Ego autem et domus

mea serviemus Domino.

DEEP TIME SERIES FORECASTING With PYTHON

An Intuitive Introduction to Deep Learning for Applied Time Series Modeling

Dr. N.D Lewis

Image photography by Deanna Lewis with helpful assistance from Naomi Lewis.

ISBN-13: 978-1540809087
ISBN-10: 1540809080

Contents

Dedicated to Angela, wife, friend and mother extraordinaire.

Acknowledgments

A special thank you to:

My wife Angela, for her patience and constant encouragement.

My daughters Deanna and Naomi, for being helpful, taking hundreds of photographs for this book and my website.

And the readers of my earlier books who contacted me with questions and suggestions.

Master Deep Time Series Forecasting with Python!

Deep Time Series Forecasting with Python takes you on a gentle, fun and unhurried practical journey to creating deep neural network models for time series forecasting with Python. It uses plain language rather than mathematics; And is designed for working professionals, office workers, economists, business analysts and computer users who want to try deep learning on their own time series data using Python.

NO EXPERIENCE REQUIRED: I'm assuming you never did like linear algebra, don't want to see things derived, dislike complicated computer code, and you're here because you want to try deep learning time series forecasting models for yourself.

THIS BOOK IS FOR YOU IF YOU WANT:

- Explanations rather than mathematical derivation.
- Practical illustrations that use real data.
- Worked examples in Python you can easily follow and immediately implement.
- Ideas you can actually use and try out with your own data.

QUICK AND EASY: Using plain language, this book offers a simple, intuitive, practical, non-mathematical, easy to follow guide to the most successful ideas, outstanding techniques and usable solutions available using Python. Examples are clearly described and can be typed directly into Python as printed on the page.

TAKE THE SHORTCUT: **Deep Time Series Forecasting with Python** was written for people who want to get up to speed as soon as possible. In this book you will learn how to**:**

- **Unleash** the power of Long Short-Term Memory Neural Networks
- **Develop** hands on skills using the Gated Recurrent Unit Neural Network.
- **Design** successful applications with Recurrent Neural Networks.
- **Deploy** Nonlinear Auto-regressive Network with Exogenous Inputs
- **Adapt** Deep Neural Networks for Time Series Forecasting.
- **Master** strategies to build superior Time Series Models.

PRACTICAL, HANDS ON: Deep learning models are finding their way into regular use by practical forecasters. Through a simple to follow process you will learn how to build deep time series forecasting models using Python. Once you have mastered the process, it will be easy for you to translate your knowledge into your own powerful applications.

GET STARTED TODAY! Everything you need to get started is contained within this book. **Deep Time series Forecasting with Python** is your very own hands on practical, tactical, easy to follow guide to mastery.

Buy this book today and accelerate your progress!

Other Books by N.D Lewis

- **Deep Learning for Business with Python**
- **Deep Learning for Business with R**
- **Deep Learning Step by Step with Python**
- **Deep Learning Made Easy with R:**
  - Volume I: A Gentle Introduction for Data Science
  - Volume II: Practical Tools for Data Science
  - Volume III: Breakthrough Techniques to Transform Performance
- **Build Your Own Neural Network TODAY!**
- **92 Applied Predictive Modeling Techniques in R**
- **100 Statistical Tests in R**
- **Visualizing Complex Data Using R**
- **Learning from Data Made Easy with R**

For further detail's visit www.AusCov.com

Preface

If you are anything like me, you hate long prefaces. I don't care about the author's background. Nor do I need a lengthy overview of the history of what I am about to learn. Just tell me what I am going to learn, and then teach me how to do it. You are about to learn how to use a set of modern neural network tools to forecast time series data using Python. Here are the tools:

- Deep Neural Networks.
- Long Short-Term Memory Neural Network.
- Gated Recurrent Unit Neural Network.
- Simple Recurrent Neural Network.
- Elman Neural Network.
- Jordan Neural Network.
- Nonlinear Auto-regressive Network with Exogenous Inputs.
- Working with Multiple Outputs.
- Strategies to Build Superior models.

Caution!

If you are looking for detailed mathematical derivations, lemmas, proofs or implementation tips, please do not purchase this book. It contains none of those things.

You don't need to know complex mathematics, algorithms or object-oriented programming to use this text. It skips all that stuff and concentrates on sharing code, examples and illustrations that gets practical stuff done.

Before you buy this book, ask yourself the following tough questions. Are you willing to invest the time, and then work through the examples and illustrations required to take your knowledge to the next level? If the answer is yes, then by all means click that buy button so I can purchase my next cappuccino.

A Promise

No matter who you are, no matter where you are from, no matter your background or schooling, you have the ability to master the ideas outlined in this book. With the appropriate software tool, a little persistence and the right guide, I personally believe deep learning methods can be successfully used in the hands of anyone who has a real interest.

When you are done with this book, you will be able to implement one or more of the ideas I've talked about in your own particular area of interest. You will be amazed at how quick and easy the techniques are to develop and test. With only a few different uses you will soon become a skilled practitioner.

I invite you therefore to put what you read in these pages into action. To help you do that, I've created "**21 Tips For Data Science Success with Python**", it is yours for **FREE**. Simply go to *http://www.AusCov.com* and download it now. It is my gift to you. It shares with you some of the very best resources you can use to boost your productivity in Python.

Now, it's your turn!

How to Get the Absolute Most Possible Benefit from this Book

On its own, this book won't turn you into a deep learning time series guru any more than a few dance lessons will turn you into the principal dancer with the Royal Ballet in London. But if you're a working professional, economist, business analyst or just interested in trying out new machine learning ideas, you will learn the basics of deep learning for time series forecasting, and get to play with some cool tools. Once you have mastered the basics, you will be able to use your own data to effortlessly (and one hopes accurately) forecast the future.

It's no accident that the words simple, easy and gentle appear so often in this text. I have shelves filled with books about time series analysis, statistics, computer science and econometrics. Some are excellent, others are good, or at least useful enough to keep. But they range from very long to the very mathematical. I believe many working professionals want something short, simple with practical examples that are easy to follow and straightforward to implement. In short, a very gentle intuitive introduction to deep learning neural networks for applied time series modeling.

Also, almost all advice on machine learning for time series forecasting comes from academics; this comes from a practitioner. I have been a practitioner for most of my working life. I enjoy boiling down complex ideas and techniques into applied, simple and easy to understand language that works. Why spend five hours ploughing through technical equations, proofs and lemmas when the core idea can be explained in ten minutes and deployed in fifteen?

I wrote this book because I don't want you to spend your time struggling with the mechanics of implementation or theoretical details. That's why we have Ivy league (in the US) or Russell Group (in the UK) professors. Even if you've never attempted to forecast anything, you can easily make your computer do the grunt work. This book will teach you how to apply the very best Python tools to solve basic time series problems.

I want you to get the absolute most possible benefit from this book in the minimum amount of time. You can achieve this by typing in the examples, reading the reference material and most importantly experimenting. This book will deliver the most value to you if you do this. Successfully applying neural networks requires work, patience, diligence and most importantly experimentation and testing. By working through the numerous examples and reading the references, you will broaden your knowledge, deepen you intuitive understanding and strengthen your practical skill set.

As implied by the title, this book is about understanding and then hands use of neural networks for time series forecasting and analysis; more precisely, it is an attempt to give you the tools you need to build deep neural networks easily and quickly using Python. The objective is to provide you the reader with the necessary tools to do the job, and provide sufficient illustrations to make you think about genuine applications in your own field of interest. I hope the process is not only beneficial but enjoyable.

Getting Python

To use this book you will need to download a copy of Python. It comes in two major versions - Python 2 and Python 3. Although Python 3 was introduced several years ago, Python 2 has been around longer and remains the most popular version used for Data Science. The programs in this book are written to run on Python 2 and may not run correctly, if at all, on Python 3. You can download a copy of Python 2 (such as version as 2.7.11) at https://www.python.org/.

Alternative Distributions of Python

There are a large number of distributions of Python, many Data Scientists use the Anaconda Python distribution (https://www.continuum.io/downloads). It comes prepackaged with many of the core software modules used in data analysis and statistical modeling. The PyPy (http://pypy.org/) variant uses just-in-time compilation to accelerate code and therefore runs deep learning code pretty quickly. For Windows users, the WinPython (https://winpython.github.io/) is one of the easiest ways to run Python, without the installation burden.

Learning Python

Python is a powerful programming language which is easy for beginners to use and learn. If you have experience in any programming language you can pick up Python very quickly.

If you are new to the language, or have not used it in a while, refresh your memory with these free resources:

- The Python Tutorial - https://docs.python.org/2/tutorial/
- Python For Beginners - https://www.python.org/about/gettingstarted/

- A Beginner's Python Tutorial - https://en.wikibooks.org/wiki/A_Beginner%27s_Python_Tutorial
- A interactive Python tutorial - http://www.learnpython.org/

This book is a guide for beginners, not a reference manual. The coding style is designed to make the Python scripts easier to understand and simple to learn, rather than to illustrate Pythonic style, or software design best practices.

NOTE... ✍

Visit the Python Software Foundation community section. You will find the latest news, tips and tricks at https://www.python.org/community/.

Using Packages

The efficient use of Python requires the use of software modules called packages or libraries. Throughout this text we will use a number of packages. If a package mentioned in the text is not installed on your machine you need to download and install it.

For details on specific packages look at the Python Package Index - a repository of packages for Python at https://pypi.python.org/pypi.

Deep learning has a reputation as being difficult to learn and complicated to implement. Fortunately, there are a growing number of high level neural network libraries that enable people interested in using deep learning to quickly build and test models without worrying about the technical details surrounding floating point operations and linear algebra. We make use of many of these libraries throughout this text.

Additional Resources to Check Out

Data Science focused Python user groups are popping up everywhere. Look for one in your local town or city. Join it! Here are a few resources to get you started:

- A great place to start connecting with the Python community is the Python Software Foundation's community website:https://www.python.org/community/
- Get in contact with local Python coders. Many have regular meetings. Here are great places to search:
  - http://www.meetup.com/topics/python/
  - https://wiki.python.org/moin/LocalUserGroups
- A global directory is listed at: https://wiki.python.org/moin/LocalUserGroups.
- Keep in touch and up to date with useful information in my free newsletter. It is focused on news, articles, software, events, tools and jobs related to data science. Sign up at www.AusCov.Com.

Also look at the following articles:

- Dhar, Vasant. "Data science and prediction." Communications of the ACM 56.12 (2013): 64-73.
- Provost, Foster, and Tom Fawcett. Data Science for Business: What you need to know about data mining and data-analytic thinking. " O'Reilly Media, Inc.", 2013.
- Schutt, Rachel, and Cathy O'Neil. Doing data science: Straight talk from the frontline. " O'Reilly Media, Inc.", 2013.
- Behnel, Stefan, et al. "Cython: The best of both worlds." Computing in Science & Engineering 13.2 (2011): 31-39.

- Millman, K. Jarrod, and Michael Aivazis. "Python for scientists and engineers." Computing in Science & Engineering 13.2 (2011): 9-12.

NOTE... ✍

As you use the ideas in this book successfully in your own area of expertise, write and let me know. I'd love to hear from you. Email or visit www.AusCov.com.

Chapter 1

The Characteristics of Time Series Data Simplified

A time series is a discrete or continuous sequence of observations that depends on time. Time is an important feature in natural processes such as air temperature, pulse of the heart or waves crashing on a sandy beach. It is also important in many business processes such the total units of a newly released books sold in the first 30 days, or the number of calls received by a customer service center over a holiday weekend.

Time is a natural element that is always present when data is collected. Time series analysis involves working with time based data in order to make predictions about the future. The time period may be measured in years, seasons, months, days, hours, minutes, seconds or any other suitable unit of time.

The Data Generating Mechanism

Time series data are the output of a "*data generating process*". As illustrated in Figure 1.1, at each point in time a new obser-

vation is generated. So, for example, at time t we might observe an observation say y. We denote this by y_t. At the next time step, say $t+1$, we observe a new observation, which we denote y_{t+1}. The time step t, might be measured in seconds, hours, days, week, months, years and so on. For example, stock market volatility is calculated daily, and the unemployment rate reported monthly.

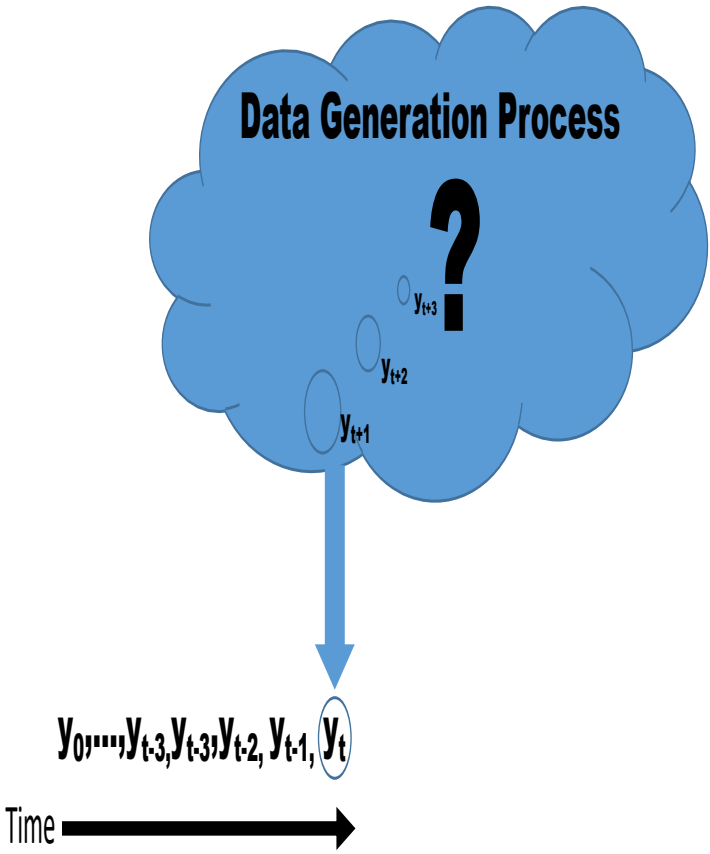


Figure 1.1: Data generating mechanism for time series data

Create a Simple Time Series using Python

Consider the simple data generation process:

$$y_t = \alpha + \beta y_{t-1} + \epsilon_t, \tag{1.1}$$

which tells us the observation today (time t) is calculated as the sum of a constant α, and a proportion yesterday's observation βy_{t-1}, plus a random error ϵ_t.

What does this series look like? Well, it clearly depends on the values of the initial observation y_0, the parameters β, α and the error ϵ_t. Suppose, $y_0 = 1$,$\beta = 0.95$, and $\alpha = -0.25$, you can generate some sample observations with the following code:

```
import pandas as pd
import numpy as np
seed =2016
np.random.seed(seed)
y_0 = 1
alpha = -0.25
beta =0.95
y=pd.Series(y_0)
num=10
for i in range(num):
        y_t = alpha+(beta*y_0)+ np.random.
           uniform(-1, 1)
        y=y.set_value(i,y_t)
        y_0=y_t
```

Here is how to read the above code:

- The first two lines import the `pandas` and `numpy` libraries. These two libraries are the twin workhorses of data science.
- The initial value of the series (y_0) is stored in the Python variable `y_0`.

- Equation parameters α, β are captured by `alpha` and `beta` respectively.
- Random errors (ϵ) are generated by calling the `random.uniform` method, and are constrained to lie between the value of -1 and +1.
- The code generates ten observation by setting (`num=10`) and using a simple `for` loop.

NOTE... ✍

The examples in this text were developed using Numpy 1.11.1rc1, Pandas 0.18.1 with Python 2.7.11.

View the Simulated Time Series

You can view the time series observations using the `print` statement:

```
print "Values of y are: \n",y
Values of y are:
0    1.000000
1    1.629219
2    1.864309
3    2.004397
4    1.578358
5    1.534571
6    0.657570
7    0.791785
8    0.996448
9    0.946840
```

Figure 1.2 plots the value of `y` for each time step, and Figure 1.3 plots the first 1000 observations.

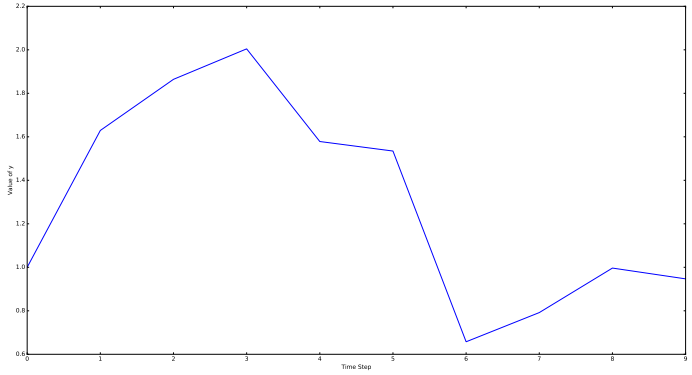


Figure 1.2: Simulated time Series

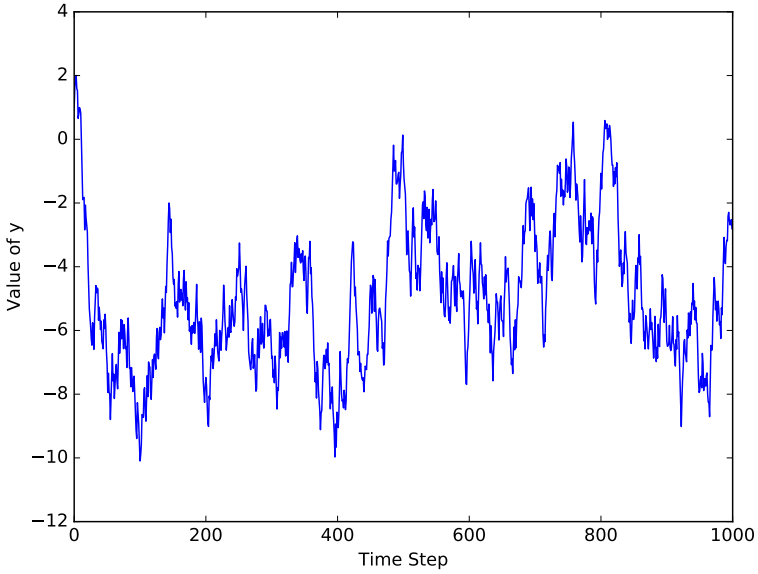


Figure 1.3: 1000 simulated time series observations

Randomness and Reproducibility

The `np.random.seed` method is used to make the random error ϵ_t repeatable. To see this try:

```
np.random.seed(2016)
print "Seed = 2016 ", np.random.rand(3)

np.random.seed(1990)
print "Seed = 1990 ", np.random.rand(3)

np.random.seed(2016)
print "Seed = 2016 ", np.random.rand(3)
print "not reset = ", np.random.rand(3)
```

You will see the output:

```
Seed = 2016   [ 0.89670536   0.73023921   0.78327576]
Seed = 1990   [ 0.72197726   0.07755006   0.09762252]
Seed = 2016   [ 0.89670536   0.73023921   0.78327576]
not reset =   [ 0.74165167   0.4620905    0.64256513]
```

If the seed is set to a specific value, the random sequence will be repeated exactly. For example, notice that when the seed is 2016 the first observation takes the value 0.89670536. This happens provided the seed is set before calling the `random.rand` method.

If the random seed is not reset, different numbers appear with every invocation. For example, on the first invocation using `Seed= 2016`, the second random value is 0.73023921. On the second invocation (`not reset`), the second random value is 0.4620905.

NOTE... ✍

Throughout this text we set the random generator seed to help with reproducibility of the results.

The Importance of Temporal Order

If you change the order of the observations in a time series, you lose the underlying dynamics of the data generating mechanism. Take for example, the series we generated using equation 1.1, suppose we reorder the data from smallest to largest:

```
print y.sort_values()
6    0.657570
7    0.791785
9    0.946840
8    0.996448
0    1.000000
5    1.534571
4    1.578358
1    1.629219
2    1.864309
3    2.004397
```

The first column reports the original time step for which the y value was observed. The values are now sorted by size, however the inherent structure captured by the time dynamics are lost, look at Figure 1.4.

Whether you are modeling the Dow-Jones Industrial Average or monthly Sunspot activity, changing the order of time series observations will result in the loss of the underlying dynamics you are most interested in capturing.

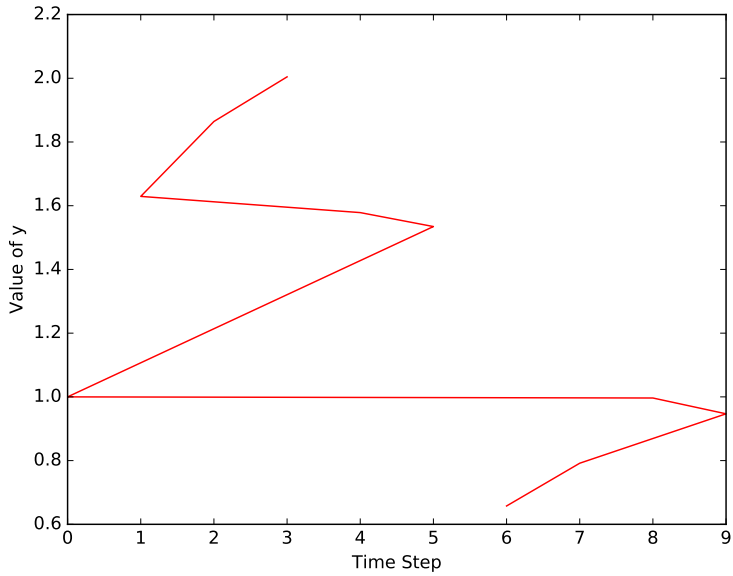


Figure 1.4: Ordered observations

The Ultimate Goal

If we fully understood the data generating mechanism, we could perfectly predict the sequence of future observations. Alas, in most cases, the data generating mechanism is unknown and we predict a new observation, say $\hat{y}_t$. Our goal is to develop forecasting models which minimize the error between our foretasted value $\hat{y}_t$ and the observed value y_t.

NOTE... ✍

Time-series data consists of sampled data points taken over time from a generally unknown process (see Figure 1.1). It has been called one of the top ten challenging problems in predictive analytics.

For Additional Exploration

The `pandas` and `numpy` packages are the workhorse of data analysis and modeling. The `pandas` library makes data manipulation and analysis a breeze. The `numpy` library provides numerous mathematical functions. It is worth spending time getting to know each package well:

- Visit the official Pandas site at http://pandas.pydata.org/
- Visit the official NumPy site at http://www.numpy.org/

To help you on your journey review the following articles:

- Shukla, Xitij U., and Dinesh J. Parmar. "Python–A comprehensive yet free programming language for statisticians." Journal of Statistics and Management Systems 19.2 (2016): 277-284.
- McKinney, Wes. "pandas: a foundational Python library for data analysis and statistics." Python for High Performance and Scientific Computing (2011): 1-9.
- Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation." Computing in Science & Engineering 13.2 (2011): 22-30.
- McKinney, Wes. "Data structures for statistical computing in python." Proceedings of the 9th Python in Science Conference. Vol. 445. 2010.
- Oliphant, Travis E. A guide to NumPy. Vol. 1. USA: Trelgol Publishing, 2006.

Chapter 2

Deep Neural Networks Explained

NEURAL networks can be used to help solve a wide variety of problems. This is because in principle, they can calculate any computable function. In practice, they are especially useful for problems which are tolerant of some error, have lots of historical or example data available, but to which hard and fast rules cannot easily be applied.

What is a Neural Network?

A Neural network is constructed from a number of interconnected nodes known as neurons, see Figure 2.1. These are usually arranged into layers. A typical feedforward neural network will have at a minimum an input layer, a hidden layer and an output layer. The input layer nodes correspond to the number of features or attributes you wish to feed into the neural network. These are akin to the covariates (independent variables) you would use in a linear regression model. The number of output nodes correspond to the number of items you wish to predict or classify. The hidden layer nodes are generally used to perform non-linear transformation on the original input at-

tributes.

In their simplest form, feed-forward neural networks propagate attribute information through the network to make a prediction, whose output is either continuous for regression or discrete for classification.

Figure 2.1 illustrates a typical feed forward neural network topology used to predict the age of a child. It has 2 input nodes, 1 hidden layer with 3 nodes, and 1 output node. The input nodes feed the attributes (Height, Weight) into the network. There is one input node for each attribute. The information is fed forward to the hidden layer. In each node a mathematical computation is performed which is then fed to the output node. The output nodes calculate a weighted sum on this data to predict child age. It is called a feed forward neural network because the information flows forward through the network.

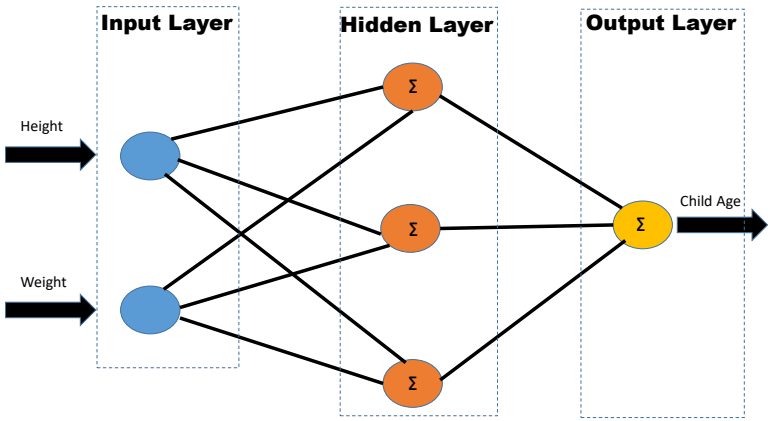


Figure 2.1: A basic neural network

The Role of Neuron

At the heart of an artificial neural network is a mathematical node, unit or neuron. It is the basic processing element. The

input layer neurons receive incoming information which they process via a mathematical function and then distribute to the hidden layer neurons. This information is processed by the hidden layer neurons and passed onto the output layer neurons.

Figure 2.2 illustrates a biological and artificial neuron. The key here is that information is processed via an activation function. Each activation function emulates brain neurons in that they are fired or not depending on the strength of the input signal. The result of this processing is then weighted and distributed to the neurons in the next layer. In essence, neurons activate each other via weighted sums. This ensures the strength of the connection between two neurons is sized according to the weight of the processed information.

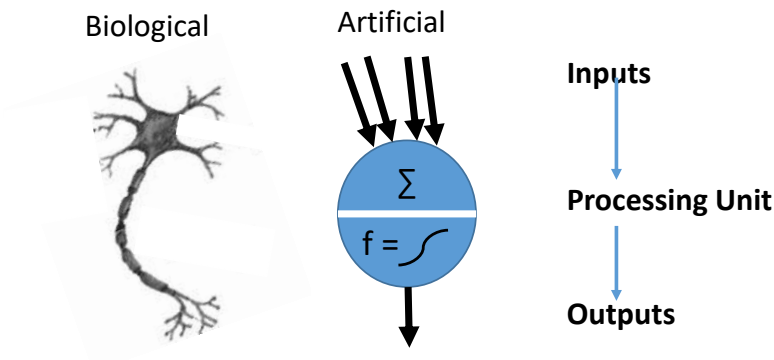


Figure 2.2: Biological and artificial neuron

Deep Learning in a Nutshell

There numerous types of deep learning model. In this text we focus on deep neural networks constructed from multiply hidden layers often called backpropagation neural networks. Historically, the foundation of deep learning is mostly a way of using backpropagation with gradient descent and a large number of nodes and hidden layers. Indeed, such back propagation neural networks were the first deep learning approach

to demonstrate widespread generality.

A deep neural network (DNN) consists of an input layer, an output layer, and a number of hidden layers sandwiched in between. As shown in Figure 2.3, the hidden layers are connected to the input layers and they combine and weight the input values to produce a new real valued number, which is then passed to the output layer. The output layer makes a classification or prediction decision using the values computed in the hidden layers.

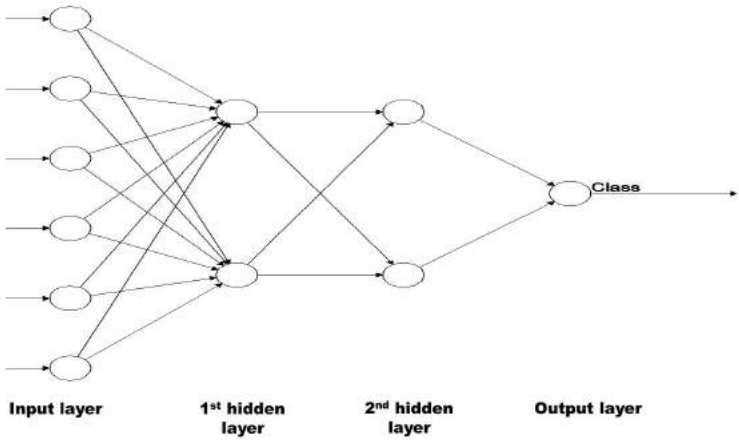


Figure 2.3: Feed forward neural network with 2 hidden layers

Deep multi-layer neural networks contain many levels of nonlinearities which allow them to compactly represent highly non-linear and/ or highly-varying functions. They are good at identifying complex patterns in data and have been set work to improve things like computer vision and natural language processing, and to solve unstructured data challenges.

As with a single layer neural network, during learning the weights of the connections between the layers are updated in order to make the output value as close as possible to the target output.

Generating Data for use with a Deep Neural Network

Let's build a DNN to approximate a function right now using Python. We will build a DNN to approximate $y = x^2$. First, let's import some basic packages. This is achieved using the `import` keyword:

```
import numpy as np
import pandas as pd
import random
```

Let's look at the code.

- The `import` keyword is used to import three packages - `numpy`, `pandas` and `random`.
- The NumPy (`numpy`) package is a core package used for scientific computing with Python. We will use it frequently throughout this text. Notice we import the package and use the shorthand `np` to access its elements later in the code. We do the same thing for the pandas package, referring to it with the shorthand `pd`.
- The `pandas` package is a library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. It is another core package used in data science.
- We will use the `random` module to generate random numbers.

Next, we generate 50 values of x randomly and then set $y = x^2$:

```
random.seed(2016)
sample_size=50
sample = pd.Series(random.sample(range
   (-10000,10000), sample_size))
x=sample/10000
y=x**2
```

One of the nice things about Python is that it is easy to read. The above code is a good example of this. Let's take a detailed look:

- The `random.seed` function is used to ensure you can reproduce the exact random sample used in the text. This is achieved by specifying a seed value, in this case 2016.
- The `random.sample` function is used to generate a random sample of 50 observations, with the result stored as a data type known as a Series. This is achieved via the argument `pd.Series()`. Notice we use the shorthand `pd` in place of `pandas`. A Series is a one-dimensional labeled array. It can contain any data type (integers, strings, floating point numbers, Python objects, etc.).
- The random sample is then scaled to lie between ± 1, and stored in the Python object `x`. Finally, the `y` object is used to store the squared values of `x`.

NOTE... ✍

You can see the "type" of a Python object by using the `type` argument. For example, to see the type of `sample`:

```
type(sample)
<class'pandas.core.series.Series'>
```

It is a `pandas` core type known as `Series`.

Exploring the Sample Data

It is a good idea to look at your data now and then, so let's take a peek at the first ten observations in `x`:

```
print x.head(10)

0     0.4758
1    -0.1026
2     0.7847
3     0.7506
4    -0.4870
5     0.3657
6     0.8647
7    -0.8361
8    -0.4067
9    -0.8568
dtype: float64
```

The first observations take a value of 0.4758, and the tenth observation a value of -0.8568. Notice the output also reports the `numpy dtype`, in this case `float64`. A `numpy` array contains elements described by a `dtype` object.

Now, look at `y`:

```
print y.head(10)

0     0.226386
1     0.010527
2     0.615754
3     0.563400
4     0.237169
5     0.133736
6     0.747706
7     0.699063
8     0.165405
9     0.734106
```

The first observation takes a value of 0.226386 and the tenth observation a value of 0.734106. A quick spot check reveals that for the first observation $(0.4758)^2 = 0.226386$, and for the tenth observation $(-0.8568)^2 = 0.734106$; so the numbers are as expected with `y` equal to the square of `x`. A visual plot of

the simulated data is shown in Figure 2.4.

