Ego autem et domus

mea serviemus Domino.

DEEP LEARNING MADE EASY WITH R

A Gentle Introduction for Data Science.

Dr. N.D. Lewis

Image photography by Deanna Lewis

ISBN: 978-1519514219
ISBN: 1519514212

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.

Master Deep Learning with this fun, practical, hands on guide.

With the explosion of big data deep learning is now on the radar. Large companies such as Google, Microsoft, and Facebook have taken notice, and are actively growing in-house deep learning teams. Other large corporations are quickly building out their own teams. If you want to join the ranks of today's top data scientists take advantage of this valuable book. It will help you get started. It reveals how deep learning models work, and takes you under the hood with an easy to follow process showing you how to build them faster than you imagined possible using the powerful, free R predictive analytics package.

NO EXPERIENCE REQUIRED - Bestselling decision scientist Dr. N.D Lewis builds deep learning models for fun. Now he shows you the shortcut up the steep steps to the very top. It's easier than you think. Through a simple to follow process you will learn how to build the most successful deep learning models used for learning from data. Once you have mastered the process, it will be easy for you to translate your knowledge into your own powerful applications.

If you want to accelerate your progress, discover the best in deep learning and act on what you have learned, this book is the place to get started.

YOU'LL LEARN HOW TO:

- **Develop Recurrent Neural Networks**
- **Build Elman Neural Networks**
- **Deploy Jordan Neural Networks**
- **Create Cascade Correlation Neural Networks**
- **Understand Deep Neural Networks**

- **Use Autoencoders**
- **Unleash the power of Stacked Autoencoders**
- **Leverage the Restricted Boltzmann Machine**
- **Master Deep Belief Networks**

Once people have a chance to learn how deep learning can impact their data analysis efforts, they want to get hands on with the tools. This book will help you to start building smarter applications today using R. Everything you need to get started is contained within this book. It is your detailed, practical, tactical hands on guide - the ultimate cheat sheet for deep learning mastery.

A book for everyone interested in machine learning, predictive analytic techniques, neural networks and decision science. Buy the book today. Your next big breakthrough using deep learning is only a page away!

Other Books by N.D Lewis

- **Build Your Own Neural Network TODAY!** Build neural network models in less time than you ever imagined possible. This book contains an easy to follow process showing you how to build the most successful neural networks used for learning from data using R.

- **92 Applied Predictive Modeling Techniques in R:** AT LAST! Predictive analytic methods within easy reach with R...This jam-packed book takes you under the hood with step by step instructions using the popular and free R predictive analytic package. It provides numerous examples, illustrations and exclusive use of real data to help you leverage the power of predictive analytics. A book for every data analyst, student and applied researcher.

- **100 Statistical Tests in R:** Is designed to give you rapid access to one hundred of the most popular statistical tests. It shows you, step by step, how to carry out these tests in the free and popular R statistical package. The book was created for the applied researcher whose primary focus is on their subject matter rather than mathematical lemmas or statistical theory.

- **Visualizing Complex Data Using R**: Wish you had fresh ways to present data, explore relationships, visualize your data and break free from mundane charts and diagrams? In this book you will find innovative ideas to unlock the relationships in your own data and create killer visuals to help you transform your next presentation from good to great. Visualizing complex relationships with ease using R begins here.

For further details visit www.AusCov.com

Preface

THIS is a book for everyone who is interested in deep learning. Deep learning isn't just for multinational corporations or large university research departments. The ideas and practical information you'll find here will work just as well for the individual data scientist working for a small advertising firm, the team of three decision scientists employed by a regional pet foods company, the student completing a project on deep learning for their coursework, or the solo consultant engaged on a forecasting project for the local health authority. You don't need to be a genius statistician or programming guru to understand and benefit from the deep learning ideas discussed in this text.

The purpose of this book is to introduce deep learning techniques to data scientists, applied researchers, hobbyists and others who would like to use these tools. It is not intended to be a comprehensive theoretical treatise on the subject. Rather, the intention is to keep the details to the minimum while still conveying a good idea of what can be done and how to do it using R. The focus is on the "how" because it is this knowledge that will drive innovation and actual practical solutions for you. As Anne Isabella Richie, daughter of writer William Makepeace Thackeray, wrote in her novel Mrs. Dymond "*...if you give a man a fish he is hungry again in an hour. If you teach him to catch a fish you do him a good turn*".

This book came out of the desire to put the powerful technology of deep learning into the hands of the everyday practitioner. The material is therefore designed to be used by the individual whose primary focus is on data analysis and modeling. The focus is solely on those techniques, ideas and strategies that have been proven to work and can be quickly digested and deployed in the minimal amount of time.

On numerous occasions, individuals in a wide variety of disciplines and industries, have asked "how can I quickly un-

derstand and use the techniques of deep learning in my area of interest?" 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, using the techniques of deep learning is now easier than ever. This book is designed to give you rapid access. It shows you, step by step, how to build each type of deep learning 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 R package. 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, numerous applications of deep learning 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 the models using R. 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.

In the end, deep learning isn't about mathematical lemmas or proving theorems. It's ultimately about real life, real people and the application of machine learning algorithms to real world problems in order to drive useful solutions. No matter who you are, no matter where you are from, no matter your background or schooling, you have the ability to use the ideas outlined in this book. With the appropriate software tool, a little persistence and the right guide, I personally believe deep learning techniques can be successfully used in the hands of anyone who has a real interest.

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 deep learning models. 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.

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. Knowledge of how to build and apply deep learning 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.

A Promise

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

I invite you therefore to put what you read in these pages into action. To help you do that, I've created **"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.

Now, it's your turn!

How to Get the Most from this Book

This is a hands on, role up your sleeves and experiment with the data and R book. You will get the maximum benefit by typing in the examples, reading the reference material and experimenting. By working through the numerous examples and reading the references, you will broaden your knowledge, deepen you intuitive understanding and strengthen your practical skill set.

There are at least two other ways to use this book. 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. Alternatively, browse through the real world examples, illustrations, case studies, tips and notes to stimulate your own ideas.

☛ *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 `RSNNS` package you would type in the R console:

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

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

```
> require(RSNNS)
```

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

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 `elman` function, you would type:

```
?elman
```

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.

> ☛ ***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)
> ```

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.

The R package 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 deep learning models; more precisely, it is an attempt to give you the tools you need to build these models easily and quickly using R. 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 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!

Chapter 1

Introduction

In God we trust. All others must bring data.

W. Edwards Deming

In elementary school, Mrs. Masters my teacher, taught me and my fellow students about perspective. Four students were "volunteered" to take part, in what turned out to be a very interesting experiment. Each volunteer was blindfolded and directed towards a large model of a donkey. The first student was led to the tail, and asked to describe what she felt. The second student was lead to the leg, and again asked to describe what he felt. The same thing was repeated for the remaining two students, each led to a different part of the donkey.

Because no one of the blindfolded students had complete information, the descriptions of what they were feeling were widely inaccurate. We laughed and giggled at the absurdity of the suggestions of our fellow blindfolded students. It was quite clearly a donkey; how could it be anything else! Once the blindfold was taken off the students, they too laughed. Mrs. Masters was an amazing teacher.

In this chapter I am going to draw a picture of the complete "donkey" for you, so to speak. I'll give an overview of deep learning, identify some of the major players, touch on areas

where it has already been deployed, outline why you should add it to your data science toolkit, and provide a primer on neural networks, the foundation of the deep learning techniques we cover in this book.

What is Deep Learning?

Deep learning is an area of machine learning that emerged from the intersection of neural networks, artificial intelligence, graphical modeling, optimization, pattern recognition and signal processing. The models in this emerging discipline have been exulted by sober minded scholars in rigorous academic journals[1] "*Deep learning networks are a revolutionary development of neural networks, and it has been suggested that they can be utilized to create even more powerful predictors.*" Faster computer processors, cheaper memory and the deluge of new forms of data allow businesses of all sizes to use deep learning to transform real-time data analysis.

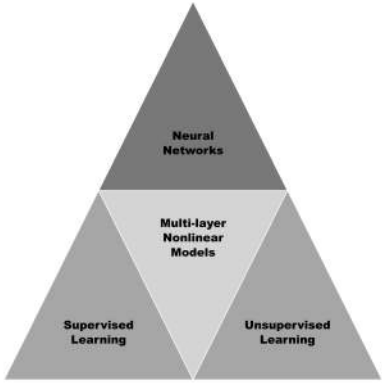


Figure 1.1: The deep learning pyramid

Deep learning is about supervised or unsupervised learn-

ing from data using multiple layered machine learning models. The layers in these models consist of multiple stages of nonlinear data transformations, where features of the data are represented at successively higher, more abstract layers.

Figure 1.1 illustrates the deep learning pyramid. At the base are the two core styles of learning from data, supervised learning and unsupervised learning. The core element of nonlinear data transformation lies at the center of the pyramid, and at the top, in this book, we consider various types of neural networks.

NOTE... ✍

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

1. **Supervised learning**: Your training data contain 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.

As we work together through the core ideas involved in deep learning and useful models using R, the general approach we take is illustrated in Figure 1.2. Whatever specific machine learning model you develop, you will always come back to this basic diagram. Input data is passed to the model and filtered through multiple non-linear layers. The final layer consists of a classifier which determines which class the object of interest belongs to.

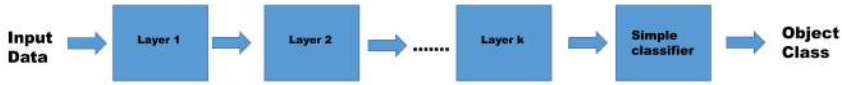


Figure 1.2: General deep learning framework

The goal in learning from data is to predict a response variable or classify a response variable using a group of given attributes. This is somewhat similar to what you might do with linear regression, where the dependent variable (response) is predicted by a linear model using a group of independent variables (aka attributes or features). However, traditional linear regression models are not considered deep because they do apply multiple layers of non-linear transformation to the data.

Other popular learning from data techniques such as decision trees, random forests and support vector machines, although powerful tools[2], are not deep. Decision trees and random forests work with the original input data with no transformations or new features generated; whilst support vector machines are considered shallow because they only consist of a kernel and a linear transformation. Similarly, single hidden layer neural networks are also not considered deep as they consist of only one hidden layer[3].

What Problems Can Deep Learning Solve?

The power of deep learning models comes from their ability to classify or predict nonlinear data using a modest number of parallel nonlinear steps[4]. A deep learning model learns the input data features hierarchy all the way from raw data input to the actual classification of the data. Each layer extracts features from the output of the previous layer.

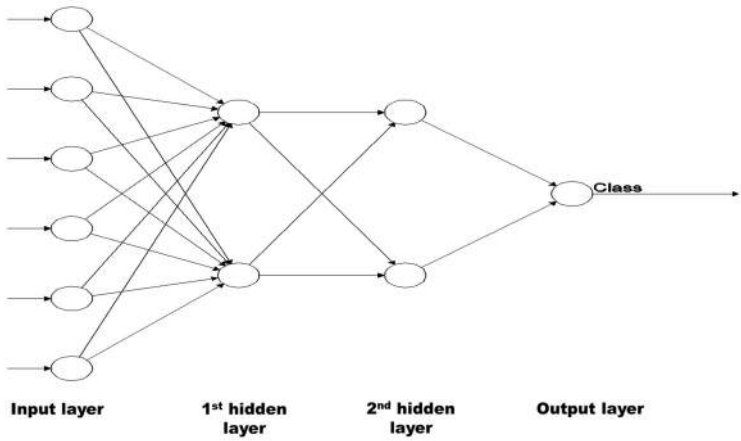


Figure 1.3: Feed forward neural network with 2 hidden layers

The deep learning models we consider throughout this text are neural networks with multiple hidden layers. The simplest form of deep neural network, as shown in Figure 1.3, contains at least two layers of hidden neurons, where each additional layer processes the output from the previous layer as input[5].

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

Who Uses Deep Learning?

Deep learning technology is being used commercially in the health care industry, medical image processing[6], natural-language processing and in advertising to improve click through rates. Microsoft, Google, IBM, Yahoo, Twitter, Baidu, Paypal and Facebook are all exploiting deep learning to understand user's preferences so that they can recommend targeted ser-

vices and products. It is popping up everywhere, it is even on your smartphone where it underpins voice assisted technology.

NOTE... ✍

The globally distributed and widely read IEEE[7] Spectrum magazine reported[8]"*the demand for data scientists is exceeding the supply. These professionals garner high salaries and large stock option packages...*" According to the McKinsey Global Institute, the United States alone faces a shortage of 140,000 to 190,000 data scientists with the appropriate skills[9]. Whilst the Harvard Business Review declared data science as the *sexiest* job of the 21st century[10].

You would be hard pressed to think of an area of commercial activity where deep learning could not be beneficial. Think about this for five minutes. Write down a list of your best ideas.

Here is a list of areas I came up with:

- Process Modeling and Control[11].
- Health Diagnostics[12].
- Investment Portfolio Management[13].
- Military Target Recognition[14].
- Analysis of MRI and X-rays[15].
- Credit Rating of individuals by banks and other financial institutions[16].
- Marketing campaigns[17].
- Voice Recognition[18].

- Forecasting the stock market[19].
- Text Searching[20].
- Financial Fraud Detection[21].
- Optical Character Recognition[22].

Richard Socher made his list, found a useful application, and co-founded MetaMind[23], an innovative company which specializes in medical image analysis and automated image recognition. Other data scientists, entrepreneurs, applied researchers, and maybe even you, will soon follow Richard into this increasingly lucrative space.

A Primer on Neural Networks

Neural networks have been an integral part of the data scientist's toolkit for over a decade[24]. Their introduction considerably improved the accuracy of predictors, and the gradual refinement of neural network training methods continues to benefit both commerce and scientific research. 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[25]. 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 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[26] it was soon discovered that they were pretty good at classification and prediction tasks[27].

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.

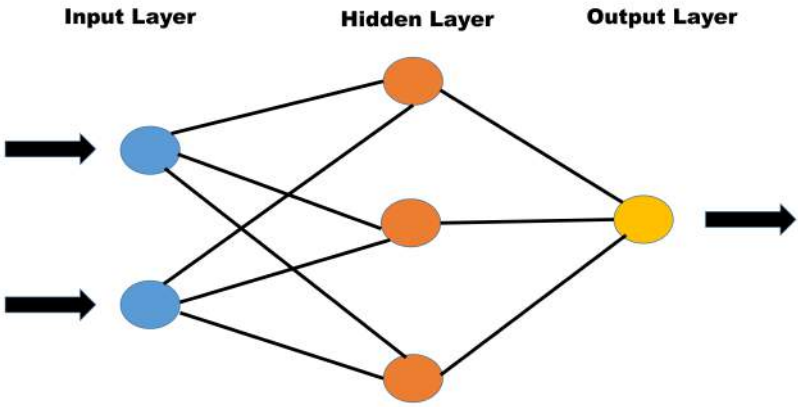


Figure 1.4: A basic neural network

A Neural network is constructed from a number of interconnected nodes known as neurons. These are usually arranged into a number of layers. A typical feedforward neural network will have at a minimum an input layer, a hidden layer and an output layer. The input layer nodes correspond to the number of features or attributes you wish to feed into the neural network. These are akin to the co-variate s you would use in a linear regression model. The number of output nodes corre-

spond to the number of items you wish to predict or classify. The hidden layer nodes are generally used to perform non-linear transformation on the original input attributes.

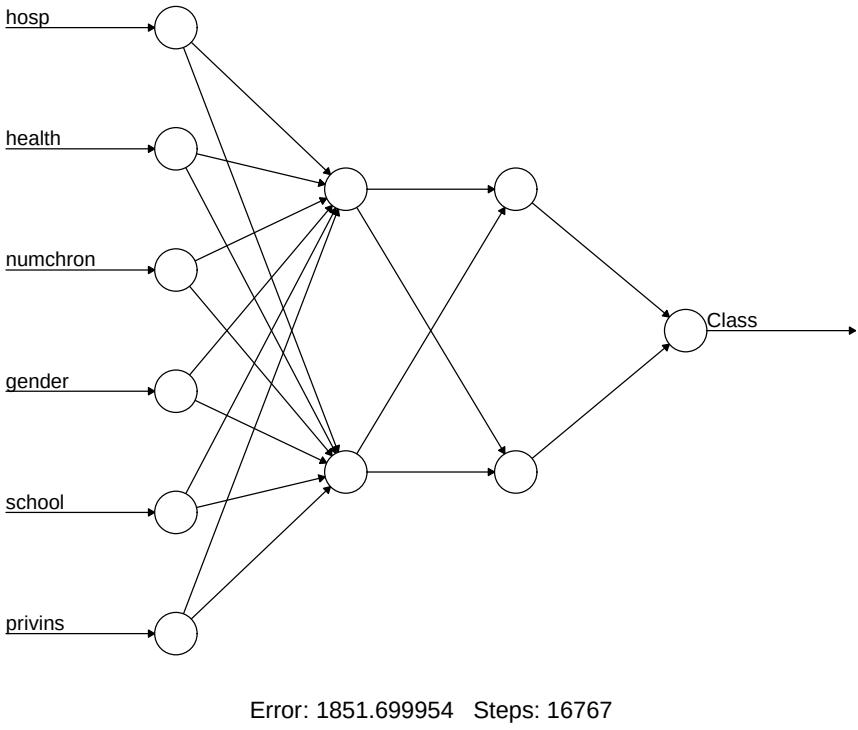


Figure 1.5: Multi-Layer Perceptron

In their simplest form, feed-forward neural networks propagate attribute information through the network to make a prediction, whose output is either continuous for regression or discrete for classification. Figure 1.4 illustrates a typical feed forward neural network topology. It has 2 input nodes, 1 hidden layer with 3 nodes, and 1 output node. The information is fed forward from the input attributes to the hidden layers and

then to the output nodes which provide the classification or regression prediction. It is called a feed forward neural network because the information flows forward through the network.

Figure 1.5 shows the topology of a typical multi-layer perceptron neural network as represented in R. This particular model 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 1.6 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 flows 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.

At the heart of an artificial neural network is a mathematical node, unit or neuron. It is the basic processing element. The input layer neurons receive incoming information which they process via a mathematical function and then distribute to the hidden layer neurons. This information is processed by the hidden layer neurons and passed onto the output layer neurons. 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.

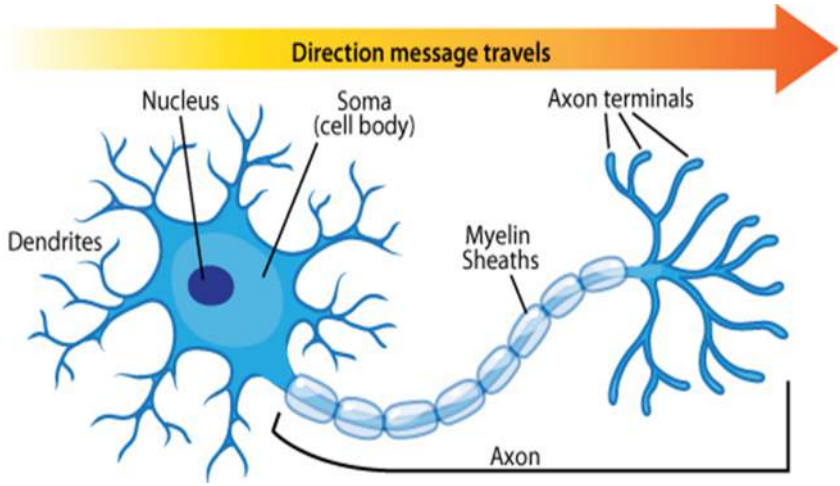


Figure 1.6: Biological Neuron. © Arizona Board of Regents / ASU Ask A Biologist. https://askabiologist.asu.edu/neuron-anatomy. See also http://creativecommons.org/licenses/by-sa/3.0/

NOTE... ✍

The original "Perceptron" model was developed at the Cornell Aeronautical Laboratory back in 1958[28]. 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.

The result of this processing is then weighted and dis-

tributed to the neurons in the next layer. In essence, neurons activate each other via weighted sums. This ensures the strength of the connection between two neurons is sized according to the weight of the processed information.

Each neuron contains an 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.

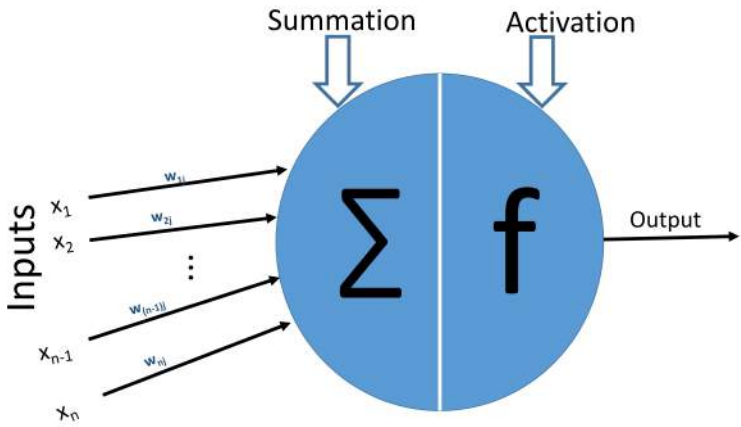


Figure 1.7: Alternative representation of neuron

Figure 1.7 illustrate the workings of an individual neuron. Given a sample of input attributes $\{x_1,...,x_n\}$ a weight w_{ij} is associated with each connection into the neuron; and the neuron then sums all inputs according to:

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

The parameter b_j is known as the bias and is similar to the intercept in a linear regression model. It allows the network to shift the activation function "upwards" or "downwards". This type of flexibility is important for successful machine learning[29].

Activation Functions

Activation functions for the hidden layer nodes are needed to introduce non linearity into the network. The activation function 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 1.8 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:

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

There are a wide number of activation functions. Four of the most popular include:

- **Linear function:** In this case:

  $$f(u) = u$$

- **Hyperbolic Tangent function:** The hyperbolic tangent function produces output in the range -1 to $+1$. This function shares many properties of the sigmoid function; however, because the output space is broader, it is

sometimes more efficient for modeling complex nonlinear relations. The function takes the form:

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

- **Softmax function**: It is common practice to use a softmax function for the output of a neural network. Doing this gives us a probability distribution over the k classes:

$$f(u) = \frac{\exp\left(\frac{u}{T}\right)}{\sum_k \exp\left(\frac{u}{T}\right)}$$

where T is the temperature (normally set to 1). Note that using a higher value for T produces a 'softer' probability distribution over the k classes. It is typically used with a response variable that has k alternative unordered classes. It can essentially be viewed as a set of binary nodes whose states are mutually constrained so that exactly one of the k states has value 1 with the remainder taking the value 0.

- **Rectified linear unit (ReLU):** Takes the form:

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

This activation function has proved popular in deep learning models because significant improvements of classification rates have been reported for speech recognition and computer vision tasks[30]. It only permits activation if a neurons output is positive; and allows the network to compute much faster than a network with sigmoid or hyperbolic tangent activation functions because it is simply a max operation. It allows sparsity of the neural network because when initialized randomly approximately half of the neurons in the entire network will be set to zero.
There is also a smooth approximation which is sometimes used because it is differentiable:

$$f(u) = \log(1 + \exp(u))$$

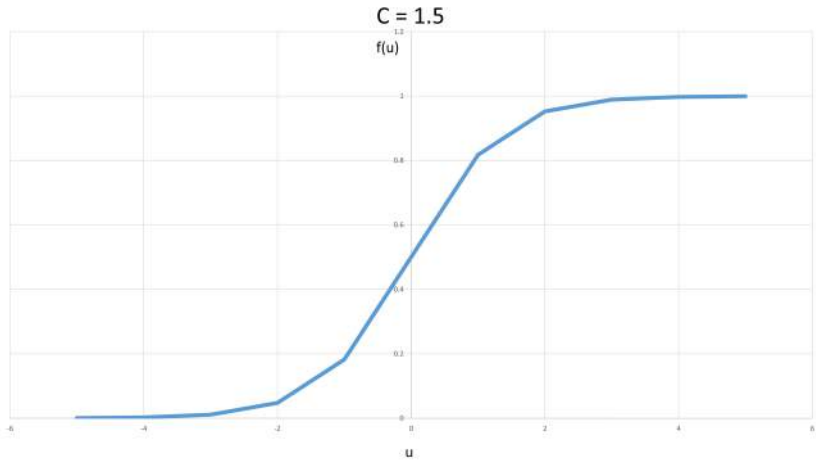


Figure 1.8: The sigmoid function with c = 1.5

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 falls below a pre-specified threshold.

The backpropagation algorithm was the first popular 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 layer is used to re-modify the weights. The algorithm propagates the error backwards through the network and computes the gradient of the change in error with respect to changes in the weight values.

- **Step 5:- Adjust**: Make adjustments to the weights using the gradients of change with the goal of reducing the error. The weights and biases of each neuron are adjusted by a factor based on the derivative of the activation function, the differences between the network output and the actual target outcome and the neuron outputs. Through this process the network "learns".

The basic idea is roughly illustrated in Figure 1.9. 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)[31]. Each cycle through this learning process is called an epoch.

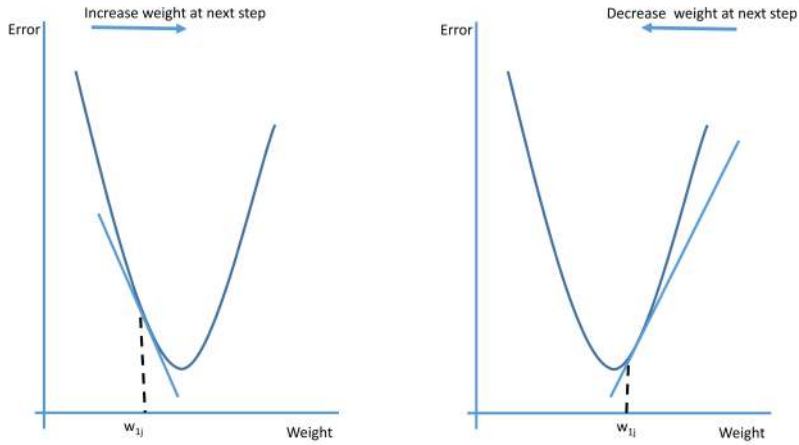


Figure 1.9: Basic idea of the backpropagation algorithm

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 input attributes.

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!

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 nonlin-

ear which makes the optimization process complex. To avoid the network getting trapped in a local minima a momentum parameter is often specified.

Deep learning 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.

Here are four things to try right now:

1. Do an internet search using terms similar to "deep learning jobs", "machine learning jobs" , "machine learning jobs salary" and "deep learning jobs salary". What do find?

2. Identify four areas where deep learning could be of personal benefit to you and/ or your organization. Now pick the area you are most interested in. Keep this topic in mind as you work through the remainder of this book. If possible, go out and collect some data on this area now. By the way, jot down your thoughts in your innovation journal. Refer back to these as you work through this text. If you don't have an innovation/ ideas journal - go out and get one! It will be your personal gold mine for new and innovative solutions[32].

3. If you are new to R, or have not used it in a while, refresh your memory by reading the amazing FREE tutorials at http://cran.r-project.org/other-docs.html. You will be "up to speed" in record time!

4. R- user groups are popping up everywhere. Look for one in you local town or city. Join it! Here are a few resources to get you started:

- For my fellow Londoners check out: http://www.londonr.org/.
- A global directory is listed at: http://blog.revolutionanalytics.com/local-r-groups.html.
- Another global directory is available at: http://r-users-group.meetup.com/.
- Keep in touch and up to date with useful information in my FREE newsletter. Sign up at www.AusCov.Com.

Notes

[1]See Spencer, Matt, Jesse Eickholt, and Jianlin Cheng. "A Deep Learning Network Approach to ab initio Protein Secondary Structure Prediction." Computational Biology and Bioinformatics, IEEE/ACM Transactions on 12.1 (2015): 103-112.

[2]If you want to learn about these, other data science techniques and how to use them in R, pick up a copy of **92 Applied Predictive Modeling Techniques in R**, available at **www.AusCov.com**.

[3]For more information on building neural networks using R, pick up a copy of the book **Build Your Own Neural Network TODAY!** from **www.AusCov.com**.

[4]It also appears deep architectures can be more efficient (in terms of number of fitting parameters) than a shallow network. See for example Y. Bengio, Y. LeCun, et al. Scaling learning algorithms towards ai. Large-scale kernel machines, 34(5), 2007.

[5]See:

- Hinton G. E., Osindero S., Teh Y. (2006). "A fast learning algorithm for deep belief nets", Neural Computation 18: 1527–1554.
- BengioY (2009) Learning deep architectures for AI, Foundations and Trends in Machine Learning 2:1–127.

[6]See for example http://www.ersatzlabs.com/

[7]The IEEE is the world's largest professional organization devoted to engineering and the applied sciences. It has over 400,000 members globally.

[8]See the report by Emily Waltz. Is Data Scientist the Sexiest Job of Our Time? IEEE Spectrum. September 2012. Also available at http://spectrum.ieee.org/tech-talk/computing/it/is-data-scientist-the-sexiest-job-of-our-time

[9]See the special report by James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. McKinsey Global Institute. May 2011. Also available at http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation.

[10]See Davenport, Thomas H., and D. J. Patil. "Data Scientist: The Sexiest Job of the 21st Century-A new breed of professional holds the key to capitalizing on big data opportunities. But these specialists aren't easy to find—And the competition for them is fierce." Harvard Business Review (2012): 70.

[11]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.

[12]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.

[13]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.

[14]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.

[15]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.

[16]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.

[17]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.

[18]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.

[19]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.

[20]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.

[21]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).

[22]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. & Eng 41 (2011): 5587-5591.

[23]https://www.metamind.io/yours

[24]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.

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

[26]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.

[27]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.

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

[29]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. Imagine a linear regression model where the intercept is fixed at zero and you will get the picture.

[30]See for example,

- Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. Regularization of neural networks using dropconnect. In Proceedings of the 30th International Conference on Machine Learning (ICML-13) (2013), pp. 1058–1066.
- Yajie Miao, F. Metze and S. Rawat: Deep maxout networks for low-resource speech recognition. IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), 398-403. 2013.

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

[32]I like Austin Kleon's "Steal Like An Artist Journal." It is an amazing notebook for creative kleptomaniacs. See his website http://austinkleon.com.

Chapter 2

Deep Neural Networks

Data Science is a series of failures punctuated by the occasional success.

N.D Lewis

REMARKABLE achievements have been made in the practical application of deep learning techniques. After decades of research, many failed applications and the abandonment of the discipline by all but a few hardcore researchers[33]. What was once thought impossible is now feasible and deep learning networks have gained immense traction because they have been shown to outperform all other state-of-the-art machine learning tools for a wide variety of applications such as object recognition[34], scene understanding[35] and occlusion detection[36].

Deep learning models are being rapidly developed and applied in a wide range of commercial areas due to their superior predictive properties including robustness to overfitting[37]. They are successfully used in a growing variety of applications ranging from natural language processing to document recognition and traffic sign classification [38].

In this chapter we discuss several practical applications of the Deep Neural Network (DNN) ranging from enhancing visibility in foggy weather, malware detection and image compres-

sion. We build a model to investigate the universal approximation theorem, develop a regression style DNN to predict median house prices, create a classification DNN for modeling diabetes and illustrate how to use multiple outputs whilst predicting waist and hip circumference. During the process you will learn to use a wide variety of R packages, master a few data science tricks and pick up a collection of useful tips to enhance DNN performance.

The Amazingly Simple Anatomy of Deep Neural Networks

As illustrated in Figure 2.1, a DNN consists of an input layer, an output layer, and a number of hidden layers sandwiched in between the two. It is akin to a multi-layer perceptron (MLP) but with many hidden layers and multiple connected neurons per layer. The multiple hidden layers of the DNN are advantageous because they can approximate extremely complex decision functions.

The hidden layers can be considered as increasingly complex feature transformations. They are connected to the input layers and they combine and weight the input values to produce a new real valued number, which is then passed to the output layer. The output layer makes a classification or prediction decision using the most abstract features computed in the hidden layers.

During DNN learning the weights of the connections between the layers are updated during the training phase in order to make the output value as close as possible to the target output. The final DNN solution finds the optimal combination of weights so that the network function approximates a given decision function as closely as possible. The network learns the decision function through implicit examples.

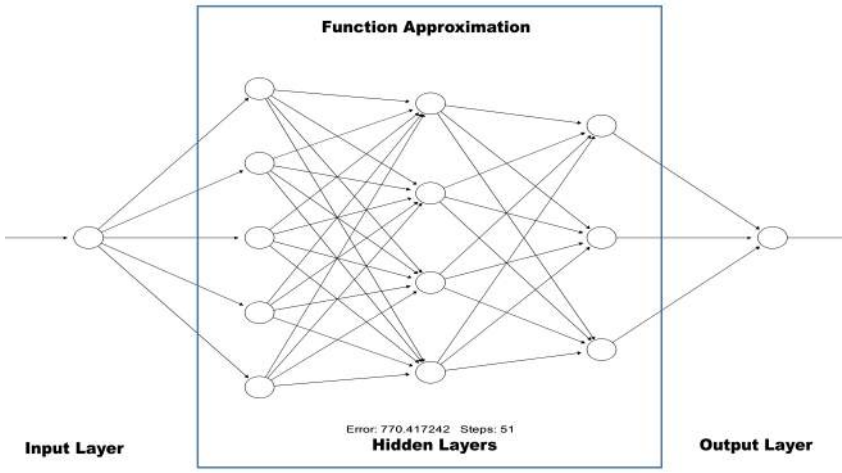


Figure 2.1: A DNN model

How to Explain a DNN in 60 Seconds or Less

As data scientists, we often have to explain the techniques we use to other data scientists who may be new to the method. Being able to do this is a great skill to acquire. Here is what to do when you absolutely, positively must explain your DNN in 60 Seconds or less. Take a deep breath and point to Figure 2.1; the hidden layers are the secret sauce of the DNN. The non-linear data transformations these neurons perform are at the heart of the power of these techniques.

We can view a DNN as a combinations of individual regression models. These models (aka neurons) are chained together to provide more flexible combinations of outputs given a set of inputs than would be possible with a single model. It is this flexibility that allows them to fit any function. The outputs from individual neurons when combined together form the likelihood of the probable and improbable. The end result being the probability of reasonable suspicion that a certain outcome

belongs to a particular class.

