MACHINE LEARNING MASTERY

Deep Learning FOR TIME SERIES FORECASTING

Predict the Future with MLPs, CNNs, and LSTMs in Python

Jason Brownlee

Disclaimer

Acknowledgements

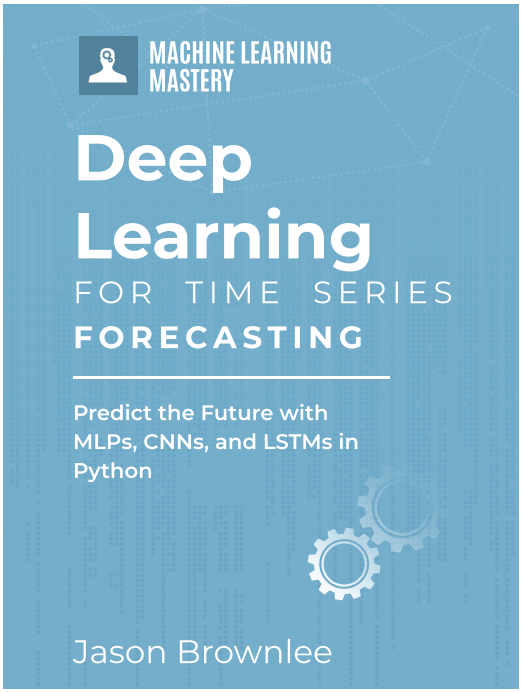
Special thanks to my copy editor Sarah Martin and my technical editors Arun Koshy and Andrei Cheremskoy.

Copyright

Edition: v1.91

This is Just a Sample

Thank-you for your interest in **Deep Learning for Time Series Forecasting**.
This is just a sample of the full text. You can purchase the complete book online from:
https://machinelearningmastery.com/deep-learning-for-time-series-forecasting/



Contents

Preface

Perhaps the topic that I am most asked about is how to use deep learning methods for time series forecasting. Questions range from how to prepare data for deep learning models to what models to use for specific types of forecasting problems. This book was carefully designed to address these questions and show you exactly how to apply deep learning methods to time series forecasting problems.

In writing this book, I imagined that you were provided with a dataset and a desire to use deep learning methods to address it. I designed the chapters to walk you through the process of first establishing a baseline of performance with naive and classical methods. I then provide step-by-step tutorials to show exactly how to develop a suite of different types of neural network models for time series forecasting. After we cover these basics, I then hammer home how to use them on real-world datasets with example after example on larger projects. This is not a book for beginners. The focus on deep learning methods means that we won't focus on many other important areas of time series forecasting, such as data visualization, how classical methods work, the development of machine learning solutions, or even depth and details on how the deep learning methods work. I assume that you are familiar with these introductory topics.

Deep learning methods do offer a lot of promise for time series forecasting, specifically the automatic learning of temporal dependence and the automatic handling of temporal structures like trends and seasonality. Unfortunately, deep learning methods are often developed for univariate time series forecasting problems and often perform worse than classical and even naive forecasting methods. This has led to their general dismissal as being unsuitable for time series forecasting in general. Nevertheless, deep learning methods are effective at more complex time series forecasting problems that involve large amounts of data, multiple variates with complex relationships, and even multi-step and time series classification tasks.

A limitation I saw in the adoption of deep learning methods for time series forecasting was in the exclusive exploration of recurrent neural networks, such as LSTM networks. These methods can work well in some situations, but can often perform better when used in hybrid models with CNNs or other variations. Therefore, I have ensured that examples of each class of neural network and hybrid models were demonstrated throughout the book. In addition to providing a playbook to show you how to develop deep learning models for your own time series forecasting problems, I designed this book to highlight the areas where deep learning methods may show the most promise. Deep learning may be the future of complex and challenging time series forecasting and I think this book will help you get started and make rapid progress on your own forecasting problems. I hope that you agree and are as excited as I am about the journey ahead.