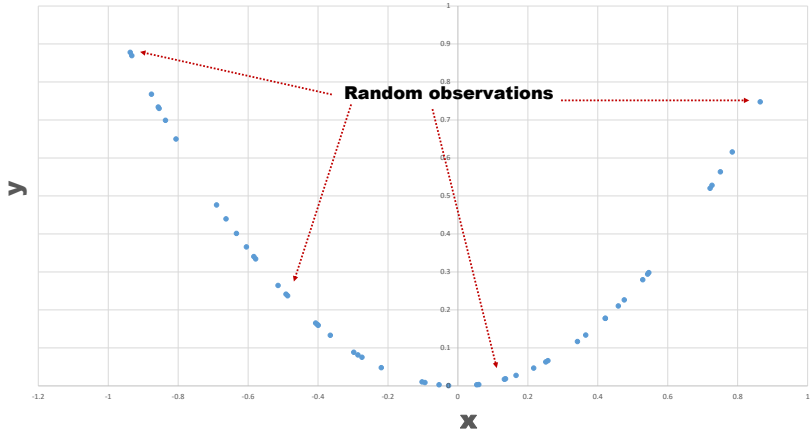


Figure 2.4: Plot of $y = x^2$ simulated data

You can also use the `describe()` method to provide a quick summary of your data. Let's use this to examine `x`:

```
print x.describe()

count    50.000000
mean     -0.100928
std       0.534392
min      -0.937000
25%      -0.562275
50%      -0.098400
75%       0.359750
max       0.864700
```

The `describe()` method provides details on the maximum, minimum, number of observations and the quartiles of the sample. As expected x lies between ± 1, with a maximum value of 0.8647 and a minimum value of -0.937.

Translating Sample Data into a Suitable Format

For this example, will use the `neuralpy` package. It is a machine learning library specializing in neural networks. Its intuitive interface makes it easy for you to quickly start training data and testing deep learning models.

In order to use this package, we need to put the data in a suitable format. To do this we will create a Python object called `dataSet` which will contain a list of the `x` and `y` observations. An easy way to do this is via a `while` loop:

```
count = 0
dataSet = [([x.ix[count]],[y.ix[count]])]
count=1
while (count < sample_size):
        print "Working on data item: ",
           count
        dataSet = (dataSet+[([x.ix[count
           ,0]],
        [y.ix[count]])])
        count = count + 1
```

Here is a quick breakdown of the above code:

- The `count` object is used to step through the loop, and is incremented after each new `x,y` example is added to `dataSet`.
- The `.ix` argument is used for indexing purposes and allows us to step through each observation in `x` and `y` and add it to `dataSet`.

While the model is running you should see something like this:

```
Working on data item:  1
Working on data item:  2
Working on data item:  3
.       .       .
```

```
Working on data item:  47
Working on data item:  48
Working on data item:  49
```

NOTE... ✍

Notice that `dataSet` is a list object:

```
type(dataSet)
<type 'list'>
```

This is the required type for the `neuralpy` package.

A Super Easy Deep Neural Network Python Tool

We will fit a DNN with two hidden layers. The first hidden layer will contain three neurons and the second hidden layer will have seven neurons, see Figure 2.5. This can be specified using the `neuralpy.Network` function:

```
import neuralpy
fit = neuralpy.Network(1, 3,7,1)
```

The maximum number of times the model iterates through the learning algorithm is controlled by the `epochs` parameter, we set this value to 100. The learning rate controls the size of the step in the gradient descent algorithm (see page 58), it is set equal to 1:

```
epochs = 100
learning_rate = 1
```

Now we are ready to fit the model using the `fit.train` function:

```
print "fitting model right now"
fit.train(dataSet, epochs, learning_rate)
```

Notice we pass `fit.train` the sample followed by the maximum number of iterations, then the learning rate. The model will take a short while to converge to a solution.

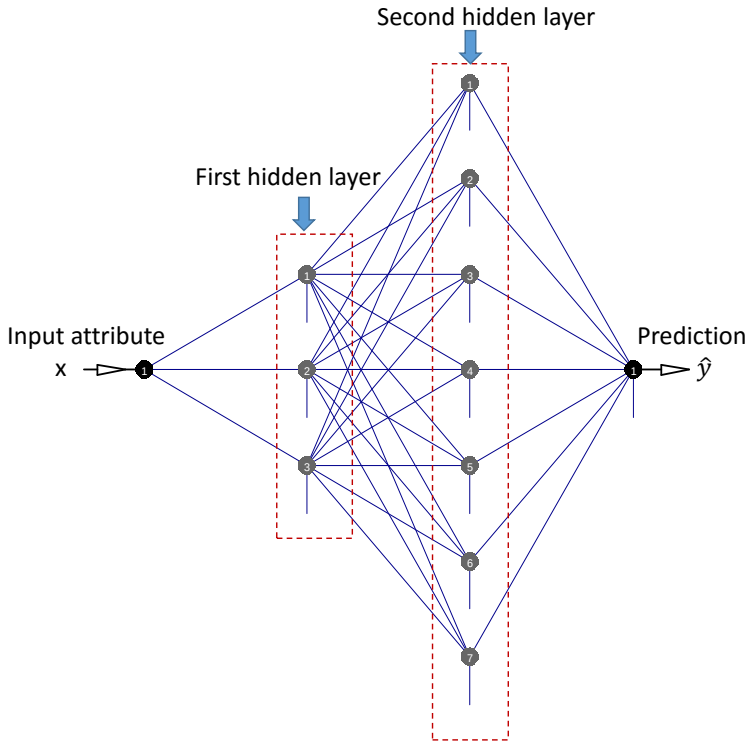


Figure 2.5: Structure of `neuralpy` neural network model.

NOTE... ✍

The `neuralpy` package only works with Python 2.7. Check your version of Python via:

```
import sys
print(sys.version)
```

Assessing Model Performance

Let's see how good this model is at approximating the values in `y`. The predicted values can be obtained using the `fit.forward` function. We will use a simple `while` loop to store the predictions in the Python object `out`, and a `print` function to display the results to the screen:

```
count = 0
pred=[]
while (count < sample_size):
    out=fit.forward(x[count])
    print ("Obs: ",count+1,
        " y = ",round(y[count],4),
        " prediction = ",
        round(pd.Series(out),4))
    pred.append(out)
    count = count + 1
```

As the model is running it will print the results to the screen. For the first five observations you should see something like the following:

```
('Obs: ', 1, ' y = ', 0.2264, ' prediction = ', 0.2283)
('Obs: ', 2, ' y = ', 0.0105, ' prediction = ', 0.0539)
('Obs: ', 3, ' y = ', 0.6158, ' prediction = ', 0.6182)
('Obs: ', 4, ' y = ', 0.5634, ' prediction = ', 0.5818)
('Obs: ', 5, ' y = ', 0.2372, ' prediction = ', 0.1999)
```

Of course, the numbers displayed to your screen will differ from those shown above, why? Because `neuralpy` sets the weights and biases at random. For each run of the model these values are different. Nevertheless, your overall results, will likely be close to the actual observations.

The reported numbers indicate the DNN provides a good, although not exact, approximation of the actual function. The predicted y and `x` values for all observations are shown in Figure 2.6. Judge for yourself, what do you think of the DNN models accuracy? How similar were your results to those shown? How might you improve the overall performance?

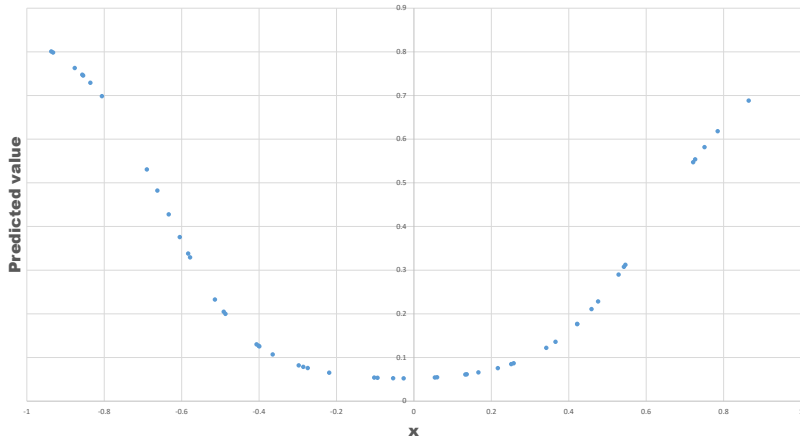


Figure 2.6: Predicted values and `x` using `neuralpy`

Additional Resources to Check Out

There is no shortage of Python neural network libraries. The `neuralpy` package has been around for a while. We use it to start with because it is easy to use, and allows you to create neural networks quickly and with very little knowledge. Additional documentation can be found at http://pythonhosted.org/neuralpy/ and https://jon--lee.github.io/neuralpy/.

Be sure to look at the following articles:

- Lin, Henry W., and Max Tegmark. "Why does deep and cheap learning work so well?." arXiv preprint arXiv:1608.08225 (2016).
- Schmidhuber, Jürgen. "Deep learning in neural networks: An overview." Neural Networks 61 (2015): 85-117.

Chapter 3

Deep Neural Networks for Time Series Forecasting the Easy Way

Now that you have a basic understanding of what a deep neural network is. Let's build a model to predict a real world time series. Through the process, we will dig a little deeper into the workings of neural networks, gain practical experience in their use, and develop a forecasting model that has considerable value for the residents and government officials of the Republic of Singapore.

Getting the Data from the Internet

If you have lived in Singapore, you will know that anyone who wants to register a new vehicle must first obtain a Certificate of Entitlement (COE). It gives the holder the right to vehicle ownership and access to the very limited road space in the tropical island city-state. The number of COEs issued is limited and they can only be obtained through an open bidding system. Figure 3.1 displays the historical time series of the COE price.

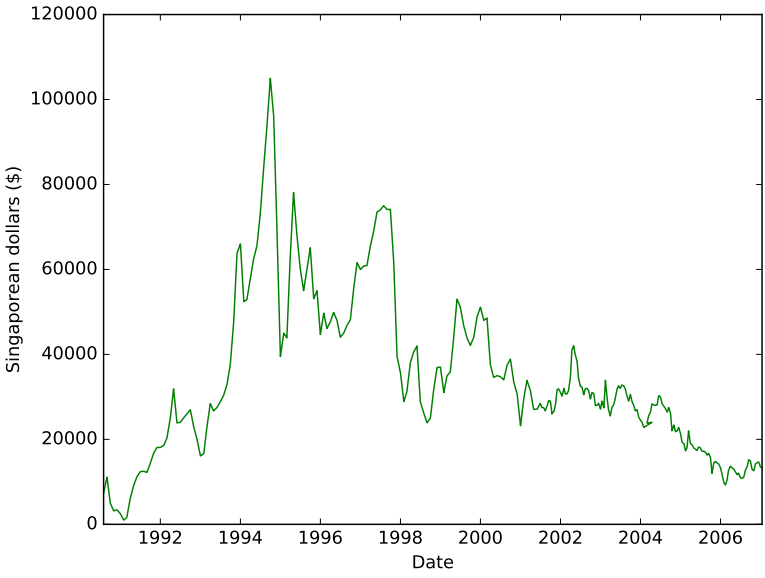


Figure 3.1: Certificate of Entitlement Price

A historical sample of the COE data can be downloaded from the internet:

```
import numpy as np
import pandas as pd
import urllib
url="https://goo.gl/WymYzd"
loc= "C:\\Data\\COE.xls"
urllib.urlretrieve(url, loc)
```

The first three lines import the required Python libraries. The `urllib` module fetches data from the World Wide Web; and the object `url` contains the location of the historical data (as a Google URL Shortener address). You could also specify the full website address using:

```
url="http://ww2.amstat.org/publications/jse
   /datasets/COE.xls"
```

The object `loc` contains the location to store the downloaded file. You will need to set this to your preferred location. The downloaded file is called `COE.XLS` and is stored on the `C:` drive in a directory called `Data`.

Cleaning up Downloaded Spreadsheet Files

The downloaded file has the XLS extension signifying the Microsoft Excel file format. It can be processed using the `pandas ExcelFile` method:

```
Excel_file = pd.ExcelFile(loc)
```

The code imports the excel file into the Python object `Excel_file`.

> ***NOTE...*** ✍
>
> You may need to install the package `xlrd`. You may be able to do this via `pip` i.e using something like:
>
> ```
> pip install xlrd
> ```

Worksheet Names

To process the data further we need to know the worksheet names associated with the spreadsheet file:

```
print Excel_file.sheet_names
[u'COE data']
```

The spreadsheet contains one sheet called `COE data`. Let's load the data into a Dataframe and view a summary of the information it contains:

```
spreadsheet = Excel_file.parse('COE data')

print spreadsheet.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 265 entries, 0 to 264
Data columns (total 6 columns):
DATE       265 non-null datetime64[ns]
COE$       265 non-null float64
COE$_1     265 non-null float64
#Bids      265 non-null int64
Quota      265 non-null int64
Open?      265 non-null int64
```

We see the spreadsheet contains six columns with 265 rows (entries). The first contains the date of the COE auction, and the remaining columns the other five variables.

The variable `COE$` contains the Certificate of Entitlement historical price. We will use it as our target variable and store it in an object called `data`:

```
data=spreadsheet['COE$']
print data.head()
```

View `data` Values

Look at the first few values of the object `data` using the `head` method:

```
print data.head()

0     7400.0
1    11100.0
2     5002.0
3     3170.0
4     3410.0
```

It appears that the first COE price (back in August 1990) was $7,400 (row 0), and the second price considerably higher at $11,100. The individuals who brought at this price got a rough deal, because the price dropped to around $5,000 and then less than $3,200 over the next few months. I'd be "kicking myself" if I'd paid $11,000 for something that trades at more than a 50% discount a month later. My guess is those who brought at $11,000 were probably too rich to care!

Adjusting Dates

Take another look at some of the dates in the spreadsheet:

```
print (spreadsheet['DATE'][193:204])
```

```
193    2004-02-01
194    2002-02-15
195    2004-03-01
196    2004-03-15
197    2004-04-01
198    2002-04-15
199    2004-05-01
200    2004-05-15
201    2004-06-01
202    2002-06-15
203    2004-07-01
```

Did you notice the error for 15th February, 15th April and 15th June? All are reported as the year 2002; they should say 2004. This quite is a common type of coding error. It won't directly impact our immediate analysis, but let's correct it anyway:

```
spreadsheet.set_value(194, 'DATE', '2004-02-15')
spreadsheet.set_value(198, 'DATE', '2004-04-15')
spreadsheet.set_value(202, 'DATE', '2004-06-15')
```

As a check, print out the dates again:

```
print (spreadsheet['DATE'][193:204])
```

```
193   2004-02-01
194   2004-02-15
195   2004-03-01
196   2004-03-15
197   2004-04-01
198   2004-04-15
199   2004-05-01
200   2004-05-15
201   2004-06-01
202   2004-06-15
203   2004-07-01
Name: DATE, dtype: datetime64[ns]
```

Saving Data

Let's save the file for later use. In this case as a comma-separated values (CSV) file. This is straightforward with the `to_csv` method. First, determine where you want to store the data. In this example, we use the object `loc` to store the location - on the `C` drive in the directory called `Data`. Second, call the `to_csv` method:

```
loc= "C:\\Data\\COE.csv"
spreadsheet.to_csv(loc)
```

Understanding Activation Functions

Each neuron contains an activation function (see Figure 3.2) and a threshold value. The threshold value is the minimum value that an input must have to activate the neuron. The activation function is applied and the output passed to the next neuron(s) in the network.

An activation function is designed to limit the output of the neuron, usually to values between 0 to 1, or -1 to +1. In most cases the same activation function is used for every neuron in a

network. Almost any nonlinear function does the job, although for the stochastic gradient descent algorithm (see page 59) it must be differentiable and it helps if the function is bounded.

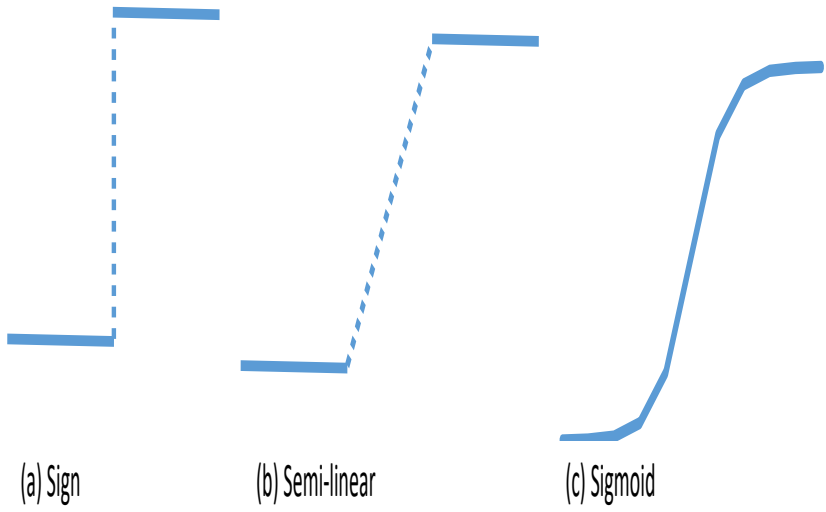


Figure 3.2: Three activation functions

NOTE... ✍

Activation functions for the hidden layer nodes are needed to introduce non linearity into the network.

The Fundamental Task of the Neuron

The task of the neuron is to perform a weighted sum of input signals and apply an activation function before passing the output to the next layer. So, we see that the input layer passes the data to the first hidden layer. The hidden layer neurons perform the summation on the information passed to them from

the input layer neurons; and the output layer neurons perform the summation on the weighted information passed to them from the hidden layer neurons.

Sigmoid Activation Function

The sigmoid (or logistic) function is a popular choice. It is an "S" shaped differentiable activation function. It is shown in Figure 3.3 where parameter c is a constant taking the value 1.5.

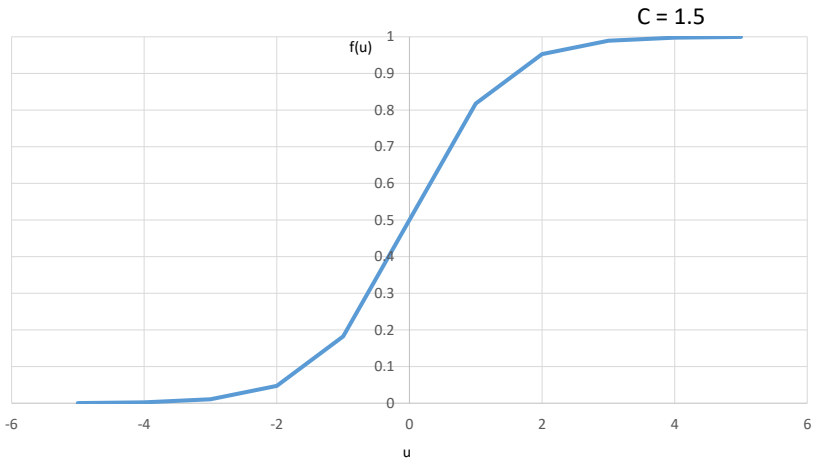


Figure 3.3: The sigmoid function with c = 1.5

The sigmoid function takes a real-valued number and "squashes" it into a range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. It is given by:

$$f(u) = \frac{1}{1 + exp(-cu)}$$

It gained popularity partly because the output of the function can be interpreted as the probability of the artificial neuron "firing".

Computational Cost

The sigmoid function is popular with basic neural networks because it can be easily differentiated and therefore reduces the computational cost during training. It turns out that:

$$\frac{\partial f(u)}{\partial u} = f(u)\,(1 - f(u))$$

So we see that the derivative $\frac{\partial f(u)}{\partial u}$ is simply the logistic function $f(u)$ multiplied by 1 minus $f(u)$. This derivative is used to learn the weight vectors via an algorithm known as stochastic gradient descent. The key thing to know is that because of this property the sigmoid function turns out to be very convenient for calculation of the gradients used in neural network learning.

NOTE... ✍

In order to carry out gradient descent (see page 58) the activation function needs to be differentiable.

How to Scale the Input attributes

Deep neural networks are sensitive to the scale of the input data, especially when the sigmoid (see page 38) or tanh (see page 44) activation functions are used. It is good practice to re-scale the data to the range -1 to +1 or 0 to 1. This can be achieved via the `MinMaxScaler` method in the `sklearn` module:

```
x=data
from sklearn import preprocessing
scaler = preprocessing.MinMaxScaler(
   feature_range=(0, 1))
```

The first line transfers the price data to our attribute variable `x`, then the preprocessing module is loaded and `MinMaxScaler`

used with the `feature_range` set to scale the attributes to the 0 to 1 range. The object `scaler`, contains the instructions required to scale the data. To see this type:

```
print scaler
MinMaxScaler(copy=True, feature_range=(0,
   1))
```

This informs us that `scaler` will scale the data into the 0,1 range. Note that `copy=True` is the default setting. If the input is already a numpy array you can set it to `False` to perform in place row normalization and avoid a copy. However, we set it to `True` because `x` (and `data`) are Pandas Series:

```
print type(x)
<class 'pandas.core.series.Series'>
```

Pandas is great for data manipulation, but for our numerical analysis we need to use a numpy array.

Here is how to transform `x` into a numpy array (format accepted by the deep neural network model, see page 45):

```
x = np.array(x).reshape((len(x), ))
```

Now, check the type of x:

```
print type(x)
<type 'numpy.ndarray'>
```

Whoopee! It is now a `numpy ndarray`.

Log Transform

When working with time series that always takes a positive value, I usually like to log transform the data. I guess this comes from my days building econometric models where applying a log transformation helps normalize the data. Anyway, here is how to do that:

```
x=np.log(x)
```

A Note on Data Shape

Sample data is stored as an array. The function `shape` returns the shape of an array. For example, to see the shape of `x`:

```
print x.shape
(265,)
```

This informs us that `x` is a 1 dimensional array with 265 rows (examples).

To scale x to lie in the [0,1] range, we pass it to the `fit_transform` function with the instructions contained in `scaler`. However, we need to pass it as a two dimensional array. The `reshape` function is an easy way to achieve this:

```
x=x.reshape(-1,1)
print x.shape
(265, 1)
```

The shape is now described by a pair of integers where the numbers denote the lengths of the corresponding array dimension.

Scale `x`

We are ready to scale the data in `x`. Here is how to do this in one line:

```
x = scaler.fit_transform(x)
```

The function `fit_transform` scales the values to the 0-1 range. Now, we convert the scaled value of `x` back to its original shape:

```
x=x.reshape(-1)
print x.shape
(265,)
```

We better check the data are scaled appropriately. A simple way to do this is to look at the maximum and minimum values of `x`:

```
print(round(x.min(),2))
0.0

print(round(x.max(),2))
1.0
```

Yep, all looks good.

Assessing Partial Autocorrelation

Now that the data is appropriately scaled, we need to determine how many past observations to include in our model. A simple way to do this is to use partial autocorrelations. The partial autocorrelation function (PACF) measures directly how an observation is correlated with an observation n time steps apart.

A partial autocorrelation is the amount of correlation between an observation x_t and a lag of itself (say x_{t-k}) that is not explained by correlations of the observations in between. For example, if x_t is a time-series observation measured at time t, then the partial correlation between x_t and x_{t-3} is the amount of correlation between x_t and x_{t-3} that is not explained by their common correlations with x_{t-1} and x_{t-2}.

Working the `statsmodels` library

The `statsmodels` library contains the function `pacf`. Here is how to use it to calculate the partial autocorrelations for 5 lags of `x`:

```
from statsmodels.tsa.stattools import pacf
x_pacf=pacf(x, nlags=5, method='ols')
```

The object `x_pacf` contains the actual partial autocorrelations. Just like regular correlations they range between -1 to +1. Where +1 indicates a strong positive association, -1

indicates a strong negative association, and 0 signifies no association.

Use the print statement to look at the values:

```
print x_pacf
```

```
[ 1.          0.95969034 -0.27093837  0.22144024
 -0.04729577  0.07360662]
```

The first value is 1 because it represents the correlation of `x_t` with itself. The second partial correlation takes the value of approximately 0.96, and indicates that x_t and x_{t-1} are highly correlated. This might be expected, as it is reasonable to assume there is some relationship between today's COE price and the price yesterday. However, the third partial autocorrelation is moderately negative, and the remaining values are rather small.

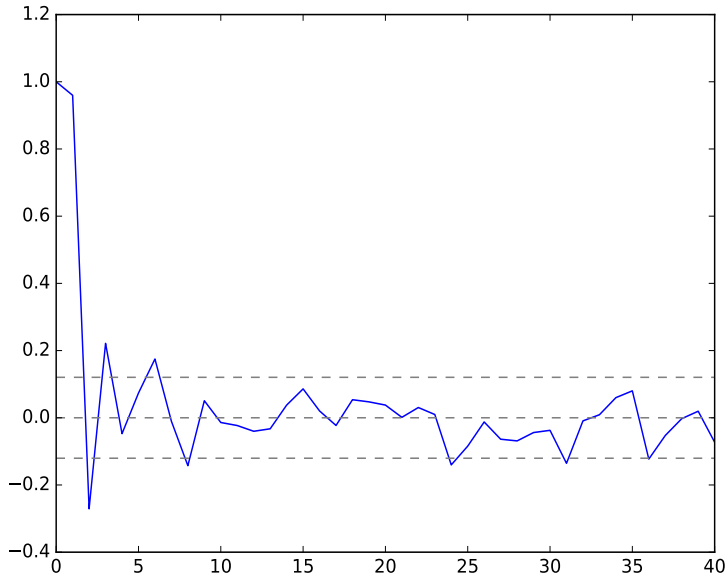


Figure 3.4: Partial autocorrelations

Figure 3.4 plots the partial autocorrelations for 40 lags alongside an approximate 95% statistical confidence interval (dotted gray line). It appears observations past one time lag have little association with the current COE price.

NOTE... ✍

The PACF measures the correlation between x_t and x_{t-k} by removing the intervening correlations.

Tanh Activation Function

The Tanh activation function is a popular alternative to the sigmoid function. It takes the form:

$$f(u) = \tanh(cu)$$

Like the sigmoid (logistic) activation function, the Tanh function is also sigmoidal ("s"-shaped), but produces output in the range -1 to $+1$. Since it is essentially a scaled sigmoid function, the Tanh function shares many of the same properties. However, because the output space is broader ([-1,+1] versus sigmoid range of [0,+1]), it is sometimes more effective for modeling complex nonlinear relations.

As shown in Figure 3.5, unlike the sigmoid function, the Tanh is symmetric around zero - only zero-valued inputs are mapped to near-zero outputs. In addition, strongly negative inputs will map to negative outputs. These properties make the network less likely to get "stuck" during training.

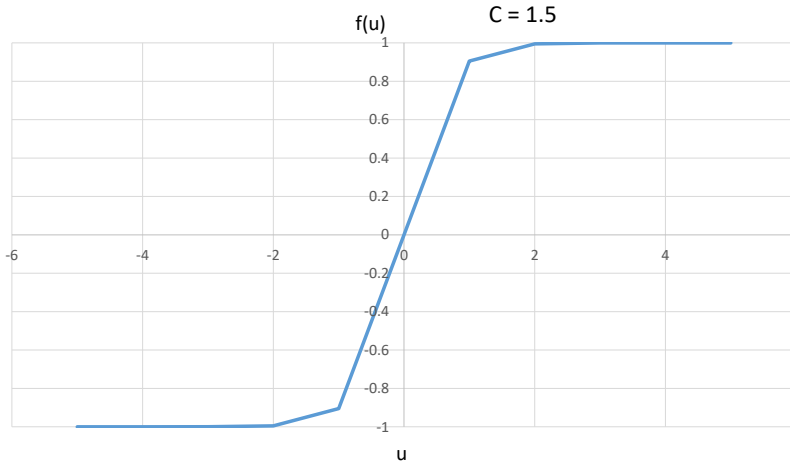


Figure 3.5: Hyperbolic Tangent Activation Function

Despite all the factors in favor of Tanh, in practice it is not a forgone conclusion that it will be a better choice than the sigmoid activation function. The best we can do is experiment.

A Neural Network Architecture for Time Series Forecasting

The `nnet-ts` library provides a neural network architecture designed explicitly for time series forecasting. The goal is to develop a model that can forecast the next COE price to within an accuracy of ±$1,500. We refer to this range as the prediction comfort interval (CI).

The `nnet-ts` package relies heavily on `numpy`, `scipy`, `pandas`, `theano` and `keras` libraries. Be sure to check their repositories and install them first. Then fetch the package from https://pypi.python.org/ using something along the lines of:

```
pip install nnet-ts
```

NOTE... ✍

The analysis in this section was carried out using `nnet-ts 1.0`.

Import `nnet_ts`

The model predicts next months COE price given this months COE price. This is called a one step ahead forecast. Let's calculate 12 one step ahead forecasts. A `while` loop will be needed to achieve this. But first, load the required package and set up the initial variables:

```
from nnet_ts import *
count = 0
ahead =12
pred=[]
```

The variable `count` will be used to iterate through the data. Since 12 one step ahead forecasts are to be calculated, the parameter `ahead` is set to equal 12. The variable `pred` will store each of the 12 one step ahead predictions.

NOTE... ✍

Python coders generally discourages the use of syntax such as `"import *"` because it can result in namespace collisions. Limit use to ad-hoc tests or situations where it is explicitly required by a package.

The `while` Loop

Here is `while` loop:

```
while (count < ahead):
```

```
end =len(x)-ahead+count
np.random.seed(2016)

fit1 = TimeSeriesNnet(hidden_layers =
    [7, 3],
activation_functions = ["tanh", "tanh"])

fit1.fit(x[0:end],lag = 1,epochs = 100)

out=fit1.predict_ahead(n_ahead = 1)

print ("Obs: ",count+1, " x = ",
round(x[count],4)," prediction = ",
round(pd.Series(out),4))

pred.append(out)

count = count + 1
```

Let's spend a few moments walking through this code.

- The model is called `fit1` and contains two hidden layers. The first hidden layer contains 7 nodes, and the second hidden layer contains 3 nodes. Both layers use the tanh activation function.
- On each iteration the model is fit to the data with a one time step lag via the `fit` function.
- This is followed by the `predict` method used to calculate the forecast. The parameter `n_ahead` is set to 1 to generate a 1 time step ahead forecast.
- Finally, the forecast at each iteration, stored in the object `out,` is appended to `pred`.

Realized and Predicted Values

As the code runs you will see the predicted values alongside the realized (normalized) observations:

```
('Obs: ', 1, ' x = ', 0.4303, ' prediction = ', 0.5147)
('Obs: ', 2, ' x = ', 0.5174, ' prediction = ', 0.5266)
('Obs: ', 3, ' x = ', 0.3462, ' prediction = ', 0.547)
('Obs: ', 4, ' x = ', 0.2482, ' prediction = ', 0.562)
('Obs: ', 5, ' x = ', 0.2639, ' prediction = ', 0.5859)
('Obs: ', 6, ' x = ', 0.1979, ' prediction = ', 0.5613)
('Obs: ', 7, ' x = ', 0.0, ' prediction = ', 0.5462)
('Obs: ', 8, ' x = ', 0.1064, ' prediction = ', 0.5525)
('Obs: ', 9, ' x = ', 0.3875, ' prediction = ', 0.5579)
('Obs: ', 10, ' x = ', 0.4726, ' prediction = ', 0.5832)
('Obs: ', 11, ' x = ', 0.5188, ' prediction = ', 0.5691)
('Obs: ', 12, ' x = ', 0.5406, ' prediction = ', 0.5575)
```

The predictions shown are the one time step ahead forecasts of the normalized COE price series stored in the Python object `x`. To convert them back to their original scale use the `inverse_transform` function and then take the exponent:

```
pred1 = scaler.inverse_transform(pred)
pred1=np.exp(pred1)
print np.round(pred1,1)
```

```
[[ 10964. ]
 [ 11585.5]
 [ 12741.5]
 [ 13661.9]
 [ 15267. ]
 [ 13617.9]
 [ 12694.4]
 [ 13073.2]
 [ 13401.3]
 [ 15080.6]
 [ 14124.5]
 [ 13377.2]]
```

Figure 3.6 plots the observed and forecast values alongside the tolerance range (dotted line). The actual observed values lie within the required CI; and the dynamics of the predictions capture some of the underlying dynamics of the original price series.

However, the match is not an exact fit and could, with some tweaking of the model, be improved. Nevertheless, you have constructed in a matter of moments a deep neural network

that adequately captures the dynamics of a rather complex time series of prices. With a little experimentation and further testing, you might even offer to sell the model to the COE seekers of Singapore!

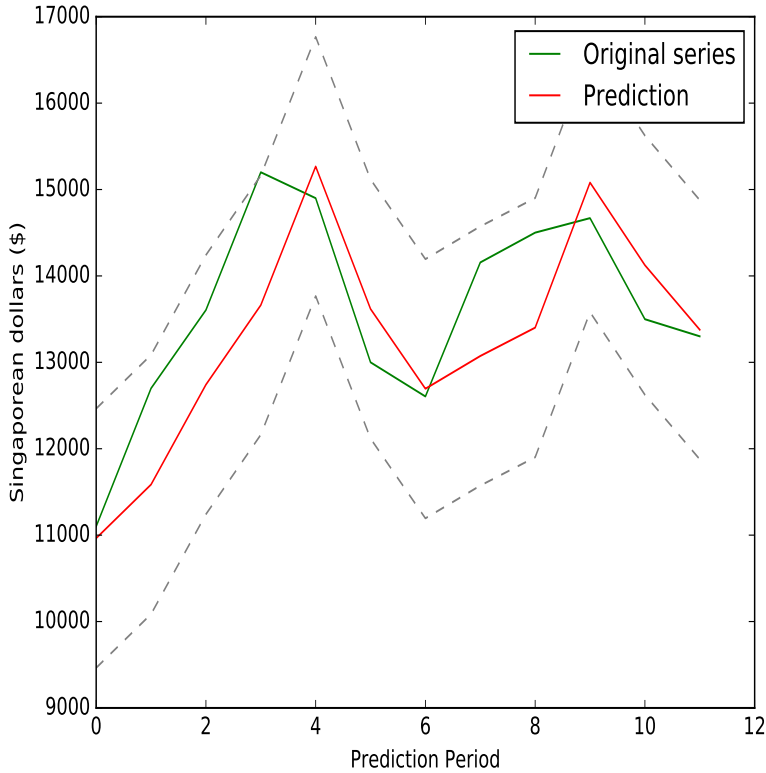


Figure 3.6: Observed and predicted values for COE

Additional Resources to Check Out

- Join a deep learning meetup group, they are springing up across the entire globe.

- Milton Boyd's article on designing neural networks for time series forecasting is a classic. See Kaastra, Iebeling, and Milton Boyd. "Designing a neural network for forecasting financial and economic time series." Neurocomputing 10.3 (1996): 215-236.

- Also, look at Frank, Ray J., Neil Davey, and Stephen P. Hunt. "Time series prediction and neural networks." Journal of intelligent and robotic systems 31.1-3 (2001): 91-103.

Chapter 4

A Simple Way to Incorporate Additional Attributes in Your Model

TRADDITIONAL time series analysis regresses the current observation on past values of itself. However, in many circumstances we may have additional explanatory attributes that are measured simultaneously. For example, an Economist might be interested in forecasting the rate of unemployment. Many potentially explanatory measures of economic activity are recorded at regular time intervals alongside the unemployment rate. Examples include interest rates, consumer price inflation, and measures of economic growth such as Gross Domestic Product. In this chapter, we'll look at how to include additional attributes into a deep neural network time series forecasting model.

Working with Additional Attributes

Let's continue with the COE data introduced in chapter 3. First, import the required libraries and open the saved csv file:

```
import numpy as np
```

```
import pandas as pd
loc= "C:\\Data\\COE.csv"
temp = pd.read_csv(loc)
```

The object temp contains the contents of the file. Let's look at the first few observations:

```
print temp.head()
```

	Unnamed: 0	DATE	COE$	COE$_1	#Bids	Quota	Open?
0	0	1990-08-01	7400.0	7750.0	656	472	0
1	1	1990-09-01	11100.0	7400.0	1462	468	0
2	2	1990-10-01	5002.0	11100.0	633	472	0
3	3	1990-11-01	3170.0	5002.0	952	511	0
4	4	1990-12-01	3410.0	3170.0	919	471	0

The data has an extra index column which will need to be removed. We don't need the date column either, so let's remove that too:

```
data= temp.drop( temp.columns [[0,1]] , axis =1)
print data.head()
```

	COE$	COE$_1	#Bids	Quota	Open?
0	7400.0	7750.0	656	472	0
1	11100.0	7400.0	1462	468	0
2	5002.0	11100.0	633	472	0
3	3170.0	5002.0	952	511	0
4	3410.0	3170.0	919	471	0

The variable `COE$_1` is the one month lagged COE price, `#Bids` is the number of resident bids for a COE, `Quota` records the available certificates. The variable `Open?` refers to whether an open-bid or closed-bid format was used.

NOTE... ✍

From August 1990 to June 2001, the COE auction used a closed-bid format. In this process bids were taken in secret and only disclosed at the end of the auction. From April 2003 onwards, the auction exclusively followed an open-bid format.

Data Processing

The target variable is `COE$` which we store in the object `y`. In addition, we will use the variables `COE$_1`, `#Bids, Quota` and `Open?` as our additional attribute set. These are stored in the object `x`:

```
y=data['COE$']
x=data.drop( data.columns [[0,4]] , axis =1)
x=x.apply(np.log)
x=pd.concat([x,data['Open?']], axis=1)
```

The second line removes `COE$` (column 0) and `Open?` (column 4) via the drop method; and stores `COE$_1, #Bids` and `Quota` in `x`. Since `Open?` is binary, it is added back to `x` via the `concat` method after the log transformation of the numerical variables.

Look at the transformed attributes:

```
print x.head()
```

	COE$_1	*#Bids*	*Quota*	*Open?*
0	8.955448	6.486161	6.156979	0
1	8.909235	7.287561	6.148468	0
2	9.314700	6.450470	6.156979	0
3	8.517593	6.858565	6.236370	0
4	8.061487	6.823286	6.154858	0

```
print x.tail()
```

	COE$_1	*#Bids*	*Quota*	*Open?*
260	9.441849	7.660114	7.215240	1
261	9.557894	7.591357	7.160069	1
262	9.581973	7.596392	7.156956	1
263	9.593492	7.443078	7.162397	1
264	9.510371	7.529943	7.173192	1

Scaling Attributes and Target

The next step is to scale the attributes into a suitable range for input into the deep neural network model:

```
from sklearn import preprocessing
scaler_x = preprocessing.MinMaxScaler(
    feature_range=(0, 1))
x = np.array(x).reshape((len(x),4 ))
x = scaler_x.fit_transform(x)
```

The attributes are scaled to lie in the 0 to 1 range. Notice that x is reshaped by number of attributes into a numpy `ndarray`. Hence the value "4" refers to the number of attributes.

A similar method is used to scale the target variable:

```
scaler_y = preprocessing.MinMaxScaler(
    feature_range=(0, 1))
y = np.array(y).reshape((len(y), 1))
y=np.log(y)
y = scaler_y.fit_transform(y)
```

The `pyneurgen` Module

The `pyneurgen` module provides libraries for use in Python programs to build neural networks and genetic algorithms and/or genetic programming. We will use this package to build our model. It requires the target and attributes be supplied as a list:

```
y=y.tolist()
x=x.tolist()
```

We now have our target y, and attributes x, in an appropriate format.

The Working of the Neuron Simplified

Figure 4.1 illustrate the workings of an individual neuron. Given a sample of input attributes $\{x_1,...,x_n\}$ a weight w_{ij} is associated with each connection into the neuron; and the neuron then sums all inputs according to:

$$f(u) = \sum_{i=1}^{n} w_{ij} x_j + b_j$$

The parameter b_j is known as the bias and is similar to the intercept in a linear regression model. It allows the network to shift the activation function "upwards" or "downwards". This type of flexibility is important for successful machine learning.

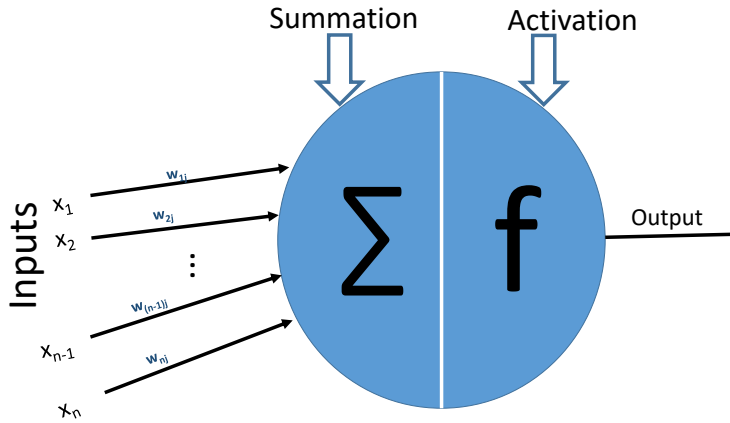


Figure 4.1: Artificial neuron

The size of a neural network is often measured by the number of parameters required to be estimated. The network in Figure 2.1 has $[2 \times 3] + [3 \times 1] = 9$ weights and $3 + 1 = 4$ biases, for a total of 13 learnable parameters. This is a lot of parameters relative to a traditional statistical model.

NOTE... ✍

In general, the more parameters, the more data is required to obtain reliable predictions.

How a Neural Network Learns

For any given set of input data and neural network weights, there is an associated magnitude of error, which is measured by the error function (also known as the cost function). This is our measure of "how well" the neural network performed with respect to its given training sample and the expected output. The goal is to find a set of weights that minimizes the mismatch between network outputs and actual target values.

The typical neural network learning algorithm applies error propagation from outputs to inputs, and gradually fine tunes the network weights to minimize the sum of error. The learning cycle is illustrated in Figure 4.2. Each cycle through this learning process is called an epoch.

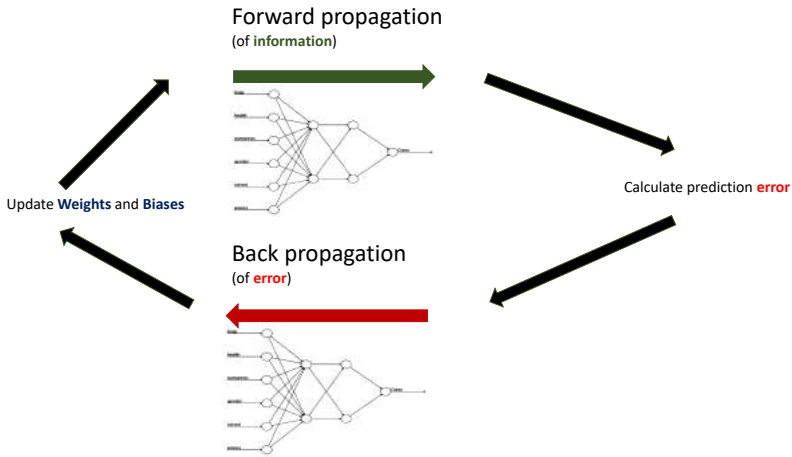


Figure 4.2: Neural network learning cycle

Backpropagation is the most widely used learning algorithm. It takes the output of the neural network and compares it to the desired value. Then it calculates how far the output was from the desired value. This is a measure of error. Next it calculates the error associated with each neuron from the preceding layer. This process is repeated until the input layer is

reached. Since the error is propagated backwards (from output to input attributes) through the network to adjust the weights and biases, the approach is known as backpropagation.

Step by Step Explanation

The basic neural network learning approach works by computing the error of the output of the neural network for a given sample, propagating the error backwards through the network while updating the weight vectors in an attempt to reduce the error. The algorithm consists of the following steps.

- **Step 1:- Initialization of the network:** The initial values of the weights need to be determined. A neural network is generally initialized with random weights.
- **Step 2:- Feed Forward:** Information is passed forward through the network from input to hidden and output layer via node activation functions and weights. The activation function is (usually) a sigmoidal (i.e., bounded above and below, but differentiable) function of a weighted sum of the nodes inputs.
- **Step 3:- Error assessment:** Assess whether the error is sufficiently small to satisfy requirements or whether the number of iterations has reached a predetermined limit. If either condition is met, then the training ends. Otherwise, the iterative learning process continues.
- **Step 4:- Propagate:** The error at the output layer is used to re-modify the weights. The algorithm propagates the error backwards through the network and computes the gradient of the change in error with respect to changes in the weight values.
- **Step 5:- Adjust**: Make adjustments to the weights using the gradients of change with the goal of reducing the error. The weights and biases of each neuron are adjusted

by a factor based on the derivative of the activation function, the differences between the network output and the actual target outcome and the neuron outputs. Through this process the network "learns".

Gradient Descent Clarified

Gradient descent is one of the most popular algorithms to perform optimization in a neural network. In general, we want to find the weights and biases with minimize the error function. Gradient descent updates the parameters iteratively to minimize the overall network error.

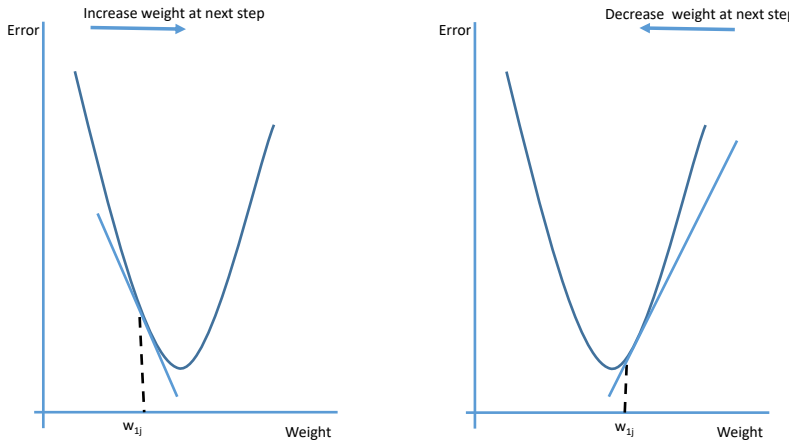


Figure 4.3: Basic idea of Stochastic Gradient minimization

It iteratively updates the weight parameters in the direction of the gradient of the loss function until a minimum is reached. In other words, we follow the direction of the slope of the loss function downhill until we reach a valley. The basic idea is roughly illustrated in Figure 4.3:

- If the partial derivative is negative, the weight is increased (left part of the figure);

- if the partial derivative is positive, the weight is decreased (right part of the figure). The parameter known as the learning rate (discussed further on the following page) determines the size of the steps taken to reach the minimum.

Stochastic Gradient Descent

In traditional gradient descent, you use the entire dataset to compute the gradient at each iteration. For large datasets, this leads to redundant computations because gradients for very similar examples are recomputed before each parameter update.

Stochastic Gradient Descent (SGD) is an approximation of the true gradient. At each iteration, it randomly selects a single example to update the parameters and moves in the direction of the gradient with respect to that example. It therefore follows a noisy gradient path to the minimum. Due in part to the lack of redundancy, it often converges to a solution much faster than traditional gradient descent.

One rather nice theoretical property of stochastic gradient descent is that it is guaranteed to find the global minimum if the loss function is convex. Provided the learning rate is decreased slowly during training, SGD has the same convergence behavior as traditional gradient descent.

NOTE... ✍

In probabilistic language SGD is almost certainly converges to a local or the global minimum for both convex and non- convex optimizations.

How to Easily Specify a Model

Load the appropriate libraries:

```
from pyneurgen.neuralnet import NeuralNet
```

Here is how to specify the neural network structure:

```
import random
random.seed(2016)
fit1 = NeuralNet()
fit1.init_layers(4, [7,3], 1)
fit1.randomize_network()
```

After instantiating the main class `NeuralNet` in `fit1`, the network layers are specified. The model contains 4 input attributes, 2 hidden layers, and 1 output layer. The first hidden layer contains 7 nodes, and the second hidden layer has 3 nodes.

NOTE... ✍

The `randomize_network` method randomizes the weights and bias of each connection.

Choosing a Learning Rate

Our next task is to specify the learning rate. It determines the size of the steps taken to reach the minimum by the gradient descent algorithm. Figure 4.4 illustrates the general situation. With a large learning rate, the network may learn very quickly, a lower learning rates takes longer to find the optimum value.

You may wonder why not set the learning rate to a high value? Well, as shown in Figure 4.5, if the learning rate is too high the network may miss the global minimum and not learn very well or even at all. Setting the learning rate involves an iterative tuning procedure in which we manually set the highest possible value.

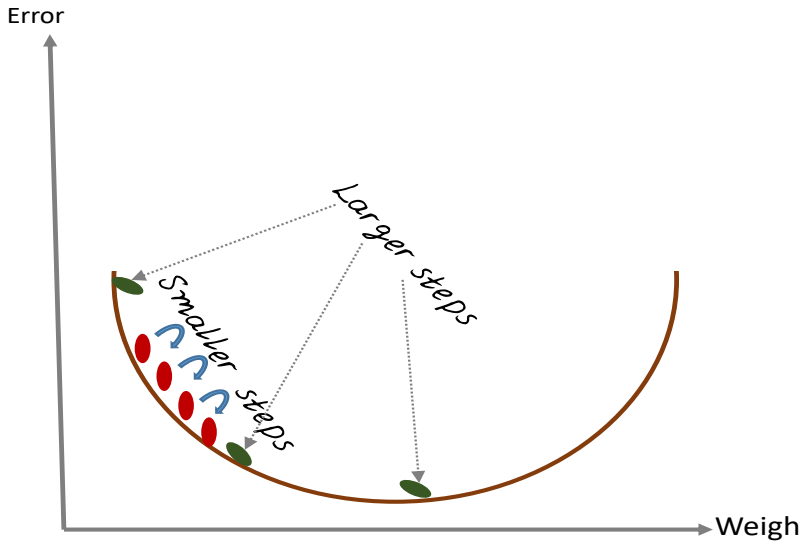
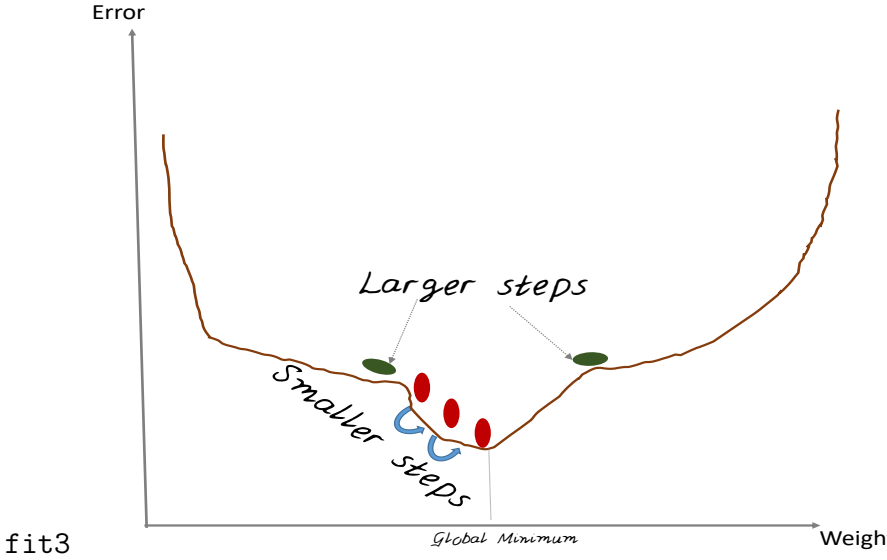


Figure 4.4: Optimization with small and large learning rates



fit3

Figure 4.5: Error surface with small and large learning rate

Setting the Learning Rate

A high learning rate can cause the system to diverge in terms of the objective function. Choosing this rate too low results in slow learning. Let's set the learning rate to a relatively low level of 0.05 and set up the training and test sets to be used by the model:

```
fit1.set_learnrate(0.05)
fit1.set_all_inputs(x)
fit1.set_all_targets(y)
length = len(x)
learn_end_point = int(length * 0.95)
fit1.set_learn_range(0, learn_end_point)
fit1.set_test_range(learn_end_point + 1,
    length-1)
```

In the above code, once the learning rate is specified, the attributes and response are passed to fit1. We use 95% of the observations for training the model, with the remainder used for the test set.

NOTE... ✍

The optimal learning rate is usually data and network architecture specific. However, you might initially try values in the range of 0.1 to 1e-6.

The Efficient Way to Run Your Model

`pyneurgen` allows you to specify activation functions for each layer. The tanh function is chosen for both layers:

```
fit1.layers[1].set_activation_type('tanh')
fit1.layers[2].set_activation_type('tanh')
```

NOTE... ✍

Experiment with alternative activation functions. In `pyneurgen` you can also use `'linear'` and `'sigmoid'`.

Run the Model

Now, we are ready to run the model. This is achieved via the `learn` function:

```
fit1.learn(epochs=200,
           show_epoch_results=True,
           random_testing=False)
```

The parameter `random_testing` is set to `False` to maintain the order of the examples. The model is run for 200 epochs, and because `show_epoch_results=True`, you should see the model performance for each epoch along the lines of:

```
epoch:  0 MSE:  0.00375778849486
epoch:  1 MSE:  0.00481815401934
epoch:  2 MSE:  0.00419000621781
epoch:  3 MSE:  0.00377844460056
epoch:  4 MSE:  0.00349184307804
epoch:  5 MSE:  0.00328394827976
epoch:  6 MSE:  0.00312792310141
```

NOTE... ✍

For this type of model you can also try `random_testing=True`. This will present the example pair (x,y) in random order to the model (see page 136 for another example).

Assess Performance

Performance is measured using the Mean Square Error (MSE). It is derived by summing the squared differences between the observed target (y) and predicted value ($\hat{y}$), and dividing by the number of test examples (n):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

The smaller the MSE, the closer the predictions match the observed target values. Ideally, we would like a MSE of zero for a perfect prediction for every example. In practice, it will nearly always be greater than zero.

Figure 4.6 shows for our model `fit1`, it declines rapidly for the first 50 epochs, and then exhibits a more gradual decline eventually leveling out by around 200 epochs.

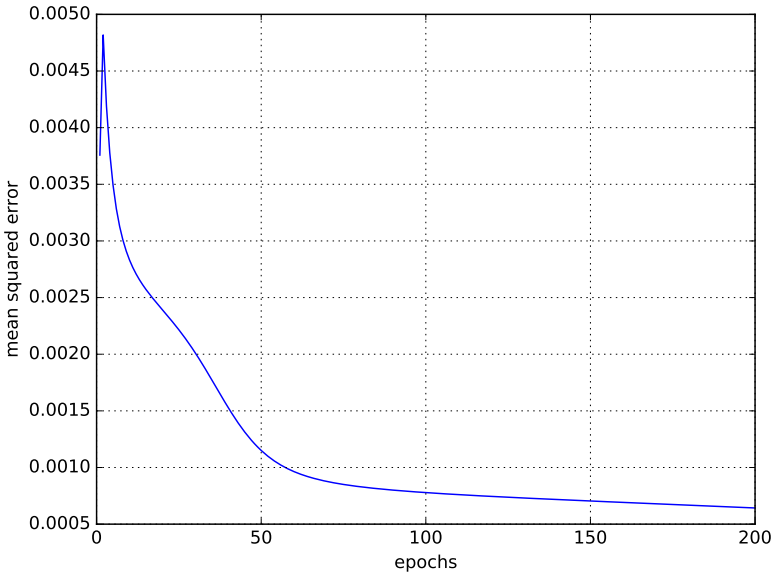


Figure 4.6: Train set MSE by epoch

The test set performance is obtained using the `test()` function:

```
mse = fit1.test()
print "test set MSE =",np.round(mse,6)
test set MSE = 0.000107
```

This value appears to be reasonably small, however, we should check it against our original criteria of accuracy within a ± $1,500 tolerance. Figure 4.7, plots the predicted and observed values to confirm this.

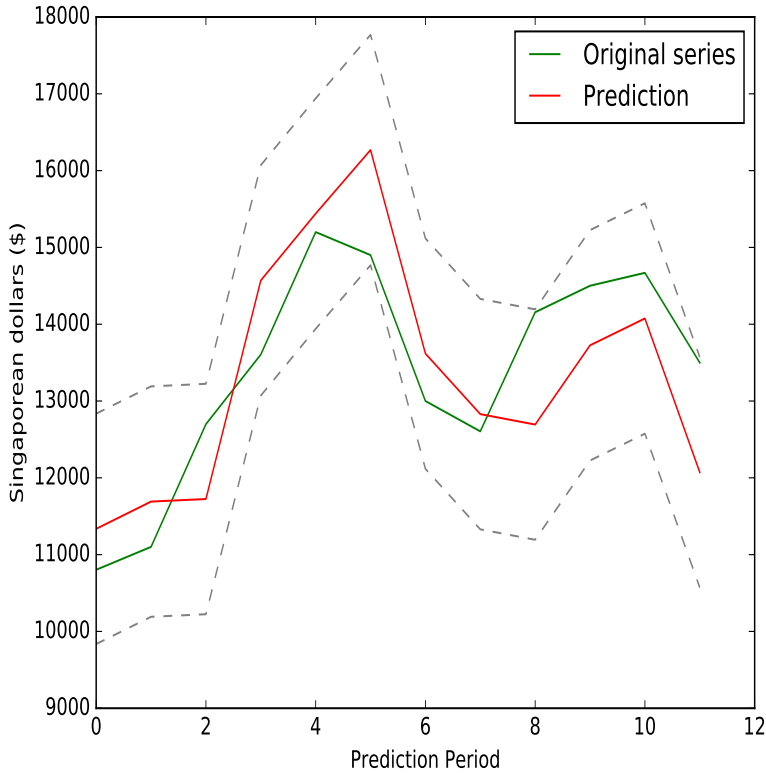


Figure 4.7: Predicted and actual values for COE model with attributes

NOTE... ✍

You can view the predictions and target value pairs via `test_targets_activations`. For example, try something along the lines of:

```
print np.round( fit1.
    test_targets_activations,4)
```

Additional Resources to Check Out

- Be sure to check out the pyneurgen documentation

Chapter 5

The Simple Recurrent Neural Network

Ever since I ran across my first Recurrent Neural Network (RNN), I have been intrigued by their ability to learn difficult problems that involve time series data. Unlike the deep neural network discussed in chapter 3 and chapter 4, RNNs contain hidden states which are distributed across time. This allows them to efficiently store a lot of information about the past. As with a regular deep neural network, the non-linear dynamics allows them to update their hidden state in complicated ways. They have been successfully used for many problems including adaptive control, system identification, and most famously in speech recognition. In this chapter, we use the Keras library to build a simple recurrent neural network to forecast COE price.

Why Use Keras?

Keras is a deep learning library for Theano and TensorFlow. Theano allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. TensorFlow is a library for numerical computation us-

ing graphs.

Theano and TensorFlow are two of the top numerical platforms in Python used for Deep Learning research. Whilst they are both very powerful libraries, they have a steep learning curve. Keras is a high-level neural networks library, capable of running on top of either TensorFlow or Theano. It allows easy and fast prototyping, was specifically developed with a focus on enabling fast experimentation, and runs on both a central processing unit and graphics processing unit.

Keras is straightforward to install. You should be able to use something along the lines of:

```
pip install keras
```

NOTE... ✍

You must have an installation of Theano or TensorFlow on your system before you install Keras.

What is a Recurrent Neural Network?

Neural networks can be classified into dynamic and static. Static neural networks calculate output directly from the input through feed forward connections. For example, in a basic feedforward neural network, the information flows in a single direction from input to output. Such neural networks have no feedback elements. The neural networks we developed in chapter 3 and chapter 4, are all examples of static neural networks.

In a dynamic neural network, the output depends on the current input to the network, and the previous inputs, outputs, and/or hidden states of the network. Recurrent neural networks are an example of a dynamic network.

Visual Representation

Two simple recurrent neural networks are illustrated in Figure 5.1. The idea is that the recurrent connections allow a memory of previous inputs to persist in the network's internal state, and thereby influence the network's output. At each time step, the network processes the input vector (x_t), updates its hidden state via an activation functions (h_t), and uses it to make a prediction of its output (y_t). Each node in the hidden layer receives the inputs from the previous layer and the outputs of the current layer from the last time step. The value held in the delay unit is fed back to the hidden units as additional inputs.

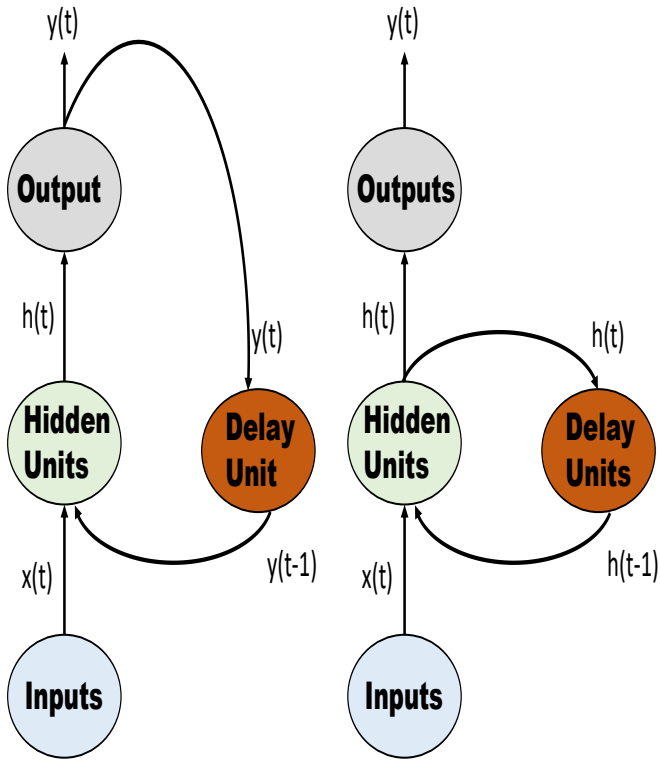


Figure 5.1: Two Simple Recurrent Neural Network Structures

It turns out that with enough neurons and time, RNNs can compute anything that can be computed by your computer. Computer Scientists refer to this as being Turing complete. Turing complete roughly means that in theory an RNN can be used to solve any computation problem. Alas, "in theory" often translates poorly into practice because we don't have infinite memory or time.

NOTE... ✍

I suppose we should really say the RNN can approximate Turing-completeness up to the limits of their available memory.

Prepare the Sample Data

Now that you have a broad overview, load the libraries and prepare the data required for our COE price forecasting model:

```
import numpy as np
import pandas as pd
loc= "C:\\Data\\COE.csv"
temp = pd.read_csv(loc)
data= temp.drop( temp.columns [[0,1]] , axis
    =1)
y=data["COE$"]
x=data.drop( data.columns [[0,4]] , axis =1)
x=x.apply(np.log)
x=pd.concat([x,data["Open?"]], axis=1)
from sklearn import preprocessing
scaler_x = preprocessing.MinMaxScaler(
    feature_range=(0, 1))
x = np.array(x).reshape((len(x),4 ))
x = scaler_x.fit_transform(x)
```

```
scaler_y = preprocessing.MinMaxScaler(
    feature_range=(0, 1))
y = np.array(y).reshape((len(y), 1))
y=np.log(y)
y = scaler_y.fit_transform(y)
```

The above code should be familiar to you by now (see chapter 4), so we won't discuss it in detail. However, here are the highlights:

- Be sure to change the value in `loc` to point to the directory where you stored `COE.csv`.
- In terms of data processing, the attributes, stored in `x`, are scaled to lie in the 0 to 1 range.
- The natural log of the target variable is stored in `y`, also scaled to lie in the 0 to 1 range.

Gain Clarity on the Role of the Delay Units

The delay unit enables the network to have short-term memory. This is because it stores the hidden layer activation values (or output) of the previous time step. It releases these values back into the network at the subsequent time step. In other words, the RNN has a "memory" which captures information about what has been calculated by the hidden units at an earlier time step.

Time-series data contain patterns ordered by time. Information about the underlying data generating mechanism is contained in these patterns. RNNs take advantage of this ordering because the delay units exhibit persistence. It is this "short term" memory feature that allows an RNN to learn and generalize across sequences of inputs.

NOTE... ✍

An RNN basically takes the output of each hidden or output layer, and feeds it back to itself (via the delay node) as an additional input. The delay loop allows information to be passed from one time step of the network to the next.

Follow this Approach to Create Your Train and Test Sets

For this illustration, the train set contains around 95% of the observations, with the remaining allocated to the test set:

```
end=len(x)-1
learn_end = int(end * 0.954)
x_train=x[0:learn_end-1,]
x_test=x[learn_end:end-1,]
y_train=y[1:learn_end]
y_test=y[learn_end+1:end]
x_train=x_train.reshape(x_train.shape + (1,))
x_test=x_test.reshape(x_test.shape + (1,))
```

The final two lines of the above code reshape the training and test set attributes into a suitable format for passing to the Keras library. The shape is of the form (`number of samples, number of features,time_steps`). For example, the train set contains 250 examples on four features for each time step:

```
print "Shape of x_train is ",x_train.shape
Shape of x_train is  (250, 4, 1)
```

And for the test set:

```
print "Shape of x_test is ",x_test.shape
Shape of x_test is  (12, 4, 1)
```

As expected, there are 12 examples in the test set on 4 features for each time step.

Parameter Sharing Clarified

From Figure 5.2 we see that a RNN is constructed of multiple copies of the same network, each passing a message to a successor. It therefore shares the same parameters across all time steps because it performs the same task at each step, just with different inputs. This greatly reduces the total number of parameters required to learn relative to a traditional deep neural network which uses a different set of weights and biases for each layer.

NOTE... ✍

The deep neural network discussed in chapter 3 and chapter 4 had no sense of time or the ability to retain memory of their previous state. This is not the case for a RNN.

Understand Backpropagation Through Time

It turns out that with a few tweaks, any feed-forward neural networks can be trained using the backpropagation algorithm. For an RNN, the tweaked algorithm is known as Backpropagation Through Time. The basic idea is to unfold the RNN through time. This sounds complicated, but the concept is quite simple. Look at Figure 5.2. It illustrates an unfolded version of Figure 5.1 (left network). By unfolding, we simply mean that we write out the network for the complete time steps under consideration.

Unfolding simple unrolls the recurrent loop over time to reveal a feedforward neural network. The unfolded RNN is essentially a deep neural network where each layer corresponds to a time step in the original RNN.

The resultant feed-forward network can be trained using the backpropagation algorithm. Computing the derivatives of error with respect to weights is reduced to computing the derivatives in each layer of a feed forward network. Once the network is trained, the feed-forward network "folds" itself obtaining the original RNN.

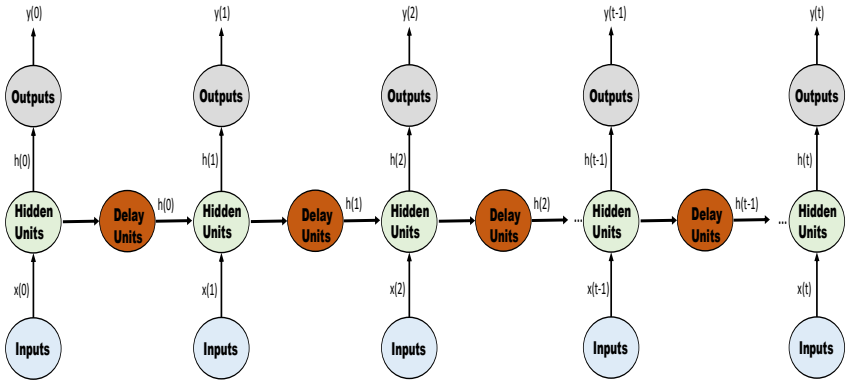


Figure 5.2: Unfolding a RNN

> ***NOTE...*** ✍
>
> The delay (also called the recurrent or context) weights in the RNN are the sum of the equivalent connection weights on each fold.

Import Keras Modules

The Keras library modules can be loaded as follows:

```
from keras.models import Sequential
```

```
from keras.optimizers import SGD
from keras.layers.core import Dense,
   Activation
from keras.layers.recurrent import
   SimpleRNN
```

Here is a quick overview:

- The first line imports the Keras Sequential model. It allows the linear stacking of layers.
- Optimization for our illustration is carried out using stochastic gradient decent (see page 59), therefore we import `SGD` from `keras.optimizers`.
- The third line imports the activation functions and a dense layer, this is a regular fully connected neural network layer. We use it for the output layer.
- The final line imports a fully-connected RNN where the output is to be fed back to input.

NOTE... ✍

The examples in this text were developed using Keras 1.0.4 with Theano 0.8.2 as the backend.

Determine Model Structure

The next step is to set up the model structure:

```
seed=2016
np.random.seed(seed)
fit1 = Sequential()
fit1.add(SimpleRNN(output_dim=8,activation
   ="tanh", input_shape=(4, 1)))
fit1.add(Dense(output_dim=1, activation='
   linear'))
```

Let's take a brief walk-through of this set up:

- The model is stored in fit1. Each layer is added via the `add` function. It consists of a RNN with 8 delay nodes with tanh activation functions.
- The `input_shape` argument takes the number of features (in this case 4) and the number of time steps (in this case 1).
- The output layer is a fully connected layer with 1 output (our COE forecast) via a linear activation function.

A Complete Intuitive Guide to Momentum

The iterative process of training a neural network continues until the error reaches a suitable minimum. We saw on page 60, that the size of steps taken at each iteration is controlled by the learning rate. Larger steps reduce training time, but increase the likelihood of getting trapped in local minima instead of the global minima.

Another technique that can help the network out of local minima is the use of a momentum term. It can take a value between 0 and 1. It adds this fraction of the previous weight update to the current one. As shown in Figure 5.3, a high value for the momentum parameter, say 0.9, can reduce training time and help the network avoid getting trapped in local minima.

However, setting the momentum parameter too high can increase the risk of overshooting the global minimum. This is further increased if you combine a high learning rate with a lot of momentum. However, if you set the momentum coefficient too low the model will lack sufficient energy to jump over local minima.

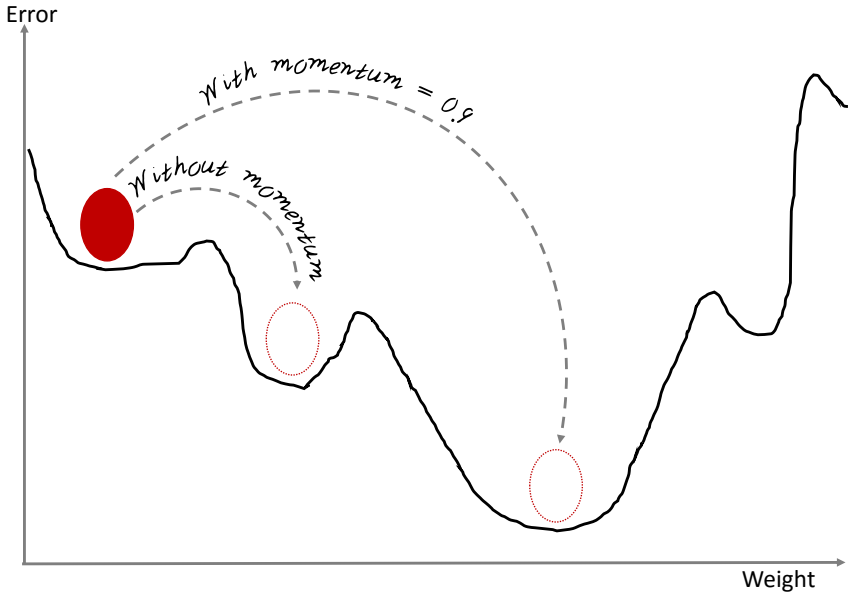


Figure 5.3: Benefit of momentum

Choosing the Momentum Value

So how do you set the optimum value of momentum? It is often helpful to experiment with different values. One rule of thumb is to reduce the learning rate when using a lot of momentum. Using this idea we select a low learning rate of 0.0001 combined with a relatively high momentum value of 0.95:

```
sgd = SGD(lr=0.0001,  momentum=0.95,
   nesterov=True)
fit1.compile(loss='mean_squared_error',
   optimizer=sgd)
```

Notice we set `nestrov = True` to use Nesterov's accelerated gradient descent. This is a first-order optimization method designed to improve stability and speed up convergence of gradient descent.

NOTE... ✍

During regular gradient descent the gradient is used to update the weights. Use of the `nestrov` algorithm adds a momentum term to the gradient updates to speed things up a little.

How to Benefit from Mini Batching

The traditional backpropagation algorithm calculates the change in neuron weights, known as delta's or gradients, for every neuron in all the layers of a deep neural network, and for every single epoch. The deltas are essentially calculus derivative adjustments designed to minimize the error between the actual output and the deep neural network output.

A very large deep neural network might have millions of weights for which the delta's need to be calculated. Think about this for a moment....Millions of weights require gradient calculations.... As you might imagine, this entire process can take a considerable amount of time. It is even possible, that the time taken for a deep neural network to converge on an acceptable solution using batch learning propagation makes its use in-feasible for a particular application.

Mini batching is one common approach to speeding up neural network computation. It involves computing the gradient on several training examples (batches) together, rather than for each individual example as happens in the original stochastic gradient descent algorithm.

A batch contains several training examples in one forward/backward pass. To get a sense of the computational efficiency of min-batching, suppose you had a batch size of 500,

with 1000 training examples. It will take only 2 iterations to complete 1 epoch.

NOTE... ✍

The larger the batch size, the more memory you will need to run the model.

Fitting the Model

The model, with a batch size of 10, is fit over 700 epochs as follows:

```
fit1.fit(x_train, y_train, batch_size=10,
   nb_epoch=700)
```

Once the model has completed, you can evaluate the train and test set MSE:

```
score_train = fit1.evaluate(x_train,
        y_train,
        batch_size=10)

score_test = fit1.evaluate(x_test,
        y_test,
        batch_size=10)

print "in train MSE = ", round(score_train
   ,6)
in train MSE =  0.003548

print "in test MSE = ", round(score_test,6)
in test MSE =  0.000702
```

And to convert the predictions back to their original scale, so we can view them individually:

```
pred1=fit1.predict(x_test)
pred1 = scaler_y.inverse_transform(np.array
   (pred1).reshape((len(pred1), 1)))

pred1=np.exp(pred1)

print np.rint(pred1)

[ 12658.]
 [ 11512.]
 [ 11353.]
 [ 11699.]
 [ 13374.]
 [ 14311.]
 [ 15937.]
 [ 14842.]
 [ 13155.]
 [ 12533.]
 [ 14358.]
 [ 14718.]]
```

Figure 5.4 plots the actual and predicted values, alongside the client comfort interval (CI). Several of the actual observations are outside of the CI. This might suggest tweaking model parameters and/ or the further pre-processing of the data. Of course, we could also select an alternative neural network model. We pursue this option further in the next chapter.

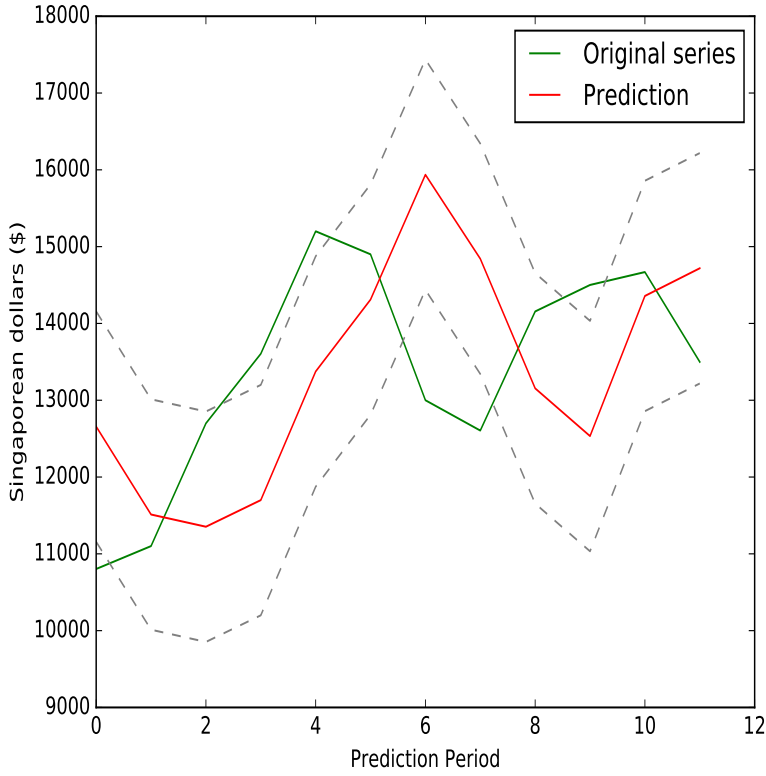


Figure 5.4: Actual and forecast values for Simple RNN

Additional Resources to Check Out

- Look at the Keras documentation. Whilst you are at it, also browse Theano and TensorFlow documentation.
- For an interesting application to weather forecasting see Abbot, John, and Jennifer Marohasy. "Using lagged and forecast climate indices with artificial intelligence to predict monthly rainfall in the Brisbane Catchment, Queensland, Australia." International Journal of Sustainable De-

velopment and Planning 10.1 (2015): 29-41.

Chapter 6

Elman Neural Networks

ELMAN neural networks are a popular partially recurrent neural network. They were initially designed to learn sequential or time-varying patterns and have been successfully used in pattern classification, control, optimization.

They are composed of an input layer, a context layer (also called a recurrent or delay layer see Figure 6.1) , a hidden layer, and an output layer. Each layer contains one or more neurons which propagate information from one layer to another by computing a nonlinear function of their weighted sum of inputs.

In an Elman neural network the number of neurons in the context layer is equal to the number of neurons in the hidden layer. In addition, the context layer neurons are fully connected to all the neurons in the hidden layer.

Similar to a regular feedforward neural network, the strength of all connections between neurons is determined by a weight. Initially, all weight values are chosen randomly and are optimized during training.

Memory occurs through the delay (context) units which are fed by hidden layer neurons. The weights of the recurrent connections from hidden layer to the delay units are fixed at 1. This result in the delay units always maintaining a copy of the previous values of the hidden units.

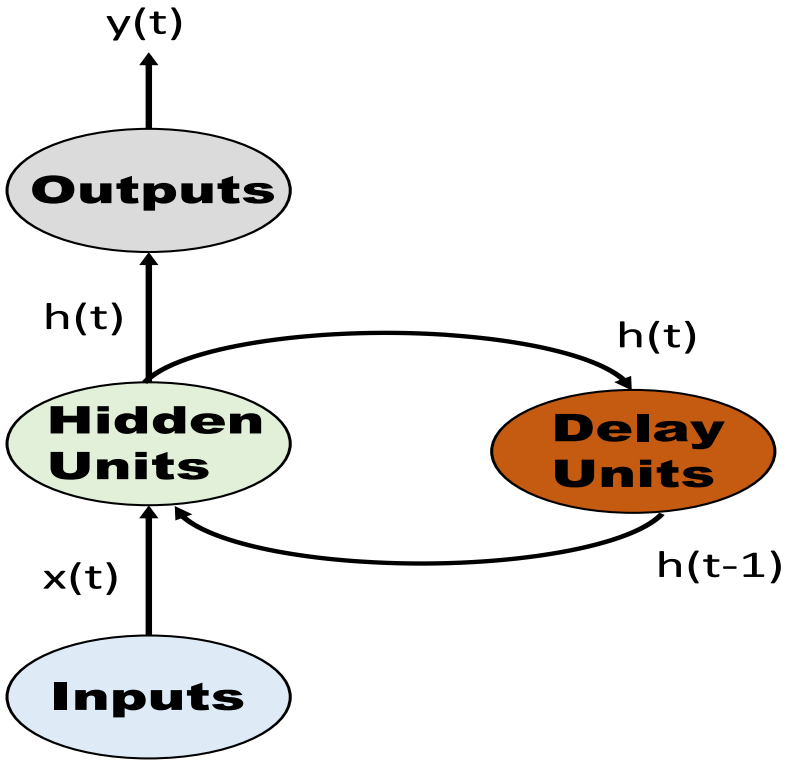


Figure 6.1: Elman Neural Network

Prepare You Data for Easy Use

To illustrative the construction of an Elman network, we continue with the COE price data. Load the saved csv file and process the data into target and attributes:

```
import numpy as np
import pandas as pd
loc= "C:\\Data\\COE.csv"
temp = pd.read_csv(loc)
data= temp.drop( temp.columns [[0,1]] ,
   axis =1)
y=data['COE$']
```

```
x=data.drop( data.columns [[0,4]] , axis
   =1)
x=x.apply(np.log)
x=pd.concat([x,data['Open?']], axis=1)
```

The above will be familiar to you from page 51. The attributes are stored in `x`, and the target (COE price) in `y`.

Scaling to [0,1] Range

Next, scale the attributes and target variable to lie in the range 0 to 1:

```
from sklearn import preprocessing
scaler_x = preprocessing.MinMaxScaler(
   feature_range=(0, 1))
x = np.array(x).reshape((len(x),4 ))
x = scaler_x.fit_transform(x)
scaler_y = preprocessing.MinMaxScaler(
   feature_range=(0, 1))
y = np.array(y).reshape((len(y), 1))
y=np.log(y)
y = scaler_y.fit_transform(y)
y=y.tolist()
x=x.tolist()
```

Much of the above will be familiar to you. However, we will use the `pyneurgen` library to build our Elman neural network. It requires both the attributes and target be passed to it as a list. Hence, the last two lines call the `tolist` method.

How to Model a Complex Mathematical Relationship with No Knowledge

If you have read any traditional books on time series analysis you will no doubt have been introduced to the importance of

estimating the seasonality of your data, gauging the trend, and ensuring the data is stationary (constant mean and variance).

Unfortunately, the presence of trend and seasonal variation can be hard to estimate and/or remove. The chief difficultly is that the underlying dynamics generating the data are unknown. Volumes have been written on how to "guesstimate" it, with little overall agreement on the optimal method.

The Unspoken Reality

It is often very difficult to define trend and seasonality satisfactorily. Moreover, even if you can correctly identify the trend it is important to ensure the right sort of trend (global or local) is modeled. This is important because traditional statistical approaches require the specification of an assumed time-series model, such as auto-regressive models, Linear Dynamical Systems, or Hidden Markov Model.

In practice, traditional statistical tools require considerable experience and skill to select the appropriate type of model for a given data set. The great thing about neural networks is that you do not need to specify the exact nature of the relationship (linear, non-linear, seasonality,trend) that exists between the input and output. The hidden layers of a deep neural network (DNN) remove the need to prespecify the nature of the data generating mechanism. This is because they can approximate extremely complex decision functions.

Figure 6.2 illustrates the general situation. The hidden layer(s) act as a generic function approximator. For a recurrent neural network the hidden and delay units perform a similar function, see Figure 6.3.

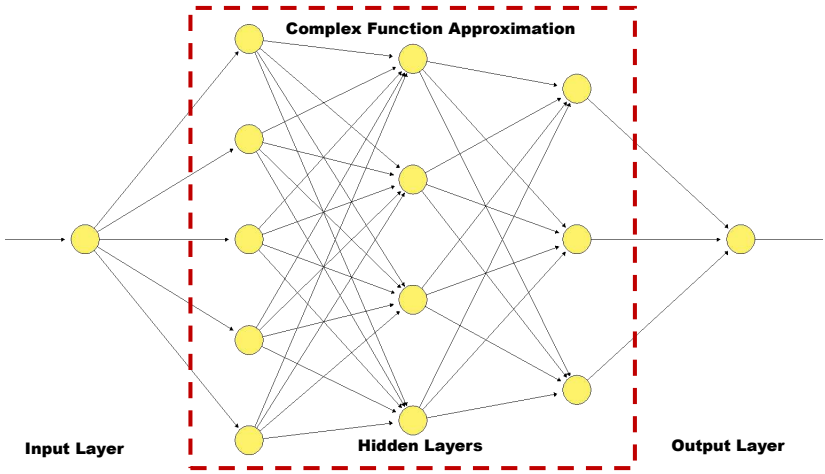


Figure 6.2: A DNN model

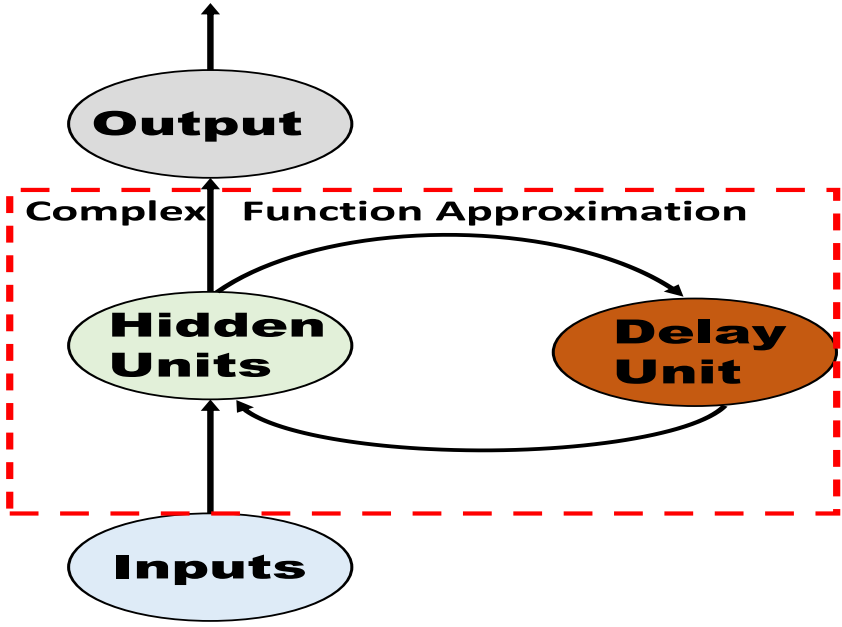


Figure 6.3: Complex function approximation in an Elman neural network

NOTE... ✍

In many real world time series problems the mechanism generating the data is either very complex and/or completely unknown. Such data cannot be adequately described by traditional analytical equations.

Use this Python Library for Rapid Results

Next, load the pyneurgen modules and specify the model:

```
from pyneurgen.neuralnet import NeuralNet
from pyneurgen.recurrent import
   ElmanSimpleRecurrent
```

Now, specify the model. It has 7 nodes in the hidden layer, 4 input nodes corresponding to each of the four attributes, and a single output node which calculates the forecast:

```
import random
random.seed(2016)
fit1 = NeuralNet()
input_nodes = 4
hidden_nodes = 7
output_nodes = 1
fit1.init_layers(input_nodes,
    [hidden_nodes],
    output_nodes,
    ElmanSimpleRecurrent())
fit1.randomize_network()
fit1.layers[1].set_activation_type('sigmoid
   ')
fit1.set_learnrate(0.05)
```

```
fit1.set_all_inputs(x)
fit1.set_all_targets(y)
```

In the above code the sigmoid activation function is used for the hidden nodes with the learning rate set to 5%. That means that 5% of the error between each instance of target and output will be communicated back down the network during training.

Exploring the Error Surface

Neural network models have a lot of weights whose values must be determined to produce an optimal solution. The output as a function of the inputs is likely to be highly nonlinear which makes the optimization process complex. Finding the globally optimal solution that avoids local minima is a challenge because the error function is in general neither convex nor concave. This means that the matrix of all second partial derivatives (often known as the Hessian) is neither positive semi definite, nor negative semi definite. The practical consequence of this observation is that neural networks can get stuck in local minima, depending on the shape of the error surface.

To make this analogous to one-variable functions notice that sin(x) is in neither convex nor concave. It has infinitely many maxima and minima, see Figure 6.4 (top panel). Whereas x^2 has only one minimum and $-x^2$ only one maximum, see Figure 6.4 (bottom panel). The practical consequence of this observation is that, depending on the shape of the error surface, neural networks can get stuck in local minima.

If you plotted the neural network error as a function of the weights, you would likely see a very rough surface with many local minima. As shown in Figure 6.5, it can have very many peaks and valleys. It may be highly curved in some directions while being flat in others. This makes the optimization process very complex.

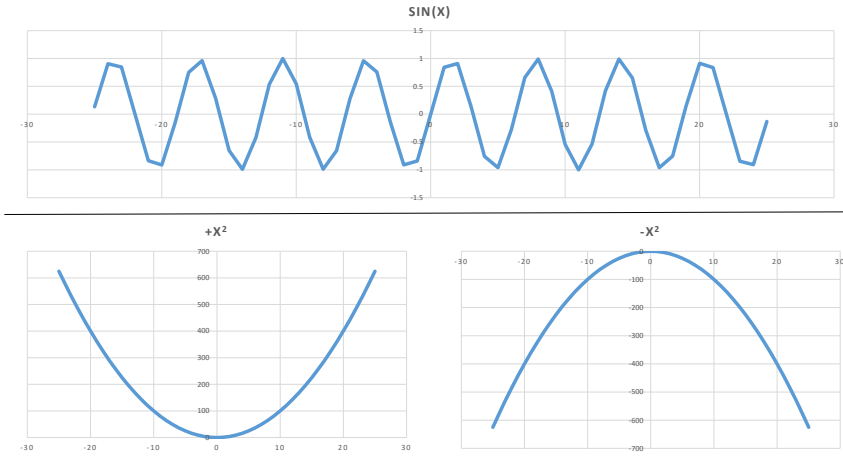
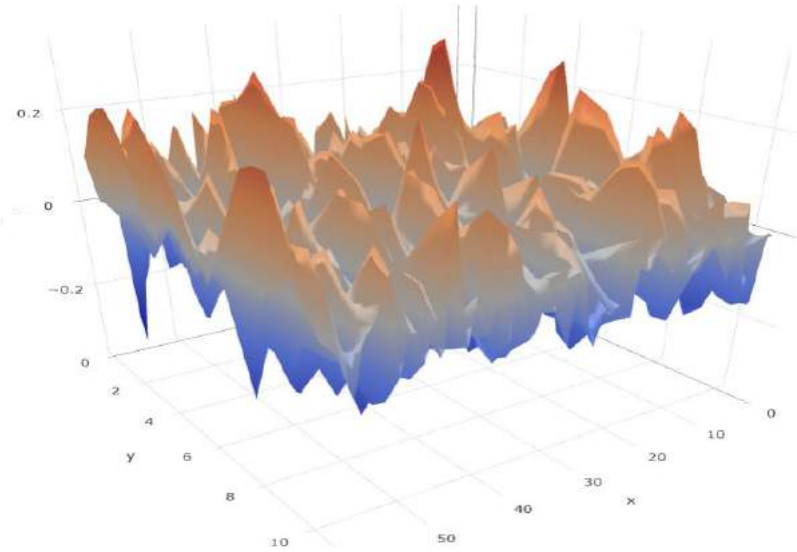


Figure 6.4: One variable functions sin(x), $+x^2$ and $-x^2$



comfort interval

Figure 6.5: Complex error surface of a typical optimization problem

A Super Simple Way to Fit the Model

The train and test samples are obtained as follows:

```
length = len(x)
learn_end_point = int(length * 0.95)
fit1.set_learn_range(0, learn_end_point)
fit1.set_test_range(learn_end_point + 1,
   length-1)
```

The train set contains approximately 95% of the samples.

The `learn` function

The `learn` function fits the model to the data. For this illustration we run the model for 100 epochs and view the results by epoch:

```
fit1.learn(epochs=100, show_epoch_results=
   True,random_testing=False)
```

As the model runs you should see the train set MSE per epoch printed to your screen:

```
epoch: 0 MSE: 0.00573194277476
epoch: 1 MSE: 0.00197294353497
epoch: 2 MSE: 0.00190746802167
epoch: 3 MSE: 0.00184641750107
epoch: 4 MSE: 0.00178925912178
epoch: 5 MSE: 0.00173554922549
epoch: 6 MSE: 0.00168491665439
```

NOTE... ✍

Figure 6.6 plots the MSE by epoch. It falls quite quickly for the first few epochs, and then declines at a gentle slope leveling off by the 100 epoch.

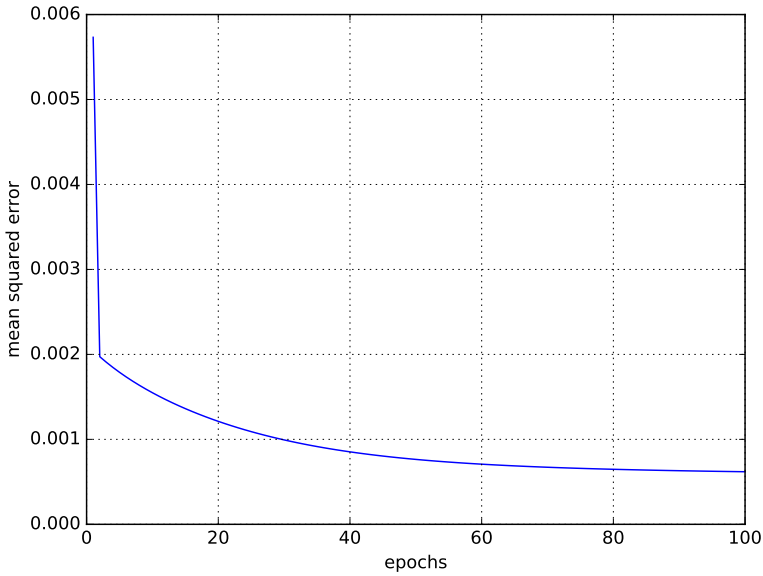


Figure 6.6: MSE by epoch for the Elman Neural Network

Test Set MSE

The final test set MSE is obtained via the `test` function:

```
mse = fit1.test()
print "test set  MSE =",np.round(mse,6)
test set  MSE = 0.000118
```

Transforming the predictions back to their original scale can be achieved via the `inverse_transform` function:

```
pred = [item[1][0] for item in fit1.
   test_targets_activations]
pred1 = scaler_y.inverse_transform(np.array
   (pred).reshape((len(pred), 1)))
pred1=np.exp(pred1)
```

Figure 6.7 plots the predictions and observed values. The model appears to capture the underlying dynamics of the data.

Furthermore, most of the realized values are within the specified forecast tolerance.

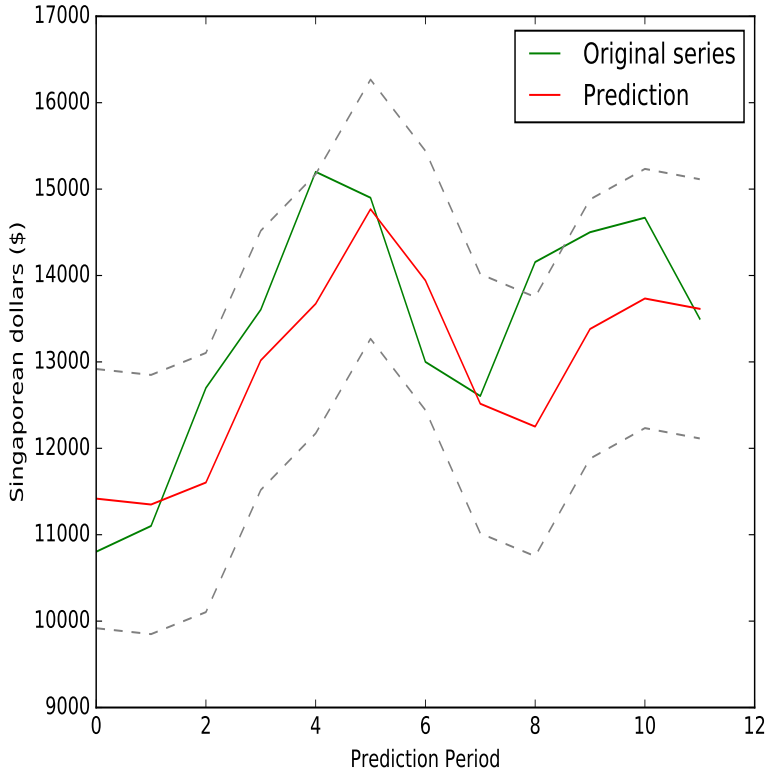


Figure 6.7: Observed and predicted values for COE using an Elman neural network.

