NEURAL NETWORKS FOR TIME SERIES FORECASTING With R

Intuitive Step by Step Blueprint for Beginners

Dr. N.D Lewis

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

ISBN-13: 978-1544752952
ISBN-10: 1544752954

Contents

Finally, A Blueprint for Neural Network Time Series Forecasting with R!

Neural Networks for Time Series Forecasting with R offers a practical tutorial that uses hands-on examples to step through real-world applications using clear and practical case studies. Through this process it takes you on a gentle, fun and unhurried journey to creating neural network models for time series forecasting with R. Whether you are new to data science or a veteran, this book offers a powerful set of tools for quickly and easily gaining insight from your data using R.

NO EXPERIENCE REQUIRED: **Neural Networks for Time Series Forecasting with R** uses plain language rather than a ton of equations; I'm assuming you never did like linear algebra, don't want to see things derived, dislike complicated computer code, and you're here because you want to try neural networks for time series forecasting for yourself.

YOUR PERSONAL BLUE PRINT: Through a simple to follow step by step process, you will learn how to build neural network time series forecasting models using R. Once you have mastered the process, it will be easy for you to translate your knowledge into your own powerful applications.

THIS BOOK IS FOR YOU IF YOU WANT:

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

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

- **Unleash** the power of Long Short-Term Memory Neural Networks.
- **Develop** hands-on skills using the Gated Recurrent Unit Neural Network.
- **Design** successful applications with Recurrent Neural Networks.
- **Deploy** Jordan and Elman Partially Recurrent Neural Networks.
- **Adapt** Deep Neural Networks for Time Series Forecasting.
- **Master** the General Method of Data Handling Type Neural Networks.

QUICK AND EASY: For each neural network model, every step in the process is detailed, from preparing the data for analysis, to evaluating the results. These steps will build the knowledge you need to apply them to your own data science tasks. Using plain language, this book offers a simple, intuitive, practical, non-mathematical, easy to follow guide to the most successful ideas, outstanding techniques and usable solutions available using R.

GET STARTED TODAY! Everything you need to get started is contained within this book. **Neural Networks for Time Series Forecasting with R** is your very own hands-on practical, tactical, easy to follow guide to mastery.

Buy this book today and accelerate your progress!

Preface

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

- Deep Neural Networks.
- Long Short-Term Memory Neural Network.
- Gated Recurrent Unit Neural Network.
- Simple Recurrent Neural Network.
- Elman Neural Network.
- Jordan Neural Network.
- General Method of Data Handling Type Neural Networks.

And here are the projects:

- Predicting the Number of New Cases of Escherichia coli.
- Forecasting the Whole Bird Spot Price of Chicken.
- Predicting Eye Movements.
- Forecasting Electrocardiogram Activity.
- Historical Wheat Price Forecasting.
- Modeling Air Temperature.
- Forecasting Monthly Sunspot Numbers.

Caution!

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

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

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

A Promise

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

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

I invite you therefore to put what you read in these pages into action. To help you do that, I've created **"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 is 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 Absolute Most Possible Benefit from this Book

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

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

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

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

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

As implied by the title, this book is about understanding and then hands-on use of neural networks for time series forecasting and analysis; more precisely, it is an attempt to give you the tools you need to build neural networks 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.

Getting R

R is a free software environment for statistical computing and graphics. It is available for all the major operating systems. Due to the popularity of Windows, examples in this book use the Windows version of R. You can download a copy of R from the R Project for Statistical Computing.

Learning R

This text is not a tutorial on using R. However, as you work through the examples you will no doubt pick up many tips and tricks. However, 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!

Using Packages

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

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

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

```
require(neuralnet)
```

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

NOTE... ✍

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()
```

Effective Use of Functions

Functions in R often have multiple parameters. The examples in this text 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 `prcomp` function, you would type:

```
?prcomp
```

Details of the function and additional parameters will appear in your default web browser. After fitting your model of interest, you are strongly encouraged to experiment with additional parameters.

I have also included the `set.seed` method in the R code samples throughout this text to assist you in reproducing the results exactly as they appear on the page.

NOTE... ✍

> 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.

Additional Resources to Check Out

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

- 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`.

NOTE... ✍

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

Chapter 1

The Characteristics of Time Series Data

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

Time Series in Practice

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

Business and Economic Time Series

Throughout the year, in every economy across the globe, business time series data is collected to record economic activity, assist management make decisions, and support policy-makers select an appropriate course of action.

Figure 1.1 shows the monthly price of chicken (whole bird) from 2001 to 2015. It is characteristic of many business and economic time series. Over the sample period, the price shows a marked upward trend, peaking at over 110 cents per pound. It also exhibits quite marked dips, and periods of increased volatility.

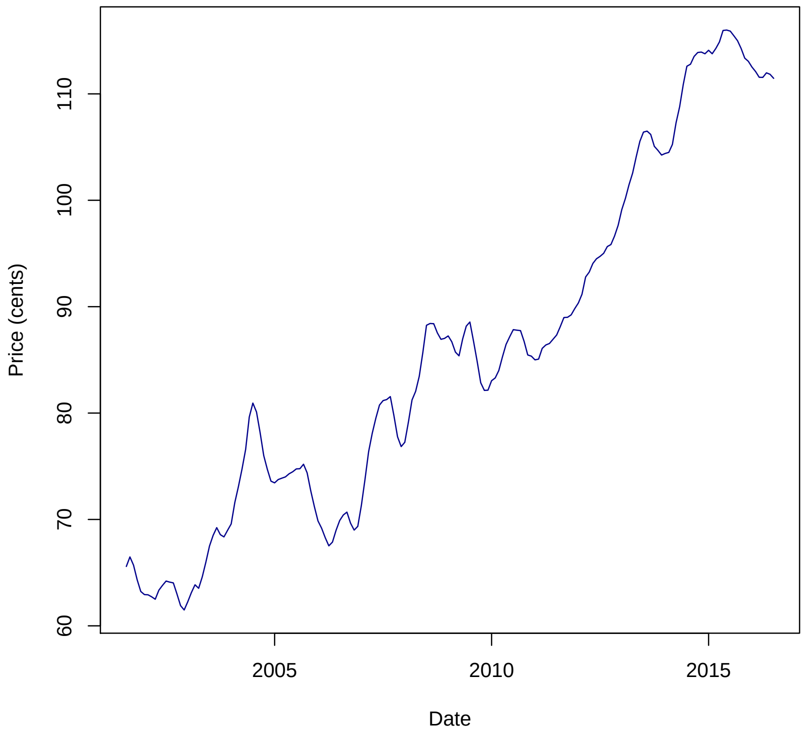


Figure 1.1: Monthly price of chicken

Health and Medical Time Series

Time Series data are routinely collected in medical and health related activities. Blood pressure, weight, heart rate, and numerous other metrics are recorded, often continuously, by various medical devices.

Figure 1.2 shows an electrocardiogram (ECG). An ECG measures the electrical activity of the heart muscle as it changes through time. It is characterized by an undulating cycle with sharp peaks and troughs.

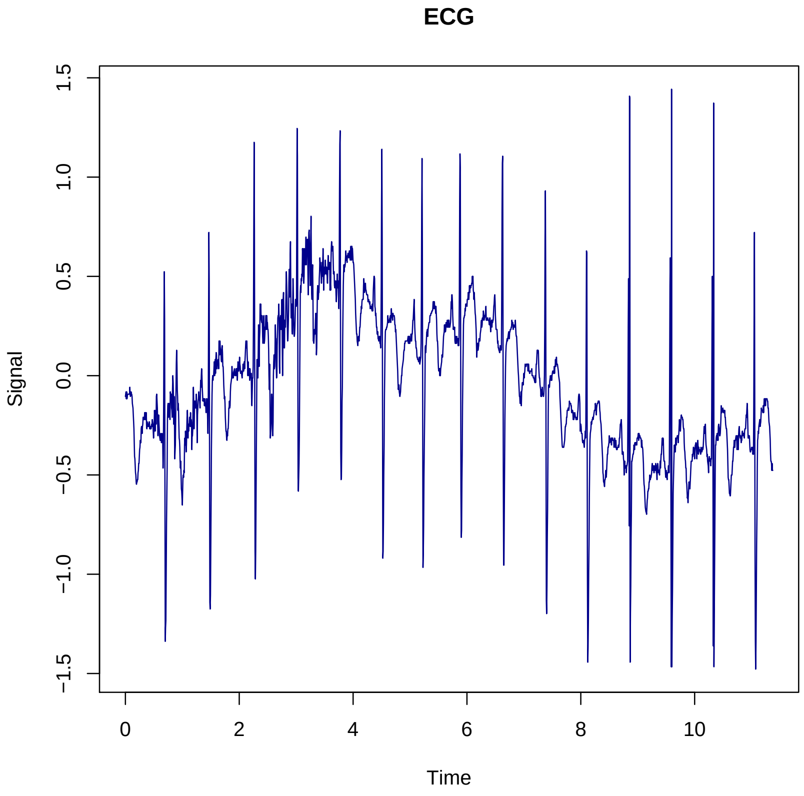


Figure 1.2: Electrocardiogram activity

Physical and Environmental Time Series

The physical and environmental sciences also collect and analyze large amounts of time series data. Environmental variables, collected over long periods of time, help assist in determining climatic trends and future possibilities.

Figure 1.3 shows a time-series plot of the average monthly temperature in Nottingham, England. There does not appear to be any trend evident in the data, however it does exhibit strong seasonality, with peak temperatures occurring in the summer months and lows during the winter.

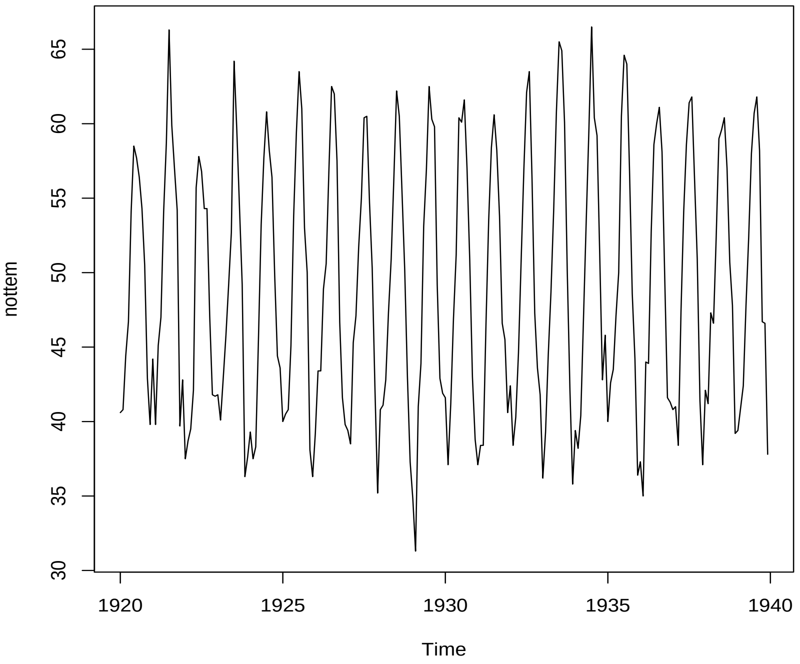


Figure 1.3: Average Monthly Temperatures at Nottingham 1920–1939

Time Series of Counts

Frequently, a time series is composed of counts through time. For example, public health officials in Germany count the number of E.coli cases occurring in a region.

Figure 1.4 shows the weekly count of E.coli cases in the state of North Rhine- Westphalia from January 2001 to May 2013. It appears the number of cases took a dramatic upward turn in late 2011. Since we do not have any information about the cause of the spike, our time series model will have to be flexible enough to handle such upward (or downward) jumps.

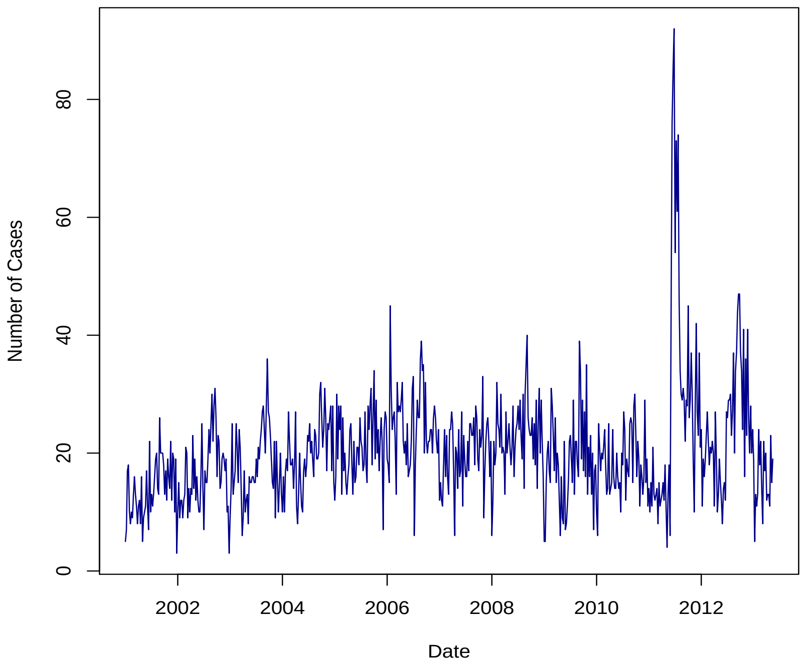


Figure 1.4: Weekly number of Cases of E. coli infection in the state of North Rhine- Westphalia

Discrete Time Series

Time series data is frequently collected on discrete events. In some cases, the events might be binary, such as whether low intensity earth tremors have taken place in the Islands of Japan.

Discrete data can also have many possible states. For example, eye movement is measured as a series of integers representing fixation position. Figure 1.5 plots eye movements for an individual listening to a conversation. It indicates quite large swings in terms of actual eye fixation position over time.

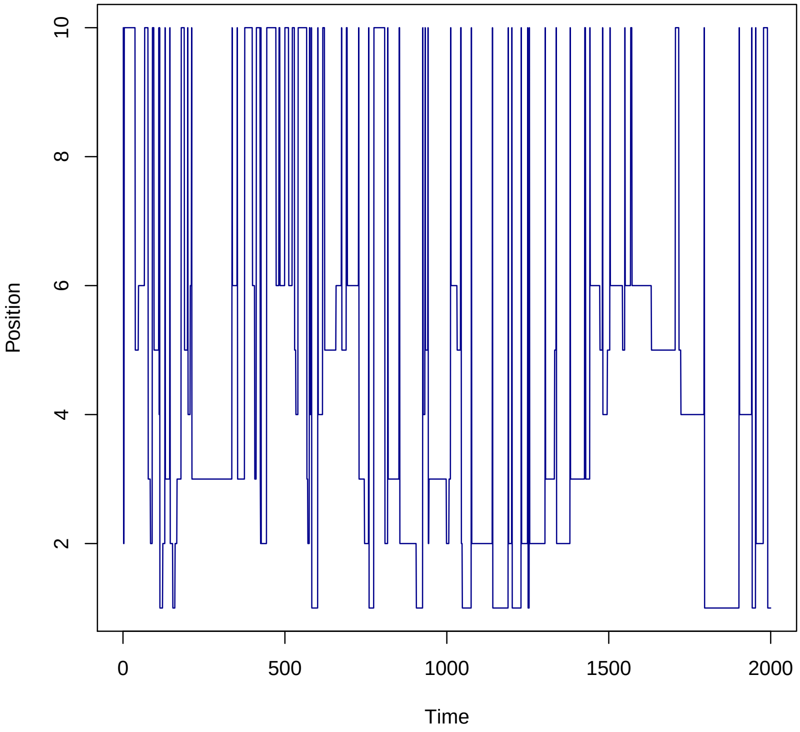


Figure 1.5: Time series plot of eye fixation positions

The Data Generating Mechanism

Time series data are the output of a "*data generating process*". As illustrated in Figure 1.6, at each point in time a new observation is generated. So, for example, at time t we might observe an observation say y. We denote this by y_t. At the next time step, say $t+1$, we observe a new observation, which we denote y_{t+1}. The time step t, might be measured in seconds, hours, days, week, months, years and so on. For example, stock market volatility is calculated daily, and the unemployment rate reported monthly.

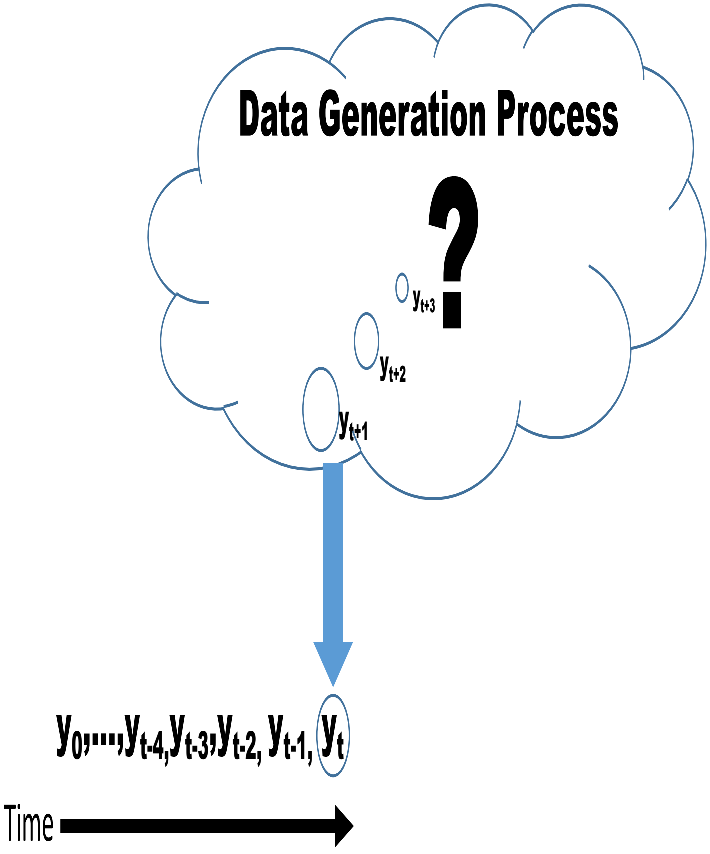


Figure 1.6: Data generating mechanism for time series data

Time Series Decomposition

The mechanism generating time series y_t is often thought to consist of three components. A seasonal component (S_t), a trend component (T_t), and a random error or remainder component (ϵ_t). Provided seasonal variation around the trend cycle does not vary with the level of the time series, we might write:

$$y_t = S_t + T_t + \epsilon_t$$

This is known as the additive model.

An Example

Take another look at Figure 1.3, it might be a good candidate for the additive model. In R you can use the `decompose` function to estimate each of the components:

```
data("nottem",package="datasets")
y<- nottem
y_dec<-decompose(y,  type = "additive")
plot(y_dec)
```

Here is how to read the above lines of R code. The first line loads the data. It is held in the R object `nottem` from the `datasets` package. The data is transferred to the R object y, and in the third line passed to the `decompose` function. The argument `type = "additive"` tells R to use the additive model. The result is stored in the R object `y_dec`, which is visualized using the `plot` function.

Figure 1.7 shows the original (top panel) and decomposed components (bottom three panels). The components can be added together to reconstruct the original time series shown in the top panel. Notice that:

1. The trend is nonlinear;

2. The seasonal component appears to be constant, so all the years in the sample have a very similar pattern;

3. The random component, shown in the bottom panel, shows no clear trend or cycles.

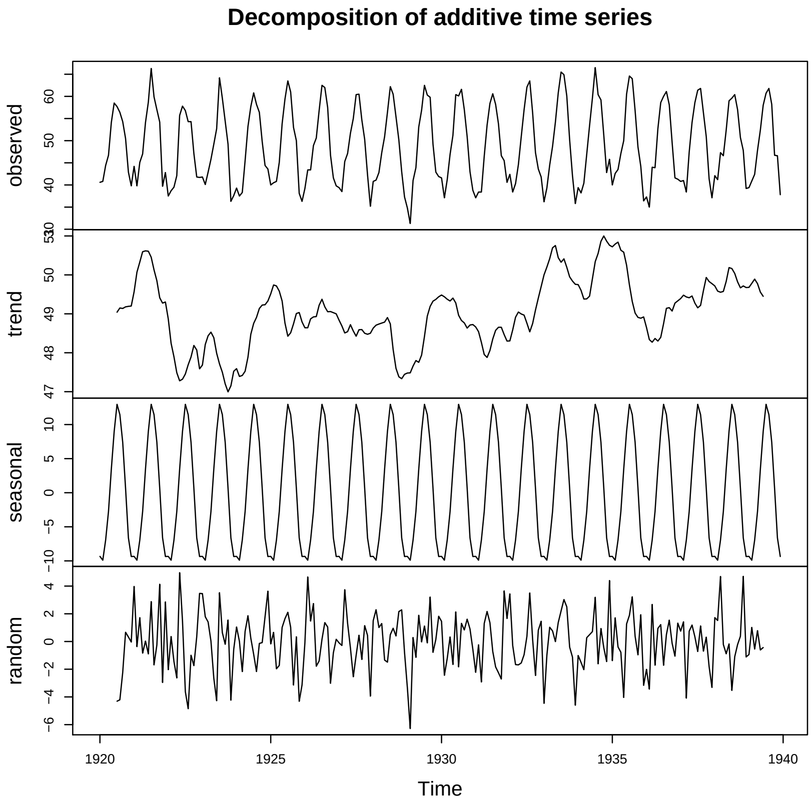


Figure 1.7: Additive model decomposition of the `nottem` dataframe

The multiplicative model

If the seasonal variation around the trend cycle varies with the level of the time series, the multiplicative decomposition can be specified as:

$$y_t = S_t \times T_t \times \epsilon_t$$

NOTE... ✍

You can access the multiplicative model in R by setting `type = "multiplicative"` in the `decompose` function.

The Benefit of Neural Networks for Time Series Modeling

If you have read any traditional books on time series analysis you will no doubt have been introduced to the importance of estimating the seasonality of your data, gauging the trend, and ensuring the data is stationary (constant mean and variance). Unfortunately, in many real-world time series problems the mechanism generating the data is either very complex and/or completely unknown. Such data cannot be adequately described by simple analytical equations.

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

The chief difficulty is that the underlying dynamics generating the time series are usually unknown. Volumes have been written on how to "guesstimate" it, with little overall agreement on the optimal method.

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

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

Summary

Time-series data consists of sampled data points taken over time from a generally unknown process (see Figure 1.6). It has been called one of the top ten challenging problems in predictive analytics.

In the remaining chapters of this text, we will attack this challenge head on with a number of powerful and highly successful neural network models. Let's get started!

Chapter 2

Feed-Forward Neural Networks Explained

NEURAL networks can be used to help solve a wide variety of problems. This is because in principle, they can calculate any computable function. They have been an integral part of the data scientist's toolkit for several decades. Their widespread adoption has improved the accuracy of predictors, and the gradual refinement of neural network training methods continues to benefit both commerce and scientific research. Neural networks 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.

This chapter is devoted to feed-forward neural networks. However, many of the lessons carry over to the other types of neural networks we discuss in the remainder of this book. In this chapter, you will:

- Learn the key concepts that define how a neural network works.

- Work with methods to improve neural network performance.

- Develop an understanding of deep neural networks.

- Build feed-forward neural networks to predict the number of case of Escherichia coli.

If you've worked with neural networks before, some of the material in this chapter may be review. Even so, it may be helpful to take another look, as the concepts outlined lie at the core of modern neural networks, and will play an important role in subsequent chapters.

Understanding Neural Networks

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

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

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

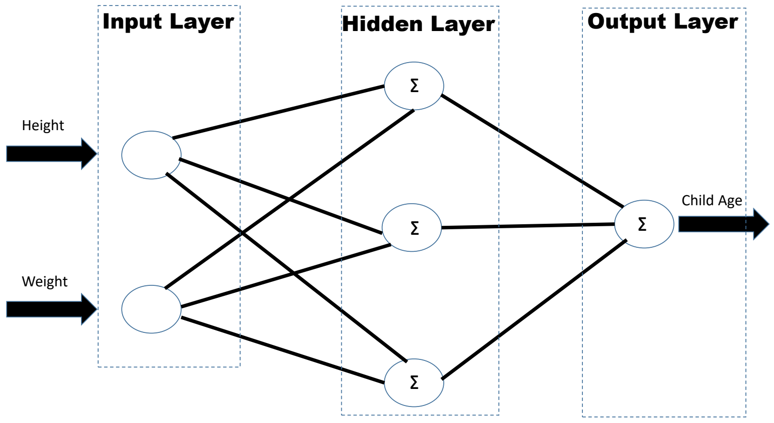


Figure 2.1: A basic neural network

The Role of Neuron

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

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

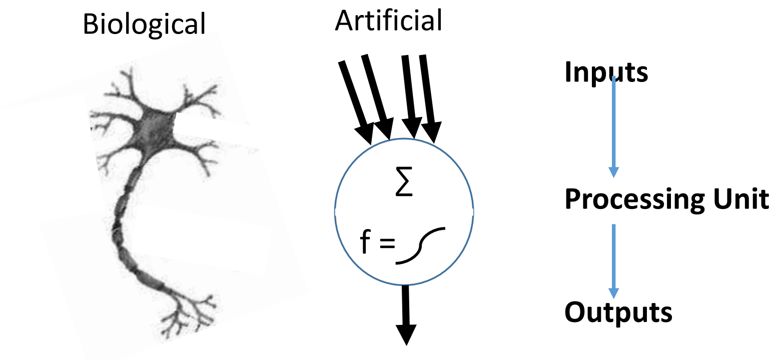


Figure 2.2: Biological and artificial neuron

Explaining Activation Functions

Each neuron contains an activation function (see Figure 2.3) and a threshold value. The threshold value is the minimum value that an input must have to activate the neuron. The activation function is applied and the output passed to the next neuron(s) in the network.

An activation function is designed to limit the output of the neuron, usually to values between 0 to 1, or -1 to +1. In most cases the same activation function is used for every neuron in a network. Almost any nonlinear function does the job, although for the stochastic gradient descent algorithm (see page 31) it must be differentiable and it helps if the function is bounded.

NOTE... ✍

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

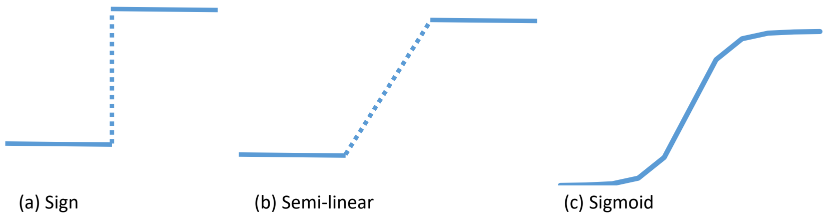


Figure 2.3: Three activation functions

The core task

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

Sigmoid Activation Function

The sigmoid (or logistic) function is a popular choice. It is an "S" shaped differentiable activation function. It is shown in Figure 2.4, where parameter c is a constant taking the value 1.5.

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

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

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

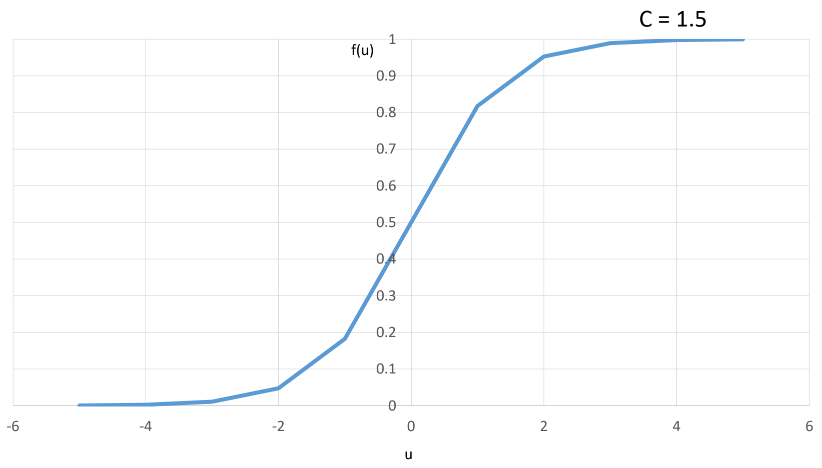


Figure 2.4: The sigmoid function with c = 1.5

Computational Cost

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

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

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

NOTE... ✍

In order to carry out gradient descent (see page 29) the activation function needs to be differentiable.

Tanh Activation Function

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

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

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

As shown in Figure 2.5, unlike the sigmoid function, the Tanh is symmetric around zero - only zero-valued inputs are mapped to near-zero outputs. In addition, strongly negative inputs will map to negative outputs. These properties make the network less likely to get "stuck" during training.

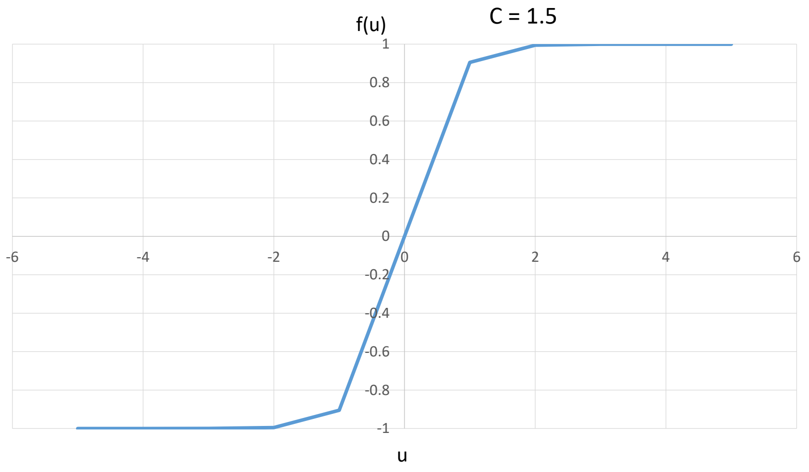


Figure 2.5: Hyperbolic Tangent Activation Function

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

The Working of the Neuron Simplified

Figure 2.6 illustrates the workings of an individual neuron. Given a sample of input attributes $\{x_1,\ldots,x_n\}$, a weight w_{ij} is associated with each connection into the neuron; and the neuron then sums all inputs according to:

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

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

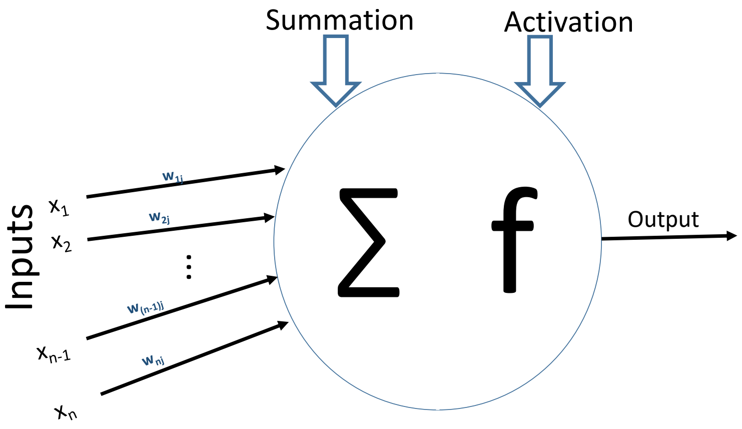


Figure 2.6: Artificial neuron

Network Size

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

NOTE... ✍

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

How a Neural Network Learns

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

Error propagation

The typical neural network learning algorithm applies error propagation from outputs to inputs, and gradually fine-tunes the network weights to minimize the sum of error. The learning cycle is illustrated in Figure 2.7. Each cycle through this learning process is called an epoch.

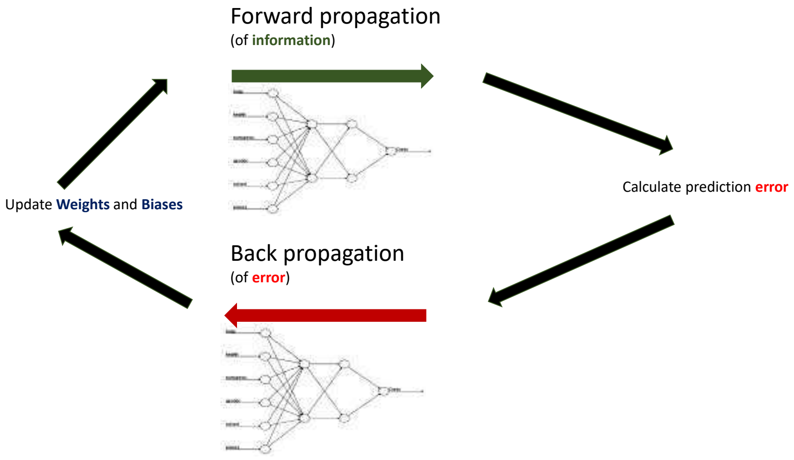


Figure 2.7: Neural network learning cycle

Backpropagation

Backpropagation is the most widely used learning algorithm. It takes the output of the neural network and compares it to the desired value. Then it calculates how far the output was from the desired value. This is a measure of error. Next it calculates the error associated with each neuron from the preceding layer. This process is repeated until the input layer is reached. Since the error is propagated backwards (from output to input attributes) through the network to adjust the weights and biases, the approach is known as backpropagation.

Step by Step Explanation

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

- **Step 1:- Initialization of the network:** The initial values of the weights need to be determined. A neural network is generally initialized with random weights.

- **Step 2:- Feed Forward:** Information is passed forward through the network from input to hidden and output layer via node activation functions and weights. The activation function is (usually) a sigmoidal (i.e., bounded above and below, but differentiable) function of a weighted sum of the nodes inputs.

- **Step 3:- Error assessment:** Assess whether the error is sufficiently small to satisfy requirements, or whether the number of iterations has reached a predetermined limit. If either condition is met, then the training ends. Otherwise, the iterative learning process continues.

- **Step 4:- Propagate:** The error at the output layer is used to re-modify the weights. The algorithm propagates the error backwards through the network and computes the gradient of the change in error with respect to changes in the weight values.

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

Gradient Descent Clarified

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

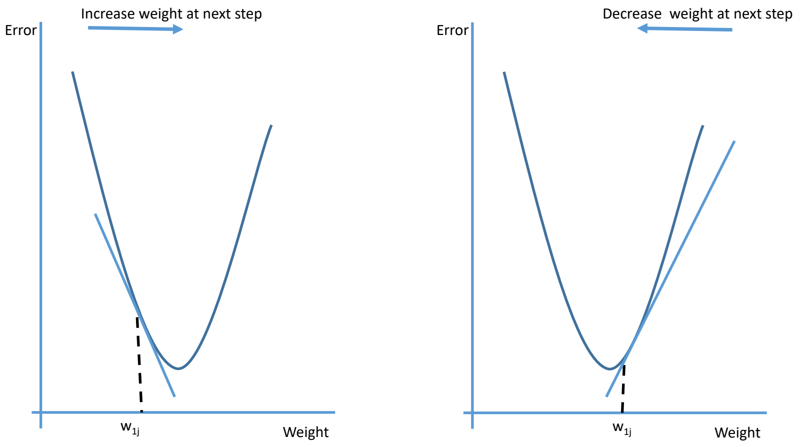


Figure 2.8: Basic idea of Stochastic Gradient minimization

It iteratively updates the weight parameters in the direction of the gradient of the loss function until a minimum is reached. In other words, we follow the direction of the slope of the loss function downhill until we reach a valley. The basic idea is roughly illustrated in Figure 2.8:

- If the partial derivative is negative, the weight is increased (left part of the figure);
- If the partial derivative is positive, the weight is decreased (right part of the figure).
- The parameter known as the learning rate (discussed further on page 33) determines the size of the steps taken to reach the minimum.

Stochastic Gradient Descent

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

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

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

NOTE... ✍

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

Exploring the Error Surface

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

A simple example

To make this analogous to one-variable functions notice that sin(x) is in neither convex nor concave. It has infinitely many maxima and minima, see Figure 2.9 (top panel). Whereas x^2 has only one minimum, and $-x^2$ only one maximum, see Figure 2.9 (bottom panel). The practical consequence of this observation is that, depending on the shape of the error surface, neural networks can get stuck in local minima.

Visualizing the error surface

If you plotted the neural network error as a function of the weights, you would likely see a very rough surface with many local minima. As shown in Figure 2.10, it can have very many peaks and valleys. It may be highly curved in some directions while being flat in others. This makes the optimization process very complex.

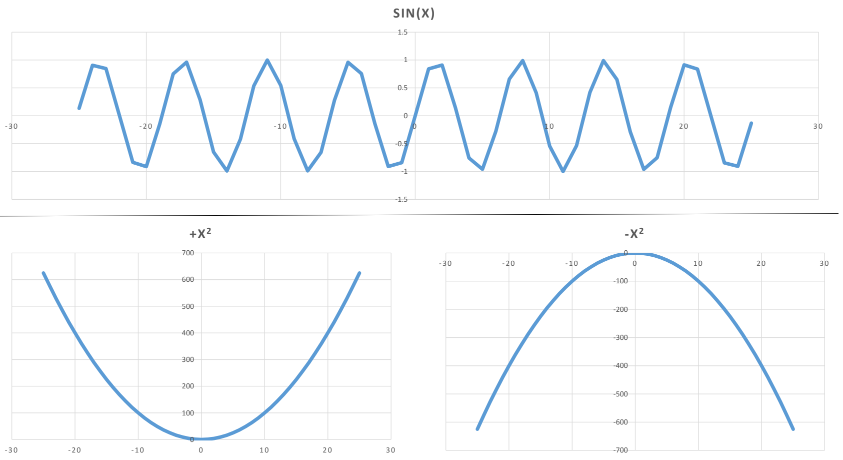


Figure 2.9: One variable functions sin(x), $+x^2$ and $-x^2$

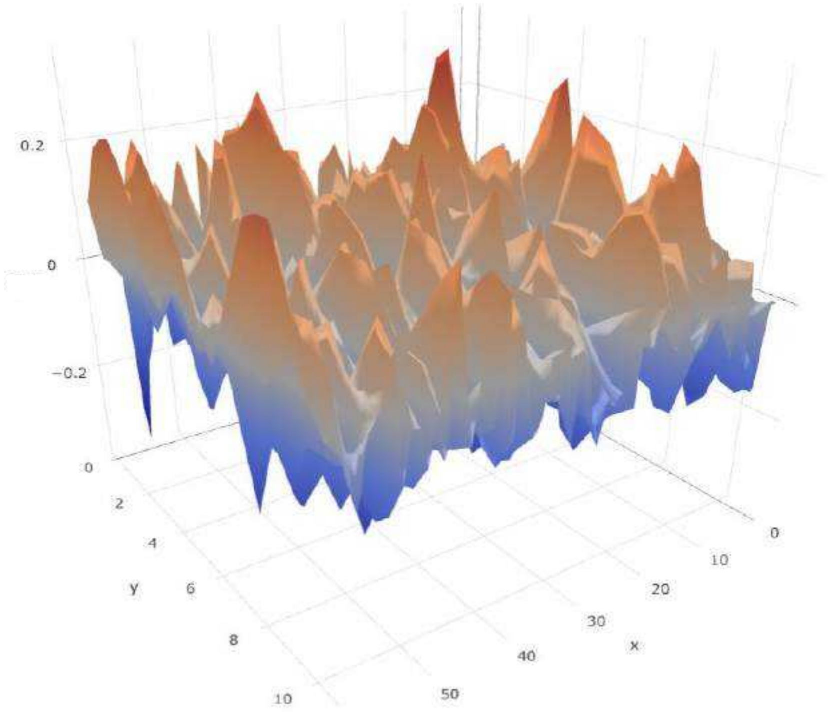


Figure 2.10: Complex error surface of a typical optimization problem

Choosing a Learning Rate

Our next task is to specify the learning rate. It determines the size of the steps taken to reach the minimum by the gradient descent algorithm. Figure 2.11 illustrates the general situation. With a large learning rate, the network may learn very quickly, a lower learning rate takes longer to find the optimum value.

You may wonder, why not set the learning rate to a high value? Well, as shown in Figure 2.12, if the learning rate is too high, the network may miss the global minimum and not learn very well or even at all. Setting the learning rate involves an iterative tuning procedure in which we manually set the highest possible value, usually by trial and error.

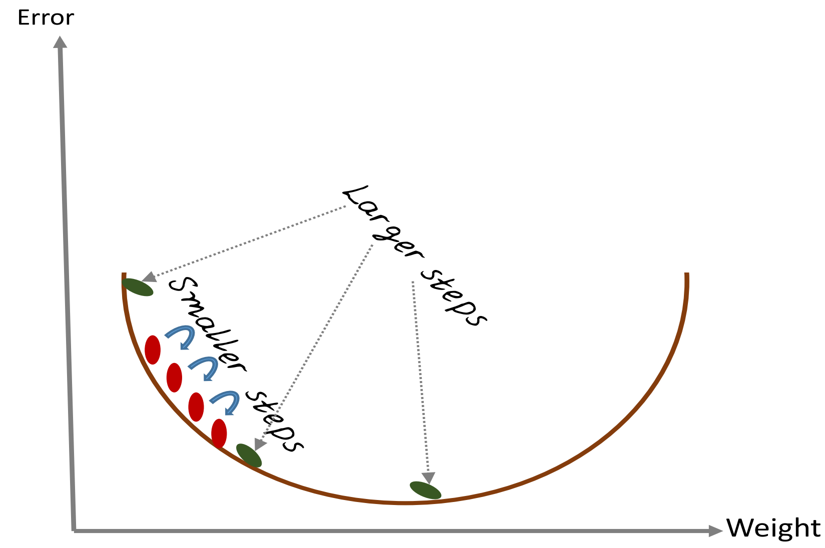


Figure 2.11: Optimization with small and large learning rates

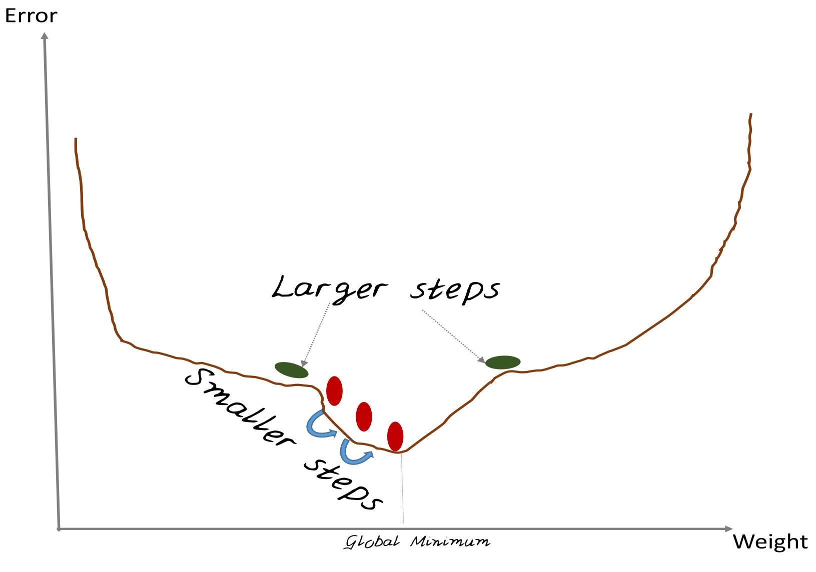


Figure 2.12: Error surface with small and large learning rate

The iterative process of training a neural network continues until the error reaches a suitable minimum. We saw on page 33, that the size of steps taken at each iteration is controlled by the learning rate. Larger steps reduce training time, but increase the likelihood of getting trapped in local minima instead of the global minima.

Another technique that can help the network out of local minima is the use of a momentum term. It can take a value between 0 and 1. It adds this fraction of the previous weight update to the current one. As shown in Figure 2.13, a high value for the momentum parameter, say 0.9, can reduce training time and help the network avoid getting trapped in local minima.

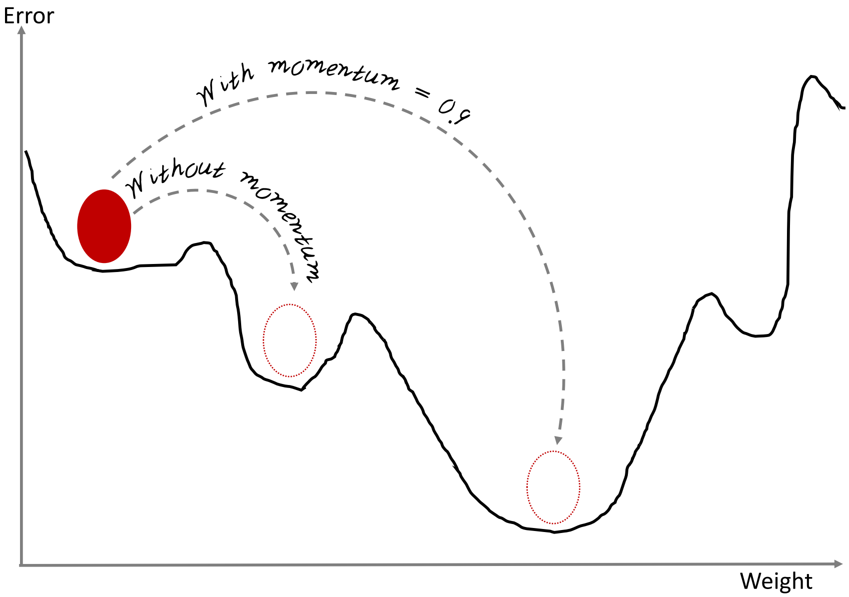


Figure 2.13: Benefit of momentum

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

Choosing the Momentum Value

So how do you set the optimum value of momentum? It is often helpful to experiment with different values. One rule of thumb is to reduce the learning rate when using a lot of momentum.

NOTE... ✍

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

Deep Neural Networks in a Nutshell

A deep neural network (DNN) consists of an input layer, an output layer, and a number of hidden layers sandwiched in between. As shown in Figure 2.14, the hidden layers are connected to the input layers. They combine and weight the input values to produce a new real valued number, which is then passed to the output layer. The output layer makes a classification or prediction decision using the values computed in the hidden layers.

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

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

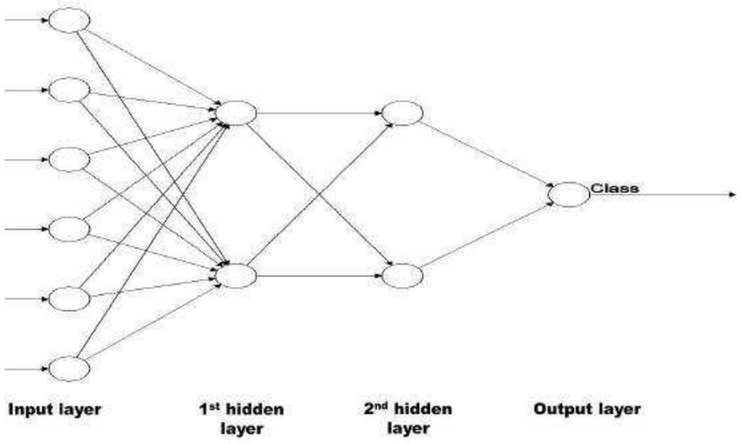


Figure 2.14: Feed forward neural network with 2 hidden layers

The Role of Batching

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

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

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

A batch contains several training examples in one forward/backward pass. To get a sense of the computational efficiency of batching, suppose you had a batch size of 500, with 1000 training examples. It will take only 2 iterations to complete 1 epoch.

NOTE... ✍

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

Practical Application of Feed-Forward Neural Networks

Neural networks have been successfully used in a very wide range of time series problems. One of the most popular applications is forecasting the stock market. We discuss an example in this section. They have also been used in agriculture; we take a look at their role in predicting crop yields. The final illustration in this section, outlines how they have been used in lake water management.

Forecasting Stock Prices

The ability to accurately predict the direction of stock prices is of importance to traders, investors and financial regulators. Large movements up or down in stock prices can have a dramatic impact on the fortunes of an economy. Researcher K.G Akintola, built a deep feed-forward neural network to predict the price movements of equity stocks. The goal was to use weekly forecasts to help buyers determine which shares to buy or sell at the appropriate time.

To assess the potential of their system, stock prices for the Intercontinental Bank Nigeria were collected over a 15 month period. The time lagged price was used to create the input features. In total, four time lagged price inputs were use. So for example, at time t, the input attributes were x_{t-1},x_{t-2},x_{t-3},x_{t-4}, where x is the price, and time is measured in weekly increments.

The neural network was designed with two hidden layers, each with four nodes; with the learning rate set to 0.3, and momentum equal 0.5. The model was optimized over 500 epochs. Figure 2.15 illustrates the network.

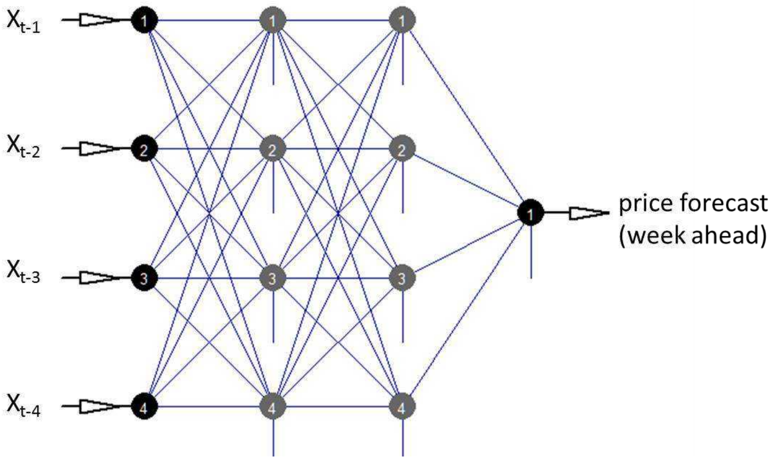


Figure 2.15: Stock Price Deep Neural Network

Predicting Crop Yields

Accurate prediction of crop yields is important for farmers, commodity traders, and government agencies. A. Shabri, R. Samsudin and Z. Ismail, explore the use of a feed-forward neural network to predict rice yields. Rice is an important food staple. About 90% of rice is produced and consumed in Asia.

The dataset collected by the researchers is rather small, consisting of only 78 observations of rice yields gathered from 1995 to 2001. The training sample consisted of 63 observations, and the test set used to assess the forecasts contained the remaining 15 observations.

A note on assessing model performance

A variety of error functions are used to train a neural network. The most common, for regression tasks, is the Mean Square Error (MSE). It is derived by summing the squared differences between the observed target (y) and predicted value ($\hat{y}$) dividing by the number of test examples (n):

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

The smaller the MSE, the closer the predictions match the observed target values.

NOTE... ✍

> MSE measures the average of the squares of the errors between the observed target and predicted values. The smaller the MSE, the closer the predictions match the observed target values.

Since the MSE is measured in squared units of the target, it is common practice to report the square root of this value, known as the root mean squared error (RMSE):

$$RMSE = \sqrt{MSE}$$

The RMSE can be interpreted as the average distance, between the predicted and observed values, measured in units of the target variable.

Another popular metric is the Mean Absolute Error (MAE):

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

Whatever metric is used, the goal is minimize the difference between the predicted and actual observations.

RMSE and MAE for stocks

A. Shabri, R. Samsudin and Z. Ismail use RMSE and MAE to assess performance. They observed that a model with 5 lagged rice yields as inputs, with 10 nodes in a single hidden layer performed the best, see Figure 2.16. The optimal model was trained for 5000 epochs, using the back-propagation algorithm with a very low learning rate of 0.001, paired with a high momentum coefficient of 0.9.

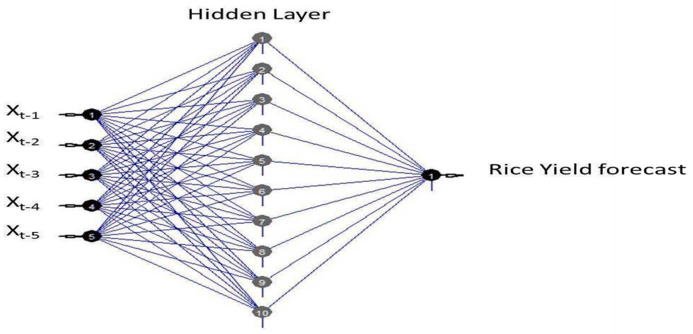


Figure 2.16: Rice yield neural network

Modeling Lake Water Levels

Effective lake water management requires accurate prediction of water level fluctuations. It is important for both leisure activities, and environmental management of the actual water body and surrounding lake shore. Taiwanese researchers Young Chih-Chieh and others, assess the value of neural networks for modeling a time series of observations about lake levels.

Yuan-Yang Lake in north-central Taiwan was used to illustrate the approach. Measurements of inflow discharge, outflow discharge, precipitation, and water level were recorded hourly from August 1 to December 31, 2009, and from January 1 to May 31, 2010.

To forecast the water level, the researchers considered a number of neural network architectures, all with one hidden layer. Between two to four input features were used. These included:

- The net volume flux (derived from inflow discharge, outflow discharge and precipitation);
- and the measured water level with a 1 to 3 hour time lag,

The various models were trained using a learning rate of 0.01, a momentum coefficient of 0.3, and maximum number of iterations set to 400. The number of hidden layer nodes was chosen by trial and error. The best performing neural network model had a MAE of 0.77 and a RMSE of 1.17.

Example - Forecasting the Number of New Cases of Escherichia coli

German bacteriologist Dr. Theodor Escherich first discovered Escherichia coli (E.coli) bacteria in the human colon way back in 1885. They normally live in our gastrointestinal tract, see Figure 2.17, and are a part of a normal and healthy bacterial flora. However, Dr. Escherich showed that certain strains of the bacterium produce a toxin that can cause infant diarrhea and gastroenteritis.

It was later discovered that this little bug could also cause urinary tract infections, and is also responsible for “traveler’s” diarrhea. It turns out the infection is acquired by consuming contaminated food or water, with symptoms appearing around 3 to 4 days after ingestion.

The ability to accurately forecast E.coli levels is of major benefit to public health. We will investigate the use of feed-forward neural networks to forecast the weekly number of reported disease cases caused by E.coli.

Figure 2.17: Escherichia coli

Step 1 – Collecting Data

We use the E.coli infections time series from the `tscount` R package. The data is stored in an object called `ecoli`, and contains the weekly number of reported disease cases caused by Escherichia coli in the state of North Rhine- Westphalia (Germany) from January 2001 to May 2013.

NOTE... ✍

To learn more about the dataset, how it is collected, and obtain recent values visit the Robert Koch Institute at https://survstat.rki.de.

You can load the data into R in one line:

```
data("ecoli",package ="tscount")
```

We can use the `head` function to view the first few observations contained in the object `ecoli`:

```
head(ecoli)
  year week cases
1 2001    1     5
2 2001    2     7
3 2001    3    17
4 2001    4    18
5 2001    5    10
6 2001    6     8
```

For the first week 5 cases were reported. By week 4, this number had risen to 18, before falling back to 8 cases by week 6.

Step 2 – Exploring and Preparing the Data

The R object `ecoli` is a R dataframe object. To see this, use the `class` function:

```
class(ecoli)
[1] "data.frame"
```

It appears to be a numeric variable:

```
class(ecoli$cases)
[1] "numeric"
```

On closer inspection, we see that it is of type `double`:

```
typeof(ecoli$cases)
[1] "double"
```

Let's take a look at how the number of cases evolved over time for the sample. To do this we store the number of cases in an R object called `data`. This object is then converted to a time series data type:

```
data<-as.numeric(unlist(ecoli[3]))
data<-ts(matrix(data),
start=c(2001,1),
end=c(2013,20),
frequency=52)
```

Here is a brief overview of each of these lines of R code:

- The first line makes use of the `unlist` function to coerce the observations into a suitable format. If you don't use `unlist` you will likely see the error message:

```
Error: (list) object cannot be coerced to
   type 'double'
```

- The second line uses the `ts` method to convert the observations in `data` into a time series object, with dates starting in week 1, 2001; and ending in week 20, 2013.

Visualizing the Data

Now, we are ready to plot the data. This is quite straightforward via the `plot` function:

```
plot(data,
xlab="Date",
ylab="Number of Cases",
col="darkblue")
```

You should now see something similar to Figure 2.18. It appears the number of cases took a dramatic upward turn in late 2011. Since we do not have any information about the cause of the spike, our model will have to be flexible enough to handle such upward jumps.

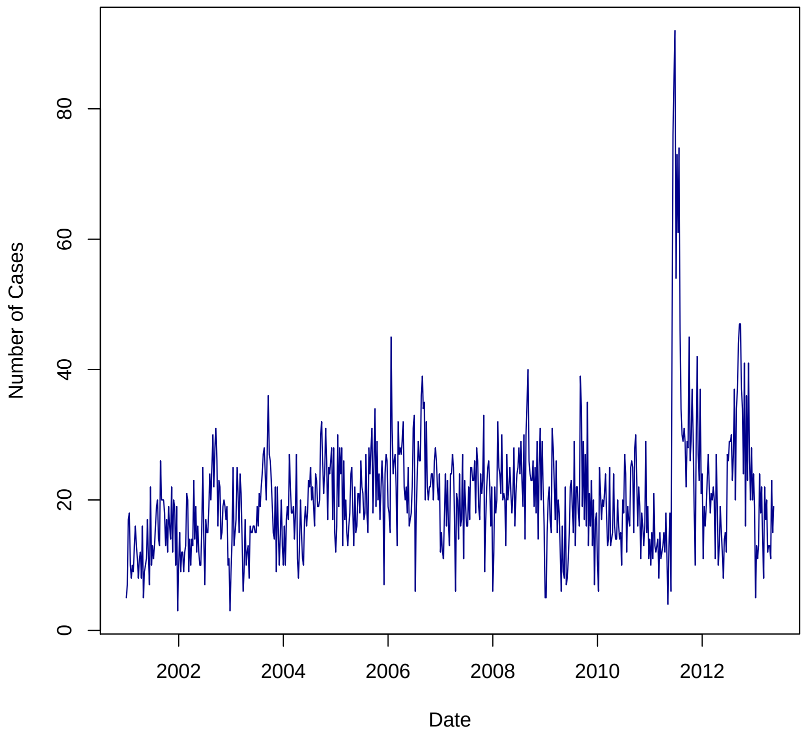


Figure 2.18: Weekly number of Cases of E. coli infection in the state of North Rhine- Westphalia

Partial Autocorrelation

We need to determine how many past observations to include in our model. A simple way to do this is to use partial autocorrelations. A partial autocorrelation is the amount of correlation between an observation x_t and a lag of itself (say x_{t-k}) that is not explained by correlations of the observations in-between. For example, if x_t is a time-series observation measured at time t, then the partial correlation between x_t and x_{t-3} is the amount of correlation between x_t and x_{t-3} that is not explained by their common correlations with x_{t-1} and x_{t-2}. The partial autocorrelation function (PACF) measures directly how an observation is correlated with an observation n time steps apart. It can be called via the `pacf` function:

```
pacf(data)
```

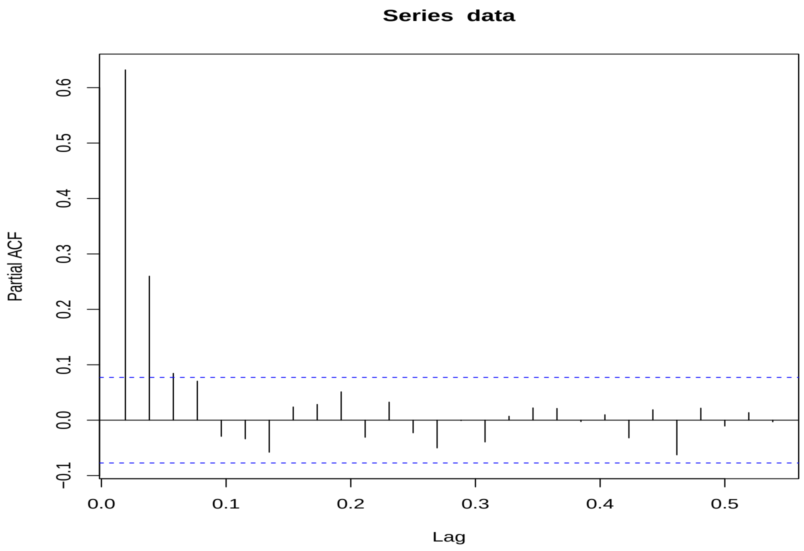


Figure 2.19: PACF of E.coli data

The resultant image, shown in Figure 2.19, plots the partial correlations at each lag. The interpretation of the PACF is somewhat subjective. We use it as a rough guide to how many past observations to include in our model. For this example, the first four partial autocorrelations are around or above 10%.

We take this as an indication that using four past values might be a decent initial guess.

The `quantmod` package

The `quantmod` package has a very intuitive lag operator. It requires the data be in the `zoo` format:

```
require(zoo)
data<-as.zoo(data)
class(data)
[1] "zooreg" "zoo"
```

The observations are now in a suitable format for use with `quantmod`. The `Lag` operator does the work for us:

```
require(quantmod)
x1<-Lag(data, k = 1)
x2<-Lag(data, k = 2)
x3<-Lag(data, k = 3)
x4<-Lag(data, k = 4)
```

`x1` contains the original observations lagged by 1 week; `x2` contains the original observations lagged by 2 weeks; And `x4` contains the original data lagged by 4 weeks.

`cbind` the observations

We will use all the lagged observations and the original series in our analysis. Next, combine them into a single R object called x:

```
x<-cbind(x1,x2,x3,x4,data)
```

The most recent observations in the new object `x` can be viewed using the `tail` function:

```
tail(x)
         Lag.1 Lag.2 Lag.3 Lag.4 data
2013(15)    12    20    17    22   13
2013(16)    13    12    20    17   13
```

```
2013(17)      13      13      12      20     11
2013(18)      11      13      13      12     23
2013(19)      23      11      13      13     15
2013(20)      15      23      11      13     19
```

The most recent observation (week 20 of 2013) measured 19 cases of E.coli (see the column `x$data`).

Missing values

Take a look at the first few observations:

```
head(x)
        Lag.1 Lag.2 Lag.3 Lag.4 data
2001(1)    NA    NA    NA    NA    5
2001(2)     5    NA    NA    NA    7
2001(3)     7     5    NA    NA   17
2001(4)    17     7     5    NA   18
2001(5)    18    17     7     5   10
2001(6)    10    18    17     7    8
```

Missing observations are identified by `NA`. The forth lagged variable has four missing values, and the first lagged variable has one missing value. This is to be expected and is a direct result of using the `Lag` function. However, before we begin our analysis, these missing values need to be removed. We can do that in one line:

```
x <- x[-(1:4),]
```

To see if it worked as expected, take a look at the first few observations using the `head` function:

```
head(x)
          Lag.1 Lag.2 Lag.3 Lag.4 data
2001(5)      18    17     7     5   10
2001(6)      10    18    17     7    8
2001(7)       8    10    18    17   10
2001(8)      10     8    10    18    9
2001(9)       9    10     8    10   13
```

```
2001(10)     13          9       10          8     16
```

Yep, all looks good here. The first column contains the values of `data` lagged by one time step (week), the second column lagged by two time steps, the third column lagged by three time steps, and the fourth column lagged by four time steps.

NOTE... ✍

> Do a quick spot check. Notice when `data` has a value of 16, `x1=13`, `x2=9`,`x3=10` and `x4=8`. All attributes have the expected lagged values.

The Importance of Normalization

In traditional statistical analysis, it is common to standardize a variable so that it's distribution is approximately Gaussian. The Gaussian distribution has some nice theoretical properties. When building a deep neural network, it is also a good idea to standardize the attributes because the algorithm might perform badly if they are not more or less distributed from a standard normal distribution. A standard normal distributed variable has a mean of 0 and a variance of 1.

Whilst there are no fixed rules about how to standardize attributes, a straightforward way to achieve this involves transforming the data to center it by removing the mean value of each attribute, then scaling it by dividing attributes by their standard deviation. For an attribute x_i:

$$z_i = \frac{x_i - \overline{x}}{\sigma_x} \tag{2.1}$$

This approach is often my first "port of call" when building traditional statistical models. It ensures the observations have a mean of 0 and a standard deviation of 1. Of course, like many things in machine learning, it does not always yield good results; and so you need alternatives: Here are two popular choices:

$$z_i = \frac{x_i}{\sqrt{SS_i}} \tag{2.2}$$

$$z_i = \frac{x_i}{x_{max} + 1} \tag{2.3}$$

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.

Yet another approach involves scaling attributes to lie between a given minimum and maximum value, often between zero and one, so that the maximum absolute value of each attribute is scaled to unit size:

$$z_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{2.4}$$

Which should you use? Well, the key to outstanding success in machine learning is prodigious experimentation. You may eventually try them all.

A custom scale function

To get us started, we use equation 2.4. This is achieved via the user defined `range_data` function:

```
range_data<-function(x) {(x-min(x))/(max(x)
   -min(x))}
```

Here is how we use the function on our data:

```
x<-data.matrix(x)
min_data<-min(x)
max_data<-max(x)
x <-range_data(x)
```

The first line is inserted to ensure `x` is a matrix. It will need to be in this format when we pass it to our feed-forward neural network. The objects `min_data` and `max_data`, in the second and third lines, will be used later when we need to unscale our data. The forth line passes `x` to our user defined function `range_data`. The scaled values are stored in `x`.

A quick check

The observations in our scaled data should lie in the range [0, 1]. The `summary` function offers a quick way to check. For the first two time lags we observe:

```
summary(x[,1:2])
        Lag.1                      Lag.2
 Min.      :0.0000     Min.      :0.0000
 1st Qu.:0.1348        1st Qu.:0.1348
 Median  :0.1798       Median  :0.1798
 Mean      :0.1956     Mean      :0.1957
 3rd Qu.:0.2472        3rd Qu.:0.2472
 Max.      :1.0000     Max.      :1.0000
 Max.      :1.0000
```

Each variable lies in the expected range.

And for the remaining two lagged variables we have:

```
summary(x[,3:4])
        Lag.3                      Lag.4
 Min.      :0.0000     Min.      :0.0000
 1st Qu.:0.1348        1st Qu.:0.1348
 Median  :0.1798       Median  :0.1798
 Mean      :0.1954     Mean      :0.1953
 3rd Qu.:0.2472        3rd Qu.:0.2472
 Max.      :1.0000     Max.      :1.0000
```

As expected, each attribute lies in the 0 to 1 range.

NOTE... ✍

You should verify for yourself that the fifth column of x (which contains the target variable) also takes similar values to `x1`, `x2`, `x3` and `x4`.

Creating the training and test samples

Now that we have the data in a suitable format for input into a neural network, we need to create our training and test sets. First, let's check to see how many observations we have to play with:

```
nrow(x)
[1] 640
```

A grand total of 640 examples. We use 600 observations for the training sample, and the remaining 40 for the test set.

To begin, transfer the target observations into the R object `y`, leaving the attributes/features in the object `x`:

```
y<-as.numeric(x[,5])
x<-x[,-5]
```

Next, create the training and test sets:

```
n_train <- 600
x_train<-x[1:n_train ,]
y_train<-y[1:n_train]
x_test<-x[(n_train+1):nrow(x) ,]
y_test<-y[(n_train+1):nrow(x)]
```

As a check, you can view the number of examples in the train and test set using the `nrow` function:

```
nrow(x_test)
[1] 40

nrow(x_train)
[1] 600
```

Yep, the numbers are as we expected.

NOTE... ✍

You should verify for yourself that the number of test and train examples in the target variable `y` match those of the feature set `x`.

Step 3 – Training a Model on the Data

We build a two hidden layer deep neural network, shown in Figure 2.20. It contains 4 input nodes, one for each of the lagged variables; 10 nodes in the first hidden layer, 2 nodes in the second hidden layer; and 1 output node which is used to report the model forecast.

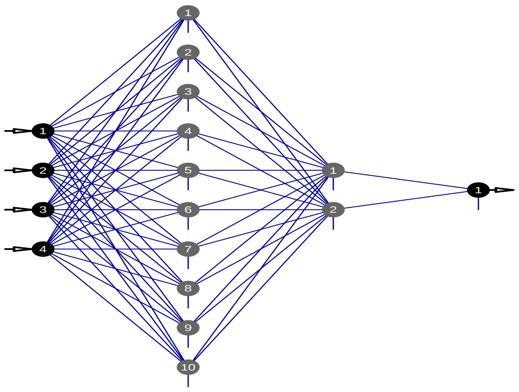


Figure 2.20: The deep neural network (`model1`)

The MxNet package

The MxNet package is used to construct the model. It is a flexible and efficient library for deep learning. The first step is to get a copy of the package. You can do that using the following R code:

```
install.packages("drat",
repos="https://cran.rstudio.com")
drat:::addRepo("dmlc")
install.packages("mxnet")
```

A deep feed-forward multi-layer perceptron neural network can be built in MxNet with the function call:

```
library(mxnet)
mx.set.seed(2018)
model1 <- mx.mlp(x_train, y_train,
hidden_node=c(10,2),
out_node=1,
activation="sigmoid",
out_activation="rmse",
num.round=100,
array.batch.size=20,
learning.rate=0.07,
momentum=0.9,
device=mx.cpu())
```

Let's take a moment to go over this:

- `mx.set.seed` is used to set the random number generator seed to be used for the initialization of the weights. This is to help with reproducibility.
- The data, in terms of the attributes and target variable, are passed first to the `mx.mlp` function.
- The number of nodes in the hidden layer is specified using the `hidden_node` argument; And the number of output node is controlled by the `out_node` argument.

- The sigmoid (logistic) function is chosen as the activation function via the `activation` argument.
- The error metric is specified using the `out_activation` argument. In this case, the root mean square error (see page 40) is used.
- The maximum number of iterations to run the model, and the batch size are determined by the `num.round` and `array.batch.size` arguments respectively. For this example, we run the model for a maximum of 100 iterations, with a sample batch size equal to 20.
- A relatively low learning rate (0.07) is paired with a high momentum (0.9).
- A nice feature of the MxNet package is that you can use your computers standard processing units (CPU) or graphical processing units (GPU). For the examples in this text we use the CPU by setting `device=mx.cpu()` .

Once the model has finished running, the optimized weights and biases are stored in the object `model1`.

NOTE... ✍

The examples in this chapter use the R version 0.9.4 of MxNet.

Step 4 – Evaluating Model Performance

To use the trained model for prediction, you simply need to invoke the `predict` function. It takes the model as the first parameter, and the attribute set as the second:

```
pred1_train = predict(model1,
x_train,
ctx=mx.cpu())
```

The first 8 fitted values are shown below:

```
pred1_train[,1:8]
[1] 0.1862900 0.1862898 0.1862897 0.1862894
    0.1862892 0.1862896 0.1862899
[8] 0.1862899
```

Performance on the training set

How well did the model perform on the training set? One way to asses this is via the root mean square error metric. It can be calculated using the `rmse` function in the `Metrics` package:

```
library (Metrics)
round(rmse(y_train,pred1_train[,1:600]) ,5)
[1] 0.10411
```

The model produces an error of around 0.104 on the training sample data. How good is this? The closer the error is to zero, the better the fit of the model. We can use 0.104 as a benchmark against which to compare alternative models.

Step 5 – Improving Model Performance

Accurate prediction requires the use of a formula for estimating the quantity to be predicted. In linear regression for example, the predicted variable (y) is assumed to be a linear function of the input attributes (x_1,x_2):

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \epsilon$$

The goal is to minimize the error term ϵ. Of course, in practice the relationships between predicted values and the input attributes are often nonlinear or unknown. While this is a hindrance to a traditional statistical model, it does not hamper neural networks to the same extent. This is because it was discovered that one hidden layer is sufficient to model any piece-wise continuous function. The following theorem was developed as a result of this discovery:

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, $|F(x) - \hat{F}(x) < \epsilon|$.

It simply 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.

A single layer model

Let's develop a single hidden layer model, and see how it performs on the training sample:

```
mx.set.seed(2018)
model2 <- mx.mlp(x_train, y_train,
hidden_node=c(10),
out_node=1,
activation="sigmoid",
out_activation="rmse",
num.round=100,
array.batch.size=20,
learning.rate=0.07,
momentum=0.9,
device=mx.cpu())
```

The model, stored in `model2`, has 10 nodes in a single hidden layer, and is otherwise identical to `model1`.

Let's see how it performed in terms of root mean squared error:

```
pred2_train = predict(model2,
x_train,
ctx=mx.cpu())
```

```
round(rmse(y_train,pred2_train[,1:600]),5)
[1] 0.08026
```

It scores 0.080, a clear improvement over `model1`.

Using an alternative activation function

Building neural network models requires a lot of experimentation, sometimes very small changes in the model specification can lead to dramatic improvement in model fit. One "trick" always worth a try is switching in different activation functions. Let's see what happens if we replace the sigmoid function with the tanh function in `model2`:

```
mx.set.seed(2018)
model3 <- mx.mlp(x_train, y_train,
hidden_node=c(10),
out_node=1,
activation="tanh",
out_activation="rmse",
num.round=100,
array.batch.size=20,
learning.rate=0.07,
momentum=0.9,
device=mx.cpu())
```

The only difference between `model2` and `model3` is the use of the tanh activation function in place of the sigmoid function.

So how did the `model3` perform on the test set?

```
pred3_train = predict(model3,
x_train,
ctx=mx.cpu())

round(rmse(y_train,pred3_train[,1:600]),5)
[1] 0.07776
```

The use of a tanh activation enhances performance even further. Of course, great training set performance does not necessarily carryover to great test set performance. However, as a general rule of thumb, I typically select the model with the smallest train set error for evaluation on the test set.

Using this rule of thumb let's select `model3` and see how it performs on the test set.

Test set performance

Simply pass `model3` along with the test set attributes to the `predict` function:

```
pred3_test = predict(model3,
x_test,
ctx=mx.cpu())

round(rmse(y_test,pred3_test[,1:40]),5)
[1] 0.09302
```

The test sample error is higher than the training set error. This quite normal. The reported test set error at 0.093 is still slightly lower than the train set error of `model1`.

Unscaling the data

If you want to look at the predictions in their original units you will need to unscale the data. This simply means reversing the data transformation we did using our custom function `range_data`. To do this we create another custom function, this time called `unscale_data`:

```
unscale_data<-function(x,max_x,min_x)
{x*(max_x-min_x)+min_x}
```

To test the function, let's use it to unscale the test set target attribute data. Since we know what these values should be, we can use them as a simple accuracy check:

```
y_actual<-unscale_data(y_test ,
max_data,
min_data)
```

Next, put the observations into an appropriate format, in this case a matrix of type time series (`ts`):

```
y_actual<-ts(matrix(y_actual) ,
end=c(2013 ,20) ,
frequency=52)
```

Now let's grab the original target variable observations (stored in the object `data`) and put them in a suitable format:

```
original_test_data<-ts(matrix(data[605:644]) ,
end=c(2013 ,20) ,
frequency=52)
```

OK, so our re-scaled target variable observations are stored in `y_actual`, and the original (unscaled) observations are stored in the R object `original_test_data`. Both objects should contain exactly the same values. Let's do a simple check that involves creating a combined dataframe and inspecting the first and last few observations:

```
checkit<-cbind(y_actual ,
original_test_data)
head(checkit)
```

```
     y_actual original_test_data
[1 ,]      37                 37
[2 ,]      20                 20
[3 ,]      34                 34
[4 ,]      37                 37
[5 ,]      44                 44
[6 ,]      47                 47
```

And using the `tail` function:

```
tail(checkit)

        y_actual  original_test_data
[35,]         13                  13
[36,]         13                  13
[37,]         11                  11
[38,]         23                  23
[39,]         15                  15
[40,]         19                  19
```

An alternative

Of course, we could also use the `identical` function to check that the columns in `checkit` are identical:

```
identical(checkit['y_actual'],
checkit['original_test_data'])
[1] TRUE
```

All looks good, the `unscale_data` function appears to be doing a nice job.

Unscaling test set predictions

Now, let's unscale the test set predictions from `model3`:

```
pred_actual<-unscale_data(pred3_test,
max_data,
min_data)

pred_actual<-ts(matrix(pred_actual),
end=c(2013,20),
frequency=52)
```

Plot the result

Finally, plot the prediction and actual observations for the test set using the `ts.plot` function:

```
ts.plot(y_actual, pred_actual,
gpars=list(xlab="Year",
ylab="Cases", lty=c(1:2),
col=c("blue","red")))
```

You should see Figure 2.21. It shows the predictions (dotted line) and actual observations (solid line). The model captures much of the trend, and some of the undulations of the original series.

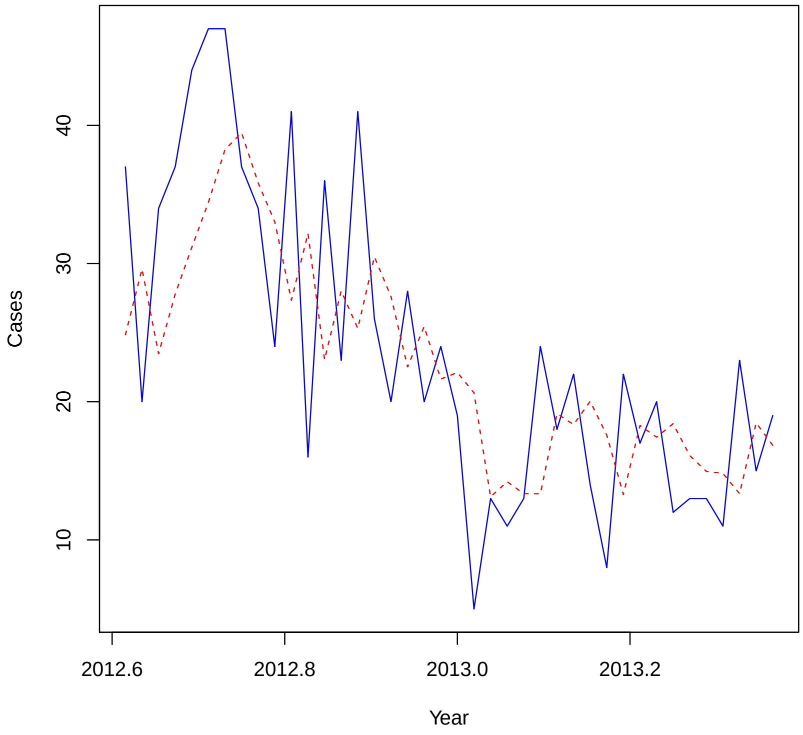


Figure 2.21: E.coli predicted and actual observations for the test set sample via `model3`

NOTE... ✍

Experiment a little with the model parameters. See if you can improve the performance further.

Summary

This chapter presented a detailed overview of feed-forward neural networks. Although deep neural networks are growing in popularity, they do not necessarily yield superior performance to neural networks with only one hidden layers. Unfortunately, without actually building and testing various numbers of layers on actual data, there is no guaranteed way to know whether one layer will be sufficient for the task. In general, the more complex the decision problem, the more layers may be required.

It is also worth mentioning that feed-forward neural networks are only one of many types of neural network useful for time series modeling. In the next chapter, we build upon our knowledge by introducing recurrent neural networks.

Suggested Reading

To find out more about the practical applications of feed-forward neural networks discussed in this chapter see:

- **Stock prices**: Akintola, K. G., B. K. Alese, and A. F. Thompson. "Time series forecasting with neural network: A case study of stock prices of intercontinental bank Nigeria." IJRRAS (2011).

- **Rice yields:** Shabri, Ani, Ruhaidah Samsudin, and Zuhaimy Ismail. "Forecasting of the rice yields time series forecasting using artificial neural network and statistical model." Journal of Applied Sciences 9.23 (2009): 4168-4173.

- **Lake levels:** Chih-Chieh Young, Wen-Cheng Liu, and Wan-Lin Hsieh, “Predicting the Water Level Fluctuation in an Alpine Lake Using Physically Based, Artificial Neural Network, and Time Series Forecasting Models,” Mathematical Problems in Engineering, vol. 2015, Article ID 708204, 11 pages, 2015. doi:10.1155/2015/708204.

Chapter 3

Recurrent Neural Networks

EVER since I ran across my first Recurrent Neural Network (RNN), I have been intrigued by their ability to learn difficult problems that involve time series data. They have been successfully used for many problems including adaptive control, system identification, and most famously in speech recognition.

This chapter, outlines the core concepts surrounding RNNs. In particular, you will

- Gain an intuitive understanding of recurrent neural networks.
- Develop insight into the role and use of delay units.
- Understand the core idea behind parameter sharing, and backpropagation through time.
- Build recurrent neural networks to predict the whole bird spot price of chicken.

Understanding Recurrent Neural Networks

Unlike the feed-forward neural network discussed in chapter 2, RNNs contain hidden states which are distributed across time. This allows them to efficiently store a lot of information about the past. As with a regular feed-forward neural network, the non-linear dynamics introduced by the nodes allows them to capture complicated time series dynamics.

The Difference between Dynamic and Static Neural Networks

Neural networks can be classified into dynamic and static. Static neural networks calculate output directly from the input through feed-forward connections. For example, in a basic feedforward neural network, the information flows in a single direction from input to output. Such neural networks have no feedback elements. The neural networks we developed in chapter 2, are all examples of static neural networks.

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

A Visual Explanation of Recurrent Neural Networks

Two simple recurrent neural networks are illustrated in Figure 3.1. The core idea is that recurrent connections allow memory of previous inputs to persist in the network's internal state, and thereby influence the network's output.

At each time step, the network processes:

1. The input attributes (x_t);

2. Updates its hidden state via activation functions (h_t);
3. and uses it to make a prediction of its output (y_t).

These steps are similar to the feed-forward neural network we discussed in chapter 2. However, the value held in the delay unit is fed back to the hidden units as additional input.

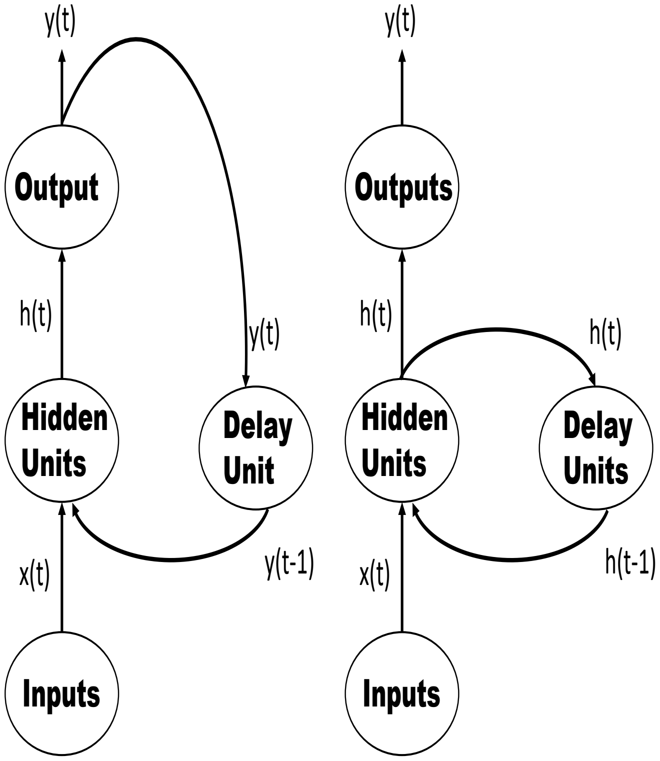


Figure 3.1: Two Simple Recurrent Neural Network Structures

A Quick Look at Turing complete

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

NOTE... ✍

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

The Role of the Delay Units

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

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

NOTE... ✍

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

Parameter Sharing Clarified

From Figure 3.2, we see that a RNN is constructed of multiple copies of the same network, each passing a message to a successor. It therefore shares the same parameters across all time steps because it performs the same task at each step, just with different inputs. This greatly reduces the total number of parameters required to learn relative to a traditional deep neural network which uses a different set of weights and biases for each layer.

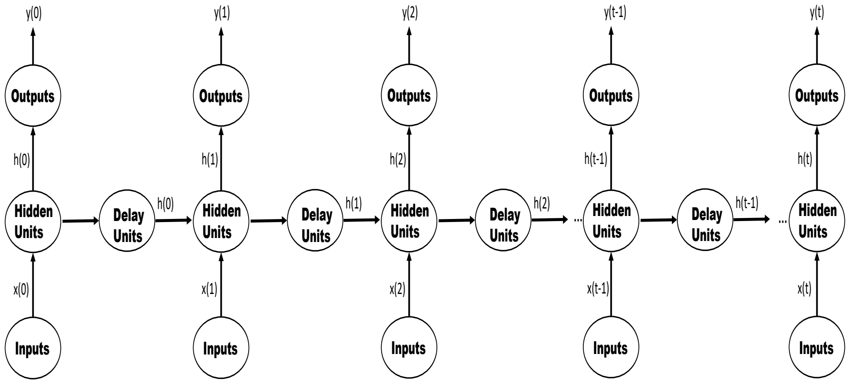


Figure 3.2: Unfolding a RNN

NOTE... ✍

The deep neural network discussed in chapter 2 had no sense of time or the ability to retain memory of their previous state. This is not the case for a RNN.

Backpropagation Through Time

It turns out, that with a few tweaks, any feed-forward neural network can be trained using the backpropagation algorithm.

For an RNN, the tweaked algorithm is known as Backpropagation Through Time.

The basic idea is to unfold the RNN through time. This sounds complicated, but the concept is quite simple. Look at Figure 3.2. It illustrates an unfolded version of Figure 3.1 (left network). By unfolding, we simply mean that we write out the network for the complete time steps under consideration.

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

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

NOTE... ✍

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

Practical Application of Recurrent Neural Networks

Before we get into the details of building our own model, let's take a quick look at some practical applications of RNNs.

Management of Phytoplankton Biomass

If you live near a fresh water lake or slow flowing river you are no doubt familiar with algal blooms. They spread rapidly covering vast areas. Whilst often pretty to look at, such blooms create stressful conditions in aquatic ecosystems. Prediction of blooms can assist in better management of freshwater systems.

Aquatic engineer D.K. Kim develop a recurrent neural network to predict one week ahead algal proliferation in the lower Nakdong River. The Nakdong River is the second largest river catchment area in South Korea. Around 10 million people live and work in the basin area.

The sample used in the analysis was made up of twenty-five features or attributes. The features were physicochemical and biological attributes such as water temperature, dams discharge, river flow, rainfall, zooplankton abundance, and nutrient concentration.

The sample was collected at a weekly frequency from June 1994 to December 2006. The training sample used data from June 1994 to 2003. In addition, a cross validation sample with data from 2004- 2005 data was also used. The data collected during 2006 formed the test set.

The recurrent neural networks had one hidden layer with between 1 to 5 nodes. The input attributes were lagged from one to twenty weeks; and the model was optimized over 5000 epochs.

River Flow Forecasting

Seasonal flooding is one of the most serious natural disasters faced by many inhabits of river basins, and low-lying countries. River flow forecasting is important for early flood warning, reservoir operations, and management of water resources. Accurate short term water forecasts result in economic benefits through more efficient irrigation allocations; improved information for recreational users; and helps to prevent unnecessary releases into a river system.

River flow forecasting is one of the most important topics in hydrology and water resources engineering. A team of civil engineers (D. Nagesh Kumar et al.) develop RNNs to forecast the monthly flow of the river Hemavathi in Karnataka state, India.

The researchers had access to 57 years of monthly data. It indicated that historical river flow ranges from 1.84 million cubic meters (M.cu.m.) in summer months, to 2894M.cu.m. in monsoon months.

The training sample consisted of the first 50 years of data, with the remaining 7 years used as the test set. The data was scaled into the [0,1] range before being fed into the RNN model.

The recurrent neural networks used time lagged river flow as features. In all, five time lagged features were used as input. The network had 2 hidden layers, with 10 nodes in each layer, and 3 delay units. The model was optimized over 50,000 iterations.

Forecasting Internet Round Trip Time

The internet round-trip time (RTT) or round-trip delay time, is time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received. In practice, such end to end communication is characterized by random delays. Predicting the expected length of delay is useful for network monitoring, routing protocol design, and flow control management.

Engineer Salem Belhaj design a RNN to predict RTT. Delay data was collected between two internet nodes over a two-week period. The input features consisted of the previous three RTT times. The model had 20 nodes in the hidden layer and was trained over 700 epochs.

Example - Forecasting the Whole Bird Spot Price of Chicken

Chicken is a popular food item in many countries across the world. In the United Kingdom, it accounts for almost half meat consumption; And Americans eat around 60 ponds of chicken per person every year.

The price of chicken is determined by a combination of available supply, and consumer demand. Price can fluctuate quite considerably over time. In this section, we build a RNN to predict the whole bird sport price of chicken.

Step 1 – Collecting Data

The data used to construct the model is stored in the dataframe `chicken` from the `astsa` package. The object `chicken`, contains the monthly whole bird spot price, Georgia docks, US cents per pound, from August 2001 to December 2015.

Let's load the data into R, and visualize it via a plot:

```
data("chicken",package ="astsa")
plot(chicken,
xlab="Date",
ylab="Price (cents)",
col="darkblue")
```

Figure 3.3 shows the resultant plot. Over the sample period, the price shows a marked upward trend, peaking at over 110 cents per pound.

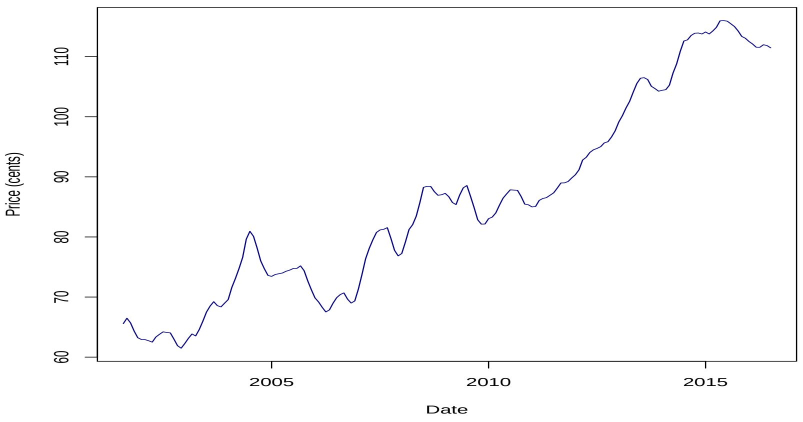


Figure 3.3: Monthly price of chicken

Step 2 – Exploring and Preparing the Data

Let's begin by transferring the observations in `chicken` to the R object `data`; And use the `quantmod` package (see page 48) to create four time lagged attributes:

```
data<-chicken
require(quantmod)
data<-as.zoo(data)
x1<-Lag(data, k = 1)
x2<-Lag(data, k = 2)
x3<-Lag(data, k = 3)
x4<-Lag(data, k = 4)
```

Much of this will be familiar to you. Notice, that for this example we use 4 features, with price lags ranging from 1 month to 4 months.

To make things a little easier to follow, we combine the attributes, and original price data into the R object `x`:

```
x<-cbind(x1,x2,x3,x4,data)
```

Natural logarithm and time lags

When dealing with price data, I often carry out analysis on the natural logarithm rather than the actual levels. This is probably a throwback from my Econometric roots, but let's do it anyway, and then take a look at the data using the `head` function:

```
x<-log(x)
head(round(x,2))
            Lag.1 Lag.2 Lag.3 Lag.4 data
2001(8)        NA    NA    NA    NA 4.18
2001(9)      4.18    NA    NA    NA 4.20
2001(10)     4.20  4.18    NA    NA 4.19
2001(11)     4.19  4.20  4.18    NA 4.16
2001(12)     4.16  4.19  4.20  4.18 4.15
2002(1)      4.15  4.16  4.19  4.20 4.14
```

There are two things to check here;

- First, that the lags are operating as expected. We see that `Lag.1` contains `data` lagged by 1 month. And this is as expected. A similar result holds for the remaining features.

- The second thing to check is that we have the correct number of `NA's` in each column. The object `Lag.4` has four missing values, and `Lag.1` has one missing value - this is as expected.

Remove missing values

Since everything checked out OK, we can remove the missing values and take a look at the result:

```
x <- x[-(1:4),]
head(round(x,2))
            Lag.1 Lag.2 Lag.3 Lag.4 data
2002(8)      4.16  4.16  4.15  4.14 4.16
```

```
2002(9)     4.16   4.16   4.16   4.15  4.16
2002(10)    4.16   4.16   4.16   4.16  4.14
2002(11)    4.14   4.16   4.16   4.16  4.13
2002(12)    4.13   4.14   4.16   4.16  4.12
```

All looks good. The numbers are as expected.

Preparing the data for input into the neural network

We can use the `range_data` function (see page 51 for further discussion) to scale the data:

```
x<-data.matrix(x)
range_data<-function(x) {(x-min(x))/(max(x)-
    min(x))}
min_data<-min(x)
max_data<-max(x)
x <-range_data(x)
```

The object `x` now contains the scaled observations. They all should lie in the range [0,1]. We can use the `max` and `min` functions to carry out a very quick check:

```
max(x)
 [1] 1

min(x)
 [1] 0
```

Yep, the numbers are as expected.

Creation of train and test samples

Let's make sure our data are in the appropriate matrix format:

```
x1<-as.matrix(x[,1])
x2<-as.matrix(x[,2])
x3<-as.matrix(x[,3])
x4<-as.matrix(x[,4])
y<-as.matrix(x[,5])
```

Our model will predict the last six months of prices. Since the data has 176 observations, this leaves 170 observations for the training sample:

```
n_train <- 170
y_train<-as.matrix(y[1:n_train])
x1_train<-as.matrix(t(x1[1:n_train ,]))
x2_train<-as.matrix(t(x2[1:n_train ,]))
x3_train<-as.matrix(t(x3[1:n_train ,]))
x4_train<-as.matrix(t(x4[1:n_train ,]))
```

Much of the above will be familiar to you by now. However, notice the feature data are transposed so that they are in the correct format for use in the `rnn` package.

A quick numbers check

We had better check the numbers align up as expected:

```
nrow(x)
[1] 176

ncol(x1_train)
[1] 170

ncol(x2_train)
[1] 170

ncol(x3_train)
[1] 170

ncol(x4_train)
[1] 170

length(y_train)
[1] 170
```

So, we see there are a total of 176 observations, of which 170 are used for the training sample. You should check for yourself that the remaining 6 observations are captured in the test set attributes and target variable.

Step 3 – Training a Model on the Data

The model is estimated using the `trainr` function in the `rnn` package. It requires the attribute data be passed as a 3 dimensional array. The first dimension is the number of samples, the second dimension time, and the third dimension captures the variables.

For the training set, we have one sample (first dimension) which consists of one hundred and seventy time points (second dimension) on four features (third dimension) .

The required 3 dimensional array can be constructed using the `array` and `dim` functions as follows:

```
x_train <- array( c(x1_train,
x2_train,
x3_train,
x4_train),
dim=c(dim(x1_train),4) )
```

This creates an array of doubles, the format required for the `trainr` function.

To check that `x_train` has the correct dimensions use the `dim` function:

```
dim(x_train)
[1]    1  170    4)
```

We see that the first dimension of `x_train` is 1 (as we only have one sample), the second dimension corresponds to the number of observations (time steps), and the third dimension correctly captures the number of features used.

Specifying the model

Our first model will have 3 nodes in the hidden layer, with a relatively low learning rate of 0.05, sigmoid activation function, and optimized over 500 epochs:

```
require(rnn)
set.seed(2018)
model1 <- trainr(Y = t(y_train),
X = x_train,
learningrate = 0.05,
hidden_dim = 3,
numepochs = 500,
network_type = "rnn",
sigmoid = "logistic")
```

The `set.seed` function is used to aid reproducibility. The RNN model is specified by setting `network_type = "rnn"`. The fully trained model is stored in the object `model1`.

NOTE... ✍

The examples in this chapter use version 0.8.0 of the `rnn` package.

Examining error by epoch

The error by epoch is accessed via the `$error` argument of the fitted model. Let's extract and plot it:

```
error_1<-t(model1$error)
rownames(error_1) <- 1:nrow(error_1)
colnames(error_1) <- "error"
plot(error_1)
```

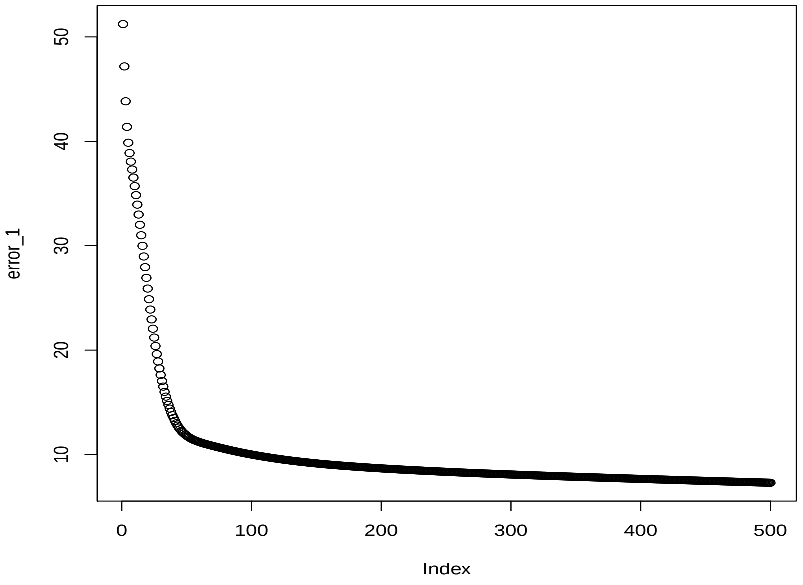


Figure 3.4: Error by epoch for `model1`

As shown in Figure 3.4, the error falls quite rapidly for the first 50 or so epochs. This indicates that the model is learning from the data. It then exhibits a more gentle decline to 500 epochs. If the model was unable to learn from the data, we might see a flat or even upward slopping line.

Step 4 – Evaluating Model Performance

The training set fitted values can be obtained using the `predictr` function. It takes the trained model and sample data as arguments. In this case, we pass it `model1`, and since we are interested in train set performance, the training set attribute data:

```
pred1_train <- t(predictr(model1, x_train))
```

We use the correlation coefficient to assess how closely the predicted values match the actual observations for the training sample:

```
round(cor(y_train,pred1_train),5)
           [,1]
[1,]  0.97937
```

The correlation is relatively high at close to 0.98. This is a decent start!
Let's plot the predicted and actual training set values via the `plot` function:

```
plot(y_train,pred1_train,ylab="pred1_train")
```

Figure 3.5 confirms the overall strong fit.

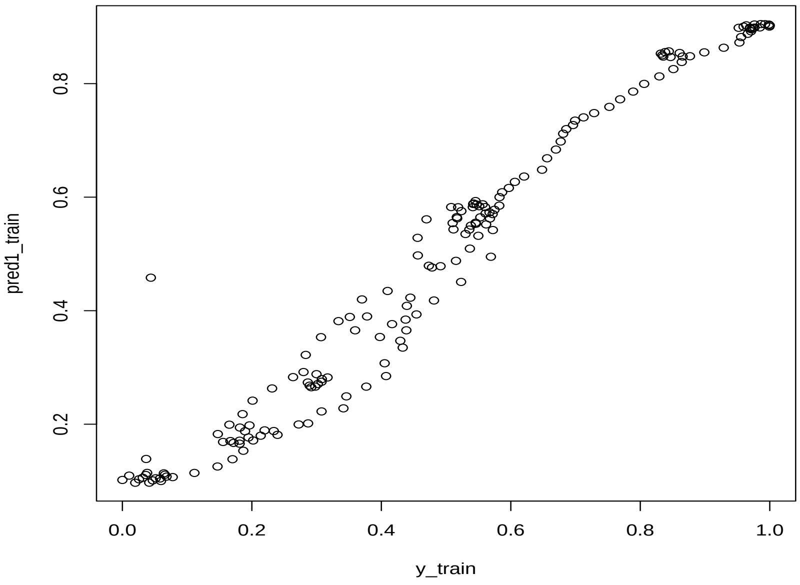


Figure 3.5: Training sample actual and fitted observations from `model1`

Step 5 – Improving Model Performance

We would like to get the correlation between the predicted and actual observations over 0.98 for the training sample. How can we achieve this?

One possible solution is to increase the number of nodes in the hidden layer. Along similar lines, we could also add an additional hidden layer to the model. Of course, this is not guaranteed to work, but let's see if it improves things here:

```
set.seed(2018)
model2 <- trainr(Y = t(y_train),
X = x_train,
learningrate = 0.05,
hidden_dim = c(3,2),
numepochs = 500,
network_type = "rnn",
sigmoid = "logistic")
```

Notice that `model2` has the same exact specification as `model1` with the exception that a second hidden layer with 2 nodes is added.

Now to estimate the training set correlation:

```
pred2_train <- t(predictr(model2, x_train))
round(cor(y_train,pred2_train),5)
            [,1]
[1,] 0.98555
```

Yes! `model2` has broken through the 0.98 barrier.

Preparing the test set data

Now let's see how each model did on the test set. First, put the data into a suitable format:

```
x1_test<-as.matrix(t(x1[(n_train+1):nrow(x1),]))
x2_test<-as.matrix(t(x2[(n_train+1):nrow(x2),]))
x3_test<-as.matrix(t(x3[(n_train+1):nrow(x3),]))
x4_test<-as.matrix(t(x4[(n_train+1):nrow(x4),]))
```

```
y_test<-as.matrix(y[(n_train+1):nrow(x4)])
```

NOTE... ✍

Remember the attributes/ features should be transposed. This is not the case for the target variable.

Next, create the 3 dimensional feature array, and check it has the appropriate dimensions:

```
x_test <- array( c(x1_test,
x2_test,
x3_test,
x4_test),
dim=c(dim(x1_test),4) )

dim(x_test)
[1] 1 6 4
```

The dimensions are as expected - one sample, with six time steps (months) on four features.

Creating the predictions

Now, we are ready to create the test set predictions for `model1` and `model2`:

```
pred1_test <- t(predictr(model1, x_test))
pred2_test <- t(predictr(model2, x_test))
```

This produces a forecast up to 6 months ahead, without the need to re-estimate the model for each new prediction. Such forecasts are often useful, particularly when it takes a significant amount of time to run/ update a model.

Unscaling the data

We want to view the predicted values in their original unscaled form. To do this we use the `unscale_data` function (see page 60). For easy plotting, the actual and predicted values are stored in the `ts` format.

First, let's work with `model1`:

```
unscale_data<-function(x,max_x,min_x)
{x*(max_x-min_x)+min_x}

pred1_actual<-unscale_data(pred1_test,
max_data,
min_data)

pred1_actual<-exp(pred1_actual)

pred1_actual<-ts(matrix(pred1_actual),
end=c(2016,7),
frequency=12)
```

The first line is a repeat of the `unscale_data` function for your convenience.

Now use the same approach for `model2`:

```
pred2_actual<-unscale_data(pred2_test,
max_data,
min_data)

pred2_actual<-exp(pred2_actual)

pred2_actual<-ts(matrix(pred2_actual),
end=c(2016,7),
frequency=12)
```

And for the actual observations:

```
y_actual<-unscale_data(y_test,
max_data,
min_data)

y_actual<-exp(y_actual)

y_actual<-ts(matrix(y_actual),
end=c(2016,7),
frequency=12)
```

Great! We have the data in the original scale. The next step is to combine all the predictions and actual values into a single dataframe called `result_all`:

```
result_all <-cbind(y_actual,
round(pred1_actual,2),
round(pred2_actual,2))
```

Now, add the appropriate column names and view the data:

```
colnames(result_all)<-c("actual",
"Model 1",
"Model 2")
result_all
          actual  Model 1  Model 2
Feb 2016  112.10   111.56   106.54
Mar 2016  111.56   109.38   111.80
Apr 2016  111.55   108.63   110.95
May 2016  111.98   108.32   110.54
Jun 2016  111.84   108.25   110.35
Jul 2016  111.46   108.21   110.24
```

Take a close look at the result. Although `model2` is a little off with the first prediction, it appears to more closely match the remaining actual prices.

Re-estimating the model for each new observation

Often, the performance of a neural network model can be improved by re-optimizing the weights as new observations become available. In our illustration, this will require the creation of six, one month ahead forecasts. This requires re-estimating the model six times. A simple `for` loop will do the job nicely!

Let's begin by setting up the core parameters:

```
start=n_train
end = 6
k=-1
```

Notice that:

- The parameter `start` contains the row value of the last month of the training sample. In this case, it takes the value 170.
- The parameter `end` contains the number of one step ahead forecasts, in this case 6.
- And `k` is a counter variable used to increment the attribute observations by 1 time step (month).

Loop overview

For ease of readability I have broken the loop down into several parts:

1. Open the for loop
2. Prepare the training data
3. Specify the model
4. Prepare the test data
5. Make predictions and close the for loop

Open the `for` loop: The `cat` function is used to print to screen the current step head forecast month the model is estimating:

```
for (i in start:((start+end)-1)) {
k=k+1
cat("Working forecast for month ",
n_train+k,
"\n")
```

Prepare the training data: As we have done previously, the data is transformed into the appropriate dimensions for input into the `trainr` function:

```
y_train_step<-as.matrix(y[1:(n_train+k)])
x1_train_step<-as.matrix(t(x1[1:(n_train+k) ,]))
x2_train_step<-as.matrix(t(x2[1:(n_train+k) ,]))
x3_train_step<-as.matrix(t(x3[1:(n_train+k) ,]))
x4_train_step<-as.matrix(t(x4[1:(n_train+k) ,]))

x_train_step <- array( c(x1_train_step,
x2_train_step,
x3_train_step,
x4_train_step),
dim=c(dim(x1_train_step),4) )
```

Notice we augment the attribute and target variable with `_step` to remind us that we are calculating the one step ahead forecast at each iteration of the loop.

Specify the model: We train a model with 2 nodes in a single hidden layer:

```
set.seed(2018)
model_step <- trainr(Y = t(y_train_step),
X = x_train_step,
learningrate = 0.05,
hidden_dim = 2,
numepochs = 500,
network_type = "rnn",
sigmoid = "logistic")
```

Prepare the test data: This step is familiar to you by now. We push the test set data into the appropriate format for use by the `predictr` function:

```
x1_test_step<-as.matrix(t(x1[(n_train+k+1),]))
x2_test_step<-as.matrix(t(x2[(n_train+k+1),]))
x3_test_step<-as.matrix(t(x3[(n_train+k+1),]))
x4_test_step<-as.matrix(t(x4[(n_train+k+1),]))

y_test_step<-as.matrix(y[(n_train+k+1)])

x_test_step <- array( c(x1_test_step,
x2_test_step,
x3_test_step,
x4_test_step),
dim=c(dim(x1_test_step),4) )
```

Make predictions and close the for loop: The one-step ahead predictions at each iteration are stored in the R object `forecast`:

```
pred_test <- t(predictr(model_step,
x_test_step))

if(i==start) forecast<-as.numeric(pred_test)

if (i>start) forecast<-cbind(forecast,
as.numeric(pred_test))

}#remember to close for loop!
```

Depending on the speed of your computer, the entire loop may take several minutes to run.

Unscale the predictions

Once the loop has finished unscale the predictions:

```
pred_actual<-unscale_data(forecast,
max_data,
min_data)

pred_actual<-exp(pred_actual)
pred_actual<-ts(matrix(pred_actual),
end=c(2016,7),
frequency=12)
```

Combining with previous results

Now, let's combine the above with the predictions from `model1` and `model2`:

```
result_all<-cbind(result_all,
round(pred_actual,2) )
colnames(result_all)<-c("actual",
"Model 1",
"Model 2",
"One step")
```

The R object `result_all` now contains all three model predictions. Let's take a look:

```
result_all
         actual Model 1 Model 2 One step
Feb 2016 112.10  111.56  106.54   112.07
Mar 2016 111.56  109.38  111.80   112.14
Apr 2016 111.55  108.63  110.95   112.20
May 2016 111.98  108.32  110.54   112.25
Jun 2016 111.84  108.25  110.35   112.33
Jul 2016 111.46  108.21  110.24   112.40
```

The one step ahead predictions are closer to the actual observed prices. This is typical of one step ahead forecasts. They tend to be more accurate because they use more recent observations/-data/information. Of course, they take considerably longer to run, since for each new observation, the model is re-estimated. However, in this case, it is worth the additional effort.

Summary

After reading this chapter you should know the core concepts surrounding recurrent neural networks, and be able to apply them using the `rnn` package to your own time series data. We saw that the use of one step ahead forecasts with the model re-estimated at each time step can help squeeze out additional performance.

In the next chapter, we look at a neural network specifically designed to handle long sequences of data - the Long Short-Term Memory Recurrent Neural Network.

Suggested Reading

To find out more about the practical applications discussed in this chapter see:

- **Management of Phytoplankton Biomass**: Kim, D. K., et al. "Machine learning for predictive management: short and long term prediction of phytoplankton biomass using genetic algorithm based recurrent neural networks." International Journal of Environmental Research 6.1 (2011): 95-108.

- **River Flow Forecasting:** Kumar, D. Nagesh, K. Srinivasa Raju, and T. Sathish. "River flow forecasting using recurrent neural networks." Water resources management 18.2 (2004): 143-161.

- **Forecasting Internet Round Trip Time:** Belhaj, Salem, and Moncef Tagina. "Modeling and prediction of the internet end-to-end delay using recurrent neural networks." Journal of Networks 4.6 (2009): 528-535.

Chapter 4

Long Short-Term Memory Recurrent Neural Network

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

This chapter introduces the LSTM, and explains how you can quickly build one for time series analysis in R. By the end of the is chapter, you will understand:

- How a LSTM differs from the RNN of the previous chapter.
- The role of gates and memory blocks.
- How others have used LSTM for time series forecasting.
- How to build a LSTM model in R.

Understanding Long Short-Term Memory Recurrent Neural Networks

Figure 4.1 outlines a simple schematic of an LSTM. It is similar to the recurrent network we saw in Figure 3.1. However, it replaces the hidden units with memory blocks.

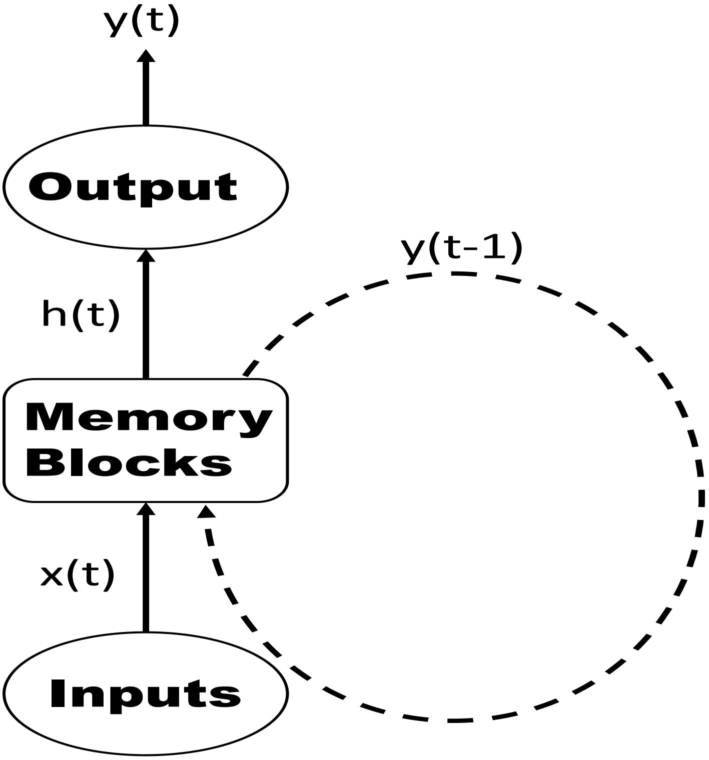


Figure 4.1: Schematic of a LSTM

The LSTM Memory Block in a Nutshell

When I first came across the LSTM memory block, it was difficult to look past the complexity. Figure 4.2 illustrates a simple LSTM memory block with only input, output, and forget gates. In practice, memory blocks may have even more gates!

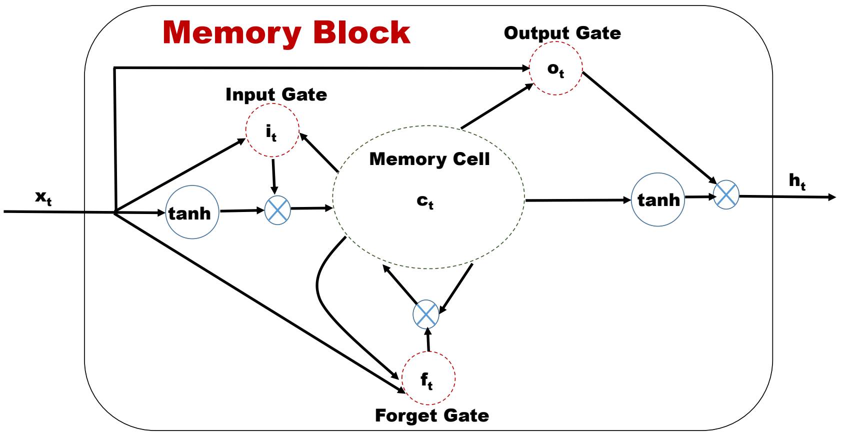


Figure 4.2: Simple memory block

The key that unlocks the door

The key to an intuitive understanding of Figure 4.2 is that:

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

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

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

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

With the above four point in mind, take another look at Figure 4.2; you should be able to get an intuitive feel for what is going on.

The Role of Gates

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

About each gate

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

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

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

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

NOTE... ✍

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

Understand the Constant Error Carousel

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

NOTE... ✍

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

Hard Sigmoid function

The hard sigmoid function, see Figure 4.3, is often used as the inner cell activation function. It is piece-wise linear, and faster to compute than the sigmoid function which it approximates.

The hard Sigmoid function can be calculated as:

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

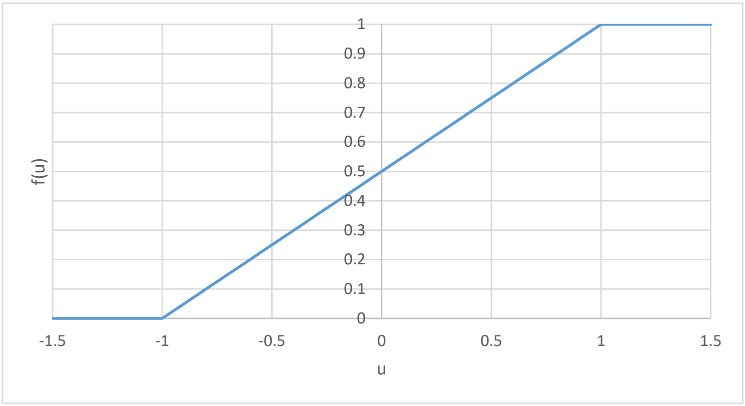


Figure 4.3: Hard Sigmoid activation function

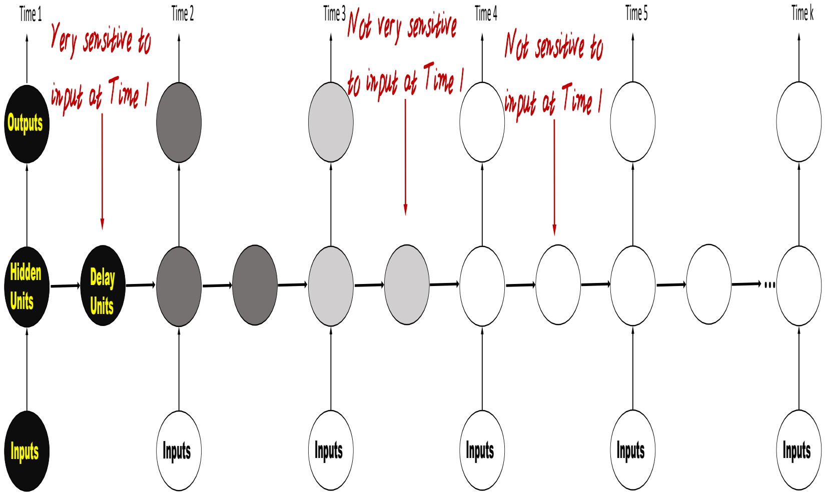
NOTE... ✍

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

A Note on Vanishing Gradients

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

Figure 4.4 illustrates the situation. The shading of the nodes in the unfolded network indicates their sensitivity to the original input at time 1. The darker the shade, the greater the sensitivity. Over time this sensitivity decays.



Time 1
Time 2
Time 3
Time 4
Time 5
Time k
Very sensitive to input at Time 1
Not very sensitive to input at Time 1
Not sensitive to input at Time 1
Outputs
Hidden Units
Delay Units
Inputs

Figure 4.4: Vanishing Gradient's in a RNN

The cause

It turns out that the evolution of the back-propagated error over time depends exponentially on the size of the weights. This is because the gradient is essentially equal to the recurrent weight matrix raised to a high power. Essentially, the hidden state is passed along for each iteration, so when back-propagating, the same weight matrix is multiplied by itself multiple times.

When raised to high powers (i.e. iterated over time) it causes the gradient to shrink (or grow) at a rate that is exponential in the number of time-steps. This is known as the “*vanishing gradient*” if the gradients shrink, or “*exploding gradients*” if the gradient grows.

The practical implication is that learning long term dependencies in the data via a simple RNN can take a prohibitive amount of time, or may not happen at all.

The solution

The LSTM avoids the vanishing gradient problem. This is because the memory block has a cell that stores the previous values and holds onto it unless a "forget gate" tells the cell to forget those values. Only the cell keeps track of the model error as it flows back in time. At each time step the cell state can be altered or remain unchanged. The cell's local error remains constant in the absence of a new input or error signal. The cell state can also be fed into the current hidden state calculations. In this way, the value of hidden states occurring early in a sequence can impact later observations.

NOTE... ✍

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

Practical Application of Long Short-Term Memory Recurrent Neural Networks

Understanding how others have used the LSTM in a time series context can help you develop your ideas. Three excellent examples include their use for effective mind control, anomaly detection, and mortality risk prediction. In this section, we explore each of these practical applications.

Effective Mind Control

Every time you have a thought your brain produces electrical signals corresponding to that thought. Electroencephalography (EEG) is the discipline of recording and then deciphering your minds electrical signals.

In the decades old movie Control Factor. Lance Bishop, an insurance salesman, becomes an involuntary subject in a top secret, and totally unethical government mind-control project. When his mind is ordered to kill his wife, he begins to fight back. Lance Bishop, was the unwitting subject of a devious electroencephalography plot!

Although Electroencephalography has been around since the 1920s, recent developments have made it possible for individuals to use their "brain waves" to control mechanical devices, and to communicate without the need to engage their muscle systems. Mind control, once science fiction, is now science fact!

A group of Australian scholars (Xiang el al.) developed a 7 layer deep LSTM model to classify raw EEG signals.

The researchers had access to a time series of around 26.4 million EEG measurements collected from 109 individuals. They used 28,000 samples from one individual to evaluate the performance of their model.

Five states or labels made up the classification problem. These were:

1. eye closed;
2. focus on left fist;
3. focus on right fist;
4. focus on both fists;
5. and focus on both feet.

The model achieved 95% accuracy. Now, here is the really amazing thing, the model was able to achieve this on raw data, without any preprocessing or feature engineering!

Anomaly Detection

The Large Hadron Collider (LHC) is located on Switzerland and France border. It is the worlds largest experimental instrument. Its superconducting magnets require careful monitoring in order to avoid malfunctions and failures.

Researcher Maciej Wielgosz develop a LSTM model to detect anomalies in the superconducting magnets. They create a single layer LSTM with 128 nodes. The model was run over 20 epochs and took around 2½ hours to train. It delivered 99% accuracy on the sample time series data.

Mortality Risk Predictions

In an innovation application of the LSTM model, Melissa Aczony analyzed electronic medical records collected over time by a Pediatric Intensive Care Unit. The LSTM model was designed to use this information to make mortality predictions.

The sample period began in March 2002 and ended in March 2016. It consisted of 12,020 patients. Time series features included physiological observations, lab work, interventions and drugs administered. A total of 75% of the patients were randomly selected for the training set.

In total, 382 features were fed into a two hidden layer LSTM model. Each hidden layer had 256 nodes. Melissa Aczony found that the RNN model outperformed traditional logistic regression, and also beat out a feed-forward neural network.

Example - Predicting Eye Movements

The dataframe `crqa` contains time series observations on the eye movements of a speaker and listener. In this section, we build a LSTM model to capture the eye movement dynamics of the listener.

Step 1 – Collecting Data

First, load the data:

```
data("crqa",package ="crqa")
```

Step 2 – Exploring and Preparing the Data

The listener eye movements are stored in the object `RDts1`:

```
typeof(RDts1)
[1] "list"

nrow(RDts1)
[1] 2000
```

The object `RDts1` is a list object containing 2000 observations. What do the observations look like? Take a look using the head and tail argument:

```
head(RDts1)
  V1
1 10
2  2
3  2
4 10
5 10
6 10

tail(RDts1)
     V1
```

```
1995    1
1996    1
1997    1
1998    1
1999    1
2000    1
```

Eye movement is measured as a series of integers representing the fixation position of the listener. Let's explore how many unique fixation positions are contained in the sample. We can do this visually using a combination of the `table` and `barplot` functions:

```
barplot(table(RDts1),
ylab="Count",
col="darkblue")
```

Figure 4.5 shows the result. We see, there are a total of seven fixation positions recorded in the sample (positions 1,2,3,4,5,6 and 10).

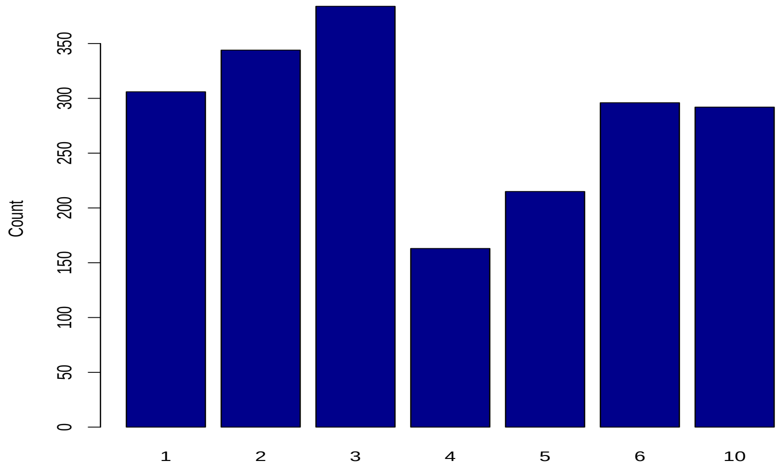


Figure 4.5: Bar plot of eye fixation positions

Time series plot

Now, let's use the plot function to see what the eye movement positions look like over time:

```
plot(as.ts(RDts1),
col="darkblue",
ylab="Position")
```

Figure 4.6 indicates quite large swings in terms of position. It would be quite challenging to build a traditional statistical model to capture this type of dynamic.

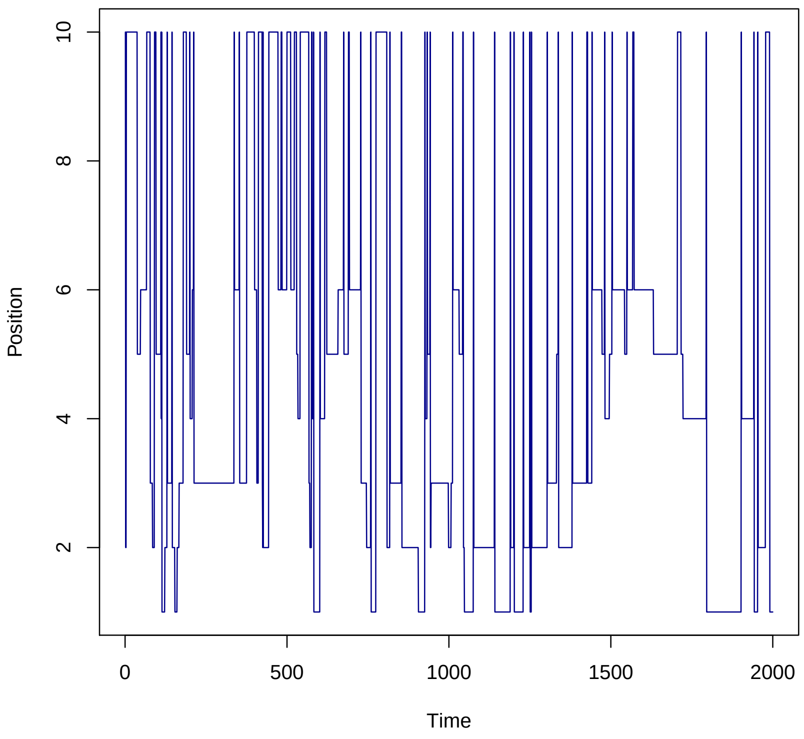


Figure 4.6: Time series plot of eye movement positions

Creation of sequences

We won't use all the sample data. Instead, the first 300 observations are used to train the model. We then use the model to predict the subsequent 100 eye movements.

To illustrate the use of different sequences with a LSTM model, the sample is broken into three sequences. Each sequence has 100 observations The sequences are combined into an R object called `data`:

```
seq_1<-RDts1[1:100,]
seq_2<-RDts1[101:200,]
seq_3<-RDts1[201:300,]
data<-cbind(seq_1,seq_2,seq_3)
```

Our goal is to use this data to predict the next eye movement:

```
lab_1<-RDts1[2:101,]
lab_2<-RDts1[102:201,]
lab_3<-RDts1[302:301,]
lab<-cbind(lab_1,lab_2,lab_3)
```

Notice the `lab_` objects contain the next eye position for each of the three sequences.

NOTE... ✍

The `lab_` series will be used as target variable labels in the subsequent analysis.

Putting the data into a suitable format

Now, we need to prepare the data for input into the LSTM model. We will use the `mxnet` package. This requires the data and target variable to be passed to it in specific 2 dimensional format. The first dimension is the number of sequences (in this case 3), and the second dimension measures the number of examples or observations (in this case 100):

```
data_set = aperm((array(c(data),
dim = c(100,3))))

labels_set= aperm(array(c(lab),
dim = c(100,3)))
```

The object `data_set` contains the training data, and `labels_set` the target observations.

We better check the dimensions of each of these objects. We can use the `dim` function to do this:

```
dim(data_set)
[1]    3  100

dim(labels_set)
[1]    3  100
```

Both objects have the correct dimensions for our analysis.

Pass data as a list

The `mxnet` package requires data to be passed to the model as a list. Here is how to take `data_set` and `labels_set`, and combine them into the appropriate form:

```
trainDat <- list(data=data_set,
label=labels_set)
```

The object `trainDat` now contains the data in a format suitable for passing to our LSTM model in the package `mxnet`. Use the `summary` function for an overview:

```
summary(trainDat)
        Length  Class   Mode
data    300     -none-  numeric
label   300     -none-  numeric
```

Step 3 – Training a Model on the Data

We are ready to specify a few of the parameters for our model. Let's do that ahead of calling the `mx.lstm` function:

```
batch_size = 1
seq.len = 3
num.hidden = 25
num.embed =2
num_layer =1
num_round = 2
update.period = 1
learning.rate= 0.3
num_labels=7
input_size=7
moment=0.9
```

Let's take a moment to review some of the above.

- We specify a LSTM with 25 nodes in the hidden layer;
- and use a learning rate of 0.3, with momentum of 0.9.
- For illustration, we will run the model over 2 iterations (see what happens if you increase this number). This is specified in the `num_round` parameter.
- Also, notice that `num_labels` corresponds to the number of eye positions in the sample.

Specify the LSTM

Using the above parameters we can specify the model:

```
require(mxnet)
mx.set.seed(0)

model <- mx.lstm(trainDat,eval.data = NULL,
ctx = mx.cpu(),
num.round = num_round,
update.period = update.period,
num.lstm.layer=num_layer,
seq.len=seq.len,
num.hidden=num.hidden,
num.embed=num.embed,
num.label=num_labels,
batch.size=batch_size,
input.size=input_size,
initializer=mx.init.uniform(0.1),
learning.rate=learning.rate,
momentum=moment)
```

Step 4 – Evaluating Model Performance

Next, we create an inference model. This is required in order to generate forecasts later in our analysis. It is created by passing the parameters of `model`, and several of our previously specified parameters to the `mx.lstm.inference` function as follows:

```
mx.set.seed(2018)
pred<-mx.lstm.inference(num.lstm.layer =
    num_layer,
input.size = input_size,
num.hidden = num.hidden,
num.embed = num.embed,
num.label = num_labels,
ctx = mx.cpu(),
arg.params = model$arg.params)
```

Notice that we reference the parameters of `model` via `$arg.params`. The object `pred` contains the inference model.

Using the `mx.lstm.forward` function

Now, the `mx.lstm.forward` function is used to predict future eye movements using lstm inference model `pred`. For example, to obtain the prediction for observation 301, you would use:

```
model_probs <- mx.lstm.forward(pred,
RDts1[301,1])
```

The object `model_probs` is a list that contains the probability for each of the seven potential eye positions. To access these probabilities append the `$prob` argument as follows:

```
model_probs$prob
                 [,1]
[1,]  2.193601e-04
[2,]  2.540698e-08
[3,]  3.541170e-04
[4,]  9.701149e-01
[5,]  1.352675e-06
[6,]  3.446032e-05
[7,]  2.927572e-02
```

The probability for each of the seven eye positions is reported. Your probabilities may look somewhat different from those shown. I've found when using the `mxnet` package that there can be some variation between forecasts on different machines / and or runs of the model. In this example, the largest probability at around 0.97, is for position 4. What was the actual eye position one time unit later?

```
RDts1[302,1]
[1]  3
```

Arrrgh! Off by one position!

NOTE... ✍

If you encounter this error:

```
Error in mx.ctx.default() :
object 'mx.ctx.internal.default.
    value' not found
```

Add the following before you call the `mx.lstm.forward` function:

```
mx.ctx.internal.default.value = list
    (device="cpu",
device_id=0,
device_typeid=1)
mx.ctx.internal.default.value
class(mx.ctx.internal.default.value)
    = "MXContext"
```

Step 5 – Viewing Training Set Performance

Rather than look at the probabilities for each example one by one, we want to use them collectively to predict the actual eye position. This can be achieved using a simple `for` loop. For this illustration, we will use our model to predict a time sequence of 100 eye positions.

Key variables

To begin, we define some of the key variables:

```
probs<-1:700
eye_class<-1:100
dim(probs)<-c(num_labels,100)
```

The object `probs` will contain the eye fixation position probabilities. There are seven eye positions and 100 observations to predict. Therefore, we create `probs` as a two dimensional array with 7 rows and 100 columns. The object `eye_class`, will contain the actual eye position predictions.

The for loop

Here is the for loop:

```
for (i in (1:100)) {
temp=as.numeric(RDts1[i+302,1])
mx.set.seed(2018)
model_prob <- mx.lstm.forward(pred,
temp , FALSE)
prob=as.array(model_prob$prob)
eye_class[i] <- which.max(prob)
}
```

Let's take a moment to walk through this code.

1. The current eye position is stored in the object `temp`.

2. It is passed with the inference model (`pred`) to the `mx.lstm.forward` function.

3. The actual estimated eye probabilities are stored in `prob`. The `which.max` function is used to retrieve the eye position with the largest probability. This value is stored in the object `eye_class`.

Viewing the results

Our final step is to collect together the predictions and actual observations, and display them as a plot. Here is how to convert the `eye_class` values into a suitable format:

```
eye_class<-as.data.frame(eye_class)
eye_class[,1][eye_class[,1] == 7] <- 10
final_pred<-as.ts(eye_class[,1])
final_pred<-as.numeric(unlist(final_pred))
```

```
final_pred<-ts(matrix(final_pred),
start=c(1),
end=c(100),
frequency=1)
```

Most of the above should be familiar to you. Just note:

1. The second line is used to ensure the 7th class has the label "10". This is to ensure comparability with the actual observations.

2. The object `final_pred` contains the predictions in the format we wish to use.

For illustrative purposes, we combine the predictions and the actual observed eye positions into a single dataframe called `result` which we then plot:

```
result<-cbind(RDts1[303:402,1],final_pred)
plot(result)
```

I've found when using the `mxnet` package that there can be some variation between forecasts on different machines / and or runs of the model. Your plot may look similar to Figure 4.7 or possibly Figure 4.8.

NOTE... ✍

Given the inherent variability in neural networks, you might run the model several times, and use the median count of the largest probability as the predicted class.

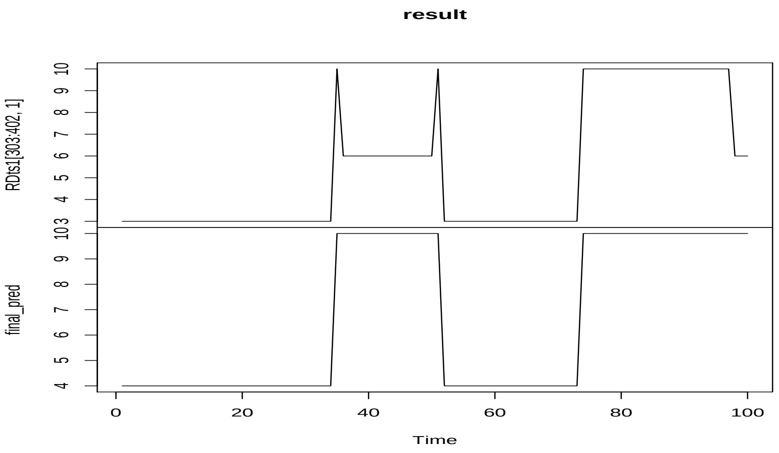


Figure 4.7: Eye movement actual (top) and predicted values (bottom)

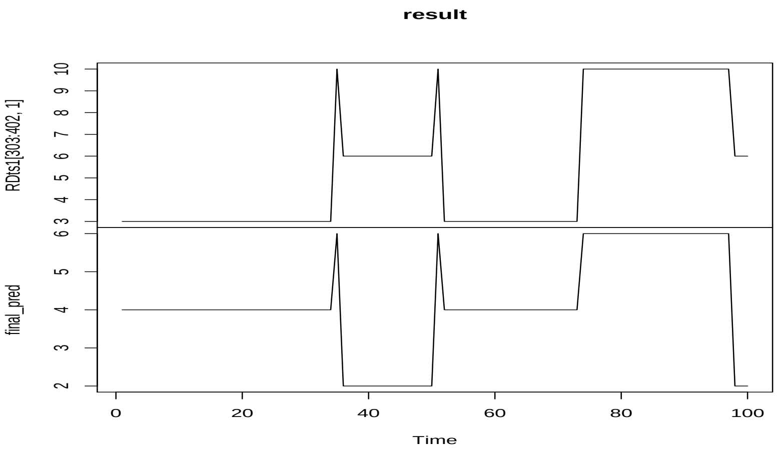


Figure 4.8: Eye movement actual (top) and predicted values (bottom)

Example - Forecasting Electrocardiogram Activity

Let's build another LSTM example, this time using electrocardiogram data. An electrocardiogram (ECG) measures the electrical activity of the heart muscle as it changes with time. Although a relatively old technology, it remains a useful tool to measure health, and for disease detection.

In this section, we build a LSTM model to forecast ECG activity.

Step 1 – Collecting Data

The `wmtsa` package contains a 2048 point ECG data series in the R object `ecg`. Let's load and plot the data:

```
require("wmtsa")
plot(ecg,xlab="Date",
ylab="Signal",
col="darkblue")
```

NOTE... ✍

The ECG data time-series represents approximately 15 beats of a normal human cardiac rhythm. It was sampled at 180 Hz measured in units of millivolts.

Figure 4.9 shows the resultant plot.

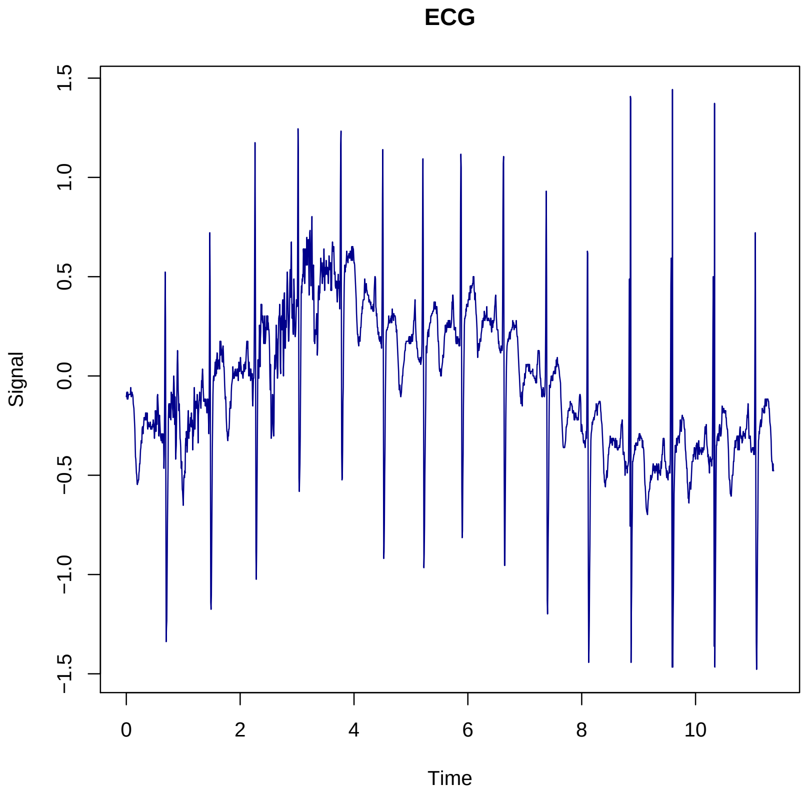


Figure 4.9: ECG data from the `wmtsa` package

Step 2 – Exploring and Preparing the Data

The data in `ecg` is stored as a `signalSeries` class:

```
class(ecg)
[1] "signalSeries"
attr(,"package")
[1] ".GlobalEnv"
```

We need to convert the data in `ecg` into numeric values. Whilst we are at it, let's also create four time lagged attributes using the `quantmod` package:

```
data<-as.numeric(ecg)
require(quantmod)
data<-as.zoo(data)
x1<-Lag(data, k = 1)
x2<-Lag(data, k = 2)
x3<-Lag(data, k = 3)
x4<-Lag(data, k = 4)
x<-cbind(x1,x2,x3,x4,data)
```

The object `x` contains the combined actual and lagged observations. These will serve as our input attributes (`x1,x2,x3,x4`) and target variable (`data`).

Scale data and remove missing observations

Now, take a look at `x` using the `head` function:

```
head(round(x,3))
    Lag.1   Lag.2   Lag.3   Lag.4    data
1      NA      NA      NA      NA  -0.105
2  -0.105      NA      NA      NA  -0.093
3  -0.093  -0.105      NA      NA  -0.082
4  -0.082  -0.093  -0.105      NA  -0.116
5  -0.116  -0.082  -0.093  -0.105  -0.082
6  -0.082  -0.116  -0.082  -0.093  -0.116
```

The next step is to remove the missing values, and scale the data using our custom `range_data` function:

```
x <- x[-(1:4),]
x<-data.matrix(x)
range_data<-function(x) {(x-min(x))/(max(x)-
    min(x))}
min_data<-min(x)
max_data<-max(x)
x <-range_data(x)

head(round(x,3))
   Lag.1  Lag.2  Lag.3  Lag.4   data
5   0.466  0.478  0.474  0.470  0.478
6   0.478  0.466  0.478  0.474  0.466
7   0.466  0.478  0.466  0.478  0.470
8   0.470  0.466  0.478  0.466  0.474
9   0.474  0.470  0.466  0.478  0.474
10  0.474  0.474  0.470  0.466  0.474
```

Create train and test sets

We will use the first 1950 observations for the training sample, with the remainder used for the test set:

```
x1<-as.matrix(x[,1])
x2<-as.matrix(x[,2])
x3<-as.matrix(x[,3])
x4<-as.matrix(x[,4])
y<-as.matrix(x[,5])

n_train <- 1950
y_train<-as.matrix(y[1:n_train])
x1_train<-as.matrix(t(x1[1:n_train,]))
x2_train<-as.matrix(t(x2[1:n_train,]))
x3_train<-as.matrix(t(x3[1:n_train,]))
x4_train<-as.matrix(t(x4[1:n_train,]))
```

We better check to ensure our training data has the correct number of rows/ observations:

```
ncol(x1_train)
[1] 1950

ncol(x2_train)
[1] 1950

ncol(x3_train)
[1] 1950

ncol(x4_train)
[1] 1950

length(y_train)
[1] 1950
```

Yep, all is as expected.

Step 3 – Training a Model on the Data

We will try the LSTM model from the package `rnn`. To use it, as we saw on page 80, requires the data be transformed into a suitable format. The first dimension refers to the number of sequences/ samples. In this case, we have one sample. The second dimension refers to the time - 1950 time units for this case. And the third dimension refers to the number of features to be fed into the model, in this case four time lagged variables:

```
x_train <- array( c(x1_train,
x2_train,
x3_train,
x4_train),
dim=c(dim(x1_train),4) )
```

```
dim(x_train)
[1]      1 1950      4
```

The shape of `x_train` is as expected.

Specifying the model

The initial model will have 3 nodes in the hidden layer, a relatively low learning rate of 0.05, and run over 300 epochs using a tanh activation function:

```
require(rnn)
set.seed(2018)
model1 <- trainr(Y = t(y_train),
X = x_train,
learningrate = 0.05,
hidden_dim = 3,
numepochs = 300,
network_type = "lstm",
sigmoid = "tanh")
```

Step 4 – Evaluating Model Performance

Let's take a look at the error by epoch:

```
error_1<-t(model1$error)
rownames(error_1) <- 1:nrow(error_1)
colnames(error_1) <- "error"

plot(error_1,
ylab="Training Error",
xlab="epochs")
```

Figure 4.10 shows a rapid decline in the error with each epoch. This is indicative that the model is learning from the available data.

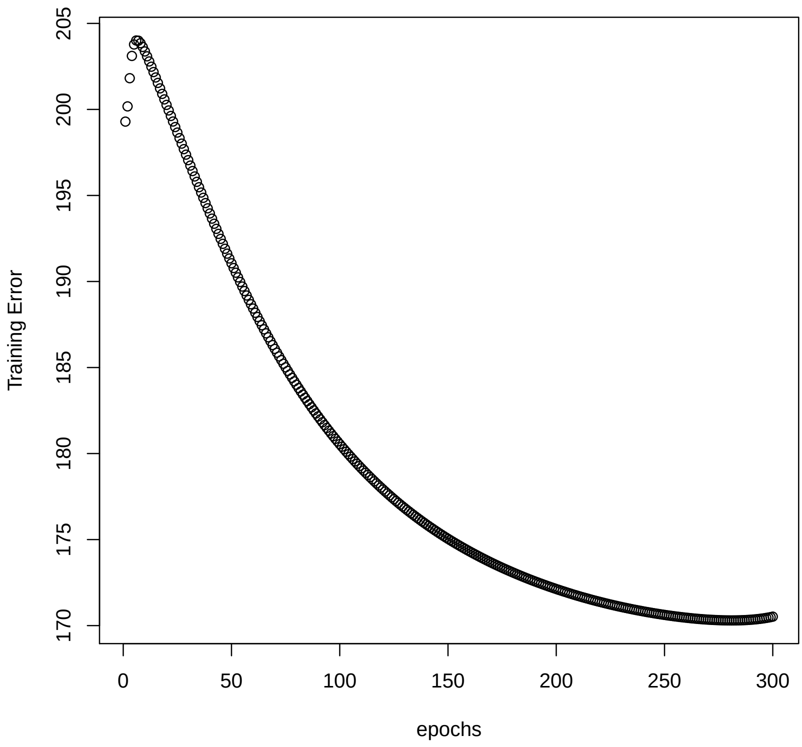


Figure 4.10: Error by epoch for ECG LSTM model

Getting the predicted values

The `predictr` function is used to obtain the model predictions. We asses performance on the training set using the correlation coefficient, and also the root mean squared error:

```
pred1_train <- t(predictr(model1, x_train))
require(Metrics)

round (rmse(y_train,pred1_train),3)
[1] 0.115
```

```
round(cor(y_train,pred1_train),3)
        [,1]
[1,]  0.701
```

Both numbers indicate a reasonable fit to the data. Visual inspection can help confirm this. So, plot the predictions as a time series, and visually assess how well they capture the dynamics of an ECG:

```
plot(as.ts(pred1_train),ylab="Signal")
```

Figure 4.11 indicates that the model does a credible job at capturing the dynamics of the data.

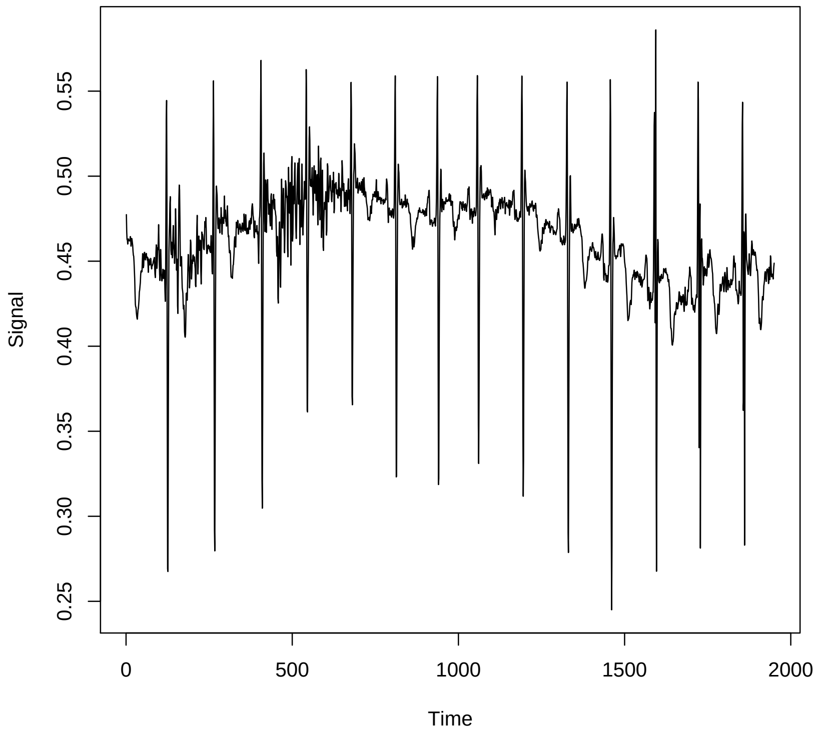


Figure 4.11: ECG estimated from the training sample

Step 5 – Assessing Test Set Performance

Since the training set performance looks very promising, let's see how the model actually performs with the test set data. First, prepare the data into a suitable format. The following code is familiar to you by now (see page 78 for further discussion):

```
x1_test<-as.matrix(t(x1[(n_train+1):nrow(x1) ,]))
x2_test<-as.matrix(t(x2[(n_train+1):nrow(x2) ,]))
x3_test<-as.matrix(t(x3[(n_train+1):nrow(x3) ,]))
x4_test<-as.matrix(t(x4[(n_train+1):nrow(x4) ,]))
y_test<-as.matrix(y[(n_train+1):nrow(x4)])

x_test <- array( c(x1_test,
x2_test,
x3_test,
x4_test),
dim=c(dim(x1_test),4)  )
dim(x_test)
[1]   1 94   4
```

The above should be familiar to you. The test set consists of 1 sample, with 94 time steps, and 4 input features.

The `predictr` function is used to perform the predictions:

```
pred1_test <- t(predictr(model1,
x_test))
```

Unscaling data

The R object `pred1_test` contains the prediction. However, we want to look at the data in the original scale. This is achieved via the `unscale_data` function:

```
unscale_data<-function(x,max_x,min_x)
{x*(max_x-min_x)+min_x}

pred1_actual<-unscale_data(pred1_test,
max_data,
min_data)
pred1_actual<-exp(pred1_actual)
pred1_actual<-ts(matrix(pred1_actual),
end=c(2016,7),
frequency=12)



y_actual<-unscale_data(y_test,
max_data,
min_data)
y_actual<-exp(y_actual)
y_actual<-ts(matrix(y_actual),
end=c(2016,7),
frequency=12)
```

Visual inspection

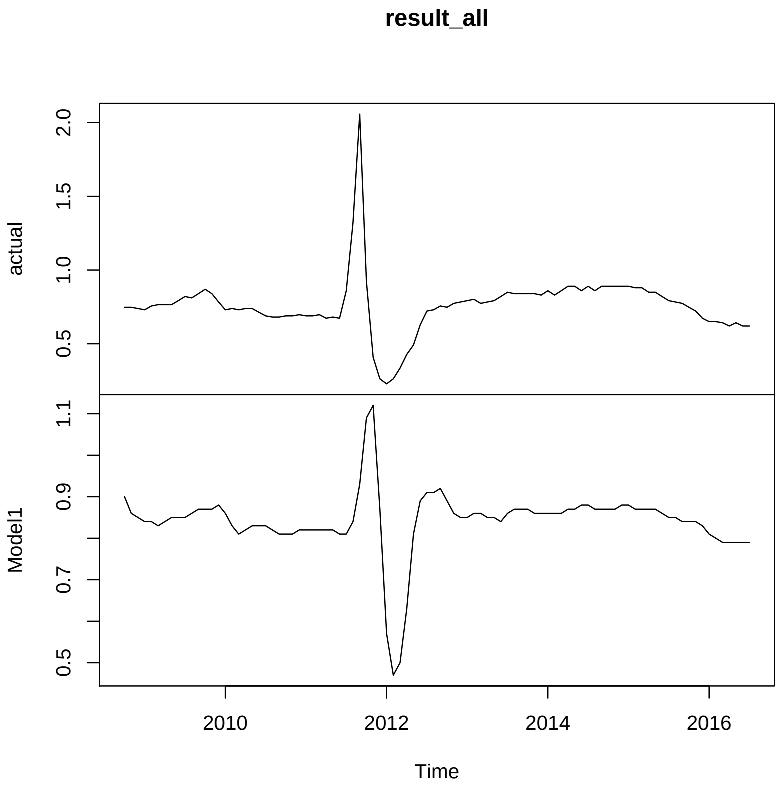
Now that we have the data in a suitable form, let's take a visual look at how well the test set predictions capture the dynamics of the original data:

```
result_all <-cbind(y_actual,
round(pred1_actual,2))

colnames(result_all)<-c("actual",
"Model1")

plot(result_all)
```

Figure 4.12 shows the resultant plot. Our initial model appears to capture the dynamics of the data, including the peaks and dips. Of course, with careful refinement, it could be further improved.



T

Figure 4.12: ECG actual (top) and predictions (bottom) from the test set

Summary

This chapter presented an overview of the LSTM model and introduced several R tools you can use to estimate this model. The LSTM neural network has delivered state-of-the-art performance in numerous tasks such as speech and handwriting recognition. As we have seen, it is also a useful tool for time series forecasting.

However, like many things in data science, there is no guarantee that this model will always deliver the best, or even acceptable performance on your data. The LSTM is only one of many emerging types of recurrent neural network useful for time series forecasting. In the next chapter, we introduce another popular model, the Gated Recurrent Unit Neural network.

Suggested Reading

To find out more about the practical applications discussed in this chapter see:

- **Mind Control:** Zhang, Xiang, et al. "Enhancing Mind Controlled Smart Living Through Recurrent Neural Networks." arXiv preprint arXiv:1702.06830 (2017).

- **Anomaly Detection:** Wielgosz, Maciej, Andrzej Skoczeń, and Matej Mertik. "Recurrent Neural Networks for anomaly detection in the Post-Mortem time series of LHC superconducting magnets." arXiv preprint arXiv:1702.00833 (2017). Forecasting Electrocardiogram Activity

- **Mortality Risk Predictions:** Aczon, Melissa, et al. "Dynamic Mortality Risk Predictions in Pediatric Critical Care Using Recurrent Neural Networks." arXiv preprint arXiv:1701.06675 (2017).

Chapter 5

Gated Recurrent Unit Neural Networks

OVER the past few years, the Gated Recurrent Unit (GRU) has emerged as an exciting new tool for modeling time-series data. They have fewer parameters than the LSTM, but often deliver similar or superior performance.

This chapter covers the GRU, and discusses some recent applications relevant for time series forecasting. By the end of this chapter you will learn:

- About the structure of a GRU.
- How they differ from an LSTM and RNN.
- How to use a GRU to perform time series prediction.

We will begin by examining, intuitively, the structure of a GRU, followed by a look at several applications. Then, we apply the GRU model to the eye movement data introduced in chapter 4.

Understanding Gated Recurrent Unit Neural Networks

Just like the LSTM, the GRU controls the flow of information, but without the use of a memory unit. Rather than a separate cell state, the GRU uses the hidden state as memory. It also merges the forget and input gates into a single "update" gate.

Figure 5.1 illustrates the topology of a GRU memory block (node). It contains an update gate (z) and reset gate (r). Here is how to think about each of these elements:

- The reset gate determines how to combine the new input with previous memory.
- The update gate defines how much of the previous memory to use in the present.

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

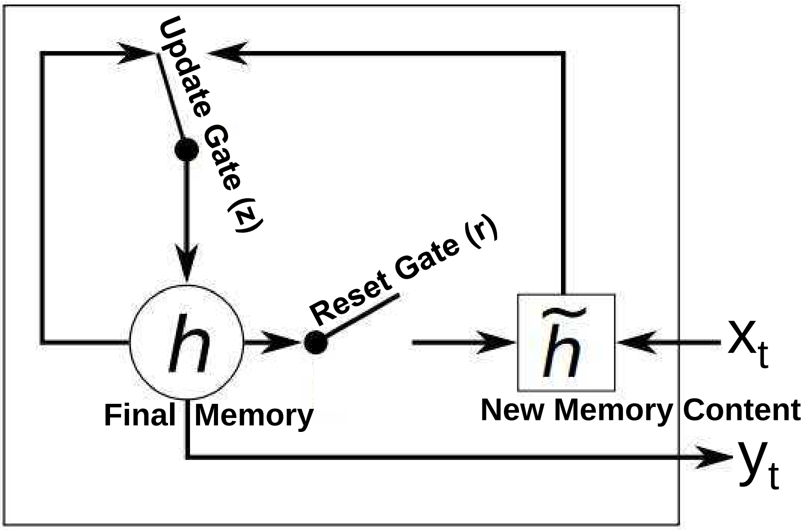


Figure 5.1: Gated Recurrent Unit topology

The Reset Gate

The reset gate determines how to combine the new input x_t with the previous hidden state h_{t-1}. It gives the model the ability to block or pass information from the previous hidden state. This allows a GRU to "reset" itself whenever a previous hidden state is no longer relevant. The reset gate is applied directly to the previous hidden state.

NOTE... ✍

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

The Update Gate

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

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

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

The GRU is designed to adaptively reset or update its memory content. The hidden activation (final memory) is simply a linear interpolation of the previous hidden activation (new memory content), with weights determined by the update gate.

Practical Application of Gated Recurrent Unit Neural Networks

The GRU, although a relatively recent addition to the family of neural networks, is increasingly being deployed for real-world time series forecasting. We discuss two successful applications, the first relates to the accurate identification of a patient's state of health. The second, is in the field of anomaly detection.

Patient State of Health

If you have spent any time in an intensive care unit, you will have no doubt noticed the array of monitoring devices hooked up to the patient. These are designed to monitor patient state of health. Any sudden change precipitates an alarm to alert the medical team that urgent action is required.

Adam McCarthy and Christopher Williams, build a GRU neural network to capture the dynamics of four patient state of health time series - electrocardiogram, systolic and diastolic arterial blood pressure, and systolic intracranial pressure.

The data was extracted from measurements on 27 patients admitted to an intensive care unit in Glasgow, Scotland. And the target variables capturing patient state of health were modeled by:

1. blood samples (BS);
2. endotracheal suctioning (SC);
3. damped traces (DT);

4. and other annotations, referred to by the authors as the X factor (X).

A number of GRU models were developed for each target variable. The number of nodes ranged from 8 for the SC model, to 64 for the BS and X models. It is interesting to notice that for all the models considered, the learning rate was set to less than 0.004.

The researchers noted wide variation in the performance of the models. For example, the BS GRU achieved an area under the curve score of 0.97; while the X GRU achieved a score of only 0.58.

These results indicate that the GRU is certainly not a "sliver bullet". However, just like any other machine leaning technique, given the appropriate sample, it can deliver outstanding performance. The only known way to determine how the GRU or any other technique will perform, is to try it out on sample data.

Anomaly Detection in Aircraft Time Series

Researchers at the Center for Air Transportation Systems Research at George Mason University, developed a GRU neural network to detect anomalies in multivariate time-series data, collected from the Flight Data Recorder of an aircraft.

The dataset included a total of 500 simulated flights. A total of 15 of these flights were anomalous. For each flight, twenty-one continuous and discrete variables were recorded for the approach phase of flight. These included factors such as the aircraft state, and automation state parameters.

A total of ten GRU neural networks were tested on the data. Four of the models had two hidden layers, with the remainder having a single hidden layer. Networks had either 30 or 60 nodes in each of their hidden layers. The models were optimized over 40 to 120 epochs. The researchers report that every single model was able to correctly detect at least 8 anomalous cases

Example - Predicting Eye Movements

The LSTM and GRU models often deliver similar performance. Let's investigate this by building a GRU model using the eye movement data we discussed on page 103.

Step 1 – Collecting Data

We mirror the step by step process outlined on page 103. First, load the required sample data:

```
data("crqa",package ="crqa")
```

Step 2 – Exploring and Preparing the Data

The data is prepared and transformed along the exact same lines as discussed in on page 103:

```
seq_1<-RDts1[1:100 ,]
seq_2<-RDts1[101:200 ,]
seq_3<-RDts1[201:300 ,]
data<-cbind(seq_1 ,
seq_2 ,
seq_3)

lab_1<-RDts1[2:101 ,]
lab_2<-RDts1[102:201 ,]
lab_3<-RDts1[302:301 ,]
lab<-cbind(lab_1 ,
lab_2 ,
lab_3)
```

```
data_set = aperm((array(c(data),
dim = c(100,3))))

labels_set= aperm(array(c(lab),
dim = c(100,3)))

trainDat <- list(data=data_set,
label=labels_set)
summary(trainDat)
```

All the above should be familiar to you. For a refresh see page 103.

Step 3 – Training a Model on the Data

We build a GRU with 1 hidden layer containing 10 nodes. The model is run over 50 iterations. Here are the details of the parameters used to specify the model:

```
batch_size = 1
seq.len = 3
num.hidden = 10
num.embed = 2
num_layer = 1
num_round = 50
update.period = 1
learning.rate = 0.8
num_labels = 7
input_size = 7
moment = 0
```

The model can be fitted to the data using the `mx.gru` function:

```
require(mxnet)
mx.set.seed(2018)
model <- mx.gru(trainDat,eval.data = NULL,
ctx = mx.cpu(),
num.round = num_round,
update.period = update.period,
num.gru.layer=num_layer,
seq.len=seq.len,
num.hidden=num.hidden,
num.embed=num.embed,
num.label=num_labels,
batch.size=batch_size,
input.size=input_size,
initializer=mx.init.uniform(0.1),
learning.rate=learning.rate,
momentum=moment)
```

Step 4 – Evaluating Model Performance

Next, we create an inference model. This is achieved by passing the parameters of `model`, and several of our previously specified parameters to the `mx.gru.inference` function:

```
mx.set.seed(2018)
pred<-mx.gru.inference(num.gru.layer =
    num_layer,
input.size = input_size,
num.hidden = num.hidden,
num.embed = num.embed,
num.label = num_labels,
ctx = mx.cpu(),
arg.params = model$arg.params)
```

Notice:

- We reference the parameters of model via `$arg.params`.

- The object `pred` contains the inference model.

Using the `mx.gru.forward` function

The `mx.gru.forward` function can be used to predict via the gru inference model `pred`. For example, to obtain the prediction for observation 301 you would use:

```
model_probs <- mx.gru.forward(pred,
RDts1[301,1])
```

The object `model_probs` is a list that contains the probability for each of the seven potential eye positions. To access these probabilities append the `$prob` argument:

```
model_probs$prob
                  [,1]
[1,]  1.930798e-02
[2,]  3.812707e-07
[3,]  3.625009e-05
[4,]  9.516750e-01
[5,]  1.144239e-02
[6,]  8.313497e-07
[7,]  1.753725e-02
```

The probability for each of the seven eye positions is reported. Your numbers may differ from those shown above. You can use the `which.max` function to automatically return the class with the largest probability:

```
which.max(as.array(model_probs$prob))
[1]  4
```

The largest probability occurs for class 4.

Step 5 – Improving Model Performance

Now, let's build a loop to look at the probabilities collectively rather than one by one. For this illustration, we use the GRU model to predict a sequence of 100 eye positions.

Key variables

Here are some key variables:

```
probs <-1:700
eye_class <-1:100
dim(probs)<-c(num_labels,100)
```

The object `probs` will contain the probabilities. There are seven eye positions, and 100 observations to be forecast. We use a two dimensional array with 7 rows and 100 columns to store the probabilities. The object `eye_class`, will contain the forecasts of eye fixation position.

The for loop

Here is the for loop:

```
for (i in (1:100)) {
temp=as.numeric(RDts1[i+302,1])
mx.set.seed(2018)
model_prob <- mx.gru.forward(pred,
temp , FALSE)
prob=as.array(model_prob$prob)
eye_class[i] <- which.max(prob)
}
```

Let's take a moment to walk through this code.

1. The current eye position is stored in the object `temp`.

2. It is passed with the inference model (`pred`) to the `mx.gru.forward` function.

3. The actual estimated eye probabilities are stored in `prob`, and the `which.max` function is used to retrieve the eye position with the largest probability. This value is stored in the object `eye_class`.

Viewing the results

Our final step, is to collect together the predictions and actual observations and display them as a plot. First, convert the `eye_class` values into a suitable format:

```
eye_class<-as.data.frame(eye_class)
eye_class[,1][eye_class[,1] == 7] <- 10
final_pred<-as.ts(eye_class[,1])
final_pred<-as.numeric(unlist(final_pred))
final_pred<-ts(matrix(final_pred),
start=c(1),
end=c(100),
frequency=1)
```

Here is a short overview of the above:

- The second line ensures the 7th class has the label “10” rather than “7”. This makes the output comparable with the actual observations.

- The object `final_pred` contains the predictions in the appropriate format we wish to use.

For illustrative purposes, we combine the predictions and the actual observed eye positions into a single dataframe called `result` which we then plot:

```
result<-cbind(RDts1[303:402,1],final_pred)
plot(result)
```

The resultant plot may look similar to Figure 5.2. Overall, the GRU delivers similar performance to the LSTM on this example.

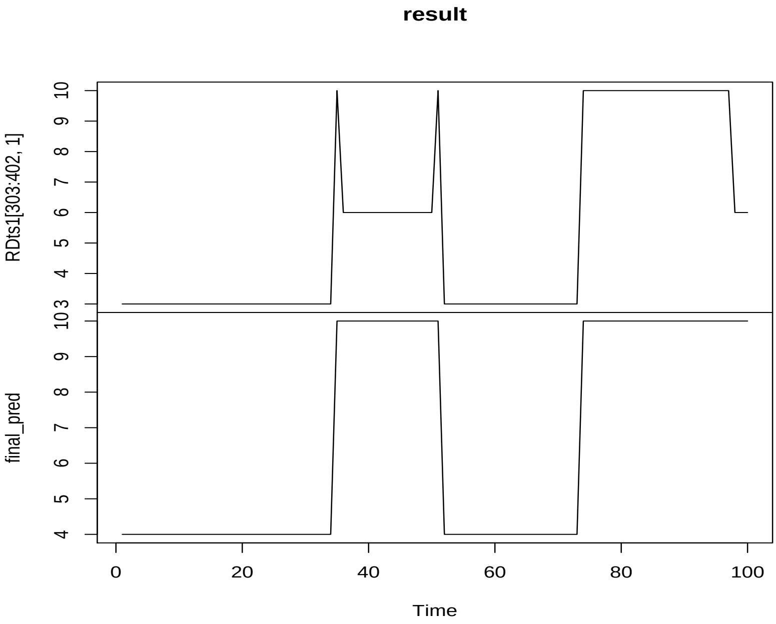


Figure 5.2: Actual (top) and predicted values (bottom) for the eye movement GRU model

NOTE... ✍

How similar are your results to those shown in the text? Take another look at Figure 4.7. and compare it to Figure 5.2.

Summary

This chapter introduced the GRU for applied time series forecasting. It is designed to adaptively reset or update its memory content. Often, it delivers similar performance to the LSTM model. However, having fewer parameters it is more parsimonious.

This chapter scratched the surface of how the GRU can be used. Be sure to take a close look at both articles mentioned in the suggested reading section.

Suggested Reading

To find out more about the practical applications discussed in this chapter see:

- **Patient State of Health:** McCarthy, Adam, and Christopher Williams. "Predicting Patient State-of-Health using Sliding Window and Recurrent Classifiers." arXiv preprint arXiv:1612.00662 (2016).

- **Anomaly Detection in Aircraft Time Series:** Nanduri, Anvardh, and Lance Sherry. "Anomaly detection in aircraft data using Recurrent Neural Networks (RNN)." Integrated Communications Navigation and Surveillance (ICNS), 2016. IEEE, 2016.

Chapter 6

Elman Neural Networks

THIS chapter extends our neural network tool kit by introducing the Elman network. It is a popular simple recurrent neural network that has delivered outstanding performance in a wide range of applications.

You will learn:

- About the structure of an Elman neural network.
- How their memory operates, and differs from the LSTM and GRU.
- Some areas where they have been successfully applied.
- How to build them quickly and efficiently using R.

Although Elman neural networks have been around for a relatively long period of time, they are an essential neural network technique to add to your time series tool kit.

Understanding Elman Neural Networks

Elman neural networks were initially designed to learn sequential or time-varying patterns. They are composed of an input layer, a context layer (also called a recurrent or delay layer see Figure 6.1), a hidden layer, and an output layer. Each layer contains one or more neurons which propagate information from one layer to another by computing a nonlinear function of their weighted sum of inputs.

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

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

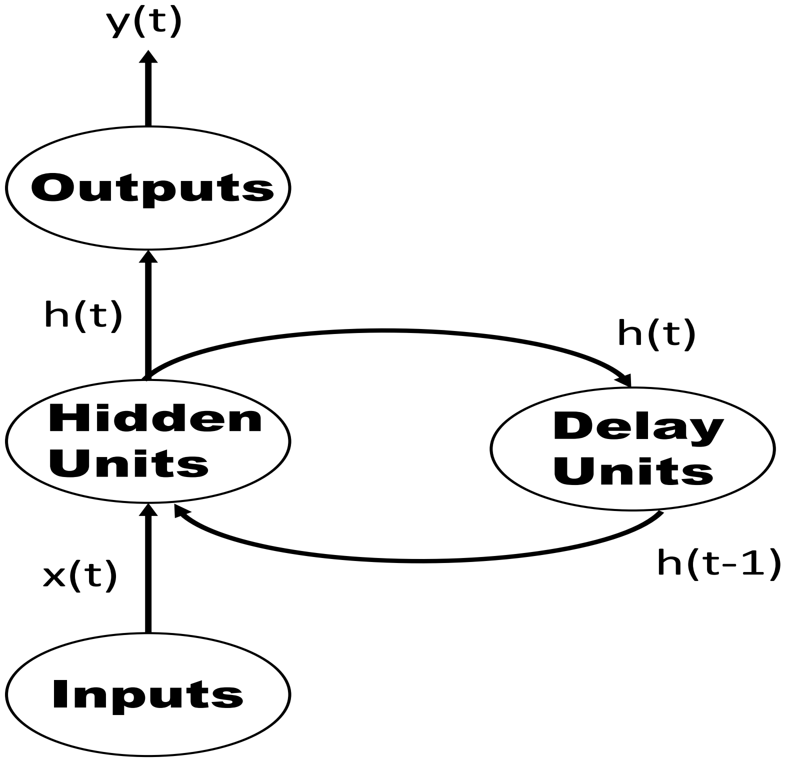


Figure 6.1: Elman Neural Network

Role of Memory

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

Practical Application of Elman Neural Networks

In this section, we take a look at several interesting applications of Elman neural networks. The first uses them to forecast highly volatile electricity prices. The second, illustrates how they can be used to predict crude oil futures prices; the third, explores their role as a weather forecasting tool. And the forth application, discuss how Elman neural networks can be used for anomaly detection. Their role in assisting in the drive to maintain water quality, and how they can be used to predict the stock market are also discussed.

Electricity Market Price Forecasting

In a deregulated electricity market, price is determined by the balance between available supply and demand. Producers of electricity would like to know the day-ahead price in order to make adjustments to available supply. Consumers also would like to know the day ahead price in order to minimize their cost of consumption.

Figure 6.2 shows the daily average wholesale electricity price from 2006 to 2013, for the United States Midwest Indiana Hub. The price series is characterized by a varying trend, changes in the underlying variance, and spikes. These characteristics make forecasting electricity prices challenging.

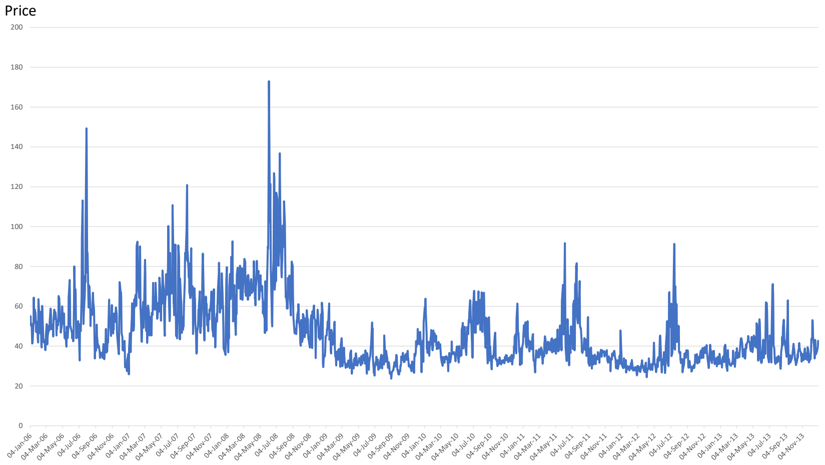


Figure 6.2: Indiana Hub wholesale electricity (daily average) spot price

Researchers S. Anbazhagan and N. Kumarappan, develop a Elman neural network model for day ahead electricity market price forecasting. The researchers use four weeks of Spanish electricity prices for the year 2002, and four weeks of New York prices for 2010.

The proposed Elman network had 10 nodes in the hidden layer, with 16 lagged prices as input features. The Spanish and New York model returned an average weekly mean absolute percentage error of around 6.6% and 3.82% respectively.

Crude Oil Futures

The price of crude oil has a significant impact on the global economy. Economists, traders, risk managers, corporations and consumers, are all invested in the price of this global commodity. Accurate prediction of future prices is of considerable value to producers, consumers and investors.

Figure 6.3 plots the daily crude oil (Light-Sweet, Cushing, Oklahoma) price from November 1999 to March 2017. It is characterized by long run and short term trends, alongside sudden price reversals.

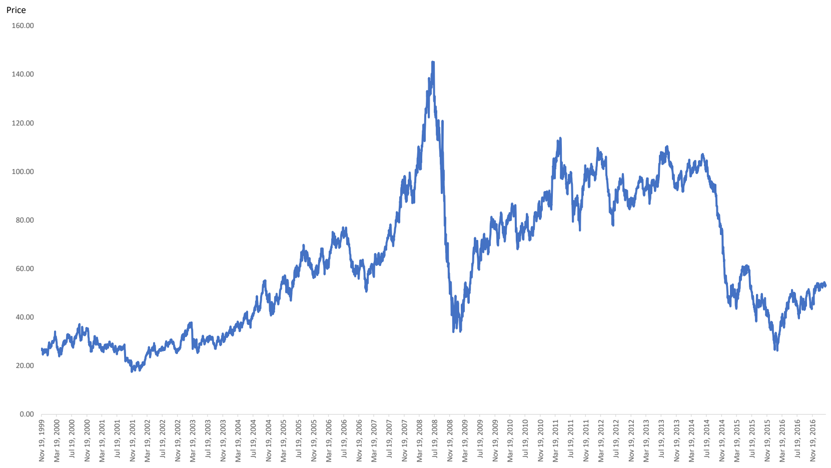


Figure 6.3: Crude oil futures price (Light-Sweet, Cushing, Oklahoma)

A team of finance experts trained Elman neural networks to predict the near-month daily futures price for a range of crude oil prices (Brent, WTI, DUBAI, and IPE). Models were trained over a minimum of ten years worth of daily data run over 1000 iterations. The researchers discovered, that for crude oil prices, the larger the training set, the better the general performance of the Elman neural network.

Weather Forecasting

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

Researchers Maqsood, Khan and Abraham 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. The test sample consisted 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.

Researcher Yao et al. 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 three neurons in the output layer. The eight inputs corresponded to various ranges on the frequency spectrum of fault signals received from the inverter over time. 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 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. The dataset 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.

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 -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.

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 was 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 an 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 accurately 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. 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 observations.

Example - Forecasting Historical Wheat Prices

Wheat is the second most cultivated crop after maize. As the global population has continued to expand, the production of wheat has grown. World production has grown by more than three times since 1960. Crop production continues to grow today. In this section, we build an Elman neural network to predict historical wheat prices.

Step 1 – Collecting Data

The r object `bev` in the `tseries` package contains the annual wheat price from 1500 to 1869, averaged over many locations in western and central Europe. Load and plot the data:

```
data("bev",package ="tseries")
plot(bev)
```

Figure 6.4 shows the resultant plot.

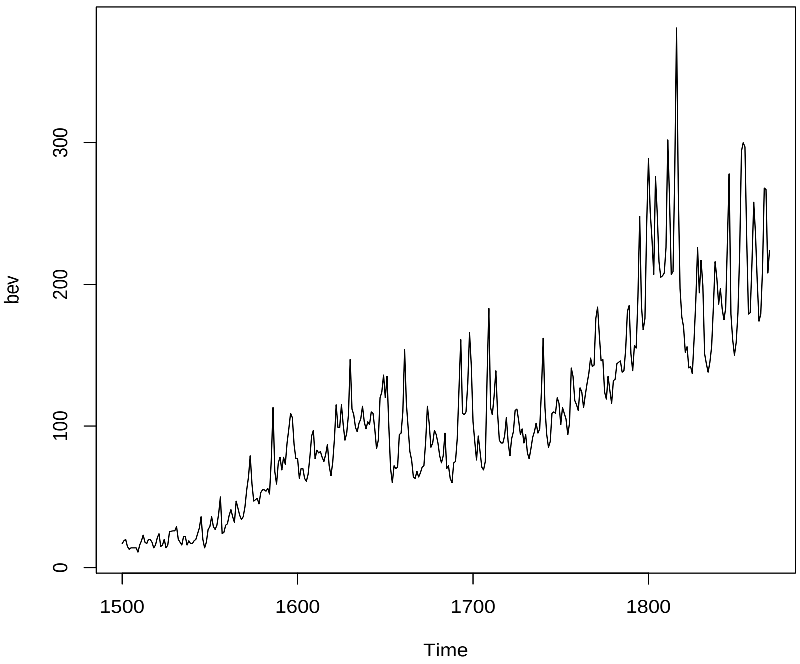


Figure 6.4: European historical wheat prices

The time series exhibits a positive long term trend, around which there is considerable volatility. The volatility increases with the general price level. There also appears to be strong seasonality in the data. This is probably responsible for driving much of the undulating nature of the series.

Step 2 – Exploring and Preparing the Data

From Figure 6.4, it appears seasonality tends to increase with the general trend suggesting a multiplicative model:

$$price_t = trend_t \times seasonal_t \times random_t$$

where:

- $price_t$ is the price of wheat at time t,
- $trend_t$ is the trend component,
- $seasonal_t$ the seasonal component,
- and $random_t$ is the random error component.

Using the `decompose` function

We can use the `decompose` function to estimate each of these components. Rather than use the price levels, we will work with the natural logarithm of prices:

```
y <- ts(log(bev), frequency = 12)
decomp<-decompose(y,
type = "multiplicative")
 plot(decomp)
```

The R object `y` contains the log prices, `decomp` the decomposed series, and Figure 6.5 shows the estimated components.

NOTE... ✍

The additive model is defined as:
$price_t = trend_t + seanaonal_t + random_t$
It can be called via the `decompose` function by setting the argument `type = "additive"`.

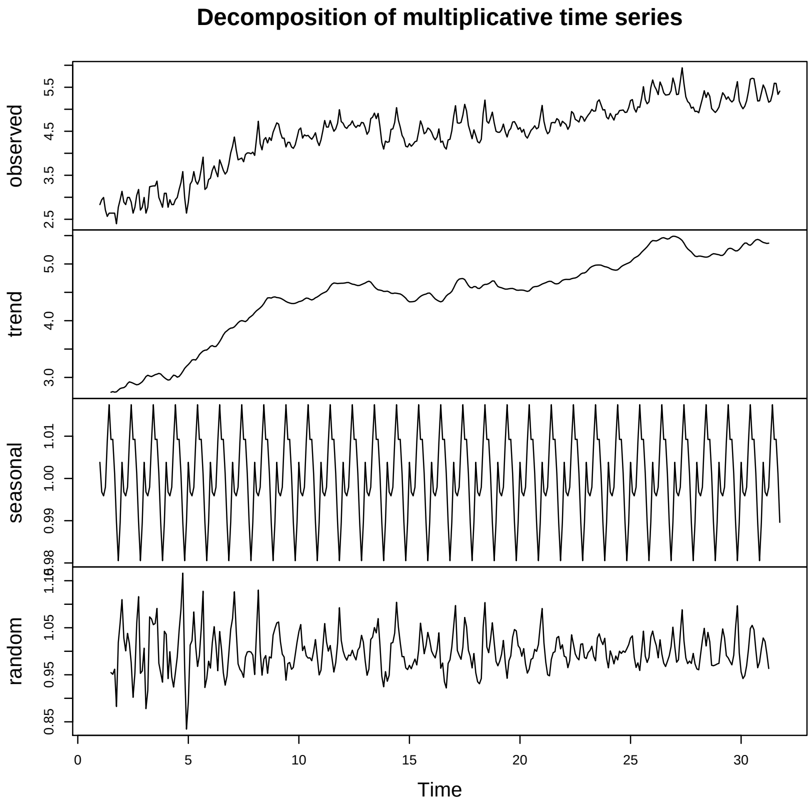


Figure 6.5: Decomposition of wheat prices

Trend component

Let's take a closer look at the trend component:

```
plot(decomp$trend,ylab="Value")
```

Figure 6.6 indicates the trend is highly nonlinear. It contains periods of rapid increase followed by relative plateaus.

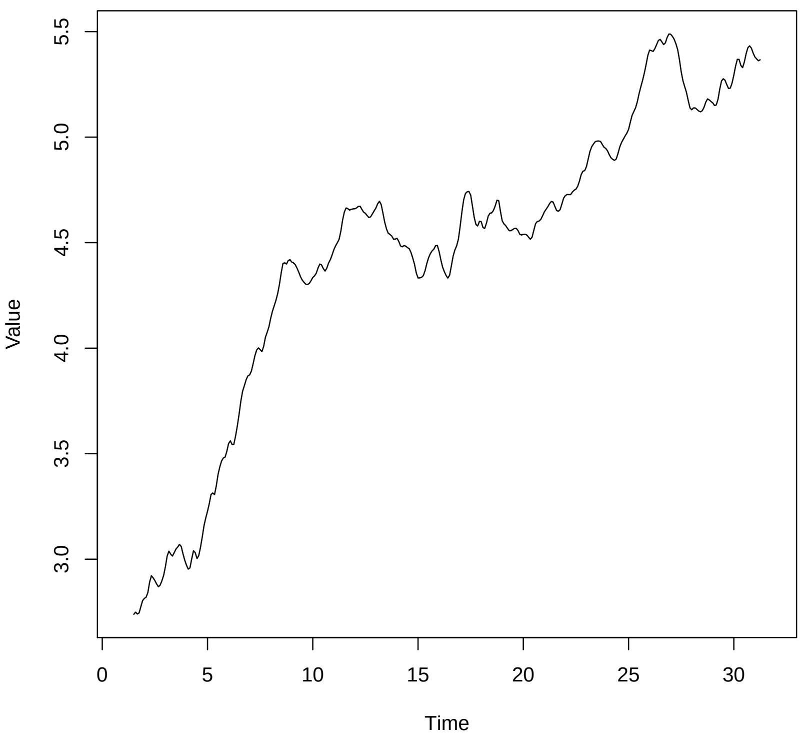


Figure 6.6: Trend component of wheat prices

Error component

It can often be useful to inspect the distribution of the error component. We use the `density` function to give us a quick visual summary:

```
random<-newdata <- na.omit(decomp$random)
plot(density(random))
```

Figure 6.7 indicates the error component follows approximately the classical bell shaped distribution of a normally distributed random variable.

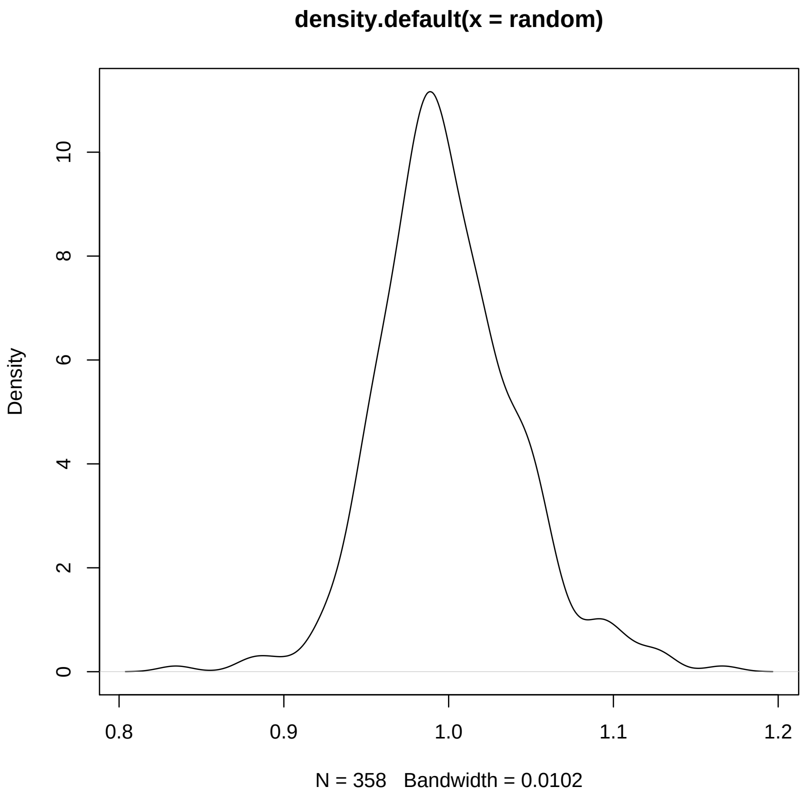


Figure 6.7: Density plot of the random component of wheat prices

We can probe a little deeper using a Normal Q-Q plot via the function `qqnorm`:

```
qqnorm(random)
```

If the `random` component values were generated by a normal distribution, the points in the Q-Q plot would lie exactly along the 45 degree line. Figure 6.8 confirms `random` is approximately normally distributed.

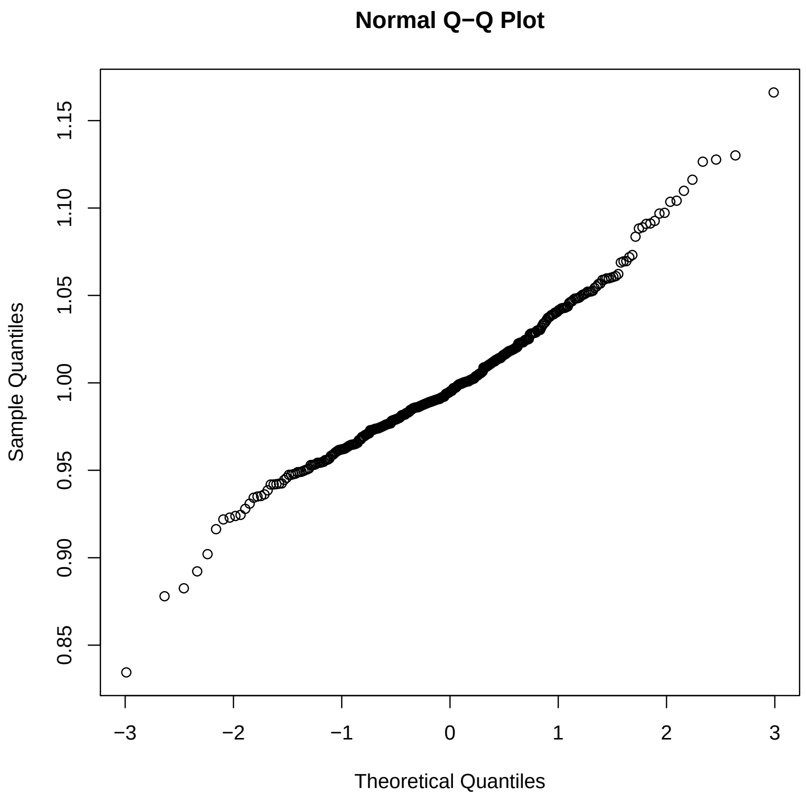


Figure 6.8: Q-Q plot of the random component of wheat prices

Inspecting Autocorrelation

Next, take a look at the autocorrelations and partial autocorrelations:

```
acf(y)
pacf(y)
```

From Figure 6.9, we see that the correlation between prices tends to persist over time. Figure 6.10 shows the partial autocorrelations. It indicates that lagged prices have some association with the current price.

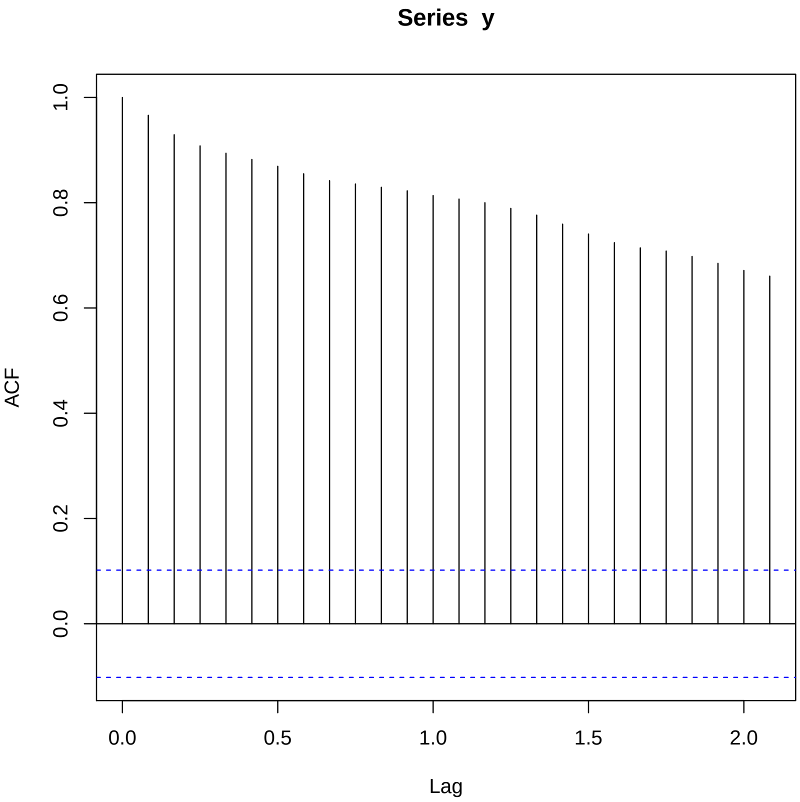


Figure 6.9: Auto correlations of wheat prices

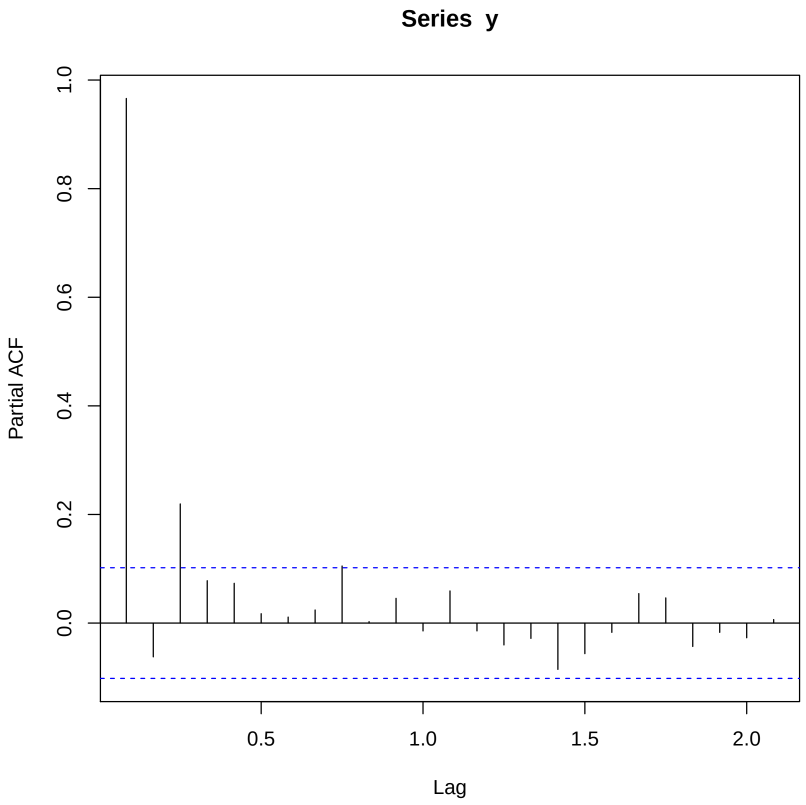


Figure 6.10: Partial autocorrelations of wheat prices

Choosing the number of time lagged variables

Given the monthly frequency of the data, we will create 12 time lagged variables as input features. First, scale the data using the custom `range_data` function:

```
range_data<-function(x)  {(x-min(x))/(max(x)-
    min(x))}

y <-range_data(bev)
min_data<-min(bev)
max_data<-max(bev)
```

Notice that we are now using the actual price levels rather than the log values.

Next, use the `quantmod` package to create the lagged features:

```
require(quantmod)
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)

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

The object `x` contains the lagged data. The last line in the above code removes the missing values that result from the use of the `Lag` function.

Number of observations

We should have sufficient observations left to do a meaningful analysis, let's check:

```
n=nrow(x)
n
```

```
[1] 358
```

Yep, a total of 358 observations. This is sufficient for our analysis.

Step 3 – Training a Model on the Data

We use 300 observations selected at random for the training set. The remainder are held back for the test set:

```
set.seed(2018)
n_train <- 300
train <- sample(1:n,n_train , FALSE)
inputs <- x[,2:13]
outputs <- x[,1]
```

The `sample` function selects the training sample at random, without replacement. The target variable is stored in the object `ouputs`, and the attributes are stored in the R object `inputs`.

Using the package `RSNNS`

The package `RSNNS` contains the function `elman` which estimates an Elman neural network. We build a model with two hidden layers each containing 2 nodes. The model is run for 1000 iterations:

```
require(RSNNS)
fit <- elman(inputs[train],
outputs[train],
size=c(2,2),
maxit=1000)
```

The argument `size` specifies the number of nodes in each of the hidden layers; and the argument `maxit` controls the number of iterations. The R object `fit` contains the fitted model.

NOTE... ✍

RSNNS version 0.4.9 is used to carry out the analysis in this section.

Step 4 – Evaluating Model Performance

The `plotIterativeError` function plots the iterative error over the sample. A rapidly declining error indicates the model is learning from the available data:

```
plotIterativeError(fit)
```

Figure 6.11 indicates the error drops rapidly for the first 150 iterations. It then declines at a more modest rate until the 1000 iteration.

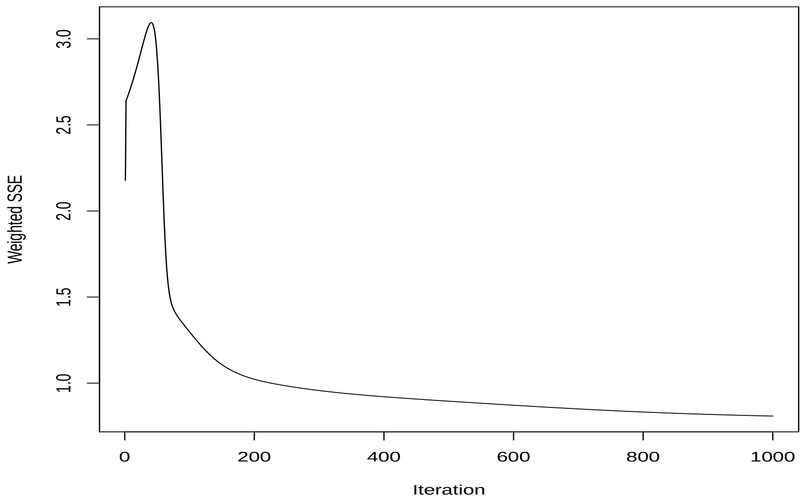


Figure 6.11: Error by iteration for Elman neural network

NOTE... ✍

Try running the model over a larger number of iterations. Investigate by how much performance changes.

The `plotRegressionError` function

The `plotRegressionError` is used to visualize the relationship between the actual target values and the predicted values:

```
plotRegressionError(outputs[train],
fit$fitted.values)
```

Figure 6.12 shows the resultant plot.

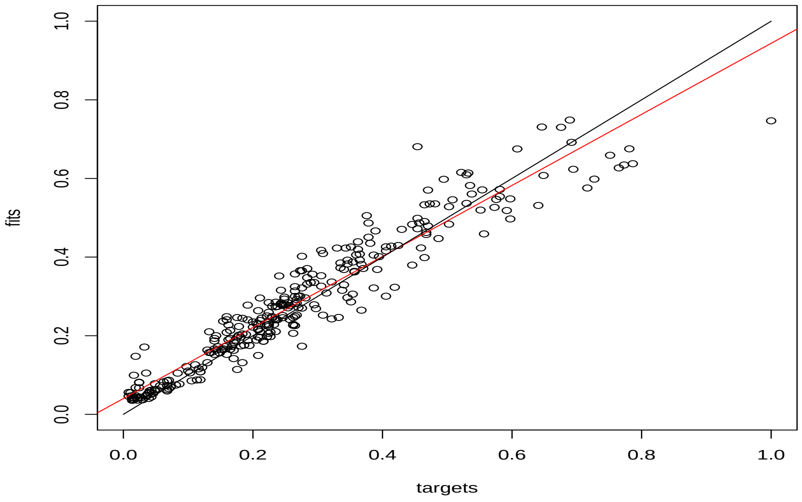


Figure 6.12: Fitted and actual values using the `plotRegressionError` function

In Figure 6.12, the target values (actual observations) lie on the x-axis; and fitted values for the training set lie on the y-axis. A perfect fit would result in a line through zero with gradient one. This is shown as the solid black line on the figure. A linear fit to the actual data is shown in red (lighter) line.

Overall, the fit to the training data looks pretty good. Let's get a quantitative assessment by calculating the squared correlation coefficient:

```
round(cor(outputs[train],
fit$fitted.values)^2,4)
          [,1]
[1,]  0.9168
```

Not too bad for an initial model!

Step 5 – Assessing Test Set Performance

The `predict` function is used to make predictions. It takes the fitted model (in this case `fit`), and the sample observations. Here, we want to assess how well the model performs on the test set, so we pass these values alongside `fit` to `predict`:

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

round(cor(outputs[-train],
pred)^2,4)
        [,1]
[1,]  0.8746
```

The test set correlation is quite respectable at around 0.87.

Visual inspection

We want to unscale the data and take a look at it visually. Our custom `unscale_data` function does the job. This is followed by the `plot` function:

```
unscale_data<-function(x,max_x,min_x)
```

```
{x*(max_x–min_x)+min_x}

output_actual<–unscale_data(outputs[−train],
max_data,min_data)

output_actual<–as.matrix(output_actual)

rownames(output_actual)<– 1:length(
    output_actual)

output_pred<–unscale_data(pred,
max_data,
min_data)

result<–cbind(as.ts(output_actual),
as.ts(output_pred))
plot(result)
```

Figure 6.13 shows the actual and predicted values.

Overall, the Elman neural network is able to capture the trend, and much of the underlying dynamics of the wheat price time series. These results were achieved without any significant feature engineering or statistical manipulation of the dataset as if often required with traditional statistical models.

NOTE... ✍

Try tweaking the model parameters. See if you can obtain an even better fit.

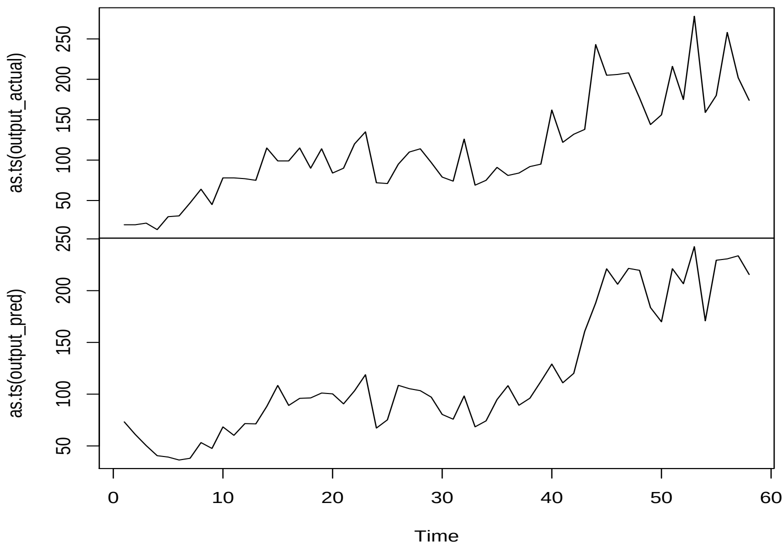


Figure 6.13: Test set actual(top) and predicted (bottom) values for the wheat prediction Elman model

Summary

In this chapter, we studied the Elman simple recurrent neural network. We used it to predict the historical price of wheat. We found that, using a randomly selected sample (rather than a subset of sequential observations as we have done in previous chapters), the Elman neural network delivered strong performance.

In the next chapter, we discuss another popular simple recurrent neural network - the Jordan neural network.

Suggested Reading

To find out more about the practical applications discussed in this chapter see:

- **Electricity Market Price Forecasting:** Anbazhagan, S., and N. Kumarappan. "Day-ahead deregulated electricity market price forecasting using neural network input featured by DCT." Energy conversion and management 78 (2014): 711-719.
- **Crude Oil Futures:** Hu, John Wei-Shan, Yi-Chung Hu, and Ricky Ray-Wen Lin. "Applying neural networks to prices prediction of crude oil futures." Mathematical Problems in Engineering 2012 (2012).
- **Weather Forecasting:** Maqsood, Imran, Muhammad Riaz Khan, and Ajith Abraham. "Canadian weather analysis using connectionist learning paradigms." Advances in Soft Computing. Springer London, 2003. 21-32.
- **Equipment Fault finding:** 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.
- **Water Quality:** Wang, Heyi, and YiGao. "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.
- **Stock market:** Wang, Jie, et al. "Financial Time Series Prediction Using Elman Recurrent Random Neural Networks." Computational Intelligence and Neuroscience 501 (2015): 613073.

Chapter 7

Jordan Neural Networks

TIME series analysis with neural networks opens up possibilities to model complex relationships that might be difficult to capture using traditional linear statistical models. This chapter covers the Jordan neural network. Like the Elman network, it is often known as a simple recurrent neural network.

By the end of this chapter you will:

- Learn the key difference between the Jordan and Elman neural networks.
- Gain an intuitive understanding of the structure of a Jordan neural network.
- Review several practical uses of the Jordan neural network.
- Apply the Jordan neural network to predict the average air temperature at the famous Nottingham Castle, England.

Jordan neural networks have proven to be a popular tool for applied time series modeling. They are often trained alongside Elman neural networks during the model selection stage of data analysis. In fact, as we will see, a Jordan neural network is very similar to the Elman neural network.

Understanding Jordan Neural Networks

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

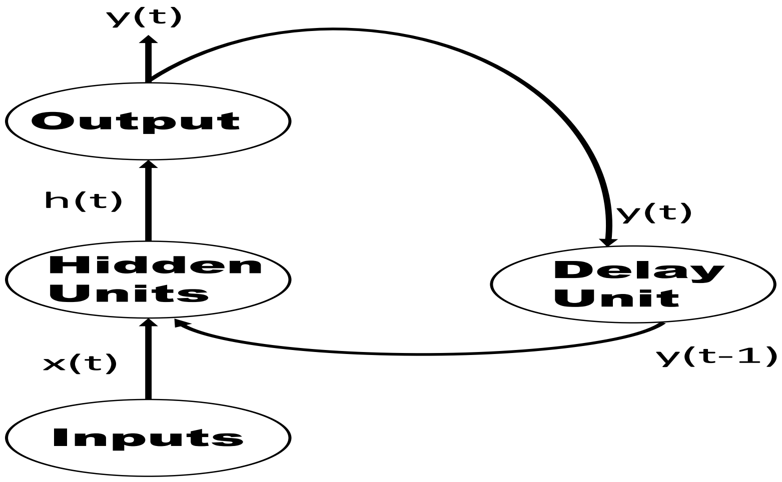


Figure 7.1: Structure of the Jordan Neural Network

Compare Figure 7.1 (Jordan) with Figure 6.1 (Elman). They are very similar. However, in a Jordan network the context layer (delay unit) is directly connected to the input of the hidden layer. Like the Elman neural network, the Jordan neural network is useful for predicting time series observations which have a short memory.

Practical Application of Jordan Neural Networks

Before we begin to build our own Jordan networks in R, it is worthwhile to look at some applications. In this section, we discuss uses that include wind speed forecasting, traffic flow prediction, and monitoring of stock market volatility.

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.

A small team of Civil Engineers (Anurag and Deo) built Jordan neural networks to forecast daily, weekly, and 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 located on 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 compared the network predictions to the popular auto regressive integrated moving average (ARIMA) time-series models; They observe:

> "*The neural network forecasting is also found to be more accurate than traditional statistical timeseries analysis.*"

Traffic Flow Forecasting

Have you ever found yourself waiting patiently at a stop light even though the intersection traffic (for which you have been stopped) is nonexistent? Well, you are not alone. It turns out that signalization of traffic lights at intersections is managed by traffic light controllers that have predefined fixed-time signalization sequences. Heavy traffic, light traffic or no traffic the signalization sequences are indifferent.

"*Hello!*", you say to yourself as you sit halted at the empty crossing, "*Has anyone thought of developing signalization sequences which respond to the traffic flow?*"

Your mind has just delivered to you a great idea, a wonderfully practical idea, and one perfectly suited to neural networks. In fact, your thought is so stunningly beneficial to other motorists that you imagine a group of German drivers, who are themselves neural network experts, creating a system suitable for the task. Ah, but that is just a fantasy, imagination, a thought and nothing more!

The Organic Traffic Control System, developed by German drivers who are neural network experts, uses a Jordan neural network for predicting upcoming traffic flows! At the core of the system lies a Jordan neural network with a sigmoid activation function, and ten units in the hidden layer.

The input attributes are the previous five time instants of traffic flow measured in vehicles per hour. These attributes are preprocessed to correct for missing values, and then normalized to a range of 0 to 1. The output layer consists of the main task and two auxiliary tasks.

The system was tested on data from a very busy intersection in the beautiful city of Hamburg, which sits on the Elbe River, a major trade route via the North Sea to the United Kingdom and beyond. The maximum flow at the intersection was reported as 1050 cars per hour.

The model predicted traffic flow five minutes into the future. The network was initially trained with 300 epochs, and training terminated when the training error fell below 0.002. Excellent predictive performance was observed on all days of the week; and the overall prediction error was estimated to be less than 5%. The researchers conclude in an understated academic voice:

> "*Based on real data for an intersection in Hamburg, Germany, the experiments for the neural network showed promising results.*"

Monitoring Stock Market Volatility

A group of Croatian Economists built a Jordan neural network to forecast stock market volatility. They collect 918 daily returns of the CROBEX index from the Zagreb Stock Exchange over the period January 2011 to September 2014. The train set consisted of 666 observations, and the test set the remaining 252 observations.

The researchers constructed a target variable to capture the volatility of the daily index. They then used a one day lag of this variable as the single input feature. The Jordan model had 1 hidden layer with 1 node, a single output layer, a learning rate set to 0.2, and the context unit weight set to 0.9. The researchers used the `RSNNS` R package to estimate the model.

The Jordan model was compared to a traditional econometric volatility forecasting model (GARCH(1,1)). The researchers report that the Jordan model showed "superior performance" on every measured metric. They conclude:

> "*The benefit of using recurrent networks in time series forecasting is reflected in the results, because JNN(1,1,1) [Jordan Neural Network] presents a better performance than the GARCH(1,1) model in forecasting volatility.*"

Example - Modeling Air Temperature

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.

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.

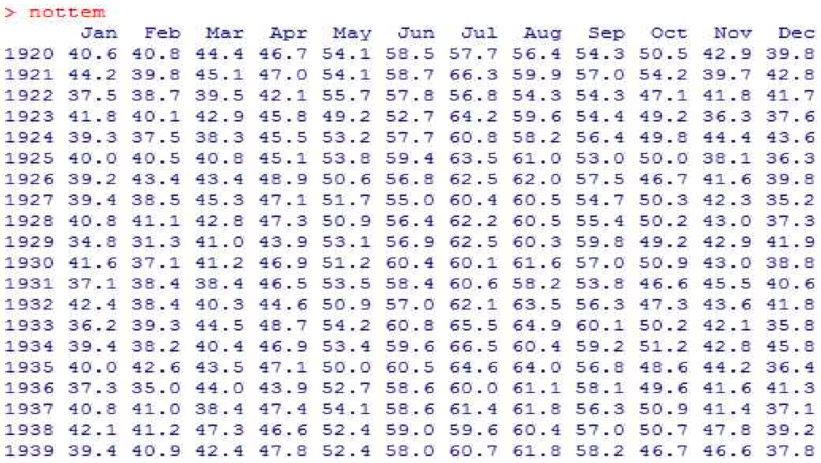
Step 1 – Collecting Data

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, a location the mythical outlaw Robin Hood would have known well:

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

Step 2 – Exploring and Preparing the Data

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.

Checking object class

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.

Viewing the data

Figure 7.2 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)
```

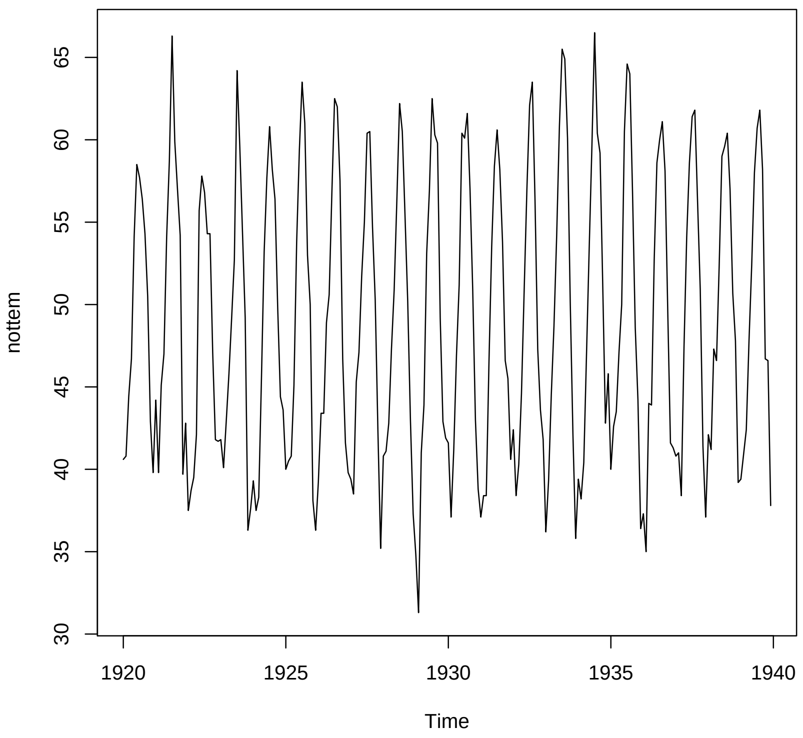


Figure 7.2: Average Monthly Temperatures at Nottingham 1920–1939

Scaling the data

In this case, given that we do not have any explanatory variables, we will take the log transformation and then use the `range_data` function to scale the data:

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

```
range_data<-function(x)
{(x-min(x))/(max(x)-min(x))}

y <-range_data(nottem)
min_data<-min(nottem)
max_data<-max(nottem)
```

Choosing lags and cleaning up

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)
```

We need to remove the lagged values that contain NA's. 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 the attributes and response variable. The result is shown in Figure 7.3, and was created using the `plot` method:

```
plot(temp)
```

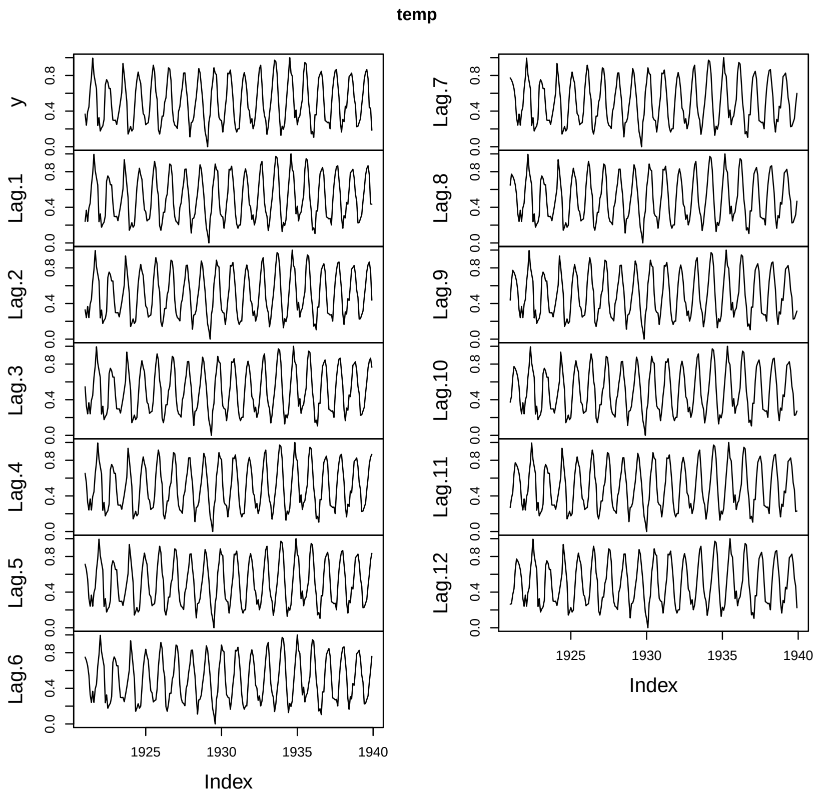


Figure 7.3: Response and attribute variables for Jordan network

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

Step 3 – Training a Model on the Data

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
```

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

```
set.seed(465)
n_train <- 190
train <- sample(1:n,n_train , FALSE)
```

The model is estimated along similar lines as the Elman neural network on page 158. 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)
```

Step 4 – Evaluating Model Performance

The plot of the training error by iteration is shown in Figure 7.4. It was created using:

```
plotIterativeError(fit)
```

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

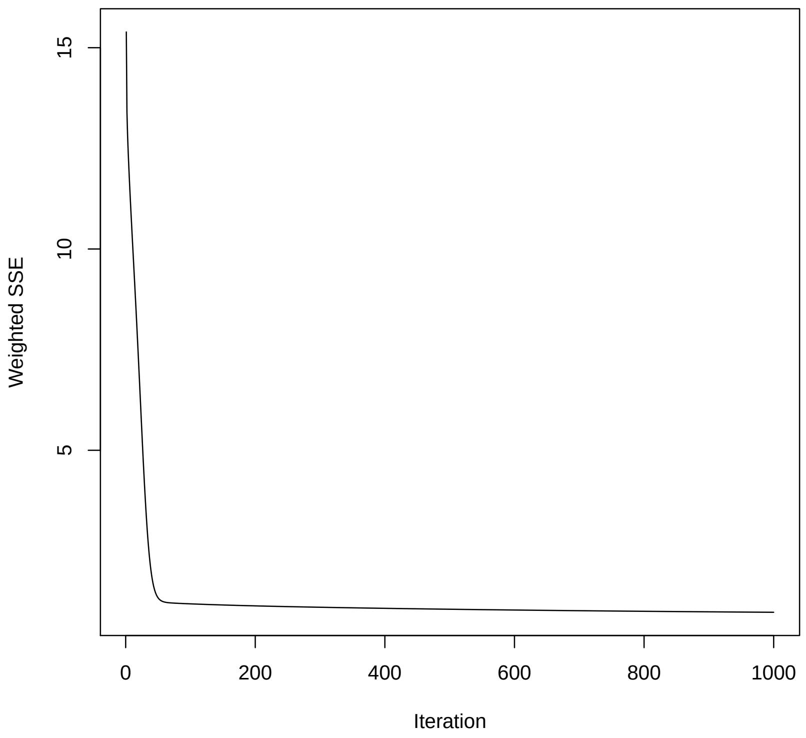


Figure 7.4: Training error by iteration

Step 5 – Assessing Test Set Performance

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])

round(cor(outputs[train],
fit$fitted.values)^2,3)
  [,1]
[1,] 0.911
```

The squared correlation coefficient is relatively high at close to 0.91.

Visualizing the data

Of course, it is also helpful to visualize the predicted and actual observations. The `unscale_data` function helps here:

```
unscale_data<-function(x,max_x,min_x)
{x*(max_x-min_x)+min_x}

output_actual<-unscale_data(outputs[-train
   ],
max_data,min_data)

output_actual<-as.matrix(output_actual)

rownames(output_actual) <- 1:length(
   output_actual)

output_pred<-unscale_data(pred,
max_data,
min_data)

result<-cbind(as.ts(output_actual),
as.ts(output_pred))
plot(result)
```

Figure 7.5 shows the actual and predicted values for our model. Notice how well it captures the underlying dynamics of the air temperature time series. Although, not an exact fit, the model provides a great benchmark from which to compare alternative model specifications. We can at least say, our initial model has good success in predicting the air temperature in Nottingham - Robin Hood would be amazed!

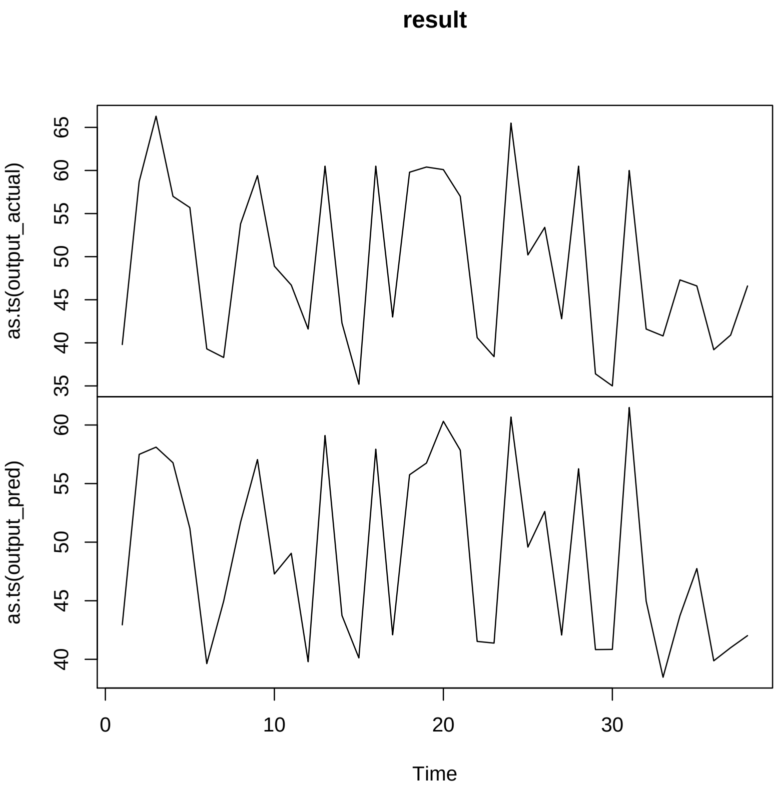


Figure 7.5: Actual (top) and predicted values using a Jordan neural network

Summary

Jordan neural networks have been applied to time-series prediction for many years; from forecasting stock market volatility and environmental factors, to traffic flow forecasting. They can be applied directly to a time-series task provided the data has been suitably pre-processed.

In practice, choosing the correct network topology and type can have huge effects on the effectiveness of time-series prediction. As we have seen in previous chapters, sometimes it is more important to have a couple of hidden layers with a few nodes; And for some problems a lot of nodes in a single hidden layer does the job.

In the next chapter, we will examine another important type of neural network. Much like the neural networks we have studied to-date, it is useful for identifying patterns in data. But unlike feed-forward neural networks, the method in the next chapter models the relationship between attributes and a target variable via a polynomial set of mathematical equations.

Suggested Reading

To find out more about the practical applications discussed in this chapter see:

- **Wind speed forecasting:** More, Anurag, and M. C. Deo. "Forecasting wind with neural networks." Marine structures 16.1 (2003): 35-49.

- **Traffic flow forecasting:** Sommer, Matthias, Sven Tomforde, and Jörg Hähner. "Using a Neural Network for Forecasting in an Organic Traffic Control Management System." Presented as part of the 2013 Workshop on Embedded Self-Organizing Systems. 2013.

- **Monitoring Stock Market Volatility:** Arnerić, Josip, Tea Poklepović, and Zdravka Aljinović. "GARCH based artificial neural networks in forecasting conditional variance of stock returns." Croatian Operational Research Review 5.2 (2014): 329-343.

Chapter 8

General Method of Data Handling Type Neural Networks

EVERY once in a while, a technique comes along that takes the data science community by storm. The General Method of Data Handling (GMDH) type neural networks are an example of such a technique. It was developed behind the "*Iron Curtain*" during the dying days of the Soviet Union. The method was not widely known or taught in the university classrooms of the major European, Asian, African or American universities.

Today, it seems strange that this powerful technique went almost unnoticed. I stumbled across the method quite by accident, as part of my search for the very best neural network techniques. It has been a permanent addition to my arsenal of learning from data algorithms ever since. In this chapter, you will:

- Gain an intuitive insight into how the technique works.
- Understand why it is considered to be so powerful.
- Study practical applications of it use.

- Discover how to develop GMDH- type neural networks using R.

As we develop our understanding of the GMDH type neural network, we touch on the important topic of deductive/ inductive inference; and a powerful learning from data principle known as Occam's Razor. But first, we turn our attention to what others have said about this truly outstanding learning algorithm.

Understanding the General Method of Data Handling Type Neural Networks

When I first came across the Ukrainian Academy of Sciences A. G. Ivakhnenko's polynomial theory of complex dynamic systems, I was immediately struck by the ingenuity of the approach, and by the audacious claim that:

> "*The polynomial theory of complex dynamic systems will bring about a complete revolution of the art of prediction, pattern recognition, identification, optimizing control with information storage, and to the other problems of engineering cybernetics.*"

The claim of A. G. Ivakhnenko was so appealing that I was literally salivating at mouth at the thought of the amazing possibilities it could bring to my modeling capabilities. Imagine it, whilst my co-workers were struggling to get their data to "*speak to them*" using linear models and classical econometrics, I would be acing it with Ivakhnenko's Self-Organizing Polynomial Neural Networks!

I simply could not wait to put Ivakhnenko's claim to the test with real data on a real project. No matter what, I would use the method in my very next suitable project.

Some Areas of Outstanding Success

In the meantime I continued to read of fantastic results by those who applied Ivakhnenko's General Method of Data Handling (GMDH) technique. Here are just a small sample of the observations made by seasoned researchers who have tried the technique, and reported their findings in, usually very subdued, scholarly journals:

- **Across all disciplines**: "*Since its introduction researchers from all over the world who used the GMDH in modeling were astonished by its prediction accuracy.*" - Abbod, Maysam, and Karishma Deshpande.

- **Poultry production**: "*In conclusion, using such powerful models can enhance our ability to predict economic traits, make precise prediction of nutrition requirements, and achieve optimal performance in poultry production.*" - Mottaghitalab, M., et al.

- **Image processing**: "*It was shown that the revised GMDH-type neural network algorithm was a useful method for the medical image recognition of the brain because the neural network architecture was automatically organized so as to minimize AIC* [Akaike's Information Criterion] *or PSS* [Prediction Sum of Squares] *values.*" - Kondo, Tadashi, and Junji Ueno.

- **Exchange rate prediction**: "*This implies that the proposed* [GMDH] *modelling approaches can be used as a feasible solution for exchange rate forecasting in financial management.*" - Taušer, Josef, and Petr Buryan.

- **Molecular sciences**: "*The resulting predictive* [GMDH] *model appears to correctly incorporate the effects of liquor impurities and is found to offer a level of performance comparable to the most sophisticated phenomenological model presented to date.*" - Bennett, Frederick R., Peter Crew, and Jennifer K. Muller.

- **Commodity prices**: "*It has been shown that GMDH type neural networks have better results when the associated system is so complex and the underline processes or relationship are not completely understandable or display chaotic properties.*" - Mehrara, Mohsen, Ali Moeini, and M. Ahrari.

Fortunately, the wait to test out Ivakhnenko's neural network model was not very long; the results were astounding! The approach worked so well it is now a permanent addition to my time series forecasting toolbox.

The General Method of Data Handling in a Nutshell

Ivakhnenko's approach, known as the General Method of Data Handling (GMDH), models the relationship between a set of p attributes and a target variable using a multilayered perceptron-type neural network structure.

In general, the connection between the input and output variables can be approximated by an infinite Volterra-Kolmogorov-Gabor polynomial of the form:

$$y = a_0 \sum_{i=1}^{p} a_i x_i + \sum_{i=1}^{p} \sum_{j=1}^{p} a_{ij} x_i x_j + \sum_{i=1}^{p} \sum_{j=1}^{p} \sum_{k=1}^{p} a_{ijk} x_i x_j x_{k+\ldots}$$

A. G. Ivakhnenko demonstrated that a second-order polynomial can reconstruct the entire Volterra-Kolmogorov-Gabor polynomial using a feedforward perceptron type procedure:

$$y = \alpha + \alpha_1 x_i + \alpha_2 x_j + \alpha_3 x_i^2 + \alpha_4 x_j^2 + \alpha_5 x_i x_j,$$

NOTE... ✍

The work of the Institute of Cybernetics, Ukrainian Academy of Sciences, Academician A. G. Ivakhnenko was revealed to the Western world when R.L Barron traveled to the Soviet Union back in 1968.

Within a matter of months, Ivakhnenko's Polynomial Theory of Complex Systems was rushed through the publication process, and appeared in the IEEE Transactions On Systems, Man, and Cybernetics.

For further details get a copy of this paper, it is a very interesting read, both in terms of the mathematical details and also the historical cold war context. See Ivakhnenko, A. G. "Polynomial theory of complex systems." Systems, Man and Cybernetics, IEEE Transactions on 4 (1971): 364-378.

Network Structure

The GMDH network is an automated heuristic driven self-organizing learning process consisting of a network of neurons (often called transfer functions).

The network architecture is fully self-determined by the GMDH algorithm; with the final network containing an input layer, hidden layers and the final output layer.

An illustrative structure with p input attributes is shown in Figure 8.1.

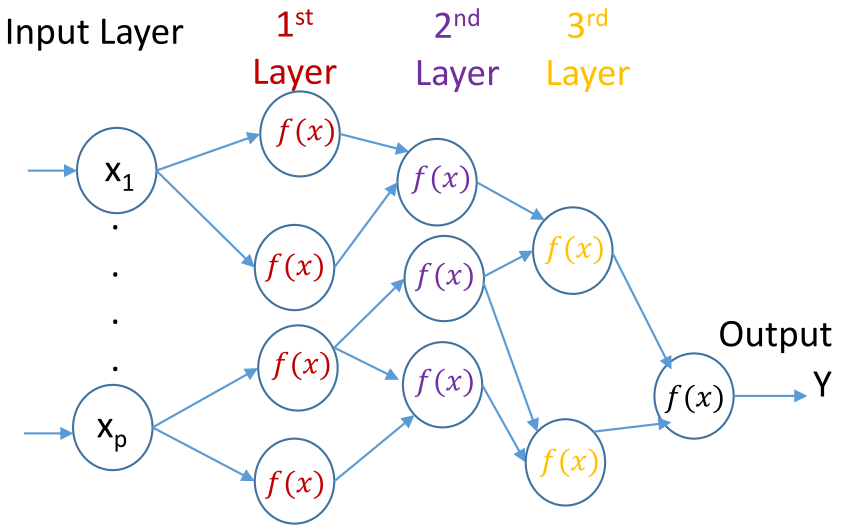


Figure 8.1: GMDH network structure.

The Main Idea

The main idea behind the algorithm is that each node in the hidden and output layer receives input from two other neurons, and outputs a quadratic function of its inputs. In other words, each neuron in the network fits its output to the target for each input vector:

$$y_{ij} = f(x) = \alpha + \alpha_1 x_i + \alpha_2 x_j + \alpha_3 x_i^2 + \alpha_4 x_j^2 + \alpha_5 x_i x_j$$

The parameters $\{\alpha_1, \alpha_{2,} \alpha_3, \alpha_4, \alpha_5\}$ are similar to the weights found in a multilayer perceptron in that they are unique for each node. The output y_{ij} is the neurons attempt to fit the overall target output of the network y. The approach is further illustrated in Figure 8.2.

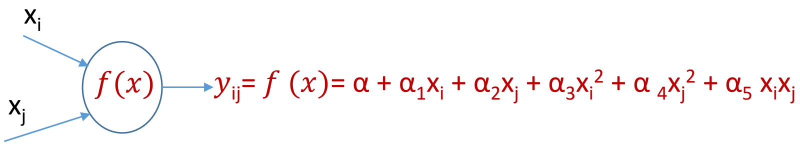

x_i
x_j
$f(x)$
$y_{ij} = f(x) = \alpha + \alpha_1 x_i + \alpha_2 x_j + \alpha_3 x_i^2 + \alpha_4 x_j^2 + \alpha_5 x_i x_j$

Figure 8.2: Node in a GMDH network.

NOTE... ✍

The functions $f(x)$ are known as partial descriptors; they are determined statistically by linear regression using the training set.

The Sample and Test Set

The sample data is divided into training and test sets. The test set is used to estimate the coefficients of the partial descriptors. They are then ranked, typically by mean squared error, and the best partial descriptors selected. The cut off criteria is typically left to the user. The selected neurons become the inputs to the subsequent layer.

NOTE... ✍

GMDH neural networks are not fully interconnected. Nodes with little predictive power are dropped from subsequent computation. It is in this sense that the networks are self organizing.

Forward Propagation

The forward propagation of signals through nodes is similar to the principle used in a feedforward neural network. However, the network is built from the data layer by layer. The first layer consists of functions of each possible pair of the p input attributes.

The input attributes are applied to first layer nodes; each node performs its own polynomial transfer function, the outputs of which are then distributed through the structure to the second layer. The second layer nodes transforms the inputs via polynomial transfer functions, and passes them onto the next layer. This is somewhat different from a traditional feed-forward network where backpropagation ensures all the layers participate simultaneously during the training process.

In a GMDH network, the first network layer consists of a set of quadratic functions of the input attributes, the second layer involves fourth degree polynomials, the third layer includes eighth degree polynomials and so on. The input nodes are connected through a network structure which is made up of several layers of neurons of progressively increasing complexity.

The network increases in depth with each training and selection cycle through the addition of new layers. The algorithm is terminated when little or no further improvement in performance is achieved.

An Important Rule for Data Science

In my very first lecture at graduate school, the brilliant mathematics professor who was to share with us the inner working of measure theory, began with the words:

> "*Lex parsimoniae. Entia non sunt multiplicanda praeter necessitatem.*"

Which roughly translates to the:

“*The law of parsimony or economy. Entities should not be multiplied unnecessarily.*”

This principal is the famous Occam’s Razor; named in honor of the late middle ages scholar William of Ockham.

Here are several interpretations of Occam’s Razor relevant to data science:

- If a smaller set of attributes fits the observations sufficiently well use those attributes. Avoid “*stacking*” additional attributes to improve the fit of a model.

- Select the modeling approach that makes the fewest assumptions.

- Only retain that subset of assumptions which make a clear difference to the predictions of the hypothesis.

- In selecting between hypotheses that explain a phenomenon equal well, it is usually best to start with the simplest one.

- If two or more models have the same prediction accuracy, choose the simplest model.

The appealing thing about Occam’s Razor, as a rule of thumb, is that it neatly captures the idea that in building a decision model you should try to find the smallest set of attributes which provide an adequate description of the data.

The inductive approach developed by GMDH neural networks adheres explicitly to the principle of Occam’s razor. In this, it differs fundamentally from many other types of neural networks, which have a tendency to over-fit data.

NOTE... ✍

Ockham is a peaceful English parish on the east side of the Wey Valley, about 20 miles from London. It is perhaps most famous for being the birthplace in 1285 of Franciscan friar William of Ockham. Friar Ockham was an intellectual rebel. He took part forcefully in the great debates of his time on the poverty of the clergy, transubstantiation, and the right of the popes to exercise temporal power. He was excommunicated by Pope John XXII in 1328.

Inductive Bias

Learning from data requires the ability to generalize from past observations in order to predict new circumstances that are related to past observations. Inductive bias is anything that causes an inductive learner to prefer some hypotheses over other hypotheses. It consists of some fundamental assumption or set of assumptions that the learner makes about the target function that enables it to generalize beyond the training data.

Here are a few examples of inductive bias, which you will no doubt have (at least implicitly) assumed as you built models to learn from data:

- If $\mathcal{H}$ represents linear regression, then $\hat{y}$ assumes the relationship between the attributes x and the target y is linear.

- In the case of the simple regression $y = \alpha + \beta x^q$; the data scientist may only wish to consider those functions where $\alpha > 0$, $3 \leq \beta \leq 7$ and $q > 1.5$.

- In an optimization algorithm, $\mathcal{H}$ consists of the space the algorithm is allowed to search.

Why Inductive Bias is so important

Inductive bias is an important element of your data science practice. This is because as Jonathan Baxter of the London School of Economics explains:

> "*Probably the most important problem in machine learning is the preliminary biasing of a learner's hypothesis space so that it is small enough to ensure good generalization* [ability to predict] *from reasonable training sets, yet large enough that it contains a good solution to the problem to be learnt.*" - see Baxter, Jonathan. "Learning internal representations." Proceedings of the eighth annual conference on Computational learning theory. ACM, 1995.

This point is so important for outstanding data science that world renowned Carnegie Mellon Professor Tom M. Mitchell, writing back in 1980, advocated:

> "*If biases and initial knowledge are at the heart of the ability to generalize beyond observed data, then efforts to study machine learning must focus on the combined use prior knowledge, biases, and observation in guiding the learning process. It would be wise to make the biases and their use in controlling learning just as explicit as past research has made the observations and their use.*" See Mitchell, Tom M. The need for biases in learning generalizations. Department of Computer Science, Laboratory for Computer Science Research, Rutgers Univ., 1980.

Inductive Bias and the GMDH algorithm

The original GMDH algorithm builds the network layer by layer until a stopping criterion is satisfied. For the first layer, all possible combinations of the p attribute variables are used to construct the regression polynomials; therefore there are a total of $\frac{p(p-1)}{2}$ neurons in the first layer. However, this process can take quite a while, and is not always feasible in practice as Josef Taušer of the University of Economics, Prague and Petr Buryan of the Czech Technical University discovered:

> "*Looking for the best structure of the GMDH net during its learning process can be considered as a state space search. The original learning process as presented in Ivakhnenko goes through the whole state space of possible combinations by iteratively creating new layers of nodes connected with the previous ones. When a solution is found that seems to be the locally best (i.e. best within its local neighbourhood formed by solutions of other already existing nodes), the model is constituted by cutting off nodes that are not connected to the chosen one. It is clear that this quickly leads to great expansion of state space to be searched and causes the algorithm to be suitable only for simpler models.*" - See Taušer, Josef, and Petr Buryan. "Exchange Rate Predictions in International Financial Management by Enhanced GMDH Algorithm." Prague Economic Papers 20.3 (2011): 232-249.

In practice, inductive bias is often introduced explicitly into the GMDH algorithm by the end user pre-selecting the number of layers to be included in the model, and thereby reducing the state space searched. This approach is known as building a GMDH-type network. I have found it to be a very efficient and effective way to use GMDH in practice.

Inductive v Deductive Inference

A key difference, as shown in Figure 8.3, between inductive and deductive approaches to inference revolves around the testing of hypothesis. Both approaches begin by observing a phenomenon of interest, however the focus in inductive methods is typically on selecting the best model for prediction.

In the deductive approach, the focus is on exploring theory, with the data used primarily to test a hypothesis about that theory. The hypothesis is either rejected or accepted based on the "*weight of evidence*" derived from the empirical data.

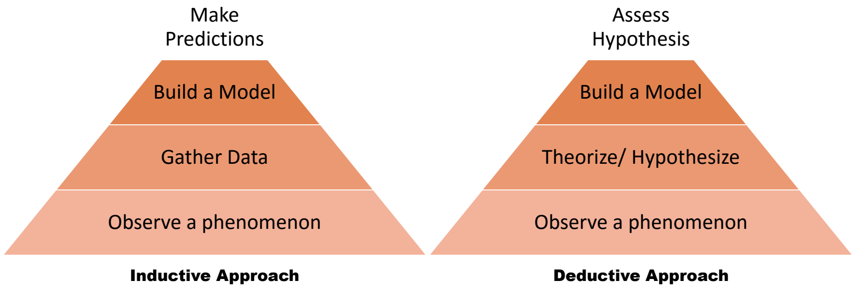


Figure 8.3: Induction and Deduction

A Short Story

Cambridge, England in the year 1833. Adolphe Quetelet, the renowned Belgian astronomer, mathematician, statistician and sociologist, collected together his papers, and hurried out of his lodgings. As he strode rapidly along, his hand gripped tightly onto the bundle. They contained data on crime in parts of France and Belgium. His destination was not far - the British Association was meeting, and he hoped to present his data. He was the official Belgian delegate. His freshly complied numbers illuminated new insights and needed "*airing*".

An old historical debate

Throughout all history there has always been excitement (and distrust) about the collection and use of fresh empirical data to develop new insights. As Sir William Petty writing in 1690 expressed:

> "*for instead of using only comparative and superlative Words, and intellectual Arguments, I have taken the course to express my self in Terms of Number, Weight, or Measure; to use only Arguments of Sense, and to consider only such Causes, as have visible Foundations in Nature; leaving those that depend upon the mutable Minds, Opinions, Appetites, and Passions of particular Men, to the Consideration of others.*" See Hull, Charles Henry. The Economic Writings of Sir William Petty. CUP Archive, 1690.

In 1833, there was considerable skepticism of the *"Political Arithmetick"* of scholars like Adolphe Quetelet. Senior officials of the British Association left no room on the agenda for the official Belgian delegate's talk! What was Adolphe Quetelet to do?

Richard Jones

Fortunately, Richard Jones offered up his private room for a discussion of *"Quetelet's data"*. Among the attendees was T.R. Malthus and Charles Babbage. *"Let us form a new section of the British Association to deal with statistics."* suggested Babbage; and with those words (or something like them) what is today known as the Royal Statistical Society was birthed. It was formed several months later - March 15th 1834 as the Statistical Society of London.

Back in 1833 the debate between deductive and inductive methods ragged even more fiercely than it does today. Babbage was well aware that if his statistics group had sought prior permission to form, the proposal would have been rejected. As computer scientist and United States Navy Rear Admiral Grace Hopper was found of saying:

"*It is often easier to ask for forgiveness than to ask for permission.*"

The group founded by Babbage was initially restricted by the British Association to:

"*facts relating to communities of men which are capable of being expressed by numbers, and which promise, when sufficiently multiplied, to indicate general laws.*"

Strangely enough, this was essentially the data driven, inductive approach popularized by Machine Learning today. It was attacked on numerous occasions as being unscientific. The accusations of "*unscientific Political Arithmetick*" peaked in an all out assault, in 1877, by the British Association to abolish its statistical section as not being properly scientific. The attached was successfully repelled.

Thrashing the wheat

Even the logo of the Statistical Society, a wheatsheaf, bears the hallmarks of the debate. It once contained the Latin motto *alils extrendum - to be threshed by other*s, see Figure 8.4. In 1857, the motto was removed, and as renowned historian I.D Hill comments:

"*Statisticians would 'thresh the wheat themselves'* ".

This is because, as I.D Hill also reports of a statistical scholar writing in 1836, it is:

"*...impossible to frame a Statistical exhibition of the present subject for practically useful purposes, without theorizing.*"

Figure 8.4: Wheatsheaf logo with original Latin motto (left). Tided up version with logo removed (right)

GMDH and the Core Objective of Machine Learning

Not too long ago, deductive analytics was *“King of the Empirical Hill”*. It ascended the steep steps to the pinnacle riding on the back of Karl Popper’s notion of testability, in which he claimed that testable hypotheses are to be preferred because they are a more efficient means for advancing scientific knowledge.

NOTE... ✍

Karl’s seminal work was originally published in German in 1939 as Logik der Forschung. Fortunately, after the second world war, he re-wrote it in English. Get a copy, it is jam packed with ideas that will challenge your thinking. See Popper, Karl R. "The logic of scientific discovery." London: Hutchinson (1959).

An economic lesson

I recall, in my theoretical economics class, the stern warning from the professor that the "*data cannot be trusted*". It seems this experience was not just limited to my class. A famous professor of econometrics explains:

> "*A widely held view in economics is that the current untrustworthiness of empirical evidence and the inability to forecast economic phenomena is largely due to the fact that the economy is too complicated and the resulting data are too heterogeneous to be amenable to statistical modeling.*" - See Spanos, Aris. "Learning from data: The role of error in statistical modeling and inference." Unpublished manuscript (2012).

Once outside of the classroom, and in the real world of empirical analysis, I soon discovered that provided you have sufficient data and access to the appropriate tools, a data driven inductive approach can yield significant insights.

One of the central objectives of Machine Learning is to automate the entire learning from data process. Considerably success has been achieved in this task. Again and again, results from industry, academia and science have proven that voluminous amounts of empirical data can be fruitfully "mined" to derive models to estimate useful relationships between specified attributes and target variables.

So rapid has been the rise of machine learning, and so stunning its successes that it has become an indispensable part of many application areas, in the sciences, engineering, the social sciences, and even in the arts. Across a broad swath of academic disciplines it has stamped its mark with best in class performance.

NOTE... ✍

Yes, even in the arts! See for example:

- Thoma, Martin. "Creativity in Machine Learning." arXiv preprint arXiv:1601.03642 (2016).
- Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. "A neural algorithm of artistic style." arXiv preprint arXiv:1508.06576 (2015).

The bottom line

Whether you are working in the area of medical diagnosis, hand-written character recognition, marketing, financial forecasting, bioinformatics, economics or any other discipline that requires empirical analysis, you will frequently face the situation where the underlying first principles are unknown or the systems under study are too complex to be mathematically described in sufficient detail to provide useful results. I have found an inductive data driven approach valuable in all of these situations, and so will you.

Practical Application of Self-Organizing Polynomial Neural Networks

GMDH is an inductive self-organizing approach because gradually more complicated models are generated based on evaluation of their performance on a set of multi-input single-output data pairs. A priori knowledge of the data generating process is not required.

This allows you to easily, quickly use GMDH type neural networks to model complex systems without having specific knowledge of the underlying system dynamics. Let's take a look at two applications of this powerful technique.

Coal Mining Accidents

Indian coal mining engineers apply the GMDH-type neural network to predict the annual number of fatal accidents from coal mining in India, over the period 1970 to 2012. Although the dataset have a very limited number of observations, the researchers were able to successfully fit a GMDH type neural network to the data.

They quickly found that the GMDH type neural network provided valuable insight about both the trend, and level of future accidents.

The researchers also built a range of alternative forecasting models, including traditional auto regressive integrated moving average (ARIMA) time series models, and Support Vector Machines. The GMDH type neural network performed relatively well against these alternative modeling techniques.

Stock Specific Price Forecasting

Stock price prediction is a popular and profitable topic. Investors, regulators and risk managers all want to gain insight into the future direction of the stock market. The GMDH type neural network is appealing for this type of task, due to its ability to capture complex time series dynamics. Its potential was highlighted by Saeed Fallahi, who built models to predict the stock price of cement companies trading on Tehran stock exchange.

Stock price dynamics are determined by many factors including the current and future earnings per share, Dividend per share, and the price-earnings ratio. These, and other factors, were selected as input attributes for the model.

Daily price data was collected over the period 1999-2008. The training sample used 80% of the available observations, with the test set using the remainder.

The optimal model had 2 hidden layers, and was used to predict the stock price of 10 individual cement companies. Performance between predicted and observed values was measured using the R-squared statistic. It ranged between 0.99 and 1.0 for all companies. The test set R-squared ranged from 0.98 to 1.0.

Example - Forecasting Monthly Sunspot Numbers

The study of sunspot activity has practical importance to geophysicists, environment scientists, climatologists, and telecommunications companies. In this section, we build a GMDH type neural network to forecast the monthly number of sunspots.

Step 1 – Collecting Data

The R data frame `sunspot.month` contains monthly numbers of sunspots from 1749. Let's grab and plot the data:

```
plot(sunspot.month,
ylab="Number of Sunspots",
xlab="Year",
col="darkblue")
```

Figure 8.5 plots the time series. It indicates there is a long term cyclical pattern to sunspot activity.

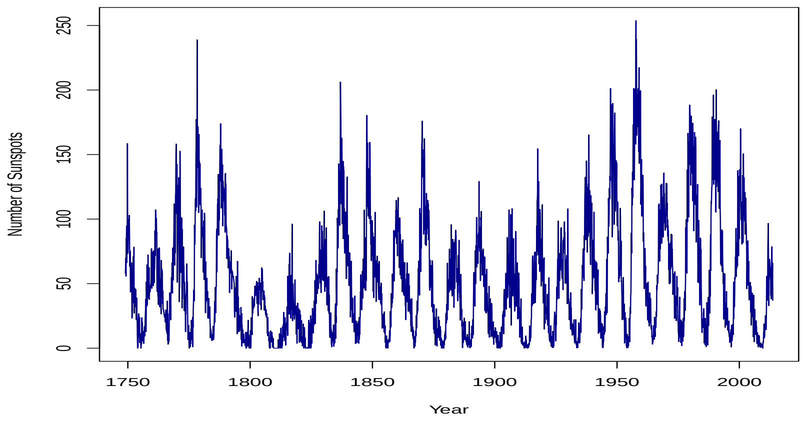


Figure 8.5: Monthly sunspot numbers since 1749.

Step 2 – Exploring and Preparing the Data

For the model, we use the previous sixty months as the training set:

```
data(sunspot.month)
data<-sunspot.month[3113:3172]
data <- ts(matrix(data),
end = c(2013, 4),
frequency = 12)
```

Notice, we use the `ts` and `matrix` functions to convert the `data` object into a `ts` [R time series] class.

Now take a quick peek at the training data:

```
plot(data,
xlab="Year",
ylab="Number of Sunspots",
col="darkblue")
```

Figure 8.6 shows the resultant plot. Over the training period, the number of sunspots has exhibited a general upward trend leveling off between 2012 and 2013.

NOTE... ✍

Analysis in this chapter was carried out using version 1.6 of the GMDH package.

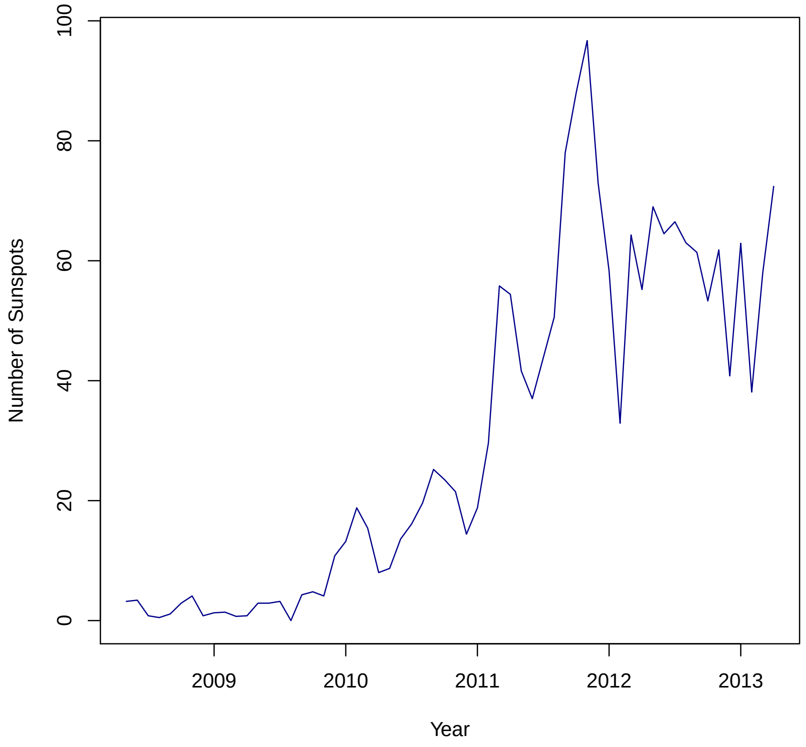


Figure 8.6: Training data used in sunspots prediction.

Step 3 – Training a Model on the Data

The model can be fitted using the `fcast` function from the `GMDH` package. The `fcast` function takes four core arguments.

- The first argument takes the training and test sample used to build the model;
- The second argument is the number of inputs into the network;
- and the third argument is the number of layers;
- Finally, the number of time units (months in our case) ahead to forecast.

For this example, we use 3 observations for input, with a 2 layers model. Since we are forecasting from May to September, the number of period ahead forecasts is set to 5:

```
require (GMDH)
fit = fcast(data,
input = 3,
layer = 2,
f.number = 5)


     Point Forecast    Lo 95    Hi 95
2068       79.61365 63.58931 95.63800
2069       77.95074 56.71379 99.18769
2070       67.21099 41.68679 92.73520
2071       57.00216 33.05195 80.95237
2072       52.65503 30.51590 74.79415
```

The model reports the forecasts along with a 95% confidence interval. For example, for May (point 2068) the model forecasts a value of 79.61. The upper range on this forecast is 95.63, and the lower range is 63.58.

How well did the model do? Figure 8.7 plots the predictions (red line), their confidence intervals (dotted line), and the actual observations (blue line). It seems all the actual observation, except for June, fall within the 95% confidence interval.

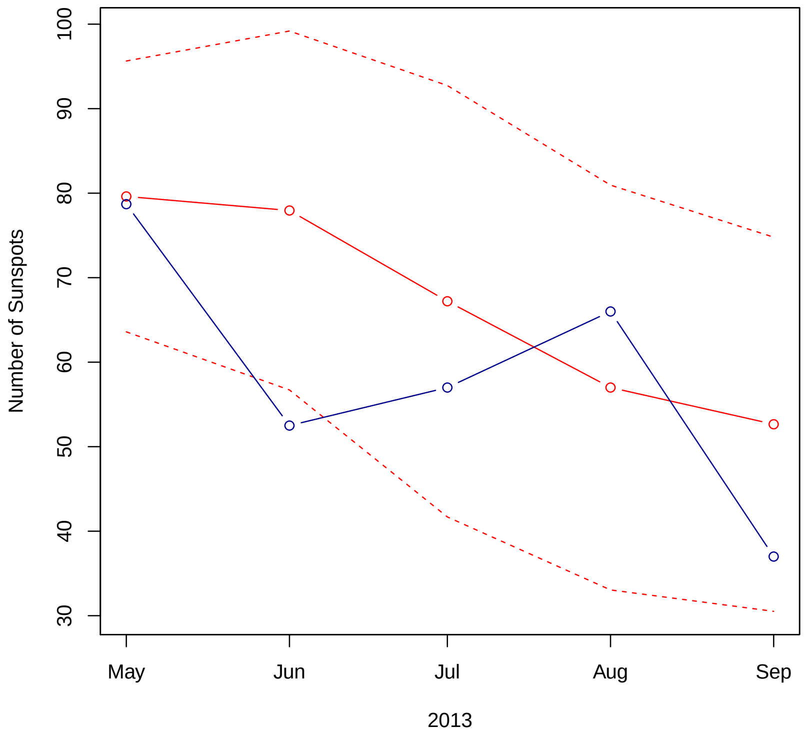


Figure 8.7: Forecast with 95% confidence interval and actual observations.

NOTE... ✍

The fitted values for the training data can be accessed by appending `fit` with the `$fitted` argument.

Step 4 – Improving Model Performance

We would like to improve the performance of the model. Ideally, all the actual observations should fall inside our 95% confidence interval. Using more observations as inputs might improve the fit. Suppose we increase the number from 3 to 15:

```
fit = fcast(data,
input = 15,
layer = 2,
f.number = 5)
```

```
     Point Forecast    Lo 95    Hi 95
2068       71.70897 55.32897 88.08898
2069       68.96809 47.07059 90.86559
2070       60.16692 34.99724 85.33660
2071       58.12570 34.05412 82.19727
2072       57.42110 34.63536 80.20685
```

Figure 8.8 shows the refitted model comfortable contains the actual observations within the 95% confidence interval.

Changing the Confidence Level

A nice feature of `fcast` is that you can change the confidence interval by adding the argument `level`. For example, to specify a 99% confidence interval add the argument `level = 0.99`. So, using our original model we would have:

```
fit = fcast(data,
input = 3,
layer = 2,
f.number = 5,
level=0.99)
```

```
     Point Forecast  Lo 0.99  Hi 0.99
2068       79.61365 79.51220 79.71510
```

```
2069         77.95074 77.81629 78.08519
2070         67.21099 67.04940 67.37258
2071         57.00216 56.85053 57.15378
2072         52.65503 52.51487 52.79518
```

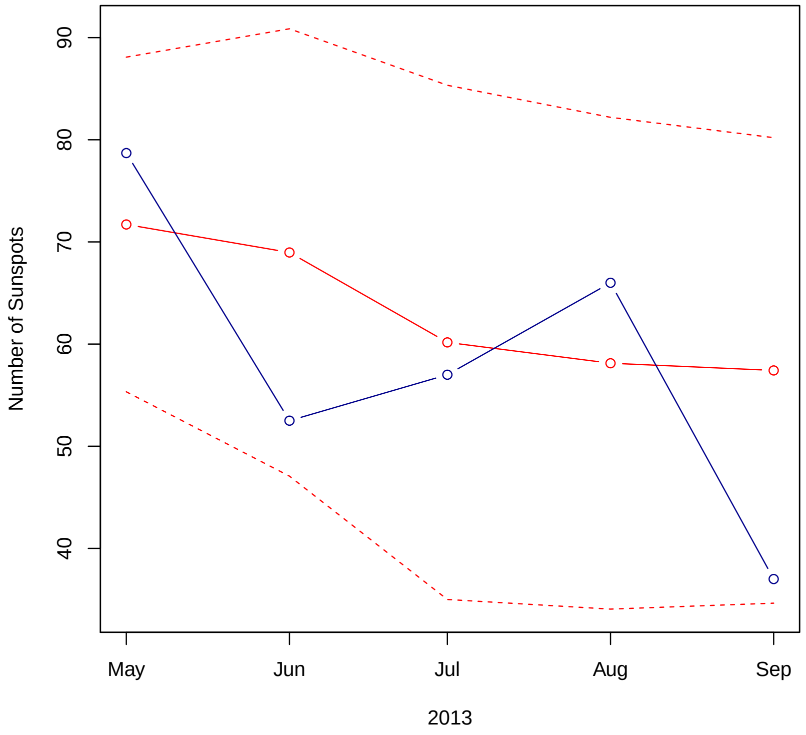
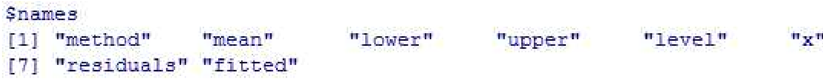


Figure 8.8: Predicted and actual values using 15 inputs.

Exploring Model Attributes

The `attributes` method provides details on the list of values stored in the fitted R object `fit`:

```
attributes (fit)
```



```
$names
[1] "method"    "mean"      "lower"     "upper"     "level"     "x"
[7] "residuals" "fitted"
```

For example, the predicted values are obtained by appending `$mean` to the `fit` object:

```
fit$mean

Time Series:
Start = 2068
End = 2072
Frequency = 1
[1] 79.61365 77.95074 67.21099 57.00216
   52.65503
```

Step 5 – Creating One Month Ahead Forecasts

Suppose you want to build a model to predict the one-month ahead number of sunspots over an extended period of time. How can you do this?

In this next illustration, I explain a simple way to achieve this. We will use sixty observations to train the model. The start date is January 2012, and we calculate the one month ahead predictions to September 2013. This will give us a total of 21 monthly predictions.

Core parameters

First, gather together a few parameters:

```
start=3157
len=60
end = 21
k=-1
```

Let's spend a moment discussing each of these parameters in turn.

- The parameter `start` contains the last month of training data. In this case, December 2011. The first forecast will be for January 2012.

- For this example, we use a rolling 60 month window, so the parameter `len`, which contains the number of observations to be used in the training sample, is set to 60.

- The parameter, `end` contains the number of 1 step ahead forecasts; in this case; 21 months.

- Finally, `k` is a simple count variable used to roll the sample data forward.

The main loop

Next, we build the main loop:

```
for (i in start:((start+end)-1))
{
k=k+1
data<-sunspot.month[(start-len+1+k):(start+
   k)]
data <- ts(matrix(data),frequency = 12)
fit = fcast(data,
input = 4,
layer = 2,
f.number = 1)
if(i==start) forecast<-as.numeric(fit$mean)
if (i>start) forecast<-cbind(forecast,as.
   numeric(fit$mean))
}
```

Let's take a moment to go over the above code:

- In this example, we fit a model using 2 layers, with 4 inputs.

- The argument `f.number` is set equal to 1, to indicate a 1 time unit ahead forecast.

- The loop cycles around for each of the 21 predictions using a rolling window of 60 observations.

`Metrics` Package

Once the loop is complete, we will need to asses the performance of fitted predictions. We use the root mean square error from the `Metrics` package:

```
y = sunspot.month[(start+1):(start+end)-1]
require(Metrics)
rmse(forecast,y)
[1] 5.920988
```

The overall error is 5.92. Figure 8.9 shows the predicted and actual observations over the 21 month window. Notice that a handful of the observations lie outside of the 95% confidence interval. One way to deal with this, as we have already seen, would be to use a different number of inputs. You could also use a different rolling sample size.

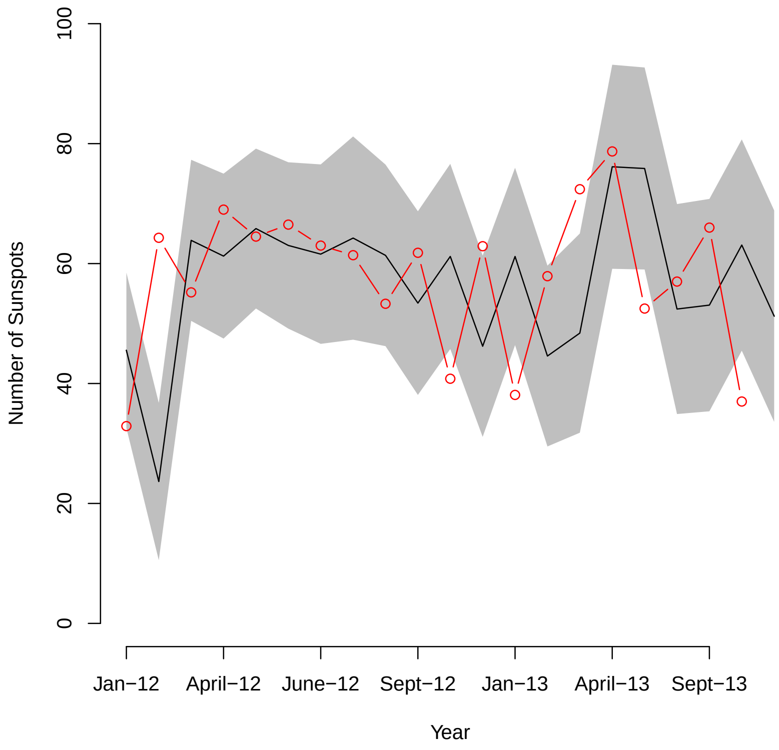


Figure 8.9: Actual and predicted values with 95% level confidence interval.

NOTE... ✍

Try rebuilding the model with a larger number of inputs and a smaller sample size. What do you observe?

Using Akaike's Information Criterion

A nice feature of the GMDH package is the ability to use Akaike's information criterion for assessing the partial descriptors. You can access this by setting the argument `method = "RGMDH"` in the `fcast` function. To see it in action, let's predict the 21 monthly observations used earlier, with a 3 layer model using 3 inputs:

```
k=-1
for (i in start:((start+end)-1)) {
k=k+1
data<-sunspot.month[(start-len+1+k):(start+
   k)]
data <- ts(matrix(data),frequency = 12)
fit = fcast(data,
input = 3,
layer = 3,
f.number = 1,
method = "RGMDH")
if(i==start) forecast1<-as.numeric(fit$mean
   )
if (i>start) forecast1<-cbind(forecast1,as.
   numeric(fit$mean))

}
```

The above code simply repeats that shown on page 208. So we won't discuss it here.

Now, calculate the performance metrics:

```
y = sunspot.month[(start+1):(start+end)-1]
require(Metrics)
rmse(forecast1,y)
[1] 6.188309
```

The model has a slightly higher error than the previous model. Nevertheless, as shown in Figure 8.10, it captures much of the dynamics of the underlying sun spots time series. Indeed, in this case, the fit is very similar to that of previous model.

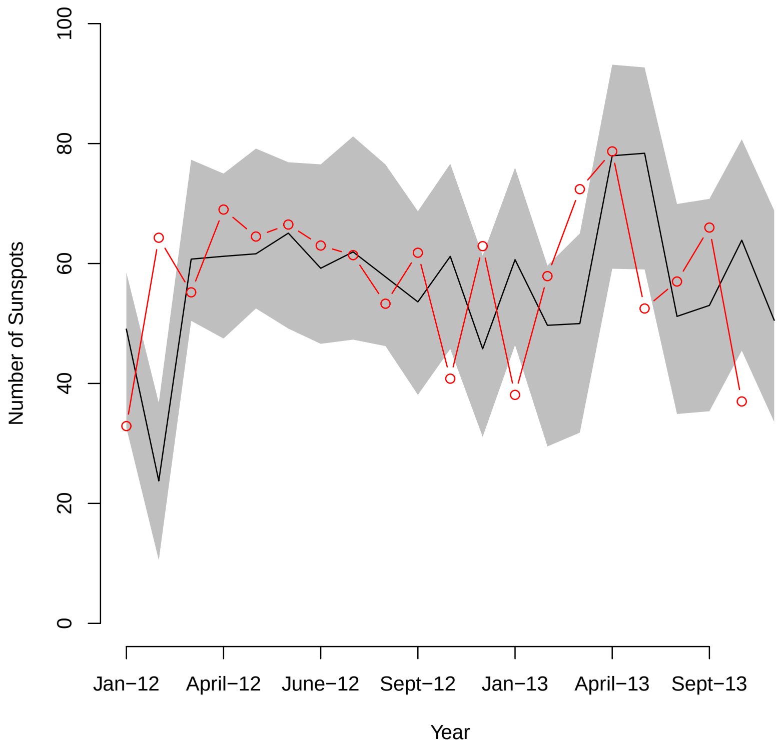


Figure 8.10: Actual and predicted values using the revised GMDH network.

Summary

In this chapter, you got a flavor of the GMDH type neural networks, details of practical applications for which it has been successfully deployed, and you got to try your hand at building GMDH type neural networks to forecast sunspot activity.

GMDH type neural networks are a relatively old technology. However, they can be used to provide useful insights for many types of time series data. For some reason, they remain a relatively obscure technique. Yet, the results they generate are well worth the effort.

Suggested Reading

To find out more about the practical applications discussed in this chapter see:

- **Coal Mining Accidents:** Kher, Anupam Anant, and Rajendra Yerpude. "Application of Forecasting Models on Indian Coal Mining Fatal Accident (Time Series) Data." International Journal of Applied Engineering Research 11.2 (2016): 1533-1537.

- **Stock Specific Price Forecasting:** Bashiri, Vahab, et al. "Application of GMDH-type neural network to stock price prediction of Iran's auto industry." International Journal of Business Forecasting and Marketing Intelligence 2.4 (2016): 359-378.

Other

- **History of the Royal Statistical Society:** Hill, I. D. "Statistical Society of London–Royal Statistical Society: The First 100 Years: 1834-1934." Journal of the Royal Statistical Society. Series A (General) (1984): 130-139.).

- **Across all disciplines**: Abbod, Maysam, and Karishma Deshpande. "Using intelligent optimization methods to improve the group method of data handling in time series prediction." Computational Science–ICCS 2008. Springer Berlin Heidelberg, 2008. 16-25.

- **Poultry production**: Mottaghitalab, M., et al. "Predicting caloric and feed efficiency in turkeys using the group method of data handling-type neural networks." Poultry science 89.6 (2010): 1325-1331.

- **Image processing**: Kondo, Tadashi, and Junji Ueno. "Medical image recognition of the brain by revised GMDH-type neural network algorithm with a feedback loop." International Journal of Innovative Computing Information and Control 2.5 (2006): 1039-1052.

- **Exchange rate prediction**: Taušer, Josef, and Petr Buryan. "Exchange Rate Predictions in International Financial Management by Enhanced GMDH Algorithm." Prague Economic Papers 20.3 (2011): 232-249.

- **Molecular sciences**: Bennett, Frederick R., Peter Crew, and Jennifer K. Muller. "A GMDH approach to modelling gibbsite solubility in Bayer process liquors." International journal of molecular sciences 5.3 (2004): 101-109.

- **Commodity prices**: Mehrara, Mohsen, Ali Moeini, and M. Ahrari. "Comparison of two approach of neural network for forecasting of oil future prices." Asia J. Bus. Manage. Sci 1.3 (2011).

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