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 three 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.

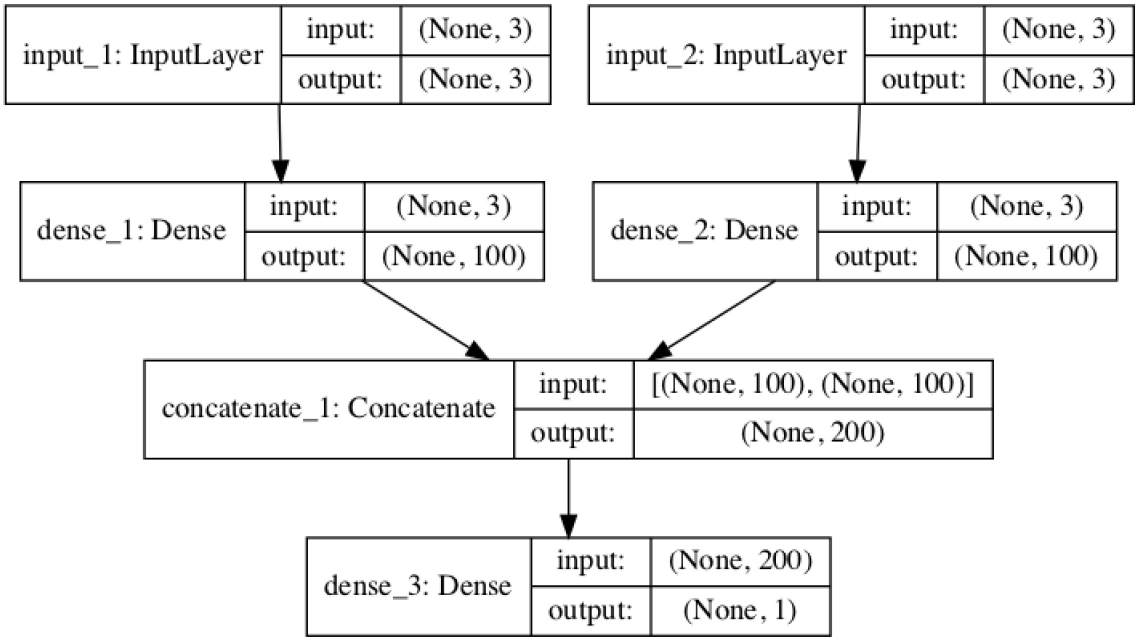

Figure 7.1: Plot of Multi-headed MLP for Multivariate Time Series Forecasting.

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, 9]` 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.

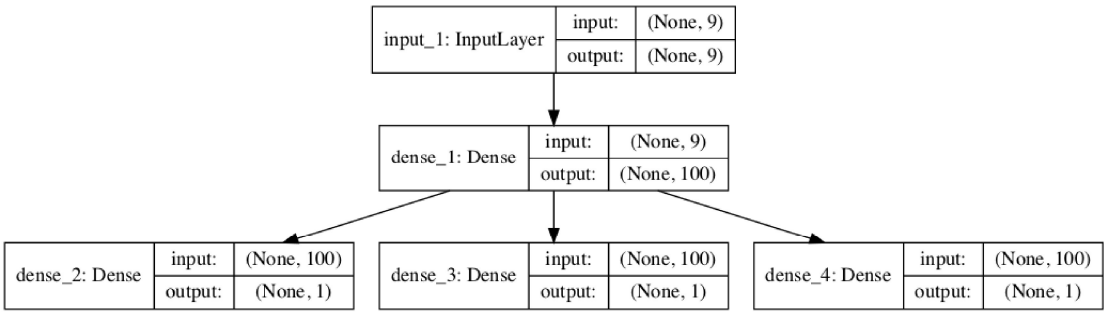

Figure 7.2: Plot of Multi-output MLP for Multivariate Time Series Forecasting.

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 where 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(64, 2, activation='relu', input_shape=(n_steps, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu', input_shape=(n_steps, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu', input_shape=(n_steps, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu', input_shape=(n_steps, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu')(visible1)
cnn1 = MaxPooling1D()(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(64, 2, activation='relu')(visible2)
cnn2 = MaxPooling1D()(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.

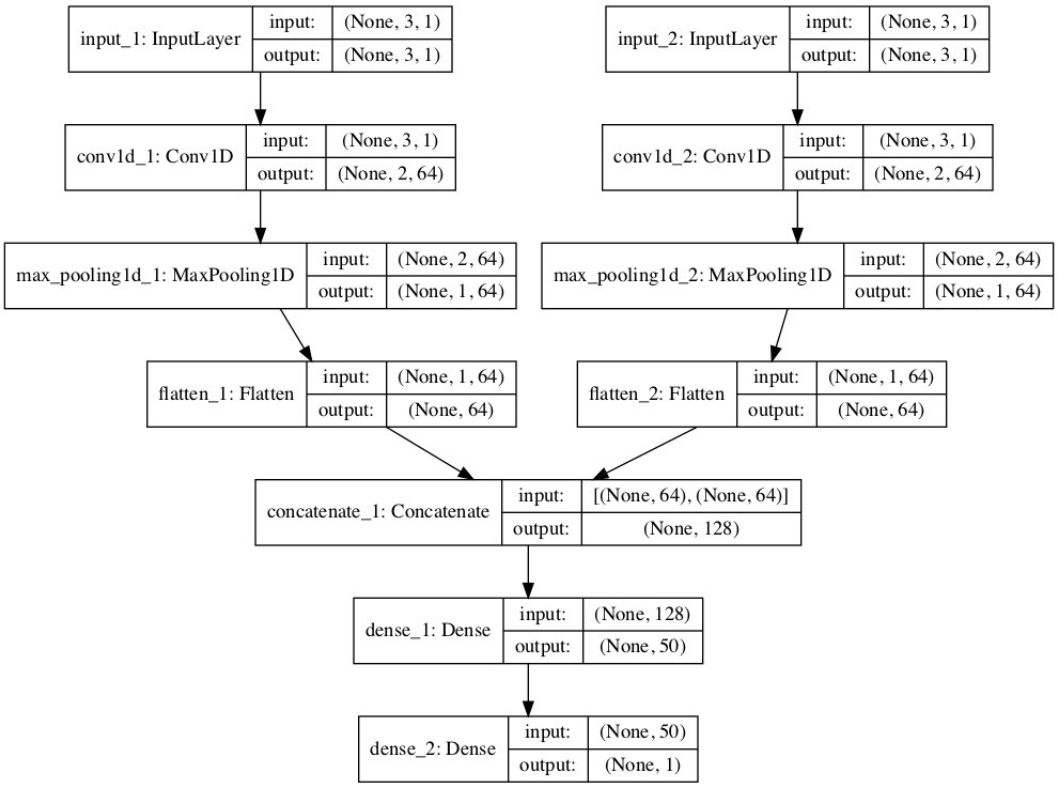

Figure 8.1: Plot of Multi-headed 1D CNN for Multivariate Time Series Forecasting.

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(64, 2, activation='relu')(visible1)
cnn1 = MaxPooling1D()(cnn1)
cnn1 = Flatten()(cnn1)
# second input model
visible2 = Input(shape=(n_steps, n_features))
cnn2 = Conv1D(64, 2, activation='relu')(visible2)
cnn2 = MaxPooling1D()(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(64, 2, activation='relu', input_shape=(n_steps, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu', input_shape=(n_steps, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu')(visible)
cnn = MaxPooling1D()(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.

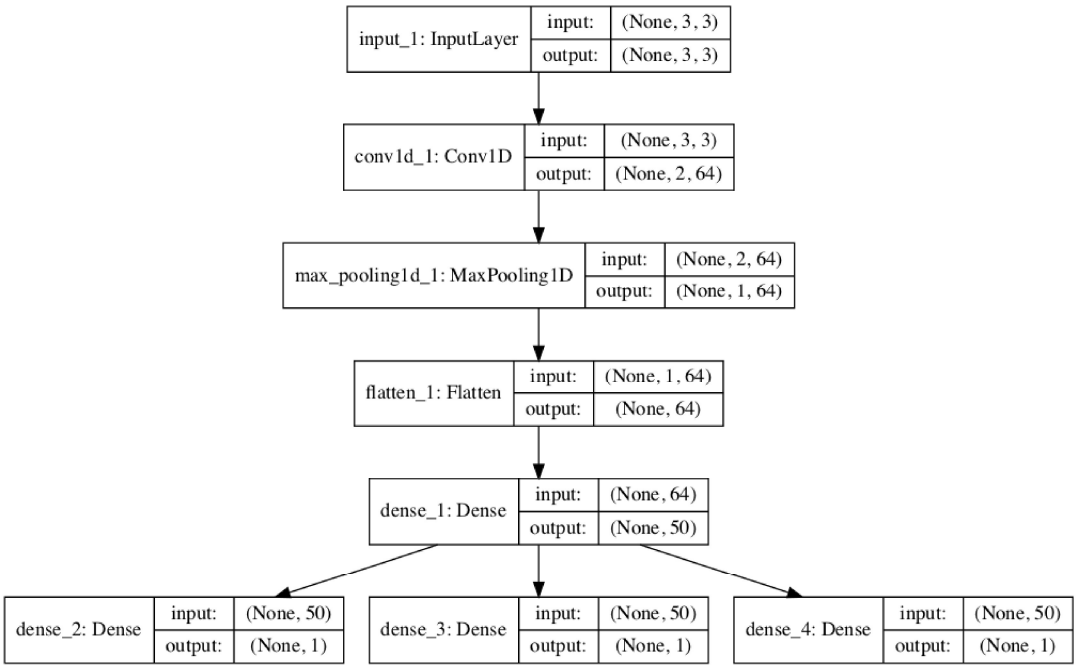

Figure 8.2: Plot of Multi-output 1D CNN for Multivariate Time Series Forecasting.

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(64, 2, activation='relu')(visible)
cnn = MaxPooling1D()(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(64, 2, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(MaxPooling1D())
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(64, 2, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(MaxPooling1D())
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(64, 1, activation='relu'), input_shape=(None, n_steps,
    n_features)))
model.add(TimeDistributed(MaxPooling1D()))
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(64, 1, activation='relu'), input_shape=(None, n_steps,
    n_features)))
model.add(TimeDistributed(MaxPooling1D()))
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(64, (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(64, (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.