Under certain conditions we can interpret each hidden layer as a simple log-linear model[39]. The log-linear model is one of the specialized cases of generalized linear models for Poisson-distributed data. It is an extension of the two-way contingency analysis. Recall from your statistics 101 class, this involved measuring the conditional relationship between two or more discrete, categorical variables by taking the natural logarithm of cell frequencies within a contingency table. If you took statistics 101 and don't recall this, don't worry; I also found it hard to stay awake in statistics 101, and I taught the class!

The key take away is that since there are many layers in a typical DNN, the hidden layers can be viewed as a stack of log-linear models which collectively approximate the posterior probability of the response class given the input attributes. The hidden layers model the posterior probabilities of conditionally independent hidden binary neurons given the input attributes. The output layer models the class posterior probabilities.

Three Brilliant Ways to Use Deep Neural Networks

One of the first hierarchical neural systems was the Neocognitron developed in the late 1970's by the NHK Broadcasting Science Research Laboratories in Tokyo, Japan[40]. It was a neural network for visual pattern recognition. Although it represented a considerable breakthrough, the technology remained in the realm of academic research for several decades.

To whet our appetite with more modern applications, let's look at three practical uses of DNN technology. What you will recognize from studying these applications is the immense potential of DNNs in a wide variety of fields. As you read through this section, reflect back to your answer to exercise 2 on page 22. Ask yourself this question, *how can I use DNNs to solve the problems I identified?* Read through this section

quickly to stimulate your thinking. Re-read it again at regular intervals. The tools you need to get started will be discussed shortly.

Enhancing Visibility in Foggy Weather

I think we can all agree that visual activities such as object detection, recognition, and navigation are much more difficult in foggy conditions. It turns out that faded scene visibility and lower contrast while driving in foggy conditions occurs because of the absorption or scattering of light by atmospheric particles such as fog, haze, and mist seriously degrades visibility. Since the reduction of visibility can dramatically degrade an operator's judgment in a vehicle and induce erroneous sensing in remote surveillance systems, visibility prediction and enhancement methods are of considerable practical value.

In general, de-fogging algorithms require a fogless image of the same scene[41], or salient objects in a foggy image such as lane markings or traffic signs in order to supply distance cues[42]. Hanyang University Professor Jechang Jeong and computer engineering student Farhan Hussain developed a deep neural network approach to de-fogging images in real time[43] which works for unseen images. The approach they adopt is quite interesting. They generate an approximate model of the fog composition in a scene using a deep neural network. Details of their algorithm are given in Figure 2.2.

Provided the model is not over trained, it turns out that it can generalize well for de-fogging unseen images. Take a look at Figure 2.3, the top image (a) shows the original scene without fog; the middle image (b) the foggy scene; and the bottom image (c) the de-fogged image using the DNN. Hussain and Jeong conclude by stating that "*The proposed method is robust as it achieves good result for a large set of unseen foggy images.*"

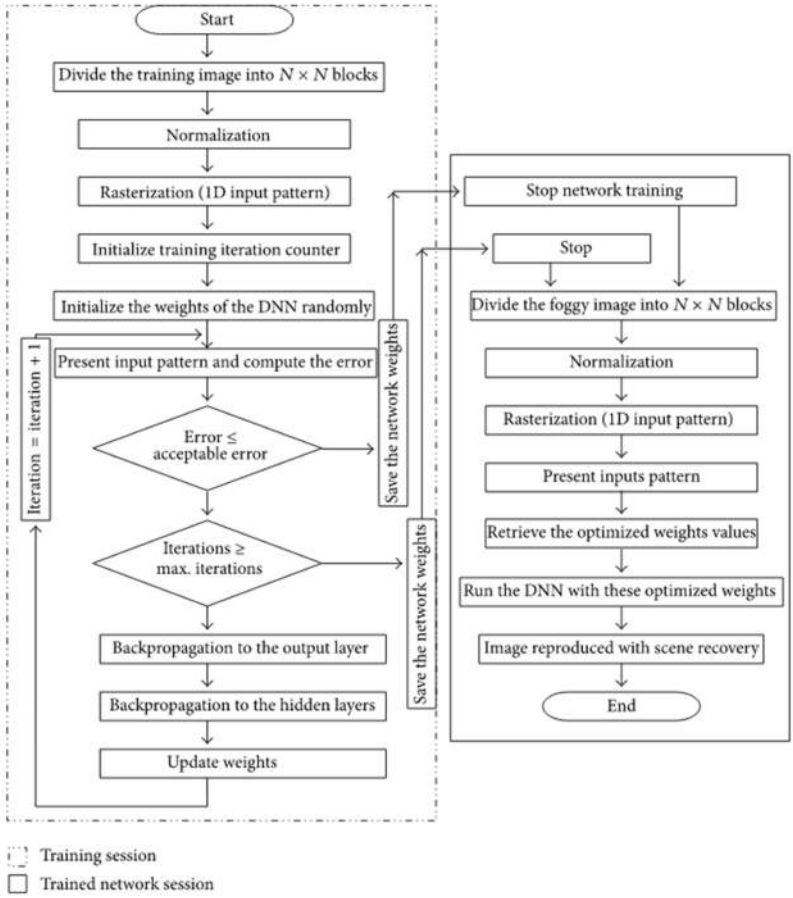


Figure 2.2: Jeong and Hussain's de-fogging DNN algorithm. Image source Hussain and Jeong cited in endnote item 43..

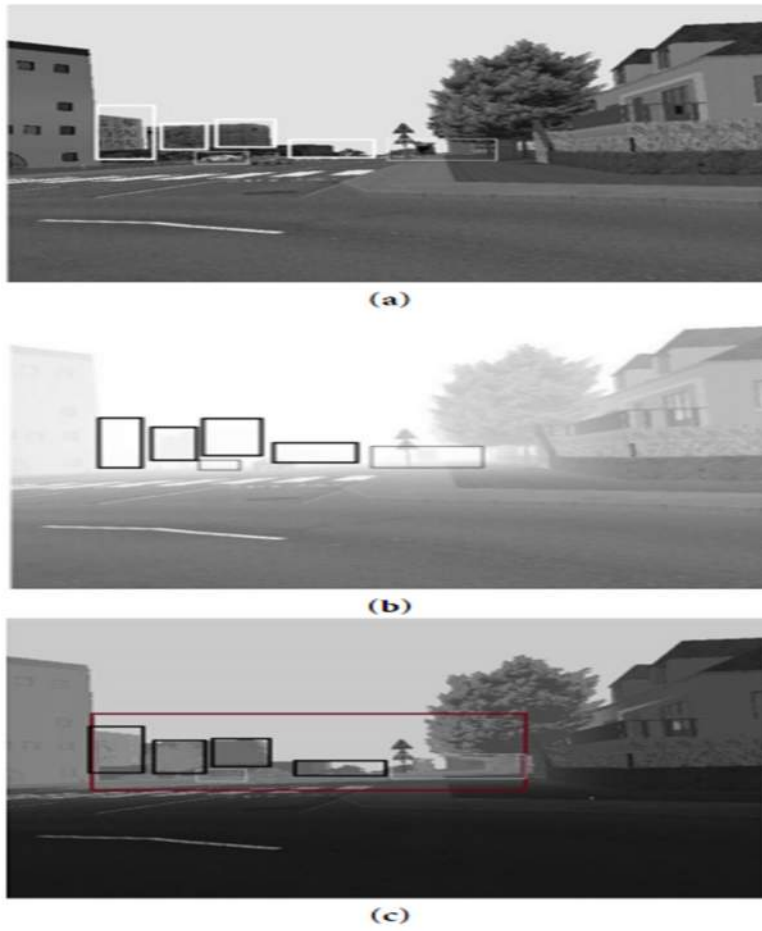


Figure 2.3: Sample image from Jeong and Hussain's de-fogging DNN algorithm. Image source Hussain and Jeong cited in endnote item 43.

A Kick in the Goolies for Criminal Hackers

Malware such as Trojans, worms, spyware, and botnets are malicious software which can shut down your computer. Criminal networks have exploited these software devices for illegitimate gain for as long as there have been uses of the internet.

A friend, working in the charitable sector, recently found his computer shut down completely by such malevolent software. Not only was the individual unable to access critical software files or computer programs, the criminals who perpetrated the attack demanded a significant "ransom" to unlock the computer system. You may have experienced something similar.

Fortunately, my friend had a backup, and the ransom went unpaid. However, the malware cost time, generated worry and several days of anxious file recovery. Such attacks impact not only individuals going about their lawful business, but corporations, and even national governments. Although many approaches have been taken to curb the rise of this malignant activity it continues to be rampant.

Joshua Saxe and Konstantin Berlin of Invincea Labs, LLC[44] use deep neural networks to help identify malicious software[45]. The DNN architecture consists of an input layer, two hidden layers and an output layer. The input layer has 1024 input features, the hidden layers each have 1024 neurons. Joshua and Konstantin test their model using more than 400,000 software binaries sourced directly from their customers and internal malware repositories.

Now here is the good news. Their DNN achieved a 95% detection rate at 0.1% false positive rate. Wow! But here is the truly amazing thing, they achieved these results by direct learning on all binaries files, without any filtering, unpacking, or manually separating binary files into categories. This is astounding!

What explains their superb results? The researchers observe that *"Neural networks also have several properties that make them good candidates for malware detection. First, they can allow incremental learning, thus, not only can they be training in batches, but they can retrained efficiently (even on an hourly or daily basis), as new training data is collected. Second, they allow us to combine labeled and unlabeled data, through pretraining of individual layers. Third, the classifiers*

are very compact, so prediction can be done very quickly using low amounts of memory."

Here is the part you should memorize, internalize and show to your boss when she asks what all this deep learning stuff is about. It is the only thing that matters in business, research, and for that matter life - results. Joshua and Konstantin conclude by saying "*Our results demonstrate that it is now feasible to quickly train and deploy a low resource, highly accurate machine learning classification model, with false positive rates that approach traditional labor intensive signature based methods, while also detecting previously unseen malware.*" And this is why deep learning matters!

The Incredible Shrinking Image

Way back in my childhood, I watched a movie called "*The Incredible Shrinking Man*." It was about a young man, by the name of Scott, who is exposed to a massive cloud of toxic radioactive waste. This was rather unfortunate, as Scott was enjoying a well deserved relaxing day fishing on his boat.

Rather than killing him outright in a matter of days, as such exposure might do to me or you; the screen writer uses a little artistic license. The exposure messes with poor Scott's biological system in a rather unusual way. You see, within a matter of days Scott begins to shrink. When I say shrink, I don't mean just his arm, leg or head, as grotesque at that would be; Scott's whole body shrinks in proportion, and he becomes known as the incredible shrinking man.

Eventually he becomes so small his wife puts him to live in a dolls house. I can tell you, Scott was not a happy camper at this stage of the movie, and neither would you be. I can't quite recall how the movie ends, the last scene I remember involves a tiny, minuscule, crumb sized Scott, wielding a steel sowing pin as a sword in a desperate attempt to fend off a vicious looking spider!

Whilst scientists have not quite solved the problem of how

to miniaturize a live human, Jeong and Hussain have figured out how to use a deep neural network to compress images[46]. A visual representation of their DNN is shown in Figure 2.4.

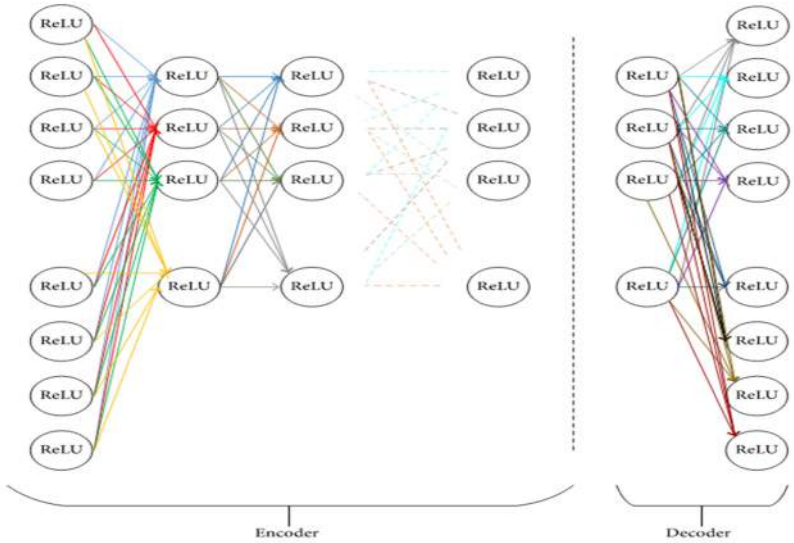


Figure 2.4: Jeong and Hussain's DNN for image compression. Image source Hussain and Jeong cited in endnote item 46.

Notice the model consists of two components - "Encoder"and "Decoder", this is because image compression consists of two phases. In the first phase the image is compressed, and in the second phase it is decompressed to recover the original image. The number of neurons in the input layer and output layer corresponds to the size of image to be compressed. Compression is achieved by specifying a smaller number of neurons in the last hidden layer than contained in the originals input attribute / output set.

The researches applied their idea to several test images and for various compression ratios. Rectified linear units were used for the activation function in the hidden layers because they "*lead to better generalization of the network and reduce the real*

compression-decompression time." Jeong and Hussain also ran the models using sigmoid activation functions.

Figure 2.5 illustrates the result for three distinct images. The original images are shown in the top panel, the compressed and reconstructed image using the rectified linear units and sigmoid function are shown in the middle and lower panel respectively. The researchers observe "*The proposed DNN learns the compression/decompression function very well.*" I would agree. What ideas does this give you?

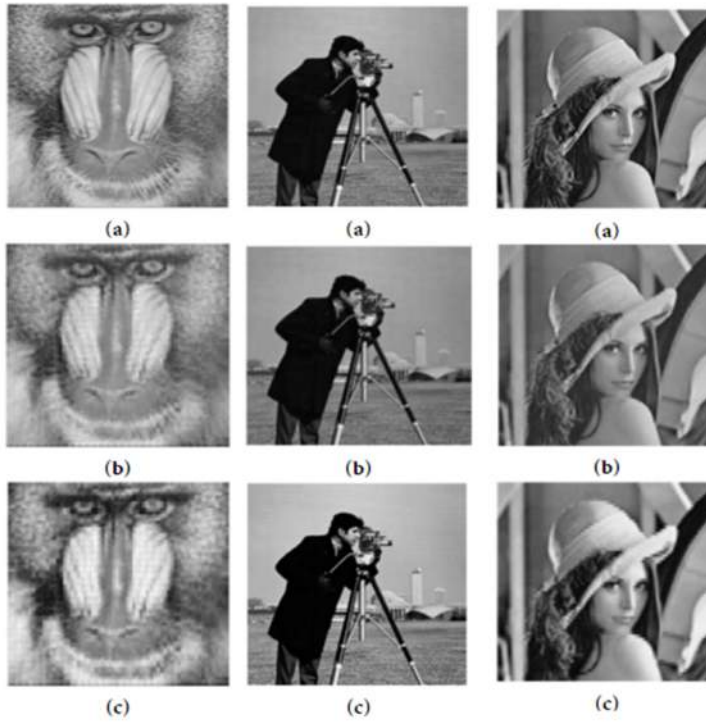


Figure 2.5: Image result of Jeong and Hussain's DNN for image compression. Image source Hussain and Jeong cited in endnote item 46

How to Immediately Approximate Any Function

A while back researchers Hornik et al.[47] 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.

Of course in practice things are a little different. For one thing, real world decision functions may not be continuous; the theorem does not specify the number of hidden neurons required. It seems, for very many real world problems many hidden layers are required for accurate classification and prediction. Nevertheless, the theorem holds some practical value.

Let's build a DNN to approximate a function right now using R. First we need to load the required packages. For this example, we will use the `neuralnet` package:

```
> library("neuralnet")
```

We will build a DNN to approximate $y = x^2$. First we create the attribute variable x, and the response variable y.

```
> set.seed(2016)
> attribute <- as.data.frame(sample(seq
   (-2,2,length=50),
50, replace=FALSE),
```

```
ncol=1)
> response<-attribute^2
```

Let's take a look at the code, line by line. First the `set.seed` method is used to ensure reproducibility. The second line generates 50 observations without replacement over the range -2 to +2. The result is stored in the R object `attribute`. The third line calculates $y = x^2$ with the result stored in the R object `response`.

Next we combine the `attribute` and `response` objects into a dataframe called `data`; this keeps things simple, which is always a good idea when you are coding in R:

```
> data <- cbind(attribute,response)
> colnames(data) <- c("attribute","response
   ")
```

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

```
> head(data,10)
     attribute   response
1   -1.2653061 1.60099958
2   -1.4285714 2.04081633
3    1.2653061 1.60099958
4   -1.5102041 2.28071637
5   -0.2857143 0.08163265
6   -1.5918367 2.53394419
7    0.2040816 0.04164931
8    1.1020408 1.21449396
9   -2.0000000 4.00000000
10  -1.8367347 3.37359434
```

The numbers are as expected with `response` equal to the square of `attribute`. A visual plot of the simulated data is shown in Figure 2.6.

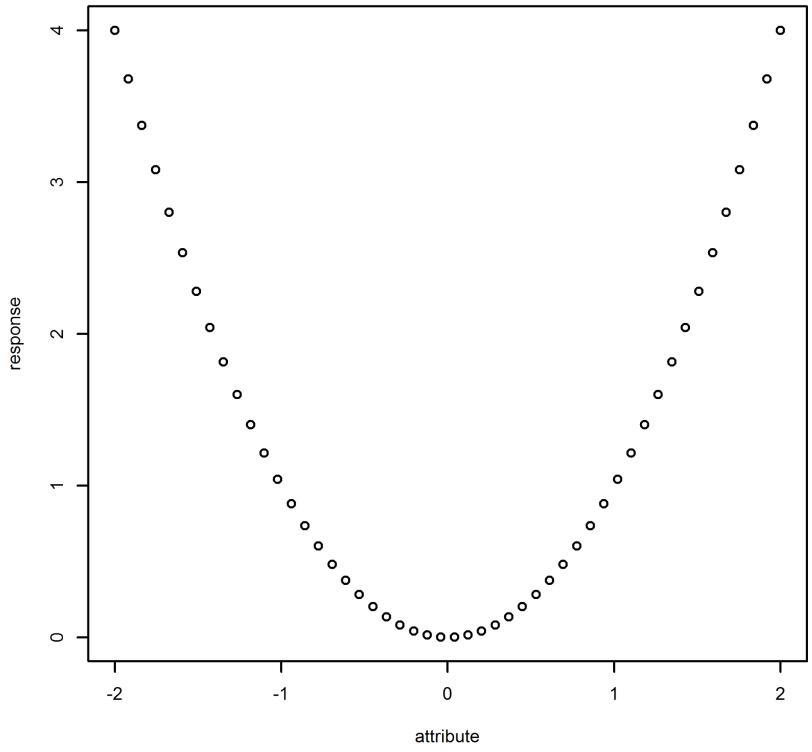


Figure 2.6: Plot of $y = x^2$ simulated data

We will fit a DNN with two hidden layers each containing three neurons. Here is how to do that:

```
>  fit<-neuralnet(response~attribute,
data=data,
hidden=c(3,3),
threshold=0.01)
```

The specification of the model formula `response~attribute` follows standard R practice. Perhaps the only item of interest is `threshold` which sets the error threshold to be used. The actual level you set will depend in large part on the application

for which the model is developed. I would image a model used for controlling the movements of a scalpel during brain surgery would require a much smaller error margin than a model developed to track the activity of individual pedestrians in a busy shopping mall. The fitted model is shown in Figure 2.7.

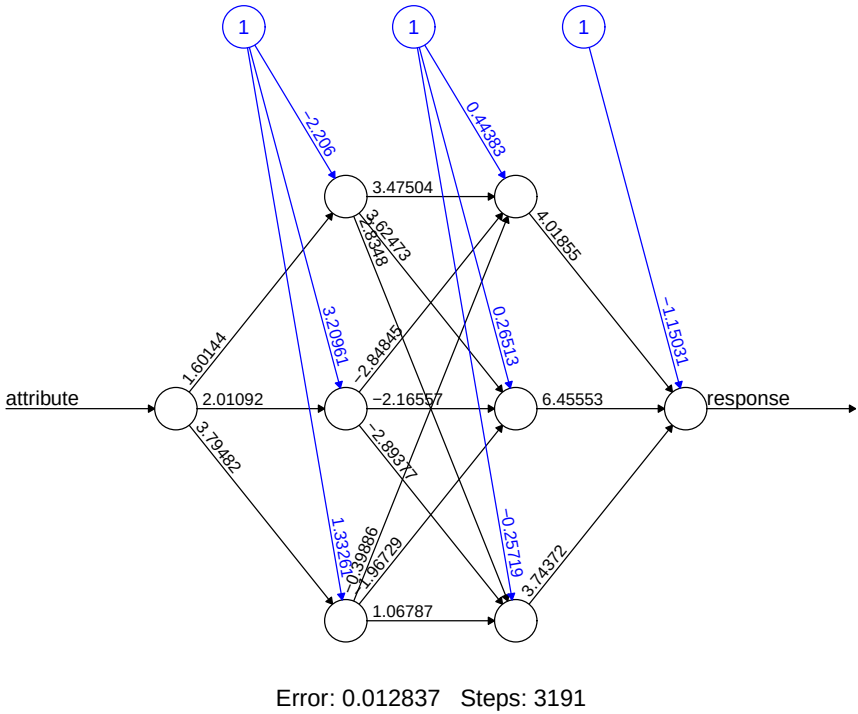


Figure 2.7: DNN model for $y = x^2$

Notice that the image shows the weights of the connections and the intercept or bias weights. Overall, it took 3191 steps for the model to converge with an error of 0.012837.

Let's see how good the model really is at approximating a function using a test sample. We generate 10 observations

from the range -2 to +2 and store the result in the R object `testdata`:

```
> testdata <- as.matrix(sample(seq(-2,2,
   length=10)
, 10, replace=FALSE)
, ncol=1)
```

Prediction in the `neuralnet` package is achieved using the `compute` function:

```
> pred <- compute(fit, testdata)
```

> ***NOTE...*** ✍
>
> To see what attributes are available in any R object simply type `attributes(object_name)`. For example, to see the attributes of `pred` you would type and see the following:
>
> ```
> > attributes(pred)
> $names
> [1] "neurons"    "net.result"
> ```

The predicted values are accessed from `pred` using `$net.result`. Here is one way to see them:

```
>  result <- cbind(testdata,pred$net.result
   ,testdata^2)
> colnames(result) <- c("Attribute","
   Prediction", "Actual")
> round(result,4)
      Attribute Prediction Actual
 [1,]    2.0000     3.9317 4.0000
 [2,]   -2.0000     3.9675 4.0000
 [3,]    0.6667     0.4395 0.4444
 [4,]   -0.6667     0.4554 0.4444
 [5,]    1.5556     2.4521 2.4198
```

```
 [6,]    -1.5556      2.4213 2.4198
 [7,]    -0.2222      0.0364 0.0494
 [8,]     0.2222      0.0785 0.0494
 [9,]    -1.1111      1.2254 1.2346
[10,]     1.1111      1.2013 1.2346
```

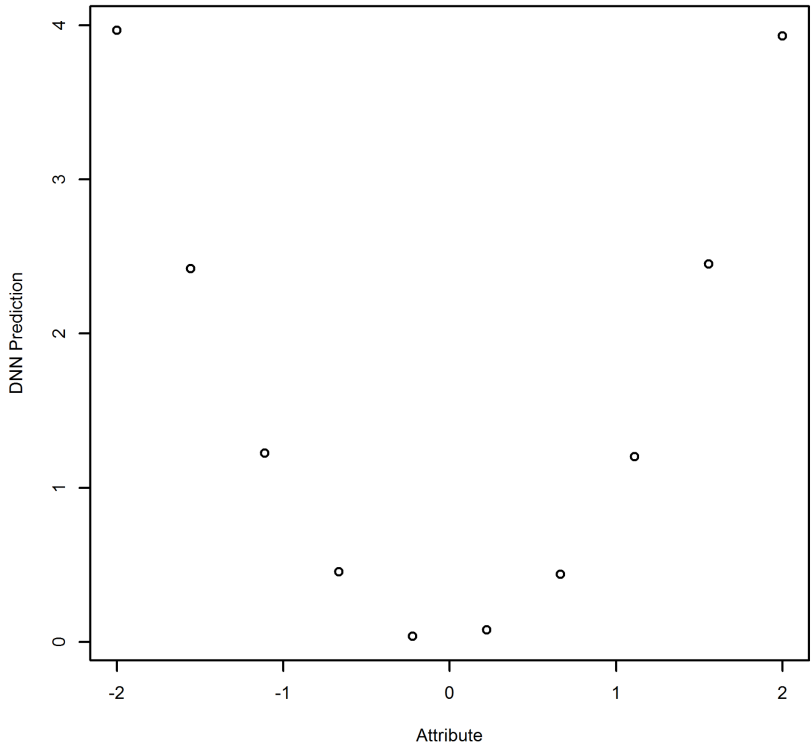


Figure 2.8: DNN predicted and actual number

The first line combines the DNN prediction (`pred$net.result`) with the actual observations (`testdata^2`) into the R object `result`. The second line gives each column a name (to ease readability when we print to the screen).

The third line prints the R object `result` rounding it to four decimal places.

The reported numbers indicate the DNN provides a good, although not exact, approximation of the actual function. The predicted and fitted values are shown in Figure 2.8. Judge for yourself, what do you think of the DNN models accuracy? How might you improve it?

The Answer to How Many Neurons to Include

As soon as you begin to see the potential of DNNs the question of exactly how many neurons to include arises. One way to think about this issue is to notice that the patterns you would like to extract from your data contain variation. These variations often arise from natural sources (i.e. are randomly drawn from a probability distribution) or they may be inherent in the process you are trying to model.

In his 1971 work of fiction Wheels, Arthur Hailey explains why cars assembled on a Monday or Friday suffer from quality problems. Cars produced on Monday were of lower quality due to the residual effect of weekend drunkenness; whilst cars manufactured on Friday suffered from the workers thinking about drunkenness!

In reality, any and every industrial production line will have some variation. Take for example the production of widgets; although widgets are all created using the same process and materials, there will be some variation in weight, size, color and texture. This variation may not matter for the intended use of the widget. However, too much variation between individual widgets will result in quality issues. In fact, the number of top executives who have been fired due to unacceptable variation in the production processes of the widgets they oversee are numerous[48].

The key point is that there will nearly always be some varia-

tion in the patterns you wish to capture from your data. Part of this variation will be systematic and can be used to distinguish between classes; and part of the variation will be noise which cannot be used to distinguish between classes. Since different problems will have a different mix of systematic variation to noise induced variation, the optimal number of hidden neurons is problem specific.

One idea is to use more neurons per layer to detect finer structure in your data. However, the more hidden neurons used the more likely is the risk of over fitting. Over-fitting occurs because with more neurons there is an increased likelihood that the DNN will learn both patterns and noise, rather than the underlying statistical structure of the data. The result is a DNN which performs extremely well in sample but poorly out of sample.

Here is the key point to keep in mind as you build your DNN models. For the best generalization ability, a DNN should have as few neurons as possible to solve the problem at hand with a tolerable error. The larger the number of training patterns, the larger the number of neurons that can be used whilst still preserving the generalization ability of a DNN.

.

The ABCs of Choosing the Optimal Number of Layers

One of the major difficulties you will face in building and using DNNs is the selection of the appropriate number of layers. The problem is challenging because it is possible to train a DNN to a very small error on the training sample only to find that it performs very badly for patterns not used in the training process.

When this happens to you, and it will, do not be perturbed. The professors who teach the data science classes, books which outline the techniques and articles which discuss the latest

trends rarely reveal this unspoken truth.

Pick up any text book and what do you see? Successful model followed by successful model. Man, by the time I had finished my Master's degree in Statistics, I thought this model building stuff was a doddle! The funny thing is, year after year I continue to meet freshly minted Master's degree students in data science and statistics who think as I once did. You see, the take away for the uninitiated is that the model building phase is easy, a breeze, a place where the data scientist can relax, put their feet up and coast.

In actual practice, *data science is a series of failures punctuated by the occasional success*. If you have spent any time deploying commercial models, you will inevitably invest a considerable amount of time, worry and intellectual capital at the model development phase.

Unfortunately, with a DNN it is quite possible to have five, six, hell yes even twenty models all performing reasonably well on the training set! Yet, every single one fail miserably on the test sample; or worse - your model makes it to production but crashes in a huge ball of bright flames, illuminating you as its creator, as it tumbles slowly back to earth. The resultant thick choking smoke surrounding you and your boss can smoother your prospects for advancement, stifle opportunity and even end careers. The selection of the appropriate number of layers is critical.

I think of finding the optimal number of layers as essentially a model selection problem. It can be partially solved using traditional model selection techniques, one of the most frequently used by novices being trial and error, followed by my personal favorite a systematic global search against selection criteria. If you consider each hidden layer as a feature detector, the more layers, the more complex feature detectors can be learned. This leads to a straightforward rule of thumb, use more layers for more complex functions.

Three Ideas to Supercharge DNN Performance

Let's face it, running neural networks over large datasets requires patience. When I coded my very first multi-layer perceptron, back in the early 1990's, I had to wait around three and a half days per run! And by today's standards my dataset was tiny. One of the major problems associated with neural networks in general is the time taken to train networks and the subsequent problem of the model over-fitting.

Imagine, the situation, the results are needed yesterday by your boss because your most important client's campaign is about to launch! Thankfully your large DNN has been trained over many thousands of connections costing several hours overnight to a small error. However, when you assess its predictive power on the test sample it fails miserably! The model has been over-fit by you! What are you supposed to tell your boss or worse yet the client? The fact is, as you continue to develop your applied skills in deep learning, you will need some tricks in your back pocket. Here are three of the best the DNN developer keeps up her sleeve.

Dropout to Enhance Success

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

The funny thing is, we also celebrate those celebrities and business tycoons who dropped out. These individuals went

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

Whilst dropping out is both celebrated and derided by our culture, it appears to also have some upside and downside in relation to DNNs. Borrowing from the idea that dropping out might boost DNN performance, suppose we ignore a random number of neurons during each training round[49]; the process of randomly omitting a fraction of the hidden neurons is called dropout, see Figure 2.9.

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

The idea is very simple and results in a weak learning model at each epoch. Weak models have low predictive power by themselves, however the predictions of many weak models can be weighted and combined to produce models with much 'stronger' predictive power.

In fact, dropout is very similar to the idea behind the machine learning technique of bagging[50]. Bagging is a type of model averaging approach where a majority vote is taken from classifiers trained on bootstraped samples of the training data. In fact, you can view dropout as implicit bagging of several neural network models. Dropout can therefore be regarded as an efficient way to perform model averaging across a large number of different neural networks.

Much of the power of DNNs comes from the fact that each neuron operates as an independent feature detector. However, in actual practice is common for two or more neurons to begin to detect the same feature repeatedly. This is called co-adaptation. It implies the DNN is not utilizing its full capacity

efficiently, in effect wasting computational resources calculating the activation for redundant neurons that are all doing the same thing.

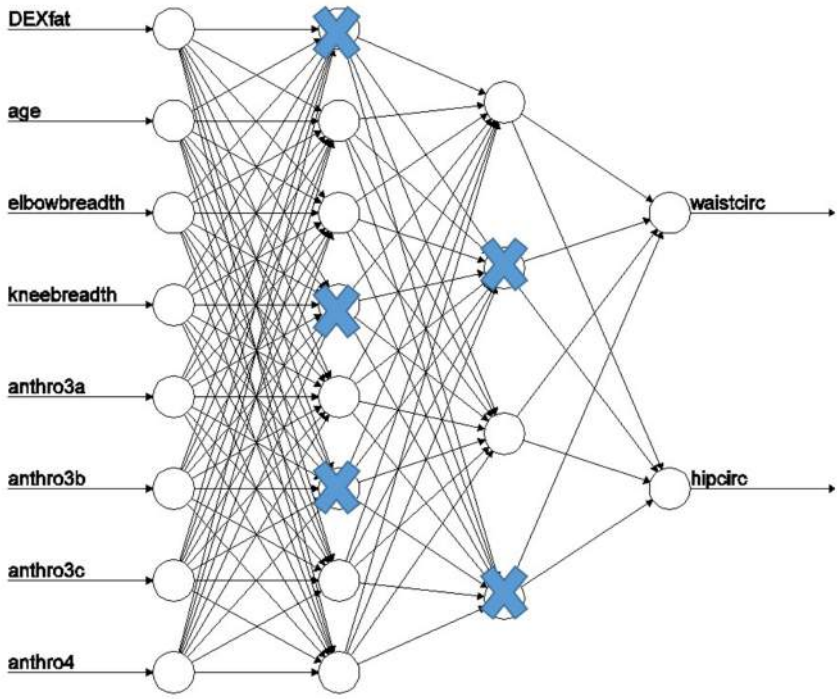


Figure 2.9: DNN Dropout

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

Dropout discourages co-adaptations of hidden neurons by dropping out a fixed fraction of the activation of the neurons

during the feed forward phase of training. Dropout can also be applied to inputs. In this case, the algorithm randomly ignore certain inputs.

One of life's lessons is that dropping out is not necessarily ruinous to future performance, but neither is it a guarantee of future success, the same is true for dropout in DNNs, there is no absolute guarantee that it will enhance performance, but it is often worth a try. Keep the following three points in mind as you develop your own DNN models:

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

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

3. It appears to provide the most benefit on very large DNN models[51].

How to Benefit from Mini Batching

The traditional backpropagation algorithm, known as batch learning backpropagation, calculates the change in neuron weights, known as delta's or gradients, for every neuron in all of the layers of a DNN and for every single epoch. An epoch is one complete forward pass and one backward pass of the error for all training examples. The deltas are essentially calculus derivative adjustments designed to minimize the error between the actual output and the DNN output.

A very large DNN might have millions of weights for which the delta's need to be calculated. Think about this for a moment....Millions of weights require gradient calculations.... As you might imagine, this entire process can take a considerable amount of time. It is even possible, that the time taken for a DNN to converge on a acceptable solution using batch learning

propagation makes its use in-feasible for a particular application.

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

A batch consists of a number of training examples in one forward/backward pass. Note that the larger the batch size, the more memory you will need to run the model. To get a sense of the computational efficiency of min-batching, suppose you had a batch size of 500, with 1000 training examples. It will take only 2 iterations to complete 1 epoch.

A common question is when should I use mini batching? The answer really depends on the size of the DNN you are building. The more neurons in the model, the greater the potential benefit from mini batching.

Another common question is about the optimal mini batch size. Although in the text books you often see mini batch sizes of 50, 100, 200 and so on. These are generally for illustrative purposes. There is a degree of art and science in choosing the optimal size. In my experience no one number is optimal for all circumstances. The best advice is to experiment with different values to get a sense of what works for your sample and specific DNN architecture.

A Simple Plan for Early Stopping

When I was in elementary school we had regular and frequent outdoor playtime's as part of our education. We usually had playtime mid-morning to whip up our appetite prior to lunch, after lunch to aid the digestive system, and on warm sunny days, our teachers would unleash us outside for an afternoon playtime - to burn up any unused energy before we were packed off home. I loved school!

The only thing that would shorten our frequent outdoor

adventures was the rain. If it rained (and it rains a lot in England) the early stopping bell would ring, the fun would be terminated and the lessons begin. The funny thing was, even after a shortened play time my ability to focus and concentrate on my school work was enhanced. I guess my teachers had an intuitive understanding of the link between physical activity, mental focus, mood and performance[52]. To this day I go outside for a brisk walk to jolt my mental faculties into action.

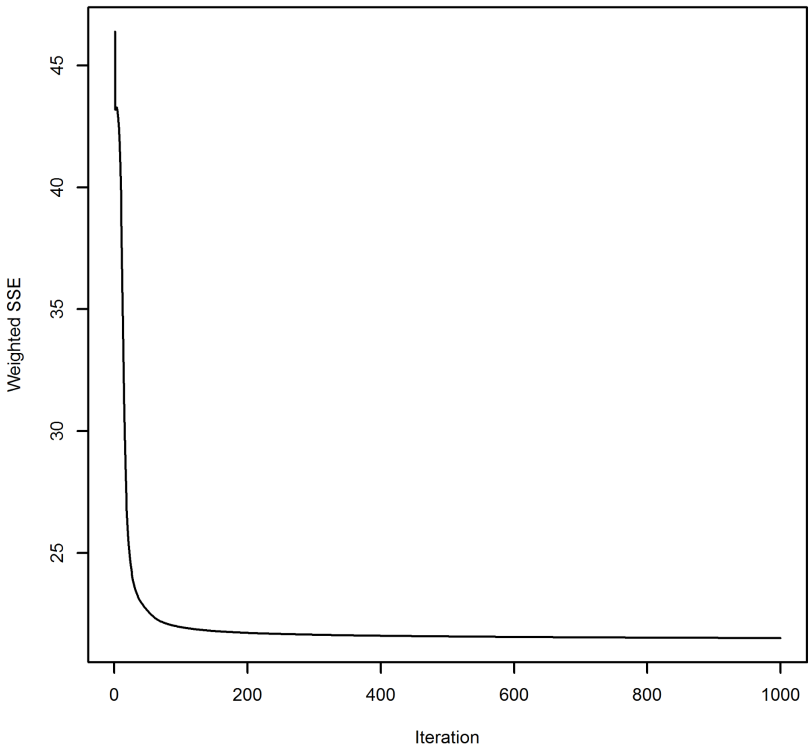


Figure 2.10: Network error by iteration

The analogy with school playtime's is that we should be willing to stop DNN training early if that makes it easier to

extract acceptable performance on the testing sample. This is the idea behind early stopping where the sample is divided into three sets. A training set, a validation set and a testing set. The test set is used to train the DNN. The training error is usually a monotonic function, that decreases further in every iteration. Figure 2.10 illustrates this situation. The error falls rapidly during the first 100 iterations. It then declines at a much shallower rate over the next three hundred iterations, before leveling off to a constant value.

The validation set is used to monitor the performance of the model. The validation error usually falls sharply during the early stages as the network rapidly learns the functional form, but then increases, indicating the model is starting to over fit. In early stopping, training is stopped at the lowest error achieved on the validation set. The validation model is then used on the test sample. This procedure has been proven to be highly effective in reducing over-fitting for a wide variety of neural network applications[53]. It is worth a try by you.

Incredibly Simple Ways to Build DNNs with R

Now we turn our attention to building a DNN model in R using real world data. For this example, we will build a regression DNN to investigate housing values in suburbs of Boston.

A Proven Way to Build Regression DNNs

We begin by loading the packages we will use:

```
> library("neuralnet")
> require(Metrics)
```

We use the functionality of the `neuralnet` package to build the DNN, and the `Metrics` package to calculate various model

fit criteria. The dataset used was collected originally by urban planners Harrison and Rubinfeld to generate quantitative estimates of the willingness of residents to pay for air quality improvements in Boston[54]. It is contained in the `MASS` package:

```
> data("Boston",package="MASS")
> data<-Boston
```

The R object `Boston` contains 506 rows and 14 columns. Each column corresponds to a specific variable. Details of each of the variable used in our analysis are given in Table 1.

Name	Description
`crim`	per capita crime rate by town.
`indus`	proportion of non-retail business acres per town.
`nox`	nitrogen oxides concentration (parts per 10 million).
`rm`	average number of rooms per dwelling.
`age`	proportion of owner-occupied units built prior to 1940.
`dis`	average distances to five Boston employment centres.
`tax`	full-value property-tax rate.
`ptratio`	pupil-teacher ratio by town.
`lstat`	lower status of the population (percent).
`medv`	median value of owner-occupied homes.

Table 1: Variables used from `Boston` data frame.

Here is how to retain only those variable we want to use:

```
> keeps <- c("crim", "indus", "nox", "rm" ,
   "age", "dis", "tax" ,"ptratio", "lstat"
   ,"medv" )
> data<-data[keeps]
```

The R object `data` now only contains the variables we are going to use. The response variable is `mdev`. Next, we perform a quick check to see if the retained data contains any missing values using the `apply` method:

```
> apply(data,2,function(x) sum(is.na(x)))
   crim   indus     nox      rm     age     dis     tax ptratio   lstat    medv
      0       0       0       0       0       0       0       0       0       0
```

Great! There are no missing values.

The relationship between `medv` and the explanatory variables is shown in Figure 2.11.

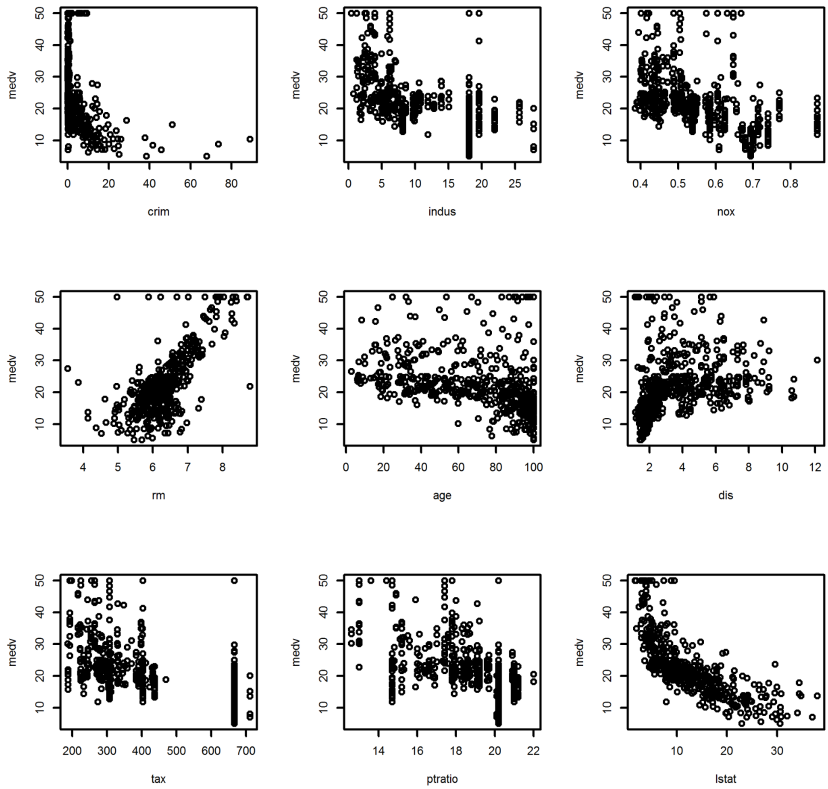


Figure 2.11: Relationship between `medv` and other attributes

The next step is to specify the model. I typically use a formula, as I find it an efficient way to write and maintain R code. In this case the formula is stored in an R object called

f. The response variable medv is to be "regressed" against the remaining nine attributes. Here is how to specify it:

```
> f<-medv~ crim + indus + nox + rm + age +
   dis + tax + ptratio + lstat
```

To set up our test sample we use the sample 400 of the 506 rows of data without replacement using the sample method:

```
> set.seed(2016)
> n=nrow(data)
> train <- sample(1:n, 400, FALSE)
```

Note the R object train contains the row location of the data to be used in the test sample. It does not contain the actual observations.

The DNN can be fitted using the neuralnet function. This can be achieved as follows:

```
> fit<- neuralnet(f,
data = data[train,],
hidden=c(10,12,20),
algorithm = "rprop+",
err.fct = "sse",
act.fct = "logistic",
threshold=0.1,
linear.output=TRUE)
```

Let's walk through this statement argument by argument. The first argument (f) contains the model formula. The second argument specifies the dataset to use (in this case data). The third argument is used to indicate how many hidden layers and nodes to include in the model; in this case we specify three hidden layers. The first hidden layer has 10 neurons, the second hidden layer twelve neurons and the third hidden layer contains 20 neurons.

The next argument specifies the learning algorithm to be use. Rather than use a traditional backpropagation algorithm I choose a more robust version called resilient backpropagation

with backtracking[55]. 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`).

The parameter `err.fct` is the error function; for regression models the sum of squared errors is chosen automatically (`err.fct = "sse"`) . A threshold is set of 0.1 with a logistic activation function. Finally, for the output neuron a linear activation function is selected using `linear.output=TRUE.`

Prediction using our DNN is straightforward as the `neuralnet` package has a handy function called `compute`. The only thing we need to do is pass it the DNN model contained in `fit` along with the test data. Here is how to do that:

```
pred <- compute(fit, data[-train,1:9])
```

Notice I pass the row numbers of the test sample by using `-train` (recall `train` identified the rows to use in the training sample). Figure 2.12 shows the fitted and predicted values alongside the linear regression (solid line). The model appears to perform reasonably well, although there are a number of outlying observations which are not well predicted. This is confirmed by the performance metrics. I calculate the squared correlation coefficient, mean squared error (`mse`) and root mean square error (`rmse`) for illustration:

```
round(cor(pred$net.result,data[-train,10])
    ^2,6)
          [,1]
[1,] 0.809458

>   mse(data [-train,10], pred$net.result)
[1] 0.2607601849

>  rmse (data [-train,10], pred$net.result)
[1] 0.5106468299
```

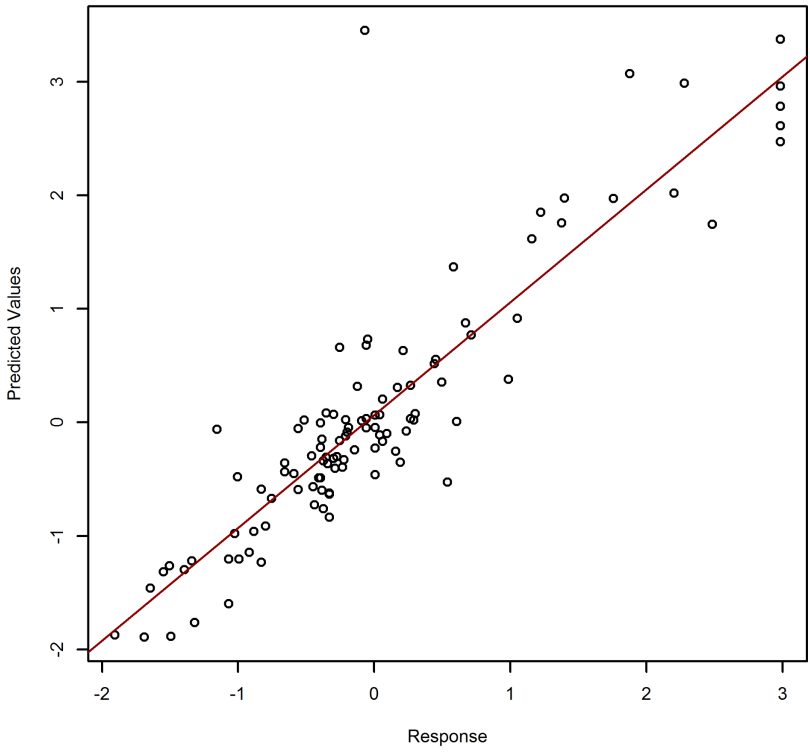


Figure 2.12: Predicted and fitted values

The Smart Person's Tip for DNN Regression

The great thing about R is that there are multiple packages for estimating neural network models. I always like to build models using a wide variety of packages and learning algorithms. Let's build a DNN, using traditional backpropagation with the `deepnet` package. First let's load the required package:

```
> require(deepnet)
```

Next, use the `set.seed` method for reproducibility, and store the attributes in the R object `X` with the response variable

in the R object `Y`:

```
set.seed(2016)
 X= data[train,1:9]
 Y= data[train,10]
```

Here is how to create a DNN:

```
fitB<-nn.train(x=X, y=Y,
 initW = NULL,
 initB = NULL,
 hidden = c(10,12,20),
learningrate = 0.58,
momentum =0.74,
learningrate_scale =1 ,
activationfun = "sigm",
output = "linear",
numepochs = 970,
batchsize = 60,
hidden_dropout = 0,
visible_dropout = 0)
```

The statement is fairly similar to what we have seen before; however, let's walk through it line by line. The neural network is stored in the R object `fitB`. Notice we pass the attributes and the response variable using the statement `x=X, y=Y`. The `deepnet` package gives you the ability to specify the starting values of the neuron weights (`initW`) and biases (`initB`); I set both values to `NULL` so that the algorithm will select their values at random. The DNN has the same topology as that estimated on page 60 i.e. three hidden layers, with 10,12 and 20 neurons in the first, second and third hidden layers respectively.

To use the backpropagation algorithm, you have to specify a learning rate, momentum and learning rate scale[56]. The learning rate controls how quickly or slowly the neural network converges. Briefly, momentum involves adding a weighted average of past gradients in the gradient descent updates. It tends to dampen noise, especially in areas of high curvature of the error function. Momentum can therefore help the network avoid

becoming trapped in local minima. All three parameters are generally set by trial and error, I choose values of 0.58, 0.74 and 1 for the learning rate, momentum and learning rate scale respectively.

The next two lines specify the activation function for the hidden and output neurons. For the hidden neurons I use a logistic function (`"sigm"`); other options include `"linear"` or `"tanh"`. For the output neuron I use a linear activation function, other options include `"sigm"` and `"softmax"`. The model is run over 970 epochs each with a batch size of 60. No neurons are dropped out in the input layer or hidden layer.

Prediction using the test sample is similar to what we have seen before:

```
> Xtest<- data[-train,1:9]
> predB <- nn.predict(fitB, Xtest)
```

With the performance metrics are calculated as:

```
> round(cor(predB,data[-train,10])^2,6)
          [,1]
[1,] 0.930665

>   mse(data [-train,10], predB)
[1] 0.08525447959

>  rmse (data [-train,10], predB)
[1] 0.2919836975
```

Overall, the process of building a model using the `deepnet` package is fairly similar to the process used with the `neuralnet` package. This is one of the great advantages of using R, packages generally work in similar ways, (although of course the parameters to be specified may be different). Such flexibility allows you to build similar DNN models rapidly using a vast variety of different learning algorithms and tuning parameters.

You will find, time and again, that DNNs with the same topology but different learning algorithms or tuning parameters

will perform on the same underlying data very differently. The important point to remember is that the process of selecting the optimal DNN model requires selection of a topology, number of neurons, layers, learning algorithm and tuning parameters. This leads to quite a complex set of combinations! Despite this, as we have seen, powerful DNN models can be rapidly constructed, trained and tested in R. This should lead to a boom in their use in a wide variety of disciplines outside of their traditional domain of image, vocal and signal processing. How will you use DNN regression in your next data science project?

The Art of Building Great DNNs for Classification

I would hazard a guess that the vast majority of published applications of DNNs to date involve solving classification problems. The bulk of these applications involve image and signal processing. In this section, we look at the application of a DNN model to a health related issue. As you work through this case study keep your mind open to alternative applications in your own field of specialized knowledge.

We will build a DNN model 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[57]. It contains 768 observations on 9 variables measured on females at least 21 years old of Pima Indian heritage. Table 2 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)
```

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 2: Response and independent variables in `PimaIndiansDiabetes2` data frame

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

The numbers are in line we what we expected.

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 the classification response variable.

Did you 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 data science life! However, 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. For example, 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)[58].

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 for as Shakespeare wrote *"Though this be madness, yet there is method in't."* We store the response variable in an R object called `y` and remove it from `temp`. Notice 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)
```

Let's push on and 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(2016)
> n=nrow(temp)

> n_train <- 600
> n_test<-n-n_train

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

To build our DNN we use the `RSNNS` package:

```
>  require(RSNNS)
```

Keeping things as simple as possible we assign the response variable to the R object `Y`, and the attributes to `X`:

```
> set.seed(2016)
> X<-temp[train,1:6]
> Y<-temp[train,7]
```

Now to the meat of the issue, here is how to specify our classification DNN in the `RSNNS` package:

```
fitMLP<- mlp(x=X, y=Y,
size = c(12,8),
maxit = 1000,
initFunc = "Randomize_Weights",
initFuncParams = c(-0.3, 0.3),
learnFunc = "Std_Backpropagation",
learnFuncParams = c(0.2, 0),
updateFunc = "Topological_Order",
updateFuncParams = c(0),
hiddenActFunc = "Act_Logistic",
shufflePatterns = TRUE,
linOut = TRUE)
```

Much of this will be familiar to you by now, so I will run through it only briefly. The network has two hidden layers; the first hidden layer contains 12 neurons; the second hidden layer contains 8 neurons. Weights and biases are initialized randomly, with a logistic activation function in the hidden layers and linear activation function for the output neuron. You

will notice, that for this small dataset, the R code executes relatively quickly.

Prediction is carried out using the `predict` method. Here is how I do that for our example:

```
predMLP<- sign(predict(fitMLP, temp[-train
   ,1:6]))
```

Since we have classification data, we should take a look at the confusion matrix. Here it is:

```
> table( predMLP,sign(temp[-train,7]) ,
dnn =c("Predicted" , " Observed"))

          Observed
Predicted -1  1
       -1 67  9
       1  21 27
```

Notice the model did not fit the data perfectly. The off diagonal elements in the confusion matrix indicate the number of misclassified observations. It is often helpful to calculate the misclassification as an error rate. Here is how I do that:

```
> error_rate = (1- sum( predMLP == sign(
   temp[-train,7])  )/ 124)
>  round( error_rate ,3)  [1] 0.242
```

The classification DNN has an overall error rate of around 24% (or an accuracy rate of around 76%).

It is worth highlighting how easily and quickly a classification DNN can be constructed using R. I can recall spending many hours writing code in C++ and C in order to complete what can now be achieved in a few lines of code in R. If you are interested in the art of data science, that is using the tools of the discipline to extract meaningful insights, use of R will certainly boost your productivity.

Asking R users how they build a particular type of model is a bit like the old Econometrician joke:- *How many econometricians does it take to change a light-bulb? Five, plus or minus*

eight! Let's face it, most of us have our own favorite package, tool and technique. Whilst this is human, a better strategy is to remain flexible, and uncover as many ways as possible to solve a problem. For R this involves using alternative tools and packages.

Let's look at how to build a classification DNN using the `AMORE` package. (By the way, you could also do this with the packages we have already used to build a regression DNN. Having as many arrows in your toolkit is essential to data science success.) First, detach the `RSNNS` package and load `AMORE` as follows:

```
> detach("package:RSNNS", unload=TRUE)
> library(AMORE)
```

Here is how to specify a classification DNN in the `AMORE` package:

```
net <- newff(n.neurons=c(6,12,8,1),
learning.rate.global=0.01,
momentum.global=0.5,
error.criterium="LMLS",
Stao=NA,
hidden.layer="sigmoid",
output.layer="purelin",
method="ADAPTgdwm")
```

Much of this is similar to what you have seen before. Note however, that the AMORE package requires you to specify the number of input and output nodes as well as hidden nodes. This is achieved using the statement `n.neurons=c(6,12,8,1)`, the first number reflects the six input attributes, the second and third values represent the number of neurons in the first and second hidden layers, and the final number the number of output neurons, in this case 1.

Most of the time data scientists train DNNs by minimizing the mean squared error of the training set. However, in the presence of outliers, the resulting model can struggle to

capture the mechanism that generates the data. For this reason, when building DNN models I also like to minimize robust error metrics. In this case I use the mean log squared error (`LMLS`)[59]. Other robust choices available in the `AMORE` package include `"TAO"` for the Tao error[60] as well as the standard mean squared error (`LMS`). The argument `method` specifies the learning method used. I have chosen to use adaptive gradient descend with momentum. `"ADAPTgdwm"`.

Next, we assign the attributes to the R object `X`, and the response variable to the R object `Y`.

```
>  X<-temp[train,]
>  Y<-temp[train,7]
```

Next we fit the model. Here is how to do that:

```
fit <- train(net,
P=X,
T=Y,
error.criterium="LMLS",
report=TRUE,
show.step=100,
n.shows=5 )
```

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

```
index.show: 1 LMLS 0.239138435481238
index.show: 2 LMLS 0.236182280077741
index.show: 3 LMLS 0.230675203275236
index.show: 4 LMLS 0.222697557309232
index.show: 5 LMLS 0.214651839732672
```

Once the model has converged, use the `sim` method to fit the model using the test sample:

```
> pred <- sign(sim(fit$net, temp[-train,]))
```

The confusion can be calculated using

```
> table( pred,sign(temp[-train,7]) ,
```

```
dnn =c("Predicted" ,
" Observed"))
          Observed
Predicted -1  1
       -1 71 10
       1  17 26
```

Finally, the error rate is given by:

```
> error_rate = (1- sum( pred == sign(temp[-
   train,7])  )/ 124)
>  round( error_rate ,3)
[1] 0.218
```

Overall, the use of adaptive gradient descend with momentum learning combined with a robust error resulted in a lower misclassification error rate of around 22% relative to the classification DNN developed using the `RSNNS` package. The key take away here is that as you build your DNN models, do so in a variety of packages, use different learning algorithms, network architecture and so on. In other words, at the heart of the art of building great DNNs is experimentation.

How to Model Multiple Response Variable's

In image processing modeling multiple response variables (aka pixels) in a single DNN is standard practice. However, in many areas of data science, models are often built containing only one response variable. This is probably a hangover from the residual dominance of the once ubiquitous linear regression model.

It was probably Sir Francis Galton's sweet pea experiment back in 1875[61] which propelled forward the notion of linear regression as a tool for modeling relationships in data. It subsequently became the defacto technique of analysis in a wide variety of disciplines. Partly, due to the ease of calculation, theoretical underpinnings and useful insights, it remains a useful tool to this day.

A key statistical reason for modeling multiple response variables jointly on the same set of attributes is that it can lead to more efficient parameter estimates[62] in regression type models. I first came across the idea of joint estimation of response variables several years ago whilst working on surrogate markers in clinical trials. The wonderful thing about this is that the generation of more efficient estimates can shorten the duration of clinical trials[63]. In short, more efficient estimates lead to faster decisions as the result of greater confidence in terms of statistical inference. I can think of multiple instances, across a wide range of commercial, industrial and research applications, where modeling multiple response variables on the same set of attributes is useful. Think for moment about an application in your line of work. Now let's look at how to do this in R.

We build our DNN using the data frame `bodyfat` contained in the `TH.data` package. The data was originally 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 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 3. Here is how to load the data:

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

Waist and hip circumference will serve as the response variables, see Figure 2.13. A visualization of the attributes is shown in Figure 2.14.

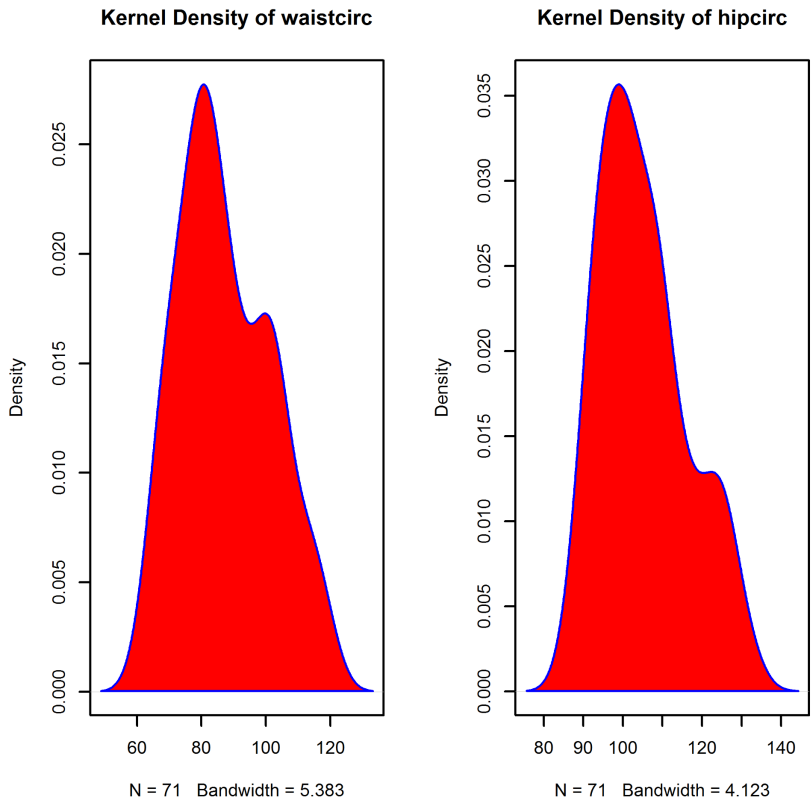


Figure 2.13: Kernel density plot for response variables

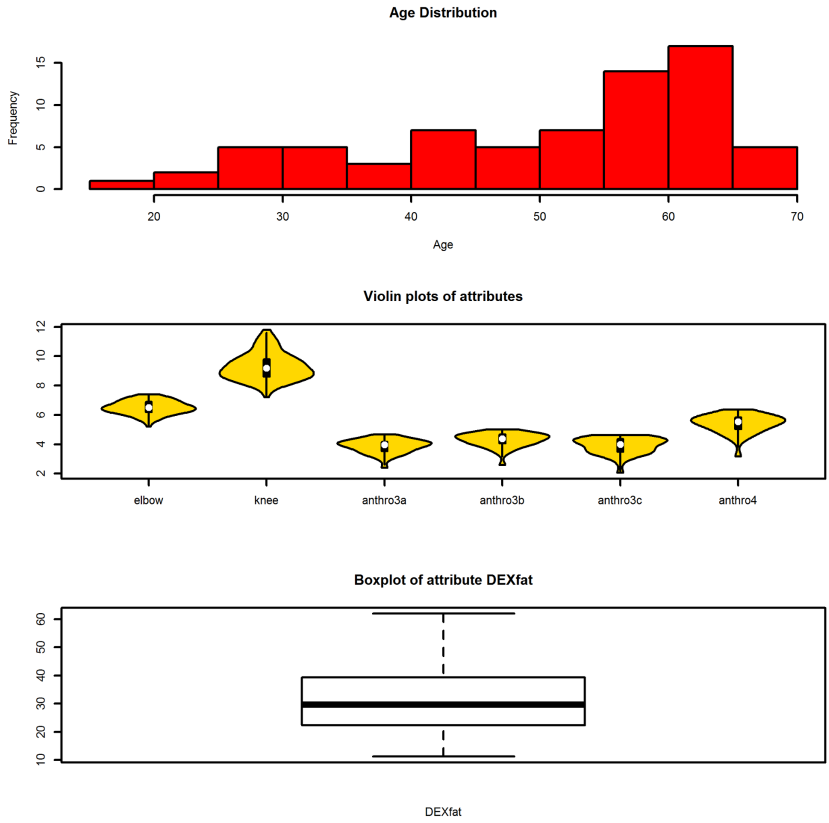


Figure 2.14: DNN `bodyfat` attributes

We will use the `neuralnet` package to build the DNN. The `Metrics` package calculates a variety of useful performance metrics and is also loaded:

```
> library(neuralnet)
> require(Metrics)
```

Since we are dealing with a rather small dataset, with only 71 observations will use 60 observations for the training sample. The data is sampled without replacement as follows:

```
> set.seed(2016)
> train <- sample(1:71,50 , FALSE)
```

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

Next, the observations are standardized, and the formula to be used to build the DNN is stored in the R object f:

```
> scale_bodyfat<-as.data.frame(scale(log(
   bodyfat)))
> f<-  waistcirc + hipcirc ~ DEXfat+age +
   elbowbreadth + kneebreadth + anthro3a +
   anthro3b + anthro3c +anthro4
```

Notice the use of waistcirc + hipcirc ~ to indicate two response variables. In general, for k response variables $\{r_1,\ldots,r_k\}$ you would use r_1+r_2+...+r_k~.

I fit a model with two hidden layers with 8 neurons in the first hidden layer and 4 neurons in the second hidden layer. The remaining parameters are similar to those we have already discussed:

```
it<- neuralnet(f,
data = scale_bodyfat[train,],
hidden=c(8,4),
 threshold=0.1,
```

```
 err.fct = "sse",
algorithm = "rprop+",
act.fct = "logistic",
linear.output=FALSE
)
```

A visualization of the fitted model is shown in Figure 2.15. The model only took 45 steps to converge with a sum of squared errors equal to 25.18.

Now a little bit of housekeeping, I copy the data in `scale_bodyfat` and store it in the R object `without_fat`. This is the object we will play with. It also allows us to preserve the original values contained in `scale_bodyfat` which we will use later. Notice that I also remove the response variables from this new R object using the `NULL` argument:

```
> without_fat<-scale_bodyfat
> without_fat$waistcirc <-NULL
> without_fat$hipcirc<-NULL
```

Now we are ready to use the model on the test sample. The approach is similar to that which we have seen before. The second line prints out the predicted values (first few only shown):

```
> pred <- compute(fit, without_fat[-train,]
    )
> pred$net.result
pred$net.result
                [,1]              [,2]
48  0.8324509945946 0.7117765670627
53  0.1464097674972 0.0389520756586
58  0.7197107887754 0.6215091479349
62  0.8886019185711 0.8420724129305
```

That's it! The DNN with two response variables has been successfully built.

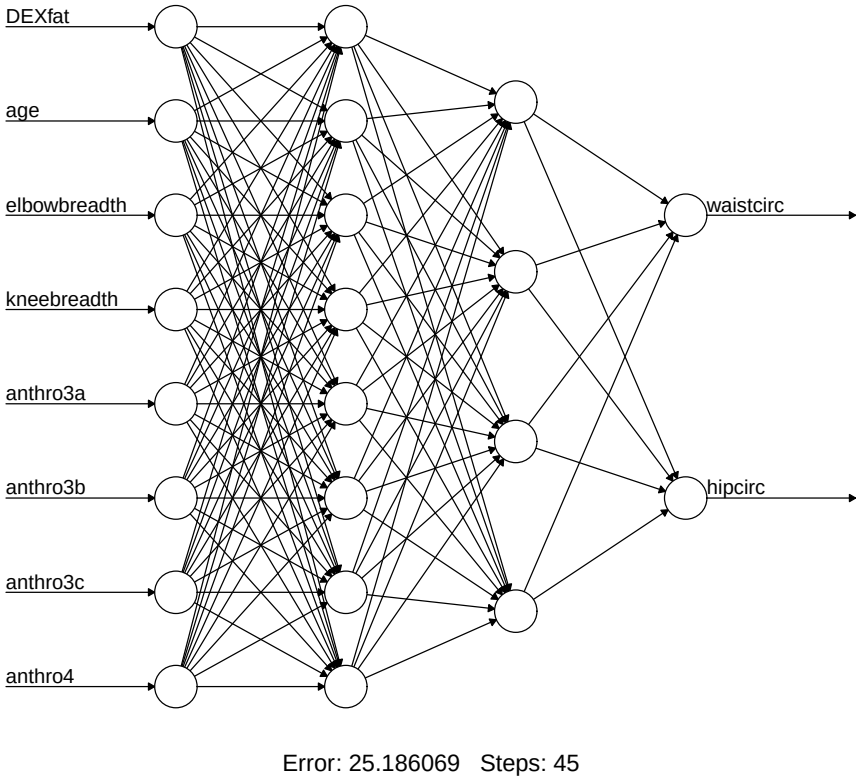


Figure 2.15: DNN model with two outputs

Hey, after all of that we may as well compare our 21st century model results to Sir Francis Galton's 19th century linear regression model. This will involve building two models. One for each of the response variables. First, create the formulas:

```
> fw<-  waistcirc ~ DEXfat+age +
   elbowbreadth + kneebreadth + anthro3a +
   anthro3b + anthro3c +anthro4
> fh<-  hipcirc ~ DEXfat+age + elbowbreadth
    + kneebreadth + anthro3a + anthro3b +
   anthro3c +anthro4
```

Now run each of the models using the training data:

```
> regw<-linReg<-lm(fw,data = scale_bodyfat[
   train,])
> regh<-linReg<-lm(fh,data = scale_bodyfat[
   train,])
```

Next, build predictions using the test sample:

```
> predw<-predict(regw, without_fat[-train
   ,])
> predh<-predict(regh, without_fat[-train
   ,])
```

Wow, that was pretty fast! So how does our DNN model compare with the linear regressions? The mean squared error statistic for the DNN model response variable `waistcirc` can be calculated as follows:

```
>  mse(scale_bodyfat [-train,10],  pred$net
   .result[,1])
[1] 0.5376666652
```

And for Galton's regression model:

```
>  mse(scale_bodyfat [-train,10], predw)
[1] 0.3670629158
```

Galton's 19th century linear regression for waist size beats the pants off our deep learning model!
What about hip size? Here is the metric for the DNN:

```
>  mse(scale_bodyfat [-train,10], pred$net.
   result[,2])
[1] 0.5530302859
```

And for Galton's hip size regression model:

```
>  mse(scale_bodyfat [-train,10], predh)
0.5229634116
```

A much closer race here, but the linear regression still wins with a smaller mean squared error.

Of course, in practice you will build a variety of models using different learning algorithms, number of neurons and so on; and the linear regression model results serve as a useful benchmark. To illustrate this point, I build a DNN using traditional backpropagation with the `deepnet` package.

First, I collect the attributes into the R object `X`, and the response variables into the R object `Y`:

```
>  set.seed(2016)
>  X= as.matrix(without_fat[train,])
>  Y= as.matrix(scale_bodyfat[train,3:4])
```

The DNN model with 2 hidden layers is estimated as follows:

```
 fitB<-nn.train(x=X, y=Y,
 initW = NULL,
 initB = NULL,
 hidden = c(8,4),
 activationfun = "sigm",
learningrate = 0.02,
momentum =0.74,
learningrate_scale =1 ,
output = "linear",
numepochs = 970,
batchsize = 60,
hidden_dropout = 0,
visible_dropout = 0)
```

The predicted values, using the test sample, are stored in the R object `predB`:

```
> Xtest<- as.matrix(without_fat[-train,])
> predB <- nn.predict(fitB, Xtest)
```

And the mean square error for `waistcirc`, and `hipcirc` are calculated as:

```
>  mse(scale_bodyfat [-train,10],  predB
   [,1])
[1] 0.1443659185
```

```
>  mse(scale_bodyfat [-train,10], predB
   [,2])
[1] 0.1409938484
```

In this case, the DNN model outperforms (in terms of mean square error) the linear regression models.

The key takeaway is not that any particular approach is absolutely inferior, rather that the data science tool you use should be selected in order to fit the nature of the problem you have at hand. As novelist Stephen King wrote[65] "*It's best to have your tools with you. If you don't, you're apt to find something you didn't expect and get discouraged.*"

Notes

[33]If you are interested in the history of deep learning see the interesting article at http://recode.net/2015/07/15/ai-conspiracy-the-scientists-behind-deep-learning/

[34]Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[35]Couprie, Camille, et al. "Indoor semantic segmentation using depth information." arXiv preprint arXiv:1301.3572 (2013).

[36]Sarkar, Soumik, et al. "Occlusion Edge Detection in RGB-D Frames using Deep Convolutional Networks." arXiv preprint arXiv:1412.7007 (2014).

[37]For a successful application in image recognition see A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classication with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097{1105, 2012.

[38]See for example:

1. Keshmiri, Soheil, et al. "Application of Deep Neural Network in Estimation of the Weld Bead Parameters." arXiv preprint arXiv:1502.04187 (2015).
2. Dahl, George E., et al. "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition." Audio, Speech, and Language Processing, IEEE Transactions on 20.1 (2012): 30-42.
3. R. Collobert and J. Weston, A unified architecture for natural language processing: Deep neural networks with multitask learning, in Int. Conf. Machine Learning, 2008.
4. D. C. Ciresan, U. Meier, J. Masci, and J. Schmidhuber, Multi-column deep neural network for traffic sign classification, Neural Networks, vol. 32, pp. 333-338, 2012.
5. Y.LeCun, L.Bottou, and P.Haffiner, Gradient-based learning applied document recognition, Proc. of the IEEE, vol.86, pp.2278-2324, 1998.

[39]For further details see Seide, Frank, et al. "Feature engineering in context-dependent deep neural networks for conversational speech transcription." Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on. IEEE, 2011.

[40]Fukushima, Kunihiko. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position." Biological cybernetics 36.4 (1980): 193-202.

[41]See for example Narasimhan, Srinivasa G., and Shree K. Nayar. "Contrast restoration of weather degraded images." Pattern Analysis and Machine Intelligence, IEEE Transactions on 25.6 (2003): 713-724.

[42]Pomerleau, Dean. "Visibility estimation from a moving vehicle using the RALPH vision system." IEEE Conference on Intelligent Transportation Systems. 1997.

[43]Farhan Hussain and Jechang Jeong, "Visibility Enhancement of Scene Images Degraded by Foggy Weather Conditions with Deep Neural Networks," Journal of Sensors, vol. 2016, Article ID 3894832, 9 pages, 2016. doi:10.1155/2016/3894832

[44]Invincea Labs provides an advanced endpoint security solutions that prevents targeted attacks, detects breaches & supports threat response by eradicating malware.

[45]Saxe, Joshua, and Konstantin Berlin. "Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features." arXiv preprint arXiv:1508.03096 (2015).

[46]See Farhan Hussain and Jechang Jeong, "Efficient Deep Neural Network for Digital Image Compression Employing Rectified Linear Neurons," Journal of Sensors, vol. 2016, Article ID 3184840, 7 pages, 2016. doi:10.1155/2016/3184840

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

[48]In any year you will find many examples. Here are a couple, from several years ago, to give you a sense of the real world importance of this issue:

- In November 2013 autoblog reported that Kwon Moon-sik, research and development president of the car manufacturer Hyundai, resigned as a result of a series of quality issues. See http://www.autoblog.com/2013/11/12/hyundai-executive-resigns-quality-issues/. See also the Reuters report at http://www.reuters.com/article/us-hyundai-rd-idUSBRE9AA0F920131111.
- Towards the end of 2014 Fiat Chrysler's quality chief, Doug Betts, left the company, one day after the automaker ranked last in a closely watched U.S. scorecard on vehicle reliability. See Automotive News article *"Betts leaves Chrysler after another poor quality showing"* by Larry P. Vellequette.

[49]See:

- Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

- Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." arXiv preprint arXiv:1207.0580 (2012).

[50]The idea was proposed by Leo Breiman, who called it "bootstrap aggregating" see Breiman, Leo. "Bagging predictors." Machine learning 24.2 (1996): 123-140.

[51]See for example Dahl, George E., Tara N. Sainath, and Geoffrey E. Hinton. "Improving deep neural networks for LVCSR using rectified linear units and dropout." Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. IEEE, 2013.

[52]The link appears to have been validated by a large number of studies. See for example Sharma A, Madaan V, Petty FD. Exercise for Mental Health. Primary Care Companion to The Journal of Clinical Psychiatry. 2006;8(2):106.

[53]Here are three examples in totally different areas of application :

- Payan, Adrien, and Giovanni Montana. "Predicting Alzheimer's disease: a neuroimaging study with 3D convolutional neural networks." arXiv preprint arXiv:1502.02506 (2015).
- Oko, Eni, Meihong Wang, and Jie Zhang. "Neural network approach for predicting drum pressure and level in coal-fired subcritical power plant." Fuel 151 (2015): 139-145.
- Torre, A., et al. "Prediction of compression strength of high performance concrete using artificial neural networks." Journal of Physics: Conference Series. Vol. 582. No. 1. IOP Publishing, 2015.

[54]Harrison, David, and Daniel L. Rubinfeld. "Hedonic housing prices and the demand for clean air." Journal of environmental economics and management 5.1 (1978): 81-102.

[55]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.

[56]See my book **Build Your Own Neural Network TODAY!** for a quick and easy explanation of these terms.

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

[58]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.

[59]See Liano, Kadir. "Robust error measure for supervised neural network learning with outliers." Neural Networks, IEEE Transactions on 7.1 (1996): 246-250.

[60]See for example:

- Pernía-Espinoza, Alpha V., et al. "TAO-robust backpropagation learning algorithm." Neural Networks 18.2 (2005): 191-204.
- Ordieres-Meré, Joaquín B., Francisco J. Martínez-de-Pisón, and Ana González-Marcos. "TAO-robust backpropagation learning algorithm." Neural networks 18.2 (2005): 191-204.
- Rusiecki, Andrzej. "Robust LTS backpropagation learning algorithm." Computational and ambient intelligence. Springer Berlin Heidelberg, 2007. 102-109.

[61]See Stanton, Jeffrey M. "Galton, Pearson, and the peas: A brief history of linear regression for statistics instructors." Journal of Statistics Education 9.3 (2001).

[62]See for example Zellner, Arnold, and Tong Hun Lee. "Joint estimation of relationships involving discrete random variables." Econometrica: Journal of the Econometric Society (1965): 382-394.

[63]See for example N.D. Lewis. Surrogate Markers in Clinical Trials. PhD Thesis. University of Cambridge

[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.

[65]See King, Stephen. "On writing: A memoir of the craft." New York: Scribner (2000). If you have not already done so, go out and purchase a copy of this book at your earliest convenience. It is jam packed full of advice that will serve you well.

Chapter 3

Elman Neural Networks

All models are wrong, but some are useful.

George E. P. Box

The Elman neural network is a recurrent neural network. 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 3.1. 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.

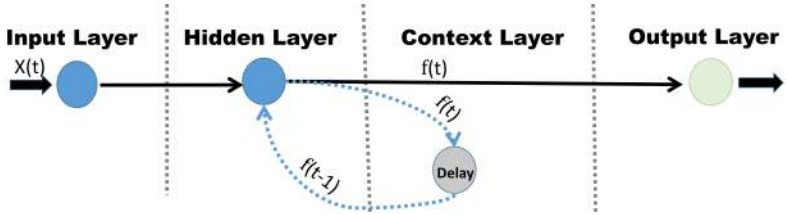


Figure 3.1: Simple recurrent neural network

What is an Elman Neural Network?

Elman neural 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.

To solidify this idea let's take a look at the Elman network developed by Khatib et al.[66] to predict hourly solar radiation. The network is illustrated in Figure 3.2. 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.

What is the 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.

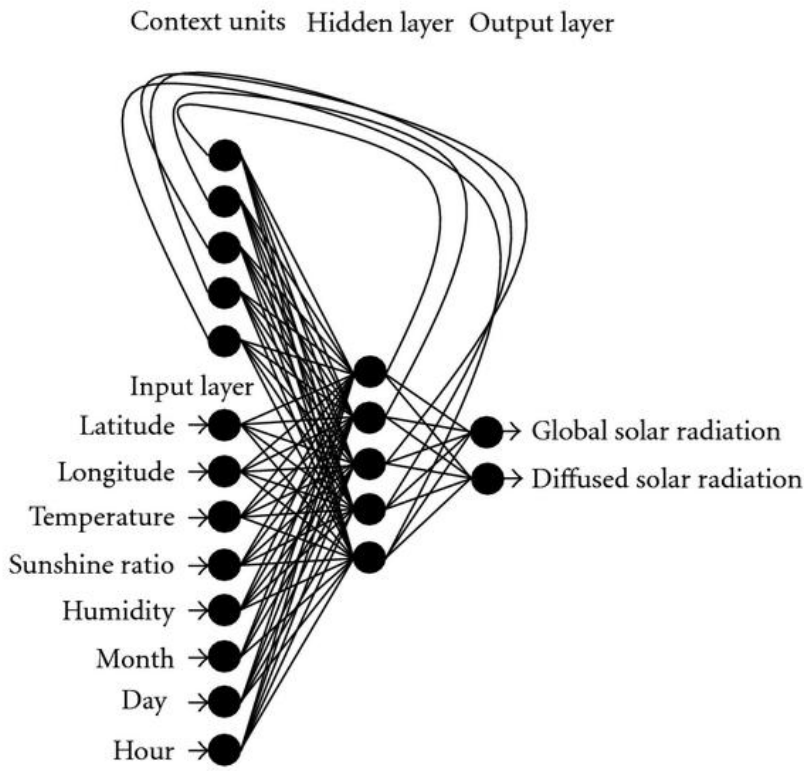


Figure 3.2: Schematic illustration of the Elman network to predict hourly solar radiation. Source Khatib et al. cited in endnote sec. 66

How to Understand 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 3.3. 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:

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

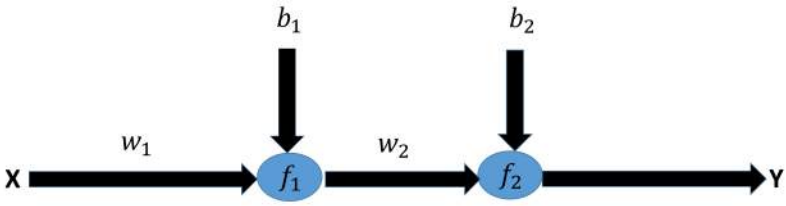


Figure 3.3: Two neuron network

In an Elman network, as shown in Figure 3.4, there is a context neuron which feeds the signal from the hidden layer neuron[67] 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. The output at time t is given by:

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

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.

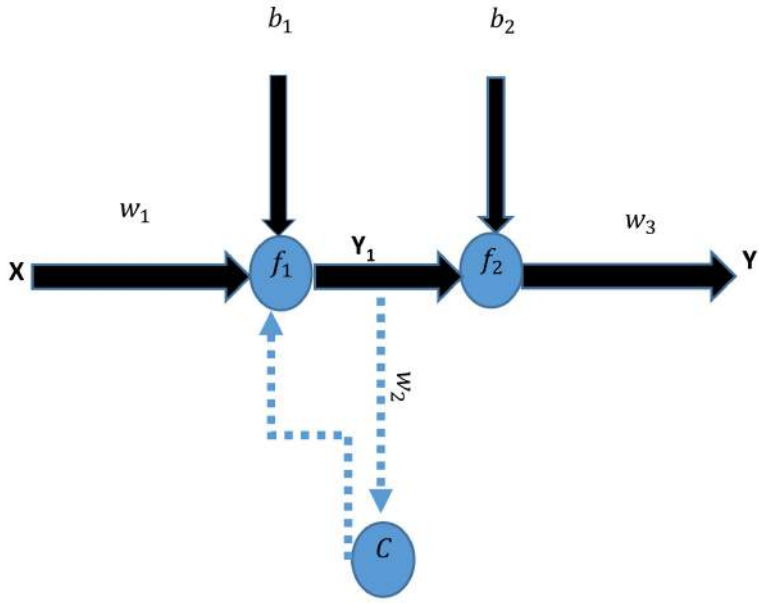


Figure 3.4: Two node Elman network

How to Use Elman Networks to Boost Your Result's

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 3.2 to predict hourly solar radiation developed by Khatib et al. The property of memory retention is also useful for pattern recognition and robot control.

Four Smart Ways to use Elman Neural Networks

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?

The Ultimate Weather Forecasting Model

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.

How to Immediately Find a Serious Fault

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.*"

An Innovative Idea for Better 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.*"

How to Make a "Killing" in 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.

The Easy Way to Build Elman Networks

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

☛ ***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)
```

Here is How to Load the Best 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)
```

Why Viewing Data is the New Science

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 3.5. 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`.

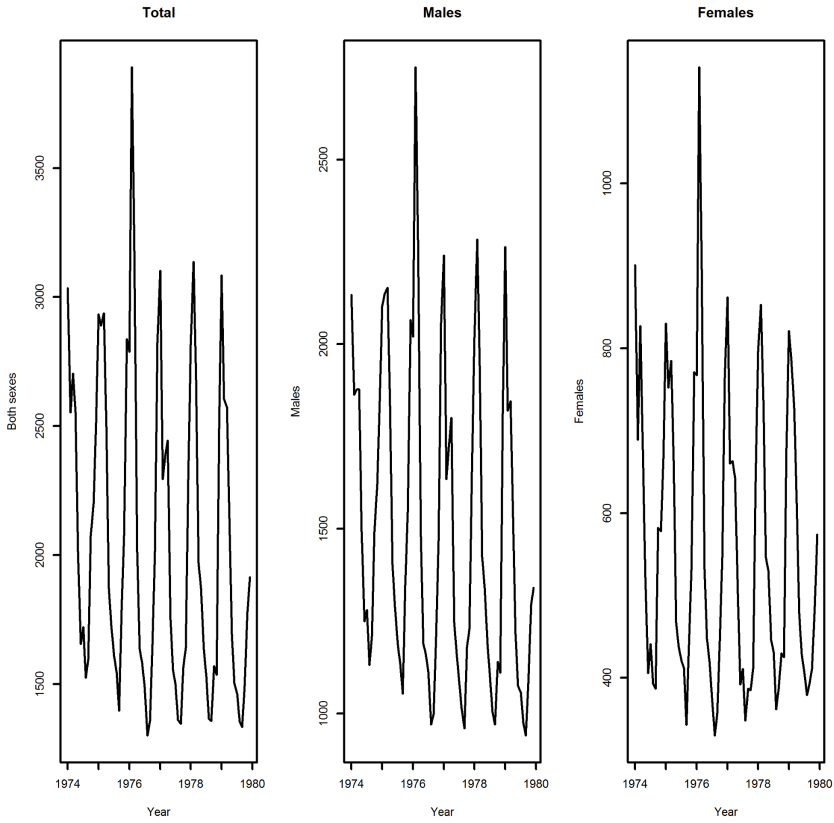


Figure 3.5: Monthly number of deaths from bronchitis, emphysema and asthma in the United Kingdom

Overall, we see that there has been considerable variation in the number of deaths over time, with 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 3.6.

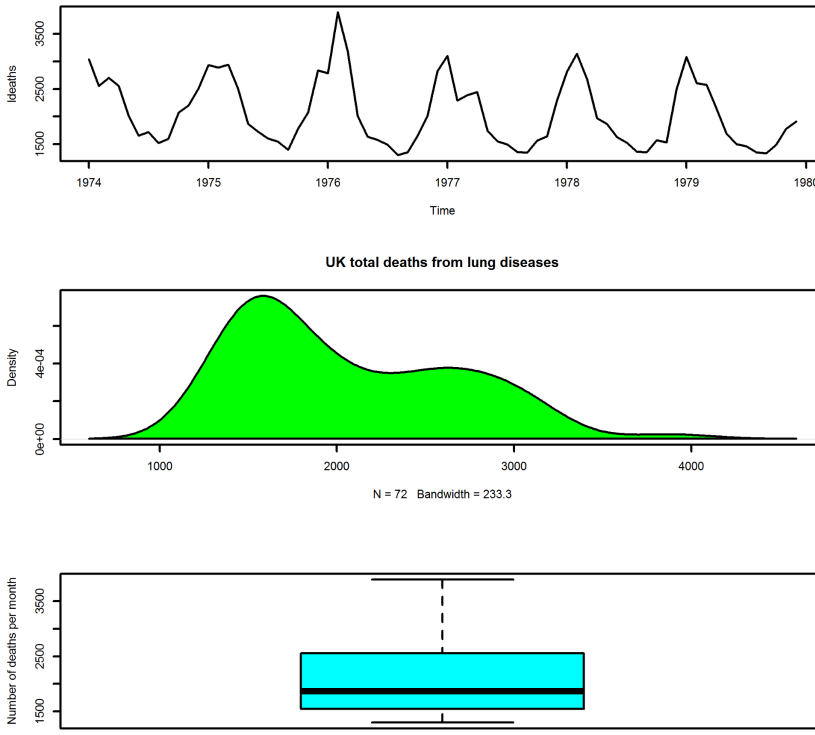


Figure 3.6: Visual summary of total number of deaths

The Secret to Transforming 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 `quantmod` package. This requires that we convert `y` into a `zoo` class object:

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

Now we are ready to use the `quantmod 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 an Interesting 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 3.7.

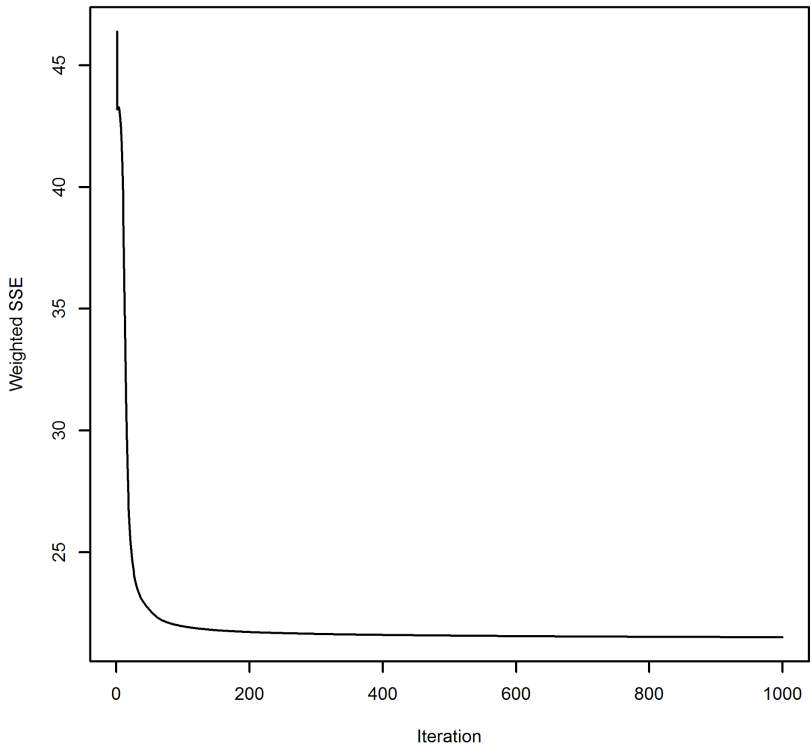


Figure 3.7: Elman neural network error plot

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.

Creating the Ideal Prediction

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 3.8.

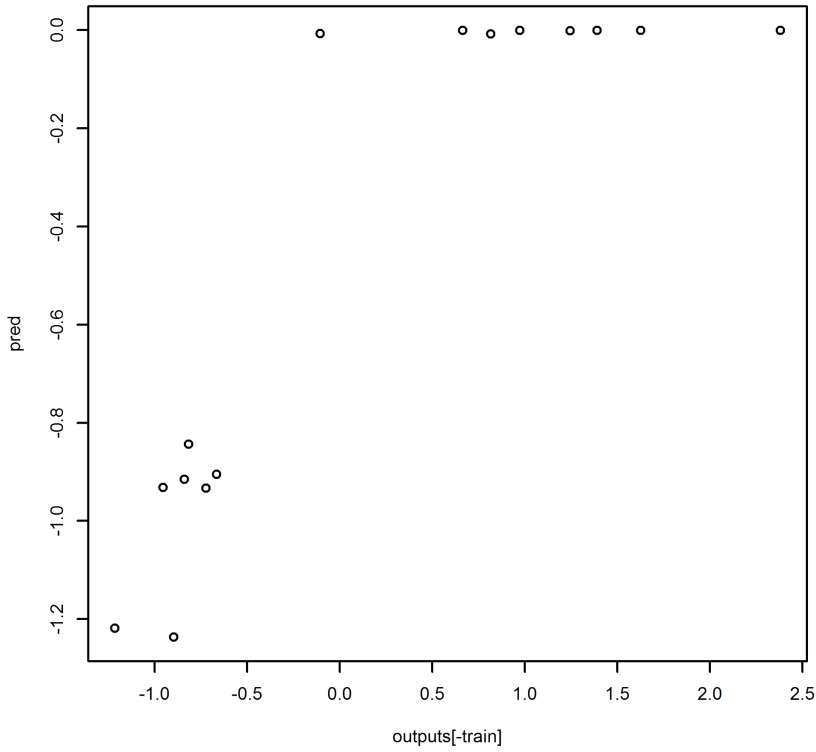


Figure 3.8: Elman network actual and predicted values

The squared correlation coefficient is relatively high at 0.78. To improve on this number so you can achieve your ideal prediction, you simply need to experiment with different model parameters. Give it a go, rebuild the model to improve the overall performance.

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

Notes

[66]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).

[67]In the Elman network the neurons typically use sigmoidal activation functions.

[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.

[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 4

Jordan Neural Networks

> *It is complete nonsense to state that all models are wrong, so let's stop using that quote.*
>
> Mark van der Laan

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 4.1.

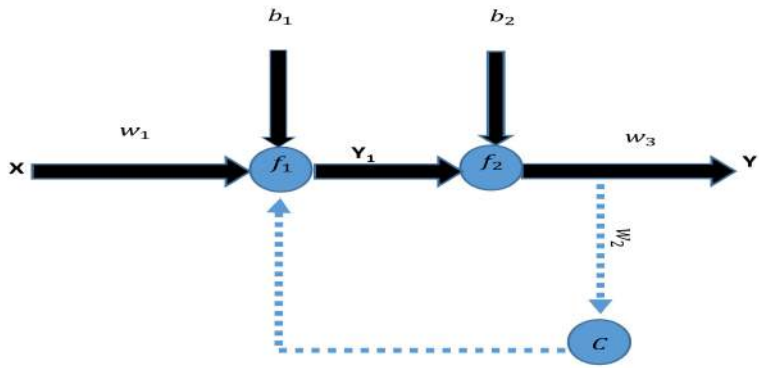


Figure 4.1: A simple Jordan neural network

The activity of the output node[s] is recurrently copied back into the context nodes. This provides the network with a mem-

ory of its previous state.

Three Problems Jordan Neural Networks Can Solve

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.

The Ultimate Guide for Wind Speed 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.*"

How to Classify 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%.

Deep Learning to Woo Spanish Speakers

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%.

Essential Elements for Effective Jordan Models in R

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 modeling 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.

Which are the Appropriate 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 4.2 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)
```

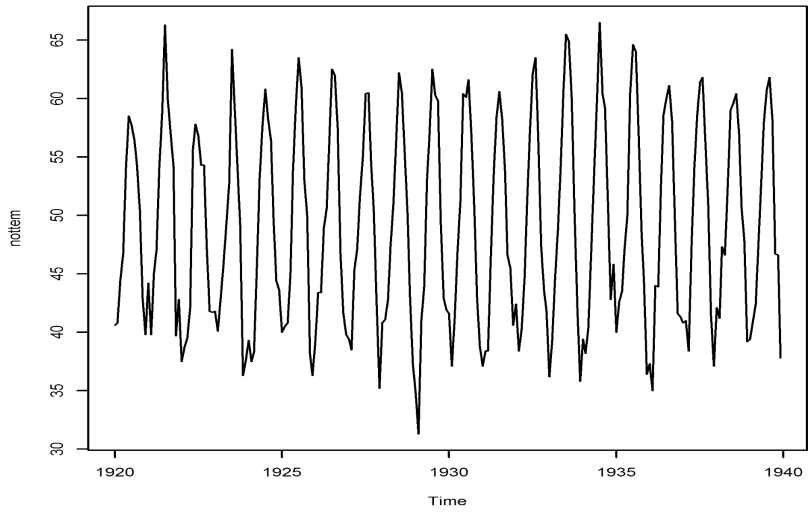


Figure 4.2: Average Monthly Temperatures at Nottingham 1920–1939

A Damn Good Way to Transform Data

For the most part I like to standardize my attribute data prior to using a neural network model. Whilst there are no fixed rules about how to normalize inputs here are four popular choices for an attribute x_i:

$$z_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{4.1}$$

$$z_i = \frac{x_i - \overline{x}}{\sigma_x} \tag{4.2}$$

$$z_i = \frac{x_i}{\sqrt{SS_i}} \tag{4.3}$$

$$z_i = \frac{x_i}{x_{max} + 1} \tag{4.4}$$

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.
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 101 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 4.3, and was created using the `plot` method:

```
>plot(temp)
```

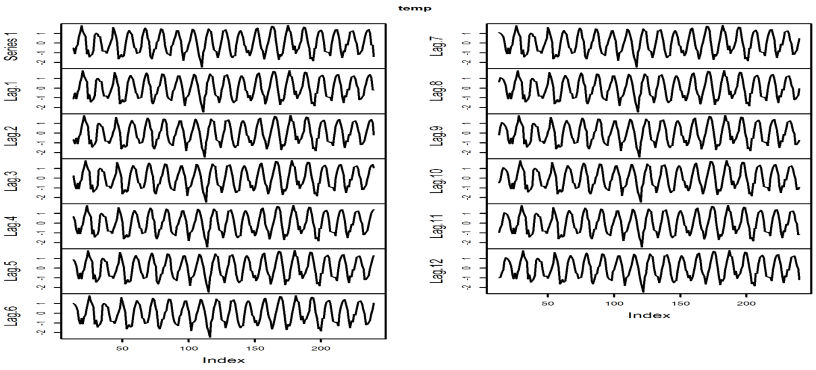


Figure 4.3: Response and attribute variables for Jordan network

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

Here is How to Select 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)
```

Use This Tip to Estimate Your Model

The model is estimated along similar lines as the Elman neural network on page 102. 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 4.4. It was created using:

```
> plotIterativeError(fit)
```

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

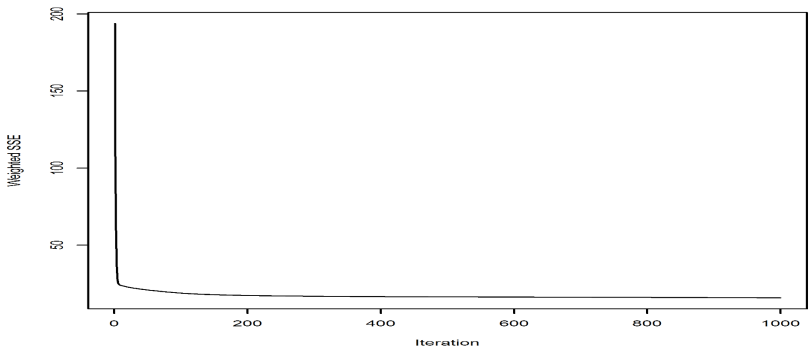


Figure 4.4: Training error by iteration

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!

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.

[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.

Chapter 5

The Secret to the Autoencoder

Though this be madness, yet there is method in't.

William Shakespeare

The autoencoder (sometimes called an autoassociator) is an unsupervised three-layer feature learning feed-forward neural network network. Architecturally, as shown in Figure 5.1, it is very similar to the multilayer perceptron; it includes an input layer, a hidden layer and an output layer. The number of neurons in the input layer is equal to the number of neurons in the output layer. The number of hidden neurons is less (or more) than the number of input neurons. It is different from an MLP because the output layer has as many nodes as the input layer, and instead of training it to predict some target value y given inputs x, an autoencoder is trained to reconstruct its own inputs x.

The autoencoder consists of an encoder and a decoder. The mapping of the input layer to the hidden layer is called encoding and the mapping of the hidden layer to the output layer is called decoding. The encoder takes the vector of input attributes and transforms them typically through sigmoid activation functions in the hidden layers into new features. The

decoder then converts these features back to the original input attributes. So for example, in Figure 5.1, the autoencoder takes the attributes x_1, x_2, x_3, x_4 encodes them using three hidden units each containing sigmoid activation functions h_1, h_2, h_3, and then decodes them to obtain estimates $\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4$ of the original attributes.

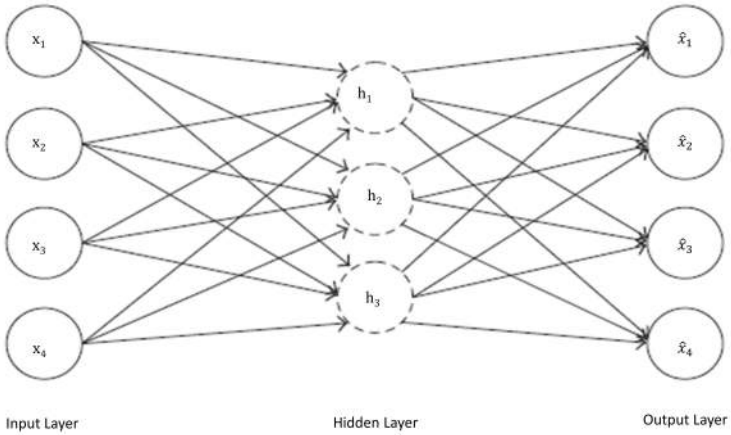


Figure 5.1: Basic Autoencoder

A Jedi Mind Trick

At this point you may be wondering is this some complicated Jedi mind trick? What is the point of taking our original attributes, transforming them into who knows what, and then obtaining estimates of their original values? What is going on here?

Well, yes there is a little bit of the Jedi Master in this. You see, one of the great challenges faced by data scientists, statisticians, psychologists and others involved in data analysis is the problem of dimensionality reduction.

Here is what this issue involves: Suppose you have 300 potential attributes that might explain your response variable.

In traditional statistical modeling, it would not be possible to include all 300 attributes in your model. What to do? Enter dimensionality reduction techniques, the most famous being principal component analysis (PCA) developed in 1901 by University College, London, professor of mathematics Karl Pearson.

I first came across PCA though the classic 1972 and 1973 papers of the then University of Kent at Canterbury scholar I. T. Jolliffe[86]; the technique has been a permanent addition to my data science toolkit ever since. Briefly, PCA is a statistical procedure that uses an orthogonal transformation to convert a number of (possibly) correlated attributes into a (smaller) number of uncorrelated variables called principal components. The principal components are linear combinations of the original attributes. Here are the key points to be aware of:

1. The number of principal components is less than or equal to the number of original variables.

2. The first principal component accounts for the largest proportion of the variability in the data; and each succeeding component accounts for as much of the remaining variability as possible conditional on being orthogonal (aka uncorrelated) to the preceding components.

3. The principal components are orthogonal because they are the eigenvectors of the covariance matrix, which is of course symmetric.

The Secret Revealed

I'll always remember what Jabba the Hutt said rather pointedly to Bib Fortuna in Return of the Jedi "*You weak-minded fool! He's using an old Jedi mind trick!*" The thing about a Jedi mind trick was once it was revealed to you, it lost much of its power or became totally ineffective. Perhaps you recall the

rather humorous scene in the Phantom Menace, when Qui-Gon Jinn, stranded on Tatooine, attempts to use a mind trick on Watto in order to encourage him to accept Republic credits for the purchase of a T-14 hyperdrive. Watto interjects "*...you think you're some kind of Jedi, waving your hand around like that? I'm a Toydarian! Mind tricks don't work on me, only money! No money, no parts, no deal!*"

Let's take another look at the autoencoder "trick",this time without the hand waving! An autoencoder is a feed forward neural network used to learn representations of data. The idea is to train a network with at least one hidden layer to reconstruct its inputs. The output values are set equal to input values, i.e. $\hat{x} = x$ in order to learn the identity function $h(x) \approx x$. The autoencoder achieves this by mapping input attributes to hidden layer nodes using an encoder function:

$$h(x) = f(Wx + b_h)$$

where x is the input vector of attributes, f denotes the sigmoid function, b_h is the vector of hidden neuron biases, and W is the matrix of hidden weights. The data is reconstructed using a linear decoder:

$$g(\hat{x}) = \lambda f(Wx + b_h) + b_g$$

After learning the weights W, each hidden neuron represents a certain feature of the input data. Thus, the hidden layer $h(x)$ can be considered as a new feature representation of the input data. The hidden representation $h(x)$ is then used to reconstruct an approximation $\hat{x}$ of the input using the decoder function $g(\hat{x})$.

The most commonly used function for the encoder and decoder is a nonlinear sigmoid function. Training is typically achieved by minimizing the squared reconstruction error $(g(\hat{x}) - x)^2$ using the backpropagation by gradient descent algorithm. In practice, the restriction $\lambda = W^T$is often imposed to reduce the number of parameters.

The key thing to keep in mind is that by learning the hidden representation $h(x)$ that can reconstruct the original input attributes, the autoencoder captures important features of the input attributes. For example, if the number of hidden layer neurons is larger than the number of input layer neurons the hidden layers map the input to a higher dimension. Similarly, if the number of hidden neurons is less than the number of input neurons the autoencoder's hidden layer essentially compresses the input attributes in such a way that they can be efficiently reconstructed[87].

The compressed representation has lower dimensionality than the original input attributes. For example, when an autoencoder has 400 input neurons and 60 hidden nodes, the original 400-dimensional input is 'reconstructed' approximately from the 60-dimensional output of the hidden layer. If we use the output of the hidden layer as a representation of an input of the network, the autoencoder plays the role of a feature extractor.

It turns out that autoencoders can implement a variety of dimensionality reduction techniques. A linear autoencoder can learn the Eigenvectors of the data equivalent to applying PCA to the inputs[88]. A nonlinear autoencoder is capable of discovering more complex, multi-modal structure in the data. In certain situations, a nonlinear autoencoder can even outperform PCA for certain dimensionality reduction tasks involved with handwriting and face recognition[89].

Autoencoders can therefore be used to learn a compressed (or expanded) representation of data with minimum reconstruction loss. As pointed out by deep learning scholar Geoffrey Hinton[90] "*It has been obvious since the 1980s that backpropagation through deep autoencoders would be very effective for nonlinear dimensionality reduction, provided that computers were fast enough, data sets were big enough, and the initial weights were close enough to a good solution. All three conditions are now satisfied. Unlike nonparametric methods, autoencoders give mappings in both directions between the data*

and code spaces, and they can be applied to very large data sets because both the pretraining and the fine-tuning scale linearly in time and space with the number of training cases."

A Practical Definition You Can Run With

We can summarize by stating that an autoencoder is an artificial neural network that attempts to reproduce its input, i.e., the target output is the input. Another way of saying this, and a little more formal is that an autoencoder is a feedforward neural network that tries to implement an identity function by setting the outputs equal to the inputs during training. The parameters are learned by minimizing a loss functions. The loss function measures the difference between input attributes and output. Stochastic Gradient Descent is typically used to minimize the loss function:

$$L(\hat{x}, x) = -\sum_{i=1}^{k} \left[x_i \log \hat{x}_i + (1 - x_i) \log \left(1 - x_i\right) \right]$$

How to Save the Brazilian Cerrado

Let's look at a very practical application of an autoencoder as a feature extractor. Since the environment impacts us all, we take our example from environmental and space science.

The Brazilian Cerrado, at almost three times the size of Texas, is the world's most biologically rich savanna. It contains over 10,000 species of plants, of which 45% are exclusive to the Cerrado, feeds three of the major water basins in South America: the Amazon, Paraguay and São Francisco Rivers.

Brazilian National Institute for Space Research researchers Costa Wanderson, Leila Fonseca, and Thales Körting report over 500,000km^2 of this pristine and marvelous landscape has been industrialized into cultivated pastures over the past few

years[91]. The rate of change is so rapid the researchers observe with concern that it outpaces even the stunning intensity of commercialization and industrialization in the Amazon region.

The consequence, for the Brazilian Cerrado, as Wanderson, Fonseca, and Körting state is that "*...nearly 50% of the cultivated pasture areas are severely degraded, causing loss of soil fertility, increased erosion and predominance of invasive species.*"

Moderate Resolution Imaging Spectroradiometer data from the Terra and Aqua satellites was collected by the researchers to perform image classification for a small region focused on the Serra da Canastra National Park.

The researchers ended up with a total of 30 input attributes, 23 of which were obtained from spectral bands. Given the number of attributes, the researchers required a technique to reduce the dimensionality. An Autoencoder was developed and successfully used to reduce the dimensionality of the data.

Analysis was performed using a reduced set of 15 attributes and a reduced set of 25 attributes. The researchers conclude "*...using a smaller number of attributes from the autoencoder had accuracies similar to those obtained in the tests that used all original attributes, emphasizing the ability of this network* [the autoencoder] *to reduce the dimensionality of the data.*"

The Essential Ingredient You Need to Know

The key ingredient in an autoencoder are the input attributes. It is important to realize that if the input attributes contain no structure then compression will prove futile. For example, if the input attributes are completely random, compression is unfeasible. Effective compression requires the attributes to be correlated or related in some way. In other words, there needs to be some structure that can be used to decompress the data.

If that structure is not present, using an autoencoder for dimensionality reduction will likely fail.

The Powerful Benefit of the Sparse Autoencoder

A sparse autoencoder uses a large number of hidden neurons, where only a small fraction are active. A nonlinear mapping of the input vector x is made by setting the number of hidden neurons[92] much larger than the number of input neurons[93] and then enforcing a sparsity constraint on them. Training therefore involves using a sparsity constraint to learn sparse representations of the data. The most popular sparsity constraint uses the Kullback-Leibler (KL) divergence.

Understanding Kullback-Leibler Divergence

Kullback- Leibler divergence is a distance measure from a "true" probability distribution to a "target" probability distribution of a Bernoulli random variable with mean p and a Bernoulli random variable with mean $\hat{p}_j$:

$$KL(p||\hat{p}_j) = p \log\left(\frac{p}{\hat{p}_j}\right) + (1-p)\log\left(\frac{1-p}{1-\hat{p}_j}\right)$$

Note that $KL(p||\hat{p}_j) = 0$ for $p = \hat{p}_j$, otherwise it is positive.

The parameter p is the sparsity parameter, which is usually set to a small value. It is the frequency of the activation of hidden nodes; for example, if p =0.07, the average activation of neuron j is 7%.

Let's denote the activation of the hidden neuron j for input attribute x_i by let $a_j^{(2)}(x_i)$ and define:

$$\hat{p}_j = \frac{1}{n}\sum_{i=1}^{n}\left[a_j^{(2)}(x_i)\right]$$

The parameter $\hat{p}_j$ is the average threshold activation of the hidden neuron j over all training samples. In order to compute $\hat{p}_j$, the entire training set needs to be propagated forward to compute all units' activation's; followed by stochastic gradient descent using backpropagation. This makes it computationally expensive relative to the standard autoencoder.

The sparsity optimization objective for unit j is $p = \hat{p}_j$ which is obtained by adding the following sparsity constraint when minimizing the squared reconstruction error or loss function:

$$\alpha\sum_{j} KL(p||\hat{p}_j)$$

where α is a hyperparmeter that determines the relative importance of the sparseness term in the overall autoencoder loss function. To see this note that in general, an autoencoder finds the weights to minimize:

$$\arg\min_{W,b} J(W,b) = L(\hat{x},x)$$

Here L is a loss function such as the squared error or cross-entropy, W the weights and b the biases. For the sparse autoencoder we have:

$$J_{Sparse}(W,b) = J(W,b) + \alpha\sum_{j} KL(p||\hat{p}_j)$$

So we see that the sparsity term adds a positive value to the overall autoencoder loss function for $p \neq \hat{p}_j$.

Three Timeless Lessons from the Sparse Autoencoder

Lesson 1: In practice many more applications are developed using the sparse autoencoder than the standard autoencoder; mainly because a greater variety of models can be learned, depending on the activation function, number of hidden units and nature of the regularization used during training.

Lesson 2: Another rationale for using a sparse encoding for classification is that features that allow for a sparse representation are more likely to encode discriminatory properties of the original input data.

Lesson 3: Finally, if the hidden layer has more neurons units than the input layer, the standard autoencoder can potentially learn the identity function and thereby not extract useful features from the input attributes.

Mixing Hollywood, Biometrics and Sparse Autoencoders

Biometrics techniques occur frequently in Hollywood Movies. You may have noticed this. I am amazed at how frequently this fantastic identification technology rears its head on the big screen. In almost every portrayal of the future, especially in dystopian futures, it plays an important role. Actors can be seen peering into eye scanners that monitor the capillary vessels located at the back of the eye or assess the colored ring that surrounds the eye's pupil. They can be seen waving their hands at sensors to open doors, start up space-craft and fire weapons. From the computer that analyses voice and facial expressions, to the sensitive door that opens when a hand is

placed on the identification pad, biometric techniques are all over the movies.

Scanning live eyeballs with blue lights, passing laser like beams over hands or even digital bulbs flashing manically as a disobedient computer assesses vocal patterns and rotates its beady camera like eye to assess facial expressions, make for added excitement in a movie.

Alas, it seems, finger vein authentication which is real and works, lacks the excitement factor demanded by Hollywood. I suspect this is due to the very limited amount of drama and visual excitement screen playwrights can generate from an individual placing their finger on a small glass plate. This is rather a shame because finger vein authentication is a highly practical, fast, non-invasive means of making sure people really are who they claim to be.

It is part of our present, and will likely persist as a biometric technique well into the future. This may be why Iranian scholar Mohsen Fayyaz and colleagues[94] created an interesting approach to finger vein verification using a sparse autoencoder.

The SDUMLA-HMT Finger Vein database[95] was used as the source of the vein images. This dataset contains 3,816 finger vein images. It was collected as follows - each subject included in the database provided images of his/her index finger, middle finger and ring finger of both hands, and the collection for each of the 6 fingers was repeated 6 times to obtain 6 finger vein images. Vein images were captured using infrared scanner, see Figure 5.2.

Fayyaz et al. apply PCA to reduce the size of input data attributes and a sparse autoencoder with 4000 neurons in the hidden layer and the Kullback- Leibler sparsity constraint. The autoencoder was trained on 3,000 images which excluded images of individual's right hand index finger. These were used for the test sample, a total of 600 images.

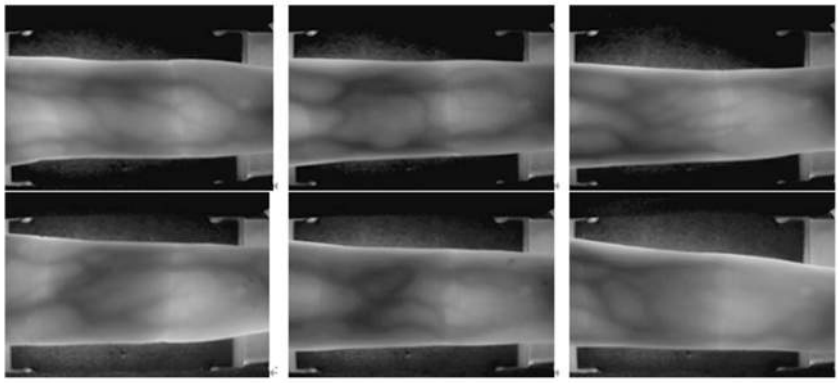


Figure 5.2: Finger Vein images captured by the SDUMLA-HMT database

The performance of the autoencoders was assessed based on the ability of the learned features to separate represented individuals' finger vein images from each other. The equal error rate (EER) and area under the curve (AUC) were the primary performance metrics. The researchers report an EER of 0.70 and an AUC of 99.67. These results handsomely outperformed alternative algorithms proposed by other researchers.

This is all very promising, because as Fayyaz et al. also points out "*Finger veins are situated inside the body and because of this natural property, it is hard to forge and spoof them.*" Well, I bet you can imagine the Hollywood movie: In the last but one screen the good guy disposes of his evil nemesis, hacks of the fallen antagonist's right hand and places the blooded finger on the sensor plate; after a slight click, a long pause, the door slowly opens[96] to freedom...Come on Hollywood this finger vein analysis technique has legs!

How to Immediately use the Autoencoder in R

Enough talking now let's get to work with R! The R logo should be pretty familiar to you. Let's put some of the ideas we have discussed into action and build a sparse autoencoder to compress the R logo image and extract hidden features. First, load the required packages:

```
> require(autoencoder)
> require(ripa)
```

The package `autoencoder` contains the functions we need to build a sparse autoencoder. The `ripa` package contains an image of the R logo. The image is loaded into R as follows:

```
> data(logo)
```

Go ahead, take a look at `logo`, you will see the image shown in Figure 5.3:

```
> image(logo)
```

Let's take a look at the characteristics of `logo`:

```
> logo
size:  77 x 101
type:  grey
```

It is a grey scale image of size 77 by 101 pixels.

Next, make of copy of the image and store in `x_train`. The image is transposed with `t()` to make it suitable for use with the package `autoencoder`.

```
> x_train<-t(logo)

> x_train
size:  101 x 77
type:  grey
```

So, we see `x_train` is a grey scale image with 101 rows (cases) and 77 columns (attributes).

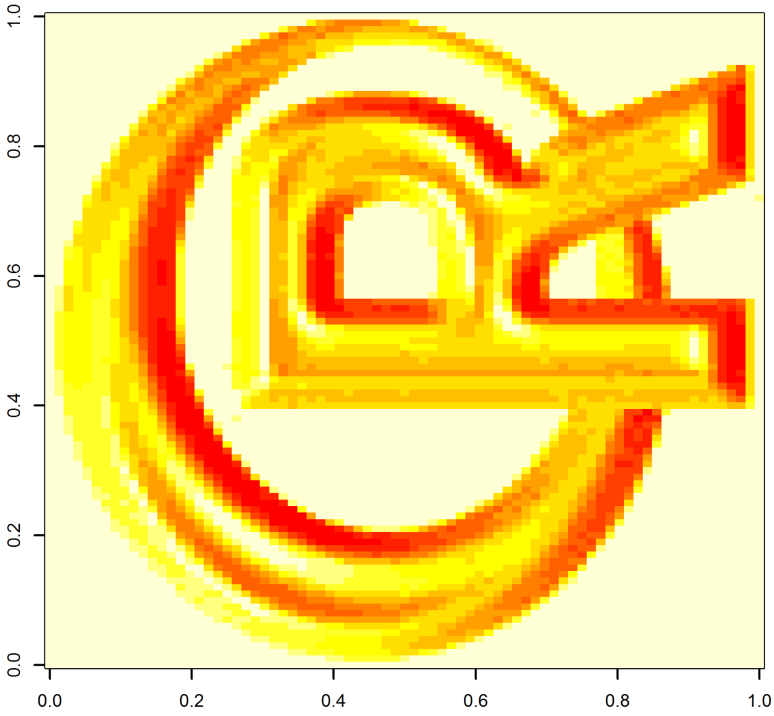


Figure 5.3: The R logo using `image`

Now we are ready to specify a sparse autoencoder using the `autoencode` function. Here is how to do that:

```
> set.seed(2016)

> fit<-autoencode(X.train=x_train,
+ X.test = NULL,
+ nl = 3,
+ N.hidden = 60,
+ unit.type = "logistic",
+ lambda =  1e-5,
+ beta =  1e-5,
```

```
+ rho = 0.3,
+ epsilon =0.1,
+ max.iterations = 100,
+ optim.method = c("BFGS"),
+ rel.tol=0.01,
+ rescale.flag = TRUE,
+ rescaling.offset = 0.001)
```

The first line of `autoencode` indicates the model will be stored in the R object `fit`; it also passes the image data in `x_train` to the function. The parameter `nl` refers to the number of layers and is set to 3. The number of hidden nodes equals 60 with logistic activation functions. The parameter `lambda` is a weight decay parameter, typically set to a small value; the same holds true of `beta`, which is the weight of the sparsity penalty term. The sparsity is set to 0.3 (`rho`) and sampled from a normal distribution N(0,`epsilon`^{2}). The maximum number of iterations is set to 100. Notice that `rescale.flag = TRUE` to uniformly rescale the training matrix `x_train` so its values lie in the range 0-1 (for the logistic activation function).

The attributes associated with fit can be viewed as follows:

```
> attributes(fit)
$names
 [1] "W"                              "b"
 [3] "unit.type"                      "rescaling"
 [5] "nl"                             "sl"
 [7] "N.input"                        "N.hidden"
 [9] "mean.error.training.set" "mean.error.
    test.set"

$class
[1] "autoencoder"
```

Let's take a look at the `"mean.error.test.set"`:

```
> fit$mean.error.training.set
[1] 0.3489713
```

As we have seen previously, it is often useful to extract the features of the hidden nodes. This is easily achieved using the `predict` function with `hidden.output=TRUE`:

```
> features<- predict(fit, X.input=x_train,
   hidden.output=TRUE)
```

Since the number of hidden nodes is set to 60 and the number of attributes (columns) is 77, the features are a compact representation of the original image. A visual representation of the features is shown in Figure 5.4 and can be obtained using:

```
> image(t(features$X.output))
```

The transpose function `t()` is used to re-orientate the features to match Figure 5.3.

NOTE... ✍

The `autoencode` function uses the Nelder–Mead, quasi-Newton and conjugate-gradient algorithms by calling the `optim` function contained in the `stats` package[97]. Available optimisation methods include:

- `"BFGS"` is a quasi-Newton method, it uses function values and gradients to build up a picture of the surface to be optimized.
- `"CG"` is a conjugate gradients method which is generally more fragile than the BFGS method. Its primary advantage is with large optimization problems because it works without having to store large matrices.
- `"L-BFGS-B"` allows each variable to be given a lower and/or upper bound.

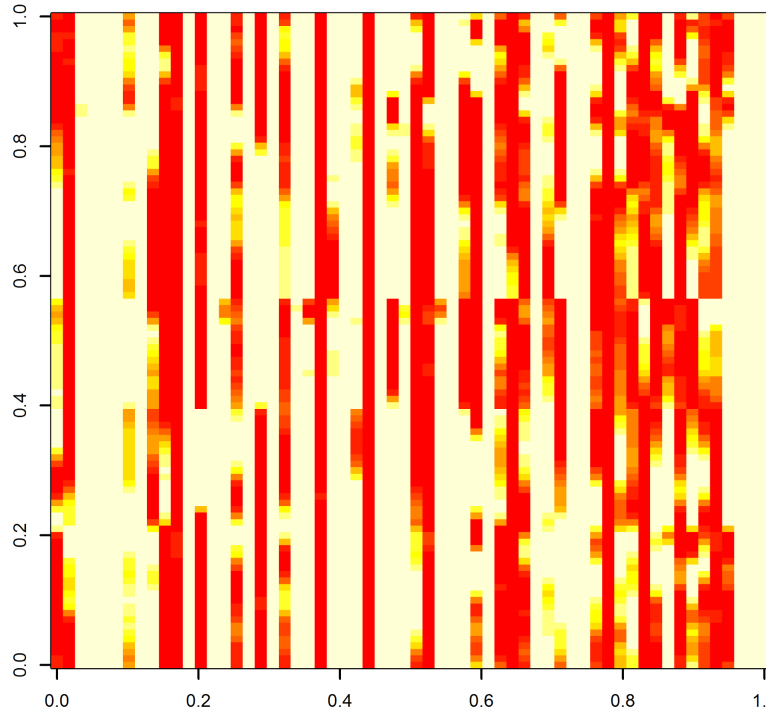


Figure 5.4: Hidden node features extracted from `fit`

How well do these compressed features capture the original image? To reconstruct values use the `predict` function with `hidden.output =FALSE`:

```
> pred<- predict(fit, X.input=x_train,
   hidden.output=FALSE)
```

The mean square error appears reasonably small:

```
> pred$mean.error
[1] 0.3503714
```

Let's take a look at the reconstructed image:

```
> recon<-pred$X.output
> image(t(recon))
```

Figure 5.5 presents the actual and reconstructed image. It appears our sparse autoencoder represents the original image pretty well!

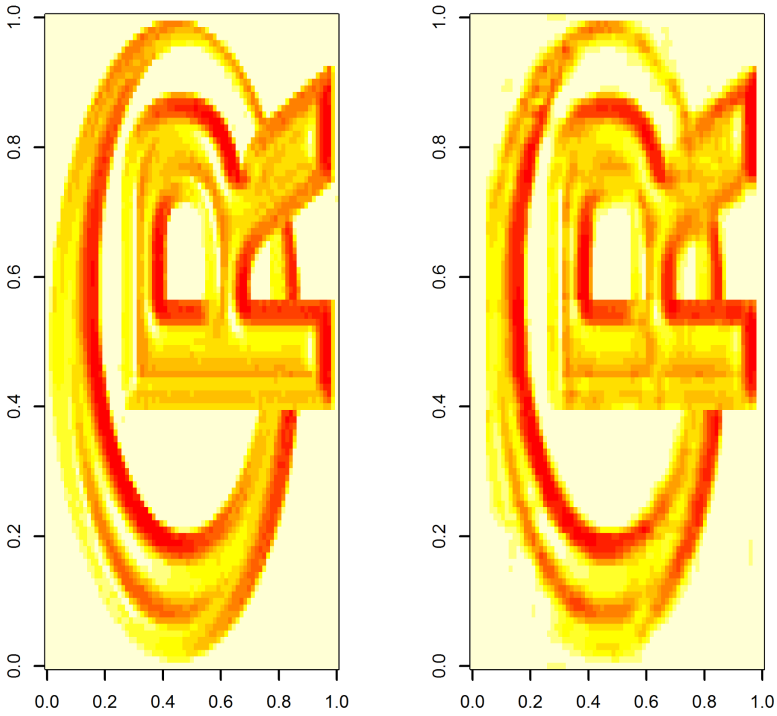


Figure 5.5: Original logo (left) and Sparse Autoencoder reconstruction (right)

An Idea for Your Own Data Science Projects with R

In this section we perform an analysis of an edible mollusk using the autoencoder and R. As you work through this section, think about how you might adapt the approach for you own research.

Abalone are, mostly sedentary, marine snails and belong to a group of sea critters (phylum Mollusca) which include clams, scallops, sea slugs, octopuses and squid. They cling to rocks while waiting for their favorite food of kelp to drift by. Once the seaweed is spotted they hold it down, with what can only be described as a "foot", and chew on it vigorously (for a snail) with radula - a rough tongue with many small teeth.

Given the healthy diet of fresh kelp it is little wonder these mollusks are considered delicious raw or cooked. They are deemed to be so flavorful that the demand for consumption outstripped supply over many years. Today, the white abalone is officially listed as an endangered species. Other species remain plentiful, are harvested regularly to the delight of foodies[98], restaurant goers and seafood chefs[99].

The abalone dataset from UCI Machine Learning Archives was collected to predict abalone age (through the number of rings on the shell) given various attributes such as shell sizes (height, length, width), and weights (shell weight, shucked weight, viscera weight, whole weight). Here is how to capture the web link to the dataset[100] using R:

```
> aburl = 'http://archive.ics.uci.edu/ml/
   machine-learning-databases/abalone/
   abalone.data'
```

The observations are loaded into R using `read.table` and stored in the R object `data`:

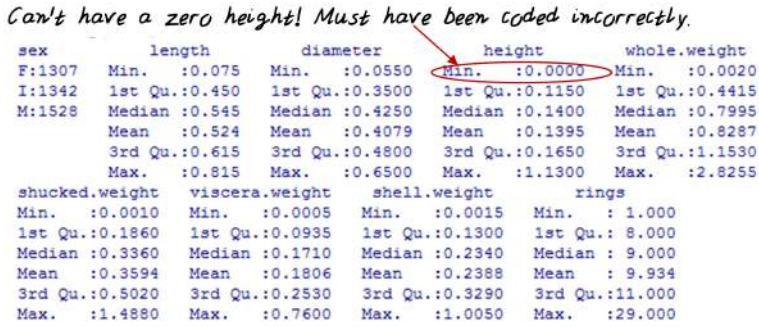
```
> names = c('sex','length',
'diameter',
'height',
```

```
'whole.weight',
'shucked.weight',
'viscera.weight',
'shell.weight','rings')

> data = read.table(aburl,
header = F ,
sep = ',',
col.names = names)
```

To investigate the data for unusual observations I often use the `summary` function. The annotated result is shown below:

```
>summary(data)
```



We can tell instantly there exists a problem with the abalone height; some of the snails are recorded as having 0 height. This is not possible! We will need to investigate further. To take a look at those values enter:

```
> data[data$height==0,]
```

```
     length diameter height whole.weight shucked.weight viscera.weight
1258  0.430     0.34      0        0.428         0.2065         0.0860
3997  0.315     0.23      0        0.134         0.0575         0.0285
     shell.weight rings
1258       0.1150     8
3997       0.3505     6
```

It appears two observations have been erroneously recorded with zero height (observation 1258 and observation 3997). We need to re-code these observations as missing and remove them from sample.

```
> data$height[data$height==0] = NA
> data<-na.omit(data)
```

Next, we drop the sex variable from the sample.

```
> data$sex<-NULL
```

We take another look at the data using `summary`:

```
>summary(data)
```

```
     length          diameter          height        whole.weight
 Min.   :0.0750   Min.   :0.0550   Min.   :0.0100   Min.   :0.0020
 1st Qu.:0.4500   1st Qu.:0.3500   1st Qu.:0.1150   1st Qu.:0.4422
 Median :0.5450   Median :0.4250   Median :0.1400   Median :0.8000
 Mean   :0.5241   Mean   :0.4079   Mean   :0.1396   Mean   :0.8290
 3rd Qu.:0.6150   3rd Qu.:0.4800   3rd Qu.:0.1650   3rd Qu.:1.1535
 Max.   :0.8150   Max.   :0.6500   Max.   :1.1300   Max.   :2.8255
 shucked.weight   viscera.weight    shell.weight        rings
 Min.   :0.0010   Min.   :0.0005   Min.   :0.0015   Min.   : 1.000
 1st Qu.:0.1862   1st Qu.:0.0935   1st Qu.:0.1300   1st Qu.: 8.000
 Median :0.3360   Median :0.1710   Median :0.2340   Median : 9.000
 Mean   :0.3595   Mean   :0.1807   Mean   :0.2388   Mean   : 9.935
 3rd Qu.:0.5020   3rd Qu.:0.2530   3rd Qu.:0.3287   3rd Qu.:11.000
 Max.   :1.4880   Max.   :0.7600   Max.   :1.0050   Max.   :29.000
```

All seems reasonable.

Next, we transpose the data, convert it to a matrix and store the result in the R object `data1`:

```
> data1<-t(data)
> data1<-as.matrix(data1)
```

Now we load the `autoencoder` package, sample 10 observations without replacement and create the basic model which is stored in the R object `fit`:

```
require(autoencoder)

> set.seed(2016)
```

```
> n=nrow(data)
> train <- sample(1:n, 10, FALSE)

> fit<-autoencode(X.train=data1[,train],
+ X.test = NULL,
+ nl = 3,
+ N.hidden = 5,
+ unit.type = "logistic",
+ lambda =  1e-5,
+ beta =  1e-5,
+ rho = 0.07,
+ epsilon =0.1,
+ max.iterations = 100,
+ optim.method = c("BFGS"),
+ rel.tol=0.01,
+ rescale.flag = TRUE,
+ rescaling.offset = 0.001)
```

Much of this we have seen before; the key thing to note is that we fit a model with 5 hidden nodes and a sparsity parameter of 7%.

Once the model is optimized, you should observe a mean square error less than 2%:

```
> fit$mean.error.training.set
[1] 0.01654644
```

The features can be observed, as we saw earlier, by setting `hidden.output=TRUE`. Since the number of hidden nodes is less than the number of features the output of `features$X.output` captures a compressed representation of the data:

```
> features<- predict(fit, X.input=data1[,
   train], hidden.output=TRUE)
> features$X.output
```

	[,1]	[,2]	[,3]	[,4]	[,5]
length	6.572353e-01	7.459814e-01	4.025650e-01	5.306412e-01	6.325756e-01
diameter	7.149880e-01	7.907795e-01	4.670981e-01	5.899761e-01	6.929413e-01
height	8.119831e-01	8.628111e-01	5.986132e-01	7.009952e-01	7.956291e-01
whole.weight	4.901213e-01	6.072550e-01	2.545919e-01	3.770476e-01	4.642603e-01
shucked.weight	7.437931e-01	8.128396e-01	5.023596e-01	6.205615e-01	7.235000e-01
viscera.weight	7.893552e-01	8.464983e-01	5.645063e-01	6.734824e-01	7.718709e-01
shell.weight	7.721268e-01	8.337987e-01	5.403436e-01	6.532020e-01	7.534961e-01
rings	1.855435e-09	1.038285e-08	1.111354e-09	9.623515e-09	1.471309e-09

Let's use the `predict` function to reconstruct the values and store the result in the R object `pred`:

```
> pred<- predict(fit, X.input=data1[,train
  ], hidden.output=FALSE)
```

Figure 5.6, Figure 5.7 and Figure 5.8 use a radar plot to visualize the reconstructed values.

Overall the reconstructed values offer a reasonable representation of the original values. However, notice the apparent poor fit of observation 5 and to a lesser extent observation 6; let's investigate a little further.

The barplot of Figure 5.9 shows the difference between the reconstructed values and observed values. It appears the reconstructed observations under-fit on all eight dimensions (attributes). This is most apparent in observation 5 with rings where the reconstructed rings attribute has a value of 1.4 versus the observed value of 3.5. A similar pattern is observed with observation 6.

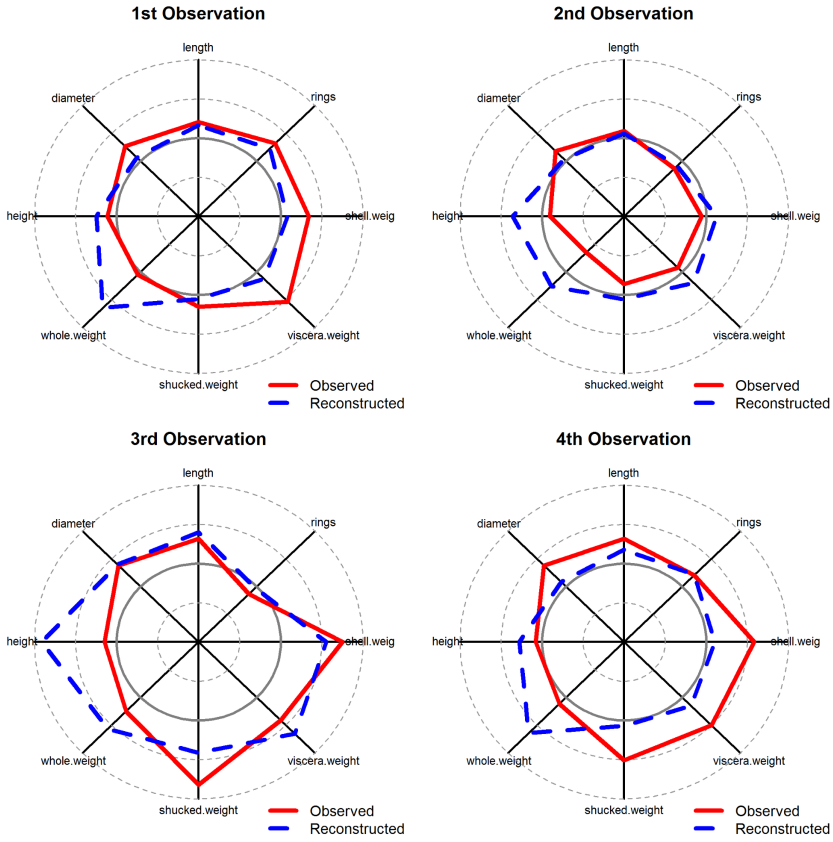


Figure 5.6: Observed and reconstructed values for observations 1 to 4

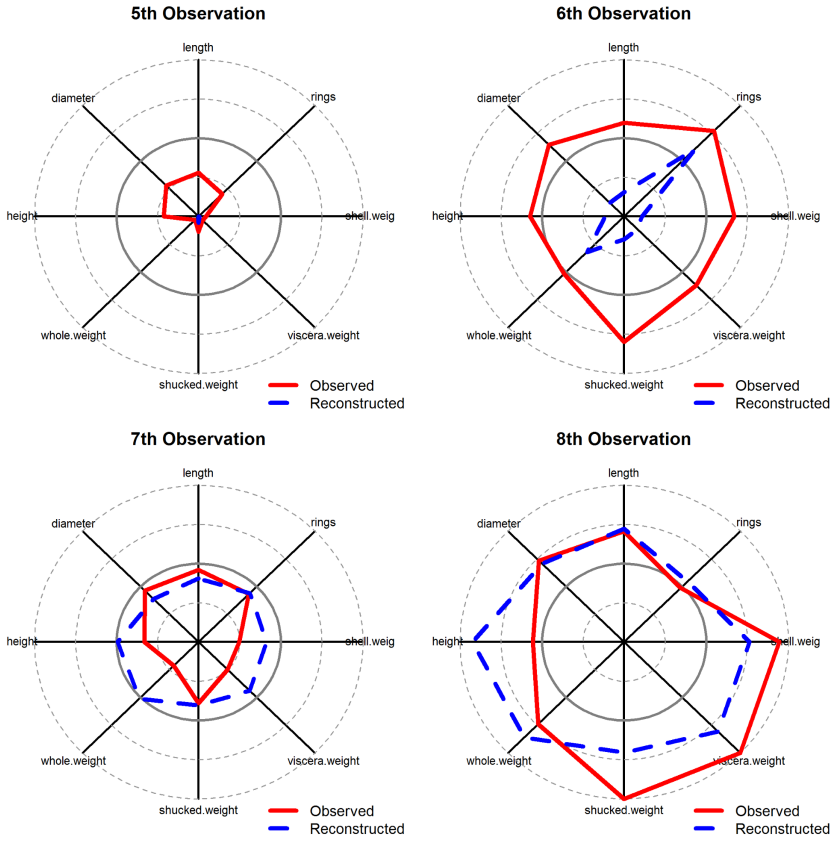


Figure 5.7: Observed and reconstructed values for observations 5 to 8

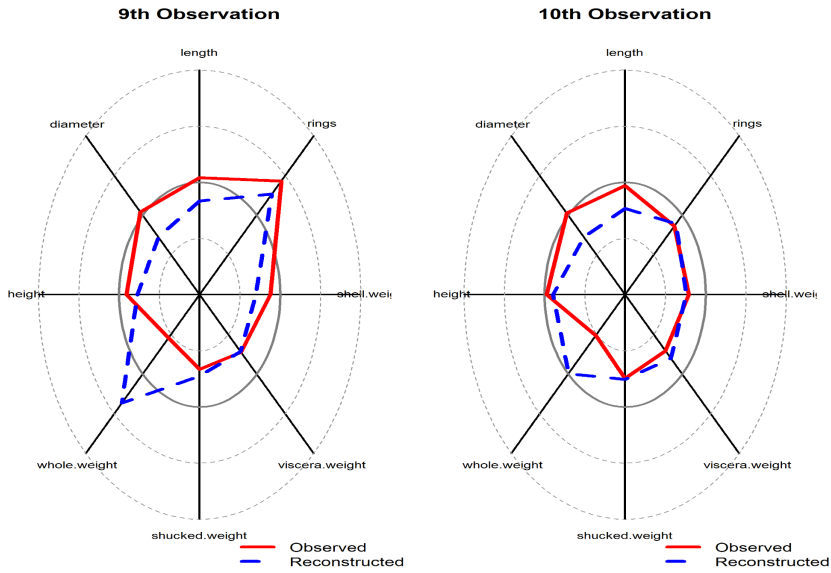


Figure 5.8: Observed and reconstructed values for observations 9 and 10

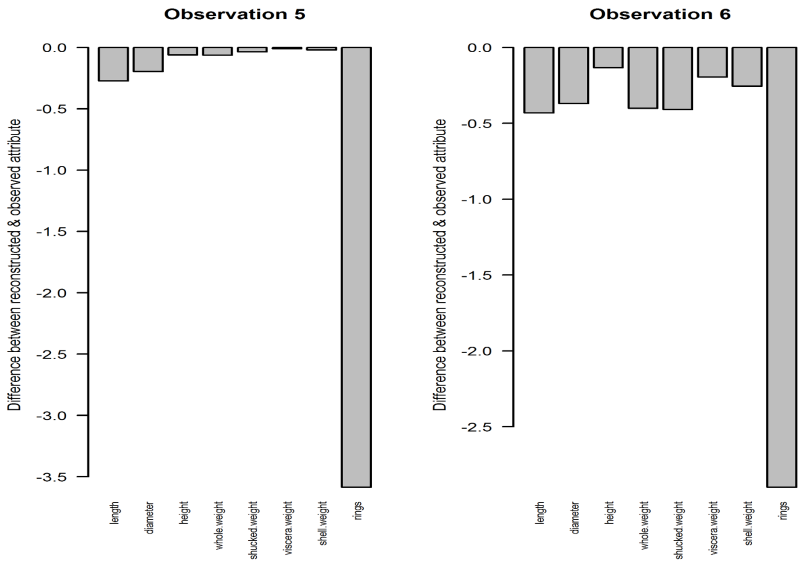


Figure 5.9: Barplot of differences

Notes

[86]I strongly recommend you get a copy and read thoroughly both of the following papers. They are pure solid gold:

- Jolliffe, Ian T. "Discarding variables in a principal component analysis. I: Artificial data." Applied statistics (1972): 160-173.
- Jolliffe, Ian T. "Discarding variables in a principal component analysis. II: Real data." Applied statistics (1973): 21-31.

[87]See:

- Bengio, Yoshua. "Learning deep architectures for AI." Foundations and trends® in Machine Learning 2.1 (2009): 1-127.
- Deng, Li, et al. "Binary coding of speech spectrograms using a deep auto-encoder." Interspeech. 2010.

[88]See Rostislav Goroshin and Yann LeCun. Saturating Auto-Encoders. International Conference on Learning Representations, 2013.

[89]See Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." Science 313.5786 (2006): 504-507.

[90]Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." Science 313.5786 (2006): 504-507.

[91]Costa, Wanderson, Leila Fonseca, and Thales Körting. "Classifying Grasslands and Cultivated Pastures in the Brazilian Cerrado Using Support Vector Machines, Multilayer Perceptrons and Autoencoders." Machine Learning and Data Mining in Pattern Recognition. Springer International Publishing, 2015. 187-198.

[92]The layer of the hidden units is often referred to as the bottleneck.

[93]The situation where the number of hidden neurons is larger than the number of input neurons is often referred to as "overcomplete".

[94]Fayyaz, Mohsen, et al. "A Novel Approach For Finger Vein Verification Based on Self-Taught Learning." arXiv preprint arXiv:1508.03710 (2015).

[95]Developed by Shandong University and available at http://mla.sdu.edu.cn/sdumla-hmt.html

[96]Yes, I am aware that technically this won't actually work. As Fayyaz makes clear "*Another key property of finger vein pattern authentication is the assurance of aliveness of the person, whose biometrics are being proved.*" Hey, but we are talking Hollywood where creative license abounds!

[97]R Core Team (2015). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

[98]For the foodies among you here are a few East Coast (USA) restaurants which serve abalone steaks:

- Allegretto Vineyard Resort Paso Robles 805-369-2500 http://www.ayreshotels.com/allegretto-resort-and-vineyard-paso-robles
- Artisan Paso Robles 805-237-8084 http://artisanpasorobles.com
- Windows on the Water Morro Bay 805-772-0677 http://www.windowsmb.com
- Madeline's Cambria 805-927-4175 http://www.madelinescambria.com
- The Black Cat Cambria 805-927-1600 http://www.blackcatbistro.com
- Linn's Restaurant Cambria 805-927-0371 http://www.linnsfruitbin.com/Linns_Restaurant.html
- Madonna Inn San Luis Obispo 805-543-30

[99]See for example Chef Rick Moonen who states "*We believe in the importance of buying and serving seafood that comes from abundant populations which are under sound management. All fish on our menu are caught or farmed in a way that is not harmful to the ocean environment or to other ocean creatures. We are strong supporters of local fishing communities and take responsibility for our role in preserving a lasting and diverse supply of seafood.*" For more information on great sustainable seafood visit http://rickmoonen.com/

[100]For additional details please see http://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.names

Chapter 6

The Stacked Autoencoder in a Nutshell

The stacked autoencoder (SA) is a deep network with multiple layers. Each layer is an autoencoder in which the outputs of each layer are wired to the inputs of the successive layers. The number of units in the intermediate layers tends to get progressively smaller in order to produce a compact representation.

Figure 6.1 illustrates a stacked autoencoder with two hidden layers for binary classification from the research paper of Jirayucharoensak, Pan-Ngum, and Israsena[101]. In this example, four attributes $\{x_1,x_2,x_3,x_4\}$ are fed into the stacked autoencoder. The nodes with +1 at their center represent the bias. The attribute data is passed through two hidden layers, each containing three nodes. The output layer consists of a softmax activation function which calculates the probability that the output belongs to a specific class (in this case class 0 or class 1). Notice that for classification the softmax layer will usually have one output node per class.

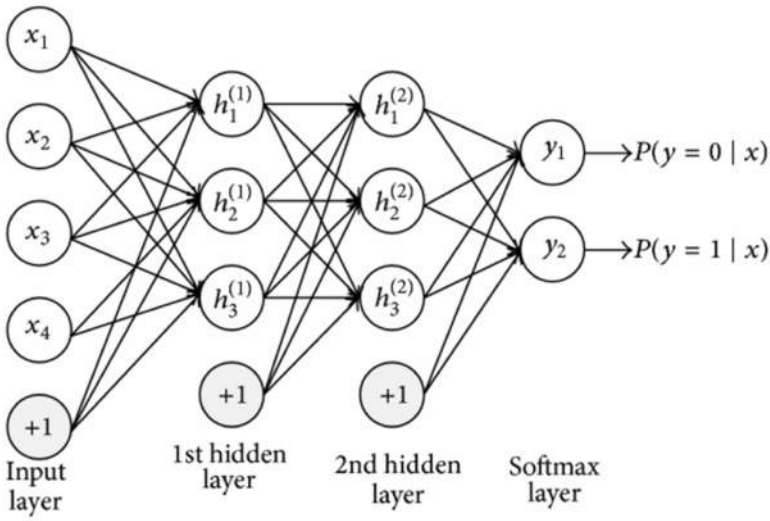


Figure 6.1: Stacked autoencoder with two hidden layers for binary classification. Image source Jirayucharoensak et al. cited in endnote *sec. 101.*

The Deep Learning Guru's Secret Sauce for Training

A few years ago a rather surprising observation emerged from the machine learning research community. It appeared, to many people's surprise, that neural networks with multiple layers were easily trainable, provided one followed a specific technique[102]. These 'deep' neural networks, the subject of this book, actually outperformed single layer models in specific tasks. What was the training process that led to the outperformance and shocked the machine learning community?

In essence the training of a SA or other deep networks for classification tasks proceeds one layer at a time through an unsupervised pre-training phase.

Here is how it works. The first hidden layer is trained on the

input attributes. During this process the weights and bias parameters are optimized using a loss function such as the squared error or cross-entropy.

Next, the algorithm propagates the input attributes through the trained first hidden layer to obtain the primary features. These features are then used to train the next hidden layer. Once trained, the primary features are propagated forward through the trained second layer resulting in a set of secondary features. These features are used as input to train the third layer. Each subsequent layer performs a similar procedure. The last hidden layer passes its trained features to the output layer which, typically using a softmax activation function, calculates the probability of each class.

Finally, the model is fine-tuned by using backpropagation across the entire network to learn weights and biases given labeled training examples. The objective being to minimize the network error, often in terms of the confusion matrix or cross-entropy.

Whilst the above procedure represents the traditional approach to training a SA some researchers have raised doubts on the necessity of the pre-training step[103]. Since data science is a series of failures punctuated by the occasional success, the best advice nearly always, is to experiment.

How Much Sleep Do You Need?

How much sleep do you enjoy each night? How much would you like? Are you getting the recommended amount?
Sleep scholars Webb and Agnew suggest 9 to 10 hours per night for optimal benefit[104]. Are you getting that much?
Don't worry if you are not consistently hitting 10 hours. You see, sleep researcher Jim Horne[105], in his comprehensive treatise, argues between 4.5 to 6 hours per night maximum. Apparently, according to Jim, you can easily adapt to a 5 or 6 hour sleep schedule; anything more than this is "optional", mere

restful "gravy", unnecessary to fulfill your physical need for sleep or prevent the accumulation of that hazy feeling which is the result of sleep deficit. It seems experts in the field, who have studied the phenomenon for years have wildly different suggestions.

The lack of a clear scientific answer has not stopped the popular media from joining the sleep bandwagon. In typical media hype, the issue is no longer about feeling a little tired (or not) when stumbling out of bed in the morning, afternoon or evening (depending on your lifestyle); instead is has somehow mutated into a desperate battle between life and death. The heavyweight Time Magazine reported[106] *"Studies show that people who sleep between 6.5 hr. and 7.5 hr. a night...live the longest. And people who sleep 8 hr. or more, or less than 6.5 hr., they don't live quite as long.*" Shawn Youngstedt, a professor in the College of Nursing and Health Innovation at Arizona State University Phoenix is reported in the Wall Street Journal[107] as noting "*The lowest mortality and morbidity is with seven hours.*" In the same article Dr. Youngstedt, a scholar in the effects of oversleeping, warns "*Eight hours or more has consistently been shown to be hazardous.*" All this adds new meaning to the phrase "Snooze you lose". It seems that extra lie in on Sunday mornings might be killing you.

Not to worry, another, perhaps more sober headed, panel of respected sleep scholars has suggested[108] "*7 to 9 hours for young adults and adults, and 7 to 8 hours of sleep for older adults.*" Whatever is the precise number of hours you get each night or actually need, I think we can all agree that a restful night of calm relaxing sleep can work wonders after a hectic, jam packed day.

According to the American Academy of Sleep Medicine[109] restful sleep is a process which passes through five primary stages. These are rapid eye movement, stages N1, N2, slow wave sleep (N3) and waking up (W). Each specific stage can be scientifically measured by electrical brain activity recorded using a polysomnogram with accompanying hypnograms (expert

annotations of sleep stages) used to identify the sleep stage. This process, as you might expect is manually intensive and subject to human error.

Imperial College, London scholars Orestis Tsinalis, Paul Matthews and Yike Guo develop a SA to automatically score which stage of sleep an individual is in[110]. The present approach to scoring sleep phases is inefficient and prone to error. As Tsinalis, Matthews and Guo explain "*...one or more experts classify each epoch into one of the five stages (N1, N2, N3, R or W) by quantitatively and qualitatively examining the signals of the PSG* [polysomnogram] *in the time and frequency domains.*"

The researchers use an open sleep dataset[111]. The sleep stages were scored by individual experts for 20 healthy subjects, 10 male and 10 female, aged 25–34 years. With a total of around 20 hours of recordings per subject. Because of an imbalance between the response classes (sleep stages) the researchers use a class-balanced random sampling scheme with an ensemble of autoencoder classifiers, each one being trained on a different sample of the data.

The final model consisted of an ensemble of 20 independent SA all with the same hyperparameters. A type of majority voting was used to determine the classification of epochs; the researchers took the mean of the class probabilities from the individual SA's outputs, selecting the class with the highest probability.

The confusion matrix is shown in Figure 6.2. For all sleep stages the classification was correct at least 60% of the time. The most accurate classification of sleep stage occurred for N3 (89%), followed by W (81%). Notice the upper and lower triangle of the confusion matrix are similar to what you might observe in a correlation matrix being almost mirror images of each other. This is an indication that the misclassification errors due to class imbalances have been successfully mitigated.

The researchers also compare their approach to other methods developed in the literature. On every performance metric considered their approach outperforms by a wide margin.

The clear superiority of their results encourages Tsinalis, Matthews and Guo to confidently proclaim "*To the best of our knowledge our method has the best performance in the literature when classification is done across all five sleep stages simultaneously using a single channel of EEG* [Electroencephalography]."

	N1 (algorithm)	N2 (algorithm)	N3 (algorithm)	R (algorithm)	W (algorithm)
N1 (expert)	**1654** (60%)	**262** (9%)	**8** (0%)	**366** (13%)	**472** (17%)
N2 (expert)	**1270** (7%)	**13,696** (78%)	**1231** (7%)	**760** (4%)	**621** (4%)
N3 (expert)	**7** (0%)	**469** (8%)	**4966** (89%)	**6** (0%)	**143** (3%)
R (expert)	**899** (12%)	**340** (4%)	**0** (0%)	**6164** (80%)	**308** (4%)
W (expert)	**441** (13%)	**34** (1%)	**23** (1%)	**138** (4%)	**2744** (81%)

Figure 6.2: Confusion matrix of Tsinalis, Matthews and Guo. *The numbers in bold are numbers of epochs. The numbers in parentheses are the percentage of epochs that belong to the class classified by the expert (rows) that were classified by their algorithm as belonging to the class indicated by the columns. Source of table:* Tsinalis, Matthews and Guo cited in endnote number sec. 110.

Tsinalis, Matthews and Guo have unleashed the power of SA to make major improvements in their field of interest. Who would disagree with the development of automatic classification machines for sleep stage? The deployment and widespread adoption of such machines could be an important step forward for researchers in the entire field of sleep studies. Who knows, maybe it will help to settle the question of "*How much sleep do you need*?"

Build a Stacked Autoencoder in Less Than 5 Minutes

Let's continue with the abalone data used on page 137. We should be able to build a stacked autoencoder pretty quickly. The model can be estimated using the `SAENET.train` function from the package `SAENET`. First we load the data following exactly the same steps as we did on page 137. First, load the packages and download the data from the internet:

```
> require(SAENET)
> aburl = 'http://archive.ics.uci.edu/ml/
   machine-learning-databases/abalone/
   abalone.data'
> names = c('sex','length','diameter','
   height','whole.weight','shucked.weight',
   'viscera.weight','shell.weight','rings')
> data = read.table(aburl, header = F , sep
    = ',', col.names = names)
```

Next, drop the gender attribute, delete the observations with miscoded height, and store the resultant sample as a matrix in the R object `data1`:

```
> data$sex<-NULL
> data$height[data$height==0] = NA
> data<-na.omit(data)
> data1<-as.matrix(data)
```

Now onto the sample. Again, for illustration we will only sample 10 observations:

```
> set.seed(2016)
> n=nrow(data)
> train <- sample(1:n, 10, FALSE)
```

Now we are ready to estimate the model. In this case we fit a model with 3 hidden layers with 5, 4 and 2 nodes respectively {`n.nodes = c(5,4,2)`}. The remainder of the following code

is familiar to you by now. The model is stored in the R object `fit`:

```
> fit<-SAENET.train(X.train=data1[train,],
+ n.nodes = c(5,4,2),
+ unit.type = "logistic",
+ lambda =  1e-5,
+ beta =  1e-5,
+ rho = 0.07,
+ epsilon =0.1,
+ max.iterations = 100,
+ optim.method = c("BFGS"),
+ rel.tol=0.01,
+ rescale.flag = TRUE,
+ rescaling.offset = 0.001)
```

The output from each layer can be viewed by typing `fit[[n]]$X.output` where `n` is the layer of interest. For example, to see the output from the two nodes in the third layer:

```
> fit[[3]]$X.output
          [,1]      [,2]
753  0.4837342 0.4885643
597  0.4837314 0.4885684
3514 0.4837309 0.4885684
558  0.4837333 0.4885653
1993 0.4837282 0.4885726
506  0.4837351 0.4885621
2572 0.4837315 0.4885684
3713 0.4837321 0.4885674
11   0.4837346 0.4885632
223  0.4837310 0.4885684
```

Figure 6.3 plots the outputs by observation, where for clarity item 753 is labeled as observation 1, item 597 is labeled as observation 2, ..., and item 223 is labeled as item 10.

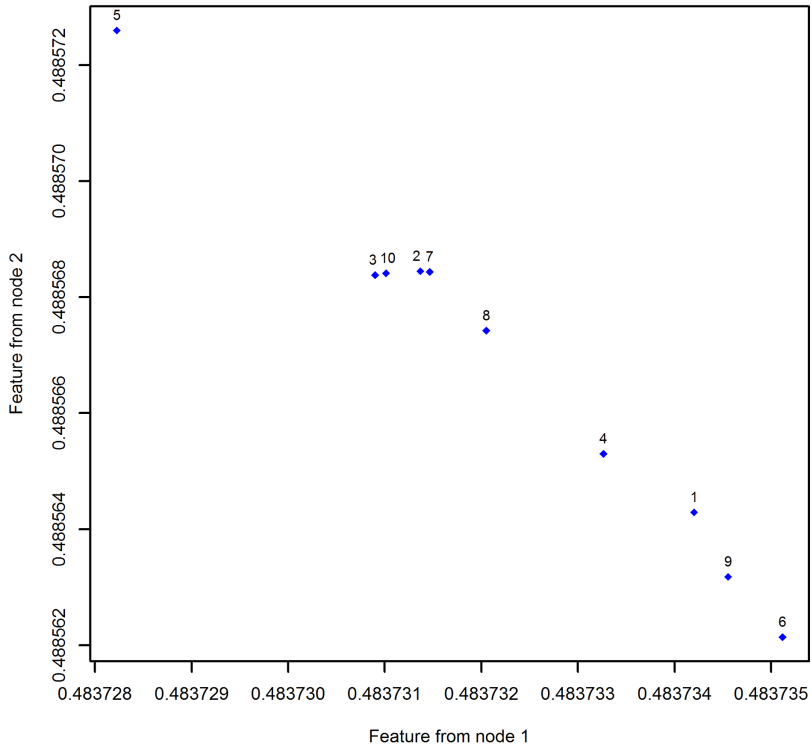


Figure 6.3: Visual representation of the features from the 3rd layer

What is a Denoising Autoencoder?

A denoising autoencoder (DA), as with the regular autoencoder, contains three layers; an input layer, hidden layer, and output layer; where the hidden layer is the encoder and the output layer the decoder. What distinguishes a denoising autoencoder from it's traditional cousin is the DA adds noise ran-

domly to the input during training. It is a stochastic variant of autoencoder. The addition of noise prevents the hidden layer from simply learning the identity function and forces it to uncover more robust features.

The Salt and Pepper of Random Masking

Given the input attribute vector x, the DA produces a new "noisy" attribute vector $\tilde{x}$. Whilst there are many ways to introduce noise, one of the most frequently used is random masking. In random masking inputs are randomly set equal to zero. Notice this idea is akin to dropout where a random number of neurons are ignored during each training round. As we saw on page 51 dropout can significantly improve performance.

A variant, often referred to as the addition of salt and pepper noise, randomly forces a fraction of x to be 0 or 1. Another popular method involves the addition of Gaussian noise to the attribute vector x. Whichever method of noise is used the DA is then trained to reconstruct the input attributes from the noisy version.

The Two Essential Tasks of a Denoising Autoencoder

Intuitively the DA attempts two things:

1. First, encode the input attribute vector x;
2. second, remove the effect of the error contained in the noisy attribute vector $\tilde{x}$.

These tasks can only be successfully achieved if the DA captures the statistical dependencies between the input attributes. In practice, a DA often achieves a much better representation of the input attributes than the standard autoencoder[112].

How to Understand Stacked Denoising Autoencoders

Denoising autoencoders can be stacked to form a deep learning network. Figure 6.4 shows a stacked denoising autoencoder (SDA) with four hidden layers.

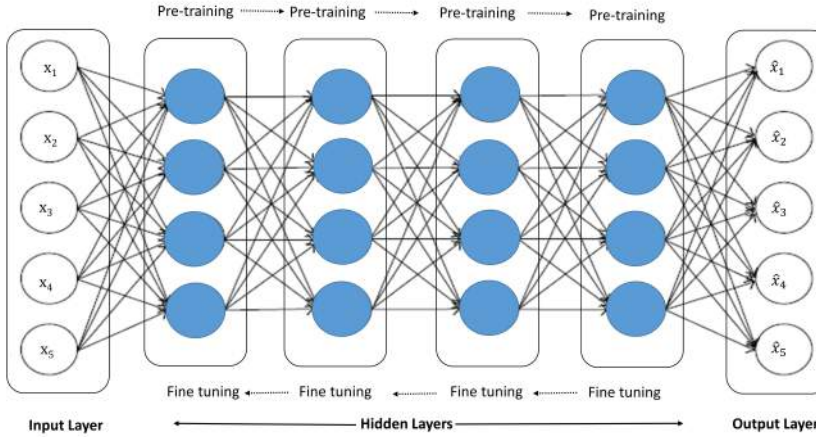


Figure 6.4: Stacked Denoising Autoencoder with four hidden layers

In a stacked denoising autoencoder (SDA) pre-training is done one layer at a time in the same way as discussed for a stacked autoencoder (see page 148). Over time it has become clear that pre-training often encourages the network to discover a better parameter space during the back-propagation phase.

In the pre-training phase each layer is trained as a denoising autoencoder by minimizing the reconstruction error. For each layer, feature extraction is carried out with the extracted hidden representation treated as the input to the next hidden layer. After the final pre-training process, the last hidden layer is classified typically with a softmax activation function and the resulting vector passed to the output layer.

Once all layers are pre-trained, the network goes through a second stage of training, often referred to as fine tuning, via supervised learning. At this stage the prediction error is minimized by backpropagation using the entire network as we would train a multilayer perceptron.

Stacking denoising autoencoders appears to improve a networks representation ability. The more layers that are added, the more abstract are the features extracted[113].

A Stunningly Practical Application

Imaging spectroscopy has been used in the laboratory by physicists and chemists for well over 100 years to identify materials and their composition. It's popularity in the laboratory comes from its ability to detect individual absorption features due to specific chemical bonds in a solid, liquid, or gas. Hyperspectral remote sensing takes what was once only possible in the darkened recesses of a few scientist's laboratory and applies it to the entire Earth.

Hyperspectral images are spectrally over-determined providing sufficient spectral information to identify and distinguish between spectrally similar but distinct materials. Hyperspectral sensors typically measure reflected radiation as a series of narrow and contiguous wavelength bands.

It is a technology with amazing potentially to provide greatly improved discriminant capability for the detection and identification of minerals, terrestrial vegetation, land cover classification, and in fact identification of any of a stunning array of man-made or nature produced materials. With advancing technology, imaging spectroscopy has increasingly been focused on analyzing the Earth's surface.

As you might hazard a guess, geologists were quick to latch onto this technology for the identification and mapping of potential mineral deposits. They soon discovered that the actual detection of economically exploitable minerals is dependent on

a complex combination of factors including:

- The range of the spectral coverage;
- available spectral resolution,
- signal-to-noise ratio of the spectrometer,
- the actual abundance of the mineral sought;
- and the strength of absorption features for that material in the wavelength region measured.

It is difficult to think of an area in industry, commerce or government where this technology could not be of tremendous benefit.

Deep learning techniques have a role to play in this booming and amazingly lucrative area. Think for a moment, how could you use deep learning to help out here?

Take a look at Figure 6.5, the answer should become immediately obvious to you. It shows a hyperspectral data cube from the innovative research of Möckel et al.[114]. These innovative scholars use hyperspectral data to discriminate between grazed vegetation belonging to different grassland successional stages. Now here is the thing, Möckel et al are essentially interested in a classification task. Guess what? SDA was designed for this very purpose.

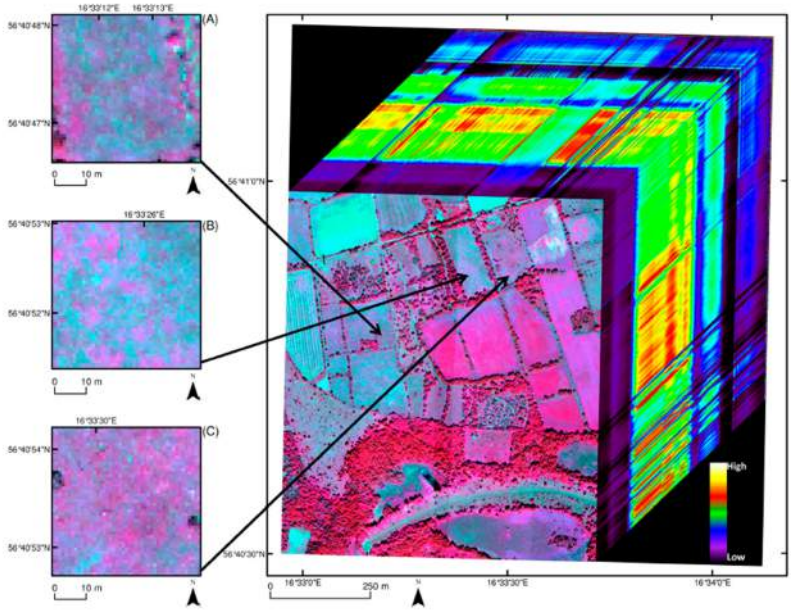


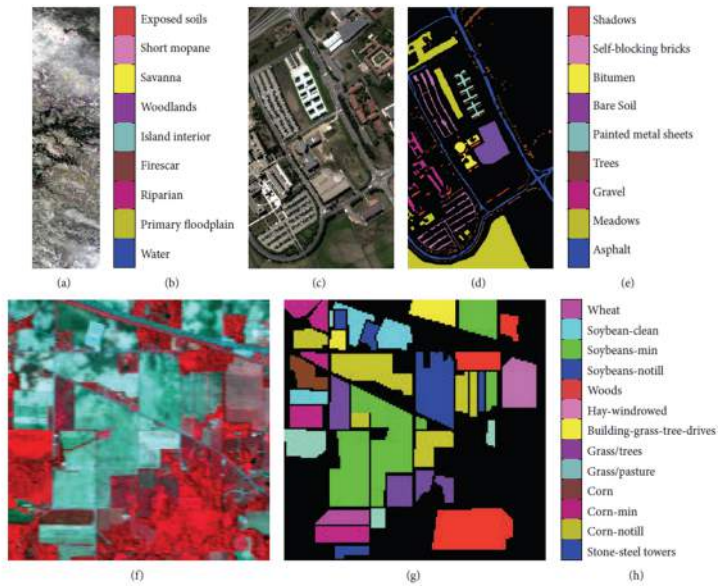
Figure 6.5: Hyperspectral data cube showing examples of data associated with (A) young; (B) intermediate-aged; and (C) old grassland sites within a subarea of 0.71 km2 within the Jordtorp study area. The color-composite on the top was obtained using three hyperspectral wavebands (RGB = 861 nm, 651 nm, and 549 nm). Image source: Möckel, Thomas, et al. as cited in endnote sec. 114.

An Innovative Idea

The potential for useful deployment of SDA for the classification of hyperspectral data is truly outstanding; a fact not missed by the trio Chen, Li, and Xiaoquan, scholars at China University of Geosciences and Huazhong University of Science and Technology[115]. This trio of researchers unleash the power of SDA for feature extraction and classification of hyperspectral data. Their study, small and experimental, uses three images created from hyperspectral data. The first image was taken

over Indian Pine (INP); the second over the Okavango Delta, Botswana (BOT); and the third over the University of Pavia (PU), Italy.

Figure 6.6 shows the RGB images and the ground referenced information with class legends for each of the images. Table 4 shows the class information of the three image datasets, the number of classes in each image and the number of labeled samples in each class.



Three band false color composite and ground references. (a) False color composite of BOT image with ground reference. (b) Class legend of BOT image. (c) False color composite of PU image. (d) Ground reference of PU image. (e) Class legend of PU image. (f) IND PINE scene. (g) Ground reference of IND PINE image. (h) Class legend of IND PINE image.

Figure 6.6: Three band false color composite and ground classification references. Source of image Chen Xing, Li Ma, and Xiaoquan Yang as cited in endnote sec. 115.

BOT		INP		PU	
ID	Class Name	ID	Class Name	ID	Class Name
1	Water (158)	1	Stone-steel Towers (95)	1	Asphalt (6631)
2	Floodplain (228)	2	Corn-notill (1434)	2	Meadows (18649)
3	Riparian (237)	3	Corn-min (834)	3	Gravel (2099)
4	Firescar (178)	4	Corn (234)	4	Trees (3064)
5	Island Interior (183)	5	Grass/Pasture (497)	5	Painted metal Sheets (1435)
6	Woodlands (199)	6	Grass/Trees (747)	6	Bare Soil (5029)
7	Savanna (162)	7	Building-Grass-Tree-Drives (380)	7	Bitumen (1330)
8	Mopane (124)	8	Hay-windrowed (489)	8	Self-Blocking Bricks (3682)
9	Exposed Soils (111)	9	Woods (1294)	9	Shadows (947)
		10	Soybeans-notill (968)		
		11	Soybeans-min (2468)		
		12	Soybean-clean (614)		
		13	Wheat (212)		

Table 4: Class information for each image taken from the research paper of Chen, Li , and Xiaoquan.Source of table Chen Xing, Li Ma, and Xiaoquan Yang as cited in endnote sec. 115.

Let's take a deep dive into some of the details of the model. The first thing to observe is that the model consists of an SDA for feature extraction and logistic regression with sigmoid activation functions (LR) for fine-tuning and classification. Therefore the model, as shown in Figure 6.7, consists of a an unsupervised element (the SDA) and a supervised element (LR).

The second thing to observe is the parameters used in the model are image specific. Three images equal three sets of model parameters; this is akin to the parameters in a simple linear regression being determined by the data. The slope and intercept terms can be expected to be different for distinctly different datasets. In a similar way, different images may require different parameter settings and therefore Chen, Li, and Xiaoquan calibrated their SDA to each specific image.

The number of hidden layers was 4, 3, and 3 for BOT, PU, and INP, respectively. The number of nodes was selected as 100, 300, and 200, for BOT, PU, and INP respectively. The standard deviation of Gaussian noise was also image specific; the researchers selected values of 0.6 for BOT/ PU, and 0.2 for INP. In addition, the number of epochs used during pretraining was set to 200 for all images; with the number of fine tuning epochs equal to 1500 for all images. The learning rates

of pretraining and fine tuning were selected as 0.01 and 0.1 respectively.

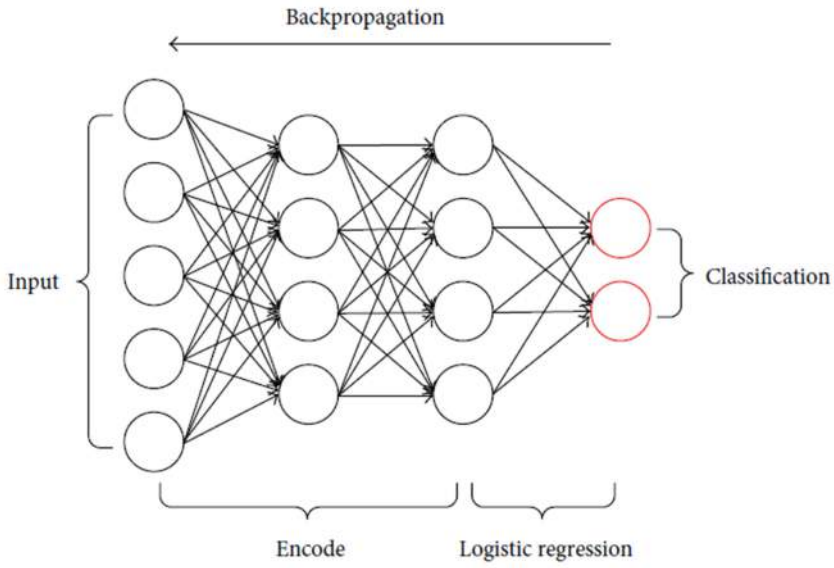


Figure 6.7: Architecture of Chen, Li , and Xiaoquan's deep learning model. Source of image Chen Xing, Li Ma, and Xiaoquan Yang as cited in endnote sec. 115.

How Chen, Li, and Xiaoquan Train their Model

Three steps are followed by the researchers in order to train their model.

- **Step 1:** Initial network weights are obtained by the SDA encoding component of the model.
- **Step 2:** The initial weights of the LR layer are set randomly.

- **Step 3:** Training data are used as input data, and their predicted classification results are produced with the initial weights of the whole network.

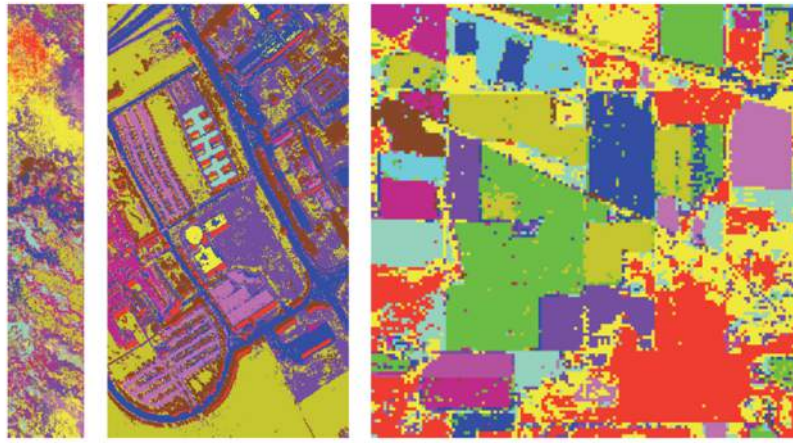
- **Step 4:** Network weights are iteratively tuned using backpropagation.

How to Avoid the Sirens Song

Figure 6.8 presents the images reconstructed from the optimal models. I think you will agree on causal observation they appear to be a pretty good representation of the actual classifications. This is a great start. However, as statistician George Box warned "*Statisticians, like artists, have the bad habit of falling in love with their models*." Falling in love with a model is very dangerous. The data scientist must exhibit the qualities of Odysseus in Homer's Odyssey, no matter how beautiful a model is to you. do not be seduced by the Sirens song. Always remember that in data science there are numerous ways to achieve the same end result.

It is always a good idea to benchmark a model against competing alternatives, even experimental or toy models. This is an important step towards objective professionalism. It will keep you safe from deep potholes, especially those strewn along the rocky road of model deployment for real life use.

Chen, Li, and Xiaoquan compare the classification ability of their model (CLX) to support vector machines with linear (LSVM) and Radial basis function (RSVM) kernels. Table 5 presents the results in terms of classification accuracy for all three models. The key thing to notice is that CLX outperformed both support vector machine for images PU and IMP. However, it under-performed RSVM for BOT.



Classification results of the whole image on BOT (a), PU (b), and INP (c) data set.

Figure 6.8: Classification results of model developed by Chen, Li , and Xiaoquan. Source Chen, Li , and Xiaoquan. Source of image Chen Xing, Li Ma, and Xiaoquan Yang as cited in endnote sec. 115.

Data	LSVM	RSVM	CLX
BOT	92.88	**96.88**	95.53
PU	80.11	93.62	**95.97**
INP	76.15	90.63	**92.06**

Table 5: Performance results of LSVM, RSVM and CLX. Adapted from Chen, Li, and Xiaoquan. Source of table Chen Xing, Li Ma, and Xiaoquan Yang as cited in endnote sec. 115.

A Challenge to You from the Author

What are we to make of this? Well, have I ever told you that *Data Science is a series of failures punctuated by the occasional*

success? Chen, Li, and Xiaoquan have two successes and it all looks rather promising. Like Alexander Graham Bell, the pioneer of the telephone, speaking into his electronic apparatus "*Watson, come here! I want to see you*!"; and then hearing his voice miraculously transmitted through wires over a short-range receiver.

NOTE... ✍

Alexander Graham Bell was first and foremost an astute businessman, it was not long before a patent for an "*apparatus for transmitting vocal or other sounds telegraphically*" was filed with the U.S. patent office[116]. It begins "*Be it known that I, ALEXANDER GRAHAM BELL,of Salem, Massachusetts, have invented certain new and useful Improvements in Telegraphy.*"

Chen, Li, and Xiaoquan's project was experimental but so stunningly successful that further refinement and commercial applications are sure to follow. Other astute individuals will harness the power of the SDA to develop useful systems and follow Mr. Bell into the history books. Will one of them be you?

The Fast Path to a Denoising Autoencoder in R

Researchers Huq and Cleland[117], as part of a regular survey on the fertility of women in Bangladesh, collected data on mobility of social freedom. The sub-sample measured the response of 8445 rural women to questions about whether they could engage in certain activities alone (see Table 6).

Name	Description
Item 1	Go to any part of the village.
Item 2	Go outside the village.
Item 3	Talk to a man you do not know.
Item 4	Go to a cinema or cultural show.
Item 5	Go shopping.
Item 6	Go to a cooperative or mothers' club.
Item 7	Attend a political meeting.
Item 8	Go to a health centre or hospital.

Table 6: Variables used by Huq and Cleland to measure women's mobility of social freedom

Let's use this sample to build our autoencoder. The data is contained in the R object `Mobility,` from the package `ltm`. We use it and the package `RcppDL`:

```
>  require(RcppDL)
>   require("ltm")
>   data(Mobility)
> data<-Mobility
```

Next, we set up the data; in this example we sample 1,000 observations without replacement from the original 8445 responses. A total of 800 responses are used for the training set, with the remaining 200 observations used for the test set:

```
> set.seed(17)
> n=nrow(data)
> sample <- sample(1:n, 1000, FALSE)
> data <-as.matrix(Mobility[sample,])
> n=nrow(data)
> train <- sample(1:n, 800, FALSE)
```

Now to create the attributes for the training sample and test sample:

```
> x_train <-matrix(as.numeric(unlist(data[
   train,])),nrow=nrow(data[train,]))
> x_test<-matrix(as.numeric(unlist(data[-
   train,])),nrow=nrow(data[-train,]))
```

Need to check to ensure we have the correct sample sizes. The train set should equal 800 observations, and 200 for the test set:

```
> nrow(x_train)
[1] 800


> nrow(x_test)
[1] 200
```

All looks good. Now we can remove the response variable from the attributes R objects. In this example we will use item 3 (*Talk to a man you do not know*) as the response variable. Here is how to remove it from the attribute objects:

```
> x_train<-x_train[,-3]
> x_test<-x_test[,-3]
```

The training and attribute R objects should now look something like this:

```
> head(x_train)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    1    1    0    0    0    0
[2,]    1    1    0    0    0    0    0
[3,]    1    1    1    0    0    0    0
[4,]    0    0    0    0    0    0    0
[5,]    1    0    0    0    0    0    0
[6,]    1    0    0    0    0    0    0
> head(x_test)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0    0    0    0    0    0    0
[2,]    0    0    0    0    0    0    0
[3,]    0    0    0    0    0    0    0
[4,]    1    0    0    0    0    0    0
```

```
[5,]    0    0    0    0    0    0    0
[6,]    1    0    1    0    0    0    0
```

Next, we prepare the response variable for use with the `RcppDL` package. The denoising autoencoder is built using the `Rsda` function from this package. We will pass the response variable to it using two columns. First, create the response variable for the training sample:

```
> y_train<-data[train,3]
> temp<-ifelse(y_train==0, 1, 0)
> y_train<-cbind(y_train,temp)
```

Take a look at the result:

```
> head(y_train)
     y_train temp
1405       1    0
3960       0    1
3175       1    0
7073       1    0
7747       1    0
8113       1    0
```

And check we have the correct number of observations:

```
> nrow(y_train)
[1] 800
```

We follow the same procedure for the test sample:

```
> y_test<-data[-train,3]
> temp1<-ifelse(y_test==0, 1, 0)
> y_test<-cbind(y_test,temp1)

> head(y_test)
     y_test temp1
3954      1     0
1579      0     1
7000      0     1
4435      1     0
```

```
7424       1      0
6764       1      0

> nrow(y_test)
[1] 200
```

Now we are ready to specify our model. Let's first build a stacked autoencoder without any noise. We will use two hidden layers each containing ten nodes:

```
> hidden = c(10,10)
> fit <- Rsda(x_train, y_train, hidden)
```

The default noise level for `Rsda` is 30%. Since, we want to begin with a regular stacked autoencoder we set the noise to 0. Here is how to do that:

```
> setCorruptionLevel (fit, x = 0.0)

> summary(fit)
$PretrainLearningRate
[1] 0.1

$CorruptionLevel
[1] 0

$PretrainingEpochs
[1] 1000

$FinetuneLearningRate
[1] 0.1

$FinetuneEpochs
[1] 500
```

NOTE... ✍

You can set a number of the parameters in `Rsda`. We used something along the lines of:

```
setCorruptionLevel (model,x)
```

You can also choose the number of epochs and learning rates for both fine tuning and pretraining:

- `setFinetuneEpochs`
- `setFinetuneLearningRate`
- `setPretrainLearningRate`
- `setPretrainEpochs`

The next step is to pretrain and fine tune the model. This is fairly straight forward:

```
> pretrain(fit)
> finetune(fit)
```

Since the sample is small the model converges pretty quickly. Let's take a look at the predicted probabilities for the response variable using the test sample:

```
> predProb<-predict(fit, x_test)

> head(predProb,6)
          [,1]      [,2]
[1,] 0.4481689 0.5518311
[2,] 0.4481689 0.5518311
[3,] 0.4481689 0.5518311
[4,] 0.6124651 0.3875349
[5,] 0.4481689 0.5518311
```

```
[6,] 0.8310412 0.1689588
```

So, we see for the first three observations the model predicts approximately a 45% probability that they belong to class 1 and 55% that they belong to class 2. Let's take a peek to see how it did:

```
> head(y_test,3)
     y_test temp1
3954      1     0
1579      0     1
7000      0     1
```

It was missed the first observation! However, it classified the second and third observations correctly. Finally, we construct the confusion matrix:

```
> pred1<-ifelse(predProb[,1]>=0.5, 1, 0)

> table( pred1,y_test[,1] ,
dnn =c("Predicted"
, " Observed"))

          Observed
Predicted   0   1
        0  15  15
        1  36 134
```

Next, we rebuild the model, this time adding 25% noise:

```
> setCorruptionLevel (fit, x = 0.25)
> pretrain(fit)
> finetune(fit)
> predProb<-predict(fit, x_test)
> pred1<-ifelse(predProb[,1]>=0.5, 1, 0)

> table( pred1,y_test[,1] ,
dnn =c("Predicted" ,
" Observed"))
          Observed
```

```
Predicted    0    1
        0   15   15
        1   36  134
```

It appears to give us the same confusion matrix as a stacked autoencoder without any noise. So in this case, adding noise was not of much benefit.

Notes

[101]See Suwicha Jirayucharoensak, Setha Pan-Ngum, and Pasin Israsena, "EEG-Based Emotion Recognition Using Deep Learning Network with Principal Component Based Covariate Shift Adaptation," The Scientific World Journal, vol. 2014, Article ID 627892, 10 pages, 2014. doi:10.1155/2014/627892

[102]See G. E. Hinton, S. Osindero, and Y. W. Teh, "A fast learning algorithm for deep belief nets," Neural Computation, vol. 18, no. 7, pp. 1527–1554, 2006.

[103]See for example:

Pugh, Justin K., Andrea Soltoggio, and Kenneth O. Stanley. "Real-time hebbian learning from autoencoder features for control tasks." (2014).

Cireşan, Dan, et al. "Multi-column deep neural network for traffic sign classification." Neural Networks 32 (2012): 333-338.

[104]Webb, W. B., and H. W. Agnew Jr. "Are we chronically sleep deprived?." Bulletin of the Psychonomic Society 6.1 (1975): 47-48.

[105]Horne, James. Why we sleep: the functions of sleep in humans and other mammals. Oxford University Press, 1988.

[106]How Much Sleep Do You Really Need? By Laura Blue Friday, June 06, 2008.

[107]Why Seven Hours of Sleep Might Be Better Than Eight by Sumathi Reddy July 21, 2014.

[108]Hirshkowitz, Max, et al. "National Sleep Foundation's sleep time duration recommendations: methodology and results summary." Sleep Health 1.1 (2015): 40-43.

[109]See http://www.aasmnet.org/

[110]Tsinalis, Orestis, Paul M. Matthews, and Yike Guo. "Automatic Sleep Stage Scoring Using Time-Frequency Analysis and Stacked Sparse Autoencoders." Annals of biomedical engineering (2015): 1-11.

[111]See:

- Goldberger, Ary L., et al. "Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals." Circulation 101.23 (2000): e215-e220.
- Also visit https://physionet.org/pn4/sleep-edfx/

[112]See:

- Bengio, Yoshua, et al. "Greedy layer-wise training of deep networks." Advances in neural information processing systems 19 (2007): 153.

- Vincent, Pascal, et al. "Extracting and composing robust features with denoising autoencoders." Proceedings of the 25th international conference on Machine learning. ACM, 2008.

[113]See Vincent, Pascal, et al. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." The Journal of Machine Learning Research 11 (2010): 3371-3408.

[114]Möckel, Thomas, et al. "Classification of grassland successional stages using airborne hyperspectral imagery." Remote Sensing 6.8 (2014): 7732-7761.

[115]See Chen Xing, Li Ma, and Xiaoquan Yang, "Stacked Denoise Autoencoder Based Feature Extraction and Classification for Hyperspectral Images," Journal of Sensors, vol. 2016, Article ID 3632943, 10 pages, 2016. doi:10.1155/2016/3632943

[116]Graham, Bell Alexander. "Improvement in telegraphy." U.S. Patent No. 174,465. 7 Mar. 1876.

[117]Huq, N. and Cleland, J. (1990) Bangladesh Fertility Survey, 1989. Dhaka: National Institute of Population Research and Training (NIPORT).

Chapter 7

Restricted Boltzmann Machines

We are drowning in information and starving for knowledge.

Rutherford D. Roger

The Restricted Boltzmann Machine (RBM) is an unsupervised learning model that approximates the probability density function of sample data. The "restricted" part of the name points to the fact that there are no connections between units in the same layer. The parameters of the model are learned by maximizing the likelihood function of the samples. Since it is used to approximate a probability density function it is often referred to in the literature as a generative model[118]. Generative learning involves making guesses about the probability distribution of the original input in order to reconstruct it.

The Four Steps to Knowledge

Figure 7.1 shows a graphical representation of an RBM. It is composed of two layers in which there are a number of units

with inner layer connections. Connections between layers are symmetric and bidirectional, allowing information transfer in both directions. It has one visible layer containing four nodes and one hidden layer containing three nodes. The visible nodes are related to the input attributes so that for each unit in the visible layer, the corresponding attribute value is observable.

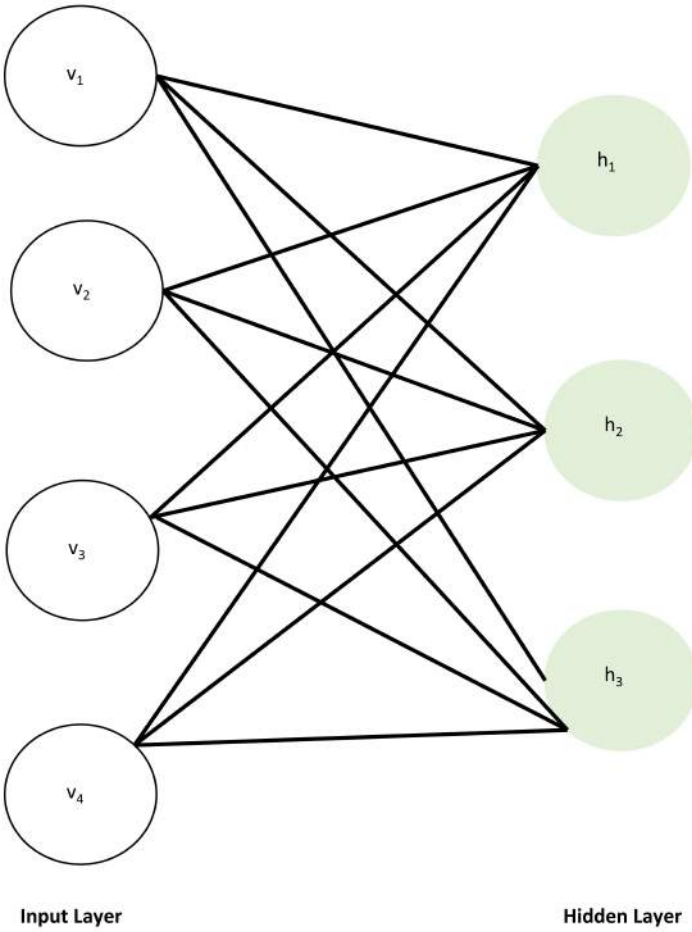


Figure 7.1: Graphical representation of an RBM model.

Here is an intuitive explanation, in four steps, of how they

work. Let's consider the commonly used situation where all nodes (neurons/units) are binary. In this case:

1. Visible layer nodes take values $\mathrm{v_i} = 0$ or $\mathrm{v_i} = 1$.

2. Each node is a locus of computation that processes input, and begins by making stochastic decisions about whether to transmit that input or not.

3. The hidden units serve to increase the expressiveness of the model and contain binary units where $\mathrm{h_j} = 0$ or $\mathrm{h_j} = 1$.

4. For each unit or node in the hidden layer, the corresponding value is unobservable and it needs to be inferred.

The Role of Energy and Probability

The RBM is a probabilistic energy-based model, meaning the probability of a specific configuration of the visible and hidden units is proportional to the negative exponentiation of an energy function, $\tilde{E}(v, h)$. For each pair of a visible vector and a hidden vector the probability of the pair (v, h) is defined as follows:

$$P(v, h) = \frac{1}{Z} \exp -\tilde{E}(v, h),$$

where the denominator Z is a normalizing constant known as the partition function.

The partition function sums over all possible pairs of visible and hidden variables is computed as:

$$Z = \sum_{v} \sum_{h} \exp -\tilde{E}(v, h),$$

It is therefore a normalizing constant such that $P(v, h)$ defines a probability distribution over all possible pairs of v and h.

NOTE... ✍

A bipartite structure or graph is math speak for a group of nodes which can only be one of two colors, with lines (edges) connecting the nodes (vertices) so that no two nodes of the same color are connected by a line. Figure 7.2 illustrates this idea.

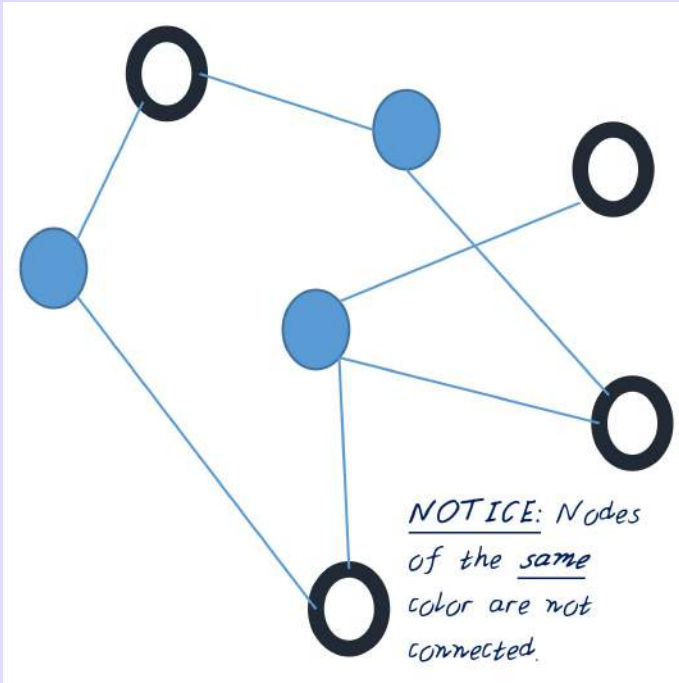


Figure 7.2: Bipartite structure

In the RBM the bipartite graph is constructed of hidden and visible nodes as illustrated in Figure 7.1.

As there are no intra-layer connections within an RBM the energy of a joint configuration (v, h), which defines a bipartite

structure, is given by:

$$\tilde{E}(v,h) = -\sum_{i=1}^{m}\sum_{j=1}^{n} w_{ij}v_i h_i - \sum_{i=1}^{m} v_i b_i - \sum_{j=1}^{n} h_j d_j,$$

where w_{ij} is the weight associated with the link between v_i and h_j. Intuitively, weights identify the correlation of the nodes. Larger weights imply a larger possibility that connected nodes concur. The parameters b_i and d_j are the biases of the j^{th} hidden and i^{th} visible nodes respectively; and m and n are the number of nodes in the visible and hidden layer. Also note that configurations with a lower energy function have a higher probability of being realized.

A Magnificent Way to Think

Another way to think about a RBM is as a parametric model of the joint probability distribution of visible and hidden variables. It is therefore a type of autoencoder used to learn a representation (encoding) in terms of the joint probability distribution for a set of data.

Since the hidden units of the RBM only connect to units outside of their specific layer, they are mutually independent given the visible units. The conditional independence of the units in the same layer is a nice property because it allows us to factorize the conditional distributions of the hidden units given the visible units as:

$$P(h|v) = \prod_j P(h_j|v),$$

with

$$P(h_j = 1|v) = \prod_j sigmoid\left(\sum_i w_{ij}v_i + d_j\right)$$

The conditional distribution of the visible units can be factorized in a similar way:

$$P(v|h) = \prod_i P(v_i|h),$$

with

$$P(v_i = 1|h) = \prod_i sigmoid\left(\sum_j w_{ij}h_j + b_i\right)$$

These conditional probabilities are important for the iterative updates between hidden and visible layers when training an RBM model. In practice, Z is difficult to calculate so computation of the joint distribution $P(v, h)$ is typically intractable.

NOTE... ✍

Notice that RBMs have two biases. This is one aspect that distinguishes them from other autoencoders. The hidden bias d_j helps the RBM produce the activations on the forward pass through the network, while the visible layer's biases b_i help the RBM learn the reconstructions during backpropagation or the backward pass.

The Goal of Model Learning

The goal in training an RBM is to adjust the parameters of the model, denoted by Θ, so that the log-likelihood function using the training data is maximized.

The gradient of the log-likelihood is:

$$\frac{\partial}{\partial\Theta}L(\Theta) = -\left\langle\frac{\partial\tilde{E}(v;\Theta)}{\partial\Theta}\right\rangle_{data} + \left\langle\frac{\partial\tilde{E}(v;\Theta)}{\partial\Theta}\right\rangle_{model}$$

where $\langle \bullet \rangle_{data}$ represents the expectation of all visible nodes v in regard to the data distribution and $\langle \bullet \rangle_{model}$ denotes the model joint distribution defined by $P(v, h)$.
Rather unfortunately the log-likelihood gradient is difficult to compute because it involves the intractable partition function Z. To solve this issue, we need a trick. Fortunately, data science has one at hand.

Training Tricks that Work Like Magic

The first training trick involves use of the expressions for $P(h|v)$ and $P(v|h)$. Since both of these expressions are tractable why not use them to sample from the joint distribution $P(v, h)$ using a Gibbs sampler?

Trick 1: O Jogo Bonito!

Gibbs sampling is a Monte Carlo Markov chain algorithm for producing samples from the joint probability distribution of multiple random variables. The basic idea is to construct a Markov chain by updating each variable based on its conditional distribution given the state of the others. This turns out to be a great idea, it involves fancy footwork even Pelé, the most successful league goal scorer in the history of the beautiful game, would be proud - O Jogo Bonito!

In Gibbs sampling the visible nodes are sampled simultaneously given fixed values of the hidden nodes. Similarly, hidden nodes are sampled simultaneously given the visible nodes. For each step in the Markov chain the hidden layer units are randomly chosen to be either 1 or 0 with probability determined by their sigmoid function; and similarly the visible layer unit are randomly chosen to be either 1 or 0 with probability determined by their sigmoid function.

As the number of steps becomes infinitely large the samples of the hidden and visible nodes converge to the joint dis-

tribution $P(v, h)$. Through this rather neat trick stochastic estimates of the gradient are easily obtained.

When I say" *easily*", take it with a "*pinch of salt*." If you are in a hurry, the results were due yesterday, the boss or the client or (heaven forbid) reporters are foaming at the mouth baying for results, then this type of "*easy*" won't work for you. This is because one iteration of alternating Gibbs sampling requires updating all of the hidden units in parallel for $P(h|v)$, followed by updating all of the visible units in parallel $P(v|h)$. Gibbs sampling therefore can take a very long time to converge. Depending on your need for speed, it may be totally unfeasible for your data science challenge.

So you may wonder, what type of trick is this? Maybe it is like the magician who saws the beautiful women in half, and then later in the show she reappears smiling and waving and totally "*together*". I'm amazed every time I see that particular trick. But of what real practical use is it?

Well hang on just a moment! There is another trick that goes along with the first trick. In fact, the second trick supercharges the first trick. If you only had the first trick you might be satisfied, and many Statisticians were. Maybe, if you find yourself with code to write, you would keep it in your back pocket for smallish datasets. In fact, you might use it so infrequently as to forget about it entirely! However, with this second trick in hand you have the keys to the Restricted Boltzmann Machine Kingdom (metaphorically speaking).

Trick 2: The Keys to the Kingdom

Here is the second trick - contrastive divergence which minimizes the Kullback–Leibler distance (see page 126). We won't go into all of the technical details, but at the core is this idea - Start the Markov chain by setting the states of the visible units to a training vector. Then the binary states of the hidden units are all computed in parallel to calculate $P(h|v)$. Once the binary states have been chosen for the hidden units, a "re-

construction" is produced by setting each v_i to 1 with a probability given by $P(v|h)$. Now run Gibbs sampling for a very small number of iterations, but don't wait for convergence.

If we use k to denote the number of Gibbs sampling iterations, then k=1 will do the job for many applications! Yes, in many applications you only need run the Gibbs sampler for 1 step. Since 1 is a lot less than infinity this saves a lot of time! In summary, sampling k (usually k = 1) steps from a Gibbs chain initialized with a data sample yields the contrastive divergence.

In practice more efficiencies are available because updates are typically performed using multiple training examples at a time by averaging over the updates produced by each example. This smooths the learning signal and also helps take advantage of the efficiency of larger matrix operations.

Using the Gibbs sample, the observed data and the free energy function[119]; we are able to approximate the log likelihood gradient. This used in conjunction with the gradient descent back propagation algorithm allows the log likelihood estimates $\hat{\Theta}$ to be efficiently obtained.

Trick 3: Back Pocket Activation's

The standard RBM model uses binary units (BU) in both the visible and hidden layers with the sigmoid activation function. However, many other types of activation can also be used. In particular, real valued data in the range of [0-1] can be modeled using a logistic activation function and contrastive divergence training without any modification.

Another popular activity function are the rectified linear units (see page 18). Vinod and Hinton report[120] rectified linear units (RLU) demonstrate strong results for image analysis. On one task using 32×32 color images, the researchers report the RLU had a prediction accuracy of 0.8073 compared to a prediction accuracy of 0.7777 for BU. However, the standard deviations were too large for the difference to be statistically significant.

Another advantage of the RLU is that they do not have more parameters than the BU, but they are much more expressive. As Vinod and Hinton state "*Compared with binary units, these units* [RLUs] *learn features that are better for object recognition on the NORB dataset*[121] *and face verification on the Labeled Faces in the Wild dataset*[122]*. Unlike binary units, rectified linear units preserve information about relative intensities as information travels through multiple layers of feature detectors.*"

Trick 4: Alternatives to Contrastive Divergence

Since an RBM defines a distribution over all of its variables, there is more than one strategy that can be used to train it. This is good for you because the art of data science lies in finding useful alternatives.

Persistent contrastive divergence is one such alternative. It uses another approximation for sampling from $p(v, h)$ that relies on a single Markov chain which has a persistent state[123], where the Markov chain is not reinitialized with a data sample after parameter updates. It uses the last chain state in the last update step. This technique has been reported to lead to better gradient approximations if the learning rate is chosen sufficiently small.

Fast Persistent Contrastive Divergence[124] is another variant which attempts to speed up learning by introducing an additional set of parameters used only for Gibbs sampling procedure during learning.

Parallel Tempering[125], also known as Replica Exchange Monte Carlo Markov Chain sampling, is another technique you might stumble across. The general idea of parallel tempering is to simulate replicas of the original system of interest by introducing supplementary Gibbs chains.

Each replica is typically at a different temperature. The higher the temperature, the 'smoother' the corresponding dis-

tribution of the chain. Parallel tempering leads to better mixing of the original chain (i.e. achieves good sampling) by allowing the systems at different temperatures to exchange complete configurations with a probability given by the Metropolis Hastings ratio. It turns out that the simulation of K replicas is more than 1/K times more efficient than a standard, single-temperature Monte Carlo simulation. One additional advantage of this technique is that it can make efficient use of large CPU clusters, where different replicas can be run in parallel.

The Key Criticism of Deep Learning

Why does the contrastive divergence approach work? I don't know. The best I can tell you is that the theoreticians don't know either! Scholars Sutskever and Tieleman[126] pointed out decisively that the learning rule is not following the gradient of any function whatsoever. They state "*Despite CD's* [contrastive divergence's] *empirical success, little is known about its theoretical convergence properties.*" As Boston University Professor of Physics Pankaj Mehta and Northwestern University Professor of biological physics David Schwab point out about deep learning[127] "*Despite the enormous success of deep learning, relatively little is understood theoretically about why these techniques are so successful at feature learning and compression.*"

A key criticism of deep learning is that the theoretical properties of the methods introduced, for example contrastive divergence, are not well understood; And this is absolutely correct. But is this a problem? Hell, no! The most successful data scientists are pragmatic. They run well ahead of the theoreticians and innovate solutions to practical problems. They leave the theoreticians several steps behind gasping for breath wrestling with the "*why*" question. If a technique works outstandingly well data scientists use it until something better is discovered. The contrastive divergence approach works well

enough to achieve success in many significant applications. The "*why*", although interesting, can wait!

Did Alexander Graham Bell understand completely the physics of his "*apparatus for transmitting vocal or other sounds telegraphically*"? No! But he made a financial fortune, changed the world and went down in history as one of the great inventors of his era. What about University College, London, professor of mathematics Karl Pearson? Back in 1901 did he understand totally and completely the theoretical properties of his very practical principal component analysis statistical tool? Does the machine learning Guru, Professor Hinton[128] who discovered contrastive divergence, understand all of its theoretical nuances?

Once the theoreticians catch up, they provide additional support for why a great idea is a great idea. And this is a very important activity. But we can't wait for them, let's keep running!

Two Ideas that can Change the World

In the past decade, the restricted Boltzmann machine (RBM) had been widely used in many applications including dimensionality reduction, classification, collaborative filtering, feature learning, and topic modeling. In this section we explore two very promising ideas, which each in their own way, can change the world.

How to Punch Cancer in the Face

Several years ago voice-over artist, writer and actress Erica McMaster, an Ottawa native, began to train as a fighter. Her mind was so focused and her determination so complete that her training activities captured the imagination of the local press. At the end of a beautiful spring day in the historic Old Mill Inn in Etobicoke, adjacent to Toronto's gentle flowing

Humber river and luscious parkland, Erica climbed into the boxing ring and let her fists do the talking.

Her opponent was cancer, and her goal: "*To punch cancer in the face.*" Erica lost her mother, two of her grandparents and a childhood friend to the disease. The Princess Margaret Cancer Foundation's Annual Fight To End Cancer Fundraiser was a huge success[129].

Whilst very few data scientists are trained fighters, the tools we wield can, and do save lives. Let's look at one example of how deep learning can help Erica McMaster "*Punch cancer in the face.*"

Clinical researchers have demonstrated conclusively that cancer sera contain antibodies which react with a unique group of autologous cellular antigens called tumor-associated antigens[130]. This is exciting news because mini-arrays of antigens have the potential to detect and diagnose various types of cancer. The trick is to associate the correct combination of antigens with a specific type of cancer. Once the mini-arrays have been created, this becomes a classification problem for which RBM's are well suited RBM.

Koziol et al.[131] examine the use of RBMs for the classification of Hepatocellular Carcinoma[132]. The standard approach to such classification is logistic regression. It is the grandson of linear regression, a member of the generalized linear family of models and has reigned supreme in the medical statistics community for several decades.

It appears the logistic model was first introduced sometime during the first half of the 19th century by statistician Alphonse Quetlet's student Pierre-François Verhulst. It was discovered, in part, as an exercise to moderate Thomas Malthus's exponential population growth model. Close to 200 years after Verhulst's discovery, the scientific community, commercial sectors and government researchers have fully embraced it as a viable tool for binary data.

Since it is based on modeling the odds of an outcome it is perfectly suited to the classification task, and has rightly

earned its place as the gold standard in many disciplines. Take for example, the United Kingdom, where the performance of trauma departments 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[133] methodology, introduced in 1981. It is a standardized approach for tracking and evaluating outcome of trauma care and is calculated as a combination index based on the revised trauma score (RTS), injury severity score (ISS) and patient's age. TRISS determines the probability of survival of a patient from two logistic regressions, one applied if the patient has a penetrating injury and the other applied for blunt injuries[134].

Today logistic regression is one of the most widely taught and used binary models in the analysis of categorical data. It is a staple in statistics 101 classes, dominates quantitative marketing, and is used with numerous mutations in classical econometrics[135]. You may have studied it and explored its power in your own research. If not, add it to your tool kit today![136] In numerous areas it has become the benchmark which challenger models must meet and defeat.

For their comparative study of logistic regression and RBM Koziol et al. collected sera samples from 175 Hepatocellular Carcinoma patients and 90 normal patients. Twelve antigens were expressed as recombinant proteins. The researchers used a 10-fold cross-validation for both the RBM and logistic regression models. The entire data set was randomly divided into 10 equally sized sub-samples. With each sub-sample stratified to preserve the ratio of cancer cases to controls. The logistic regression and RBM were trained on 9 sub-samples; the 10th sample was used for validation.

NOTE... ✍

Sensitivity is the ability of a classifier to identify positive results, while specificity is the ability to distinguish negative results.

$$Sensitivity = \frac{NTP}{NTP + NTN} \times 100 \qquad (7.1)$$

$$Specificity = \frac{NTN}{NTP + NTN} \times 100 \qquad (7.2)$$

NTP is the number of true positives and NTN is the number of true negatives.

The results for dichotomized data using 12 antigens are shown in Table 7. These results, although promising, are certainly not a home run for RBM; and this is to be expected because data science is all about obtaining a slight edge. It is all you need to win. If data scientists like you can develop models which give us a slight edge over cancer, we will be well on the way to "*punching it in the face.*"

Koziol et al.conclude tentatively "*Their relatively good performance in our classification problem is promising, and RBMs merit further consideration in similar investigations. We encourage others to explore further the utility of the RBM approach in similar settings.*" Other researcher will perform follow up studies, maybe even you?

A Magnificent Way to Assist the Anaesthetist

Just prior to writing this book I found myself hospitalized. I was barely able to walk and rushed into the emergency ward. Within a matter of minutes, the medical professionals diagnosed my issue and began to stabilize my biological metrics.

Model	Sensitivity	Specificity
Logistic Regression	0.697	0.811
RBM	0.720	0.800

Table 7: Results for dichotomized data using 12 antigens. Source Koziol et al.

The very next day I went into surgery, the last thing I recall was the Anaesthetist administrating general anaesthesia.

Surgery lasted twenty minutes, I awoke groggy, pain free and quickly recovered. The incident got me thinking, wouldn't it be an amazing if data science techniques could assist monitoring and administrating of general anaesthesia?

Professor of bio medical engineering, bio-signal processing, medical ultrasound and meditronics, Pei-Jarn Chen and associates[137] had a similar idea.

The researchers develop a RBM to predict the depth of sedation using biological signals as input attributes/ features. Two distinct types of features are extracted.

The first group of attributes were related to a patient's autonomic nervous system which regulates the functions of internal organs such as the heart, stomach and intestines. They include the heart rate, blood pressure and peripheral capillary oxygen saturation (SpO_2).

The second group of attributes measured metrics related to the patient's state induced by an anesthesia. This included the fractional anesthetic concentration, end-tidal carbon dioxide[138], fraction inspiration carbon dioxide[139], and minimum alveolar concentration[140].

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. When this is completed, the results are used to calculate the accuracy of the model.

Continuous data, including the response variable depth of sedation, was collected on 27 patients subject to anaesthesia. The training set consisted of 5000 randomly selected epochs and 1000 epochs for the sample. Leave one- out cross-validation with the mean square error (MSE) was used to evaluate performance.

As a benchmark for performance the researchers compared the MSE of the RBM model to a feed-forward neural network (ANN) and modular neural networks using different features (MN–1, MN–2,MN–3).

Figure 7.3 illustrates the results. It shows the average MSE for ANN,MN–1, MN–2,MN–3, and RBM model. The MSE were 0.0665, 0.0590, 0.0551, 0.0515, and 0.0504, respectively. The RBM model has the smallest error. This is a positive sign for the use of RBM, although not conclusive because the MSE appears to be statistically indistinguishable from that of MN–3.

Chen et al. are slightly bolder than I am (wearing my statistician hat), and assert "*The experimental results demonstrated*

that the proposed approach outperforms feed-forward neural network and modular neural network. Therefore, the proposed approach is able to ease patient monitoring services by using biological systems and promote healthcare quality." Professor Chen's research is a clear step forward. This is an exciting time to be involved in data science!

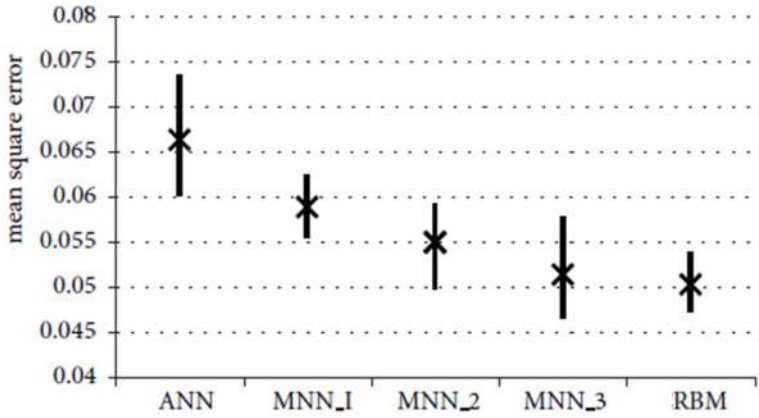


Figure 7.3: MSE results of different modeling approaches. Source adapted from Chen et al. cited in endnote sec. 137

Secrets to the Restricted Boltzmann Machine in R

Let's build an RBM using the `Mobility` dataframe discussed previously on page 166. First we load the required packages and data:

```
>require(RcppDL)
>   require("ltm")
>   data(Mobility)
> data<-Mobility
```

The package `RcppDL` contains functions to estimate a RBM. The package `ltm` contains the `Mobility` dataset.

We sample 1000 observations for our analysis, using 800 for the training set:

```
> set.seed(2395)
> n=nrow(data)
> sample <- sample(1:n, 1000, FALSE)
> data <-as.matrix(Mobility[sample,])
> n=nrow(data)
> train <- sample(1:n, 800, FALSE)
```

Now create the train and test attribute objects in R (`x_train,x_test`). For this example we will remove item 4 and item 6 from the sample (see Table 6):

```
> x_train <-matrix(as.numeric(unlist(data[
   train,])),nrow=nrow(data[train,]))
> x_test<-matrix(as.numeric(unlist(data[-
   train,])),nrow=nrow(data[-train,]))

> x_train<-x_train[,-c(4,6)]
> x_test<-x_test[,-c(4,6)]

> head(x_train)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    1    1    0    1
[2,]    1    0    1    0    0    0
[3,]    1    1    1    0    0    0
[4,]    1    0    0    0    0    0
[5,]    1    0    0    0    0    0
[6,]    1    1    1    0    0    0
> head(x_test)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    1    0    0    0    0    0
[3,]    1    0    0    0    0    0
[4,]    1    0    1    0    0    0
```

```
[5,]    1    1    1    1    0    0
[6,]    0    0    1    0    0    0
```

Specification of the model is similar to what we have previously seen:

```
> fit <- Rrbm(x_train)
```

The r object `fit` contains the model specification. We set 3 hidden nodes with a learning rate of 0.01:

```
> setHiddenRepresentation (fit, x = 3)
> setLearningRate(fit, x = 0.01)
```

Here is the summary of the model:

```
> summary(fit)
$LearningRate
[1] 0.01

$ContrastiveDivergenceStep
[1] 1

$TrainingEpochs
[1] 1000

$HiddenRepresentation
[1] 3
```

Now the parameters have been selected, training is achieved using:

```
> train(fit)
```

That's it! We have built a RBM.

NOTE... ✍

You can set a number of the parameters in `Rrbm` including:

- `setStep signature`
- `setHiddenRepresentation`
- `setLearningRate`
- `setTrainingEpochs`

Now, let's see if we can reconstruct the original values in a similar way to what we did in chapter 5. To achieve this use the `reconstruct` function to obtain the estimated probabilities:

```
reconProb<-reconstruct(fit,x_train)
```

```
> head(reconProb,6)
          [,1]      [,2]      [,3]       [,4]       [,5]       [,6]
[1,] 0.7598778 0.3435336 0.6941071 0.11650449 0.10796628 0.14369203
[2,] 0.8237427 0.3128176 0.7621758 0.06838804 0.06258921 0.08983742
[3,] 0.8187479 0.3163145 0.7565158 0.07186695 0.06600808 0.09402979
[4,] 0.8092837 0.3198017 0.7461960 0.07815092 0.07162580 0.10108005
[5,] 0.8092837 0.3198017 0.7461960 0.07815092 0.07162580 0.10108005
[6,] 0.8187479 0.3163145 0.7565158 0.07186695 0.06600808 0.09402979
```

Then convert the probabilities into binary values:

```
> recon<-ifelse(reconProb>=0.5, 1, 0)
```

You should now see the reconstructed values something like this:

```
> head(recon)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    1    0    0    0
[2,]    1    0    1    0    0    0
```

```
[3,]    1    0    1    0    0    0
[4,]    1    0    1    0    0    0
[5,]    1    0    1    0    0    0
[6,]    1    0    1    0    0    0
```

Finally, we can create a global confusion matrix for all six attributes combined:

```
> table( recon,x_train , dnn =c("Predicted"
    , " Observed"))
          Observed
Predicted    0    1
        0 2786  414
        1  371 1229
```

It can also be useful to view an image of the reconstructed values as shown in Figure 7.4. This is achieved by:

```
> par( mfrow =c(1, 2))
> image(x_train,main="Train")
> image(recon,main="Reconstruction")
```

The reconstructed image appears to contain the main block like features, although not the finer details in terms of streaking across the major blocks. It is left as an exercise for the reader to further refine and improve the model.

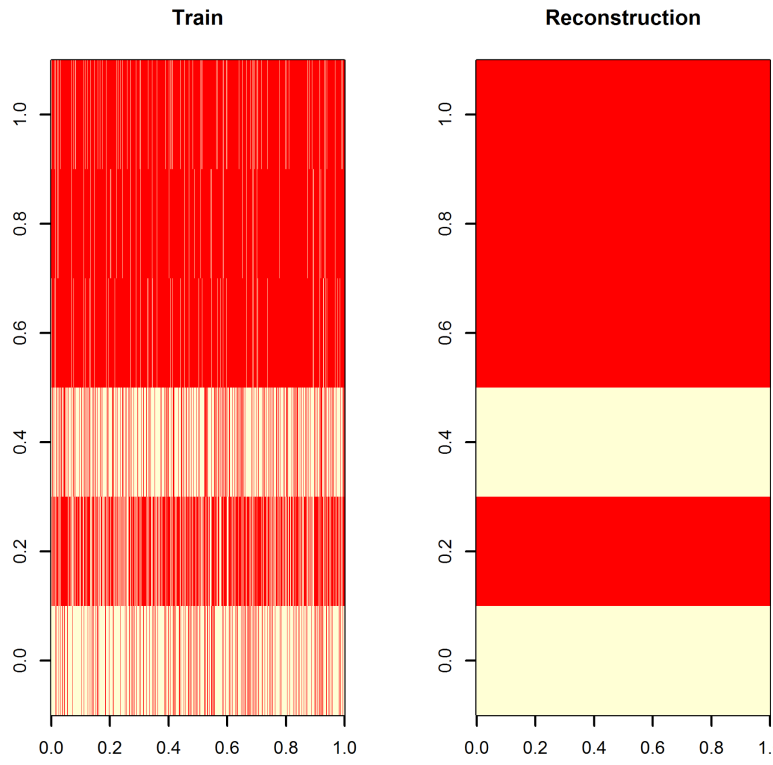


Figure 7.4: RBM train set and reconstructed values using `Mobility`

NOTE... ✍

You can also estimate a RBM with the `deepnet` package. Here is how to do that:

```
> fit2<-rbm.train(x_train,
+ hidden=3,
+ numepochs = 3,
+ batchsize = 100,
+ learningrate = 0.8,
+ learningrate_scale = 1,
+ momentum = 0.5,
+ visible_type = "bin",
+ hidden_type = "bin",
+ cd = 1)
```

Much of this should be familiar to you. Notice that `rbm.train` allows you to specify the batch size which is useful in very large samples. the parameter `cd` refers to the number of steps for contrastive divergence (see sec. 7).

RBMs are powerful learning machines in their own right, however arguably their most important use is as learning modules used in the construction of deep belief networks; and that is the subject of the next chapter.

Notes

[118]A RBM is a bipartite Markov random field.

[119]Unless you are writing computer code to calculate the estimates, the specific details of the algorithm are less important than an intuitive understanding. However,

- A detailed description which is very easy to follow can be found in Bengio, Yoshua. "Learning deep architectures for AI." Foundations and trends® in Machine Learning 2.1 (2009): 1-127.
- Another easy to follow guide is Hinton, Geoffrey. "A practical guide to training restricted Boltzmann machines." Momentum 9.1 (2010): 926.
- If you would like additional details on how contrastive divergence works you can email me at info@NigelDLewis. I'd be delighted to share updated resources with you.

[120]Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." Proceedings of the 27th International Conference on Machine Learning (ICML-10). 2010.

[121]A dataset which contains images of 50 toys belonging to 5 generic categories: four-legged animals, human figures, airplanes, trucks, and cars. The objects were imaged by two cameras under 6 lighting conditions, 9 elevations, and 18 azimuths. For further details please see: LeCun, Yann, Fu Jie Huang, and Leon Bottou. "Learning methods for generic object recognition with invariance to pose and lighting." Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on. Vol. 2. IEEE, 2004.

[122]A database of face photographs designed for studying the problem of unconstrained face recognition. The data set contains more than 13,000 images of faces collected from the web. For further details please see Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. University of Massachusetts, Amherst, Technical Report 07-49, October, 2007.

[123]For details see Tieleman, Tijmen. "Training restricted Boltzmann machines using approximations to the likelihood gradient." Proceedings of the 25th international conference on Machine learning. ACM, 2008.

[124]Tieleman, Tijmen, and Geoffrey Hinton. "Using fast weights to improve persistent contrastive divergence." Proceedings of the 26th Annual International Conference on Machine Learning. ACM, 2009.

[125]See:

- C. J. Geyer, in Computing Science and Statistics Proceedings of the 23rd Symposium on the Interface, American Statistical Association, New York, 1991, p. 156.
- Cho, KyungHyun, Tapani Raiko, and Alexander Ilin. "Parallel tempering is efficient for learning restricted Boltzmann machines." IJCNN. 2010.
- Fischer, Asja, and Christian Igel. "A bound for the convergence rate of parallel tempering for sampling restricted Boltzmann machines." Theoretical Computer Science (2015).

126

- See Sutskever, Ilya, and Tijmen Tieleman. "On the convergence properties of contrastive divergence." International Conference on Artificial Intelligence and Statistics. 2010.
- Also note that the observation that the contrastive divergence update is not the gradient of any objective function was first proved by university of Toronto student Tijmen. See: Some investigations into energy based models. Master's thesis, University of Toronto.

[127]Mehta, Pankaj, and David J. Schwab. "An exact mapping between the variational renormalization group and deep learning." arXiv preprint arXiv:1410.3831 (2014).

[128]Hinton, Geoffrey E. "Training products of experts by minimizing contrastive divergence." Neural computation 14.8 (2002): 1771-1800.

[129]See Toronto Sun article 'Punch cancer in the face' by Steve Buffery. Sunday, March 08, 2015.

[130]See for example:

- Liu, W., et al. "Evaluation of Tumour Associated Antigen (TAA) Miniarray in Immunodiagnosis of Colon Cancer." Scandinavian journal of immunology 69.1 (2009): 57-63.
- Tan, Hwee Tong, et al. "Serum autoantibodies as biomarkers for early cancer detection." FEBS journal 276.23 (2009): 6880-6904.
- Ye, Hua, et al. "Mini-array of multiple tumor-associated antigens (TAAs) in the immunodiagnosis of breast cancer." Oncology letters 5.2 (2013): 663-668.
- Coronell, Johana A. Luna, et al. "The current status of cancer biomarker research using tumour-associated antigens for minimal invasive and early cancer diagnostics." Journal of proteomics 76 (2012): 102-115.

[131]James A. Koziol, Eng M. Tan, Liping Dai, Pengfei Ren, and Jian-Ying Zhang, "Restricted Boltzmann Machines for Classification of Hepatocellular Carcinoma," Computational Biology Journal, vol. 2014, Article ID 418069, 5 pages, 2014. doi:10.1155/2014/418069

[132] Liver cancer

[133]see:

- Champion, Howard R., et al. "Trauma score." Critical care medicine 9.9 (1981): 672-676.
- Champion, Howard R., et al. "A revision of the Trauma Score." Journal of Trauma and Acute Care Surgery 29.5 (1989): 623-629.
- 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.

[134]To try out TRISS for yourself visit http://www.trauma.org/index.php/main/article/387/

[135]See for example Maddala's book. Add it to your personal library. It is a classic and worth every penny:

- Maddala, Gangadharrao S. Limited-dependent and qualitative variables in econometrics. No. 3. Cambridge university press, 1986.

[136]To get you started, here are five books that will tell you all you need to know as a data scientist:

- Allison, Paul D. Logistic regression using SAS: Theory and application. SAS Institute, 2012.
- Harrell, Frank E. Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis. Springer Science & Business Media, 2013.
- Hosmer Jr, David W., Stanley Lemeshow, and Rodney X. Sturdivant. Applied logistic regression. Vol. 398. John Wiley & Sons, 2013.
- Menard, Scott. Applied logistic regression analysis. Vol. 106. Sage, 2002.
- Kleinbaum, David G., and Mitchel Klein. Analysis of Matched Data Using Logistic Regression. Springer New York, 2010.

[137]Yeou-Jiunn Chen, Shih-Chung Chen, and Pei-Jarn Chen, "Prediction of Depth of Sedation from Biological Signals Using Continuous Restricted Boltzmann Machine," Mathematical Problems in Engineering, vol. 2014, Article ID 189040, 6 pages, 2014. doi:10.1155/2014/189040

[138]Measurement of exhaled carbon dioxide.

[139]The fraction or percentage of oxygen in the space being measured. It is used to represent the percentage of oxygen participating in gas-exchange.

[140]The concentration of vapor in the lungs that is required to prevent movement in half of subjects in response to surgical stimulus.

Chapter 8

Deep Belief Networks

Statisticians, like artists, have the bad habit of falling in love with their models.

George Box

A deep belief network (DBN) is a probabilistic generative multilayer neural network composed of several stacked Restricted Boltzmann Machines. In a DBN, every two sequential hidden neural layers form an RBM. The input of the current RBM is actually the output features of a previous one. Each model performs a nonlinear transformation on its input vectors and produces as output, the vectors that will be used as input for the next model in the sequence. Thus each RBM in the stack that makes up a DBN receives a different representation of the data.

DBNs usually consists of two types of layer. They are visible layer and hidden layer. Visible layers contain input nodes and output nodes, and hidden layers contain hidden nodes. The training of DBN can be divided into two steps: pretraining and fine-tuning using discriminative backpropagation.

How to Train a Deep Belief Network

Pre-training followed by fine tuning is a powerful mechanism for training a deep belief networks[141]. Let's take a look at each of these elements.

The Key Elements of Pre-training

Pre-training helps generalization; it uses a layer-wise greedy learning strategy in which each RBM is trained individually layer by layer using contrastive divergence and then stacked on top of each other.

When the first RBM has been trained, its parameters are fixed, and the hidden unit values are used as the visible unit values for the second RBM. This is done by training it using the training sample as the visible layer. Then weights of the learned RBM are then used to predict the probability of activating the hidden nodes for each example in the training dataset.

These activation probabilities are then used as the visible layer in the second RBM. This procedure is repeated several times to initialize the weights for several hidden layers until the final RBM is trained. Overall, the DBN, as with other multilayer neural networks, can be efficiently trained by using the feature activation's of one layer as the training data for the next.

Since pretraining is unsupervised, no class labels are required at this stage. In practice a batch-learning method is often applied to accelerate the pre-training process with the weights of the RBMs updated every mini-batch.

The many layers of hidden units create many layers of feature detectors that become progressively more complex. Pre-training is believed to help capture high level features of the data which assist supervised learning when labels are provided in the fine tuning stage.

The Heart of Fine-tuning

The fine-tuning stage uses the standard back-propagation algorithm through the whole pre-trained network to adjust the features in every layer to make them more useful for classification. This, as we have seen earlier, involves adjusting the weights for enhanced discriminative ability by minimizing the error between the network predicted class and the actual class.

A softmax output layer is typical used as a multiclass classifier. The softmax classifier therefore learns a joint model of the features extracted by the RBMs and the corresponding label of the test samples where the number of nodes in the output layer is equal to the number of classes.

Since fine-tuning involves supervised learning, the corresponding labels for the training data are needed. After training, the predicted class label of a test sample are obtained in the usual manner - by propagation of the test data forward from the first layer visible through to the softmax output layer.

How to Deliver a Better Call Waiting Experience

Call routing is the task of getting callers to the right place in the call center, which could be the appropriate live agent or automated service. Several years ago this was thought best achieved with a list of menu options for callers to select using a telephone touch-tone keypad. These menu drive systems have provided comedians with a goldmine of funny material. I myself have been frustrated, baffled and annoyed on many occasions trying to navigate through menu driven systems. Glyn Richard's YouTube video captures the real world experience of many[142].

Imagine my joy when telecommunications giant Verizon Corporate Services Group Inc. filed patent number US7092888 B1 - Unsupervised training in natural language call routing[143].

Natural language call routing allows callers to describe the reason for their call in their own words, instead of presenting them with a frustrating and limiting menu!

Well over a decade has passed since the original filing date. Over that time, we have seen a steady rise in natural language call routing. Alas, purely menu driven systems are still with us!

Deep learning has an important role to play in this space. IBM researcher Ruhi Sarikaya[144] investigated the role of DBNs in natural language call routing. This is a very interesting application because it involves real time data collected from a customer call center for a Fortune 500 company. Such companies typically have a wide variety of call types to deal with and the call–routing system for this particular company is coded to respond to 35 different call types.

The training sample consisted of 27,000 automatically transcribed utterances amounting to 178,000 words in total. In order to assess the impact of data size on learning method, Ruhi divided the sample into a range of sets from 1,000 to 10,000 and also included the original sample size of 27,000.

Pre-training was achieved with contrastive divergence. The model was fine-tuned with backpropagation (stochastic gradient descent) using an early stopping rule (see page 57) which limited the number of iterations to 100.

Ruhi reports that the performance improved as the sample data was increased from 1,000 to 10,000. For example, the reported prediction accuracy was 78.1% for the smallest sample size of 1,000, 87% for the sample size of 6,000, and 90.3% for the full sample of 27,000.

In line with data science best practice Ruhi used a range of alternative data analytic models as benchmarks, including the support vector machine (SVM). Ruhi and collaborators conclude by stating "*DBNs performed as well as or slightly better than SVMs for all sizes of training set.*"

A World First Idea that You Can Easily Emulate

Many longstanding residents of the Great State of Texas can still recall the drought of 1949 to 1956. Ask them, and they will tell you how Lake Dallas shrunk down to 1/10 of its size, and how west of the Pecos River only eight inches of rain was recorded for the entire year of 1953. Of course the drought ended, as all droughts do with a deluge of rain that soaked the entire state in the spring of 1956.

I often recount my own memories of the Great Drought of 75 and 76 in England, where hot weather dried up lakes and resulted in water restrictions. It was reported as the hottest summer average temperature in the United Kingdom since records began. And since the British have kept records since at least 1066, that is a dam long time! It too ended in a deluge of rain in the autumn of 76.

Now here is the interesting thing, data scientists have been attempting to predict drought conditions, since before statistics was considered a formal scientific discipline. Governments, local authorities and regional water boards would all like to know when drought conditions are likely to occur.

Researchers Junfei Chen, Qiongji Jin of Hohai university and Jing Chao of Nanjing university investigate this issue using DBNs. Collectively, they have developed what appears to be the worlds first short-term drought prediction model based on deep belief networks[145]. The goal of the researchers was to investigate whether DBNs can be used to predict drought in the Huaihe River Basin.

Data on monthly rainfall from four hydrologic stations (Bengbu, Fuyang, Xuchang, and Zhumadian) was collected from January 1958 to 2006 and then transformed into different time scales using the Standardized Precipitation Index[146] (SPI). SPI is an index used to characterize meteorological drought on a range of timescales. On short timescales, the SPI

is closely related to soil moisture, while at longer timescales, the SPI can be related to groundwater and reservoir storage.

Four timescales were used by the researchers SPI3, SPI6, SPI9, and SPI12. The training sample consisted of data from the period 1958–1999, and the testing sample used observations from 2000–2006.

A total of 16 DBN models were developed, one for each site and SPI index. The optimal number of nodes in each model is given in Table 8. We see, for example, the DBN developed using the SP13 series at the Bengbu measuring station has 9 input nodes, 5 hidden nodes in the first layer, and 10 hidden nodes in the second layer. Whilst the DBN for SPI12 for the same weather station has 8 input nodes, with 5 nodes in each of the hidden layers.

SPI series	Station	Number of input nodes	Number of first hidden nodes	Number of second hidden nodes
SPI3	Bengbu	9	5	10
	Fuyang	8	20	15
	Xuchang	8	20	15
	Zhumadian	8	20	15
SPI6	Bengbu	10	5	10
	Fuyang	7	5	5
	Xuchang	7	5	5
	Zhumadian	8	5	5
SPI9	Bengbu	10	5	5
	Fuyang	7	5	5
	Xuchang	6	5	5
	Zhumadian	10	5	15
SPI12	Bengbu	8	5	5
	Fuyang	10	5	5
	Xuchang	9	5	5
	Zhumadian	10	5	5

Table 8: The optimal network structures of DBN. Source of table Chen et al. cited in endnote sec. 145.

Figure 8.1 illustrates the fit of the DBN models for the Bengbu measuring station. Visually, the fit of each model looks reasonable. The root mean square error (RMSE) and mean absolute error (MAE) are also given.

The researchers benchmark their models against a back-propagation (BP) neural network and find the DBN clearly outperforms. This leads them to conclude "*The errors results show that the DBN model outperforms the BP neural network. This study shows that the DBN model is a useful tool for drought prediction.*" And with this study Junfei Chen, Qiongji Jin and Jing Chao, have created a world first; and so can you!

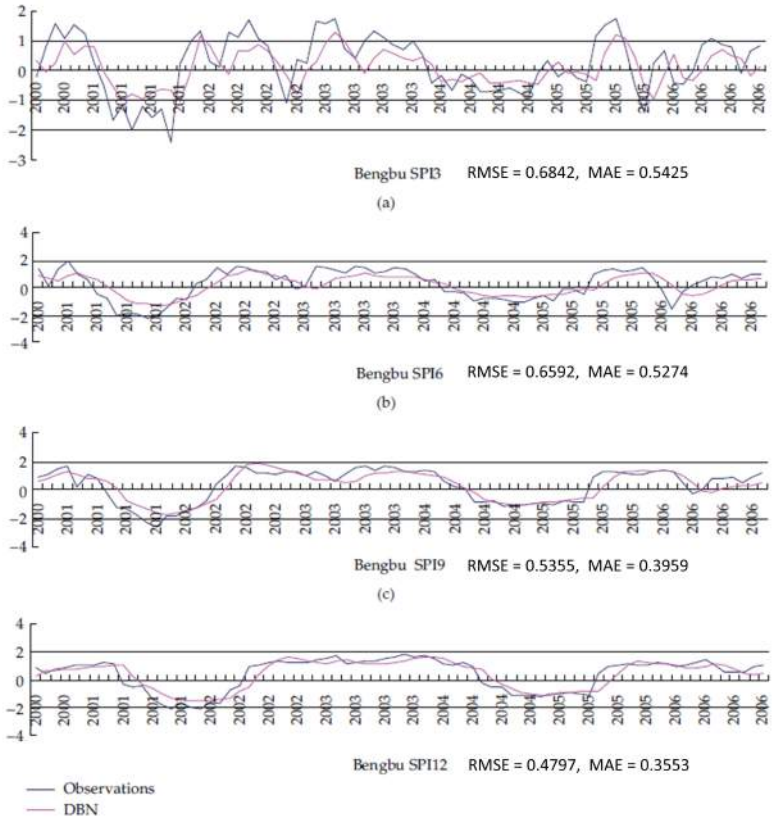


Figure 8.1: Observed and predicted values for different time-scale SPI series at the Bengbu station. Source of figure: Chen et al. cited in endnote sec. 145.

Steps to Building a Deep Belief Network in R

We will continue with the analysis of page 194 and use the `Rdbn` function in the package `RcppDL`. For this illustration we create a combined response variable of item 4 and item 6, from Table 6, by taking the row-wise maximum:

```
> y<-apply(cbind(data[,4],data[,6]), 1, max
   , na.rm = TRUE)
```

The test and training response variable has to be formatted so we can pass it to the `Rdbn` function. Essentially, this involves creating a response variable R object with two columns. First for the test sample:

```
> y_train<-as.numeric(y[train])
> temp<-ifelse(y_train==0, 1, 0)
> y_train<-cbind(y_train,temp)
> head(y_train)
     y_train temp
[1,]       1    0
[2,]       0    1
[3,]       1    0
[4,]       0    1
[5,]       0    1
[6,]       1    0
```

And now for the training sample:

```
> y_test<-as.numeric(y[-train])
> temp1<-ifelse(y_test==0, 1, 0)
> y_test<-cbind(y_test,temp1)
> head(y_test)
     y_test temp1
[1,]      0     1
[2,]      0     1
[3,]      1     0
[4,]      0     1
```

```
[5,]       1      0
[6,]       0      1
```

A quick check to ensure we have the correct number of observations in the training (800) and test (200) sets:

```
> nrow(y_train)
[1] 800

> nrow(y_test)
[1] 200
```

We choose a model with two hidden layers. Let's specify the model and then look at the default settings:

```
> hidden = c(12,10)
> fit <- Rdbn(x_train, y_train, hidden)

> summary(fit)
$PretrainLearningRate
[1] 0.1

$PretrainingEpochs
[1] 1000

$FinetuneLearningRate
[1] 0.1

$FinetuneEpochs
[1] 500

$ContrastiveDivergenceStep
[1] 1
```

So we have a model with two hidden layers with 12 nodes in the first hidden layer and 10 nodes in the second hidden layer. This time we will run with the default settings. As we have already seen pre-training followed by fine tuning is a powerful mechanism for training a deep belief networks. Here is how to

do this with our sample:

```
> pretrain(fit)
> finetune(fit)
```

The next step is to use the fitted model to predict the classes of the test sample:

```
> predProb<-predict(fit, x_test)
```

We have seen this basic format before. The output is in terms of probabilities. Let's take a look at the first few predictions:

```
> head(predProb,6)
          [,1]      [,2]
[1,] 0.3942584 0.6057416
[2,] 0.4137407 0.5862593
[3,] 0.4137407 0.5862593
[4,] 0.4332217 0.5667783
[5,] 0.4970219 0.5029781
[6,] 0.4142550 0.5857450
```

Next we convert these probabilities into binary values and calculate the confusion matrix:

```
> pred1<-ifelse(predProb[,1]>=0.5, 1, 0)

> table( pred1,y_test[,1] , dnn =c("
   Predicted" , " Observed"))
         Observed
Predicted   0   1
        0 115  75
        1   0  10
```

That's it, we are done!

Notes

[141]See Hinton, Geoffrey E. "To recognize shapes, first learn to generate images." Progress in brain research 165 (2007): 535-547.

[142]Watch it only if you don't know what I am talking about. Seehttps://youtu.be/grSMuM8MXRs

[143]McCarthy, Daniel J., and Premkumar Natarajan. "Unsupervised training in natural language call routing." U.S. Patent No. 7,092,888. 15 Aug. 2006.

[144]Sarikaya, Ruhi, Geoffrey E. Hinton, and Bhuvana Ramabhadran. "Deep belief nets for natural language call-routing." Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on. IEEE, 2011.

[145]Junfei Chen, Qiongji Jin, and Jing Chao, "Design of Deep Belief Networks for Short-Term Prediction of Drought Index Using Data in the Huaihe River Basin," Mathematical Problems in Engineering, vol. 2012, Article ID 235929, 16 pages, 2012. doi:10.1155/2012/235929

[146]For further details see McKee, Thomas B., Nolan J. Doesken, and John Kleist. "The relationship of drought frequency and duration to time scales." Proceedings of the 8th Conference on Applied Climatology. Vol. 17. No. 22. Boston, MA, USA: American Meteorological Society, 1993.

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.

I've spoken to thousands of people over the past few years. I'd love to hear your experiences using the ideas in this book. Contact me with your stories, questions and suggestions at *Info@NigelDLewis.com*.

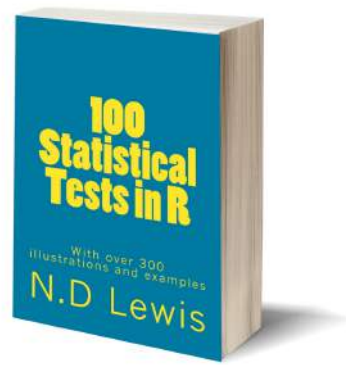
Good luck!

P.S. Thanks for allowing me to partner with you on your data science journey.

Index

OTHER BOOKS YOU WILL ALSO ENJOY

Over 100 Statistical Tests at Your Fingertips!



100 Statistical Tests in R is designed to give you rapid access to one hundred of the most popular statistical tests.

It shows you, step by step, how to carry out these tests in the free and popular R statistical package.

The book was created for the applied researcher whose primary focus is on their subject matter rather than mathematical lemmas or statistical theory.

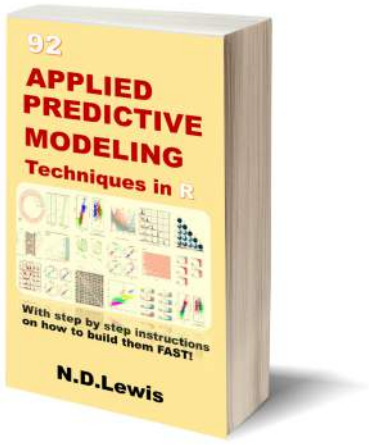
Step by step examples of each test are clearly described, and can be typed directly into R as printed on the page.

To accelerate your research ideas, over three hundred applications of statistical tests across engineering, science, and the social sciences are discussed.

100 Statistical Tests in R

ORDER YOUR COPY TODAY!

AT LAST! Predictive analytic methods within easy reach with R...



This jam-packed book takes you under the hood with step by step instructions using the popular and free R predictive analytic package.

It provides numerous examples, illustrations and exclusive use of real data to help you leverage the power of predictive analytics.

A book for every data analyst, student and applied researcher.

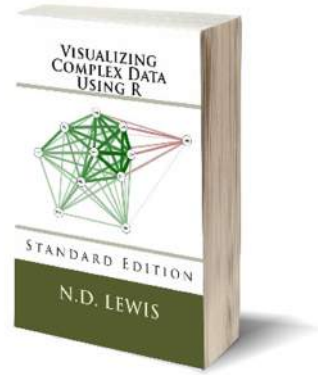
ORDER YOUR COPY TODAY!

"They Laughed As They Gave Me The Data To Analyze...But Then They Saw My Charts!"

Wish you had fresh ways to present data, explore relationships, visualize your data and break free from mundane charts and diagrams?

Visualizing complex relationships with ease using R begins here.

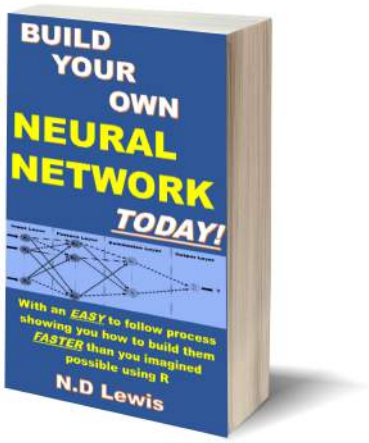
In this book you will find innovative ideas to unlock the relationships in your own data and create killer visuals to help you transform your next presentation from good to great.



Visualizing Complex Data Using R

ORDER YOUR COPY TODAY!

ANNOUNCING...The Fast & Easy Way to Master Neural Networks



This rich, fascinating, accessible hands on guide, puts neural networks firmly into the hands of the practitioner. It reveals how they work, and takes you under the hood with an easy to follow process showing you how to build them faster than you imagined possible using the powerful, free R predictive analytics package.

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

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

Buy the book today and master neural networks the fast & easy way!

ORDER YOUR COPY TODAY!

Write your notes here:

Write your notes here:

Write your notes here:

Write your notes here:

Write your notes here:

Write your notes here:

Write your notes here:

Write your notes here:

Write your notes here: