90.0
80.0
70.0
60.0
50.0
40.0
30.0
20.0
10.0
```

Listing 11.3: Example output from making a persistence forecast.

We can now look at developing a function for the average forecast strategy. Averaging the last n observations is straight-forward; for example:

```
# mean forecast
from numpy import mean
result = mean(history[-n:])
```

Listing 11.4: Example of averaging prior observations.

We may also want to test out the median in those cases where the distribution of observations is non-Gaussian.

```
# median forecast
from numpy import median
result = median(history[-n:])
```

Listing 11.5: Example of calculating the median prior observations.

The `average_forecast()` function below implements this taking the historical data and a config array or tuple that specifies the number of prior values to average as an integer, and a string that describe the way to calculate the average (`mean` or `median`).

```
# one-step average forecast
def average_forecast(history, config):
  n, avg_type = config
  # mean of last n values
  if avg_type is 'mean':
    return mean(history[-n:])
  # median of last n values
  return median(history[-n:])
```

Listing 11.6: Example of a function for calculating an average forecast.

The complete example on a small contrived dataset is listed below.

```
# example of an average forecast
from numpy import mean
from numpy import median

# one-step average forecast
def average_forecast(history, config):
```

```
  n, avg_type = config
  # mean of last n values
  if avg_type is 'mean':
    return mean(history[-n:])
  # median of last n values
  return median(history[-n:])

# define dataset
data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
print(data)
# test naive forecast
for i in range(1, len(data)+1):
  print(average_forecast(data, (i, 'mean')))
```

Listing 11.7: Example of making an average forecast.

Running the example forecasts the next value in the series as the mean value from contiguous subsets of prior observations from -1 to -10, inclusively.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
100.0
95.0
90.0
85.0
80.0
75.0
70.0
65.0
60.0
55.0
```

Listing 11.8: Example output from making an average forecast.

We can update the function to support averaging over seasonal data, respecting the seasonal offset. An offset argument can be added to the function that when not set to 1 will determine the number of prior observations backwards to count before collecting values from which to include in the average. For example, if n=1 and offset=3, then the average is calculated from the single value at $n \times$ `offset` or $1 \times 3 = -3$. If $n = 2$ and `offset` $= 3$, then the average is calculated from the values at 1×3 or -3 and 2×3 or -6. We can also add some protection to raise an exception when a seasonal configuration ($n \times$ offset) extends beyond the end of the historical observations. The updated function is listed below.

```
# one-step average forecast
def average_forecast(history, config):
  n, offset, avg_type = config
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
```

```
  # mean of last n values
  if avg_type is 'mean':
    return mean(values)
  # median of last n values
  return median(values)
```

Listing 11.9: Example of a function for calculating an average forecast with support for seasonality.

We can test out this function on a small contrived dataset with a seasonal cycle. The complete example is listed below.

```
# example of an average forecast for seasonal data
from numpy import mean
from numpy import median

# one-step average forecast
def average_forecast(history, config):
  n, offset, avg_type = config
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # mean of last n values
  if avg_type is 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# define dataset
data = [10.0, 20.0, 30.0, 10.0, 20.0, 30.0, 10.0, 20.0, 30.0]
print(data)
# test naive forecast
for i in [1, 2, 3]:
  print(average_forecast(data, (i, 3, 'mean')))
```

Listing 11.10: Example of making an average forecast with seasonality.

Running the example calculates the mean values of `[10]`, `[10, 10]` and `[10, 10, 10]`.

```
[10.0, 20.0, 30.0, 10.0, 20.0, 30.0, 10.0, 20.0, 30.0]
10.0
10.0
10.0
```

Listing 11.11: Example output from making an average forecast with seasonality.

It is possible to combine both the naive and the average forecast strategies together into the same function. There is a little overlap between the methods, specifically the n-offset into the history that is used to either persist values or determine the number of values to average.

It is helpful to have both strategies supported by one function so that we can test a suite of configurations for both strategies at once as part of a broader grid search of simple models. The `simple_forecast()` function below combines both strategies into a single function.

```
# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)
```

Listing 11.12: Example of a function that combines persistence and average forecasts.

Next, we need to build up some functions for fitting and evaluating a model repeatedly via walk-forward validation, including splitting a dataset into train and test sets and evaluating one-step forecasts. We can split a list or NumPy array of data using a slice given a specified size of the split, e.g. the number of time steps to use from the data in the test set. The `train_test_split()` function below implements this for a provided dataset and a specified number of time steps to use in the test set.

```
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

Listing 11.13: Example of a function for splitting data into train and test sets.

After forecasts have been made for each step in the test dataset, they need to be compared to the test set in order to calculate an error score. There are many popular error scores for time series forecasting. In this case, we will use root mean squared error (RMSE), but you can change this to your preferred measure, e.g. MAPE, MAE, etc. The `measure_rmse()` function below will calculate the RMSE given a list of actual (the test set) and predicted values.

```
# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))
```

Listing 11.14: Example of a function for calculating RMSE.

We can now implement the walk-forward validation scheme. This is a standard approach to evaluating a time series forecasting model that respects the temporal ordering of observations. First, a provided univariate time series dataset is split into train and test sets using the `train_test_split()` function. Then the number of observations in the test set are enumerated. For each we fit a model on all of the history and make a one step forecast. The true observation for the time step is then added to the history, and the process is repeated. The `simple_forecast()` function is called in order to fit a model and make a prediction. Finally, an error score is calculated by comparing all one-step forecasts to the actual test set by calling the `measure_rmse()` function. The `walk_forward_validation()` function below implements this, taking a univariate time series, a number of time steps to use in the test set, and an array of model configuration.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error
```

Listing 11.15: Example of a function for performing walk-forward validation.

If you are interested in making multi-step predictions, you can change the call to `predict()` in the `simple_forecast()` function and also change the calculation of error in the `measure_rmse()` function. We can call `walk_forward_validation()` repeatedly with different lists of model configurations. One possible issue is that some combinations of model configurations may not be called for the model and will throw an exception.

We can trap exceptions and ignore warnings during the grid search by wrapping all calls to `walk_forward_validation()` with a try-except and a block to ignore warnings. We can also add debugging support to disable these protections in the case we want to see what is really going on. Finally, if an error does occur, we can return a `None` result; otherwise, we can print some information about the skill of each model evaluated. This is helpful when a large number of models are evaluated. The `score_model()` function below implements this and returns a tuple of (key and result), where the key is a string version of the tested model configuration.

```
# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
```

```
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)
```

Listing 11.16: Example of a function the robust evaluation of a model.

Next, we need a loop to test a list of different model configurations. This is the main function that drives the grid search process and will call the `score_model()` function for each model configuration. We can dramatically speed up the grid search process by evaluating model configurations in parallel. One way to do that is to use the Joblib library[1]. We can define a Parallel object with the number of cores to use and set it to the number of scores detected in your hardware.

```
# define executor
executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
```

Listing 11.17: Example of preparing a Joblib executor.

We can then create a list of tasks to execute in parallel, which will be one call to the `score_model()` function for each model configuration we have.

```
# define list of tasks
tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
```

Listing 11.18: Example of preparing a task list for a Joblib executor.

Finally, we can use the Parallel object to execute the list of tasks in parallel.

```
# execute list of tasks
scores = executor(tasks)
```

Listing 11.19: Example of running a Joblib executor.

On some systems, such as windows that do not support the `fork()` function, it is necessary to add a check to ensure that the entry point of the script is only executed by the main process and not child processes.

```
...
if __name__ == '__main__':
  ...
```

Listing 11.20: Example of wrapping the entry point into the script in a check for the main process.

[1]Note, you may have to install Joblib: `pip install joblib`

That's it. We can also provide a non-parallel version of evaluating all model configurations in case we want to debug something.

```
# execute list of tasks sequentially
scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
```

Listing 11.21: Example of evaluating a suite of configurations in sequence.

The result of evaluating a list of configurations will be a list of tuples, each with a name that summarizes a specific model configuration and the error of the model evaluated with that configuration as either the RMSE or `None` if there was an error. We can filter out all scores set to `None`.

```
# order scores
scores = [r for r in scores if r[1] != None]
```

Listing 11.22: Example of filtering out scores for invalid configurations.

We can then sort all tuples in the list by the score in ascending order (best are first), then return this list of scores for review. The `grid_search()` function below implements this behavior given a univariate time series dataset, a list of model configurations (list of lists), and the number of time steps to use in the test set. An optional parallel argument allows the evaluation of models across all cores to be tuned on or off, and is on by default.

```
# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores
```

Listing 11.23: Example of a function for grid searching configurations.

We're nearly done. The only thing left to do is to define a list of model configurations to try for a dataset. We can define this generically. The only parameter we may want to specify is the periodicity of the seasonal component in the series (offset), if one exists. By default, we will assume no seasonal component. The `simple_configs()` function below will create a list of model configurations to evaluate. The function only requires the maximum length of the historical data as an argument and optionally the periodicity of any seasonal component, which is defaulted to 1 (no seasonal component).

```
# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
```

```
        cfg = [i, o, t]
        configs.append(cfg)
  return configs
```

Listing 11.24: Example of a function for defining simple forecast configurations to grid search.

We now have a framework for grid searching simple model hyperparameters via one-step walk-forward validation. It is generic and will work for any in-memory univariate time series provided as a list or NumPy array. We can make sure all the pieces work together by testing it on a contrived 10-step dataset. The complete example is listed below.

```
# grid search simple forecasts
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error

# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
```

```
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # define dataset
  data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
  # data split
  n_test = 4
  # model configs
  max_length = len(data) - n_test
  cfg_list = simple_configs(max_length)
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 11.25: Example of demonstrating the grid search infrastructure.

Running the example first prints the contrived time series dataset. Next, the model configurations and their errors are reported as they are evaluated. Finally, the configurations and the error for the top three configurations are reported. We can see that the persistence model with a configuration of 1 (e.g. persist the last observation) achieves the best performance of the simple models tested, as would be expected.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]

...
> Model[[4, 1, 'mean']] 25.000
> Model[[3, 1, 'median']] 20.000
> Model[[6, 1, 'mean']] 35.000
> Model[[5, 1, 'median']] 30.000
> Model[[6, 1, 'median']] 35.000
done

[1, 1, 'persist'] 10.0
[2, 1, 'mean'] 15.0
[2, 1, 'median'] 15.0
```

Listing 11.26: Example output from demonstrating the grid search infrastructure.

Now that we have a robust framework for grid searching simple model hyperparameters, let's test it out on a suite of standard univariate time series datasets. All datasets in this tutorial were drawn from the Time Series Dataset Library on the DataMarket website[2]. The results

[2] https://datamarket.com/data/list/?q=provider:tsdl

demonstrated on each dataset provide a baseline of performance that can be used to compare more sophisticated methods, such as SARIMA, ETS, and even machine learning methods.

11.4 Case Study 1: No Trend or Seasonality

The *daily female births* dataset summarizes the daily total female births in California, USA in 1959. You can download the dataset directly from here:

- daily-total-female-births.csv [3]

Save the file with the filename `daily-total-female-births.csv` in your current working directory. We can load this dataset as a Pandas `Series` using the function `read_csv()` and summarize the shape of the dataset.

```
# load
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
```

Listing 11.27: Example of loading the daily female births dataset.

We can then create a line plot of the series and inspect it for systematic structures like trends and seasonality.

```
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.28: Example of plotting the daily female births dataset.

The complete example is listed below.

```
# load and plot daily births dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.29: Example of loading and plotting the daily female births dataset.

Running the example first summarizes the shape of the loaded dataset. The dataset has one year, or 365 observations. We will use the first 200 for training and the remaining 165 as the test set.

[3] https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-total-female-births.csv

```
(365, 1)
```

Listing 11.30: Example output from loading and summarizing the shape of the daily female births dataset

A line plot of the series is also created. We can see that there is no obvious trend or seasonality.

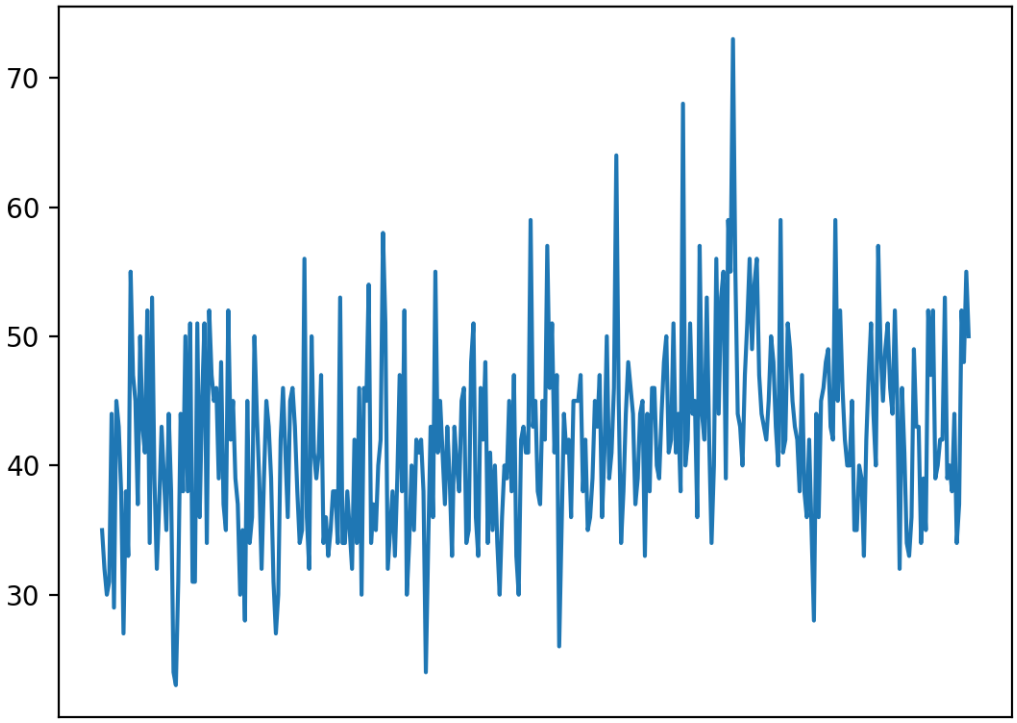

Figure 11.1: Line Plot of the Daily Female Births Dataset.

We can now grid search naive models for the dataset. The complete example grid searching the daily female univariate time series forecasting problem is listed below.

```
# grid search simple forecast for daily female births
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
from pandas import read_csv
```

```
# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
```

```
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # define dataset
  series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 165
  # model configs
  max_length = len(data) - n_test
  cfg_list = simple_configs(max_length)
  # grid search
```

```
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 11.31: Example of grid searching naive models for the daily female births dataset.

Running the example prints the model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run.

```
...
> Model[[186, 1, 'mean']] 7.523
> Model[[200, 1, 'median']] 7.681
> Model[[186, 1, 'median']] 7.691
> Model[[187, 1, 'persist']] 11.137
> Model[[187, 1, 'mean']] 7.527
done

[22, 1, 'mean'] 6.930411499775709
[23, 1, 'mean'] 6.932293117115201
[21, 1, 'mean'] 6.951918385845375
```

Listing 11.32: Example output from grid searching naive models for the daily female births dataset.

We can see that the best result was an RMSE of about 6.93 births with the following configuration:

- **Strategy**: Average
- **n**: 22
- **function**: mean()

This is surprising given the lack of trend or seasonality, I would have expected either a persistence of -1 or an average of the entire historical dataset to result in the best performance.

11.5 Case Study 2: Trend

The *monthly shampoo sales* dataset summarizes the monthly sales of shampoo over a three-year period. You can download the dataset directly from here:

- monthly-shampoo-sales.csv [4]

Save the file with the filename `monthly-shampoo-sales.csv` in your current working directory. We can load this dataset as a Pandas `Series` using the function `read_csv()` and summarize the shape of the dataset.

[4] https://raw.githubusercontent.com/jbrownlee/Datasets/master/shampoo.csv

```
# load
series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
```

Listing 11.33: Example of loading the monthly shampoo sales dataset.

We can then create a line plot of the series and inspect it for systematic structures like trends and seasonality.

```
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.34: Example of plotting the monthly shampoo sales dataset.

The complete example is listed below.

```
# load and plot monthly shampoo sales dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.35: Example of loading and plotting the monthly shampoo sales dataset.

Running the example first summarizes the shape of the loaded dataset. The dataset has three years, or 36 observations. We will use the first 24 for training and the remaining 12 as the test set.

```
(36, 1)
```

Listing 11.36: Example output from loading and summarizing the shape of the monthly shampoo sales dataset

A line plot of the series is also created. We can see that there is an obvious trend and no obvious seasonality.

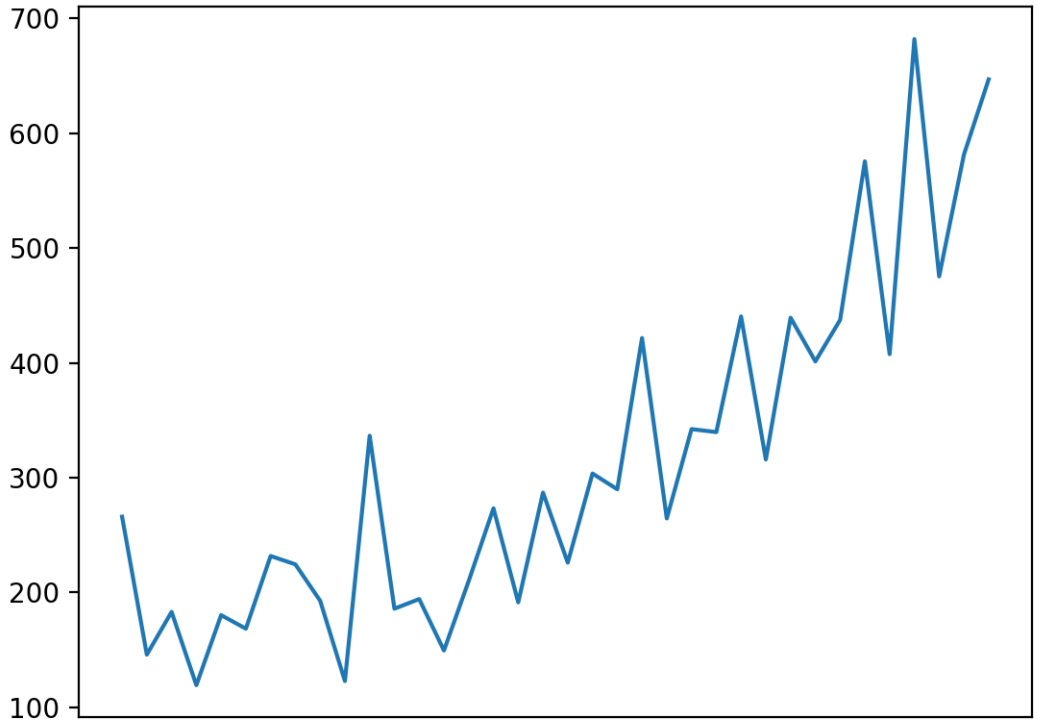

Figure 11.2: Line Plot of the Monthly Shampoo Sales Dataset.

We can now grid search naive models for the dataset. The complete example grid searching the shampoo sales univariate time series forecasting problem is listed below.

```
# grid search simple forecast for monthly shampoo sales
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
```

```
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
```

```
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
  max_length = len(data) - n_test
  cfg_list = simple_configs(max_length)
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 11.37: Example of grid searching naive models for the monthly shampoo sales dataset.

Running the example prints the configurations and the RMSE are printed as the models are

evaluated. The top three model configurations and their error are reported at the end of the run.

```
...
> Model[[23, 1, 'mean']] 209.782
> Model[[23, 1, 'median']] 221.863
> Model[[24, 1, 'persist']] 305.635
> Model[[24, 1, 'mean']] 213.466
> Model[[24, 1, 'median']] 226.061
done

[2, 1, 'persist'] 95.69454007413378
[2, 1, 'mean'] 96.01140340258198
[2, 1, 'median'] 96.01140340258198
```

Listing 11.38: Example output from grid searching naive models for the monthly shampoo sales dataset.

We can see that the best result was an RMSE of about 95.69 sales with the following configuration:

- **Strategy**: Persist
- **n**: 2

This is surprising as the trend structure of the data would suggest that persisting the previous value (-1) would be the best approach, not persisting the second last value.

11.6 Case Study 3: Seasonality

The *monthly mean temperatures* dataset summarizes the monthly average air temperatures in Nottingham Castle, England from 1920 to 1939 in degrees Fahrenheit. You can download the dataset directly from here:

- monthly-mean-temp.csv [5]

Save the file with the filename `monthly-mean-temp.csv` in your current working directory. We can load this dataset as a Pandas `Series` using the function `read_csv()` and summarize the shape of the dataset.

```
# load
series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
```

Listing 11.39: Example of loading the monthly mean temperature dataset.

We can then create a line plot of the series and inspect it for systematic structures like trends and seasonality.

[5] https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-mean-temp.csv

```
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.40: Example of plotting the monthly mean temperature dataset.

The complete example is listed below.

```
# load and plot monthly mean temp dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.41: Example of loading and plotting the monthly mean temperature dataset.

Running the example first summarizes the shape of the loaded dataset. The dataset has 20 years, or 240 observations.

```
(240, 1)
```

Listing 11.42: Example output from loading and summarizing the shape of the monthly mean temperature dataset

A line plot of the series is also created. We can see that there is no obvious trend and an obvious seasonality structure.

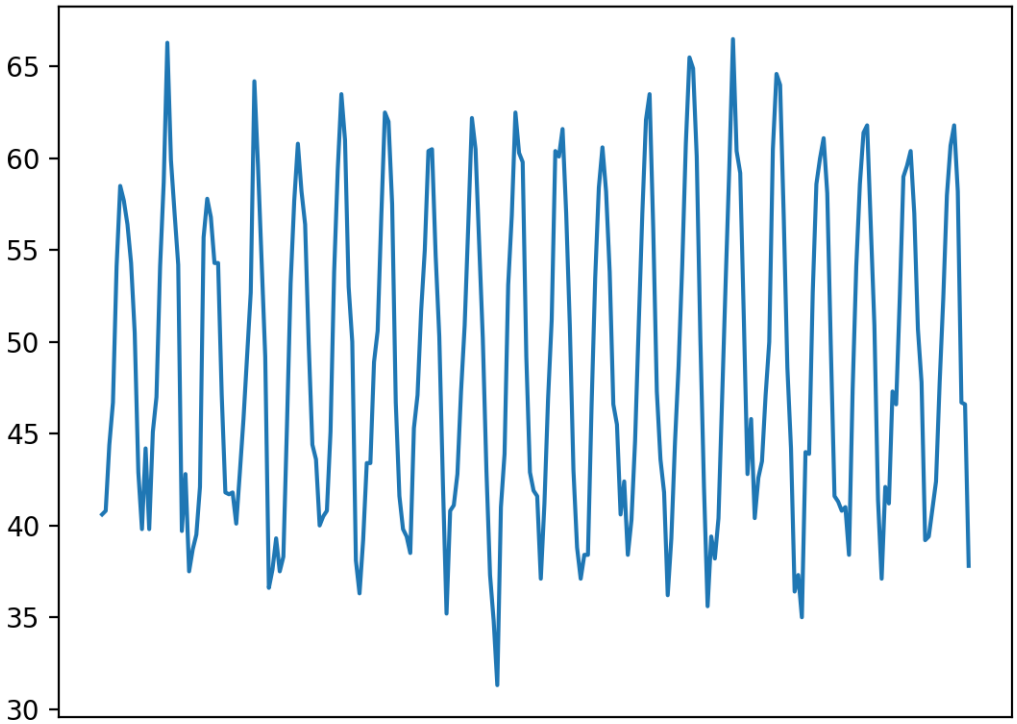

Figure 11.3: Line Plot of the Monthly Mean Temperatures Dataset.

We will trim the dataset to the last five years of data (60 observations) in order to speed up the model evaluation process and use the last year or 12 observations for the test set.

```
# trim dataset to 5 years
data = data[-(5*12):]
```

Listing 11.43: Example of reducing the size of the dataset.

The period of the seasonal component is about one year, or 12 observations. We will use this as the seasonal period in the call to the `simple_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = simple_configs(seasonal=[0, 12])
```

Listing 11.44: Example of specifying some seasonal configurations.

We can now grid search naive models for the dataset. The complete example grid searching the monthly mean temperature time series forecasting problem is listed below.

```
# grid search simple forecast for monthly mean temperature
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
```

```
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
```

```
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # define dataset
  series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
```

```
data = series.values
# data split
n_test = 12
# model configs
max_length = len(data) - n_test
cfg_list = simple_configs(max_length, offsets=[1,12])
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 11.45: Example of grid searching naive models for the monthly mean temperature dataset.

Running the example prints the model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run.

```
...
> Model[[227, 12, 'persist']] 5.365
> Model[[228, 1, 'persist']] 2.818
> Model[[228, 1, 'mean']] 8.258
> Model[[228, 1, 'median']] 8.361
> Model[[228, 12, 'persist']] 2.818
done

[4, 12, 'mean'] 1.5015616870445234
[8, 12, 'mean'] 1.5794579766489512
[13, 12, 'mean'] 1.586186052546763
```

Listing 11.46: Example output from grid searching naive models for the monthly mean temperature dataset.

We can see that the best result was an RMSE of about 1.50 degrees with the following configuration:

- **Strategy**: Average
- **n**: 4
- **offset**: 12
- **function**: mean()

This finding is not too surprising. Given the seasonal structure of the data, we would expect a function of the last few observations at prior points in the yearly cycle to be effective.

11.7 Case Study 4: Trend and Seasonality

The *monthly car sales* dataset summarizes the monthly car sales in Quebec, Canada between 1960 and 1968. You can download the dataset directly from here:

- monthly-car-sales.csv [6]

Save the file with the filename `monthly-car-sales.csv` in your current working directory. We can load this dataset as a Pandas `Series` using the function `read_csv()` and summarize the shape of the dataset.

```
# load
series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
```

Listing 11.47: Example of loading the monthly car sales dataset.

We can then create a line plot of the series and inspect it for systematic structures like trends and seasonality.

```
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.48: Example of plotting the monthly car sales dataset.

The complete example is listed below.

```
# load and plot monthly car sales dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.49: Example of loading and plotting the monthly car sales dataset.

Running the example first summarizes the shape of the loaded dataset. The dataset has 9 years, or 108 observations. We will use the last year or 12 observations as the test set.

```
(108, 1)
```

Listing 11.50: Example output from loading and summarizing the shape of the monthly shampoo sales dataset

A line plot of the series is also created. We can see that there is an obvious trend and seasonal components.

[6] https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-car-sales.csv

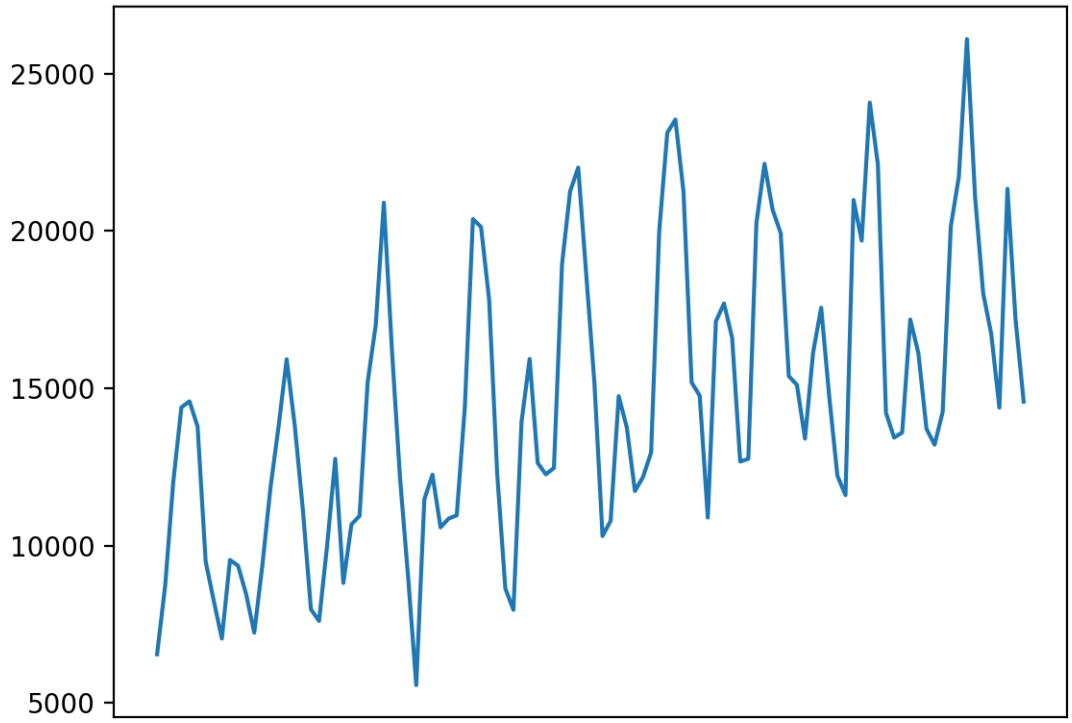

Figure 11.4: Line Plot of the Monthly Car Sales Dataset.

The period of the seasonal component could be six months or 12 months. We will try both as the seasonal period in the call to the `simple_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = simple_configs(seasonal=[0,6,12])
```

Listing 11.51: Example of specifying some seasonal configurations.

We can now grid search naive models for the dataset. The complete example grid searching the monthly car sales time series forecasting problem is listed below.

```
# grid search simple forecast for monthly car sales
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
from pandas import read_csv
```

```
# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
```

```
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # define dataset
  series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
  max_length = len(data) - n_test
  cfg_list = simple_configs(max_length, offsets=[1,12])
  # grid search
```

```
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 11.52: Example of grid searching naive models for the monthly car sales dataset.

Running the example prints the model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run.

```
...
> Model[[79, 1, 'median']] 5124.113
> Model[[91, 12, 'persist']] 9580.149
> Model[[79, 12, 'persist']] 8641.529
> Model[[92, 1, 'persist']] 9830.921
> Model[[92, 1, 'mean']] 5148.126
done

[3, 12, 'median'] 1841.1559321976688
[3, 12, 'mean'] 2115.198495632485
[4, 12, 'median'] 2184.37708988932
```

Listing 11.53: Example output from grid searching naive models for the monthly car sales dataset.

We can see that the best result was an RMSE of about 1841.15 sales with the following configuration:

- `Strategy`: Average
- `n`: 3
- `offset`: 12
- `function`: median()

It is not surprising that the chosen model is a function of the last few observations at the same point in prior cycles, although the use of the median instead of the mean may not have been immediately obvious and the results were much better than the mean.

11.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Plot Forecast**. Update the framework to re-fit a model with the best configuration and forecast the entire test dataset, then plot the forecast compared to the actual observations in the test set.
- **Drift Method**. Implement the drift method for simple forecasts and compare the results to the average and naive methods.

- **Another Dataset**. Apply the developed framework to an additional univariate time series problem (e.g. from the Time Series Dataset Library).

If you explore any of these extensions, I'd love to know.

11.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

11.9.1 Datasets

- Time Series Dataset Library, DataMarket.
  https://datamarket.com/data/list/?q=provider:tsdl
- Daily Female Births Dataset, DataMarket.
  https://datamarket.com/data/set/235k/daily-total-female-births-in-california-1959
- Monthly Shampoo Sales Dataset, DataMarket.
  https://datamarket.com/data/set/22r0/sales-of-shampoo-over-a-three-year-period
- Monthly Mean Temperature Dataset, DataMarket.
  https://datamarket.com/data/set/22li/mean-monthly-air-temperature-deg-f-nottingham-
- Monthly Car Sales Dataset, DataMarket.
  https://datamarket.com/data/set/22n4/monthly-car-sales-in-quebec-1960-1968

11.9.2 APIs

- `numpy.mean` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html
- `numpy.median` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.median.html
- `sklearn.metrics.mean_squared_error` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html
- `pandas.read_csv` API.
  https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html
- Joblib: running Python functions as pipeline jobs.
  https://pythonhosted.org/joblib/

11.9.3 Articles

- Forecasting, Wikipedia.
  https://en.wikipedia.org/wiki/Forecasting

11.10 Summary

In this tutorial, you discovered how to develop a framework from scratch for grid searching simple naive and averaging strategies for time series forecasting with univariate data. Specifically, you learned:

- How to develop a framework for grid searching simple models from scratch using walk-forward validation.
- How to grid search simple model hyperparameters for daily time series data for births.
- How to grid search simple model hyperparameters for monthly time series data for shampoo sales, car sales, and temperature.

11.10.1 Next

In the next lesson, you will discover how to develop exponential smoothing models for univariate time series forecasting problems.

Chapter 12

How to Develop ETS Models for Univariate Forecasting

Exponential smoothing is a time series forecasting method for univariate data that can be extended to support data with a systematic trend or seasonal component. It is common practice to use an optimization process to find the model hyperparameters that result in the exponential smoothing model with the best performance for a given time series dataset. This practice applies only to the coefficients used by the model to describe the exponential structure of the level, trend, and seasonality. It is also possible to automatically optimize other hyperparameters of an exponential smoothing model, such as whether or not to model the trend and seasonal component and if so, whether to model them using an additive or multiplicative method.

In this tutorial, you will discover how to develop a framework for grid searching all of the exponential smoothing model hyperparameters for univariate time series forecasting. After completing this tutorial, you will know:

- How to develop a framework for grid searching ETS models from scratch using walk-forward validation.
- How to grid search ETS model hyperparameters for daily time series data for female births.
- How to grid search ETS model hyperparameters for monthly time series data for shampoo sales, car sales, and temperature.

Let's get started.

12.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Develop a Grid Search Framework
2. Case Study 1: No Trend or Seasonality
3. Case Study 2: Trend
4. Case Study 3: Seasonality
5. Case Study 4: Trend and Seasonality

12.2 Develop a Grid Search Framework

In this section, we will develop a framework for grid searching exponential smoothing model hyperparameters for a given univariate time series forecasting problem. For more information on exponential smoothing for time series forecasting, also called ETS, see Chapter 5. We will use the implementation of Holt-Winters Exponential Smoothing provided by the Statsmodels library. This model has hyperparameters that control the nature of the exponential performed for the series, trend, and seasonality, specifically:

- `smoothing_level` (alpha): the smoothing coefficient for the level.
- `smoothing_slope` (beta): the smoothing coefficient for the trend.
- `smoothing_seasonal` (gamma): the smoothing coefficient for the seasonal component.
- `damping_slope` (phi): the coefficient for the damped trend.

All four of these hyperparameters can be specified when defining the model. If they are not specified, the library will automatically tune the model and find the optimal values for these hyperparameters (e.g. `optimized=True`). There are other hyperparameters that the model will not automatically tune that you may want to specify; they are:

- `trend`: The type of trend component, as either `add` for additive or `mul` for multiplicative. Modeling the trend can be disabled by setting it to `None`.
- `damped`: Whether or not the trend component should be damped, either True or False.
- `seasonal`: The type of seasonal component, as either `add` for additive or `mul` for multiplicative. Modeling the seasonal component can be disabled by setting it to `None`.
- `seasonal_periods`: The number of time steps in a seasonal period, e.g. 12 for 12 months in a yearly seasonal structure.
- `use_boxcox`: Whether or not to perform a power transform of the series (True/False) or specify the lambda for the transform.

If you know enough about your problem to specify one or more of these parameters, then you should specify them. If not, you can try grid searching these parameters. We can start-off by defining a function that will fit a model with a given configuration and make a one-step forecast. The `exp_smoothing_forecast()` below implements this behavior. The function takes an array or list of contiguous prior observations and a list of configuration parameters used to configure the model. The configuration parameters in order are: the trend type, the dampening type, the seasonality type, the seasonal period, whether or not to use a Box-Cox transform, and whether or not to remove the bias when fitting the model.

```
# one-step Holt Winter's Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
```

```
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]
```

Listing 12.1: Example of a function for making an ETS forecast.

In this tutorial, we will use the grid searching framework developed in Chapter 11 for tuning and evaluating naive forecasting methods. One important modification to the framework is the function used to perform the walk-forward validation of the model named `walk_forward_validation()`. This function must be updated to call the function for making an ETS forecast. The updated version of the function is listed below.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error
```

Listing 12.2: Example of a function for walk-forward validation with ETS forecasts.

We're nearly done. The only thing left to do is to define a list of model configurations to try for a dataset. We can define this generically. The only parameter we may want to specify is the periodicity of the seasonal component in the series, if one exists. By default, we will assume no seasonal component. The `exp_smoothing_configs()` function below will create a list of model configurations to evaluate. An optional list of seasonal periods can be specified, and you could even change the function to specify other elements that you may know about your time series. In theory, there are 72 possible model configurations to evaluate, but in practice, many will not be valid and will result in an error that we will trap and ignore.

```
# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
```

```
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models
```

Listing 12.3: Example of a function for defining configurations for ETS models to grid search.

We now have a framework for grid searching triple exponential smoothing model hyperparameters via one-step walk-forward validation. It is generic and will work for any in-memory univariate time series provided as a list or NumPy array. We can make sure all the pieces work together by testing it on a contrived 10-step dataset. The complete example is listed below.

