Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks

Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks

Jeff Heaton

Heaton Research, Inc.
St. Louis, MO, USA

Do not make illegal copies of this ebook

Title	AIFH, Volume 3: Deep Learning and Neural Networks
Author	Jeff Heaton
Published	December 31, 2015
Copyright	Copyright 2015 by Heaton Research, Inc., All Rights Reserved.
ISBN	978-1505714340
Price	9.99 USD
File Created	Sun Nov 08 15:28:07 CST 2015

Publisher: Heaton Research, Inc.
Artificial Intelligence for Humans, Volume 3: Neural Networks and Deep Learning
December, 2015
Author: Jeff Heaton
Editor: Tracy Heaton
ISBN: 978-1505714340
Edition: 1.0

Heaton Research, Encog, the Encog Logo and the Heaton Research logo are all trademarks of Heaton Research, Inc., in the United States and/or other countries.

TRADEMARKS: Heaton Research has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, so the content is based upon the final release of software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

SOFTWARE LICENSE AGREEMENT: TERMS AND CONDITIONS

The media and/or any online materials accompanying this book that are available now or in the future contain programs and/or text files (the "Software") to be used in connection with the book. Heaton Research, Inc. hereby grants to you a license to use and distribute software programs that make use of the compiled binary form of this book's source code. You may not redistribute the source code contained in this book, without the written permission of Heaton Research, Inc. Your purchase, acceptance, or use of the Software will constitute your acceptance of such terms.

The Software compilation is the property of Heaton Research, Inc. unless otherwise indicated and is protected by copyright to Heaton Research, Inc. or other copyright owner(s) as indicated in the media files (the "Owner(s)"). You are hereby granted a license to use and distribute the Software for your personal, noncommercial use only. You may not reproduce, sell, distribute, publish, circulate, or commercially exploit the Software, or any portion thereof, without the written consent of Heaton Research, Inc. and the specific copyright owner(s) of any component software included on this media.

In the event that the Software or components include specific license requirements or end-user agreements, statements of condition, disclaimers, limitations or warranties ("End-User License"), those End-User Licenses supersede the terms and conditions herein as to that particular Software component. Your purchase, acceptance, or use of the Software will constitute your acceptance of such End-User Licenses.

By purchase, use or acceptance of the Software you further agree to comply with all export laws and regulations of the United States as such laws and regulations may exist from time to time.

SOFTWARE SUPPORT

Components of the supplemental Software and any offers associated with them may be supported by the specific Owner(s) of that material but they are not supported by Heaton Research, Inc.. Information regarding any available support may be obtained from the Owner(s) using the information provided in the appropriate README files or listed elsewhere on the media.

Should the manufacturer(s) or other Owner(s) cease to offer support or decline to honor any offer, Heaton Research, Inc. bears no responsibility. This notice concerning support for the Software is provided for your information only. Heaton Research, Inc. is not the agent or principal of the Owner(s), and Heaton Research, Inc. is in no way responsible for providing any support for the Software, nor is it liable or responsible for any support provided, or not provided, by the Owner(s).

WARRANTY

Heaton Research, Inc. warrants the enclosed media to be free of physical defects for a period of ninety (90) days after purchase. The Software is not available from Heaton Research, Inc. in any other form or media than that enclosed herein or posted to www.heatonresearch.com. If you discover a defect in the media during this warranty period, you may obtain a replacement of identical format at no charge by sending the defective media, postage prepaid, with proof of purchase to:

Heaton Research, Inc.
Customer Support Department
1734 Clarkson Rd #107
Chesterfield, MO 63017-4976
Web: www.heatonresearch.com
E-Mail: support@heatonresearch.com

DISCLAIMER

Heaton Research, Inc. makes no warranty or representation, either expressed or implied, with respect to the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will Heaton Research, Inc., its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, Heaton Research, Inc. further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by Heaton Research, Inc. reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

SHAREWARE DISTRIBUTION

This Software may use various programs and libraries that are distributed as shareware. Copyright laws apply to both shareware and ordinary commercial software, and the copyright Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

This book is dedicated to my mom Mary,
thank you for all the love
and encouragement over the years.
.

Contents

Introduction

- Series Introduction
- Example Computer Languages
- Prerequisite Knowledge
- Fundamental Algorithms
- Other Resources
- Structure of this Book

This book is the third in a series covering select topics in artificial intelligence (AI), a large field of study that encompasses many sub-disciplines. In this introduction, we will provide some background information for readers who might not have read Volume 1 or 2. It is not necessary to read Volume 1 or 2 before this book. We introduce needed information from both volumes in the following sections.

0.1 Series Introduction

This series of books introduces the reader to a variety of popular topics in artificial intelligence. By no means are these volumes intended to be an exhaustive AI resource. However, each book presents a specific area of AI to familiarize the reader with some of the latest techniques in this field of computer science.

In this series, we teach artificial intelligence concepts in a mathematically gentle manner, which is why we named the series *Artificial Intelligence for Humans*. As a result, we always follow the theories with real-world programming examples and pseudocode instead of relying solely on mathematical formulas. Still, we make these assumptions:

- The reader is proficient in at least one programming language.
- The reader has a basic understanding of college algebra.
- The reader does not necessarily have much experience with formulas from calculus, linear algebra, differential equations, and statistics. We will introduce these formulas when necessary.

Finally, the book's examples have been ported to a number of programming languages. Readers can adapt the examples to the language that fits their particular programming needs.

0.1.1 Programming Languages

Although the book's text stays at the pseudocode level, we provide example packs for Java, C# and Python. The Scala programming language has a community-supplied port, and readers are also working on porting the examples to additional languages. So, your favorite language might have been ported since this printing. Check the book's GitHub repository for more information. We highly encourage readers of the books to help port to other languages. If you would like to get involved, Appendix A has more information to get you started.

0.1.2 Online Labs

Many of the examples from this series use JavaScript and are available to run online, using HTML5. Mobile devices must also have HTML5 capability to run the programs. You can find all online lab materials at the following web site:

`http://www.aifh.org`

These online labs allow you to experiment with the examples even as you read the e-book from a mobile device.

0.1.3 Code Repositories

All of the code for this project is released under the Apache Open Source License v2 and can be found at the following GitHub repository:

`https://github.com/jeffheaton/aifh`

If you find something broken, misspelled, or otherwise botched as you work with the examples, you can fork the project and push a commit revision to GitHub. You will also receive credit among the growing number of contributors. Refer to Appendix A for more information on contributing code.

0.1.4 Books Planned for the Series

The following volumes are planned for this series:

- Volume 0: Introduction to the Math of AI
- Volume 1: Fundamental Algorithms
- Volume 2: Nature-Inspired Algorithms
- Volume 3: Deep Learning and Neural Networks

We will produce Volumes 1, 2, and 3 in order. Volume 0 is a planned prequel that we will create near the end of the series. While all the books will include the required mathematical formulas to implement the programs, the prequel will recap and expand on all the concepts from the earlier volumes. We also intend to produce more books on AI after the publication of Volume 3.

In general, you can read the books in any order. Each book's introduction will provide some background material from previous volumes. This organization allows you to jump quickly to the volume that contains your area of interest. If you want to supplement your knowledge at a later point, you can read the previous volume.

0.1.5 Other Resources

Many other resources on the Internet will be very useful as you read through this series of books.

The first resource is Khan Academy, a nonprofit, educational website that provides videos to demonstrate many areas of mathematics. If you need additional review on any mathematical concept in this book, Khan Academy probably has a video on that information.

`http://www.khanacademy.org/`

The second resource is the Neural Network FAQ. This text-only resource has a great deal of information on neural networks and other AI topics.

`http://www.faqs.org/faqs/ai-faq/neural-nets/`

Although the information in this book is not necessarily tied to Encog, the Encog home page has a fair amount of general information on machine learning.

`http://www.encog.org`

0.2 Neural Networks Introduction

Neural networks have been around since the 1940s, and, as a result, they have quite a bit of history. This book will cover the historic aspects of neural networks because you need to know some of the terminology. A good example of this historic progress is the activation function, which scales values passing through neurons in the neural network. Along with threshold activation functions, researchers introduced neural networks, and this advancement gave way to sigmoidal activation functions, then to hyperbolic tangent functions and now to the rectified linear unit (ReLU). While most current literature suggests using the ReLU activation function exclusively, you need to understand sigmoidal and hyperbolic tangent to see the benefits of ReLU.

Whenever possible, we will indicate which architectural component of a neural network to use. We will always identify the architectural components now accepted as the recommended choice over older classical components. We will bring many of these architectural elements together and provide you with some concrete recommendations for structuring your neural networks in Chapter 14, “Architecting Neural Networks.”

Neural networks have risen from the ashes of discredit several times in their history. McCulloch, W. and Pitts, W. (1943) first introduced the idea of a neural network. However, they had no method to train these neural networks. Programmers had to craft by hand the weight matrices of these early networks. Because this process was tedious, neural networks fell into disuse for the first time.

Rosenblatt, F. (1958) provided a much-needed training algorithm called backpropagation, which automatically creates the weight matrices of neural networks. It fact, backpropagation has many layers of neurons that simulate the architecture of animal brains. However, backpropagation is slow, and, as the layers increase, it becomes even slower. It appeared as if the addition of computational power in the 1980s and early 1990s helped neural networks perform tasks, but the hardware and training algorithms of this era could not effectively train neural networks with many layers, and, for the second time, neural networks fell into disuse.

The third rise of neural networks occurred when Hinton (2006) provided a radical new way to train deep neural networks. The recent advances in high-speed graphics processing units (GPU) allowed programmers to train neural networks with three or more layers and led to a resurgence in this technology as programmers realized the benefits of deep neural networks.

In order to establish the foundation for the rest of the book, we begin with an analysis of classic neural networks, which are still useful for a variety of tasks. Our analysis includes concepts, such as self-organizing maps (SOMs), Hopfield neural networks, and Boltzmann machines. We also introduce the feedforward neural network and show several ways to train it.

A feedforward neural network with many layers becomes a deep neural network. The book contains methods, such as GPU support, to train deep networks. We also explore technologies related to deep learning, such as dropout, regularization, and convolution. Finally, we demonstrate these techniques through several real-world examples of deep learning, such as predictive modeling and image recognition.

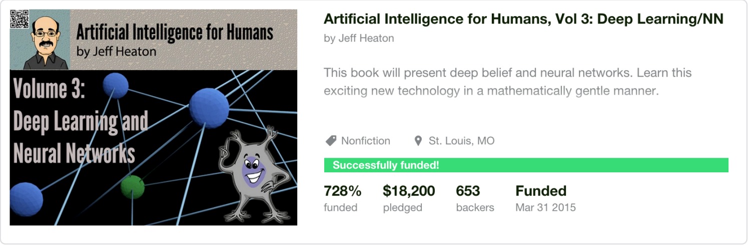
If you would like to read in greater detail about the three phases of neural network technology, the following article presents a great overview:

`http://chronicle.com/article/The-Believers/190147/`

0.3 The Kickstarter Campaign

In 2013, we launched this series of books after a successful Kickstarter campaign. Figure 1 shows the home page of the Kickstarter project for Volume 3:

Figure 1: The Kickstarter Campaign



Artificial Intelligence for Humans, Vol 3: Deep Learning/NN
by Jeff Heaton

This book will present deep belief and neural networks. Learn this exciting new technology in a mathematically gentle manner.

Nonfiction St. Louis, MO

Successfully funded!

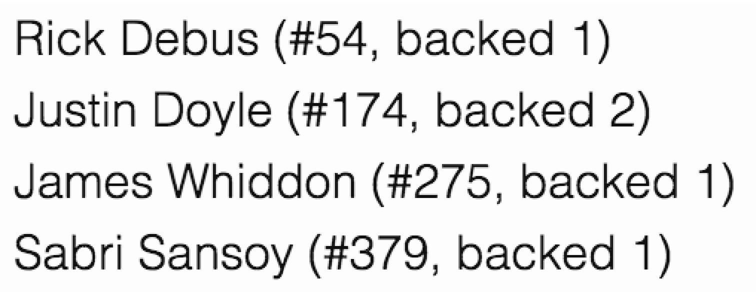
728%	$18,200	653	Funded
funded	pledged	backers	Mar 31 2015

You can visit the original Kickstarter at the following link:

`https://goo.gl/zW4dht`

We would like to thank all of the Kickstarter backers of the project. Without your support, this series might not exist. We would like to extend a huge thank you to those who backed at the $250 and beyond level:

Figure 2: Gold Level Backers



Rick Debus (#54, backed 1)

Justin Doyle (#174, backed 2)

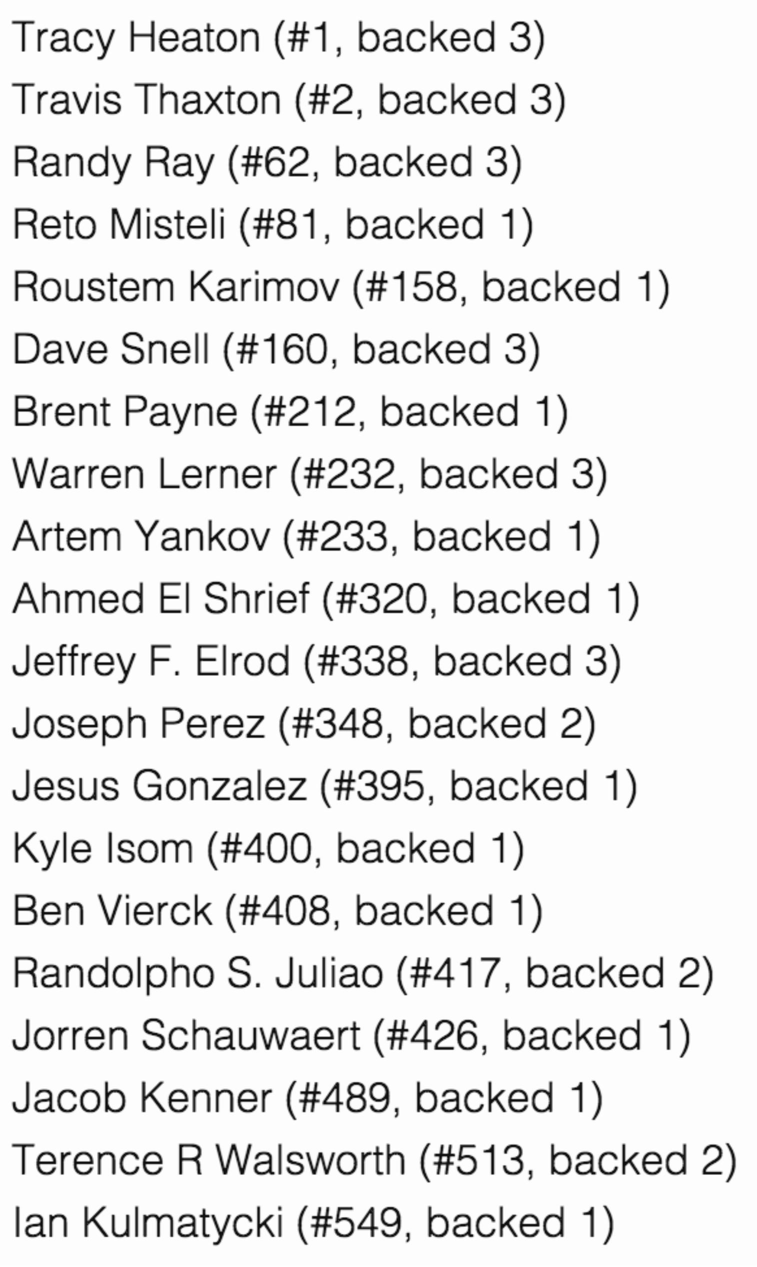
James Whiddon (#275, backed 1)

Sabri Sansoy (#379, backed 1)

It will be great discussing your projects with you. Thank you again for your support.

We would also like to extend a special thanks to those backers who supported the book at the $100 and higher levels. They are listed here in the order that they backed:

Figure 3: Silver Level Backers



Tracy Heaton (#1, backed 3)
Travis Thaxton (#2, backed 3)
Randy Ray (#62, backed 3)
Reto Misteli (#81, backed 1)
Roustem Karimov (#158, backed 1)
Dave Snell (#160, backed 3)
Brent Payne (#212, backed 1)
Warren Lerner (#232, backed 3)
Artem Yankov (#233, backed 1)
Ahmed El Shrief (#320, backed 1)
Jeffrey F. Elrod (#338, backed 3)
Joseph Perez (#348, backed 2)
Jesus Gonzalez (#395, backed 1)
Kyle Isom (#400, backed 1)
Ben Vierck (#408, backed 1)
Randolpho S. Juliao (#417, backed 2)
Jorren Schauwaert (#426, backed 1)
Jacob Kenner (#489, backed 1)
Terence R Walsworth (#513, backed 2)
Ian Kulmatycki (#549, backed 1)

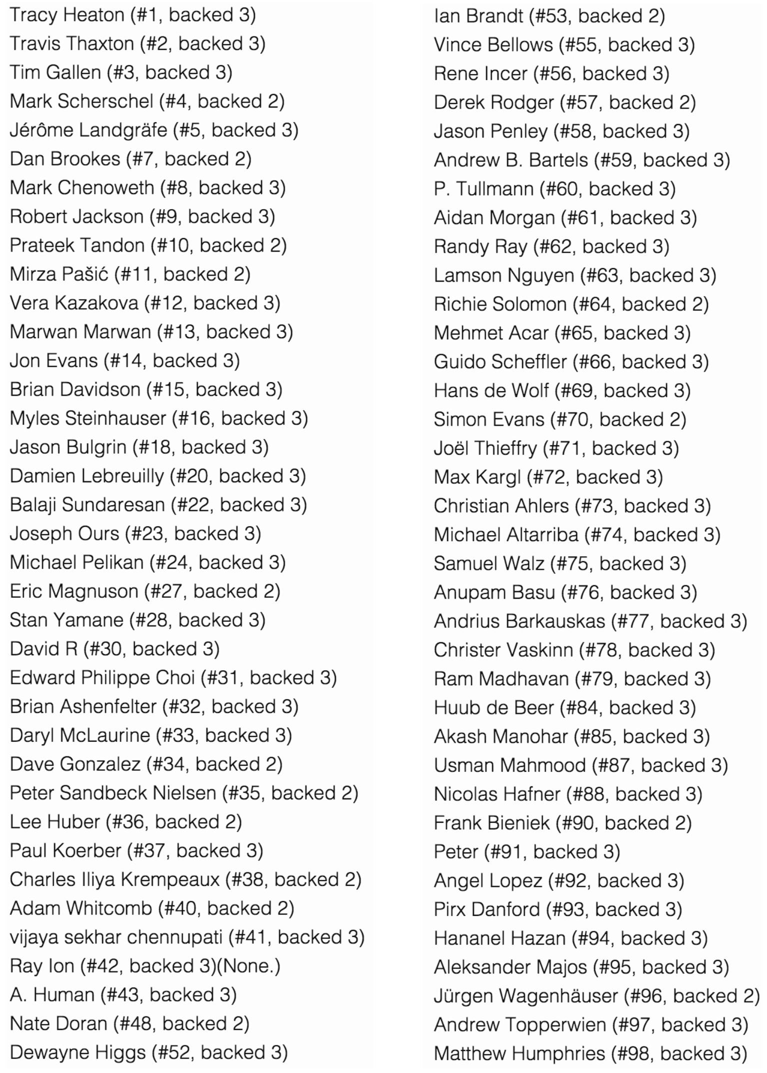
A special thank you to my wife, Tracy Heaton, who edited the previous two volumes.

There have been three volumes so far; the repeat backers have been very valuable to this campaign! It is amazing to me how many repeat backers there are!

Thank you, everyone–you are the best!

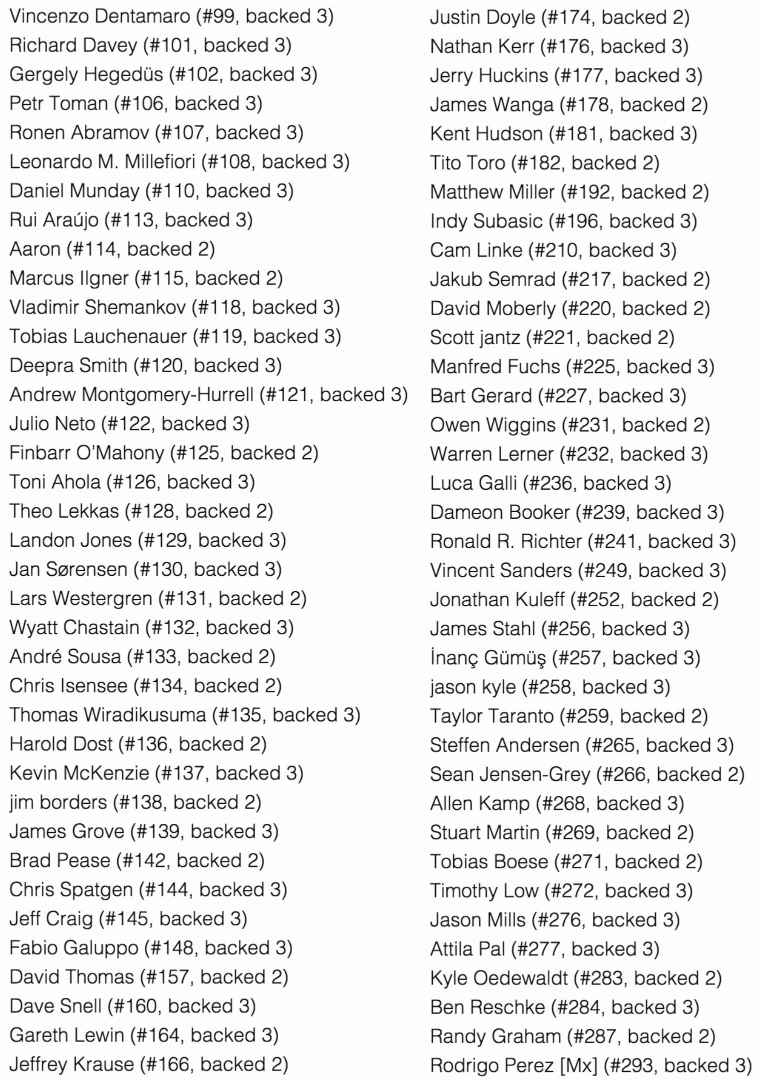
`http://www.heatonresearch.com/ThankYou/`

Figure 4: Repeat Backers 1/4



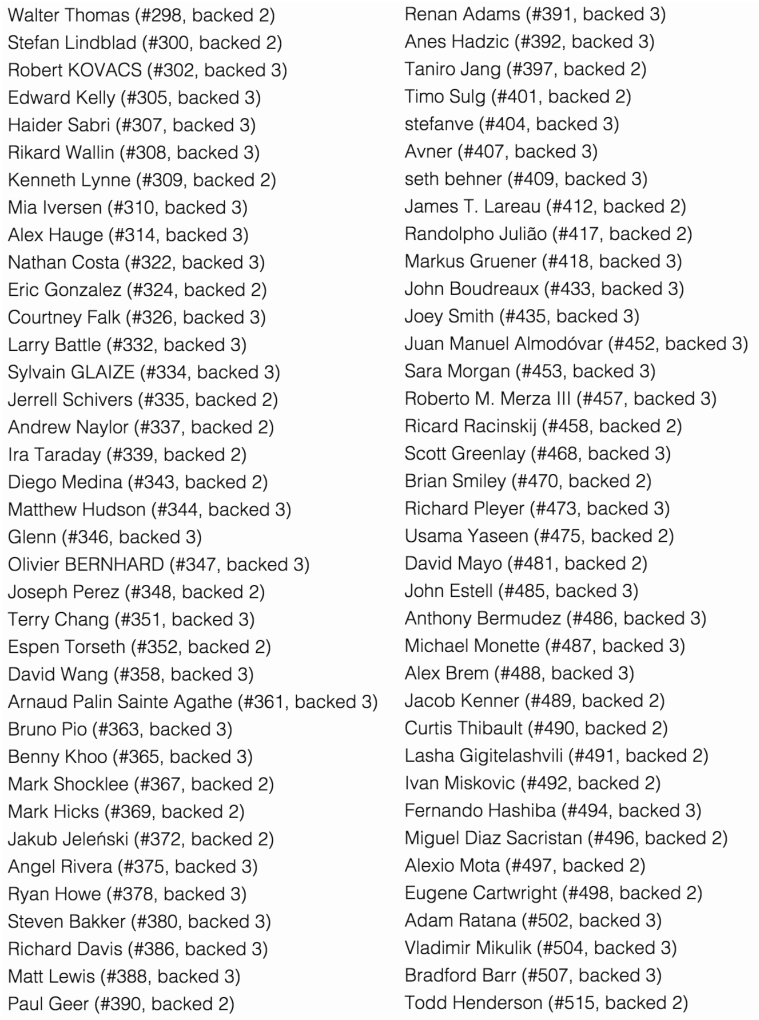
Tracy Heaton (#1, backed 3)
Travis Thaxton (#2, backed 3)
Tim Gallen (#3, backed 3)
Mark Scherschel (#4, backed 2)
Jérôme Landgräfe (#5, backed 3)
Dan Brookes (#7, backed 2)
Mark Chenoweth (#8, backed 3)
Robert Jackson (#9, backed 3)
Prateek Tandon (#10, backed 2)
Mirza Pašić (#11, backed 2)
Vera Kazakova (#12, backed 3)
Marwan Marwan (#13, backed 3)
Jon Evans (#14, backed 3)
Brian Davidson (#15, backed 3)
Myles Steinhauser (#16, backed 3)
Jason Bulgrin (#18, backed 3)
Damien Lebreuilly (#20, backed 3)
Balaji Sundaresan (#22, backed 3)
Joseph Ours (#23, backed 3)
Michael Pelikan (#24, backed 3)
Eric Magnuson (#27, backed 2)
Stan Yamane (#28, backed 3)
David R (#30, backed 3)
Edward Philippe Choi (#31, backed 3)
Brian Ashenfelter (#32, backed 3)
Daryl McLaurine (#33, backed 3)
Dave Gonzalez (#34, backed 2)
Peter Sandbeck Nielsen (#35, backed 2)
Lee Huber (#36, backed 2)
Paul Koerber (#37, backed 3)
Charles Iliya Krempeaux (#38, backed 2)
Adam Whitcomb (#40, backed 2)
vijaya sekhar chennupati (#41, backed 3)
Ray Ion (#42, backed 3)(None.)
A. Human (#43, backed 3)
Nate Doran (#48, backed 2)
Dewayne Higgs (#52, backed 3)
Ian Brandt (#53, backed 2)
Vince Bellows (#55, backed 3)
Rene Incer (#56, backed 3)
Derek Rodger (#57, backed 2)
Jason Penley (#58, backed 3)
Andrew B. Bartels (#59, backed 3)
P. Tullmann (#60, backed 3)
Aidan Morgan (#61, backed 3)
Randy Ray (#62, backed 3)
Lamson Nguyen (#63, backed 3)
Richie Solomon (#64, backed 2)
Mehmet Acar (#65, backed 3)
Guido Scheffler (#66, backed 3)
Hans de Wolf (#69, backed 3)
Simon Evans (#70, backed 2)
Joël Thieffry (#71, backed 3)
Max Kargl (#72, backed 3)
Christian Ahlers (#73, backed 3)
Michael Altarriba (#74, backed 3)
Samuel Walz (#75, backed 3)
Anupam Basu (#76, backed 3)
Andrius Barkauskas (#77, backed 3)
Christer Vaskinn (#78, backed 3)
Ram Madhavan (#79, backed 3)
Huub de Beer (#84, backed 3)
Akash Manohar (#85, backed 3)
Usman Mahmood (#87, backed 3)
Nicolas Hafner (#88, backed 3)
Frank Bieniek (#90, backed 2)
Peter (#91, backed 3)
Angel Lopez (#92, backed 3)
Pirx Danford (#93, backed 3)
Hananel Hazan (#94, backed 3)
Aleksander Majos (#95, backed 3)
Jürgen Wagenhäuser (#96, backed 2)
Andrew Topperwien (#97, backed 3)
Matthew Humphries (#98, backed 3)

Figure 5: Repeat Backers 2/4



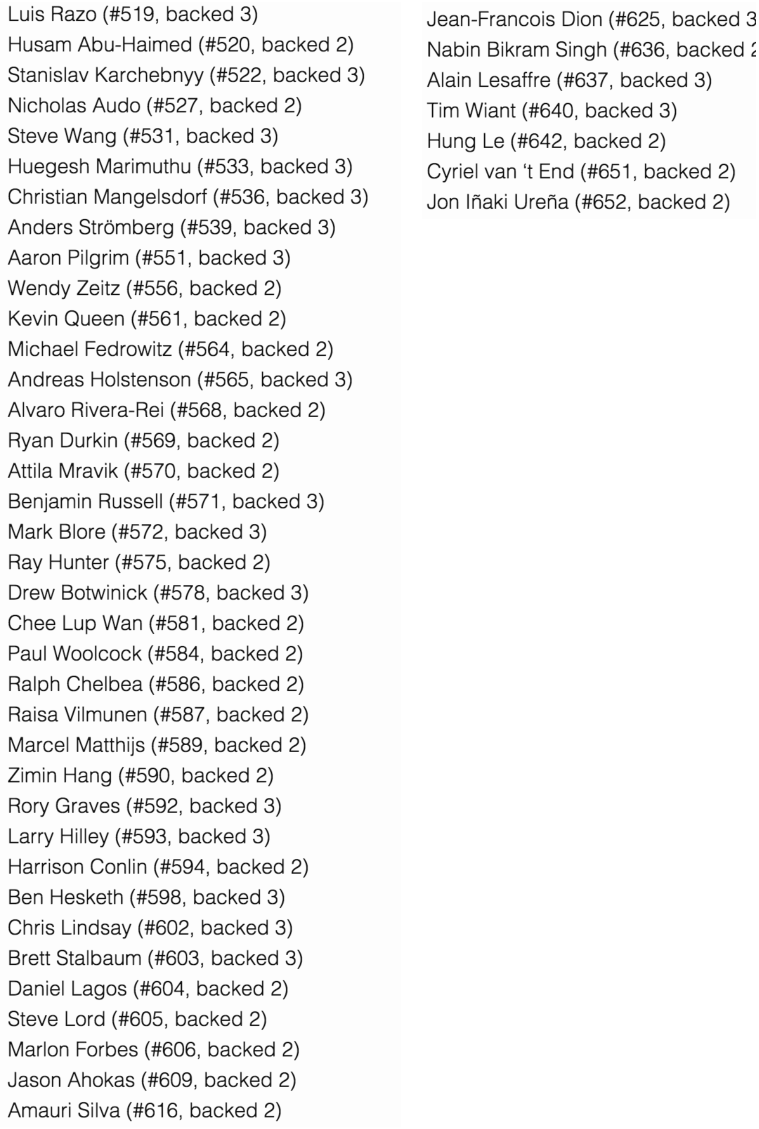
Vincenzo Dentamaro (#99, backed 3)
Richard Davey (#101, backed 3)
Gergely Hegedüs (#102, backed 3)
Petr Toman (#106, backed 3)
Ronen Abramov (#107, backed 3)
Leonardo M. Millefiori (#108, backed 3)
Daniel Munday (#110, backed 3)
Rui Araújo (#113, backed 3)
Aaron (#114, backed 2)
Marcus Ilgner (#115, backed 2)
Vladimir Shemankov (#118, backed 3)
Tobias Lauchenauer (#119, backed 3)
Deepra Smith (#120, backed 3)
Andrew Montgomery-Hurrell (#121, backed 3)
Julio Neto (#122, backed 3)
Finbarr O'Mahony (#125, backed 2)
Toni Ahola (#126, backed 3)
Theo Lekkas (#128, backed 2)
Landon Jones (#129, backed 3)
Jan Sørensen (#130, backed 3)
Lars Westergren (#131, backed 2)
Wyatt Chastain (#132, backed 3)
André Sousa (#133, backed 2)
Chris Isensee (#134, backed 2)
Thomas Wiradikusuma (#135, backed 3)
Harold Dost (#136, backed 2)
Kevin McKenzie (#137, backed 3)
jim borders (#138, backed 2)
James Grove (#139, backed 3)
Brad Pease (#142, backed 2)
Chris Spatgen (#144, backed 3)
Jeff Craig (#145, backed 3)
Fabio Galuppo (#148, backed 3)
David Thomas (#157, backed 2)
Dave Snell (#160, backed 3)
Gareth Lewin (#164, backed 3)
Jeffrey Krause (#166, backed 2)
Justin Doyle (#174, backed 2)
Nathan Kerr (#176, backed 3)
Jerry Huckins (#177, backed 3)
James Wanga (#178, backed 2)
Kent Hudson (#181, backed 3)
Tito Toro (#182, backed 2)
Matthew Miller (#192, backed 2)
Indy Subasic (#196, backed 3)
Cam Linke (#210, backed 3)
Jakub Semrad (#217, backed 2)
David Moberly (#220, backed 2)
Scott jantz (#221, backed 2)
Manfred Fuchs (#225, backed 3)
Bart Gerard (#227, backed 3)
Owen Wiggins (#231, backed 2)
Warren Lerner (#232, backed 3)
Luca Galli (#236, backed 3)
Dameon Booker (#239, backed 3)
Ronald R. Richter (#241, backed 3)
Vincent Sanders (#249, backed 3)
Jonathan Kuleff (#252, backed 2)
James Stahl (#256, backed 3)
İnanç Gümüş (#257, backed 3)
jason kyle (#258, backed 3)
Taylor Taranto (#259, backed 2)
Steffen Andersen (#265, backed 3)
Sean Jensen-Grey (#266, backed 2)
Allen Kamp (#268, backed 3)
Stuart Martin (#269, backed 2)
Tobias Boese (#271, backed 2)
Timothy Low (#272, backed 3)
Jason Mills (#276, backed 3)
Attila Pal (#277, backed 3)
Kyle Oedewaldt (#283, backed 2)
Ben Reschke (#284, backed 3)
Randy Graham (#287, backed 2)
Rodrigo Perez [Mx] (#293, backed 3)

Figure 6: Repeat Backers 3/4



Walter Thomas (#298, backed 2)
Stefan Lindblad (#300, backed 2)
Robert KOVACS (#302, backed 3)
Edward Kelly (#305, backed 3)
Haider Sabri (#307, backed 3)
Rikard Wallin (#308, backed 3)
Kenneth Lynne (#309, backed 2)
Mia Iversen (#310, backed 3)
Alex Hauge (#314, backed 3)
Nathan Costa (#322, backed 3)
Eric Gonzalez (#324, backed 2)
Courtney Falk (#326, backed 3)
Larry Battle (#332, backed 3)
Sylvain GLAIZE (#334, backed 3)
Jerrell Schivers (#335, backed 2)
Andrew Naylor (#337, backed 2)
Ira Taraday (#339, backed 2)
Diego Medina (#343, backed 2)
Matthew Hudson (#344, backed 3)
Glenn (#346, backed 3)
Olivier BERNHARD (#347, backed 3)
Joseph Perez (#348, backed 2)
Terry Chang (#351, backed 3)
Espen Torseth (#352, backed 2)
David Wang (#358, backed 3)
Arnaud Palin Sainte Agathe (#361, backed 3)
Bruno Pio (#363, backed 3)
Benny Khoo (#365, backed 3)
Mark Shocklee (#367, backed 2)
Mark Hicks (#369, backed 2)
Jakub Jeleński (#372, backed 2)
Angel Rivera (#375, backed 3)
Ryan Howe (#378, backed 3)
Steven Bakker (#380, backed 3)
Richard Davis (#386, backed 3)
Matt Lewis (#388, backed 3)
Paul Geer (#390, backed 2)
Renan Adams (#391, backed 3)
Anes Hadzic (#392, backed 3)
Taniro Jang (#397, backed 2)
Timo Sulg (#401, backed 2)
stefanve (#404, backed 3)
Avner (#407, backed 3)
seth behner (#409, backed 3)
James T. Lareau (#412, backed 2)
Randolpho Julião (#417, backed 2)
Markus Gruener (#418, backed 3)
John Boudreaux (#433, backed 3)
Joey Smith (#435, backed 3)
Juan Manuel Almodóvar (#452, backed 3)
Sara Morgan (#453, backed 3)
Roberto M. Merza III (#457, backed 3)
Ricard Racinskij (#458, backed 2)
Scott Greenlay (#468, backed 3)
Brian Smiley (#470, backed 2)
Richard Pleyer (#473, backed 3)
Usama Yaseen (#475, backed 2)
David Mayo (#481, backed 2)
John Estell (#485, backed 3)
Anthony Bermudez (#486, backed 3)
Michael Monette (#487, backed 3)
Alex Brem (#488, backed 3)
Jacob Kenner (#489, backed 2)
Curtis Thibault (#490, backed 2)
Lasha Gigitelashvili (#491, backed 2)
Ivan Miskovic (#492, backed 2)
Fernando Hashiba (#494, backed 3)
Miguel Diaz Sacristan (#496, backed 2)
Alexio Mota (#497, backed 2)
Eugene Cartwright (#498, backed 2)
Adam Ratana (#502, backed 3)
Vladimir Mikulik (#504, backed 3)
Bradford Barr (#507, backed 3)
Todd Henderson (#515, backed 2)

Figure 7: Repeat Backers 4/4



Luis Razo (#519, backed 3)
Husam Abu-Haimed (#520, backed 2)
Stanislav Karchebnyy (#522, backed 3)
Nicholas Audo (#527, backed 2)
Steve Wang (#531, backed 3)
Huegesh Marimuthu (#533, backed 3)
Christian Mangelsdorf (#536, backed 3)
Anders Strömberg (#539, backed 3)
Aaron Pilgrim (#551, backed 3)
Wendy Zeitz (#556, backed 2)
Kevin Queen (#561, backed 2)
Michael Fedrowitz (#564, backed 2)
Andreas Holstenson (#565, backed 3)
Alvaro Rivera-Rei (#568, backed 2)
Ryan Durkin (#569, backed 2)
Attila Mravik (#570, backed 2)
Benjamin Russell (#571, backed 3)
Mark Blore (#572, backed 3)
Ray Hunter (#575, backed 2)
Drew Botwinick (#578, backed 3)
Chee Lup Wan (#581, backed 2)
Paul Woolcock (#584, backed 2)
Ralph Chelbea (#586, backed 2)
Raisa Vilmunen (#587, backed 2)
Marcel Matthijs (#589, backed 2)
Zimin Hang (#590, backed 2)
Rory Graves (#592, backed 3)
Larry Hilley (#593, backed 3)
Harrison Conlin (#594, backed 2)
Ben Hesketh (#598, backed 3)
Chris Lindsay (#602, backed 3)
Brett Stalbaum (#603, backed 3)
Daniel Lagos (#604, backed 2)
Steve Lord (#605, backed 2)
Marlon Forbes (#606, backed 2)
Jason Ahokas (#609, backed 2)
Amauri Silva (#616, backed 2)

Jean-Francois Dion (#625, backed 3
Nabin Bikram Singh (#636, backed 2
Alain Lesaffre (#637, backed 3)
Tim Wiant (#640, backed 3)
Hung Le (#642, backed 2)
Cyriel van ‘t End (#651, backed 2)
Jon Iñaki Ureña (#652, backed 2)

0.4 Background Information

You can read *Artificial Intelligence for Humans* in any order. However, this book does expand on some topics introduced in Volumes 1 and 2. The goal of this section is to help you understand what a neural network is and how to use it. Most people, even non-programmers, have heard of neural networks. Many science fiction stories have plots that are based on ideas related to neural networks. As a result, sci-fi writers have created an influential but somewhat inaccurate view of the neural network.

Most laypeople consider neural networks to be a type of artificial brain. According to this view, neural networks could power robots or carry on intelligent conversations with human beings. However, this notion is a closer definition of artificial intelligence (AI) than of neural networks. Although AI seeks to create truly intelligent machines, the current state of computers is far below this goal. Human intelligence still trumps computer intelligence.

Neural networks are a small part of AI. As they currently exist, neural networks carry out miniscule, highly specific tasks. Unlike the human brain, computer-based neural networks are not general-purpose computational devices. Furthermore, the term neural network can create confusion because the brain is a network of neurons just as AI uses neural networks. To avoid this problem, we must make an important distinction.

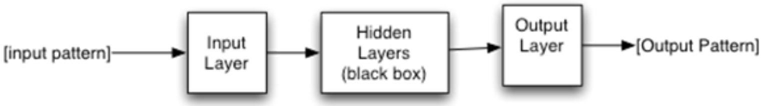
We should really call the human brain a biological neural network (BNN). Most texts do not bother to make the distinction between a BNN and artificial neural networks (ANNs). Our book follows this pattern. When we refer to neural networks, we're dealing with ANNs. We are not talking about BNNs when we use the term neural network.

Biological neural networks and artificial neural networks share some very basic similarities. For instance, biological neural networks have inspired the mathematical constructs of artificial neural networks. Biological plausibility describes various artificial neural network algorithms. This term defines how close an artificial neural network algorithm is to a biological neural network.

As previously mentioned, programmers design neural networks to execute one small task. A full application will likely use neural networks to accomplish certain parts of the application. However, the entire application will not be implemented as a neural network. It may consist of several neural networks of which each has a specific task.

Pattern recognition is a task that neural networks can easily accomplish. For this task, you can communicate a pattern to a neural network, and it communicates a pattern back to you. At the highest level, a typical neural network can perform only this function. Although some network architectures might achieve more, the vast majority of neural networks work this way. Figure 8 illustrates a neural network at this level:

Figure 8: A Typical Neural Network



As you can see, the above neural network accepts a pattern and returns a pattern. Neural networks operate synchronously and will only output when it has input. This behavior is not like that of a human brain, which does not operate synchronously. The human brain responds to input, but it will produce output anytime it feels like it!

0.4.1 Neural Network Structure

Neural networks consist of layers of similar neurons. Most have at least an input layer and an output layer. The program presents the input pattern to the input layer. Then the output pattern is returned from the output layer. What happens between the input and an output layer is a black box. By black box, we mean that you do not know exactly why a neural network outputs what it does. At this point, we are not yet concerned with the internal structure of the neural network, or the black box. Many different architectures define the interaction between the input and output layer. Later, we will examine some of these architectures.

The input and output patterns are both arrays of floating-point numbers. Consider the arrays in the following ways:

```
Neural Network Input: [ −0.245, .283, 0.0 ]
Neural Network Output: [ 0.782, 0.543 ]
```

The above neural network has three neurons in the input layer, and two neurons are in the output layer. The number of neurons in the input and output layers does not change, even if you restructure the interior of the neural network.

To utilize the neural network, you must express your problem so that the input of the problem is an array of floating-point numbers. Likewise, the solution to the problem must be an array of floating-point numbers. Ultimately, this expression is the only process that that neural networks can perform. In other words, they take one array and transform it into a second. Neural networks do not loop, call subroutines, or perform any of the other tasks you might think of with traditional programming. Neural networks simply recognize patterns.

You might think of a neural network as a hash table in traditional programming that maps keys to values. It acts somewhat like a dictionary. You can consider the following as a type of hash table:

- “hear” -> “to perceive or apprehend by the ear”
- “run” -> “to go faster than a walk”
- “write” -> “to form (as characters or symbols) on a surface with an instrument (as a pen)”

This table creates a mapping between words and provides their definitions. Programming languages usually call this a hash map or a dictionary. This hash table uses a key of type string to reference another value that is also of the same type string. If you’ve not worked with hash tables before, they simply map one value to another, and they are a form of indexing. In other words, the dictionary returns a value when you provide it with a key. Most neural networks function in this manner. One neural network called bidirectional associative memory (BAM) allows you to provide the value and receive the key.

Programming hash tables contain keys and values. Think of the pattern sent to the input layer of the neural network as the key to the hash table. Likewise, think of the value returned from the hash table as the pattern that is returned from the output layer of the neural network. Although the comparison between a hash table and a neural network is appropriate to help you understand the concept, you need to realize that the neural network is much more than a hash table.

What would happen with the previous hash table if you were to provide a word that is not a key in the map? To answer the question, we will pass in the key of “wrote.” For this example, a hash table would return **null**. It would indicate in some way that it could not find the specified key. However, neural networks do not return **null**; they find the closest match. Not only do they find the closest match, they will modify the output to estimate the missing value. So if you passed in “wrote” to the above neural network, you would likely receive what you would have expected for “write.” You would likely get the output from one of the other keys because not enough data exist for the neural network to modify the response. The limited number of samples (in this case, there are three) causes this result.

The above mapping raises an important point about neural networks. As previously stated, neural networks accept an array of floating-point numbers and return another array. This behavior provokes the question about how to put string, or textual, values into the above neural network. Although a solution exists, dealing with numeric data rather than strings is much easier for the neural network.

In fact, this question reveals one of the most difficult aspects of neural network programming. How do you translate your problem into a fixed-length array of floating-point numbers? In the examples that follow, you will see the complexity of neural networks.

0.4.2 A Simple Example

In computer programming, it is customary to provide a “Hello World” application that simply displays the text “Hello World.” If you have previously read about neural networks, you have no doubt seen examples with the exclusive or (XOR) operator, which is one of the “Hello World” applications of neural network programming. Later in this section, we will describe more complex scenarios than XOR, but it is a great introduction. We shall begin by looking at the XOR operator as though it were a hash table. If you are not familiar with the XOR operator, it works similarly to the AND / OR operators. For an AND to be true, both sides must be true. For an OR to be true, either side must be true. For an XOR to be true, both of the sides must be different from each other. The following truth table represents an XOR:

```
False XOR False = False
True XOR False = True
False XOR True = True
True XOR True = False
```

To continue the hash table example, you would represent the above truth table as follows:

```
[ 0.0 , 0.0 ] -> [ 0.0 ]
[ 1.0 , 0.0 ] -> [ 1.0 ]
[ 0.0 , 1.0 ] -> [ 1.0 ]
[ 1.0 , 1.0 ] -> [ 0.0 ]
```

These mappings show input and the ideal expected output for the neural network.

0.4.3 Training: Supervised and Unsupervised

When you specify the ideal output, you are using supervised training. If you did not provide ideal outputs, you would be using unsupervised training. Supervised training teaches the neural network to produce the ideal output. Unsupervised training usually teaches the neural network to place the input data into a number of groups defined by the output neuron count.

Both supervised and unsupervised training are iterative processes. For supervised training, each training iteration calculates how close the actual output is to the ideal output and expresses this closeness as an error percent. Each iteration modifies the internal weight matrices of the neural network to decrease the error rate to an acceptably low level.

Unsupervised training is also an iterative process. However, calculating the error is not as easy. Because you have no expected output, you cannot measure how far the unsupervised neural network is from your ideal output. Thus, you have no ideal output. As a result, you will just iterate for a fixed number of iterations and try to use the network. If the neural network needs more training, the program provides it.

Another important aspect to the above training data is that you can take it in any order. The result of two zeros, with XOR applied (0 XOR 0) is going to be 0, regardless of which case that you used. This characteristic is not true of all neural networks. For the XOR operator, we would probably use a type of neural network called a feedforward neural network in which the order of the training set does not matter. Later in this book, we will examine recurrent neural networks that do consider the order of the training data. Order is an essential component of a simple recurrent neural network.

Previously, you saw that the simple XOR operator utilized training data. Now we will analyze a situation with more complex training data.

0.4.4 Miles per Gallon

In general, neural network problems involve a set of data that you use to predict values for later sets of data. These later sets of data result after you've already trained your neural network. The power of a neural network is to predict outcomes for entirely new data sets based on knowledge learned from past data sets. Consider a car database that contains the following fields:

- Car Weight
- Engine Displacement
- Cylinder Count
- Horse Power
- Hybrid or Gasoline
- Miles per Gallon

Although we are oversimplifying the data, this example demonstrates how to format data. Assuming you have collected some data for these fields, you should be able to construct a neural network that can predict one field value, based on the other field values. For this example, we will try to predict miles per gallon.

As previously demonstrated, we will need to define this problem in terms of an input array of floating-point numbers mapped to an output array of floating-point numbers. However, the problem has one additional requirement. The numeric range on each of these array elements should be between 0 and 1 or -1 and 1. This range is called normalization. It takes real-world data and turns it into a form that the neural network can process.

First, we determine how to normalize the above data. Consider the neural network format. We have six total fields. We want to use five of these fields to predict the sixth. Consequently, the neural network would have five input neurons and one output neuron.

Your network would resemble the following:

- Input Neuron 1: Car Weight
- Input Neuron 2: Engine Displacement
- Input Neuron 3: Cylinder Count
- Input Neuron 4: Horse Power
- Input Neuron 5: Hybrid or Gasoline
- Output Neuron 1: Miles per Gallon

We also need to normalize the data. To accomplish this normalization, we must think of reasonable ranges for each of these values. We will then transform input data into a number between 0 and 1 that represents an actual value's position within that range. Consider this example with the reasonable ranges for the following values:

- Car Weight: 100-5000 lbs.
- Engine Displacement: 0.1 to 10 liters
- Cylinder Count: 2-12

- Horse Power: 1-1000
- Hybrid or Gasoline: true or false
- Miles per Gallon: 1-500

Given today's cars, these ranges may be on the large end. However, this characteristic will allow minimal restructuring to the neural network in the future. We also want to avoid having too much data at the extreme ends of the range.

To illustrate this range, we will consider the problem of normalizing a weight of 2,000 pounds. This weight is 1,900 into the range (2000 - 100). The size of the range is 4,900 pounds (5000-100). The percent of the range size is 0.38 (1,900 / 4,900). Therefore, we would feed the value of 0.38 to the input neuron in order to represent this value. This process satisfies the range requirement of 0 to 1 for an input neuron.

The hybrid or regular value is a true/false. To represent this value, we will use 1 for hybrid and 0 for regular. We simply normalize a true/false into two values.

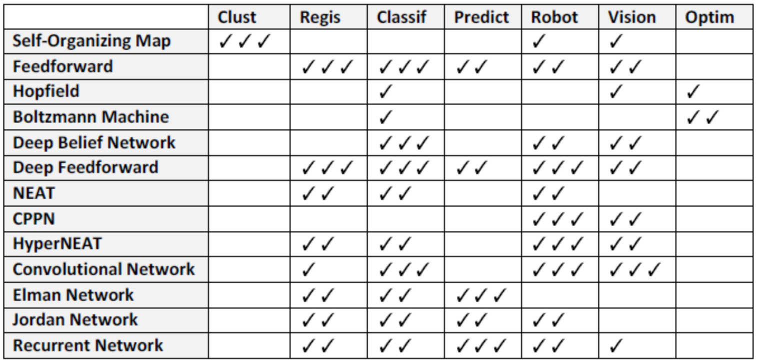
Now that you've seen some of the uses for neural networks, it is time to determine how to select the appropriate neural network for your specific problem. In the succeeding section, we provide a roadmap for the various neural networks that are available.

0.5 A Neural Network Roadmap

This volume contains a wide array of neural network types. We will present these neural networks along with examples that will showcase each neural network in a specific problem domain. Not all neural networks are designed to tackle every problem domain. As a neural network programmer, you need to know which neural network to use for a specific problem.

This section provides a high-level roadmap to the rest of the book that will guide your reading to areas of the book that align with your interests. Figure 9 shows a grid of the neural network types in this volume and their applicable problem domains:

Figure 9: Neural Network Types & Problem Domains



	Clust	Regis	Classif	Predict	Robot	Vision	Optim
Self-Organizing Map	✓✓✓				✓	✓	
Feedforward		✓✓✓	✓✓✓	✓✓	✓✓	✓✓	
Hopfield			✓			✓	✓
Boltzmann Machine			✓				✓✓
Deep Belief Network			✓✓✓		✓✓	✓✓	
Deep Feedforward		✓✓✓	✓✓✓	✓✓	✓✓✓	✓✓	
NEAT		✓✓	✓✓		✓✓		
CPPN					✓✓✓	✓✓	
HyperNEAT		✓✓	✓✓		✓✓✓	✓✓	
Convolutional Network		✓	✓✓✓		✓✓✓	✓✓✓	
Elman Network		✓✓	✓✓	✓✓✓			
Jordan Network		✓✓	✓✓	✓✓	✓✓		
Recurrent Network		✓✓	✓✓	✓✓✓	✓✓	✓	

The problem domains listed above are the following:

- **Clust** - Unsupervised clustering problems
- **Regis** - Regression problems, the network must output a number based on input.
- **Classif** - Classification problems, the network must classify data points into predefined classes.
- **Predict** - The network must predict events in time, such as signals for finance applications.
- **Robot** - Robotics, using sensors and motor control
- **Vision** - Computer Vision (CV) problems require the computer to understand images.
- **Optim** - Optimization problems require that the network find the best ordering or set of values to achieve an objective.

The number of checkmarks gives the applicability of each of the neural network types to that particular problem. If there are no checks, you cannot apply that network type to that problem domain.

All neural networks share some common characteristics. Neurons, weights, activation functions, and layers are the building blocks of neural networks. In the first chapter of this book, we will introduce these concepts and present the basic characteristics that most neural networks share.

0.6 Data Sets Used in this Book

This book contains several data sets that allow us to show application of the neural networks to real data. We chose several data sets in order to cover topics such as regression, classification, time-series, and computer vision.

0.6.1 MNIST Handwritten Digits

Several examples use the MNIST handwritten digits data set. The MNIST database (Mixed National Institute of Standards and Technology database) is a large database of handwritten digits that programmers use for training various image processing systems. This classic data set is often presented in conjunction with neural networks. This data set is essentially the "Hello World" program of neural networks. You can obtain it from the following URL:

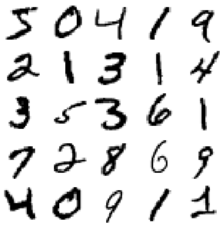
`http://yann.lecun.com/exdb/mnist/`

The data set in the above listing is stored in a special binary format. You can also find this format at the above URL. The example programs provided for this chapter are capable of reading this format.

This data set contains many handwritten digits. It also includes a training set of 60,000 examples and a test set of 10,000 examples. We provide labels on both sets to indicate what each digit is supposed to be. MNIST is a highly studied data set that programmers frequently use as a benchmark for new machine learning algorithms and techniques. Furthermore, researchers have published many scientific papers about their attempts to achieve the lowest error rate. In one study, the researcher managed to achieve an error rate on the MNIST database of 0.23 percent while using a hierarchical system of convolutional neural networks (Schmidhuber, 2012).

We show a small sampling of the data set in Figure 10:

Figure 10: MNIST Digits



We can use this data set for classification neural networks. The networks learn to look at an image and classify it into the appropriate place among the ten digits. Even though this data set is an image-based neural network, you can think of it as a traditional data set. These images are 28 pixels by 28 pixels, resulting in a total of 784 pixels. Despite the impressive images, we begin the book by using regular neural networks that treat the images as a 784-input-neuron neural network. You would use exactly the same type of neural network to handle any classification problem that has a large number of inputs. Such problems are high dimensional. Later in the book, we will see how to use neural networks that were specifically designed for image recognition. These neural networks will perform considerably better on the MNIST digits than the more traditional neural networks.

The MNIST data set is stored in a propriety binary format that is described at the above URL. We provide a decoder in the book's examples.

0.6.2 Iris Data Set

Because AI frequently uses the iris data set (Fisher, 1936), you will see it several times in this book. Sir Ronald Fisher (1936) collected these data as an example of discriminant analysis. This data set has become very popular in machine learning even today. The following URL contains the iris data set:

`https://archive.ics.uci.edu/ml/datasets/Iris`

The iris data set contains measurements and species information for 150 iris flowers, and the data are essentially represented as a spreadsheet with the following columns or features:

- Sepal length
- Sepal width
- Petal length
- Petal width
- Iris species

Petals refer to the innermost petals of the iris, and sepal refers to the outermost petals of the iris flower. Even though the data set seems to have a vector of length 5, the species feature must be handled differently than the other four. In other words, vectors typically contain only numbers. So, the first four features are inherently numerical. The species feature is not.

One of the primary applications of this data set is to create a program that will act as a classifier. That is, it will consider the flower's features as inputs (sepal length, petal width, etc.) and ultimately determine the species. This classification would be trivial for a complete and known data set, but our goal is to see whether the model can correctly identify the species using data from unknown irises.

Only simple numeric encoding translates the iris species to a single dimension. We must use additional dimensional encodings, such as one-of-n or equilateral, so that the species encodings are equidistant from each other. If we are classifying irises, we do not want our encoding process to create any biases.

Thinking of the iris features as dimensions in a higher-dimensional space makes a great deal of sense. Consider the individual samples (the rows in the iris data set) as points in this search space. Points closer together likely share similarities. Let's take a look at these similarities by studying the following three rows from the iris data set:

```
5.1, 3.5, 1.4, 0.2, Iris-setosa
7.0, 3.2, 4.7, 1.4, Iris-versicolour
6.3, 3.3, 6.0, 2.5, Iris-virginica
```

The first line has 5.1 as the sepal length, 3.5 as the sepal width, 1.4 as the petal length, and 0.2 as the petal width. If we use one-of-n encoding to the range 0 to 1, the above three rows would encode to the following three vectors:

```
[ 5.1, 3.5, 1.4, 0.2, 1, 0, 0 ]
[ 7.0, 3.2, 4.7, 1.4, 0, 1, 0 ]
[ 6.3, 3.3, 6.0, 2.5, 0, 0, 1 ]
```

Chapter 4, "Feedforward Neural Networks," will cover one-of-n encoding.

0.6.3 Auto MPG Data Set

The auto miles per gallon (MPG) data set is commonly used for regression problems. The data set contains attributes of several cars. Using these attributes, we can train neural networks to predict the fuel efficiency of the car. The UCI Machine Learning Repository provides this data set, and you can download it from the following URL:

`https://archive.ics.uci.edu/ml/datasets/Auto+MPG`

We took these data from the StatLib library, which is maintained at Carnegie Mellon University. In the exposition for the American Statistical Association, programmers used the data in 1983, and no values are missing. Quinlan (1993), the author of the study, used this data set to describe fuel consumption. "The data concern city-cycle fuel consumption in miles per gallon, to be projected in terms of three multi-valued discrete and five continuous attributes" (Quinlan, 1993).

The data set contains the following attributes:

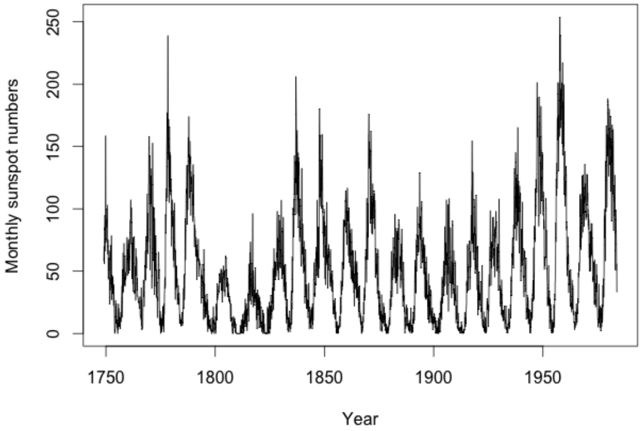
```
1. mpg: continuous
2. cylinders: multi-valued discrete
3. displacement: continuous
4. horsepower: continuous
5. weight: continuous
6. acceleration: continuous
7. model year: multi-valued discrete
8. origin: multi-valued discrete
9. car name: string (unique for each instance)
```

0.6.4 Sunspots Data Set

Sunspots are temporary phenomena on the surface of the sun that appear visibly as dark spots compared to surrounding regions. Intense magnetic activity causes sunspots. Although they occur at temperatures of roughly 3,000-4,500 K (2,727-4,227 °C), the contrast with the surrounding material at about 5,780 K leaves them clearly visible as dark spots. Sunspots appear and disappear with regularity, making them a good data set for time series prediction.

Figure 11 shows sunspot activity over time:

Figure 11: Sunspots Activity



The sunspot data file contains information similar to the following:

```
YEAR MON   SSN    DEV
1749   1   58.0   24.1
1749   2   62.6   25.1
1749   3   70.0   26.6
1749   4   55.7   23.6
1749   5   85.0   29.4
1749   6   83.5   29.2
1749   7   94.8   31.1
1749   8   66.3   25.9
1749   9   75.9   27.7
```

The above data provide the year, month, sunspot count, and standard deviation of sunspots observed. Many world organizations track sunspots. The following URL contains a table of sunspot readings:

`http://solarscience.msfc.nasa.gov/greenwch/spot_num.txt`

0.6.5 XOR Operator

The exclusive or (XOR) operator is a Boolean operator. Programmers frequently use the truth table for the XOR as an ultra-simple sort of ”Hello World” training set for machine learning. We refer to the table as the XOR data set. This operator is related to the XOR parity operator, which accepts three inputs and has the following truth table:

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

We utilize the XOR operator for cases in which we would like to train or evaluate the neural network by hand.

0.6.6 Kaggle Otto Group Challenge

In this book, we will also utilize the Kaggle Otto Group Challenge data set. Kaggle is a platform that fosters competition among data scientists on new data sets. We use this data set to classify products into several groups based on unknown attributes. Additionally, we will employ a deep neural network to tackle this problem. We will also discuss advanced ensemble techniques in this chapter that you can use to compete in Kaggle. We will describe this data set in greater detail in Chapter 16.

We will begin this book with an overview of features that are common to most neural networks. These features include neurons, layers, activation functions, and connections. For the remainder of the book, we will expand on these topics as we introduce more neural network architectures.

Chapter 1

Neural Network Basics

- Neurons and Layers
- Neuron Types
- Activation Functions
- Logic Gates

This book is about neural networks and how to train, query, structure, and interpret them. We present many neural network architectures as well as the plethora of algorithms that can train these neural networks. Training is the process in which a neural network is adapted to make predictions from data. In this chapter, we will introduce the basic concepts that are most relevant to the neural network types featured in the book.

Deep learning, a relatively new set of training techniques for multilayered neural networks, is also a primary topic. It encompasses several algorithms that can train complex types of neural networks. With the development of deep learning, we now have effective methods to train neural networks with many layers.

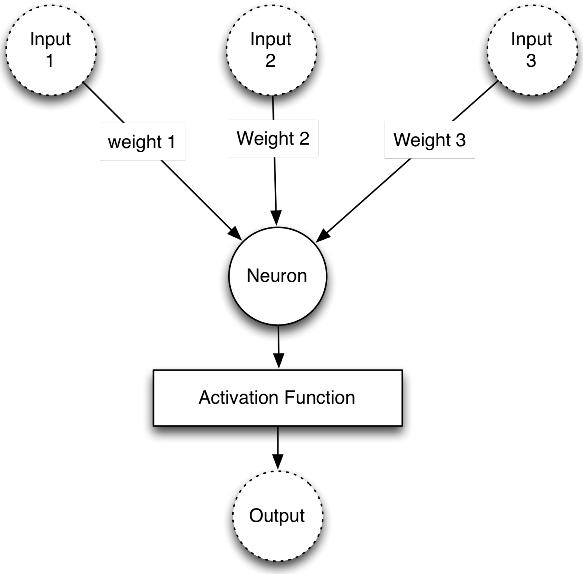
This chapter will include a discussion of the commonalities among the different neural networks. Additionally, you will learn how neurons form weighted connections, how these neurons create layers, and how activation functions affect the output of a layer. We begin with neurons and layers.

1.1 Neurons and Layers

Most neural network structures use some type of neuron. Many different kinds of neural networks exist, and programmers introduce experimental neural network structures all the time. Consequently, it is not possible to cover every neural network architecture. However, there are some commonalities among neural network implementations. An algorithm that is called a neural network will typically be composed of individual, interconnected units even though these units may or may not be called neurons. In fact, the name for a neural network processing unit varies among the literature sources. It could be called a node, neuron, or unit.

Figure 1.1 shows the abstract structure of a single artificial neuron:

Figure 1.1: An Artificial Neuron



The artificial neuron receives input from one or more sources that may be other neurons or data fed into the network from a computer program. This input is usually floating-point or binary. Often binary input is encoded to floating-point by representing true or false as 1 or 0. Sometimes the program also depicts the binary input as using a bipolar system with true as 1 and false as -1.

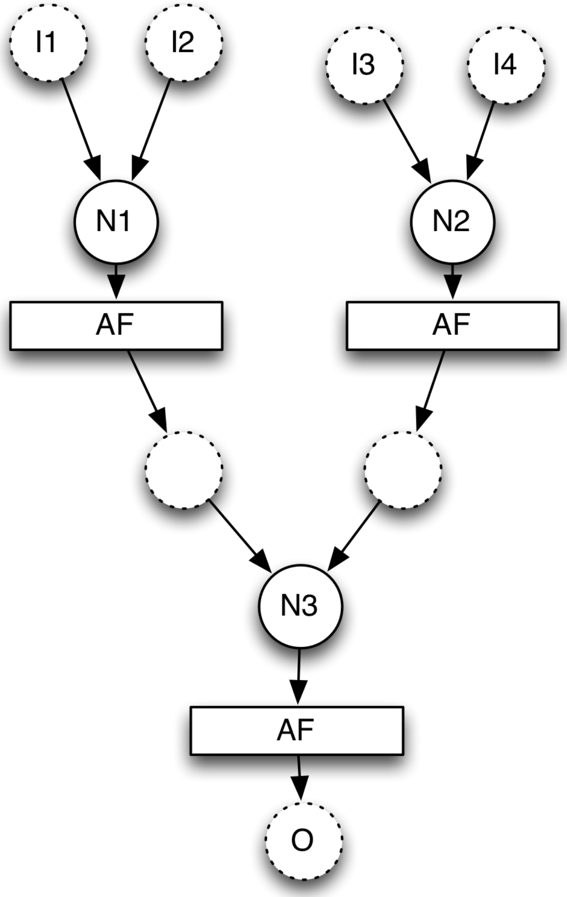
An artificial neuron multiplies each of these inputs by a weight. Then it adds these multiplications and passes this sum to an activation function. Some neural networks do not use an activation function. Equation 1.1 summarizes the calculated output of a neuron:

$$f(x_i, w_i) = \phi(\sum_i (w_i \cdot x_i)) \tag{1.1}$$

In the above equation, the variables x and w represent the input and weights of the neuron. The variable i corresponds to the number of weights and inputs. You must always have the same number of weights as inputs. Each weight is multiplied by its respective input, and the products of these multiplications are fed into an activation function that is denoted by the Greek letter ϕ (phi*)*. This process results in a single output from the neuron.

Figure 1.1 shows the structure with just one building block. You can chain together many artificial neurons to build an artificial neural network (ANN). Think of the artificial neurons as building blocks for which the input and output circles are the connectors. Figure 1.2 shows an artificial neural network composed of three neurons:

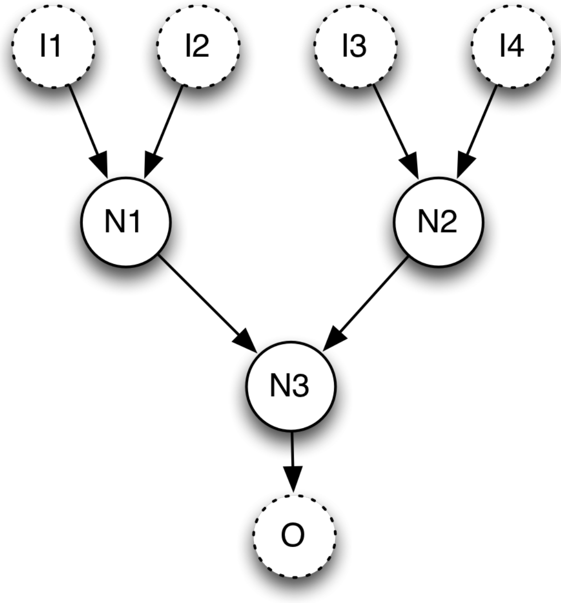
Figure 1.2: Simple Artificial Neural Network (ANN)



The above diagram shows three interconnected neurons. This representation is essentially Figure 1.1, minus a few inputs, repeated three times and then connected. It also has a total of four inputs and a single output. The output of neurons N1 and N2 feed N3 to produce the output O. To calculate the output for Figure 1.2, we perform Equation 1.1 three times. The first two times calculate N1 and N2, and the third calculation uses the output of N1 and N2 to calculate N3.

Neural network diagrams do not typically show the level of detail seen in Figure 1.2. To simplify the diagram, we can omit the activation functions and intermediate outputs, and this process results in Figure 1.3:

Figure 1.3: Simplified View of ANN

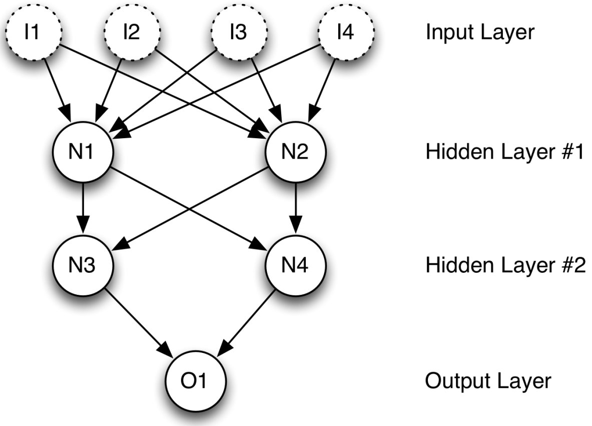


Looking at Figure 1.3, you can see two additional components of neural networks. First, consider the inputs and outputs that are shown as abstract dotted line circles. The input and output could be parts of a larger neural network. However, the input and output are often a special type of neuron that accepts data from the computer program using the neural network, and the output neurons return a result back to the program. This type of neuron is called an input neuron. We will discuss these neurons in the next section.

Figure 1.3 also shows the neurons arranged in layers. The input neurons are the first layer, the N1 and N2 neurons create the second layer, the third layer contains N3, and the fourth layer has O. While most neural networks arrange neurons into layers, this is not always the case. Stanley (2002) introduced a neural network architecture called Neuroevolution of Augmenting Topologies (NEAT). NEAT neural networks can have a very jumbled, non-layered architecture.

The neurons that form a layer share several characteristics. First, every neuron in a layer has the same activation function. However, the layers themselves might have different activation functions. Second, layers are fully connected to the next layer. In other words, every neuron in one layer has a connection to neurons in the previous layer. Figure 1.3 is not fully connected. Several layers are missing connections. For example, I1 and N2 do not connect. Figure 1.4 is a new version of Figure 1.3 that is fully connected and has an additional layer.

Figure 1.4: Fully Connected Network



In Figure 1.4, you see a fully connected, multilayered neural network. Networks, such as this one, will always have an input and output layer. The number of hidden layers determines the name of the network architecture. The network in Figure 1.4 is a two-hidden-layer network. Most networks will have between zero and two hidden layers. Unless you have implemented deep learning strategies, networks with more than two hidden layers are rare.

You might also notice that the arrows always point downward or forward from the input to the output. This type of neural network is called a feedforward neural network. Later in this book, we will see recurrent neural networks that form inverted loops among the neurons.

1.2 Types of Neurons

In the last section, we briefly introduced the idea that different types of neurons exist. Now we will explain all the neuron types described in the book. Not every neural network will use every type of neuron. It is also possible for a single neuron to fill the role of several different neuron types.

1.2.1 Input and Output Neurons

Nearly every neural network has input and output neurons. The input neurons accept data from the program for the network. The output neuron provides processed data from the network back to the program. These input and output neurons will be grouped by the program into separate layers called the input and output layer. However, for some network structures, the neurons can act as both input and output. The Hopfield neural network, which we will discuss in Chapter 3, "Hopfield & Boltzmann Machines," is an example of a neural network architecture in which neurons are both input and output.

The program normally represents the input to a neural network as an array or vector. The number of elements contained in the vector must be equal to the number of input neurons. For example, a neural network with three input neurons might accept the following input vector:

```
[0.5, 0.75, 0.2]
```

Neural networks typically accept floating-point vectors as their input. Likewise, neural networks will output a vector with length equal to the number of output neurons. The output will often be a single value from a single output neuron. To be consistent, we will represent the output of a single output neuron network as a single-element vector.

Notice that input neurons do not have activation functions. As demonstrated by Figure 1.1, input neurons are little more than placeholders. The input is simply weighted and summed. Furthermore, the size of the input and output vectors for the neural network will be the same if the neural network has neurons that are both input and output.

1.2.2 Hidden Neurons

Hidden neurons have two important characteristics. First, hidden neurons only receive input from other neurons, such as input or other hidden neurons. Second, hidden neurons only output to other neurons, such as output or other hidden neurons. Hidden neurons help the neural network understand the input, and they form the output. However, they are not directly connected to the incoming data or to the eventual output. Hidden neurons are often grouped into fully connected hidden layers.

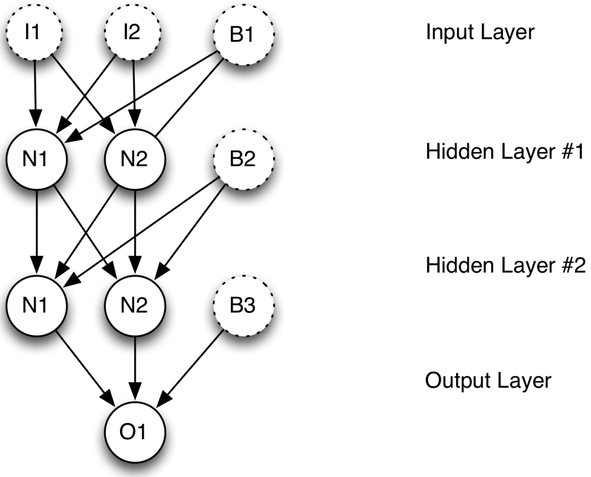
A common question for programmers concerns the number of hidden neurons in a network. Since the answer to this question is complex, more than one section of the book will include a relevant discussion of the number of hidden neurons. Prior to deep learning, it was generally suggested that anything more than a single-hidden layer is excessive (Hornik, 1991). Researchers have proven that a single-hidden-layer neural network can function as a universal approximator. In other words, this network should be able to learn to produce (or approximate) any output from any input as long as it has enough hidden neurons in a single layer.

Another reason why researchers used to scoff at the idea of additional hidden layers is that these layers would impede the training of the neural network. Training refers to the process that determines good weight values. Before researchers introduced deep learning techniques, we simply did not have an efficient way to train a deep network, which are neural networks with a large number of hidden layers. Although a single-hidden-layer neural network can theoretically learn anything, deep learning facilitates a more complex representation of patterns in the data.

1.2.3 Bias Neurons

Programmers add bias neurons to neural networks to help them learn patterns. Bias neurons function like an input neuron that always produces the value of 1. Because the bias neurons have a constant output of 1, they are not connected to the previous layer. The value of 1, which is called the bias activation, can be set to values other than 1. However, 1 is the most common bias activation. Not all neural networks have bias neurons. Figure 1.5 shows a single-hidden-layer neural network with bias neurons:

Figure 1.5: Network with Bias Neurons



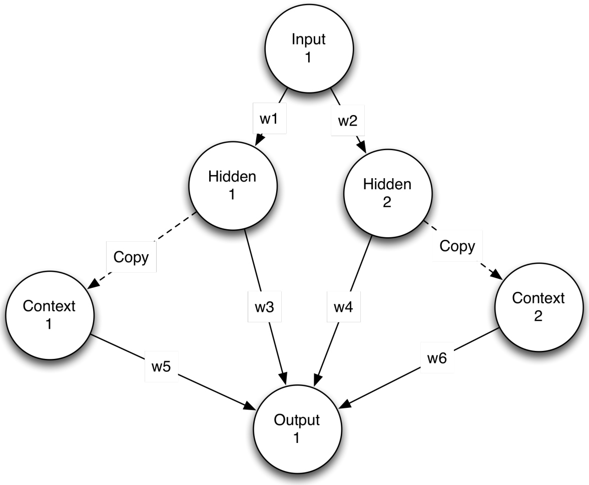
The above network contains three bias neurons. Every level, except for the output layer, contains a single bias neuron. Bias neurons allow the output of an activation function to be shifted. We will see exactly how this shifting occurs later in the chapter when activation functions are discussed.

1.2.4 Context Neurons

Context neurons are used in recurrent neural networks. This type of neuron allows the neural network to maintain state. As a result, a given input may not always produce exactly the same output. This inconsistency is similar to the workings of biological brains. Consider how context factors in your response when you hear a loud horn. If you hear the noise while you are crossing the street, you might startle, stop walking, and look in the direction of the horn. If you hear the horn while you are watching an action adventure film in a movie theatre, you don't respond in the same way. Therefore, prior inputs give you the context for processing the audio input of a horn.

Time series is one application of context neurons. You might need to train a neural network to learn input signals to perform speech recognition or to predict trends in security prices. Context neurons are one way for neural networks to deal with time series data. Figure 1.6 shows how context neurons might be arranged in a neural network:

Figure 1.6: Context Neurons



This neural network has a single input and output neuron. Between the input and output layers are two hidden neurons and two context neurons. Other than the two context neurons, this network is the same as previous networks in the chapter.

Each context neuron holds a value that starts at 0 and always receives a copy of either hidden 1 or hidden 2 from the previous use of the network. The two dashed lines in Figure 1.6 mean that the context neuron is a direct copy with no other weighting. The other lines indicate that the output is weighted by one of the six weight values listed above. Equation 1.1 still calculates the output in the same way. The value of the output neuron would be the sum of all four inputs, multiplied by their weights, and applied to the activation function.

A type of neural network called a simple recurrent neural network (SRN) uses context neurons. Jordan and Elman networks are the two most common types of SRN. Figure 1.6 shows an Elman SRN. Chapter 13, “Time Series and Recurrent Networks,” includes a discussion of both types of SRN.

1.2.5 Other Neuron Types

The individual units that comprise a neural network are not always called neurons. Researchers will sometimes refer to these neurons as nodes, units or summations. In later chapters of the book, we will explore deep learning that utilizes Boltzmann machines to fill the role of neurons. Regardless of the type of unit, neural networks are almost always constructed of weighted connections between these units.

1.3 Activation Functions

In neural network programming, activation or transfer functions establish bounds for the output of neurons. Neural networks can use many different activation functions. We will discuss the most common activation functions in this section.

Choosing an activation function for your neural network is an important consideration because it can affect how you must format input data. In this chapter, we will guide you on the selection of an activation function. Chapter 14, “Architecting Neural Networks,” will also contain additional details on the selection process.

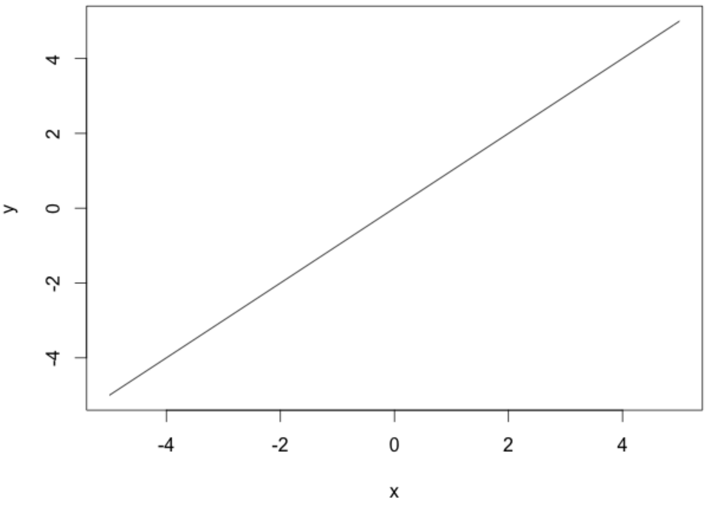
1.3.1 Linear Activation Function

The most basic activation function is the linear function because it does not change the neuron output at all. Equation 1.2 shows how the program typically implements a linear activation function:

$$\phi(x) = x \tag{1.2}$$

As you can observe, this activation function simply returns the value that the neuron inputs passed to it. Figure 1.7 shows the graph for a linear activation function:

Figure 1.7: Linear Activation Function



Regression neural networks, those that learn to provide numeric values, will usually use a linear activation function on their output layer. Classification neural networks, those that determine an appropriate class for their input, will usually utilize a softmax activation function for their output layer.

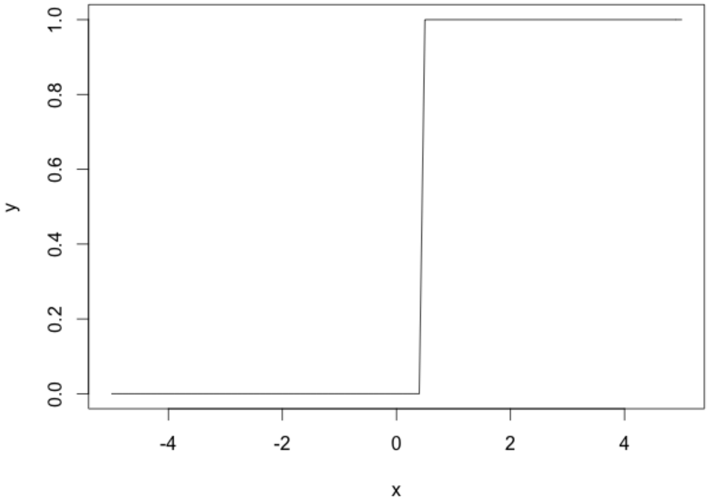
1.3.2 Step Activation Function

The step or threshold activation function is another simple activation function. Neural networks were originally called perceptrons. McCulloch & Pitts (1943) introduced the original perceptron and used a step activation function like Equation 1.3:

$$\phi(x) = \begin{cases} 1, \text{if } x \geq 0.5. \\ 0, \text{otherwise.} \end{cases} \tag{1.3}$$

Equation 1.3 outputs a value of 1.0 for incoming values of 0.5 or higher and 0 for all other values. Step functions are often called threshold functions because they only return 1 (true) for values that are above the specified threshold, as seen in Figure 1.8:

Figure 1.8: Step Activation Function



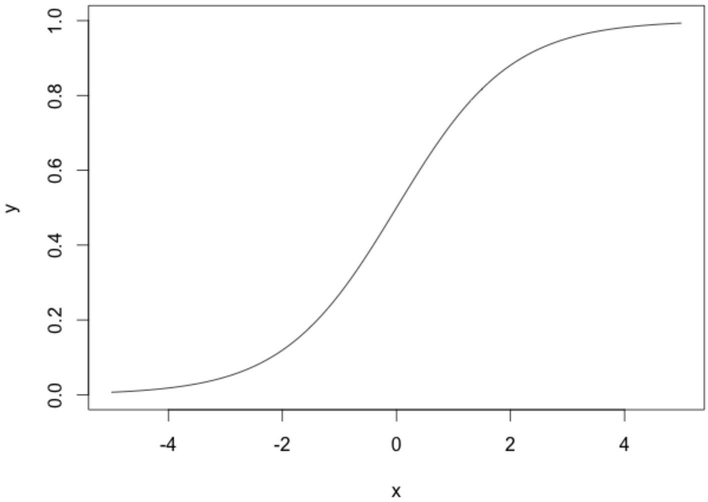
1.3.3 Sigmoid Activation Function

The sigmoid or logistic activation function is a very common choice for feedforward neural networks that need to output only positive numbers. Despite its widespread use, the hyperbolic tangent or the rectified linear unit (ReLU) activation function is usually a more suitable choice. We introduce the ReLU activation function later in this chapter. Equation 1.4 shows the sigmoid activation function:

$$\phi(x) = \frac{1}{1 + e^{-x}} \tag{1.4}$$

Use the sigmoid function to ensure that values stay within a relatively small range, as seen in Figure 1.9:

Figure 1.9: Sigmoid Activation Function



As you can see from the above graph, values above or below 0 are compressed to the approximate range between 0 and 1.

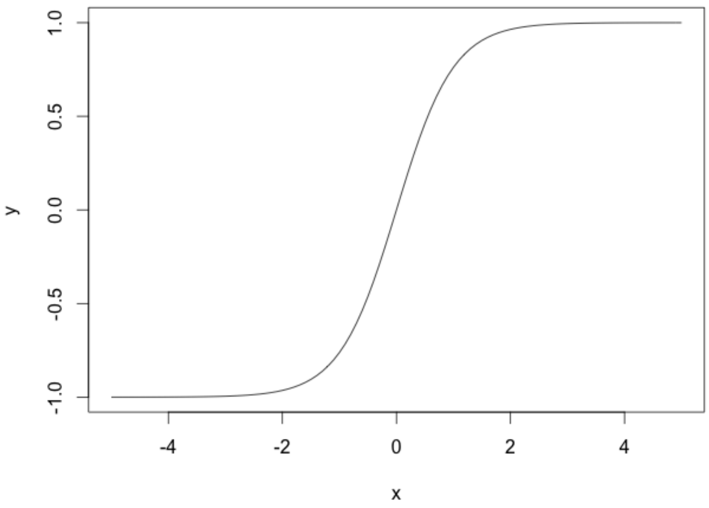
1.3.4 Hyperbolic Tangent Activation Function

The hyperbolic tangent function is also a very common activation function for neural networks that must output values in the range between -1 and 1. This activation function is simply the hyperbolic tangent (tanh) function, as shown in Equation 1.5:

$$\phi(x) = \tanh(x) \tag{1.5}$$

The graph of the hyperbolic tangent function has a similar shape to the sigmoid activation function, as seen in Figure 1.10:

Figure 1.10: Hyperbolic Tangent Activation Function



The hyperbolic tangent function has several advantages over the sigmoid activation function. These involve the derivatives used in the training of the neural network, and they will be covered in Chapter 6, “Backpropagation Training.”

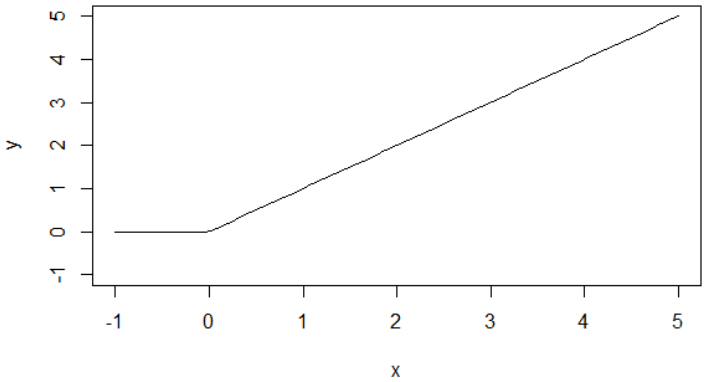
1.4 Rectified Linear Units (ReLU)

Introduced in 2000 by Teh & Hinton, the rectified linear unit (ReLU) has seen very rapid adoption over the past few years. Prior to the ReLU activation function, the hyperbolic tangent was generally accepted as the activation function of choice. Most current research now recommends the ReLU due to superior training results. As a result, most neural networks should utilize the ReLU on hidden layers and either softmax or linear on the output layer. Equation 1.6 shows the very simple ReLU function:

$$\phi(x) = \max(0, x) \tag{1.6}$$

We will now examine why ReLU typically performs better than other activation functions for hidden layers. Part of the increased performance is due to the fact that the ReLU activation function is a linear, non-saturating function. Unlike the sigmoid/logistic or the hyperbolic tangent activation functions, the ReLU does not saturate to -1, 0, or 1. A saturating activation function moves towards and eventually attains a value. The hyperbolic tangent function, for example, saturates to -1 as x decreases and to 1 as x increases. Figure 1.11 shows the graph of the ReLU activation function:

Figure 1.11: ReLU Activation Function



Most current research states that the hidden layers of your neural network should use the ReLU activation. The reasons for the superiority of the ReLU over hyperbolic tangent and sigmoid will be demonstrated in Chapter 6, “Backpropagation Training.”

1.4.1 Softmax Activation Function

The final activation function that we will examine is the softmax activation function. Along with the linear activation function, softmax is usually found in the output layer of a neural network. The softmax function is used on a classification neural network. The neuron that has the highest value claims the input as a member of its class. Because it is a preferable method, the softmax activation function forces the output of the neural network to represent the probability that the input falls into each of the classes. Without the softmax, the neuron’s outputs are simply numeric values, with the highest indicating the winning class.

To see how the softmax activation function is used, we will look at a common neural network classification problem. The iris data set contains four measurements for 150 different iris flowers. Each of these flowers belongs to one of three species of iris. When you provide the measurements of a flower, the softmax function allows the neural network to give you the probability that these measurements belong to each of the three species. For example, the neural network might tell you that there is an 80% chance that the iris is setosa, a 15% probability that it is virginica and only a 5% probability of versicolour. Because these are probabilities, they must add up to 100%. There could not be an 80% probability of setosa, a 75% probability of virginica and a 20% probability of versicolour–this type of a result would be nonsensical.

To classify input data into one of three iris species, you will need one output neuron for each of the three species. The output neurons do not inherently specify the probability of each of the three species. Therefore, it is desirable to provide probabilities that sum to 100%. The neural network will tell you the probability of a flower being each of the three species. To get the probability, use the softmax function in Equation 1.7:

$$\phi_i = \frac{e^{z_i}}{\sum\limits_{j \in group} e^{z_j}} \tag{1.7}$$

In the above equation, i represents the index of the output neuron (o) being calculated, and j represents the indexes of all neurons in the group/level. The variable z designates the array of output neurons. It's important to note that the softmax activation is calculated differently than the other activation functions in this chapter. When softmax is the activation function, the output of a single neuron is dependent on the other output neurons. In Equation 1.7, you can observe that the output of the other output neurons is contained in the variable z, as none of the other activation functions in this chapter utilize z. Listing 1.1 implements softmax in pseudocode:

Listing 1.1: The Softmax Function

```
def softmax(neuron_output):
  sum = 0
  for v in neuron_output:
    sum = sum + v

  sum = math.exp(sum)
  proba = []
  for i in range(len(neuron_output))
proba[i] = math.exp(neuron_output[i])/sum
  return proba
```

To see the softmax function in operation, refer to the following URL:

`http://www.heatonresearch.com/aifh/vol3/softmax.html`

Consider a trained neural network that classifies data into three categories, such as the three iris species. In this case, you would use one output neuron for each of the target classes. Consider if the neural network were to output the following:

```
Neuron 1: setosa: 0.9
Neuron 2: versicolour: 0.2
Neuron 3: virginica: 0.4
```

From the above output, we can clearly see that the neural network considers the data to represent a setosa iris. However, these numbers are not probabilities. The 0.9 value does not represent a 90% likelihood of the data representing a setosa. These values sum to 1.5. In order for them to be treated as probabilities, they must sum to 1.0. The output vector for this neural network is the following:

```
[0.9,0.2,0.4]
```

If this vector is provided to the softmax function, the following vector is returned:

```
[0.47548495534876745 , 0.2361188410001125 , 0.28839620365112]
```

The above three values do sum to 1.0 and can be treated as probabilities. The likelihood of the data representing a setosa iris is 48% because the first value in the vector rounds to 0.48 (48%). You can calculate this value in the following manner:

```
sum=exp(0.9)+exp(0.2)+exp(0.4)=5.17283056695839
j0= exp(0.9)/sum = 0.47548495534876745
j1= exp(0.2)/sum = 0.2361188410001125
j2= exp(0.4)/sum = 0.28839620365112
```

1.4.2 What Role does Bias Play?

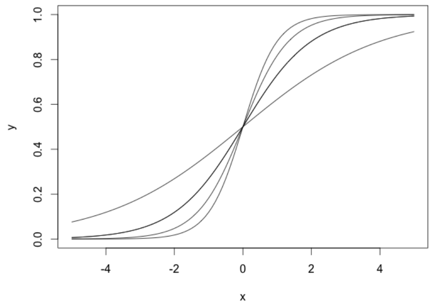
The activation functions seen in the previous section specify the output of a single neuron. Together, the weight and bias of a neuron shape the output of the activation to produce the desired output. To see how this process occurs, consider Equation 1.8. It represents a single-input sigmoid activation neural network.

$$f(x, w, b) = \frac{1}{1 + e^{-(wx+b)}} \tag{1.8}$$

The x variable represents the single input to the neural network. The w and b variables specify the weight and bias of the neural network. The above equation is a combination of the Equation 1.1 that specifies a neural network and Equation 1.4 that designates the sigmoid activation function.

The weights of the neuron allow you to adjust the slope or shape of the activation function. Figure 1.12 shows the effect on the output of the sigmoid activation function if the weight is varied:

Figure 1.12: Adjusting Neuron Weight



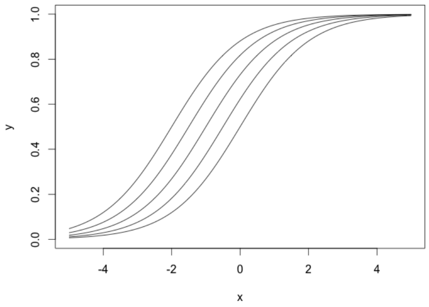
The above diagram shows several sigmoid curves using the following parameters:

```
f(x,0.5,0.0)
f(x,1.0,0.0)
f(x,1.5,0.0)
f(x,2.0,0.0)
```

To produce the curves, we did not use bias, which is evident in the third parameter of 0 in each case. Using four weight values yields four different sigmoid curves in Figure 1.11. No matter the weight, we always get the same value of 0.5 when x is 0 because all of the curves hit the same point when x is 0. We might need the neural network to produce other values when the input is near 0.5.

Bias does shift the sigmoid curve, which allows values other than 0.5 when x is near 0. Figure 1.13 shows the effect of using a weight of 1.0 with several different biases:

Figure 1.13: Adjusting Neuron Bias



The above diagram shows several sigmoid curves with the following parameters:

```
f(x,1.0,1.0)
f(x,1.0,0.5)
f(x,1.0,1.5)
f(x,1.0,2.0)
```

We used a weight of 1.0 for these curves in all cases. When we utilized several different biases, sigmoid curves shifted to the left or right. Because all the curves merge together at the top right or bottom left, it is not a complete shift.

When we put bias and weights together, they produced a curve that created the necessary output from a neuron. The above curves are the output from only one neuron. In a complete network, the output from many different neurons will combine to produce complex output patterns.

1.5 Logic with Neural Networks

As a computer programmer, you are probably familiar with logical programming. You can use the programming operators AND, OR, and NOT to govern how a program makes decisions. These logical operators often define the actual meaning of the weights and biases in a neural network. Consider the following truth table:

```
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
NOT 0 = 1
NOT 1 = 0
```

The truth table specifies that if both sides of the AND operator are true, the final output is also true. In all other cases, the result of the AND is false. This definition fits with the English word "and" quite well. If you want a house with a nice view AND a large backyard, then both requirements must be fulfilled for you to choose a house. If you want a house that has a nice view or a large backyard, then only one needs to be present.

These logical statements can become more complex. Consider if you want a house that has a nice view and a large backyard. However, you would also be satisfied with a house that has a small backyard yet is near a park. You can express this idea in the following way:

```
([nice view] AND [large yard]) OR ((NOT [large yard]) and [park])
```

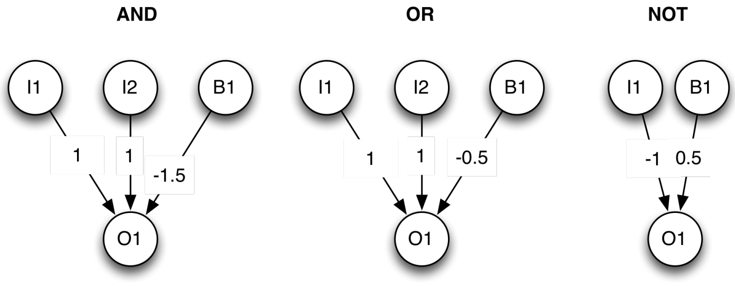
You can express the previous statement with the following logical operators:

$$(NV \wedge LY) \vee (\neg LY \wedge PK) \tag{1.9}$$

In the above statement, the OR looks like a letter "v," the AND looks like an upside down "v," and the NOT looks like half of a box.

We can use neural networks to represent the basic logical operators of AND, OR, and NOT, as seen in Figure 1.14:

Figure 1.14: Basic Logic Operators



The above diagram shows the weights and bias weight for each of the three fundamental logical operators. You can easily calculate the output for any of these operators using Equation 1.1. Consider the AND operator with two true (1) inputs:

```
(1*1) + (1*1) + (-1.5) = 0.5
```

We are using a step activation function. Because 0.5 is greater than or equal to 0.5, the output is 1 or true. We can evaluate the expression where the first input is false:

```
(1*1) + (0*1) + (-1.5) = -0.5
```

Because of the step activation function, this output is 0 or false.

We can build more complex logical structures from these neurons. Consider the exclusive or (XOR) operator that has the following truth table:

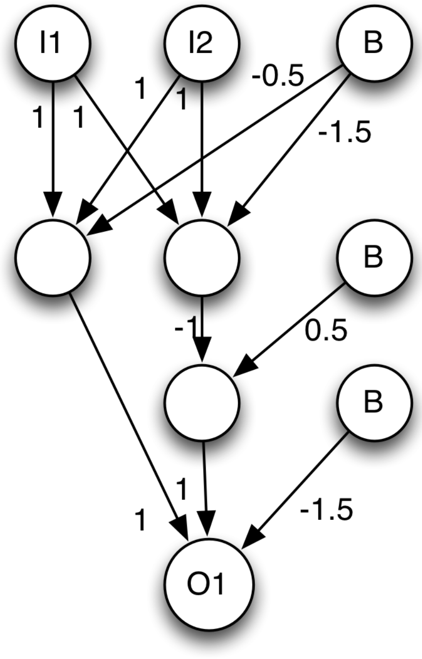
```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

The XOR operator specifies that one, but not both, of the inputs can be true. For example, one of the two cars will win the race, but not both of them will win. The XOR operator can be written with the basic AND, OR, and NOT operators as follows:

$$p \oplus q = (p \lor q) \land \neg(p \land q) \tag{1.10}$$

The plus with a circle is the symbol for the XOR operator, and p and q are the two inputs to evaluate. The above expression makes sense if you think of the XOR operator meaning p or q, but not both p and q. Figure 1.15 shows a neural network that can represent an XOR operator:

Figure 1.15: XOR Neural Network



Calculating the above neural network would require several steps. First, you must calculate the values for every node that is directly connected to the inputs. In the case of the above neural network, there are two nodes. We will show an example of calculating the XOR with the inputs [0,1]. We begin by calculating the two topmost, unlabeled (hidden) nodes:

```
(0*1) + (1*1) - 0.5 = 0.5 = True
(0*1) + (1*1) - 1.5 = -0.5 = False
```

Next we calculate the lower, unlabeled (hidden) node:

```
(0*-1)+0.5 = 0.5 = True
```

Finally, we calculate O1:

```
(1*1)+(1*1)-1.5 = 0.5 = True
```

As you can see, you can manually wire the connections in a neural network to produce the desired output. However, manually creating neural networks is very tedious. The rest of the book will include several algorithms that can automatically determine the weight and bias values.

1.6 Chapter Summary

In this chapter, we showed that a neural network is comprised of neurons, layers, and activation functions. Fundamentally, the neurons in a neural network might be input, hidden, or output in nature. Input and output neurons pass information into and out of the neural network. Hidden neurons occur between the input and output neurons and help process information.

Activation functions scale the output of a neuron. We also introduced several activation functions. The two most common activation functions are the sigmoid and hyperbolic tangent. The sigmoid function is appropriate for networks in which only positive output is needed. The hyperbolic tangent function supports both positive and negative output.

A neural network can build logical statements, and we demonstrated the weights to generate AND, OR, and NOT operators. Using these three basic operators, you can build more complex, logical expressions. We presented an example of building an XOR operator.

Now that we've seen the basic structure of a neural network, we will explore in the next two chapters several classic neural networks so that you can use this abstract structure. Classic neural network structures include the self-organizing map, the Hopfield neural network, and the Boltzmann machine. These classical neural networks form the foundation of other architectures that we present in the book.

Chapter 2

Self-Organizing Maps

- Self-Organizing Maps
- Neighborhood Functions
- Unsupervised Training
- Dimensionality

Now that you have explored the abstract nature of a neural network introduced in the previous chapter, you will learn about several classic neural network types. This chapter covers one of the earliest types of neural networks that are still useful today. Because neurons can be connected in various ways, many different neural network architectures exist and build on the fundamental ideas from Chapter 1, "Neural Network Basics." We begin our examination of classic neural networks with the self-organizing map (SOM).

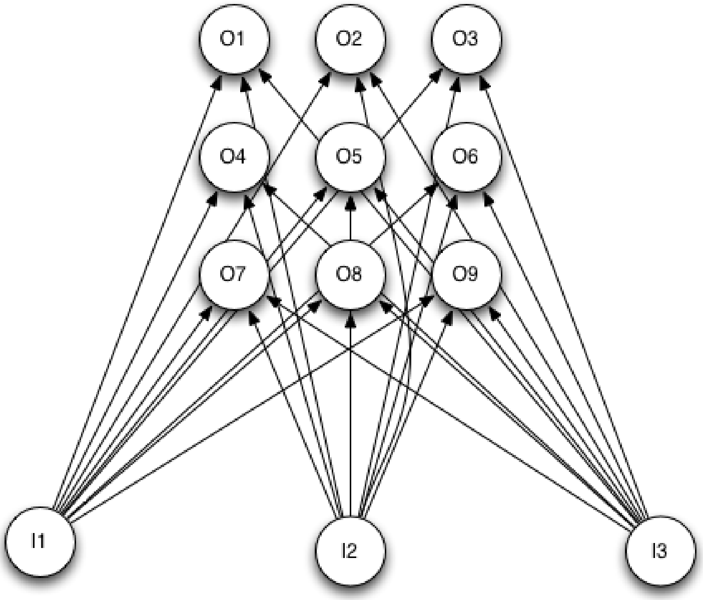
The SOM is used to classify neural input data into one of several groups. Training data is provided to the SOM, as well as the number of groups into which you wish to classify these data. While training, the SOM will group these data into groups. Data that have the most similar characteristics will be grouped together. This process is very similar to clustering algorithms, such as *k*-means. However, unlike *k*-means, which only groups an initial set of data, the SOM can continue classifying new data beyond the initial data set that was used for training. Unlike most of the neural networks in this book, SOM is unsupervised–you do not tell it what groups you expect the training data to fall into. The SOM simply figures out the groups itself, based on your training data, and then it classifies any future data into similar groups. Future classification is performed using what the SOM learned from the training data.

2.1 Self-Organizing Maps

Kohonen (1988) introduced the self-organizing map (SOM), a neural network consisting of an input layer and an output layer. The two-layer SOM is also known as the Kohonen neural network and functions when the input layer maps data to the output layer. As the program presents patterns to the input layer, the output neuron is considered the winner when it contains the weights most similar to the input. This similarity is calculated by comparing the Euclidean distance between the set of weights from each output neuron. The shortest Euclidean distance wins. Calculating Euclidean distance is the focus of the next section.

Unlike the feedforward neural network discussed in Chapter 1, there are no bias values in the SOM. It just has weights from the input layer to the output layer. Additionally, it uses only a linear activation function. Figure 2.1 shows the SOM:

Figure 2.1: Self-Organizing Map



The SOM pictured above shows how the program maps three input neurons to nine output neurons arranged in a three-by-three grid. The output neurons of the SOM are often arranged into a grid, cube, or other higher-dimensional construct. Because the ordering of the output neurons in most neural networks typically conveys no meaning at all, this arrangement is very different. For example, the close proximity of the output neurons #1 and #2 in most neural networks is not significant. However, for the SOM, the closeness of one output neuron to another is important. Computer vision applications make use of the closeness of neurons to identify images more accurately. Convolutional neural networks (CNNs), which will be examined in Chapter 10, “Convolutional Neural Networks,” group neurons into overlapping regions based on how close these input neurons are to each other. When recognizing images, it is very important to consider which pixels are near each other. The program recognizes patterns such as edges, solid regions, and lines by looking at pixels near each other.

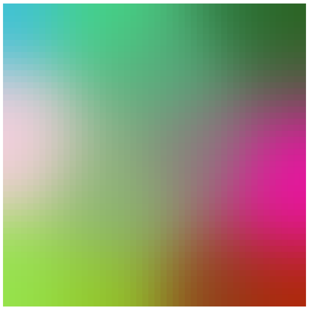
Common structures for the output neurons of SOMs include the following:

- **One-Dimensional:** Output neurons are arranged in a line.
- **Two-Dimensional:** Output neurons are arranged in a grid.
- **Three-Dimensional:** Output neurons are arranged in a cube.

We will now see how to structure a simple SOM that learns to recognize colors that are given as RGB vectors. The individual red, green, and blue values can range between -1 and +1. Black or the absence of color designates -1, and +1 expresses the full intensity of red, green or blue. These three-color components comprise the neural network input.

The output will be a 2,500-neuron grid arranged into 50 rows by 50 columns. This SOM will organize similar colors near each other in this output grid. Figure 2.2 shows this output:

Figure 2.2: The Output Grid



Although the above figure may not be as clear in the black and white editions of this book as it is in the color e-book editions, you can see similar colors grouped near each other. A single, color-based SOM is a very simple example that allows you to visualize the grouping capabilities of the SOM.

How are SOMs trained? The training process will update the weight matrix, which is 3 by 2,500. To start, the program initializes the weight matrix to random values. Then it randomly chooses 15 training colors.

The training will progress through a series of iterations. Unlike other neural network types, the training for SOM networks involves a fixed number of iterations. To train the color-based SOM, we will use 1,000 iterations.

Each iteration will choose one random color sample from the training set, a collection of RGB color vectors that each consist of three numbers. Likewise, the weights between each of the 2,500 output neurons and the three input neurons are a vector of three numbers. As training progresses, the program will calculate the Euclidean distance between each weight vector and the current training pattern. A Euclidean distance determines the difference between two vectors of the same size. In this case, both vectors are three numbers that represent an RGB color. We compare the color from the training data to the three weights of each neuron. Equation 2.1 shows the Euclidean distance calculation:

$$\mathrm{d}(\mathbf{p},\mathbf{w}) = \sqrt{\sum_{i=1}^{n}(p_i - w_i)^2} \tag{2.1}$$

In the above equation, the variable p represents the training pattern. The variable w corresponds to the weight vector. By squaring the differences between each vector component and taking the square root of the resulting sum, we calculate the Euclidean distance. This calculation measures the difference between each weight vector and the input training pattern.

The program calculates the Euclidean distance for every output neuron, and the one with the shortest distance is called the best matching unit (BMU). This neuron will learn the most from the current training pattern. The neighbors of the BMU will learn less. To perform this training, the program loops over every neuron and determines the extent to which it should be trained. Neurons that are closer to the BMU will receive more training. Equation 2.2 can make this determination:

$$W_v(t+1) = W_v(t) + \theta(v,t)\alpha(t)(D(t) - W_v(t)) \tag{2.2}$$

In the above equation, the variable t, also known as the iteration number, represents time. The purpose of the equation is to calculate the resulting weight vector Wv(t+1). You will determine the next weight by adding to the current weight, which is Wv(t). The end goal is to calculate how different the current weight is from the input vector, and it is done by the clause D(T)-Wv(t). Training the SOM is the process of making a neuron's weights more similar to the training element. We do not want to simply assign the training element to the output neurons weights, making them identical. Rather, we calculate the difference between the training element and the neurons weights and scale this difference by multiplying it by two ratios. The first ratio, represented by θ (theta), is the neighborhood function. The second ratio, represented by α (alpha), is a monotonically decreasing learning rate. In other words, as the training progresses, the learning rate falls and never rises.

The neighborhood function considers how close each output neuron is to the BMU. For neurons that are nearer, the neighborhood function will return a value that approaches 1. For distant neighbors, the neighborhood function will approach 0. This range between 0 and 1 controls how near and far neighbors are trained. Nearer neighbors will receive more of the training adjustment to their weights. In the next section, we will analyze how the neighborhood function determines the training adjustments. In addition to the neighborhood function, the learning rate also scales how much the program will adjust the output neuron.

2.1.1 Understanding Neighborhood Functions

The neighborhood function determines the degree to which each output neuron should receive a training adjustment from the current training pattern. The function usually returns a value of 1 for the BMU. This value indicates that the BMU should receive the most training. Those neurons farther from the BMU will receive less training. The neighborhood function determines this weighting.

If the output neurons are arranged in only one dimension, you should use a simple one-dimensional neighborhood function, which will treat the output as one long array of numbers. For instance, a one-dimensional network might have 100 output neurons that form a long, single-dimensional array of 100 values.

A two-dimensional SOM might take these same 100 values and represent them as a grid, perhaps of 10 rows and 10 columns. The actual structure remains the same; the neural network has 100 output neurons. The only difference is the neighborhood function. The first would utilize a one-dimensional neighborhood function; the second would use a two-dimensional neighborhood function. The function must consider this additional dimension and factor it into the distance returned.

It is also possible to have three, four, and even more dimensional functions for the neighborhood function. Typically, neighborhood functions are expressed in vector form so that the number of dimensions does not matter. To represent the dimensions, the Euclidian norm (represented by two vertical bars) of all inputs is taken, as seen in Equation 2.3:

$$||p - w|| = \sqrt{\sum_{i=1}^{n}(p_i - w_i)^2} \tag{2.3}$$

For the above equation, the variable p represents the dimensional inputs. The variable w represents the weights. A single dimension has only a single value for p. Calculating the Euclidian norm for *[2-0]* would simply be the following:

$$||2 - 0|| = \sqrt{2^2} = 2 \tag{2.4}$$

Calculating the Euclidean norm for *[2-0, 3-0]* is only slightly more complex:

$$||\ [2 - 0, 3 - 0]\ || = \sqrt{2^2 + 3^2} = 3.605551 \tag{2.5}$$

The most popular choice for SOMs is the two-dimensional neighborhood function. One-dimensional neighborhood functions are also common. However, neighborhood functions with three or more dimensions are more unusual. Choosing the number of dimensions really comes down to the programmer deciding how many ways an output neuron can be close to another. This decision should not be taken lightly because each additional dimension significantly affects the amount of memory and processing power needed. This additional processing is why most programmers choose two or three dimensions for the SOM application.

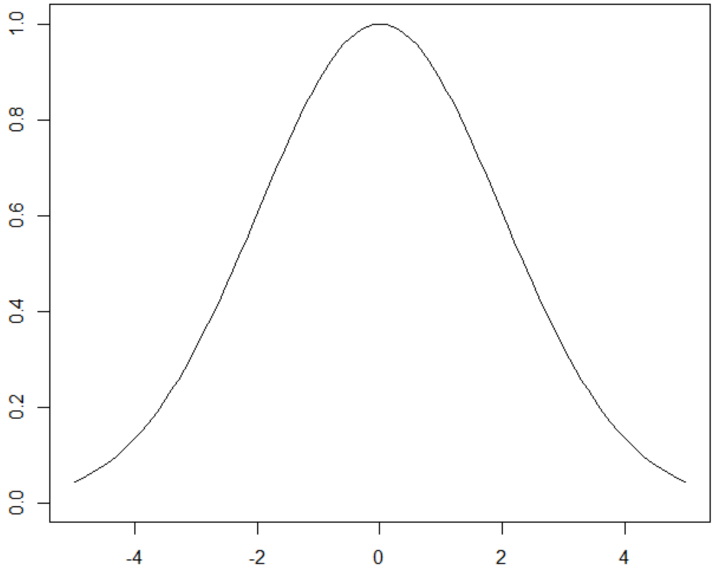
It can be difficult to understand why you might have more than three dimensions. The following analogy illustrates the limitations of three dimensions. While at the grocery store, John noticed a package of dried apples. As he turned his head to the left or right, traveling in the first dimension, he saw other brands of dried apples. If he looked up or down, traveling in the second dimension, he saw other types of dried fruit. The third dimension, depth, simply gives him more of exactly the same dried apples. He reached behind the front item and found additional stock. However, there is no fourth dimension, which could have been useful to allow fresh apples to be located near to the dried apples. Because the supermarket only had three dimensions, this type of link is not possible. Programmers do not have this limitation, and they must decide if the extra processing time is necessary for the benefits of additional dimensions.

The Gaussian function is a popular choice for a neighborhood function. Equation 2.4 uses the Euclidean norm to calculate the Gaussian function for any number of dimensions:

$$f(x, c, w) = e^{-(w||x-c||)^2} \tag{2.6}$$

The variable x represents the input to the Gaussian function, c represents the center of the Gaussian function, and w represents the widths. The variables x, w and c all are vectors with multiple dimensions. Figure 2.3 shows the graph of the two-dimensional Gaussian function:

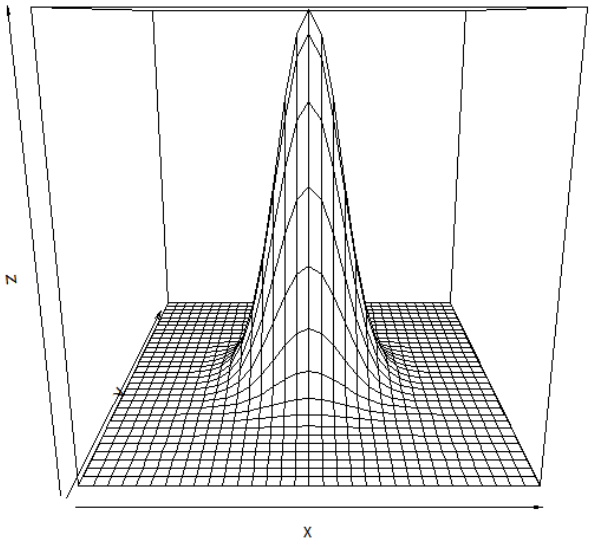
Figure 2.3: A Single-Dimensional Gaussian Function



This figure illustrates why the Gaussian function is a popular choice for a neighborhood function. Programmers frequently use the Gaussian function to show the normal distribution, or bell curve. If the current output neuron is the BMU, then its distance (x-axis) will be 0. As a result, the training percent (y-axis) is 1.0 (100%). As the distance increases either positively or negatively, the training percentage decreases. Once the distance is large enough, the training percent approaches 0.

If the input vector to the Gaussian function has two dimensions, the graph appears as Figure 2.4:

Figure 2.4: A Two-Dimensional Gaussian Function



How does the algorithm use Gaussian constants with a neural network? The center (c) of a neighborhood function is always 0, which centers the function on the origin. If the algorithm moves the center from the origin, a neuron other than the BMU would receive the full learning. It is unlikely you would ever want to move the center from the origin. For a multi-dimensional Gaussian, set all centers to 0 in order to position the curve at the origin.

The only remaining Gaussian parameter is the width. You should set this parameter to something slightly less than the entire width of the grid or array. As training progresses, the width gradually decreases. Just like the learning rate, the width should decrease monotonically.

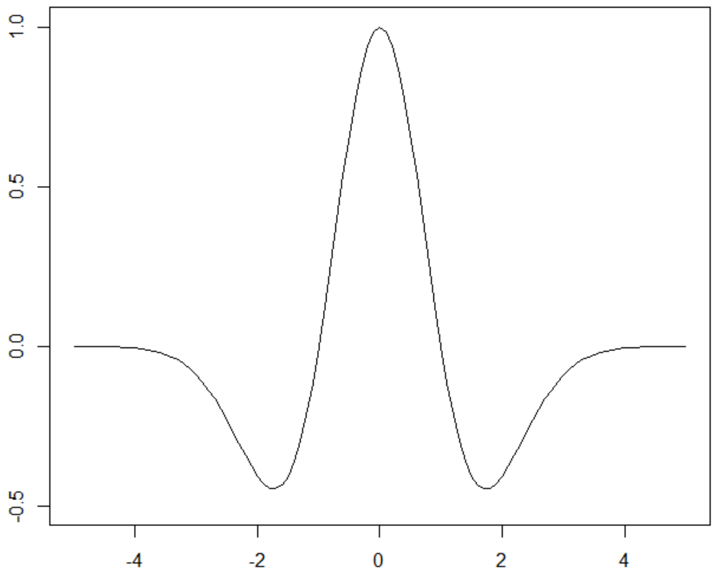
2.1.2 Mexican Hat Neighborhood Function

Though it is the most popular, the Gaussian function is not the only neighborhood function available. The Ricker wave, or Mexican hat function, is another popular neighborhood function. Just like the Gaussian neighborhood function, the vector length of the x dimensions is the basis for the Mexican hat function, as seen in Equation 2.5:

$$f(x,c,w) = \left(1 - \frac{||x-c||^2}{w}\right) e^{-\frac{||x-c||^2}{2w}} \tag{2.7}$$

Much the same as the Gaussian, the programmer can use the Mexican hat function in one or more dimensions. Figure 2.5 shows the Mexican hat function with one dimension:

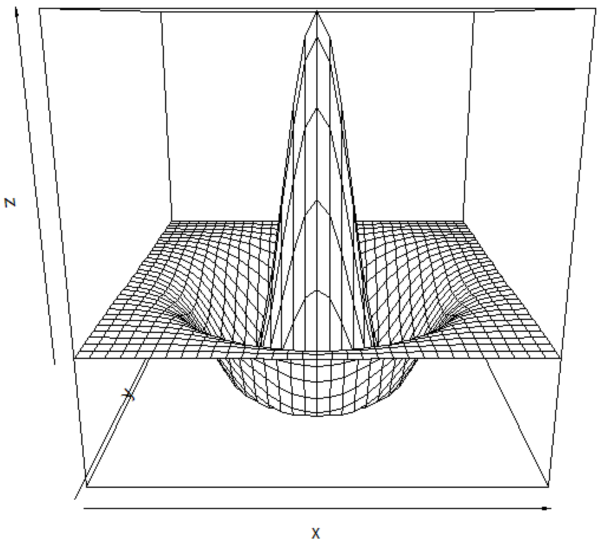
Figure 2.5: A One-Dimensional Mexican Hat Function



You must be aware that the Mexican hat function penalizes neighbors that are between 2 and 4, or -2 and -4 units from the center. If your model seeks to penalize near misses, the Mexican hat function is a good choice.

You can also use the Mexican hat function in two or more dimensions. Figure 2.6 shows a two-dimensional Mexican hat function:

Figure 2.6: A Two-Dimensional Mexican Hat Function



Just like the one-dimensional version, the above Mexican hat penalizes near misses. The only difference is that the two-dimensional Mexican hat function utilizes a two-dimensional vector, which looks more like a Mexican sombrero than the one-dimensional variant. Although it is possible to use more than two dimensions, these variants are hard to visualize because we perceive space in three dimensions.

2.1.3 Calculating SOM Error

Supervised training typically reports an error measurement that decreases as training progresses. Unsupervised models, such as the SOM network, cannot directly calculate an error because there is no expected output. However, an estimation of the error can be calculated for the SOM (Masters, 1993).

We define the error as the longest Euclidean distance of all BMUs in a training iteration. Each training set element has its own BMU. As learning progresses, the longest distance should decrease. The results also indicate the success of the SOM training since the values will tend to decrease as the training continues.

2.2 Chapter Summary

In the first two chapters, we explained several classic neural network types. Since Pitts (1943) introduced the neural network, many different neural network types have been invented. We have focused primarily on the classic neural network types that still have relevance and that establish the foundation for other architectures that we will cover in later chapters of the book.

This chapter focused on the self-organizing map (SOM) that is an unsupervised neural network type that can cluster data. The SOM has an input neuron count equal to the number of attributes for the data to be clustered. An output neuron count specifies the number of groups into which the data should be clustered. The SOM neural network is trained in an unsupervised manner. In other words, only the data points are provided to the neural network; the expected outputs are not provided. The SOM network learns to cluster the data points, especially the data points similar to the ones with which it trained.

In the next chapter, we will examine two more classic neural network types: the Hopfield neural network and the Boltzmann machine. These neural network types are similar in that they both use an energy function during their training process. The energy function measures the amount of energy in the network. As training progresses, the energy should decrease as the network learns.

Chapter 3

Hopfield & Boltzmann Machines

- Hopfield Networks
- Energy Functions
- Hebbian Learning
- Associative Memory
- Optimization
- Boltzmann Machines

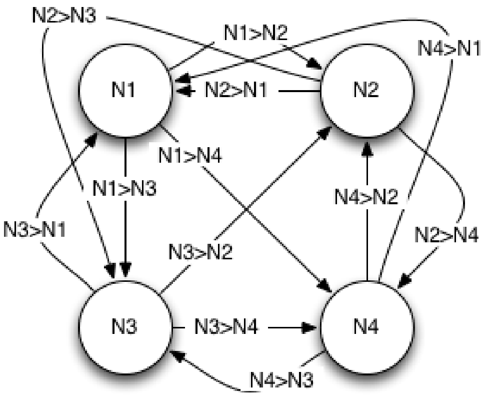
This chapter will introduce the Hopfield network as well as the Boltzmann machine. Though neither of these classic neural networks is used extensively in modern AI applications, both are foundational to more modern algorithms. The Boltzmann machine forms the foundation of the deep belief neural network (DBNN), which is one of the fundamental algorithms of deep learning. Hopfield networks are a very simple type of neural network that utilizes many of the same features that the more complex feedforward neural networks employ.

3.1 Hopfield Neural Networks

The Hopfield neural network (Hopfield, 1982) is perhaps the simplest type of neural network because it is a fully connected single layer, auto-associative network. In other words, it has a single layer in which each neuron is connected to every other neuron. Additionally, the term auto-associative means that the neural network will return the entire pattern if it recognizes a pattern. As a result, the network will fill in the gaps of incomplete or distorted patterns.

Figure 3.1 shows a Hopfield neural network with just four neurons. While a four-neuron network is handy because it is small enough to visualize, it can recognize a few patterns.

Figure 3.1: A Hopfield Neural Network with 12 Connections



Because every neuron in a Hopfield neural network is connected to every other neuron, you might assume that a four-neuron network would contain a four-by-four matrix, or 16 connections. However, 16 connections would require that every neuron be connected to itself as well as to every other neuron. In a Hopfield neural network, 16 connections do not occur; the actual number of connections is 12.

These connections are weighted and stored in a matrix. A four-by-four matrix would store the network pictured above. In fact, the diagonal of this matrix would contain 0's because there are no self-connections. All neural network examples in this book will use some form of matrix to store their weights.

Each neuron in a Hopfield network has a state of either true (1) or false (-1). These states are initially the input to the Hopfield network and ultimately become the output of the network. To determine whether a Hopfield neuron's state is -1 or 1, use Equation 3.1:

$$s_i \leftarrow \begin{cases} +1 & \text{if } \sum_j w_{ij} s_j \geq \theta_i, \\ -1 & \text{otherwise.} \end{cases} \tag{3.1}$$

The above equation calculates the state (s) of neuron i. The state of a given neuron greatly depends on the states of the other neurons. The equation multiplies and sums the weight (w) and state (s) of the other neurons (j). Essentially, the state of the current neuron (i) is +1 if this sum is greater than the threshold (θ, theta). Otherwise it is -1. The threshold value is usually 0.

Because the state of a single neuron depends on the states of the remaining neurons, the order in which the equation calculates the neurons is very important. Programmers frequently employ the following two strategies to calculate the states for all neurons in a Hopfield network:

- **Asynchronous**: This strategy updates only one neuron at a time. It picks this neuron at random.
- **Synchronous**: It updates all neurons at the same time. This method is less realistic since biological organisms lack a global clock that synchronizes the neurons.

You should typically run a Hopfield network until the values of all neurons stabilize. Despite the fact that each neuron is dependent on the states of the others, the network will usually converge to a stable state.

It is important to have some indication of how close the network is to converging to a stable state. You can calculate an energy value for Hopfield networks. This value decreases as the Hopfield network moves to a more stable state. To evaluate the stability of the network, you can use the energy function. Equation 3.2 shows the energy calculation function:

$$E = -\left(\sum_{i<j} w_{ij}\, s_i\, s_j + \sum_i \theta_i\, s_i\right) \tag{3.2}$$

Boltzmann machines, discussed later in the chapter, also utilize this energy function. Boltzmann machines share many similarities with Hopfield neural networks. When the threshold is 0, the second term of Equation 3.2 drops out. Listing 3.1 contains the code to implement Equation 3.1:

Listing 3.1: Hopfield Energy

```
def energy(weights,state,threshold):
  # First term
  a = 0
  for i in range(neuron_count):
    for j in range(neuron_count):
      a = a + weight[i][j] * state[i] * state[j]

  a = a * -0.5
  # Second term
  b = 0
  for i in range(neuron_count):
    b = b + state[i] * threshold[i]

  # Result
  return a + b
```

3.1.1 Training a Hopfield Network

You can train Hopfield networks to arrange their weights in a way that allows the network to converge to desired patterns, also known as the training set.

These desired training patterns are a list of patterns with a Boolean value for each of the neurons that comprise the Boltzmann machine. The following data might represent a four-pattern training set for a Hopfield network with eight neurons:

```
1 1 0 0 0 0 0 0
0 0 0 0 1 1 0 0
1 0 0 0 0 0 0 1
0 0 0 1 1 0 0 0
```

The above data are completely arbitrary; however, they do represent actual patterns to train the Hopfield network. Once trained, a pattern similar to the one listed below should find equilibrium with a pattern close to the training set:

```
1 1 1 0 0 0 0 0
```

Therefore, the state of the Hopfield machine should change to the following pattern:

```
1 1 0 0 0 0 0 0
```

You can train Hopfield networks with either Hebbian (Hopfield, 1982) or Storkey (Storkey, 1999) learning. The Hebbian process for learning is biologically plausible, and it is often expressed as, "cells that fire together, wire together." In other words, two neurons will become connected if they frequently react to the same input stimulus. Equation 3.3 summarizes this behavior mathematically:

$$w_{ij} = \frac{1}{n} \sum_{\mu=1}^{n} \epsilon_i^{\mu} \epsilon_j^{\mu} \tag{3.3}$$

The constant n represents the number of training set elements (ϵ, epsilon). The weight matrix will be square and will contain rows and columns equal to the number of neurons. The diagonal will always be 0 because a neuron is not connected to itself. The other locations in the matrix will contain values specifying how often two values in the training pattern are either +1 or -1. Listing 3.2 contains the code to implement Equation 3.3:

Listing 3.2: Hopfield Hebbian Training

```
def add_pattern(weights,pattern,n):
  for i in range(neuron_count):
    for j in range(neuron_count):
      if i==j:
        weights[i][j] = 0
      else:
        weights[i][j] = weights[i][j]
          + ((pattern[i] * pattern[j])/n)
```

We apply the **add_pattern** method to add each of the training elements. The parameter **weights** specifies the weight matrix, and the parameter **pattern** specifies each individual training element. The variable **n** designates the number of elements in the training set.

It is possible that the equation and the code are not sufficient to show how the weights are generated from input patterns. To help you visualize this process, we provide an online Javascript application at the following URL:

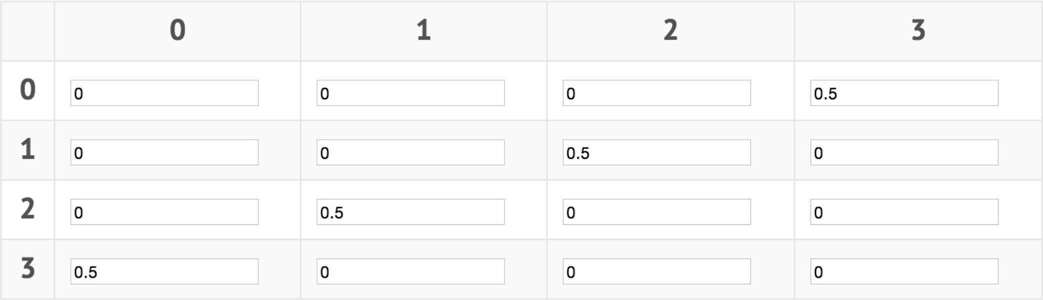
`http://www.heatonresearch.com/aifh/vol3/hopfield.html`

Consider the following data to train a Hopfield network:

```
[1 ,0 ,0 ,1]
[0 ,1 ,1 ,0]
```

The previous data should produce a weight matrix like Figure 3.2:

Figure 3.2: Hopfield Matrix



	0	1	2	3
0	0	0	0	0.5
1	0	0	0.5	0
2	0	0.5	0	0
3	0.5	0	0	0

To calculate the above matrix, divide 1 by the number of training set elements. The result is 1/2, or 0.5. The value 0.5 is placed into every row and column that has a 1 in the training set. For example, the first training element has a 1 in neurons #0 and #3, resulting in a 0.5 being added to row 0, column 3 and row 3, column 0. The same process continues for the other training set element.

Another common training technique for Hopfield neural networks is the Storkey training algorithm. Hopfield neural networks trained with Storkey have a greater capacity of patterns than the Hebbian method just described. The Storkey algorithm is more complex than the Hebbian algorithm.

The first step in the Storkey algorithm is to calculate a value called the local field. Equation 3.4 calculates this value:

$$h_{ij} = \sum_{k=1,k\neq i,j} w_{ik}\epsilon_k \tag{3.4}$$

We calculate the local field value (h) for each weight element (i & j). Just as before, we use the weights (w) and training set elements (ϵ, epsilon). Listing 3.3 provides the code to calculate the local field:

Listing 3.3: Calculate Storkey Local Field

```
def calculate_local_field(weights, i, j, pattern):
  sum = 0
  for k in range(len(pattern)):
    if k != i:
      sum = sum + weights[i][k] * pattern[k]
  return sum
```

Equation 3.5 has the local field value that calculates the needed change (ΔW):

$$\Delta w_{ij} = \frac{1}{n}\epsilon_i\epsilon_j - \frac{1}{n}\epsilon_i h_{ji} - \frac{1}{n}\epsilon_j h_{ij} \tag{3.5}$$

Listing 3.4 calculates the values of the weight deltas:

Listing 3.4: Storkey Learning

```
def add_pattern(weights, pattern):
  sum_matrix = matrix(len(pattern),len(pattern))
  n = len(pattern)
  for i in range(n):
    for j in range(n):
      t1 = (pattern[i] * pattern[j])/n
      t2 = (pattern[i] *
          calculate_local_field(weights,j,i,pattern))/n
      t3 = (pattern[j] *
          calculate_local_field(weights,i,j,pattern))/n
      d = t1-t2-t3;
      sum_matrix[i][j] = sum_matrix[i][j] + d
  return sum_matrix
```

Once you calculate the weight deltas, you can add them to the existing weight matrix. If there is no existing weight matrix, simply allow the delta weight matrix to become the weight matrix.

3.2 Hopfield-Tank Networks

In the last section, you learned that Hopfield networks can recall patterns. They can also optimize problems such as the traveling salesman problem (TSP). Hopfield and Tank (1984) introduced a special variant, the Hopfield-Tank network, to find solutions to optimization problems.

The structure of a Hopfield-Tank network is somewhat different than a standard Hopfield network. The neurons in a regular Hopfield neural network can hold only the two discrete values of 0 or 1. However, a Hopfield-Tank neuron can have any number in the range 0 to 1. Standard Hopfield networks possess discrete values; Hopfield-Tank networks keep continuous values over a range. Another important difference is that Hopfield-Tank networks use sigmoid activation functions.

To utilize a Hopfield-Tank network, you must create a specialized energy function to express the parameters of each problem to solve. However, producing such an energy function can be a time-consuming task. Hopfield & Tank (2008) demonstrated how to construct an energy function for the traveling salesman problem (TSP). Other optimization functions, such as simulated annealing and Nelder-Mead, do not require the creation of a complex energy function. These general-purpose optimization algorithms typically perform better than the older Hopfield-Tank optimization algorithms.

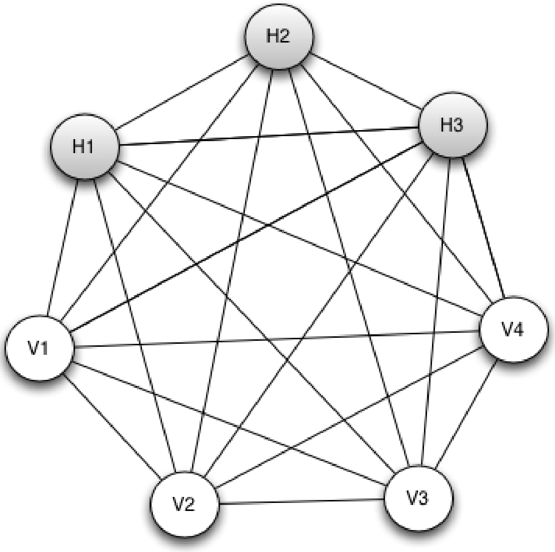
Because other algorithms are typically better choices for optimizations, this book does not cover the optimization Hopfield-Tank network. Nelder-Mead and simulated annealing were demonstrated in *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms.* Chapter 6, “Backpropagation Training,” will have a review of stochastic gradient descent (SGD), which is one of the best training algorithms for feedforward neural networks.

3.3 Boltzmann Machines

Hinton & Sejnowski (1985) first introduced Boltzmann machines, but this neural network type has not enjoyed widespread use until recently. A special type of Boltzmann machine, the restricted Boltzmann machine (RBM), is one of the foundational technologies of deep learning and the deep belief neural network (DBNN). In this chapter, we will introduce classic Boltzmann machines. Chapter 9, "Deep Learning," will include deep learning and the restricted Boltzmann machine.

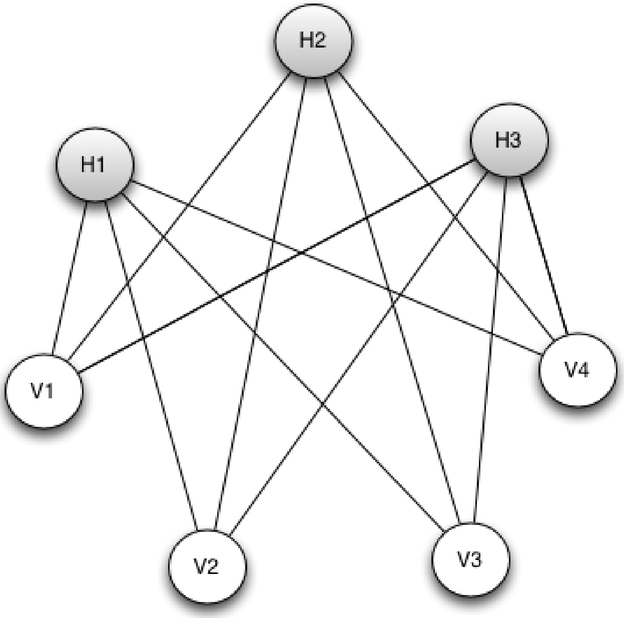
A Boltzmann machine is essentially a fully connected, two-layer neural network. We refer to these layers as the visual and hidden layers. The visual layer is analogous to the input layer in feedforward neural networks. Despite the fact that a Boltzmann machine has a hidden layer, it functions more as an output layer. This difference in the meaning of hidden layer is often a source of confusion between Boltzmann machines and feedforward neural networks. The Boltzmann machine has no hidden layer between the input and output layers. Figure 3.3 shows the very simple structure of a Boltzmann machine:

Figure 3.3: Boltzmann Machine



The above Boltzmann machine has three hidden neurons and four visible neurons. A Boltzmann machine is fully connected because every neuron has a connection to every other neuron. However, no neuron is connected to itself. This connectivity is what differentiates a Boltzmann machine from a restricted Boltzmann machine (RBM), as seen in Figure 3.4:

Figure 3.4: Restricted Boltzmann Machine (RBM)



The above RBM is not fully connected. All hidden neurons are connected to each visible neuron. However, there are no connections among the hidden neurons nor are there connections among the visible neurons.

Like the Hopfield neural network, a Boltzmann machine's neurons acquire only binary states, either 0 or 1. While there is some research on continuous Boltzmann machines capable of assigning decimal numbers to the neurons, nearly all research on the Boltzmann machine centers on binary units. Therefore, this book will not include information on continuous Boltzmann machines.

Boltzmann machines are also called a generative model. In other words, a Boltzmann machine does not generate constant output. The values presented to the visible neurons of a Boltzmann machine, when considered with the weights, specify a probability that the hidden neurons will assume a value of 1, as opposed to 0.

Although a Boltzmann machine and Hopfield neural networks have some characteristics in common, there are several important differences:

- Hopfield networks suffer from recognizing certain false patterns.
- Boltzmann machines can store a greater capacity of patterns than Hopfield networks.
- Hopfield networks require the input patterns to be uncorrelated.
- Boltzmann machines can be stacked to form layers.

3.3.1 Boltzmann Machine Probability

When the program queries the value of 1 of the Boltzmann machine's hidden neurons, it will randomly produce a 0 or 1. Equation 3.6 obtains the calculated probability for that neuron with a value of 1:

$$p_{\text{i=on}} = \frac{1}{1 + \exp(-\frac{\Delta E_i}{T})} \tag{3.6}$$

The above equation will calculate a number between 0 and 1 that represents a probability. For example, if the value 0.75 were generated, the neuron would return a 1 in 75% of the cases. Once it calculates the probability, it can produce the output by generating a random number between 0 and 1 and returning 1 if the random number is below the probability.

The above equation returns the probability for neuron i being on and is calculated with the delta energy (ΔE) at i. The equation also uses the value T, which represents the temperature of the system. Equation 3.2, from earlier in the chapter, can calculate T. The value θ (theta) is the neuron's bias value.

The change in energy is calculated using Equation 3.7:

$$\Delta E_i = \sum_j w_{ij} \, s_j + \theta_i \tag{3.7}$$

This value is the energy difference between 1 (on) and 0 (off) for neuron i. It is calculated using the θ (theta), which represents the bias.

Although the values of the individual neurons are stochastic (random), they will typically fall into equilibrium. To reach this equilibrium, you can repeatedly calculate the network. Each time, a unit is chosen while Equation 3.6 sets its state. After running for an adequate period of time at a certain temperature, the probability of a global state of the network will depend only upon that global state's energy.

In other words, the log probabilities of global states become linear in their energies. This relationship is true when the machine is at thermal equilibrium, which means that the probability distribution of global states has converged. If we start running the network from a high temperature and gradually decrease it until we reach a thermal equilibrium at a low temperature, then we may converge to a distribution where the energy level fluctuates around the global minimum. We call this process simulated annealing.

3.4 Applying the Boltzmann Machine

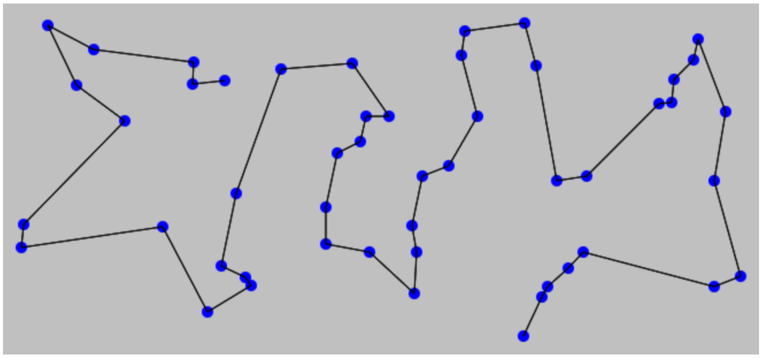
Most research around Boltzmann machines has moved to the restricted Boltzmann machine (RBM) that we will explain in Chapter 9, "Deep Learning." In this section, we will focus on the older, unrestricted form of the Boltzmann, which has been applied to both optimization and recognition problems. We will demonstrate an example of each type, beginning with an optimization problem.

3.4.1 Traveling Salesman Problem

The traveling salesman problem (TSP) is a classic computer science problem that is difficult to solve with traditional programming techniques. Artificial intelligence can be applied to find potential solutions to the TSP. The program must determine the order of a fixed set of cities that minimizes the total distance covered. The traveling salesman is called a combinational problem. If you are already familiar with TSP or you have read about it in a previous volume in this series, you can skip this section.

TSP involves determining the shortest route for a traveling salesman who must visit a certain number of cities. Although he can begin and end in any city, he may visit each city only once. The TSP has several variants, some of which allow multiple visits to cities or assign different values to cities. The TSP in this chapter simply seeks the shortest possible route to visit each city one time. Figure 3.5 shows the TSP problem used here, as well as a potential shortest route:

Figure 3.5: The Traveling Salesman



Finding the shortest route may seem like an easy task for a normal iterative program. However, as the number of cities increases, the number of possible combinations increases drastically. If the problem has one or two cities, only one or two routes are possible. If it includes three cities, the possible routes increase to six. The following list shows how quickly the number of paths grows:

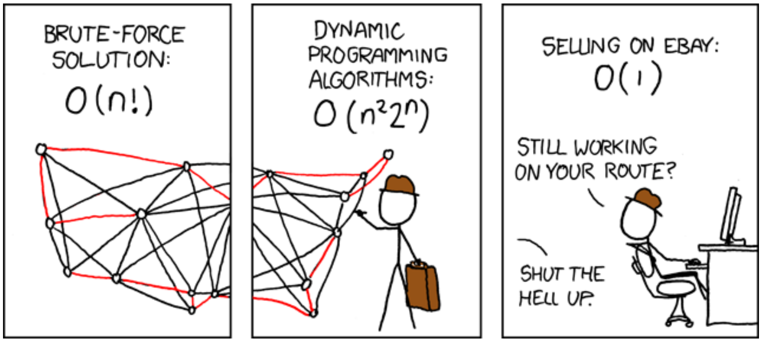
```
1 city has 1 path
2 cities have 2 paths
3 cities have 6 paths
4 cities have 24 paths
5 cities have 120 paths
6 cities have 720 paths
7 cities have 5,040 paths
8 cities have 40,320 paths
9 cities have 362,880 paths
10 cities have 3,628,800 paths
11 cities have 39,916,800 paths
12 cities have 479,001,600 paths
```

```
13 cities have 6,227,020,800 paths
...
50 cities have 3.041 * 10^64 paths
```

In the above table, the formula to calculate total paths is the factorial. The number of cities, *n*, is calculated using the factorial operator (*!*). The factorial of some arbitrary value *n* is given by *n* * (*n* - 1) * (*n* - 2) * ... * 3 * 2 * 1. These values become incredibly large when a program must do a brute-force search. The traveling salesman problem is an example of a non-deterministic polynomial time (NP) hard problem. Informally, NP-hard is defined as any problem that lacks an efficient way to verify a correct solution. The TSP fits this definition for more than 10 cities. You can find a formal definition of NP-hard in *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Garey, 1979).

Dynamic programming is another common approach to the traveling salesman problem, as seen in xkcd.com comic in Figure 3.6:

Figure 3.6: The Traveling Salesman (from xkcd.com)



Although this book does not include a full discussion of dynamic programming, understanding its essential function is valuable. Dynamic programming breaks a large problem, such as the TSP, into smaller problems. You can reuse work for many of the smaller programs, thereby decreasing the amount of iterations required by a brute-force solution.

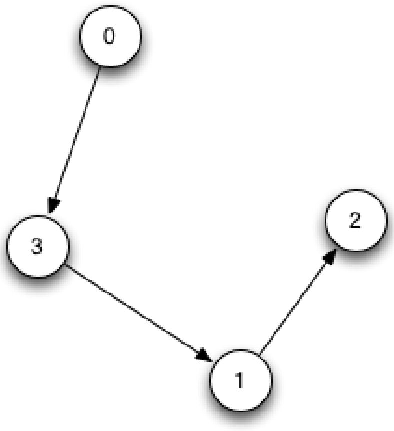
Unlike brute-force solutions and dynamic programming, a genetic algorithm is not guaranteed to find the best solution. Although it will find a good solution, the score might not be the best. The sample program examined in the next section shows how a genetic algorithm produced an acceptable solution for the 50-city problem in a matter of minutes.

3.4.2 Optimization Problems

To use the Boltzmann machine for an optimization problem, it is necessary to represent a TSP solution in such a way that it fits onto the binary neurons of the Boltzmann machine. Hopfield (1984) devised an encoding for the TSP that both Boltzmann and Hopfield neural networks commonly use to represent this combinational problem.

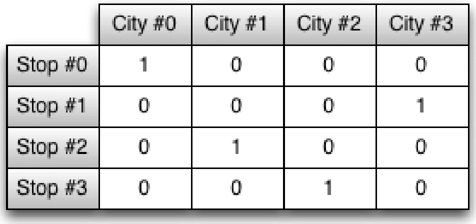
The algorithm arranges the neurons of the Hopfield or Boltzmann machine on a square grid with the number of rows and columns equal to the number of cities. Each column represents a city, and each row corresponds to a segment in the journey. The number of segments in the journey is equal to the number of cities, resulting in a square grid. Each row in the matrix should have exactly one column with a value of 1. This value designates the destination city for each of the trip segments. Consider the city path shown in Figure 3.7:

Figure 3.7: Four Cities to Visit



Because the problem includes four cities, the solution requires a four-by-four grid. The first city visited is City #0. Therefore, the program marks 1 in the first column of the first row. Likewise, visiting City #3 second produces a 1 in the final column of the second row. Figure 3.8 shows the complete path:

Figure 3.8: Encoding of Four Cities



	City #0	City #1	City #2	City #3
Stop #0	1	0	0	0
Stop #1	0	0	0	1
Stop #2	0	1	0	0
Stop #3	0	0	1	0

Of course, the Boltzmann machines do not arrange neurons in a grid. To represent the above path as a vector of values for the neuron, the rows are simply placed sequentially. That is, the matrix is flattened in a row-wise manner, resulting in the following vector:

```
[1 ,0 ,0 ,0 ,    0 ,0 ,0 ,1 ,    0 ,1 ,0 ,0 ,    0 ,0 ,1 ,0]
```

To create a Boltzmann machine that can provide a solution to the TSP, the program must align the weights and biases in such a way that allows the states of the Boltzmann machine neurons to stabilize at a point that minimizes the total distance between cities. Keep in mind that the above grid can also find itself in many invalid states. Therefore, a valid grid must have the following:

- A single 1 value per row.
- A single 1 value per column.

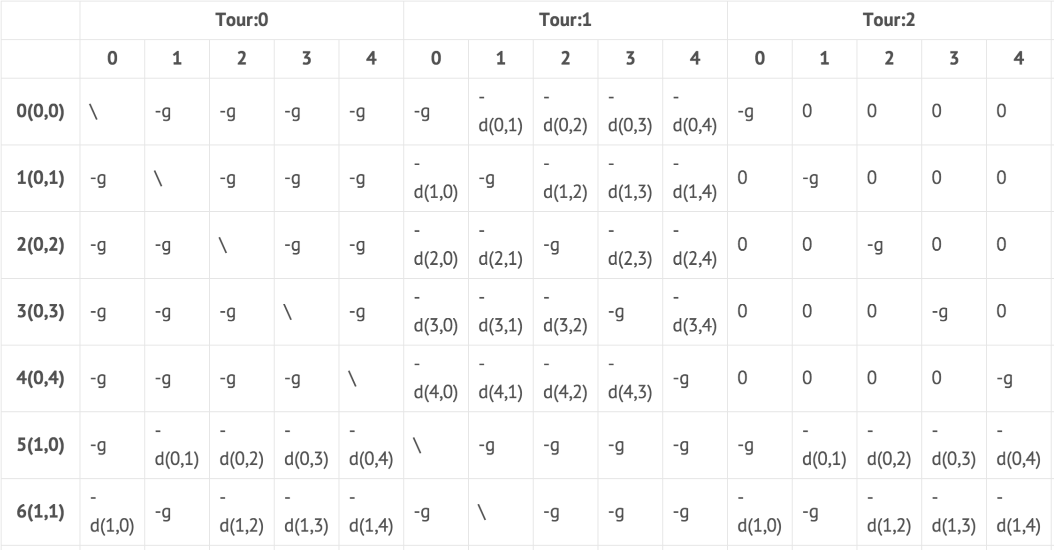
As a result, the program needs to construct the weights so that the Boltzmann machine will not reach equilibrium in an invalid state. Listing 3.5 shows the pseudocode that will generate this weight matrix:

Listing 3.5: Boltzmann Weights for TSP

```
gamma = 7
# Source
for source_tour in range(NUM_CITIES):
  for source_city in range(NUM_CITIES):
    source_index = source_tour * NUM_CITIES + source_city
# Target
    for targetTour in range(NUM_CITIES):
      for (int target_city in range(NUM_CITIES):
        target_index = target_tour * NUM_CITIES + target_city
# Calculate the weight
        weight = 0
# Diagonal weight is 0
        if source_index != target_index:
# Determine the next and previous element in the tour.
# Wrap between 0 and last element.
          prev_target_tour = wrapped next target tour
          next_target_tour = wrapped previous target tour
# If same tour element or city, then -gama
          if (source_tour == target_tour)
             or (source_city == target_city):
            weight = -gamma
# If next or previous city, -gamma
          elif ((source_tour == prev_target_tour)
            or (source_tour == next_target_tour))
            weight = -distance(source_city,target_city)
# Otherwise 0
        set_weight(source_index, target_index, weight)
# All biases are -gamma/2
    set_bias(source_index, -gamma / 2)
```

Figure 3.9 displays part of the created weight matrix for four cities:

Figure 3.9: Boltzmann Machine Weights for TSP (4 cities)



	Tour:0					Tour:1					Tour:2				
	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
0(0,0)	\	-g	-g	-g	-g	-g	-d(0,1)	-d(0,2)	-d(0,3)	-d(0,4)	-g	0	0	0	0
1(0,1)	-g	\	-g	-g	-g	-d(1,0)	-g	-d(1,2)	-d(1,3)	-d(1,4)	0	-g	0	0	0
2(0,2)	-g	-g	\	-g	-g	-d(2,0)	-d(2,1)	-g	-d(2,3)	-d(2,4)	0	0	-g	0	0
3(0,3)	-g	-g	-g	\	-g	-d(3,0)	-d(3,1)	-d(3,2)	-g	-d(3,4)	0	0	0	-g	0
4(0,4)	-g	-g	-g	-g	\	-d(4,0)	-d(4,1)	-d(4,2)	-d(4,3)	-g	0	0	0	0	-g
5(1,0)	-g	-d(0,1)	-d(0,2)	-d(0,3)	-d(0,4)	\	-g	-g	-g	-g	-g	-d(0,1)	-d(0,2)	-d(0,3)	-d(0,4)
6(1,1)	-d(1,0)	-g	-d(1,2)	-d(1,3)	-d(1,4)	-g	\	-g	-g	-g	-d(1,0)	-g	-d(1,2)	-d(1,3)	-d(1,4)

Depending on your viewing device, you might have difficulty reading the above grid. Therefore, you can generate it for any number of cities with the Javascript utility at the following URL:

`http://www.heatonresearch.com/aifh/vol3/boltzmann_tsp_grid.html`

Essentially, the weights have the following specifications:

- Matrix diagonal is assigned to 0. Shown as "\" in Figure 3.9.
- Same source and target position, set to -γ (gamma). Shown as -*g* in Figure 3.9.
- Same source and target city, set to -γ (gamma). Shown as -*g* in Figure 3.9.
- Source and target next/previous cities, set to -*distance*. Shown as *d(x,y)* in Figure 3.9.
- Otherwise, set to 0.

The matrix is symmetrical between the rows and columns.

3.4.3 Boltzmann Machine Training

The previous section showed the use of hard-coded weights to construct a Boltzmann machine that was capable of finding solutions to the TSP. The program constructed these weights through its knowledge of the problem. Manually setting the weights is a necessary and difficult step for applying Boltzmann machines to optimization problems. However, this book will not include information about constructing weight matrices for general optimization problems because Nelder-Mead and simulated annealing are more often used for general-purpose algorithms.

3.5 Chapter Summary

In this chapter, we explained several classic neural network types. Since Pitts (1943) introduced the neural network, many different neural network types have been invented. We have focused primarily on the classic neural network types that still have relevance and that establish the foundation for other architectures that we will cover in later chapters of the book.

The self-organizing map (SOM) is an unsupervised neural network type that can cluster data. The SOM has an input neuron count equal to the number of attributes for the data to be clustered. An output neuron count specifies the number of groups into which the data should be clustered.

The Hopfield neural network is a simple neural network type that can recognize patterns and optimize problems. You must create a special energy function for each type of optimization problem that requires the Hopfield neural network. Because of this quality, programmers choose algorithms like Nelder-Mead or simulated annealing instead of the optimized version of the Hopfield neural network.

The Boltzmann machine is a neural network architecture that shares many characteristics with the Hopfield neural network. However, unlike the Hopfield network, you can stack the deep belief neural network (DBNN). This stacking ability allows the Boltzmann machine to play a central role in the implementation of the deep belief neural network (DBNN), the basis of deep learning.

In the next chapter, we will examine the feedforward neural network, which remains one of the most popular neural network types. This chapter will focus on classic feedforward neural networks that use sigmoid and hyperbolic tangent activation functions. New training algorithms, layer types, activation functions and other innovations allow the classic feedforward neural network to be used with deep learning.

Chapter 4

Feedforward Neural Networks

- Classification
- Regression
- Network Layers
- Normalization

In this chapter, we shall examine one of the most common neural network architectures, the feedforword neural network. Because of its versatility, the feedforward neural network architecture is very popular. Therefore, we will explore how to train it and how it processes a pattern.

The term feedforward describes how this neural network processes and recalls patterns. In a feedforward neural network, each layer of the neural network contains connections to the next layer. For example, these connections extend forward from the input to the hidden layer, but no connections move backward. This arrangement differs from the Hopfield neural network featured in the previous chapter. The Hopfield neural network was fully connected, and its connections were both forward and backward. We will analyze the structure of a feedforward neural network and the way it recalls a pattern later in the chapter.

We can train feedforward neural networks with a variety of techniques from the broad category of backpropagation algorithms, a form of supervised training that we will discuss in greater detail in the next chapter. We will focus on applying optimization algorithms to train the weights of a neural network in this chapter. If you need more information about optimization algorithms, Volumes 1 and 2 of *Artificial Intelligence for Humans* contain sections on this subject. Although we can employ several optimization algorithms to train the weights, we will primarily direct our attention to simulated annealing.

Optimization algorithms adjust a vector of numbers to achieve a good score from an objective function. The objective function gives the neural network a score based closely on the neural network's output that matches the expected output. This score allows any optimization algorithm to train neural networks.

A feedforward neural network is similar to the types of neural networks that we have already examined. Just like other types of neural networks, the feedforward neural network begins with an input layer that may connect to a hidden layer or to the output layer. If it connects to a hidden layer, the hidden layer can subsequently connect to another hidden layer or to the output layer. Any number of hidden layers can exist.

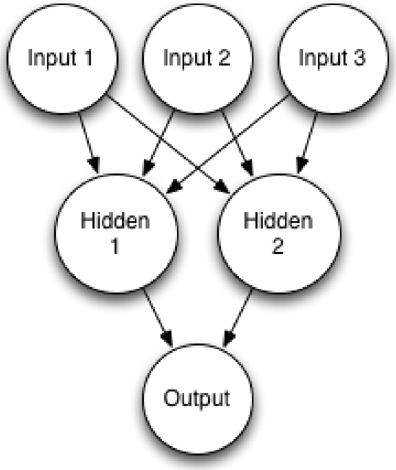
4.1 Feedforward Neural Network Structure

In Chapter 1, "Neural Network Basics," we discussed that neural networks could have multiple hidden layers and analyzed the purposes of these layers. In this chapter, we will focus more on the structure of the input and output neurons, beginning with the structure of the output layer. The type of problem dictates the structure of the output layer. A classification neural network will have an output neuron for each class, whereas a regression neural network will have one output neuron.

4.1.1 Single-Output Neural Networks for Regression

Though feedforward neural networks can have more than one output neuron, we will begin by looking at a single-output neuron network in a regression problem. A regression network is capable of predicting a single numeric value. Figure 4.1 illustrates a single-output feedforward neural network:

Figure 4.1: Single-Output Feedforward Network



This neural network will output a single numeric value. We can use this type of neural network in the following ways:

- **Regression** - Compute a number based on the inputs. (e.g., How many miles per gallon (MPG) will a specific type of car achieve?)
- **Binary Classification** - Decide between two options, based on the inputs. (e.g., Of the given characteristics, which is a cancerous tumor?)

We provide a regression example for this chapter that utilizes data about various car models and predicts the miles per gallon that the car will achieve. You can find this data set at the following URL:

https://archive.ics.uci.edu/ml/datasets/Auto+MPG

A small sampling of this data is shown here:

```
mpg,cylinders,displacement,horsepower,weight,acceleration,
   model_year,origin,car_name
18,8,307,130,3504,12,70,1,”chevrolet chevelle malibu”
15,8,350,165,3693,11,70,1,”buick skylark 320”
18,8,318,150,3436,11,70,1,”plymouth satellite”
16,8,304,150,3433,12,70,1,”amc rebel sst”
```

For a regression problem, the neural network would create columns such as cylinders, displacement, horsepower, and weight to predict the MPG. These values are all fields used in the above listing that specify qualities of each car. In this case, the target is MPG; however, we could also utilize MPG, cylinders, horsepower, weight, and acceleration to predict displacement.

To make the neural network perform regression on multiple values, you might apply multiple output neurons. For example, cylinders, displacement, and horsepower can predict both MPG and weight. Although a multi-output neural network is capable of performing regression on two variables, we don’t recommend this technique. You will usually achieve better results with separate neural networks for each regression outcome that you are trying to predict.

4.2 Calculating the Output

In Chapter 1, "Neural Network Basics," we explored how to calculate the individual neurons that comprise a neural network. As a brief review, the output of an individual neuron is simply the weighted sum of its inputs and a bias. This summation is passed to an activation function. Equation 4.1 summarizes the calculated output of a neural network:

$$f(x_i, w_i) = \phi(\sum_i (w_i \cdot x_i)) \tag{4.1}$$

The neuron multiplies the input vector (x) by the weights (w) and passes the result into an activation function (ϕ, phi). The bias value is the last value in the weight vector (w), and it is added by concatenating a 1 value to the input. For example, consider a neuron that has two inputs and a bias. If the inputs were 0.1 and 0.2, the input vector would appear as follows:

```
[0.1, 0.2, 1.0]
```

In this example, add the value 1.0 to support the bias weight. We can also calculate the value with the following weight vector:

```
[0.01, 0.02, 0.3]
```

The values 0.01 and 0.02 are the weights for the two inputs to the neuron. The value 0.3 is the bias. The weighted sum is calculated as follows:

```
(0.1*0.01) + (0.2*0.02) + (1.0*0.3) = 0.305
```

The value 0.305 is then passed to an activation function.

Calculating an entire neural network is essentially a matter of following this same procedure for each neuron in the network. This process allows you to work your way from the input neurons to the output. You can implement this process by creating objects for each connection in the network or by aligning these connection values into matrices.

Object-oriented programming allows you to define an object for each neuron and its weights. This approach can produce very readable code, but it has two significant problems:

- The weights are stored across many objects.
- Performance suffers because it takes many function calls and memory accesses to piece all the weights together.

It is valuable to create weights in the neural network as a single vector. A variety of different optimization algorithms can adjust a vector to perfect a scoring function. *Artificial Intelligence for Humans, Volumes 1* & 2 include a discussion of these optimization functions. Later in this chapter, we will see how simulated annealing optimizes the weight vector for the neural network.

To construct a weight vector, we will first look at a network that has the following attributes:

- Input Layer: 2 neurons, 1 bias
- Hidden Layer: 2 neurons, 1 bias
- Output Layer: 1 neuron

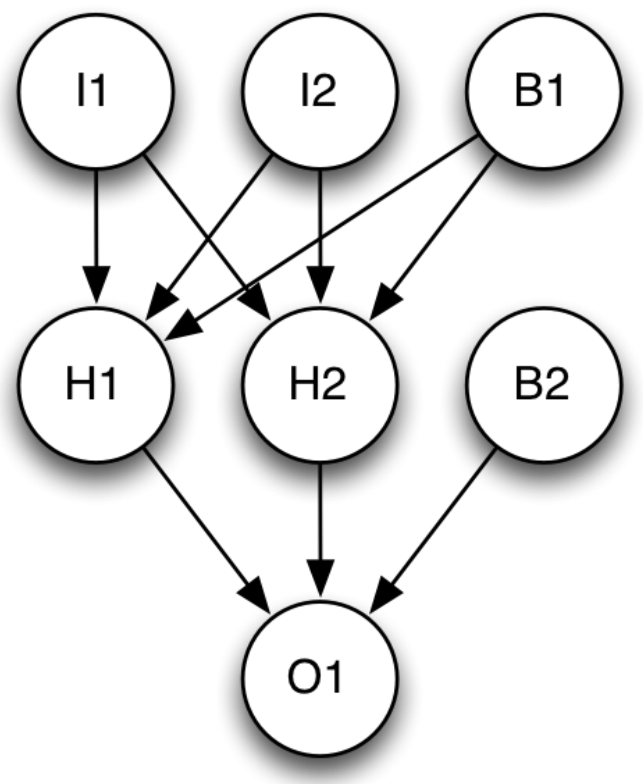
These characteristics give this network a total of 7 neurons.

You can number these neurons for the vector in the following manner:

```
Neuron 0: Output 1
Neuron 1: Hidden 1
Neuron 2: Hidden 2
Neuron 3: Bias 2 (set to 1, usually)
Neuron 4: Input 1
Neuron 5: Input 2
Neuron 6: Bias 1 (set to 1, usually)
```

Graphically, you can see the network as Figure 4.2:

Figure 4.2: Simple Neural Network



You can create several additional vectors to define the structure of the network. These vectors hold index values to allow the quick navigation of the weight vector. These vectors are listed here:

```
layerFeedCounts: [1, 2, 2]
layerCounts: [1, 3, 3]
layerIndex: [0, 1, 4]
layerOutput: [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0]
weightIndex: [0, 3, 9]
```

Each vector stores the values for the output layer first and works its way to the input layer. The *layerFeedCounts* vector holds the count of non-bias neurons in each layer. This characteristic is essentially the count of non-bias neurons. The *layerOutput* vector holds the current value of each neuron. Initially, all neurons start with 0.0 except for the bias neurons, which start at 1.0. The *layerIndex* vector holds indexes to where each layer begins in the *layerOuput* vector. The *weightIndex* holds indexes to the location of each layer in the weight vector.

The weights are stored in their own vector and structured as follows:

```
Weight 0: H1->O1
Weight 1: H2->O1
Weight 2: B2->O1
Weight 3: I1->H1
Weight 4: I2->H1
Weight 5: B1->H1
Weight 6: I1->H2
Weight 7: I2->H2
Weight 8: B1->H2
```

Once the vectors have been arranged, calculating the output of the neural network is relatively easy. Listing 4.1 can accomplish this calculation:

Listing 4.1: Calculate Feedforward Output

```python
def compute(net, input):
  sourceIndex = len(net.layerOutput)
    - net.layerCounts[len(net.layerCounts) - 1]
  # Copy the input into the layerOutput vector
  array_copy(input, 0, net.layerOutput, sourceIndex, net.
      inputCount)
  # Calculate each layer
  for i in reversed(range(0,len(layerIndex))):
    compute_layer(i)
  # update context values
  offset = net.contextTargetOffset[0]
  # Create result
  result = vector(net.outputCount)
  array_copy(net.layerOutput, 0, result, 0, net.outputCount)
  return result
```

```
def compute_layer(net,currentLayer):
  inputIndex = net.layerIndex[currentLayer]
  outputIndex = net.layerIndex[currentLayer - 1]
  inputSize = net.layerCounts[currentLayer]
  outputSize = net.layerFeedCounts[currentLayer - 1]
  index = this.weightIndex[currentLayer - 1]
  limit_x = outputIndex + outputSize
  limit_y = inputIndex + inputSize
  # weight values
  for x in range(outputIndex,limit_x):
    sum = 0;
    for y in range(inputIndex,limit_y):
      sum += net.weights[index] * net.layerOutput[y]
        net.layerSums[x] = sum
        net.layerOutput[x] = sum
        index = index + 1

  net.activationFunctions[currentLayer - 1]
     .activation_function(
 net.layerOutput, outputIndex, outputSize)
```

4.3 Initializing Weights

The weights of a neural network determine the output for the neural network. The process of training can adjust these weights so the neural network produces useful output. Most neural network training algorithms begin by initializing the weights to a random state. Training then progresses through a series of iterations that continuously improve the weights to produce better output.

The random weights of a neural network impact how well that neural network can be trained. If a neural network fails to train, you can remedy the problem by simply restarting with a new set of random weights. However, this solution can be frustrating when you are experimenting with the architecture of a neural network and trying different combinations of hidden layers and neurons. If you add a new layer, and the network's performance improves, you must ask yourself if this improvement resulted from the new layer or from a new set of weights. Because of this uncertainty, we look for two key attributes in a weight initialization algorithm:

- How consistently does this algorithm provide good weights?
- How much of an advantage do the weights of the algorithm provide?

One of the most common, yet least effective, approaches to weight initialization is to set the weights to random values within a specific range. Numbers between -1 and +1 or -5 and +5 are often the choice. If you want to ensure that you get the same set of random weights each time, you should use a seed. The seed specifies a set of predefined random weights to use. For example, a seed of 1000 might produce random weights of 0.5, 0.75, and 0.2. These values are still random; you cannot predict them, yet you will always get these values when you choose a seed of 1000.

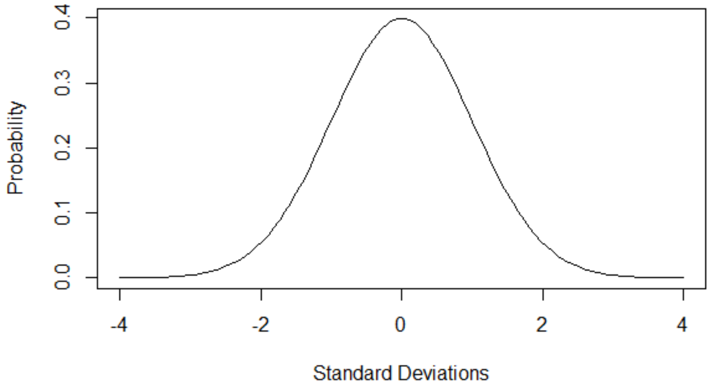
Not all seeds are created equal. One problem with random weight initialization is that the random weights created by some seeds are much more difficult to train than others. In fact, the weights can be so bad that training is impossible. If you find that you cannot train a neural network with a particular weight set, you should generate a new set of weights using a different seed.

Because weight initialization is a problem, there has been considerable research around it. Over the years we have studied this research and added six different weight initialization routines to the Encog project. From our research, the Xavier weight initialization algorithm, introduced in 2006 by Glorot & Bengio, produces good weights with reasonable consistency. This relatively simple algorithm uses normally distributed random numbers.

To use the Xavier weight initialization, it is necessary to understand that normally distributed random numbers are not the typical random numbers between 0 and 1 that most programming languages generate. In fact, normally distributed random numbers are centered on a mean (μ, mu) that is typically 0. If 0 is the center (mean), then you will get an equal number of random numbers above and below 0. The next question is how far these random numbers will venture from 0. In theory, you could end up with both positive and negative numbers close to the maximum positive and negative ranges supported by your computer. However, the reality is that you will more likely see random numbers that are between 0 and three standard deviations from the center.

The standard deviation σ (sigma) parameter specifies the size of this standard deviation. For example, if you specified a standard deviation of 10, then you would mainly see random numbers between -30 and +30, and the numbers nearer to 0 have a much higher probability of being selected. Figure 4.3 shows the normal distribution:

Figure 4.3: The Normal Distribution



The above figure illustrates that the center, which in this case is 0, will be generated with a 0.4 (40%) probability. Additionally, the probability decreases very quickly beyond -2 or +2 standard deviations. By defining the center and how large the standard deviations are, you are able to control the range of random numbers that you will receive.

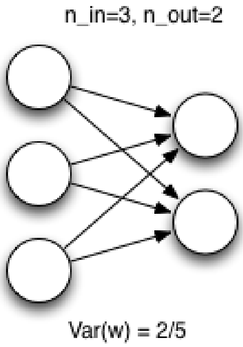
Most programming languages have the capability of generating normally distributed random numbers. In general, the Box-Muller algorithm is the basis for this functionality. The examples in this volume will either use the built-in normal random number generator or the Box-Muller algorithm to transform regular, uniformly distributed random numbers into a normal distribution. *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms* contains an explanation of the Box-Muller algorithm, but you do not necessarily need to understand it in order to grasp the ideas in this book.

The Xavier weight initialization sets all of the weights to normally distributed random numbers. These weights are always centered at 0; however, their standard deviation varies depending on how many connections are present for the current layer of weights. Specifically, Equation 4.2 can determine the standard deviation:

$$Var(W) = \frac{2}{n_{in} + n_{out}} \tag{4.2}$$

The above equation shows how to obtain the variance for all of the weights. The square root of the variance is the standard deviation. Most random number generators accept a standard deviation rather than a variance. As a result, you usually need to take the square root of the above equation. Figure 4.4 shows how one layer might be initialized:

Figure 4.4: Xavier Initialization of a Layer



This process is completed for each layer in the neural network.

4.4 Radial-Basis Function Networks

Radial-basis function (RBF) networks are a type of feedforward neural network introduced by Broomhead and Lowe (1988). These networks can be used for both classification and regression. Though they can solve a variety of problems, RBF networks seem to losing popularity. By their very definition, RBF networks cannot be used in conjunction with deep learning.

The RBF network utilizes a parameter vector, a model that specifies weights and coefficients, in order to allow the input to generate the correct output. By adjusting a random parameter vector, the RBF network produces output consistent with the iris data set. The process of adjusting the parameter vector to produce the desired output is called training. Many different methods exist for training an RBF network. The parameter vectors also represent its long-term memory.

In the next section, we will briefly review RBFs and describe the exact makeup of these vectors.

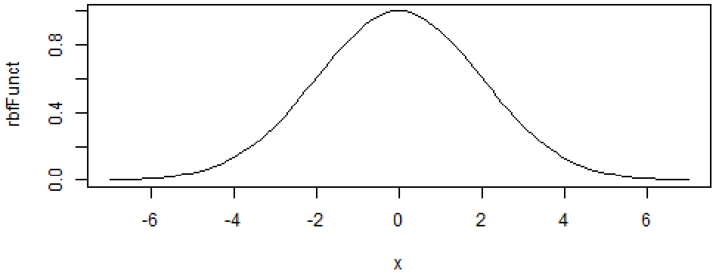
4.4.1 Radial-Basis Functions

Because many AI algorithms utilize radial-basis functions, they are a very important concept to understand. A radial-basis function is symmetric with respect to its center, which is usually somewhere along the x-axis. The RBF will reach its maximum value or peak at the center. Whereas a typical setting for the peak in RBF networks is 1, the center varies accordingly.

RBFs can have many dimensions. Regardless of the number of dimensions in the vector passed to the RBF, its output will always be a single scalar value.

RBFs are quite common in AI. We will start with the most prevalent, the Gaussian function. Figure 4.5 shows a graph of a 1D Gaussian function centered at 0:

Figure 4.5: Gaussian Function



You might recognize the above curve as a normal distribution or a bell curve, which is a radial-basis function. The RBFs, such as a Gaussian function, can selectively scale numeric values. Consider Figure 4.5 above. If you applied this function to scale numeric values, the result would have maximum intensity at the center. As you moved from the center, the intensity would diminish in either the positive or negative directions.

Before we can look at the equation for the Gaussian RBF, we must consider how to process the multiple dimensions. RBFs accept multi-dimensional input and return a single value by calculating the distance between the input and the center vector. This distance is called r. The RBF center and input to the RBF must always have the same number of dimensions for the calculation to occur. Once we calculate r, we can determine the individual RBF. All of the RBFs use this calculated r.

Equation 4.3 shows how to calculate r:

$$r = ||\mathbf{x} - \mathbf{x}_i|| \tag{4.3}$$

The double vertical bars that you see in the above equation signify that the function describes a distance or a norm. In certain cases, these distances can vary; however, RBFs typically utilize Euclidean distance. As a result, the examples that we provide in this book always apply the Euclidean distance. Therefore, r is simply the Euclidean distance between the center and the x vector. In each of the RBFs in this section, we will use this value r. Equation

4.4 shows the equation for a Gaussian RBF:

$$\phi(r) = e^{-r^2} \tag{4.4}$$

Once you've calculated r, determining the RBF is fairly easy. The Greek letter ϕ, which you see at the left of the equation, always represents the RBF. The constant e in Equation 4.4 represents Euler's number, or the natural base, and is approximately 2.71828.

4.4.2 Radial-Basis Function Networks

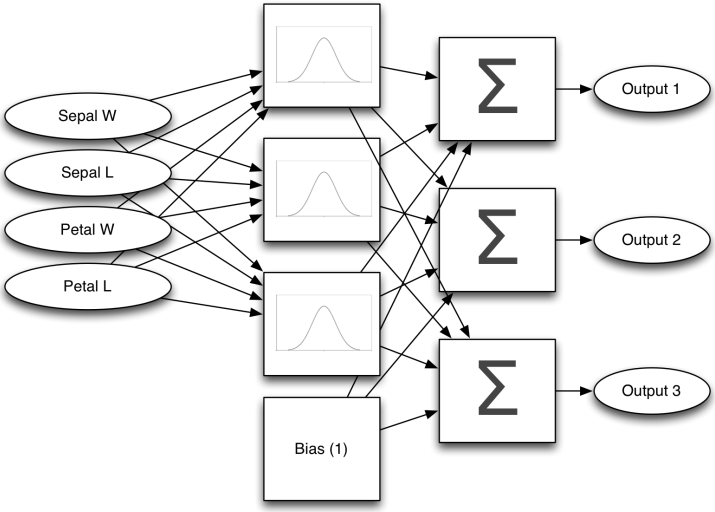
RBF networks provide a weighted summation of one or more radial-basis functions; each of these functions receives the weighted input attributes in order to predict the output. Consider the RBF network as a long equation that contains the parameter vector. Equation 4.5 shows the equation needed to calculate the output of this network:

$$f(X) = \sum_{i=1}^{N} a_i p(||b_i X - c_i||) \tag{4.5}$$

Note that the double vertical bars in the above equation signify that you must calculate the distance. Because these symbols do not specify which distance algorithm to use, you can select the algorithm. In the above equation, X is the input vector of attributes; c is the vector center of the RBF; p is the chosen RBF (Gaussian, for example); a is the vector coefficient (or weight) for each RBF; and b specifies the vector coefficient to weight the input attributes.

In our example, we will apply an RBF network to the iris data set. Figure 4.6 provides a graphic representation of this application:

Figure 4.6: The RBF Network for the Iris Data



The above network contains four inputs (the length and width of petals and sepals) that indicate the features that describe each iris species. The above diagram assumes that we are using one-of-n encoding for the three different iris species. Using equilateral encoding for only two outputs is also possible. To keep things simple, we will use one-of-n and arbitrarily choose three RBFs. Even though additional RBFs allow the model to learn more complex data sets, they require more time to process.

Arrows represent all coefficients from the equation. In Equation 4.5, b represents the arrows between the input attributes and the RBFs. Similarly, a represents the arrows between the RBFs and the summation. Notice also the bias box, which is a synthetic function that always returns a value of 1. Because the bias function's output is constant, the program does not require inputs. The weights from the bias to the summation specify the y-intercept for the equation. In short, bias is not always bad. This case demonstrates that bias is an important component to the RBF network. Bias nodes are also very common in neural networks.

Because multiple summations exist, you can see the development of a classification problem. The highest summation specifies the predicted class. A regression problem indicates that the model will output a single numeric value.

You will also notice that Figure 4.4 contains a bias node in the place of an additional RBF. Unlike the RBF, the bias node does not accept any input. It always outputs a constant value of 1. Of course, this constant value of 1 is multiplied by a coefficient value, which always causes the coefficient to be directly added to the output, regardless of the input. When the input is 0, bias nodes are very useful because they allow the RBF layer to output values despite the low value of the input.

The long-term memory vector for the RBF network has several different components:

- Input coefficients
- Output/Summation coefficients
- RBF width scalars (same width in all dimensions)
- RBF center vectors

The RBF network will store all of these components as a single vector that will become its long-term memory. Then an optimization algorithm can set the vector to values that will produce the correct iris species for the features presented. This book contains several optimization algorithms that can train an RBF network.

In conclusion, this introduction provided a basic overview of vectors, distance, and RBF networks. Since this discussion included only the prerequisite material to understand Volume 3, refer to Volumes 1 and 2 for a more thorough explanation of these topics.

4.5 Normalizing Data

Normalization was briefly mentioned previously in this book. In this section, we will see exactly how it is performed. Data are not usually presented to the neural network in exactly the same raw form as you found it. Usually data are scaled to a specific range in a process called normalization. There are many different ways to normalize data. For a full summary, refer to *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms.* This chapter will present a few normalization methods most useful for neural networks.

4.5.1 One-of-N Encoding

If you have a categorical value, such as the species of an iris, the make of an automobile, or the digit label in the MNIST data set, you should use one-of-*n* encoding. This type of encoding is sometimes referred to as one-hot encoding. To encode in this way, you would use one output neuron for each class in the problem. Recall the MNSIT data set from the book's introduction, where you have images for digits between 0 and 9. This problem is most commonly encoded as ten output neurons with a softmax activation function that gives the probability of the input being one of these digits. Using one-of-n encoding, the ten digits might be encoded as follows:

```
0 -> [1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0]
1 -> [0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0]
2 -> [0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0]
3 -> [0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0]
4 -> [0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0]
5 -> [0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0]
6 -> [0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0]
7 -> [0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0]
8 -> [0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0]
0 -> [0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1]
```

One-of-n encoding should always be used when the classes have no ordering. Another example of this type of encoding is the make of an automobile. Usually the list of automakers is unordered unless there is some meaning you wish to convey by this ordering. For example, you might order the automakers by the number of years in business. However, this classification should only be done if the number of years in business has meaning to your problem. If there is truly no order, then one-of-n should always be used.

Because you can easily order the digits, you might wonder why we use one-of-n encoding for them. However, the order of the digits does not mean the program can recognize them. The fact that "1" and "2" are numerically next to each other does nothing to help the program recognize the image. Therefore, we should not use a single-output neuron that simply outputs the digit recognized. The digits 0-9 are categories, not actual numeric values. Encoding categories with a single numeric value is detrimental to the neural network's decisions process.

Both the input and output can use one-of-n encoding. The above listing used 0's and 1's. Normally you will use the rectified linear unit (ReLU) and softmax activation, and this type of encoding is normal. However, if you are working with a hyperbolic tangent activation function, you should utilize a value of -1 for the 0's to match the hyperbolic tangent's range of -1 to 1.

If you have an extremely large number of classes, one-of-n encoding can become cumbersome because you must have a neuron for every class. In such cases, you have several options. First, you might find a way to order your categories. With this ordering, your categories can now be encoded as a numeric value, which would be the current category's position within the ordered list.

Another approach to dealing with an extremely large number of categories is frequency-inverse document frequency (TF-IDF) encoding because each class essentially becomes the probability of that class's occurrence relative to the others. In this way, TF-IDF allows the program to map a large number of classes to a single neuron. A complete discussion of TF-IDF is beyond the scope of this book; however, it is built into many machine learning frameworks for languages such as R, Python, and some others.

4.5.2 Range Normalization

If you have a real number or an ordered list of categories, you might choose range normalization because it simply maps the input data's range into the range of your activation function. Sigmoid, ReLU and softmax use a range between 0 and 1, whereas hyperbolic tangent uses a range between -1 and 1.

You can normalize a number with Equation 4.6:

$$norm(x, d_L, d_H, n_L, n_H) = \frac{(x-d_L)(n_H-n_L)}{(d_H-d_L)} + n_L \tag{4.6}$$

To perform the normalization, you need the high and low values of the data to be normalized, given by *dl* and *dh* in the equation above. Similarly, you need the high and low values to normalize into (usually 0 and 1), given by *nl* and *nh*.

Sometimes you will need to undo the normalization performed on a number and return it to a denormalized state. Equation 4.7 performs this operation:

$$denorm(x, d_L, d_H, n_L, n_H) = \frac{(d_L - d_H)x - (n_H \cdot d_L) + d_H \cdot n_L}{(n_L - n_H)} \tag{4.7}$$

A very simple way to think of range normalization is percentages. Consider the following analogy. You see an advertisement stating that you will receive a $10 (USD) reduction on a product, and you have to decide if this deal is worthwhile. If you are buying a t-shirt, this offer is probably a good deal; however, if you are buying a car, $10 does not really matter. Furthermore, you need to be familiar with the current value of US dollars in order to make your decision. The situation changes if you learn that the merchant had offered a 10% discount. Thus, the value is now more meaningful. No matter if you are buying a t-shirt, car or even a house, the 10% discount has clear ramifications on the problem because it transcends currencies. In other words, the percentage is a type of normalization. Just like in the analogy, normalizing to a range helps the neural network evaluate all inputs with equal significance.

4.5.3 Z-Score Normalization

Z-score normalization is the most common normalization for either a real number or an ordered list. For nearly all applications, z-score normalization should be used in place of range normalization. This normalization type is based on the statistical concept of z-scores, the same technique for grading exams on a curve. Z-scores provide even more information than percentages.

Consider the following example. Student A scored 85% of the points on her exam. Student B scored 75% of the points on his exam. Which student earned the better grade? If the professor is simply reporting the percentage of correct points, then student A earned a better score. However, you might change your answer if you learned that the average (mean) score for student A's very easy exam was 95%. Similarly, you might reconsider your position if you discovered that student B's class had an average score of 65%. Student B performed above average on his exam. Even though student A earned a better score, she performed below average. To truly report a curved score (a z-score) you must have the mean score and the standard deviation. Equation 4.8 shows the calculation of a mean:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.8}$$

You can calculate the mean (μ, mu) by adding all of the scores and dividing by the number of scores. This process is the same as taking an average. Now that you have the average, you need the standard deviation. If you had a mean score of 50 points, then everyone taking the exam varied from the mean by some amount. The average amount that students varied from the mean is essentially the standard deviation. Equation 4.9 shows the calculation of the standard deviation (σ, sigma):

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{4.9}$$

Essentially, the process of taking a standard deviation is squaring and summing each score's difference from the mean. These values are added together and the square root is taken of this total. Now that you have the standard deviation, you can calculate the z-score with Equation 4.10:

$$z = \frac{x - \mu}{\sigma} \tag{4.10}$$

Listing 4.2 shows the pseudocode needed to calculate a z-score:

Listing 4.2: Calculate a Z-Score

```
# Data to score:
data = [ 5, 10, 3, 20, 4]
# Sum the values
sum = 0
for d in data:
    sum = sum + d
# Calculate mean
mean = float(sum) / len(data)
print( ”Mean: ” + mean )
# Calculate the variance
variance = 0
for d in data:
    variance = variance + ((mean-d)**2)
variance = variance / len(data)
```

```
print( ”Variance: ” + variance )
# Calculate the standard deviation
sdev = sqrt(variance)
print( ”Standard Deviation: ” + sdev )
# Calculate zscore
zscore = []
for d in data:
    zscore.append( (d–mean)/sdev)
print(”Z–Scores: ” + str(zscore) )
```

The above code will result in the following output:

```
Mean: 8.4
Variance: 39.440000000000005
Standard Deviation: 6.280127387243033
Z–Scores: [−0.5413902920037097, 0.2547719021193927,
    −0.8598551696529507, 1.8470962903655976, −0.7006227308283302]
```

The z-score is a numeric value where 0 represents a score that is exactly the mean. A positive z-score is above average; a negative z-score is below average. To help visualize z-scores, consider the following mapping between z-scores and letter grades:

```
<−2.0 = D+
−2.0  = C–
−1.5  = C
−1.0  = C+
−0.5  = B–
0.0   = B
+0.5  = B+
+1.0  = A–
+1.5  = A
+2.0  = A+
```

We took the mapping listed above from an undergraduate syllabus. There is a great deal of variation on z-score to letter grade mapping. Most professors will set the 0.0 z-score to either a C or a B, depending on if the professor/university considers C or B to represent an average grade. The above professor considered B to be average. The z-score works well for neural network input as it is centered at 0 and will very rarely go above +3 and below -3.

4.5.4 Complex Normalization

The input to a neural network is commonly called its feature vector. The process of creating a feature vector is critical to mapping your raw data to a form that the neural network can comprehend. The process of mapping the raw data to a feature vector is called encoding. To see this mapping at work, consider the auto MPG data set:

```
1. mpg:           numeric
2. cylinders:     numeric, 3 unique
3. displacement:  numeric
4. horsepower:    numeric
5. weight:        numeric
6. acceleration:  numeric
7. model year:    numeric, 3 unique
8. origin:        numeric, 7 unique
9. car name:      string (unique for each instance)
```

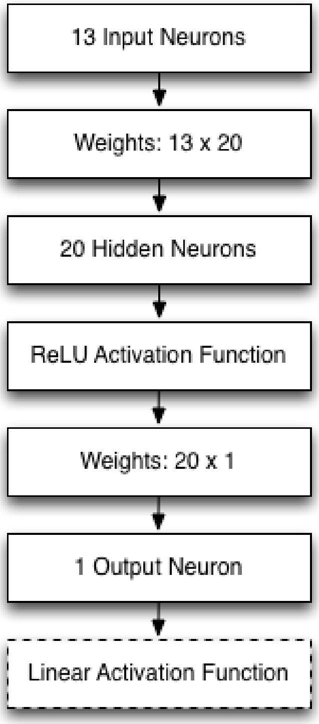
To encode the above data, we will use MPG as the output and treat the data set as regression. The MPG feature will be z-score encoded, and it falls within the range of the linear activation function that we will use on the output.

We will discard the car name. Cylinders and model-year are both one-of-n encoded, the remaining fields will be z-score encoded. The following feature vector results:

```
Input Feature Vector:
Feature 1: cylinders-2, -1 no, +1 yes
Feature 2: cylinders-4, -1 no, +1 yes
Feature 3: cylinders-8, -1 no, +1 yes
Feature 4: displacement z-score
Feature 5: horsepower z-score
Feature 6: weight z-score
Feature 7: acceleration z-score
Feature 8: model year-1977, -1 no, +1 yes
Feature 9: model year-1978, -1 no, +1 yes
Feature 10: model year-1979, -1 no, +1 yes
Feature 11: origin-1
Feature 12: origin-2
Feature 13: origin-3
Output:
mpg z-score
```

As you can see, the feature vector has grown from the nine raw fields to thirteen features plus an output. A neural network for these data would have thirteen input neurons and a single output. Assuming a single-hidden layer of twenty neurons with the ReLU activation, this network would look like Figure 4.7:

Figure 4.7: Simple Regression Neural Network



4.6 Chapter Summary

Feedforward neural networks are one of the most common algorithms in artificial intelligence. In this chapter, we introduced the multilayer feedforward neural network and the radial-basis function (RBF) neural network. Classification and regression apply both of these types of neural network.

Feedforward networks have well-defined layers. The input layer accepts the input from the computer program. The output layer returns the processing result of the neural network to the calling program. Between these layers are hidden neurons that help the neural network to recognize a pattern presented at the input layer and produce the correct result on the output layer.

RBF neural networks use a series of radial-basis functions for their hidden layer. In addition to the weights, it is also possible to change the widths and centers of these RBFs. Though an RBF and feedforward network can approximate any function, they go about the process in different ways.

So far, we've seen only how to calculate the values for neural networks. Training is the process by which we adjust the weights of neural networks so that the neural network outputs the values that we desire. To train neural networks, we also need to have a way to evaluate it. The next chapter introduces both training and validation of neural networks.

Chapter 5

Training & Evaluation

- Mean Squared Error
- Sensitivity & Specificity
- ROC Curve
- Simulated Annealing

So far we've seen how to calculate a neural network based on its weights; however, we have not seen where these weight values actually come from. Training is the process where a neural network's weights are adjusted to produce the desired output. Training uses evaluation, which is the process where the output of the neural network is evaluated against the expected output.

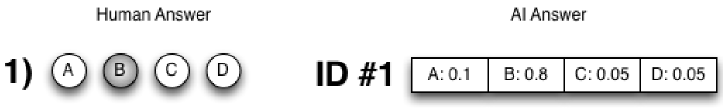
This chapter will cover evaluation and introduce training. Because neural networks can be trained and evaluated in many different ways, we need a consistent method to judge them. An objective function evaluates a neural network and returns a score. Training adjusts the neural network in ways that might achieve better results. Typically, the objective function wants lower scores. The process of attempting to achieve lower scores is called minimization. You might establish maximization problems, in which the objective function wants higher scores. Therefore, you can use most training algorithms for either minimization or maximization problems.

You can optimize weights of a neural network with any continuous optimization algorithm, such as simulated annealing, particle swarm optimization, genetic algorithms, hill climbing, Nelder-Mead, or random walk. In this chapter, we will introduce simulated annealing as a simple training algorithm. However, in addition to optimization algorithms, you can train neural networks with backpropagation. Chapter 6, “Backpropagation Training,” and Chapter 7, “Other Propagation Training,” will introduce several algorithms that were based on the backpropagation training algorithms introduced in Chapter 6.

5.1 Evaluating Classification

Classification is the process by which a neural network attempts to classify the input into one or more classes. The simplest way of evaluating a classification network is to track the percentage of training set items that were classified incorrectly. We typically score human examples in this manner. For example, you might have taken multiple-choice exams in school in which you had to shade in a bubble for choices A, B, C, or D. If you chose the wrong letter on a 10-question exam, you would earn a 90%. In the same way, we can grade computers; however, most classification algorithms do not simply choose A, B, C, or D. Computers typically report a classification as their percent confidence in each class. Figure 5.1 shows how a computer and a human might both respond to question #1 on an exam:

Figure 5.1: Human Exam versus Computer Classification



As you can see, the human test taker marked the first question as “B.” However, the computer test taker had an 80% (0.8) confidence in “B” and was also somewhat sure with 10% (0.1) on “A.” The computer then distributed the remaining points on the other two. In the simplest sense, the machine would get 80% of the score for this question if the correct answer were “B.”

The machine would get only 5% (0.05) of the points if the correct answer were "D."

5.1.1 Binary Classification

Binary classification occurs when a neural network must choose between two options, which might be true/false, yes/no, correct/incorrect, or buy/sell. To see how to use binary classification, we will consider a classification system for a credit card company. This classification system must decide how to respond to a new potential customer. This system will either "issue a credit card" or "decline a credit card."

When you have only two classes that you can consider, the objective function's score is the number of false positive predictions versus the number of false negatives. False negatives and false positives are both types of errors, and it is important to understand the difference. For the previous example, issuing a credit card would be the positive. A false positive occurs when a credit card is issued to someone who will become a bad credit risk. A false negative happens when a credit card is declined to someone who would have been a good risk.

Because only two options exist, we can choose the mistake that is the more serious type of error, a false positive or a false negative. For most banks issuing credit cards, a false positive is worse than a false negative. Declining a potentially good credit card holder is better than accepting a credit card holder who would cause the bank to undertake expensive collection activities.

A classification problem seeks to assign the input into one or more categories. A binary classification employs a single-output neural network to classify into two categories. Consider the auto MPG data set that is available from the University of California at Irvine (UCI) machine learning repository at the following URL:

`https://archive.ics.uci.edu/ml/datasets/Auto+MPG`

For the auto MPG data set, we might create classifications for cars built inside of the United States. The field named *origin* provides information on the location of the car assembly. Thus, the single-output neuron would give a number that indicates the probability that the car was built in the USA.

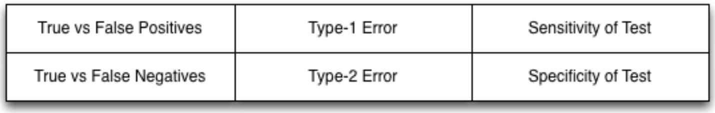
To perform this prediction, you need to change the *origin* field to hold values between 1 and the low-end range of the activation function. For example, the low end of the range for the sigmoid function is 0; for the hyperbolic tangent, it is -1. The neural network will output a value that indicates the probability of a car being made in the USA or elsewhere. Values closer to 1 indicate a higher probability of the car originating in the USA; values closer to 0 or -1 indicate a car originating from outside the USA.

You must choose a cutoff value that differentiates these predictions into either USA or non-USA. If USA is 1.0 and non-USA is 0.0, we could just choose 0.5 as the cutoff value. Consequently, a car with an output of 0.6 would be USA, and 0.4 would be non-USA.

Invariably, this neural network will produce errors as it classifies cars. A USA-made car might yield an output of 0.45; however, because the neural network is below the cutoff value, it would not put the car in the correct category. Because we designed this neural network to classify USA-made cars, this error would be called a false negative. In other words, the neural network indicated that the car was non-USA, creating a negative result because the car was actually from the USA. Thus, the negative classification was false. This error is also known as a type-2 error.

Similarly, the network might falsely classify a non-USA car as USA. This error is a false positive, or a type-1. Neural networks prone to produce false positives are characterized as more specific. Similarly, neural networks that produce more false negatives are labeled as more sensitive. Figure 5.2 summarizes these relationships between true/false, positives/negatives, type-1 & type-2 errors, and sensitivity/specificity:

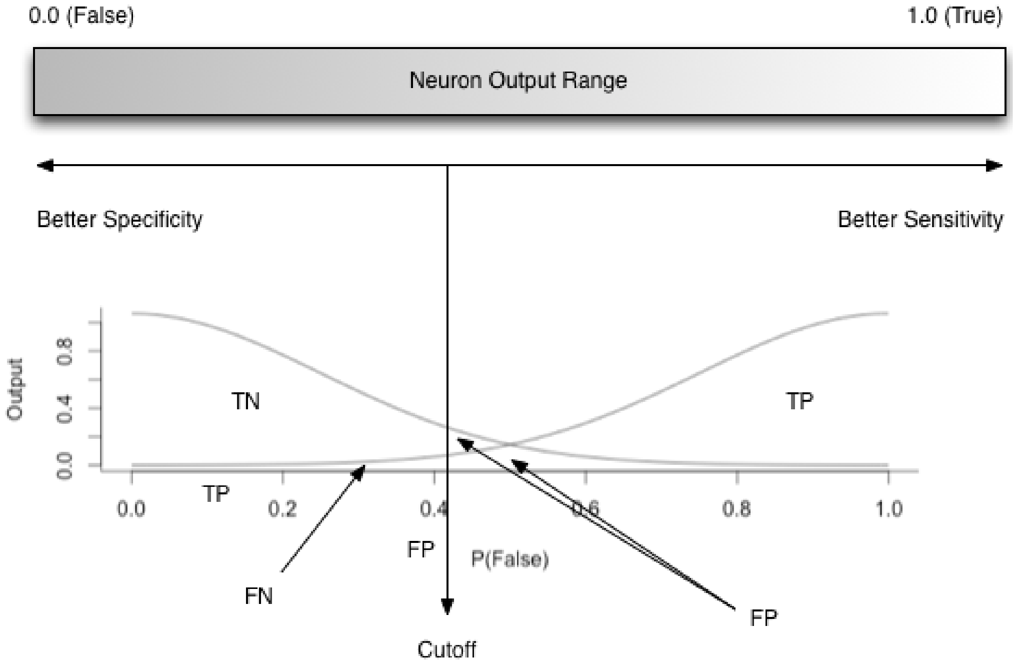
Figure 5.2: Types of Errors



True vs False Positives	Type-1 Error	Sensitivity of Test
True vs False Negatives	Type-2 Error	Specificity of Test

Setting the cutoff for the output neuron selects whether sensitivity or specificity is more important. It is possible to make a neural network more sensitive or specific by adjusting this cutoff, as illustrated in Figure 5.3:

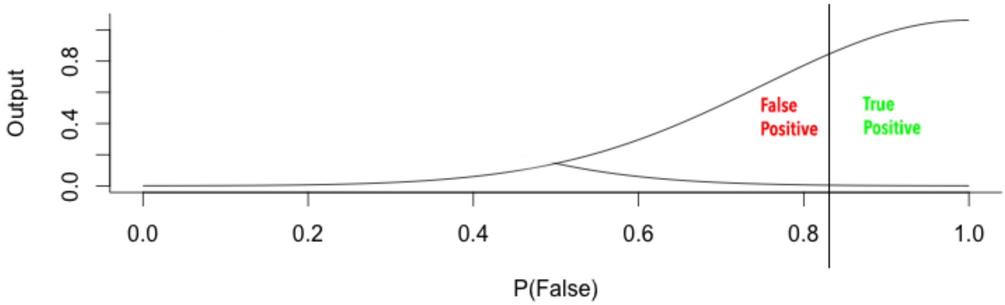
Figure 5.3: Sensitivity vs. Specificity



As the limit line moves left, the network becomes more specific. The decrease in the size of the true negative (TN) area makes this specificity evident. Conversely, as the limit line moves right, the network becomes more sensitive. This sensitivity is evident in the decrease in size of the true positive (TP) area.

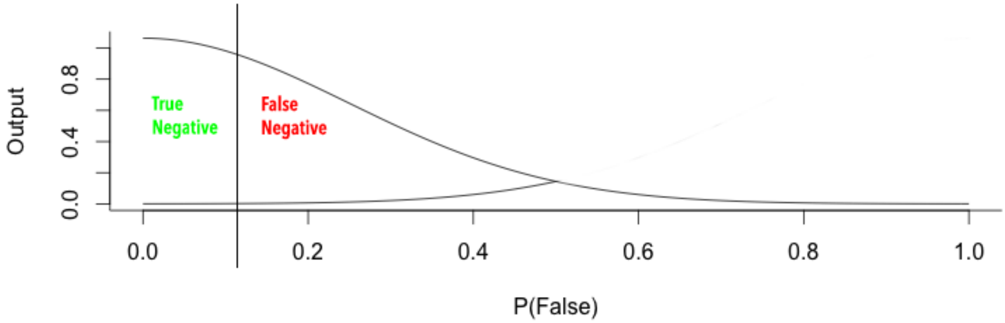
Increases in sensitivity will usually result in a decrease of specificity. Figure 5.4 shows a neural limit designed to make the neural network very sensitive:

Figure 5.4: Sensitive Cutoff



The neural network can also be calibrated for greater sensitivity, as shown in Figure 5.5:

Figure 5.5: Specific Cutoff

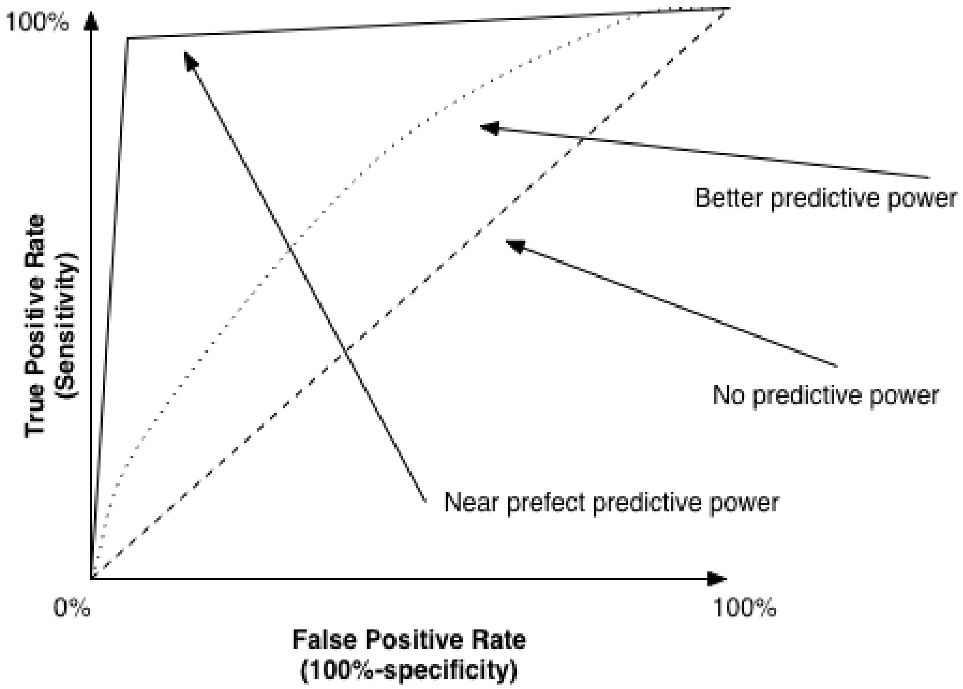


Attaining 100% specificity or sensitivity is not necessarily good. A medical test can reach 100% specificity by simply predicting that everyone does not have the disease. This test will never commit a false positive error because it never gave a positive answer. Obviously, this test is not useful. Highly specific or sensitive neural networks produce the same meaningless result. We need a way to evaluate the total effectiveness of the neural network that is independent of the cutoff point. The total prediction rate combines the percentage of true positives and true negatives. Equation 5.1 can calculate the total prediction rate:

$$TPR = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

Additionally, you can visualize the total prediction rate (TPR) with a receiver operator characteristic (ROC) chart, as seen in Figure 5.6:

Figure 5.6: Receiver Operator Characteristic (ROC) Chart



The above chart shows three different ROC curves. The dashed line shows an ROC with zero predictive power. The dotted line shows a better neural network, and the solid line shows a nearly perfect neural network. To understand how to read an ROC chart, look first at the origin, which is marked by 0%. All ROC lines always start at the origin and move to the upper-right corner where true positive (TP) and false positive (FP) are both 100%.

The y-axis shows the TP percentages from 0 to 100. As you move up the y-axis, both TP and FP increase. As TP increases, so does sensitivity; however, specificity falls. The ROC chart allows you to select the level of sensitivity you need, but it also shows you the number of FPs you must accept to achieve that level of sensitivity.

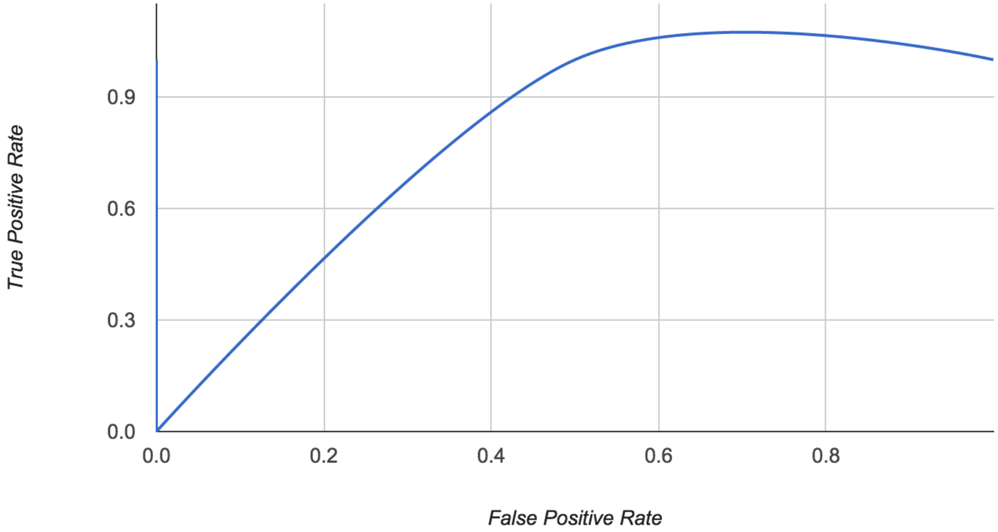
The worst network, the dashed line, always has a 50% total prediction rate. Given that there are only two outcomes, this result is no better than random guessing. To get 100% TP, you must also have a 100% FP, which still results in half of the predictions being wrong.

The following URL allows you to experiment with a simple neural network and ROC curve:

`http://www.heatonresearch.com/aifh/vol3/anneal_roc.html`

We can train the neural network at the above URL with simulated annealing. Each time an annealing epoch is completed, the neural network improves. We can measure this improvement by the mean squared error calculation (MSE). As the MSE drops, the ROC curve stretches towards the upper left corner. We will describe the MSE in greater detail later in this chapter. For now, simply think of it as a measurement of the neural network's error when you compare it to the expected output. A lower MSE is desirable. Figure 5.7 shows the ROC curve after we have trained the network for a number of iterations:

Figure 5.7: ROC Curve



It is important to note that the goal is not always to maximize the total prediction rate. Sometimes a false positive (FP) is better than a false negative (FN.) Consider a neural network that predicts a bridge collapse. A FP means that the program predicts a collapse when the bridge was actually safe. In this case, checking a structurally sound bridge would waste an engineer's time. On the other hand, a FN would mean that the neural network predicted the bridge was safe when it actually collapsed. A bridge collapsing is a much worse outcome than wasting the time of an engineer. Therefore, you should arrange this type of neural network so that it is overly specific.

To evaluate the total effectiveness of the network, you should consider the area under the curve (AUC). The optimal AUC would be 1.0, which is a 100% (1.0) x 100% (1.0) rectangle that pushes the area under the curve to the maximum. When reading an ROC curve, the more effective neural networks have more space under the curve. The curves shown previously, in Figure 5.6, correspond with this assessment.

5.1.2 Multi-Class Classification

If you want to predict more than one outcome, you will need more than one output neuron. Because a single neuron can predict two outcomes, a neural network with two output neurons is somewhat rare. If there are three or more outcomes, there will be three or more output neurons. *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms* does show a method that can encode three outcomes into two output neurons.

Consider Fisher's iris data set. This data set contains four different measurements for three different species of iris flower. The following URL contains this data set:

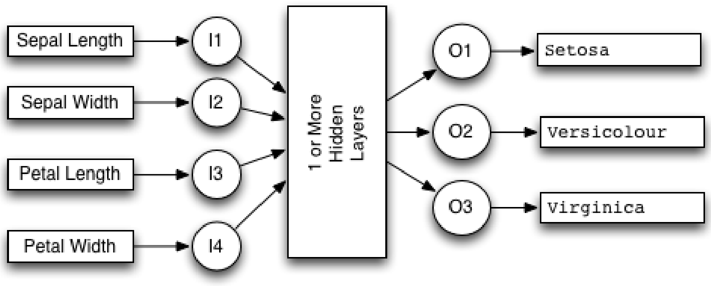
`https://archive.ics.uci.edu/ml/datasets/Iris`

Sample data from the iris data set is shown here:

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolour
6.4,3.2,4.5,1.5,Iris-versicolour
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
```

Four measurements can predict the species. If you are interested in reading more about how to measure an iris flower, refer to the above link. For this prediction, the meaning of the four measurements does not really matter. These measurements will teach the neural network to predict. Figure 5.8 shows a neural network structure that can predict the iris data set:

Figure 5.8: Iris Data Set Neural Network



The above neural network accepts the four measurements and outputs three numbers. Each output corresponds with one of the iris species. The output neuron that produces the highest number determines the species predicted.

5.1.3 Log Loss

Classification networks can derive a class from the input data. For example, the four iris measurements can group the data into the three species of iris. One easy method to evaluate classification is to treat it like a multiple-choice exam and return a percent score. Although this technique is common, most machine learning models do not answer multiple-choice questions like you did in school. Consider how the following question might appear on an exam:

```
1. Would an iris setosa have a sepal length of 5.1 cm,
   a sepal width of 3.5 cm, a petal length of 1.4 cm, and
   a petal width of 0.2 cm?
A) True
B) False
```

This question is exactly the type that a neural network must face in a classification task. However, the neural network will not respond with an answer of “True” or “False.” It will answer the question in the following manner:

```
True: 80%
```

The above response means that the neural network is 80% sure that the flower is a setosa. This technique would be very handy in school. If you could not decide between true and false, you could simply place 80% on “True.” Scoring is relatively easy because you receive your percentage value for the correct answer. In this case, if “True” were the correct answer, your score would be 80% for that question.

However, log loss is not quite that simple. Equation 5.2 is the equation for log loss:

$$\text{log loss} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right) \tag{5.2}$$

You should use this equation only as an objective function for classifications that have two outcomes. The variable *y-hat* is the neural network’s prediction, and the variable y is the known correct answer. In this case, y will always be 0 or 1. The training data have no probabilities. The neural network classifies it either into one class (1) or the other (0).

The variable N represents the number of elements in the training set–the number of questions in the test. We divide by N because this process is customary for an average. We also begin the equation with a negative because the log function is always negative over the domain 0 to 1. This negation allows a positive score for the training to minimize.

You will notice two terms are separated by the addition (+). Each contains a log function. Because y will be either 0 or 1, then one of these two terms will cancel out to 0. If y is 0, then the first term will reduce to 0. If y is 1, then the second term will be 0.

If your prediction for the first class of a two-class prediction is *y-hat*, then your prediction for the second class is 1 minus *y-hat*. Essentially, if your prediction for class A is 70% (0.7), then your prediction for class B is 30% (0.3). Your score will increase by the log of your prediction for the correct class. If the neural network had predicted 1.0 for class A, and the correct answer was A, your score would increase by log (1), which is 0. For log loss, we seek a low score, so a correct answer results in 0. Some of these log values for a neural network's probability estimate for the correct class:

- -log(1.0) = 0
- -log(0.95) = 0.02
- -log(0.9) = 0.05
- -log(0.8) = 0.1
- -log(0.5) = 0.3
- -log(0.1) = 1
- -log(0.01) = 2
- -log(1.0e-12) = 12
- -log(0.0) = negative infinity

As you can see, giving a low confidence to the correct answer affects the score the most. Because log (0) is negative infinity, we typically impose a minimum value. Of course, the above log values are for a single training set element. We will average the log values for the entire training set.

5.1.4 Multi-Class Log Loss

If more than two outcomes are classified, then we must use multi-class log loss. This loss function is very closely related to the binary log loss just described. Equation 5.3 shows the equation for multi-class log loss:

$$\text{multi-class log loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(\hat{y}_{i,j}) \tag{5.3}$$

In the above equation, N is the number of training set elements, and M represents the number of categories for the classification process. Conceptually, the multi-class log loss objective function works similarly to single log loss. The above equation essentially gives you a score that is the average of the negative-log of your prediction for the correct class on each of the data sets. The inner most sigma-summation in the above equation functions as an if-then statement and allows only the correct class with a y of 1.0 to contribute to the summation.

5.2 Evaluating Regression

Mean squared error (MSE) calculation is the most commonly utilized process for evaluating regression machine learning. Most Internet examples of neural networks, support vector machines, and other models apply MSE (Draper, 1998), shown in Equation 5.4:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2 \tag{5.4}$$

In the above equation, y is the ideal output and *y-hat* is the actual output. The mean squared error is essentially the mean of the squares of the individual differences. Because the individual differences are squared, the positive or negative nature of the difference does not matter to MSE.

You can evaluate classification problems with MSE. To evaluate classification output with MSE, each class's probability is simply treated as a numeric output. The expected output simply has a value of 1.0 for the correct class, and 0 for the others. For example, if the first class were correct, and the other three classes incorrect, the expected outcome vector would look like the following:

```
[1.0, 0, 0, 0]
```

You can use nearly any regression objective function for classification in this way. A variety of functions, such as root mean square (RMS) and sum of squares error (SSE) can evaluate regression, and we discussed these functions in *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms.*

5.3 Training with Simulated Annealing

To train a neural network, you must define its tasks. An objective function, otherwise known as scoring or loss functions, can generate these tasks. Essentially, an objective function evaluates the neural network and returns a number indicating the usefulness of the neural network. The training process modifies the weights of the neural network in each iteration so the value returned from the objective function improves.

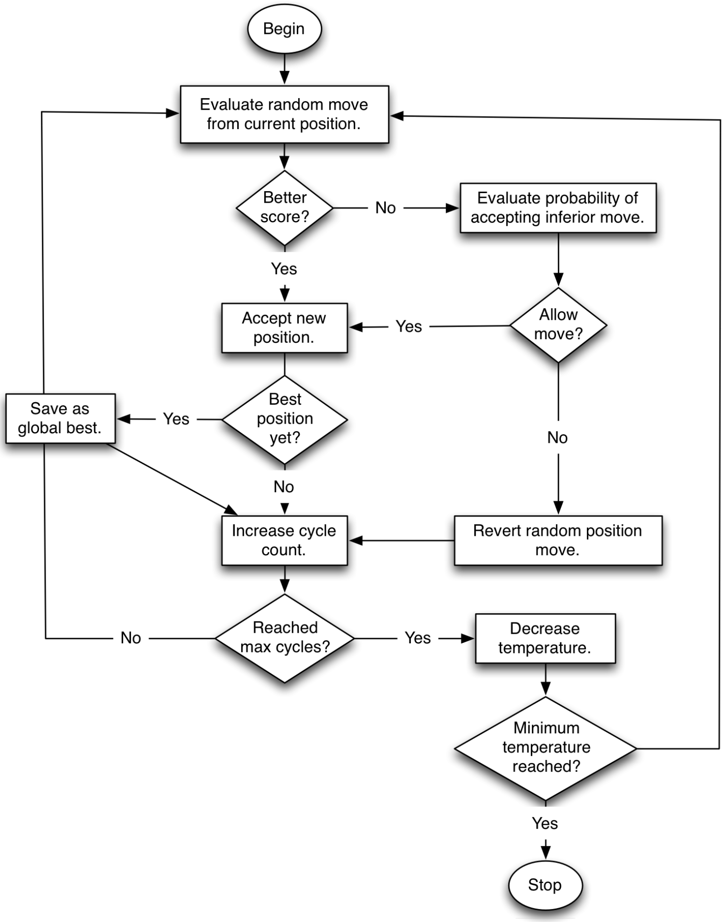
Simulated annealing is an effective optimization technique that we examined in *Artificial Intelligence for Humans Volume 1.* In this chapter, we will review simulated annealing as well as show you how any vector optimization function can improve the weights of a feedforward neural network. In the next chapter, we will examine even more advanced optimization techniques that take advantage of the differentiable loss function.

As a review, simulated annealing works by first assigning the weight vector of a neural network to random values. This vector is treated like a position, and the program evaluates every possible move from that position. To understand how a neural network weight vector translates to a position, think of a neural network with just three weights. In the real world, we consider position in terms of the x, y and z coordinates. We can write any position as a vector of 3. If we are willing to move in a single dimension, we could move in a total of six different directions. We would have the option of moving forward or backwards in the x, y or z dimensions.

Simulated annealing functions by moving forward or backwards in all available dimensions. If the algorithm takes the best move, a simple hill-climbing algorithm would result. Hill climbing only improves scores. Therefore, it is called a greedy algorithm. To reach the best position, an algorithm will sometime need to move to a lower position. As a result, simulated annealing very much follows the expression of two steps forward, one step back.

In other words, simulated annealing will sometimes allow a move to a weight configuration with a worse score. The probability of accepting such a move starts high and decreases. This probability is known as the current temperature, and it simulates the actual metallurgical annealing process where a metal cools and achieves greater hardness. Figure 5.9 shows the entire process:

Figure 5.9: Simulated Annealing



A feedforward neural network can utilize simulated annealing to learn the iris data set. The following program shows the output from this training:

```
Iteration #1, Score=0.3937, k=1,kMax=100,t=343.5891,prob=0.9998
Iteration #2, Score=0.3937, k=2,kMax=100,t=295.1336,prob=0.9997
Iteration #3, Score=0.3835, k=3,kMax=100,t=253.5118,prob=0.9989
Iteration #4, Score=0.3835, k=4,kMax=100,t=217.7597,prob=0.9988
Iteration #5, Score=0.3835, k=5,kMax=100,t=187.0496,prob=0.9997
Iteration #6, Score=0.3835, k=6,kMax=100,t=160.6705,prob=0.9997
Iteration #7, Score=0.3835, k=7,kMax=100,t=138.0116,prob=0.9996
```

```
...
Iteration #99, Score=0.1031, k=99,kMax=100,t=1.16E-4,prob=2.8776E
    -7
Iteration #100, Score=0.1031, k=100,kMax=100,t=9.9999E-5,prob
    =2.1443E-70
Final score: 0.1031
[0.22222222222222213, 0.6249999999999999, 0.06779661016949151,
    0.04166666666666667] -> Iris-setosa, Ideal: Iris-setosa
[0.1666666666666668, 0.41666666666666663, 0.06779661016949151,
    0.04166666666666667] -> Iris-setosa, Ideal: Iris-setosa
...
[0.6666666666666666, 0.41666666666666663, 0.711864406779661,
    0.9166666666666666] -> Iris-virginica, Ideal: Iris-virginica
[0.5555555555555555, 0.20833333333333331, 0.6779661016949152,
    0.75] -> Iris-virginica, Ideal: Iris-virginica
[0.611111111111111, 0.41666666666666663, 0.711864406779661,
    0.7916666666666666] -> Iris-virginica, Ideal: Iris-virginica
[0.5277777777777778, 0.5833333333333333, 0.7457627118644068,
    0.9166666666666666] -> Iris-virginica, Ideal: Iris-virginica
[0.44444444444444453, 0.41666666666666663, 0.6949152542372881,
    0.7083333333333334] -> Iris-virginica, Ideal: Iris-virginica
[1.178018083703488, 16.66575553359515, -0.6101619300462806,
    -3.9894606091020965, 13.989551673146842, -8.87489712462323,
    8.027287801488647, -4.615098285283519, 6.426489182215509,
    -1.4672962642199618, 4.136699061975335, 4.20036115439746,
    0.9052469139543605, -2.8923515248132063, -4.733219252086315,
    18.6497884912826, 2.5459600552510895, -5.618872440836617,
    4.638827606092005, 0.8887726364890928, 8.730809901357286,
    -6.4963370793479545, -6.4003385330186795, -11.820235441582424,
     -3.29494170904095, -1.5320936828139837, 0.1094081633203249,
    0.26353076268018827, 3.935780218339343, 0.8881280604852664,
    -5.048729642423418, 8.288232057956957, -14.686080237582006,
    3.058305829324875, -2.4144038920292608, 21.76633883966702,
    12.151853576801647, -3.6372061664901416, 6.28253174293219,
    -4.209863472970308, 0.8614258660906541, -9.382012074551428,
    -3.346419915864691, -0.6326977049713416, 2.1391118323593203,
    0.44832732990560714, 6.853600355726914, 2.8210824313745957,
    1.3901883615737192, -5.962068350552335, 0.502596306917136]
```

The initial random neural network starts out with a high multi-class log loss score of 30. As the training progresses, this value falls until it is low enough for training to stop. For this example, the training stops as soon as the error falls below 10. To determine a good stopping point for the error, you should evaluate how well the network is performing for your intended use. A log loss below 0.5 is often in the acceptable range; however, you might not be able to achieve this score with all data sets.

The following URL shows an example of a neural network trained with simulated annealing:

`http://www.heatonresearch.com/aifh/vol3/anneal_roc.html`

5.4 Chapter Summary

Objective functions can evaluate neural networks. They simply return a number that indicates the success of the neural network. Regression neural networks will frequently utilize mean squared error (MSE). Classification neural networks will typically use a log loss or multi-class log loss function. These neural networks create custom objective functions.

Simulated annealing can optimize the neural network. You can utilize any of the optimization algorithms presented in Volumes 1 and 2 of *Artificial Intelligence for Humans*. In fact, you can optimize any vector in this way because the optimization algorithms are not tied to a neural network. In the next chapter, you will see several training methods designed specifically for neural networks. While these specialized training algorithms are often more efficient, they require objective functions that have a derivative.

Chapter 6

Backpropagation Training

- Gradient Calculation
- Backpropagation
- Learning Rate & Momentum
- Stochastic Gradient Descent

Backpropagation is one of the most common methods for training a neural network. Rumelhart, Hinton, & Williams (1986) introduced backpropagation, and it remains popular today. Programmers frequently train deep neural networks with backpropagation because it scales really well when run on graphical processing units (GPUs). To understand this algorithm for neural networks, we must examine how to train it as well as how it processes a pattern.

Classic backpropagation has been extended and modified to give rise to many different training algorithms. In this chapter, we will discuss the most commonly used training algorithms for neural networks. We begin with classic backpropagation and then end the chapter with stochastic gradient descent (SGD).

6.1 Understanding Gradients

Backpropagation is a type of gradient descent, and many texts will use these two terms interchangeably. Gradient descent refers to the calculation of a gradient on each weight in the neural network for each training element. Because the neural network will not output the expected value for a training element, the gradient of each weight will give you an indication about how to modify each weight to achieve the expected output. If the neural network did output exactly what was expected, the gradient for each weight would be 0, indicating that no change to the weight is necessary.

The gradient is the derivative of the error function at the weight's current value. The error function measures the distance of the neural network's output from the expected output. In fact, we can use gradient descent, a process in which each weight's gradient value can reach even lower values of the error function.

With respect to the error function, the gradient is essentially the partial derivative of each weight in the neural network. Each weight has a gradient that is the slope of the error function. A weight is a connection between two neurons. Calculating the gradient of the error function allows the training method to determine whether it should increase or decrease the weight. In turn, this determination will decrease the error of the neural network. The error is the difference between the expected output and actual output of the neural network. Many different training methods called propagation-training algorithms utilize gradients. In all of them, the sign of the gradient tells the neural network the following information:

- **Zero gradient** - The weight is not contributing to the error of the neural network.
- **Negative gradient** - The weight should be increased to achieve a lower error.
- **Positive gradient** - The weight should be decreased to achieve a lower error.

Because many algorithms depend on gradient calculation, we will begin with an analysis of this process.

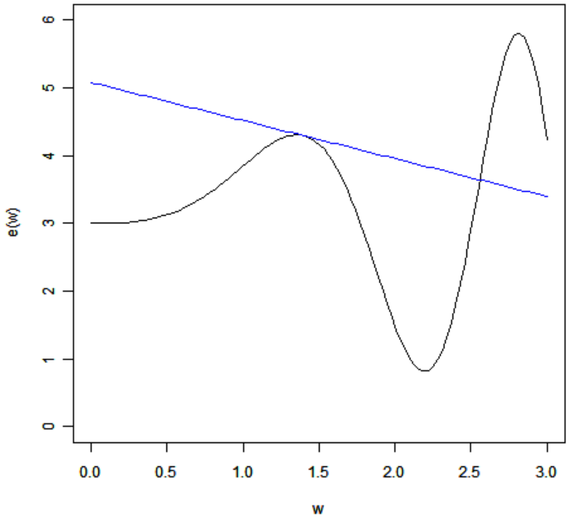
6.1.1 What is a Gradient

First of all, let's examine the gradient. Essentially, training is a search for the set of weights that will cause the neural network to have the lowest error for a training set. If we had an infinite amount of computation resources, we would simply try every possible combination of weights to determine the one that provided the lowest error during the training.

Because we do not have unlimited computing resources, we have to use some sort of shortcut to prevent the need to examine every possible weight combination. These training methods utilize clever techniques to avoid performing a brute-force search of all weight values. This type of exhaustive search would be impossible because even small networks have an infinite number of weight combinations.

Consider a chart that shows the error of a neural network for each possible weight. Figure 6.1 is a graph that demonstrates the error for a single weight:

Figure 6.1: Gradient of a Single Weight



Looking at this chart, you can easily see that the optimal weight is the location where the line has the lowest y-value. The problem is that we see only the error for the current value of the weight; we do not see the entire graph because that process would require an exhaustive search. However, we can determine the slope of the error curve at a particular weight. In the above chart, we see the slope of the error curve at 1.5. The straight line that barely touches the error curve at 1.5 gives the slope. In this case, the slope, or gradient, is -0.5622. The negative slope indicates that an increase in the weight will lower the error.

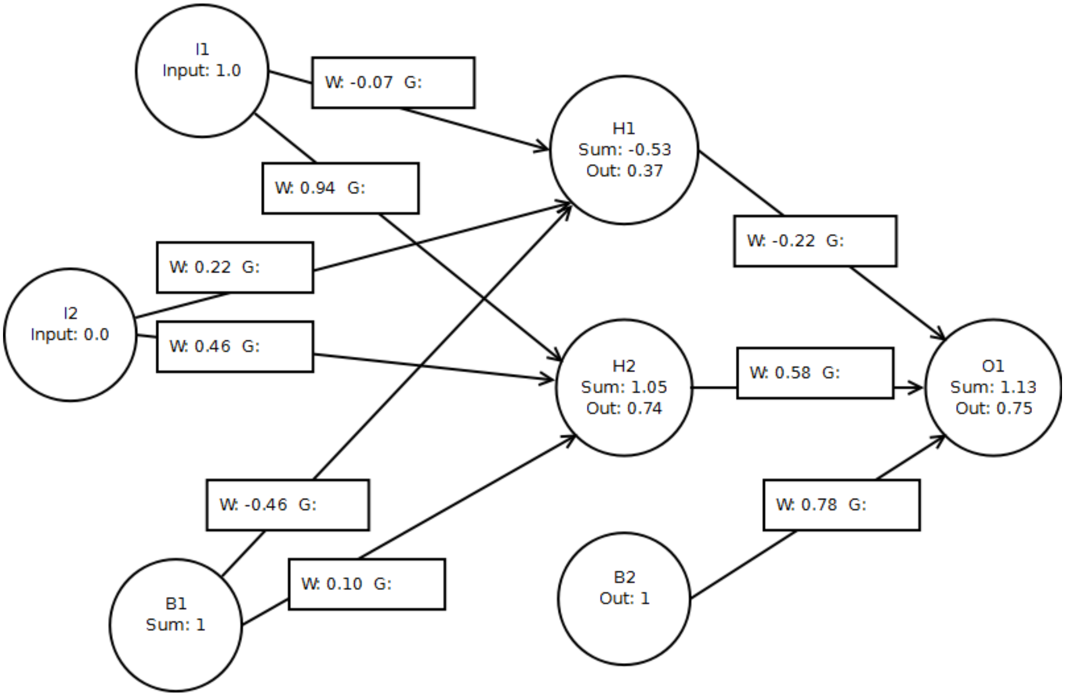
The gradient is the instantaneous slope of the error function at the specified weight. The derivative of the error curve at that point gives the gradient. This line tells us the steepness of the error function at the given weight.

Derivatives are one of the most fundamental concepts in calculus. For the purposes of this book, you just need to understand that a derivative provides the slope of a function at a specific point. A training technique and this slope can give you the information to adjust the weight for a lower error. Using our working definition of the gradient, we will now show how to calculate it.

6.1.2 Calculating Gradients

We will calculate an individual gradient for each weight. Our focus is not only the equations but also the applications in actual neural networks with real numbers. Figure 6.2 shows the neural network that we will use:

Figure 6.2: An XOR Network



Additionally, we use this same neural network in several examples on the website for this book. In this chapter, we will show several calculations that demonstrate the training of a neural network. We must use the same starting weights so that these calculations are consistent. However, the above weights have no special characteristic; the program generated them randomly.

The aforementioned neural network is a typical three-layer feedforward network like the ones we have previously studied. The circles indicate neurons. The lines connecting the circles are the weights. The rectangles in the middle of the connections give the weight for each connection.

The problem that we now face is calculating the partial derivative for each of the weights in the neural network. We use a partial derivative when an equation has more than one variable. Each of the weights is considered a variable because these weight values will change independently as the neural network changes. The partial derivatives of each weight simply show each weight's independent effect on the error function. This partial derivative is the gradient.

We can calculate each partial derivative with the chain rule of calculus. We will begin with one training set element. For Figure 6.2 we provide an input of [1,0] and expect an output of [1]. You can see that we apply the input on the above figure. The first input neuron has an input value of 1.0, and the second input neuron has an input value of 0.0.

This input feeds through the network and eventually produces an output. Chapter 4, "Feedforward Neural Networks," covers the exact process to calculate the output and sums. Backpropagation has both a forward and backwards pass. The forward pass occurs when we calculate the output of the neural network. We will calculate the gradients only for this item in the training set. Other items in the training set will have different gradients. We will discuss how to combine the gradients for the individual training set element later in the chapter.

We are now ready to calculate the gradients. The steps involved in calculating the gradients for each weight are summarized here:

- Calculate the error, based on the ideal of the training set.
- Calculate the node (neuron) delta for the output neurons.
- Calculate the node delta for the interior neurons.
- Calculate individual gradients.

We will discuss these steps in the subsequent sections.

6.2 Calculating Output Node Deltas

Calculating a constant value for every node, or neuron, in the neural network is the first step. We will start with the output nodes and work our way backwards through the neural network. The term backpropagation comes from this process. We initially calculate the errors for the output neurons and propagate these errors backwards through the neural network.

The node delta is the value that we will calculate for each node. Layer delta also describes this value because we can calculate the deltas one layer at a time. The method for determining the node deltas can differ if you are calculating for an output or interior node. The output nodes are calculated first, and they take into account the error function for the neural network. In this volume, we will examine the quadratic error function and the cross entropy error function.

6.2.1 Quadratic Error function

Programmers of neural networks frequently use the quadratic error function. In fact, you can find many examples of the quadratic error function on the Internet. If you are reading an example program, and it does not mention a specific error function, the program is probably using the quadratic error function, also known as the mean squared error (MSE) function discussed in Chapter 5, "Training and Evaluation." Equation 6.1 shows the MSE function:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{6.1}$$

The above equation compares the neural network's actual output (y) with the expected output (*y-hat*). The variable n contains the number of training elements times the number of output neurons. MSE handles multiple output neurons as individual cases. Equation 6.2 shows the node delta used in conjunction with the quadratic error function:

$$\delta_i = (\hat{y}_i - y_i)\phi_i' \tag{6.2}$$

The quadratic error function is very simple because it takes the difference between the expected and actual output for the neural network. The Greek letter ϕ (phi-prime) represents the derivative of the activation function.

6.2.2 Cross Entropy Error Function

The quadratic error function can sometimes take a long time to properly adjust the weight. Equation 6.3 shows the cross entropy error function:

$$CE = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)] \tag{6.3}$$

The node delta calculation for the cross entropy error turns out to be much less complex than the MSE, as seen in Equation 6.4.

$$\delta_i = \hat{y}_i - y_i \tag{6.4}$$

The cross entropy error function will typically better results than the quadratic it will create a much steeper gradient for errors. You should always use the cross entropy error function.

6.3 Calculating Remaining Node Deltas

Now that the output node delta has been calculated according to the appropriate error function, we can calculate the node deltas for the interior nodes, as demonstrated by Equation 6.5:

$$\delta_i = \phi_i' \sum_{k} w_{ki}\delta_k \tag{6.5}$$

We will calculate the node delta for all hidden and non-bias neurons, but we do not need to calculate the node delta for the input and bias neurons. Even though we can easily calculate the node delta for input and bias neurons with Equation 6.5, gradient calculation does not require these values. As you will soon see, gradient calculation for a weight only considers the neuron to which the weight is connected. Bias and input neurons are only the beginning point for a connection; they are never the end point.

If you would like to see the gradient calculation process, several JavaScript examples will show the individual calculations. These examples can be found at the following URL:

`http://www.heatonresearch.com/aifh/vol3/`

6.4 Derivatives of the Activation Functions

The backpropagation process requires the derivatives of the activation functions, and they often determine how the backpropagation process will perform. Most modern deep neural networks use the linear, softmax, and ReLU activation functions. We will also examine the derivatives of the sigmoid and hyperbolic tangent activation functions so that we can see why the ReLU activation function performs so well.

6.4.1 Derivative of the Linear Activation Function

The linear activation function is barely an activation function at all because it simply returns whatever value it is given. For this reason, the linear activation function is sometimes called the identity activation function. The derivative of this function is 1, as demonstrated by Equation 6.6:

$$\phi\prime(x) = 1 \tag{6.6}$$

The Greek letter ϕ (phi) represents the activation function, as in previous chapters. However, the apostrophe just above and to the right of ϕ (phi) means that we are using the derivative of the activation function. This is one of several ways that a derivative is expressed in a mathematical form.

6.4.2 Derivative of the Softmax Activation Function

In this volume, the softmax activation function, along with the linear activation function, is used only on the output layer of the neural networks. As mentioned in Chapter 1, “Neural Network Basics,” the softmax activation function is different from the other activation functions in that its value is dependent on the other output neurons, not just on the output neuron currently being calculated. For convenience, the softmax activation function is repeated in Equation 6.7:

$$\phi_i = \frac{e^{z_i}}{\sum\limits_{j \in group} e^{z_j}} \tag{6.7}$$

The z vector represents the output from all output neurons. Equation 6.8 shows the derivative of this activation function:

$$\frac{\partial \phi_i}{\partial z_i} = \phi_i(1 - \phi_i) \tag{6.8}$$

We used slightly different notation for the above derivative. The ratio, with the cursive-stylized "d" symbol means a partial derivative, which occurs when you differentiate an equation with multiple variables. To take a partial derivative, you differentiate the equation relative to one variable, holding all others constant. The top "d" tells you what function you are differentiating. In this case, it is the activation function ϕ (phi). The bottom "d" denotes the respective differentiation of the partial derivative. In this case, we are calculating the output of the neuron. All other variables are treated as constant. A derivative is the instantaneous rate of change–only one thing can change at once.

You will not use the derivative of the linear or softmax activation functions to calculate the gradients of the neural network if you use the cross entropy error function. You should use the linear and softmax activation functions only at the output layer of a neural network. Therefore, we do not need to worry about their derivatives for the interior nodes. For the output nodes with cross entropy, the derivative of both linear and softmax is always 1. As a result, you will never use the linear or softmax derivatives for interior nodes.

6.4.3 Derivative of the Sigmoid Activation Function

Equation 6.9 shows the derivative of the sigmoid activation function:

$$\phi\prime(x) = \phi(x)(1 - \phi(x)) \tag{6.9}$$

Machine learning frequently utilizes the sigmoid function represented in the above equation. We derived the formula through algebraic manipulation of the sigmoid derivative in order to use the sigmoid activation function in its own derivative. For computational efficiency, the Greek letter ϕ (phi) in the above activation function represents the sigmoid function. During the feedforward pass, we calculated the value of the sigmoid function. Retaining the sigmoid function makes the sigmoid derivative a simple calculation. If you are interested in how to obtain Equation 6.9, you can refer to the following URL:

`http://www.heatonresearch.com/aifh/vol3/deriv_sigmoid.html`

6.4.4 Derivative of the Hyperbolic Tangent Activation Function

Equation 6.10 shows the derivative of the hyperbolic tangent activation function:

$$\phi\prime(x) = 1.0 - \phi^2(x) \tag{6.10}$$

We recommend that you always use the hyperbolic tangent activation function instead of the sigmoid activation function.

6.4.5 Derivative of the ReLU Activation Function

Equation 6.11 shows the derivative of the ReLU function:

$$\frac{dy}{dx}\phi(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{6.11}$$

Strictly speaking, the ReLU function does not have a derivative at 0. However, because of convention, the gradient of 0 is substituted when x is 0. Deep neural networks with sigmoid and hyperbolic tangent activation functions can be difficult to train using backpropagation. Several factors cause this difficulty. The vanishing gradient problem is one the most common causes. Figure 6.3 shows the hyperbolic tangent function, along with its gradient/derivative:

Figure 6.3: Tanh Activation Function & Derivative

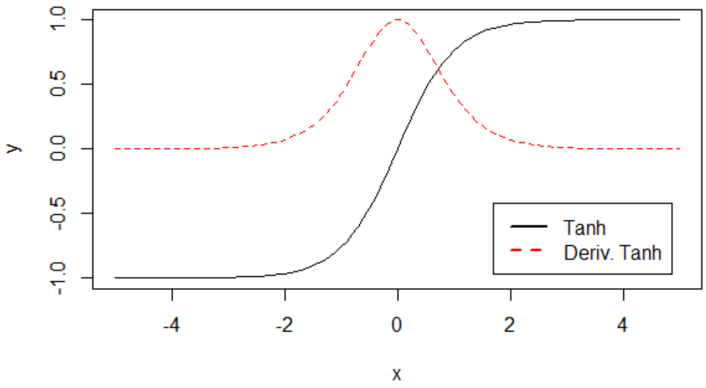
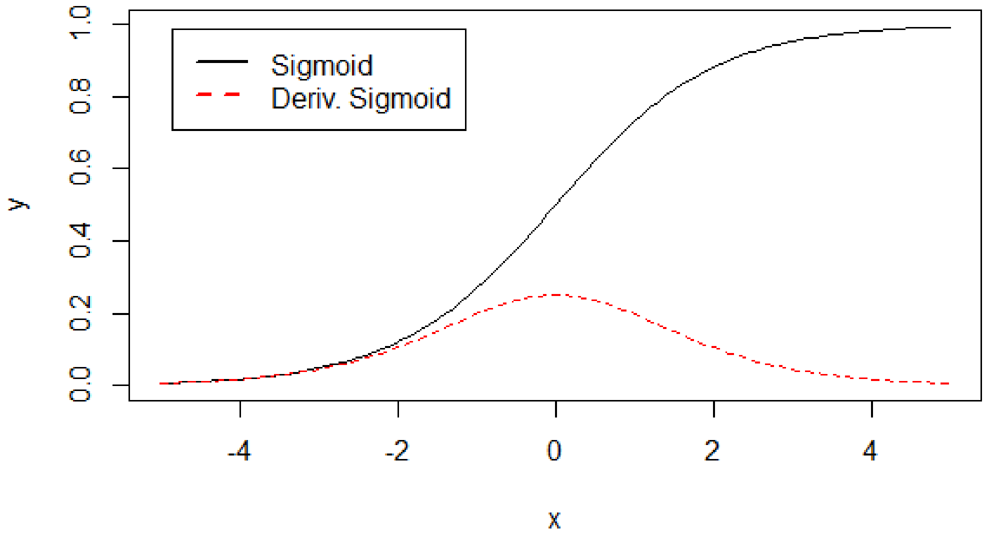


Figure 6.3 shows that as the hyperbolic tangent (blue line) saturates to -1 and 1, the derivative of the hyperbolic tangent (red line) vanishes to 0. The sigmoid and hyperbolic tangent activation functions both have this problem, but ReLU doesn't. Figure 6.4 shows the same graph for the sigmoid activation function and its vanishing derivative:

Figure 6.4: Tanh Activation Function & Derivative



6.5 Applying Backpropagation

Backpropagation is a simple training method that adjusts the weights of the neural network with its calculated gradients. This method is a form of gradient descent since we are descending the gradients to lower values. As the program adjusts these weights, the neural network should produce more desirable output. The global error of the neural network should fall as it trains. Before we can examine the backpropagation weight update process, we must examine two different ways to update the weights.

6.5.1 Batch and Online Training

We have already shown how to calculate the gradients for an individual training set element. Earlier in this chapter, we calculated the gradients for a case in which we gave the neural network an input of [1,0] and expected an output of

[1]. This result is acceptable for a single training set element. However, most training sets have many elements. Therefore, we can handle multiple training set elements through two approaches called online and batch training.

Online training implies that you modify the weights after every training set element. Using the gradients obtained in the first training set element, you calculate and apply a change to the weights. Training progresses to the next training set element and also calculates an update to the neural network. This training continues until you have used every training set element. At this point, one iteration, or epoch, of training has completed.

Batch training also utilizes all the training set elements. However, we have not updated the weights. Instead, we sum the gradients for each training set element. Once we have summed the training set elements, we can update the neural network weights. At this point, the iteration is complete.

Sometimes, we can set a batch size. For example, you might have a training set size of 10,000 elements. You might choose to update the weights of the neural network every 1,000 elements, thereby causing the neural network weights to update ten times during the training iteration.

Online training was the original method for backpropagation. If you would like to see the calculations for the batch version of this program, refer to the following online example:

```
http://www.heatonresearch.com/aifh/vol3/xor_batch.html
```

6.5.2 Stochastic Gradient Descent

Batch and online training are not the only choices for backpropagation. Stochastic gradient descent (SGD) is the most popular of the backpropagation algorithms. SGD can work in either batch or online mode. Online stochastic gradient descent simply selects a training set element at random and then calculates the gradient and performs a weight update. This process continues until the error reaches an acceptable level. Choosing random training set elements will usually converge to an acceptable weight faster than looping through the entire training set for each iteration.

Batch stochastic gradient descent works by choosing a batch size. For each iteration, a mini-batch is chosen by randomly selecting a number of training set elements up to the chosen batch size. The gradients from the mini-batch are summed just as regular backpropagation batch updating. This update is very similar to regular batch updating except that the mini-batches are randomly chosen each time they are needed. The iterations typically process a single batch in SGD. Batches are usually much smaller than the entire training set size. A common choice for the batch size is 600.

6.5.3 Backpropagation Weight Update

We are now ready to update the weights. As previously mentioned, we will treat the weights and gradients as a single-dimensional array. Given these two arrays, we are ready to calculate the weight update for an iteration of backpropagation training. Equation 6.6 shows the formula to update the weights for backpropagation:

$$\Delta w_{(t)} = -\epsilon \frac{\partial E}{\partial w_{(t)}} + \alpha \Delta w_{(t-1)} \tag{6.12}$$

The above equation calculates the change in weight for each element in the weight array. You will also notice that the above equation calls for the weight change from the previous iteration. You must keep these values in another array. As previously mentioned, the direction of the weight update is inversely related to the sign of the gradient–a positive gradient should cause a weight decrease, and vice versa. Because of this inverse relationship Equation 6.12 begins with a negative.

The above equation calculates the weight delta as the product of the gradient and the learning rate (represented by ϵ, epsilon). Furthermore, we add the product of the previous weight change and the momentum value (represented by α, alpha). The learning rate and momentum are two parameters that we must provide to the backpropagation algorithm. Choosing values for learning rate and momentum is very important to the performance of the training. Unfortunately, the process for determining learning rate and momentum is mostly trial and error.

The learning rate scales the gradient and can slow down or speed up learning. A learning rate below 0 will slow down learning. For example, a learning rate of 0.5 would decrease every gradient by 50%. A learning rate above 1.0

would accelerate training. In reality, the learning rate is almost always below 1.

Choosing a learning rate that is too high will cause your neural network to fail to converge and have a high global error that simply bounces around instead of converging to a low value. Choosing a learning rate that is too low will cause the neural network to take a great deal of time to converge.

Like the learning rate, the momentum is also a scaling factor. Although it is optional, momentum determines the percent of the previous iteration's weight change that should be applied to the iteration. If you do not want to use momentum, just specify a value of 0.

Momentum is a technique added to backpropagation that helps the training escape local minima, which are low points on the error graph that are not the true global minimum. Backpropagation has a tendency to find its way into a local minimum and not find its way back out again. This process causes the training to converge to a higher undesirable error. Momentum gives the neural network some force in its current direction and may allow it to break through a local minimum.

6.5.4 Choosing Learning Rate and Momentum

Momentum and learning rate contribute to the success of the training, but they are not actually part of the neural network. Once training is complete, the trained weights remain and no longer utilize momentum or the learning rate. They are essentially part of the temporary scaffolding that creates a trained neural network. Choosing the correct momentum and learning rate can impact the effectiveness of your training.

The learning rate affects the speed at which your neural network trains. Decreasing the learning rate makes the training more meticulous. Higher learning rates might skip past optimal weight settings. A lower training rate will always produce better results. However, lowering the training rate can greatly increase runtime. Lowering the learning rate as the network trains can be an effective technique.

You can use the momentum to combat local minima. If you find the neural network stagnating, a higher momentum value might push the training past the local minimum that it encountered. Ultimately, choosing good values for momentum and learning rate is a process of trial and error. You can vary both as training progresses. Momentum is often set to 0.9 and the learning rate to 0.1 or lower.

6.5.5 Nesterov Momentum

The stochastic gradient descent (SGD) algorithm can sometimes produce erratic results because of the randomness introduced by the mini-batches. The weights might get a very beneficial update in one iteration, but a poor choice of training elements can undo it in the next mini-batch. Therefore, momentum is a resourceful tool that can mitigate this sort of erratic training result.

Nesterov momentum is a relatively new application of a technique invented by Yu Nesterov in 1983 and updated in his book, *Introductory Lectures on Convex Optimization: A Basic Course* (Nesterov, 2003). This technique is occasionally referred to as Nesterov's accelerated gradient descent. Although a full mathematical explanation of Nesterov momentum is beyond the scope of this book, we will present it for the weights in sufficient detail so you can implement it. This book's examples, including those for the online JavaScript, contain an implementation of Nesterov momentum. Additionally, the book's website contains Javascript that output example calculations for the weight updates of Nesterov momentum.

Equation 6.13 calculates a partial weight update based on both the learning rate (ϵ, epsilon) and momentum (α, alpha):

$$n_0 = 0 \ , \ n_t = \alpha n_{t-1} + \epsilon \frac{\partial E}{\partial w_t} \tag{6.13}$$

The current iteration is signified by t, and the previous iteration by t-1. This partial weight update is called n and initially starts out at 0. Subsequent calculations of the partial weight update are based on the previous value of the partial weight update. The partial derivative in the above equation is the gradient of the error function at the current weight. Equation 6.14 shows the Nesterov momentum update that replaces the standard backpropagation weight update shown earlier in Equation 6.12:

$$\Delta w_t = \alpha n_{t-1} - (1 + \alpha) n_t \tag{6.14}$$

The above weight change is calculated as an amplification of the partial weight change. The delta weight shown in the above equation is added to the current weight. Stochastic gradient descent (SGD) with Nesterov momentum is one of the most effective training algorithms for deep learning.

6.6 Chapter Summary

This chapter introduced classic backpropagation as well as stochastic gradient descent (SGD). These methods are all based on gradient descent. In other words, they optimized individual weights with derivatives. For a given weight value, the derivative gave the program the slope of the error function. The slope allowed the program to determine how to change the weight value. Each training algorithm interprets this slope, or gradient, differently.

Despite the fact that backpropagation is one of the oldest training algorithms, it remains one of the most popular ones. Backpropagation simply adds the gradient to the weight. A negative gradient will increase the weight, and a positive gradient will decrease the weight. We scale the weight by the learning rate in order to prevent the weights from changing too rapidly. A learning rate of 0.5 would mean to add half of the gradient to the weight, whereas a learning rate of 2.0 would mean to add twice the gradient.

There are a number of variants to the backpropagation algorithm. Some of these, such as resilient propagation, are somewhat popular. The next chapter will introduce some backpropagation variants. Though these variants are useful to know, stochastic gradient descent (SGD) remains the most common deep learning training algorithm.

Chapter 7

Other Propagation Training

- Resilient Propagation
- Levenberg-Marquardt
- Hessian and Jacobean Matrices

The backpropagation algorithm has influenced many training algorithms, such as the stochastic gradient descent (SGD), introduced in the previous chapter. For most purposes, the SGD algorithm, along with Nesterov momentum, is a good choice for a training algorithm. However, other options exist. In this chapter, we examine two popular algorithms inspired by elements from backpropagation.

To make use of these two algorithms, you do not need to understand every detail of their implementation. Essentially, both algorithms accomplish the same objective as backpropagation. Thus, you can substitute them for backpropagation or stochastic gradient descent (SGD) in most neural network frameworks. If you find SGD is not converging, you can switch to resilient propagation (RPROP) or Levenberg-Marquardt algorithm in order to experiment. However, you can skip this chapter if you are not interested in the actual implementation details of either algorithm.

7.1 Resilient Propagation

RPROP functions very much like backpropagation. Both backpropagation and RPROP must first calculate the gradients for the weights of the neural network. However, backpropagation and RPROP differ in the way they use the gradients. Reidmiller & Braun (1993) introduced the RPROP algorithm.

One important feature of the RPROP algorithm is that it has no necessary training parameters. When you utilize backpropagation, you must specify the learning rate and momentum. These two parameters can greatly impact the effectiveness of your training. Although RPROP does include a few training parameters, you can almost always leave them at their default.

The RPROP protocol has several variants. Some of the variants are listed below:

- RPROP+
- RPROP-
- iRPROP+
- iRPROP-

We will focus on classic RPROP, as described by Reidmiller & Braun (1994). The other four variants described above are relatively minor adaptations of classic RPROP. In the next sections, we will describe how to implement the classic RPROP algorithm.

7.2 RPROP Arguments

As previously mentioned, one advantage RPROP has over backpropagation is that you don't need to provide any training arguments in order to use RPROP. However, this doesn't mean that RPROP lacks configuration settings. It simply means that you usually do not need to change the configuration settings for RPROP from their defaults. However, if you really want to change them, you can choose among the following configuration settings:

- Initial Update Values
- Maximum Step

As you will see in the next section, RPROP keeps an array of update values for the weights, which determines how much you will alter each weight. This change is similar to the learning rate in backpropagation, but it is much better because the algorithm adjusts the update value of every weight in the neural network as training progresses. Although some backpropagation algorithms will vary the learning rate and momentum as learning progresses, most will use a single learning rate for the entire neural network. Therefore, the RPROP approach has an advantage over backpropagation algorithms.

We start these update values at the default of 0.1, according to the initial update values argument. As a general rule, we should never change this default. However, we can make an exception to this rule if we have already trained the neural network. In the case of a previously trained neural network, some of the initial update values are going to be too strong, and the neural network will regress for many iterations before it can improve. As a result, a trained neural network may benefit from a much smaller initial update.

Another approach for an already trained neural network is to save the update values once training stops and use them for the new training. This method will allow you to resume training without the initial spike in errors that you would normally see when resuming resilient propagation training. This approach will only work if you are continuing resilient propagation on an already trained network. If you were previously training the neural network with a different training algorithm, then you will be able to restore from an array of update values.

As training progresses, you will use the gradients to adjust the updates up and down. The *maximum step* argument defines the maximum upward step size that the gradient can take over the update values. The default value for the maximum step argument is 50. It is unlikely that you will need to change the value of this argument.

In addition to these arguments, RPROP keeps constants during processing. These are values that you can never change. The constants are listed as follows:

- Delta Minimum (1e-6)
- Negative η (Eta) (0.5)
- Positive -η (Eta) (1.2)
- Zero Tolerance (1e-16)

Delta minimum specifies the minimum value that any of the update values can reach. If an update value were at 0, it would never be able to increase beyond 0. We will describe negative and positive η (eta) in the next sections.

The *zero tolerance* defines how closely a number should reach 0 before that number is equal to 0. In computer programming, it is typically bad practice to compare a floating-point number to 0 because the number would have to equal 0 exactly. Rather, you typically see if the absolute value of a number is below an arbitrarily small number. A sufficiently small number is considered 0.

7.3 Data Structures

You must keep several data structures in memory while you perform RPROP training. These structures are all arrays of floating-point numbers. They are summarized here:

- Current Update Values
- Last Weight Change Values
- Current Weight Change Values
- Current Gradient Values
- Previous Gradient Values

You keep the current update values for the training. If you want to resume training at some point, you must store this update value array. Each weight has one update value that cannot go below the minimum delta constant. Likewise, these update values cannot exceed the maximum step argument.

RPROP must keep several values between iterations. You must also track the last weight delta value. Backpropagation keeps the previous weight delta for momentum. RPROP uses this delta value in a different way that we will examine in the next section. You also need the current and previous gradients. RPROP needs to know when the sign changes from the current gradient to the previous gradient. This change indicates that you must act on the update values. We will discuss these actions in the next section.

7.4 Understanding RPROP

In the previous sections, we examined the arguments, constants, and data structures necessary for RPROP. In this section, we will show you an iteration of RPROP. When we discussed backpropagation in earlier sections, we mentioned the online and batch weight update methods. However, RPROP does not support online training so all weight updates for RPROP will be performed in batch mode. As a result, each iteration of RPROP will receive gradients that are the sum of the individual gradients of each training set. This aspect is consistent with backpropagation in batch mode.

7.4.1 Determine Sign Change of Gradient

At this point, we have the gradients that are the same as the gradients calculated by the backpropagation algorithm. Because we use the same process to obtain gradients in both RPROP and backpropagation, we will not repeat it here. For the first step, we compare the gradient of the current iteration to the gradient of the previous iteration. If there is no previous iteration, then we can assume that the previous gradient was 0.

To determine whether the gradient sign has changed, we will use the sign (sgn) function. Equation 7.1 defines the sgn function:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases} \tag{7.1}$$

The sgn function returns the sign of the number provided. If x is less than 0, the result is -1. If x is greater than 0, then the result is 1. If x is equal to 0, then the result is 0. We usually implement the sgn function to use a tolerance for 0, since it is nearly impossible for floating-point operations to hit 0 precisely on a computer.

To determine whether the gradient has changed sign, we use Equation 7.2:

$$c = \frac{\partial E}{\partial w_{ij}}^{(t)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t-1)} \tag{7.2}$$

Equation 7.2 will result in a constant c. We evaluate this value as negative or positive or close to 0. A negative value for c indicates that the sign has changed. A positive value indicates that there is no change in sign for the gradient. A value near 0 indicates a very small change in sign or almost no change in sign.

Consider the following situations for these three outcomes:

```
-1 * 1 = -1  (negative, changed from negative to positive)
1 * 1 = 1 (positive, no change in sign)
1.0 * 0.000001 = 0.000001 (near zero, almost changed signs, but
    not quite)
```

Now that we have calculated the constant c, which gives some indication of sign change, we can calculate the weight change. The next section includes a discussion of this calculation.

7.4.2 Calculate Weight Change

Now that we have the change in sign of the gradient, we can observe what happens in each of the three cases mentioned in the previous section. Equation 7.3 summarizes these three cases:

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)} & \text{, if } c > 0 \\ +\Delta_{ij}^{(t)} & \text{, if } c < 0 \\ 0 & \text{, otherwise} \end{cases} \tag{7.3}$$

This equation calculates the actual weight change for each iteration. If the value of c is positive, then the weight change will be equal to the negative of the weight update value. Similarly, if the value of c is negative, the weight change will be equal to the positive of the weight update value. Finally, if the value of c is near 0, there will be no weight change.

7.4.3 Modify Update Values

We use the weight update values from the previous section to update the weights of the neural network. Every weight in the neural network has a separate weight update value that works much better than the single learning rate of backpropagation. We modify these weight update values during each training iteration, as seen in Equation 7.4:

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^{+} \cdot \Delta_{ij}^{(t-1)} & \text{, if } c > 0 \\ \eta^{-} \cdot \Delta_{ij}^{(t-1)} & \text{, if } c < 0 \\ \Delta_{ij}^{(t-1)} & \text{, otherwise} \end{cases} \tag{7.4}$$

We can modify the weight update values in a way that is very similar to the changes of the weights. We base these weight update values on the previously calculated value c, just like the weights.

If the value of c is positive, then we multiply the weight update value by the value of positive $+\eta$ (eta). Similarly, if the value of c is negative, we multiply the weight update value by negative -η (eta). Finally, if the value of c is near 0, then we don't change the weight update value.

The JavaScript example site for this book has examples of the RPROP update as well as examples of the previous equations and sample calculations.

7.5 Levenberg-Marquardt Algorithm

The Levenberg-Marquardt algorithm (LMA) is a very efficient training method for neural networks. In many cases, LMA will outperform RPROP. As a result, every neural network programmer should consider this training algorithm. Levenberg (1940) introduced the foundation for the LMA, and Marquardt (1963) expanded its methods.

LMA is a hybrid algorithm that is based on Newton's method (GNA) and on gradient descent (backpropagation). Thus, LMA combines the strengths of GNA and backpropagation. Although gradient descent is guaranteed to converge to a local minimum, it is slow. Newton's method is fast, but it often fails to converge. By using a damping factor to interpolate between the two, we create a hybrid method. To understand how this hybrid works, we will first examine Newton's method. Equation 7.5 shows Newton's method:

$$W_{min} = W_0 - H^{-1}g \tag{7.5}$$

You will notice several variables in the above equation. The result of the equation is that you can apply deltas to the weights of the neural network. The variable H represents the Hessian, which we will discuss in the next section. The variable g represents the gradients of the neural network. You will also notice the -1 "exponent" on the variable H, which specifies that we are doing a matrix decomposition of the variables H and g.

We could easily spend an entire chapter on matrix decomposition. However, we will simply treat matrix decomposition as a black box atomic operator for the purposes of this book. Because we will not explain how to calculate matrix decomposition, we have included a common piece of code taken from the JAMA package. Many mathematical computer applications have used this public domain code, adapted from a FORTRAN program. To perform matrix decomposition, you can use JAMA or another source.

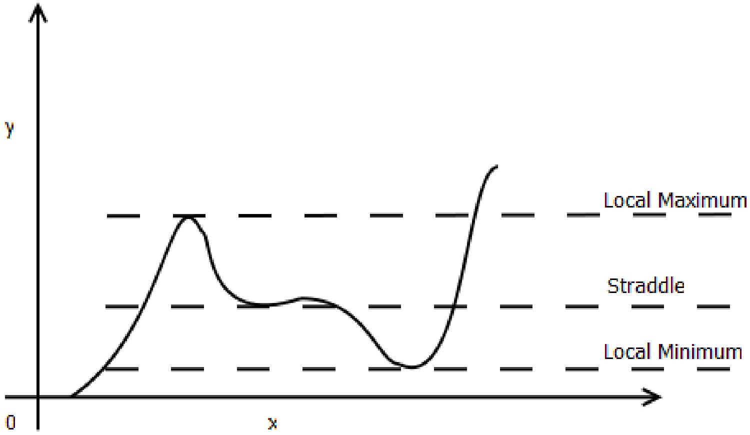
Although several types of matrix decomposition exist, we are going to use the LU decomposition, which requires a square matrix. This decomposition works well because the Hessian matrix has the same number of rows as columns. Every weight in the neural network has a row and column. The LU decomposition takes the Hessian, which is a matrix of the second derivative of the partial derivatives of the output of each of the weights. The LU decomposition solves the Hessian by the gradients, which are the square of the error of each weight. These gradients are the same as those that we calculated in Chapter 6, "Backpropagation Training," except they are squared. Because the errors are squared, we must use the sum of square error when dealing with LMA.

Second derivative is an important term to know. It is the derivative of the first derivative. Recall from Chapter 6, "Backpropagation Training," that the derivative of a function is the slope at any point. This slope shows the direction that the curve is approaching for a local minimum. The second derivative is also a slope, and it points in a direction to minimize the first derivative. The goal of Newton's method, as well as of the LMA, is to reduce all of the gradients to 0.

It's interesting to note that the goal does not include the error. Newton's method and LMA can be oblivious to the error because they try to reduce all the gradients to 0. In reality, they are not completely oblivious to the error because they use it to calculate the gradients.

Newton's method will converge the weights of a neural network to a local minimum, a local maximum, or a straddle position. We achieve this convergence by minimizing all the gradients (first derivatives) to 0. The derivatives will be 0 at local minima, maxima, or straddle position. Figure 7.1 shows these three points:

Figure 7.1: Local Minimum, Straddle and Local Maximum



The algorithm implementation must ensure that local maxima and straddle points are filtered out. The above algorithm works by taking the matrix decomposition of the Hessian matrix and the gradients. The Hessian matrix is typically estimated. Several methods exist to estimate the Hessian matrix. However, if it is inaccurate, it can harm Newton's method.

LMA enhances Newton's algorithm to the following formula in Equation 7.6:

$$W_{min} = W_0 - (H + \lambda I)^{-1} g \tag{7.6}$$

In this equation, we add a damping factor multiplied by an identity matrix. The damping factor is represented by λ (lambda), and I represents the identity matrix, which is a square matrix with all the values at 0 except for a northwest (NW) line of values at 1. As lambda increases, the Hessian will be factored out of the above equation. As lambda decreases, the Hessian becomes more significant than gradient descent, allowing the training algorithm to interpolate between gradient descent and Newton's method. Higher lambda favors gradient descent; lower lambda favors Newton. A training iteration of LMA begins with a low lambda and increases it until a desirable outcome is produced.

7.6 Calculation of the Hessian

The Hessian matrix is a square matrix with rows and columns equal to the number of weights in the neural network. Each cell in this matrix represents the second order derivative of the output of the neural network with respect to a given weight combination. Equation 7.7 shows the Hessian:

$$H(e) = \begin{bmatrix} \frac{\partial^2 e}{\partial w_1^2} & \frac{\partial^2 e}{\partial w_1 \, \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_1 \, \partial w_n} \\ \frac{\partial^2 e}{\partial w_2 \, \partial w_1} & \frac{\partial^2 e}{\partial w_2^2} & \cdots & \frac{\partial^2 e}{\partial w_2 \, \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 e}{\partial w_n \, \partial w_1} & \frac{\partial^2 e}{\partial w_n \, \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_n^2} \end{bmatrix} \tag{7.7}$$

It is important to note that the Hessian is symmetrical about the diagonal, which you can use to enhance performance of the calculation. Equation 7.8 calculates the Hessian by calculating the gradients:

$$\frac{\partial E}{\partial w_{(i)}} = 2(y - t)\frac{\partial y}{\partial w_{(i)}} \tag{7.8}$$

The second derivative of the above equation becomes an element of the Hessian matrix. You can use Equation 7.9 to calculate it:

$$\frac{\partial^2 E}{\partial w_i w_j} = 2\left(\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j} + (y - t)\frac{\partial^2 y}{\partial w_j \partial w_j}\right) \tag{7.9}$$

If not for the second component, you could easily calculate the above formula. However, this second component involves the second partial derivative and that is difficult to calculate. Because the component is not important, you can actually drop it because its value does not significantly contribute to the outcome. While the second partial derivative might be important for an individual training case, its overall contribution is not significant. The second component of Equation 7.9 is multiplied by the error of that training case. We assume that the errors in a training set are independent and evenly distributed about 0. On an entire training set, they should essentially cancel each other out. Because we are not using all components of the second derivative, we have only an approximation of the Hessian, which is sufficient to get a good training result.

Equation 7.10 uses the approximation, resulting in the following:

$$\frac{\partial^2 E}{\partial w_i w_j} = 2\left(\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j}\right) \tag{7.10}$$

While the above equation is only an approximation of the true Hessian, the simplification of the algorithm to calculate the second derivative is well worth the loss in accuracy. In fact, an increase in λ (lambda) will account for the loss of accuracy.

To calculate the Hessian and gradients, we must determine the partial first derivatives of the output of the neural network. Once we have these partial first derivatives, the above equations allow us to easily calculate the Hessian and gradients.

Calculation of the first derivatives of the output of the neural network is very similar to the process that we used to calculate the gradients for backpropagation. The main difference is that we take the derivative of the output. In standard backpropagation, we take the derivative of the error function. We will not review the entire backpropagation process here. Chapter 6, "Backpropagation Training," covers backpropagation and gradient calculation.

7.7 LMA with Multiple Outputs

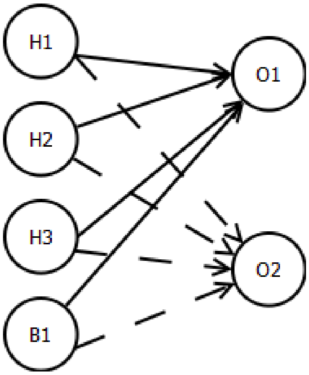
Some implementations of LMA support only a single-output neuron because LMA has roots in mathematical function approximation. In mathematics, functions typically return only a single value. As a result, many books and papers do not contain discussions of multiple-output LMA. However, you can use LMA with multiple outputs.

Support for multiple-output neurons involves summing each cell of the Hessian as you calculate the additional output neurons. The process works as if you calculated a separate Hessian matrix for each output neuron and then summed the Hessian matrices together. Encog (Heaton, 2015) uses this approach, and it leads to fast convergence times.

You need to realize that you will not use every connection with multiple outputs. You will need to calculate independently an update for the weight of each output neuron. Depending on the output neuron you are currently calculating, there will be unused connections for the other output neurons. Therefore, you must set the partial derivative for each of these unused connections to 0 when you are calculating the other output neurons.

For example, consider a neural network that has two output neurons and three hidden neurons. Each of these two output neurons would have a total of four connections from the hidden layer. Three connections result from the three hidden neurons, and a fourth comes from the bias neuron. This segment of the neural network would resemble Figure 7.2:

Figure 7.2: Calculating Output Neuron 1



Here we are calculating output neuron 1. Notice that output neuron 2 has four connections that must have their partial derivatives treated as 0. Because we are calculating output 1 as the current neuron, it only uses its normal partial derivatives. You can repeat this process for each output neuron.

7.8 Overview of the LMA Process

So far, we have examined only the math behind LMA. To be effective, LMA must be part of an algorithm. The following steps summarize the LMA process:

```
1. Calculate the first derivative of output of the neural network
   with respect to every weight.
2. Calculate the Hessian.
3. Calculate the gradients of the error (ESS) with respect to
   every weight.
4. Either set lambda to a low value (first iteration) or the
   lambda of the previous iteration.
5. Save the weights of the neural network.
6. Calculate delta weight based on the lambda, gradients, and
   Hessian.
7. Apply the deltas to the weights and evaluate error.
8. If error has improved, end the iteration.
9. If error has not improved, increase lambda (up to a max lambda)
   , restore the weights, and go back to step 6.
```

As you can see, the process for LMA revolves around setting the lambda value low and then slowly increasing it if the error rate does not improve. You must save the weights at each change in lambda so that you can restore them if the error does not improve.

7.9 Chapter Summary

Resilient propagation (RPROP) solves two limitations of simple backpropagation. First, the program assigns each weight a separate learning rate, allowing the weights to learn at different speeds. Secondly, RPROP recognizes that while the gradient's sign is a great indicator of the direction to move the weight, the size of the gradient does not indicate how far to move. Additionally, while the programmer must determine an appropriate learning rate and momentum for backpropagation, RPROP automatically sets similar arguments.

Genetic algorithms (GAs) are another means of training neural networks. There is an entire family of neural networks that use GAs to evolve every aspect of the neural network, from weights to the overall structure. This family includes the NEAT, CPPN and HyperNEAT neural networks that we will discuss in the next chapter. The GA used by NEAT, CPPN and HyperNEAT is not just another training algorithm because these neural networks introduce a new architecture based on the feedforward neural networks examined so far in this book.

Chapter 8

NEAT, CPPN & HyperNEAT

- NEAT
- Genetic Algorithms
- CPPN
- HyperNEAT

In this chapter, we discuss three closely related neural network technologies: NEAT, CPPN and HyperNEAT. Kenneth Stanley's EPLEX group at the University of Central Florida conducts extensive research for all three technologies. Information about their current research can be found at the following URL:

`http://eplex.cs.ucf.edu/`

NeuroEvolution of Augmenting Topologies (NEAT) is an algorithm that evolves neural network structures with genetic algorithms. The compositional pattern-producing network (CPPN) is a type of evolved neural network that can create other structures, such as images or other neural networks. Hypercube-based NEAT, or HyperNEAT, a type of CPPN, also evolves other neural networks. Once HyperNEAT train the networks, they can easily handle much higher resolutions of their dimensions.

Many different frameworks support NEAT and HyperNEAT. For Java and C#, we recommend our own Encog implementation, which can be found at the following URL:

`http://www.encog.org`

You can find a complete list of NEAT implementations at Kenneth Stanley's website:

`http://www.cs.ucf.edu/~kstanley/neat.html`

Kenneth Stanley's website also includes a complete list of HyperNEAT implementations:

`http://eplex.cs.ucf.edu/hyperNEATpage/`

For the remainder of this chapter, we will explore each of these three network types.

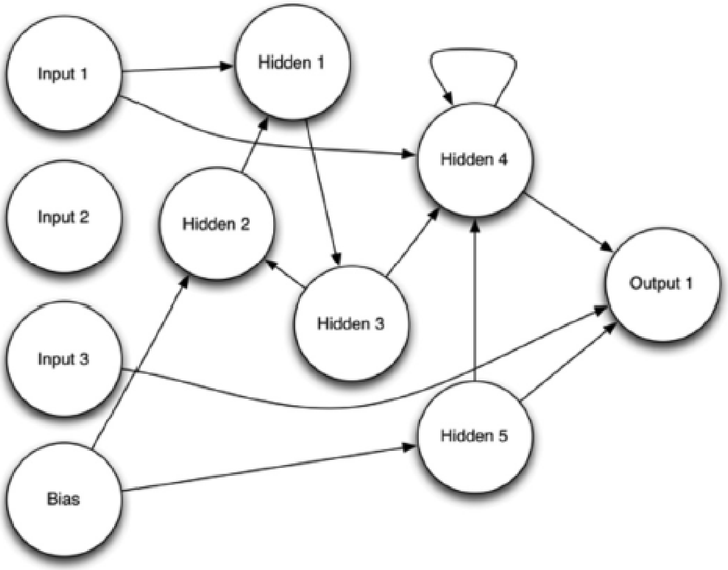
8.1 NEAT Networks

NEAT is a neural network structure developed by Stanley and Miikkulainen (2002). NEAT optimizes both the structure and weights of a neural network with a genetic algorithm (GA). The input and output of a NEAT neural network are identical to a typical feedforward neural network, as seen in previous chapters of this book.

A NEAT network starts out with only bias neurons, input neurons, and output neurons. Generally, none of the neurons have connections at the outset. Of course, a completely unconnected network is useless. NEAT makes no assumptions about whether certain input neurons are actually needed. An unneeded input is said to be statistically independent of the output. NEAT will often discover this independence by never evolving optimal genomes to connect to that statistically independent input neuron.

Another important difference between a NEAT network and an ordinary feedforward neural network is that other than the input and output layers, NEAT networks do not have clearly defined hidden layers. However, the hidden neurons do not organize themselves into clearly delineated layers. One similarity between NEAT and feedforward networks is that they both use a sigmoid activation function. Figure 8.1 shows an evolved NEAT network:

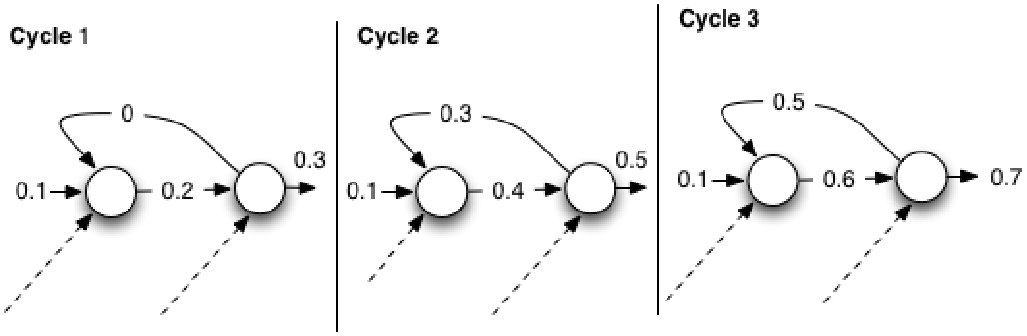
Figure 8.1: NEAT Network



Input 2 in the above image never formed any connections because the evolutionary process determined that input 2 was unnecessary. A recurrent connection also exists between hidden 3 and hidden 2. Hidden 4 has a recurrent connection to itself. Overall, you will note that a NEAT network lacks a clear delineation of layers.

You can calculate a NEAT network in exactly the same way as you do for a regular weighted feedforward network. You can manage the recurrent connections by running the NEAT network multiple times. This works by having the recurrent connection input start at 0 and update them each type you cycle through the NEAT network. Additionally, you must define a hyperparameter to specify the number of times to calculate the NEAT network. Figure 8.2 shows recurrent link calculation when a NEAT network is instructed to cycle three times to calculate recurrent connections:

Figure 8.2: Cycling to Calculate Recurrences



The above diagram shows the outputs from each neuron, over each connection, for three cycles. The dashed lines indicate the additional connections. For simplicity, the diagram doesn't have the weights. The purpose of Figure 8.2 is to show that the recurrent output stays one cycle behind.

For the first cycle, the recurrent connection provided a 0 to the first neuron because neurons are calculated left to right. The first cycle has no value for the recurrent connection. For the second cycle, the recurrent connection now has the output 0.3, which the first cycle provided. Cycle 3 follows the same pattern, taking the 0.5 output from cycle 2 as the recurrent connection's output. Since there would be other neurons in the calculation, we have contrived these values, which the dashed arrows show at the bottom. However, Figure 8.2 does illustrate that the recurrent connections are cycled through previous cycles.

NEAT networks extensively use genetic algorithms, which we examined in *Artificial Intelligence for Humans, Volume 2: Nature-Inspired Algorithms.* Although you do not need to understand completely genetic algorithms to follow the discussion of them in this chapter, you can refer to Volume 2, as needed.

NEAT uses a typical genetic algorithm that includes:

- **Mutation** - The program chooses one fit individual to create a new individual that has a random change from its parent.
- **Crossover** - The program chooses two fit individuals to create a new individual that has a random sampling of elements from both parents.

All genetic algorithms engage the mutation and crossover genetic operators with a population of individual solutions. Mutation and crossover choose with greater probability the solutions that receive higher scores from an objective function. We explore mutation and crossover for NEAT networks in the next two sections.

8.1.1 NEAT Mutation

NEAT mutation consists of several mutation operations that can be performed on the parent genome. We discuss these operations here:

- **Add a neuron**: By selecting a random link, we can add a neuron. A new neuron and two links replace this random link. The new neuron effectively splits the link. The program selects the weights of each of the two new links to provide nearly the same effective output as the link being replaced.
- **Add a link**: The program chooses a source and destination, or two random neurons. The new link will be between these two neurons. Bias neurons can never be a destination. Output neurons cannot be a source. There will never be more than two links in the same direction between the same two neurons.

- **Remove a link**: Links can be randomly selected for removal. If there are no remaining links interacting with them, you can remove the hidden neurons, which are neurons that are not input, output, or the single bias neuron.

- **Perturb a weight**: You can choose a random link. Then multiply its weight by a number from a normal random distribution with a gamma of 1 or lower. Smaller random numbers will usually cause a quicker convergence. A gamma value of 1 or lower will specify that a single standard deviation will sample a random number of 1 or lower.

You can increase the probability of the mutation so that the weight perturbation occurs more frequently, thereby allowing fit genomes to vary their weights and further adapt through their children. The structural mutations happen with much less frequency. You can adjust the exact frequency of each operation with most NEAT implementations.

8.1.2 NEAT Crossover

NEAT crossover is more complex than many genetic algorithms because the NEAT genome is an encoding of the neurons and connections that comprise an individual genome. Most genetic algorithms assume that the number of genes is consistent across all genomes in the population. In fact, child genomes in NEAT that result from both mutation and crossover may have a different number of genes than their parents. Managing this number discrepancy requires some ingenuity when you implement the NEAT crossover operation.

NEAT keeps a database of all the changes made to a genome through mutation. These changes are called innovations, and they exist in order to implement mutations. Each time an innovation is added, it is given an ID. These IDs will also be used to order the innovations. We will see that it is important to select the innovation with the lower ID when choosing between two innovations.

It is important to realize that the relationship between innovations and mutations is not one to one. It can take several innovations to achieve one mutation. The only two types of innovation are creating a neuron and a link between two neurons. One mutation might result from multiple innovations. Additionally, a mutation might not have any innovations. Only mutations that add to the structure of the network will generate innovations. The following list summarizes the innovations that the previously mentioned mutation types could potentially create.

- **Add a neuron**: One new neuron innovation and two new link innovations
- **Add a link**: One new link innovation
- **Remove a link**: No innovations
- **Perturb a weight**: No innovations

You also need to note that NEAT will not recreate innovation records if you have already attempted this type of innovation. Furthermore, innovations do not contain any weight information; innovations only contain structural information.

Crossover for two genomes occurs by considering the innovations, and this trait allows NEAT to ensure that all prerequisite innovations are also present. A naïve crossover, such as those that many genetic algorithms use, would potentially combine links with nonexistent neurons. Listing 8.1 shows the entire NEAT crossover function in pseudocode:

Listing 8.1: NEAT Crossover

```
def neat_crossover(rnd,mom,dad):
# Choose best genome (by objective function), if tie, choose
    random.
  best = favor_parent(rnd, mom, dad)
  not_best = dad if (best <> mom) else mom
  selected_links = []
  selected_neurons = []
# current gene index from mom and dad
  cur_mom = 0
  cur_dad = 0
  selected_gene = None
# add in the input and bias, they should always be here
always_count = mom.input_count + mom.output_count + 1
```

```
for i from 0 to always_count -1:
  selected_neurons.add(i, best, not_best)
# Loop over all genes in both mother and father
  while (cur_mom < mom.num_genes) or (cur_dad < dad.num_genes):
# The mom and dad gene object
    mom_gene = None
    mom_innovation = -1
    dad_gene = None
    dad_innovation = -1
# grab the actual objects from mom and dad for the specified
# indexes
# if there are none, then None
    if cur_mom < mom.num_genes:
      mom_gene = mom.links[cur_mom];
      mom_innovation = mom_gene.innovation_id
    if cur_dad < dad.num_genes:
      dad_gene = dad.links[cur_dad]
      dad_innovation_id = dad_gene.innovation_id
# now select a gene fror mom or dad. This gene is for the baby
# Dad gene only, mom has run out
    if mom_gene == None and dad_gene <> None:
      cur_dad = cur_dad + 1
      selected_gene = dad_gene
# Mom gene only, dad has run out
    else if dadGene == null and momGene <> null:
      cur_mom = cur_mom + 1
      selected_gene = mom_gene
# Mom has lower innovation number
    else if mom_innovation_id < dad_innovation_id:
      cur_mom = cur_mom + 1
      if best == mom:
        selected_gene = mom_gene
# Dad has lower innovation number
    else if dad_innovation_id < mom_innovation_id:
      cur_dad = cur_dad + 1
      if best == dad:
        selected_gene = dad_gene
# Mom and dad have the same innovation number
# Flip a coin.
    else if dad_innovation_id == mom_innovation_id:
      cur_dad = cur_dad + 1
      cur_mom = cur_mom + 1
      if rnd.next_double() >0.5:
        selected_gene = dad_gene
      else:
        selected_gene = mom_gene
# If a gene was chosen for the child then process it.
# If not, the loop continues.
    if selected_gene <> None:
# Do not add the same innovation twice in a row.
      if selected_links.count == 0:
        selected_links.add(selected_gene)
      else:
        if selected_links[selected_links.count-1]
            .innovation_id <> selected_gene.innovation_id {
          selected_links.add(selected_gene)
# Check if we already have the nodes referred to in
# SelectedGene.
# If not, they need to be added.
      selected_neurons.add(
        selected_gene.from_neuron_id, best, not_best)
      selected_neurons.add(
        selected_gene.to_neuron_id, best, not_best)
# Done looping over parent's genes
  baby = new NEATGenome(selected_links, selected_neurons)
  return baby
```

The above implementation of crossover is based on the NEAT crossover operator implemented in Encog. We provide the above comments in order to explain the critical sections of code. The primary evolution occurs on the links contained in the mother and father. Any neurons needed to support these links are brought along when the child genome is created. The code contains a main loop that loops over both parents, thereby selecting the most suitable link gene from each parent. The link genes from both parents are essentially stitched together so they can find the most suitable gene. Because the parents might be different lengths, one will likely exhaust its genes before this process is complete.

Each time through the loop, a gene is chosen from either the mother or father according to the following criteria:

- If mom or dad has run out, choose the other. Move past the chosen gene.
- If mom has a lower innovation ID number, choose mom if she has the best score. In either case, move past mom's gene.
- If dad has a lower innovation ID number, choose dad if he has the best score. In either case, move past dad's gene.
- If mom and dad have the same innovation ID, pick one randomly, and move past their gene.

You can consider that the mother and father's genes are both on a long tape. A marker for each tape holds the current position. According to the rules above, the marker will move past a parent's gene. At some point, each parent's marker moves to the end of the tape, and that parent runs out of genes.

8.1.3 NEAT Speciation

Crossover is a tricky for computers to properly perform. In the animal and plant kingdoms, crossover occurs only between members of the same species. What exactly do we mean by species? In biology, scientists define species as members of a population that can produce viable offspring. Therefore, a crossover between a horse and humming bird genome would be catastrophically unsuccessful. Yet a naive genetic algorithm would certainly try something just as disastrous with artificial computer genomes!

The NEAT speciation algorithm has several variants. In fact, one of the most advanced variants can group the population into a predefined number of clusters with a type of k-means clustering. You can subsequently determine the relative fitness of each species. The program gives each species a percentage of the next generation's population count. The members of each species then compete in virtual tournaments to determine which members of the species will be involved in crossover and mutation for the next generation.

A tournament is an effective way to select parents from a species. The program performs a certain number of trials. Typically we use five trials. For each trial, the program selects two random genomes from the species. The fitter of each genome advances to the next trial. This process is very efficient for threading, and it is also biologically plausible. The advantage to this selection method is that the winner doesn't have to beat the best genome in the species. It has to beat the best genome in the trials. You must run a tournament for each parent needed. Mutation requires one parent, and crossover needs two parents.

In addition to the trials, several other factors determine the species members chosen for mutation and crossover. The algorithm will always carry one or more elite genomes to the next species. The number of elite genomes is configurable. The program gives younger genomes a bonus so they have a chance to try new innovations. Interspecies crossover will occur with a very low probability.

All of these factors together make NEAT a very effective neural network type. NEAT removes the need to define how the hidden layers of a neural network are structured. The absence of a strict structure of hidden layers allows NEAT neural networks to evolve the connections that are actually needed.

8.2 CPPN Networks

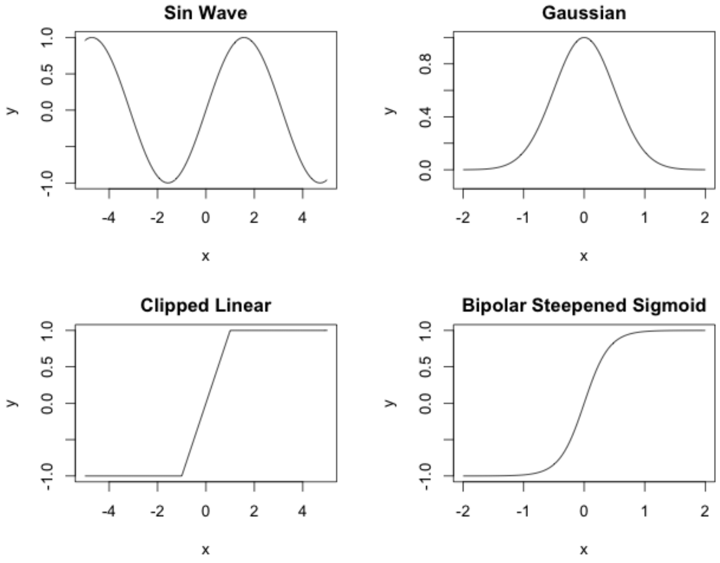
The compositional pattern-producing network (CPPN) was invented by Stanley (2007) and is a variation of the artificial neural network. CPPN recognizes one biologically plausible fact. In nature, genotypes and phenotypes are not identical. In other words, the genotype is the DNA blueprint for an organism. The phenotype is what actually results from that plan.

In nature, the genome is the instructions for producing a phenotype that is much more complex than the genotype. In the original NEAT, as seen in the last section, the genome describes link for link and neuron for neuron how to produce the phenotype. However, CPPN is different because it creates a population of special NEAT genomes. These genomes are special in two ways. First, CPPN doesn't have the limitations of regular NEAT, which always uses a sigmoid activation function. CPPN can use any of the following activation functions:

- Clipped linear
- Bipolar steepened sigmoid
- Gaussian
- Sine
- Others you might define

You can see these activation functions in Figure 8.3:

Figure 8.3: CPPN Activation Functions



The second difference is that the NEAT networks produced by these genomes are not the final product. They are not the phenotype. However, these NEAT genomes do know how to create the final product.

The final phenotype is a regular NEAT network with a sigmoid activation function. We can use the above four activation functions only for the genomes. The ultimate phenotype always has a sigmoid activation function.

8.2.1 CPPN Phenotype

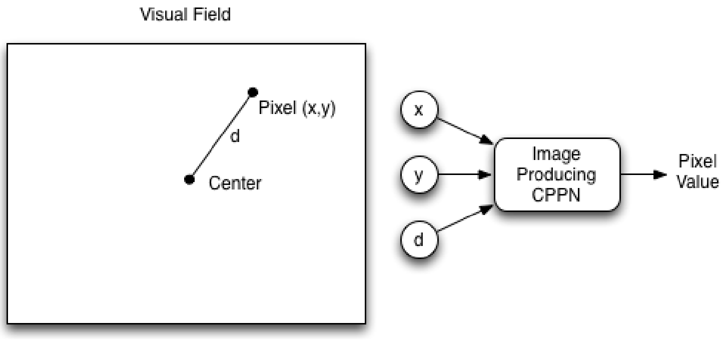
CPPNs are typically used in conjunction with images, as the CPPN phenotype is usually an image. Though images are the usual product of a CPPN, the only real requirement is that the CPPN compose something, thereby earning its name of compositional pattern-producing network. There are cases where a CPPN does not produce an image. The most popular non-image producing CPPN is HyperNEAT, which is discussed in the next section.

Creating a genome neural network to produce a phenotype neural network is a complex but worthwhile endeavor. Because we are dealing with a large number of input and output neurons, the training times can be considerable. However, CPPNs are scalable and can reduce the training times.

Once you have evolved a CPPN to create an image, the size of the image (the phenotype) does not matter. It can be 320x200, 640x480 or some other resolution altogether. The image phenotype, generated by the CPPN will grow to the size needed. As we will see in the next section, CPPNs give HyperNEAT the same sort of scalability.

We will now look at how a CPPN, which is itself a NEAT network, produces an image, or the final phenotype. The NEAT CPPN should have three input values: the coordinate on the horizontal axis (x), the coordinate on the vertical axis (y), and the distance of the current coordinate from the center (d). Inputting d provides a bias towards symmetry. In biological genomes, symmetry is important. The output from the CPPN corresponds to the pixel color at the x-coordinate and y-coordinate. The CPPN specification only determines how to process a grayscale image with a single output that indicates intensity. For a full-color image, you could use output neurons for red, green, and blue. Figure 8.4 shows a CPPN for images:

Figure 8.4: CPPN for Images



You can query the above CPPN for every x-coordinate and y-coordinate needed. Listing 8.2 shows the pseudocode that you can use to generate the phenotype:

Listing 8.2: Generate CPPN Image

```
def render_cppn(net,bitmap):
  for y from 1 to bitmap.height:
    for x from 1 to bitmap.width:
# Normalize x and y to -1,1
      norm_x = (2*(x/bitmap.width))-1
      norm_y = (2*(y/bitmap.height))-1
# Distance from center
      d = sqrt( (norm_x/2)^2
    + (norm_y /2)^2 )
```

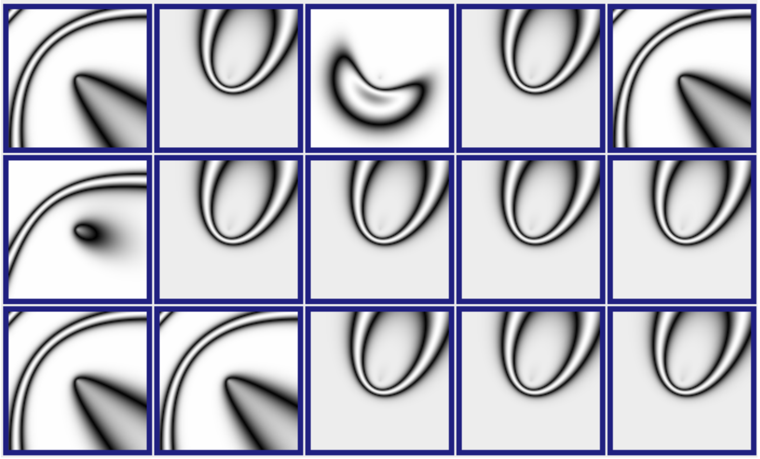
```
# Call CPPN
      input = [x,y,d]
  color = net.compute(input)
# Output pixel
  bitmap.plot(x-1,y-1, color)
```

The above code simply loops over every pixel and queries the CPPN for the color at that location. The x-coordinate and y-coordinate are normalized to being between -1 and +1. You can see this process in action at the Picbreeder website at following URL:

`http://picbreeder.org/`

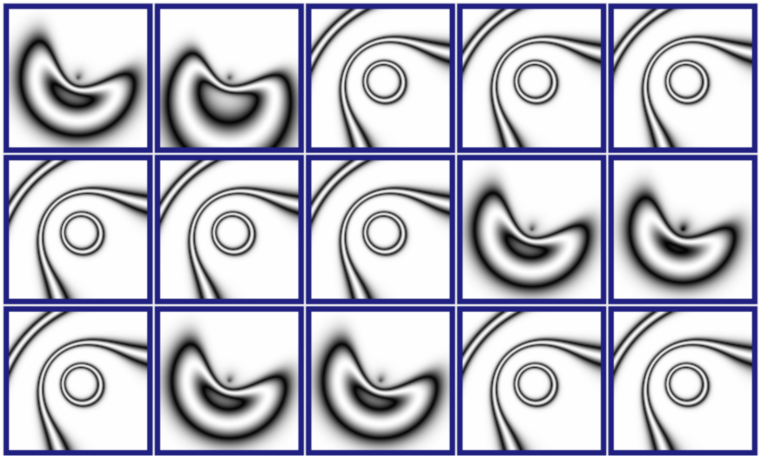
Depending on the complexity of the CPPN, this process can produce images similar to Figure 8.5:

Figure 8.5: A CPPN-Produced Image (picbreeder.org)



Picbreeder allows you to select one or more parents to contribute to the next generation. We selected the image that resembles a mouth, as well as the image to the right. Figure 8.6 shows the subsequent generation that Picbreeder produced.

Figure 8.6: A CPPN-Produced Image (picbreeder.org)



CPPN networks handle symmetry just like human bodies. With two hands, two kidneys, two feet, and other body part pairs, the human genome seems to have a hierarchy of repeated features. Instructions for creating an eye or various tissues do not exist. Fundamentally, the human genome does not have to describe every detail of an adult human being. Rather, the human genome only has to describe how to build an adult human being by generalizing many of the steps. This greatly simplifies the amount of information that is needed in a genome.

Another great feature of the image CPPN is that you can create the above images at any resolution and without retraining. Because the x-coordinate and y-coordinate are normalized to between -1 and +1, you can use any resolution.

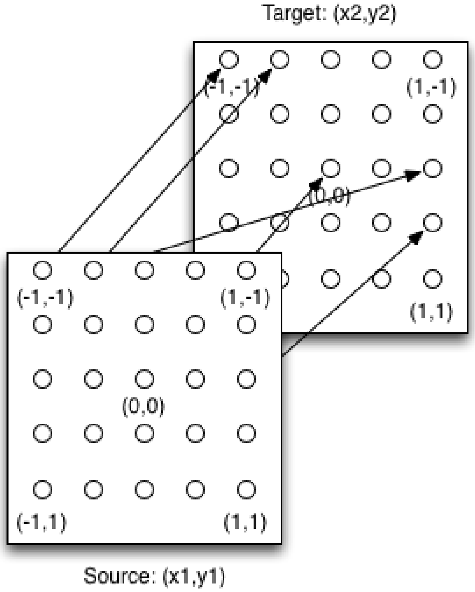
8.3 HyperNEAT Networks

HyperNEAT networks, invented by Stanley, D'Ambrosio, & Gauci (2009), are based upon the CPPN; however, instead of producing an image, a HyperNEAT network creates another neural network. Just like the CPPN in the last section, HyperNEAT can easily create much higher resolution neural networks without retraining.

8.3.1 HyperNEAT Substrate

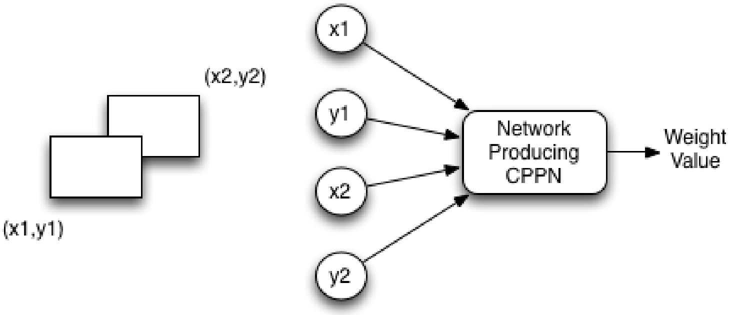
One interesting hyper-parameter of the HyperNEAT network is the substrate that defines the structure of a HyperNEAT network. A substrate defines the x-coordinate and the y-coordinate for the input and output neurons. Standard HyperNEAT networks usually employ two planes to implement the substrate. Figure 8.7 shows the sandwich substrate, one of the most common substrates:

Figure 8.7: HyperNEAT Sandwich Substrate



Together with the above substrate, a HyperNEAT CPPN is capable of creating the phenotype neural network. The source plane contains the input neurons, and the target plane contains the output neurons. The x-coordinate and the y-coordinate for each are in the -1 to +1 range. There can potentially be a weight between each of the source neurons and every target neuron. Figure 8.8 shows how to query the CPPN to determine these weights:

Figure 8.8: CPPN for HyperNEAT

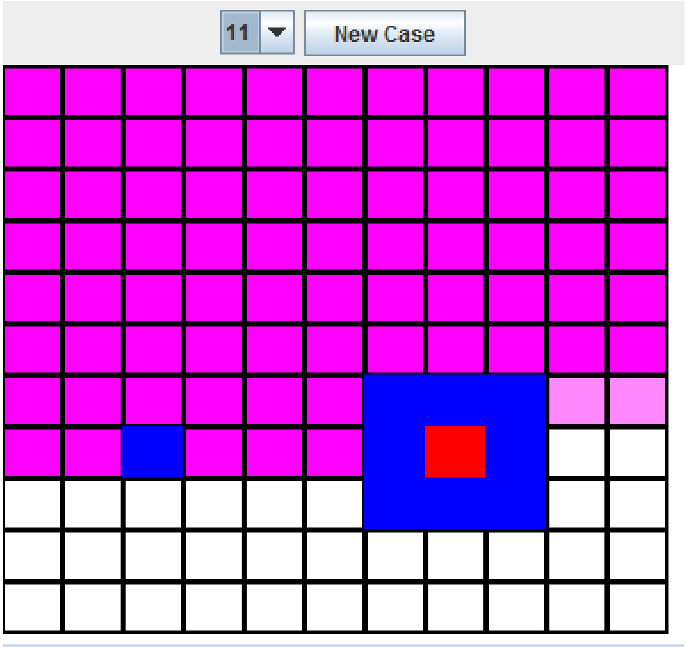


The input to the CPPN consists of four values: *x1*, *y1*, *x2*, and *y2*. The first two values *x1* and *y1* specify the input neuron on the source plane. The second two values *x2* and *y2* specify the input neuron on the target plane. HyperNEAT allows the presence of as many different input and output neurons as desired, without retraining. Just like the CPPN image could map more and more pixels between -1 and +1, so too can HyperNEAT pack in more input and output neurons.

8.3.2 HyperNEAT Computer Vision

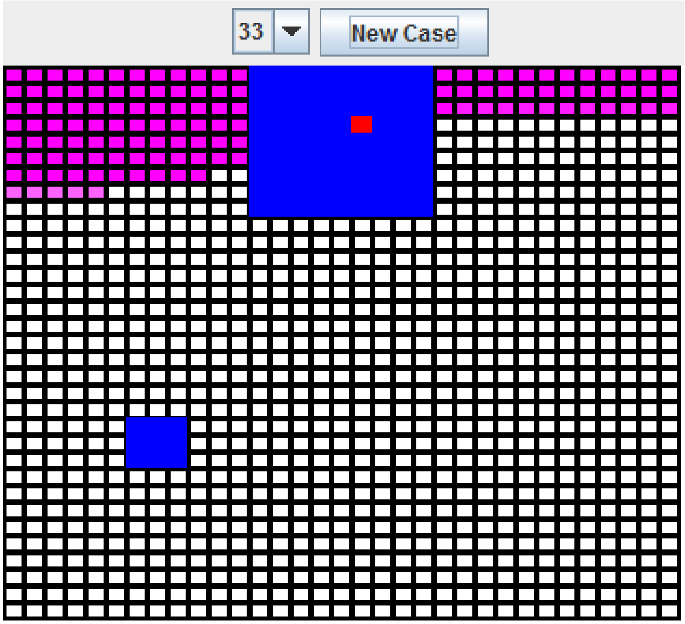
Computer vision is a great application of HyperNEAT, as demonstrated by the rectangles experiment provided in the original HyperNEAT paper by Stanley, Kenneth O., et al. (2009). This experiment placed two rectangles in a computer's vision field. Of these two rectangles, one is always larger than the other. The neural network is trained to place a red rectangle near the center of the larger rectangle. Figure 8.9 shows this experiment running under the Encog framework:

Figure 8.9: Boxes Experiment (11 resolution)



As you can see from the above image, the red rectangle is placed directly inside of the larger of the two rectangles. The "New Case" button can be pressed to move the rectangles, and the program correctly finds the larger rectangle. While this works quite well at 11x11, the size can be increased to 33x33. With the larger size, no retraining is needed, as shown in Figure 8.10:

Figure 8.10: Boxes Experiment (33 resolution)



When the dimensions are increased to 33x33, the neural network is still able to place the red square inside of the larger rectangle.

The above example uses a sandwich substrate with the input and output plane both equal to the size of the visual field, in this case 33x33. The input plane provides the visual field. The neuron in the output plane with the highest output is the program's guess at the center of the larger rectangle. The fact that the position of the large rectangle does not confuse the network shows that HyperNEAT possesses some of the same features as the convolutional neural networks that we will see in Chapter 10, "Convolutional Networks."

8.4 Chapter Summary

This chapter introduced NEAT, CPPN, and HyperNEAT. Kenneth Stanley's EPLEX group at the University of Central Florida extensively researches all three technologies. NeuroEvolution of Augmenting Topologies (NEAT) is an algorithm that uses genetic algorithms to automatically evolve neural network structures. Often the decision of the structure of a neural network can be one of the most complex aspects of neural network design. NEAT neural networks can evolve their own structure and even decide what input features are important.

The compositional pattern-producing network (CPPN) is a type of neural network that is evolved to create other structures, such as images or other neural networks. Image generation is a common task for CPPNs. The Picbreeder website allows new images to be bred based on previous images generated at this site. CPPNs can generate more than just images. The HyperNEAT algorithm is an application of CPPNs for producing neural networks.

Hypercube-based NEAT, or HyperNEAT, is a type of CPPN that evolves other neural networks that can easily handle much higher resolutions of their dimensions as soon as they are trained. HyperNEAT allows a CPPN to be evolved that can create neural networks. Being able to generate the neural network allows you to introduce symmetry, and it gives you the ability to change the resolution of the problem without retraining.

Neural networks have risen and declined in popularity several times since their introduction. Currently, there is interest in neural networks that use deep learning. In fact, deep learning involves several different concepts. The next chapter introduces deep neural networks, and we expand this topic throughout the remainder of this book.

Chapter 9

Deep Learning

- Convolutional Neural Networks & Dropout
- Tools for Deep Learning
- Contrastive Divergence
- Gibb's Sampling

Deep learning is a relatively new advancement in neural network programming and represents a way to train deep neural networks. Essentially, any neural network with more than two layers is deep. The ability to create deep neural networks has existed since Pitts (1943) introduced the multilayer perceptron. However, we haven't been able to effectively train neural networks until Hinton (1984) became the first researcher to successfully train these complex neural networks.

9.1 Deep Learning Components

Deep learning is comprised of a number of different technologies, and this chapter is an overview of these technologies. Subsequent chapters will contain more information on these technologies. Deep learning typically includes the following features:

- Partially Labeled Data
- Rectified Linear Units (ReLU)
- Convolutional Neural Networks
- Dropout

The succeeding sections provide an overview of these technologies.

9.2 Partially Labeled Data

Most learning algorithms are either supervised or unsupervised. Supervised training data sets provide an expected outcome for each data item. Unsupervised training data sets do not provide an expected outcome, which is called a label. The problem is that most data sets are a mixture of labeled and unlabeled data items.

To understand the difference between labeled and unlabeled data, consider the following real-life example. When you were a child, you probably saw many vehicles as you grew up. Early in your life, you did not know if you were seeing a car, truck, or van. You simply knew that you were seeing some sort of vehicle. You can consider this exposure as the unsupervised part of your vehicle-learning journey. At that point, you learned commonalities of features among these vehicles.

Later in your learning journey, you were given labels. As you encountered different vehicles, an adult told you that you were looking at a car, truck, or van. The unsupervised training created your foundation, and you built upon that knowledge. As you can see, supervised and unsupervised learning are very common in real life. In its own way, deep learning does well with a combination of unsupervised and supervised learning data.

Some deep learning architectures handle partially labeled data and initialize the weights by using the entire training set without the outcomes. You can independently train the individual layers without the labels. Because you can train the layers in parallel, this process is scalable. Once the unsupervised phase has initialized these weights, the supervised phase can tweak them.

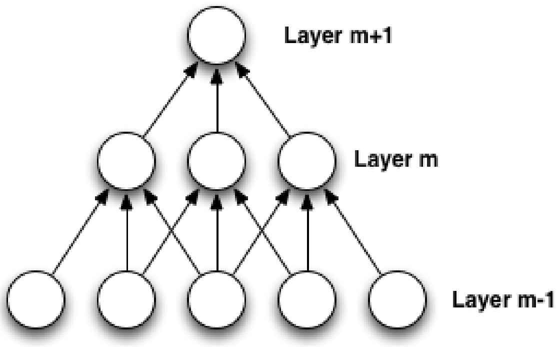
9.3 Rectified Linear Units

The Rectified linear unit (ReLU) has become the standard activation function for the hidden layers of a deep neural network. However, the restricted Boltzmann machine (RBM) is the standard for the deep belief neural network (DBNN). In addition to the ReLU activation functions for the hidden layers, deep neural networks will use a linear or softmax activation function for the output layer, depending on if the neural network supports regression or classification. We introduced ReLUs in Chapter 1, “Neural Network Basics,” and expanded upon this information in “Chapter 6, Backpropagation Training.”

9.4 Convolutional Neural Networks

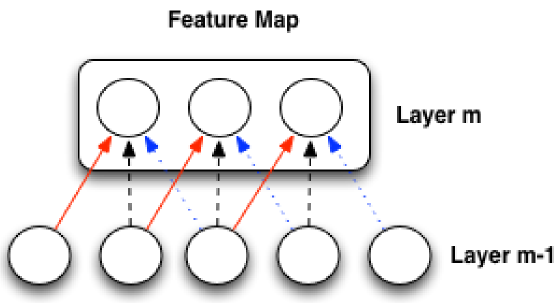
Convolution is an important technology that is often combined with deep learning. Hinton (2014) introduced convolution to allow image-recognition networks to function similarly to biological systems and achieve more accurate results. One approach is sparse connectivity in which we do not create every possible weight. Figure 9.1 shows sparse connectivity:

Figure 9.1: Sparse Connectivity



A regular feedforward neural network usually creates every possible weight connection between two layers. In deep learning terminology, we refer to these layers as dense layers. In addition to not representing every weight possible, convolutional neural networks will also share weights, as seen in Figure 9.2:

Figure 9.2: Shared Weights



As you can see in the above figure, the neurons share only three individual weights. The red (solid), black (dashed), and blue (dotted) lines indicate the individual weights. Sharing weights allows the program to store complex structures while maintaining memory and computation efficiency.

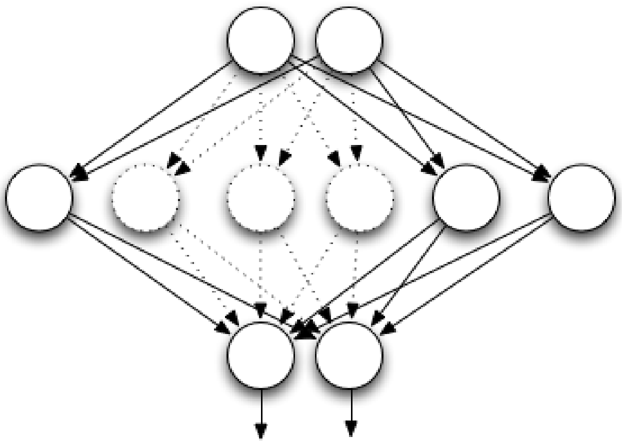
This section presented an overview of convolutional neural networks. Chapter 10, "Convolutional Neural Networks," is devoted entirely to this network type.

9.5 Neuron Dropout

Dropout is a regularization technique that holds many benefits for deep learning. Like most regularization techniques, dropout can prevent overfitting. You can also apply dropout to a neural network in a layer-by-layer fashion as you do in convolution. You must designate a single layer as a dropout layer. In fact, you can mix these dropout layers with regular layers and convolutional layers in the neural network. Never mix the dropout and convolutional layers within a single layer.

Hinton (2012) introduced dropout as a simple and effective regularization algorithm to reduce overfitting. Dropout works by removing certain neurons in the dropout layer. The act of dropping these neurons prevents other neurons from becoming overly dependent on the dropped neurons. The program removes these chosen neurons, along with all of their connections. Figure 9.3 illustrates this process:

Figure 9.3: Dropout Layer



From left to right, the above neural network contains an input layer, a dropout layer, and an output layer. The dropout layer has removed several of the neurons. The circles, made of dotted lines, indicate the neurons that the dropout algorithm removed. The dashed connector lines indicate the weights that the dropout algorithm removed when it eliminated the neurons.

Both dropout and other forms of regularization are extensive topics in the field of neural networks. Chapter 12, "Dropout and Regularization," covers regularization with particular focus on dropout. That chapter also contains an explanation on the L1 and L2 regularization algorithms. L1 and L2 discourage neural networks from the excessive use of large weights and the inclusion of certain irrelevant inputs. Essentially, a single neural network commonly uses dropout as well as other regularization algorithms.

9.6 GPU Training

Hinton (1987) introduced a very novel way to train the deep belief neural network (DBNN) efficiently. We examine this algorithm and DBNNs later in this chapter. As mentioned previously, deep neural networks have existed almost as long as the neural network. However, until Hinton's algorithm, no effective way to train deep neural networks existed. The backpropagation algorithms are very slow, and the vanishing gradient problem hinders the training.

The graphics processing unit (GPU), the part of the computer that is responsible for graphics display, is the way that researchers solved the training problem of feedforward neural networks. Most of us are familiar with GPUs because of modern video games that utilize 3D graphics. Rendering these graphical images is mathematically intense, and, to perform these operations, early computers relied on the central processing unit (CPU). However, this approach was not effective. The graphics systems in modern video games require dedicated circuitry, which became the GPU, or video card. Essentially, modern GPUs are computers that function within your computer.

As researchers discovered, the processing power contained in a GPU can be harnessed for mathematically intense tasks, such as neural network training. We refer to this utilization of the GPU for general computing tasks, aside from computer graphics, as general-purpose use of the GPU (GPGPU). When applied to deep learning, the GPU performs extraordinarily well. Combining it with ReLU activation functions, regularization, and regular backpropagation can produce amazing results.

However, GPGPU can be difficult to use. Programs written for the GPU must employ a very low-level programming language called C99. This language is very similar to the regular C programming language. However, in many ways, the C99 required by the GPU is much more difficult than the regular C programming language. Furthermore, GPUs are good only at certain tasks–even those conducive to the GPU because optimizing the C99 code is challenging. GPUs must balance several classes of memory, registers, and synchronization of hundreds of processor cores. Additionally, GPU processing has two competing standards–CUDA and OpenCL. Two standards create more processes for the programmer to learn.

Fortunately, you do not need to learn GPU programming to exploit its processing power. Unless you are willing to devote a considerable amount of effort to learn the nuances of a complex and evolving field, we do not recommend that you learn to program the GPU because it is quite different from CPU programming. Good techniques that produce efficient, CPU-based programs will often produce horribly inefficient GPU programs. The reverse is also true. If you would like to use GPU, you should work with an off-the-shelf package that supports it. If your needs do not fit into a deep learning package, you might consider using a linear algebra package, such as CUBLAS, which contains many highly optimized algorithms for the sorts of linear algebra that machine learning commonly requires.

The processing power of a highly optimized framework for deep learning and a fast GPU can be amazing. GPUs can achieve outstanding results based on sheer processing power. In 2010, the Swiss AI Lab IDSIA showed that, despite the vanishing gradient problem, the superior processing power of GPUs made backpropagation feasible for deep feedforward neural networks (Ciresan et al., 2010). The method outperformed all other machine learning techniques on the famous MNIST handwritten digit problem.

9.7 Tools for Deep Learning

One of the primary challenges of deep learning is the processing time to train a network. We often run training algorithms for many hours, or even days, seeking neural networks that fit well to the data sets. We use several frameworks for our research and predictive modeling. The examples in this book also utilize these frameworks, and we will present all of these algorithms in sufficient detail for you to create your own implementation. However, unless your goal is to conduct research to enhance deep learning itself, you are best served by working with an established framework. Most of these frameworks are tuned to train very fast.

We can divide the examples from this book into two groups. The first group shows you how to implement a neural network or to train an algorithm. However, most of the examples in this book are based on algorithms, and we examine the algorithm at its lowest level.

Application examples are the second type of example contained in this book. These higher-level examples show how to use neural network and deep learning algorithms. These examples will usually utilize one of the frameworks discussed in this section. In this way, the book strikes a balance between theory and real-world application.

9.7.1 H2O

H2O is a machine learning framework that supports a wide variety of programming languages. Though H2O is implemented in Java, it is designed as a web service. H2O can be used with R, Python, Scala, Java, and any language that can communicate with H2O's REST API.

Additionally, H2O can be used with Apache Spark for big data and big compute operations. The Sparking Water package allows H2O to run large models in memory across a grid of computers. For more information about H2O, refer to the following URL:

`http://0xdata.com/product/deep-learning/`

In addition to deep learning, H2O supports a variety of other machine learning models, such as logistic regression, decision trees, and gradient boosting.

9.7.2 Theano

Theano is a mathematical package for Python, similar to the widely used Python package, Numpy (J. Bergstra, O. Breuleux, F. Bastien, et al., J. Bergstra, O. Breuleux, F. Bastien, 2012). Like Numpy, Theano primarily targets mathematics. Though Theano does not directly implement deep neural networks, it provides all of the mathematical tools necessary for the programmer to create deep neural network applications. Theano also directly supports GPGPU. You can find the Theano package at the following URL:

`http://deeplearning.net/software/theano/`

The creators of Theano also wrote an extensive tutorial for deep learning, using Theano that can be found at the following URL:

`http://deeplearning.net/`

9.7.3 Lasagne and NoLearn

Because Theano does not directly support deep learning, several packages have been built upon Theano to make it easy for the programmer to implement deep learning. One pair of packages, often used together, is Lasagne and

Nolearn. Nolearn is a package for Python that provides abstractions around several machine learning algorithms. In this way, Nolearn is similar to the popular framework, Scikit-Learn. While Scikit-Learn focuses widely on machine learning, Nolearn specializes on neural networks. One of the neural network packages supported by Nolearn is Lasagne, which provides deep learning and can be found at the following URL:

`https://pypi.python.org/pypi/Lasagne/0.1dev`

You can access the Nolearn package at the following URL:

`https://github.com/dnouri/nolearn`

The deep learning framework Lasange takes its name from the Italian food lasagna. The spellings "lasange" and "lasagna" are both considered valid spellings of the Italian food. In the Italian language, "lasange" is singular, and "lasagna" is the plural form. Regardless of the spelling used, lasagna is a good name for a deep learning framework. Figure 9.4 shows that, like a deep neural network, lasagna is made up of many layers:

Figure 9.4: Lasagna Layers

9.7.4 ConvNetJS

Deep learning support has also been created for Javascript. The ConvNetJS package implements many deep learning algorithms, particularly in the area of convolutional neural networks. ConvNetJS primarily targets the creation of deep learning examples on websites. We used ConvNetJS to provide many of the deep learning JavaScript examples on this book's website:

`http://cs.stanford.edu/people/karpathy/convnetjs/`

9.8 Deep Belief Neural Networks

The deep belief neural network (DBNN) was one of the first applications of deep learning. A DBNN is simply a regular belief network with many layers. Belief networks, introduced by Neil in 1992 are different from regular feedforward neural networks. Hinton (2007) describes DBNNs as "probabilistic generative models that are composed of multiple layers of stochastic, latent variables." Because this technical description is complicated, we will define some terms.

- **Probabilistic** - DBNNs are used to classify, and their output is the probability that an input belongs to each class.
- **Generative** - DBNNs can produce plausible, randomly created values for the input values. Some DBNN literatures refer to this trait as dreaming.
- **Multiple layers** - Like a neural network, DBNNs can be made of multiple layers.
- **Stochastic, latent variables** - DBNNs are made up of Boltzmann machines that produce random (stochastic) values that cannot be directly observed (latent).

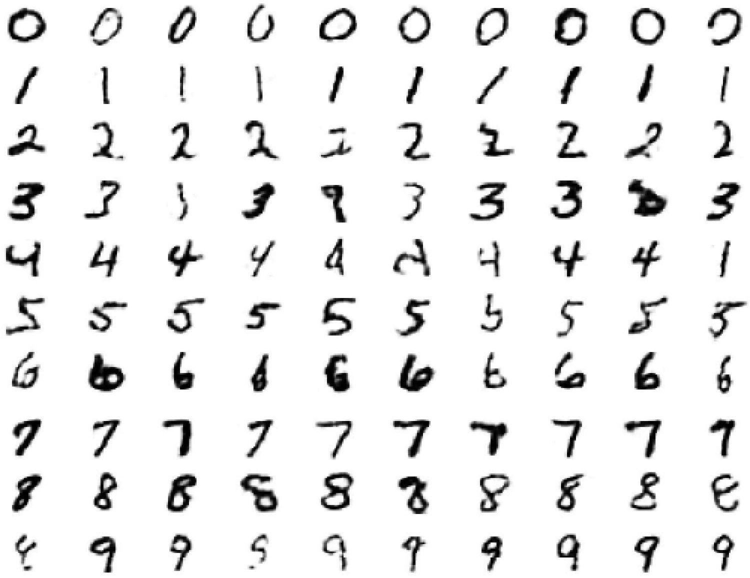
The primary differences between a DBNN and a feedforward neural network (FFNN) are summarized as follows:

- Input to a DBNN must be binary; input to a FFNN is a decimal number.
- The output from a DBNN is the class to which the input belongs; the output from a FFNN can be a class (classification) or a numeric prediction (regression).
- DBNNs can generate plausible input based on a given outcome. FFNNs cannot perform like the DBNNs.

These are important differences. The first bullet item is one of the most limiting factors of DBNNs. The fact that a DBNN can accept only binary input often severely limits the type of problem that it can tackle. You also need to note that a DBNN can be used only for classification and not for regression. In other words, a DBNN could classify stocks into categories such as buy, hold, or sell; however, it could not provide a numeric prediction about the stock, such as the amount that may be attained over the next 30 days. If you need any of these features, you should consider a regular deep feedforward network.

Compared to feedforward neural networks, DBNNs may initially seem somewhat restrictive. However, they do have the ability to generate plausible input cases based on a given output. One of the earliest DBNN experiments was to have a DBNN classify ten digits, using handwritten samples. These digits were from the classic MNIST handwritten digits data set that was included in this book's introduction. Once the DBNN is trained on the MNIST digits, it can produce new representations of each digit, as seen in Figure 9.5:

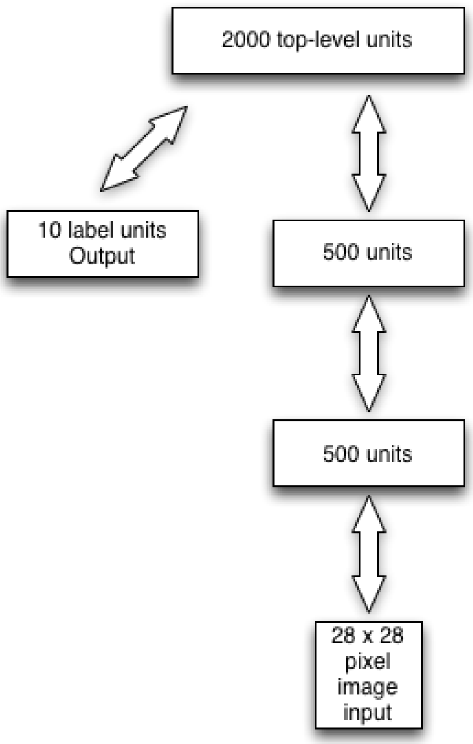
Figure 9.5: DBNN Dreaming of Digits



The above digits were taken from Hinton's (2006) deep learning paper. The first row shows a variety of different zeros that the DBNN generated from its training data.

The restricted Boltzmann machine (RBM) is the center of the DBNN. Input provided to the DBNN passes through a series of stacked RBMs that make up the layers of the network. Creating additional RBM layers causes deeper DBNNs. Though RBMs are unsupervised, the desire is for the resulting DBNN to be supervised. To accomplish the supervision, a final logistic regression layer is added to distinguish one class from another. In the case of Hinton's experiment, shown in Figure 9.6, the classes are the ten digits:

Figure 9.6: Deep Belief Neural Network (DBNN)



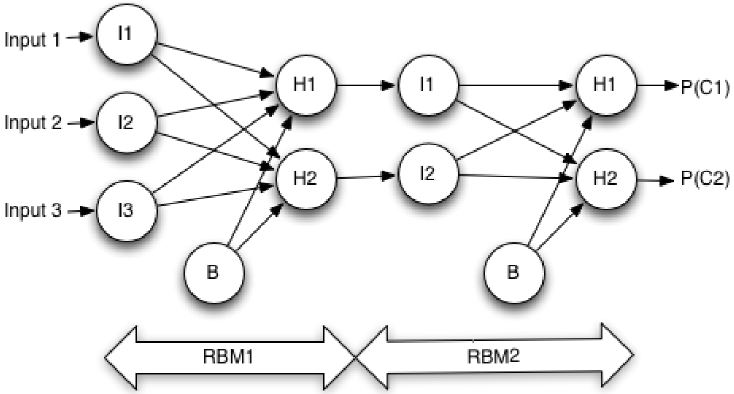
The above diagram shows a DBNN that uses the same hyper-parameters as Hinton's experiment. Hyper-parameters specify the architecture of a neural network, such as the number of layers, hidden neuron counts, and other settings. Each of the digit images presented to the DBNN is 28x28 pixels, or vectors of 784 pixels. The digits are monochrome (black & white) so these 784 pixels are single bits and are thus compatible with the DBNN's requirement that all input be binary. The above network has three layers of stacked RBMs, containing 500 neurons, a second 500-neuron layer, and 2,000 neurons, respectively.

The following sections discuss a number of algorithms used to implement DBNNs.

9.8.1 Restricted Boltzmann Machines

Because Chapter 3, "Hopfield & Boltzmann Machines," includes a discussion of Boltzmann machines, we will not repeat this material here. This chapter deals with the restricted version of the Boltzmann machine and stacking these RBMs to achieve depth. Figure 2.10, from Chapter 3, shows an RBM. The primary difference with an RBM is that the visible (input) neurons and the hidden (output) neurons have the only connections. In the case of a stacked RBM, the hidden units become the output to the next layer. Figure 9.7 shows how two Boltzmann machines are stacked:

Figure 9.7: Stacked RBMs



We can calculate the output from an RBM exactly as shown in Chapter 3, "Hopfield & Boltzmann Machines," in Equation 3.6. The only difference is now we have two Boltzmann machines stacked. The first Boltzmann machine receives three inputs passed to its visible units. The hidden units pass their output directly to the two inputs (visible units) of the second RBM. Notice that there are no weights between the two RBMs, and the output from the H1 and H2 units in RBM1 pass directly to I1 and I2 from RBM2.

9.8.2 Training a DBNN

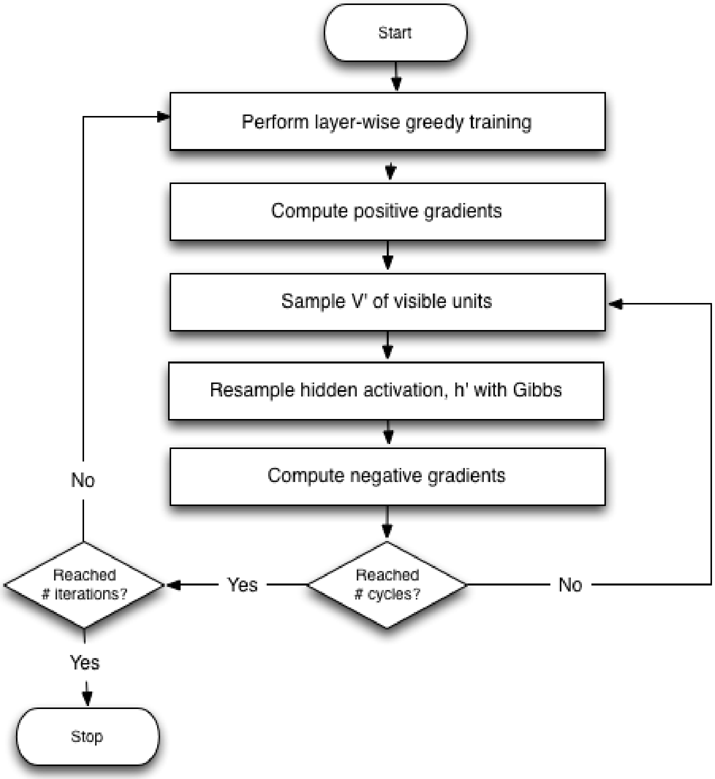
The process of training a DBNN requires a number of steps. Although the mathematics behind this process can become somewhat complex, you don't need to understand every detail for training DBNNs in order to use them. You just need to know the following key points:

- DBNNs undergo supervised and unsupervised training.
- During the unsupervised portion, the DBNN uses training data without their labels, which allows DBNNs to have a mix of supervised and unsupervised data.
- During the supervised portion, only training data with labels are used.
- Each DBNN layer is trained independently during the unsupervised portion.
- It is possible to train the DBNN layers concurrently (with threads) during the unsupervised portion.
- After the unsupervised portion is complete, the output from the layers is refined with supervised logistic regression.
- The top logistic regression layer predicts the class to which the input belongs.

Armed with this knowledge, you can skip ahead to the deep belief classification example in this chapter. However, if you wish to learn the specific details of DBNN training, read on.

Figure 9.8 provides a summary of the steps of DBNN training:

Figure 9.8: DBNN Training


Start
Perform layer-wise greedy training
Compute positive gradients
Sample V' of visible units
Resample hidden activation, h' with Gibbs
Compute negative gradients
Reached # cycles?
No
Yes
Reached # iterations?
No
Yes
Stop

9.8.3 Layer-Wise Sampling

The first step when performing unsupervised training on an individual layer is to calculate all values of the DBNN up to that layer. You will do this calculation for every training set, and the DBNN will provide you with sampled values at the layer that you are currently training. Sampled refers to the fact that the neural network randomly chooses a true/false value based on a probability.

You need to understand that sampling uses random numbers to provide you with your results. Because of this randomness, you will not always get the same result. If the DBNN determines that a hidden neuron's probability of true is 0.75, then you will get a value of true 75% of the time. Layer-wise sampling is very similar to the method that we used to calculate the output of Boltzmann machines in Chapter 3, "Hopfield & Boltzmann Machines." We will use Equation 3.6, from chapter 3 to compute the probability. The only difference is that we will use the probability given by Equation 3.6 to generate a random sample.

The purpose of the layer-wise sampling is to produce a binary vector to feed into the contrastive divergence algorithm. When training each RBM, we always provide the output of the previous RBM as the input to the current RBM. If we are training the first RBM (closest to the input), we simply use the training input vector for contrastive divergence. This process allows each of the RBMs to be trained. The final softmax layer of the DBNN is not trained during the unsupervised phase. The final logistic regression phase will train the softmax layer.

9.8.4 Computing Positive Gradients

Once the layer-wise training has processed each of the RBM layers, we can utilize the up-down algorithm, or the contrastive divergence algorithm. This complete algorithm includes the following steps, covered in the next sections of this book:

- Computing Positive Gradients
- Gibbs Sampling
- Update Weights and Biases
- Supervised Backpropagation

Like many of the gradient-descent-based algorithms presented in Chapter 6, "Backpropagation Training," the contrastive divergence algorithm is also based on gradient descent. It uses the derivative of a function to find the inputs to the function that produces the lowest output for that function. Several different gradients are estimated during contrastive divergence. We can use these estimates instead of actual calculations because the real gradients are too complex to calculate. For machine learning, an estimate is often good enough.

Additionally, we must calculate the mean probability of the hidden units by propagating the visible units to the hidden ones. This computation is the "up" portion of the up-down algorithm. Equation 9.1 performs this calculation:

$$\bar{h}_i^+ = sigmoid(\sum_j w_j v_j + b_i) \tag{9.1}$$

The above equation calculates the mean probability of each of the hidden neurons (h). The bar above the h designates it as a mean, and the positive subscript indicates that we are calculating the mean for the positive (or up) part of the algorithm. The bias is added to the sigmoid function value of the weighted sum of all visible units.

Next a value must be sampled for each of the hidden neurons. This value will randomly be either true (1) or false (0) with the mean probability just calculated. Equation 9.2 accomplishes this sampling:

$$h_i^+ = \begin{cases} 1 & r < \bar{h}_i^+ \\ 0 & r \geq \bar{h}_i^+ \end{cases} \tag{9.2}$$

This equation assumes that r is a uniform random value between 0 and 1. A uniform random number simply means that every possible number in that range has an equal probability of being chosen.

9.8.5 Gibbs Sampling

The calculation of the negative gradients is the "down" phase of the up-down algorithm. To accomplish this calculation, the algorithm uses Gibbs sapling to estimate the mean of the negative gradients. Geman and Geman (1984) introduced Gibbs sampling and named it after the physicist Josiah Willard Gibbs. The technique is accomplished by looping through k iterations that improve the quality of the estimate. Each iteration performs two steps:

- Sample visible neurons give hidden neurons.
- Sample hidden neurons give visible neurons.

For the first iteration of Gibbs sampling, we start with the positive hidden neuron samples obtained from the last section. We will sample visible neuron average probabilities from these (first bullet above). Next, we will use these visible hidden neurons to sample hidden neurons (second bullet above). These new hidden probabilities are the negative gradients. For the next cycle, we will use the negative gradients in place of the positive ones. This continues for each iteration and produces better negative gradients. Equation 9.3 accomplishes sampling of the visible neurons (first bullet):

$$\bar{v}_i^- = sigmoid(\sum_j w_j h_j + b_i) \tag{9.3}$$

This equation is essentially the reverse of Equation 9.1. Here, we determine the average visible mean using the hidden values. Again, just like we did for the positive gradients, we sample a negative probability using Equation 9.4:

$$v_i^- = \begin{cases} 1 & r < \bar{v}_i^- \\ 0 & r \geq \bar{v}_i^- \end{cases} \tag{9.4}$$

The above equation assumes that r is a uniform random number between 0 and 1.

The above two equations are only half of the Gibbs sampling step. These equations accomplished the first bullet point above because they sample visible neurons, given hidden neurons. Next, we must accomplish the second bullet point. We must sample hidden neurons, given visible neurons. This process is very similar to the above section, “Computing Positive Gradients.” This time, however, we are calculating the negative gradients.

The visible unit samples just calculated can obtain hidden means, as shown in Equation 9.5:

$$\bar{h}_i^- = sigmoid(\sum_j w_j v_j + b_i) \tag{9.5}$$

Just as before, mean probability can sample an actual value. In this case, we use the hidden mean to sample a hidden value, as demonstrated by Equation 9.6:

$$h_i^- = \begin{cases} 1 & r < \bar{h}_i^- \\ 0 & r \geq \bar{h}_i^- \end{cases} \tag{9.6}$$

The Gibbs sampling process continues. The negative hidden samples can process each iteration. Once this calculation is complete, you have the following six vectors:

- Positive mean probabilities of the hidden neurons
- Positive sampled values of the hidden neurons
- Negative mean probabilities of visible neurons
- Negative sampled values of visible neurons
- Negative mean probabilities of hidden neurons
- Negative sampled values of hidden neurons

These values will update the neural network’s weights and biases.

9.8.6 Update Weights & Biases

The purpose of any neural network training is to update the weights and biases. This adjustment is what allows the neural network to learn to perform the intended task. This is the final step for the unsupervised portion of the DBNN training process. In this step, the weights and biases of a single layer (Boltzmann machine) will be updated. As previously mentioned, the Boltzmann layers are trained independently.

The weights and biases are updated independently. Equation 9.7 shows how to update a weight:

$$\Delta_{ij} = \frac{\epsilon(\bar{h}_i^+ x_j - \bar{h}_i^- v_j^-)}{|x|} \tag{9.7}$$

The learning rate (ϵ, epsilon) specifies how much of a calculated change should be applied. High learning rates will learn quicker, but they might skip over an optimal set of weights. Lower learning rates learn more slowly, but they might have a higher quality result. The value x represents the current training set element. Because x is a vector (array), the x enclosed in two bars represents the length of x. The above equation also uses the positive mean hidden probabilities, the negative mean hidden probabilities, and the negative sampled values.

Equation 9.8 calculates the biases in a similar fashion:

$$\Delta b_i = \frac{\epsilon(h_i^+ - \bar{h}_i^-)}{||x||} \tag{9.8}$$

The above equation uses the sampled hidden value from the positive phase and the mean hidden value from the negative phase, as well as the input vector. Once the weights and biases have been updated, the unsupervised portion of the training is done.

9.8.7 DBNN Backpropagation

Up to this point, the DBNN training has focused on unsupervised training. The DBNN used only the training set inputs (x values). Even if the data set provided an expected output (y values), the unsupervised training didn't use it. Now the DBNN is trained with the expected outputs. We use only data set items that contain an expected output during this last phase. This step allows the program to use DBNN networks with data sets where each item does not necessarily have an expected output. We refer to the data as partially labeled data sets.

The final layer of the DBNN is simply a neuron for each class. These neurons have weights to the output of the previous RBM layer. These output neurons all use sigmoid activation functions and a softmax layer. The softmax layer ensures that the output for each of the classes sum to 1.

Regular backpropagation trains this final layer. The final layer is essentially the output layer of a feedforward neural network that receives its input from the top RBM. Because Chapter 6, "Backpropagation Training," contains a discussion of backpropagation, we will not repeat the information here. The main idea of a DBNN is that the hierarchy allows each layer to interpret the data for the next layer. This hierarchy allows the learning to spread across the layers. The higher layers learn more abstract notions while the lower layers form from the input data. In practice, DBNNs can process much more complex of patterns than a regular backpropagation-trained feedforward neural network.

9.8.8 Deep Belief Application

This chapter presents a simple example of the DBNN. This example simply accepts a series of input patterns as well as the classes to which these input patterns belong. The input patterns are shown below:

```
[[1, 1, 1, 1, 0, 0, 0, 0],
 [1, 1, 0, 1, 0, 0, 0, 0],
 [1, 1, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 1, 1, 1],
 [0, 0, 0, 0, 1, 1, 0, 1],
 [0, 0, 0, 0, 1, 1, 1, 0]]
```

We provide the expected output from each of these training set elements. This information specifies the class to which each of the above elements belongs and is shown below:

```
[[1, 0],
 [1, 0],
 [1, 0],
 [0, 1],
 [0, 1],
 [0, 1]]
```

The program provided in the book's example creates a DBNN with the following configuration:

- Input Layer Size: 8
- Hidden Layer #1: 2
- Hidden Layer #2: 3
- Output Layer Size: 2

First, we train each of the hidden layers. Finally, we perform logistic regression on the output layer. The output from this program is shown here:

```
Training Hidden Layer #0
Training Hidden Layer #1
Iteration: 1, Supervised training: error = 0.2478464544753616
Iteration: 2, Supervised training: error = 0.23501688281192523
Iteration: 3, Supervised training: error = 0.2228704042138232
...
Iteration: 287, Supervised training: error = 0.0010805100324100002
Iteration: 288, Supervised training: error = 7.821742124428358E-4
[0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0] -> [0.9649828726012807,
   0.03501712739871941]
[1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0] -> [0.9649830045627616,
   0.03501699543723843 5]
[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0] -> [0.03413161595489315,
   0.9658683840451069]
[0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0] -> [0.03413137188719462,
   0.9658686281128055]
```

As you can see, the program first trained the hidden layers and then went through 288 iterations of regression. The error level dropped considerably during these iterations. Finally, the sample data quizzed the network. The network responded with the probability of the input sample being in each of the two classes that we specified above.

For example, the network reported the following element:

```
[0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0]
```

This element had a 96% probability of being in class 1, but it had only a 4% probability of being in class 2. The two probabilities reported for each item must sum to 100%.

9.9 Chapter Summary

This chapter provided a high-level overview of many of the components of deep learning. A deep neural network is any network that contains more than two hidden layers. Although deep networks have existed for as long as multilayer neural networks, they have lacked good training methods until recently. New training techniques, activation functions, and regularization are making deep neural networks feasible.

Overfitting is a common problem for many areas of machine learning; neural networks are no exception. Regularization can prevent overfitting. Most forms of regularization involve modifying the weights of a neural network as the training occurs. Dropout is a very common regularization technique for deep neural networks that removes neurons as training progresses. This technique prevents the network from becoming overly dependent on any one neuron.

We ended the chapter with the deep belief neural network (DBNN), which classifies data that might be partially labeled. First, both labeled and unlabeled data can initialize the weights of the neural network with unsupervised training. Using these weights, a logistic regression layer can fine-tune the network to the labeled data.

We also discussed the convolutional neural networks (CNN) in this chapter. This type of neural network causes the weights to be shared between the various neurons in the network. This neural network allows the CNN to deal with the types of overlapping features that are very common in computer vision. We provided only a general overview of CNN because we will examine the CNNs in greater detail in the next chapter.

Chapter 10

Convolutional Neural Networks

- Sparse Connectivity
- Shared Weights
- Max-pooling

The convolutional neural network (CNN) is a neural network technology that has profoundly impacted the area of computer vision (CV). Fukushima (1980) introduced the original concept of a convolutional neural network, and LeCun, Bottou, Bengio & Haffner (1998) greatly improved this work. From this research, Yan LeCun introduced the famous LeNet-5 neural network architecture. This chapter follows the LeNet-5 style of convolutional neural network.

Although computer vision primarily uses CNNs, this technology has some application outside of the field. You need to realize that if you want to utilize CNNs on non-visual data, you must find a way to encode your data so that it can mimic the properties of visual data.

CNNs are somewhat similar to the self-organizing map (SOM) architecture that we examined in Chapter 2, "Self-Organizing Maps." The order of the vector elements is crucial to the training. In contrast, most neural networks that are not CNNs or SOMs treat their input data as a long vector of values, and the order that you arrange the incoming features in this vector is irrelevant. For these types of neural networks, you cannot change the order after you have trained the network. In other words, CNNs and SOMs do not follow the standard treatment of input vectors.

The SOM network arranged the inputs into a grid. This arrangement worked well with images because the pixels in closer proximity to each other are important to each other. Obviously, the order of pixels in an image is significant. The human body is a relevant example of this type of order. For the design of the face, we are accustomed to eyes being near to each other. In the same way, neural network types like SOMs adhere to an order of pixels. Consequently, they have many applications to computer vision.

Although SOMs and CNNs are similar in the way that they map their input into 2D grids or even higher-dimension objects such as 3D boxes, CNNs take image recognition to higher level of capability. This advance in CNNs is due to years of research on biological eyes. In other words, CNNs utilize overlapping fields of input to simulate features of biological eyes. Until this breakthrough, AI had been unable to reproduce the capabilities of biological vision.

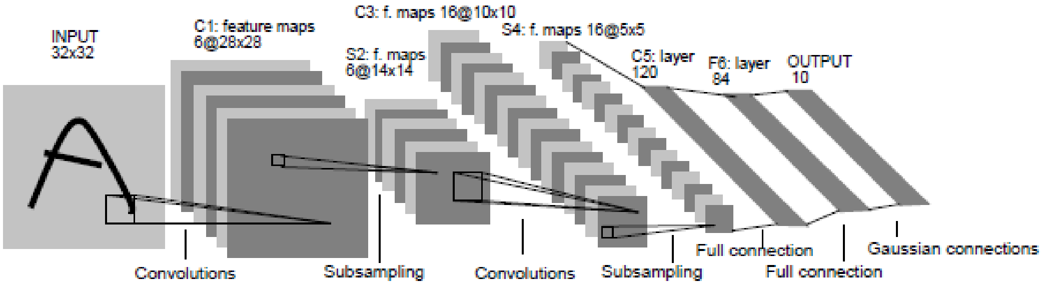
Scale, rotation, and noise have presented challenges in the past for AI computer vision research. You can observe the complexity of biological eyes in the example that follows. A friend raises a sheet of paper with a large number written on it. As your friend moves nearer to you, the number is still identifiable. In the same way, you can still identify the number when your friend rotates the paper. Lastly, your friend creates noise by drawing lines on top of the page, but you can still identify the number. As you can see, these examples demonstrate the high function of the biological eye and allow you to understand better the research breakthrough of CNNs. That is, this neural network has the ability to process scale, rotation, and noise in the field of computer vision.

10.1 LeNET-5

We can use the LeNET-5 architecture primarily for the classification of graphical images. This network type is similar to the feedforward network that we examined in previous chapters. Data flow from input to the output. However, the LeNET-5 network contains several different layer types, as Figure 10.1

illustrates:

Figure 10.1: A LeNET-5 Network (LeCun, 1998)



Several important differences exist between a feedforward neural network and a LeNET-5 network:

- Vectors pass through feedforward networks; 3D cubes pass through LeNET-5 networks.
- LeNET-5 networks contain a variety of layer types.
- Computer vision is the primary application of the LeNET-5.

However, we have also explored the many similarities between the networks. The most important similarity is that we can train the LeNET-5 with the same backpropagation-based techniques. Any optimization algorithm can train the weights of either a feedforward or LeNET-5 network. Specifically, you can utilize simulated annealing, genetic algorithms, and particle swarm for training. However, LeNET-5 frequently uses backpropagation training.

The following three layer types comprise the original LeNET-5 neural networks:

- Convolutional Layers
- Max-pool Layers
- Dense Layers

Other neural network frameworks will add additional layer types related to computer vision. However, we will not explore these additions beyond the LeNET-5 standard. Adding new layer types is a common means of augmenting existing neural network research. Chapter 12, "Dropout and Regularization," will introduce an additional layer type that is designed to reduce overfitting by adding a dropout layer.

For now, we focus our discussion on the layer types associated with convolutional neural networks. We will begin with convolutional layers.

10.2 Convolutional Layers

The first layer that we will examine is the convolutional layer. We will begin by looking at the hyper-parameters that you must specify for a convolutional layer in most neural network frameworks that support the CNN:

- Number of filters
- Filter Size
- Stride
- Padding
- Activation Function/Non-Linearity

The primary purpose for a convolutional layer is to detect features such as edges, lines, blobs of color, and other visual elements. The filters can detect these features. The more filters that we give to a convolutional layer, the more features it can detect.

A filter is a square-shaped object that scans over the image. A grid can represent the individual pixels of a grid. You can think of the convolutional layer as a smaller grid that sweeps left to right over each row of the image. There is also a hyper-parameter that specifies both the width and height of the square-shaped filter. Figure 10.1 shows this configuration in which you see the six convolutional filters sweeping over the image grid:

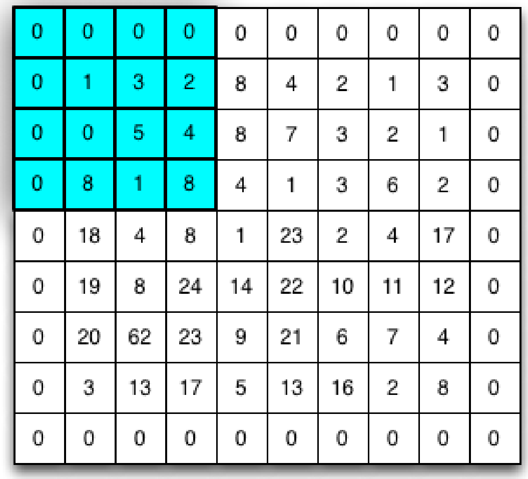
A convolutional layer has weights between it and the previous layer or image grid. Each pixel on each convolutional layer is a weight. Therefore, the number of weights between a convolutional layer and its predecessor layer or image field is the following:

```
[Filter Size] * [Filter Size] * [# of Filters]
```

For example, if the filter size were 5 (5x4) for 10 filters, there would be 250 weights.

You need to understand how the convolutional filters sweep across the previous layer's output or image grid. Figure 10.2 illustrates the sweep:

Figure 10.2: Convolutional Filter



0	0	0	0	0	0	0	0	0	0
0	1	3	2	8	4	2	1	3	0
0	0	5	4	8	7	3	2	1	0
0	8	1	8	4	1	3	6	2	0
0	18	4	8	1	23	2	4	17	0
0	19	8	24	14	22	10	11	12	0
0	20	62	23	9	21	6	7	4	0
0	3	13	17	5	13	16	2	8	0
0	0	0	0	0	0	0	0	0	0

The above figure shows a convolutional filter with a size of 4 and a padding size of 1. The padding size is responsible for the boarder of zeros in the area that the filter sweeps. Even though the image is actually 8x7, the extra padding provides a virtual image size of 9x8 for the filter to sweep across. The stride specifies the number of positions at which the convolutional filters will stop. The convolutional filters move to the right, advancing by the number of cells specified in the stride. Once the far right is reached, the convolutional filter moves back to the far left, then it moves down by the stride amount and continues to the right again.

Some constraints exist in relation to the size of the stride. Obviously, the stride cannot be 0. The convolutional filter would never move if the stride were set to 0. Furthermore, neither the stride, nor the convolutional filter size can be larger than the previous grid. There are additional constraints on the stride (s), padding (p) and the filter width (f) for an image of width (w). Specifically, the convolutional filter must be able to start at the far left or top boarder, move a certain number of strides, and land on the far right or bottom boarder. Equation 10.1 shows the number of steps a convolutional operator must take to cross the image:

$$steps = \frac{w - f + 2p}{s + 1} \tag{10.1}$$

The number of steps must be an integer. In other words, it cannot have decimal places. The purpose of the padding (p) is to be adjusted to make this equation become an integer value.

We can use the same set of weights as the convolutional filter sweeps over the image. This process allows convolutional layers to share weights and greatly reduce the amount of processing needed. In this way, you can recognize the image in shift positions because the same convolutional filter sweeps across the entire image.

The input and output of a convolutional layer are both 3D boxes. For the input to a convolutional layer, the width and height of the box is equal to the width and height of the input image. The depth of the box is equal to the color depth of the image. For an RGB image, the depth is 3, equal to the components of red, green, and blue. If the input to the convolutional layer is another layer, then it will also be a 3D box; however, the dimensions of that 3D box will be dictated by the hyper-parameters of that layer.

Like any other layer in the neural network, the size of the 3D box output by a convolutional layer is dictated by the hyper-parameters of the layer. The width and height of this box are both equal to the filter size. However, the depth is equal to the number of filters.

10.3 Max-Pool Layers

Max-pool layers downsample a 3D box to a new one with smaller dimensions. Typically, you can always place a max-pool layer immediately following a convolutional layer. Figure 10.1 shows the max-pool layer immediately after layers C1 and C3. These max-pool layers progressively decrease the size of the dimensions of the 3D boxes passing through them. This technique can avoid overfitting (Krizhevsky, Sutskever & Hinton, 2012).

A pooling layer has the following hyper-parameters:

- Spatial Extent (f)
- Stride (s)

Unlike convolutional layers, max-pool layers do not use padding. Additionally, max-pool layers have no weights, so training does not affect them. These layers simply downsample their 3D box input.

The 3D box output by a max-pool layer will have a width equal to Equation 10.2:

$$w_2 = \frac{w_1 - f}{s + 1} \tag{10.2}$$

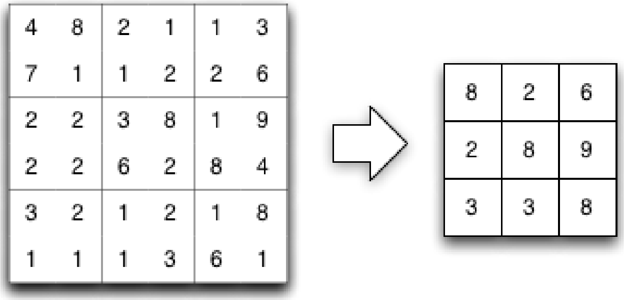
The height of the 3D box produced by the max-pool layer is calculated similarly with Equation 10.3:

$$h_2 = \frac{h_1 - f}{s + 1} \tag{10.3}$$

The depth of the 3D box produced by the max-pool layer is equal to the depth the 3D box received as input.

The most common setting for the hyper-parameters of a max-pool layer are f=2 and s=2. The spatial extent (f) specifies that boxes of 2x2 will be scaled down to single pixels. Of these four pixels, the pixel with the maximum value will represent the 2x2 pixel in the new grid. Because squares of size 4 are replaced with size 1, 75% of the pixel information is lost. Figure 10.3 shows this transformation as a 6x6 grid becomes a 3x3:

Figure 10.3: Max-pooling (f=2,s=2)



4	8	2	1	1	3
7	1	1	2	2	6
2	2	3	8	1	9
2	2	6	2	8	4
3	2	1	2	1	8
1	1	1	3	6	1

8	2	6
2	8	9
3	3	8

Of course, the above diagram shows each pixel as a single number. A grayscale image would have this characteristic. For an RGB image, we usually take the average of the three numbers to determine which pixel has the maximum value.

10.4 Dense Layers

The final layer type in a LeNET-5 network is a dense layer. This layer type is exactly the same type of layer as we've seen before in feedforward neural networks. A dense layer connects every element (neuron) in the previous layer's output 3D box to each neuron in the dense layer. The resulting vector is passed through an activation function. LeNET-5 networks will typically use a ReLU activation. However, we can use a sigmoid activation function; this technique is less common. A dense layer will typically contain the following hyper-parameters:

- Neuron Count
- Activation Function

The neuron count specifies the number of neurons that make up this layer. The activation function indicates the type of activation function to use. Dense layers can employ many different kinds of activation functions, such as ReLU, sigmoid or hyperbolic tangent.

LeNET-5 networks will typically contain several dense layers as their final layers. The final dense layer in a LeNET-5 actually performs the classification. There should be one output neuron for each class, or type of image, to classify. For example, if the network distinguishes between dogs, cats, and birds, there will be three output neurons. You can apply a final softmax function to the final layer to treat the output neurons as probabilities. Softmax allows each neuron to provide the probability of the image representing each class. Because the output neurons are now probabilities, softmax ensures that they sum to 1.0 (100%). To review softmax, you can reread Chapter 4, "Feedforward Neural Networks."

10.5 ConvNets for the MNIST Data Set

In Chapter 6, "Backpropagation Training," we used the MNIST handwritten digits as an example of using backpropagation. In Chapter 10, we present an example about improving our recognition of the MNIST digits, as a deep convolutional neural network. The convolutional network, being a deep neural network, will have more layers than the feedforward neural network seen in Chapter 6. The hyper-parameters for this network are as follows:

- Input: Accepts box of [1,96,96]
- Convolutional Layer: filters=32, filter_size=[3,3]
- Max-pool Layer: [2,2]
- Convolutional Layer: filters=64, filter_size=[2,2]
- Max-pool Layer: [2,2]
- Convolutional Layer: filters=128, filter_size=[2,2]
- Max-pool Layer: [2,2]

- Dense Layer: 500 neurons
- Output Layer: 30 neurons

This network uses the very common pattern to follow each convolutional layer with a max-pool layer. Additionally, the number of filters decreases from the input to the output layer, thereby allowing a smaller number of basic features, such as edges, lines, and small shapes to be detected near the input field. Successive convolutional layers roll up these basic features into larger and more complex features. Ultimately, the dense layer can map these higher-level features into each x-coordinate and y-coordinate of the actual 15 digit features.

Training the convolutional neural network takes considerable time, especially if you are not using GPU processing. As of July 2015, not all frameworks have equal support of GPU processing. At this time, using Python with a Theano-based neural network framework, such as Lasange, provides the best results. Many of the same researchers who are improving deep convolutional networks are also working with Theano. Thus, they promote it before other frameworks on other languages.

For this example, we used Theano with Lasange. The book's example download may have other languages available for this example as well, depending on the frameworks available for those languages. Training a convolutional neural network for digit feature recognition on Theano took less time with a GPU than a CPU, as a GPU helps considerably for convolutional neural networks. The exact amount of performance will vary according to hardware and platform. The accuracy comparison between the convolutional neural network and the regular ReLU network is shown here:

```
Relu:
Best valid loss was 0.068229 at epoch 17.
Incorrect 170/10000 (1.7000000000000002%)
ReLU+Conv:
Best valid loss was 0.065753 at epoch 3.
Incorrect 150/10000 (1.5%)
```

If you compare the results from the convolutional neural network to the standard feedforward neural network from Chapter 6, you will see the convolutional neural network performed better. The convolutional neural network is capable of recognizing sub-features in the digits to boost its performance over the standard feedforward neural network. Of course, these results will vary, depending on the platform used.

10.6 Chapter Summary

Convolutional neural networks are a very active area in the field of computer vision. They allow the neural network to detect hierarchies of features, such as lines and small shapes. These simple features can form hierarchies to teach the neural network to recognize complex patterns composed of the more simple features. Deep convolutional networks can take considerable processing power. Some frameworks allow the use of GPU processing to enhance performance.

Yann LeCun introduced the LeNET-5, the most common type of convolutional network. This neural network type is comprised of dense layers, convolutional layers and max-pool layers. The dense layers work exactly the same way as traditional feedforward networks. Max-pool layers can downsample the image and remove detail. Convolutional layers detect features in any part of the image field.

There are many different approaches to determine the best architecture for a neural network. Chapter 8, "NEAT, CPPN and HyperNEAT," introduced a neural network algorithm that could automatically determine the best architecture. If you are using a feedforward neural network you will most likely arrive at a structure through pruning and model selection, which we discuss in the next chapter.

Chapter 11

Pruning and Model Selection

- Pruning a Neural Network
- Model Selection
- Random vs. Grid Search

In previous chapters, we learned that you could better fit the weights of a neural network with various training algorithms. In effect, these algorithms adjust the weights of the neural network in order to lower the error of the neural network. We often refer to the weights of a neural network as the parameters of the neural network model. Some machine learning models might have parameters other than weights. For example, logistic regression (which we discussed in *Artificial Intelligence for Humans, Volume 1*) has coefficients as parameters.

When we train the model, the parameters of any machine learning model change. However, these models also have hyper-parameters that do not change during training algorithms. For neural networks, the hyper-parameters specify the architecture of the neural network. Examples of hyper-parameters for neural networks include the number of hidden layers and hidden neurons.

In this chapter, we will examine two algorithms that can actually modify or suggest a structure for the neural network. Pruning works by analyzing how much each neuron contributes to the output of the neural network. If a particular neuron's connection to another neuron does not significantly affect the output of the neural network, the connection will be pruned. Through this process, connections and neurons that have only a marginal impact on the output are removed.

The other algorithm that we introduce in this chapter is model selection. While pruning starts with an already trained neural network, model selection creates and trains many neural networks with different hyper-parameters. The program then selects the hyper-parameters producing the neural network that achieves the best validation score.

11.1 Understanding Pruning

Pruning is a process that makes neural networks more efficient. Unlike the training algorithms already discussed in this book, pruning does not increase the training error of the neural network. The primary goal of pruning is to decrease the amount of processing required to use the neural network. Additionally, pruning can sometimes have a regularizing effect by removing complexity from the neural network. This regularization can sometimes decrease the amount of overfitting in the neural network. This decrease can help the neural network perform better on data that were not part of the training set.

Pruning works by analyzing the connections of the neural network. The pruning algorithm looks for individual connections and neurons that can be removed from the neural network to make it operate more efficiently. By pruning unneeded connections, the neural network can be made to execute faster and possibly decrease overfitting. In the next two sections, we will examine how to prune both connections and neurons.

11.1.1 Pruning Connections

Connection pruning is central to most pruning algorithms. The program analyzes the individual connections between the neurons to determine which connections have the least impact on the effectiveness of the neural network. Connections are not the only thing that the program can prune. Analyzing the pruned connections will reveal that the program can also prune individual neurons.

11.1.2 Pruning Neurons

Pruning focuses primarily on the connections between the individual neurons of the neural network. However, to prune individual neurons, we must examine the connections between each neuron and the other neurons. If one particular neuron is surrounded entirely by weak connections, there is no reason to keep that neuron. If we apply the criteria discussed in the previous section, neurons that have no connections are the end result because the program has pruned all of the neuron's connections. Then the program can prune this type of a neuron.

11.1.3 Improving or Degrading Performance

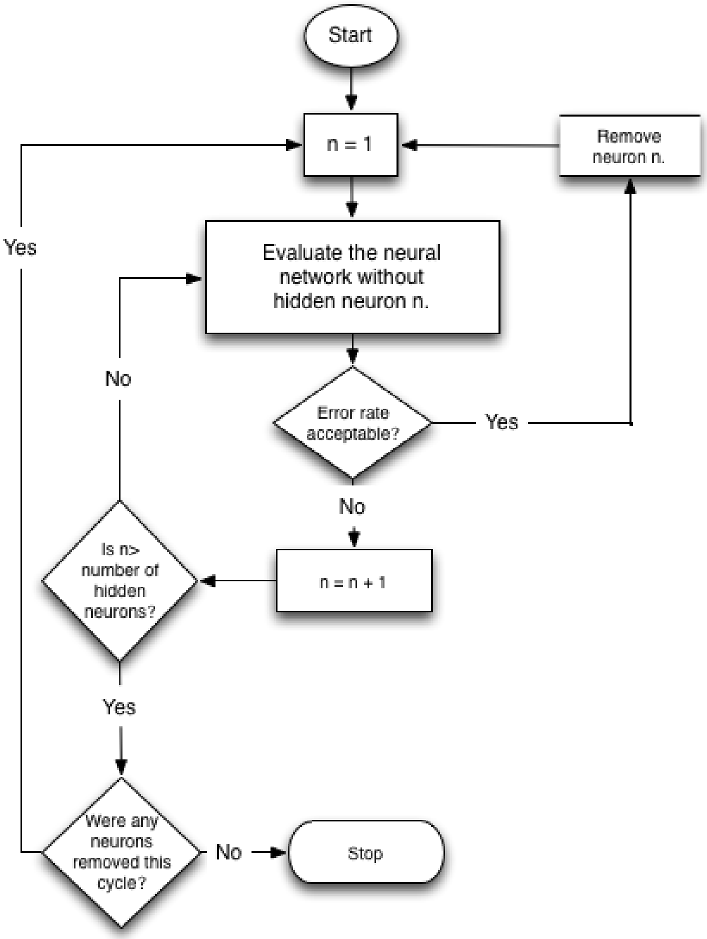
It is possible that pruning a neural network may improve its performance. Any modifications to the weight matrix of a neural network will always have some impact on the accuracy of the recognitions made by the neural network. A connection that has little or no impact on the neural network may actually be degrading the accuracy with which the neural network recognizes patterns. Removing this weak connection may improve the overall output of the neural network.

Unfortunately, pruning can also decrease the effectiveness of the neural network. Thus, you must always analyze the effectiveness of the neural network before and after pruning. Since efficiency is the primary benefit of pruning, you must be careful to evaluate whether an improvement in the processing time is worth a decrease in the neural network's effectiveness. We will evaluate the overall effectiveness of the neural network both before and after pruning in one of the programming examples from this chapter. This analysis will give us an idea of the impact that the pruning process has on the effectiveness of the neural network.

11.2 Pruning Algorithm

We will now review exactly how pruning takes place. Pruning works by examining the weight matrices of a previously trained neural network. The pruning algorithm will then attempt to remove neurons without disrupting the output of the neural network. Figure 11.1 shows the algorithm used for selective pruning:

Figure 11.1: Pruning a Neural Network


Start
n = 1
Remove neuron n.
Yes
Evaluate the neural network without hidden neuron n.
No
Error rate acceptable?
Yes
No
Is n> number of hidden neurons?
n = n + 1
Yes
Were any neurons removed this cycle?
No
Stop

As you can see, the pruning algorithm has a trial-and-error approach. The pruning algorithm attempts to remove neurons from the neural network until it cannot remove additional neurons without degrading the performance of the neural network.

To begin this process, the selective pruning algorithm loops through each of the hidden neurons. For each hidden neuron encountered, the program evaluates the error level of the neural network both with and without the specified neuron. If the error rate jumps beyond a predefined level, the program retains the neuron and evaluates the next. If the error rate does not improve significantly, the program removes the neuron.

Once the program has evaluated all neurons, it repeats the process. This cycle continues until the program has made one pass through the hidden neurons without removing a single neuron. Once this process is complete, a new neural network is achieved that performs acceptably close to the original, yet it has fewer hidden neurons.

11.3 Model Selection

Model selection is the process where the programmer attempts to find a set of hyper-parameters that produce the best neural network, or other machine learning model. In this book, we have mentioned many different hyper-parameters that are the settings that you must provide to the neural network framework. Examples of neural network hyper-parameters include:

- The number of hidden layers
- The order of the convolutional, pooling, and dropout layers
- The type of activation function
- The number of hidden neurons
- The structure of pooling and convolutional layers

As you've read through these chapters that mention hyper-parameters, you've probably been wondering how you know which settings to use. Unfortunately, there is no easy answer. If easy methods existed that determine these settings, programmers would have constructed the neural network frameworks that automatically set these hyper-parameters for you.

While we will provide more insight into hyper-parameters in Chapter 14, “Architecting Neural Networks,” you will still need to use the model selection processes described in this chapter. Unfortunately, model selection is very time-consuming. We spent 90% of our time performing model selection during our last Kaggle competition. Often, success in modeling is closely related to the amount of time you have to spend on model selection.

11.3.1 Grid Search Model Selection

Grid search is a trial-and-error, brute-force algorithm. For this technique, you must specify every combination of the hyper-parameters that you would like to use. You must be judicious in your selection because the number of search iterations can quickly grow. Typically, you must specify the hyper-parameters that you would like to search. This specification might look like the following:

- Hidden Neurons: 2 to 10, step size 2
- Activation Functions: tanh, sigmoid & ReLU

The first item states that the grid search should try hidden neuron counts between 2 and 10 counting by 2, which results in the following: 2, 4, 6, 8, and 10 (5 total possibilities.) The second item states that we should also try the activation functions tanh, sigmoid, and ReLU for each neuron count. This process results in a total of fifteen iterations because five possibilities times three possibilities is fifteen total. These possibilities are listed here:

```
Iteration #1: [2][sigmoid]
Iteration #2: [4][sigmoid]
Iteration #3: [6][sigmoid]
Iteration #4: [8][sigmoid]
Iteration #5: [10][sigmoid]
Iteration #6: [2][ReLU]
Iteration #7: [4][ReLU]
Iteration #8: [6][ReLU]
Iteration #9: [8][ReLU]
Iteration #10: [10][ReLU]
Iteration #11: [2][tanh]
Iteration #12: [4][tanh]
Iteration #13: [6][tanh]
Iteration #14: [8][tanh]
Iteration #15: [10][tanh]
```

Each set of possibilities is called an axis. These axes rotate through all possible combinations before they finish. You can visualize this process by thinking of a car's odometer. The far left dial (or axis) is spinning the fastest. It counts between 0 and 9. Once it hits 9 and needs to go to the next number, it forward back to 0, and the next place, to the left, rolls forward by one. Unless that next place was also on 9, the rollover continues to the left. At some point, all digits on the odometer are at 9, and the entire device would roll back over to 0. When this final rollover occurs, the grid search is done.

Most frameworks allow two axis types. The first type is a numeric range with a step. The second type is a list of values, like the activation functions above. The following Javascript example allows you to try your own sets of axes to see the number of iterations produced:

`http://www.heatonresearch.com/aifh/vol3/grid_iter.html`

Listing 11.1 shows the pseudocode necessary to roll through all iterations of several sets of values:

Listing 11.1: Grid Search

```
# The variable axes contains a list of each axis.
# Each axes (in axes) is a list of possible values
# for that axis.
# Current index of each axis is zero, create an array
# of zeros.
indexes = zeros(len(axes))
done = false
while not done:
# Prepare vector of current iterationŠs
# hyper-parameters.
  iteration = []
  for i from 0 to len(axes)
    iteration.add(axes [i][indexes[i]] )
```

```
# Perform one iteration, passing in the hyper-parameters
# that are stored in the iteration list.  This function
# should train the neural network according to the
# hyper-parameters and keep note of the best trained
# network so far.
  perform_iteration(iteration)
# Rotate the axes forward one unit, like a carŠs
# odometer.
  indexes[0] = indexes[0] + 1;
  var counterIdx = 0;
# roll forward the other places, if needed
  while not done and  indexes[counterIdx]>=
        len(axes [counterIdx]):
    indexes[counterIdx] = 0
    counterIdx = counterIdx + 1
    if counterIdx>=len(axes):
      done = true
    else:
      indexes[counterIdx] = indexes[counterIdx] + 1
```

The code above uses two loops to pass through every possible set of the hyper-parameters. The first loop continues while the program is still producing hyper-parameters. Each time through, this loop increases the first hyper-parameter to the next value. The second loop detects if the first hyper-parameter has rolled over. The inner loop keeps moving forward to the next hyper-parameter until no more rollovers occur. Once all the hyper-parameters roll over, the process is done.

As you can see, the grid search can quickly result in a large number of iterations. Consider if you wished to search for the optimal number of hidden neurons on five layers, where you allowed up to 200 neurons on each level. This value would be equal to 200 multiplied by itself five times, or 200 to the fifth power. This process results in 320 billion iterations. Because each iteration involves training a neural network, iterations can take minutes, hours or even days to execute.

When performing grid searches, multi-threading and grid processing can be beneficial. Running the iterations through a thread pool can greatly speed up the search. The thread pool should have a size equal to the number of cores on the computer's machine. This trait allows a machine with eight cores to work on eight neural networks simultaneously. The training of the individual models must be single threaded when you run the iterations simultaneously. Many frameworks will use all available cores to train a single neural network. When you have a large number of neural networks to train, you should always train several neural networks in parallel, running them one a time so that each network uses the machines cores.

11.3.2 Random Search Model Selection

It is also possible to use a random search for model selection. Instead of systematically trying every hyper-parameter combination, the random search method chooses random values for hyper-parameters. For numeric ranges, you no longer need to specify a step value, the random model selection will choose a continuous range of floating point numbers between your specified beginning and ending points. For a random search, the programmer typically specifies either a time or an iteration limit. The following shows a random search, using the same axes as above, but it is limited to ten iterations:

```
Iteration #1: [3.298266736790538][sigmoid]
Iteration #2: [9.569985574809834][ReLU]
Iteration #3: [1.241154231596738][sigmoid]
Iteration #4: [9.140498645836487][sigmoid]
Iteration #5: [8.041758658131585][tanh]
Iteration #6: [2.363519841339439][ReLU]
Iteration #7: [9.72388393455185][tanh]
Iteration #8: [3.411276006139815][tanh]
Iteration #9: [3.1166220877785236][sigmoid]
Iteration #10: [8.559433702612296][sigmoid]
```

As you can see, the first axis, which is the hidden neuron count, is now taking on floating-point values. You can solve this problem by rounding the neuron count to the nearest whole number. It is also advisable to avoid retesting the same hyper-parameters more than once. As a result, the program should keep a list of previously tried hyper-parameters so that it doesn't repeat any hyper-parameters that were with a small range of a previously tried set.

The following URL uses Javascript to show random search in action:

`http://www.heatonresearch.com/aifh/vol3/random_iter.html`

11.3.3 Other Model Selection Techniques

Model selection is a very active area of research, and, as a result, many innovative ways exist to perform it. Think of the hyper-parameters as a vector of values and the process of finding the best neural network score for those hyper-parameters as an objective function. You can consider these hyper-parameters as an optimization problem. We have previously examined many optimization algorithms in earlier volumes of this book series. These algorithms are the following:

- Ant Colony Optimization (ACO)
- Genetic Algorithms
- Genetic Programming
- Hill Climbing
- Nelder-Mead
- Particle Swarm Optimization (PSO)
- Simulated Annealing

We examined many of these algorithms in detail in Volumes 1 and 2 of *Artificial Intelligence for Humans*. Although the list of algorithms is long, the reality is that most of these algorithms are not suited for model selection because the objective function for model selection is computationally expensive. It might take minutes, hours or even days to train a neural network and determine how well a given set of hyper-parameters can train a neural network.

Nelder-Mead and sometimes hill climbing turn out to be the best options if you wish to apply an optimization function to model selection. These algorithms attempt to minimize calls to the objective function. Calls to the objective function are very expensive for a parameter search because a neural network must be trained. A good technique for optimization is to generate a set of hyper-parameters to use as a starting point for Nelder-Mead and allow Nelder-Mead to improve these hyper-parameters. Nelder-Mead is a good choice for a hyper-parameter search because it results in a relatively small number of calls to the objective function.

Model selection is a very common part of Kaggle data science competitions. Based on competition discussions and reports, most participants use grid and random searches for model selection.. Nelder-Mead is also popular. Another technique that is gaining in popularity is the use of Bayesian optimization, as described by Snoek, Larochelle, Hugo & Adams (2012). An implementation of this algorithm, written in Python, is called Spearmint, and you can find it at the following URL:

`https://github.com/JasperSnoek/spearmint`

Bayesian optimization is a relatively new technique for model selection on which we have only recently conducted research. Therefore, this current book does not contain a more profound examination of it. Future editions may include more information of this technique.

11.4 Chapter Summary

As you learned in this chapter, it is possible to prune neural networks. Pruning a neural network removes connections and neurons in order to make the neural network more efficient. Execution speed, number of connections, and error are all measures of efficiency. Although neural networks must be effective at recognizing patterns, efficiency is the main goal of pruning. Several different algorithms can prune a neural network. In this chapter, we examined two of these algorithms. If your neural network is already operating sufficiently fast, you must evaluate whether the pruning is justified. Even when efficiency is important, you must weigh the trade-offs between efficiency and a reduction in the effectiveness of your neural network.

Model selection plays a significant role in neural network development. Hyper-parameters are settings such as hidden neuron, layer count, and activation function selection. Model selection is the process of finding the set of hyper-parameters that will produce the best-trained neural network. A variety of algorithms can search through the possible settings of the hyper-parameters and find the best set.

Pruning can sometimes lead to a decrease in the tendency for neural networks to overfit. This reduction in overfitting is typically only a byproduct of the pruning process. Algorithms that reduce overfitting are called regularization algorithms. Although pruning will sometimes have a regularizing effect, an entire group of algorithms, called regularization algorithms, exist to reduce overfitting. We will focus exclusively on these algorithms in the next chapter.

Chapter 12

Dropout and Regularization

- Regularization
- L1 & L2 Regularization
- Dropout Layers

Regularization is a technique that reduces overfitting, which occurs when neural networks attempt to memorize training data, rather than learn from it. Humans are capable of overfitting as well. Before we examine the ways that a machine accidentally overfits, we will first explore how humans can suffer from it.

Human programmers often take certification exams to show their competence in a given programming language. To help prepare for these exams, the test makers often make practice exams available. Consider a programmer who enters a loop of taking the practice exam, studying more, and then taking the practice exam again. At some point, the programmer has memorized much of the practice exam, rather than learning the techniques necessary to figure out the individual questions. The programmer has now overfit to the practice exam. When this programmer takes the real exam, his actual score will likely be lower than what he earned on the practice exam.

A computer can overfit as well. Although a neural network received a high score on its training data, this result does not mean that the same neural network will score high on data that was not inside the training set. Regularization is one of the techniques that can prevent overfitting. A number of different regularization techniques exist. Most work by analyzing and potentially modifying the weights of a neural network as it trains.

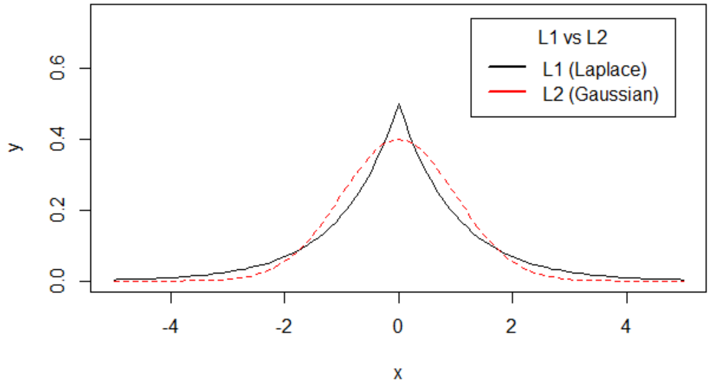
12.1 L1 and L2 Regularization

L1 and L2 regularization are two common regularization techniques that can reduce the effects of overfitting (Ng, 2004). Both of these algorithms can either work with an objective function or as a part of the backpropagation algorithm. In both cases the regularization algorithm is attached to the training algorithm by adding an additional objective.

Both of these algorithms work by adding a weight penalty to the neural network training. This penalty encourages the neural network to keep the weights to small values. Both L1 and L2 calculate this penalty differently. For gradient-descent-based algorithms, such as backpropagation, you can add this penalty calculation to the calculated gradients. For objective-function-based training, such as simulated annealing, the penalty is negatively combined with the objective score.

Both L1 and L2 work differently in the way that they penalize the size of a weight. L1 will force the weights into a pattern similar to a Gaussian distribution; the L2 will force the weights into a pattern similar to a Laplace distribution, as demonstrated by Figure 12.1:

Figure 12.1: L1 vs L2



As you can see, L1 algorithm is more tolerant of weights further from 0, whereas the L2 algorithm is less tolerant. We will highlight other important differences between L1 and L2 in the following sections. You also need to note that both L1 and L2 count their penalties based only on weights; they do not count penalties on bias values.

12.1.1 Understanding L1 Regularization

You should use L1 regularization to create sparsity in the neural network. In other words, the L1 algorithm will push many weight connections to near 0. When a weight is near 0, the program drops it from the network. Dropping weighted connections will create a sparse neural network.

Feature selection is a useful byproduct of sparse neural networks. Features are the values that the training set provides to the input neurons. Once all the weights of an input neuron reach 0, the neural network training determines that the feature is unnecessary. If your data set has a large number of input features that may not be needed, L1 regularization can help the neural network detect and ignore unnecessary features.

Equation 12.1 shows the penalty calculation performed by L1:

$$E_1 = \lambda_1 \sum_w |w| \tag{12.1}$$

Essentially, a programmer must balance two competing goals. He or she must decide the greater value of achieving a low score for the neural network or regularizing the weights. Both results have value, but the programmer has to choose the relative importance. If regularization is the main goal, the λ (lambda) value determines that the L1 objective is more important than the neural network's error. A value of 0 means L1 regularization is not considered at all. In this case, a low network error would have more importance. A value of 0.5 means L1 regularization is half as important as the error objective. Typical L1 values are below 0.1 (10%).

The main calculation performed by L1 is the summing of the absolute values (as indicated by the vertical bars) of all the weights. The bias values are not summed.

If you are using an optimization algorithm, such as simulated annealing, you can simply combine the value returned by Equation 12.1 to the score. You should combine this value to the score in such a way so that it has a negative effect. If you are trying to minimize the score, then you should add the L1 value. Similarly, if you are trying to maximize the score, then you should subtract the L1 value.

If you are using L1 regularization with a gradient-descent-based training algorithm, such as backpropagation, you need to use a slightly different error term, as shown by Equation 12.2:

$$E_1 = \frac{\lambda_1}{n} \sum_w |w| \tag{12.2}$$

Equation 12.2 is nearly the same as Equation 12.1 except that we divide by n. The value n represents the number of training set evaluations. For example, if there were 100 training set elements and three output neurons, n would be 300. We derive this number because the program has three values to evaluate for each of those 100 elements. It is necessary to divide by n because the program applies Equation 12.2 at every training evaluation. This characteristic contrasts with Equation 12.1, which is applied once per training iteration.

To use Equation 12.2, we need to take its partial derivative with respect to the weight. Equation 12.3 shows the partial derivative of Equation 12.2:

$$\frac{\partial}{\partial w} E_1 = \frac{\lambda_1}{n} sgn(w) \tag{12.3}$$

To use this gradient, we add this value to every weight gradient calculated by the gradient-descent algorithm. This addition is only performed for weight values; the biases are left alone.

12.1.2 Understanding L2 Regularization

You should use L2 regularization when you are less concerned about creating a space network and are more concerned about low weight values. The lower weight values will typically lead to less overfitting.

Equation 12.4 shows the penalty calculation performed by L2:

$$E_2 = \lambda_2 \sum_w w^2 \tag{12.4}$$

Like the L1 algorithm, the λ (lambda) value determines how important the L2 objective is compared to the neural network's error. Typical L2 values are below 0.1 (10%). The main calculation performed by L2 is the summing of the squares of all of the weights. The bias values are not summed.

If you are using an optimization algorithm, such as simulated annealing, you can simply combine the value returned by Equation 12.4 to the score. You should combine this value with the score in such a way so that it has a negative effect. If you are trying to minimize the score, then you should add the L2 value. Similarly, if you are trying to maximize the score, then you should subtract the L2 value.

If you are using L2 regularization with a gradient-descent-based training algorithm, such as backpropagation, you need to use a slightly different error term, as shown by Equation 12.5:

$$E_2 = \frac{\lambda_2}{n} \sum_w w^2 \tag{12.5}$$

Equation 12.5 is nearly the same as Equation 12.4, except that, unlike L1, we take the squares of the weights. To use Equation 12.5, we need to take the partial derivative with respect to the weight. Equation 12.6 shows the partial derivative of Equation 12.6:

$$\frac{\partial}{\partial w} E_2 = \frac{\lambda_2}{n} w \tag{12.6}$$

To use this gradient, you need to add this value to every weight gradient calculated by the gradient-descent algorithm. This addition is only performed on weight values; the biases are left alone.

12.2 Dropout Layers

Hinton, Srivastava, Krizhevsky, Sutskever, & Salakhutdinov (2012) introduced the dropout regularization algorithm. Although dropout works in a different way than L1 and L2, it accomplishes the same goal–the prevention of overfitting. However, the algorithm goes about the task by actually removing neurons and connections–at least temporarily. Unlike L1 and L2, no weight penalty is added. Dropout does not directly seek to train small weights.

Dropout works by causing hidden neurons of the neural network to be unavailable during part of the training. Dropping part of the neural network causes the remaining portion to be trained to still achieve a good score even without the dropped neurons. This decreases coadaption between neurons, which results in less overfitting.

12.2.1 Dropout Layer

Most neural network frameworks implement dropout as a separate layer. Dropout layers function as a regular, densely connected neural network layer. The only difference is that the dropout layers will periodically drop some of their neurons during training. You can use dropout layers on regular feedforward neural networks. In fact, they can also become layers in convolutional LeNET-5 networks like we studied in Chapter 10, “Convolutional Neural Networks.”

The usual hyper-parameters for a dropout layer are the following:

- Neuron Count
- Activation Function
- Dropout Probability

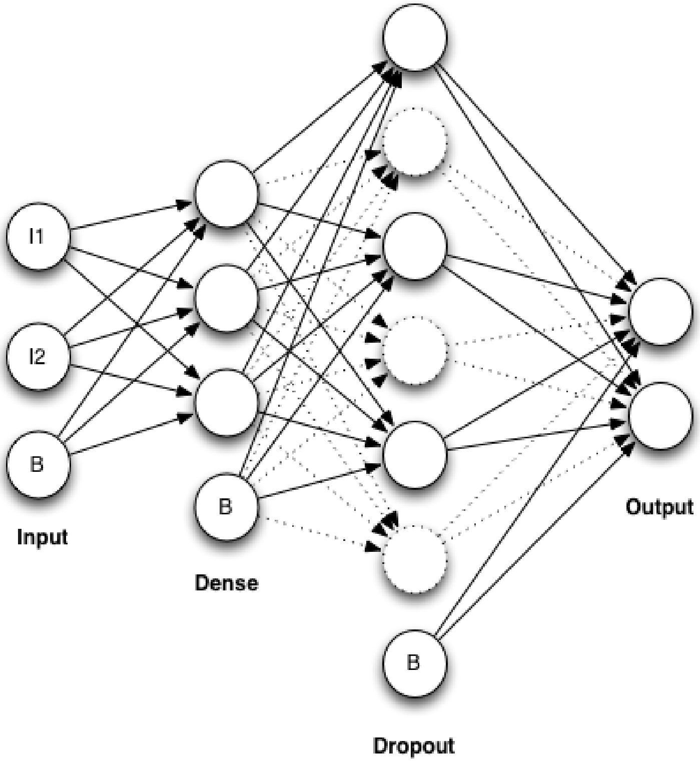
The neuron count and activation function hyper-parameters work exactly the same way as their corresponding parameters in the dense layer type mentioned in Chapter 10, "Convolutional Neural Networks." The neuron count simply specifies the number of neurons in the dropout layer. The dropout probability indicates the likelihood of a neuron dropping out during the training iteration. Just as it does for a dense layer, the program specifies an activation function for the dropout layer.

12.2.2 Implementing a Dropout Layer

The program implements a dropout layer as a dense layer that can eliminate some of its neurons. Contrary to popular belief about the dropout layer, the program does not permanently remove these discarded neurons. A dropout layer does not lose any of its neurons during the training process, and it will still have exactly the same number of neurons after training. In this way, the program only temporarily masks the neurons rather than dropping them.

Figure 12.2 shows how a dropout layer might be situated with other layers:

Figure 12.2: Dropout Layer



The discarded neurons and their connections are shown as dashed lines. The input layer has two input neurons as well as a bias neuron. The second layer is a dense layer with three neurons as well as a bias neuron. The third layer is a dropout layer with six regular neurons even though the program has dropped 50% of them. While the program drops these neurons, it neither calculates nor trains them. However, the final neural network will use all of these neurons for the output. As previously mentioned, the program only temporarily discards the neurons.

During subsequent training iterations, the program chooses different sets of neurons from the dropout layer. Although we chose a probability of 50% for dropout, the computer will not necessarily drop three neurons. It is as if we flipped a coin for each of the dropout candidate neurons to choose if that neuron was dropped out. You must know that the program should never drop the bias neuron. Only the regular neurons on a dropout layer are candidates.

The implementation of the training algorithm influences the process of discarding neurons. The dropout set frequently changes once per training iteration or batch. The program can also provide intervals where all neurons are present. Some neural network frameworks give additional hyper-parameters to allow you to specify exactly the rate of this interval.

Why dropout is capable of decreasing overfitting is a common question. The answer is that dropout can reduce the chance of a codependency developing between two neurons. Two neurons that develop a codependency will not be able to operate effectively when one is dropped out. As a result, the neural network can no longer rely on the presence of every neuron, and it trains accordingly. This characteristic decreases its ability to memorize the information presented to it, thereby forcing generalization.

Dropout also decreases overfitting by forcing a bootstrapping process upon the neural network. Bootstrapping is a very common ensemble technique. We will discuss ensembling in greater detail in Chapter 16, "Modeling with Neural Networks." Basically, ensembling is a technique of machine learning that combines multiple models to produce a better result than those achieved by individual models. Ensemble is a term that originates from the musical ensembles in which the final music product that the audience hears is the combination of many instruments.

Bootstrapping is one of the most simple ensemble techniques. The programmer using bootstrapping simply trains a number of neural networks to perform exactly the same task. However, each of these neural networks will perform differently because of some training techniques and the random numbers used in the neural network weight initialization. The difference in weights causes the performance variance. The output from this ensemble of neural networks becomes the average output of the members taken together. This process decreases overfitting through the consensus of differently trained neural networks.

Dropout works somewhat like bootstrapping. You might think of each neural network that results from a different set of neurons being dropped out as an individual member in an ensemble. As training progresses, the program creates more neural networks in this way. However, dropout does not require the same amount of processing as does bootstrapping. The new neural networks created are temporary; they exist only for a training iteration. The final result is also a single neural network, rather than an ensemble of neural networks to be averaged together.

12.3 Using Dropout

In this chapter, we will continue to evolve the book's MNIST handwritten digits example. We examined this data set in the book introduction and used it in several examples.

The example for this chapter uses the training set to fit a dropout neural network. The program subsequently evaluates the test set on this trained network to view the results. Both dropout and non-dropout versions of the neural network have results to examine.

The dropout neural network used the following hyper-parameters:

- Activation Function: ReLU
- Input Layer: 784 (28x28)
- Hidden Layer 1: 1000

- Dropout Layer: 500 units, 50%
- Hidden Layer 2: 250
- Output Layer: 10 (because there are 10 digits)

We selected the above hyper-parameters through experimentation. By rounding the number of input neurons up to the next even unit, we chose a first hidden layer of 1000. The next three layers constrained this amount by half each time. Placing the dropout layer between the two hidden layers provided the best improvement in the error rate. We also tried placing it both before hidden layer 1 and after hidden layer 2. Most of the overfitting occurred between the two hidden layers.

We used the following hyper-parameters for the regular neural network. This process is essentially the same as the dropout network except that an additional hidden layer replaces the dropout layer.

- Activation Function: ReLU
- Input Layer: 784 (28x28)
- Hidden Layer 1: 1000
- Hidden Layer 2: 500
- Hidden Layer 3: 250
- Output Layer: 10 (because there are 10 digits)

The results are shown here:

```
Relu:
Best valid loss was 0.068229 at epoch 17.
Incorrect 170/10000 (1.7000000000000002%)
ReLU+Dropout:
Best valid loss was 0.065753 at epoch 5.
Incorrect 120/10000 (1.2%)
```

As you can see, dropout neural network achieved a better error rate than the ReLU only neural network from earlier in the book. By reducing the amount of overfitting, the dropout network got a better score. You should also notice that, although the non-dropout network did achieve a better training score, this result is not good. It indicates overfitting. Of course, these results will vary, depending on the platform used.

12.4 Chapter Summary

We introduced several regularization techniques that can reduce overfitting. When the neural network memorizes the input and expected output, overfitting occurs because the program has not learned to generalize. Many different regularization techniques can force the neural network to learn to generalize. We examined L1, L2, and dropout. L1 and L2 work similarly by imposing penalties for weights that are too large. The purpose of these penalties is to reduce complexity in the neural network. Dropout takes an entirely different approach by randomly removing various neurons and forcing the training to continue with a partial neural network.

The L1 algorithm penalizes large weights and forces many of the weights to approach 0. We consider the weights that contain a zero value to be dropped from the neural network. This reduction produces a sparse neural network. If all weighted connections between an input neuron and the next layer are removed, you can assume that the feature connected to that input neuron is unimportant. Feature selection is choosing input features based on their importance to the neural network. The L2 algorithm penalizes large weights, but it does not tend to produce neural networks that are as sparse as those produced by the L1 algorithm.

Dropout randomly drops neurons in a designated dropout layer. The neurons that were dropped from the network are not gone as they were in pruning. Instead, the dropped neurons are temporarily masked from the neural network. The set of dropped neurons changes during each training iteration. Dropout forces the neural network to continue functioning when neurons are removed. This makes it difficult for the neural network to memorize and overfit.

So far, we have explored only feedforward neural networks in this volume.

In this type of network, the connections only move forward from the input layer to hidden layers and ultimately to the output layer. Recurrent neural networks allow backward connections to previous layers. We will analyze this type of neural network in the next chapter.

Additionally, we have focused primarily on using neural networks to recognize patterns. We can also teach neural networks to predict future trends. By providing a neural network with a series of time-based values, it can predict subsequent values. In the next chapter, we will also demonstrate predictive neural networks. We refer to this type of neural network as a temporal neural network. Recurrent neural networks can often make temporal predictions.

Chapter 13

Time Series and Recurrent Networks

- Time Series
- Elman Networks
- Jordan Networks
- Deep Recurrent Networks

In this chapter, we will examine time series encoding and recurrent networks, two topics that are logical to put together because they are both methods for dealing with data that spans over time. Time series encoding deals with representing events that occur over time to a neural network. There are many different methods to encode data that occur over time to a neural network. This encoding is necessary because a feedforward neural network will always produce the same output vector for a given input vector. Recurrent neural networks do not require encoding of time series data because they are able to automatically handle data that occur over time.

The variation in temperature during the week is an example of time series data. For instance, if we know that today's temperature is 25 degrees, and tomorrow's temperature is 27 degrees, the recurrent neural networks and time series encoding provide another option to predict the correct temperature for the week. Conversely, a traditional feedforward neural network will always respond with the same output for a given input. If a feedforward neural network is trained to predict tomorrow's temperature, it should respond 27 for 25. The fact that it will always output 27 when given 25 might be a hindrance to its predictions. Surely the temperature of 27 will not always follow 25. It would be better for the neural network to consider the temperatures for a series of days before the day being predicted. Perhaps the temperature over the last week might allow us to predict tomorrow's temperature. Therefore, recurrent neural networks and time series encoding represent two different approaches to the problem of representing data over time to a neural network.

So far the neural networks that we've examined have always had forward connections. The input layer always connects to the first hidden layer. Each hidden layer always connects to the next hidden layer. The final hidden layer always connects to the output layer. This manner to connect layers is the reason that these networks are called "feedforward." Recurrent neural networks are not so rigid, as backward connections are also allowed. A recurrent connection links a neuron in a layer to either a previous layer or the neuron itself. Most recurrent neural network architectures maintain state in the recurrent connections. Feedforward neural networks don't maintain any state. A recurrent neural network's state acts as a sort of short-term memory for the neural network. Consequently, a recurrent neural network will not always produce the same output for a given input.

13.1 Time Series Encoding

As we saw in previous chapters, neural networks are particularly good at recognizing patterns, which helps them predict future patterns in data. We refer to a neural network that predicts future patterns as a predictive, or temporal, neural network. These predictive neural networks can anticipate future events, such as stock market trends and sun spot cycles.

Many different kinds of neural networks can predict. In this section, the feedforward neural network will attempt to learn patterns in data so it can predict future values. Like all problems applied to neural networks, prediction is a matter of intelligently determining how to configure input and interpret output neurons for a problem. Because the type of feedforward neural networks in this book always produce the same output for a given input, we need to make sure that we encode the input correctly.

A wide variety of methods can encode time series data for a neural network. The sliding window algorithm is one of the simplest and most popular encoding algorithms. However, more complex algorithms allow the following considerations:

- Weighting older values as less important than newer
- Smoothing/averaging over time
- Other domain-specific (e.g. finance) indicators

We will focus on the sliding window algorithm encoding method for time series. The sliding window algorithm works by dividing the data into two windows that represent the past and the future. You must specify the sizes of both windows. For example, if you want to predict future prices with the daily closing price of a stock, you must decide how far into the past and future that you wish to examine. You might want to predict the next two days using the last five closing prices. In this case, you would have a neural network with five input neurons and two output neurons.

13.1.1 Encoding Data for Input and Output Neurons

Consider a simple series of numbers, such as the sequence shown here:

```
1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1
```

A neural network that predicts numbers from this sequence might use three input neurons and a single output neuron. The following training set has a prediction window of size 1 and a past window size of 3:

```
[1,2,3] -> [4]
[2,3,4] -> [3]
[3,4,3] -> [2]
[4,3,2] -> [1]
```

As you can see, the neural network is prepared to receive several data samples in a sequence. The output neuron then predicts how the sequence will continue. The idea is that you can now feed any sequence of three numbers, and the neural network will predict the fourth number. Each data point is called a time slice. Therefore, each input neuron represents a known time slice. The output neurons represent future time slices.

It is also possible to predict more than one value into the future. The following training set has a prediction window of size 2 and a past window size of 3:

```
[1,2,3] -> [4,3]
[2,3,4] -> [3,2]
[3,4,3] -> [2,1]
[4,3,2] -> [1,2]
```

The last two examples have only a single stream of data. It is possible to use multiple streams of data to predict. For example, you might predict the price with the price of a stock and its volume. Consider the following two streams:

```
Stream #1: 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1
Stream #2: 10, 20, 30, 40, 30, 20, 10, 20, 30, 40, 30, 20, 10
```

You can predict stream #1 with stream #1 and #2. You simply need to add the stream #2 values next to the stream #1 values. A training set can perform this calculation. The following set shows a prediction window of size 1 and a past window size of 3:

```
[1,10,2,20,3,30] -> [4]
[2,20,3,30,4,40] -> [3]
[3,30,4,40,3,30] -> [2]
[4,40,3,30,2,20] -> [1]
```

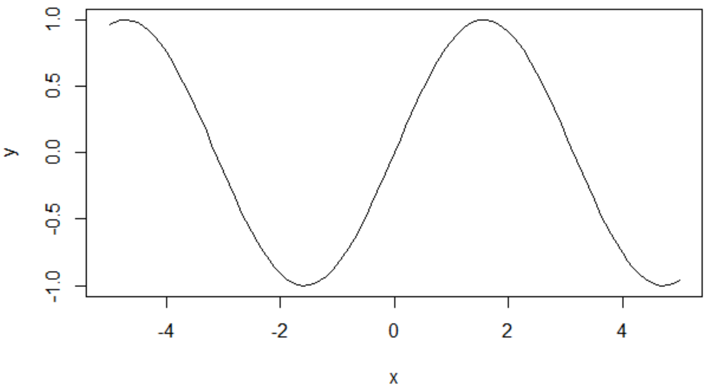
This same technique works for any number of streams. In this case, stream #1 helps to predict itself. For example, you can use the stock prices of IBM and Apple to predict Microsoft. This technique uses three streams. The stream that we're predicting doesn't need to be among the streams providing the data to form the prediction.

13.1.2 Predicting the Sine Wave

The example in this section is relatively simple. We present a neural network that predicts the sine wave, which is mathematically predictable. However, programmers can easily understand the sine wave, and it varies over time. These qualities make it a good introduction to predictive neural networks.

You can see the sine wave by plotting the trigonometric sine function. Figure 13.1 shows the sine wave:

Figure 13.1: The sine wave



The sine wave function trains the neural network. Backpropagation will adjust the weights to emulate the sine wave. When you first execute the sine wave example, you will see the results of the training process. Typical output from the sine wave predictor's training process follows:

```
Iteration #1 Error:0.48120350975475823 Iteration #2 Error:
0.36753445768855236 Iteration #3 Error:0.3212066601426759
Iteration #4 Error:0.2952410514715732 Iteration #5 Error:
0.2780102928778258 Iteration #6 Error:0.26556861969786527
Iteration #7 Error:0.25605359706505776 Iteration #8 Er236
Introduction to Neural Networks with Java, Second Edition
ror:0.24842242500053566 Iteration #9 Error:0.24204767544134156
   Iteration
#10 Error:0.23653845782593882
...
```

```
Iteration #4990 Error:0.02319397662897425 Iteration #4991 Error:
0.02319310934886356 Iteration #4992 Error:0.02319224224668851 5
Iteration #4993 Error:0.02319137532183077 Iteration #4994 Error:
0.02319050857367285 8 Iteration #4995 Error:0.02318964200159761
Iteration #4996 Error:0.02318877560498862 Iteration #4997 Error:
0.02318790938322986 Iteration #4998 Error:0.02318704333570586 7
Iteration #4999 Error:0.02318617746180174 5
```

In the beginning, the error rate is fairly high at 48%. By the second iteration, this rate quickly begins to fall to 36.7%. By the time the 4,999th iteration has occurred, the error rate has fallen to 2.3%. The program is designed to stop before hitting the 5,000th iteration. This succeeds in reducing the error rate to less than 0.03.

Additional training would produce a better error rate; however, by limiting the iterations, the program is able to finish in only a few minutes on a regular computer. This program took about two minutes to execute on an Intel I7 computer.

Once the training is complete, the sine wave is presented to the neural network for prediction. You can see the output from this prediction here:

```
5:Actual=0.76604:Predicted=0.7892166200864351:Difference=2.32% 6:A
ctual=0.86602:Predicted=0.8839210963512845:Difference=1.79% 7:Ac
tual=0.93969:Predicted=0.934526031234053:Difference=0.52% 8:Act
ual=0.9848:Predicted=0.9559577688326862:Difference=2.88% 9:Actu
al=1.0:Predicted=0.9615566601973113:Difference=3.84% 10:Actual=
0.9848:Predicted=0.9558060932656686:Difference=2.90% 11:Actual=
0.93969:Predicted=0.9354447787244102:Difference=0.42% 12:Actual
=0.86602:Predicted=0.8894014978439005:Difference=2.34% 13:Actua
l=0.76604:Predicted=0.801342405700056:Difference=3.53% 14:Actua
l=0.64278:Predicted=0.6633506809125252:Difference=2.06% 15:Actu
al=0.49999:Predicted=0.4910483600917853:Difference=0.89% 16:Act
ual=0.34202:Predicted=0.3128615278064510 5:Difference=2.92% 17:A
ctual=0.17364:Predicted=0.1460832526356813 4:Difference=2.76%
18:Actual=0.0:Predicted=−0.00836001679623843 4:Difference=0.84%
19:Actual=−0.17364:Predicted=−0.1557538146013282 3:Difference=1.79%
20:Actual=−0.34202:Predicted=−0.302177515855955 9:Difference=3.98%
...
490:Actual=−0.64278:Predicted=−0.6515076637590029:Difference=0.87%
491:Actual=−0.76604:Predicted=−0.8133333939237001:Difference=4.73%
492:Actual=−0.86602:Predicted=−0.9076496572125671:Difference=4.16%
493:Actual=−0.93969:Predicted=−0.9492579517460149:Difference=0.96%
```

```
494:Actual=-0.9848:Predicted=-0.9644567437192423:Difference=2.03%
495:Actual=-1.0:Predicted=-0.9664801515670861:Difference=3.35%
496:Actual=-0.9848:Predicted=-0.9579489752650393:Difference=2.69%
497:Actual=-0.93969:Predicted=-0.9340105440194074:Difference=0.57%
498:Actual=-0.86602:Predicted=-0.8829925066754494:Difference=1.70%
499:Actual=-0.76604:Predicted=-0.7913823031308845:Difference=2.53%
```

As you can see, we present both the actual and predicted values for each element. We trained the neural network for the first 250 elements; however, the neural network is able to predict beyond the first 250. You will also notice that the difference between the actual values and the predicted values rarely exceeds 3%.

Sliding window is not the only way to encode time series. Other time series encoding algorithms can be very useful for specific domains. For example, many technical indicators exist that help to find patterns in the value of securities such as stocks, bonds, and currency pairs.

13.2 Simple Recurrent Neural Networks

Recurrent neural networks do not force the connections to flow only from one layer to the next, from input layer to output layer. A recurrent connection occurs when a connection is formed between a neuron and one of the following other types of neurons:

- The neuron itself
- A neuron on the same level
- A neuron on a previous level

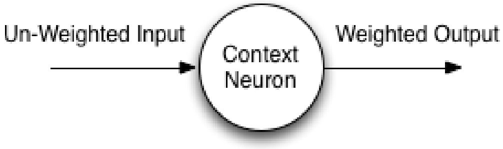
Recurrent connections can never target the input neurons or the bias neurons.

The processing of recurrent connections can be challenging. Because the recurrent links create endless loops, the neural network must have some way to know when to stop. A neural network that entered an endless loop would not be useful. To prevent endless loops, we can calculate the recurrent connections with the following three approaches:

- Context neurons
- Calculating output over a fixed number of iterations
- Calculating output until neuron output stabilizes

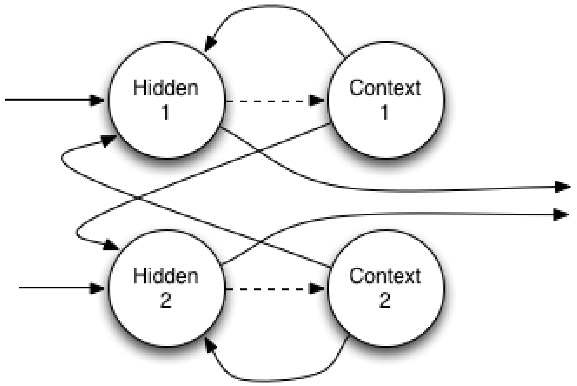
We refer to neural networks that use context neurons as a simple recurrent network (SRN). The context neuron is a special neuron type that remembers its input and provides that input as its output the next time that we calculate the network. For example, if we gave a context neuron 0.5 as input, it would output 0. Context neurons always output 0 on their first call. However, if we gave the context neuron a 0.6 as input, the output would be 0.5. We never weight the input connections to a context neuron, but we can weight the output from a context neuron just like any other connection in a network. Figure 13.2 shows a typical context neuron:

Figure 13.2: Context Neuron



Context neurons allow us to calculate a neural network in a single feed-forward pass. Context neurons usually occur in layers. A layer of context neurons will always have the same number of context neurons as neurons in its source layer, as demonstrated by Figure 13.3:

Figure 13.3: Context Layer



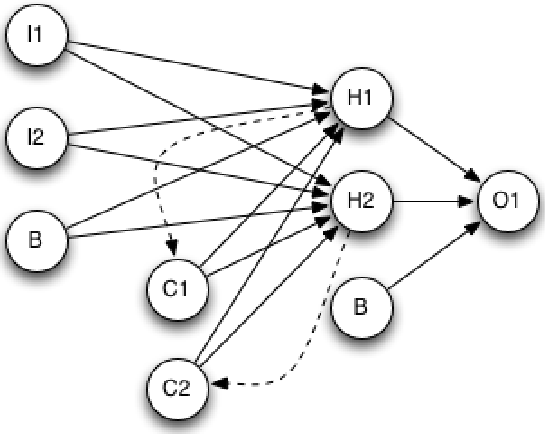
As you can see from the above layer, two hidden neurons that are labeled hidden 1 and hidden 2 directly connect to the two context neurons. The dashed lines on these connections indicate that these are not weighted connections. These weightless connections are never dense. If these connections were dense, hidden 1 would be connected to both hidden 1 and hidden 2. However, the direct connection simply joins each hidden neuron to its corresponding context neuron. The two context neurons form dense, weighted connections to the two hidden neurons. Finally, the two hidden neurons also form dense connections to the neurons in the next layer. The two context neurons would form two connections to a single neuron in the next layer, four connections to two neurons, six connections to three neurons, and so on.

You can combine context neurons with the input, hidden, and output layers of a neural network in many different ways. In the next two sections, we explore two common SRN architectures.

13.2.1 Elman Neural Networks

In 1990, Elman introduced a neural network that provides pattern recognition to time series. This neural network type has one input neuron for each stream that you are using to predict. There is one output neuron for each time slice you are trying to predict. A single-hidden layer is positioned between the input and output layer. A layer of context neurons takes its input from the hidden layer output and feeds back into the same hidden layer. Consequently, the context layers always have the same number of neurons as the hidden layer, as demonstrated by Figure 13.4:

Figure 13.4: Elman SRN

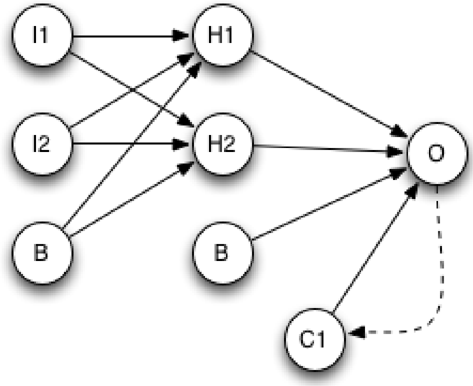


The Elman neural network is a good general-purpose architecture for simple recurrent neural networks. You can pair any reasonable number of input neurons to any number of output neurons. Using normal weighted connections, the two context neurons are fully connected with the two hidden neurons. The two context neurons receive their state from the two non-weighted connections (dashed lines) from each of the two hidden neurons.

13.2.2 Jordan Neural Networks

In 1993, Jordan introduced a neural network to control electronic systems. This style of SRN is similar to Elman networks. However, the context neurons are fed from the output layer instead of the hidden layer. We also refer to the context units in a Jordan network as the state layer. They have a recurrent connection to themselves with no other nodes on this connection, as seen in Figure 13.5:

Figure 13.5: Jordan SRN



The Jordan neural network requires the same number of context neurons and output neurons. Therefore, if we have one output neuron, the Jordan network will have a single context neuron. This equality can be problematic if you have only a single output neuron because you will be able to have just one single-context neuron.

The Elman neural network is applicable to a wider array of problems than the Jordan network because the large hidden layer creates more context neurons. As a result, the Elman network can recall more complex patterns because it captures the state of the hidden layer from the previous iteration. This state is never bipolar since the hidden layer represents the first line of feature detectors.

Additionally, if we increase the size of the hidden layer to account for a more complex problem, we also get more context neurons with an Elman network. The Jordan network doesn't produce this effect. To create more context neurons with the Jordan network, we must add more output neurons. We cannot add output neurons without changing the definition of the problem.

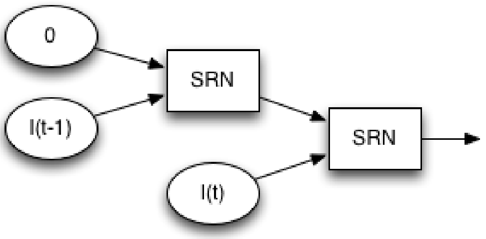
When to use a Jordan network is a common question. Programmers originally developed this network type for robotics research. Neural networks that are designed for robotics typically have input neurons connected to sensors and output neurons connected to actuators (typically motors). Because each motor has its own output neuron, neural networks for robots will generally have more output neurons than regression neural networks that predict a single value.

13.2.3 Backpropagation through Time

You can train SRNs with a variety of methods. Because SRNs are neural networks, you can train their weights with any optimization algorithm, such as simulated annealing, particle swarm optimization, Nelder-Mead or others. Regular backpropagation-based algorithms can also train of the SRN. Mozer (1995), Robinson & Fallside (1987) and Werbos (1988) each invented an algorithm specifically designed for SRNs. Programmers refer to this algorithm as backpropagation through time (BPTT). Sjoberg, Zhang, Ljung, et al. (1995) determined that backpropagation through time provides superior training performance than general optimization algorithms, such as simulated annealing. Backpropagation through time is even more sensitive to local minima than standard backpropagation.

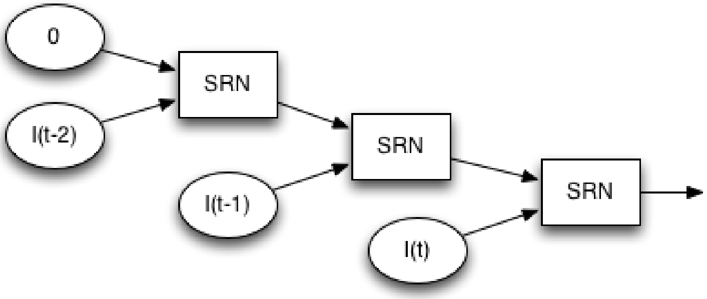
Backpropagation through time works by unfolding the SRN to become a regular neural network. To unfold the SRN, we construct a chain of neural networks equal to how far back in time we wish to go. We start with a neural network that contains the inputs for the current time, known as *t*. Next we replace the context with the entire neural network, up to the context neuron's input. We continue for the desired number of time slices and replace the final context neuron with a 0. Figure 13.6 illustrates this process for two time slices.

Figure 13.6: Unfolding to Two Time Slices



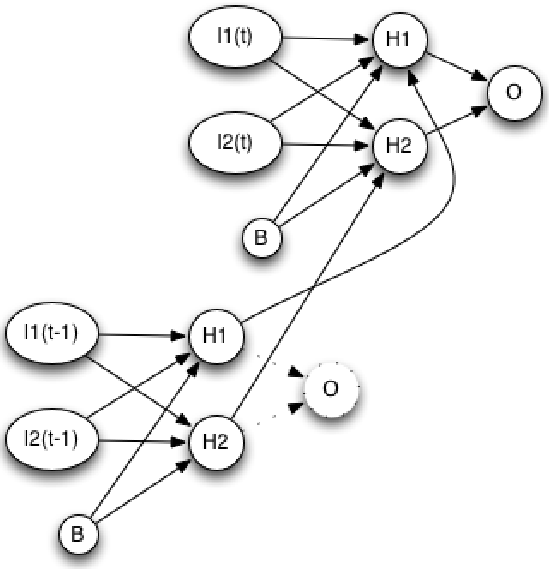
This unfolding can continue deeper; Figure 13.7 shows three time slices:

Figure 13.7: Unfolding to Two Time Slices



You can apply this abstract concept to the actual SRNs. Figure 13.8 illustrates a two-input, two-hidden, one-output Elman neural network unfolded to two time slices:

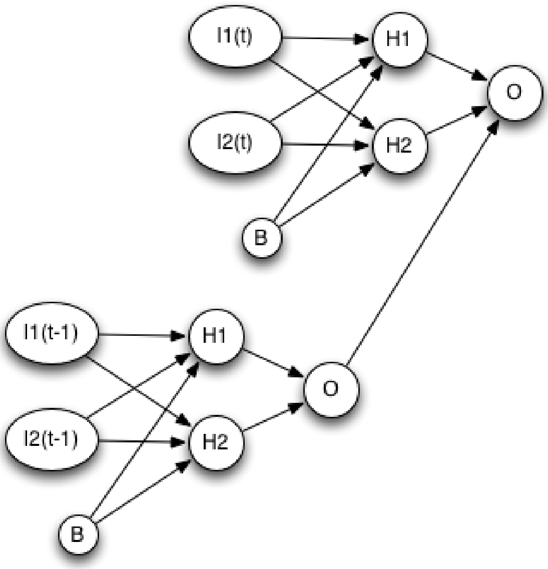
Figure 13.8: Elman Unfolded to Two Time Slices



As you can see, there are inputs for both *t* (current time) and *t-1* (one time slice in the past). The bottom neural network stops at the hidden neurons because you don't need everything beyond the hidden neurons to calculate the context input. The bottom network structure becomes the context to the top network structure. Of course, the bottom structure would have had a context as well that connects to its hidden neurons. However, because the output neuron above does not contribute to the context, only the top network (current time) has one.

It is also possible to unfold a Jordan neural network. Figure 13.9 shows a two-input, two-hidden, one-output Jordan network unfolded to two time slices.

Figure 13.9: Jordan Unfolded to Two Time Slices



Unlike the Elman network, you must calculate the entire Jordan network to determine the context. As a result, we can calculate the previous time slice (bottom network) all the way to the output neuron.

To train the SRN, we can use regular backpropagation to train the unfolded network. However, at the end of the iteration, we average the weights of all folds to obtain the weights for the SRN. Listing 13.1 describes the BPTT algorithm:

Listing 13.1: Backpropagation Through Time (BPTT):

```
def bptt(a, y)
# a[t] is the input at time t. y[t] is the output
   .. unfold the network to contain k instances of f
   .. see above figure..
   while stopping criteria no met:
# x is the current context
     x = []
     for t from 0 to n Ű 1:
# t is time. n is the length of the training sequence
     .. set the network inputs to x, a[t], a[t+1], ..., a[t+k−1]
```

```
    p = .. forward−propagation of the inputs
        .. over the whole unfolded network
# error = target − prediction
    e = y[t+k] − p
    .. Back−propagate the error, e, back across
    .. the whole unfolded network

    .. Update all the weights in the network
    .. Average the weights in each instance of f together,
    .. so that each f is identical
# compute the context for the next time−step
    x = f(x)
```

13.2.4 Gated Recurrent Units

Although recurrent neural networks have never been as popular as the regular feedforward neural networks, active research on them continues. Chung, Hyun & Bengio (2014) introduced the gated recurrent unit (GRU) to allow recurrent neural networks to function in conjunction with deep neural network by solving some inherent limitations of recurrent neural networks. GRUs are neurons that provide a similar role to the context neurons seen previously in this chapter.

It is difficult to train RNNs to capture long-term dependencies because the gradients tend to either vanish (most of the time) or explode (rarely, but with severe effects), as demonstrated by Chung, Hyun & Bengio (2015).

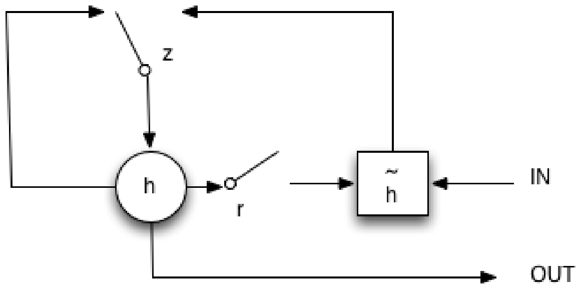
As of the 2015 publication of this book, GRUs are less than a year old. Because of the cutting edge nature of GRUs, most major neural network frameworks do not currently include them. If you would like to experiment with GRUs, the Python Theano-based framework Keras includes them. You can find the framework at the following URL:

`https://github.com/fchollet/keras`

Though we usually use Lasange, Keras is one of many Theano-based frameworks for Python, and it is also one of the first to support GRUs. This section contains a brief, high-level introduction to GRU, and we will update the book's examples as needed to support this technology as it becomes available. Refer to the book's example code for up-to-date information on example availability for GRU.

A GRU uses two gates to overcome these limitations, as shown in Figure 13.10:

Figure 13.10: Gated Recurrent Unit (GRU)



The gates are indicated by *z*, the update gate, and *r*, the reset gate. The values *h* and *tilde-h* represent the activation (output) and candidate activation. It is important to note that the switches specify ranges, rather than simply being on or off.

The primary difference between the GRU and traditional recurrent neural networks is that the entire context value does not change its value each iteration as it does in the SRN. Rather, the update gate governs the degree of update to the context activation that occurs. Additionally, the program provides a reset gate that allows the context to be reset.

13.3 Chapter Summary

In this chapter, we introduced several methods that can handle time series data with neural networks. A feedforward neural network produces the same output when provided the same input. As a result, feedforward neural networks are said to be deterministic. This quality does not allow a feedforward neural network the ability to produce output, given a series of inputs. If your application must provide output based on a series of inputs, you have two choices. You can encode the time series into an input feature vector or use a recurrent neural network.

Encoding a time series is a way of capturing time series information in a feature vector that is fed to a feedforward neural network. A number of methods encode time series data. We focused on sliding window encoding. This method specifies two windows. The first window determines how far into the past to use for prediction. The second window determines how far into the future to predict.

Recurrent neural networks are another method to deal with time series data. Encoding is not necessary with a recurrent neural network because it is able to remember previous inputs to the neural network. This short-term memory allows the neural network to be able to see patterns in time. Simple recurrent networks use a context neuron to remember the state from previous computations. We examined Elman and Jordan SRNs. Additionally, we introduced a very new neuron type called the gated recurrent unit (GRU). This neuron type does not immediately update its context value like the Elman and Jordan networks. Two gates govern the degree of update.

Hyper-parameters define the structure of a neural network and ultimately determine its effectiveness for a particular problem. In the previous chapters of this book, we introduced hyper-parameters such as the number of hidden layers and neurons, the activation functions, and other governing attributes of neural networks. Determining the correct set of hyper-parameters is often a difficult task of trial and error. However, some automated processes can make this process easier. Additionally, some rules of thumb can help architect these neural networks. We cover these pointers, as well as automated processes, in the next chapter.

Chapter 14

Architecting Neural Networks

- Hyper-parameters
- Learning Rate & Momentum
- Hidden Structure
- Activation Functions

Hyper-parameters, as mentioned in previous chapters, are the numerous settings for models such as neural networks. Activation functions, hidden neuron counts, layer structure, convolution, max-pooling and dropout are all examples of neural network hyper-parameters. Finding the optimal set of hyper-parameters can seem a daunting task, and, indeed, it is one of the most time-consuming tasks for the AI programmer. However, do not fear, we will provide you with a summary of the current research on neural network architecture in this chapter. We will also show you how to conduct experiments to help determine the optimal architecture for your own networks.

We will make architectural recommendations in two ways. First, we will report on recommendations from scientific literature in the field of AI. These recommendations will include citations so that you can examine the original paper. However, we will strive to present the key concept of the article in an approachable manner. The second way will be through experimentation. We will run several competing architectures and report the results.

You need to remember that a few hard and fast rules do not dictate the optimal architecture for every project. Every data set is different, and, as a result, the optimal neural network for every data set is also different. Thus, you must always perform some experimentation to determine a good architecture for your network.

14.1 Evaluating Neural Networks

Neural networks start with random weights. Additionally, some training algorithms use random values as well. All considered, we're dealing with quite a bit of randomness in order to make comparisons. Random number seeds are a common solution to this issue; however, a constant seed does not provide an equal comparison, given that we are evaluating neural networks with different neuron counts.

Let's compare a neural network with 32 connections against another network with 64 connections. While the seed guarantees that the first 32 connections retain the same value, there are now 32 additional connections that will have new random values. Furthermore, those 32 weights in the first network might not be in the same locations in the second network if the random seed is maintained between only the two initial weight sets.

To compare architectures, we must perform several training runs and average the final results. Because these extra training runs add to the total runtime of the program, excessive numbers of runs will quickly become impractical. It can also be beneficial to choose a training algorithm that is deterministic (one that does not use random numbers). The experiments that we will perform in this chapter will use five training runs and the resilient propagation (RPROP) training algorithm. RPROP is deterministic, and five runs are an arbitrary choice that provides a reasonable level of consistency. Using the Xavier weight initialization algorithm, introduced in Chapter 4, "Feedforward Neural Networks," will also help provide consistent results.

14.2 Training Parameters

Training algorithms themselves have parameters that you must tune. We don't consider the parameters related to training as hyper-parameters because they are not evident after a neural network has been trained. You can examine a trained neural network to determine easily what hyper-parameters are present. A simple examination of the network reveals the neuron counts and activation function in use. However, determining training parameters such as learning rate and momentum is not possible. Both training parameters and hyper-parameters greatly affect the error rates that the neural network can obtain. However, we can use training parameters only during the actual training.

The three most common training parameters for neural networks are listed here:

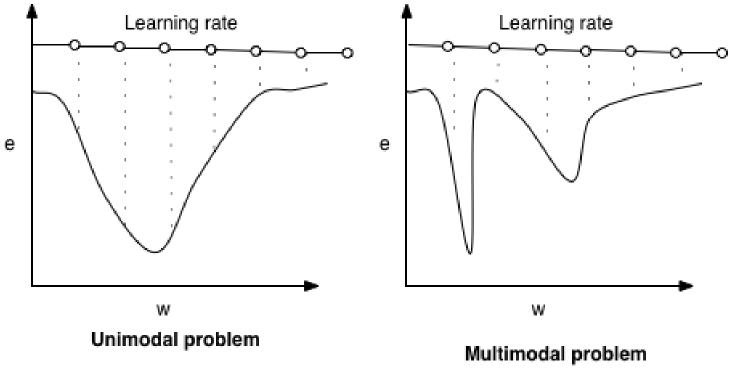
- Learning Rate
- Momentum
- Batch Size

Not all learning algorithms have these parameters. Additionally, you can vary the values chosen for these parameters as learning progresses. We discuss these training parameters in the subsequent sections.

14.2.1 Learning Rate

The learning rate allows us to determine how far each iteration of training will take the weight values. Some problems are very simple to solve, and a high training rate will yield a quick solution. Other problems are more difficult, and a quick learning might disregard a good solution. Other than the runtime of your program, there is no disadvantage in choosing a small learning rate. Figure 14.1 shows how a learning rate might fare on both a simple (unimodal) and complex (multimodal) problem:

Figure 14.1: Learning Rates



The above two charts show the relationship between weight and the score of a network. As the program increases or decreases a single weight, the score changes. A unimodal problem is typically easy to solve because its graph has only one bump, or optimal solution. In this case, we consider a good score to be a low error rate.

A multimodal problem has many bumps, or possible good solutions. If the problem is simple (unimodal), a fast learning rate is optimal because you can charge up the hill to a great score. However, haste makes waste on the second chart, as the learning rate fails to find the two optimums.

Kamiyama, Iijima, Taguchi, Mitsui, et al. (1992) stated that most literature use a learning rate of 0.2 and a momentum of 0.9. Often this learning rate and momentum can be good starting points. In fact, many examples do use these values. The researchers suggest that Equation 14.1 has a strong likelihood of attaining better results.

$$\epsilon = K(1-\alpha) \tag{14.1}$$

The variable α (alpha) is the momentum; ϵ (epsilon) is the learning rate, and K is a constant related to the hidden neurons. Their research suggests that the tuning of momentum (discussed in the next section) and learning rate are related. We define the constant K by the number of hidden neurons. Smaller numbers of hidden neurons should use a larger K. In our own experimentations, we do not use the equation directly because it is difficult to choose a concrete value of K. The following calculations show several learning rates based on learning rate and K.

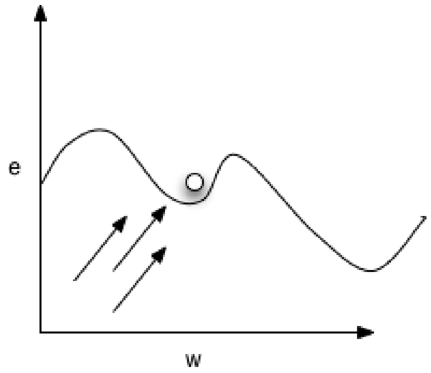
```
k=0.500000, alpha=0.200000 -> epsilon=0.400000
k=0.500000, alpha=0.300000 -> epsilon=0.350000
k=0.500000, alpha=0.400000 -> epsilon=0.300000
k=1.000000, alpha=0.200000 -> epsilon=0.800000
k=1.000000, alpha=0.300000 -> epsilon=0.700000
k=1.000000, alpha=0.400000 -> epsilon=0.600000
k=1.500000, alpha=0.200000 -> epsilon=1.200000
k=1.500000, alpha=0.300000 -> epsilon=1.050000
k=1.500000, alpha=0.400000 -> epsilon=0.900000
```

The lower values of K represent higher hidden neuron counts; therefore the hidden neuron count is decreasing as you move down the list. As you can see, for all momentums (α, alpha) of 0.2, the suggested learning rate (ϵ, epsilon) increases as the hidden neuron counts decrease. The learning rate and momentum have an inverse relationship. As you increase one, you should decrease the other. However, the hidden neuron count controls how quickly momentum and learning rate should diverge.

14.2.2 Momentum

Momentum is a learning property that causes the weight change to continue in its current direction, even if the gradient indicates that the weight change should reverse direction. Figure 14.2 illustrates this relationship:

Figure 14.2: Momentum and a Local Optima



A positive gradient encourages the weight to decrease. The weight has followed the negative gradient up the hill but now has settled into a valley, or a local optima. The gradient now moves to 0 and positive as the other side of the local optima is hit. Momentum allows the weight to continue in this direction and possibly escape from the local-optima valley and possibly find the lower point to the right.

To understand exactly how learning rate and momentum are implemented, recall Equation 6.6, from Chapter 6, "Backpropagation Training," that is repeated as Equation 14.2 for convenience:

$$\Delta w_{(t)} = -\epsilon \frac{\partial E}{\partial w_{(t)}} + \alpha \Delta w_{(t-1)} \tag{14.2}$$

This equation shows how we calculate the change in weight for training iteration t. This change is the sum of two terms that are each governed by the learning rate ϵ (epsilon) and momentum α (alpha). The gradient is the weight's partial derivative of the error rate. The sign of the gradient determines if we should increase or decrease the gradient. The learning rate simply tells backpropagation the percentage of this gradient that the program should apply to the weight change. The program always applies this change to the original weight and then retains it for the next iteration. The momentum α (alpha) subsequently determines the percentage of the previous iteration's weight change that the program should apply to this iteration. Momentum allows the previous iteration's weight change to carry through to the current iteration. As a result, the weight change maintains its direction. This process essentially gives it "momentum."

Jacobs (1988) discovered that learning rate should be decreased as training progresses. Additionally, as previously discussed, Kamiyama, et al. (1992) asserted that momentum should be increased as the learning rate is decayed. A decreasing learning rate, coupled with an increasing momentum, is a very common pattern in neural network training. The high learning rate allows the neural network to begin exploring a larger area of the search space. Decreasing the learning rate forces the network to stop exploring and begin exploiting a more local region of the search space. Increasing momentum at this point helps guard against local minima in this smaller search region.

14.2.3 Batch Size

The batch size specifies the number of training set elements that you must calculate before the weights are actually updated. The program sums all of the gradients for a single batch before it updates the weights. A batch size of 1 indicates that the weights are updated for each training set element. We refer to this process as online training. The program often sets the batch size to the size of the training set for full batch training.

A good starting point is a batch size equal to 10% of the entire training set. You can increase or decrease the batch size to see its effect on training efficiency. Usually a neural network will have vastly fewer weights than training set elements. As a result, cutting the batch size by a half, or even a fourth, will not have a drastic effect on the runtime of an iteration in standard backpropagation.

14.3 General Hyper-Parameters

In addition to the training parameters just discussed, we must also consider the hyper-parameters. They are significantly more important than training parameters because they determine the neural networks ultimate learning capacity. A neural network with a reduced learning capacity cannot overcome this deficiency with further training.

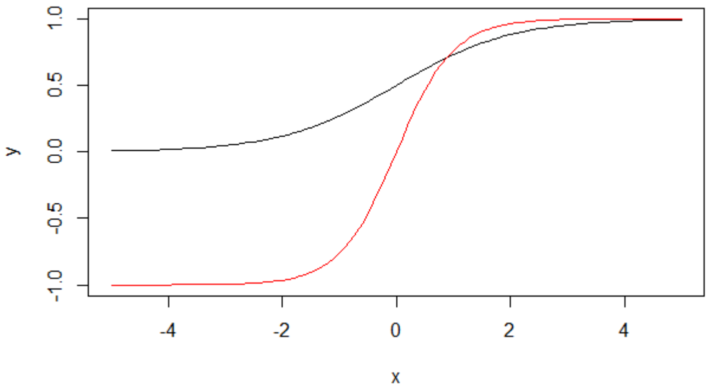
14.3.1 Activation Functions

Currently, the program utilizes two primary types of activation functions inside of a neural network:

- Sigmoidal: Logistic (sigmoid) & Hyperbolic Tangent (tanh)
- Linear: ReLU

The sigmoidal (s-shaped) activation functions have been a mainstay of neural networks, but they are now losing ground to the ReLU activation function. The two most common s-shaped activation functions are the namesake sigmoid activation function and the hyperbolic tangent activation function. The name can cause confusion because sigmoid refers both to an actual activation function and to a class of activation functions. The actual sigmoid activation function has a range between 0 and 1, whereas the hyperbolic tangent function has a range of -1 and 1. We will first tackle hyperbolic tangent versus sigmoid (the activation function). Figure 14.3 shows the overlay of these two activations:

Figure 14.3: Sigmoid and Tanh



As you can see from the figure, the hyperbolic tangent stretches over a much larger range than tanh. Your choice of these two activations will affect the way that you normalize your data. If you are using hyperbolic tangent at the output layer of your neural network, you must normalize the expected outcome between -1 and 1. Similarly, if you are using the sigmoid function for the output layer, you must normalize the expected outcome between -1 and 1. You should normalize the input to -1 to 1 for both of these activation functions. The x-values (input) above +1 will saturate to +1 (y-values) for both sigmoid and hyperbolic tangent. As x-values go below -1, the sigmoid activation function saturates to y-values of 0, and hyperbolic tangent saturates to y-values of -1.

The saturation of sigmoid to values of 0 in the negative direction can be problematic for training. As a result, Kalman & Kwasny (1992) recommend hyperbolic tangent in all situations instead of sigmoid. This recommendation corresponds with many literature sources. However, this argument only extends to the choice between sigmoidal activation functions. A growing body of recent research favors the ReLU activation function in all cases over the sigmoidal activation functions.

Zeiler et al. (2014), Maas, Hannun, Awni & Ng (2013) and Glorot, Bordes & Bengio (2013) all recommend the ReLU activation function over its sigmoidal counterparts. "Chapter 9, "Deep Learning," includes the advantages of the ReLU activation function. In this section, we will examine an experiment that compares the ReLU to the sigmoid, we used a neural network with a hidden layer of 1,000 neurons. We ran this neural network against the MNIST data set. Obviously, we adjusted the number of input and output neurons to match the problem. We ran each activation function five times with different random weights and kept the best results:

```
Sigmoid:
Best valid loss was 0.068866 at epoch 43.
Incorrect 192/10000 (1.92%)
ReLU:
Best valid loss was 0.068229 at epoch 17.
Incorrect 170/10000 (1.7000000000000002%)
```

The accuracy rates for each of the above neural networks on a validation set. As you can see, the ReLU activation function did indeed have the lowest error rate and achieved it in fewer training iterations/epochs. Of course, these results will vary, depending on the platform used.

14.3.2 Hidden Neuron Configurations

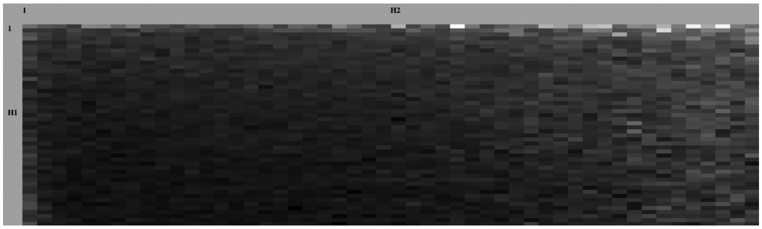
Hidden neuron configurations have been a frequent source of questions. Neural network programmers often wonder exactly how to structure their networks. As of the writing of this book, a quick scan of Stack Overflow shows over 50 questions related to hidden neuron configurations. You can find the questions at the following link:

`http://goo.gl/ruWpcb`

Although the answers may vary, most of them simply advise that the programmer "experiment and find out." According to the universal approximation theorem, a single-hidden-layer neural network can theoretically learn any pattern (Hornik, 1991). Consequently, many researchers suggest only single-hidden-layer neural networks. Although a single-hidden-layer neural network can learn any pattern, the universal approximation theorem does not state that this process is easy for a neural network. Now that we have efficient techniques to train deep neural networks, the universal approximation theorem is much less important.

To see the effects of hidden neurons and neuron counts, we will perform an experiment that will look at one-layer and two-layer neural networks. We will try every combination of hidden neurons up to two 50-neuron layers. This neural network will use a ReLU activation function and RPROP. This experiment took over 30 hours to run on an Intel I7 quad-core. Figure 14.4 shows a heat map of the results:

Figure 14.4: Heat Map of Two-Layer Network (first experiment)



The best configuration reported by the experiment was 35 neurons in hidden layer 1, and 15 neurons in hidden layer 2. The results of this experiment will vary when repeated. The above diagram shows the best-trained networks in the lower-left corner, as indicated by the darker squares. This indicates that the network favors a large first hidden layer with smaller second hidden layers. The heat map shows the relationships between the different configurations. We achieved better results with smaller neuron counts on the second hidden layer. This occurred because the neuron counts constricted the information flow to the output layer. This approach is consistent with research into auto-encoders in which successively smaller layers force the neural network to generalize information, rather than overfit. In general, based on the experiment here, we advise using at least two hidden layers with successively smaller layers.

14.4 LeNet-5 Hyper-Parameters

The LeNet-5 convolutional neural networks introduce additional layer types that bring more choices in the construction of neural networks. Both the convolutional and max-pooling layers create other choices for hyper-parameters. Chapter 10, “Convolutional Neural Networks” contains a complete list of hyper-parameters that the LeNet-5 network introduces. In this section, we will review LeNet-5 architectural recommendations recently suggested in scientific papers.

Most literature on LeNet-5 networks supports the use of a max-pool layer to follow every convolutional layer. Ideally, several convolutional/max-pool layers reduce the resolution at each step. Chapter 6, "Convolutional Neural Networks" includes this demonstration. However, very recent literature seems to indicate that max-pool layers should not be used at all.

On November 7, 2014, the website Reddit featured Dr. Geoffrey Hinton for an "ask me anything (AMA)" session. Dr. Hinton is the foremost researcher in deep learning and neural networks. During the AMA session, Dr. Hinton was asked about max-pool layers. You can read his complete response here:

`https://goo.gl/TgBakL`

Overall, Dr. Hinton begins his answer saying, "The pooling operation used in convolutional neural networks is a big mistake, and the fact that it works so well is a disaster." He then proceeds with a technical description of why you should never use max-pooling. At the time of this book's publication, his response is fairly recent and controversial. Therefore we suggest that you try the convolutional neural networks both with and without max-pool layers, as their future looks uncertain.

14.5 Chapter Summary

Selecting a good set of hyper-parameters is one of the most difficult tasks for the neural network programmer. The number of hidden neurons, activation functions, and layer structures are all examples of neural network hyper-parameters that the programmer must adjust and fine-tune. All these hyper-parameters can affect the overall capacity of the neural network to learn patterns. As a result, you must choose them correctly.

Most current literature suggests using the ReLU activation function in place of the sigmoidal (s-shaped) activation functions. If you are going to use a sigmoidal activation, most literature recommends that you use the hyperbolic tangent activation function instead of the sigmoidal activation function. The ReLU activation function is more compatible with deep neural networks.

The number of hidden layers and neurons is also an important hyper-parameter for neural networks. It is generally advisable that successive hidden layers contain a smaller number of neurons than the immediately previous layer. This adjustment has the effect of constraining the data from the inputs and forcing the neural network to generalize and not memorize, which results in overfitting.

We do not consider training parameters as hyper-parameters because they do not affect the structure of the neural network. However, you still must choose a proper set of training parameters. The learning rate and momentum are two of the most common training parameters for neural networks. Generally, you should initially set the learning rate high and decrease it as training continues. You should move the momentum value inversely with the learning rate.

In this chapter, we examined how to structure neural networks. While we provided some general recommendations, the data set generally drives the architecture of the neural network. Consequently, you must analyze the data set. We will introduce the t-SNE dimension reduction algorithm in the next chapter. This algorithm will allow you to visualize graphically your data set and see issues that occur while you are creating a neural network for that data set.

Chapter 15

Visualization

- Confusion Matrices
- PCA
- t-SNE

We frequently receive the following question about neural networks: “I’ve created a neural network, but when I train it, my error never goes to an acceptable level. What should I do?” The first step in this investigation is to determine if one of the following common errors has occurred.

- Correct number of input and output neurons
- Data set normalized correctly
- Some fatal design decision of the neural network

Obviously, you must have the correct number of input neurons to match how your data are normalized. Likewise, you should have a single-output neuron for regression problems or usually one output neuron per class for a classification problem. You should normalize input data to fit the activation function that you use. In a similar way, fatal mistakes, such as no hidden layer or a learning rate of 0, can create a bad situation.

However, once you eliminate all these errors, you must look to your data. For classification problems, your neural network may have difficulties differentiating between certain pairs of classes. To help you resolve this issue, some visualization algorithms exist that allow you to see the problems that your neural network might encounter. The two visualizations presented in this chapter will show the following issues with data:

- Classes that are easily confused for others
- Noisy data
- Dissimilarity between classes

We describe each issue in the subsequent sections and offer some potential solutions. We will present these potential solutions in the form of two algorithms of increasing complexity. Not only is the topic of visualization important for data analysis, it was also chosen as a topic by the readers of this book, which earned its initial funding through a Kickstarter campaign. The project's original 653 backers chose visualization from among several competing project topics. As a result, we will present two visualizations. Both examples will use the MNIST handwritten digits data set that we have examined in previous chapters of this book.

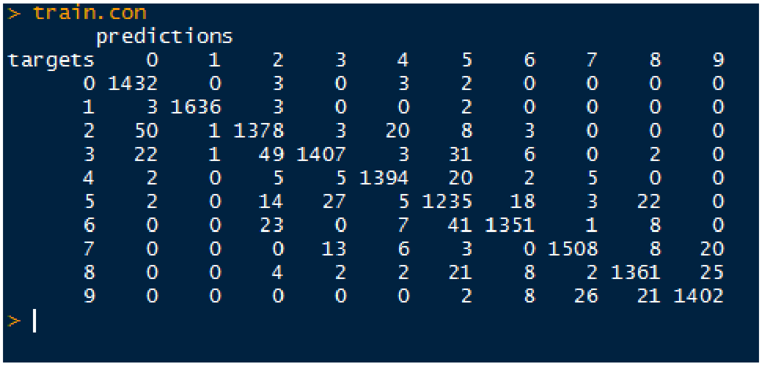
15.1 Confusion Matrix

A neural network trained for the MNIST data set should be able to take a handwritten digit and predict what digit was actually written. Some digits are more easily confused for others. Any classification neural network has the possibility of misclassifying data. A confusion matrix can measure these misclassifications.

15.1.1 Reading a Confusion Matrix

A confusion matrix is always presented as a square grid. The number of rows and columns will both be equal to the number of classes in your problem. For MNIST, this will be a 10x10 grid, as shown by Figure 15.1:

Figure 15.1: MNIST Confusion Matrix



```
> train.con
         predictions
targets    0    1    2    3    4    5    6    7    8    9
      0 1432    0    3    0    3    2    0    0    0    0
      1    3 1636    3    0    0    2    0    0    0    0
      2   50    1 1378    3   20    8    3    0    0    0
      3   22    1   49 1407    3   31    6    0    2    0
      4    2    0    5    5 1394   20    2    5    0    0
      5    2    0   14   27    5 1235   18    3   22    0
      6    0    0   23    0    7   41 1351    1    8    0
      7    0    0    0   13    6    3    0 1508    8   20
      8    0    0    4    2    2   21    8    2 1361   25
      9    0    0    0    0    0    2    8   26   21 1402
>
```

A confusion matrix uses the columns to represent predictions. The rows represent what would have been a correct prediction. If you look at row 0 column 0, you will see the number 1,432. This result means that the neural network correctly predicted a "0" 1,432 times. If you look at row 3 column 2, you will see that a "2" was predicted 49 times when it should have been a "3." The problem occurred because it's easy to mistake a handwritten "3" for a "2," especially when a person with bad penmanship writes the numbers. The confusion matrix lets you see which digits are commonly mistaken for each other. Another important aspect of the confusion matrix is the diagonal from (0,0) to (9,9). If the program trains the neural network properly, the largest numbers should be in the diagonal. Thus, a perfectly trained neural network will only have numbers in the diagonal.

15.1.2 Generating a Confusion Matrix

You can create a confusion matrix with the following steps:

- Separate the data set into training and validation.
- Train a neural network on the training set.
- Set the confusion matrix to all zeros.
- Loop over every element in the validation set.

- For every element, increase the cell: row=expected, column=predicted.
- Report the confusion matrix.

Listing 15.1 shows this process in the following pseudocode:

Listing 15.1: Compute a Confusion Matrix

```
# x - contains dataset inputs
# y - contains dataset expected values (ordinals, not strings)
def confusion_matrix(x,y,network):
# Create square matrix equal to number of classifications
  confusion = matrix( network.num_classes, network.num_classes)
# Loop over every element
  for i from 0 to len(x):
    prediction = net.compute(x[i])
      target = y[i]
      confusion[target][prediction] = confusion[target][prediction
         ] + 1
# Return result
  return confusion
```

Confusion matrices are one of the classic visualizations for classification data problems. You can use them with any classification problem, not just neural networks.

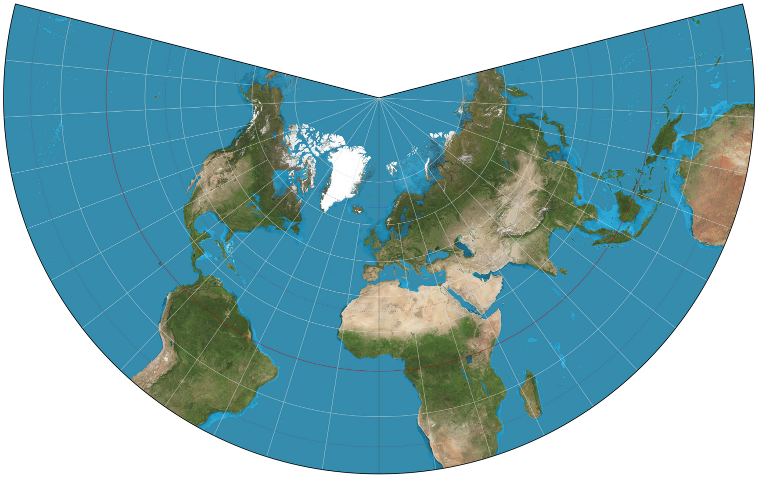
15.2 t-SNE Dimension Reduction

The t-Distributed Stochastic Neighbor Embedding (t-SNE) is a type of dimensionality reduction algorithm that programmers frequently use for visualization. We will first define dimension reduction and show its advantages for visualization and problem simplification.

The dimensions of a data set are the number of input (x) values that the program uses to make predictions. The classic iris data set has four dimensions because we measure the iris flowers in four dimensions. Chapter 4, "Feedforward Networks," has an explanation of the iris data set. The MNIST digits are images of 28x28 grayscale pixels, which result in a total of 784 input neurons (28 x 28). As a result, the MNIST data set has 784 dimensions.

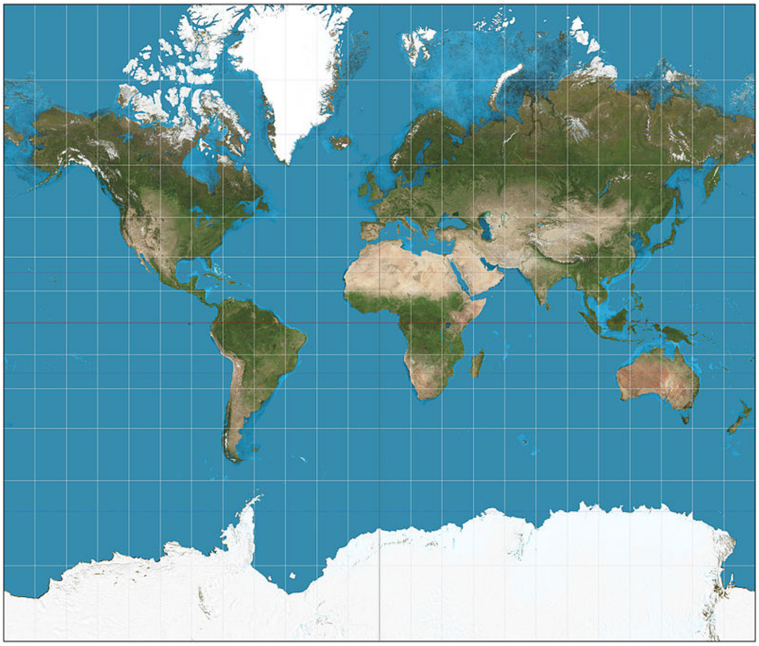
For dimensionality reduction, we need to ask the following question: “Do we really need 784 dimensions or could we project this data set into fewer dimensions?” Projections are very common in cartography. Earth exists in at least three dimensions that we can directly observe. The only true three-dimensional map of Earth is a globe. However, globes are inconvenient to store and transport. As long as it still contains the information that we require, a flat (2D) representation of Earth is useful for spaces where a globe will not fit. We can project the globe on a 2D surface in many ways. Figure 15.2 shows the Lambert projection (from Wikipedia) of Earth:

Figure 15.2: Lambert Projection (cone)



Johann Heinrich Lambert introduced the Lambert projection in 1772. Conceptually, this projection works by placing a cone over some region of the globe and projecting the image onto the globe. Once the cone is unrolled, you have a flat 2D map. Accuracy is better near the tip of the cone and worsens towards the base of the cone. The Lambert projection is not the only way to project the globe and produce a map, Figure 15.3 shows the popular Mercator projection:

Figure 15.3: Mercator Projection (cylinder)



Gerardus Mercator presented the Mercator projection in 1569. This projection works by essentially wrapping a cylinder about the globe at the equator. Accuracy is best at the equator and worsens near the poles. You can see this characteristic by examining the relative size of Greenland in both projections. Along with the two projections just mentioned, many other types exist. Each is designed to show Earth in ways that are useful for different applications.

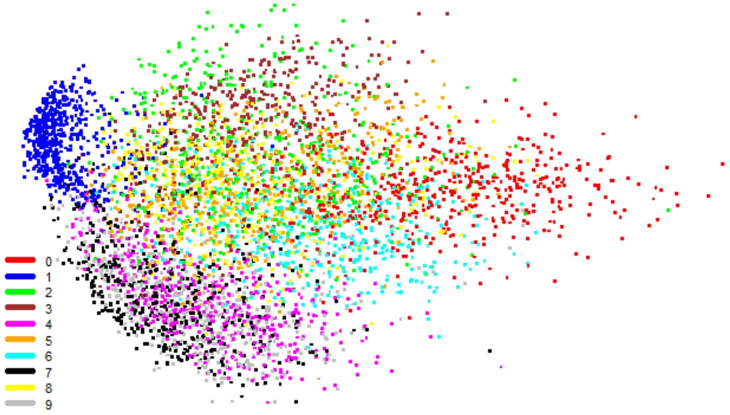
The projections above are not strictly 2D because they create a type of third dimension with other aspects like color. The map projections can convey additional information such as altitude, ground cover, or even political divisions with color. Computer projections also utilize color, as we will discover in the next section.

15.2.1 t-SNE as a Visualization

If we can reduce the MNIST 764 dimensions down to two or three with a dimension reduction algorithm, then we can visualize the data set. Reducing to two dimensions is popular because an article or a book can easily capture the visualization. It is important to remember that a 3D visualization is not actually 3D, as true 3D displays are extremely rare, as of the writing of this book. A 3D visualization will be rendered onto a 2D monitor. As a result, it is necessary to “fly” through the space and see how parts of the visualization really appear. This flight through space is very similar to a computer video game where you do not see all aspects of a scene until you fly completely around the object being viewed. Even in the real world, you cannot see both the front and back of an object you are holding–it is necessary to rotate the object with your hands to see all sides.

Karl Pearson in 1901 invented one of the most common dimensionality reduction algorithms. Principal component analysis (PCA) creates a number of principal components that match the number of dimensions to be reduced. For a 2D reduction, there would be two principal components. Conceptually, PCA is attempting to pack the higher-dimensional items into the principal components that maximize the amount of variability in the data. By ensuring that the distant values in high-dimensional space remain distant, PCA can complete its function. Figure 15.4 shows a PCA reduction of the MNIST digits to two dimensions:

Figure 15.4: 2D PCA Visualization of MNIST



The first principal component is the x-axis (left and right). As you can see, the matrix positions the blue dots (0’s) at the far left, and the red dots

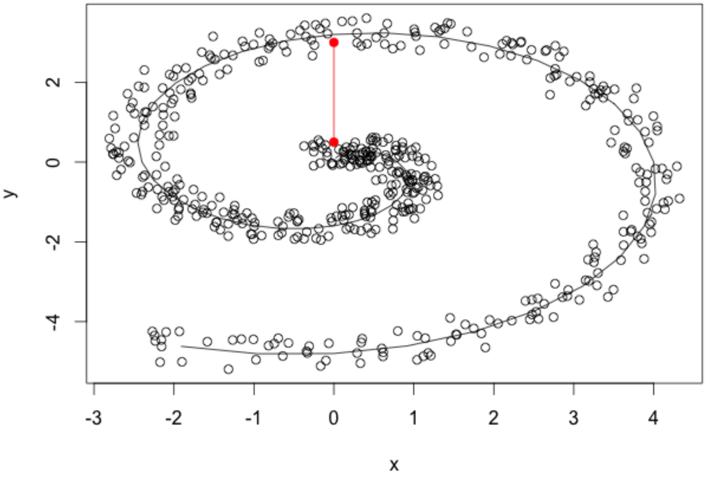
(1's) are placed towards the right. Handwritten 1's and 0's are the easiest to differentiate–they have the highest variability. The second principal component is the y-axis (up and down). On the top, you have green (2's) and brown (3's), which look somewhat similar. On the bottom are purple (4's), gray (9's) and black (7's), which also look similar. Yet the variability between these two groups is high–it is easier to tell 2's and 3's from 4's, 9's and 7's.

Color is very important to the above image. If you are reading this book in a black-and-white form, this image may not make as much sense. The color represents the digit that PCA classified. You must note that PCA and t-SNE are both unsupervised; therefore, they do not know the identities of the input vectors. In other words, they don't know which digit was selected. The program adds the colors so that we can see how well PCA classified the digits. If the above diagram is black and white in your version, you can see that the program did not place the digits into many distinct groups. We can therefore conclude that PCA does not work well as a clustering algorithm.

The above figure is also very noisy because the dots overlap in large regions. The most well-defined region is blue, where the "1" digits reside. You can also see that purple (4), black (7), and gray (9) are easy to confuse. Additionally, brown (3), green (2), and yellow (8) can be misleading.

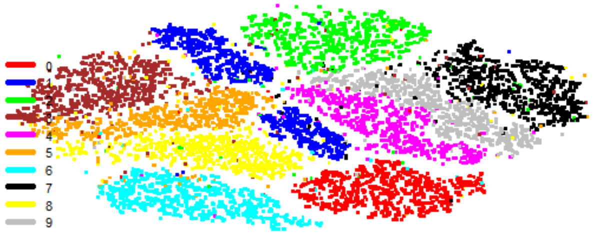
PCA analyzes the pair-wise distances of all data points and preserves large distances. As previously stated, if two points are distant in PCA, they will remain distant. However, we have to question the importance of distance. Consider Figure 15.5 that shows two points that appear to be somewhat close:

Figure 15.5: Apparent Closeness on a Spiral



The points in question are the two red, solid points that are connected by a line. The two points, when connected by a straight line, are somewhat close. However, if the program follows the pattern in the data, the points are actually far apart, as indicated by the solid spiral line that follows all of the points. PCA would attempt to keep these two points close as they appear in Figure 15.5. The t-SNE algorithm invented by van der Maaten & Hinton (2008), works somewhat differently. Figure 15.6 shows the t-SNE visualization for the same data set as featured for PCA:

Figure 15.6: 2D PCA Visualization of MNIST



The t-SNE for the MNIST digits shows a much clearer visual for the different digits. Again, the program adds color to indicate where the digits landed. However, even in black and white, you would see some divisions between clusters. Digits located nearer to each other share similarities. The amount of noise is reduced greatly, but you can still see some red dots (0's) sprinkled in the yellow cluster (8's) and cyan cluster (6's), as well as other clusters. You can produce a visualization for a Kaggle data set using the t-SNE algorithm. We will examine this process in Chapter 16, "Modeling with Neural Networks."

Implementations of t-SNE exist for most modern programming languages. Laurens van der Maaten's home page contains a list at the following URL:

`http://lvdmaaten.github.io/tsne/`

15.2.2 t-SNE Beyond Visualization

Although t-SNE is primarily an algorithm for reducing dimensions for visualization, feature engineering also utilizes it. The algorithm can even serve as a model component. Feature engineering occurs when you create additional input features. A very simple example of feature engineering is when you consider health insurance applicants, and you create an additional feature called

BMI, based on the features weight and height, as seen in equation 15.1:

$$BMI = \frac{\text{weight in kg}}{(\text{height in meters})^2} \tag{15.1}$$

BMI is simply a calculated field that allows humans to combine height and weight to determine how healthy someone is. Such features can sometimes help neural networks as well. You can build some additional features with a data point's location in either 2D or 3D space.

In Chapter 16, "Modeling with Neural Networks," we will discuss building neural networks for the Otto Group Kaggle challenge. Several Kaggle top-ten solutions for this competition used features that were engineered with t-SNE. For this challenge, you had to organize data points into nine classes. The distance between an item and the nearest neighbor of each of the nine classes on a 3D t-SNE projection was a beneficial feature. To calculate this feature, we simply map the entire training set into t-SNE space and obtain the 3D t-SNE coordinates for each feature. Then we generate nine features with the Euclidean distance between the current data point and its nearest neighbor of each of these nine classes. Finally, the program adds these nine fields to the 92 fields already being presented to the neural network.

As a visualization or as part of the input to another model, the t-SNE algorithm provides a great deal of information to the program. The programmer can use this information to see how the data are structured, and the model gains more details on the structure of the data. Most implementations of t-SNE also contain adaptions for large data sets or for very high dimensions. Before you construct a neural network to analyze data, you should consider the t-SNE visualization. After you train the neural network to analyze its results, you can use the confusion matrix.

15.3 Chapter Summary

Visualization is an important part of neural network programming. Each data set presents unique challenges to a machine learning algorithm or a neural network. Visualization can expose these challenges, allowing you to design your approach to account for known issues in the data set. We demonstrated two visualization techniques in this chapter.

The confusion matrix is a very common visualization for machine learning classification. It is always a square matrix with rows and columns equal to the number of classes in the problem. The rows represent the expected values, and the columns represent the value that the neural network actually classified. The diagonal, where the row and column numbers are equal, represents the number of times the neural network correctly classified that particular class. A well-trained neural network will have the largest numbers along the diagonal. The other cells count the number of times a misclassification occurred between each expected class and actual value.

Although you usually run the confusion matrices after the program generates a neural network, you can run the dimension reduction visualizations beforehand to expose some challenges that might be present in your data set. You can reduce the dimensions of your data set to 2D or 3D with the t-SNE algorithm. However, it becomes less effective in dimensions higher than 3D. With the 2D dimension reduction, you can create informative scatter plots that will show the relationship between several classes.

In the next chapter, we will present a Kaggle challenge as a way to synthesize many of the topics previously discussed. We will use the t-SNE visualization as an initial. Additionally, we will decrease the neural network's tendency to overfit with the use of dropout layers.

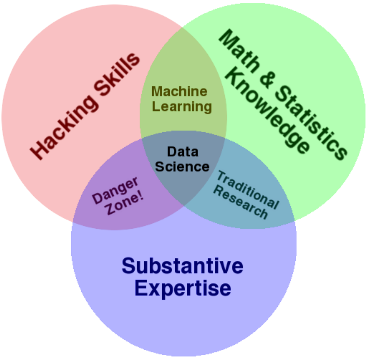
Chapter 16

Modeling with Neural Networks

- Data Science
- Kaggle
- Ensemble Learning

In this chapter, we present a capstone project on modeling, a business-oriented approach for artificial intelligence, and some aspects of data science. Drew Conway (2013), a leading data scientist, characterizes data science as the intersection of hacking skills, math and statistics knowledge, and substantive expertise. Figure 16.1 depicts this definition:

Figure 16.1: Conway's Data Science Venn Diagram



Hacking skills are essentially a subset of computer programming. Although the data scientist does not necessarily need the infrastructure knowledge of an information technology (IT) professional, these technical skills will permit him or her to create short, effective programs for processing data. In the field of data science, we refer to information processing as data wrangling.

Math and statistics knowledge covers statistics, probability, and other inferential methods. Substantive knowledge describes the business knowledge as well as the comprehension of actual data. If you combine only two of these topics, you don't have all the components for data science, as Figure 16.1 illustrates. In other words, the combination of statistics and substantive expertise is simply traditional research. Those two skills alone don't encompass the capabilities, such as machine learning, required for data science.

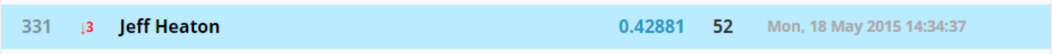
This book series deals with hacking skills and math and statistical knowledge, two of the circles in Figure 16.1. Additionally, it teaches you to create your own models, which is more pertinent to the field of computer science than data science. Substantive expertise is more difficult to obtain because it is dependent on the industry that utilizes the data science applications. For example, if you want to apply data science in the insurance industry, substantive knowledge refers to the actual business operations of these companies.

To provide a data science capstone project, we will use the Kaggle Otto Group Product Classification Challenge. Kaggle is a platform for competitive data science. You can find the Otto Group Product Classification Challenge at the following URL:

`https://www.kaggle.com/c/otto-group-product-classification-challenge`

The Otto Group was the first (and currently only) non-tutorial Kaggle competition in which we've competed. After obtaining a top 10% finish, we achieved one of the criteria for the Kaggle Master designation. To become a Kaggle Master, one must place in the top 10 of a competition once and in the top 10% of two other competitions. Figure 16.2 shows the results of our competition entry on the leaderboard:

Figure 16.2: Results in the Otto Group Product Classification Challenge



331 ↓3 Jeff Heaton 0.42881 52 Mon, 18 May 2015 14:34:37

The above line shows several pieces of information.

- We were in position 331 of 3514 (9.4%).
- We dropped three spots in the final day.
- Our multi-class log loss score was 0.42881.
- We made 52 submissions, up to May 18, 2015.

We will briefly describe the Otto Group Product Classification Challenge. For a complete description, refer to the Kaggle challenge website (found above). The Otto Group, the world's largest mail order company and currently one of the biggest e-commerce companies, introduced this challenge. Because the group has many products sold over numerous countries, they wanted to classify these products into nine categories with 93 features (columns). These 93 columns represented counts and were often 0.

The data were completely redacted (hidden). The competitors did not know the nine categories nor did they know the meaning behind the 93 features. They knew only that the features were integer counts. Like most Kaggle competitions, this challenge provided the competitors with a test and training data set. For the training data set, the competitors received the outcomes, or correct answers. For the test set, they got only the 93 features, and they had to provide the outcome.

The competition divided the test and training sets in the following way:

- Test Data: 144K rows
- Training Data: 61K rows

During the competition, participants did not submit their actual models to Kaggle. Instead, they submitted their model's predictions based on the test data. As a result, they could have used any platform to make these predictions. For this competition there were nine categories, so the competitors submitted a nine-number vector that held the probability of each of these nine categories being the correct answer.

The answer in the vector that held the highest probability was the chosen class. As you can observe, this competition was not like a multiple-choice test in school where students must submit their answer as A, B, C, or D. Instead, Kaggle competitors had to submit their answers in the following way:

- A: 80% probability
- B: 16% probability
- C: 2% probability
- D: 2% probability

College exams would not be so horrendous if students could submit answers like those in the Kaggle competition. In many multiple-choice tests, students have confidence about two of the answers and eliminate the remaining two. The Kaggle-like multiple-choice test would allow students to assign a probability to each answer, and they could achieve a partial score. In the above example, if A were the correct answer, students would earn 80% of the points.

Nevertheless, the actual Kaggle score is slightly more complex. The program grades the answers with a logarithm-based scale, and participants face heavy penalties if they have a lower probability on the correct answer. You can see the Kaggle format from the following CSV file submission:

```
1,0.0003,0.2132,0.2340,0.5468,6.2998e
    -05,0.0001,0.0050,0.0001,4.3826e-05
2,0.0011,0.0029,0.0010,0.0003,0.0001,0.5207,0.0013,0.4711,0.0011
3,3.2977e-06,4.1419e-06,7.4524e-06,2.6550e-06,5.0014e
    -07,0.9998,5.2621e-06,0.0001,6.6447e-06
4,0.0001,0.6786,0.3162,0.0039,3.3378e-05,4.1196e
    -05,0.0001,0.0001,0.0006
5,0.1403,0.0002,0.0002,6.734e
    -05,0.0001,0.0027,0.0009,0.0297,0.8255
```

As you can see, each line starts with a number that specifies the data item that is being answered. The sample above shows the answers for items one through five. The next nine values are the probabilities for each of the product classes. These probabilities must add up to 1.0 (100%).

16.0.1 Lessons from the Challenge

Having success in Kaggle requires you to understand the following topics and the corresponding tools:

- Deep Learning - Using H2O and Lasagne
- Gradient Boosting Machines (GBM) - Using XGBOOST
- Ensemble Learning - Using NumPy
- Feature Engineering - Using NumPy and Scikit-Learn
- GPU is really important for deep learning. It is best to use a deep learning package that supports it, such as H2O, Theano or Lasagne.
- The t-SNE visualization is awesome for high-dimension visualization and creating features.
- Ensembling is very important.

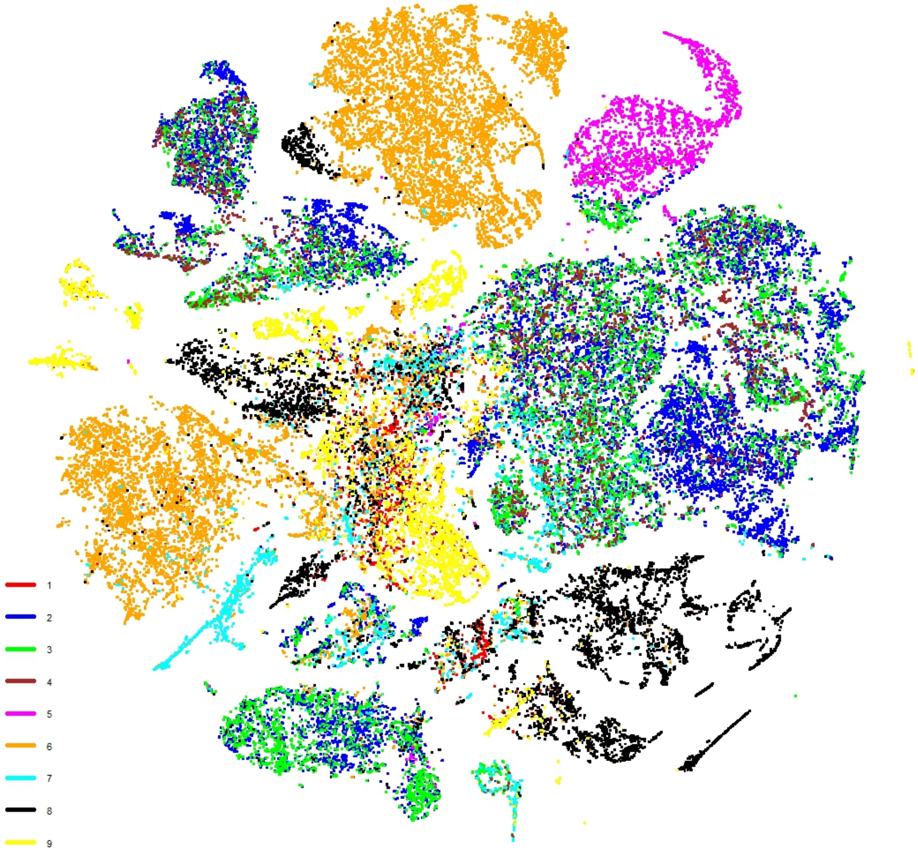
For our submission, we used Python with Scikit-Learn. However, you can use any language capable of generating a CSV file. Kaggle does not actually run your code; they score a submission file. The two most commonly used programming languages for Kaggle are R and Python. Both of these languages have strong data science frameworks available for them. R is actually a domain-specific language (DSL) for statistical analysis.

During this challenge, we learned the most about GBM parameter tuning and ensemble learning. GBMs have quite a few hyper-parameters to tune, and we became proficient at tuning a GBM. The individual scores for our GBMs were in line with those of the top 10% of the teams. However, the solution in this chapter will use only deep learning. GBM is beyond the scope of this book. In a future volume or edition of this series, we plan to examine GBM.

Although computer programmers and data scientists might typically utilize a single model like neural networks, participants in Kaggle need to use multiple models to be successful in the competition. These ensembled models produce better results than each of the models could generate independently.

We worked with t-SNE, examined in Chapter 15, “Visualization,” for the first time in this competition. This model works like principal component analysis (PCA) in that it is capable of reducing dimensions. However, the data points separate in such a way that the visualization is often clearer than PCA. The program achieves the clear visualization by using a stochastic nearest neighbor process. Figure 16.3 shows the data from the Otto Group Product Classification Challenge visualized in t-SNE:

Figure 16.3: Challenge t-SNE

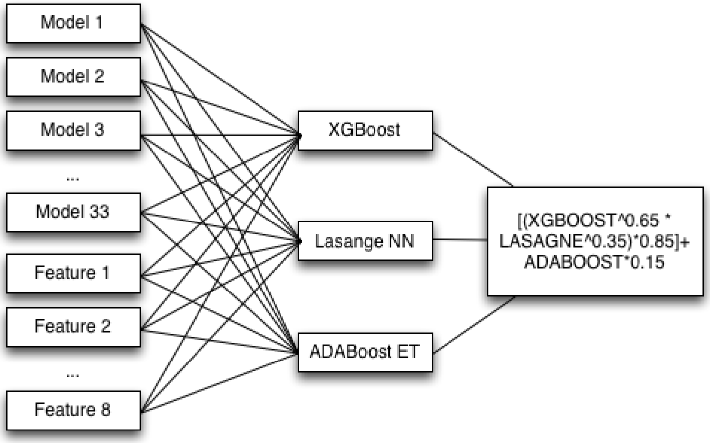


16.0.2 The Winning Approach to the Challenge

Kaggle is very competitive. Our primary objective as we entered the challenge was to learn. However, we also hoped to rank in the top 10% by the end in order to reach one of the steps in becoming a Kaggle master. Earning a top 10% was difficult; in the last few weeks of the challenge, other competitors knocked us out of the bracket almost daily. The last three days were especially turbulent. Before we reveal our solution, we will show you the winning one. The following description is based on the information publically posted about the winning solution.

The winners of the Otto Group Product Classification Challenge were Gilberto Titericz & Stanislav Semenov. They competed as a team and used a three-level ensemble, as seen in Figure 16.4:

Figure 16.4: Challenge Winning Ensemble



We will provide only a high-level overview of their approach. You can find the full description at the following URL:

`https://goo.gl/fZrJA0`

The winning approach employed both the R and Python programming languages. Level 1 used a total of 33 different models. Each of these 33 models provided its output to three models in level 2. Additionally, the program generated eight calculated features. An engineered feature is one that is calculated based on the others. A simple example of an engineered feature might be body mass index (BMI), which is calculated based on an individual's height and weight. The BMI value provides insights that height and weight alone might not.

The second level combined the following three model types:

- XGBoost - Gradient boosting
- Lasange Neural Network - Deep learning
- ADABoost Extra Trees

These three used the output of 33 models and eight features as input. The output from these three models was the same nine-number probability vector previously discussed. It was as if each model were being used independently, thereby producing a nine-number vector that would have been suitable as an answer submission to Kaggle. The program averaged together these output vectors with the third layer, which was simply a weighting. As you can see, the winners of the challenge used a large and complex ensemble. Most of the winning solutions in Kaggle followed a similar pattern.

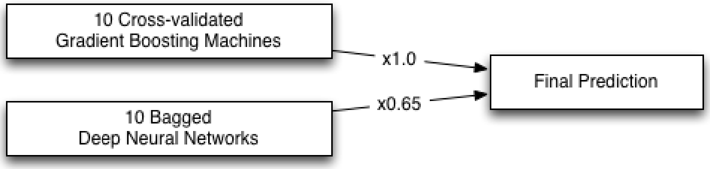
A complete discussion on exactly how they constructed this model is beyond the scope of this book. Quite honestly, such a discussion is also beyond our own current knowledge of ensemble learning. Although these complex ensembles are very effective for Kaggle, they are not always necessary for general data science purposes. These types of models are the blackest of black boxes. It is impossible to explain the reasons behind the model's predictions.

However, learning about these complex models is fascinating for research, and future volumes of this series will likely include more information about these structures.

16.0.3 Our Approach to the Challenge

So far, we've worked only with single model systems. These models that contain ensembles that are "built in", such as random forests and gradient boosting machines (GBM). However, it is possible to create higher-level ensembles of these models. We used a total of 20 models, which included ten deep neural networks and ten gradient boosting machines. Our deep neural network system provided one prediction, and the gradient boosting machines provided the other. The program blended these two predictions with a simple ratio. Then we normalized the resulting prediction vector so that the sum equaled 1.0 (100%). Figure 16.5 shows the ensemble model:

Figure 16.5: Our Challenge Group Entry



You can find our entry, written in Python, at the following URL:

`https://github.com/jeffheaton/kaggle-otto-group`

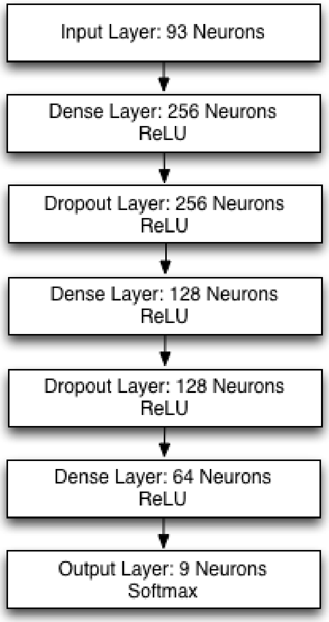
16.1 Modeling with Deep Learning

To stay within the scope of this book, we will present a solution to the Kaggle competition based on our entry. Because gradient boosting machines (GBM) are beyond the subject matter of this book, we will focus on using a deep neural network. To introduce ensemble learning, we will use bagging to combine ten trained neural networks together. Ensemble methods, such as bagging, will usually cause the aggregate of ten neural networks to score better than a single neuron. If you would like to use gradient boosting machines and replicate our solution, see the link provided above for the source code.

16.1.1 Neural Network Structure

For this neural network, we used a deep learning structure composed of dense layers and dropout layers. Because this structure was not an image network, we did not use convolutional layers or max-pool layers. These layer types required that input neurons in close proximity have some relevance to each other. However, the 93 input values that comprised the data set might not have been relevant. Figure 16.6 shows the structure of the deep neural network:

Figure 16.6: Deep Neural Network for the Challenge



As you can see, the input layer of the neural network had 93 neurons that corresponded to the 93 input columns in the data set. Three hidden layers had 256, 128 and 64 neurons each. Additionally, two dropout layers each had layers of 256 and 128 neurons and a dropout probability of 20%. The output was a softmax layer that classified the nine output groups. We normalized the input data to the neural network to take their z-scores.

Our strategy was to use two dropout layers tucked between three dense layers. We chose a power of 2 for the first dense layer. In this case we used 2 to the power of 8 (256). Then we divided by 2 to obtain each of the next two dense layers. This process resulted in 256, 128 and then 64. The pattern of using a power of 2 for the first layer and two more dense layers dividing by 2, worked well. As the experiments continued, we tried other powers of 2 in the first dense layer.

We trained the network with stochastic gradient descent (SGD). The program divided the training data into a validation set and a training set. The

SGD training used only the training data set, but it monitored the validation set's error. We trained until our validation set's error did not improve for 200 iterations. At this point, the training stopped, and the program selected the best-trained neural network over those 200 iterations. We refer to this process as early stopping, and it helps to prevent overfitting. When a neural network is no longer improving the score on the validation set, overfitting is likely occurring.

Running the neural network produces the following output:

```
Input      (None, 93)   produces      93 outputs
dense0     (None, 256)  produces     256 outputs
dropout0   (None, 256)  produces     256 outputs
dense1     (None, 128)  produces     128 outputs
dropout1   (None, 128)  produces     128 outputs
dense2     (None, 64)   produces      64 outputs
output     (None, 9) roduces       9 outputs
epoch     train loss    valid loss    train/val    valid acc
-------  ------------  ------------  -----------  -----------
      1       1.07019       0.71004      1.50723      0.73697
      2       0.78002       0.66415      1.17447      0.74626
      3       0.72560       0.64177      1.13061      0.75000
      4       0.70295       0.62789      1.11955      0.75353
      5       0.67780       0.61759      1.09750      0.75724
...
    410       0.40410       0.50785      0.79572      0.80963
    411       0.40876       0.50930      0.80260      0.80645
Early stopping.
Best valid loss was 0.495116 at epoch 211.
Wrote submission to file las-submit.csv.
Wrote submission to file las-val.csv.
Bagged LAS model: 1, score: 0.49511558950601003, current mlog:
   0.379456064667434, bagged mlog: 0.379456064667434
Early stopping.
Best valid loss was 0.502459 at epoch 221.
Wrote submission to file las-submit.csv.
Wrote submission to file las-val.csv.
Bagged LAS model: 2, score: 0.5024587499599558, current mlog:
   0.38050303230483773, bagged mlog: 0.3720715012362133
  epoch    train loss    valid loss    train/val    valid acc
-------  ------------  ------------  -----------  -----------
      1       1.07071       0.70542      1.51785      0.73658
      2       0.77458       0.66499      1.16479      0.74670
```

```
...
    370        0.41459       0.50696       0.81779       0.80760
    371        0.40849       0.50873       0.80296       0.80642
    372        0.41383       0.50855       0.81376       0.80787
Early stopping.
Best valid loss was 0.500154 at epoch 172.
Wrote submission to file las-submit.csv.
Wrote submission to file las-val.csv.
Bagged LAS model: 3, score: 0.5001535314594113, current mlog:
   0.3872396776865103, bagged mlog: 0.3721509601621992
...
Bagged LAS model: 4, score: 0.4984386022067697, current mlog:
   0.39710688423724777, bagged mlog: 0.37481605169768967
...
```

In general, the neural network gradually decreases its training and validation error. If you run this example, you might see different output, based on the programming language from which the example originates. The above output is from Python and the Lasange/NoLearn frameworks.

It is important to understand why there is a validation error and a training error. Most neural network training algorithms will separate the training set into a training and validation set. This split might be 80% for training and 20% for validation. The neural network will use the 80% to train, and then it reports that error as the training error. You can also use the validation set to generate an error, which is the validation error. Because it represents the error on the data that are not trained with the neural network, the validation error is the most important measure. As the neural network trains, the training error will continue to drop even if the neural network is overfitting. However, once the validation error stops dropping, the neural network is probably beginning to overfit.

16.1.2 Bagging Multiple Neural Networks

Bagging is a simple yet effective method to ensemble multiple models together. The example program for this chapter trains ten neural networks independently. Each neural network will produce its own set of nine probabilities that correspond to the nine classes provided by Kaggle. Bagging simply takes the average of each of these nine Kaggle-provided classes. Listing 16.1 provides the pseudocode to perform the bagging:

Listing 16.1: Bagging Neural Network

```
# Final results is a matrix with rows = to rows in training set
# Columns = number of outcomes (1 for regression, or class count
    for classification)
final_results = [][]
for i from 1 to 5:
   network = train_neural_network()
   results = evaluate_network(network)
   final_results = final_results + results

# Take the average
final_weights = weights / 5
```

We performed the bagging on the test data set provided by Kaggle. Although the test provided the 93 columns, it did not tell us the classes that it supplied. We had to produce a file that contained the ID of the item for which we were answering and then the nine probabilities. On each row, the probabilities should sum to 1.0 (100%). If we submitted a file that did not sum to 1.0, Kaggle would have scaled our values so that they did sum to 1.0.

To see the effects of bagging, we submitted two test files to Kaggle. The first test file was the first neural network that we trained. The second test file was the bagged average of all ten. The results were as follows:

- Best Single Network: 0.3794
- Five Bagged Networks: 0.3717

As you can see, the bagged networks achieved a better score than a single neural network. The complete results are shown here:

```
Bagged LAS model: 1, score: 0.4951, current mlog: 0.3794, bagged
    mlog: 0.3794
Bagged LAS model: 2, score: 0.5024, current mlog: 0.3805, bagged
    mlog: 0.3720
Bagged LAS model: 3, score: 0.5001, current mlog: 0.3872, bagged
    mlog: 0.3721
Bagged LAS model: 4, score: 0.4984, current mlog: 0.3971, bagged
    mlog: 0.3748
```

```
Bagged LAS model: 5, score: 0.4979, current mlog: 0.3869, bagged
    mlog: 0.3717
```

As you can see, the first neural network had a multi-class log loss (mlog) error of 0.3794. The mlog measure was discussed in Chapter 5, “Training & Evaluation.” The bagged score was the same because we had only one network. The amazing part happens when we bagged the second network to the first. The current scores of the first two networks were 0.3794 and 0.3804. However, when we bagged them together, we had 0.3720, which was lower than both networks. Averaging the weights of these two networks produced a new network that was better than both. Ultimately, we settled on a bagged score of 0.3717, which was better than any of the previous single network (current) scores.

16.2 Chapter Summary

In the final chapter of this book, we showed how to apply deep learning to a real-world problem. We trained a deep neural network to produce a submission file for the Kaggle Otto Group Product Classification Challenge. We used dense and dropout layers to create this neural network.

We can utilize ensembles to combine several models into one. Usually, the resulting ensemble model will achieve better scores than the individual ensemble methods. We also examined how to bag ten neural networks together and generate a Kaggle submission CSV.

After analyzing neural networks and deep learning in this final chapter as well as the previous chapters, we hope that you have learned new and useful information. If you have any comments about this volume, we would love to hear from you. In the future, we plan to create additional editions of the volumes to include more technologies. Therefore, we would be interested in discovering your preferences on the technologies that you would like us to explore in future editions. You can contact us through the following website:

`http://www.jeffheaton.com`

Appendix A

Examples

- Downloading Examples
- Structure of Example Download
- Keeping Updated

A.1 Artificial Intelligence for Humans

These examples are part of a series of books that is currently under development. Check the website to see which volumes have been completed and are available:

`http://www.heatonresearch.com/aifh`

The following volumes are planned for this series:

- Volume 0: Introduction to the Math of AI
- Volume 1: Fundamental Algorithms
- Volume 2: Nature-Inspired Algorithms
- Volume 3: Deep Learning and Neural Networks

A.2 Latest Versions

In this appendix, we describe how to obtain the *Artificial Intelligence for Humans* (AIFH) book series examples.

This area is probably the most dynamic of the book. Computer languages are always changing and adding new versions. We will update the examples as it becomes necessary, fixing bugs and making corrections. As a result, make sure that you are always using the latest version of the book examples.

Because this area is so dynamic, this file may become outdated. You can always find the latest version at the following location:

`https://github.com/jeffheaton/aifh`

A.3 Obtaining the Examples

We provide the book's examples in many programming languages. Core example packs exist for Java, C#, C/C++, Python, and R for most volumes. Volume 3, as of publication, includes Java, C#, and Python. Other languages, such as R and C/C++ are planned. We may have added other languages since publication. The community may have added other languages as well. You can find all examples at the GitHub repository:

`https://github.com/jeffheaton/aifh`

You have your choice of two different ways to download the examples.

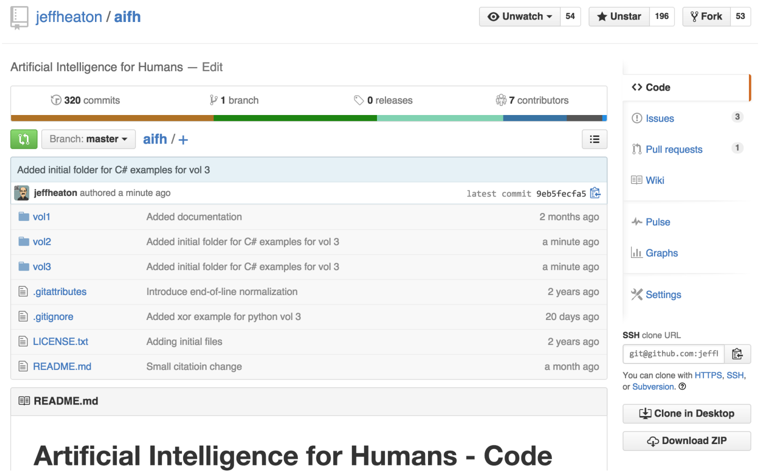
A.3.1 Download ZIP File

GitHub provides an icon that allows you to download a ZIP file that contains all of the example code for the series. A single ZIP file has all of the examples for the series. As a result, we frequently update the contents of this ZIP. If you are starting a new volume, it is important that you verify that you have the latest copy. You can perform the download from the following URL:

`https://github.com/jeffheaton/aifh`

You can see the download link in Figure A.1:

Figure A.1: GitHub



Artificial Intelligence for Humans — Edit

320 commits · 1 branch · 0 releases · 7 contributors

Added initial folder for C# examples for vol 3

jeffheaton authored a minute ago — latest commit 9eb5fecfa5

vol1	Added documentation	2 months ago
vol2	Added initial folder for C# examples for vol 3	a minute ago
vol3	Added initial folder for C# examples for vol 3	a minute ago
.gitattributes	Introduce end-of-line normalization	2 years ago
.gitignore	Added xor example for python vol 3	20 days ago
LICENSE.txt	Adding initial files	2 years ago
README.md	Small citatioin change	a month ago

SSH clone URL

git@github.com:jeffl

You can clone with HTTPS, SSH, or Subversion.

README.md

Artificial Intelligence for Humans - Code

A.3.2 Clone the Git Repository

You can obtain all the examples with the source control program **git** if it is installed on your system. The following command clones the examples to your computer: (Cloning simply refers to the process of copying the example files.)

```
git clone https://github.com/jeffheaton/aifh.git
```

You can also pull the latest updates with the following command:

```
git pull
```

If you would like an introduction to **git**, refer to the following URL:

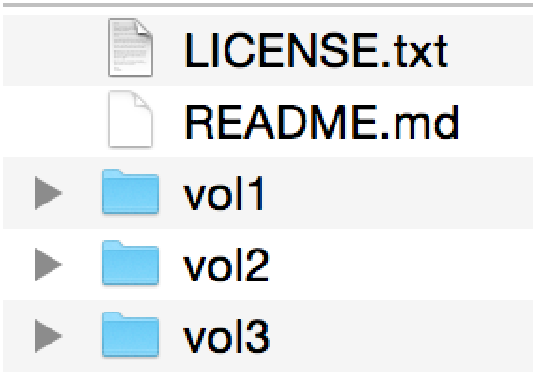
`http://git-scm.com/docs/gittutorial`

A.4 Example Contents

The entire *Artificial Intelligence for Humans* series is contained in one download that is a zip file.

Once you open the examples file, you will see the contents in Figure A.2:

Figure A.2: Examples Download



LICENSE.txt
README.md
vol1
vol2
vol3

The license file describes the license for the book examples. All of the examples for this series are released under the Apache v2.0 license, a free and open-source software (FOSS) license. In other words, we do retain a copyright to the files. However, you can freely reuse these files in both commercial and non-commercial projects without further permission.

Although the book source code is provided free, the book text is not provided free. These books are commercial products that we sell through a variety of channels. Consequently, you may not redistribute the actual books. This restriction includes the PDF, MOBI, EPUB and any other format of the book. However, we provide all books in DRM-free form. We appreciate your support of this policy because it contributes to the future growth of these books.

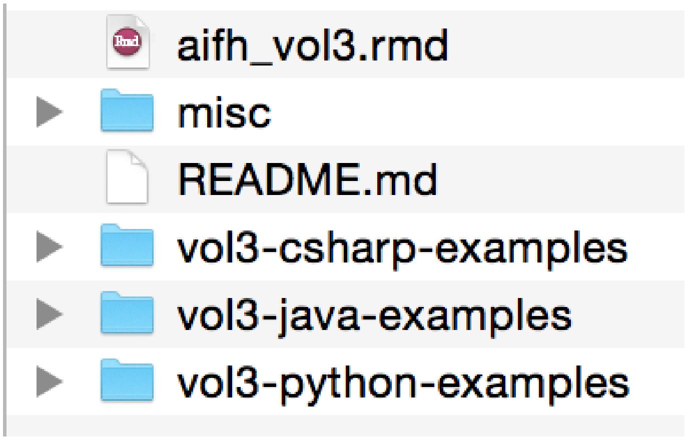
The download also includes a README file. The README.md is a "markdown" file that contains images and formatting. This file can be read either as a standard text file or in a markdown viewer. The GitHub browser automatically formats MD files. For more information on MD files, refer to the following URL:

`https://help.github.com/articles/github-flavored-markdown`

You will find a README file in many folders of the book's examples. The README file in the examples root (seen above) has information about the book series.

You will also notice the individual volume folders in the download. These are named vol1, vol2, vol3, etc. You may not see all of the volumes in the download because they have not yet been written. All of the volumes have the same format. For example, if you open Volume 3, you will see the contents listed in Figure A.3. Other volumes will have a similar layout, depending on the languages that are added.

Figure A.3: Inside Volume 3 (other volumes have same structure)



aifh_vol3.rmd
misc
README.md
vol3-csharp-examples
vol3-java-examples
vol3-python-examples

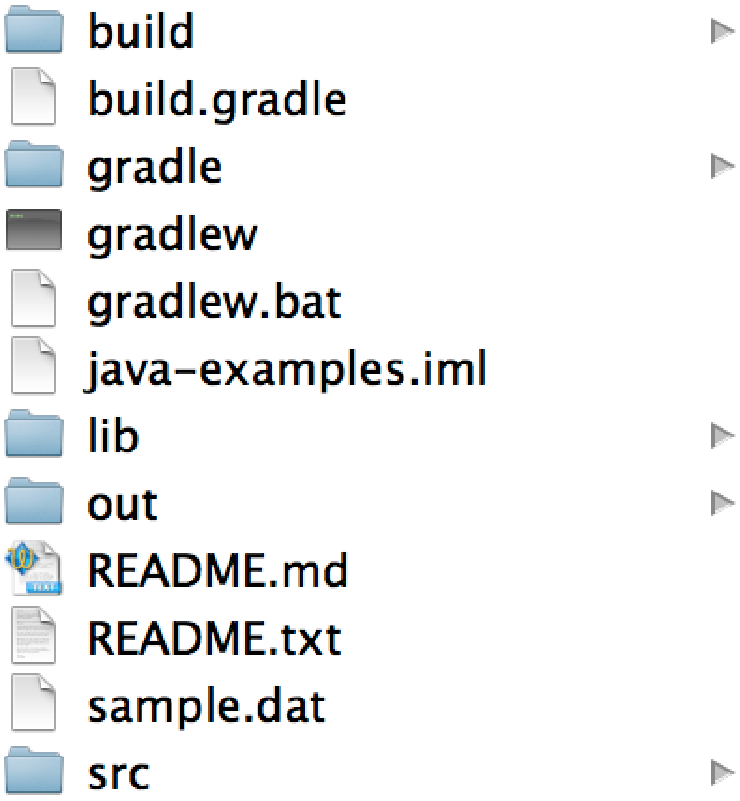
Again, you see the README file that contains information unique to this particular volume. The most important information in the volume level README files is the current status of the examples. The community often contributes example packs. As a result, some of the example packs may not be complete. The README for the volume will let you know this important information. The volume README also contains the FAQ for a volume.

You should also see a file named “aifh_vol3.RMD”. This file contains the R markdown source code that we used to create many charts in the book. We produced nearly all the graphs and charts in the book with the R programming language. The file ultimately allows you to see the equations behind the pictures. Nevertheless, we do not translate this file to other programming languages. We utilize R simply for the production of the book. If we used another language, like Python, to produce some of the charts, you would see a “charts.py” along with the R code.

Additionally, the volume currently has examples for C#, Java, and Python. However, you may see that we add other languages. So, always check the README file for the latest information on language translations.

Figure A.4 shows the contents of a typical language pack:

Figure A.4: The Java Language Pack



build
build.gradle
gradle
gradlew
gradlew.bat
java-examples.iml
lib
out
README.md
README.txt
sample.dat
src

Pay attention to the README files. The README files in a language folder are important because you will find information about the Java examples. If you have difficulty using the book's examples with a particular language, the README file should be your first step to solving the problem. The other files in the above image are all unique to Java. The README file describes these files in much greater detail.

A.5 Contributing to the Project

If you would like to translate the examples to a new language or if you have found an error in the book, you can help. Fork the project and push a commit revision to GitHub. We will credit you among the growing number of contributors.

The process begins with a fork. You create an account on GitHub and fork the AIFH project. This step creates a new project that has a copy of the AIFH files. You will then clone your new project through GitHub. Once you make your changes, you submit a "pull request." When we receive this request, we will evaluate your changes/additions and merge it with the main project.

You can find a more detailed article on contributing through GitHub at this URL:

`https://help.github.com/articles/fork-a-repo`

References

This section lists the reference materials for this book.

Ackley, H., Hinton, E., & Sejnowski, J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, 147-169.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G. Bengio, Y. (2010, June). Theano: a CPU and GPU math expression compiler. In *Proceedings of the python for scientific computing conference (SciPy).* (Oral Presentation)

Broomhead, D., & Lowe, D. (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2, 321-355.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. CoRR, abs/1412.3555.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14 (2), 179-211.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36, 193-202.

Garey, M. R., & Johnson, D. S. (1990). *Computers and intractability; a guide to the theory of np-completeness.* New York, NY, USA: W. H. Freeman & Co.

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In G. J. Gordon, D. B. Dunson, & M. Dudk (Eds.), *Aistats* (Vol. 15, p. 315-323). JMLR.org.

Hebb, D. (2002). *The organization of behavior: a neuropsychological theory.* Mahwah N.J.: L. Erlbaum Associates.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580 .

Hopfield, J. J. (1988). Neurocomputing: Foundations of research. In J. A. Anderson & E. Rosenfeld (Eds.), (pp. 457-464). Cambridge, MA, USA: MIT Press.

Hopfield, J. J., & Tank, D. W. (1985). "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52, 141-152.

Hornik, K. (1991, March). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4 (2), 251-257.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. Neural Networks, 1 (4), 295-307.

Jacobs, R., & Jordan, M. (1993, Mar). Learning piecewise control strategies in a modular neural network architecture. IEEE Transactions on Systems, Man and Cybernetics, 23 (2), 337-345.

Jordan, M. I. (1986). *Serial order: A parallel distributed processing approach* (Tech. Rep. No. ICS Report 8604). Institute for Cognitive Science, University of California, San Diego.

Kalman, B., & Kwasny, S. (1992, Jun). Why TANH: choosing a sigmoidal function. In Neural networks, 1992. *IJCNN, International Joint Conference on Neural Networks* (Vol. 4, p. 578-581 vol.4).

Kamiyama, N., Iijima, N., Taguchi, A., Mitsui, H., Yoshida, Y., & Sone, M. (1992, Nov). Tuning of learning rate and momentum on back-propagation. In *Singapore ICCS/ISITA '92. 'Communications on the move'* (p. 528-532, vol.2).

Keogh, E., Chu, S., Hart, D., & Pazzani, M. (1993). Segmenting time series: A survey and novel approach. In an edited volume, *data mining in time series databases.* Published by World Scientific Publishing Company (pp. 1-22).

Kohonen, T. (1988). Neurocomputing: Foundations of research. In J. A. Anderson & E. Rosenfeld (Eds.), (pp. 509-521). Cambridge, MA, USA: MIT Press.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (n.d.). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (p. 2012).

LeCun, Y., Bottou, L., Bengio, Y., & Haner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE* (pp.2278-2324).

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In International conference on machine learning (*ICML*).

van der Maaten, L., & Hinton, G. (n.d.). Visualizing high-dimensional data using t-SNE. *Journal of Machine Learning Research (JMLR)*, 9, 2579-2605.

Marquardt, D. (1963). An algorithm for least-squares estimation of nonlinear parameters. *SIAM Journal on Applied Mathematics*, 11 (2), 431-441.

Matviykiv, O., & Faitas, O. (2012). Data classification of spectrum analysis using neural network. *Lviv Polytechnic National University.*

McCulloch, W., & Pitts, W. (1943, December 21). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5 (4), 115-133.

Mozer, M. C. (1995). Backpropagation. In Y. Chauvin & D. E. Rumelhart (Eds.), (pp. 137{169). Hillsdale, NJ, USA: L. Erlbaum Associates Inc.

Nesterov, Y. (2004). *Introductory lectures on convex optimization: a basic course.* Kluwer Academic Publishers.

Ng, A. Y. (2004). Feature selection, l1 vs. l2 regularization, and rotational invariance. *In Proceedings of the twenty first international conference on machine learning* (pp. 78-). New York, NY, USA: ACM.

Neal, R. M. (1992, July). Connectionist learning of belief networks. *Artificial Intelligence*, 56 (1), 71-113.

Riedmiller, M., & Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *IEEE international conference on neural networks* (pp. 586-591).

Robinson, A. J., & Fallside, F. (1987). *The utility driven dynamic error propagation network* (Tech. Rep. No. CUED/F-INFENG/TR.1). Cambridge: Cambridge University Engineering Department.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Neurocomputing: Foundations of research. In J. A. Anderson & E. Rosenfeld (Eds.), (pp.696-699). Cambridge, MA, USA: MIT Press.

Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *Proceedings of the 2012 IEEE conference on computer vision and pattern recognition (cvpr)* (pp. 3642-3649). Washington, DC, USA: IEEE Computer Society.

Sjberg, J., Zhang, Q., Ljung, L., Benveniste, A., Deylon, B., yves Glorennec, P., Juditsky, A. (1995). Nonlinear black-box modeling in system identification: a unified overview. *Automatica*, 31, 1691-1724.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. Burges, L. Bottou, & K. Weinberger (Eds.), *Advances in neural information processing systems 25* (pp. 2951{2959). Curran Associates, Inc.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10 (2), 99-127.

Stanley, K. O., DAmbrosio, D. B., & Gauci, J. (2009, April). A hypercubebased encoding for evolving large-scale neural networks. *Artificial Life*, 15 (2), 185-212.

Teh, Y. W., & Hinton, G. E. (2000). Rate-coded restricted Boltzmann machines for face recognition. In T. K. Leen, T. G. Dietterich, & V. Tresp (Eds.), *Nips* (p. 908-914). MIT Press.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1.

Zeiler, M. D., Ranzato, M., Monga, R., Mao, M. Z., Yang, K., Le, Q. V., Hinton, G. E. (2013). On rectified linear units for speech processing. In *ICASSP* (p. 3517-3521). IEEE.

Index