Jason Brownlee
2018

Part I

Introduction

Welcome

Welcome to *Deep Learning for Time Series Forecasting*. Deep learning methods, such as Multilayer Perceptrons, Convolutional Neural Networks, and Long Short-Term Memory Networks, can be used to automatically learn the temporal dependence structures for challenging time series forecasting problems. Neural networks may not be the best solution for all time series forecasting problems, but for those problems where classical methods fail and machine learning methods require elaborate feature engineering, deep learning methods can be used with great success. This book was designed to teach you, step-by-step, how to develop deep learning methods for time series forecasting with concrete and executable examples in Python.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that may know some applied machine learning. Maybe you know how to work through a predictive modeling problem end-to-end, or at least most of the main steps, with popular tools. The lessons in this book do assume a few things about you, such as:

- You know your way around basic Python for programming.
- You know some basic time series forecasting with classical methods such as naive forecasts, ARIMA, or exponential smoothing.
- You know how to work through a predictive modeling problem using machine learning or deep learning methods and libraries such as scikit-learn or Keras.
- You have some background knowledge on how neural network models work, perhaps at a high level.
- You want to learn how to develop a suite of deep learning methods to get more from new or existing time series forecasting problems.

This guide was written in the top-down and results-first machine learning style that you're used to from Machine Learning Mastery.

About Your Outcomes

This book will teach you the practical deep learning skills that you need to know for time series forecasting. After reading and working through this book, you will know:

- About the promise of neural networks and deep learning methods in general for time series forecasting.
- How to transform time series data in order to train a supervised learning algorithm, such as deep learning methods.
- How to develop baseline forecasts using naive and classical methods by which to determine whether forecasts from deep learning models have skill or not.
- How to develop Multilayer Perceptron, Convolutional Neural Network, Long Short-Term Memory Networks, and hybrid neural network models for time series forecasting.
- How to forecast univariate, multivariate, multi-step, and multivariate multi-step time series forecasting problems in general.
- How to transform sequence data into a three-dimensional structure in order to train convolutional and LSTM neural network models.
- How to grid search deep learning model hyperparameters to ensure that you are getting good performance from a given model.
- How to prepare data and develop deep learning models for forecasting a range of univariate time series problems with different temporal structures.
- How to prepare data and develop deep learning models for multi-step forecasting a real-world household electricity consumption dataset.
- How to prepare data and develop deep learning models for a real-world human activity recognition project.

This new understanding of applied deep learning methods will impact your practice of working through time series forecasting problems in the following ways:

- Confidently use naive and classical methods like SARIMA and ETS to quickly develop robust baseline models for a range of different time series forecasting problems, the performance of which can be used to challenge whether more elaborate machine learning and deep learning models are adding value.
- Transform native time series forecasting data into a form for fitting supervised learning algorithms and confidently tune the amount of lag observations and framing of the prediction problem.
- Develop MLP, CNN, RNN, and hybrid deep learning models quickly for a range of different time series forecasting problems, and confidently evaluate and interpret their performance.

This book is not a substitute for an undergraduate course in deep learning or time series forecasting, nor is it a textbook for such courses, although it could be a useful complement. For a good list of top courses, textbooks, and other resources on statistics, see the *Further Reading* section at the end of each tutorial.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific method or type of problem, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed around major deep learning techniques that are directly relevant to time series forecasting. There are a lot of things you could learn about deep learning and time series forecasting, from theory to abstract concepts to APIs. My goal is to take you straight to developing an intuition for the elements you must understand with laser-focused tutorials. I designed the tutorials to focus on how to get results with deep learning methods. The tutorials give you the tools to both rapidly understand and apply each technique or operation. There is a mixture of both tutorial lessons and projects to both introduce the methods and give plenty examples and opportunity to practice using them.

Each of the tutorials are designed to take you about one hour to read through and complete, excluding the extensions and further reading. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I would recommend picking a schedule and sticking to it. The tutorials are divided into five parts; they are:

- **Part 1: Foundations**. Provides a gentle introduction to the promise of deep learning methods for time series forecasting, a taxonomy of the types of time series forecasting problems, how to prepare time series data for supervised learning, and a high-level procedure for getting the best performing model on time series forecasting problems in general.

- **Part 2: Deep Learning Modeling**. Provides a step-by-step introduction to deep learning methods applied to different types of time series forecasting problems with additional tutorials to better understand the 3D-structure required for some models.

- **Part 3: Univariate Forecasting**. Provides a methodical approach to univariate time series forecasting with a focus on naive and classical methods that are generally known to out-perform deep learning methods and how to grid search deep learning model hyperparameters.

- **Part 4: Multi-step Forecasting**. Provides a step-by-step series of tutorials for working through a challenging multi-step time series forecasting problem for predicting household electricity consumption using classical and deep learning methods.

- **Part 5: Time Series Classification**. Provides a step-by-step series of tutorials for working through a challenging time series classification problem for predicting human activity from accelerometer data using deep learning methods.

Each part targets a specific learning outcome, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of digressions. The tutorials were not designed to teach you everything there is to know about each of the methods. They were designed to give you an understanding of how they work, how to use them, and how to interpret the results the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third-parties required beyond the installation of the required packages. A complete working example is presented with each tutorial for you to inspect and copy-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues.

The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead. All code examples were tested on a POSIX-compatible machine with Python 3.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Books and book chapters.
- API documentation.
- Articles and Webpages.

Wherever possible, I try to list and link to the relevant API documentation for key functions used in each lesson so you can learn more about them. I have tried to link to books on Amazon so that you can learn more about them. I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry; you are not alone.

- **Help with a Technique?** If you need help with the technical aspects of a specific operation or technique, see the *Further Reading* section at the end of each tutorial.

- **Help with APIs?** If you need help with using the Keras library, see the list of resources in the *Further Reading* section at the end of each lesson, and also see *Appendix A*.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.
- **Help with practice problems?** If you are looking for test sets on which to practice developing deep learning models, you can see a prepared list of standard time series forecasting problems in *Appendix A*.
- Help in general? You can shoot me an email. My details are in *Appendix A*.

Summary

Are you ready? Let's dive in! Next up you will discover a gentle introduction to the promise of deep learning methods for time series forecasting.

Part II

Deep Learning Methods

Chapter 1

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.

1.1 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.

1.2 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

1.2.1 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 1.1: 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 1.2: 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 1.3: 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 1.4: 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 1.5: 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.

1.2.2 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 1.6: 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 1.7: 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 1.8: 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 1.9: 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 1.10: 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 **??**. For an example of grid searching MLP hyperparameters on a univariate time series forecasting problem, see Chapter **??**.

1.3 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.

1.3.1 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 1.11: 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 1.12: 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 1.13: 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 1.14: 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 1.15: Example input data for the first data sample.

Output:

```
65
```

Listing 1.16: 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 1.17: 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 1.18: 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 1.19: 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 1.20: Example input data for the first data sample.

Becomes:

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

Listing 1.21: 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 1.22: 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 1.23: 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 1.24: 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 1.25: 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 1.26: 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 1.27: 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 1.28: 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 1.29: 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 1.30: 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 1.31: 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.

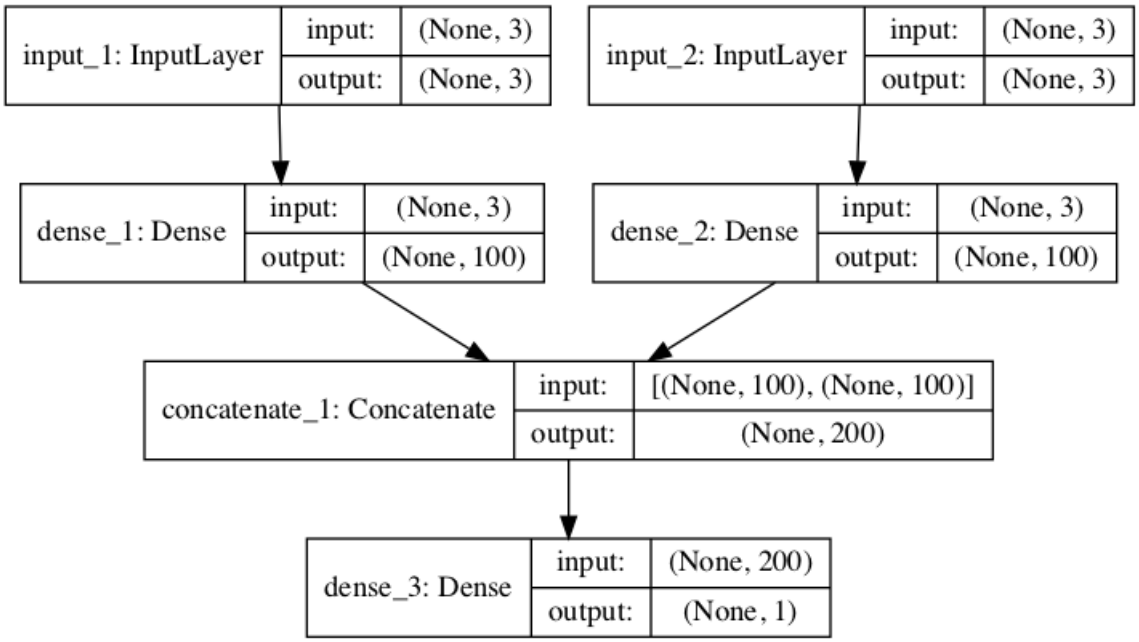


Figure 1.1: Plot of Multi-headed MLP for Multivariate Time Series Forecasting.

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 1.32: 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 1.33: 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 1.34: 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 1.35: 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 1.36: Example output from using a Multi-headed MLP to forecast a dependent time series.

1.3.2 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 1.37: 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 1.38: Example input from the first data sample.

Output:

```
40, 45, 85
```

Listing 1.39: 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 1.40: 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 1.41: 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 1.42: 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 1.43: 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 1.44: 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 1.45: 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 1.46: 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 1.47: 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 1.48: 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 1.49: 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 1.50: 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 1.51: 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 1.52: 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 1.53: 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.

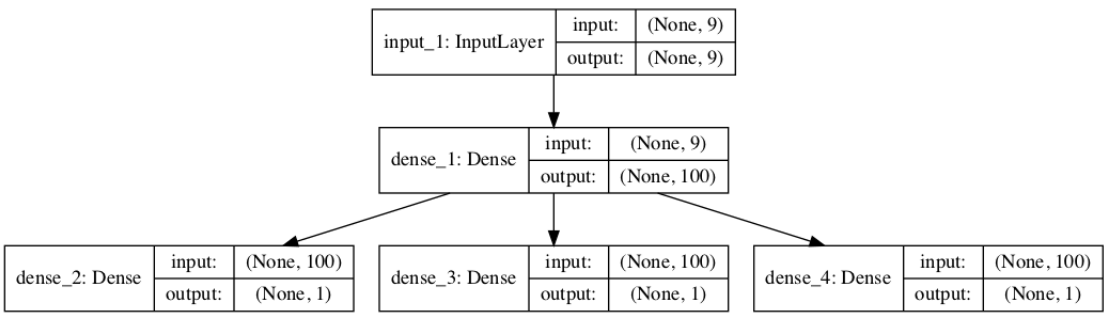


Figure 1.2: Plot of Multi-output MLP for Multivariate Time Series Forecasting.

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 1.54: 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 1.55: 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 1.56: 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 1.57: Example output from a multi-output MLP for multivariate time series forecasting.

1.4 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.

1.4.1 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 1.58: 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 1.59: Example input for a multi-step forecast.

Output:

```
[40, 50]
```

Listing 1.60: 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 1.61: 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 1.62: 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 1.63: 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.

1.4.2 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 1.64: 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 1.65: Example input for making an out-of-sample multi-step forecast.

We would expect the predicted output to be:

```
[100, 110]
```

Listing 1.66: 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 1.67: 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 1.68: 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 1.69: Example output from an MLP for multi-step time series forecasting.

1.5 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.

1.5.1 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 1.70: 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 1.71: Example input for a multi-step forecast for a dependent series.

Output:

```
65
85
```

Listing 1.72: 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 1.73: 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 1.74: 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 1.75: 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 1.76: 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 1.77: Example output from an MLP model for multi-step forecasting for a dependent series.

1.5.2 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 1.78: 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 1.79: Example input for the first a multivariate time series sample.

Output:

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

Listing 1.80: 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 1.81: 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 1.82: 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 1.83: 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 1.84: 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 1.85: 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 1.86: 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 1.87: Example output from an MLP model for multi-step forecasting for a multivariate series.

1.6 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.

1.7 Further Reading

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

1.7.1 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

1.7.2 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/

1.8 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.

1.8.1 Next

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

This is Just a Sample

Thank-you for your interest in **Deep Learning for Time Series Forecasting**.
This is just a sample of the full text. You can purchase the complete book online from:
https://machinelearningmastery.com/deep-learning-for-time-series-forecasting/