Additional Resources to Check Out

The following articles further explore the use of Elman neural networks:

- Case Studies for Applications of Elman Recurrent Neural Networks. In the book. Recurrent Neural Networks,

Edited by Xiaolin Hu. 2008. InTech Press.

- Sundaram, N. Mohana, S. N. Sivanandam, and R. Subha. "Elman Neural Network Mortality Predictor for Prediction of Mortality Due to Pollution." International Journal of Applied Engineering Research 11.3 (2016): 1835-1840.

- Wysocki, Antoni, and Maciej Ławryńczuk. "Elman neural network for modeling and predictive control of delayed dynamic systems." Archives of Control Sciences 26.1 (2016): 117-142.

- Tan, Chao, et al. "A pressure control method for emulsion pump station based on elman neural network." Computational intelligence and neuroscience 2015 (2015): 29.

- Liu, Hongmei, Jing Wang, and Chen Lu. "Rolling bearing fault detection based on the teager energy operator and elman neural network." Mathematical problems in engineering 2013 (2013).

Chapter 7

Jordan Neural Networks

A Jordan network is a single hidden layer feed forward neural network. It is similar to the Elman neural network. The only difference is that the context (delay) neurons are fed from the output layer instead of the hidden layer, see Figure 7.1. It therefore "*remembers*" the output from the previous time-step. Like the Elman neural network, it is useful for predicting time series observations which have a short memory.

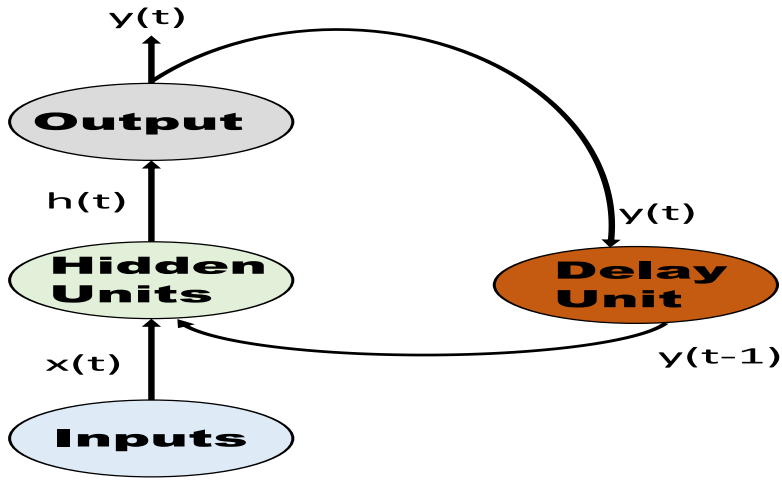


Figure 7.1: Structure of the Jordan Neural Network

NOTE... ✍

In a Jordan network the context layer is directly connected to the input of the hidden layer with a single delay.

The Fastest Path to Data Preparation

We illustrate the construction of a Jordan neural network using the COE price series. Import the data from the saved csv file and create the target (`y`) and attribute sets (`x`):

```
import numpy as np
import pandas as pd
loc= "C:\\Data\\COE.csv"
temp = pd.read_csv(loc)
data= temp.drop( temp.columns [[0,1]] ,
   axis =1)
y=data['COE$']
x=data.drop( data.columns [[0,4]] , axis
   =1)
x=x.apply(np.log)
x=pd.concat([x,data['Open?']], axis=1)
```

The next step, which you are now familiar, is to scale the attributes/ target variable. This is followed by use of the `tolist` method to put the data into an input format suitable for the `pyneurgen` library:

```
from sklearn import preprocessing
scaler_x = preprocessing.MinMaxScaler(
   feature_range=(0, 1))
x = np.array(x).reshape((len(x),4 ))
x = scaler_x.fit_transform(x)
scaler_y = preprocessing.MinMaxScaler(
   feature_range=(0, 1))
```

```
y = np.array(y).reshape((len(y), 1))
y=np.log(y)
y = scaler_y.fit_transform(y)
y=y.tolist()
x=x.tolist()
```

A Straightforward Module for Jordan Neural Networks

Import the required modules, in this case `NeuralNet` and `JordanRecurrent`, and specify the model parameters:

```
from pyneurgen.neuralnet import NeuralNet
from pyneurgen.recurrent import
   JordanRecurrent
import random
random.seed(2016)
fit1 = NeuralNet()
input_nodes = 4
hidden_nodes = 7
output_nodes = 1
existing_weight_factor = 0.9
```

The python object fit1 will contain the model. It has 7 nodes in the hidden layer, one input node for each attribute and a single output node The `existing_weight_factor` is set to 90%.

NOTE... ✍

The `existing_weight_factor` is associated with the delay node. It controls the weight associated with the previous output.

Specifying the Model

The model can be specified as follows:

```
fit1.init_layers(input_nodes,
    [hidden_nodes],
    output_nodes,
    JordanRecurrent(existing_weight_factor))
fit1.randomize_network()
fit1.layers[1].set_activation_type('sigmoid')
fit1.set_learnrate(0.05)
fit1.set_all_inputs(x)
fit1.set_all_targets(y)
```

The learning rate is set to 5%, with sigmoid activation functions used in the hidden nodes.

As we did earlier, 95% of the sample is used for training. The training and test samples are determined as follows:

```
length = len(x)
learn_end_point = int(length * 0.95)
fit1.set_learn_range(0, learn_end_point)
fit1.set_test_range(learn_end_point + 1,
   length-1)
```

Assessing Model Fit and Performance

Now that the model has been fully specified it can be fit to the data:

```
fit1.learn(epochs=100, show_epoch_results=
   True,random_testing=False)
```

As the model is running, you should see the MSE per epoch along the lines of:

```
epoch: 0 MSE: 0.00392271180035
epoch: 1 MSE: 0.00148887861103
epoch: 2 MSE: 0.00148825922623
epoch: 3 MSE: 0.00148975092454
```

```
epoch: 4 MSE: 0.00149294789624
epoch: 5 MSE: 0.00149740333037
epoch: 6 MSE: 0.0015026753066
epoch: 7 MSE: 0.00150836654247
```

Figure 7.2 plots the train set MSE by epoch. It falls sharply for the first few epochs; followed by a more moderate decline to the 100th epoch.

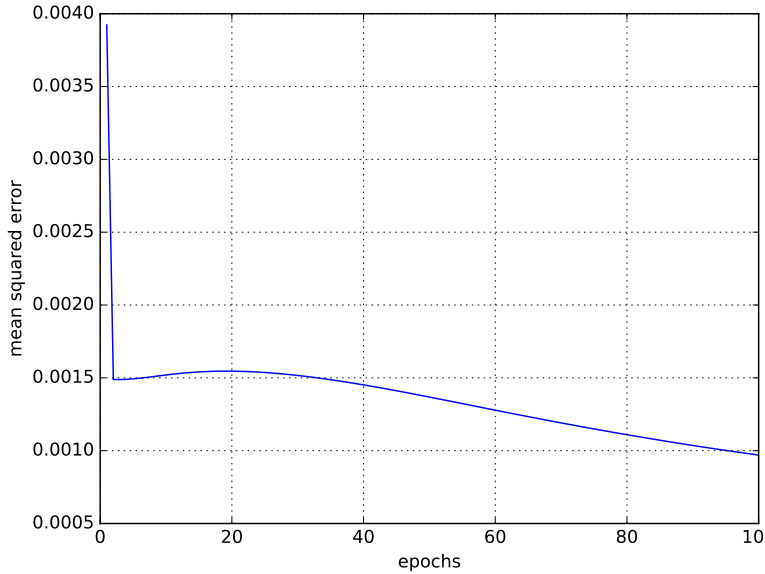


Figure 7.2: MSE by epoch for the Jordan neural network

The overall test set MSE can be calculated using:

```
mse = fit1.test()
print "test set  MSE =",np.round(mse,6)
test set  MSE = 8.4e-05
```

Figure 7.3 plots the predicted and actual observations. Overall, the observed values lie within the tolerance range, and the model appears to adequately capture the dynamics of the underlying time series.

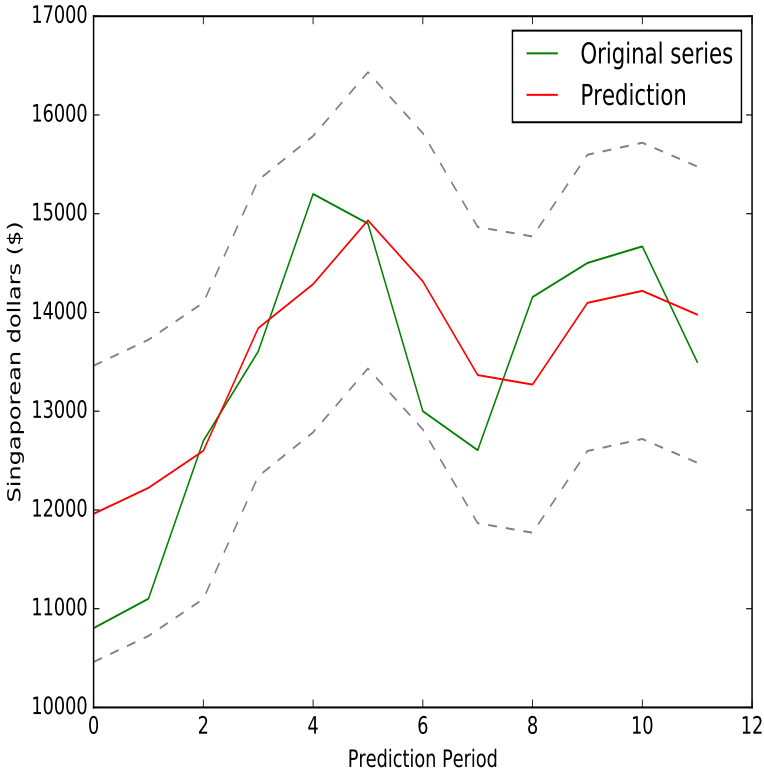


Figure 7.3: Predicted and actual values for Jordan neural network

Additional Resources to Check Out

The following articles further explore the use of Jordan neural networks:

- Abdulkarim, S. A. "Time series prediction with simple recurrent neural networks." Bayero Journal of Pure and Applied Sciences 9.1 (2016): 19-24.
- Wysocki, Antoni, and Maciej Ławryńczuk. "Jordan neu-

ral network for modelling and predictive control of dynamic systems." Methods and Models in Automation and Robotics (MMAR), 2015 20th International Conference on. IEEE, 2015.

- Bilski, Jarosław, and Jacek Smoląg. "Parallel approach to learning of the recurrent jordan neural network." International Conference on Artificial Intelligence and Soft Computing. Springer Berlin Heidelberg, 2013.

- Maraqa, Manar, et al. "Recognition of Arabic sign language (ArSL) using recurrent neural networks." (2012).

Chapter 8

Nonlinear Auto-regressive Network with Exogenous Inputs

A nonlinear auto-regressive network with exogenous inputs neural network (NARX) is a type of recurrent dynamic neural network. They have been used for modeling nonlinear dynamic systems such as heat exchangers, waste water treatment plants and petroleum refinery catalytic reforming systems, movement in biological systems, and even to predict solar radiation.

In this chapter, we illustrate the construction of a NARX model to predict the United Kingdom annual Unemployment rate. Our goal is to create a model that predicts the actual level to within a comfort interval of 2 standard deviations of the model prediction.

What is a NARX Network?

Figure 8.1 outlines a simple schematic of a NARX neural network. The inputs are fed into delay units, which act as memory of previous inputs. The output of the network is also stored in

delay units, which are fed directly into the hidden units.

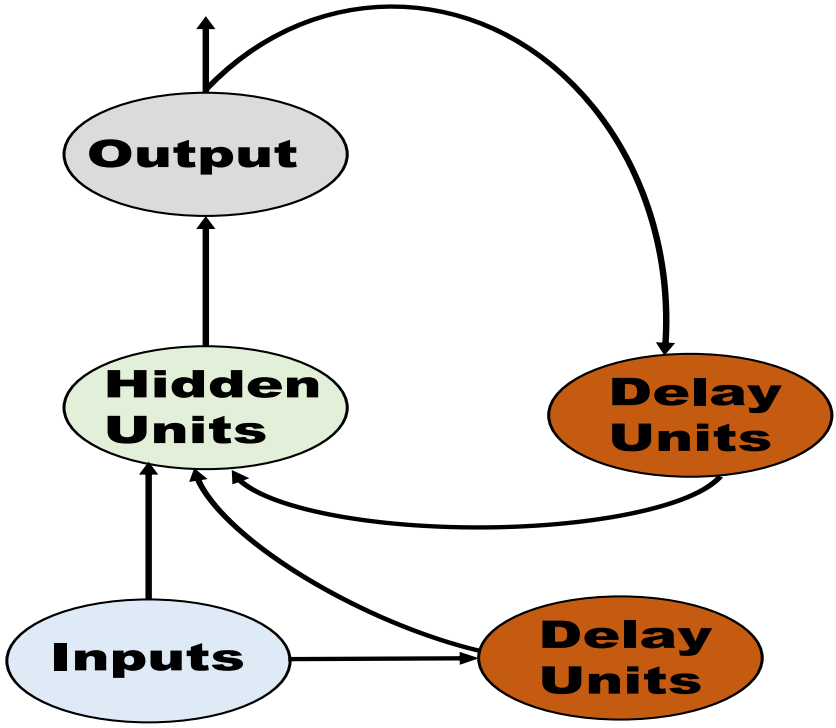


Figure 8.1: Schematic of a NARX network

Further Explanation

In traditional time series modeling the nonlinear autoregressive exogenous model is defined by the nonlinear mapping function f:

$$y_t = f\left[y_{t-1}, y_{t-2}, ..., y_{t-d_y}, x_{t-1}, x_{t-2}, ..., x_{t-d_x}\right]$$

where y (target) and x (attributes) are the past and present inputs to the model; and $d_y \geq 1, d_x \geq 1, d_y \geq d_x$ are the input memory and output memory orders (or delays).

The nonlinear mapping $f(\cdot)$ is generally unknown and needs to be approximated. The approximation can be carried in many ways; when it is approximated by a multilayer perceptron, the resulting neural network is called nonlinear auto-regressive network with exogenous inputs neural network.

NARX neural networks are useful for modelling long term dependencies in time series data because of the extended time delays captured by d_y and d_x. They can remember the output over longer time steps than a Jordan or Elman neural network.

Spreadsheet Files Made Easy with Pandas

The data used for our analysis comes from the Bank of England's three centuries of macroeconomic data research study. The data is hosted at url=http://www.bankofengland.co.uk/research/Documents/onebank/threecenturies_v2.3.xlsx. Here is how to download it:

```
import numpy as np
import pandas as pd
import urllib
url="https://goo.gl/0WVZDW"
loc= "C:\\Data\\UK_Economic.xls"
urllib.urlretrieve(url, loc)
```

In the above code the url address is shortened using the google url shortener. The object loc contains the location you wish to store the file `UK_Economic.xls`. You should adjust it to your desired location.

The spreadsheet file is organized into two parts. The first contains a broad set of annual data covering the UK national accounts and other financial and macroeconomic data stretching back in some cases to the late 17th century.

Using Pandas for Ease

Pandas is the best way to go with this type of file format. Here is how to read it into python:

```
Excel_file = pd.ExcelFile(loc)
```

The spreadsheet contains a lot of worksheets. To view them all use the `print` statement:

```
print Excel_file.sheet_names

[u'Disclaimer', u'Front page', u"What's new
    in V2.3", u'A1. Headline series',..., u
   'M14. Mthly Exchange rates 1963+', u'M15
   . Mthly $-\xa3 1791-2015']
```

The data we need is contained in the `"A1. Headline series"` worksheet. To parse it into Python use:

```
spreadsheet = Excel_file.parse('A1.
   Headline series')
```

Data Dates

The first year used for the analysis is 1855, and the last year is 2015. To see these dates extracted from the spreadsheet use the `print` statement:

```
print spreadsheet.iloc[201,0]
1855

print spreadsheet.iloc[361,0]
2015
```

So we see that row 201 contains the first year (1855), and row 361 the last year used in our analysis (2015).

Working with Macroeconomic Variables

The target variable is the annual unemployment rate. As shown in Figure 8.2, it is subject to considerable variation tied to the business cycle of economic boom followed by bust.

The unemployment rate is read from `spreadsheet` as follows:

```
unemployment = spreadsheet.iloc[201:362,15]
```

The first unemployment observation, stored in column 15 of `spreadsheet`, is located in row 201. The last observation used in our analysis is located in row 362.

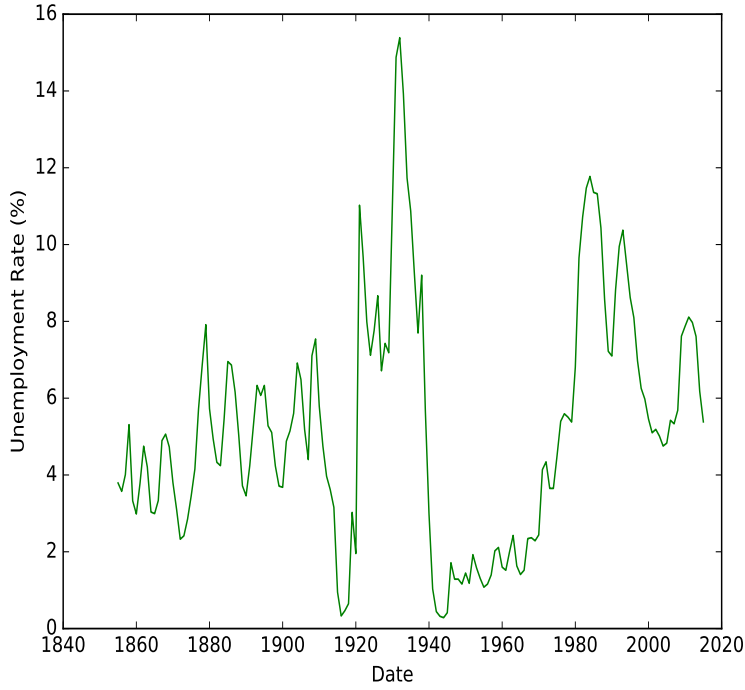


Figure 8.2: UK annual unemployment rate

Working with the Attributes

The attribute set consists of four macroeconomic variables:

1. The consumer price inflation rate;
2. the central bank rank;
3. National debt as percentage of nominal Gross Domestic Product;
4. and the deviation of economic growth (GDP) from trend.

We are interested in predicting the unemployment rate next year, given the value of each of the attributes this year. We can do this by selecting our attribute data to match each year's unemployment rate with last year's value of the attributes. In other words by lagging the attributes by time step (in this case year).

Let's use lagged values of each of the above attributes. Here is one way to do this:

```
inflation = spreadsheet.iloc[200:361,28]
bank_rate =spreadsheet.iloc[200:361,30]
debt= spreadsheet.iloc[200:361,57]
GDP_trend =spreadsheet.iloc[200:361,3]
```

Figure 8.3 plots each of these attributes. Although they all have an impact on general economic conditions, did you notice how each has a unique time dynamic? In fact, the relationship between the annual unemployment rate and each of these variables is quite complex. Figure 8.4 illustrates this. It shows the scatter plot of each attribute against the unemployment rate. The relationships appear to be highly non-linear. A situation for which neural networks are especially suited.

Before moving on to the next section, we need to group the attributes together into the object `x`:

```
x= pd.concat([GDP_trend,debt,bank_rate,
   inflation],axis=1)
```

```
x.columns = ["GDP_trend","debt","bank_rate"
    ,"inflation"]
```

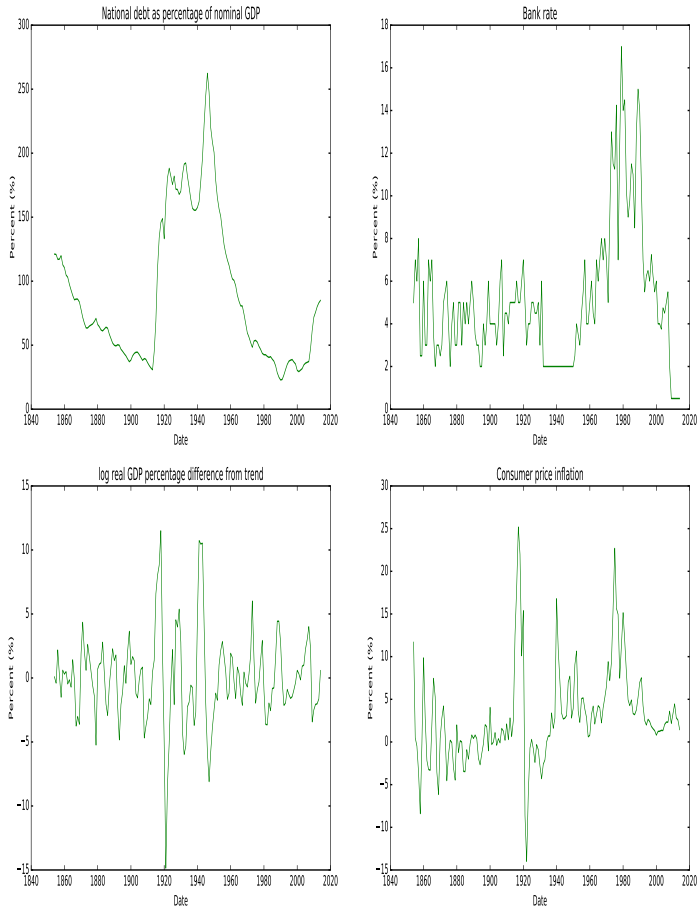


Figure 8.3: Macroeconomic attributes. National debt (top left), Bank rate (top right), GDP trend (bottom left), Consumer price inflation (bottom right)

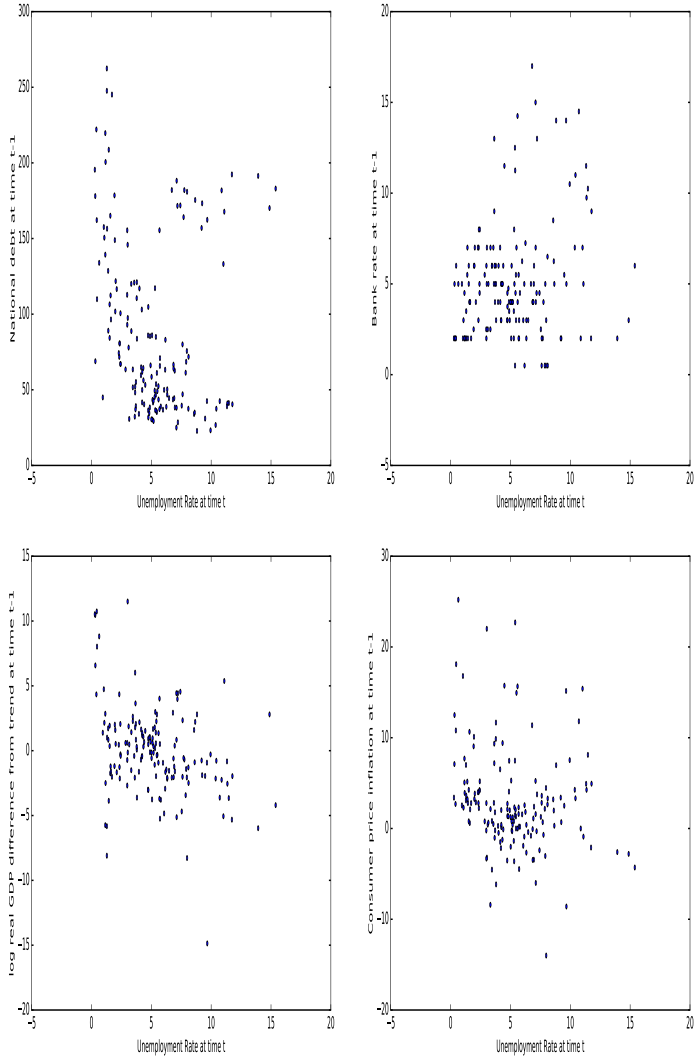


Figure 8.4: Scatter plot of unemployment rate and Macroeconomic attributes. National debt (top left), Bank rate (top right), GDP trend (bottom left), Consumer price inflation (bottom right)

Python and Pandas Data Types

There are three main types of data in Python:

- `strings`,
- `int`.
- `floats`.

How information is stored in a pandas DataFrame (`strings, int, float`) affects what we can do with it. What type is our attribute matrix `x`? To view the types use:

```
print x.dtypes
```

```
GDP_trend     object
debt          object
bank_rate     object
inflation     object
dtype: object
```

Each attribute in `x` is of data type Object. Object is a pandas data type and corresponds to the native Python `string`. In other words, the information in these attributes is stored as text strings. They need to be converted into numerical values to be used in a neural network. Here is how to convert them into the numerical type `float`:

```
x['debt'] = x['debt'].astype('float64')
x['bank_rate'] = x['bank_rate'].astype('float64')
x['GDP_trend'] = x['GDP_trend'].astype('float64')
x['inflation'] = x['inflation'].astype('float64')
```

Pandas and base Python use slightly different names for data types. The type `float64` is actually a pandas data type and corresponds to the native Python type `float`. Table 1 lists basic Pandas and Python types.

Pandas Type	Native Python Type	Comment
`object`	`string`	Text strings
`int64`	`int`	Numeric (integer)
`float64`	`float`	Numeric (decimal)

Table 1: Pandas and Python types.

A Quick Check

Now check things converted as expected:

```
print x.dtypes
GDP_trend     float64
debt          float64
bank_rate     float64
inflation     float64
dtype: object
```

Great, the conversion was successful.

Create the target variable using a similar technique:

```
y=unemployment
print y.dtype
y=pd.to_numeric(y)
```

Finally, save `x` and `y` as separate csv files, we will use them again in chapter 12:

```
loc= "C:\\Data\\economic_x.csv"
x.to_csv(loc)
loc= "C:\\Data\\economic_y.csv"
y.to_csv(loc)
```

Scaling Data

Let's work with the data we just saved. First load it into Python:

```
loc= "C:\\Data\\economic_x.csv"
x = pd.read_csv(loc)
x= x.drop( x.columns [[0]] , axis =1)
loc= "C:\\Data\\economic_y.csv"
y = pd.read_csv(loc, header=None)
y= y.drop( y.columns [[0]] , axis =1)
```

The saved csv file for `y` did not include a header hence in reading it into Python the argument `header` was set to `None`.

Both the attribute and target are scaled to lie in the range 0 to 1, and converted to a list type for use with the `pyneurgen` library:

```
from sklearn import preprocessing
scaler_x = preprocessing.MinMaxScaler(
   feature_range=(0, 1))
x = np.array(x).reshape((len(x),4 ))
x = scaler_x.fit_transform(x)
scaler_y = preprocessing.MinMaxScaler(
   feature_range=(0, 1))
y = np.array(y).reshape((len(y), 1))
y = scaler_y.fit_transform(y)
y=y.tolist()
x=x.tolist()
```

A Tool for Rapid NARX Model Construction

The `pyneurgen` library provides the tools we need. First load the required modules:

```
from pyneurgen.neuralnet import NeuralNet
from pyneurgen.recurrent import NARXRecurrent
```

The model will have 10 hidden nodes with a delay of 3 time steps on the input attributes, and a delay of 1 time step for the output. In a NARX network you can also specify the incoming

weight from the output, and the income weight from the input. We set these to 10% and 80% respectively.

Here are the initial model parameters:

```
import random
random.seed(2016)
input_nodes = 4
hidden_nodes = 10
output_nodes = 1
output_order = 1
input_order = 3
incoming_weight_from_output = 0.1
incoming_weight_from_input = 0.8
```

The model is specified as follows:

```
fit1 = NeuralNet()
fit1.init_layers(input_nodes, [hidden_nodes
   ], output_nodes,
         NARXRecurrent(
             output_order,
             incoming_weight_from_output,
             input_order,
             incoming_weight_from_input))
fit1.randomize_network()
fit1.layers[1].set_activation_type('sigmoid
   ')
fit1.set_learnrate(0.35)
fit1.set_all_inputs(x)
fit1.set_all_targets(y)
```

The above code follows the format we have seen previously. The sigmoid is specified as the activation function, and the learning rate is set to a moderate level of 35%.

NOTE... ✍

NARX can be trained using backpropagation through time.

The train set consists of 85% of the examples, with the remaining observations used for the test set:

```
length = len(x)
learn_end_point = int(length * 0.85)
fit1.set_learn_range(0, learn_end_point)
fit1.set_test_range(learn_end_point + 1,
   length-1)
```

How to Run the Model

The learn function runs the model given the sample data:

```
fit1.learn(epochs=12,
        show_epoch_results=True,
        random_testing=False)
```

The model is run for 12 epochs. At each epoch you should see the MSE reported along the lines of:

```
epoch: 0 MSE: 0.0124873980979
epoch: 1 MSE: 0.00642738416024
epoch: 2 MSE: 0.00603644943984
epoch: 3 MSE: 0.00575579771091
epoch: 4 MSE: 0.00554626175573
epoch: 5 MSE: 0.00538648176859
epoch: 6 MSE: 0.0052644996795
epoch: 7 MSE: 0.00517325931655
epoch: 8 MSE: 0.00510812675451
epoch: 9 MSE: 0.00506549744663
epoch: 10 MSE: 0.00504201592181
```

```
epoch: 11 MSE: 0.00503417011242
```

Figure 8.5 plots the MSE by epoch values. It falls sharply for the first two epochs, then declines smoothly to the last epoch.

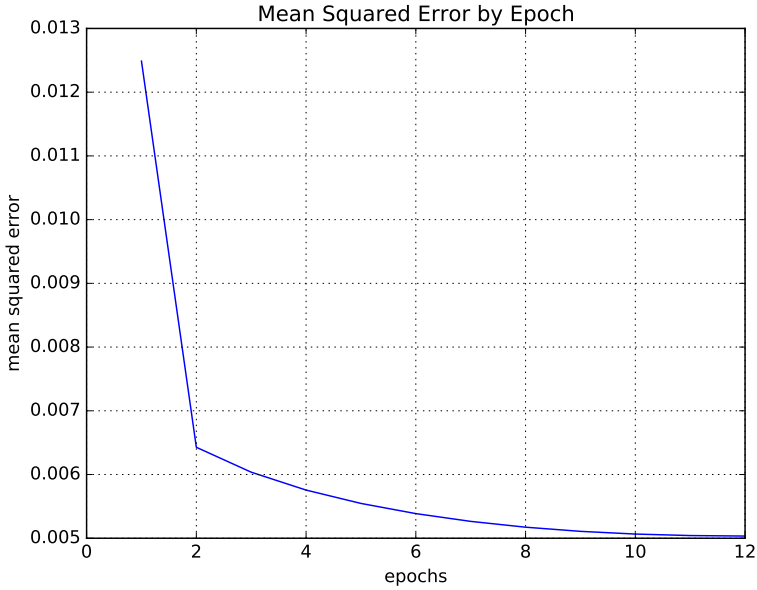


Figure 8.5: MSE by epoch for NARX model

The MSE and fitted model predictions (re scaled) are given by:

```
mse = fit1.test()
print "MSE for test set = ",round(mse,6)
MSE for test set =  0.003061

pred = [item[1][0] for item in fit1.
   test_targets_activations]
pred1 = scaler_y.inverse_transform(np.array
   (pred).reshape((len(pred), 1)))
```

Figure 8.6 plots the actual and predicted values. Actual unemployment is, for the most part, adequately contained within the tolerance range.

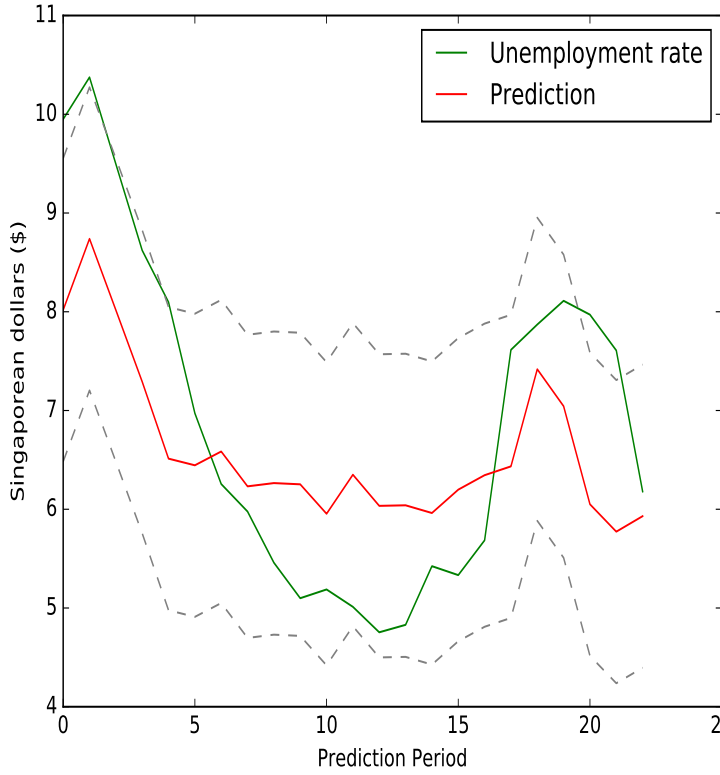


Figure 8.6: Actual and predicted values for NARX model

Additional Resources to Check Out

The NARX model is growing in popularity. Take look at the following articles:

- Caswell, Joseph M. "A Nonlinear Autoregressive Approach to Statistical Prediction of Disturbance Storm

Time Geomagnetic Fluctuations Using Solar Data." Journal of Signal and Information Processing 5.2 (2014): 42.

- William W. Guo and Heru Xue, “Crop Yield Forecasting Using Artificial Neural Networks: A Comparison between Spatial and Temporal Models,” Mathematical Problems in Engineering, vol. 2014, Article ID 857865, 7 pages, 2014. doi:10.1155/2014/857865

- Diaconescu, Eugen. "The use of NARX neural networks to predict chaotic time series." Wseas Transactions on computer research 3.3 (2008): 182-191.

Chapter 9

Long Short-Term Memory Recurrent Neural Network

Long Short-Term Memory recurrent neural networks (LSTM) have outperformed state-of-the-art deep neural networks in numerous tasks such as speech and handwriting recognition. They were specifically designed for sequential data which exhibit patterns over many time steps. In time series analysis, these patterns are called cyclical.

Cyclical Patterns in Time Series Data

Time series data often have cyclic patterns, where the observations rise and fall over long periods of time. For example, Figure 9.1 shows the monthly sales of new one family houses sold in the USA. The time series show a cyclic pattern. The highs and lows over time are tied to the business cycle. Strong demand associated with economic expansion pushes up the demand for homes and hence sales. Whilst weak demand is associated with economic slow-down with a resultant decline in sales.

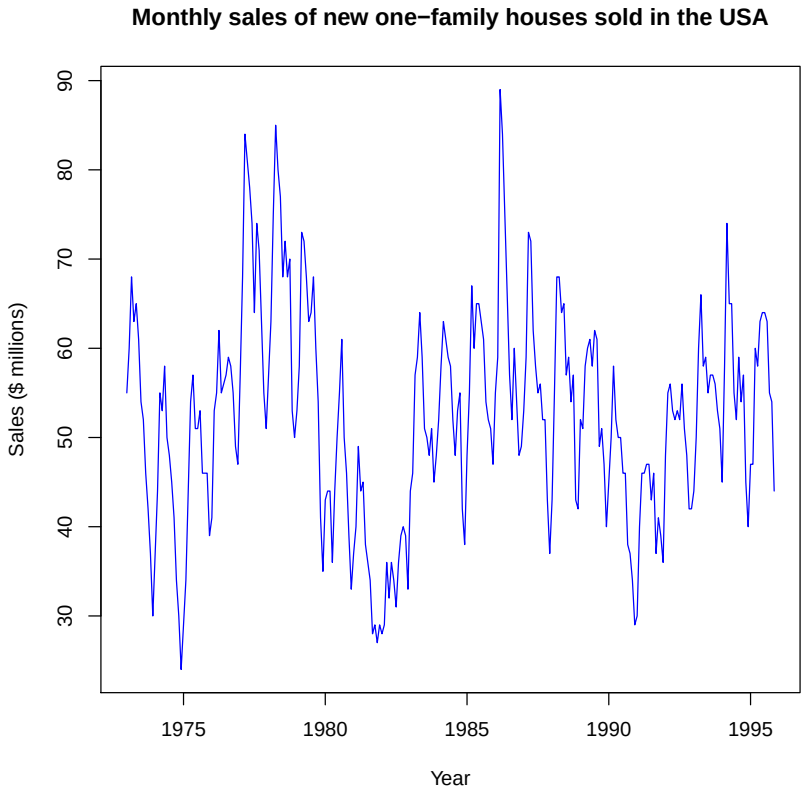


Figure 9.1: Monthly sales of new family homes

Cyclic components are also observed in the natural world. Figure 9.2 shows the monthly number of Sunspots since the 18th century collected by the Royal Observatory of Belgium, Brussels. The data follow an 11 year cycle with variation between 9 to 14 years. Sunspot activity is of interest to Telecom companies because they affect ionospheric propagation of radio waves. Scientists and telecom companies would like to have advanced warning of their level.

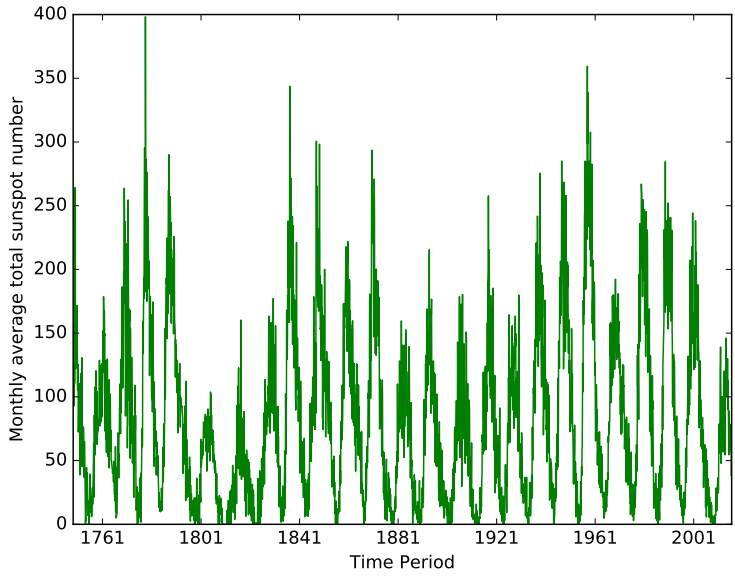


Figure 9.2: Monthly Sunspot activity

In this chapter we develop a LSTM to predict the level of Sunspot activity one month in advance using the data shown in Figure 9.2. The goal is to create a forecasting model that is accurate to a ± 50 tolerance/Comfort Interval. The data can be downloaded directly from the Royal Observatory of Belgium, Brussels:

```
import numpy as np
import pandas as pd
import urllib
url ="https://goo.gl/uWbihf"
data = pd.read_csv(url,sep=";")
loc= "C:\\Data\\Monthly Sunspots.csv"
data.to_csv(loc,index = False)
data_csv = pd.read_csv(loc,header=None)
```

Remember to change `loc` to point to your preferred download location. The final line reads the data back into Python via

the object data_csv. If you prefer to use the full url you can set:

```
url="http://www.sidc.be/silso/INFO/
   snmtotcsv.php"
```

You can also download the data manually by visiting http://www.sidc.be/silso/datafiles.

What is an LSTM?

Figure 9.3 outlines a simple schematic of an LSTM. It is similar to the simple recurrent network we saw in Figure 5.1. However, it replaces the hidden units with memory blocks.

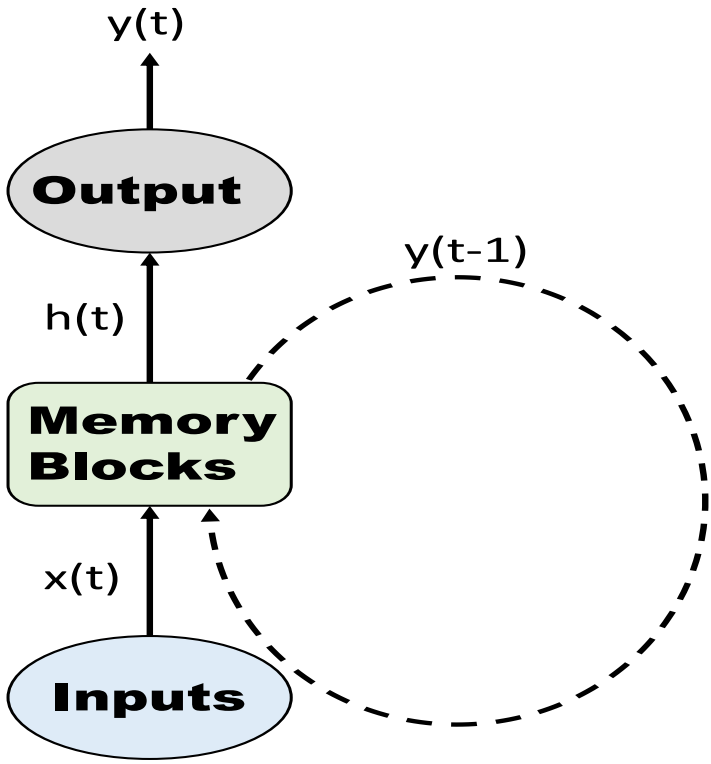


Figure 9.3: Schematic of a LSTM

Efficiently Explore and Quickly Understand Data

The data begin in January 1749 and end at the present day. For our analysis, we use all the data up to June 2016:

```
yt= data_csv.iloc[0:3210,3]
print yt.head()
0     96.7
1    104.3
2    116.7
3     92.8
4    141.7

print yt.tail()
3205    56.4
3206    54.1
3207    37.9
3208    51.5
3209    20.5
```

Figure 9.4 plots the autocorrelation function for the data. It declines over several months to low levels by month 7.

NOTE... ✍

Cyclic components in time series data are the result of a long run structural component in the mechanism generating the data.

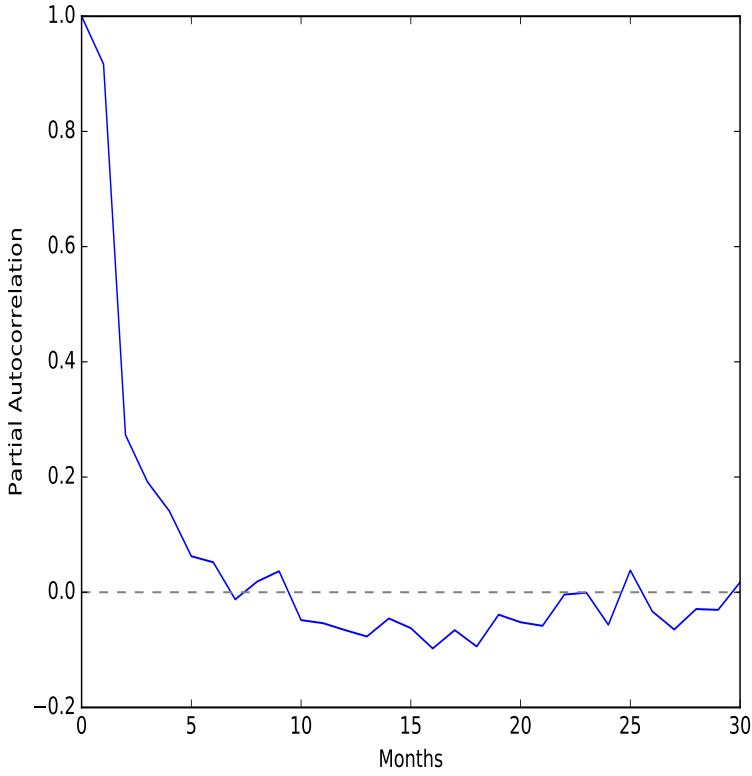


Figure 9.4: Partial autocorrelation function for Monthly Sunspots

Adding Time Lags to Data

For our model we use 5 time lags of the data. A quick way to time lag data is to use the `shift` function:

```
yt_1=yt.shift(1)
yt_2=yt.shift(2)
yt_3=yt.shift(3)
yt_4=yt.shift(4)
yt_5=yt.shift(5)
```

```
data=pd.concat([yt,yt_1, yt_2,yt_3,yt_4,
   yt_5], axis=1)
data.columns = ['yt', 'yt_1', 'yt_2','yt_3'
   ,'yt_4','yt_5']
```

Here is a quick summary of what we have done:

- The variable `yt` contains the current month's number of Sunspots;
- yt_1 contains the previous months number of Sunspots;
- and yt-5, the number of Sunspots five months ago.

To get a visual handle on this use the `tail` function:

```
print data.tail(6)
     yt   yt_1   yt_2   yt_3   yt_4   yt_5
3204  57.0   58.0   62.2   63.6   78.6   64.4
3205  56.4   57.0   58.0   62.2   63.6   78.6
3206  54.1   56.4   57.0   58.0   62.2   63.6
3207  37.9   54.1   56.4   57.0   58.0   62.2
3208  51.5   37.9   54.1   56.4   57.0   58.0
3209  20.5   51.5   37.9   54.1   56.4   57.0
```

You can see in June 2016 (the last observation - row 3209) that `yt` had a value of 20.5, the previous months value was 51.5 which is also the current value of `yt_1`. The pattern is repeated as expected for `yt_2`, `yt_3`, `yt_4`, and `yt_5`.

Missing Data as a result of Data Lag

Look at the first few observations of the data:

```
print data.head(6)
      yt    yt_1    yt_2    yt_3    yt_4   yt_5
0   96.7     NaN     NaN     NaN     NaN    NaN
1  104.3    96.7     NaN     NaN     NaN    NaN
2  116.7   104.3    96.7     NaN     NaN    NaN
3   92.8   116.7   104.3    96.7     NaN    NaN
4  141.7    92.8   116.7   104.3    96.7    NaN
5  139.2   141.7    92.8   116.7   104.3   96.7
```

There are missing values as a consequence of the lagging process. `yt-1` has one missing value (as expected), and `yt-5` has five missing values (again as expected). Remove these missing value with the `dropna()`method, and store the attributes in `x` and the current number of Sunspots (our target variable) in `y`:

```
data = data.dropna()
y=data['yt']
cols=['yt_1','yt_2','yt_3','yt_4','yt_5']
x=data[cols]
```

It is always a good idea to check to see if things are progressing as expected. First, look at the target variable:

```
print y.tail()
3205    56.4
3206    54.1
3207    37.9
3208    51.5
3209    20.5

print y.head()
5      139.2
6      158.0
7      110.5
8      126.5
9      125.8
10     264.3
```

All looks good. Now for `x`:

```
print x.tail()
      yt_1  yt_2  yt_3  yt_4  yt_5
3205  57.0  58.0  62.2  63.6  78.6
3206  56.4  57.0  58.0  62.2  63.6
3207  54.1  56.4  57.0  58.0  62.2
3208  37.9  54.1  56.4  57.0  58.0
3209  51.5  37.9  54.1  56.4  57.0

print x.head()
```

	yt_1	yt_2	yt_3	yt_4	yt_5
5	141.7	92.8	116.7	104.3	96.7
6	139.2	141.7	92.8	116.7	104.3
7	158.0	139.2	141.7	92.8	116.7
8	110.5	158.0	139.2	141.7	92.8
9	126.5	110.5	158.0	139.2	141.7

Again the numbers are in line with expectations.

The LSTM Memory Block in a Nutshell

When I first came across the LSTM memory block, it was difficult to look past the complexity. Figure 9.5 illustrates a simple LSTM memory block with only input, output, and forget gates. In practice, memory blocks may have even more gates!

The key to an intuitive understanding is this:

1. It contains a memory cell and three multiplicative gate units - the input gate, the output gate, and the forget gate.

2. Input to the memory block is multiplied by the activation of the input gate.

3. The output is multiplied by the output gate, and the previous cell values are multiplied by the forget gate.

4. The gates control the information flow into and out of the memory cell.

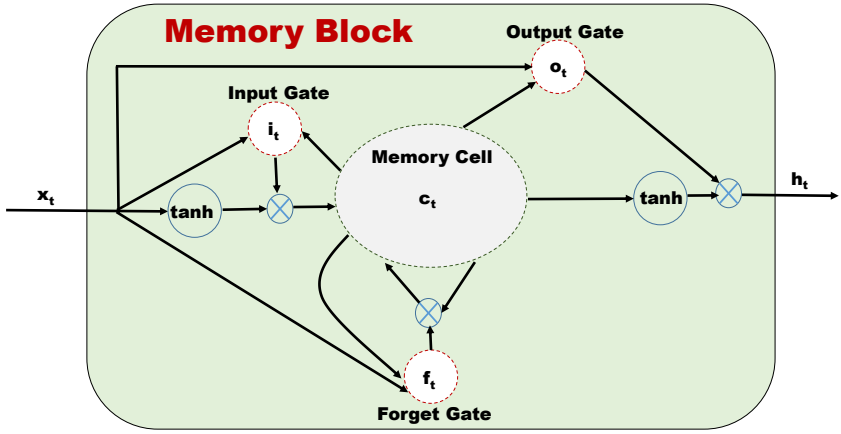


Figure 9.5: Simple memory block

NOTE... ✍

The purpose of gates is to prevent the rest of the network from changing the value of the memory cell over multiple time-steps. This allows the model to preserve information for much longer than in a RNN.

Straightforward Data Transformation for the Train and Test Sets

The data are scaled to lie in the -1 to +1 range via `MinMaxScaler`:

```
from sklearn import preprocessing

scaler_x = preprocessing.MinMaxScaler(
   feature_range=(-1, 1))
```

```
x = np.array(x).reshape((len(x),5 ))
x = scaler_x.fit_transform(x)

scaler_y = preprocessing.MinMaxScaler(
   feature_range=(-1, 1))

y = np.array(y).reshape((len(y), 1))
y = scaler_y.fit_transform(y)
```

The variables `x` and `y` now contain the scaled attributes (lagged target variable) and target variable.

The Train Set

The train set contains data from the start of the series to December 2014. We will use this data to calculate the one month ahead monthly number of Sunspots from January 2015 to June 2016:

```
train_end = 3042
x_train=x[0:train_end,]
x_test=x[train_end+1:3205,]
y_train=y[0:train_end]
y_test=y[train_end+1:3205]
x_train=x_train.reshape(x_train.shape +
   (1,))
x_test=x_test.reshape(x_test.shape + (1,))
```

The Python variable `train_end` contains the row number of the last month of training (December 2014). The attributes were reshaped for passing to the Keras LSTM function. For example, x_train has the shape:

```
print x_train.shape
(3042, 5, 1)
```

It consists of 3,042 examples, 5 lagged variables over 1 time step.

Clarify the Role of Gates

The gating mechanism is what allows LSTMs to explicitly model long-term dependencies. The input, forget, and output gate learn what information to store in the memory, how long to store it, and when to read it out. By acquiring this information from the data, the network learns how its memory should behave.

Input: The input gate learns to protect the cell from irrelevant inputs. It controls the flow of input activations into the memory cell. Information gets into the cell whenever its "write" gate is on.

Forget: Information stays in the cell so long as its forget gate is "off". The forget gate allows the cell to reset itself to zero when necessary.

Output: The output gate controls the output flow of cell activations into the rest of the network. Information can be read from the cell by turning on its output gate. The LSTM learns to turn off a memory block that is generating irrelevant output.

NOTE... ✍

The input gate takes input from the cell at the previous time-step, and the output gate from the current time-step. A memory block can choose to "forget" if necessary or retain their memory over very long periods of time.

Understand the Constant Error Carousel

The memory cell can maintain its state over time. It is often called the "Constant Error Carousel". This is because at its core it is a recurrently self-connected linear unit which recirculates activation and error signals indefinitely. This allows it to provide short-term memory storage for extended time periods.

The hard sigmoid function, see Figure 9.6, is often used as the inner cell activation function. It is piece-wise linear and faster to compute than the sigmoid function for which it approximates. It can be calculated as:

$$f(u) = \max\left(0, \min\left(1, \frac{u+1}{2}\right)\right)$$

Different software libraries have slightly different implementations of the hard sigmoid function.

NOTE... ✍

Each memory block contains a constant error carousel. This is a memory cell containing a self-connected linear unit that enforces a constant local error flow.

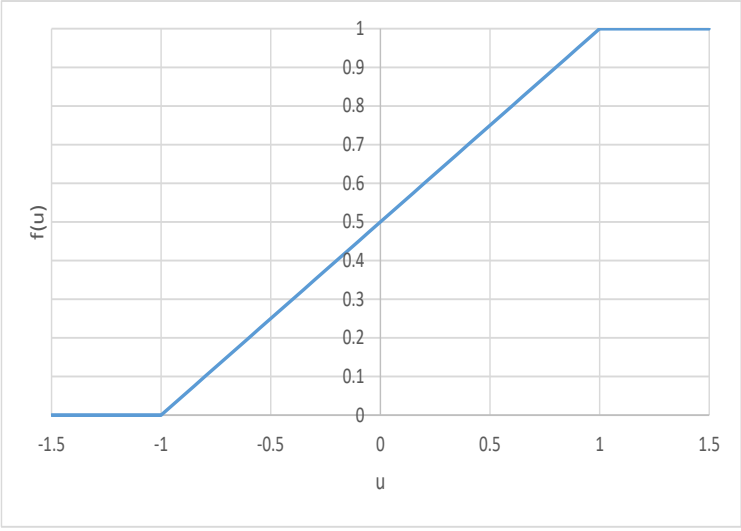


Figure 9.6: Hard Sigmoid activation function

Specifying a LSTM Model the Easy Way

The number of memory blocks is controlled by the `output_dim` argument in the LSTM function. For illustration, four memory blocks are specified. The activation function for the inner cell is set to hard sigmoid:

```
from keras.models import Sequential
```

```
from keras.layers.core import Dense,
   Activation
from keras.layers.recurrent import LSTM
seed=2016
np.random.seed(seed)
fit1 = Sequential()
fit1.add(LSTM(output_dim=4,
        activation='tanh',
        inner_activation='hard_sigmoid',
        input_shape=(5, 1)))
fit1.add(Dense(output_dim=1, activation='
   linear'))
```

As with all of the Keras models we have seen so far, the `Sequential()` model is called first, followed by the model specification. The output layer is called via the `Dense` function, which results in a fully connected layer with a linear activation function.

An alternative to Stochastic Gradient Descent

So far in this text we have used stochastic gradient descent (`optimizer=sgd`). This involves a form of static tuning (global learning rate set a specific value and forget it). During stochastic gradient descent, the update vector components for the weights will take various values. Some updates will be very large, whilst others tiny. Whatever the size or direction a single learning weight is applied across all updates. Whilst this is certainly a very popular learning rule, success depends in a large part on how the learning rate is tuned.

Setting the learning rate involves an iterative tuning procedure in which we manually set the highest possible value. Choosing too high a value can cause the system to diverge,

and choosing this rate too low results in slow learning. Determining a good learning rate is more of an art than science for many problems.

There is some evidence that automatic learning rate adjustments during stochastic gradient descent can deliver enhanced performance. The `rmsprop` algorithm uses a different learning rate for each update vector component. It normalizes the gradients using an exponential moving average of the magnitude of the gradient for each parameter divided by the root of this average. To use the rmsprop algorithm set `optimizer="rmsprop"` in the `compile` function:

```
fit1.compile(loss="mean_squared_error",
   optimizer="rmsprop")
```

Now fit the model:

```
fit1.fit(x_train, y_train, batch_size=1,
   nb_epoch=10,shuffle=False)
```

For this illustration the model is run for 10 epochs, with a batch size of 1. We do not shuffle the training examples to preserve the order in which they were created.

A nice feature of Keras models is the `summary` function, which give a broad overview of a model:

```
print fit1.summary()
```

Layer (type)	Output Shape	Param #	Connected to
lstm_1 (LSTM)	(None, 4)	96	lstm_input_1[0][0]
dense_1 (Dense)	(None, 1)	5	lstm_1[0][0]

Total params: 101

It reports the model has a total of 101 parameters, with 96 associated with the LSTM layer and 5 associated with the output (`Dense`) layer.

Train and Test MSE

Look at the train and test set MSE:

```
score_train = fit1.evaluate(x_train,
   y_train,batch_size=1)
score_test = fit1.evaluate(x_test, y_test,
   batch_size=1)

print "in train MSE = ", round(score_train
   ,4)
in train MSE =    0.0447

print "in test MSE = ", round(score_test,4)
in test MSE =    0.0434
```

Predicted values can be obtained via the `predict` function and converted to their original scale by `inverse_transform`:

```
pred1=fit1.predict(x_test)
pred1 = scaler_y.inverse_transform(np.array
   (pred1).reshape((len(pred1), 1)))
```

Figure 9.7 plots the actual and predicted values alongside the comfort interval/ tolerance range. Overall, the model tends to overestimate the number of Sunspots, although it captures the general trend, and the majority of actual observations are located within the comfort interval.

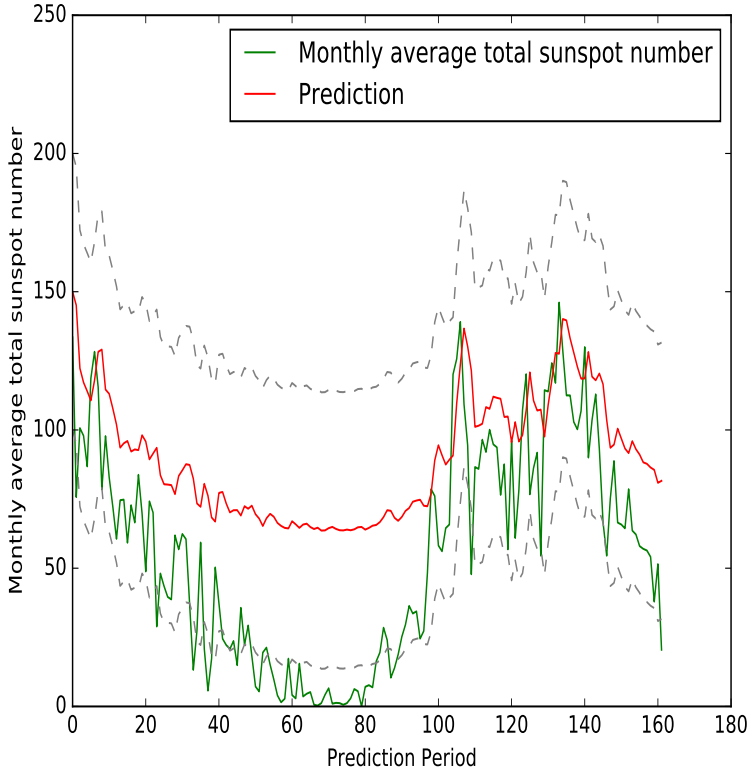


Figure 9.7: Predicted and actual values with comfort interval

Shuffling Examples to Improve Generalization

How might you improve the model? You could tweak some parameters, say by adding additional memory blocks. One hack that occasionally works is to set `shuffle=True`. This means that the training examples will be randomly shuffled at each epoch.

Recall, we have set up the data so that each example consists of the set y_t and $x_t = [y_{t-1}, y_{t-2}, y_{t-3}, y_{t-4}, y_{t-5}]$. So, example 1 contains $E_1 = [y_1, x_1]$, and example 2 contains

$E_2 = [y_2, x_2]$. In general, we have a series of examples $E_1, E_2, E_3, ...E_n$, where n is the length of the training sample.

Just for illustration, suppose we only have five examples:

With `shuffle=False` the model uses them in order E_1, E_2, E_3, E_4, E_5. When `shuffle=True,` we might end up training in the order E_3, E_1, E_5, E_4, E_2. Provided you have chosen an appropriate lag order for your data this approach can improve generalization.

Figure 9.8 shows the predicted and actual values for `shuffle=True`. The model more closely predicts the actual observations.

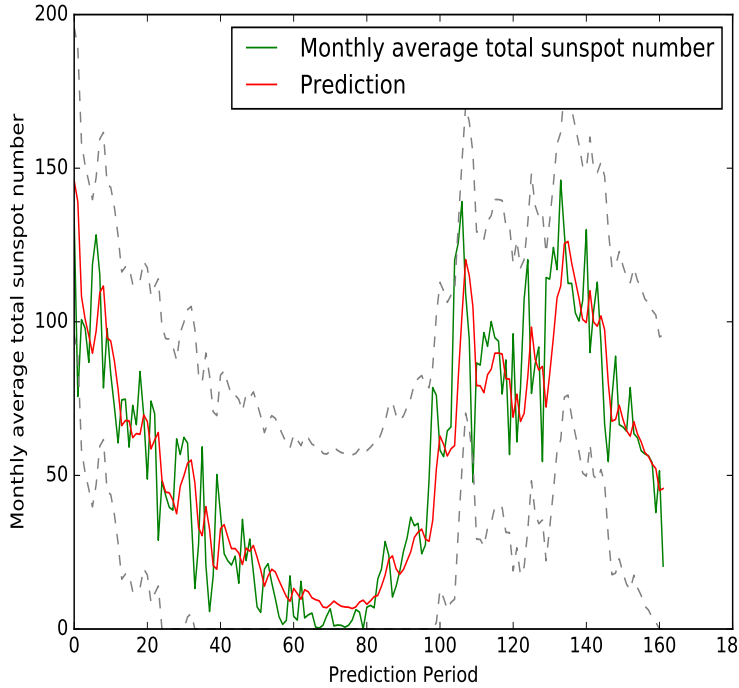


Figure 9.8: Predicted and actual values with comfort interval for `shuffle=True`.

A Note on Vanishing Gradients

The LSTM is widely used because the architecture overcomes the vanishing gradient problem that plagues recurrent neural networks. Errors in a backpropagation neural network are used to drive weight changes. With conventional learning algorithms, such as Back-Propagation Through Time or Real-Time Recurrent Learning, the back-propagated error signals tend to shrink or grow exponentially fast. This causes the error signals used for adapting network weights to become increasingly difficult to propagate through the network.

Figure 9.9 illustrates the situation. The shading of the nodes in the unfolded network indicates their sensitivity to the original input at time 1. The darker the shade, the greater the sensitivity. Over time this sensitivity decays.

It turns out that the evolution of the back-propagated error over time depends exponentially on the size of the weights. This is because the gradient is essentially equal to the recurrent weight matrix raised to a high power. Essentially, the hidden state is passed along for each iteration, so when back-propagating, the same weight matrix is multiplied by itself multiple times. When raised to high powers (i.e. iterated over time) it causes the gradient to shrink (or grow) at a rate that is exponential in the number of time-steps. This is known as the "*vanishing gradient*" if the gradients shrink, or "*exploding gradients*" if the gradient grows. The practical implication is that learning long term dependencies in the data via a simple RNN can take a prohibitive amount of time, or may not happen at all.

The LSTM avoids the vanishing gradient problem. This is because the memory block has a cell that stores the previous values and holds onto it unless a "forget gate" tells the cell to forget those values. Only the cell keeps track of the model error as it flows back in time. At each time step the cell state can be altered or remain unchanged. The cell's local error remains constant in the absence of a new input or error signals.

The cell state can also be fed into the current hidden state calculations. In this way the value of hidden states occurring early in a sequence can impact later observations.

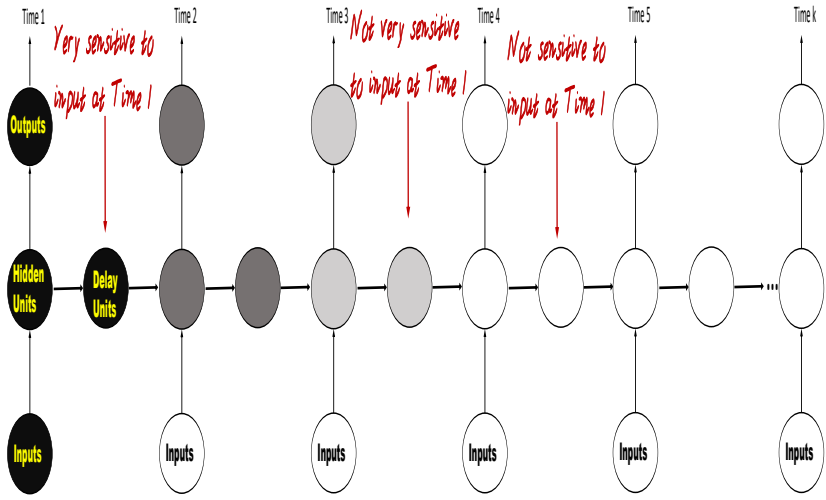


Figure 9.9: Vanishing Gradient's in a RNN

NOTE... ✍

Exploding gradients can be solved by shrinking gradients whose values exceed a specific threshold. This is known as gradient clipping.

Follow these Steps to Build a Stateful LSTM

The LSTM model has the facility to be stateful. This simply means that the states for the samples of each batch are remembered and reused as initial states for the samples in the next batch. This is a useful feature because it allows the model to

carry states across sequence-batches, which can be useful for experimenting with time series data.

Set Up for Statefulness

To use statefulness, you have to explicitly specify the batch size you are using, by passing a `batch_input_shape` argument to the first layer in your model; and set `stateful=True` in your LSTM layer(s):

```
seed=2016
np.random.seed(seed)
fit2 = Sequential()
fit2.add(LSTM(output_dim=4,
        stateful=True,
        batch_input_shape=(1,5,1),
        activation='tanh',
        inner_activation='hard_sigmoid'))

fit2.add(Dense(output_dim=1, activation='
   linear'))
fit2.compile(loss="mean_squared_error",
   optimizer="rmsprop")
```

The `batch_input_shape` takes the batch size (1 in our example), number of attributes (5 time lagged variables) and number of time steps (1 month forecast).

> ***NOTE...*** ✍
>
> The batch size argument passed to `predict` or `evaluate` should match the batch size specified in your model setup.

Forecasting One Time Step Ahead

The model we develop will continue to forecast the 1 month ahead monthly number of Sunspots from January 2015 to June 2016. However, rather than using all the available data the model will forecast the next month based on the last 500 rolling months of data:

```
end_point =len(x_train)
start_point =end_point-500
```

The model has to be trained one epoch at a time with the state reset after each epoch. Here is how to achieve this with a `for` loop:

```
for i in range(len(x_train[start_point:
   end_point])):
        print "Fitting example ",i
        fit2.fit(x_train[start_point:
           end_point], y_train[start_point:
           end_point], nb_epoch=1,
           batch_size=1, verbose=2, shuffle
           =False)
        fit2.reset_states()
```

The last line resets the states of all layers in the model.

The model will take longer to run than the previous model. While, it is running you should see output along the lines of:

```
Fitting example  0
Epoch 1/1
3s - loss: 0.0327
Fitting example  1
Epoch 1/1
3s - loss: 0.0200
Fitting example  2
Epoch 1/1
3s - loss: 0.0184
Fitting example  3
```

Model Performance

As we have previously seen, the predictions using the test set can be obtained via:

```
pred2=fit2.predict(x_test,batch_size=1)
pred2 = scaler_y.inverse_transform(np.array
   (pred2).reshape((len(pred2), 1)))
```

Figure 9.10 plots the observed and predicted values. The model appears to capture the underlying dynamics of the monthly Sunspots series, and is very similar to Figure 9.8. The vast majority of observations lie well within the comfort interval, however the model tends to overestimate the actual forecast. Figure 9.11 shows the performance of the model with `shuffle=True`. It appears very similar to Figure 9.8.

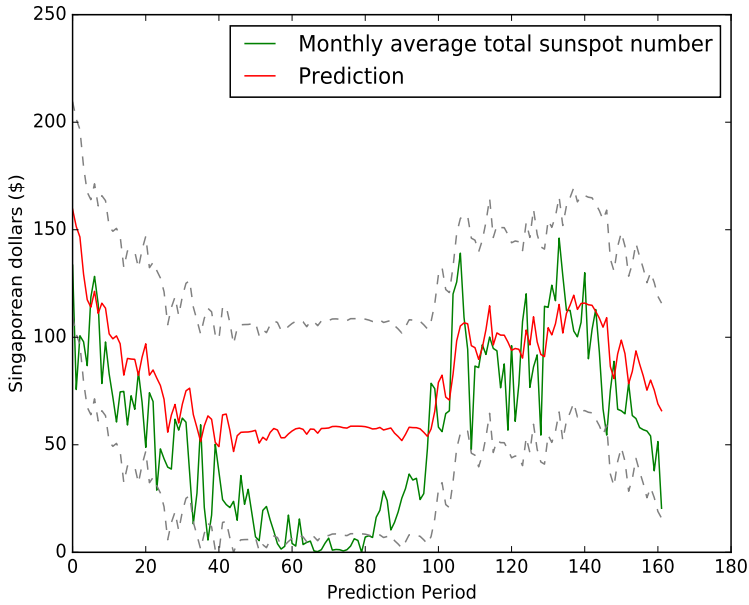


Figure 9.10: Observed and predicted values for the stateful LSTM

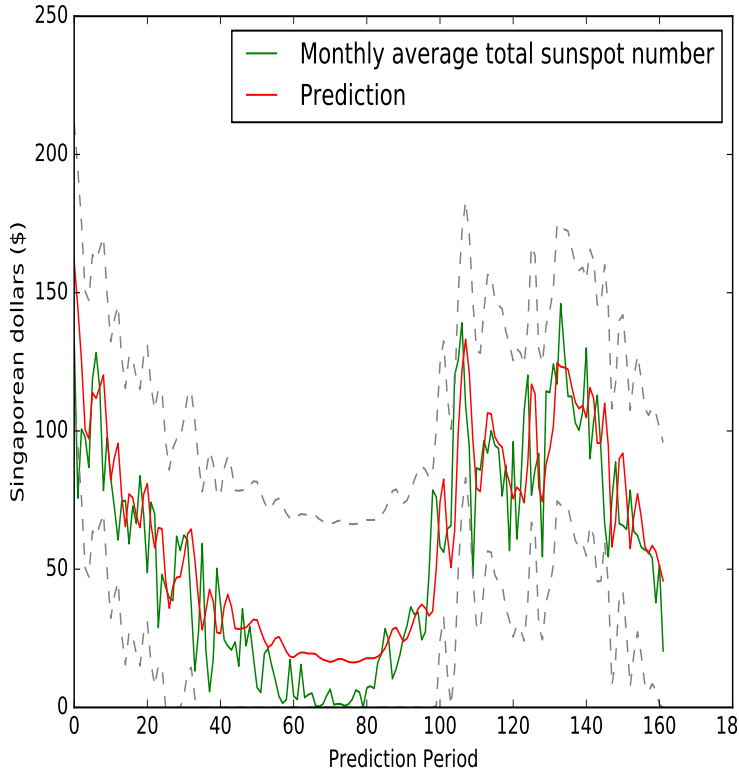


Figure 9.11: Observed and predicted values for the stateful LSTM with `shuffle=True`.

NOTE... ✍

You can stack multiple LSTM layers to make the network structure deep. You can even combine two separate LSTM networks that run in forward and backward directions to implement a bidirectional architecture.

Additional Resources to Check Out

- Be sure to explore Sepp Hochreiter's well thought out article discussing LSTM - See Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.
- For additional details on the hard sigmoid function activation function look at the article by Matthieu Courbariaux. Courbariaux, Matthieu, et al. "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to+ 1 or−."

Chapter 10

Gated Recurrent Unit

Over the past few years, the Gated Recurrent Unit (GRU) has emerged as an exciting new tool for modeling time-series data. They have fewer parameters than LSTM but often deliver similar or superior performance. Just like the LSTM the GRU controls the flow of information, but without the use of a memory unit.

> ***NOTE...*** ✍
>
> Rather than a separate cell state, the GRU uses the hidden state as memory. It also merges the forget input gates into a single "update" gate.

The Gated Recurrent Unit in a Nutshell

Figure 10.1 illustrates the topology of a GRU memory block (node). It contains an update gate (z) and reset gate (r).

- The reset gate determines how to combine the new input with previous memory.

- The update gate defines how much of the previous memory to use in the present.

Together these gates give the model the ability to explicitly save information over many time-steps.

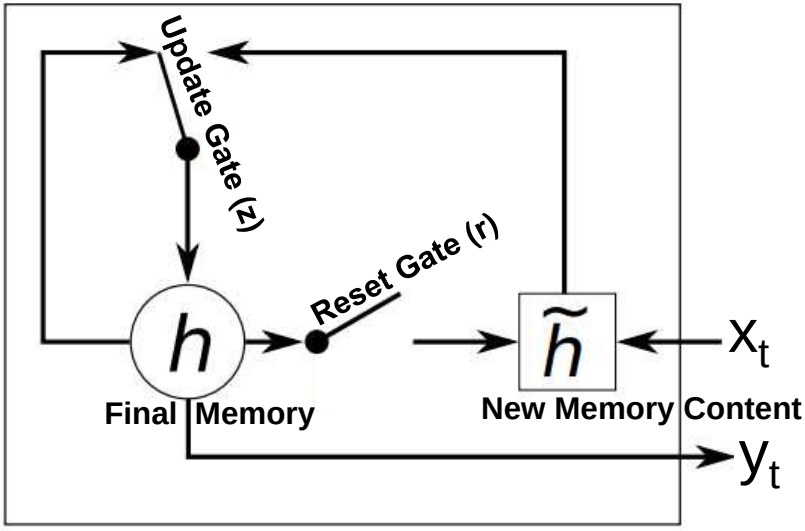


Figure 10.1: Gated Recurrent Unit topology

NOTE... ✍

The GRU is designed to adaptively reset or update its memory content.

The Reset Gate

The reset gate determines how to combine the new input x_t with the previous hidden state h_{t-1}. It gives the model the ability to block or pass information from the previous hidden state. This allows a GRU to "reset" itself whenever a previous

hidden state is no longer relevant. The reset gate is applied directly to the previous hidden state.

NOTE... ✍

If the reset gate is set to 0, it ignores previous memory. This behavior allows a GRU to drop information that is irrelevant.

The Update Gate

The update gate helps the GRU capture long-term dependencies. It determines how much of the previous hidden state h_{t-1} to retain in the current hidden state h_t. In other words, it controls how much of the past hidden state is relevant at time t by controlling how much of the previous memory content is to be forgotten and how much of the new memory content is to be added.

Whenever memory content is considered to be important for later use, the update gate will be closed. This allows the GRU to carry the current memory content across multiple time-steps and thereby capture long term dependencies.

The final memory or activation h_t is a weighted combination of the current new memory content $\widetilde{h}_t$ and previous memory activation h_{t-1}, where the weights are determined by the value of the update gate z_t.

Final Memory

In a GRU the hidden activation (final memory) is simply a linear interpolation of the previous hidden activation (new memory content), with weights determined by the update gate.

A Simple Approach to Gated Recurrent Unit Construction

Let's continue with the Sunspots example, and use the data transformations, train and test sets developed in chapter 9 (for a quick refresh see page 128).

A GRU can be built using the Keras library. We build a model with 4 units as follows:

```
from keras.models import Sequential
from keras.layers.core import Dense,
   Activation
from keras.layers.recurrent import GRU
seed=2016
np.random.seed(seed)
fit1 = Sequential()
fit1.add(GRU(output_dim=4,
        return_sequences=False,
        activation='tanh',
        inner_activation='hard_sigmoid',
        input_shape=(5, 1)))
fit1.add(Dense(output_dim=1, activation='
   linear'))
fit1.compile(loss="mean_squared_error",
   optimizer="rmsprop")
```

The specification is similar to the LSTM. The argument `return_sequences` is set to `False`. For multiple targets it can be set to `True`.

Fit the Model

The model is fit in the usual way, but for illustration, over a single epoch:

```
fit1.fit(x_train, y_train, batch_size=1,
   nb_epoch=10)
```

Since GRU has fewer parameters than the LSTM of chapter 9 it trains much faster:

```
print fit1.summary()
```

Layer (type)	Output Shape	Param #	Connected to
gru_1 (GRU)	(None, 4)	72	gru_input_1[0][0]
dense_1 (Dense)	(None, 1)	5	gru_1[0][0]

Total params: 77

Train and Test set MSE

The train set and test set MSE are given by:

```
score_train = fit1.evaluate(x_train,
   y_train,batch_size=1)
score_test = fit1.evaluate(x_test, y_test,
   batch_size=1)

print "in train MSE = ", round(score_train
   ,5)
in train MSE =  0.01746

print "in test MSE = ", round(score_test,5)
in test MSE =    0.00893
```

And the predictions can be easily calculated and converted to their original scale via the `predict` and `inverse_transform` methods:

```
pred1=fit1.predict(x_test)
pred1 = scaler_y.inverse_transform(np.array
   (pred1).reshape((len(pred1), 1)))
```

Figure 10.2 plots the predicted and actual values. The observations are well contained within the ± 50 comfort interval.

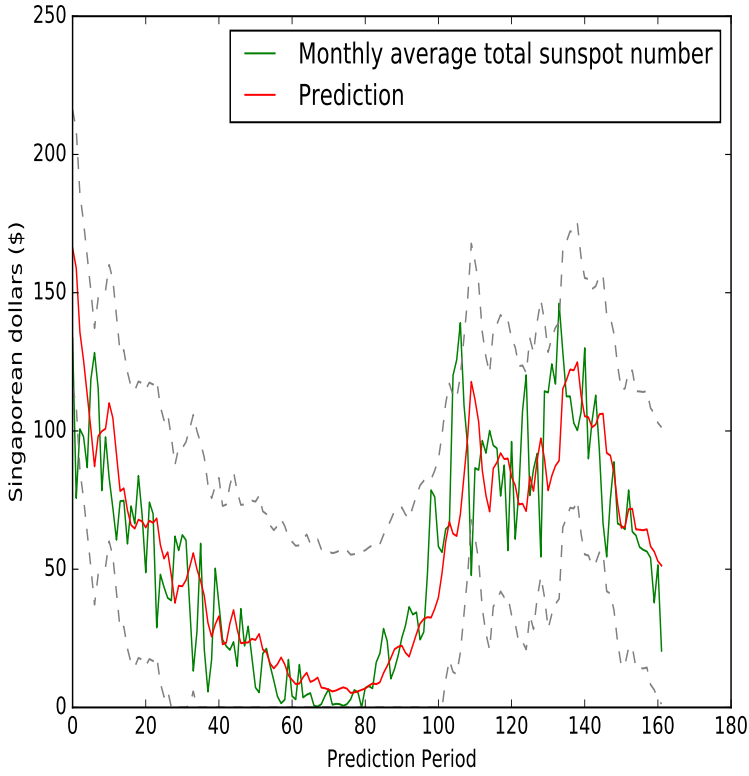


Figure 10.2: Actual and predicted values for GRU Model

A Quick Recap

So far, our data has been in the form: `[examples, features,time steps=1]` and we have framed the problem as one time step for each sample. For each time step we passed one example, so for `x[1]` we have:

```
print x[1]
[-0.30085384 -0.28829734 -0.53390256
   -0.41386238 -0.47614264]
```

The first value represents the scaled y_{t-1}, the second value y_{t-2} and so on at time 1. For $x[2]$ we have:

```
print x[2]
[-0.20642893 -0.30085384 -0.28829734
   -0.53390256 -0.41386238]
```

where the first value again represents the lagged values of y_t at time 2. This corresponds to the traditional approach to modelling time series data.

In order to put the data into this format we re-scaled the input data using (see page 129):

```
x_train=x_train.reshape(x_train.shape +
   (1,))
x_test=x_test.reshape(x_test.shape + (1,))
```

This reshaped the data into the format `[examples, features, time steps]`.

NOTE... ✍

By the way an alternative way to achieve the same reshaped data would be to use:

```
x_train = np.reshape(x_train, (
   x_train.shape[0], 1, x_train.
   shape[1]))
x_test = np.reshape(x_test, (
   x_test.shape[0], 1, x_test.
   shape[1]))
```

How to Use Multiple Time Steps

We can also use multiple time steps to make the prediction for the next time step. In this case, you would reshape the input data so that it has the form `[examples, features=1, time`

steps=5]. Let's look at what this looks like for the Sunspots example:

```
x_train=x[0:train_end,]
x_test=x[train_end+1:3205,]
y_train=y[0:train_end]
y_test=y[train_end+1:3205]
x_train = np.reshape(x_train, (x_train.
   shape[0], 1, x_train.shape[1]))
x_test = np.reshape(x_test, (x_test.shape
   [0], 1, x_test.shape[1]))

print "Shape of x_train is ",x_train.shape
Shape of x_train is  (3042, 1, 5)

print "Shape of x_test is ",x_test.shape
Shape of x_test is  (162, 1, 5)
```

You can see that we have reshaped the data to have 5 time steps with 1 feature. Next, specify the model:

```
seed=2016
num_epochs=1
np.random.seed(seed)
fit1 = Sequential()
fit1.add(GRU(output_dim=4,
        activation='tanh',
        inner_activation='hard_sigmoid',
        input_shape=(1, 5)))
fit1.add(Dense(output_dim=1, activation='
   linear'))
fit1.compile(loss="mean_squared_error",
   optimizer="rmsprop")
fit1.fit(x_train, y_train, batch_size=1,
   nb_epoch=num_epochs)
```

The above will be familiar to you. The key difference is that you have to pass the new input shape to `input_shape` in the `GRU` function.

Figure 10.3 plots the actual and predicted values. The model captures the general dynamics of the actual observations. The actual predictions are well contained within the tolerance range. However, the model appears to consistently predict a higher number of Sunspots than actually observed. Since the model has not been optimized, performance could be improved with a little parameter tweaking.

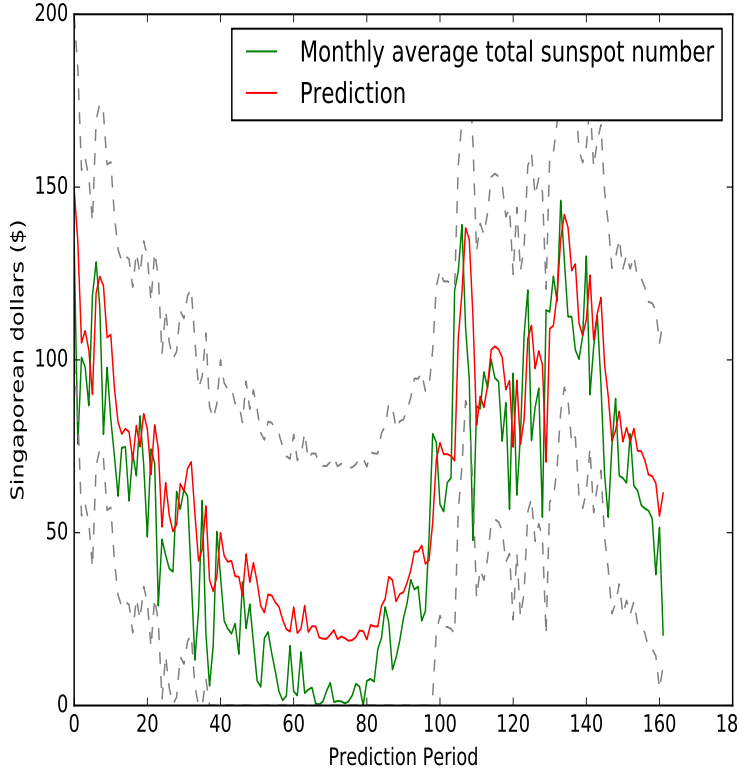


Figure 10.3: Actual and predicted values for multi time step GRU

Additional Resources to Check Out

- Junyoung Chung has put together a very nice comparison of the GRU, LSTM and simple RNN using music, and speech signal modeling. See Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling."arXiv preprint arXiv:1412.3555 (2014).

Chapter 11

Forecasting Multiple Outputs

So far, we have focused on building models where there is one response variable. This is akin to the traditional approach of univariate time series modelling. In many circumstances observations are take simultaneously on two or more time series. For example, in hydrology we might observe precipitation, discharge, soil humidity and water temperature at the same site for the same sequence of time points. In financial economics, we may be interested in modelling the opening and closing price of IBM, Microsoft and Apple over a several weeks or months. Multivariate data such as this can be easily handled in a neural network.

In this chapter, we build a RNN to predict the historical volatility of the FTSE100 stock market index and the Dow Jones Industrial index. The FTSE is an index of the 100 companies listed on the London Stock Exchange with the highest market capitalization. The Dow Jones Industrial Average is the average value of 30 large, industrial stocks traded on the New York Stock Exchange and the NASDAQ.

Figure 11.1 shows the historical performance of both indices from November 1995 to May 2001. Although the levels and precise shape of the indices are different, they appear to have

similar underlying time series dynamics.

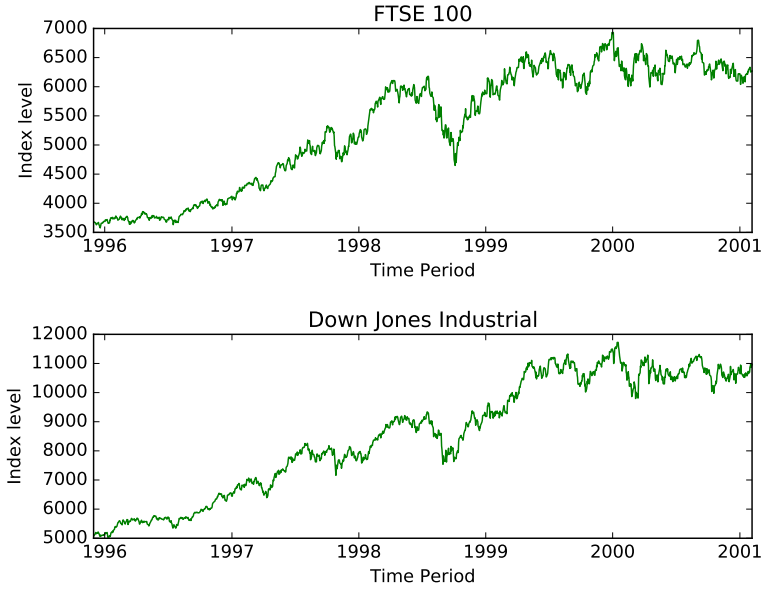


Figure 11.1: Historical performance of stock market indices

Working with Zipped Files

For our analysis, we will use historical data stored in a zip file on the internet. The `zipfile` module is used to manipulate ZIP archive files. Here is how to download and unzip the data:

```
import numpy as np
import pandas as pd
import urllib
import zipfile
url="http://www.economicswebinstitute.org/
   data/stockindexes.zip"
loc="C:\\Data\\stockindexes.zip"
dest_location="C:\\Data"
```

```
unzip = zipfile.ZipFile(loc, 'r')
unzip.extractall(dest_location)
unzip.close()
```

Extraction of Spreadsheet data

The extracted file is a `xls` spreadsheet which can be handled by the pandas `ExcelFile` method. First, read `stockindexes.xls` file into the object `Excel_file`:

```
loc= "C:\\Datastockindexes.xls"
Excel_file = pd.ExcelFile(loc)
print Excel_file.sheet_names
 [u'Description', u'Dow Jones Industrial',
    u'S&P500', u'NIKKEI 300', u'Dax30', u'
    CAC40', u'Swiss Market-Price Index', u'
    Mib30', u'IBEX 35I', u'Bel20', u'
    FTSE100']
```

The spreadsheet contains several worksheets with historical data on various stock markets. We want the `'FTSE100'` and `'Dow Jones Industrial'` worksheets:

```
ftse_data = Excel_file.parse('FTSE100')
dj_data = Excel_file.parse('Dow Jones
   Industrial')
```

Now, look at the first few observations contained in `ftse100`:

```
print ftse_data.head()
                  Start      1995-11-30 00:00:00
0                   End      2000-02-18 00:00:00
1             Frequency                        D
2                  Name  FTSE 100 - PRICE INDEX
3                  Code                  FTSE100
4  1995-11-30 00:00:00                   3664.3
```

The observations don't start until the first date "`1995-11-30 0 0 : 0 0 : 0 0`" when the FTSE 1000 had the value 3664.3.

A similar pattern is found with the Dow Jones data:

```
print dj_data.head()
                  Start                          1995-11-30
                     00:00:00
0                   End                          2000-02-18
    00:00:00
1             Frequency
                                                        D
2                  Name  DOW JONES INDUSTRIALS - PRICE
    INDEX
3                  Code
    DJINDUS
4  1995-11-30  00:00:00
    5074.49
```

In this case the actual data we are interested in begins with a value of 5074.49.

Let's transfer the price index data into the Python objects `ftse100` and `dj`:

```
ftse100= ftse_data.iloc[4:1357,1]
dj= dj_data.iloc[4:1357,1]
```

Checking Data Values

Now, take a quick look at the data in both objects:

```
print ftse100.head()
4    3664.3
5    3680.4
6    3669.7
7    3664.2
8    3662.8

print ftse100.tail()
1352    6334.53
1353    6297.53
1354    6251.83
1355    6256.43
1356    6269.21
```

```
print dj.head()
4    5074.49
5    5087.13
6    5139.52
7    5177.45
8    5199.13

print dj.tail()
1352    10881.2
1353    10887.4
1354    10983.6
1355    10864.1
1356    10965.9
```

How to Work with Multiple Targets

The goal is to predict the 30 day historical volatility (standard deviation) of the daily price change. Our first step is to create the target variables. Let's begin by concatenating the price series into one Python object - `yt`:

```
yt=pd.concat([ftse100,dj], axis=1)
print yt.head()
  1995-11-30 1995-11-30
4     3664.3    5074.49
5     3680.4    5087.13
6     3669.7    5139.52
7     3664.2    5177.45
8     3662.8    5199.13
```

Notice that the column names are actually dates and the index begins at 4. Let's fix both issues:

```
yt = yt.reset_index(drop=True)
yt.columns = ['ftse100', 'dj']
print yt.head()
```

```
  ftse100        dj
0   3664.3   5074.49
1   3680.4   5087.13
2   3669.7   5139.52
3   3664.2   5177.45
4   3662.8   5199.13
```

That looks much better.

The next step is to convert the price levels into daily price percent changes and calculate the 30 day rolling standard deviation. Here is how to do that:

```
yt=yt.pct_change(1)
win =30
vol_t=yt.rolling(window=win,center=True).
   std()
```

Figure 11.2 shows the resultant time series for both stock markets.

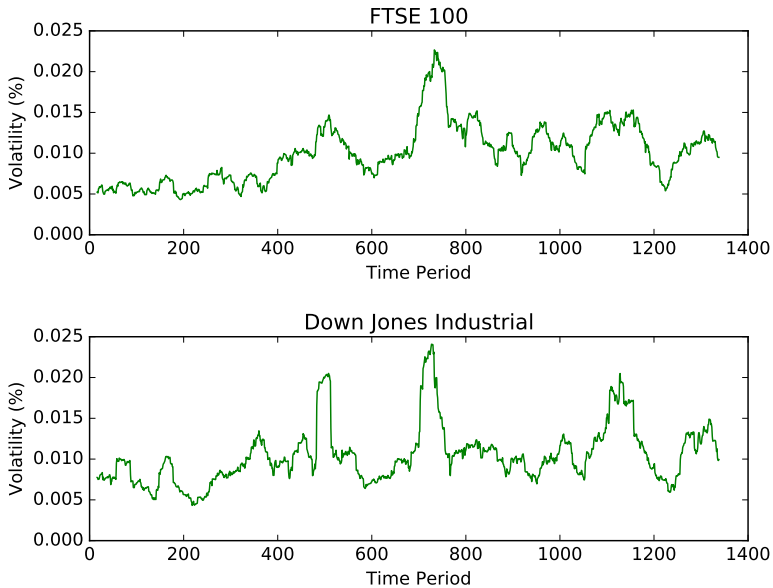


Figure 11.2: 30 day historical volatility

Creation of Hand Crafted Features

The design of a deep learning system is usually considered consisting of two major steps. The first involves preprocessing, and feature extraction; and the second step involves model building and classification or prediction.

Hand crafting features is both art and science. To spice things up a little, we add five hand crafted features. Each constructed along the following lines:

$$\left[\frac{vol_{t-1}}{vol_{t-k}}\right] \times vol_{t-1},\ k = 1, 2, 3, 4, 5$$

In other words, at time `t` we use yesterday's volatility adjusted by the ratio of it to recent past values:

```
x1=np.log((vol_t.shift(1)/vol_t.shift(2))*vol_t.shift(1))
x2=np.log((vol_t.shift(1)/vol_t.shift(3))*vol_t.shift(1))
x3=np.log((vol_t.shift(1)/vol_t.shift(4))*vol_t.shift(1))
x4=np.log((vol_t.shift(1)/vol_t.shift(5))*vol_t.shift(1))
x5=np.log((vol_t.shift(1)/vol_t.shift(6))*vol_t.shift(1))
```

These five features are combined into the object `data`, columns given a name/ header, and missing values (from the calculation of the 30 day volatility) dropped:

```
data=pd.concat([vol_t,x1,x2,x3,x4,x5], axis
   =1)
data.columns = ['ftse_t', 'dj_t',
        'ftse_t-1','dj_t-1',
        'ftse_t-2','dj_t-2',
        'ftse_t-3','dj_t-3',
        'ftse_t-4','dj_t-4',
        'ftse_t-5','dj_t-5']
data = data.dropna()
```

Target and Features in One Place

Finally, to be consistent with previous analysis, we can create the target variable `y` and the feature data x:

```
cols_y=['ftse_t', 'dj_t']
y=data[cols_y]
cols=['ftse_t-1','dj_t-1',
        'ftse_t-2','dj_t-2',
        'ftse_t-3','dj_t-3',
        'ftse_t-4','dj_t-4',
        'ftse_t-5','dj_t-5'
        ]
x=data[cols]
```

Scaling Data

The data is scaled to lie in the 0 to 1 range using `MinMaxScaler`:

```
from sklearn import preprocessing
num_attrib=10
scaler_x = preprocessing.MinMaxScaler(
   feature_range=(-1, 1))

x = np.array(x).reshape((len(x),num_attrib
   ))
x = scaler_x.fit_transform(x)

num_response=2

scaler_y = preprocessing.MinMaxScaler(
   feature_range=(0, 1))

y = np.array(y).reshape((len(y),
   num_response))
y = scaler_y.fit_transform(y)
```

All the above is similar to that discussed in earlier chapters. However, the number of attributes is equal to 10. This is because we have 5 hand crafted features $*$ 2 response variables.

Train and Test Sets

The train and test sets are created as follows:

```
train_end = 1131
data_end =len(y)
x_train=x[0:train_end,]
x_test=x[train_end+1:data_end,]
y_train=y[0:train_end]
y_test=y[train_end+1:data_end]
x_train = np.reshape(x_train, (x_train.
   shape[0], 1, x_train.shape[1]))
x_test = np.reshape(x_test, (x_test.shape
   [0], 1, x_test.shape[1]))

print "Shape of x_train is ",x_train.shape
Shape of x_train is  (1131, 1, 10)

print "Shape of x_test is ",x_test.shape
Shape of x_test is  (185, 1, 10)
```

For this example, we follow the approach outlined on page 151 and re-frame the problem using multiple time-steps (10 in this case). Therefore, the test set has 1131 examples on 10 time steps.

Model Specification and Fit

The difficult work has been done. Now the data is in an appropriate format, specifying and fitting the model follows the steps we have already worked through in the earlier chapters. First, load the appropriate libraries:

```
from keras.models import Sequential
from keras.layers.core import Dense,
   Activation
from keras.layers.recurrent import
   SimpleRNN
from keras.optimizers import SGD
```

The model is a simple RNN built using the Keras package. We will specify 10 nodes in the hidden layer, use Stochastic Gradient Descent with a learning rate of 0.01 and a momentum of 0.90, with the model run over a single epoch:

```
seed=2016
num_epochs=20
np.random.seed(seed)
fit1 = Sequential()
fit1.add(SimpleRNN(output_dim=10,
        activation='sigmoid',
        input_shape=(1, num_attrib)))
fit1.add(Dense(output_dim=num_response,
   activation='linear'))
sgd = SGD(lr=0.01,  momentum=0.90, nesterov
   =True)
fit1.compile(loss='mean_squared_error',
   optimizer=sgd)
fit1.fit(x_train, y_train, batch_size=1,
   nb_epoch=num_epochs)
```

The train and test set performance are:

```
score_train = fit1.evaluate(x_train,
   y_train,batch_size=1)
score_test = fit1.evaluate(x_test, y_test,
   batch_size=1)

print "in train MSE = ", round(score_train
   ,5)
in train MSE =  0.00158
```

```
print "in test MSE = ", round(score_test,5)
in test MSE =   0.00114
```

Now, to calculate the predictions and transform them back to their original scale:

```
pred1=fit1.predict(x_test)0.00114387514427
pred1 = scaler_y.inverse_transform(np.array
   (pred1).reshape((len(pred1), 2)))
```

Figure 11.3 and Figure 11.4 plot the actual and predicted values. For both time series, the model closely mirrors the dynamics of the actual series.

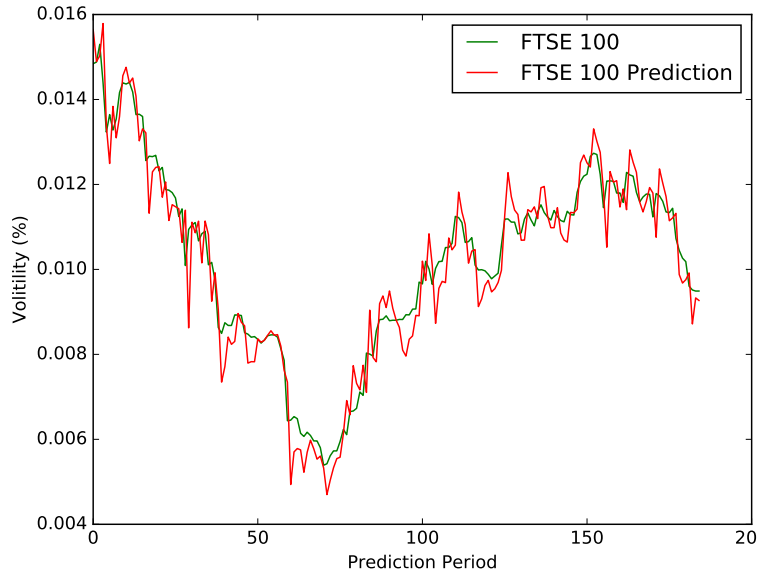


Figure 11.3: FTSE 100 Volatility model actual and predicted values

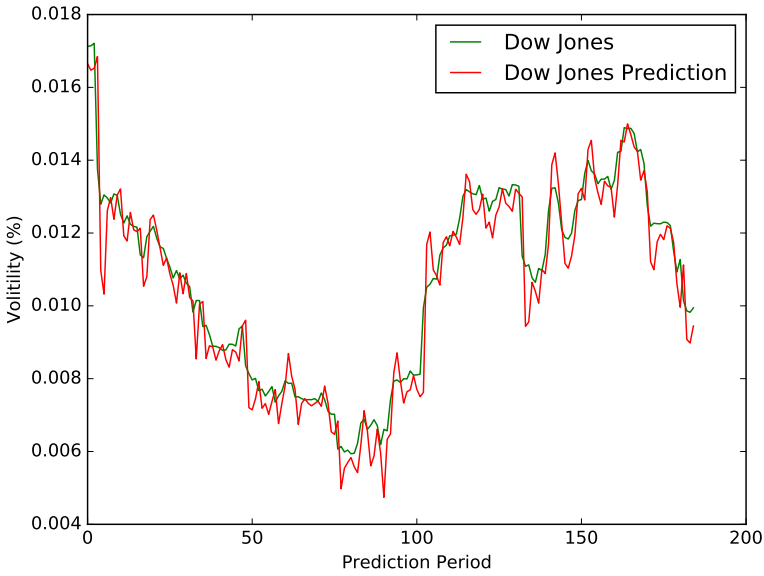


Figure 11.4: Dow Jones Volatility model actual and predicted values

Additional Resources to Check Out

- Tamal Datta Chaudhuri has written a nice easy to read paper explaining how he used a neural network to predict stock market volatility. See Chaudhuri, Tamal Datta, and Indranil Ghosh. "Forecasting Volatility in Indian Stock Market using Artificial Neural Network with Multiple Inputs and Outputs." arXiv preprint arXiv:1604.05008 (2016).

- For a highly informative discussion of recurrent neural networks for multivariate time series with missing values look at the article by Zhengping Che. It is a little technical, but a thoroughly good read. Che, Zhengping, et al. "Recurrent Neural Networks for Multivariate Time Series

with Missing Values." arXiv preprint arXiv:1606.01865 (2016).

Chapter 12

Strategies to Build Superior Models

Let's face it, building neural networks for time series modeling requires a large dose of patience. When I coded my very first multi-layer neural network, I had to wait around three and a half days per run! And by today's standards my dataset was tiny. In fact, every model we have developed in this book runs in a matter of minutes rather than days.

As you continue to develop your applied skills in deep learning for time series modeling, you will need some tricks in your back pocket to boost performance. In this chapter we outline a number of the very best ideas that can help you out of a sticky situation. But remember, paraphrasing the disclaimer I often encountered in my days managing investment portfolios, "*Deep learning tricks come with substantial risk of failure, and are not suitable for every situation!*"

UK Unemployment Rate Data

Let's load up on some data, in this case the economic data we encountered in chapter 8. Recall, we saved two files `economic_x` which contain the attributes, and `economic_y`

which contained the target variable - the annual unemployment rate:

```
import numpy as np
import pandas as pd
loc= "C:\\Users\\Data\\economic_x.csv"
x = pd.read_csv(loc)
x= x.drop( x.columns [[0]] , axis =1)
loc= "C:\\Data\\\\economic_y.csv"
y = pd.read_csv(loc, header=None)
y= y.drop( y.columns [[0]] , axis =1)
```

Remember to replace `loc` with the location to which you previously saved `economic_x` , and `economic_y`.

A Quick Peek

To steady our nerves a little (and to refresh our memory), look right now at both `x` and `y`:

```
print x.head()
GDP_trend          debt   bank_rate   inflation
0    0.079944   121.127584         5.0   11.668484
1   -0.406113   120.856960         7.0    0.488281
2    2.193677   117.024347         6.0   -0.485909
3    0.190602   117.183618         8.0   -3.613281
4   -1.505673   120.018119         2.5   -8.409321

print y.head()
           1
0   3.790930
1   3.572757
2   4.008832
3   5.309585
4   3.325983
```

Oh, we did not save y with a column name! Well, that is OK, we know that it represents the unemployment rate.

Adjust the Data Scale

Now, scale the attributes and target to lie in the range 0 to 1:

```
from sklearn import preprocessing
scaler_x = preprocessing.MinMaxScaler(
    feature_range=(0, 1))
x = np.array(x).reshape((len(x),4 ))
x = scaler_x.fit_transform(x)

scaler_y = preprocessing.MinMaxScaler(
    feature_range=(0, 1))
y = np.array(y).reshape((len(y), 1))
y = scaler_y.fit_transform(y)
```

Create Train and Test Set

Finally, let's create the train and test sets for both attributes and target variable:

```
x_train=x[0:136,]
x_test=x[137:161,]
y_train=y[0:136]
y_test=y[137:161]
```

Limitations of the Sigmoid Activation Function

One of the limitations of the sigmoid function is its gradient becomes increasingly smaller as `x` increases or decreases. This is a problem if we are using gradient descent or similar methods and is known as the vanishing gradient problem (also see page 138 where we discuss this in the context of a RNN); as the gradient gets smaller a change in the parameter's value results in a very small change in the network's output. This slows down learning to a crawl.

The slow down is amplified as the number of layers increase because the gradients of the network's output with respect to the parameters in the early layers can become extremely small. This happens because the sigmoid function takes a real-valued number and "squashes" it into values between zero and one; in other words, it maps the real number line onto the relatively small range of [0, 1].

In particular, large negative numbers become 0 and large positive numbers become 1. This implies that large areas of the input space are mapped into an extremely small range. The problem is that in these regions of the input space, even a very large change in the input will only produce a tiny change in the output and hence the gradient is small.

NOTE... ✍

The problem in a nutshell is that when the sigmoid activation function saturates at either 1 or 0, the gradient at these regions is very shallow, almost zero.

One Activation Function You Need to Add to Your Deep Learning Toolkit

The vanishing gradient problem can be avoided by using activation functions which do not squash the input space into a narrow range. In fact, I have found that swapping out a "squashing" activation function, such as the sigmoid, for an alternative "non-squashing" function can often lead to a dramatic improvement in performance of a deep neural network.

One of my favorites to try is the Rectified linear unit (relu). It is defined as:

$$f(x) = \max(0, x)$$

where x is the input to a neuron. It performs a threshold operation, where any input value less than zero is set to zero, see Figure 12.1.

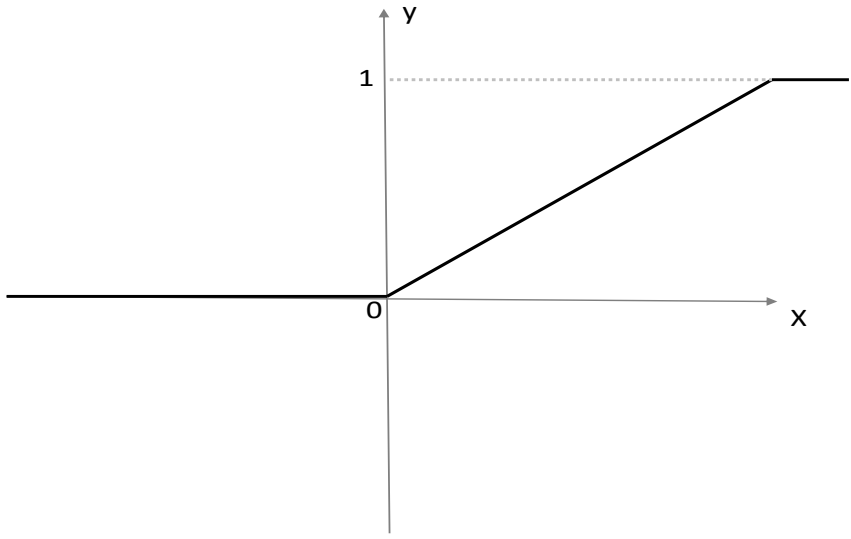


Figure 12.1: ReLU activation function

This activation function has proved popular in deep learning models because significant improvements of classification rates have been reported for speech recognition and computer vision tasks. It only permits activation if a neurons output is positive; and allows the network to compute much faster than a network with sigmoid activation functions because it is simply a max operation. It also encourages sparsity of the neural network because when initialized randomly approximately half of the neurons in the entire network will be set to zero.

Relu with Keras

Let's try out the relu activation function. We build the model with Keras. It has two hidden layers, with 40 and 20 nodes in

the first and second hidden layers respectively:

```
from keras.models import Sequential
from keras.layers import Dense
seed = 2016
np.random.seed(seed)
fit1 = Sequential()
fit1.add(Dense(40, input_dim=4, init='
   uniform', activation='relu'))
fit1.add(Dense(20, init='uniform',
   activation='relu'))
fit1.add(Dense(1, init='normal'))
```

Next, the model is run over 3,000 epochs with a batch size of 10:

```
epochs =3000
fit1.compile(loss='mean_squared_error',
   optimizer='adam')
fit1.fit(x_train, y_train, nb_epoch=epochs
   , batch_size=10)
```

We use the `'adam'` optimizer. Adam, is a fairly new algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. It was introduced a few years ago, and often gives very good results with empirical data.

NOTE... ✍

You can think of a neuron as being "active" (or as "firing") if its output value is close to 1; and "inactive" if its output value is close to 0. Sparsity constrains the neurons to be inactive most of the time. It often leads to better generalization performance (see page 176).

Viewing Predictive Performance

Figure 12.1 plots the predicted and actual values, alongside the comfort interval/ tolerance range. The model does a nice job at capturing the underlying trend. Indeed, over the forecast period, the majority of the actual observations are well within the specified comfort interval. The predictiveness is impressive for such a small model!

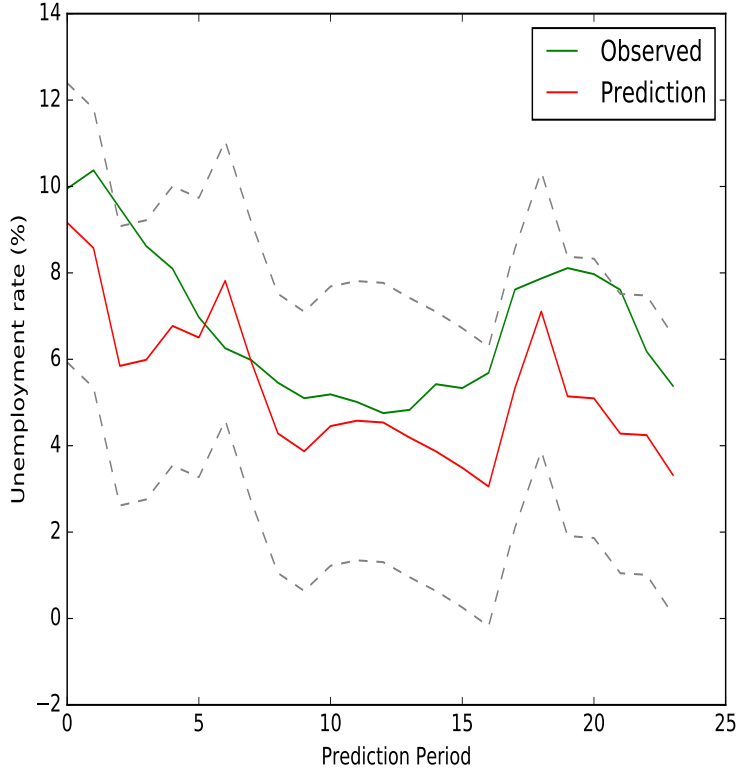


Figure 12.2: Actual and predicted values using relu activation function

Try This Simple Idea to Enhance Success

The image of the derelict who drops out of school, and wastes her life away in the shady recesses of the big city is firmly imprinted in our culture. Parents warn their children to concentrate on school lessons, pass the exams and get a good job. Whatever they do, don't drop out. If you do, you will find yourself living in the dark recesses of the big city. When it comes to going to school, we are advised against dropping out because of the fear that it will have a deleterious impact on our future.

The funny thing is, we also celebrate those celebrities and business tycoons who dropped out. These individuals went onto enormous success. Heights which would have been impossible had they stayed in school! In fact, I cannot think of an area in industry, education, science, politics or religion where individuals who dropped out have not risen to amazing heights of success. I'd hazard a guess the software or hardware you are using right now is direct consequence of one such drop out.

The Power of the Drop Out

Whilst dropping out is both celebrated and derided by our culture, it appears to also have some upside and downside in relation to DNNs. Borrowing from the idea that dropping out might boost DNN performance, suppose we ignore a random number of neurons during each training round; the process of randomly omitting a fraction of the hidden neurons is called dropout, see Figure 12.3.

To state this idea a little more formally: for each training case, each hidden neuron is randomly omitted from the network with a probability of `p`. Since the neurons are selected at random, different combinations of neurons will be selected for each training instance.

The idea is very simple and results in a weak learning model at each epoch. Weak models have low predictive power

by themselves, however the predictions of many weak models can be weighted and combined to produce models with much 'stronger' predictive power.

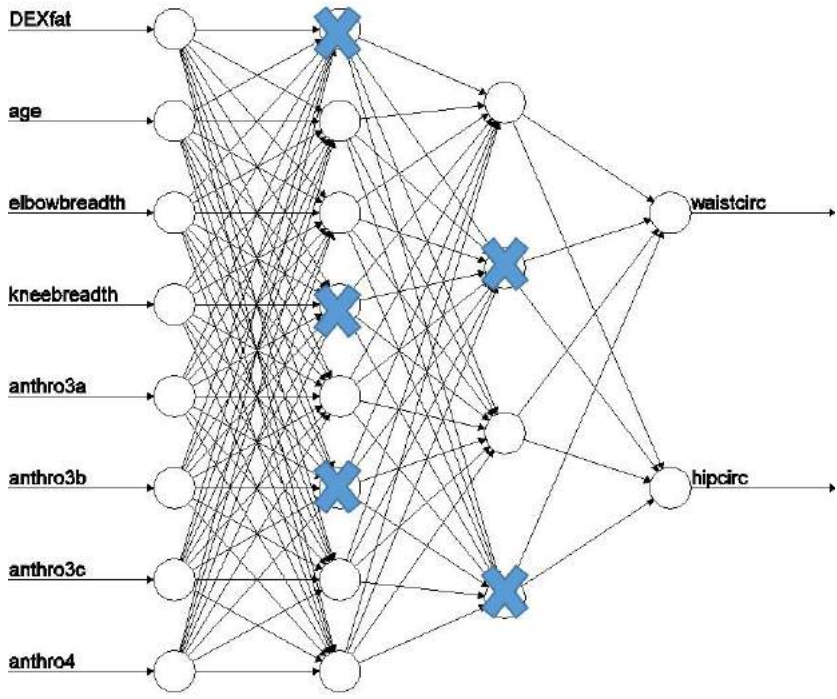


Figure 12.3: DNN Dropout

Similarity

In fact, dropout is very similar to the idea behind the machine learning technique of bagging. Bagging is a type of model averaging approach where a majority vote is taken from classifiers trained on bootstrapped samples of the training data. In fact, you can view dropout as implicit bagging of several neural network models.

Dropout can therefore be regarded as an efficient way to perform model averaging across many neural networks.

Co-adapation

Much of the power of DNNs comes from the fact that each neuron operates as an independent feature detector. However, in actual practice it is common for two or more neurons to begin to detect the same feature repeatedly. This is called co-adaptation. It implies the DNN is not utilizing its full capacity efficiently, in effect wasting computational resources calculating the activation for redundant neurons that are all doing the same thing.

In many ways co-adaptation, is similar to the idea of collinearity in linear regression, where two or more covariates are highly correlated. It implies the covariates contain similar information; in particular, that one covariate can be linearly predicted from the others with a very small error. In essence, one or more of the covariates are statistically redundant. Collinearity can be resolved by dropping one or more of the covariates from the model.

Dropout discourages co-adaptations of hidden neurons by dropping out a fixed fraction of the activation of the neurons during the feed forward phase of training. Dropout can also be applied to inputs. In this case, the algorithm randomly ignores a fixed proportion of input attributes.

A Lesson

One of life's lessons is that dropping out is not necessarily ruinous to future performance, but neither is it a guarantee of future success. The same is true for dropout in DNNs; there is no absolute guarantee that it will enhance performance, but it is often worth a try.

Keep the following three points in mind as you develop your own DNN models:

1. Dropout can reduce the likelihood of co-adaptation in noisy samples by creating multiple paths to correct classification throughout the DNN.

2. The larger the dropout fraction the more noise is introduced during training; this slows down learning,

3. Dropout appears to offer the most benefit on very large DNN models.

Adding Drop Out with Keras

Drop out can be added to our previous model via the `Dropout` function. To illustrate how you might use it, we add 5% drop out to each of the hidden layers:

```
from keras.layers import Dropout
dropout1 = 0.05
dropout2= 0.05
fit2 = Sequential()
fit2.add(Dense(40, input_dim=4, init='
   uniform', activation='relu'))
fit2.add(Dropout(dropout1))
fit2.add(Dense(20, init='uniform',
   activation='relu'))
fit2.add(Dropout(dropout2))
fit2.add(Dense(1, init='normal'))
fit2.compile(loss='mean_squared_error',
   optimizer='adam')
fit2.fit(x_train, y_train, nb_epoch=epochs,
    batch_size=10)
```

Notice, to add drop out we simply add another line after the layer specification.

Figure 12.3 plots the predicted and actual values. While a few of the early and later observations lie above the upper comfort interval, the overall prediction dynamics, mirrors closely that of the underlying target variable.

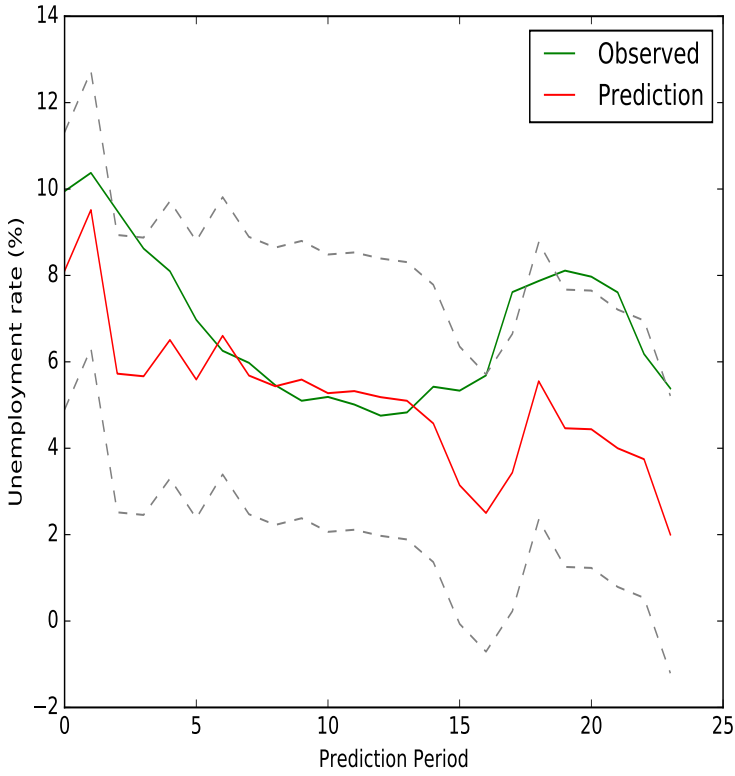


Figure 12.4: Actual and predicted values with drop out.

A Simple Plan for Early Stopping

When I was in elementary school we had regular and frequent outdoor playtime's as part of our education. We usually had playtime mid-morning to whip up our appetite prior to lunch, after lunch to aid the digestive system, and on warm sunny days, our teachers would unleash us outside for an afternoon playtime - to burn up any unused energy before we were packed off home. I loved school!

The only thing that would shorten our frequent outdoor

adventures was the rain. If it rained (and it rains a lot in England) the early stopping bell would ring, the fun would be terminated and the lessons begin. The funny thing was, even after a shortened play time my ability to focus and concentrate on my school work was enhanced. I guess my teachers had an intuitive understanding of the link between physical activity, mental focus, mood and performance. To this day I go outside for a brisk walk to jolt my mental faculties into action.

The analogy with school playtime's is that we should be willing to stop DNN training early if that makes it easier to extract acceptable performance on the testing sample. This is the idea behind early stopping where the sample is divided into three sets. A training set, a validation set and a testing set. The train set is used to train the DNN. The training error is usually a monotonic function, that decreases with every iteration. Figure 6.6 illustrates this situation. The error falls rapidly during the first 5 epochs. It then declines at a much shallower rate to one hundred epochs where it appears to level off to a constant value.

A validation set is used to monitor the performance of a model. The validation error usually falls sharply during the early stages as the network rapidly learns the functional form, but then increases, indicating the model is starting to over-fit. In early stopping, training is stopped at the lowest error achieved on the validation set. Early stopping has proven to be highly effective in reducing over-fitting for a wide variety of neural network applications. It is worth a try by you.

Using Early Stopping in Keras

The first step is to specify the model. Keras supports early stopping through an `EarlyStopping` callback class:

```
from keras.callbacks import EarlyStopping
fit3 = Sequential()
fit3.add(Dense(40, input_dim=4, init='
   uniform', activation='relu'))
```

```
fit3.add(Dense(20, init='uniform',
   activation='relu'))
fit3.add(Dense(1, init='normal'))
fit3.compile(loss='mean_squared_error',
   optimizer='adam')
```

NOTE... ✍

Callbacks are methods that are called after each epoch of training upon supplying a callbacks parameter to the `fit` method of a model. They get a "view" on internal states and statistics of a model during training.

Validation Set

We use 24 examples for validation set:

```
y_valid=y_train[112:136]
x_valid=x_train[112:136]
```

And, now fit the model:

```
fit3.fit(x_train, y_train, nb_epoch=epochs,
    batch_size=10,validation_data=(x_valid,
   y_valid),
          callbacks=[EarlyStopping(monitor='
             val_loss',patience=100, verbose
             =2,mode='auto')])
```

In early stopping, you stop training once the validation loss hasn't decreased for a fixed number of epochs. The number of epochs is specified in a parameter known as `patience`. In the above code, we set it to the value 100.

Figure 12.5 shows the predicted and actual values. Again, the model performs fairly well with the majority of actual observations contained within the comfort interval.

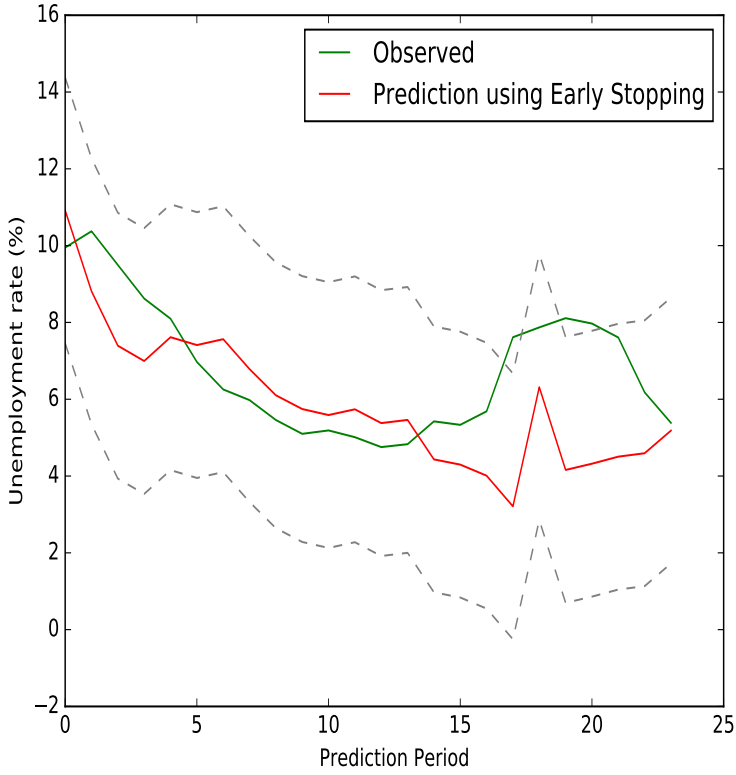


Figure 12.5: Actual and predicted values with early stopping

NOTE... ✍

To see the actual fitted model weights use:

```
fit1.get_weights()
```

Additional Resources to Check Out

- For more details on the adam optimizer, see the amazing paper by Diederik Kingma, Jimmy Ba - Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).
- Lutz Prechelt has written a well thought out article on the use and abuse of early stopping. Prechelt, Lutz. "Early stopping—but when?." Neural Networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. 53-67.
- Be sure to read the wonderful article Dropout: a simple way to prevent neural networks from overfitting. See Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." Journal of Machine Learning Research 15.1 (2014): 1929-1958.

Congratulations!

You made it to the end. Here are three things you can do next.

1. Pick up your FREE copy of **21 Tips For Data Science Success with Python** at *http://www.auscov.com*

2. Gift a copy of this book to your friends, co-workers, teammates or your entire organization.

3. If you found this book useful and have a moment to spare, I would really appreciate a short review. Your help in spreading the word is gratefully received.

I've spoken to thousands of people over the past few years. I'd love to hear your experiences using the ideas in this book. Contact me with your stories, questions and suggestions at *Info@NigelDLewis.com*.

Good luck!

P.S. Thanks for allowing me to partner with you on your data science journey.

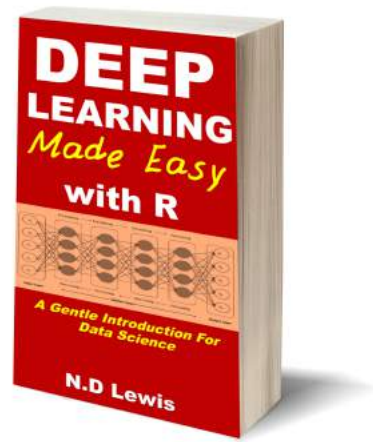
Index

OTHER BOOKS YOU WILL ALSO ENJOY

FINALLY... The Ultimate Cheat Sheet For Deep Learning Mastery

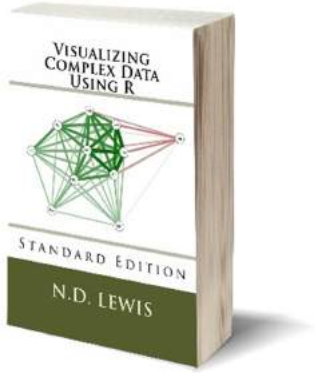
If you want to join the ranks of today's top data scientists take advantage of this valuable book. It will help you get started. It reveals how deep learning models work, and takes you under the hood with an easy to follow process showing you how to build them faster than you imagined possible using the powerful, free R predictive analytics package.

Buy the book today. Your next big breakthrough using deep learning is only a page away!



ORDER YOUR COPY TODAY!

"They Laughed As They Gave Me The Data To Analyze...But Then They Saw My Charts!"



Wish you had fresh ways to present data, explore relationships, visualize your data and break free from mundane charts and diagrams?

Visualizing complex relationships with ease using R begins here.

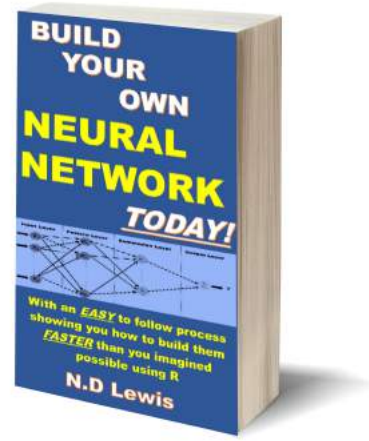
In this book you will find innovative ideas to unlock the relationships in your own data and create killer visuals to help you transform your next presentation from good to great.

Visualizing Complex Data Using R

ORDER YOUR COPY TODAY!

ANNOUNCING...The Fast & Easy Way to Master Neural Networks

This rich, fascinating, accessible hands on guide, puts neural networks firmly into the hands of the practitioner. It reveals how they work, and takes you under the hood with an easy to follow process showing you how to build them faster than you imagined possible using the powerful, free R predictive analytics package.



Here are some of the neural network models you will build:

- Multilayer Perceptrons
- Probabilistic Neural Networks
- Generalized Regression Neural Networks
- Recurrent Neural Networks

Buy the book today and master neural networks the fast & easy way!

ORDER YOUR COPY TODAY!

Write your notes here: