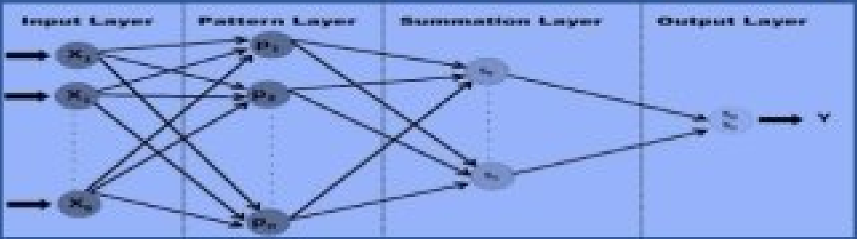
BUILD YOUR OWN NEURAL NETWORK TODAY!



With an ***EASY*** to follow process showing you how to build them ***FASTER*** than you imagined possible using R

N.D Lewis

BUILD YOUR OWN NEURAL NETWORK TODAY!

With an EASY to follow process showing you how to build them FASTER than you imagined possible using R

Dr. N.D. Lewis

Ordering Information: Quantity sales. Special discounts are available on quantity purchases by corporations, associations, and others. For details, email: info@NigelDLewis.com

Image photography by Deanna Lewis

ISBN: 978-1519101235
ISBN: 1519101236

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 daughter Deanna, for taking hundreds of photographs for this book and my website.

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

About This Book

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. Everything you need to get started is contained within this book. It is your detailed, practical, tactical hands on guide. To accelerate your success, it contains exercises with fully worked solutions also provided. Once you have mastered the process, it will be easy for you to translate your knowledge into other powerful applications. A book for everyone interested in machine learning, predictive analytics, neural networks and decision science. Here is what it can do for you:

- SAVE TIME: Imagine having at your fingertips easy access to the very best neural network models without getting bogged down in mathematical details. In this book, you'll learn fast effective ways to build powerful neural network models easily using R.

- LEARN EASILY: **Build Your Own Neural Network TODAY!** Contains an easy to follow process showing you how to build the most successful neural networks used for learning from data; use this guide and build them easily and quickly.

- BOOST PRODUCTIVITY: Bestselling author and data scientist Dr. N.D. Lewis will show you how to build neural network models in less time than you ever imagined possible! Even if you're a busy professional, a student or hobbyist with little time, you will rapidly enhance your knowledge.

- EFFORTLESS SUCCESS: By spending as little as 10 minutes a day working through the dozens of real world examples, illustrations, practitioner tips and notes, you'll be able to make giant leaps forward in your knowledge, broaden your skill-set and generate new ideas for your own personal use.

- ELIMINATE ANXIETY: Forget trying to master every single mathematical detail, instead your goal is to simply follow the process using real data that only takes about 5 to 15 minutes to complete. Within this process is a series of actions by which the neural network model is explained and constructed. All you have to do is follow the process. It is your checklist for use and reuse.

For people interested in statistics, machine learning, data analysis, data mining, and future hands-on practitioners seeking a career in the field, it sets a strong foundation, delivers the prerequisite knowledge, and whets your appetite for more.

Here are some of the neural network models you will build:

- Multi-layer Perceptrons
- Probabilistic Neural Networks
- Generalized Regression Neural Networks
- Recurrent Neural Networks

Buy the book today. Your next big breakthrough using neural networks is only a page away!

Preface

No matter who you are, no matter where you are from, no matter your background or schooling, you have the ability to use the techniques outlined in this book. With dedicated focus, persistence and the right guide I personally believe neural networks can be successfully used in the hands of anyone.

As a former working Economist, discovering neural networks was like getting a glimpse of the view from the fiftieth floor of a business skyscraper. There is a whole other world out there - a world of modeling possibilities far superior to the 19th century analytic techniques of my former discipline. Several decades have passed since my discovery, and I still remain entranced by the shear power of the methods we will discuss in this text.

They possess many advantages such as the ability to learn from data, classification capabilities and generalization for situations not contained in original training data set, computationally fastness once trained due to parallel processing, and tolerance of noisy data. These advantages have seen neural networks successfully applied in many real-word problems, including: speech recognition, medical diagnosis, image computing, process control and economic forecasting.

I'd like to share with you some of what I have seen, to take you to the fiftieth floor with me, so you too can glimpse the view. I want you to be the person that creates the next big breakthrough in modeling for your business or research or personal pleasure. The neural networks discussed in the following pages may just be the tool that does the job for you.

This book came out of the desire to put the powerful technology of neural networks into the hands of the everyday practitioner. The material is therefore designed to be used by the individual whose primary focus is on building predictive analytic models in order to drive results.

In writing this text my intention was to collect together in a single place practical tips, suggestions and a workable plan for anyone who can use a word processor to build neural network models easily. My focus is solely on those techniques, ideas and strategies that have been proven to work and can be quickly digested in the minimal amount of time.

On numerous occasions, individuals in a wide variety of disciplines and industries, have asked "how can I quickly understand and build a neural network?" The answer used to involve reading complex mathematical texts and then programming complicated formulas in languages such as C, C++ and Java.

With the rise of R, predictive analytics is now easier than ever. This book is designed to give you rapid access to the most popular types of neural networks. It shows you, step by step, how to build each type of model in the free and popular R statistical package. Examples are clearly described and can be typed directly into R as printed on the page.

The least stimulating aspect of the subject for the practitioner is the mechanics of calculation. Although many of the horrors of this topic are necessary for the theoretician, they are of minimal importance to the practitioner and can be eliminated almost entirely by the use of the modern computer. It is inevitable that a few fragments remain, these are fully discussed in the course of this text. However, as this is a hands on, role up your sleeves and apply the ideas to real data book, I do not spend much time dealing with the minutiae of algorithmic matters, proving theorems, discussing lemmas or providing proofs.

Those of us who deal with data are primarily interested in extracting meaningful structure. For this reason, it is always a good idea to study how other users and researchers have applied a technique in actual practice. This is primarily because practice often differs substantially from the classroom or theoretical text books. To this end and to accelerate your progress, actual real world applications of neural network use are given throughout this text.

These illustrative applications cover a vast range of disciplines and are supplemented by the actual case studies which take you step by step through the process of building a neural network. I have also provided detailed references for further personal study in the notes section at the end of each chapter. In keeping with the zeitgeist of R, copies of the vast majority of applied articles referenced in this text are available for free.

New users to R can use this book easily and without any prior knowledge. This is best achieved by typing in the examples as they are given and reading the comments which follow. Copies of R and free tutorial guides for beginners can be downloaded at `https://www.r-project.org/`. If you are totally new to R take a look at the amazing tutorials at `http://cran.r-project.org/other-docs.html`; they do a great job introducing R to the novice.

The master painter Vincent van Gough once said "*Great things are not done by impulse, but by a series of small things brought together.*" This instruction book is your step by step, detailed, practical, tactical guide to building and deploying neural networks. It is an enlarged, revised, and updated collection of my previous works on the subject. I've condensed into this volume the best practical ideas available.

The material you are about to read is based on my personal experience, articles I've written, hundreds of scholarly articles I've read over the years, experimentation some successful some failed, conversations I've had with data scientists in various fields and feedback I've received from numerous presentations to people just like you.

I have found, over and over, that an individual who has exposure to a broad range of modeling tools and applications will run circles around the narrowly focused genius who has only been exposed to the tools of their particular discipline. Gaining knowledge of how to build and apply neural network models will add considerably to your own personal toolkit.

Greek philosopher Epicurus once said "I write this not for the many, but for you; each of us is enough of an audience for the other." Although the ideas in this book reach out to thousands of individuals, I've tried to keep Epicurus's principle in mind–to have each page you read give meaning to just one person - YOU.

I invite you to put what you read in these pages into action. To help you do that, I've created **"12 Resources to Supercharge Your Productivity in R**", it is yours for **FREE**. Simply go to *http://www.auscov.com* and download it now. It's my gift to you. It shares with you 12 of the very best resources you can use to boost your productivity in R.

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*.

Now, it's your turn!

Dr. Nigel D. Lewis

*P.S. Don't forget to sign-up for your **FREE** copy of 12 Resources to Supercharge Your Productivity in R at **http://www.auscov.com***

Introduction

NEURAL networks have been around for several decades[1]. However, I first came across them in the spring of 1992 when I was completing a thesis for my Master's degree in Economics. In the grand old and dusty Foyles bookstore located on Charing Cross Road, I stumbled across a slim volume called Neural Computing by Russell Beale and Tom Jackson[2]. I devoured the book, and decided to build a neural network to predict foreign exchange rate movements.

After several days of coding in GW-BASIC my neural network model was ready to be tested. I fired up my Amstrad 2286 computer and let it rip on the data. Three and a half days later (computers were much slower back then) it delivered the results. The numbers compared well to a wide variety of time series statistical models, and it outperformed every economic theory of exchange rate movement I was able to find. I ditched Economic theory but was hooked on predictive analytics! I have been building, deploying and toying with neural networks and other marvelous predictive analytic models ever since.

Neural networks came out of the desire to simulate the physiological structure and function of the human brain. Although the desire never quite materialized[3] it was soon discovered that they were pretty good at classification and prediction tasks[4].

They 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.

Neural networks are useful in areas where classification and/or prediction is required. Anyone interested in prediction and classification problems in any area of commerce, industry or research should have them in their toolkit. In essence, provided you have sufficient historical data or case studies for which prediction or classification is required you can build a neural network model.

Who Uses Neural Networks?

Because of their flexibility they are of growing interest to many different disciplines:

- Computer scientists investigating the properties of non-symbolic information processing.
- Economists and financial analysts who seek to add greater flexibility to their modeling toolkit.
- Investment portfolio managers and traders who use them to make optimal decision about buying and selling commodities, equities, bonds, derivatives and other financial assets.
- Educationalists and cognitive scientists who want to describe and understand thinking, conscience and learning systems in general.
- Engineers of all varieties who deploy them to solve their applied problems.
- Physicists may use neural networks to model phenomena in statistical mechanics.

- Biologists use neural networks to interpret nucleotide sequences.
- Data Scientists use these models to beat the pants of models derived from economic theory.
- And now You!

What Problems Can Neural Networks Solve?

They have been successfully applied in a huge variety of applications. Here is a very partial list.

- Process Modeling and Control[5].
- Health Diagnostics[6].
- Investment Portfolio Management[7].
- Military Target Recognition[8].
- Analysis of MRI and X-rays[9].
- Credit Rating of individuals by banks and other financial institutions[10].
- Marketing campaigns[11].
- Voice Recognition[12].
- Forecasting the stock market[13].
- Text Searching[14].
- Financial Fraud Detection[15].
- Optical Character Recognition[16].

We will explore and discuss a wide range of other specific real world applications as we walk together through the remainder of this text.

How to Get the Most from this Book

There are at least three ways to use this book. First, you can dip into it as an efficient reference tool. Flip to the chapter you need and quickly see how calculations are carried out in R. For best results type in the example given in the text, examine the results, and then adjust the example to your own data. Second, browse through the real world examples, illustrations, case studies, tips and notes to stimulate your own research ideas. Third, by working through the numerous examples and exercises, you will strengthen your knowledge and understanding of both neural network modeling and R.

☛ PRACTITIONER TIP ☚

If you are using Windows you can easily upgrade to the latest version of R using the `installr` package. Enter the following:

```
> install.packages("installr")
> installr::updateR()
```

If a package mentioned in the text is not installed on your machine you can download it by typing `install.packages("package_name")`. For example, to download the `neuralnet` package you would type in the R console:

```
> install.packages("neuralnet")
```

Once a package is installed, you must call it. You do this by typing in the R console:

```
> require(neuralnet)
```

The `neuralnet` package is now ready for use. You only need to type this once, at the start of your R session.

☛ ***PRACTITIONER TIP*** ☚

You should only download packages from CRAN using encrypted HTTPS connections. This provides much higher assurance that the code you are downloading is from a legitimate CRAN mirror rather than from another server posing as one. Whilst downloading a package from a HTTPS connection you may run into an error message something like:
`"unable to access index for repository https://cran.rstudio.com/..."`
This is particularly common on Windows. The internet2 dll has to be activated on versions before R-3.2.2. If you are using an older version of R before downloading a new package enter the following:

```
> setInternet2(TRUE)
```

Functions in R often have multiple parameters. In the examples in this text I focus primarily on the key parameters required for rapid model development. For information on additional parameters available in a function type in the R console `?function_name`. For example, to find out about additional parameters in the `neuralnet` function, you would type:

```
?neuralnet
```

Details of the function and additional parameters will appear in your default web browser. After fitting your model of interest you are strongly encouraged to experiment with additional parameters. I have also included the `set.seed` method in the R code samples throughout this text to assist you in reproducing the results exactly as they appear on the page. R is available for all the major operating systems. Due to the popularity of windows, examples in this book use the windows version of R.

☛ ***PRACTITIONER TIP*** ☚

Can't remember what you typed two hours ago! Don't worry, neither can I! Provided you are logged into the same R session you simply need to type:

```
> history(Inf)
```

It will return your entire history of entered commands for your current session.

You don't have to wait until you have read the entire book to incorporate the ideas into your own analysis. You can experience their marvelous potency for yourself almost immediately. You can go straight to the section of interest and immediately test, create and exploit it in your own research and analysis.

.

☛ ***PRACTITIONER TIP*** ☚

On 32-bit Windows machines, R can only use up to 3Gb of RAM, regardless of how much you have installed. Use the following to check memory availability:

```
> memory.limit()
```

To remove all objects from memory:

```
rm(list=ls())
```

As implied by the title, this book is about understanding and then hands on building of neural networks; more precisely, it is an attempt to give you the tools you need to build these models easily and quickly. 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.

Applying the ideas in this book will transform your data science practice. If you utilize even one tip or idea from each chapter, you will be far better prepared not just to survive but to excel when faced by the challenges and opportunities of the ever expanding deluge of exploitable data.

As you use these models successfully in your own area of expertise, write and let me know. I'd love to hear from you. Contact me at info@NigelDLewis.com or visit www.AusCov.com.

Now let's get started!

Notes

[1]A historical overview dating back to the 1940's can be found in Yadav, Neha, Anupam Yadav, and Manoj Kumar. An Introduction to Neural Network Methods for Differential Equations. Dordrecht: Springer Netherlands, 2015.

[2]Beale, Russell, and Tom Jackson. Neural Computing-an introduction. CRC Press, 1990.

[3]Part of the reason is that a artificial neural network might have anywhere from a few dozen to a couple hundred neurons. In comparison, the human nervous system is believed to have at least $3x10^{10}$neurons.

[4]When I am talking about a neural network, I should really say "artificial neural network", because that is what we mean most of the time. Biological neural networks are much more complicated in their elementary structures than the mathematical models used in artificial neural networks.

[5]See for example:

1. Shaw, Andre M., Francis J. Doyle, and James S. Schwaber. "A dynamic neural network approach to nonlinear process modeling." Computers & chemical engineering 21.4 (1997): 371-385.
2. Omidvar, Omid, and David L. Elliott. Neural systems for control. Elsevier, 1997.
3. Rivals, Isabelle, and Léon Personnaz. "Nonlinear internal model control using neural networks: application to processes with delay and design issues." Neural Networks, IEEE Transactions on 11.1 (2000): 80-90.

[6]See for example:

1. Lisboa, Paulo JG. "A review of evidence of health benefit from artificial neural networks in medical intervention." Neural networks 15.1 (2002): 11-39.
2. Baxt, William G. "Application of artificial neural networks to clinical medicine." The lancet 346.8983 (1995): 1135-1138.
3. Turkoglu, Ibrahim, Ahmet Arslan, and Erdogan Ilkay. "An intelligent system for diagnosis of the heart valve diseases with wavelet packet neural networks." Computers in Biology and Medicine 33.4 (2003): 319-331.

[7]See for example:

1. Khoury, Pascal, and Denise Gorse. "Investing in emerging markets using neural networks and particle swarm optimisation." Neural

Networks (IJCNN), 2015 International Joint Conference on. IEEE, 2015.

2. Freitas, Fabio D., Alberto F. De Souza, and Ailson R. de Almeida. "Prediction-based portfolio optimization model using neural networks." Neurocomputing 72.10 (2009): 2155-2170.
3. Vanstone, Bruce, and Gavin Finnie. "An empirical methodology for developing stock market trading systems using artificial neural networks." Expert Systems with Applications 36.3 (2009): 6668-6680.

[8]See for example:

1. Rogers, Steven K., et al. "Neural networks for automatic target recognition." Neural networks 8.7 (1995): 1153-1184.
2. Avci, Engin, Ibrahim Turkoglu, and Mustafa Poyraz. "Intelligent target recognition based on wavelet packet neural network." Expert Systems with Applications 29.1 (2005): 175-182.
3. Shirvaikar, Mukul V., and Mohan M. Trivedi. "A neural network filter to detect small targets in high clutter backgrounds." Neural Networks, IEEE Transactions on 6.1 (1995): 252-257.

[9]See for example:

1. Vannucci, A., K. A. Oliveira, and T. Tajima. "Forecast of TEXT plasma disruptions using soft X rays as input signal in a neural network." Nuclear Fusion 39.2 (1999): 255.
2. Kucian, Karin, et al. "Impaired neural networks for approximate calculation in dyscalculic children: a functional MRI study." Behavioral and Brain Functions 2.31 (2006): 1-17.
3. Amartur, S. C., D. Piraino, and Y. Takefuji. "Optimization neural networks for the segmentation of magnetic resonance images." IEEE Transactions on Medical Imaging 11.2 (1992): 215-220.

[10]See for example:

1. Huang, Zan, et al. "Credit rating analysis with support vector machines and neural networks: a market comparative study." Decision support systems 37.4 (2004): 543-558.
2. Atiya, Amir F. "Bankruptcy prediction for credit risk using neural networks: A survey and new results." Neural Networks, IEEE Transactions on 12.4 (2001): 929-935.
3. Jensen, Herbert L. "Using neural networks for credit scoring." Managerial finance 18.6 (1992): 15-26.

[11]See for example:

1. Potharst, Rob, Uzay Kaymak, and Wim Pijls. "Neural networks for target selection in direct marketing." ERIM report series reference no. ERS-2001-14-LIS (2001).
2. Vellido, A., P. J. G. Lisboa, and K. Meehan. "Segmentation of the on-line shopping market using neural networks." Expert systems with applications 17.4 (1999): 303-314.
3. Hill, Shawndra, Foster Provost, and Chris Volinsky. "Network-based marketing: Identifying likely adopters via consumer networks." Statistical Science (2006): 256-276.

[12]See for example:

1. Waibel, Alexander, et al. "Phoneme recognition using time-delay neural networks." Acoustics, Speech and Signal Processing, IEEE Transactions on 37.3 (1989): 328-339.
2. Lippmann, Richard P. "Review of neural networks for speech recognition." Neural computation 1.1 (1989): 1-38.
3. Nicholson, Joy, Kazuhiko Takahashi, and Ryohei Nakatsu. "Emotion recognition in speech using neural networks." Neural computing & applications 9.4 (2000): 290-296.

[13]See for example:

1. Kimoto, Tatsuya, et al. "Stock market prediction system with modular neural networks." Neural Networks, 1990., 1990 IJCNN International Joint Conference on. IEEE, 1990.
2. Fernandez-Rodrıguez, Fernando, Christian Gonzalez-Martel, and Simon Sosvilla-Rivero. "On the profitability of technical trading rules based on artificial neural networks: Evidence from the Madrid stock market." Economics letters 69.1 (2000): 89-94.
3. Guresen, Erkam, Gulgun Kayakutlu, and Tugrul U. Daim. "Using artificial neural network models in stock market index prediction." Expert Systems with Applications 38.8 (2011): 10389-10397.

[14]See for example:

1. Chung, Yi-Ming, William M. Pottenger, and Bruce R. Schatz. "Automatic subject indexing using an associative neural network." Proceedings of the third ACM conference on Digital libraries. ACM, 1998.

2. Frinken, Volkmar, et al. "A novel word spotting method based on recurrent neural networks." Pattern Analysis and Machine Intelligence, IEEE Transactions on 34.2 (2012): 211-224.
3. Zhang, Min-Ling, and Zhi-Hua Zhou. "Multilabel neural networks with applications to functional genomics and text categorization." Knowledge and Data Engineering, IEEE Transactions on 18.10 (2006): 1338-1351.

[15]See for example:

1. Maes, Sam, et al. "Credit card fraud detection using Bayesian and neural networks." Proceedings of the 1st international naiso congress on neuro fuzzy technologies. 2002.
2. Brause, R., T. Langsdorf, and Michael Hepp. "Neural data mining for credit card fraud detection." Tools with Artificial Intelligence, 1999. Proceedings. 11th IEEE International Conference on. IEEE, 1999.
3. Sharma, Anuj, and Prabin Kumar Panigrahi. "A review of financial accounting fraud detection based on data mining techniques." arXiv preprint arXiv:1309.3944 (2013).

[16]See for example:

1. Yu, Qiang, et al. "Application of Precise-Spike-Driven Rule in Spiking Neural Networks for Optical Character Recognition." Proceedings of the 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems-Volume 2. Springer International Publishing, 2015.
2. Barve, Sameeksha. "Optical character recognition using artificial neural network." International Journal of Advanced Research in Computer Engineering & Technology 1.4 (2012).
3. Patil, Vijay, and Sanjay Shimpi. "Handwritten English character recognition using neural network." Elixir Comp. Sci. & Engg 41 (2011): 5587-5591.

Part I

Multi-Layer Perceptron

In this part of the book we walk step by step through the process of building a variety of Multi-+Layer Perceptron (MLP) models. We begin by describing the MLP and explain how it works. Next we explain how to build MLPs in R.

There are several issues involved in designing and training a multilayer perceptron which we will cover:

1. Selecting how many hidden layers to use in the network.

2. Deciding how many neurons to use in each hidden layer.

3. Finding a globally optimal solution that avoids local minima in a reasonable period of time.

4. Validating the neural network to test for over fitting.

Since real world data is messy, our first step will be to review, prepare and assess the data to be used in the analysis. Then we will estimate the MLP model and use it for the classification task. During the process we will explore multiple layer models and use aggregation methods with majority voting in an attempt to improve model fit.

In addition to using a variant of backpropagation we will also build and compare an MLP which uses an alternative optimization procedure. Since we will be working with real data on a challenging problem, expect disappoints and failures along the way!

Chapter 1

MLP in a Nutshell

A neural network is constructed from a number of interconnected nodes known as neurons. These are arranged into an input layer, a hidden layer and an output layer. The input layer nodes correspond to the number of features you wish to feed into the neural network, and the number of output nodes correspond to the number of items you wish to predict or classify.

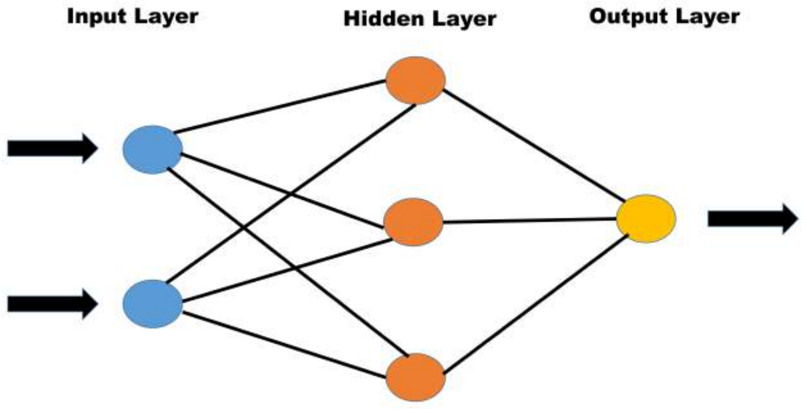


Figure 1.1: A basic neural network

Figure 1.1 illustrates a neural network topology. It has 2 input nodes, 1 hidden layer with 3 nodes and 1 output node. A neural network with one input layer, one or more hidden layers and an output layer is called a Multi-Layer Perceptron (MLP).

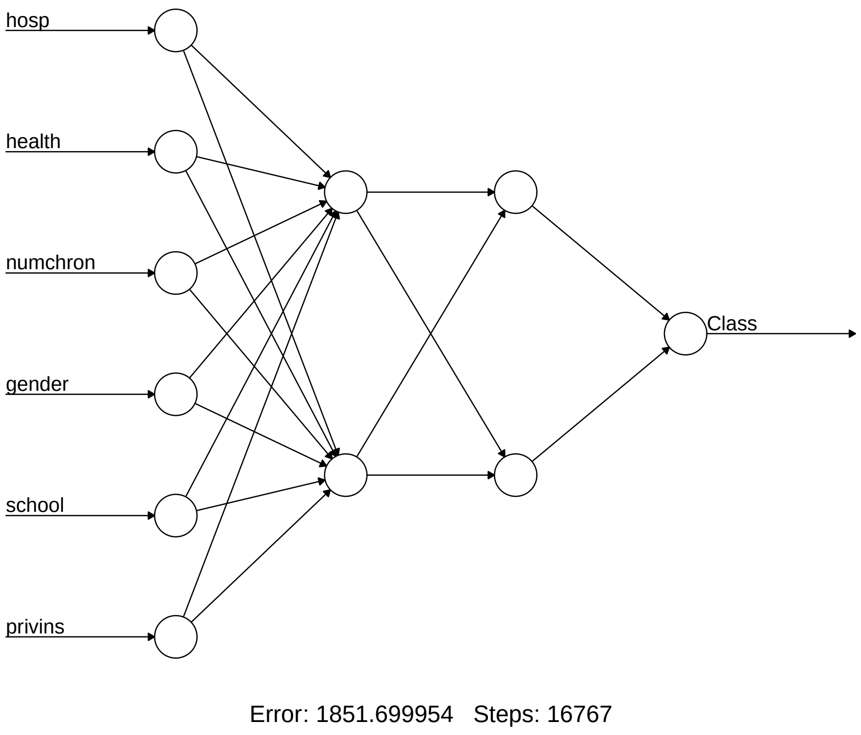


Figure 1.2: Multi-Layer Perceptron

Figure 1.2 shows the topology of a typical multi-layer perceptron as represented in R. This particular perceptron has six input nodes. To the left of each node is the name of the attribute in this case `hosp, health, numchron, gender, school` and `privins` respectively. The network has two hidden layers, each containing two nodes. The response or output variable is called `Class`. The figure also reports the network error, in this case 1851 and the number of steps required for the network to converge.

The Role of the Neuron

Figure 1.3 illustrates the working of a biological neuron. Biological neurons pass signals or messages to each other via electrical signals. Neighboring neurons receive these signals through their dendrites. Information in terms of these signals or messages flow from the dendrites to the main cell body, known as the soma and via the axon to the axon terminals. In essence, biological neurons are computation machines passing messages between each other about various biological functions.

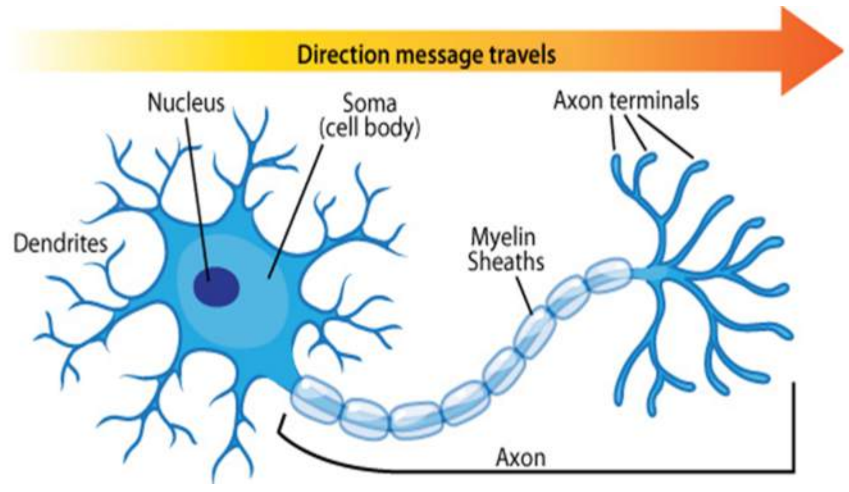


Figure 1.3: Biological Neuron. © Arizona Board of Regents / ASU Ask A Biologist. https://askabiologist.asu.edu/neuron-anatomy. See also `http://creativecommons.org/licenses/by-sa/3.0/`

At the heart of an artificial neural network is an artificial neuron. It, like it's biological cousin, is the basic processing unit. The input layer neurons receive incoming information which they process and then distribute to the hidden layer neurons. This information is processed by the hidden layer neurons and passed onto the output layer neurons. The key here is that information is processed via an activation function. The 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. This ensures the strength of the connection between two neuron is sized according to the weight of the processed information.

NOTE... ✍

The original "Perceptron" model was developed by a researcher at the Cornell Aeronautical Laboratory back in 1958[17]. It consisted of three layers with no feedback:

1. A "retina" that distributed inputs to the second layer;
2. association units that combine the inputs with weights and a threshold step function;
3. the output layer.

Each neuron contains a activation function and a threshold value. The threshold value is the minimum value that a input must have to activate the neuron. The task of the neuron therefore 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 performs this summation on the input data. The middle layer neurons perform the summation on the weighted 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 middle layer neurons.

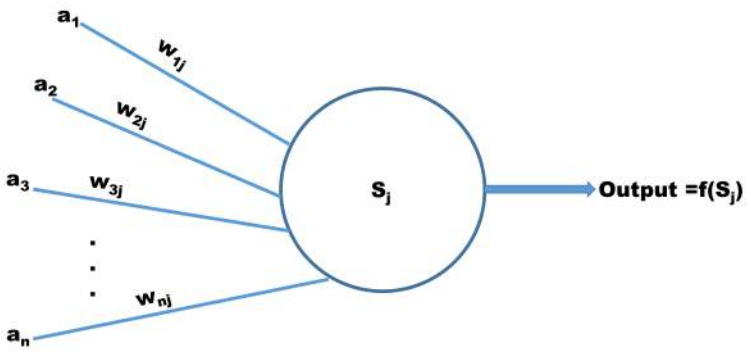


Figure 1.4: An artificial neuron

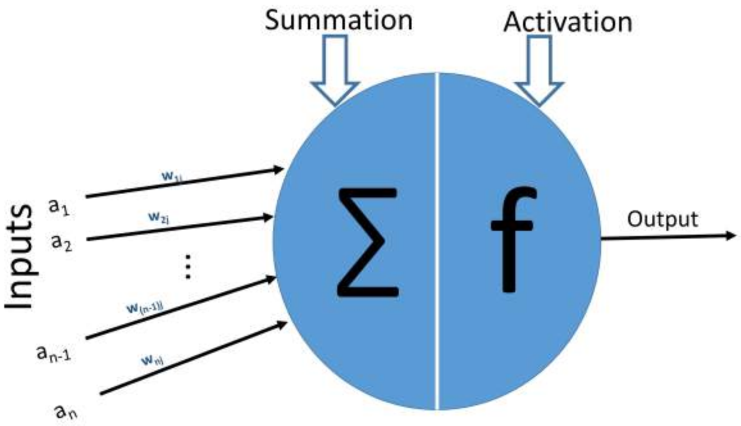


Figure 1.5: Alternative representation of neuron

Figure 1.4 and Figure 1.5 illustrate the workings of an individual neuron. Given a sample of input attributes $\{a_1,...,a_n\}$ a weight w_{ij} is associated with each connection into the neuron; and the neuron then sums all inputs according to:

$$S_j = \sum_{i=1}^{n} w_{ij} a_j + b_j$$

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[18].