```
# grid search holt winter's exponential smoothing
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
```

```
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
```

```
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # define dataset
  data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
  print(data)
  # data split
  n_test = 4
  # model configs
  cfg_list = exp_smoothing_configs()
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 12.4: Example of demonstrating the grid search infrastructure.

Running the example first prints the contrived time series dataset. Next, the model configurations and their errors are reported as they are evaluated. Finally, the configurations and the error for the top three configurations are reported.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]

 > Model[[None, False, None, None, True, True]] 1.380
 > Model[[None, False, None, None, True, False]] 10.000
 > Model[[None, False, None, None, False, True]] 2.563
 > Model[[None, False, None, None, False, False]] 10.000
done

[None, False, None, None, True, True] 1.379824445857423
[None, False, None, None, False, True] 2.5628662672606612
[None, False, None, None, False, False] 10.0
```

Listing 12.5: Example output from demonstrating the grid search infrastructure.

We do not report the model parameters optimized by the model itself. It is assumed that you can achieve the same result again by specifying the broader hyperparameters and allow the library to find the same internal parameters. You can access these internal parameters by refitting a standalone model with the same configuration and printing the contents of the `params` attribute on the model fit; for example:

```
# access model parameters
```

```
print(model_fit.params)
```

Listing 12.6: Example of accessing the automatically fit model parameters.

Now that we have a robust framework for grid searching ETS model hyperparameters, let's test it out on a suite of standard univariate time series datasets. The datasets were chosen for demonstration purposes; I am not suggesting that an ETS model is the best approach for each dataset, and perhaps an SARIMA or something else would be more appropriate in some cases.

12.3 Case Study 1: No Trend or Seasonality

The *daily female births* dataset summarizes the daily total female births in California, USA in 1959. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- daily-total-female-births.csv [1]

Save the file with the filename `daily-total-female-births.csv` in your current working directory. The dataset has one year, or 365 observations. We will use the first 200 for training and the remaining 165 as the test set. The complete example grid searching the daily female univariate time series forecasting problem is listed below.

```
# grid search ets models for daily female births
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))
```

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-total-female-births.csv

```
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
```

```
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 165
  # model configs
  cfg_list = exp_smoothing_configs()
  # grid search
  scores = grid_search(data[:,0], cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 12.7: Example of grid searching ETS models for the daily female births dataset.

Running the example may take a few minutes as fitting each ETS model can take about a minute on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > Model[['mul', False, None, None, True, False]] 6.985
 > Model[[None, False, None, None, True, True]] 7.169
```

```
 > Model[[None, False, None, None, True, False]] 7.212
 > Model[[None, False, None, None, False, True]] 7.117
 > Model[[None, False, None, None, False, False]] 7.126
done

['mul', False, None, None, True, True] 6.960703917145126
['mul', False, None, None, True, False] 6.984513598720297
['add', False, None, None, True, True] 7.081359856193836
```

Listing 12.8: Example output from grid searching ETS models for the daily female births dataset.

We can see that the best result was an RMSE of about 6.96 births. A naive model achieved an RMSE of 6.93 births, meaning that the best performing ETS model is not skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Trend**: Multiplicative
- **Damped**: False
- **Seasonal**: None
- **Seasonal Periods**: None
- **Box-Cox Transform**: True
- **Remove Bias**: True

What is surprising is that a model that assumed an multiplicative trend performed better than one that didn't. We would not know that this is the case unless we threw out assumptions and grid searched models.

12.4 Case Study 2: Trend

The *monthly shampoo sales* dataset summarizes the monthly sales of shampoo over a three-year period. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-shampoo-sales.csv [2]

Save the file with the filename `monthly-shampoo-sales.csv` in your current working directory. The dataset has three years, or 36 observations. We will use the first 24 for training and the remaining 12 as the test set. The complete example grid searching the shampoo sales univariate time series forecasting problem is listed below.

```
# grid search ets models for monthly shampoo sales
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
```

[2] https://raw.githubusercontent.com/jbrownlee/Datasets/master/shampoo.csv

```
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
```

```
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
```

```
cfg_list = exp_smoothing_configs()
# grid search
scores = grid_search(data[:,0], cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 12.9: Example of grid searching ETS models for the monthly shampoo sales dataset.

Running the example is fast given there are a small number of observations. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > Model[['mul', True, None, None, False, False]] 102.152
> Model[['mul', False, None, None, False, True]] 86.406
> Model[['mul', False, None, None, False, False]] 83.747
> Model[[None, False, None, None, False, True]] 99.416
> Model[[None, False, None, None, False, False]] 108.031
done

['mul', False, None, None, False, False] 83.74666940175238
['mul', False, None, None, False, True] 86.40648953786152
['mul', True, None, None, False, True] 95.33737598817238
```

Listing 12.10: Example output from grid searching ETS models for the monthly shampoo sales dataset.

We can see that the best result was an RMSE of about 83.74 sales. A naive model achieved an RMSE of 95.69 sales on this dataset, meaning that the best performing ETS model is skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Trend**: Multiplicative
- **Damped**: False
- **Seasonal**: None
- **Seasonal Periods**: None
- **Box-Cox Transform**: False
- **Remove Bias**: False

12.5 Case Study 3: Seasonality

The *monthly mean temperatures* dataset summarizes the monthly average air temperatures in Nottingham Castle, England from 1920 to 1939 in degrees Fahrenheit. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-mean-temp.csv [3]

Save the file with the filename `monthly-mean-temp.csv` in your current working directory. The dataset has 20 years, or 240 observations. We will trim the dataset to the last five years of data (60 observations) in order to speed up the model evaluation process and use the last year or 12 observations for the test set.

```
# trim dataset to 5 years
data = data[-(5*12):]
```

Listing 12.11: Example of reducing the size of the dataset.

The period of the seasonal component is about one year, or 12 observations. We will use this as the seasonal period in the call to the `exp_smoothing_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = exp_smoothing_configs(seasonal=[0, 12])
```

Listing 12.12: Example of specifying some seasonal configurations.

The complete example grid searching the monthly mean temperature time series forecasting problem is listed below.

```
# grid search ets hyperparameters for monthly mean temp dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
```

[3] https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-mean-temp.csv

```
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
```

```
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
  data = series.values
  # trim dataset to 5 years
  data = data[-(5*12):]
  # data split
  n_test = 12
  # model configs
  cfg_list = exp_smoothing_configs(seasonal=[0,12])
  # grid search
  scores = grid_search(data[:,0], cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 12.13: Example of grid searching ETS models for the monthly mean temperature dataset.

Running the example is relatively slow given the large amount of data. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[['mul', True, None, 12, False, False]] 4.593
 > Model[['mul', False, 'add', 12, True, True]] 4.230
 > Model[['mul', False, 'add', 12, True, False]] 4.157
 > Model[['mul', False, 'add', 12, False, True]] 1.538
 > Model[['mul', False, 'add', 12, False, False]] 1.520
done

[None, False, 'add', 12, False, False] 1.5015527325330889
[None, False, 'add', 12, False, True] 1.5015531225114707
[None, False, 'mul', 12, False, False] 1.501561363221282
```

Listing 12.14: Example output from grid searching ETS models for the monthly mean temperature dataset.

We can see that the best result was an RMSE of about 1.50 degrees. This is the same RMSE found by a naive model on this problem, suggesting that the best ETS model sits on the border of being unskillful. We can unpack the configuration of the best performing model as follows:

- **Trend**: None
- **Damped**: False
- **Seasonal**: Additive
- **Seasonal Periods**: 12
- **Box-Cox Transform**: False
- **Remove Bias**: False

12.6 Case Study 4: Trend and Seasonality

The *monthly car sales* dataset summarizes the monthly car sales in Quebec, Canada between 1960 and 1968. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-car-sales.csv [4]

Save the file with the filename `monthly-car-sales.csv` in your current working directory. The dataset has 9 years, or 108 observations. We will use the last year or 12 observations as the test set. The period of the seasonal component could be six months or 12 months. We will try both as the seasonal period in the call to the `exp_smoothing_configs()` function when preparing the model configurations.

[4] https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-car-sales.csv

```
# model configs
cfg_list = exp_smoothing_configs(seasonal=[0,6,12])
```

Listing 12.15: Example of specifying some seasonal configurations.

The complete example grid searching the monthly car sales time series forecasting problem is listed below.

```
# grid search ets models for monthly car sales
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
```

```
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
```

```
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
  cfg_list = exp_smoothing_configs(seasonal=[0,6,12])
  # grid search
  scores = grid_search(data[:,0], cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 12.16: Example of grid searching ETS models for the monthly car sales dataset.

Running the example is slow given the large amount of data. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[['mul', True, 'add', 6, False, False]] 3745.142
 > Model[['mul', True, 'add', 12, True, True]] 2203.354
 > Model[['mul', True, 'add', 12, True, False]] 2284.172
 > Model[['mul', True, 'add', 12, False, True]] 2842.605
 > Model[['mul', True, 'add', 12, False, False]] 2086.899
done

['add', False, 'add', 12, False, True] 1672.5539372356582
['add', False, 'add', 12, False, False] 1680.845043013083
['add', True, 'add', 12, False, False] 1696.1734099400082
```

Listing 12.17: Example output from grid searching ETS models for the monthly car sales dataset.

We can see that the best result was an RMSE of about 1,672 sales. A naive model achieved an RMSE of 1841.15 sales on this problem, suggesting that the best performing ETS model is skillful. We can unpack the configuration of the best performing model as follows:

- **Trend**: Additive
- **Damped**: False
- **Seasonal**: Additive

- **Seasonal Periods**: 12
- **Box-Cox Transform**: False
- **Remove Bias**: True

This is a little surprising as I would have guessed that a six-month seasonal model would be the preferred approach.

12.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Transforms**. Update the framework to support configurable data transforms such as normalization and standardization.
- **Plot Forecast**. Update the framework to re-fit a model with the best configuration and forecast the entire test dataset, then plot the forecast compared to the actual observations in the test set.
- **Tune Amount of History**. Update the framework to tune the amount of historical data used to fit the model (e.g. in the case of the 10 years of max temperature data).

If you explore any of these extensions, I'd love to know.

12.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

12.8.1 Books

- Chapter 7 Exponential smoothing, *Forecasting: principles and practice*, 2013. https://amzn.to/2xlJsfV
- Section 6.4. Introduction to Time Series Analysis, *Engineering Statistics Handbook*, 2012. https://www.itl.nist.gov/div898/handbook/
- *Practical Time Series Forecasting with R*, 2016. https://amzn.to/2LGKzKm

12.8.2 APIs

- `statsmodels.tsa.holtwinters.ExponentialSmoothing` API.
- `statsmodels.tsa.holtwinters.HoltWintersResults` API.

12.8.3 Articles

- Exponential smoothing, Wikipedia.
  https://en.wikipedia.org/wiki/Exponential_smoothing

12.9 Summary

In this tutorial, you discovered how to develop a framework for grid searching all of the exponential smoothing model hyperparameters for univariate time series forecasting. Specifically, you learned:

- How to develop a framework for grid searching ETS models from scratch using walk-forward validation.
- How to grid search ETS model hyperparameters for daily time series data for births.
- How to grid search ETS model hyperparameters for monthly time series data for shampoo sales, car sales and temperature.

12.9.1 Next

In the next lesson, you will discover how to develop autoregressive models for univariate time series forecasting problems.

Chapter 13

How to Develop SARIMA Models for Univariate Forecasting

The Seasonal Autoregressive Integrated Moving Average, or SARIMA, model is an approach for modeling univariate time series data that may contain trend and seasonal components. It is an effective approach for time series forecasting, although it requires careful analysis and domain expertise in order to configure the seven or more model hyperparameters. An alternative approach to configuring the model that makes use of fast and parallel modern hardware is to grid search a suite of hyperparameter configurations in order to discover what works best. Often, this process can reveal non-intuitive model configurations that result in lower forecast error than those configurations specified through careful analysis.

In this tutorial, you will discover how to develop a framework for grid searching all of the SARIMA model hyperparameters for univariate time series forecasting. After completing this tutorial, you will know:

- How to develop a framework for grid searching SARIMA models from scratch using walk-forward validation.
- How to grid search SARIMA model hyperparameters for daily time series data for births.
- How to grid search SARIMA model hyperparameters for monthly time series data for shampoo sales, car sales, and temperature.

Let's get started.

13.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Develop a Grid Search Framework
2. Case Study 1: No Trend or Seasonality
3. Case Study 2: Trend
4. Case Study 3: Seasonality
5. Case Study 4: Trend and Seasonality

13.2 Develop a Grid Search Framework

In this section, we will develop a framework for grid searching SARIMA model hyperparameters for a given univariate time series forecasting problem. For more information on SARIMA for time series forecasting, see Chapter 5. We will use the implementation of SARIMA provided by the Statsmodels library. This model has hyperparameters that control the nature of the model performed for the series, trend and seasonality, specifically:

- `order`: A tuple `p`, `d`, and `q` parameters for the modeling of the trend.
- `seasonal_order`: A tuple of `P`, `D`, `Q`, and `m` parameters for the modeling the seasonality
- `trend`: A parameter for controlling a model of the deterministic trend as one of `'n'`, `'c'`, `'t'`, and `'ct'` for no trend, constant, linear, and constant with linear trend, respectively.

If you know enough about your problem to specify one or more of these parameters, then you should specify them. If not, you can try grid searching these parameters. We can start-off by defining a function that will fit a model with a given configuration and make a one-step forecast. The `sarima_forecast()` below implements this behavior.

The function takes an array or list of contiguous prior observations and a list of configuration parameters used to configure the model, specifically two tuples and a string for the trend order, seasonal order trend, and parameter. We also try to make the model robust by relaxing constraints, such as that the data must be stationary and that the MA transform be invertible.

```
# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]
```

Listing 13.1: Example of a function for making a SARIMA forecast.

In this tutorial, we will use the grid searching framework developed in Chapter 11 for tuning and evaluating naive forecasting methods. One important modification to the framework is the function used to perform the walk-forward validation of the model named `walk_forward_validation()`. This function must be updated to call the function for making an SARIMA forecast. The updated version of the function is listed below.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
```

```
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error
```

Listing 13.2: Example of a function for walk-forward validation with SARIMA forecasts.

We're nearly done. The only thing left to do is to define a list of model configurations to try for a dataset. We can define this generically. The only parameter we may want to specify is the periodicity of the seasonal component in the series, if one exists. By default, we will assume no seasonal component. The `sarima_configs()` function below will create a list of model configurations to evaluate.

The configurations assume each of the AR, MA, and I components for trend and seasonality are low order, e.g. off (0) or in `[1,2]`. You may want to extend these ranges if you believe the order may be higher. An optional list of seasonal periods can be specified, and you could even change the function to specify other elements that you may know about your time series. In theory, there are 1,296 possible model configurations to evaluate, but in practice, many will not be valid and will result in an error that we will trap and ignore.

```
# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
                  cfg = [(p,d,q), (P,D,Q,m), t]
                  models.append(cfg)
  return models
```

Listing 13.3: Example of a function for defining configurations for SARIMA models to grid search.

We now have a framework for grid searching SARIMA model hyperparameters via one-step walk-forward validation. It is generic and will work for any in-memory univariate time series

provided as a list or NumPy array. We can make sure all the pieces work together by testing it on a contrived 10-step dataset. The complete example is listed below.

```
# grid search sarima hyperparameters
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
```

```
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
                  cfg = [(p,d,q), (P,D,Q,m), t]
```

```
                models.append(cfg)
  return models

if __name__ == '__main__':
  # define dataset
  data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
  print(data)
  # data split
  n_test = 4
  # model configs
  cfg_list = sarima_configs()
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 13.4: Example of demonstrating the grid search infrastructure.

Running the example first prints the contrived time series dataset. Next, the model configurations and their errors are reported as they are evaluated, truncated below for brevity. Finally, the configurations and the error for the top three configurations are reported. We can see that many models achieve perfect performance on this simple linearly increasing contrived time series problem.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]

...
 > Model[[(2, 0, 0), (2, 0, 0, 0), 'ct']] 0.001
 > Model[[(2, 0, 0), (2, 0, 1, 0), 'ct']] 0.000
 > Model[[(2, 0, 1), (0, 0, 0, 0), 'n']] 0.000
 > Model[[(2, 0, 1), (0, 0, 1, 0), 'n']] 0.000
done

[(2, 1, 0), (1, 0, 0, 0), 'n'] 0.0
[(2, 1, 0), (2, 0, 0, 0), 'n'] 0.0
[(2, 1, 1), (1, 0, 1, 0), 'n'] 0.0
```

Listing 13.5: Example output from demonstrating the grid search infrastructure.

Now that we have a robust framework for grid searching SARIMA model hyperparameters, let's test it out on a suite of standard univariate time series datasets. The datasets were chosen for demonstration purposes; I am not suggesting that a SARIMA model is the best approach for each dataset; perhaps an ETS or something else would be more appropriate in some cases.

13.3 Case Study 1: No Trend or Seasonality

The *daily female births* dataset summarizes the daily total female births in California, USA in 1959. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- daily-total-female-births.csv [1]

Save the file with the filename `daily-total-female-births.csv` in your current working directory. The dataset has one year, or 365 observations. We will use the first 200 for training and the remaining 165 as the test set. The complete example grid searching the daily female univariate time series forecasting problem is listed below.

```
# grid search sarima hyperparameters for daily female dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
```

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-total-female-births.csv

```
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
```

```
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
                  cfg = [(p,d,q), (P,D,Q,m), t]
                  models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 165
  # model configs
  cfg_list = sarima_configs()
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 13.6: Example of grid searching SARIMA models for the daily female births dataset.

Running the example may take a while on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[[(2, 1, 2), (1, 0, 1, 0), 'ct']] 6.905
> Model[[(2, 1, 2), (2, 0, 0, 0), 'ct']] 7.031
> Model[[(2, 1, 2), (2, 0, 1, 0), 'ct']] 6.985
> Model[[(2, 1, 2), (1, 0, 2, 0), 'ct']] 6.941
> Model[[(2, 1, 2), (2, 0, 2, 0), 'ct']] 7.056
done

[(1, 0, 2), (1, 0, 1, 0), 't'] 6.770349800255089
[(0, 1, 2), (1, 0, 2, 0), 'ct'] 6.773217122759515
[(2, 1, 1), (2, 0, 2, 0), 'ct'] 6.886633191752254
```

Listing 13.7: Example output from grid searching SARIMA models for the daily female births dataset.

We can see that the best result was an RMSE of about 6.77 births. A naive model achieved an RMSE of 6.93 births suggesting that the best performing SARIMA model is skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Order**: (1, 0, 2)

- **Seasonal Order**: (1, 0, 1, 0)
- **Trend Parameter**: 't' for linear trend

It is surprising that a configuration with some seasonal elements resulted in the lowest error. I would not have guessed at this configuration and would have likely stuck with an ARIMA model.

13.4 Case Study 2: Trend

The *monthly shampoo sales* dataset summarizes the monthly sales of shampoo over a three-year period. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-shampoo-sales.csv [2]

Save the file with the filename `monthly-shampoo-sales.csv` in your current working directory. The dataset has three years, or 36 observations. We will use the first 24 for training and the remaining 12 as the test set. The complete example grid searching the shampoo sales univariate time series forecasting problem is listed below.

```
# grid search sarima hyperparameters for monthly shampoo sales dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

[2] https://raw.githubusercontent.com/jbrownlee/Datasets/master/shampoo.csv

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
```

```
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
                  cfg = [(p,d,q), (P,D,Q,m), t]
                  models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
  cfg_list = sarima_configs()
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 13.8: Example of grid searching SARIMA models for the monthly shampoo sales dataset.

Running the example may take a while on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[[(2, 1, 2), (1, 0, 1, 0), 'ct']] 68.891
> Model[[(2, 1, 2), (2, 0, 0, 0), 'ct']] 75.406
```

```
> Model[[(2, 1, 2), (1, 0, 2, 0), 'ct']] 80.908
> Model[[(2, 1, 2), (2, 0, 1, 0), 'ct']] 78.734
> Model[[(2, 1, 2), (2, 0, 2, 0), 'ct']] 82.958
done

[(0, 1, 2), (2, 0, 2, 0), 't'] 54.767582003072874
[(0, 1, 1), (2, 0, 2, 0), 'ct'] 58.69987083057107
[(1, 1, 2), (0, 0, 1, 0), 't'] 58.709089340600094
```

Listing 13.9: Example output from grid searching naive models for the monthly shampoo sales dataset.

We can see that the best result was an RMSE of about 54.76 sales. A naive model achieved an RMSE of 95.69 sales on this dataset, meaning that the best performing SARIMA model is skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Trend Order**: (0, 1, 2)
- **Seasonal Order**: (2, 0, 2, 0)
- **Trend Parameter**: 't' (linear trend)

13.5 Case Study 3: Seasonality

The *monthly mean temperatures* dataset summarizes the monthly average air temperatures in Nottingham Castle, England from 1920 to 1939 in degrees Fahrenheit. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-mean-temp.csv [3]

Save the file with the filename `monthly-mean-temp.csv` in your current working directory. The dataset has 20 years, or 240 observations. We will trim the dataset to the last five years of data (60 observations) in order to speed up the model evaluation process and use the last year or 12 observations for the test set.

```
# trim dataset to 5 years
data = data[-(5*12):]
```

Listing 13.10: Example of reducing the size of the dataset.

The period of the seasonal component is about one year, or 12 observations. We will use this as the seasonal period in the call to the `sarima_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = sarima_configs(seasonal=[0, 12])
```

Listing 13.11: Example of specifying some seasonal configurations.

[3] https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-mean-temp.csv

The complete example grid searching the monthly mean temperature time series forecasting problem is listed below.

```
# grid search sarima hyperparameters for monthly mean temp dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
```

```
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
```

```
                cfg = [(p,d,q), (P,D,Q,m), t]
                models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
  data = series.values
  # trim dataset to 5 years
  data = data[-(5*12):]
  # data split
  n_test = 12
  # model configs
  cfg_list = sarima_configs(seasonal=[0, 12])
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 13.12: Example of grid searching SARIMA models for the monthly mean temperature dataset.

Running the example may take a while on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[[(2, 1, 2), (2, 1, 0, 12), 't']] 4.599
> Model[[(2, 1, 2), (1, 1, 0, 12), 'ct']] 2.477
> Model[[(2, 1, 2), (2, 0, 0, 12), 'ct']] 2.548
> Model[[(2, 1, 2), (2, 0, 1, 12), 'ct']] 2.893
> Model[[(2, 1, 2), (2, 1, 0, 12), 'ct']] 5.404
done

[(0, 0, 0), (1, 0, 1, 12), 'n'] 1.5577613610905712
[(0, 0, 0), (1, 1, 0, 12), 'n'] 1.6469530713847962
[(0, 0, 0), (2, 0, 0, 12), 'n'] 1.7314448163607488
```

Listing 13.13: Example output from grid searching SARIMA models for the monthly mean temperature dataset.

We can see that the best result was an RMSE of about 1.55 degrees. A naive model achieved an RMSE of 1.50 degrees, suggesting that the best performing SARIMA model is not skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Trend Order**: (0, 0, 0)
- **Seasonal Order**: (1, 0, 1, 12)

- **Trend Parameter**: 'n' (no trend)

As we would expect, the model has no trend component and a 12-month seasonal ARIMA component.

13.6 Case Study 4: Trend and Seasonality

The *monthly car sales* dataset summarizes the monthly car sales in Quebec, Canada between 1960 and 1968. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-car-sales.csv [4]

Save the file with the filename `monthly-car-sales.csv` in your current working directory. The dataset has 9 years, or 108 observations. We will use the last year or 12 observations as the test set. The period of the seasonal component could be six months or 12 months. We will try both as the seasonal period in the call to the `sarima_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = sarima_configs(seasonal=[0,6,12])
```

Listing 13.14: Example of specifying some seasonal configurations.

The complete example grid searching the monthly car sales time series forecasting problem is listed below.

```
# grid search sarima hyperparameters for monthly car sales dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
```

[4] https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-car-sales.csv

```
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
```

```
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
                  cfg = [(p,d,q), (P,D,Q,m), t]
                  models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
  cfg_list = sarima_configs(seasonal=[0,6,12])
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 13.15: Example of grid searching SARIMA models for the monthly car sales dataset.

Running the example may take a while on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider

running the example a few times.

```
...
> Model[[(2, 1, 2), (2, 0, 2, 12), 'ct']] 10710.462
> Model[[(2, 1, 2), (2, 1, 2, 6), 'ct']] 2183.568
> Model[[(2, 1, 2), (2, 1, 0, 12), 'ct']] 2105.800
> Model[[(2, 1, 2), (2, 1, 1, 12), 'ct']] 2330.361
> Model[[(2, 1, 2), (2, 1, 2, 12), 'ct']] 31580326686.803
done

[(0, 0, 0), (1, 1, 0, 12), 't'] 1551.8423920342414
[(0, 0, 0), (2, 1, 1, 12), 'c'] 1557.334614575545
[(0, 0, 0), (1, 1, 0, 12), 'c'] 1559.3276311282675
```

Listing 13.16: Example output from grid searching SARIMA models for the monthly car sales dataset.

We can see that the best result was an RMSE of about 1,551.84 sales. A naive model achieved an RMSE of 1,841.15 sales on this problem, suggesting that the best performing SARIMA model is skillful. We can unpack the configuration of the best performing model as follows:

- **Trend Order**: (0, 0, 0)
- **Seasonal Order**: (1, 1, 0, 12)
- **Trend Parameter**: `'t'` (linear trend)

13.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Transforms**. Update the framework to support configurable data transforms such as normalization and standardization.
- **Plot Forecast**. Update the framework to re-fit a model with the best configuration and forecast the entire test dataset, then plot the forecast compared to the actual observations in the test set.
- **Tune Amount of History**. Update the framework to tune the amount of historical data used to fit the model (e.g. in the case of the 10 years of max temperature data).

If you explore any of these extensions, I'd love to know.

13.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

13.8.1 Books

- Chapter 8 ARIMA models, *Forecasting: principles and practice*, 2013.
  https://amzn.to/2xlJsfV
- Chapter 7, Non-stationary Models, *Introductory Time Series with R*, 2009.
  https://amzn.to/2smB9LR

13.8.2 APIs

- Statsmodels Time Series Analysis by State Space Methods.
  http://www.statsmodels.org/dev/statespace.html
- `statsmodels.tsa.statespace.sarimax.SARIMAX` API.
  http://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html
- `statsmodels.tsa.statespace.sarimax.SARIMAXResults` API.
  http://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAXResults.html
- Statsmodels SARIMAX Notebook.
  http://www.statsmodels.org/dev/examples/notebooks/generated/statespace_sarimax_stata.html

13.8.3 Articles

- Autoregressive integrated moving average, Wikipedia.
  https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average

13.9 Summary

In this tutorial, you discovered how to develop a framework for grid searching all of the SARIMA model hyperparameters for univariate time series forecasting. Specifically, you learned:

- How to develop a framework for grid searching SARIMA models from scratch using walk-forward validation.
- How to grid search SARIMA model hyperparameters for daily time series data for births.
- How to grid search SARIMA model hyperparameters for monthly time series data for shampoo sales, car sales and temperature.

13.9.1 Next

In the next lesson, you will discover how to develop deep learning models for univariate time series forecasting problems.

Chapter 14

How to Develop MLPs, CNNs and LSTMs for Univariate Forecasting

Deep learning neural networks are capable of automatically learning and extracting features from raw data. This feature of neural networks can be used for time series forecasting problems, where models can be developed directly on the raw observations without the direct need to scale the data using normalization and standardization or to make the data stationary by differencing. Impressively, simple deep learning neural network models are capable of making skillful forecasts as compared to naive models and tuned SARIMA models on univariate time series forecasting problems that have both trend and seasonal components with no pre-processing.

In this tutorial, you will discover how to develop a suite of deep learning models for univariate time series forecasting. After completing this tutorial, you will know:

- How to develop a robust test harness using walk-forward validation for evaluating the performance of neural network models.
- How to develop and evaluate simple Multilayer Perceptron and convolutional neural networks for time series forecasting.
- How to develop and evaluate LSTMs, CNN-LSTMs, and ConvLSTM neural network models for time series forecasting.

Let's get started.

14.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Time Series Problem
2. Model Evaluation Test Harness
3. Multilayer Perceptron Model
4. Convolutional Neural Network Model
5. Recurrent Neural Network Models

14.2 Time Series Problem

In this tutorial we will focus on one dataset and use it as the context to demonstrate the development of a range of deep learning models for univariate time series forecasting. We will use the *monthly car sales* dataset as this context as it includes the complexity of both trend and seasonal elements. The *monthly car sales* dataset summarizes the monthly car sales in Quebec, Canada between 1960 and 1968. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-car-sales.csv [1]

Save the file with the filename `monthly-car-sales.csv` in your current working directory. The dataset is monthly and has nine years, or 108 observations. In our testing, will use the last year, or 12 observations, as the test set. A line plot is created. The dataset has an obvious trend and seasonal component. The period of the seasonal component could be six months or 12 months. From prior experiments, we know that a naive model can achieve a root mean squared error, or RMSE, of 1,841.155 by taking the median of the observations at the three prior years for the month being predicted (see Chapter 11); for example:

```
yhat = median(-12, -24, -36)
```

Listing 14.1: Example of an effective naive forecast model.

Where the negative indexes refer to observations in the series relative to the end of the historical data for the month being predicted. From prior experiments, we know that a SARIMA model can achieve an RMSE of 1,551.84 with the configuration of `SARIMA(0, 0, 0),(1, 1, 0),12` where no elements are specified for the trend and a seasonal difference with a period of 12 is calculated and an AR model of one season is used (see Chapter 13).

The performance of the naive model provides a lower bound on a model that is considered skillful. Any model that achieves a predictive performance of lower than 1,841.15 on the last 12 months has skill. The performance of the SARIMA model provides a measure of a good model on the problem. Any model that achieves a predictive performance lower than 1,551.84 on the last 12 months should be adopted over a SARIMA model. Now that we have defined our problem and expectations of model skill, we can look at defining the test harness.

14.3 Model Evaluation Test Harness

In this section, we will develop a test harness for developing and evaluating different types of neural network models for univariate time series forecasting. This test harness is a modified version of the framework presented in Chapter 11 and may cover much of the same of the same ground. This section is divided into the following parts:

1. Train-Test Split
2. Series as Supervised Learning
3. Walk-Forward Validation

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-car-sales.csv

4. Repeat Evaluation
5. Summarize Performance
6. Worked Example

14.3.1 Train-Test Split

The first step is to split the loaded series into train and test sets. We will use the first eight years (96 observations) for training and the last 12 for the test set. The `train_test_split()` function below will split the series taking the raw observations and the number of observations to use in the test set as arguments.

```
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

Listing 14.2: Example of a function for splitting data into train and test sets.

14.3.2 Series as Supervised Learning

Next, we need to be able to frame the univariate series of observations as a supervised learning problem so that we can train neural network models (covered in Chapter 4). A supervised learning framing of a series means that the data needs to be split into multiple examples that the model learn from and generalize across. Each sample must have both an input component and an output component. The input component will be some number of prior observations, such as three years or 36 time steps. The output component will be the total sales in the next month because we are interested in developing a model to make one-step forecasts.

We can implement this using the `shift()` function on the Pandas `DataFrame`. It allows us to shift a column down (forward in time) or back (backward in time). We can take the series as a column of data, then create multiple copies of the column, shifted forward or backward in time in order to create the samples with the input and output elements we require. When a series is shifted down, `NaN` values are introduced because we don't have values beyond the start of the series. For example, the series defined as a column:

```
(t)
1
2
3
4
```

Listing 14.3: Example of a time series as a column of data.

Can be shifted and inserted as a column beforehand:

```
(t-1),    (t)
Nan,       1
1,         2
2,         3
3,         4
4,         NaN
```

Listing 14.4: Example of column time series data with an added shifted column.

We can see that on the second row, the value 1 is provided as input as an observation at the prior time step, and 2 is the next value in the series that can be predicted, or learned by the model to be predicted when 1 is presented as input. Rows with `NaN` values can be removed. The `series_to_supervised()` function below implements this behavior, allowing you to specify the number of lag observations to use in the input and the number to use in the output for each sample. It will also remove rows that have `NaN` values as they cannot be used to train or test a model.

```
# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values
```

Listing 14.5: Example of a function for transforming a univariate series into a supervised learning dataset.

Note, this is a more generic way of transforming a time series dataset into samples than the specialized methods presented in Chapters 7, 8, and 9.

14.3.3 Walk-Forward Validation

Time series forecasting models can be evaluated on a test set using walk-forward validation. Walk-forward validation is an approach where the model makes a forecast for each observation in the test dataset one at a time. After each forecast is made for a time step in the test dataset, the true observation for the forecast is added to the test dataset and made available to the model. Simpler models can be refit with the observation prior to making the subsequent prediction. More complex models, such as neural networks, are not refit given the much greater computational cost. Nevertheless, the true observation for the time step can then be used as part of the input for making the prediction on the next time step. First, the dataset is split into train and test sets. We will call the `train_test_split()` function to perform this split and pass in the pre-specified number of observations to use as the test data.

A model will be fit once on the training dataset for a given configuration. We will define a generic `model_fit()` function to perform this operation that can be filled in for the given type of neural network that we may be interested in later. The function takes the training dataset and the model configuration and returns the fit model ready for making predictions.

```
# fit a model
def model_fit(train, config):
  return None
```

Listing 14.6: Example of a function for fitting a model with a configuration.

Each time step of the test dataset is enumerated. A prediction is made using the fit model. Again, we will define a generic function named `model_predict()` that takes the fit model, the history, and the model configuration and makes a single one-step prediction.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  return 0.0
```

Listing 14.7: Example of a function for making a forecast with a fit model.

The prediction is added to a list of predictions and the true observation from the test set is added to a list of observations that was seeded with all observations from the training dataset. This list is built up during each step in the walk-forward validation, allowing the model to make a one-step prediction using the most recent history. All of the predictions can then be compared to the true values in the test set and an error measure calculated. We will calculate the root mean squared error, or RMSE, between predictions and the true values.

RMSE is calculated as the square root of the average of the squared differences between the forecasts and the actual values. The `measure_rmse()` implements this below using the `mean_squared_error()` scikit-learn function to first calculate the mean squared error, or MSE, before calculating the square root.

```
# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))
```

Listing 14.8: Example of a function for calculating RMSE for a forecast.

The complete `walk_forward_validation()` function that ties all of this together is listed below. It takes the dataset, the number of observations to use as the test set, and the configuration for the model, and returns the RMSE for the model performance on the test set.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error
```

Listing 14.9: Example of a function for walk-forward validation of a forecast model configuration.

14.3.4 Repeat Evaluation

Neural network models are stochastic. This means that, given the same model configuration and the same training dataset, a different internal set of weights will result each time the model is trained that will in turn have a different performance. This is a benefit, allowing the model to be adaptive and find high performing configurations to complex problems. It is also a problem when evaluating the performance of a model and in choosing a final model to use to make predictions.

To address model evaluation, we will evaluate a model configuration multiple times via walk-forward validation and report the error as the average error across each evaluation. This is not always possible for large neural networks and may only make sense for small networks that can be fit in minutes or hours. The `repeat_evaluate()` function below implements this and allows the number of repeats to be specified as an optional parameter that defaults to 30 and returns a list of model performance scores: in this case, RMSE values.

```
# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores
```

Listing 14.10: Example of a function for the repeated evaluation of a forecast model configuration.

14.3.5 Summarize Performance

Finally, we need to summarize the performance of a model from the multiple repeats. We will summarize the performance first using summary statistics, specifically the mean and the standard deviation. We will also plot the distribution of model performance scores using a box and whisker plot to help get an idea of the spread of performance. The `summarize_scores()` function below implements this, taking the name of the model that was evaluated and the list of scores from each repeated evaluation, printing the summary and showing a plot.

```
# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()
```

Listing 14.11: Example of a function for summarizing the performance of a forecast model.

14.3.6 Worked Example

Now that we have defined the elements of the test harness, we can tie them all together and define a simple persistence model. Specifically, we will calculate the median of a subset of prior observations relative to the time to be forecasted. We do not need to fit a model so the `model_fit()` function will be implemented to simply return `None`.

```
# fit a model
def model_fit(train, config):
  return None
```

Listing 14.12: Example of a function for fitting a forecast model.

We will use the config to define a list of index offsets in the prior observations relative to the time to be forecasted that will be used as the prediction. For example, 12 will use the observation 12 months ago (-12) relative to the time to be forecasted.

```
# define config
config = [12, 24, 36]
```

Listing 14.13: Example of a configuration to evaluate.

The `model_predict()` function can be implemented to use this configuration to collect the observations, then return the median of those observations.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  values = list()
  for offset in config:
    values.append(history[-offset])
  return median(values)
```

Listing 14.14: Example of a function for making a naive forecast.

The complete example of using the framework with a simple persistence model is listed below.

```
# persistence forecast for monthly car sales dataset
from math import sqrt
from numpy import median
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, interval):
  return [data[i] - data[i - interval] for i in range(interval, len(data))]

# fit a model
def model_fit(train, config):
  return None

# forecast with a pre-fit model
def model_predict(model, history, config):
```

```
  values = list()
  for offset in config:
    values.append(history[-offset])
  return median(values)

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [12, 24, 36]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('persistence', scores)
```

Listing 14.15: Example of demonstrating the forecast infrastructure with a naive model.

Running the example prints the RMSE of the model evaluated using walk-forward validation

on the final 12 months of data. The model is evaluated 30 times, although, because the model has no stochastic element, the score is the same each time.

```
...
 > 1841.156
 > 1841.156
 > 1841.156
 > 1841.156
 > 1841.156
persistence: 1841.156 RMSE (+/- 0.000)
```

Listing 14.16: Example output from demonstrating the forecast infrastructure with a naive model.

We can see that the RMSE of the model is 1841 sales, providing a lower-bound of performance by which we can evaluate whether a model is skillful or not on the problem. A box and whisker plot is also created, but is not reproduced here because there is no distribution to summarize, e.g. the plot is not interesting in this case as all skill scores have the same value. Now that we have a robust test harness, we can use it to evaluate a suite of neural network models.

14.4 Multilayer Perceptron Model

The first network that we will evaluate is a Multilayer Perceptron, or MLP for short. This is a simple feedforward neural network model that should be evaluated before more elaborate models are considered. MLPs can be used for time series forecasting by taking multiple observations at prior time steps, called lag observations, and using them as input features and predicting one or more time steps from those observations. This is exactly the framing of the problem provided by the `series_to_supervised()` function in the previous section. The training dataset is therefore a list of samples, where each sample has some number of observations from months prior to the time being forecasted, and the forecast is the next month in the sequence. For example:

```
X,                      y
month1, month2, month3,  month4
month2, month3, month4,  month5
month3, month4, month5,  month6
...
```

Listing 14.17: Example framing of the forecast problem for training a model.

The model will attempt to generalize over these samples, such that when a new sample is provided beyond what is known by the model, it can predict something useful; for example:

```
X,                      y
month4, month5, month6,  ???
```

Listing 14.18: Example framing of the forecast problem for making a forecast.

We will implement a simple MLP using the Keras deep learning library. For more details on modeling a univariate time series with an MLP, see Chapter 7. The model will have an input layer with some number of prior observations. This can be specified using the `input_dim` argument when we define the first hidden layer. The model will have a single hidden layer with some number of nodes, then a single output layer. We will use the rectified linear activation function on the hidden layer as it performs well. We will use a linear activation function (the

default) on the output layer because we are predicting a continuous value. The loss function for the network will be the mean squared error loss, or MSE, and we will use the efficient Adam flavor of stochastic gradient descent to train the network.

```
# define model
model = Sequential()
model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
```

Listing 14.19: Example of defining an MLP forecast model.

The model will be fit for some number of training epochs (exposures to the training data) and batch size can be specified to define how often the weights are updated within each epoch. The `model_fit()` function for fitting an MLP model on the training dataset is listed below. The function expects the config to be a list with the following configuration hyperparameters:

- `n_input`: The number of lag observations to use as input to the model.
- `n_nodes`: The number of nodes to use in the hidden layer.
- `n_epochs`: The number of times to expose the model to the whole training dataset.
- `n_batch`: The number of samples within an epoch after which the weights are updated.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch = config
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  # define model
  model = Sequential()
  model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.20: Example of a function for defining and fitting a MLP forecast model.

Making a prediction with a fit MLP model is as straightforward as calling the `predict()` function and passing in one sample worth of input values required to make the prediction.

```
# make a prediction
yhat = model.predict(x_input, verbose=0)
```

Listing 14.21: Example of making a prediction with a fit MLP model.

In order to make a prediction beyond the limit of known data, this requires that the last n known observations are taken as an array and used as input. The `predict()` function expects one or more samples of inputs when making a prediction, so providing a single sample requires the array to have the shape `[1, n_input]`, where `n_input` is the number of time steps that the

model expects as input. Similarly, the `predict()` function returns an array of predictions, one for each sample provided as input. In the case of one prediction, there will be an array with one value. The `model_predict()` function below implements this behavior, taking the model, the prior observations, and model configuration as arguments, formulating an input sample and making a one-step prediction that is then returned.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _ = config
  # prepare data
  x_input = array(history[-n_input:]).reshape(1, n_input)
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

Listing 14.22: Example of a function for making a forecast with a fit MLP model.

We now have everything we need to evaluate an MLP model on the monthly car sales dataset. Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- `n_input`: 24 (e.g. 24 months)
- `n_nodes`: 500
- `n_epochs`: 100
- `n_batch`: 100

This configuration can be defined as a list:

```
# define config
config = [24, 500, 100, 100]
```

Listing 14.23: Example of good configuration for an MLP forecast model.

Note that when the training data is framed as a supervised learning problem, there are only 72 samples that can be used to train the model. Using a batch size of 72 or more means that the model is being trained using batch gradient descent instead of mini-batch gradient descent. This is often used for small datasets and means that weight updates and gradient calculations are performed at the end of each epoch, instead of multiple times within each epoch. The complete code example is listed below.

```
# evaluate mlp for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
```

```
from keras.layers import Dense
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch = config
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  # define model
  model = Sequential()
  model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _ = config
  # prepare data
  x_input = array(history[-n_input:]).reshape(1, n_input)
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
```

```
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [24, 500, 100, 100]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('mlp', scores)
```

Listing 14.24: Example of an MLP model for forecasting monthly car sales.

Running the example prints the RMSE for each of the 30 repeated evaluations of the model. At the end of the run, the average and standard deviation RMSE are reported of about 1,526 sales. We can see that, on average, the chosen configuration has better performance than both the naive model (1,841.155) and the SARIMA model (1,551.842). This is impressive given that the model operated on the raw data directly without scaling or the data being made stationary.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > 1458.993
 > 1643.383
 > 1457.925
 > 1558.934
 > 1708.278

mlp: 1526.688 RMSE (+/- 134.789)
```

Listing 14.25: Example output from an MLP model for forecasting monthly car sales.

A box and whisker plot of the RMSE scores is created to summarize the spread of the performance for the model. This helps to understand the spread of the scores. We can see that although on average the performance of the model is impressive, the spread is large. The standard deviation is a little more than 134 sales, meaning a worse case model run that is 2 or 3 standard deviations in error from the mean error may be worse than the naive model. A challenge in using the MLP model is in harnessing the higher skill and minimizing the variance of the model across multiple runs.

This problem applies generally for neural networks. There are many strategies that you could use, but perhaps the simplest is simply to train multiple final models on all of the available data and use them in an ensemble when making predictions, e.g. the prediction is the average of 10-to-30 models.

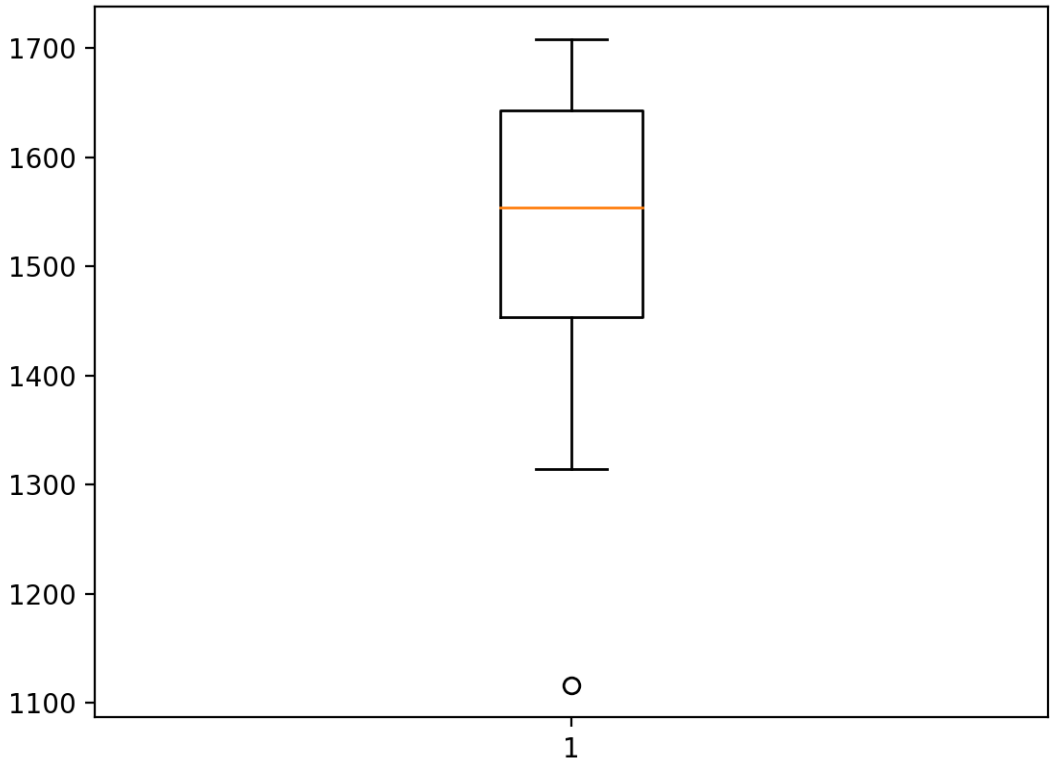

Figure 14.1: Box and Whisker Plot of Multilayer Perceptron RMSE Forecasting Car Sales.

14.5 Convolutional Neural Network Model

Convolutional Neural Networks, or CNNs, are a type of neural network developed for two-dimensional image data, although they can be used for one-dimensional data such as sequences of text and time series. When operating on one-dimensional data, the CNN reads across a sequence of lag observations and learns to extract features that are relevant for making a prediction. For more information on using CNNs for univariate time series forecasting, see Chapter 8. We will define a CNN with two convolutional layers for extracting features from the input sequences. Each will have a configurable number of filters and kernel size and will use the rectified linear activation function. The number of filters determines the number of parallel fields on which the weighted inputs are read and projected. The kernel size defines the number of time steps read within each snapshot as the network reads along the input sequence.

```
# define convolutional layers
model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
    input_shape=(n_input, 1)))
model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu'))
```

Listing 14.26: Example of defining convolutional layers.

A max pooling layer is used after convolutional layers to distill the weighted input features

into those that are most salient, reducing the input size by 1/4. The pooled inputs are flattened to one long vector before being interpreted and used to make a one-step prediction.

```
# define pooling and output layers
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(1))
```

Listing 14.27: Example of defining pooling, flatten and output layers.

The CNN model expects input data to be in the form of multiple samples, where each sample has multiple input time steps, the same as the MLP in the previous section. One difference is that the CNN can support multiple features or types of observations at each time step, which are interpreted as channels of an image. We only have a single feature at each time step, therefore the required three-dimensional shape of the input data will be `[n_samples, n_input, 1]`.

```
# reshape training data
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
```

Listing 14.28: Example of reshaping input data for the CNN model.

The `model_fit()` function for fitting the CNN model on the training dataset is listed below. The model takes the following five configuration parameters as a list:

- `n_input`: The number of lag observations to use as input to the model.
- `n_filters`: The number of parallel filters.
- `n_kernel`: The number of time steps considered in each read of the input sequence.
- `n_epochs`: The number of times to expose the model to the whole training dataset.
- `n_batch`: The number of samples within an epoch after which the weights are updated.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_filters, n_kernel, n_epochs, n_batch = config
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
      input_shape=(n_input, 1)))
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.29: Example of a function for defining and fitting a CNN forecast model.

Making a prediction with the fit CNN model is very much like making a prediction with the fit MLP model in the previous section. The one difference is in the requirement that we specify the number of features observed at each time step, which in this case is 1. Therefore, when making a single one-step prediction, the shape of the input array must be: `[1, n_input, 1]`. The `model_predict()` function below implements this behavior.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, _ = config
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

Listing 14.30: Example of a function for making a forecast with a fit CNN model.

Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- `n_input`: 36 (e.g. 3 years or 3×12)
- `n_filters`: 256
- `n_kernel`: 3
- `n_epochs`: 100
- `n_batch`: 100 (e.g. batch gradient descent)

This can be specified as a list as follows:

```
# define config
config = [36, 256, 3, 100, 100]
```

Listing 14.31: Example of good configuration for a CNN forecast model.

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

```
# evaluate cnn for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from matplotlib import pyplot
```

```
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_filters, n_kernel, n_epochs, n_batch = config
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
      input_shape=(n_input, 1)))
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, _ = config
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [36, 256, 3, 100, 100]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('cnn', scores)
```

Listing 14.32: Example of a CNN model for forecasting monthly car sales.

Running the example first prints the RMSE for each repeated evaluation of the model. At the end of the run, we can see that indeed the model is skillful, achieving an average RMSE of 1,524.06, which is better than the naive model, the SARIMA model, and even the MLP model in the previous section. This is impressive given that the model operated on the raw data directly without scaling or the data being made stationary.

The standard deviation of the score is large, at about 57 sales, but is $\frac{1}{3}$ the size of the standard deviation observed with the MLP model in the previous section. We have some confidence that in a bad-case scenario (3 standard deviations), the model RMSE will remain below (better than) the performance of the naive model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> 1489.795
> 1652.620
> 1537.349
> 1443.777
> 1567.179

cnn: 1524.067 RMSE (+/- 57.148)
```

Listing 14.33: Example output from a CNN model for forecasting monthly car sales.

A box and whisker plot of the scores is created to help understand the spread of error across the runs. We can see that the spread does seem to be biased towards larger error values, as we would expect, although the upper whisker of the plot (in this case, the largest error that are not outliers) is still limited at an RMSE of 1,650 sales.

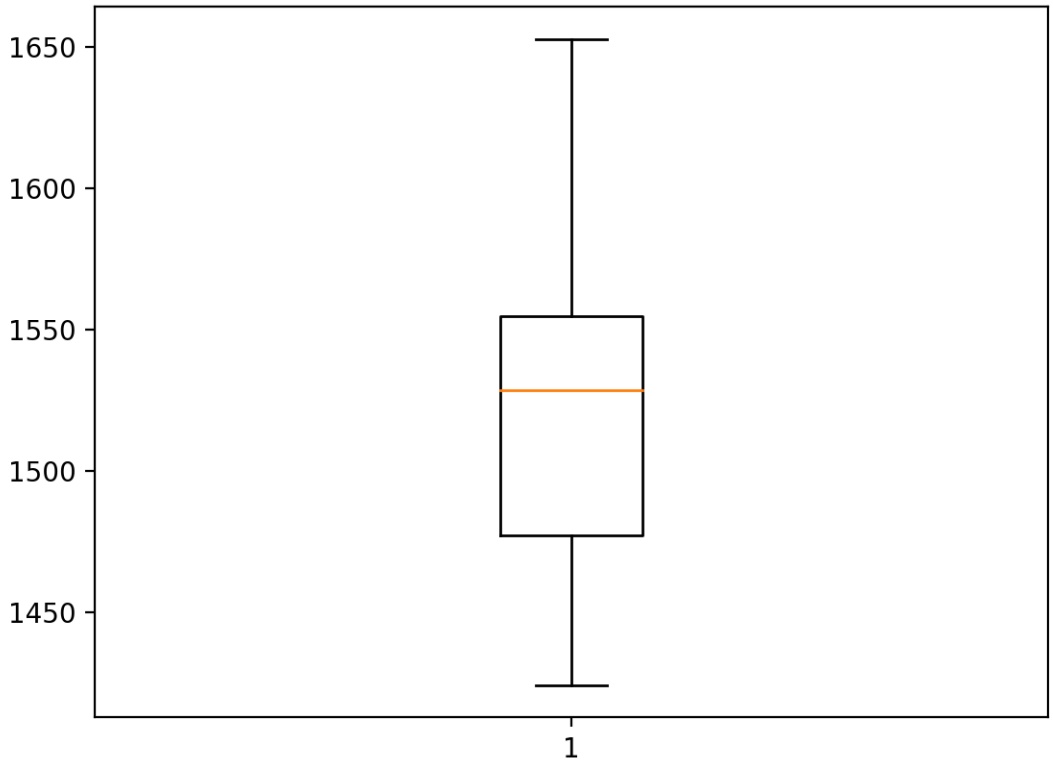

Figure 14.2: Box and Whisker Plot of Convolutional Neural Network RMSE Forecasting Car Sales.

14.6 Recurrent Neural Network Models

Recurrent neural networks, or RNNs, are those types of neural networks that use an output of the network from a prior step as an input in attempt to automatically learn across sequence data. The Long Short-Term Memory, or LSTM, network is a type of RNN whose implementation addresses the general difficulties in training RNNs on sequence data that results in a stable model. It achieves this by learning the weights for internal gates that control the recurrent connections within each node. Although developed for sequence data, LSTMs have not proven effective on time series forecasting problems where the output is a function of recent observations, e.g. an autoregressive type forecasting problem, such as the car sales dataset. Nevertheless, we can develop LSTM models for autoregressive problems and use them as a point of comparison with other neural network models. For more information on LSTMs for univariate time series forecasting, see Chapter 9. In this section, we will explore three variations on the LSTM model for univariate time series forecasting; they are:

- **Vanilla LSTM**: The LSTM network as-is.
- **CNN-LSTM**: A CNN network that learns input features and an LSTM that interprets them.

- **ConvLSTM**: A combination of CNNs and LSTMs where the LSTM units read input data using the convolutional process of a CNN.

14.6.1 LSTM

The LSTM neural network can be used for univariate time series forecasting. As an RNN, it will read each time step of an input sequence one step at a time. The LSTM has an internal memory allowing it to accumulate internal state as it reads across the steps of a given input sequence. At the end of the sequence, each node in a layer of hidden LSTM units will output a single value. This vector of values summarizes what the LSTM learned or extracted from the input sequence. This can be interpreted by a fully connected layer before a final prediction is made.

```
# define model
model = Sequential()
model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, 1)))
model.add(Dense(n_nodes, activation='relu'))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
```

Listing 14.34: Example of defining an LSTM forecast model.

Like the CNN, the LSTM can support multiple variables or features at each time step. As the car sales dataset only has one value at each time step, we can fix this at 1, both when defining the input to the network in the `input_shape` argument `[n_input, 1]`, and in defining the shape of the input samples.

```
# reshape input samples
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
```

Listing 14.35: Example of reshaping input data for the LSTM model.

Unlike the MLP and CNN that do not read the sequence data one-step at a time, the LSTM does perform better if the data is stationary. This means that difference operations are performed to remove the trend and seasonal structure. In the case of the car sales dataset, we can make the data stationery by performing a seasonal adjustment, that is subtracting the value from one year ago from each observation.

```
# seasonal differencing
adjusted = value - value[-12]
```

Listing 14.36: Example of seasonal differencing.

This can be performed systematically for the entire training dataset. It also means that the first year of observations must be discarded as we have no prior year of data to difference them with. The `difference()` function below will difference a provided dataset with a provided offset, called the difference order, e.g. 12 for one year of months prior.

```
# difference dataset
def difference(data, interval):
  return [data[i] - data[i - interval] for i in range(interval, len(data))]
```

Listing 14.37: Example of a function for differencing a series.

We can make the difference order a hyperparameter to the model and only perform the operation if a value other than zero is provided. The `model_fit()` function for fitting an LSTM model is provided below. The model expects a list of five model hyperparameters; they are:

- `n_input`: The number of lag observations to use as input to the model.
- `n_nodes`: The number of LSTM units to use in the hidden layer.
- `n_epochs`: The number of times to expose the model to the whole training dataset.
- `n_batch`: The number of samples within an epoch after which the weights are updated.
- `n_diff`: The difference order or 0 if not used.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # define model
  model = Sequential()
  model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, 1)))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.38: Example of a function for fitting an LSTM forecast model.

Making a prediction with the LSTM model is the same as making a prediction with a CNN model. A single input must have the three-dimensional structure of samples, time steps, and features, which in this case we only have 1 sample and 1 feature: `[1, n_input, 1]`. If the difference operation was performed, we must add back the value that was subtracted after the model has made a forecast. We must also difference the historical data prior to formulating the single input used to make a prediction. The `model_predict()` function below implements this behavior.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
```

```
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]
```

Listing 14.39: Example of a function for making a forecast with a fit LSTM model.

Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- `n_input`: 36 (i.e. 3 years or 3×12)
- `n_nodes`: 50
- `n_epochs`: 100
- `n_batch`: 100 (i.e. batch gradient descent)
- `n_diff`: 12 (i.e. seasonal difference)

This can be specified as a list:

```
# define config
config = [36, 50, 100, 100, 12]
```

Listing 14.40: Example of good configuration for an LSTM forecast model.

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

```
# evaluate lstm for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
```

```
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, interval):
  return [data[i] - data[i - interval] for i in range(interval, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # define model
  model = Sequential()
  model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, 1)))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
```

```
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [36, 50, 100, 100, 12]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('lstm', scores)
```

Listing 14.41: Example of an LSTM model for forecasting monthly car sales.

Running the example, we can see the RMSE for each repeated evaluation of the model. At the end of the run, we can see that the average RMSE is about 2,109, which is worse than the naive model. This suggests that the chosen model is not skillful, and it was the best that could be found given the same resources used to find model configurations in the previous sections. This provides further evidence (although weak evidence) that LSTMs, at least alone, are perhaps a bad fit for autoregressive-type sequence prediction problems.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
```

```
> 2266.130
> 2105.043
> 2128.549
> 1952.002
> 2188.287

lstm: 2109.779 RMSE (+/- 81.373)
```

Listing 14.42: Example output from an LSTM model for forecasting monthly car sales.

A box and whisker plot is also created summarizing the distribution of RMSE scores. Even the base case for the model did not achieve the performance of a naive model.

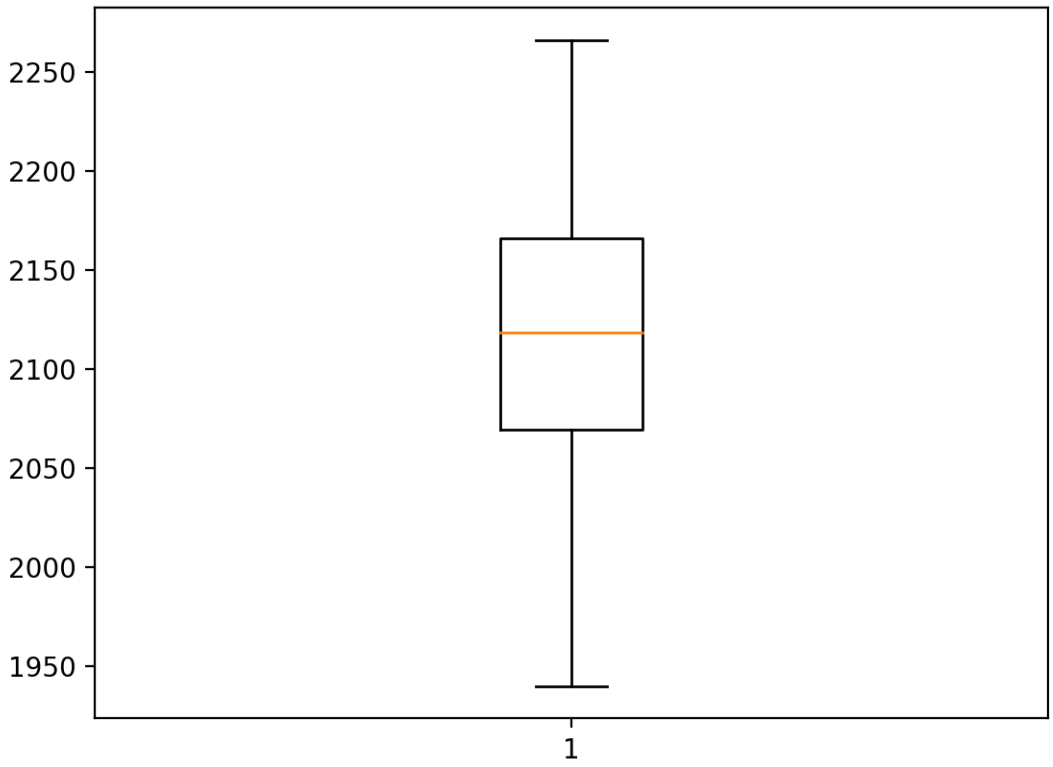

Figure 14.3: Box and Whisker Plot of Long Short-Term Memory Neural Network RMSE Forecasting Car Sales.

14.6.2 CNN LSTM

We have seen that the CNN model is capable of automatically learning and extracting features from the raw sequence data without scaling or differencing. We can combine this capability with the LSTM where a CNN model is applied to sub-sequences of input data, the results of which together form a time series of extracted features that can be interpreted by an LSTM model. This combination of a CNN model used to read multiple subsequences over time by an LSTM is called a CNN-LSTM model. The model requires that each input sequence, e.g. 36

months, is divided into multiple subsequences, each read by the CNN model, e.g. 3 subsequence of 12 time steps. It may make sense to divide the sub-sequences by years, but this is just a hypothesis, and other splits could be used, such as six subsequences of six time steps. Therefore, this splitting is parameterized with the `n_seq` and `n_steps` for the number of subsequences and number of steps per subsequence parameters.

```
# reshape input samples
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
```

Listing 14.43: Example of reshaping input data for the CNN-LSTM model.

The number of lag observations per sample is simply (`n_seq` × `n_steps`). This is a 4-dimensional input array now with the dimensions: `[samples, subsequences, timesteps, features]`. The same CNN model must be applied to each input subsequence. We can achieve this by wrapping the entire CNN model in a `TimeDistributed` layer wrapper.

```
# define CNN input model
model = Sequential()
model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
    activation='relu', input_shape=(None,n_steps,1))))
model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
    activation='relu')))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
```

Listing 14.44: Example of defining the CNN input model.

The output of one application of the CNN submodel will be a vector. The output of the submodel to each input subsequence will be a time series of interpretations that can be interpreted by an LSTM model. This can be followed by a fully connected layer to interpret the outcomes of the LSTM and finally an output layer for making one-step predictions.

```
# define LSTM and output model
model.add(LSTM(n_nodes, activation='relu'))
model.add(Dense(n_nodes, activation='relu'))
model.add(Dense(1))
```

Listing 14.45: Example of defining the LSTM output model.

The complete `model_fit()` function is listed below. The model expects a list of seven hyperparameters; they are:

- `n_seq`: The number of subsequences within a sample.
- `n_steps`: The number of time steps within each subsequence.
- `n_filters`: The number of parallel filters.
- `n_kernel`: The number of time steps considered in each read of the input sequence.
- `n_nodes`: The number of LSTM units to use in the hidden layer.
- `n_epochs`: The number of times to expose the model to the whole training dataset.
- `n_batch`: The number of samples within an epoch after which the weights are updated.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = config
  n_input = n_seq * n_steps
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, n_steps, 1))
  # define model
  model = Sequential()
  model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
      activation='relu', input_shape=(None,n_steps,1))))
  model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
      activation='relu')))
  model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
  model.add(TimeDistributed(Flatten()))
  model.add(LSTM(n_nodes, activation='relu'))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.46: Example of a function for defining and fitting a CNN-LSTM model.

Making a prediction with the fit model is much the same as the LSTM or CNN, although with the addition of splitting each sample into subsequences with a given number of time steps.

```
# prepare data
x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
```

Listing 14.47: Example of reshaping data for making a prediction.

The updated `model_predict()` function is listed below.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_seq, n_steps, _, _, _, _, _ = config
  n_input = n_seq * n_steps
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

Listing 14.48: Example of defining a function for making a forecast with a fit CNN-LSTM model.

Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- `n_seq`: 3 (i.e. 3 years)

- n_steps: 12 (i.e. 1 year of months)
- n_filters: 64
- n_kernel: 3
- n_nodes: 100
- n_epochs: 200
- n_batch: 100 (i.e. batch gradient descent)

We can define the configuration as a list; for example:

```
# define config
config = [3, 12, 64, 3, 100, 200, 100]
```

Listing 14.49: Example of defining a good configuration for the CNN-LSTM model.

The complete example of evaluating the CNN-LSTM model for forecasting the univariate monthly car sales is listed below.

```
# evaluate cnn-lstm for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import TimeDistributed
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
```

```
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# fit a model
def model_fit(train, config):
  # unpack config
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = config
  n_input = n_seq * n_steps
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, n_steps, 1))
  # define model
  model = Sequential()
  model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
      activation='relu', input_shape=(None,n_steps,1))))
  model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
      activation='relu')))
  model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
  model.add(TimeDistributed(Flatten()))
  model.add(LSTM(n_nodes, activation='relu'))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_seq, n_steps, _, _, _, _, _ = config
  n_input = n_seq * n_steps
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
```

```
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [3, 12, 64, 3, 100, 200, 100]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('cnn-lstm', scores)
```

Listing 14.50: Example of a CNN-LSTM model for forecasting monthly car sales.

Running the example prints the RMSE for each repeated evaluation of the model. The final averaged RMSE is reported at the end of about 1,626, which is lower than the naive model, but still higher than a SARIMA model. The standard deviation of this score is also very large, suggesting that the chosen configuration may not be as stable as the standalone CNN model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > 1289.794
 > 1685.976
 > 1498.123
 > 1618.627
 > 1448.361

cnn-lstm: 1626.735 RMSE (+/- 279.850)
```

Listing 14.51: Example output from a CNN-LSTM model for forecasting monthly car sales.

A box and whisker plot is also created summarizing the distribution of RMSE scores. The plot shows one single outlier of very poor performance just below 3,000 sales.

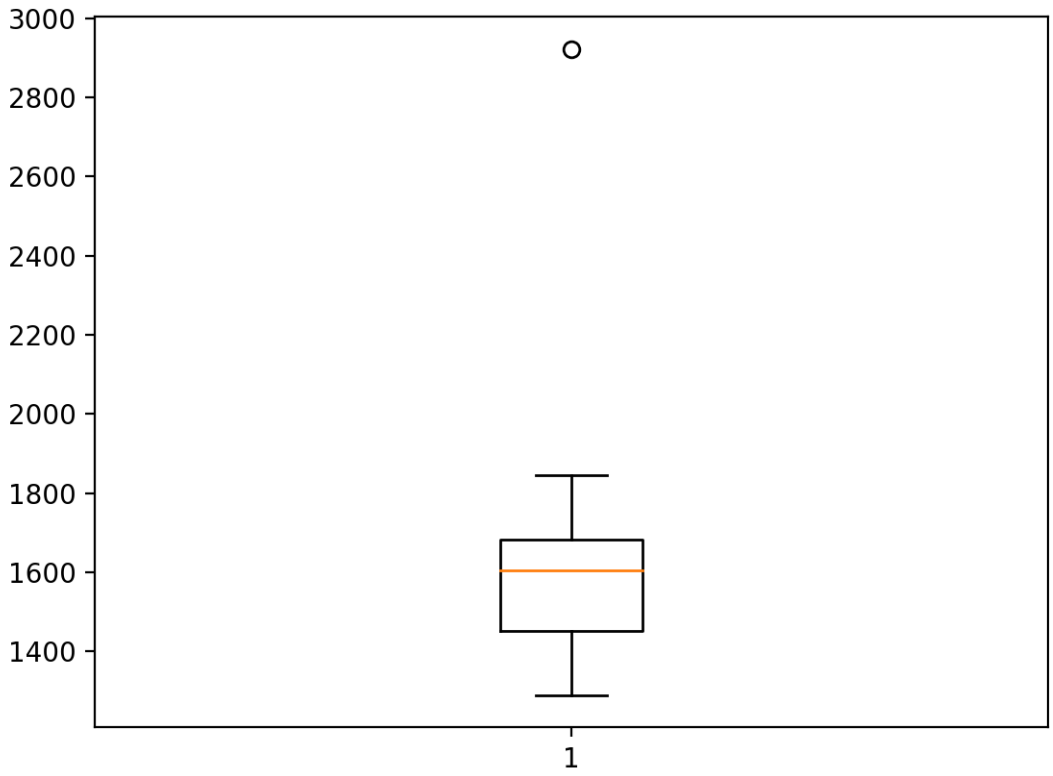

Figure 14.4: Box and Whisker Plot of CNN-LSTM RMSE Forecasting Car Sales.

14.6.3 ConvLSTM

It is possible to perform a convolutional operation as part of the read of the input sequence within each LSTM unit. This means, rather than reading a sequence one step at a time, the LSTM would read a block or subsequence of observations at a time using a convolutional process, like a CNN. This is different to first reading an extracting features with an LSTM and interpreting the result with an LSTM; this is performing the CNN operation at each time step as part of the LSTM.

This type of model is called a Convolutional LSTM, or ConvLSTM for short. It is provided in Keras as a layer called `ConvLSTM2D` for 2D data. We can configure it for use with 1D sequence data by assuming that we have one row with multiple columns. As with the CNN-LSTM, the input data is split into subsequences where each subsequence has a fixed number of time steps, although we must also specify the number of rows in each subsequence, which in this case is fixed at 1.

```
# reshape input samples
train_x = train_x.reshape((train_x.shape[0], n_seq, 1, n_steps, 1))
```

Listing 14.52: Example of reshaping input data for the ConvLSTM model.

The shape is five-dimensional, with the dimensions: `[samples, subsequences, rows, columns, features]`.

Like the CNN, the ConvLSTM layer allows us to specify the number of filter maps and the size of the kernel used when reading the input sequences.

```
# define convlstm layer
model.add(ConvLSTM2D(filters=n_filters, kernel_size=(1,n_kernel), activation='relu',
    input_shape=(n_seq, 1, n_steps, 1)))
```

Listing 14.53: Example of defining a ConvLSTM layer for univariate data.

The output of the layer is a sequence of filter maps that must first be flattened before it can be interpreted and followed by an output layer. The model expects a list of seven hyperparameters, the same as the CNN-LSTM; they are:

- `n_seq`: The number of subsequences within a sample.
- `n_steps`: The number of time steps within each subsequence.
- `n_filters`: The number of parallel filters.
- `n_kernel`: The number of time steps considered in each read of the input sequence.
- `n_nodes`: The number of LSTM units to use in the hidden layer.
- `n_epochs`: The number of times to expose the model to the whole training dataset.
- `n_batch`: The number of samples within an epoch after which the weights are updated.

The `model_fit()` function that implements all of this is listed below.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = config
  n_input = n_seq * n_steps
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, 1, n_steps, 1))
  # define model
  model = Sequential()
  model.add(ConvLSTM2D(filters=n_filters, kernel_size=(1,n_kernel), activation='relu',
      input_shape=(n_seq, 1, n_steps, 1)))
  model.add(Flatten())
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.54: Example of a function for defining and fitting a ConvLSTM forecast model.

A prediction is made with the fit model in the same way as the CNN-LSTM, although with the additional rows dimension that we fix to 1.

```
# prepare data
x_input = array(history[-n_input:]).reshape((1, n_seq, 1, n_steps, 1))
```

Listing 14.55: Example of reshaping input data for making a forecast with a ConvLSTM model.

The `model_predict()` function for making a single one-step prediction is listed below.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_seq, n_steps, _, _, _, _, _ = config
  n_input = n_seq * n_steps
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_seq, 1, n_steps, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

Listing 14.56: Example of a function for making a forecast with a fit ConvLSTM model.

Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- `n_seq`: 3 (i.e. 3 years)
- `n_steps`: 12 (i.e. 1 year of months)
- `n_filters`: 256
- `n_kernel`: 3
- `n_nodes`: 200
- `n_epochs`: 200
- `n_batch`: 100 (i.e. batch gradient descent)

We can define the configuration as a list; for example:

```
# define config
config = [3, 12, 256, 3, 200, 200, 100]
```

Listing 14.57: Example of defining a good configuration for the ConvLSTM model.

We can tie all of this together. The complete code listing for the ConvLSTM model evaluated for one-step forecasting of the monthly car sales dataset is listed below.

```
# evaluate convlstm for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
```

```
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import ConvLSTM2D
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, interval):
  return [data[i] - data[i - interval] for i in range(interval, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = config
  n_input = n_seq * n_steps
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, 1, n_steps, 1))
  # define model
  model = Sequential()
  model.add(ConvLSTM2D(filters=n_filters, kernel_size=(1,n_kernel), activation='relu',
      input_shape=(n_seq, 1, n_steps, 1)))
  model.add(Flatten())
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
```

```
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_seq, n_steps, _, _, _, _, _ = config
  n_input = n_seq * n_steps
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_seq, 1, n_steps, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [3, 12, 256, 3, 200, 200, 100]
```

```
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('convlstm', scores)
```

Listing 14.58: Example of a ConvLSTM model for forecasting monthly car sales.

Running the example prints the RMSE for each repeated evaluation of the model. The final averaged RMSE is reported at the end of about 1,660, which is lower than the naive model, but still higher than a SARIMA model. It is a result that is perhaps on par with the CNN-LSTM model. The standard deviation of this score is also very large, suggesting that the chosen configuration may not be as stable as the standalone CNN model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > 1653.084
 > 1650.430
 > 1291.353
 > 1558.616
 > 1653.231

convlstm: 1660.840 RMSE (+/- 248.826)
```

Listing 14.59: Example output from a ConvLSTM model for forecasting monthly car sales.

A box and whisker plot is also created, summarizing the distribution of RMSE scores.

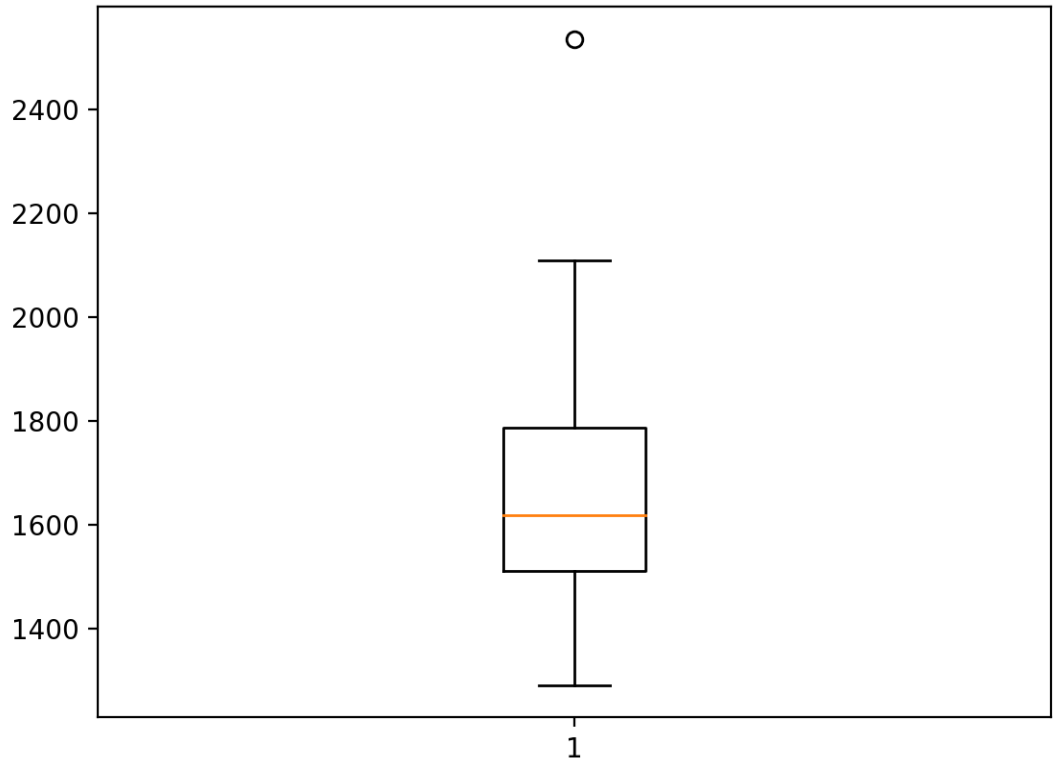

Figure 14.5: Box and Whisker Plot of ConvLSTM RMSE Forecasting Car Sales.

14.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Preparation**. Explore whether data preparation, such as normalization, standardization, and/or differencing can list the performance of any of the models.
- **Grid Search Hyperparameters**. Implement a grid search of the hyperparameters for one model to see if you can further lift performance.
- **Learning Curve Diagnostics**. Create a single fit of one model and review the learning curves on train and validation splits of the dataset, then use the diagnostics of the learning curves to further tune the model hyperparameters in order to improve model performance.
- **History Size**. Explore different amounts of historical data (lag inputs) for one model to see if you can further improve model performance
- **Reduce Variance of Final Model**. Explore one or more strategies to reduce the variance for one of the neural network models.

- **Update During Walk-Forward**. Explore whether re-fitting or updating a neural network model as part of walk-forward validation can further improve model performance.
- **More Parameterization**. Explore adding further model parameterization for one model, such as the use of additional layers.

If you explore any of these extensions, I'd love to know.

14.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- `pandas.DataFrame.shift` API.
  http://pandas-docs.github.io/pandas-docs-travis/generated/pandas.DataFrame.shift.html
- `matplotlib.pyplot.boxplot` API.
  https://matplotlib.org/api/_as_gen/matplotlib.pyplot.boxplot.html
- Keras Sequential Model API.
  https://keras.io/models/sequential/
- Keras Core Layers API.
  https://keras.io/layers/core/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/
- Keras Pooling Layers API.
  https://keras.io/layers/pooling/
- Keras Recurrent Layers API.
  https://keras.io/layers/recurrent/

14.9 Summary

In this tutorial, you discovered how to develop a suite of deep learning models for univariate time series forecasting. Specifically, you learned:

- How to develop a robust test harness using walk-forward validation for evaluating the performance of neural network models.
- How to develop and evaluate simple Multilayer Perceptron and convolutional neural networks for time series forecasting.
- How to develop and evaluate LSTMs, CNN-LSTMs, and ConvLSTM neural network models for time series forecasting.

14.9.1 Next

In the next lesson, you will discover how to develop a framework to grid search deep learning models for univariate time series forecasting problems.

Chapter 15

How to Grid Search Deep Learning Models for Univariate Forecasting

Grid searching is generally not an operation that we can perform with deep learning methods. This is because deep learning methods often require large amounts of data and large models, together resulting in models that take hours, days, or weeks to train. In those cases where the datasets are smaller, such as univariate time series, it may be possible to use a grid search to tune the hyperparameters of a deep learning model. In this tutorial, you will discover how to develop a framework to grid search hyperparameters for deep learning models. After completing this tutorial, you will know:

- How to develop a generic grid searching framework for tuning model hyperparameters.
- How to grid search hyperparameters for a Multilayer Perceptron model on the airline passengers univariate time series forecasting problem.
- How to adapt the framework to grid search hyperparameters for convolutional and long short-term memory neural networks.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Time Series Problem
2. Grid Search Framework
3. Multilayer Perceptron Model
4. Convolutional Neural Network Model
5. Long Short-Term Memory Network Model

15.2 Time Series Problem

In this tutorial we will focus on one dataset and use it as the context to demonstrate the development of a grid searching framework for range of deep learning models for univariate time series forecasting. We will use the *monthly airline passenger* dataset as this context as it includes the complexity of both trend and seasonal elements. The *monthly airline passenger* dataset summarizes the monthly total number of international passengers in thousands on for an airline from 1949 to 1960. Download the dataset directly from here:

- monthly-airline-passengers.csv [1]

Save the file with the filename `monthly-airline-passengers.csv` in your current working directory. We can load this dataset as a Pandas `DataFrame` using the function `read_csv()`.

```
# load
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
```

Listing 15.1: Load the dataset.

Once loaded, we can summarize the shape of the dataset in order to determine the number of observations.

```
# summarize shape
print(series.shape)
```

Listing 15.2: Summarize the shape of the dataset.

We can then create a line plot of the series to get an idea of the structure of the series.

```
# plot
pyplot.plot(series)
pyplot.show()
```

Listing 15.3: Create a line plot of the dataset.

We can tie all of this together; the complete example is listed below.

```
# load and plot monthly airline passengers dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 15.4: Example of loading and plotting the monthly airline passengers dataset.

Running the example first prints the shape of the dataset.

```
(144, 1)
```

Listing 15.5: Example output from loading and plotting the monthly airline passengers dataset.

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv

The dataset is monthly and has 12 years, or 144 observations. In our testing, we will use the last year, or 12 observations, as the test set. A line plot is created. The dataset has an obvious trend and seasonal component. The period of the seasonal component is 12 months.

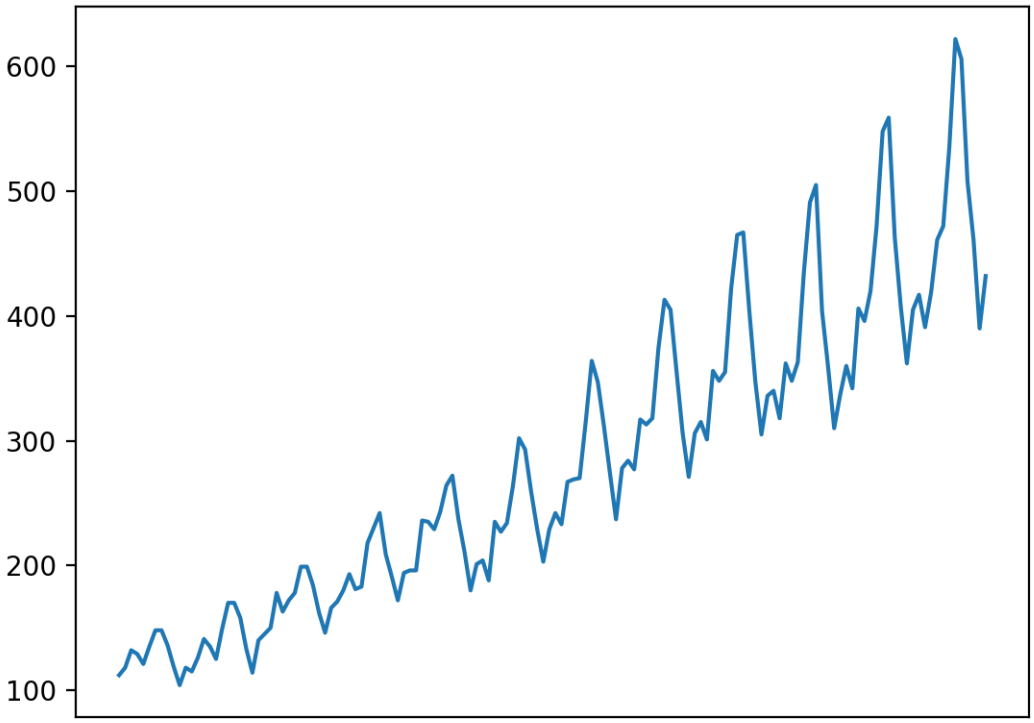

Figure 15.1: Line Plot of Monthly International Airline Passengers.

In this tutorial, we will introduce the tools for grid searching, but we will not optimize the model hyperparameters for this problem. Instead, we will demonstrate how to grid search the deep learning model hyperparameters generally and find models with some skill compared to a naive model. From prior experiments, a naive model can achieve a root mean squared error, or RMSE, of 50.70 (remember the units are thousands of passengers) by persisting the value from 12 months ago (relative index -12). The performance of this naive model provides a bound on a model that is considered skillful for this problem. Any model that achieves a predictive performance of lower than 50.70 on the last 12 months has skill.

It should be noted that a tuned ETS model can achieve an RMSE of 17.09 and a tuned SARIMA can achieve an RMSE of 13.89. These provide a lower bound on the expectations of a well-tuned deep learning model for this problem. Now that we have defined our problem and expectations of model skill, we can look at defining the grid search test harness.

15.3 Develop a Grid Search Framework

In this section, we will develop a grid search test harness that can be used to evaluate a range of hyperparameters for different neural network models, such as MLPs, CNNs, and LSTMs. This section is divided into the following parts:

1. Train-Test Split
2. Series as Supervised Learning
3. Walk-Forward Validation
4. Repeat Evaluation
5. Summarize Performance
6. Worked Example

Note, much of this framework was presented already in Chapter 14. Some elements are duplicated here given the changes needed to adapt it for grid searching model hyperparameters.

15.3.1 Train-Test Split

The first step is to split the loaded series into train and test sets. We will use the first 11 years (132 observations) for training and the last 12 for the test set. The `train_test_split()` function below will split the series taking the raw observations and the number of observations to use in the test set as arguments.

```
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

Listing 15.6: Example of a function to split the dataset into train and test sets.

15.3.2 Series as Supervised Learning

Next, we need to be able to frame the univariate series of observations as a supervised learning problem so that we can train neural network models. A supervised learning framing of a series means that the data needs to be split into multiple examples that the model learns from and generalizes across. Each sample must have both an input component and an output component. The input component will be some number of prior observations, such as three years, or 36 time steps.

The output component will be the total sales in the next month because we are interested in developing a model to make one-step forecasts. We can implement this using the `shift()` function on the Pandas `DataFrame`. It allows us to shift a column down (forward in time) or back (backward in time). We can take the series as a column of data, then create multiple copies of the column, shifted forward or backward in time in order to create the samples with the input and output elements we require. When a series is shifted down, `NaN` values are introduced because we don't have values beyond the start of the series.

```
(t)
1
2
3
4
```

Listing 15.7: Example of a time series as a column.

This column can be shifted and inserted as a column beforehand:

```
(t-1),    (t)
Nan,       1
1,         2
2,         3
3,         4
4,         NaN
```

Listing 15.8: Example of an added columns with the shifted time series.

We can see that on the second row, the value 1 is provided as input as an observation at the prior time step, and 2 is the next value in the series that can be predicted, or learned by the model to be predicted when 1 is presented as input. Rows with `NaN` values can be removed. The `series_to_supervised()` function below implements this behavior, allowing you to specify the number of lag observations to use in the input and the number to use in the output for each sample. It will also remove rows that have `NaN` values as they cannot be used to train or test a model.

```
# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values
```

Listing 15.9: Example of a function to transform a time series into samples.

Note, this is a more generic way of transforming a time series dataset into samples than the specialized methods presented in Chapters 7, 8, and 9.

15.3.3 Walk-Forward Validation

Time series forecasting models can be evaluated on a test set using walk-forward validation. Walk-forward validation is an approach where the model makes a forecast for each observation in the test dataset one at a time. After each forecast is made for a time step in the test dataset, the true observation for the forecast is added to the test dataset and made available to

the model. Simpler models can be refit with the observation prior to making the subsequent prediction. More complex models, such as neural networks, are not refit given the much greater computational cost. Nevertheless, the true observation for the time step can then be used as part of the input for making the prediction on the next time step.

First, the dataset is split into train and test sets. We will call the `train_test_split()` function to perform this split and pass in the pre-specified number of observations to use as the test data. A model will be fit once on the training dataset for a given configuration. We will define a generic `model_fit()` function to perform this operation that can be filled in for the given type of neural network that we may be interested in later. The function takes the training dataset and the model configuration and returns the fit model ready for making predictions.

```
# fit a model
def model_fit(train, config):
  return None
```

Listing 15.10: Example of a dummy function for fitting a model.

Each time step of the test dataset is enumerated. A prediction is made using the fit model. Again, we will define a generic function named `model_predict()` that takes the fit model, the history, and the model configuration and makes a single one-step prediction.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  return 0.0
```

Listing 15.11: Example of a dummy function for making a prediction with a fit model.

The prediction is added to a list of predictions and the true observation from the test set is added to a list of observations that was seeded with all observations from the training dataset. This list is built up during each step in the walk-forward validation, allowing the model to make a one-step prediction using the most recent history. All of the predictions can then be compared to the true values in the test set and an error measure calculated. We will calculate the root mean squared error, or RMSE, between predictions and the true values.

RMSE is calculated as the square root of the average of the squared differences between the forecasts and the actual values. The `measure_rmse()` implements this below using the `mean_squared_error()` scikit-learn function to first calculate the mean squared error, or MSE, before calculating the square root.

```
# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))
```

Listing 15.12: Example of a function for calculating the error for a forecast.

The complete `walk_forward_validation()` function that ties all of this together is listed below. It takes the dataset, the number of observations to use as the test set, and the configuration for the model, and returns the RMSE for the model performance on the test set.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
```

```
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error
```

Listing 15.13: Example of a function for the walk-forward evaluation of a deep learning model.

15.3.4 Repeat Evaluation

Neural network models are stochastic. This means that, given the same model configuration and the same training dataset, a different internal set of weights will result each time the model is trained that will, in turn, have a different performance. This is a benefit, allowing the model to be adaptive and find high performing configurations to complex problems. It is also a problem when evaluating the performance of a model and in choosing a final model to use to make predictions.

To address model evaluation, we will evaluate a model configuration multiple times via walk-forward validation and report the error as the average error across each evaluation. This is not always possible for large neural networks and may only make sense for small networks that can be fit in minutes or hours. The `repeat_evaluate()` function below implements this and allows the number of repeats to be specified as an optional parameter that defaults to 10 and returns the mean RMSE score from all repeats.

```
# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)
```

Listing 15.14: Example of a function for the repeated evaluation of a model.

15.3.5 Grid Search

We now have all the pieces of the framework. All that is left is a function to drive the search. We can define a `grid_search()` function that takes the dataset, a list of configurations to search, and the number of observations to use as the test set and perform the search. Once mean scores

are calculated for each config, the list of configurations is sorted in ascending order so that the best scores are listed first. The complete function is listed below.

```
# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores
```

Listing 15.15: Example of a function for coordinating the grid search of model hyperparameters.

15.3.6 Worked Example

Now that we have defined the elements of the test harness, we can tie them all together and define a simple persistence model. We do not need to fit a model so the `model_fit()` function will be implemented to simply return `None`.

```
# fit a model
def model_fit(train, config):
  return None
```

Listing 15.16: Example of a dummy function for fitting a model.

We will use the config to define a list of index offsets in the prior observations relative to the time to be forecasted that will be used as the prediction. For example, 12 will use the observation 12 months ago (-12) relative to the time to be forecasted.

```
# define config
cfg_list = [1, 6, 12, 24, 36]
```

Listing 15.17: Example of a set of configurations to search.

The `model_predict()` function can be implemented to use this configuration to persist the value at the negative relative offset.

```
# forecast with a pre-fit model
def model_predict(model, history, offset):
  history[-offset]
```

Listing 15.18: Example of a function for making persistence forecasts.

The complete example of using the framework with a simple persistence model is listed below.

```
# grid search persistence models for monthly airline passengers dataset
from math import sqrt
from numpy import mean
from pandas import read_csv
from sklearn.metrics import mean_squared_error

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

```
# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# fit a model
def model_fit(train, config):
  return None

# forecast with a pre-fit model
def model_predict(model, history, offset):
  return history[-offset]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# define dataset
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
data = series.values
```

```
# data split
n_test = 12
# model configs
cfg_list = [1, 6, 12, 24, 36]
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 10 configs
for cfg, error in scores[:10]:
  print(cfg, error)
```

Listing 15.19: Example of demonstrating the grid search framework for evaluating a persistence model on the airline passengers dataset.

Running the example prints the RMSE of the model evaluated using walk-forward validation on the final 12 months of data. Each model configuration is evaluated 10 times, although, because the model has no stochastic element, the score is the same each time. At the end of the run, the configurations and RMSE for the top three performing model configurations are reported. We can see, as we might have expected, that persisting the value from one year ago (relative offset -12) resulted in the best performance for the persistence model.

```
...
 > 110.274
 > 110.274
 > 110.274
> Model[36] 110.274
done

12 50.708316214732804
1 53.1515129919491
24 97.10990337413241
36 110.27352356753639
6 126.73495965991387
```

Listing 15.20: Example output from demonstrating the grid search framework for evaluating a persistence model on the airline passengers dataset.

Now that we have a robust test harness for grid searching model hyperparameters, we can use it to evaluate a suite of neural network models.

15.4 Multilayer Perceptron Model

In this section we will grid search hyperparameters for an MLPs for univariate time series forecasting. For more details on modeling a univariate time series with an MLP, see Chapter 7. There are many aspects of the MLP that we may wish to tune. We will define a very simple model with one hidden layer and define five hyperparameters to tune. They are:

- `n_input`: The number of prior inputs to use as input for the model (e.g. 12 months).
- `n_nodes`: The number of nodes to use in the hidden layer (e.g. 50).
- `n_epochs`: The number of training epochs (e.g. 1000).

- `n_batch`: The number of samples to include in each mini-batch (e.g. 32).
- `n_diff`: The difference order (e.g. 0 or 12).

Modern neural networks can handle raw data with little pre-processing, such as scaling and differencing. Nevertheless, when it comes to time series data, sometimes differencing the series can make a problem easier to model. Recall that differencing is the transform of the data such that a value of a prior observation is subtracted from the current observation, removing trend or seasonality structure. We will add support for differencing to the grid search test harness, just in case it adds value to your specific problem. It does add value for the internal airline passengers dataset. The `difference()` function below will calculate the difference of a given order for the dataset.

```
# difference dataset
def difference(data, order):
  return [data[i] - data[i - order] for i in range(order, len(data))]
```

Listing 15.21: Example of a function for differencing a dataset.

Differencing will be optional, where an order of 0 suggests no differencing, whereas an order 1 or order 12 will require that the data be differenced prior to fitting the model and that the predictions of the model will need the differencing reversed prior to returning the forecast. We can now define the elements required to fit the MLP model in the test harness. First, we must unpack the list of hyperparameters.

```
# unpack config
n_input, n_nodes, n_epochs, n_batch, n_diff = config
```

Listing 15.22: Example of unpacking MLP hyperparameters.

Next, we must prepare the data, including the differencing, transforming the data to a supervised format and separating out the input and output aspects of the data samples.

```
# prepare data
if n_diff > 0:
  train = difference(train, n_diff)
# transform series into supervised format
data = series_to_supervised(train, n_input)
# separate inputs and outputs
train_x, train_y = data[:, :-1], data[:, -1]
```

Listing 15.23: Example of preparing data for fitting the MLP model.

We can now define and fit the model with the provided configuration.

```
# define model
model = Sequential()
model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
# fit model
model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
```

Listing 15.24: Example of defining an MLP model.

The complete implementation of the `model_fit()` function is listed below.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # define model
  model = Sequential()
  model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit model
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 15.25: Example of a function for fitting an MLP model for a given configuration.

The five chosen hyperparameters are by no means the only or best hyperparameters of the model to tune. You may modify the function to tune other parameters, such as the addition and size of more hidden layers and much more. Once the model is fit, we can use it to make forecasts. If the data was differenced, the difference must be inverted for the prediction of the model. This involves adding the value at the relative offset from the history back to the value predicted by the model.

```
# invert difference
correction = 0.0
if n_diff > 0:
  correction = history[-n_diff]
...
# correct forecast if it was differenced
return correction + yhat[0]
```

Listing 15.26: Example of inverting any differencing performed.

It also means that the history must be differenced so that the input data used to make the prediction has the expected form.

```
# calculate difference
history = difference(history, n_diff)
```

Listing 15.27: Example of differencing the history prior to making a prediction.

Once prepared, we can use the history data to create a single sample as input to the model for making a one-step prediction. The shape of one sample must be `[1, n_input]` where `n_input` is the chosen number of lag observations to use.

```
# shape input for model
x_input = array(history[-n_input:]).reshape((1, n_input))
```

Listing 15.28: Example of preparing one sample ready for making a forecast.

Finally, a prediction can be made.

```
# make forecast
yhat = model.predict(x_input, verbose=0)
```

Listing 15.29: Example of making a forecast with a single sample of data.

The complete implementation of the `model_predict()` function is listed below. Next, we must define the range of values to try for each hyperparameter. We can define a `model_configs()` function that creates a list of the different combinations of parameters to try. We will define a small subset of configurations to try as an example, including a differencing of 12 months, which we expect will be required. You are encouraged to experiment with standalone models, review learning curve diagnostic plots, and use information about the domain to set ranges of values of the hyperparameters to grid search.

You are also encouraged to repeat the grid search to narrow in on ranges of values that appear to show better performance. An implementation of the `model_configs()` function is listed below.

```
# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_nodes = [50, 100]
  n_epochs = [100]
  n_batch = [1, 150]
  n_diff = [0, 12]
  # create configs
  configs = list()
  for i in n_input:
    for j in n_nodes:
      for k in n_epochs:
        for l in n_batch:
          for m in n_diff:
            cfg = [i, j, k, l, m]
            configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs
```

Listing 15.30: Example of a function for preparing a list of model configurations to evaluate.

We now have all of the pieces needed to grid search MLP models for a univariate time series forecasting problem. The complete example is listed below.

```
# grid search mlps for monthly airline passengers dataset
from math import sqrt
from numpy import array
from numpy import mean
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

```
# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, order):
  return [data[i] - data[i - order] for i in range(order, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_in=n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # define model
  model = Sequential()
  model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit model
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with the fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  # shape input for model
  x_input = array(history[-n_input:]).reshape((1, n_input))
```

```
  # make forecast
  yhat = model.predict(x_input, verbose=0)
  # correct forecast if it was differenced
  return correction + yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_nodes = [50, 100]
  n_epochs = [100]
  n_batch = [1, 150]
  n_diff = [0, 12]
  # create configs
  configs = list()
```

```
  for i in n_input:
    for j in n_nodes:
      for k in n_epochs:
        for l in n_batch:
          for m in n_diff:
            cfg = [i, j, k, l, m]
            configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs

# define dataset
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# model configs
cfg_list = model_configs()
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 15.31: Example of demonstrating the grid search framework for evaluating a MLP model configurations on the airline passengers dataset.

Running the example, we can see that there are a total of eight configurations to be evaluated by the framework. Each config will be evaluated 10 times; that means 10 models will be created and evaluated using walk-forward validation to calculate an RMSE score before an average of those 10 scores is reported and used to score the configuration. The scores are then sorted and the top 3 configurations with the lowest RMSE are reported at the end. A skillful model configuration was found as compared to a naive model that reported an RMSE of 50.70. We can see that the best RMSE of 18.98 was achieved with a configuration of `[12, 100, 100, 1, 12]`, which we know can be interpreted as:

- `n_input`: 12
- `n_nodes`: 100
- `n_epochs`: 100
- `n_batch`: 1
- `n_diff`: 12

A truncated example output of the grid search is listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
Total configs: 8
 > 20.707
 > 29.111
 > 17.499
 > 18.918
 > 28.817
...
 > 21.015
 > 20.208
 > 18.503
> Model[[12, 100, 100, 150, 12]] 19.674
done

[12, 100, 100, 1, 12] 18.982720013625606
[12, 50, 100, 150, 12] 19.33004059448595
[12, 100, 100, 1, 0] 19.5389405532858
```

Listing 15.32: Example output from demonstrating the grid search framework for evaluating a MLP model configurations on the airline passengers dataset.

15.5 Convolutional Neural Network Model

We can now adapt the framework to grid search CNN models. For more details on modeling a univariate time series with a CNN, see Chapter 8. Much the same set of hyperparameters can be searched as with the MLP model, except the number of nodes in the hidden layer can be replaced by the number of filter maps and kernel size in the convolutional layers. The chosen set of hyperparameters to grid search in the CNN model are as follows:

- `n_input`: The number of prior inputs to use as input for the model (e.g. 12 months).
- `n_filters`: The number of filter maps in the convolutional layer (e.g. 32).
- `n_kernel`: The kernel size in the convolutional layer (e.g. 3).
- `n_epochs`: The number of training epochs (e.g. 1000).
- `n_batch`: The number of samples to include in each mini-batch (e.g. 32).
- `n_diff`: The difference order (e.g. 0 or 12).

Some additional hyperparameters that you may wish to investigate are the use of two convolutional layers before a pooling layer, the repetition of the convolutional and pooling layer pattern, the use of dropout, and more. We will define a very simple CNN model with one convolutional layer and one max pooling layer.

```
# define model
model = Sequential()
model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
    input_shape=(n_input, n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
```

```
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
```

Listing 15.33: Example of defining a CNN model.

The data must be prepared in much the same way as for the MLP. Unlike the MLP that expects the input data to have the shape `[samples, features]`, the 1D CNN model expects the data to have the shape `[samples, timesteps, features]` where features maps onto channels and in this case 1 for the one variable we measure each month.

```
# reshape input data into [samples, timesteps, features]
n_features = 1
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
```

Listing 15.34: Example of reshaping data for the CNN model.

The complete implementation of the `model_fit()` function is listed below.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_filters, n_kernel, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # reshape input data into [samples, timesteps, features]
  n_features = 1
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
      input_shape=(n_input, n_features)))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 15.35: Example of a function for fitting a CNN model for a given configuration.

Making a prediction with a fit CNN model is very much like making a prediction with a fit MLP. Again, the only difference is that the one sample worth of input data must have a three-dimensional shape.

```
x_input = array(history[-n_input:]).reshape((1, n_input, 1))
```

Listing 15.36: Example of reshaping one sample for making a forecast.

The complete implementation of the `model_predict()` function is listed below.

```
# forecast with the fit model
def model_predict(model, history, config):
```

```
  # unpack config
  n_input, _, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]
```

Listing 15.37: Example of a function for making a forecast with a fit CNN model.

Finally, we can define a list of configurations for the model to evaluate. As before, we can do this by defining lists of hyperparameter values to try that are combined into a list. We will try a small number of configurations to ensure the example executes in a reasonable amount of time. The complete `model_configs()` function is listed below.

```
# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_filters = [64]
  n_kernels = [3, 5]
  n_epochs = [100]
  n_batch = [1, 150]
  n_diff = [0, 12]
  # create configs
  configs = list()
  for a in n_input:
    for b in n_filters:
      for c in n_kernels:
        for d in n_epochs:
          for e in n_batch:
            for f in n_diff:
              cfg = [a,b,c,d,e,f]
              configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs
```

Listing 15.38: Example of a function for preparing a list of model configurations to evaluate.

We now have all of the elements needed to grid search the hyperparameters of a convolutional neural network for univariate time series forecasting. The complete example is listed below.

```
# grid search cnn for monthly airline passengers dataset
from math import sqrt
from numpy import array
from numpy import mean
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
```

```
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, order):
  return [data[i] - data[i - order] for i in range(order, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_filters, n_kernel, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_in=n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # reshape input data into [samples, timesteps, features]
  n_features = 1
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
      input_shape=(n_input, n_features)))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

```
# forecast with the fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores
```

```
# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_filters = [64]
  n_kernels = [3, 5]
  n_epochs = [100]
  n_batch = [1, 150]
  n_diff = [0, 12]
  # create configs
  configs = list()
  for a in n_input:
    for b in n_filters:
      for c in n_kernels:
        for d in n_epochs:
          for e in n_batch:
            for f in n_diff:
              cfg = [a,b,c,d,e,f]
              configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs

# define dataset
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# model configs
cfg_list = model_configs()
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 10 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 15.39: Example of demonstrating the grid search framework for evaluating a CNN model configurations on the airline passengers dataset.

Running the example, we can see that only eight distinct configurations are evaluated. We can see that a configuration of `[12, 64, 5, 100, 1, 12]` achieved an RMSE of 18.89, which is skillful as compared to a naive forecast model that achieved 50.70. We can unpack this configuration as:

- `n_input`: 12
- `n_filters`: 64
- `n_kernel`: 5
- `n_epochs`: 100
- `n_batch`: 1
- `n_diff`: 12

A truncated example output of the grid search is listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
Total configs: 8
 > 23.372
 > 28.317
 > 31.070
...
 > 20.923
 > 18.700
 > 18.210
> Model[[12, 64, 5, 100, 150, 12]] 19.152
done

[12, 64, 5, 100, 1, 12] 18.89593462072732
[12, 64, 5, 100, 150, 12] 19.152486150334234
[12, 64, 3, 100, 150, 12] 19.44680151564605
```

Listing 15.40: Example output from demonstrating the grid search framework for evaluating a CNN model configurations on the airline passengers dataset.

15.6 Long Short-Term Memory Network Model

We can now adopt the framework for grid searching the hyperparameters of an LSTM model. For more details on modeling a univariate time series with an LSTM network, see Chapter 9. The hyperparameters for the LSTM model will be the same five as the MLP; they are:

- `n_input`: The number of prior inputs to use as input for the model (e.g. 12 months).
- `n_nodes`: The number of nodes to use in the hidden layer (e.g. 50).
- `n_epochs`: The number of training epochs (e.g. 1000).
- `n_batch`: The number of samples to include in each mini-batch (e.g. 32).
- `n_diff`: The difference order (e.g. 0 or 12).

We will define a simple LSTM model with a single hidden LSTM layer and the number of nodes specifying the number of units in this layer.

```
# define model
model = Sequential()
model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, n_features)))
model.add(Dense(n_nodes, activation='relu'))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
# fit model
model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
```

Listing 15.41: Example of defining an LSTM model.

It may be interesting to explore tuning additional configurations such as the use of a bidirectional input layer, stacked LSTM layers, and even hybrid models with CNN or ConvLSTM input models. As with the CNN model, the LSTM model expects input data to have a three-dimensional shape for the samples, time steps, and features.

```
# reshape input data into [samples, timesteps, features]
n_features = 1
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
```

Listing 15.42: Example of reshaping training data for the LSTM model.

The complete implementation of the `model_fit()` function is listed below.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # reshape input data into [samples, timesteps, features]
  n_features = 1
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
  # define model
  model = Sequential()
  model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, n_features)))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit model
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 15.43: Example of a function for fitting an LSTM model.

Also like the CNN, the single input sample used to make a prediction must also be reshaped into the expected three-dimensional structure.

```
# reshape sample into [samples, timesteps, features]
x_input = array(history[-n_input:]).reshape((1, n_input, 1))
```

Listing 15.44: Example of reshaping a single sample for making a prediction.

The complete `model_predict()` function is listed below.

```
# forecast with the fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
```

```
  # reshape sample into [samples, timesteps, features]
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]
```

Listing 15.45: Example of a function for making a forecast with a fit LSTM model.

We can now define the function used to create the list of model configurations to evaluate. The LSTM model is quite a bit slower to train than MLP and CNN models; as such, you may want to evaluate fewer configurations per run. We will define a very simple set of two configurations to explore: stochastic and batch gradient descent.

```
# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_nodes = [100]
  n_epochs = [50]
  n_batch = [1, 150]
  n_diff = [12]
  # create configs
  configs = list()
  for i in n_input:
    for j in n_nodes:
      for k in n_epochs:
        for l in n_batch:
          for m in n_diff:
            cfg = [i, j, k, l, m]
            configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs
```

Listing 15.46: Example of a function for defining model configurations to evaluate.

We now have everything we need to grid search hyperparameters for the LSTM model for univariate time series forecasting. The complete example is listed below.

```
# grid search lstm for monthly airline passengers dataset
from math import sqrt
from numpy import array
from numpy import mean
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
```

```
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, order):
  return [data[i] - data[i - order] for i in range(order, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_in=n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # reshape input data into [samples, timesteps, features]
  n_features = 1
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
  # define model
  model = Sequential()
  model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, n_features)))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit model
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with the fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  # reshape sample into [samples, timesteps, features]
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
```

```
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_nodes = [100]
  n_epochs = [50]
  n_batch = [1, 150]
  n_diff = [12]
  # create configs
  configs = list()
  for i in n_input:
```

```
    for j in n_nodes:
      for k in n_epochs:
        for l in n_batch:
          for m in n_diff:
            cfg = [i, j, k, l, m]
            configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs

# define dataset
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# model configs
cfg_list = model_configs()
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 10 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 15.47: Example of demonstrating the grid search framework for evaluating an LSTM model configurations on the airline passengers dataset.

Running the example, we can see that only two distinct configurations are evaluated. We can see that a configuration of `[12, 100, 50, 1, 12]` achieved an RMSE of 21.24, which is skillful as compared to a naive forecast model that achieved 50.70. The model requires a lot more tuning and may do much better with a hybrid configuration, such as having a CNN model as input. We can unpack this configuration as:

- `n_input`: 12
- `n_nodes`: 100
- `n_epochs`: 50
- `n_batch`: 1
- `n_diff`: 12

A truncated example output of the grid search is listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
Total configs: 2
 > 20.488
 > 17.718
 > 21.213
...
 > 22.300
 > 20.311
```

```
 > 21.322
> Model[[12, 100, 50, 150, 12]] 21.260
done

[12, 100, 50, 1, 12] 21.243775750634093
[12, 100, 50, 150, 12] 21.259553398553606
```

Listing 15.48: Example output from demonstrating the grid search framework for evaluating an LSTM model configurations on the airline passengers dataset.

15.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **More Configurations**. Explore a large suite of configurations for one of the models and see if you can find a configuration that results in better performance.
- **Data Scaling**. Update the grid search framework to also support the scaling (normalization and/or standardization) of data both before fitting the model and inverting the transform for predictions.
- **Network Architecture**. Explore the grid searching larger architectural changes for a given model, such as the addition of more hidden layers.
- **New Dataset**. Explore the grid search of a given model in a new univariate time series dataset.
- **Multivariate**. Update the grid search framework to support small multivariate time series datasets, e.g. datasets with multiple input variables.

If you explore any of these extensions, I'd love to know.

15.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Keras Sequential Model API.
  https://keras.io/models/sequential/
- Keras Core Layers API.
  https://keras.io/layers/core/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/
- Keras Pooling Layers API.
  https://keras.io/layers/pooling/
- Keras Recurrent Layers API.
  https://keras.io/layers/recurrent/

15.9 Summary

In this tutorial, you discovered how to develop a framework to grid search hyperparameters for deep learning models. Specifically, you learned:

- How to develop a generic grid searching framework for tuning model hyperparameters.
- How to grid search hyperparameters for a Multilayer Perceptron model on the airline passengers univariate time series forecasting problem.
- How to adapt the framework to grid search hyperparameters for convolutional and long short-term memory neural networks.

15.9.1 Next

This is the final lesson of this part, the next part will focus how to systematically work through a real-world multivariate multi-step time series problem to forecast household energy usage.