Activation Functions

Activation functions for the hidden layer nodes are needed to introduce non linearity into the network. The activation function $f(S_j)$ is applied and the output passed to the next neuron(s) in the network. It 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 backpropagation algorithm it must be differentiable and it helps if the function is bounded.

The sigmoid function is a popular choice. It is an "S" shape differentiable activation function. It is shown in Figure 1.6 where parameter c is a constant taking the value 1.5. It is popular partly because it can be easily differentiated and therefore reduces the computational cost during training. It also produces an output between the values 0 and 1 and is given by:

$$f(S_j) = \frac{1}{1 + exp(-cS_j)}$$

Two other popular activation functions are:

1. **Linear function:** The output of the neuron is the weighted sum of the inputs:

$$f(S_j) = S_j$$

2. **Hyperbolic Tangent function:** Produces a value between -1 and 1. The function takes the form:

   $f(S_j) = \tanh(cS_j)$

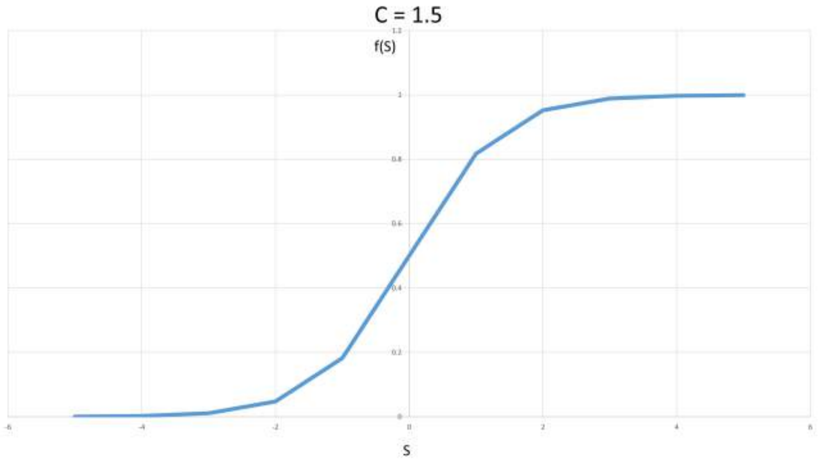


Figure 1.6: The sigmoid function with c = 1.5

Neural Network Learning

To learn from data a neural network uses a specific learning algorithm. There are many learning algorithms, but in general they all train the network by iteratively modifying the connection weights until the error between the output produced by the network and the desired output fall below a pre-specified threshold.

The backpropagation algorithm was the first leaning algorithm and is still widely used. It uses gradient descent as the core learning mechanism. Starting from random weights the backpropagation algorithm calculates the network weights making small changes and gradually making adjustments determined by the error between the result produced by the network and the desired outcome.

The algorithm applies error propagation from outputs to inputs and gradually fine tunes the network weights to minimize the sum of error using the gradient descent technique. Learning therefore 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:** The output of the network is assessed relative to known output. If the error is below a pre-specified threshold the network is trained and the algorithm terminated.

- **Step 4:- Propagate:** The error at the output layers 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".

This basic idea is roughly illustrated in Figure 1.7. 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)[19]. Each cycle through this process is called a epoch.

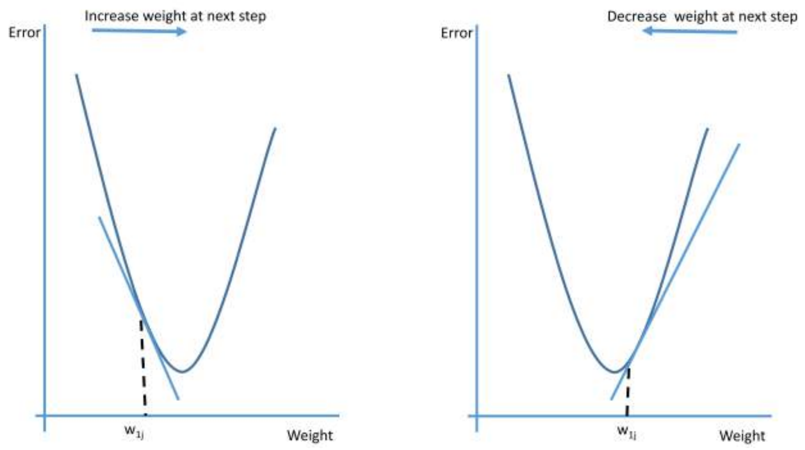


Figure 1.7: Basic idea of the backpropagation algorithm

NOTE... ✍

Neural networks are initialized by setting random values to the weights and biases. One rule of thumb is to set the random values to lie in the range (-2 n to 2 n), where n is the number of inputs.

I discovered early on that backpropagation using gradient descent often converges very slowly or not at all. In my first coded neural network I used the backpropagation algorithm and it took over 3 days for the network to converge to a solution! Fortunately, computers are much faster today than they were in the 1990s! In addition, more efficient learning algorithms have been developed.

Despite the relatively slow learning rate associated with backpropagation, being a feedforward algorithm, it is quite rapid during the prediction or classification phase.

Finding the globally optimal solution that avoids local minima is a challenge. This is because a typical neural network may have hundreds of weights whose combined values are used to generate a solution. The solution is often highly nonlinear which makes the optimization process complex. To avoid the network getting trapped in a local minima, a momentum parameter is often specified.

NOTE... ✍

There are two basic types of learning used in decision science:

1. **Supervised learning**: Your training data contains the known outcomes. The model is trained relative to these outcomes.

2. **Unsupervised learning:** Your training data does not contain any known outcomes. In this case the algorithm self-discovers relationships in your data.

Practical Applications

Let's take a look at some real world practical applications of the MLP in action. The variety of places where this modeling tool has been found useful is quite staggering. In science, industry and research these nifty little models have added significant value to their uses. Those who have added them to their toolkit have certainly gained new insights. I like to think about MLPs as legendary Jedi Master Yoda thought about life in the movie Star wars when he said "Do. Or do not. There is no try." Take a look at these uses and see what you think.

Sheet Sediment Transport

Tayfur[20]model sediment transport using artificial neural networks (ANNs). The sample consisted of the original experimental hydrological data of Kilinc & Richardson[21]

A three-layer feed-forward artificial neural network with two neurons in the input layer, eight neurons in the hidden layer, and one neuron in the output layer was built. The sigmoid function was used as an activation function in the training of the network and the learning of the ANN was accomplished by the backpropagation algorithm. A random value of 0.2, and —1.0 were assigned for the network weights and biases, before starting the training process.

The ANNs performance was assessed relative to popular physical hydrological models (flow velocity, shear stress, stream power, and unit stream power) against a combination of slope types (mild, steep and very steep) and rain intensity (low, high, very high).

The ANN outperformed the popular hydrological models for very high intensity rainfall on both steep and very steep slope, see Table 1.

	Mild slope	Steep slope	Very steep slope
Low Intensity	physical	physical	ANN
High Intensity	physical	physical	physical
Very high Intensity	physical	ANN	ANN

Table 1: Which model is best? Model transition framework derived from Tayfur's analysis.

☛ ***PRACTITIONER TIP*** ☚

Tayfur's results highlight an important issue facing the data scientist. Not only is it often a challenge to find the best model among competing candidates, it is even more difficult to identify a single model that works in all situations. A great solution, offered by Tayfur is to have a model transition matrix, that is determine which model(s) perform well under specific conditions and then use the appropriate model for a given condition.

Stock Market Volatility

Mantri et al.[22] investigate the performance of a multilayer perceptron relative to standard econometric models of stock market volatility (GARCH, Exponential GARCH, Integrated GARCH, and the Glosten, Jagannathan and Runkle GARCH model).

Since the multilayer perceptron does not make assumptions about the distribution of stock market innovations it is of interest to Financial Analysts and Statisticians. The sample consisted of daily data collected on two Indian stock indices (BSE SENSEX and the NSE NIFTY) over the period January 1995 to December 2008.

The researchers found no statistical difference between the volatility of the stock indices estimated by the multilayer perceptron and standard econometric models of stock market volatility.

Trauma Survival

The performance of trauma departments in the United Kingdom is widely audited by applying predictive models that assess the probability of survival, and examining the rate of unexpected survivals and deaths. The standard approach is the TRISS methodology which consists of two logistic regressions, one applied if the patient has a penetrating injury and the other applied for blunt injuries[23].

Hunter, Henry and Ferguson[24] assess the performance of the TRISS methodology against alternative logistic regression models and a multilayer perceptron.

The sample consisted of 15,055 cases, gathered from Scottish Trauma departments over the period 1992-1996. The data was divided into two subsets: the training set, containing 7,224 cases from 1992-1994; and the test set, containing 7,831 cases gathered from 1995-1996. The researcher's logistic regression models and the neural network were optimized using the training set.

The neural network was optimized ten times with the best resulting model selected. The researchers conclude that neural networks can yield better results than logistic regression.

Brown Trout Reds

Lek et al.[25] compare the ability of multiple regression and neural networks to predict the density of brown trout redds in southwest France. Twenty nine observation stations distributed on six rivers and divided into 205 morphodynamic units collected information on 10 ecological metrics.

The models were fitted using all the ecological variables and also with a sub-set of four variables. Testing consisted of random selection for the training set (75% of observations and the test set 25% of observations). The process was repeated a total of five times.

The average correlation between the observed and estimated values over the five samples is reported in Table 2. The researchers conclude that both multiple regression and neural networks can be used to predict the density of brown trout redds, however the neural network model had better prediction accuracy.

Neural Network		Multiple Regression	
Train	Test	Train	Test
0.900	0.886	0.684	0.609

Table 2: Let et al's reported correlation coefficients between estimate and observed values in training and test samples

Chlorophyll Dynamics

Wu et al.[26] developed two modeling approaches - artificial neural networks (ANN) and multiple linear regression (MLR), to simulate the daily Chlorophyll *a* dynamics in a northern German lowland river. Chlorophyll absorbs sunlight to synthesize carbohydrates from CO_2 and water. It is often used as a proxy for the amount of phytoplankton present in water bodies.

Daily Chlorophyll *a* samples were taken over an 18 month period. In total 426 daily samples were obtained. Each 10th daily sample was assigned to the validation set resulting in 42 daily observations. The calibration set contained 384 daily observations.

A three layer backpropagation neural network was used. The input layer consisted of 12 neurons corresponding to the independent variables shown in Table 3. The same independent variables were also used in the multiple regression model. The dependent variable in both models was the daily concentration of Chlorophyll *a*.

The results of the ANN and MLR illustrated a good agreement between the observed and predicted daily concentration of Chlorophyll *a*, see Table 4.

Air temperature	Ammonium nitrogen
Average daily discharge	Chloride
Chlorophyll a concentration	Daily precipitation
Dissolved inorganic nitrogen	Nitrate nitrogen
Nitrite nitrogen	Orthophosphate phosphorus
Sulfate	Total phosphorus

Table 3: Wu et al's input variables

Model	R-Square	NS	RMSE
MLR	0.53	0.53	2.75
NN	0.63	0.62	1.94

Table 4: Wu et al's performance metrics (NS = Nash Sutcliffe efficiency, RMSE = root mean square error)

☛ ***PRACTITIONER TIP*** ☚

To see which packages are installed on your machine use the following:

```
pack <- as.data.frame
(installed.packages()[,c(1,3:4)])

rownames(pack) <- NULL

pack <- pack[is.na(pack$Priority),
1:2,drop=FALSE]

print(pack, row.names=FALSE)
```

Notes

[17]Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.

[18]This is because a multilayer perceptron, say with a step activation function, and with n inputs collectively define an n-dimensional space. In such a network any given node creates a separating hyperplane producing an "on" output on one side and an "off" output on the other. The weights determine where this hyperplane lies in the input space. Without a bias term, this separating hyperplane is constrained to pass through the origin of the space defined by the inputs. This, in many cases, would severely restrict a neural networks ability to learn.

[19]For a detailed mathematical explanation see, R. Rojas. Neural Networks. Springer-Verlag, Berlin, 1996

[20]Tayfur, Gokmen. "Artificial neural networks for sheet sediment transport." Hydrological Sciences Journal 47.6 (2002): 879-892.

[21]Kilinc, Mustafa. "Mechanics of soil erosion from overland flow generated by simulated rainfall." Colorado State University. Hydrology Papers (1973).

[22]Mantri, Jibendu Kumar, P. Gahan, and Braja B. Nayak. "Artificial neural networks–An application to stock market volatility." Soft-Computing in Capital Market: Research and Methods of Computational Finance for Measuring Risk of Financial Instruments (2014): 179.

[23]For further details see H.R. Champion et.al, Improved Predictions from a Severity Characterization of Trauma (ASCOT) over Trauma and Injury Severity Score (TRISS): Results of an Independent Evaluation. Journal of Trauma: Injury, Infection and Critical Care, 40 (1), 1996.

[24]Hunter, Andrew, et al. "Application of neural networks and sensitivity analysis to improved prediction of trauma survival." Computer methods and programs in biomedicine 62.1 (2000): 11-19.

[25]Lek, Sovan, et al. "Application of neural networks to modelling nonlinear relationships in ecology." Ecological modelling 90.1 (1996): 39-52.

[26]Wu, Naicheng, et al. "Modeling daily chlorophyll a dynamics in a German lowland river using artificial neural networks and multiple linear regression approaches." Limnology 15.1 (2014): 47-56.

Chapter 2

Building a Single Layer MLP

THE process of building an MLP in R involves several stages. The general approach taken throughout this book is outlined in Figure 2.1.

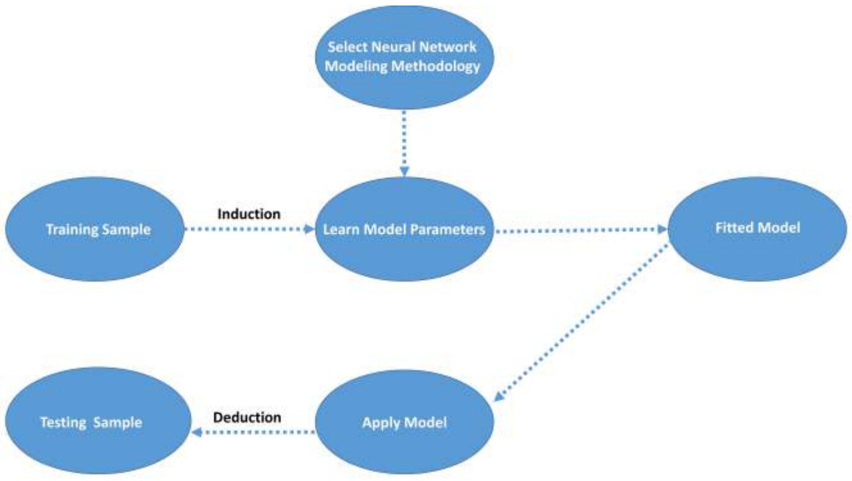


Figure 2.1: Neural network model building framework

An important part of this process is identifying the problem to be modeled, determine and load the appropriate packages, selection of the data, identifying a suitable response variable, cleaning the data, selection of relevant attributes, specification of the formula for the model, estimation of the model and assessment of the predictions. This chapter walks you through each of these steps. The goal is to build an MLP model to predict whether an individual is likely to have a higher than average number of physician office visits.

Which Packages Should I Use?

The first task is to load the required packages. The package `neuralnet` is R's work engine for backpropagation and we will make heavy use of this package in our analysis. It is also useful to try other learning mechanisms so the package `nnet`, which uses optimization based on Nelder–Mead, quasi-Newton and conjugate-gradient algorithm, is also loaded. Finally, we use the `plyr` package for a custom function to count agreement between columns:

```
> require(neuralnet)
> require(nnet)
> library(plyr)
```

Understanding Your Data

Research psychologist Daniel B. Wright gave invaluable advice when he stated[27] "*Conducting data analysis is like drinking a fine wine. It is important to swirl and sniff the wine, to unpack the complex bouquet and to appreciate the experience. Gulping the wine doesn't work.*" We need to take his advice to heart. Let's spend a little while getting to know our data.

Deb and Trivedi[28] model counts of medical care utilization by the elderly in the United States using data from the National Medical Expenditure Survey. They analyze data on 4406 individuals, aged 66 and over, who are covered by Medicare, a public insurance program[29].

Our goal is to use the data reported by Deb and Trivedi[30] to build a neural network model to predict which individuals are likely to have a higher than average number of physician office visits. The data we need is contained in the `DebTrivedi` data frame available in the `MixAll` package. It can be loaded into R using the following command:

```
> data("DebTrivedi",package="MixAll")
```

`DebTrivedi` contains 19 columns and 4406 rows corresponding to the individuals aged 66 and over who were part of Deb and Trivedi's study:

```
> ncol(DebTrivedi)
[1] 19

> nrow(DebTrivedi)
[1] 4406
```

The `str` method can be used to check and compactly display the structure of an R object. Let's use it with the `DebTrivedi` data frame:

```
> str(DebTrivedi, list.len = 5)
'data.frame':   4406 obs. of  19 variables:
 $ ofp     : int  5 1 13 16 3 17 9 3 1 0
    ...
 $ ofnp    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ opp     : int  0 2 0 5 0 0 0 0 0 0 ...
 $ opnp    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ emer    : int  0 2 3 1 0 0 0 0 0 0 ...
  [list output truncated]
```

Setting `list.len = 5` shows the first five columns only. As expected `DebTrivedi` is identified as a data frame with 4406 observations on 19 variables. Notice that each row provides details of the name of the attribute, type of attribute and the first few observations. For example, `ofp` is an integer with the first, second and third observations taking the value 5, 1, and

13 respectively. It turns out that `ofp` is the number of physician office visits, and we will use it to derive the classification variable used in our analysis.

☛ ***PRACTITIONER TIP*** ☚

To see the type of an object in R use the `class` method. For example:

```
> class(DebTrivedi$health)
[1] "factor"

> class(DebTrivedi$hosp)
[1] "integer"
```

Let's make a copy of `DebTrivedi` and store it in a data frame called `data`:

```
> data<-DebTrivedi
```

Missing data are a fact of life, individuals die, equipment breaks, you forget to measure something, you can't read your writing, etc. It is always a good idea to check sample data for missing values; these are coded as `NA` in R. Let's make a quick check:

```
> data<-na.omit(data)

> nrow(DebTrivedi)
[1] 4406

> ncol(DebTrivedi)
[1] 19
```

It looks good, so let's continue with a visual inspection of `ofp` using the `plot` method.

Since `ofp` is an integer of counts, we create a table and then call the `plot` method, the result is shown in Figure 2.2.

```
>  plot(table(data$ofp),
xlab="Number of physician office visits (
   ofp)",
ylab="Frequency")
```

The histogram illustrates that the distribution exhibits both substantial variation and a rather large number of zeros. If we were interested in modeling `ofp` directly we could use one of the approaches developed by Deb and Trivedi for count data[31]. However, we are interested in using a neural network to classify whether an individual will have a higher than average number of physician office visits.

One solution is to create a binary variable which for any individual in the study takes the value -1 if the number of office visits is greater than or equal to the median, and 1 otherwise. This is the approach we take, storing the result in the variable called `Class`:

```
> data$Class <- ifelse(DebTrivedi$ofp >=
   median(DebTrivedi$ofp), -1, 1)
```

Notice we use the median as our measure of "average", of course you could also use the mean. Let's take a look at the distribution of the response variable, this is shown in Figure 2.3.

```
> barplot(table(data$Class), ylab="
   Frequency")
```

We use the following six attributes as covariates in the neural network:

1. `hosp`: Number of hospital visits.
2. `health`: Classified as "poor", "average" or "excellent".
3. `numchron`: Number of chronic conditions (cancer, heart attack, gall bladder problems, emphysema, arthritis, diabetes, other heart disease).
4. `gender`: male or female.

5. `school`: Number of years of education

6. `privins`:Private insurance classified as "yes" or "no".

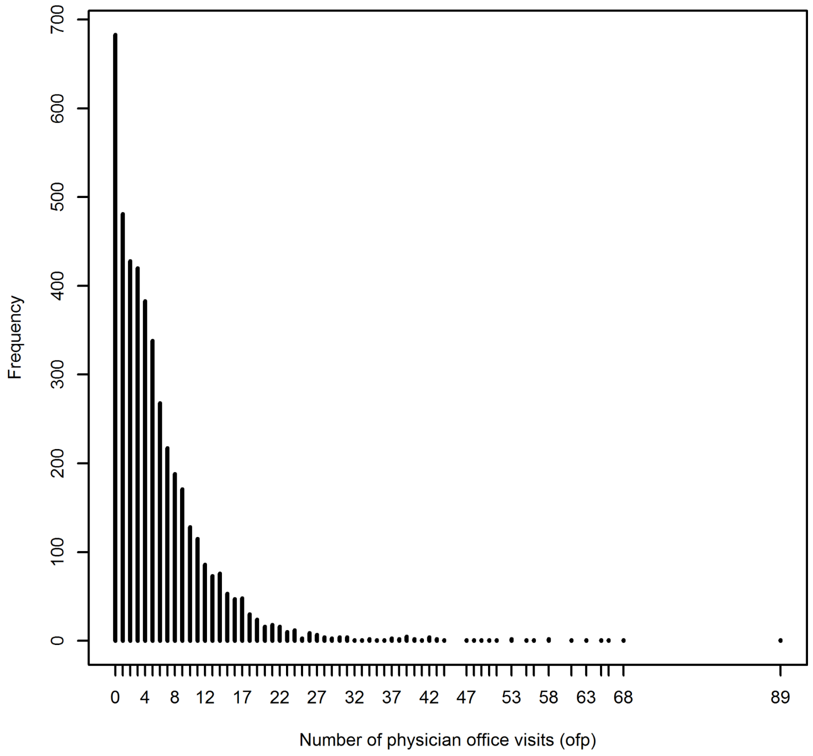


Figure 2.2: Number of physician office visits (`ofp`)

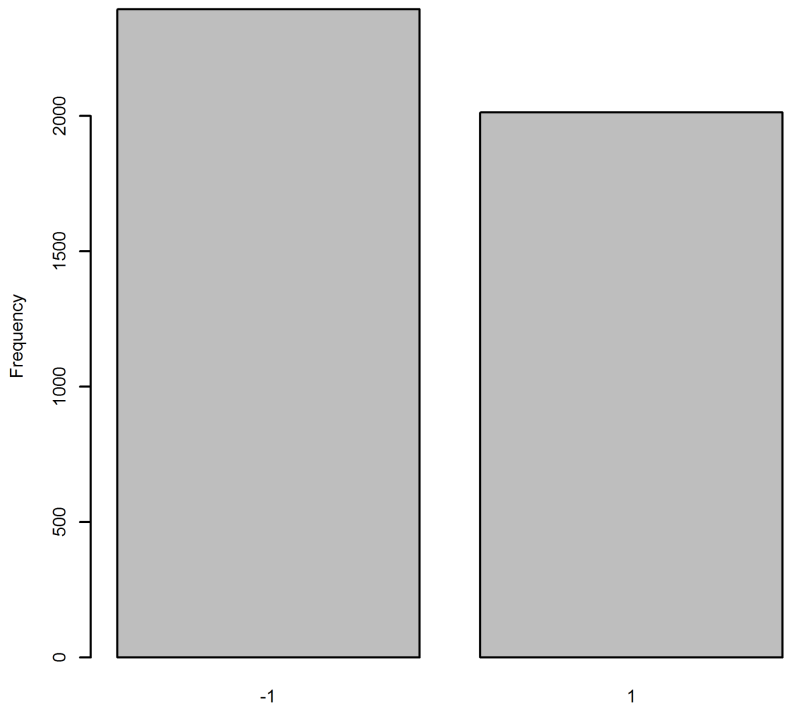


Figure 2.3: Barplot of dichotomous variable `Class`

NOTE... ✍

Before we begin our analysis go to the Appendix and on page 193 enter the R code for the `cfac` function. We will use it to display some of the data contained in `school`, `numchron` and `hosp`.

With the `cfac` function in hand we are ready to explore the relationship between `Class` and each of the covariate attributes. To do this we will use six charts; and since it is useful to see them all on one screen the par method is used as follows:

```
>  par( mfrow = c(3, 2))

> plot(Class ~ cfac(hosp),
data = data,
xlab="Number of hospital stays (hosp)")

> plot(Class ~ cfac(numchron),
data = data,
xlab= "Number of chronic conditions (
   numchron)" )

> plot(Class ~ cfac(school),
data = data,
xlab="Number of years of education (school)
   ")

> plot(Class~health,data=data)

> plot(Class~gender,data=data)

> plot(Class~privins,data=data,
xlab="Private insurance (privins)")
```

The results are shown in Figure 2.4. It seems intuitive that above median physician office visits are associated with increasing hospital stays and the number of chronic conditions. It appears individuals who have less than 6 years of schooling visit their physician's office less than the rest of the sample. There is a similar pattern for individuals who do not have private insurance. Overall there is little difference between male and female; however, those with poor to average health have above median office visits relative to individuals in excellent health.

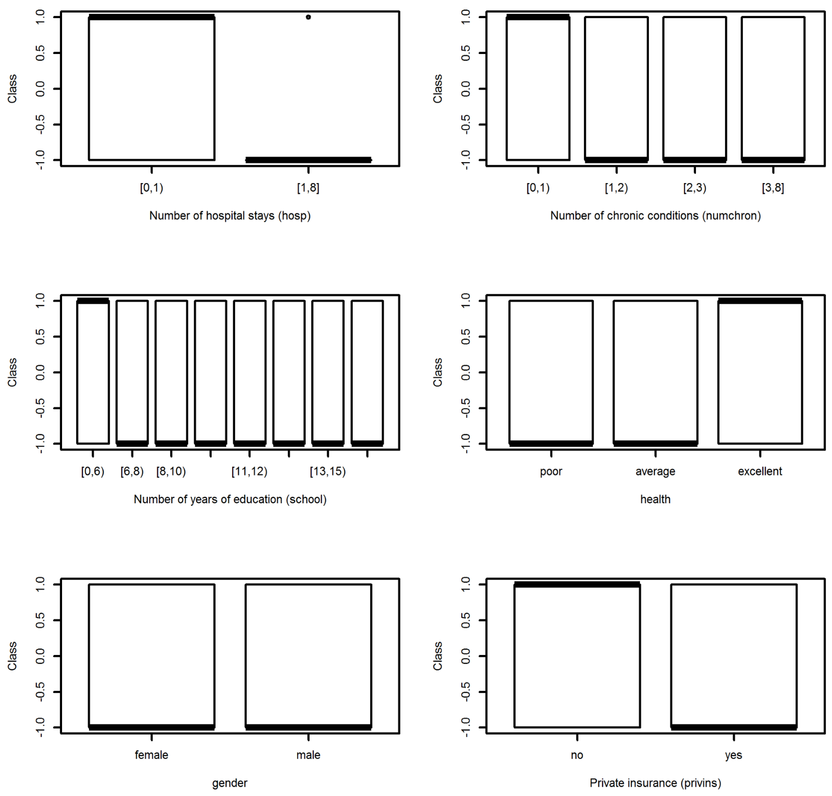


Figure 2.4: Relationships of each of the six attributes to `Class`

We need to convert **gender**, **privins** and **health** to numerical variables. This can be achieved as follows:

```
> levels(data$gender) <- c("-1","1")
> data$gender<-as.numeric(as.character(data
   $gender))

> levels(data$privins) <- c("-1","1")
> data$privins<-as.numeric(as.character(
   data$privins))

> levels(data$health) <- c("0","1","2")
> data$health<-as.numeric(as.character(data
   $health))
```

The remaining count variables can be converted in a similar fashion:

```
> data$hosp<-as.numeric(as.character(data$
   hosp))

> data$numchron<-as.numeric(as.character(
   data$numchron))

> data$school<-as.numeric(as.character(data
   $school))
```

To keep things simple, it is a good idea to only keep those variables that are going to be used. Here is how to do that:

```
> keeps <- c("Class","hosp","health",
"numchron", "gender",
"school" , "privins")

> data<-data[keeps]
```

The `data` data frame now only contains those variables we are actually going to use in our analysis.

Role of Standardization

I generally like to standardize my data, although it is not absolutely necessary I have found it useful in practice. The `scale` method does the job:

```
> data<-scale(data)
```

Whilst there are no fixed rules about how to normalize inputs here are four popular choices for an attribute x_i:

$$z_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{2.1}$$

$$z_i = \frac{x_i - \overline{x}}{\sigma_x} \tag{2.2}$$

$$z_i = \frac{x_i}{\sqrt{SS_i}} \tag{2.3}$$

$$z_i = \frac{x_i}{x_{max} + 1} \tag{2.4}$$

SS_i is the sum of squares of x_i, and $\bar{x}$ and σ_x are the mean and standard deviation of x_i.

How to Choose the Optimal Number of Nodes

One of the very first questions asked about neural networks is how many nodes should be included in the hidden layer? The straightforward, shoot from the hip, answer is there is no fixed rule as to how many nodes to include. Hey wait! but my professor told me...

Here is the "skinny" on this issue; at the start of an empirical investigation the answer to the how many nodes question is "We don't know". However, this is not good enough for most of us, so many researchers, authors, professors and even practitioners (including myself) guesstimate using "rules of thumb". They often take the number of inputs plus the number of outputs possibly divided by two with a sign and maybe a square root thrown in for good measure. These rules are frequently quite crude, fail often, and are the source of endless squabbles about which is the best.

In reality there is no exact answer to this question, partly because it depends on the complexity of the problem you are trying to solve. The best we can say is that you should probably have at least one or two examples for each parameter you are trying to estimate. In other words, if you have lots of examples in your data, you can probably safely include lots of hidden nodes.

Don't get too hung up on rules of thumb. I'll throw some decent ones out as we go along; but know that they are simply a tool to get you started - they are not the answer. So on the number of nodes in a hidden layer, I take my advice from Thomas A. Edison when he said about his journey toward the discovery of the electric light bulb "*I have not failed. I've just found 10,000 ways that won't work.*" Trial and error, guided by experience and knowledge of the data is often the best place to start. In any case, did I mention that Data Science is a series of failures punctuated by the occasional success?

Creating a Neural Network Formula

I find it efficient, when working in R, to use formula for the models I develop. This allows complex equations to be contained in a single R object. It is pretty straightforward to do this in R, here is how:

```
> f<-Class~hosp + health + numchron +
   gender + school + privins
```

We use the `<-` operator to assign the formula to `f`, and the ~ operator to indicate `Class` as the response variable which is to be modeled against the six specified attributes. For completeness, let's take a peek at the class of f:

```
>  class(f)
[1] "formula"
```

Ensuring Reproducible Results

Let's generate five random values from the normal distribution:

```
> rnorm(5)
[1] -0.01645672  0.27054771 -0.24473636
   1.59675103  0.15970867
```

The first random value takes the value -0.01645672, and the last the value 0.15970867. Since they are random they each take on different values. Now, if we generate another five random values this is what we observe:

```
> rnorm(5)
[1]  0.1228062 -0.7922208  0.8522540
   2.1538066 -1.0498416
```

These values are different from the first five we generated. In fact, you will have noticed if you are typing in the examples as you read this book, that your five random numbers are different from those shown here. Now, just suppose we needed to reproduce the exact sequence of random values - so that you could see the exact same results as printed on the page. How would we do that? Throughout this book the `set.seed` method is used to ensure the reproducibility of the results.

Let's generate five more random variables, this time using the `set.seed` method:

```
> set.seed(103)
> rnorm(5)
[1] -0.7859732  0.0547389 -1.1725603
   -0.1673128 -1.8650316
```

Notice, all the values are different from each other. Now let's reproduce this result exactly:

```
> set.seed(103)
> rnorm(5)
[1] -0.7859732  0.0547389 -1.1725603
   -0.1673128 -1.8650316
```

We get the exact same sequence of observations; in essence we have reproduced the result. As you build your own models you may also wish to use this method so that your fellow workers or those interested in your research can reproduce your exact result.

The count of the number of observations in the sample is stored in the R object `nrow`. A total of four thousand randomly selected individual observations will be used to train the model. They are selected without replacement using the `sample` method. The remaining 406 observations will be retained for use in the test sample. Here is the R code to achieve this:

```
> set.seed(103)
> n=nrow(data)
> train <- sample(1:n, 4000, FALSE)
```

Estimating the Model

At last we have everything in pace to estimate a neural network. We have loaded the appropriate packages, selected and cleaned our data, determined the form of the response variable, selected which attributes to use and specified the formula for the model. This is a process you will repeat over and over again as you build and successfully develop multiple neural network models.

NOTE... ✍

Neural networks learn by modification of their weights. The ultimate goal of the training process is to find the set of weight values that result in the output of the neural network closely matching the actual observed target values.

Our task is to estimate the optimal weights of our MLP model. To do this we use the `neuralnet` function in the `neuralnet` package. This function is the workhorse of neural networking modeling using backpropagation in R[32].

Let's begin by estimating an MLP with one hidden layer. Here is how to do that:

```
> fit<- neuralnet(f,
data = data[train,],
hidden=1,
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",
linear.output=FALSE)
```

Notice we begin by giving the `neuralnet` method the model formula stored in `f`. This is followed by the `data` argument with which we pass to our training data; `hidden` is set to 1 and controls the number of nodes in the hidden layer. Rather than use a traditional backpropagation algorithm, I have had considerable success with a more robust version called resilient backpropagation with backtracking[33]. This is selected by specifying `algorithm = "rprop+"`. By the way you can use traditional backpropagation if you wish by setting `algorithm ="backprop"`. If you use this option, you will also need to specify a learning rate (i.e. `learningrate =0.01`).

Note that `err.fct` is the error function; you can choose between the sum of squared errors (`err.fct = "sse"`) and cross-entropy (`err.fct = "ce"`). It is also necessary to specify the type of activation function we chose a logistic activation function.

The output node is influenced by the parameter `linear.output`. If set to `"TRUE"` the node's output is not transformed by the specified activation function; it is in essence linear. This is the linear activation function discussed on page 22.

Let's take a look at some of the details of the model. We can use the `print` method as follows:

```
> print(fit)
Call: neuralnet(formula = f, data = data[
   train, ], hidden = 1, algorithm = "rprop
   +",     err.fct = "sse", act.fct = "
   logistic", linear.output = FALSE)

1 repetition was calculated.
```

```
        Error Reached Threshold Steps
1 1854.659153    0.008971185555 15376
```

The first line gives the formula we typed in. It is always useful to review this line, often times what you thought you typed is different from what you actually typed. The second line reports the number of repetitions, in this case 1. We discuss the role of repetitions further on page 58. Finally, the output informs us the model converged after 15,376 steps with an error of 1854.

It is fun to visualize neural networks; this can be achieved using the `plot` method. If you use this option with the `neuralnet` package you will have a number of useful choices available. To show the fitted weights and intercept values on your visualization you set the `intercept` and `show.weights` parameters to `TRUE`. However, for the moment, we will just visualize the fitted topology using the `plot` method as follows:

```
> plot(fit,intercept = FALSE,
show.weights = FALSE)
```

Figure 2.5 displays the result. The image shows the six input nodes and their associate attribute names, the hidden layer node, and the output node associated with the response variable `Class`. At the bottom of the figure are the error estimate (1854) and number of steps required for the algorithm to converge (15,376).

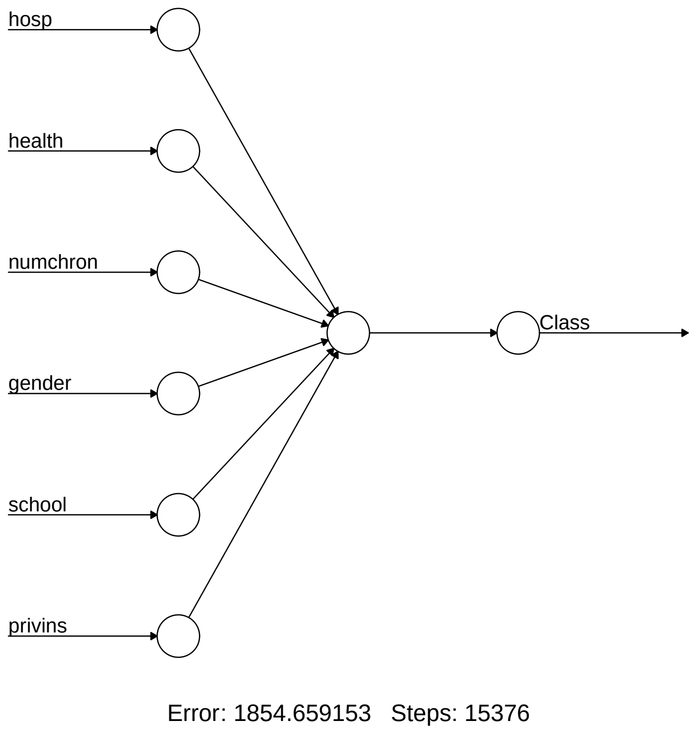
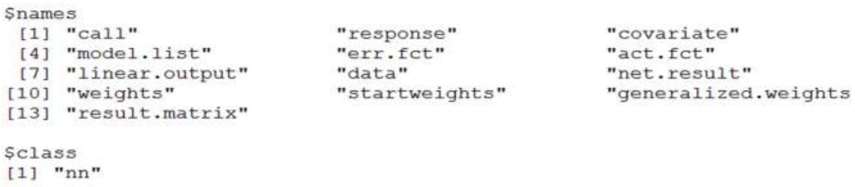


Figure 2.5: Fitted topology of MLP with 1 hidden node

All objects in R have associated attributes that can be accessed using the `attribute` method. We stored the results of the MLP in an object called `fit`. To see what `fit` contains we call the `attribute` method:

```
> attributes(fit)
```



```
$names
 [1] "call"               "response"           "covariate"
 [4] "model.list"         "err.fct"            "act.fct"
 [7] "linear.output"      "data"               "net.result"
[10] "weights"            "startweights"       "generalized.weights
[13] "result.matrix"

$class
[1] "nn"
```

Details of a particular attribute can be viewed using the $ operator. For example, a summary of the fitted network is contained in `result.matrix`. Let's take a look:

```
> fit$result.matrix
                                        1
error                    1854.659153417289
reached.threshold           0.008971185555
steps                   15376.000000000000
Intercept.to.1layhid1      -1.476505618519
hosp.to.1layhid1            0.936995772797
health.to.1layhid1         -0.308589104362
numchron.to.1layhid1        1.065004008360
gender.to.1layhid1         -0.190692443760
school.to.1layhid1          0.159883623634
privins.to.1layhid1         0.363787899579
Intercept.to.Class          1.648630276189
1layhid.1.to.Class        -40.196940113888
```

Notice, once again the error is given (1854), followed by the threshold and steps (15,376). This is followed by the estimated optimum weights. The synapse between `hosp` and the hidden neuron (`hosp.to.1layhid1`) has a weight of 0.936; whilst the synapse between gender to the hidden neuron (`gender.to.1layhid1`) has a weight of -0.19.

A nice feature of the `neuralnet` package is that we can also visualize these results using the plot method:

```
> plot(fit,intercept = TRUE,show.weights =
   TRUE)
```

The result is shown in Figure 2.6.

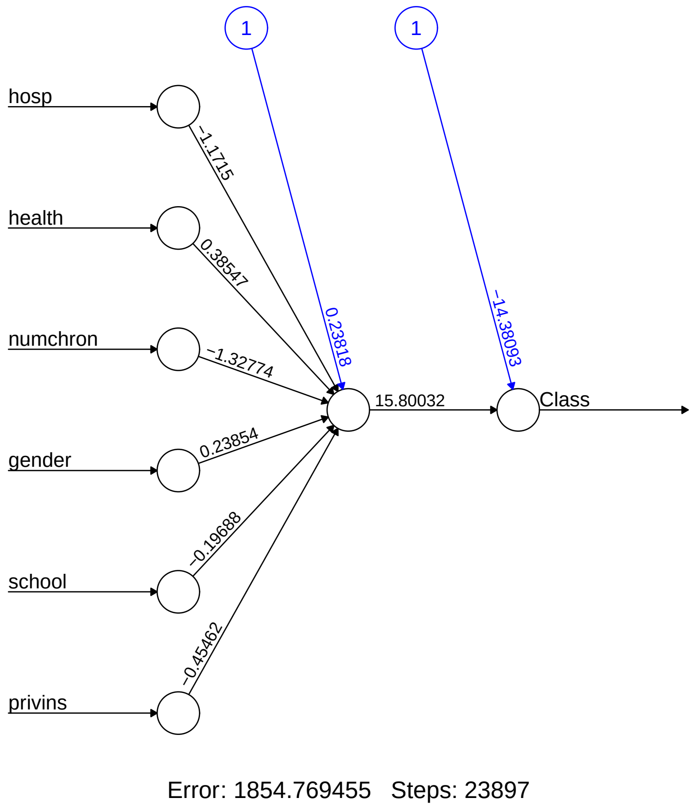


Figure 2.6: Fitted topology of MLP with 1 hidden node and weights

Predicting New Cases

Now we are ready to investigate how well the fitted model performs on the test sample. The `compute` method provides the means to do this:

```
> pred <- compute(fit, data[-train,2:7] )
```

We use the argument `data[-train,2:7]` to pass the test sample along with `fit` to the `compute` method. The element `-train` indicates we are using the testing sample[34]. The element 2:7 simple refers to the attributes which are contained in columns 2 to 7 of data.

To view the first few predictions use $net.result along with pred:

```
> head(pred$net.result ,6)
                         [,1]
1   0.000001149730617861
15  0.000004387003430401
16  0.666976073629486299
46  0.000000001263003749
76  0.000000469604180798
82  0.000217696667194658
```

These numbers can be viewed essentially as the probability of an individual belonging to the below median or above median group in Class. Let's convert them to the same -1, +1 scale as used in Class. We do this using the ifelse function with probabilities less than or equal to 0.5 being assigned to group -1 and the remainder assigned to group 1. Here is how to do this:

```
>  r2 <- ifelse(pred$net.result <= 0.5, -1,
    1)
> head(r2,6)
   [,1]
1    -1
15   -1
16    1
46   -1
76   -1
82   -1
```

Our next step is to calculate the confusion matrix; it contains information about actual and predicted classifications and is often used to assess the quality of a model's prediction. Here is how to do that:

```
> table(sign(r2),sign(data[-train,1]) ,
   dnn =c("Predicted" , " Observed"))
          Observed
Predicted   -1    1
       -1  205  157
       1    11   33
```

Of the 406 observations 205 were correctly classified as belonging to group -1, and 33 were correctly classified as belonging to group +1. The error rate is calculated measuring the misclassified observations as a proportion of the total:

```
> error_rate = (1- sum( sign(r2) == sign(
   data[-train,1])  )/406 )
>  round( error_rate ,2)
[1] 0.41
```

Overall, 41% of individuals were misclassified. This implies a prediction accuracy rate of around 59%.

Exercises

- **Question 1:** Re-build the model on page 46, but this time using six hidden nodes.
- **Question 2:** Re-estimate the model you developed in question 1, but using resilient backpropagation without back tracking.
- **Question 3:** Suppose a domain expert informed you only `hosp`, `health` and `numchron` were relevant attributes. Build a model with 2 hidden nodes using resilient backpropagation without back tracking.
- **Question 4:** Assess the performance of the model developed in questions 3 on the test sample.

Notes

[27]Wright, Daniel B. "Making friends with your data: Improving how statistics are conducted and reported." British Journal of Educational Psychology 73.1 (2003): 123-136.

[28]Deb, Partha, and Pravin K. Trivedi. "Demand for medical care by the elderly: a finite mixture approach." Journal of applied Econometrics 12.3 (1997): 313-336.

[29]The data came from the National Medical Expenditure Survey which seeks to provide a comprehensive picture of how Americans use and pay for health services. It is a representative, national probability sample of more than 38,000 individuals admitted to long-term care facilities.

[30]Deb and Trivedi used their sample to develop a finite mixture negative binomial count model that accommodates unobserved heterogeneity in data and excessive number of zeros to estimated various measures of medical care demand by the elderly.

[31]To see how to do this in see my book `92 Applied Predictive Modeling Techniques in R,` available at www.AusCov.com

[32]You are advised to become familiar with the package documentation which can be found `https://cran.r-project.org/web/packages/neuralnet/`.

[33]For additional details on resilient backpropagation see:

- Riedmiller M. (1994) Rprop - Description and Implementation Details. Technical Report. University of Karlsruhe.
- Riedmiller M. and Braun H. (1993) A direct adaptive method for faster backpropagation learning: The RPROP algorithm. Proceedings of the IEEE International Conference on Neural Networks (ICNN), pages 586-591. San Francisco.

[34]i.e. of the entire sample use only those elements not identified by `train`.

Chapter 3

Building MLPs With More Than One Layer

In this chapter we build on the R code of the previous chapter and delve head first into how to go about creating a model with more than one layer. Our goal is to explore whether adding an extra layer will reduce the overall misclassification error of the test sample.

During our exploration we answer the question of how many layers to include, explain the role of local minima, and demonstrate how to fit multiple layers in R. We also uncover some surprising results, and gain a little intuitive insight that will assist us in constructing MLPs more efficiently. Be prepared for both frustration and disappointment, and maybe some pleasant surprises. In any-case, I think I mentioned that Data Science is a series of failures punctuated by the occasional success, well this is never more apt than when building MLPs with more than one layer.

How Many Hidden Layers should I choose?

Since we are going to build an MLP with more than one layer, the natural question is "*how many layers should I choose?*" This is the very question I asked myself back in the spring of 1992 when I coded up my first MLP. It is probably a question you are asking yourself now as you read. Back then my answer was pragmatic - what is the smallest number greater than one that I can get away with? I was in a hurry to try out this amazing new technology, to see how it would perform relative to other statistical techniques and models derived from economic theory; plus, I had a thesis to complete! Each successive layer added complexity and therefore took more time to program and run. I settled on building models with at most two layers and so should you. Here is why:

A while back researchers Hornik et al.[35] showed that one hidden layer is sufficient to model any piecewise continuous function. Their theorem is a good one and worth repeating here:

> **Hornik et al. theorem**: Let F be a continuous function on a bounded subset of n-dimensional space. Then there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximate F arbitrarily well. Namely, for all x in the domain of F, $|F(x) - \hat{F}(x) < \epsilon|$.

This is a remarkable theorem. Practically, it says that for *any* continuous function F and some error tolerance measured by ε, it is possible to build a neural network with one hidden layer that can calculate F. This suggests, theoretically at least, that for very many problems, one hidden layer should be sufficient.

Now, before we get too carried away and think we are done, the Hornik et al. theorem has the word "*continuous*" in it. Two hidden layers are required to model data with discontinuities such as the saw tooth wave pattern shown in Figure 3.1.

There appears to be little theoretical reason for using more than two hidden layers; and in practice, for the most part, the higher the number of layers the greater the risk of converging to a local minima. So, in most situations, there is probably no need to stuff your network with tons of layers! (Unless of course you are into deep learning, but that amazing subject is the focus of another of my books).

Three layer models with one hidden layer is my recommended starting point. However, if you must go higher ask yourself what is the smallest number greater than one that you can get away with?

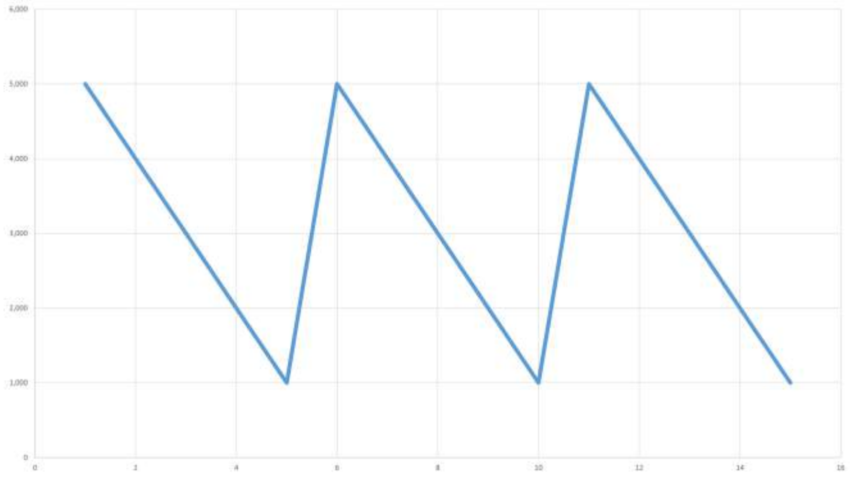


Figure 3.1: Saw tooth pattern

So how do you specify a model with more than one layer in R? To estimate more than one layer using the `neuralnet` package you would specify something like `hidden = c(`n_1`,`n_2`,...,`n_n`)`, where n_n is the number of nodes in the first layer and n_2 is the number of nodes in the second layer and so on[36].

Dealing With Local Minima

The neural network model we build in this chapter has over twenty weights whose values must be found in order 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. As we have previously observed, for any given set of input data and 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. Recall, the goal of learning is to find a set of weights that minimizes the mismatch between network outputs and actual target values.

It turns out that 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 3.2 (top panel). Whereas x^2 has only one minimum and $-x^2$ only one maximum, see Figure 3.2 (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. Figure 3.3 presents a highly simplified picture of this situation. It only represents a single weight value (on the horizontal axis). In the network we build in this chapter, you would actually observe a multiple dimension surface with many local valleys.

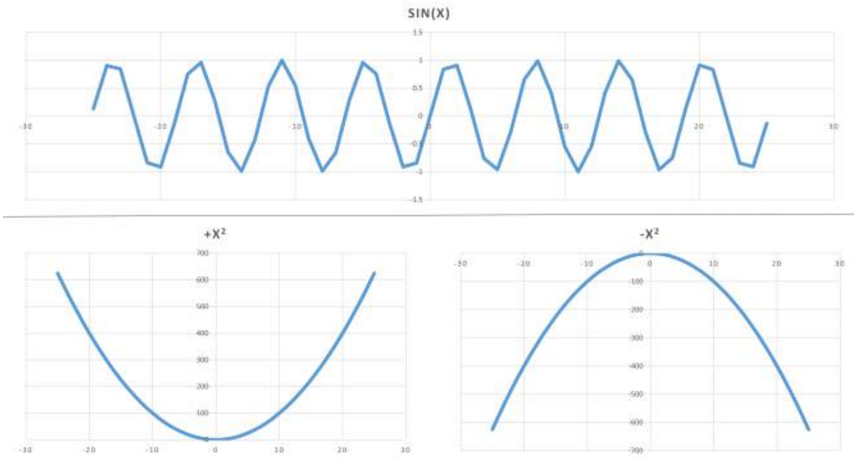


Figure 3.2: One variable functions sin(x), +x^2 and -x^2

Backpropagation can get stuck in a local minima especially if it begins its search in a valley near a local minimum. Several methods have been proposed to avoid local minima. The simplest, and the one we use here, is to try a number of random starting points and use the one with the best value. This can be achieved using the `rep` parameter in the `neuralnet` method. It specifies the number of random repetitions to be used in the neural network's training[37].

As you build your own neural network models, you will use the `rep` parameter often to give your models multiple starting points. You should set it to a large number; in exploratory analysis a value of 1000 might be a reasonable starting point.

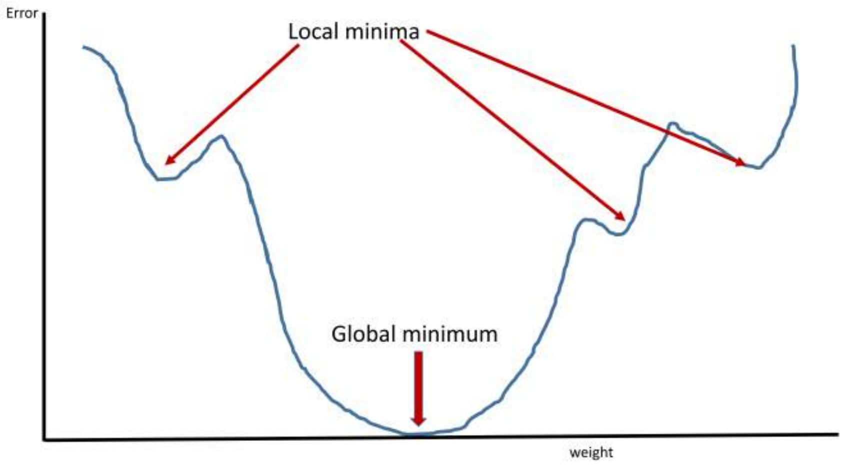


Figure 3.3: Error and network weight

Building the Model

Continuing on from where we left off in the last chapter let's try `hidden = c(2,2)`. This will fit two hidden layers each with two nodes. We set a maximum step size of 25,000 (`stepmax` =25000), and to illustrate the use of `rep` we set it to a small value of 10:.

```
>  fitM<- neuralnet(f, data = data[train,],
hidden=c(2,2),
stepmax = 25000,
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",
rep=10,
linear.output=FALSE)
```

After a short while you will see a message along the following lines:

```
Warning message:
```

```
algorithm did not converge in 9 of 10
   repetition(s) within the stepmax
```

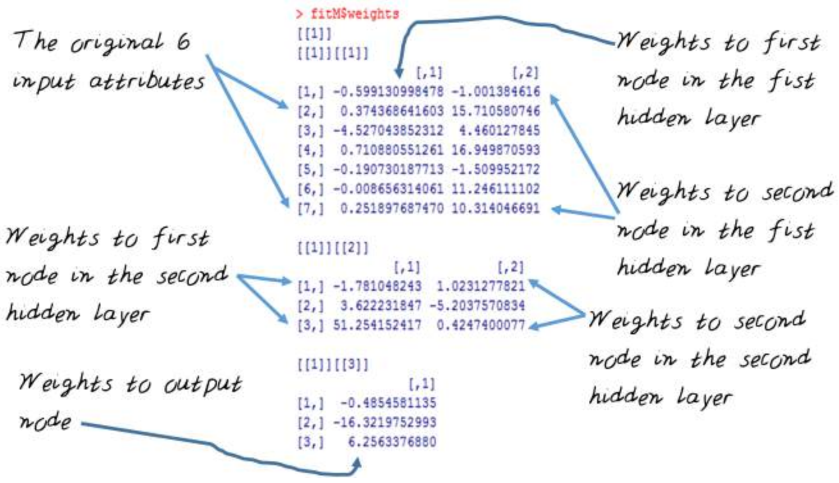
It appears for most of the epochs the algorithm failed to converge. In fact, only 1 out of the 10 runs converged to a solution within 25,000 steps. Why is this? Perhaps the algorithm descended very quickly at some steep valley, got to the opposite slope and then bounced back and forth failing to converge. Whatever the reason, the successful convergence rate was around 10%.

What can we do about this? Well, we could try fiddling around with the parameters in the model, maybe increase the step size, change the error function, select an alternative activation function or even specify a different algorithm. However, we have one success so let's work with that for now.

Understanding the Weight Vectors

The estimated weights are shown along with handwritten comments in Figure 3.4. You can see the six original input variables are listed by their column location plus 1. This is because the first row of each vector represents the intercept estimates. For example, the reported value of -0.599130998478 is the intercept estimate for the first node in the hidden layer; and the value 1.0231277821 is the intercept estimate of the second node in the second layer.

Notice how the weights are separated by layer. So the first block of numbers represents the weights into the first hidden layer, the second block of numbers capture the weights into the second hidden layer, and the third block of number measure the weights of the output node.


The original 6 input attributes
Weights to first node in the fist hidden layer
Weights to second node in the fist hidden layer
Weights to first node in the second hidden layer
Weights to second node in the second hidden layer
Weights to output node

```
> fitM$weights
[[1]]
[[1]][[1]]
             [,1]         [,2]
[1,] -0.599130998478 -1.001384616
[2,]  0.374368641603 15.710580746
[3,] -4.527043852312  4.460127845
[4,]  0.710880551261 16.949870593
[5,] -0.190730187713 -1.509952172
[6,] -0.008656314061 11.246111102
[7,]  0.251897687470 10.314046691

[[1]][[2]]
            [,1]         [,2]
[1,] -1.781048243  1.0231277821
[2,]  3.622231847 -5.2037570834
[3,] 51.254152417  0.4247400077

[[1]][[3]]
              [,1]
[1,]  -0.4854581135
[2,] -16.3219752993
[3,]   6.2563376880
```

Figure 3.4: Estimated weights for `fitM`

Can we interpret the weights in a similar way to linear regression? I use regression models when I need to make an assessment of the relative contribution of the attributes (in regression called independent variables or covariates) to the outcome variable (called the dependent variable in regression). In a one hidden layer neural network you might possibly be able to interpret high weights as indicative of positive outcomes and low weights as indicative of negative outcomes. MLPs with more than one layer can be difficult to interpret in this way. As the number of layers increase it becomes very difficult to try to understand what the different hidden neurons are doing[38].

The topology of the fitted model is illustrated in Figure 3.5. It took 14,394 steps to converge with an error of 1841.

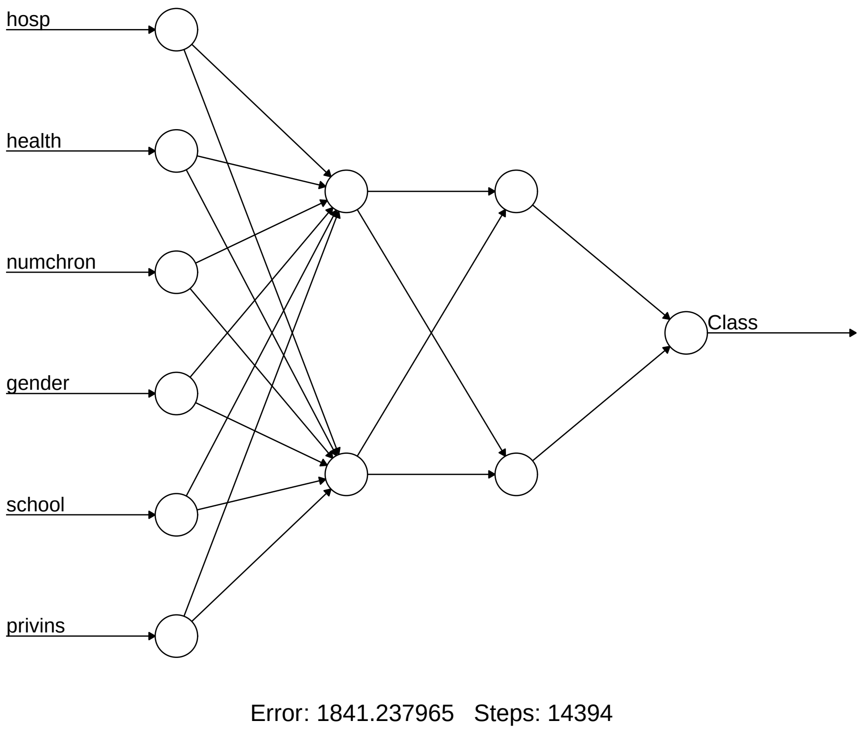


Figure 3.5: Two layer MLP

Predicting Class Outcomes

Now we are ready to predict using the test sample. We use the `compute` method and follow the same steps discussed earlier - save the prediction results in `pred`, scale to +1 and -1 storing the result in `rM`, calculate the confusion matrix and then the error rate. Remember that `data[-train,2:7]` specifies use of the test sample with all six attributes.

```
>  pred <- compute(fitM, data[-train,2:7] ,
   rep = 1)
```

```
>  rM <- ifelse(pred$net.result <= 0.5, -1,
    1)

> table( sign(rM),sign(data[-train,1]) ,
   dnn =c("Predicted" , " Observed"))
          Observed
Predicted  -1   1
       -1 200 159
       1   16  31

> error_rate = (1- sum( sign(rM) == sign(
   data[-train,1])  )/406 )

>  round( error_rate ,2)
[1] 0.43
```

The misclassification error rate is 43% and slightly higher than the single layer model we developed on page 53. As we indicated earlier in this chapter, using two or more hidden layers does not necessarily improve model performance; this appears to be true in this example. A little disappointing? Certainly, but then Data Science is a series of failures punctuated by the occasional success...

Exercises

- **Question 1:** Using the `DebTrivedi` data with `f<-Class~hosp + health + numchron + gender + school + privins`, fit an MLP with one hidden layer and one node in each layer. Assess the performance using the misclassification error.
- **Question 2:** What would you expect to happen to the number of training steps if you use the smallest learning rate algorithm (i.e. set `algorithm = "slr"`?

- **Question 3:** Re-run your analysis for question 1 using the original unscaled attributes and unscaled response variable.
- **Question 4:** fit the model on page 60 using the linear transformation function for the output node.

Notes

[35]See Hornik, M. Stichcombe, and H. White. Multilayer feedforward networks are universal approximators. Neural Networks, 2:359–366, 1989.

[36]In practice this package allows you to specify up to four layers.

[37]As a practical matter this simple strategy of randomly initializing the weights of the network and using backpropagation often finds poor solutions in neural networks with multiple (say 3 or more) hidden layers. It is probably the reason why so many useful MLPs are limited to one or two hidden layers.

[38]Various attempts have been made to open the so called neural network black-box and cast light on the interpretability of weights. Articles which discuss and outline ways to do this include:

1. Intrator, Orna, and Nathan Intrator. "Interpreting neural-network results: a simulation study." Computational statistics & data analysis 37.3 (2001): 373-393.

2. Paliwal, Mukta, and Usha A. Kumar. "Assessing the contribution of variables in feed forward neural network." Applied Soft Computing 11.4 (2011): 3690-3696.

3. Duh, Mei-Sheng, Alexander M. Walker, and John Z. Ayanian. "Epidemiologic interpretation of artificial neural networks." American journal of epidemiology 147.12 (1998): 1112-1122.

Chapter 4

Using Multiple Models

WHILST textbooks often focus on building a single MLP, in practice you will build very many models in your attempt to solve a particular problem. Much of the time you will find that very many of your MLPs will produce similar results with no one providing the best solution by a huge margin. In this chapter we build a variety of MLPs and explore how to combine MLP predictions.

MLP models are stochastic in their solution; that is different runs of a model will result in different weights and predictions. This variation could be large and therefore might influence the quality of the predictions on new unseen data. We illustrate how to capture the distribution of the misclassification error to help assess the quality of a selected MLPs predictions.

Estimating Alternative Models

Our first task is to rapidly specify and estimate three alternative neural network models. One strategy is to simply estimate multiple one layer MLPs with a range of hidden nodes. This is the strategy we employ. In addition to the models we have already developed we estimate three additional one hidden layer models, with the number of hidden nodes ranging from 1 to 4.

This can be achieved as follows:

```
> set.seed(103)

> fit1<- neuralnet(f, data = data[train,],
hidden=1,
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",rep=1)

> fit3<- neuralnet(f, data = data[train,],
hidden=3,
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",rep=1)

> fit4<- neuralnet(f, data = data[train,],
hidden=4,
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",rep=1)
```

Note that `fit1`, `fit3` and `fit4` contain the result of the fitted MLPs. We also continue to use resilient backpropagation with backtracking, sum of squares error function and a logistic activation function. To keep things simple, we keep the `rep` parameter set at 1.

Next, we collect together the predictions and store them in R objects, `pred1`, `pred3` and `pred4`:

```
> pred1 <- compute(fit1, data[-train,2:7] )

> pred3 <- compute(fit3, data[-train,2:7] )

> pred4 <- compute(fit4, data[-train,2:7] )
```

☛ ***PRACTITIONER TIP*** ☚

When naming a variable it is always useful to adopt meaningful (to you) names. If you have to review the code again in six months from now, your task will be much easier if the object names are intuitive.

Now we scale the predictions to match `Class`:

```
> r1<- ifelse(pred1$net.result <= 0.5, -1,
   1)

> r3<- ifelse(pred3$net.result <= 0.5, -1,
   1)

> r4<- ifelse(pred4$net.result <= 0.5, -1,
   1)
```

Combining Predictions

Let's combine the predictions of all the models we have developed so far into a single object called `agg_pred`:

```
> agg_pred<-cbind(r1,r2,r3,r4,rM)
```

Take a peek at the values inside (first ten rows using the `head` method):

```
> head(agg_pred,10)
     [,1] [,2] [,3] [,4] [,5]
1      -1   -1   -1   -1   -1
15     -1   -1   -1   -1   -1
16      1    1    1    1    1
46     -1   -1   -1   -1   -1
76     -1   -1   -1   -1   -1
```

```
82     -1    -1    -1    -1    -1
86     -1    -1    -1    -1    -1
93     -1    -1    -1    -1    -1
103    -1    -1    -1    -1    -1
105    -1    -1    -1    -1    -1
```

Since all of the models are using the same data, and differ primarily by the number of hidden nodes, we might expect them to have similar predictions. We can check the concordance between predictions with a little personal R coded function called **res**:

```
res <- aaply(agg_pred, 2, function(col) {
  aaply(agg_pred, 2, function(col2) {
    sum(col==col2)/length(col)
  })
})
```

PRACTITIONER TIP

It good practice to comment your R code as you develop it. This is achieved with the # operator. For example:

```
#store predictions of MLP
#with 1 hidden layer in pred1
> pred1 <- compute(fit1, data[-
   train,2:7] )
```

When we call **res** this is what we observe:

```
> round(res,3)

X1       1     2     3     4     5
  1 1.000 0.995 0.961 0.973 0.963
  2 0.995 1.000 0.956 0.968 0.958
  3 0.961 0.956 1.000 0.973 0.978
  4 0.973 0.968 0.973 1.000 0.975
  5 0.963 0.958 0.978 0.975 1.000
```

If there was zero agreement between two columns the value reported by `res` would be 0. If there is perfect agreement, the value reported would equal 1. Notice that the values reported are all greater than 0.95. So we conclude the agreement between all five models is very high.

How can we combine the separate predictions into an overall prediction? There are many ways to do this. We will use a form of majority voting. We take as the majority vote the median row value in `agg_pred`. This can be achieved as follows:

```
> rmeds <- apply(agg_pred, 1, median)
```

The object `rmeds` now contains the majority vote using all five models. Let's combine it into the `agg_pred` data frame and look at the first few observations:

```
> agg_pred<-cbind(r1,r2,r3,r4,rM,rmeds)

> colnames(agg_pred) <- c("r1","r2","r3","
  r4","rM","rmeds")

> head(agg_pred,10)
    r1 r2 r3 r4 rM rmeds
1   -1 -1 -1 -1 -1    -1
15  -1 -1 -1 -1 -1    -1
16   1  1  1  1  1     1
46  -1 -1 -1 -1 -1    -1
76  -1 -1 -1 -1 -1    -1
82  -1 -1 -1 -1 -1    -1
86  -1 -1 -1 -1 -1    -1
93  -1 -1 -1 -1 -1    -1
103 -1 -1 -1 -1 -1    -1
105 -1 -1 -1 -1 -1    -1
```

Now we are ready to calculate the confusion matrix and report the error rate using the majority vote:

```
> table( sign(agg_pred[,"rmeds"]),
sign(data[-train ,1]),
dnn = c("Predicted" , " Observed"))

          Observed
Predicted  -1   1
       -1 200 155
       1   16  35

> error_rate = (1- sum( sign(agg_pred[,"
   rmeds"]) == sign(data[-train ,1])  )/406
   )

>  round( error_rate ,2)
[1] 0.42
```

Overall, we see misclassification error rate of 42% which implies a prediction accuracy of around 58%.

A Practical Alternative to Backpropagation

When building MLPs it is useful to try other learning mechanisms. The nnet package builds MLPs using optimization based on Nelder–Mead, quasi-Newton and conjugate-gradient algorithm.

A one layer MLP with eight hidden nodes can be fitted using the nnet package as follows:

```
>  fitN<- nnet(f, data = data[train ,],size
   =8, rang = 0.1,decay = 5e-4, maxit =
   200,softmax=FALSE)
```

The argument size is the number of nodes, rang the initial random weights, decay the parameter for weight decay, maxit the maximum number of iterations, the softmax argument applies to multinomial responses since we have a binary response we set it to FALSE.

As the model is iterating you should see output along the lines of:

```
# weights:  65
initial  value 4986.262229
iter  10 value 4000.116514
iter  20 value 3997.511724
iter  30 value 3997.497784
              .
              .
              .
iter 190 value 3672.643548
iter 200 value 3672.414548
final  value 3672.414548
stopped after 200 iterations
```

The fitted model is stored in fitN. By typing the fitted models name we can view useful information:

```
>  fitN
a 6-8-1 network with 65 weights
inputs: hosp health numchron gender school
   privins
output(s): Class
options were - decay=0.0005
```

We can also view the weights using $wts. The first few weights are shown below:

```
>  fitN$wts
  [1]   -4.58871100719
  [5]   -6.02430895518
  [9]   14.35126627623
 [13]    6.25112643467
 [17]  -11.04323722943
 [21]    0.78767417187
```

Now let's predict using the fitted model and the test sample. This is easily achieved using the `predict` method:

```
> predN<-predict(fitN, data[-train,2:7],
   type = c("raw"))
```

Notice we set `type = c("raw")` to obtain individual probabilities. The first few are as follows:

```
> head(round(predN,3),7)
     [,1]
1   0.000
15  0.000
16  0.623
46  0.000
76  0.000
82  0.000
86  0.000
```

Finally, as before, we scale the values in `predN` to lie in the range -1 to +1, build the confusion matrix and calculate the error:

```
>  rN <- ifelse(predN <= 0.5, -1, 1)

> table( sign(rN),
sign(data[-train,1]),
dnn =c("Predicted", " Observed"))

          Observed
Predicted  -1   1
       -1 201 152
       1   15  38
> error_rate = (1- sum( sign(rN) == sign(
   data[-train,1])  )/406 )

>  round( error_rate ,2)
[1] 0.41
```

The misclassification error rate is similar to that observed using resilient backpropagation in the neuralnet package.

Bootstrapping the Error

It is often of value to obtain the distribution of error across multiple runs of a model. We can do this as follows. First we set up the initial parameters; boot contains the number of bootstraps we want to run, error will contain the misclassification error for each run of the model and i is a loop parameter:

```
> boot<-1000

> i=0

> error<-(1:boot)
> dim(error)
> dim(error) <- c(boot)
```

We run 1000 bootstraps using the following while loop:

```
while(i<=boot)
{

train <- sample(1:n, 4000, FALSE)

fitB<- nnet(f, data = data[train,],size =8,
    rang = 0.1,decay = 5e-4, maxit = 200,
   softmax=FALSE,trace=FALSE)

predB<-predict(fitB, data[-train,2:7], type
    = c("raw"))

rB <- ifelse(predB <= 0.5, -1, 1)

error[i] = (1- sum( sign(rB) == sign(data[-
   train,1])  )/406 )
```

```
  i=i+1
}
```

The result is visualized in the barplot shown in Figure 4.1. We observe the misclassification error ranges from a low of 0.32 to a high of 0.47, the median and mean are around 0.39.

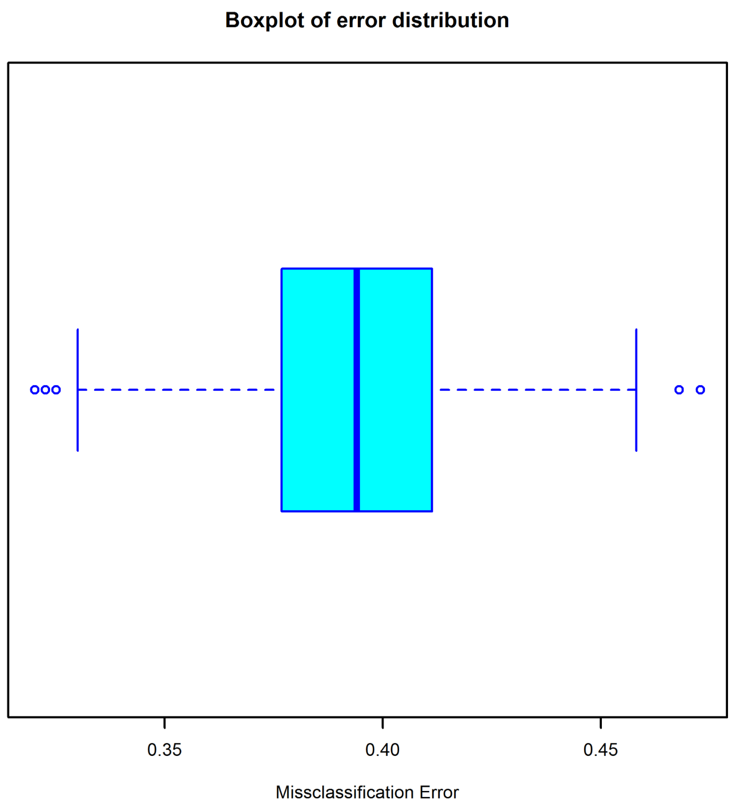


Figure 4.1: Distribution of error using `boot` =1000

Exercises

- **Question 1:** Use the `nnet` package to estimate a model with 1 hidden node and calculate the misclassification error.

- **Question 2:** Re-estimate the model you developed in question 1, but now using 2 nodes.
- **Question 3:** Assess the performance of the model developed in questions 3 on the test sample.
- **Question 4:** Use the bootstrap procedure to calculate the misclassification error distribution (mean, min, max and median) of the model you estimated in question 2.

Notes

[35]See Hornik, M. Stichcombe, and H. White. Multilayer feedforward networks are universal approximators. Neural Networks, 2:359–366, 1989.

[36]In practice this package allows you to specify up to four layers.

[37]As a practical matter this simple strategy of randomly initializing the weights of the network and using backpropagation often finds poor solutions in neural networks with multiple (say 3 or more) hidden layers. It is probably the reason why so many useful MLPs are limited to one or two hidden layers.

[38]Various attempts have been made to open the so called neural network black-box and cast light on the interpretability of weights. Articles which discuss and outline ways to do this include:

1. Intrator, Orna, and Nathan Intrator. "Interpreting neural-network results: a simulation study." Computational statistics & data analysis 37.3 (2001): 373-393.

2. Paliwal, Mukta, and Usha A. Kumar. "Assessing the contribution of variables in feed forward neural network." Applied Soft Computing 11.4 (2011): 3690-3696.

3. Duh, Mei-Sheng, Alexander M. Walker, and John Z. Ayanian. "Epidemiologic interpretation of artificial neural networks." American journal of epidemiology 147.12 (1998): 1112-1122.

Part II

Probabilistic Neural Network

THE Probabilistic Neural Network (PNN) has become one of the most popular forms of feedforward neural network. In this section of the book we describe the topology of PNNs, outline how they work and underscore why we should consider including them as instrument in our predictive analytic toolbox. To demonstrate the flexibility of this tool a wide variety of applications are discussed, ranging from leaf recognition to grading the quality of pearls.

We dig deep into the application of this method. Using R to generate rapid answers we walk step by step through the process of PNN model development and assessment. Here are the broad steps we will take:

1. Identify the appropriate packages;
2. clarify our modeling objective;
3. source the relevant data;
4. scrub the data;
5. select the smoothing parameter;
6. assess various alternative models;
7. classify the test sample and assess model performance.

Throughout the entire process you will be "knee deep" in practical matters surrounding both the use of R and the application of PNNs to real world data. The problem is demanding, the data messy and the best solution is not always obvious. Ready for the challenge? Then let's get started!

Chapter 5

Understanding PNNs

A PNN is a multilayered feed forward neural network that learns to approximate the probability density function of the training samples. It is a supervised learning algorithm but differs from the MLP in the way that learning occurs. As can be seen in Figure 5.1, the architecture of a PNN consists of four layers; the input layer, the pattern layer, the summation layer, and the output layer.

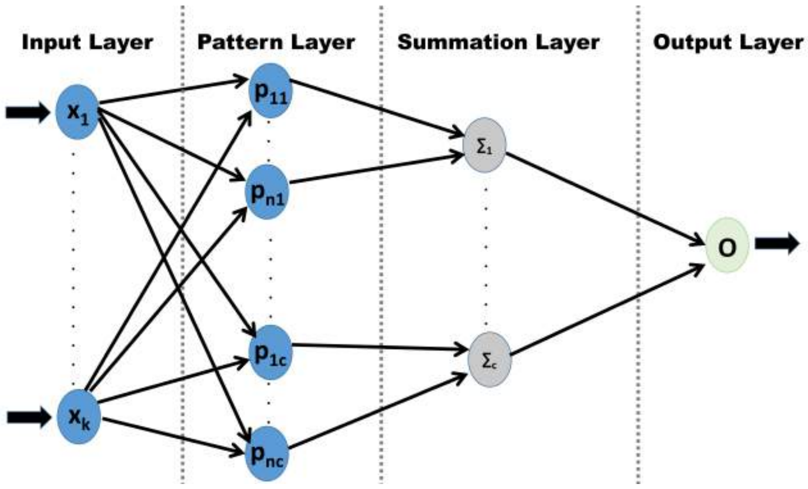


Figure 5.1: PNN network

The number of neurons in the input layer is determined by the number of attributes included in the training sample. If your input vector contains k attributes, then the input layer will contain k neurons. The input layer neurons do not perform any computations. They simply distribute the input to the neurons in the pattern layer.

The pattern layer is fully connected to the input layer. However, unlike the MLP, each pattern layer neuron represents the information contained in one training sample. If there are c classes in your data and n samples in the training set, there are a total of $n \times c$ pattern layer neurons. Each pattern layer neuron calculates the probability of how well it's sample input vector fits into the class represented by that neuron.

The third layer is the summation layer, which consists of as many neurons as the number of classes in the training data. If there are c classes, there will be c summation layer neurons. For each class of testing inputs the summation layer neurons sum the contributions of the previous layer output, and produce an output vector of probabilities related to the class they represent.

Finally, the output layer calculates the probabilities for each of the possible classes for which the input data can be classified.

How PNNs Work

Let's look at how this works in practice. Suppose an input vector of attributes $x=\{x_1,x_2\ldots,x_k\}$ is given to a PNN model.

- Step 1: The input layer distributes the input to the neurons in the pattern layer dividing them into c groups, one for each class.
- Step 2: The pattern layer neurons then compute their output. The activation function used in a pattern layer neuron is a probability density function. A Parzen window probability density function estimator with a Gaus-

sian kernel function is typical used. There are many kernel functions, but the Gaussian kernel function is most widely used in machine learning. For example, the pattern layer neuron $\mathrm{x_{ij}}$ computes its output using a Gaussian kernel of the form:

$$\phi_{ic}\left(x\right) = \frac{1}{\left(2\pi\sigma^2\right)^{\frac{k}{2}}}\, exp\left(-\frac{||x - x_{ic}||^2}{2\sigma^2}\right)$$

The parameter σ is known as the smoothing parameter and controls the spread of the Gaussian function.

- Step 3: The summation layer neurons sum the inputs from the pattern layers neurons that correspond to the class from which the training pattern was selected. This produces a vector of probabilities for each specific class:

$$f_c\left(x\right) = \textstyle\sum_{i=1}^{n} w_{ic} \times \phi_{ic}(x) \text{ where } \sum_{i=1}^{n} w_{ic} = 1.$$

- Step 4: The output layer neurons classify the pattern vector x using Bayes's decision rule:

$$C(x) = \arg\max_{i \leq c \leq C}\left(f_c\right)$$

Fundamentally a PNN is a Bayesian classifier machine that can map any input pattern to a number of classifications.

Why Use PNNs?

When I built my first neural network I used a MLP with backpropagation. Because the incremental adaptation time of back propagation was a significant fraction of the total computation time the MLP took around three and a half days to converge. I was not alone in experiencing the “long wait” often associated with backpropagation.

Researchers Marchette and Priebe[39] experienced a similar issue with backpropagation. They had a small data set consisting of 113 emitter reports of three continuous input parameters. The goal was to classify hulls into one of six possible groups. Because the data set was rather small a leave-one-out cross-validation was suggested. This was a good idea, especially for such a small sample. However, it proved infeasible as the researchers estimated it would take in excess of three weeks of continuous computing time on a Digital Equipment Corp. VAX 8650.

NOTE... ✍

Cross-validation refers to a technique used to allow for the training and testing of inductive models. Leave-one-out cross-validation involves taking out one observation from your sample and training the model with the rest. The predictor just trained is applied to the excluded observation. One of two possibilities will occur: the predictor is correct on the previously unseen control, or not. The removed observation is then returned, and the next observation is removed, and again training and testing are done. This process is repeated for all observations.

Maloney[40] investigated the same dataset. Leave-one-out cross-validation using a PNN took nine seconds[41]. Other researchers observed similar results. For example, Donald Specht of Lockheed Missiles & Space Company, built a PNN on the same dataset using a PC/AT 386 with a 20 MHz clock the time required was 0.7 seconds with better classification accuracy than a MLP which ran over several days. He commented: "*This compared to back-propagation running over the weekend which resulted in 82% accuracy, this result again represents a* *speed improvement of 200,000 to 1 with slightly superior accuracy.*"

PNNs are fast because they operate completely in parallel without a need for feedback from the individual neurons to the inputs. Their probabilistic approach to classify data is essentially learning in one-step. Therefore, a minimal amount of time is required to train a network. This is also an important advantage when adding new information. For MLPs new information requires retraining and is therefore computationally expensive. New information can be added to PNNs with almost no retraining allowing them to be used in real time knowledge discovery.

PNNs also generally work well with a few training samples, are relatively insensitive to outliers, and are guaranteed to converge to the Bayes optimal classifier as the size of the training set increases[42]. They also generate classification probability scores, and do not suffer from the local minima issues associated with MLPs.

Practical Applications

When I was little, I'd sit and watch television for hours. In between the regular programming, loud, noisy, flashy advertisements boomed out. One of the phrases that stuck with me is "try before you buy". You have probably heard that phrase yourself. Perhaps, you can see the salesperson in the advertisement now as you read; pointing their finger at the camera, smiling and saying "try before you buy, you'll love it!" Well, let's take a look at some of those who have brought the PNN technology. It seems the application of this neural network technique to really useful problems is vast.

Leaf Recognition

Motivated by the need to automate the process of plant recognition Wu et al.[43] develop a PNN for automated leaf recognition. A total of 12 leaf features were used, with the majority extracted automatically from digital images. The leaf features were then orthogonalized by Principal Components Analysis into five principal input attributes. The PNN was trained by 1800 leaves to classify 32 kinds of plants.

The researchers observe that using the training sample, for some species, the PNN correctly classified 100% of the samples. For example, for the Chinese horse chestnut (*Aesculus chinensis*) 63 training samples were used with all correctly classified. However, for other species the training error was relatively large, for example for the species sweet osmanthus (*Osmanthus fragrans Lour*) 55 training samples were used with 5 incorrectly classified.

Overall, The PNN achieved an accuracy around 90%. The researchers observe that the ultimate goal is use this technology in order to develop a database for plant protection. The first step toward this objective, the researchers argue, is to teach a computer how to classify plants.

Emotional Speech in Children

Kumar and Mohanty[44] use a PNN to classify emotional speech in children. Emotions are classified as - boredom, angry, sad and happy. They use a database of 500 utterances collected in three different languages (English, Hindi, and Odia). The researchers use a variety of vocal feature extraction techniques such as the Hurst parameter and time-frequency pH features.

The data is randomly partitioned into training, validation and testing sets. A total of 60% of the observations are used for training, and 20% used for validation and testing respectively.

Kumar and Mohanty use the validation set primarily to find the optimal sigma (smoothing parameter). A range of values were tested by trial and error including 0.25, 0.42 and 1.1. The value 1.1 was determined to be the optimal. Using this value, they observe an overall average classification error of around 12.7%. The researchers conclude: "*The results will be helpful for the later research on emotion classification.*"

Grading Pearl Quality

A pearl grader has to determine quality and assign a grade. Features that are considered include mollusk species, nacre thickness, luster, surface, shape, color and pearl size. A pearl grader has to quantify visual observations and to assign a grading level to a pearl. Kustrin and Morton[45] use a PNN to predict pearl quality parameters from ultraviolet–visible reflectance spectra. The sample consisted of twenty eight naturally-colored cultured pearls obtained from commercial pearl farms, 11 freshwater pearls from Zhuji (Zhejiang, China), 4 Akoya pearls from Japan, 5 Tahitian pearls from the South pacific and 8 pearls from a farm in Bali, Indonesia.

The pearls were scanned and their UV spectra collected. The spectra were acquired at two different locations on each pearl in order to assess surface homogeneity. The collected spectral data, suitably transformed, were used as inputs into the PNN with the pearl quality assessment parameters (mussel species, pearl color and shape complexity, donor color, and donor condition) as separate categorical outputs.

The researchers report the PNN successfully classified and graded 28 different pearls with 25 out of 28 mollusk species correctly classified and 90% of pearl color correctly predicted. In the case of donor color, 1 prediction was misclassified, and for luster and surface complexity 2 observations were misclassified.

Classification of Enzymes

Enzymes are biological catalysts which serve as important inputs in many industrial process. Khan et al.[46] use a PNN to classify enzymes. Two benchmark datasets were used, the first contained 217 enzymes consisting of 105 acidic enzymes and 112 alkaline enzymes. The second dataset was composed of 122 enzymes of which 54 were acidic enzymes and 68 alkaline enzymes.

Two approaches are used to extract features from the datasets. Pseudo amino acid composition (PseAAC) and split amino acid composition (SAAC). The features extracted from these methods were fed into two separate PNN models.

The success rates of classifiers using the first dataset was 94.5% using SAAC, and 96.3% for PseAAC. For the second data set the success rates were 92.6% using SAAC, and 99.2% for PseAAC.

The researchers conclude by stating "*In this research study, we have established an effective model for discriminating acidic and alkaline enzyme using two discrete protein sequence representation methods including PseAAC and SAAC*"

Notes

[39]Marchette, D., and C. Priebe. "An application of neural networks to a data fusion problem." Proceedings, 1987 Tri-Service Data Fusion Symposium. Vol. 1. 1987.

[40]Maloney, P. S. "An application of probabilistic neural networks to a hull-to-emitter correlation problem." 6th Annual Intelligence Community AI Symposium, Washington, DC. 1988.

[41]On an IBM PC running at (8 MHz).

[42]A Bayes optimal classifier is a classifier that minimizes a certain probabilistic error measure (often called the risk).

[43]Wu, Stephen Gang, et al. "A leaf recognition algorithm for plant classification using probabilistic neural network." Signal Processing and Information Technology, 2007 IEEE International Symposium on. IEEE, 2007.

[44]Palo, Hemanta Kumar, and Mihir Narayan Mohanty. "Classification of Emotional Speech of Children Using Probabilistic Neural Network." International Journal of Electrical and Computer Engineering (IJECE) 5.2 (2015): 311-317.

[45]Kustrin, S. A., and D. W. Morton. "The use of probabilistic neural network and UV reflectance spectroscopy as an objective cultured pearl quality grading method." Modern Chemistry and Applications 3.152 (2015): 2.

[46]Khan, Zaheer Ullah, Maqsood Hayat, and Muazzam Ali Khan. "Discrimination of acidic and alkaline enzyme using Chou's pseudo amino acid composition in conjunction with probabilistic neural network model." Journal of theoretical biology 365 (2015): 197-203.

Chapter 6

Classification Using a PNN

Zig Ziglar was fond of saying "*You can have everything in life you want, if you will just help enough other people get what they want.*"[47] It was an ancient lesson which the oracles of Greek antiquity understood well. People would flock from far and wide to hear the sage words of the oracle. Why? Because they wanted to know what was going to happen in their lives in the future. They sought guidance on difficult issues. Senior government officials, governors and even kings sought them out. In the process of serving these people the oracles made a pretty good living!

Whilst PNNs are no oracle, they are pretty good at classification and prediction tasks. In this chapter we build a PNN to help classify the diabetes status of women. Being able to accurately, quickly and easily discern the future was once the province of the oracle in ancient Greece. The employment of PNNs in your line of research or work offers you one way to help many people get what they want.

Loading the Appropriate Packages

A probabilistic neural network can be built using the `pnn` package. If you don't have a copy on your machine use the `install.packages` command to get a copy. Type something like this:

```
> install.packages("pnn")
```

Once the package has installed, be sure to load it into your current R session:

```
> library(pnn)
```

Getting the Data

We will build a PNN using the `PimaIndiansDiabetes2` data frame contained in the `mlbench` package. This dataset was collected by the National Institute of Diabetes and Digestive and Kidney Diseases[48]. It contains 768 observations on 9 variables measured on females at least 21 years old of Pima Indian heritage. Table 5 contains a description of each variable collected.

The data can be loaded into your R session as follows:

```
> data("PimaIndiansDiabetes2",package="
   mlbench")
```

Let's do a quick check to ensure we have the expected number of columns and rows:

```
> ncol(PimaIndiansDiabetes2)
[1] 9
> nrow(PimaIndiansDiabetes2)
[1] 768
```

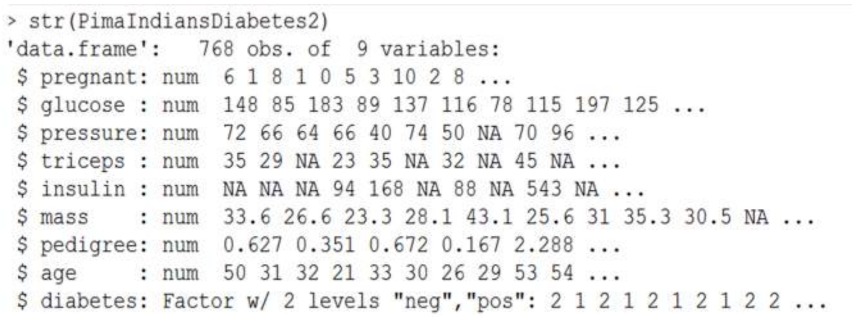
The numbers are in line we what we expected.

Name	Description
`pregnant`	Number of times pregnant
`glucose`	Plasma glucose concentration
`pressure`	Diastolic blood pressure (mm Hg)
`triceps`	Triceps skin fold thickness (mm)
`insulin`	2-Hour serum insulin (mu U/ml)
`mass`	Body mass index
`pedigree`	Diabetes pedigree function
`age`	Age (years)
`diabetes`	test for diabetes - Class variable (neg / pos)

Table 5: Response and independent variables in `PimaIndiansDiabetes2` data frame

Dealing with Missing Values

Now we can use the `str` method to check and compactly display the structure of the `PimaIndiansDiabetes2` data frame:

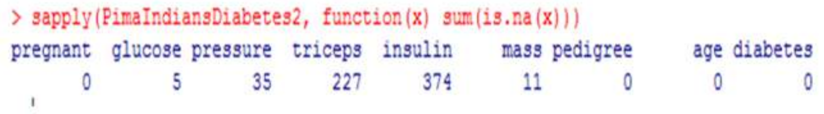


```
> str(PimaIndiansDiabetes2)
'data.frame':   768 obs. of  9 variables:
 $ pregnant: num  6 1 8 1 0 5 3 10 2 8 ...
 $ glucose : num  148 85 183 89 137 116 78 115 197 125 ...
 $ pressure: num  72 66 64 66 40 74 50 NA 70 96 ...
 $ triceps : num  35 29 NA 23 35 NA 32 NA 45 NA ...
 $ insulin : num  NA NA NA 94 168 NA 88 NA 543 NA ...
 $ mass    : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 NA ...
 $ pedigree: num  0.627 0.351 0.672 0.167 2.288 ...
 $ age     : num  50 31 32 21 33 30 26 29 53 54 ...
 $ diabetes: Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
```

As expected `PimaIndiansDiabetes2` is identified as a data frame with 768 observations on 9 variables. Notice that each row provides details of the name of the attribute, type of attribute and the first few observations. For example, `diabetes` is a factor with two levels “neg” (negative) and “pos” (positive). We will use it as our classification variable.

You may have also notice the `NA` values in `pressure`, triceps, insulin and mass. These are missing values. We all face the problem of missing data at some point in our work. People refuse or forget to answer a question, data is lost or not recorded properly. It is a fact of modeling life!

There seem to be rather a lot, we better check to see the actual numbers:



```
> sapply(PimaIndiansDiabetes2, function(x) sum(is.na(x)))
pregnant  glucose pressure  triceps  insulin     mass pedigree      age diabetes
       0        5       35      227      374       11        0        0        0
```

Wow! there are a large number of missing values particularly for the attributes of insulin and triceps. How should we deal with this? The most common method and the easiest to apply is to use only those individuals for which we have complete information. An alternative is to impute with a plausible value the missing observations. You might replace the NA's with the attribute mean or median. A more sophisticated approach would be to use a distributional model for the data (such as maximum likelihood and multiple imputation)[49].

Given the large number of missing values in insulin and triceps we remove these two attributes from the sample and use the na.omit method to remove any remaining missing values. The cleaned data is stored in temp:

```
> temp<-(PimaIndiansDiabetes2)
> temp$insulin  <- NULL
> temp$triceps <- NULL
> temp<-na.omit(temp)
```

We should have sufficient observations left to do meaningful analysis. It is always best to check:

```
> nrow(temp)
[1] 724
> ncol(temp)
[1] 7
```

So we are left with 724 individuals and (as expected) 7 columns.

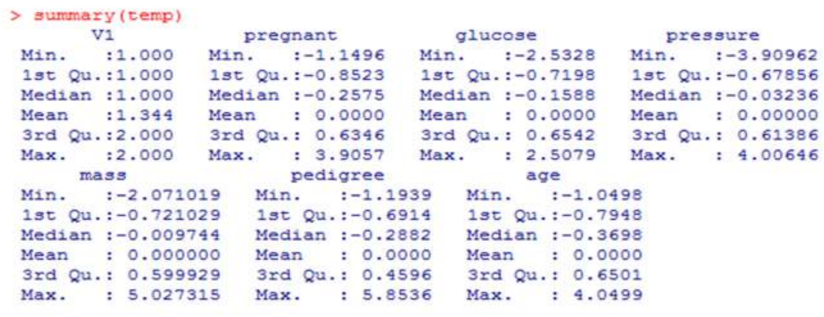
Now, watch this next move very closely! We store the response variable in an R object called `y` and remove it from `temp`. Notice that now the matrix `temp` only contains the covariate attributes. The `scale` method is used to standardize the attribute data; and then we combine `y` (as a factor) back into temp:

```
> y<-(temp$diabetes)
> temp$diabetes<-NULL

> temp<-scale(temp)

> temp<-cbind(as.factor(y),temp)
```

Check to ensure that `class` is of a matrix type:

```
> class(temp) [1] "matrix"
```

You can also use the `summary` method. You should see something like this:



```
> summary(temp)
       V1              pregnant           glucose            pressure
 Min.   :1.000   Min.   :-1.1496   Min.   :-2.5328   Min.   :-3.90962
 1st Qu.:1.000   1st Qu.:-0.8523   1st Qu.:-0.7198   1st Qu.:-0.67856
 Median :1.000   Median :-0.2575   Median :-0.1588   Median :-0.03236
 Mean   :1.344   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.00000
 3rd Qu.:2.000   3rd Qu.: 0.6346   3rd Qu.: 0.6542   3rd Qu.: 0.61386
 Max.   :2.000   Max.   : 3.9057   Max.   : 2.5079   Max.   : 4.00646
      mass              pedigree             age
 Min.   :-2.071019   Min.   :-1.1939   Min.   :-1.0498
 1st Qu.:-0.721029   1st Qu.:-0.6914   1st Qu.:-0.7948
 Median :-0.009744   Median :-0.2882   Median :-0.3698
 Mean   : 0.000000   Mean   : 0.0000   Mean   : 0.0000
 3rd Qu.: 0.599929   3rd Qu.: 0.4596   3rd Qu.: 0.6501
 Max.   : 5.027315   Max.   : 5.8536   Max.   : 4.0499
```

Finally, we select the training sample. Let's use 600 out of the 724 observations to train the model. This can be achieved as follows:

```
> set.seed(103)
> n=nrow(temp)
> n_train <- 600
> n_test<-n-n_train

> train <- sample(1:n, n_train, FALSE)
```

Building the Model

Now that we have our data in place and in a suitable format we can build the model. This is achieved using the `learn` method as follows:

```
> fit_basic <- learn(data.frame(y[train],
temp[train,]))
```

Notice we pass the response variable `y` to the `learn` method first, followed by the attribute matrix `temp`. The result is stored in the R object `fit_basic`.

We are ready to fit the model, which requires the specification of a smoothing parameter and the use of the `perf` method. First we use the `smooth` method with sigma (our smoothing parameter) set to 0.5. Followed by passing `fit` to the `perf` method:

```
> fit <- smooth(fit_basic,sigma=0.5)
> result<- perf(fit)
```

Wow! That was pretty quick (compared to the MLPs of Part I). Let's run the code again, this time measuring the time it took using the `system.time` function.

```
> system.time (perf(fit))
   user  system elapsed
  22.62    0.00   22.62
```

Even on the ancient computer I am using to code this section it only took around 23 seconds. The time taken on your computer will probably be much faster. Unless of course, you collect old machines like me!

How to Choose the Optimal Smoothing Parameter

The only parameter that needs to be selected prior to training a PNN is the smoothing factor. Care has to be taken in the value chosen; too small will result in very spiky approximation which cannot generalize well; too large tends to smooth out details. So how is it chosen? It most cases by trial and error. For example, Kumar and Mohanty (see page 88) use a PNN to classify emotional speech in children. They tested a range of values including 0.25 and 0.42 before settling on the value of 1.

The value of 0.5 we choose was our first take. Let's try two other guesses, a value of 1 and a value of 1.5:

```
>fit1 <- smooth(fit_basic,sigma=1)
>fit2 <- smooth(fit_basic,sigma=1.5)
```

As before we store the results:

```
> result1<- perf(fit)
> result2<- perf(fit)
```

As an alternative to simply guessing, we can use an optimization algorithm of some sort. The `pnn` packages allows you to do this through the `smooth` method. It actually makes use of the optimization routine in the `rgenoud` package[50]. The routine, developed by Walter et al[51], combines evolutionary search algorithms with derivative-based (Newton or quasi-Newton) methods. It is can be useful for solving problems that are nonlinear or perhaps even discontinuous in the parameters of the function to be optimized. Let's give it a go. Here is how:

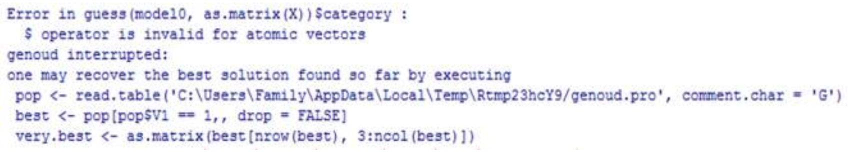
```
> set.seed(103)
> sigma<-smooth(fit,limits = c(0, 10))
```

The `limits` parameter is set to keep the search range for the smoothing between 0 and 10. Now this may take several minutes to run. At some point you will see the message:

```
Error in guess(model0, as.matrix(X))$
   category :
  $ operator is invalid for atomic vectors
genoud interrupted:
```

Alas, the algorithm failed to converge. What can we do about this? Did I ever tell you that Data Science is a series of failures punctuated by the occasional success...I guess by now you get that!

Well, more often than not optimization techniques fail; this is a practical fact that is rarely discussed in the text books or journal articles. We just have to deal with it. Fortunately, the developer of this algorithm understood this reality, so all is not totally lost. Take a close look at the R output and you will see the message:



```
Error in guess(model0, as.matrix(X))$category :
  $ operator is invalid for atomic vectors
genoud interrupted:
one may recover the best solution found so far by executing
 pop <- read.table('C:\Users\Family\AppData\Local\Temp\Rtmp23hcY9/genoud.pro', comment.char = 'G')
 best <- pop[pop$V1 == 1,, drop = FALSE]
 very.best <- as.matrix(best[nrow(best), 3:ncol(best)])
```

It provides instructions on how to recover a value if the algorithm fails. The first line tells us that we can recover the best solution found so far. The second line gives the file location. Note this location will differ on your computer. On windows machines we have to use "/" rather than "\", so we will adjust that. As an alternative you could always directly search for the file genoud.pro and then copy the file location. Let's follow its instructions:

```
>  pop <- read.table
('C:/Users/Family/AppData/Local/Temp/
   RtmpaIYCas/genoud.pro'
, comment.char = 'G')

> best <- pop[pop$V1 == 1,, drop = FALSE]
> very.best <- as.matrix(best[nrow(best),
3:ncol(best)])

> very.best
         [,1]
[1,] 0.51134
```

It seems on this run the optimal value is around 0.51[52], which is close to our first guess of 0.5. However, because the algorithm did not fully execute, on this occasion, we stick with our guesstimate of 0.50.

Assessing Training Sample Performance

At this stage we have three models to compare, `fit`, `fit1` and `fit2`. The performance statistics of the fitted model are assessed using the `$` operator. For example, enter the following to see first few observations and predicted values of `fit`:

```
> head(result$observed,5)
[1] pos pos pos pos neg
Levels: neg pos

> head(result$guessed,5)
[1] pos pos pos neg neg
Levels: neg pos
```

You can also view the number of observations successfully classified and the number misclassified:

```
> result$success
[1] 576

>  result$fails
[1] 24
```

Since the PNN model is essentially a Bayesian classifier machine, we can view the Bayesian information criterion[53] (`bic`):

```
>  result$bic
[1] -1885.546
```

This is a measure of overall goodness of fit of a model. In practice, you would estimate a number of models (as we did in chapter 4) and use this criteria to select the fitted candidate model corresponding to the minimum value of `bic`. This is the model corresponding to the highest Bayesian posterior probability.

Finally, we should take a look at the overall success rate for the training data:

```
>  result$success_rate
[1] 0.96
```

The model appears to fit the data very well, with a success rate of 96%.

Let's look at the `bic` and success rate for `fit1` and `fit2`:

```
>  result1$bic
[1] -1209.939

>  result2$bic
[1] -1047.552
```

Both of these values are larger than the `bic` value of `fit`. So we select `fit` as our model of choice. Just as a cross check, let's also look at the success rate:

```
>  result1$success_rate
[1] 0.8766667

>  result2$success_rate
[1] 0.8383333
```

It appears the success rate on `fit` at 96% is considerably higher than either of the two models.

There is always a danger in overfitting PNN models. This risk increases the smaller the value of the smoothing parameter. Given, the `bic` value and the success rate, out gut instinct is that the results will be generalizable to the test sample. However, our gut can be wrong!

Assessing Prediction Accuracy

We begin by predicting the first response value in the test sample. To do this we use the `guess` method. It takes the PNN model contained in `fit` and the training data. We use the `$category` parameter to show the prediction in terms of the classes contained in the response variable:

```
> guess(fit, as.matrix(temp[-train,][1,]))$
   category
[1] "neg"
```

So the prediction for the first women in the test sample in terms of diabetes outcome is negative. Was this actually what occurred? Let's take a look:

```
> y[-train][1]
[1] neg
Levels: neg pos
```

Yes, it appears our PNN was correct with the first individual. As a matter of curiosity, you might be interested in the value of the attributes of this woman. Here they are:

```
> round(temp[-train,][1,],2)
         pregnant  glucose pressure
    1.00    -0.85    -1.07    -0.52


mass pedigree      age
-0.63    -0.93    -1.05
```

It is interesting that the scaled score for this individual are all negative. If there was time we could investigate a little further.

Now let's take a look at the associated probabilities. We can access these using the `$probabilities` parameter:

```
> guess(fit, as.matrix(temp[-train,][1,]))$
   probabilities
        neg          pos
0.996915706 0.003084294
```

In this case there is over a 99% probability associated with the neg class. The PNN was pretty confident that this women was at a negative risk of diabetes. If, for some reason, you want to see both prediction classes and associated probabilities simply enter:

```
> guess(fit, as.matrix(temp[-train,][1,]))
$category
[1] "neg"

$probabilities
        neg          pos
0.996915706 0.003084294
```

OK, now we are ready to predict all the response values in the test sample. We can do this with a few lines of R code:

```
> pred<-1:n_test
> for (i in 1:n_test)
 {
 pred[i]<-guess(fit,
as.matrix(temp[-train,][i,]))$category
 }
```

The code simply loops through the test sample using the `guess` method for each observation and storing the result in `pred[i]`.

Let's create a confusion matrix, so we can see how well the PNN performed on the test sample.

```
> table( pred,y[-train] ,
dnn =c("Predicted" , " Observed"))

         Observed
Predicted neg pos
      neg  79   2
```

```
        pos    2   41
```

The numbers look pretty good. Only four misclassified observations. Let's calculate the error rate:

```
> error_rate = (1- sum( pred == y[-train])
    / n_test)
>  round( error_rate ,3)
[1] 0.032
```

The misclassification error rate is around 3.2% which implies a prediction accuracy of around 96.8%. A tad higher than that observed in the training sample - not bad at all!

Exercises

- **Question 1:** Assess the performance of `fit1` using the test sample.
- **Question 2:** Assess the performance of `fit2` using the test sample.
- **Question 3:** Re-build the PNN model with a smoothing parameter of 0.1 and assess the fit on the training sample.
- **Question 4:** Assess the performance of the model built in question 3 using the testing sample.

Notes

[47]See Ziglar, Zig. See You at the Top. Pelican Publishing, 2010.

[48]See http://www.niddk.nih.gov/

[49]For alternative approaches to dealing with missing values see:

1. Roth, Philip L. "Missing data: A conceptual review for applied psychologists." Personnel psychology 47.3 (1994): 537-560.
2. Afifi, A. A., and R. M. Elashoff. "Missing observations in multivariate statistics I. Review of the literature." Journal of the American Statistical Association 61.315 (1966): 595-604.
3. Pigott, Therese D. "A review of methods for missing data." Educational research and evaluation 7.4 (2001): 353-383.
4. Little, Roderick J., et al. "The prevention and treatment of missing data in clinical trials." New England Journal of Medicine 367.14 (2012): 1355-1360.

[50]For additional documentation see http://sekhon.berkeley.edu/rgenoud .

[51]See Walter Mebane, Jr. and Jasjeet S. Sekhon. 2011."Genetic Optimization Using Derivatives: The rgenoud package for R." Journal of Statistical Software, 42(11): 1-26.

[52]The number you observe may be different from this value.

[53]The Bayesian information criterion is often called the Schwarz information criterion after its creator. See Schwarz, Gideon. "Estimating the dimension of a model." The annals of statistics 6.2 (1978): 461-464.

Part III

Generalized Regression Network

English Statistician George E. P. Box's once said "*Statisticians, like artists, have the bad habit of falling in love with their models.*" I remind myself of this quote frequently; and so should you. Once you have successfully applied the Generalized Regression Neural Network[54] (GRNN) to a problem, like the alcoholic visiting the local pub, you may have a strong tendency to revisit it more frequently than is good for you!

GRNNs have proven popular in regression type forecasting problems due to their speed and accuracy. In chapter 7 the value of GRNNs for linear and non-linear regression problems is emphasized. Details are given of the role of each layer in a GRNN, and practical applications ranging from the design of DC motors to the modeling of gaseous pollutants produced by swine production plants.

We get our hands on practical R code in chapter 8, where we build a regression style model using a GRNN to predict body fat using measurements of skinfold thickness, circumferences, and bone breadth as our primary input attributes.

Chapter 7

The GRNN Model

It was Epictetus who wrote *"The key is to keep company only with people who uplift you, whose presence calls forth your best."* What holds true for people, in this instance, also holds true for modeling techniques. Keep in your toolkit only those techniques that allow you to generate the very bests insights from your data. GRNNs are one such tool. In this chapter we outline what GRNNs are used for, discuss their topology and describe the role and function of each network layer. Finally, to illustrate their flexibility, we outline their use in surface radiation measurement, swine gas production and the design of brushless DC motors.

What is GRNN Used For?

GRNNs are used to estimate linear or nonlinear regression where the dependent variable is continuous. They are useful for regression modeling problems because they have strong non-linear mapping capability and can be computed relatively quickly. They are robust to outliers and can solve any function approximation problem if sufficient data is provided. In addition, they tend to perform well in terms of prediction accuracy, even with small samples.

Suppose we have n observations on an attribute x and the associated dependent variable y. The expected value of y given x is given by:

$$E\left[y|x\right]=\frac{\int_{-\infty}^{+\infty}yf(x,y)dy}{\int_{-\infty}^{+\infty}f(x,y)dy},$$

where $f\left(x,y\right)$is the joint probability density function of x and y.

Since $f\left(x,y\right)$ is unknown the GRNN estimates it from the sample values. In practice we will have a sample of n observations on a vector of k attributes $\mathrm{X}=[\mathrm{X}_1,\mathrm{X}_2\ldots,\mathrm{X}_\mathrm{k}]^\mathrm{T}$ along with the associated dependent variable Y.

The Role of each Layer

GRNNs have four layers, namely an input layer, a pattern layer, a summation layer (which contains two neurons) and a output or decision layer, as shown in Figure 7.1.

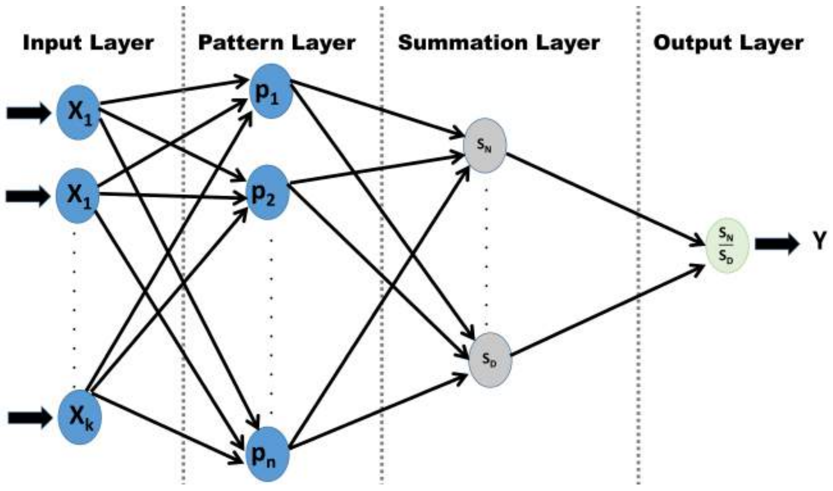


Figure 7.1: GRNN topology

The input layer contains one node for each attribute. If there are k attributes, there will be k input nodes. It serves to distribute the input data to the pattern layer.

The pattern layer contains one node for each training case. For a sample consisting of n training cases there will be n pattern layer nodes. The activation function of pattern layer neuron i is:

$$p_i = \exp\left[-\frac{D^2}{2\sigma^2}\right],$$

where $D^2 = (X - X_i)^T (X - X_i)$ is the squared Euclidean distance between the input vector X and the ith training input vector X_i; and σ determines the spread/ shape of the distribution. It is known as the smoothing factor.

Here is one way to interpret the smoothing parameter. Suppose we had a single input, the greater the value of smoothing factor, the more significant distant training cases become for the predicted value. The larger the smoothing value the smoother the function is, and in the limit it becomes a multivariate Gaussian. Smaller values of the smoothing parameter allow the estimated density to assume non-Gaussian shapes.

The output of the neuron is therefore a measure of the distance of the input from the observed patterns. The computed values are then passed to the summation layer nodes. Notice that each pattern layer neuron is connected to two neurons in the summation layer.

The summation layer has two types of summation neurons. The first (S_N), computes the sum of the weighted outputs of the pattern layer:

$$S_N = \textstyle\sum_{i=1}^{n} y_i p_i \ ,$$

and the second (S_D) calculates the unweighted outputs of the pattern neurons:

$$S_D = \sum_{i=1}^{n} p_i$$

The Output node performs a normalization by dividing the output from the summation layer to generate the prediction:

$$\hat{Y}_i(X) = \frac{S_N}{S_D}$$

Training a GRNN consists of optimizing factors to minimize the error on the training set. The Conjugate Gradient Descent optimization method is used to accomplish this task. The error used during each iteration of the network is typically the Mean Square Error.

Practical Applications

I am always amazed at the uses to which neural networks are applied. It seems their application is limited only by the creativity of their users. Here are three, very different applications which serve to illustrate the practicality of GRNNs to help solve real world problems.

Surface Radiation

The available radiative energy at the Earth's surface is measured by the difference between total upward and downward radiation. This metric is known as net all-wave surface radiation (Rn). It is important because it is a key component of global environmental processes such as evaporation, evapotranspiration, air and soil fluxes as well as other smaller energy-consuming processes such as photosynthesis. For this reason, it has been a central focus of scientists and industrialists who work in the field of remote sensing.

The researchers Jiang et al.[55] explore the potential of the GRNN to estimate global Rn from remotely sensing, surface measurements, and meteorological reading. Measurements were obtained from 251 global sites, illustrated in Figure 7.2.

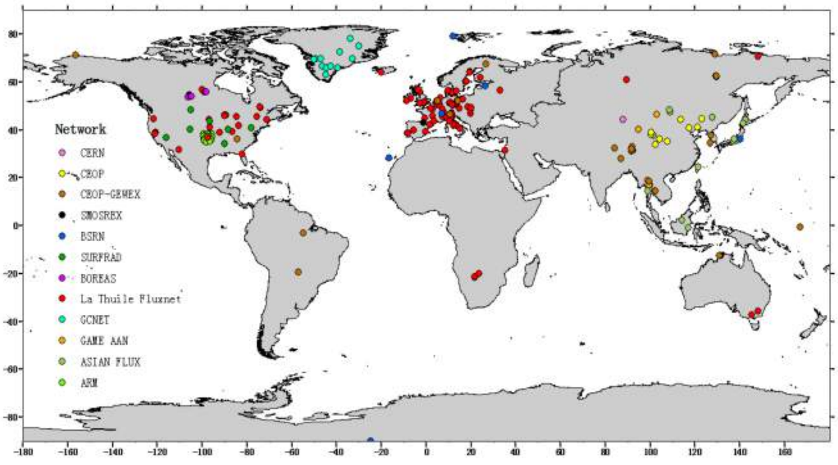
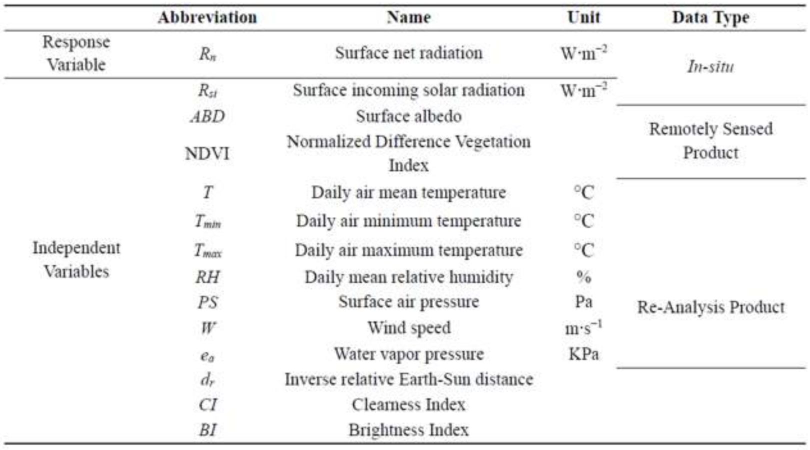


Figure 7.2: Observation sites used by Jiang et al. *Source of figure: Jiang et al.*

A total of 13 attributes were used as inputs into the GRNN model, see Table 6 for details. The data measured by these attributes covered the period 1991–2010.



	Abbreviation	Name	Unit	Data Type
Response Variable	R_n	Surface net radiation	$W \cdot m^{-2}$	*In-situ*
Independent Variables	R_{si}	Surface incoming solar radiation	$W \cdot m^{-2}$	
	ABD	Surface albedo		Remotely Sensed Product
	NDVI	Normalized Difference Vegetation Index		
	T	Daily air mean temperature	°C	Re-Analysis Product
	T_{min}	Daily air minimum temperature	°C	
	T_{max}	Daily air maximum temperature	°C	
	RH	Daily mean relative humidity	%	
	PS	Surface air pressure	Pa	
	W	Wind speed	$m \cdot s^{-1}$	
	e_a	Water vapor pressure	KPa	
	d_r	Inverse relative Earth-Sun distance		
	CI	Clearness Index		
	BI	Brightness Index		

Table 6: Summary of response and attributes used in Jiang et al. *Source of table: Jiang et al.*

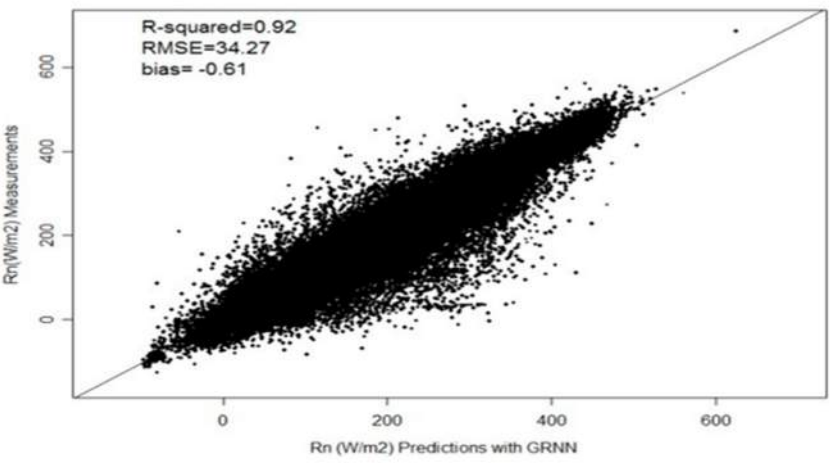


Figure 7.3: Scatter plot of GRNN predicted and measured Rn. *Source of figure Jiang et al.*

The researchers observe that the GRNN model had a coefficient of determination of 0.92 and a root mean square error of 34.27 $W \cdot m^{-2}$ (see Figure 7.3); and conclude that GRNNs offers a feasible approach to Rn prediction - "*This study concluded that ANN* [artificial neural network] *methods are a potentially powerful tool for global Rn estimation.*"

Swine Gas

Gaseous pollutants and distasteful odors generated by industrial swine production are of concern to local residents, legislators and meat producers[56]. Predictive analytic tools including atmospheric dispersion models have been used by regulatory agencies and state planners to determine reasonable setback distances between industrial animal production facilities and neighboring residential areas.

Sun, Hoff, Zelle and Nelson[57] use GRNNs to model gas (NH_3, H^2S, and CO_2) and PM_{10} concentrations and emission rates from industrial pork production facilities. The data was collected hourly for $1\frac{1}{4}$ years from two identical deep pit swine finishing buildings in Iowa[58]. The primary attributes used by the researchers included: outdoor temperature. animal units, total building ventilation rate, and indoor temperature (Tin).

The smoothing parameter was obtained by trial and error. For example, for the GRNN used to model NH_3, a range of values were investigated including 1, 0.5, 0.1 and 0.05. In this particular case the coefficient of determination increased rapidly as the smoothing parameter was reduced. For example, for a value of 1, the coefficient of determination had an average value of 0.49; for a smoothing parameter equal to 0.5 it had an average value of 0.67; and for a soothing parameter equal to 0.05 it had an average value of 0.96.

The coefficient of determination for the developed GRNN models ranged from 0.85 to 0.99. Sun, Hoff, Zelle and Nelson comment that all of the GRNN models used in the study have "excellent predicting abilities..." and conclude by stating "*...artificial neural network (ANN) technologies were capable of accurately modeling source air quality within and from the animal operations.*"

Design Optimization of Brushless DC Motors

Brushless DC motors are more efficient than brushed DC motors and require considerably less maintenance due to the absence of brushes. Their ability in the speed and torque domain means they are also more versatile. However, they are prone to cogging torque[59].

Umadevi et al. [60]develop a GRNN for design optimization of brushless DC motors. Two response variables were jointly modeled - cogging torque and average torque. The objective was the maximization of average torque and the minimization of cogging torque using as features magnet length, stator slot opening and air gap.

The initial design had an average torque of 0.363 N-m and a cogging torque of 0.012 N-m. After optimization and with a smoothing factor of 0.5 the optimal GRNN model produced a design with an average torque of 0.379 N-m and a cogging torque of 0.0084 N-m. The researchers conclude by stating "*The results of GRNN disclose an optimized BLDC* [brushless DC] *motor design with a significant improvement in the torque profile.*"

Notes

[54]See Specht, Donald F. "A general regression neural network." Neural Networks, IEEE Transactions on 2.6 (1991): 568-576.

[55]Jiang, Bo, et al. "Surface Daytime Net Radiation Estimation Using Artificial Neural Networks." Remote Sensing 6.11 (2014): 11031-11050.

[56]See for example:

- Heber, A. J., D. J. Jones, and A. L. Sutton. "Methods and Practices to Reduce Odour from Swine Facilities." Online at http://persephone. agcom. purdue. edu/AgCom/Pubs/AE/AQ-2/AQ-2. html.
- Schmidt, D. R. "Evaluation of various swine waste storage covers for odor control." Final report prepared for the University of Minnesota Extension Service, Department of Biosystems and Agricultural Engineering, University of Minnesota, St. Paul, MN (1997).
- Meszaros, G. "Minimizing emissions of pig husbandry using low cost biofilter and bioactive deep litter." REUR Technical Series (FAO) (1994).

[57]Sun, Gang, et al. "Development and comparison of backpropagation and generalized regression neural network models to predict diurnal and seasonal gas and PM10 concentrations and emissions from swine buildings." (2008).

[58]Data collected from January 2003 to April 2004.

[59]A pulsating motion produced by the variation of the air-gap permeance or reluctance of the stator teeth and slots above the magnets as a electric rotor rotates.

[60]Umadevi, N., et al. "Data Interpolation and Design Optimisation of Brushless DC Motor Using Generalized Regression Neural Network." Journal of Electrical Engineering & Technology 10.1 (2015): 188-194.

Chapter 8

Regression Using GRNNs

SHELLEY Winters, the American actress, poked fun at herself when she said "*I'm not overweight. I'm just nine inches too short.*" Yes, it is a fact that we humans are growing heavier by the decade! Right across the industrialized world, illnesses resulting from overweight and obesity have become major health concerns[61]. Diseases including diabetes and cardiovascular disease, as well certain forms of cancer are believed to be a direct consequence of our overfed lifestyle[62].

Predictive analytic and statistical tools can provide valuable information to assist in understanding and tackling this challenging issue[63]. In this chapter we develop a GRNN to predict body fat using measurements of skinfold thickness, circumferences, and bone breadth as our primary input attributes.

Getting the Required Packages and Data

The package grnn contains the functions we need to build a GRNN. It can be loaded as follows:

```
> require(grnn)
```

The GRNN is built using the data frame `bodyfat` contained in the `TH.data` package. The data was collected by Garcia et al[64] to develop reliable predictive regression equations for body fat by measuring skinfold thickness, circumferences, and bone breadth on men and women. The value in using these metrics as attributes is that they are easy to measure, relatively inexpensive to obtain, can be measured in the field setting without large scale clinical equipment, and are of potential value as covariates in long term epidemiological studies.

The original study collected data from 117 healthy German subjects, 46 men and 71 women. The `bodyfat` data frame contains the data collected on 10 variables for the 71 women, see Table 7.

Name	Description
`DEXfat`	measure of body fat.
`age`	age in years.
`waistcirc`	waist circumference.
`hipcirc`	hip circumference.
`elbowbreadth`	breadth of the elbow.
`kneebreadth`	breadth of the knee.
`anthro3a`	SLT.
`anthro3b`	SLT.
`anthro3c`	SLT.
`anthro4`	SLT.

Table 7: Response and independent variables in `bodyfat` data frame. Note SLT = sum of logarithm of three anthropometric measurements.

Here is how to load the data:

```
> data("bodyfat",package="TH.data")
```

Figure 8.1 shows the kernel density plot of the response variable `DEXfat`; and Figure 8.2 shows boxplots for the attributes.

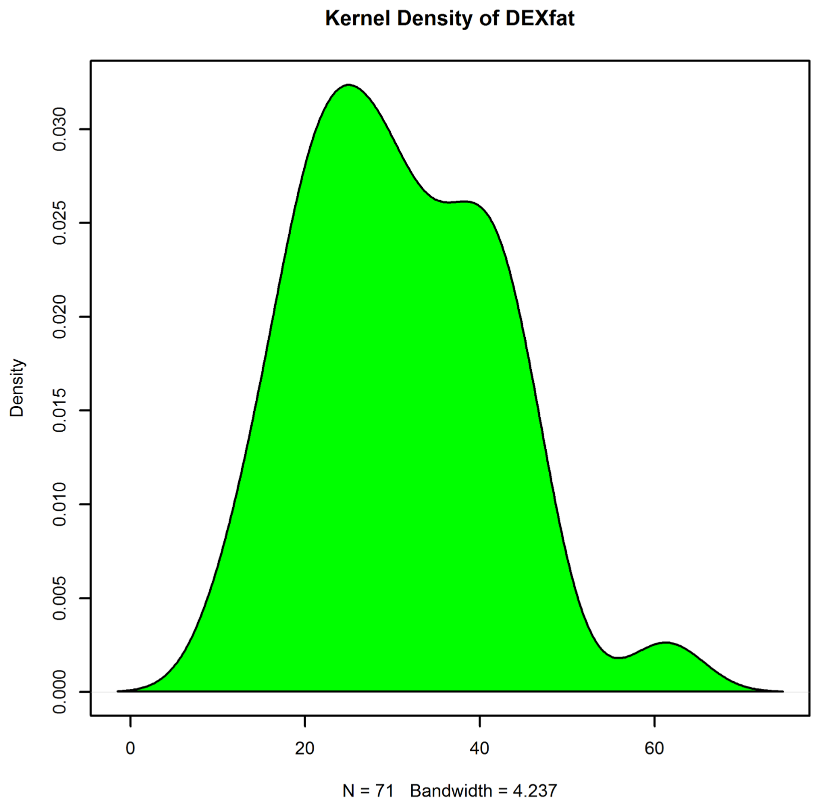


Figure 8.1: Kernel Density of DEXfat

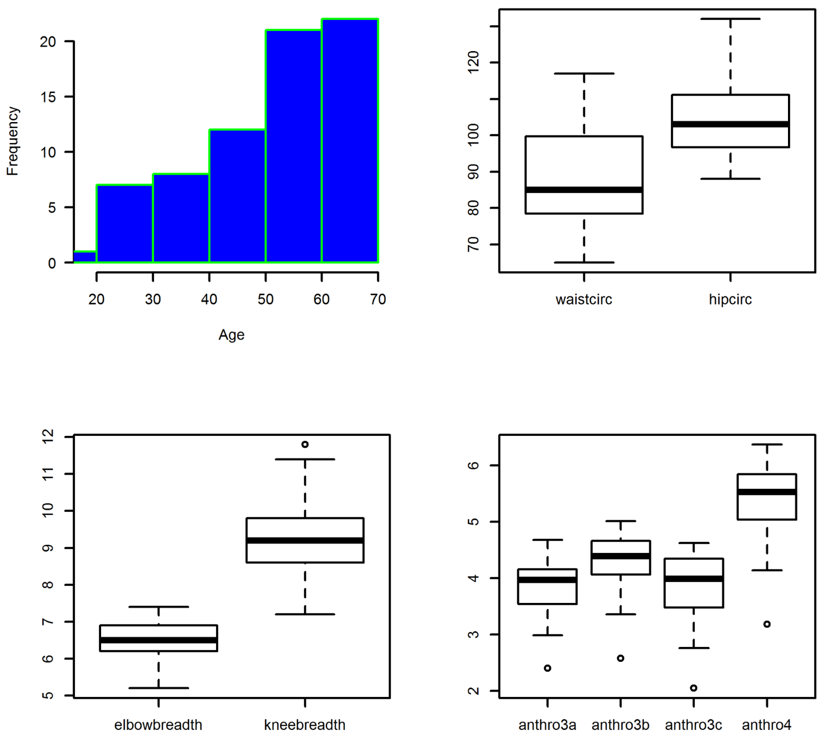


Figure 8.2: Boxplots of attributes

Prepare Data

We use 45 out of the 71 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 71 observations:

```
> set.seed(465)
> n <- nrow(bodyfat)

> n_train <- 45
> train <- sample(1:n,n_train , FALSE)
```

Let's go through the above line by line. The first line uses the `set.seed` method to ensure reproducibility of the results. The number of observations are stored in the R object `n` using the `nrow` method. Finally, the row numbers of the sample observations to be used in the training sample are selected without replacement using the `sample` method.

Next, we take the log of the response variable and store it in `y`. We do the same for the attributes, storing them in the R object `x`. The we use the `NULL` parameter to delete the response variable from `x` (so that `x` only contains attributes):

```
> y<-log(bodyfat$DEXfat)
> x<-log(bodyfat)
> x$DEXfat<-NULL
```

How to Estimate the Model

No we are ready to estimate the GRNN model. The only parameter we need to supply is the smoothing value. We saw earlier on page 116 that the researchers Sun, Hoff, Zelle and Nelson selected the value of this parameter by trial and error. We do the same in our analysis. We use the `learn` method to build the model and the `smooth` method with the smoothing parameter `sigma` equal to 2.5 and store the resultant model in `fit`:

```
> fit_basic <- learn(data.frame(y[train],x[
   train,]))
> fit <- smooth(fit_basic, sigma = 2.5)
```

Building Predictions

Let's take a look at the first row of the covariates in the testing set:

```
> round(x[-train,][1,],2)

    age waistcirc hipcirc elbowbreadth
47 4.04      4.61    4.72         1.96

kneebreadth anthro3a anthro3b anthro3c
       2.24     1.49      1.6      1.5

   anthro4
47    1.81
```

To see the first observation of the response variable in the test sample you can use a similar technique:

```
> y[-train][1]
[1] 3.730021
```

Now we are ready to predict the first response value in the test set using the covariates. Here is how to do that:

```
> guess(fit, as.matrix(x[-train,][1,]))
[1] 3.374901
```

Of course, we will want to predict using all the values in the test sample. We can achieve this using the `guess` method and the following lines of R code:

```
> pred<-1:n_test

> for (i in 1:n_test)
 {
 pred[i]<-guess(fit, as.matrix(x[-train,][i
    ,]))
 }
```

Finally, we plot in Figure 8.3 the observed and fitted values, and on the same chart we also plot the linear regression line of the predicted versus observed values:

```
> plot(y[-train],pred,
xlab="log(DEXfat)",
ylab="Predicted Values",
main="Test Sample Model Fit")

> abline(linReg<-lm(pred~ y[-train]),col=
   "darkred")

> round(cor(pred,y[-train])^2,3)
[1] 0.915
```

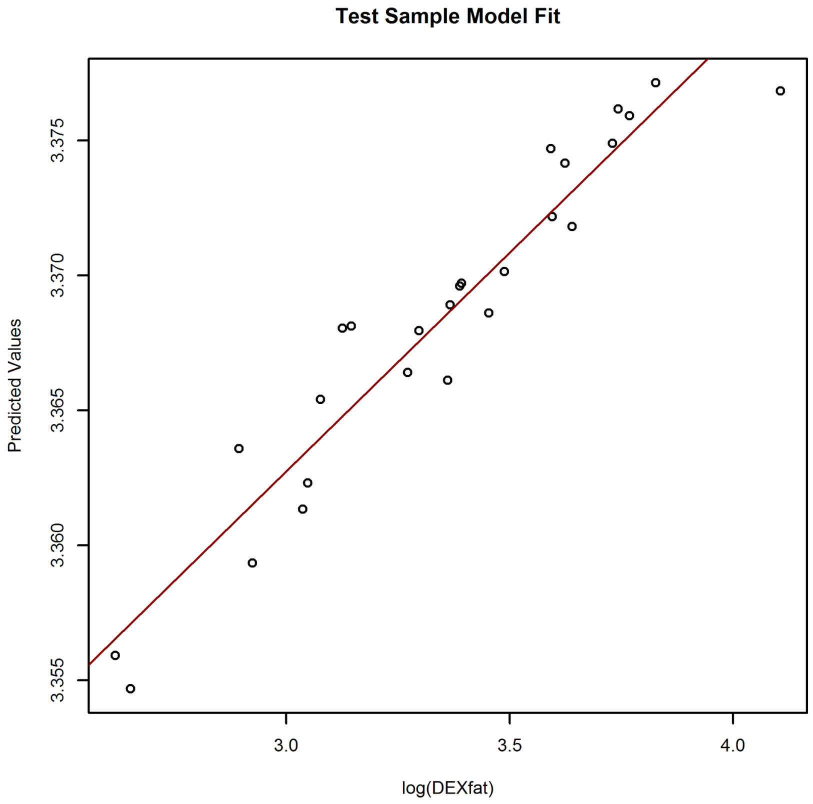


Figure 8.3: General Regression Neural Network observed and predicted values using `bodyfat`

The overall squared correlation is 0.915. So, it seems our initial guess of 2.5 for the smoothing parameter yielded a pretty good fit.

Exercises

- **Question 1:** Re-estimate the model using a smoothing parameter of 0.1 and assess fit.
- **Question 2:** Re - estimate model with a smoothing parameter of 4 and assess fit.
- **Question 3:** Estimate the model only using the attributes `waistcirc`, `hipcirc`, `elbowbreadth`, `kneebreadth` with `sigma` = 2.5 and assess fit.
- **Question 4:** Re-estimate the model in question 3 with `sigma` = 0.5 and assess fit

Notes

[61]See for example:

- World Health Organization. Obesity: preventing and managing the global epidemic. No. 894. World Health Organization, 2000.
- Allison, David B., et al. "Annual deaths attributable to obesity in the United States." Jama 282.16 (1999): 1530-1538.
- Leung, Man-Yee Mallory, et al. "Life Years Lost and Lifetime Health Care Expenditures Associated With Diabetes in the US, National Health Interview Survey, 1997–2000." Diabetes care 38.3 (2015): 460-468.
- Jackson, S. E., et al. "Perceived weight discrimination in England: a population-based study of adults aged $\geq$; 50 years." International Journal of Obesity 39.5 (2015): 858-864.
- Onubi, Ojochenemi J., et al. "Maternal obesity in Africa: a systematic review and meta-analysis." Journal of Public Health (2015): fdv138.

[62]See for example:

- Hamer, Mark, and Emmanuel Stamatakis. "Metabolically healthy obesity and risk of all-cause and cardiovascular disease mortality." The Journal of Clinical Endocrinology & Metabolism 97.7 (2012): 2482-2488.
- Steppan, Claire M., et al. "The hormone resistin links obesity to diabetes." Nature 409.6818 (2001): 307-312.
- Kopelman PG. Obesity as a medical problem. Nature. 2000;404:635– 43.
- Calle, Eugenia E., et al. "Overweight, obesity, and mortality from cancer in a prospectively studied cohort of US adults." New England Journal of Medicine 348.17 (2003): 1625-1638.
- Seidell JC, Flegal KM. Assessing obesity: classification and epidemiology. Br Med Bull. 1997;53:238 –52.
- Trayhurn, Paul. "Hypoxia and adipose tissue function and dysfunction in obesity." Physiological reviews 93.1 (2013): 1-21.

[63]See for example:

- Petkeviciene, Janina, et al. "Anthropometric measurements in childhood and prediction of cardiovascular risk factors in adulthood: Kaunas cardiovascular risk cohort study." BMC public health 15.1 (2015): 218.

- Graversen, Lise, et al. "Prediction of adolescent and adult adiposity outcomes from early life anthropometrics." Obesity 23.1 (2015): 162-169.
- Muñoz-Cachón, M. J., et al. "Overweight and obesity: prediction by silhouettes in young adults." Obesity 17.3 (2009): 545-549.
- O'callaghan, M. J., et al. "Prediction of obesity in children at 5 years: a cohort study." Journal of paediatrics and child health 33.4 (1997): 311-316.

[64]Garcia, Ada L., et al. "Improved prediction of body fat by measuring skinfold thickness, circumferences, and bone breadths." Obesity Research 13.3 (2005): 626-634.

Part IV

Recurrent Neural Networks

UP until now we have covered feed forward neural networks which model static input-output functions where information is fed in a single forward direction through the network. In these networks one input pattern produces one output. They have no sense of time or retain memory of their previous state. These neural networks are good at classification and regression tasks. In this part of the book we explore and use recurrent neural networks.

Why should we include recurrent neural networks in our toolkit? Partly because of the Universal Approximation Theorem which informs us that:

> **Universal Approximation Theorem**: Any non-linear dynamical system can be approximated to any accuracy by a recurrent neural network, with no restrictions on the compactness of the state space, provided that the network has enough sigmoidal hidden units.

However, our primary reason is pragmatic. They often outperform other modeling techniques in actual practice.

In a recurrent neural network neurons connect back to other neurons, information flow is multi-directional so the activation of neurons can flow around in a loop. This type of neural network has a sense of time and memory of earlier networks states which enables it to learn sequences which vary over time, perform classification tasks and develop predictions of future states. As a result, recurrent neural networks are used for classification, stochastic sequence modeling and associative memory tasks.

A simple form of recurrent neural network is illustrated in Figure 8.4. A delay neuron is introduced in the context layer. It has a memory in the sense that it stores the activation values of an earlier time step. It releases these values back into the network at the next time step.

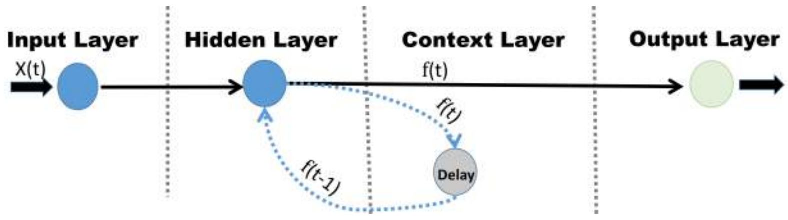


Figure 8.4: Simple recurrent neural network

We discuss Elman networks[65] in chapter 9 and Jordan networks[66] in chapter 11. Our emphasis is on practical knowledge, actual real world applications and how to build them using R.

Chapter 9

Elman Neural Networks

ELMAN networks are akin to the multi-layer perceptron augmented with one or more context layers . 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.

Khatib et al.[67] develop a Elman network to predict hourly solar radiation. The network is illustrated in Figure 9.1. It is a 3-layered network with eight input attributes (Latitude, Longitude, Temperature, Sunshine ratio, Humidity, Month, Day, Hour); five hidden neurons, five context neurons; and two output neurons predicting global solar radiation and diffused solar radiation.

Role of Context Layer Neurons

The neurons in the context layer are included because they remember the previous internal state of the network by storing hidden layer neuron values. The stored values are delayed by one time step; and are used during the next time step as additional inputs to the network for either activating the current layer or some other layer.

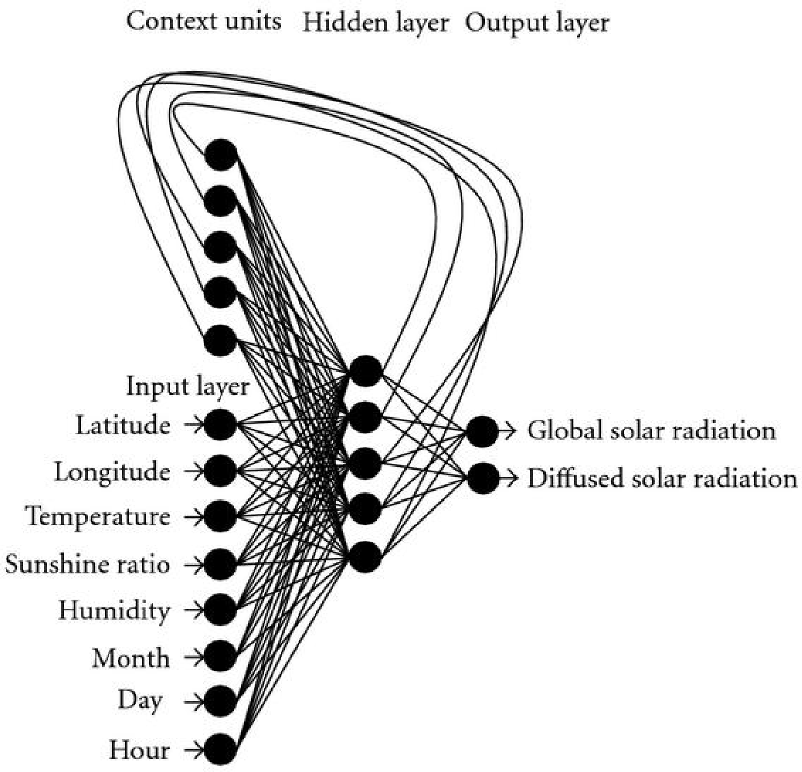


Figure 9.1: Schematic illustration of the Elman network to predict hourly solar radiation. Source Khatib et al.

Understanding the Information Flow

To help strengthen our intuition about this useful model let's look at a simplified illustration. Suppose we have a neural network consisting of only two neurons, as illustrated in Figure 9.2. The network has one neuron in each layer. Each neuron has a bias denoted by b_1 for the first neuron, and b_2 for the second neuron. The associated weights are w_1 and w_2 with activation function f_1 and f_2. Since there are only two neurons the output

Y as a function of the input attribute X is given by:

$$Y = f_2 (w_2 f_1 (w_1 X + b_1) + b_2)$$

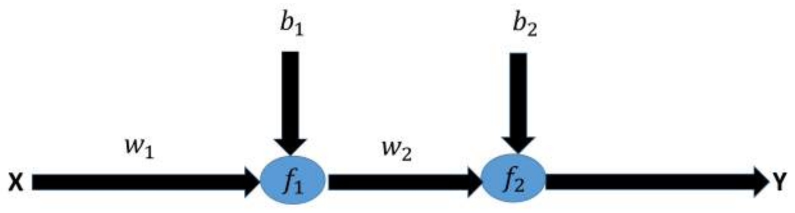


Figure 9.2: Two neuron network

In an Elman network, as shown in Figure 9.3, there is a context neuron which feeds the signal from the hidden layer neuron back to that same neuron. The signal from the context neuron is delayed by one time step and multiplied by w_2 before being fed back into the network.

In the Elman network the neurons typically use sigmoidal activation functions. The output at time t is given by:

$$Y[t] = f_2 (w_3 f_1 (w_1 X[t] + w_2 C + b_1) + b_2)$$

where,

$$C = Y_1[t-1]$$

During the training process the weights and biases are iteratively adjusted to minimize the network error, typically measured using the mean squared error. If the error is insufficiently small another iteration or epoch is initiated. We see, in this simple example, that the hidden layer is fully connected to inputs and the network exhibits recurrent connections; and the use of delayed memory creates a more dynamic neural network system.

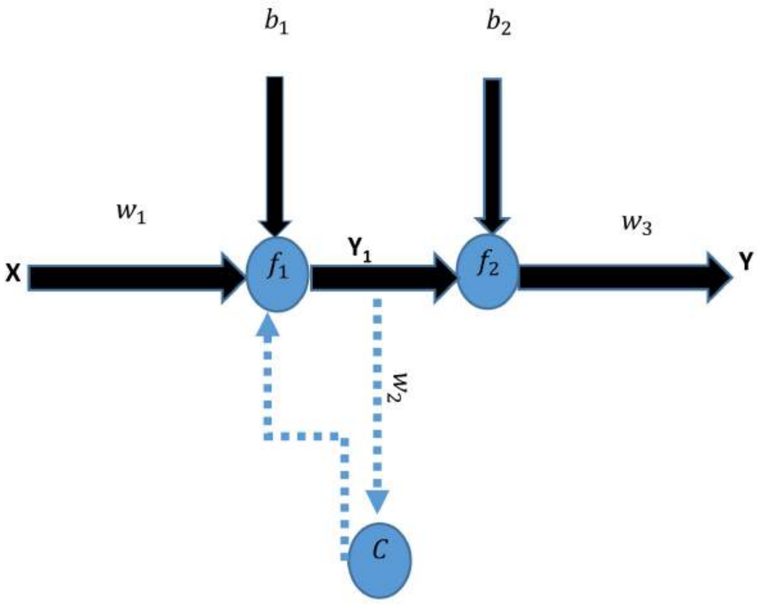


Figure 9.3: Two node Elman network

Advantages of Elman Neural Networks

Elman Neural Networks are useful in applications where we are interested in predicting the next output in given sequence. Their dynamic nature can capture time-dependent patterns, which are important for many practical applications. This type of flexibility is important in the analysis of time series data.

A typical example is the Elman network, shown in Figure 9.1 to predict hourly solar radiation developed by Khatib et al. This property of memory retention is also useful for pattern recognition and robot control.

Practical Applications

Let's take a look at some interesting applications of Elman neural networks. The applications range from fault prediction, weather forecasting to predicting the stock market. In each illustration discussed below, the power of this predictive tool becomes evident. Take a look, then answer the question of how will you apply this incredible tool in your data modeling challenges?

Weather Forecasting

Accurate weather forecasts are of keen interest to all sections of society. Farmers look to weather conditions to guide the planting and harvesting of their crops, transportation authorities seek information on the weather to determine whether to close or open specific transportation corridors, and individuals monitor the weather as they go about their daily activities.

Maqsood, Khan and Abraham[68] develop a Elman neural network to forecast the weather of Vancouver, British Columbia, Canada. They specifically focus on creating models to forecast the daily maximum temperature, minimum daily temperature and wind-speed. The dataset consisted of one year of daily observations. The first eleven months were used to train the network with the testing set consisting of the final month's data (1st to 31st August). The optimal model had 45 hidden neurons with Tan-sigmoid activation functions.

The peak temperature forecast had an average correlation of 0.96 with the actual observations, the minimum temperature forecast had an average correlation of 0.99 with the actual observations, and the wind-speed forecast had an average correlation of 0.99.

Urban Rail Auxiliary Inverter Fault Prediction

Auxiliary inverters are an important component in urban rail vehicles. Their failure can cause economic loss, delays and commuter dissatisfaction with the service quality and reliability. Yao et al.[69] apply the Elman neural network to the task of fault recognition and classification in this vital piece of equipment.

The network they construct consisted of eight input neurons, seven hidden layer neurons and 3 neurons in the output layer. The eight inputs corresponded to various ranges on the frequency spectrum of fault signals received from the inverter. The three outputs corresponded to the response variables of the voltage fluctuation signal, impulsive transient signal and the frequency variation signal. The researchers observe that "*Elman neural network analysis technology can identify the failure characteristics of the urban rail vehicle auxiliary inverter.*"

Forecasting the Quality of Water

The quality of water is often assessed by the total nitrogen (TN), total phosphorus (TP) and dissolved oxygen (DO) content present. Heyi and Gao[70] use Elman networks to predict the water quality in Lake Taihu, the third largest freshwater lake in China. Measurements were taken at three different sites in Gonghu Bay[71]. The training sample consisted of 70% of the observations. The observations were chosen at random. The remaining 30% of observations made up the test sample.

Ten parameters were selected as important covariates for water quality[72] Separate Elman networks were developed to predict TN, TP and DO. Each model was site specific, with a total of nine models developed for testing. Each model composed of one input layer, one hidden layer and one output layer. Trial and error was used to determine the optimum number of nodes in the hidden layer of each model.

For TN the researchers report a R-squared statistic of 0.91,0.72 and 0.92 for site 1, site 2 and site 3 respectively. They observe "*The developed Elman models accurately simulated the TN concentrations during water diversion at three sites in Gonghu Bay of Lake Taihu.*"

For TP R-squared values of 0.68,0.45 and 0.61 were reported for site 1, site 2 and site 3 respectively. These values, although not as high as the TN models, were deemed acceptable.

The r-squared values for DO were considerably lower at 0.3, 0.39 and 0.83 for site 1, site 2 and site 3 respectively. These lower values were addressed by the researchers with the suggestion that "*The accuracy of the model can be improved not only by adding more data for the training and testing of three sites but also by inputting variables related to water diversion.*"

Forecasting the Stock Market

The ability to predict financial indices such as the stock market is one of the appealing and practical uses of neural networks. Elman networks are particularly well suited to this task. Wang et al.[73] successfully use Elman networks for this task.

They use Elman networks to predict daily changes in four important stock indices - the Shanghai Stock Exchange (SSE) Composite Index, Taiwan Stock Exchange Capitalization Weighted Stock Index (TWSE), Korean Stock Price Index (KOSPI), and Nikkei 225 Index (Nikkei225). Data on the closing prices covering 2000 trading days were used to construct the models.

The researchers developed four models, one for each index. The SSE model had 9 hidden nodes, the TWSE 12 hidden nodes, and the KOSPI and NIKKEI225 each had 10 hidden nodes. The researchers report all four models have a correlation coefficient of 0.99 with the actual observed observations.

Notes

[65]Elman, Jeffrey L. "Finding structure in time." Cognitive science 14.2 (1990): 179-211.

[66]Jordan, Michael I. "Serial order: A parallel distributed processing approach." Advances in psychology 121 (1997): 471-495.

[67]Khatib, Tamer, Azah Mohamed, Kamarulzaman Sopian, and M. Mahmoud. "Assessment of artificial neural networks for hourly solar radiation prediction." International journal of Photoenergy 2012 (2012).

[68]Maqsood, Imran, Muhammad Riaz Khan, and Ajith Abraham. "Canadian weather analysis using connectionist learning paradigms." Advances in Soft Computing. Springer London, 2003. 21-32.

[69]Yao, Dechen, et al. "Fault Diagnosis and Classification in Urban Rail Vehicle Auxiliary Inverter Based on Wavelet Packet and Elman Neural Network." Journal of Engineering Science and Technology Review 6.2 (2013): 150-154.

[70]Wang, Heyi, and Yi Gao. "Elman's Recurrent neural network Applied to Forecasting the quality of water Diversion in the Water Source Of Lake Taihu." Energy Procedia 11 (2011): 2139-2147.

[71]The data set was collected from continuous monitoring of water quality from May 30 to Sep 19 in 2007, Apr 16 to Jun 19 in 2008, and May 5 to Jun 30 in 2009.

[72]Water temperature, water pH, secchi depth, dissolved oxygen, permanganate index, total nitrogen, total phosphorus, ammonical nitrogen, Chl-a, and the average input rate of water into the lake.

[73]Wang, Jie, et al. "Financial Time Series Prediction Using Elman Recurrent Random Neural Networks." Computational Intelligence and Neuroscience 501 (2015): 613073.

Chapter 10

Building Elman Networks

ELMAN networks are particularly useful for modeling time-series data. In this chapter we build a model to predict the total number of deaths from bronchitis, emphysema and asthma in the United Kingdom[74].

Loading the Required Packages

We will use a couple of packages as we build our Elman network. The first is `RSNNS` which uses the Stuttgart Neural Network Simulator library[75] (SNNS). This is a library containing many standard implementations of neural networks. What's great for us is that the `RNSS` package wraps SNNS functionality to make it available from within R. We will also use the `quantmod` package, which has a very easy to use lag operator function. This will be useful because we will be building our model using timeseries data. Let's load the packages now:

```
> require(RSNNS)
> require(quantmod)
```

Getting and Cleaning the Data

The data we use is contained in the datasets package. The data frame we want is called UKLungDeaths. The datasets package comes preloaded in R, so we don't technically need to load it. However, it is good practice; here is how to do it efficiently:

```
> data("UKLungDeaths",package="datasets")
```

The data frame UKLungDeaths contains three timeseries giving the monthly number of deaths from bronchitis, emphysema and asthma in the United Kingdom from 1974 to 1979[76]. The first timeseries is the total number of deaths (ldeaths), the second timeseries males only (mdeaths) and the third timeseries females only (fdeaths). We can visualize these timeseries using the plot method as follows:

```
> par(mfrow=c(3,1))

> plot(ldeaths, xlab="Year",
ylab="Both sexes",
main="Total")

> plot(mdeaths, xlab="Year",
ylab="Males",
main="Males")

>plot(fdeaths, xlab="Year",
ylab="Females",
main="Females")
```

The first line uses the par method to combine multiple plots into one overall graph as shown in Figure 10.1. The first plot method creates a chart for total monthly deaths contained in the R object ldeaths. The second plot method creates a chart for monthly male deaths contained in the R object mdeaths; and the third plot method creates a chart for monthly female deaths contained in the R object fdeaths.

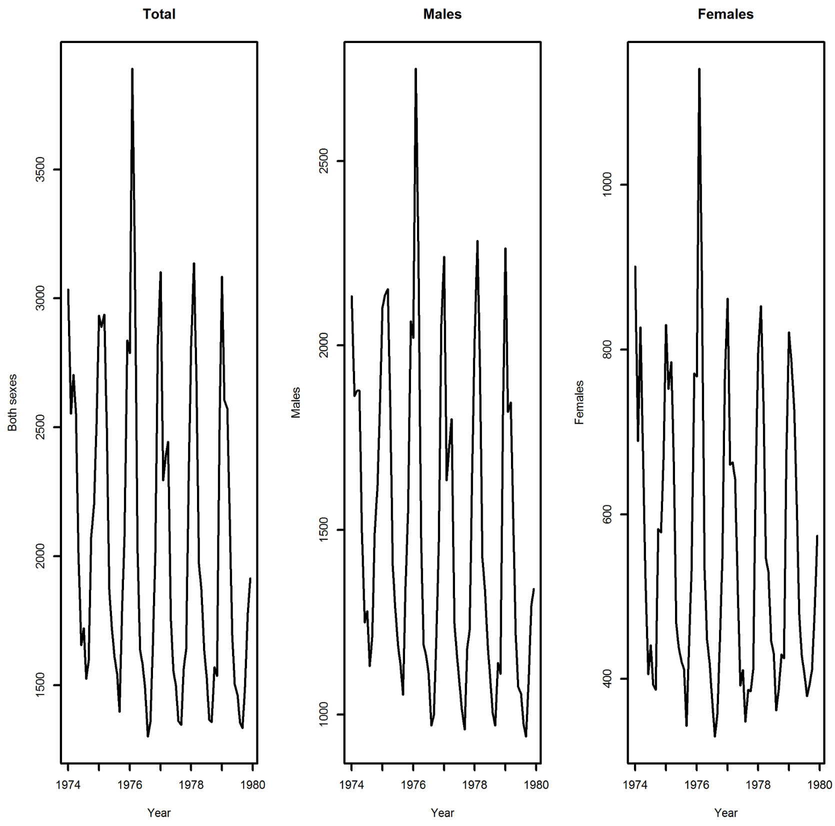


Figure 10.1: Monthly number of deaths from bronchitis, emphysema and asthma in the United Kingdom

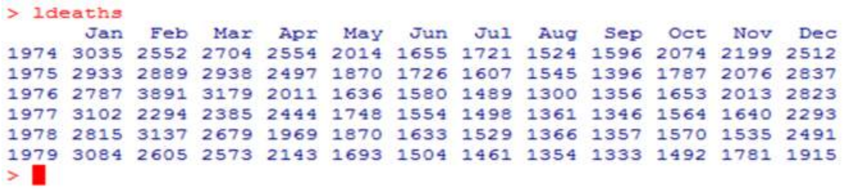
Overall, we see that there has been considerable variation in the number of deaths over time the number of deaths undulating by the month in a clearly visible pattern. The highest and lowest total monthly deaths were observed in 1976. It is interesting to note that for males the lows for each cycle exhibit a strong downward trend. The overall trend as we enter 1979 is quite clearly downward.

Since we are interested in modeling the total number of deaths,we will focus our analysis on the `ldeaths` data frame. It's always a good idea to do a quick check for missing values.

These are coded as NA in R. We sum the number of missing values using the is.na method:

```
> sum(is.na(ldeaths))
[1] 0
```

So we have zero NA's in the dataset. There do not appear to be are any missing values, but it is always a good idea to check. We can also check visually:



```
> ldeaths
      Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
1974 3035 2552 2704 2554 2014 1655 1721 1524 1596 2074 2199 2512
1975 2933 2889 2938 2497 1870 1726 1607 1545 1396 1787 2076 2837
1976 2787 3891 3179 2011 1636 1580 1489 1300 1356 1653 2013 2823
1977 3102 2294 2385 2444 1748 1554 1498 1361 1346 1564 1640 2293
1978 2815 3137 2679 1969 1870 1633 1529 1366 1357 1570 1535 2491
1979 3084 2605 2573 2143 1693 1504 1461 1354 1333 1492 1781 1915
>
```

Next we check the class of the ldeaths object:

```
> class(ldeaths)
[1] "ts"
```

It is a timeseries object. This is good to know as we shall see shortly.

Let's summarize ldeaths visually. To do this we plot the timeseries, kernel density plot and boxplot as follows:

```
> par( mfrow = c(3, 1))

> plot(ldeaths)

> x<-density(ldeaths)
> plot(x, main="UK total deaths from lung
   diseases")
> polygon(x, col="green", border="black")

> boxplot(ldeaths,col="cyan",ylab="Number
   of deaths per month")
```

The resultant plot is show in Figure 10.2.

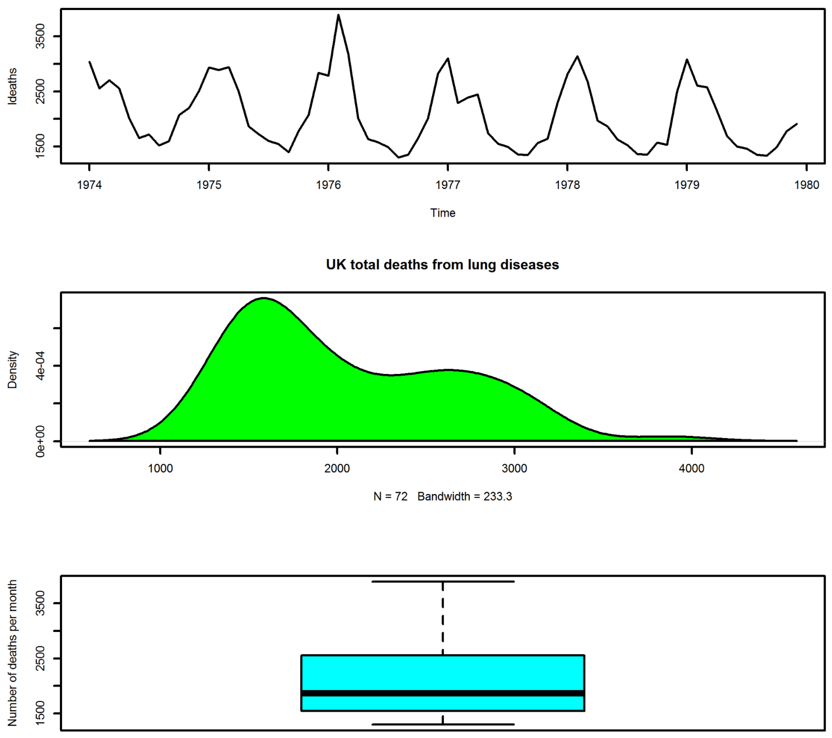


Figure 10.2: Visual summary of total number of deaths

Transforming the Data

Since the data appears to be in order, we are ready to transform it into a suitable format for use with the Elman neural network. The first thing we do is make a copy of the data, storing the result in the R object Y.

```
y<-as.ts(ldeaths)
```

When working with timeseries 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:

```
y<-log(y)
```

Now, we can standardize the observations using the `scale` method:

```
y<- as.ts(scale(y))
```

Note the `as.ts` method ensures we retain a `ts` object.

Since we have no other explanatory attributes for this data, we will use a pure timeseries approach. The main question in this modeling methodology is how many lags of the dependent variable to use? Well, since we have monthly observations over a number of years, let's use 12 lags at the monthly frequency. One way to do this is to use the `Lag` operation in the `quantmond` package. This requires that we convert `y` into a `zoo` class object:

```
> y<-as.zoo(y)
```

Now we are ready to use the `quantmond Lag` operator:

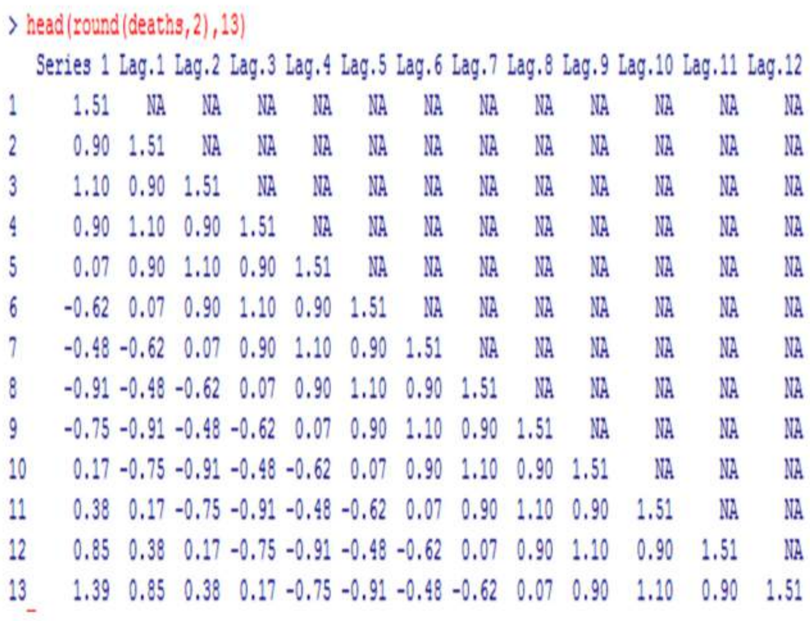
```
> x1<-Lag(y, k = 1)
> x2<-Lag(y, k = 2)
> x3<-Lag(y, k = 3)
> x4<-Lag(y, k = 4)
> x5<-Lag(y, k = 5)
> x6<-Lag(y, k = 6)
> x7<-Lag(y, k = 7)
> x8<-Lag(y, k = 8)
> x9<-Lag(y, k = 9)
> x10<-Lag(y, k = 10)
> x11<-Lag(y, k = 11)
> x12<-Lag(y, k = 12)
```

This gives us 12 attributes to feed as inputs into our neural network.

The next step is to combine the observations into a single data frame:

```
> deaths<-cbind(x1,x2,x3,x4,x5,x6,x7,x8,x9,
   x10,x11,x12)
> deaths<-cbind(y,deaths)
```

Let's take a quick peak at what `deaths` contains:



```
> head(round(deaths,2),13)
   Series 1 Lag.1 Lag.2 Lag.3 Lag.4 Lag.5 Lag.6 Lag.7 Lag.8 Lag.9 Lag.10 Lag.11 Lag.12
1      1.51    NA    NA    NA    NA    NA    NA    NA    NA    NA     NA     NA     NA
2      0.90  1.51    NA    NA    NA    NA    NA    NA    NA    NA     NA     NA     NA
3      1.10  0.90  1.51    NA    NA    NA    NA    NA    NA    NA     NA     NA     NA
4      0.90  1.10  0.90  1.51    NA    NA    NA    NA    NA    NA     NA     NA     NA
5      0.07  0.90  1.10  0.90  1.51    NA    NA    NA    NA    NA     NA     NA     NA
6     -0.62  0.07  0.90  1.10  0.90  1.51    NA    NA    NA    NA     NA     NA     NA
7     -0.48 -0.62  0.07  0.90  1.10  0.90  1.51    NA    NA    NA     NA     NA     NA
8     -0.91 -0.48 -0.62  0.07  0.90  1.10  0.90  1.51    NA    NA     NA     NA     NA
9     -0.75 -0.91 -0.48 -0.62  0.07  0.90  1.10  0.90  1.51    NA     NA     NA     NA
10     0.17 -0.75 -0.91 -0.48 -0.62  0.07  0.90  1.10  0.90  1.51     NA     NA     NA
11     0.38  0.17 -0.75 -0.91 -0.48 -0.62  0.07  0.90  1.10  0.90   1.51     NA     NA
12     0.85  0.38  0.17 -0.75 -0.91 -0.48 -0.62  0.07  0.90  1.10   0.90   1.51     NA
13     1.39  0.85  0.38  0.17 -0.75 -0.91 -0.48 -0.62  0.07  0.90   1.10   0.90   1.51
```

Notice the NA's as the number of lags increases from 1 to 12. This is as expected, since we are using lagged values. However, we do need to remove the NA observations from our dataset:

```
> deaths <- deaths[-(1:12),]
```

We are ready to begin creating our training and testing samples. First, we count the number of rows and use the `set.seed` method for reproducibility:

```
>  n=nrow(deaths)
>  n
[1] 60
> set.seed(465)
```

As a quick check we see there are 60 observations left for use in analysis. This is as expected because we deleted 12 rows of observations due to lagging the data.

Let's use 45 rows of data to build the model, and the remaining 15 rows of data we use for the test sample. We select the training data randomly without replacement as follows:

```
> n_train <- 45
> train <- sample(1:n,n_train , FALSE)
```

How to Estimate the Model

To make things as easy as possible we store the covariate attributes containing the lagged values in the R object `inputs`, and the response variable in the object `ouputs`:

```
> inputs <- deaths[,2:13]
> outputs <- deaths[,1]
```

We fit a neural network with two hidden layers each containing one node. We set the learning rate to 0.1 and the maximum number of iterations to 1000:

```
> fit <- elman(inputs[train],
outputs[train],
size=c(1,1),
learnFuncParams=c(0.1),
maxit=1000)
```

Given the relatively small size of the dataset the model converges pretty quickly. Let's plot the error function:

```
 plotIterativeError(fit)
```

It is illustrated in Figure 10.3.

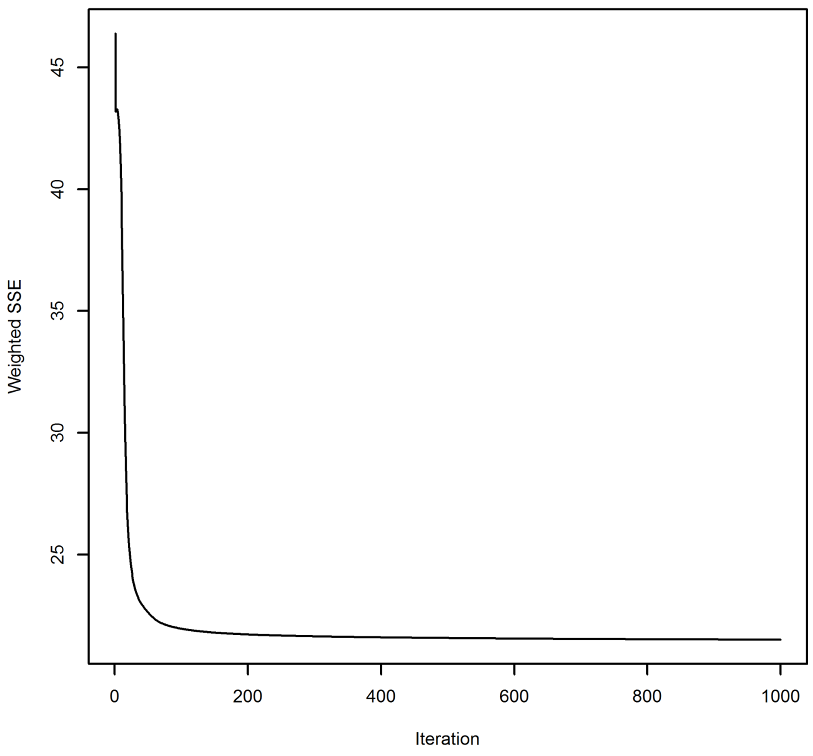
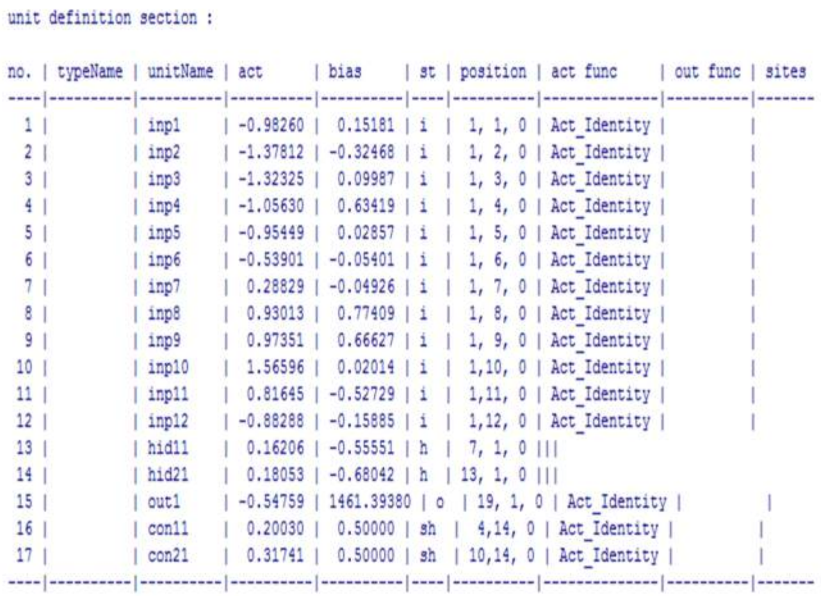


Figure 10.3: Elman neural network error plot

The error drops really quickly, leveling off by around 500 iterations. We can also use the `summary` method to get further details on the network:

```
> summary(fit)
```

You will see a large block of R output. Look for the section that looks like this:



```
unit definition section :

no. | typeName | unitName | act      | bias     | st | position | act func     | out func | sites
----|----------|----------|----------|----------|----|----------|--------------|----------|-------
  1 |          | inp1     | -0.98260 |  0.15181 | i  |  1, 1, 0 | Act_Identity |          |
  2 |          | inp2     | -1.37812 | -0.32468 | i  |  1, 2, 0 | Act_Identity |          |
  3 |          | inp3     | -1.32325 |  0.09987 | i  |  1, 3, 0 | Act_Identity |          |
  4 |          | inp4     | -1.05630 |  0.63419 | i  |  1, 4, 0 | Act_Identity |          |
  5 |          | inp5     | -0.95449 |  0.02857 | i  |  1, 5, 0 | Act_Identity |          |
  6 |          | inp6     | -0.53901 | -0.05401 | i  |  1, 6, 0 | Act_Identity |          |
  7 |          | inp7     |  0.28829 | -0.04926 | i  |  1, 7, 0 | Act_Identity |          |
  8 |          | inp8     |  0.93013 |  0.77409 | i  |  1, 8, 0 | Act_Identity |          |
  9 |          | inp9     |  0.97351 |  0.66627 | i  |  1, 9, 0 | Act_Identity |          |
 10 |          | inp10    |  1.56596 |  0.02014 | i  |  1,10, 0 | Act_Identity |          |
 11 |          | inp11    |  0.81645 | -0.52729 | i  |  1,11, 0 | Act_Identity |          |
 12 |          | inp12    | -0.88288 | -0.15885 | i  |  1,12, 0 | Act_Identity |          |
 13 |          | hid11    |  0.16206 | -0.55551 | h  |  7, 1, 0 |||
 14 |          | hid21    |  0.18053 | -0.68042 | h  | 13, 1, 0 |||
 15 |          | out1     | -0.54759 | 1461.39380 | o  | 19, 1, 0 | Act_Identity |          |
 16 |          | con11    |  0.20030 |  0.50000 | sh |  4,14, 0 | Act_Identity |          |
 17 |          | con21    |  0.31741 |  0.50000 | sh | 10,14, 0 | Act_Identity |          |
----|----------|----------|----------|----------|----|----------|--------------|----------|-------
```

The first column provides a count of the number of nodes or neurons; we see the entire network has a total of 17. The third column describes the type of neuron. We see there are 12 input neurons, 2 hidden neurons, 2 neurons in the context layer, and 1 output neuron. The forth and fifth columns give the value of the activation function and the bias for each neuron. For example, the first neuron has a activation function value of -0.98260, and a bias of 0.15181.

Prediction with an Elman Network

Now we are ready to use the model with the test sample. The `predict` method can be used to help out here:

```
> pred<-predict(fit, inputs[-train])
```

A scatter plot of the predictions and actual values is shown in Figure 10.4.

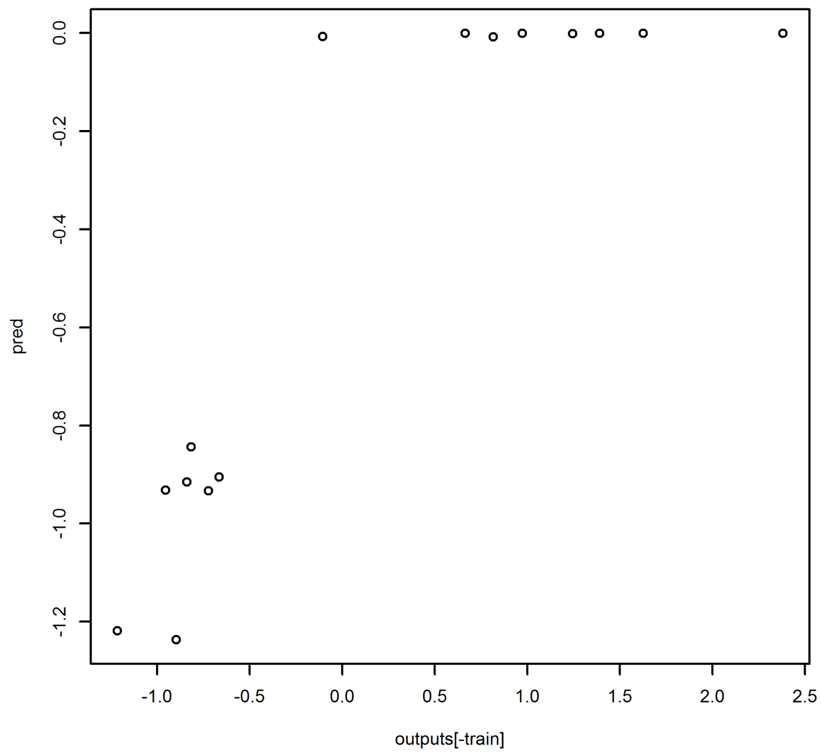


Figure 10.4: Elman network actual and predicted values

The squared correlation coefficient is relatively high at 0.78.

```
> cor(outputs[-train], pred)^2
        [,1]
[1,] 0.7845
```

Exercises

- **Question 1:** Build an Elman network with one hidden layer containing 12 nodes and assess the performance on the test sample.

- **Question 2:** What happens to performance if you use two hidden layers containing 2 nodes each?
- **Question 3:** Rebuild the model used in the text but using only x1 to x3 as the input attributes
- **Question 4:** Rebuild the model used in the text, but using only x1 to x5 as the input attributes.

Notes

[74]For additional context see:

- Doll, Richard, and Richard Peto. "Mortality in relation to smoking: 20 years' observations on male British doctors." BMJ 2.6051 (1976): 1525-1536.
- Doll, Richard, et al. "Mortality in relation to smoking: 50 years' observations on male British doctors." Bmj 328.7455 (2004): 1519.
- Burney, P., D. Jarvis, and R. Perez-Padilla. "The global burden of chronic respiratory disease in adults." The International Journal of Tuberculosis and Lung Disease 19.1 (2015): 10-20.

[75]See http://www.ra.cs.uni-tuebingen.de/SNNS/

[76]For further details see P. J. Diggle (1990) Time Series: A Biostatistical Introduction. Oxford,

Chapter 11

Jordan Neural Networks

JORDAN neural networks are similar to the Elman neural network. The only difference is that the context neurons are fed from the output layer instead of the hidden layer as illustrated in Figure 11.1.

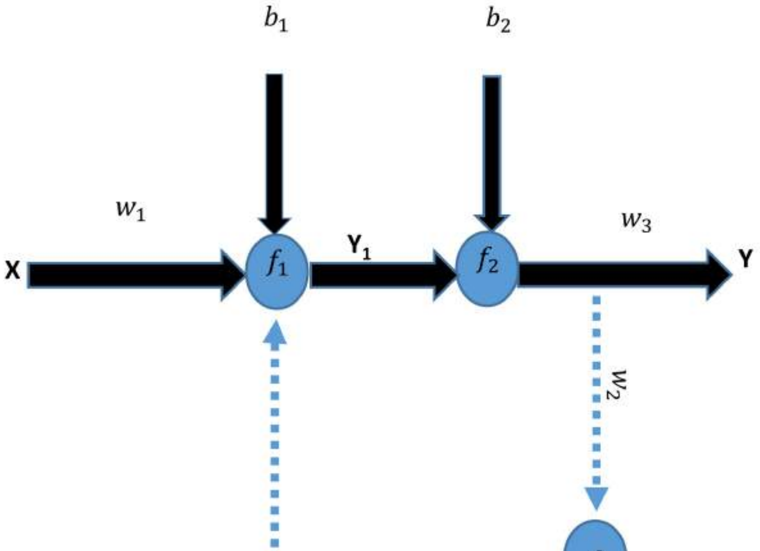


Figure 11.1: A simple Jordan neural network

The activity of the output node[s] is recurrently copied back into the context nodes. This provides the network with a memory of its previous state.

Practical Applications

Jordan neural networks have found an amazing array of uses. They appear capably of modeling timeseries data and are useful for classification problems. Below we outline a few illustrative examples of real world use of this predictive analytic tool.

Wind Forecasting

Accurate forecasts of wind speed in coastal regions are important for a wide range of industrial, transportation and social marine activity. For example, the prediction of wind speed is useful in determining the expected power output from wind turbines, operation of aircraft and navigation by ships, yachts and other watercraft. Civil Engineers Anurag and Deo[77] build Jordan neural networks in order to forecast daily, weekly as well as monthly wind speeds at two coastal locations in India.

Data was obtained from the India Meteorological Department covering a period of 12 years for the coastal location of Colaba within the Greater Mumbai (Bombay) region along the west coast of India. Three Jordan networks for daily, weekly and monthly wind speed forecasting were developed.

All three models had a mean square error less than 10%. However, the daily forecasts were more accurate than the weekly forecasts; and the weekly forecasts were more accurate than the monthly forecasts. The engineers also compare the network predictions to auto regressive integrated moving average (ARIMA) timeseries models; They observe "*The neural network forecasting is also found to be more accurate than traditional statistical timeseries analysis.*"

Classification of Protein-Protein interaction

Protein-protein interaction refers to the biological functions carried out by the proteins within the cell by interacting with other proteins in other cells as a result of biochemical events and/or electrostatic forces. Such interactions are believed to be important in understanding disease pathogenesis and developing new therapeutic approaches. A number of different perspectives have been used to study these interactions ranging from biochemistry, quantum chemistry, molecular dynamics, signal transduction, among others[78] All this information enables the creation of large protein interaction databases.

Computer scientists Dilpreet and Singh [79] apply Jordan neural networks to classify protein-protein interactions. The sample used in their analysis was derived from three existing databases[80]. It contained 753 positive patterns and 656 negative patterns[81]. Using amino acid composition of proteins as input to the Jordan network to classify the percentage of interacting and non-interacting proteins the researchers report a classification accuracy of 97.25%.

Recognizing Spanish Digits

Neural networks have been successfully applied to the difficult problem of speech recognition in English[82]. Accurate classification of the numerical digits 0 through 9 in Spanish using Jordan neural networks was investigated by researcher Tellez Paola[83].

In a rather small study, the speech of the ten numerical digits was recorded with voices from three women and three men. Each person was requested to repeat each digit four times. The network was then trained to classify the digits. Using nine random initializations' Tellez reports an average classification accuracy of 96.1%.

Notes

[77]More, Anurag, and M. C. Deo. "Forecasting wind with neural networks." Marine structures 16.1 (2003): 35-49.

[78]See for example:

- Herce, Henry D., et al. "Visualization and targeted disruption of protein interactions in living cells." Nature Communications 4 (2013).
- Hoppe, Philipp S., Daniel L. Coutu, and Timm Schroeder. "Single-cell technologies sharpen up mammalian stem cell research." Nature cell biology 16.10 (2014): 919-927.
- Li, Yao-Cheng, et al. "A Versatile Platform to Analyze Low-Affinity and Transient Protein-Protein Interactions in Living Cells in Real Time." Cell reports 9.5 (2014): 1946-1958.
- Qin, Weihua, et al. "DNA methylation requires a DNMT1 ubiquitin interacting motif (UIM) and histone ubiquitination." Cell Research (2015).

[79]Kaur, Dilpreet, and Shailendra Singh. "Protein-Protein Interaction Classification Using Jordan Recurrent Neural Network."

[80]Pfam, 3did and Negatome. For further details see:

- Robert D. Finn, John Tate et. al., “The Pfam protein families database”, Nucleic Acids Research, vol. 36, pp. 281–288, 2008.
- Amelie Stein, Robert B. Russell and Patrick Aloy, “3did: interacting protein domains of known three-dimensional structure”, Nucleic Acids Research, vol. 33, pp. 413–417, 2005.
- Pawel Smialowski, Philipp Page et. al., “The Negatome database: a reference set of non-interacting protein pairs”, Nucleic Acids Research, pp. 1–5, 2009.

[81]Positive patterns contain interacting residues in its center. Negative patterns contain non-interacting residues in its center.

[82]See:

- Lippmann, Richard P. "Review of neural networks for speech recognition." Neural computation 1.1 (1989): 1-38.
- Arisoy, Ebru, et al. "Bidirectional recurrent neural network language models for automatic speech recognition." Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on. IEEE, 2015.

- Sak, Haşim, et al. "Fast and Accurate Recurrent Neural Network Acoustic Models for Speech Recognition." arXiv preprint arXiv:1507.06947 (2015).
- Hinton, Geoffrey, et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups." Signal Processing Magazine, IEEE 29.6 (2012): 82-97.

[83]Paola, Tellez. "Recurrent Neural Prediction Model for Digits Recognition."International Journal of Scientific & Engineering Research Volume 2, Issue 11, November-2011.

Chapter 12

Modeling with Jordan Networks

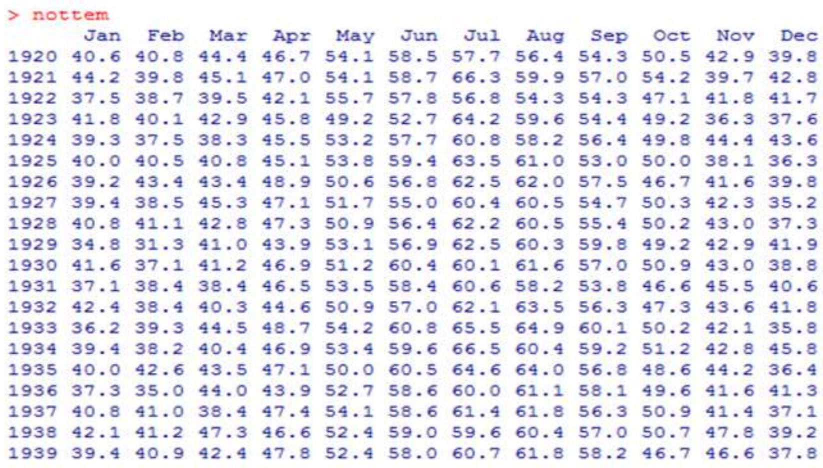
I grew up in the heart of England where the weather is always on the move. If you want to experience all four seasons in one day, central England is the place to visit! Anyway, in England, the weather is always a great conversation starter. So let's start our exploration of Jordan networks modelling British weather. To be specific, we will model the temperature of the city of Nottingham located in Nottinghamshire, England. You may recall this area was the hunting ground of the people's bandit Robin Hood[84]. Let's get our hands dirty and build a Jordan neural network right now! As with Elman networks, Jordan networks are great for modeling timeseries data.

Loading the Required Packages

We will use the `RSNNS` package along with the `quantmod` package. The data frame `nottem,` in the `datasets` package, contains monthly measurements on the average air temperature at Nottingham Castle[85], a location Robin Hood would have known well:

```
> require(RSNNS)
> data("nottem",package="datasets")
> require(quantmod)
```

Let's take a quick peek at the data held in `nottem`:



```
> nottem
      Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
1920 40.6 40.8 44.4 46.7 54.1 58.5 57.7 56.4 54.3 50.5 42.9 39.8
1921 44.2 39.8 45.1 47.0 54.1 58.7 66.3 59.9 57.0 54.2 39.7 42.8
1922 37.5 38.7 39.5 42.1 55.7 57.8 56.8 54.3 54.3 47.1 41.8 41.7
1923 41.8 40.1 42.9 45.8 49.2 52.7 64.2 59.6 54.4 49.2 36.3 37.6
1924 39.3 37.5 38.3 45.5 53.2 57.7 60.8 58.2 56.4 49.8 44.4 43.6
1925 40.0 40.5 40.8 45.1 53.8 59.4 63.5 61.0 53.0 50.0 38.1 36.3
1926 39.2 43.4 43.4 48.9 50.6 56.8 62.5 62.0 57.5 46.7 41.6 39.8
1927 39.4 38.5 45.3 47.1 51.7 55.0 60.4 60.5 54.7 50.3 42.3 35.2
1928 40.8 41.1 42.8 47.3 50.9 56.4 62.2 60.5 55.4 50.2 43.0 37.3
1929 34.8 31.3 41.0 43.9 53.1 56.9 62.5 60.3 59.8 49.2 42.9 41.9
1930 41.6 37.1 41.2 46.9 51.2 60.4 60.1 61.6 57.0 50.9 43.0 38.8
1931 37.1 38.4 38.4 46.5 53.5 58.4 60.6 58.2 53.8 46.6 45.5 40.6
1932 42.4 38.4 40.3 44.6 50.9 57.0 62.1 63.5 56.3 47.3 43.6 41.8
1933 36.2 39.3 44.5 48.7 54.2 60.8 65.5 64.9 60.1 50.2 42.1 35.8
1934 39.4 38.2 40.4 46.9 53.4 59.6 66.5 60.4 59.2 51.2 42.8 45.8
1935 40.0 42.6 43.5 47.1 50.0 60.5 64.6 64.0 56.8 48.6 44.2 36.4
1936 37.3 35.0 44.0 43.9 52.7 58.6 60.0 61.1 58.1 49.6 41.6 41.3
1937 40.8 41.0 38.4 47.4 54.1 58.6 61.4 61.8 56.3 50.9 41.4 37.1
1938 42.1 41.2 47.3 46.6 52.4 59.0 59.6 60.4 57.0 50.7 47.8 39.2
1939 39.4 40.9 42.4 47.8 52.4 58.0 60.7 61.8 58.2 46.7 46.6 37.8
```

There do not appear to be any missing values or rouge observations. So, we can continue.

We check the class of `nottem` using the `class` method:

```
> class(nottem)
[1] "ts"
```

It is a timeseries object of class `ts`. Knowing the class of your observations is critically important, especially if you are using dates or your analysis mixes different types of R classes. This can be a source of many hours of frustration. Fortunately, we now know `nottem` is of class `ts`; this will assist us in our analysis.

Figure 12.1 shows a timeseries plot of the observations in `nottem`. The data cover the years 1920 to 1939. There does not appear to be any trend evident in the data, however it does exhibit strong seasonality. You can replicate the chart by typing:

```
> plot(nottem)
```

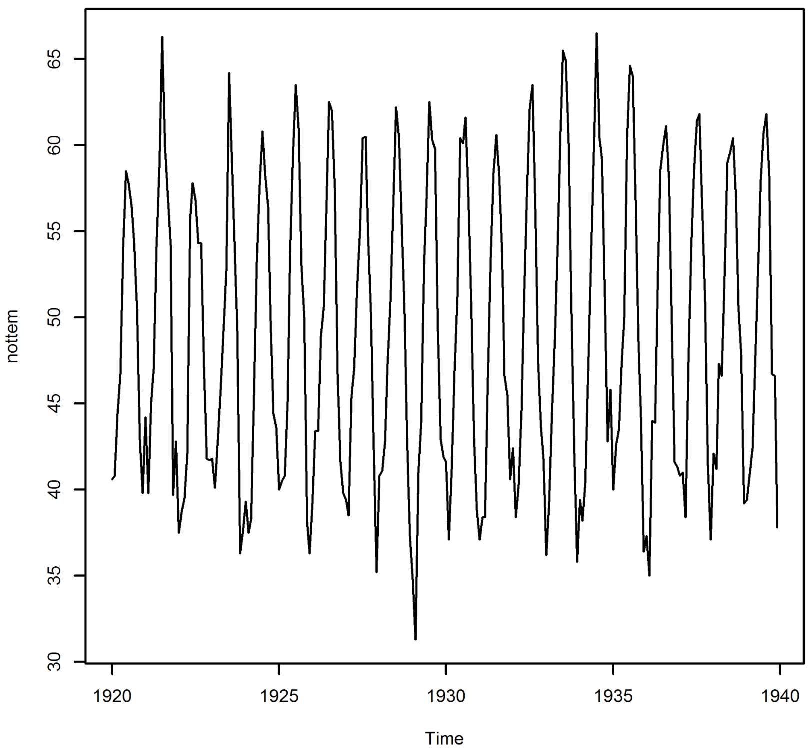


Figure 12.1: Average Monthly Temperatures at Nottingham 1920–1939

How to Transform Your Data

For the most part I like to standardize my attribute data prior to using a neural network model. In this case, given that we do not have any explanatory variables, we will take the log transformation and then use the `scale` method to standardize the data:

```
> y<-as.ts(nottem)
> y<-log(y)
> y<- as.ts(scale(y))
```

Since we are modeling data with strong seasonality characteristics which appear to depend on the month, we use monthly lags going back a full 12 months. This will give us 12 attributes to feed as inputs into the Jordan network. To use the `Lag` function in `quantmod` we need `y` to be a `zoo` class:

```
> y<-as.zoo(y)
> x1<-Lag(y, k = 1)
> x2<-Lag(y, k = 2)
> x3<-Lag(y, k = 3)
> x4<-Lag(y, k = 4)
> x5<-Lag(y, k = 5)
> x6<-Lag(y, k = 6)
> x7<-Lag(y, k = 7)
> x8<-Lag(y, k = 8)
> x9<-Lag(y, k = 9)
> x10<-Lag(y, k = 10)
> x11<-Lag(y, k = 11)
> x12<-Lag(y, k = 12)
```

As with the Elman network, we need to remove the lagged values that contain NA's (see page 149 for further details). The final cleaned values are stored in the R object `temp`:

```
> temp<-cbind(x1,x2,x3,x4,x5,x6,x7,x8,x9,
   x10,x11,x12)
> temp<-cbind(y,temp)
> temp <- temp[-(1:12),]
```

As a final check, let's visually inspect all of the attributes and response variable. The result is shown in Figure 12.2, and was created using the `plot` method:

```
>plot(temp)
```

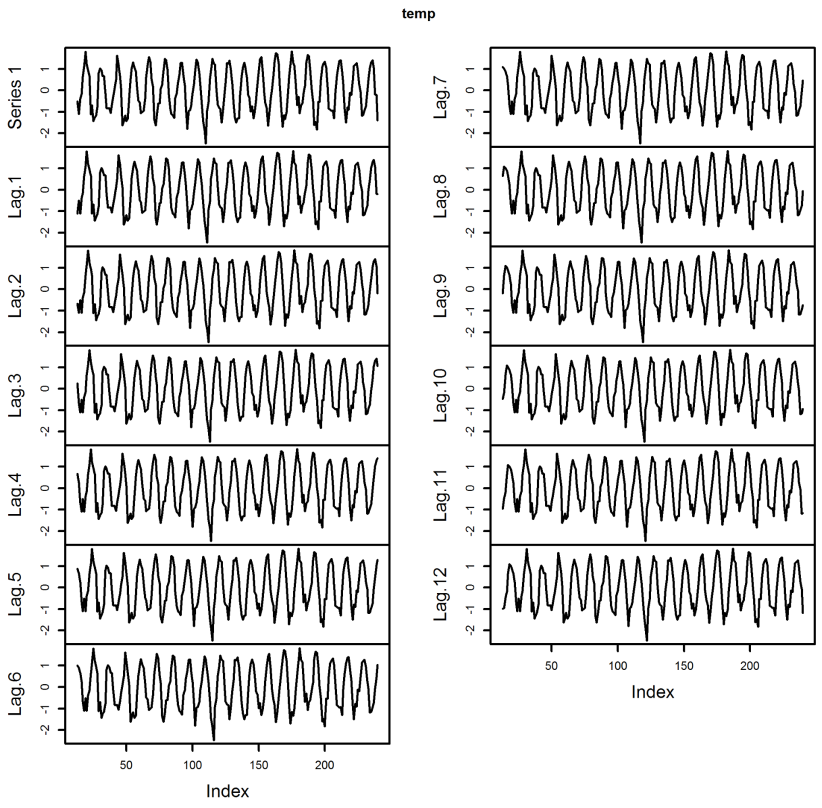


Figure 12.2: Response and attribute variables for Jordan network

Notice that `Series 1` is the response variable `y`, and `Lag 1`, `Lag 2`,...,`Lag 12` are the input attributes `x1`, `x2`,...,`x12`.

Selecting the Training Sample

First we check the number of observations (should be equal to 228), then we use the `set.seed` method to ensure reproducibility:

```
>  n=nrow(temp)
>  n
[1] 228
> set.seed(465)
```

For the training sample 190 observations are randomly selected without replacement as follows:

```
> n_train <- 190
> train <- sample(1:n,n_train , FALSE)
```

How to Estimate the Model

The model is estimated along similar lines as the Elman neural network on page 150. The attributes are stored in the R object `inputs`. The response variable is stored in the R object `outputs`. The model is then fitted using 2 hidden nodes, with a maximum of 1000 iterations and a learning rate parameter set to 0.01:

```
> inputs <- temp[,2:13]
> outputs <- temp[,1]

> fit <- jordan(inputs[train],
outputs[train],
size=2,
learnFuncParams=c(0.01),
maxit=1000)
```

The plot of the training error by iteration is shown in Figure 12.3. It was created using:

```
> plotIterativeError(fit)
```

The error falls sharply within the first 100 or so iterations and by around 300 iterations is stable.

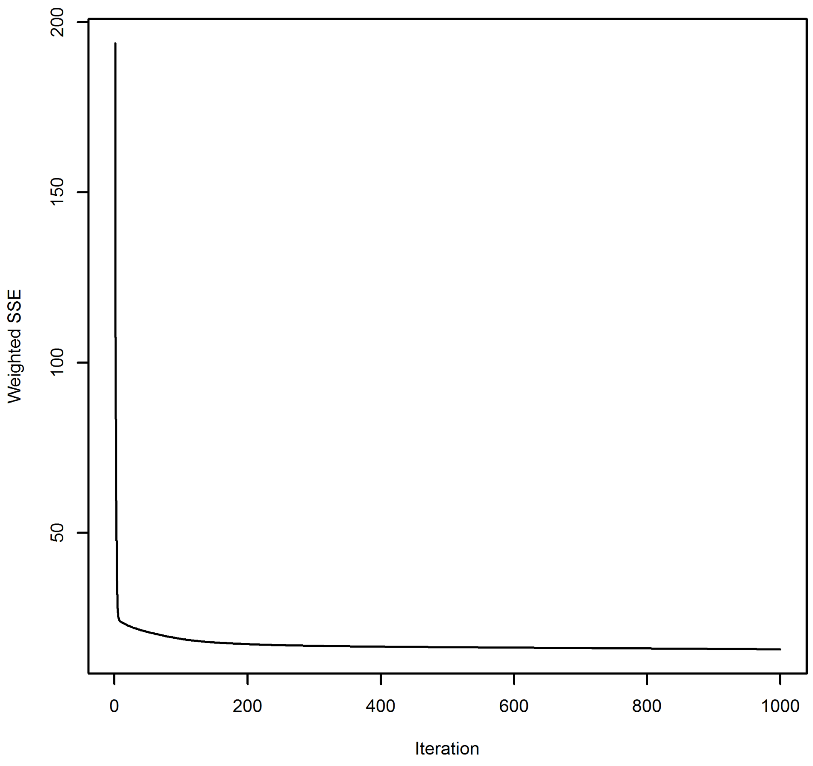


Figure 12.3: Training error by iteration

Finally, we can use the **predict** method to forecast the values from the test sample. We also calculate the squared correlation between the test sample response and the predicted values:

```
> pred<-predict(fit, inputs[-train])

> cor(outputs[-train], pred)^2
          [,1]
[1,] 0.9050079
```

The squared correlation coefficient is relatively high at 0.90. We can at least say our model has had some success in predicting the weather in Nottingham - Robin Hood would be amazed!

Exercises

- **Question 1:** Re-estimate the model given in the text with a learning rate of 0.1 and assess the fit on the test sample.
- **Question 2:** Keeping the learning rate at 0.01, estimate the model with 6 hidden layer nodes.
- **Question 3:** Now re-estimate using 12 hidden layer neurons.
- **Question 4:** Rebuild the model used in question 1 but using only x1 to x5 as the input attributes.

Notes

[84]See Knight, Stephen. Robin Hood: a complete study of the English outlaw. Blackwell Publishers, 1994.

[85]If you are planning to visit, before you go take a look at:

- Hutchinson, Lucy, and Julius Hutchinson. Memoirs of the Life of Colonel Hutchinson, Govenor of Nottingham Castle and Town. G. Bell and sons, 1906.
- Drage, Christopher. Nottingham Castle: a place full royal. Nottingham Civic Society [in association with] The Thoroton Society of Nottinghamshire, 1989.
- Hooper-Greenhill, Eilean, and Theano Moussouri. Visitors' Interpretive Strategies at Nottingham Castle Museum and Art Gallery. No. 2. Research Centre for Museums and Galleries and University of Leicester, 2001.

Solution to Exercises

Chapter 2

- **Question 1:**

All we need to do is set `hidden = 6` and re-run code. You should see something like this:

```
> set.seed(103)
> fit1<- neuralnet(f,
 data = data[train,],
 hidden=6,
 algorithm = "rprop+",
 err.fct = "sse",
 act.fct = "logistic",
 linear.output=FALSE)

> print(fit1)
Call: neuralnet(formula = f, data = data[
   train, ], hidden = 6, algorithm = "rprop
   +",     err.fct = "sse", act.fct = "
   logistic", linear.output = FALSE)

1 repetition was calculated.

        Error Reached Threshold Steps
1 1828.890459    0.009396198984  3698
```

- **Question 2:**

Set `algorithm = "rprop-"` and re-run the code:

```
>  set.seed(103)
> fit2<- neuralnet(f,
 data = data[train,],
 hidden=6,
algorithm = "rprop-",
err.fct = "sse",
 act.fct = "logistic",
 linear.output=FALSE)

> print(fit2)
Call: neuralnet(formula = f, data = data[
   train, ], hidden = 6, algorithm = "rprop
   -",     err.fct = "sse", act.fct = "
   logistic", linear.output = FALSE)

1 repetition was calculated.

        Error Reached Threshold Steps
1 1823.284806    0.009739367719 28568
```

- **Question 3:**

Need to specify `f<-Class ~hosp + health + numchron` and set `hidden=2`:

```
> set.seed(103)
> f<-Class ~hosp + health + numchron
> fit3<- neuralnet(f,
 data = data[train,],
 hidden=2,
 algorithm = "rprop-",
 err.fct = "sse",
 act.fct = "logistic",
 linear.output=FALSE)

> print(fit3)
Call: neuralnet(formula = f, data = data[
   train, ], hidden = 2, algorithm = "rprop
   -",     err.fct = "sse", act.fct = "
   logistic", linear.output = FALSE)

1 repetition was calculated.

        Error Reached Threshold Steps
1 1878.360131     0.008920075304  1119
```

- **Question 4:**

Use the `compute` method and then calculate the confusion matrix as follows:

```
> pred <- compute(fit3, data[-train,2:4] )

>  r2 <- ifelse(pred$net.result <= 0.5, -1,
    1)

> table( sign(r2),sign(data[-train,1]) ,
   dnn =c("Predicted" , " Observed"))
         Observed
Predicted  -1   1
       -1 214 178
       1    2  12
> error_rate = (1- sum( sign(r2) == sign(
   data[-train,1])  )/406 )
>  round( error_rate ,2)
[1] 0.44
```

Chapter 3

- **Question 1:**

First we fit the model:

```
> set.seed(103)
> fit1<- neuralnet(f, data = data[train,],
hidden=c(1,1),
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",
linear.output=FALSE)
```

Next the predictions:

```
> pred <- compute(fit1, data[-train,2:7] )

>  head(round(pred$net.result,3),6)
    [,1]
1  0.000
15 0.000
16 0.666
46 0.000
76 0.000
82 0.000

>  r2 <- ifelse(pred$net.result <= 0.5, -1,
    1)

> head(r2,6)
   [,1]
1    -1
15   -1
16    1
46   -1
76   -1
82   -1
```

Finally, the error:

```
> table( sign(r2),sign(data[-train,1]) ,
   dnn =c("Predicted" , " Observed"))
```

```
          Observed
Predicted  -1   1
       -1 205 157
       1   11  33

> error_rate = (1- sum( sign(r2) == sign(
   data[-train,1])  )/406 )
>  round( error_rate ,2)
[1] 0.41
```

- **Question 2:**

The number of steps before the algorithm finds a solution should increase dramatically. Of course, a solution may not be found. Let's check the number of steps taken by `fit1`:

```
> fit1
Call: neuralnet(formula = f,
data = data[train, ],
hidden = c(1, 1),
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",
linear.output = FALSE)

1 repetition was calculated.

        Error Reached Threshold Steps
1 1854.648392    0.009376957938 16285
```

Now we run a new model using `slr`:

```
> fit2<- neuralnet(f, data = data[train,],
   hidden=c(1,1),algorithm = "slr",err.fct
   = "sse", act.fct = "logistic",linear.
   output=FALSE)
```

We get the message:

```
Warning message:
algorithm did not converge in 1 of 1
   repetition(s) within the stepmax
```

- **Question 3:**

Lots of ways to do this. I'll use the `unscale` function in the `grt` package:

```
> require(grt)
> z<-unscale(data)
> set.seed(103)
> fit3<- neuralnet(f, data = z[train,],
hidden=c(1,1),
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",
linear.output=FALSE)

>  pred <- compute(fit3, z[-train,2:7] )

> r2 <- ifelse(pred$net.result <= 0.5, -1,
   1)

> table( sign(r2),sign(z[-train,1]) ,  dnn
   =c("Predicted" , " Observed"))
           Observed
Predicted   -1   1
       -1  206 168
       1    10  22

> error_rate = (1- sum( sign(r2) == sign(z
   [-train,1])  )/406 )

>  round( error_rate ,2)
[1] 0.44
```

- **Question 4:**

All we need to do here is set `linear.output=TRUE`. You should observe results similar to these:

```
> set.seed(103)
>  fitM4<- neuralnet(f, data = z[train,],
  hidden=c(2,2),
  rep=10,
  algorithm = "rprop+",
  err.fct = "sse",
  act.fct = "logistic",
  linear.output=TRUE)

Warning message:
algorithm did not converge in 9 of 10
   repetition(s) within the stepmax
```

.

We will use the 1 repetition that did converge to complete our calculation:

```
>  pred <- compute(fitM4, z[-train,2:7] )

> r2 <- ifelse(pred$net.result <= 0.5, -1,
   1)

> table( sign(r2),sign(z[-train,1]) ,  dnn
   =c("Predicted" , " Observed"))
          Observed
Predicted  -1   1
       -1 211 169
       1    5  21

> error_rate = (1- sum( sign(r2) == sign(z
   [-train,1])  )/406 )
>  round( error_rate ,2)
[1] 0.43
```

Chapter 4

- **Question 1:**

Fit a model similar to that shown in the chapter but with `size` = 1.

```
> set.seed(103)
> fitN1<- nnet(f, data = data[train,],size
   =1, rang = 0.1,decay = 5e-4, maxit =
   200,softmax=FALSE)
# weights:  9
initial  value 4903.068779
iter  10 value 4021.733983
iter  20 value 3997.764689
iter  30 value 3997.041811
iter  40 value 3730.602456
iter  50 value 3710.435482
iter  60 value 3709.988307
iter  70 value 3709.641878
final  value 3709.636706
converged

> predN1<-predict(fitN1, data[-train,2:7],
   type = c("raw"))

>  rN1 <- ifelse(predN1 <= 0.5, -1, 1)
> table( sign(rN1),
sign(data[-train,1]) ,
dnn =c("Predicted" , " Observed"))

          Observed
Predicted  -1   1
       -1 205 157
       1   11  33
> error_rate = (1- sum( sign(rN1) == sign(
   data[-train,1])  )/406 )
```

```
>  round( error_rate ,2)
[1] 0.41
```

- **Question 2:**

Set the size = 2.

```
> set.seed(103)
>   fitN2<- nnet(f, data = data[train,],
   size =2, rang = 0.1,decay = 5e-4, maxit
   = 200,softmax=FALSE)

# weights:  17
initial  value 4974.587218
iter  10 value 4036.661248
iter  20 value 4005.879135
iter  30 value 3809.832794
iter  40 value 3698.657586
iter  50 value 3685.832963
iter  60 value 3683.384473
iter  70 value 3682.655357
iter  80 value 3678.127361
iter  90 value 3677.715787
final  value 3677.708854
converged
```

- **Question 3:**

Use the `table` method. We get a misclassification error rate of 35%.

```
> table( sign(rN2),
sign(data[-train,1]),
dnn =c("Predicted" , " Observed"))

          Observed
Predicted  -1   1
```

```
       -1  223  125
       1    16   42
> error_rate = (1- sum( sign(rN2) == sign(
   data[-train,1])  )/406 )
>  round( error_rate ,2)
[1] 0.35
```

- **Question 4:**

Use the `while` loop given in the text and set `size = 2`.

```
> set.seed(103)

> boot<-1000
> i=0

> error<-(1:boot)
> dim(error)

> dim(error) <- c(boot)

> while(i<=boot)
 {

 train <- sample(1:n, 4000, FALSE)


fitB<- nnet(f, data = data[train,],
size =2,
rang = 0.1,
decay = 5e-4,
maxit = 200,
softmax=FALSE,
trace=FALSE)

predB<-predict(fitB, data[-train,2:7], type
    = c("raw"))
rB <- ifelse(predB <= 0.5, -1, 1)
error[i] = (1- sum( sign(rB) == sign(data[-
   train,1])  )/406 )

 i=i+1

+ }

> round(min(error),3)
[1] 0.303
> round(max(error),3)
[1] 0.495
> round(median(error),3)
[1] 0.394
> round(mean(error),3)
[1] 0.395
```

Chapter 6

- **Question 1:**

Add `fit1` to the for loop, create confusion matrix and calculate the error rate:

```
> pred<-1:n_test
> for (i in 1:n_test)
 {
 pred[i]<-guess(fit1,
as.matrix(temp[-train,][i,]))$category
 }

> table( pred,y[-train] ,  dnn =c("
   Predicted" , " Observed"))

          Observed
Predicted neg pos
```

```
      neg  76   3
      pos   5  40
> error_rate = (1- sum( pred == y[-train])
    / n_test)
>  round( error_rate ,3)
[1] 0.065
```

- **Question 2:**

Add `fit2` to the for loop, create confusion matrix and calculate the error rate:

```
> pred<-1:n_test
> for (i in 1:n_test)
 {
 pred[i]<-guess(fit2,
as.matrix(temp[-train,][i,]))$category
 }

> table( pred,y[-train] ,  dnn =c("
   Predicted" , " Observed"))

         Observed
Predicted neg pos
      neg  73   5
      pos   8  38
> error_rate = (1- sum( pred == y[-train])
    / n_test)
>  round( error_rate ,3)
[1] 0.105
```

- **Question 3:**

Use `smooth` method and report important performance statistics:

```
> fit3 <- smooth(fit_basic,sigma=0.1)

> result3<- perf(fit3)

> result3$bic
[1] -1712.937

> result3$success_rate
[1] 0.9466667
```

- **Question 4:**

Add `fit3` to the `for` loop, create confusion matrix and calculate the error rate:

```
> pred<-1:n_test
> for (i in 1:n_test)
 {
 pred[i]<-guess(fit3, as.matrix(temp[-train
    ,][i,]))$category
 }

> table( pred,y[-train] ,  dnn =c("
   Predicted" , " Observed"))

          Observed
Predicted neg pos
      neg  79   4
      pos   2  39
> error_rate = (1- sum( pred == y[-train])
    / n_test)

>  round( error_rate ,3)
[1] 0.048
```

Chapter 8

- **Question 1:**

Use `smooth` method with sigma set to the appropriate value:

```
> fit <- smooth(fit_basic, sigma = 0.1)
> pred<-1:n_test

> for (i in 1:n_test)
+ {
+ pred[i]<-guess(fit, as.matrix(x[-train,][
   i,]))
+ }
>  round(cor(pred,y[-train])^2,3)
[1] 0.884
```

- **Question 2:**

Use `smooth` method with sigma set to the appropriate value:

```
> fit <- smooth(fit_basic, sigma = 4)
> pred<-1:n_test
> for (i in 1:n_test)
+ {
+ pred[i]<-guess(fit, as.matrix(x[-train,][
   i,]))
+ }
>  round(cor(pred,y[-train])^2,3)
[1] 0.916
```

- **Question 3:**

Here is one way to achieve this:

```
> keeps<-c("waistcirc",  "hipcirc"
, "elbowbreadth",    "kneebreadth")
```

```
> x <-x[keeps]

> fit_basic <- learn(data.frame(y[train],x[
   train,]))
> fit <- smooth(fit_basic, sigma = 2.5)
> pred<-1:n_test

> for (i in 1:n_test)
+ {
+ pred[i]<-guess(fit, as.matrix(x[-train,][
   i,]))
+ }

>  round(cor(pred,y[-train])^2,3)
[1] 0.884
```

- **Question 4:**

Similar to question 3 with smoothing parameter set to 0.5:

```
> keeps<-c("waistcirc",  "hipcirc" ,
"elbowbreadth",   "kneebreadth")

> x <-x[keeps]
> fit_basic <- learn(data.frame(y[train],x[
   train,]))
> fit <- smooth(fit_basic, sigma = 0.5)
> pred<-1:n_test
> for (i in 1:n_test)
+ {
+ pred[i]<-guess(fit, as.matrix(x[-train,][
   i,]))
+ }

>  round(cor(pred,y[-train])^2,3)
[1] 0.882
```

Chapter 10

- **Question 1:**

We need to set `size = c(12)`:

```
> set.seed(2016)
> fit1 <- elman(inputs[train],
outputs[train],
size=c(12),
learnFuncParams=c(0.1),
maxit=1000)

> pred1<-predict(fit1, inputs[-train])
> cor(outputs[-train], pred1)^2
          [,1]
[1,] 0.5712067
```

- **Question 2:**

Set `size=c(2,2)`:

```
> set.seed(2016)

> fit2 <- elman(inputs[train],
outputs[train],
size=c(2,2),
learnFuncParams=c(0.1),
maxit=1000)

> pred2<-predict(fit2, inputs[-train])
> cor(outputs[-train], pred2)^2
          [,1]
[1,] 0.7315711
```

- **Question 3:**

We collect together the required attributes using `inputs3 <- deaths[,2:4]`:

```
> set.seed(2016)
> inputs3 <- deaths[,2:4]
> fit3 <- elman(inputs3[train],
outputs[train],
size=c(1,1),
learnFuncParams=c(0.1),
maxit=1000)

> pred3<-predict(fit3, inputs3[-train])
> cor(outputs[-train], pred3)^2
          [,1]
[1,] 0.6869871
```

- **Question 4:**

We collect together the required attributes using `inputs3 <- deaths[,2:6]`:

```
> set.seed(2016)
> inputs3 <- deaths[,2:6]

> fit3 <- elman(inputs3[train],
outputs[train],
size=c(1,1),
learnFuncParams=c(0.1),
maxit=1000)

> pred3<-predict(fit3, inputs3[-train])
> cor(outputs[-train], pred3)^2
          [,1]
[1,] 0.8046986
```

Chapter 12

- **Question 1:**

Set `learnFuncParams=c(0.01)`:

```
> set.seed(2016)
> fit1 <- jordan(inputs[train],
outputs[train],
size=2,
learnFuncParams=c(0.01),
maxit=1000)

> pred1<-predict(fit1, inputs[-train])

> cor(outputs[-train], pred1)^2
          [,1]
[1,] 0.9151276
```

- **Question 2:**

Set `size=6`:

```
> set.seed(2016)

> fit2 <- jordan(inputs[train],
outputs[train],
size=6,
learnFuncParams=c(0.01),
maxit=1000)

> pred2<-predict(fit2, inputs[-train])
> cor(outputs[-train], pred2)^2
          [,1]
[1,] 0.9141914
```

- **Question 3:**

Set size=12:

```
> set.seed(2016)
> fit3 <- jordan(inputs[train],
outputs[train],
size=12,
learnFuncParams=c(0.01),
maxit=1000)

> pred3<-predict(fit3, inputs[-train])

> cor(outputs[-train], pred3)^2
          [,1]
[1,] 0.8735576
```

- **Question 4:**

Use inputs4 <- temp[,2:6]:

```
> set.seed(2016)
> inputs4 <- temp[,2:6]

> fit4 <- jordan(inputs4[train],
outputs[train],
size=12,
learnFuncParams=c(0.01),
maxit=1000)

> pred4<-predict(fit4, inputs4[-train])

> cor(outputs[-train], pred4)^2
          [,1]
[1,] 0.8834135
```

Appendix

The cfac function

The `cfac` function, given below, is from the research paper of Zeileis and Kleiber[86].

```
cfac <- function(x, breaks = NULL)
{
if(is.null(breaks))
breaks <- unique(quantile(x, 0:10/10))
x <- cut(x,
breaks,
include.lowest = TRUE,
right = FALSE)
levels(x) <- paste(breaks[-length(breaks)],
ifelse(diff(breaks) > 1,
c(paste("-",
breaks[-c(1, length(breaks))] - 1,
sep = ""), "+"), ""),
sep = "") + return(x)
}
```

Notes

[86]Zeileis, A. and Kleiber, C. and Jackma, S. (2008). "Regression Models for Count Data in R". JSS 27, 8, 1–25. The paper is available at http://www.jstatsoft.org/v27/i08/paper

Index

Congratulations!

You made it to the end. Here are three things you can do next.

1. Pick up your **FREE** copy of **12 Resources to Supercharge Your Productivity in R** 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.

Good luck!

Dr. Nigel D. Lewis

P.S. Thanks for allowing me to partner with you on your data analysis journey.