Data Mining with Neural Networks

Solving Business Problems from Application Development to Decision Support

Joseph P. Bigus

McGraw-Hill
New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

McGraw-Hill

A Division of The McGraw-Hill Companies

Library of Congress Cataloging-in-Publication Data

Bigus, Joseph P.
Data mining with neural networks : solving business problems—from application development to decision support / by Joseph P. Bigus.
p. cm.
Includes bibliographical references and index.
ISBN 0-07-005779-6 (pbk.)
1. Neural networks (Computer science) 2. Database management.
3. Knowledge acquisition (Expert systems) I. Title
QA76.87.B55 1996
658.4'038'028563—dc20 96-33779
CIP

pbk 2 3 4 5 6 7 8 9 0 DOC/DOC 9 0 0 9 8 7

ISBN 0-07-005779-6

The sponsoring editor of this book was Jennifer Holt Di Giovanna, the editor of this book was Sally Anne Glover, and the executive editor was Robert E. Ostrander. The director of production was Katherine G. Brown. This book was set in ITC Century Light. It was composed in Blue Ridge Summit, Pa.

Printed and bound by R. R. Donnelley and Sons Company, Crawfordsville, Indiana.

McGraw-Hill books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please write to the Director of Special Sales, McGraw-Hill, 11 West 19th Street, New York, NY 10011. Or contact your local bookstore.

To Jennifer, my wife and best friend.

Contents

Acknowledgments

I would like to extend my thanks to the many people who have helped me with this book. First, I would be remiss not to thank my children, Sarah and Alex, for their patience when I monopolized our computer while working on the book. They gave up many opportunities to play computer games so that I could meet my schedule. My wife, Jennifer, put up with my working strange hours (stranger than usual, I should say) while juggling my job at IBM with the writing chores. I truly could not have written this without her help and support. I'd like to thank my manager, E.J. Limvere, for his understanding while I simultaneously completed this book, built a new house, and participated in a worldwide software design and development effort.

All of the graphic illustrations in this book were created by Gordy Hall. I would like to thank Gordy for his fine work and his patience as we turned my ideas into illustrations. If a picture is worth a thousand words, then Gordy saved me plenty of writing.

This book has benefited from the comments and suggestions of Jennifer Bigus, Cindy Hitchcock, E.J. Limvere, Jeff Pilgrim, Don Schlosnagle, Sally Glover, and Alex Berson. I thank them for the time spent reading through my drafts and for sharing their opinions with me.

I would like to thank my editor at McGraw-Hill, Jennifer Holt DiGiovanna, for following up after our first meeting at the IBM Technical Interchange in New Orleans. Her interest in my idea for "a book about neural network applications" prompted me to submit my proposal and resulted in this book being written.

Over the years,. many people at IBM have helped develop the Neural Network Utility product. These include Shawn Austvold, Steve Lund, John Henckel, Paul Hospers, Larry McMains, Lisa Barouh, Karl Schultz, Scott Peterson, Helen Fung, Matt Latcham, Cindy Hitchcock, Jeff Pilgrim, Todd Strand, Ed Seifts and Don Schlosnagle. I would like to thank these people for the contributions they made, and continue to make, to the NNU technology.

Introduction

In my position at IBM, I regularly brief executives, managers, and computer professionals on data mining and neural network technology. In my briefings, I cover the fundamentals of data mining and neural networks, and I also discuss specific applications relevant to the customers' businesses. Since time is usually limited, my goal is to quickly give them a basic understanding of data mining and to spark their imaginations so they can visualize how the technology can be used in their own enterprises. When I succeed, it is satisfying to see their excitement as they "ponder the possibilities." In the question-and-answer period following my presentations, I am invariably asked for a recommendation on a "good book on neural networks" so they can learn more. With few exceptions, these people do not want to know how neural networks work; they want to know how neural networks can be applied to solve business problems, using terminology they can understand and real-world examples to which they can relate.

While there are many neural network books available today, most focus on the inner workings of the technology. These texts approach neural networks from either a cognitive science or an engineering perspective, with a corresponding emphasis either on philosophical arguments or on a detailed treatment of the complex mathematics underlying the various neural network models. Other neural network books discuss academic applications, which have little or no relation to real business problems, and are full of C or C++ source code showing nitty-gritty implementation details. None of these titles would fit my definition of a "good book on neural networks" that is appropriate for a business-oriented audience.

This book, however, is targeted directly at executives, managers, and computer professionals by explaining data mining and neural networks from a business information systems and management perspective. It presents data mining with neural networks as a strategic business technology, with the focus on the practical, competitive advantages they offer. In addition,

the book provides a general methodology for neural network data mining and application development using a set of realistic business problems as examples. The examples are developed using a commercially available neural network data mining tool, the IBM Neural Network Utility.

Data mining, the idea of extracting valuable information from data, is not new. What is new is the wholesale computerization of business transactions and the consequential flood of business data. What is new is the distributed computer processing and storage technologies, which allow gigabytes and terabytes of data to remain online, available for processing by client/server applications. What is new are neural networks and the development of advanced algorithms for knowledge discovery. When combined, these new capabilities offer the promise of a lifesaver to businesses drowning in a sea of their own data.

Neural networks are a computing technology whose fundamental purpose is to recognize patterns in data. Based on a computing model similar to the underlying structure of the human brain, neural networks share the brain's ability to learn or adapt in response to external inputs. When exposed to a stream of training data, neural networks can discover previously unknown relationships and learn complex nonlinear mappings in the data. Neural networks provide some fundamental, new capabilities for processing business data. However, tapping these new neural network data mining functions requires a completely different application development process from traditional programming. So even though the commercial use of neural network technology has surged in the past 10 years, constructing successful neural network applications is still considered a "black art" by many in the software development community. As will be shown, this is an unfortunate and inaccurate perception of the state of the art of neural network application development.

This book presents a comprehensive view of all the major issues related to data mining and the practical application of neural networks to solving real-world business problems. Beginning with an introduction to data mining and neural network technologies from a business orientation, the book continues with an examination of the data mining process and ends with application examples. Appendices describe related technologies such as fuzzy logic and genetic algorithms.

The data mining process starts with data preparation issues, including data selection, cleansing, and preprocessing. Next, neural network model and architecture selection is discussed, with the focus on the problems the various models can solve, not the mathematical details of how they solve them. Then the neural network training and testing process is described, followed by a discussion of the use of data mining for decision support and application development. Automated data mining through the use of intelligent agents is also presented.

The application case studies deal with common business problems. The specific examples are chosen from a broad range of industries in order to be relevant to most readers. The data mining functions of classification, clustering, modeling, and time-series forecasting are illustrated in the examples.

When you finish reading this book, you will know what data mining is, what problems neural networks can solve today, how to determine if a problem is appropriate for a neural network solution, how to set up the problem for solution, and finally how to solve it. In short, I will try to illuminate the "black art" of developing neural network applications and place it in a context with other application development technologies such as object-oriented computing and incremental development and prototyping techniques.

For business executives, managers, or computer professionals, this book provides a thorough introduction to neural network technology and the issues related to its application without getting bogged down in complex math or needless details. The reader will be able to identify common business problems that are amenable to the neural network approach and will be sensitized to the issues that can affect successful completion of such applications.

The book uses clear, nontechnical language and a case-study approach to explore the issues involved in using neural networks to solve business problems. This text can be used profitably by someone trying to use neural networks to implement application solutions using commercial neural network tools, or by managers trying to understand how neural networks can be applied to their businesses. Each chapter includes a summary at the end, along with a reference list for further reading.

Part 1 spans eight chapters, including introductory chapters 1 and 2, and then provides a comprehensive methodology and overview of the key issues in data mining with neural networks for decision support and application development.

Chapter 1 describes the business and information technology trends that are contributing to the requirements for data mining applications. Key developments include the corporate data warehouse and the distributed computing models. The major steps in the data mining process are detailed, and a data mining architecture is presented. Data mining as enhanced decision support and as application development are examined. A catalog of example data mining applications in specific industries is described.

In chapter 2, neural networks are introduced as a fundamentally new computing and problem-solving paradigm for approaching data mining applications. Neural computing is presented as an alternative path on the evolution of intelligent computing, a path that was dominated by symbolic artificial intelligence. The key factors responsible for the initial rejection and the recent reemergence of neural networks are discussed.

Chapter 2 also describes the paradigm shift required for problem solving with neural networks as opposed to traditional computer programming. An example of a knowledge worker is used to compare and contrast the use of neural networks for similar tasks. Next, the neural processing element and the mechanism for adaptive behavior is discussed. Then I focus on the ba-

sic neural network computing functions: classification, clustering, modeling, and time-series forecasting.

Chapter 3 discusses the data preparation step, beginning with an overview of the current state of the art in database management systems. Next, I highlight the importance of data selection and representation to the neural network application development process. Data representation schemes for numeric and symbolic variables using real numbers and coded data types are covered. Data preprocessing operations—including symbolic mapping, taxonomics, and scaling or thresholding of numeric values—are described. Common techniques for data set management, including the quantity and quality of the data, are discussed.

Chapter 4 presents a survey of the basic neural network learning paradigms, including supervised, unsupervised, and reinforcement learning. The major neural network topologies are discussed. Next, the most popular types of neural network models and their capabilities are described. The focus is on the functional differences between neural network models, not on their mathematical derivations. The chapter ends with a discussion of the key issues in selecting a neural network model for a particular problem.

Chapter 5 walks the reader though a typical neural network development process. First I highlight the importance of selecting an appropriate error measure to indicate when the network training is complete. Next, I describe the most important training parameters used to control the training time and quality of the trained neural network. The iterative neural network development process is examined, and throughout I give a feel for the "normal" evolution and how to detect "abnormal" problems.

Chapter 6 discusses the analysis of neural network models created through data mining. This process of "discovering what the neural network learned" is required for decision support applications. This chapter presents the most common techniques for visualization of neural networks and data mining results. Rule generation from neural networks and input sensitivity analysis are also described.

Chapter 7 describes the use of trained neural networks for the deployment of operational applications. I discuss data pre- and postprocessing requirements at run time, how neural networks can be treated as simple subroutines, and how neural network prediction accuracy can be monitored. Application maintenance issues are also addressed.

Chapter 8 deals with the topic of intelligent agents and how the data mining techniques and neural networks can be used to add learning to intelligent agents. As computer systems become more complex, users are increasingly looking to advanced software to ease their burdens. Intelligent agents can automate both user and system management tasks through a combination of pattern recognition and domain knowledge.

Part 2 gives four detailed examples of how neural networks can be applied to solving business problems. Each application follows the data mining methodology used in Part 1, including a discussion of how the specific example given can be generalized to solve other similar business problems. A comprehensive list of application references is provided. Each chapter in Part 2 can stand alone; no order is implied or suggested.

Chapter 9 combines customer database and sales transaction data to define target markets. This application uses neural networks to segment the customers by creating clusters based on similarities of the customer attributes. This information can then be used to target promotions at members of the group who have the attributes in which we are interested. This chapter includes a discussion of the analysis of clusters or segments in data mining applications.

Chapter 10 uses market data on properties and selling prices to build a price estimator for real estate appraisals. This is a classic example of using neural networks for data mining. It is a simple modeling application, with multiple input variables and one output (market price or cost). Any business that must make proposals can use its past experience with similar projects to make fast, accurate estimates.

Chapter 11 mines customer profile information to rank customers or suppliers. This application uses the information a business has available on its current and past customers to build a neural network model that ranks them in order of "goodness" (i.e., profitability). Prospective new customers can be targeted or selected using their expected profitability.

Chapter 12 uses an inventory and sales transaction database to build a replenishment system. This time-series forecasting application deals with data that changes over time. The idea is to use past history to predict future behavior. Issues unique to forecasting are discussed in depth.

Appendix A presents an overview of the IBM Neural Network Utility products and their capabilities. The focus is on features of the product that support the data mining and neural network application methodology presented in Part 1.

Appendix B is an introduction to fuzzy sets and fuzzy logic. Often used in conjunction with neural networks, fuzzy logic, through fuzzy expert systems, provides an excellent way to add domain knowledge to data mining operations.

Appendix C describes evolutionary programming and genetic algorithms. Like neural networks, genetic algorithms are biologically inspired. They use a metaphor for the process of natural selection to perform parallel searches. Genetic algorithms are used to find optimal neural network architectures and to adjust connection weights.

The glossary provides a list of the most common terms used in data mining and neural network application development. The annotated bibliography contains a resource list of neural network reference books and business-oriented application papers and articles, with brief descriptions of their contents.

Part

1

The Data Mining Process Using Neural Networks

Part 1 presents a methodology for data mining with neural networks. Structured around the major steps of data preparation, data mining, and analysis of the mining results, the eight chapters in this section highlight the issues specific to neural network algorithms. The introduction mentioned the "black art" label often used to refer to the neural network development process. While perhaps not strictly cookbook in approach, a careful reading of this material will considerably enhance your chances of successfully training your neural network. For those familiar with traditional data analysis and model building, as well as those used to object-oriented development, this process will seem comfortably familiar. The emphasis is on the key steps and practical considerations, not on the theoretical issues involved.

Part 1 begins with an introduction to data mining and neural networks. Then the discussion turns to the many aspects of data preparation, the first step required for data mining, regardless of the data mining algorithm used. Of specific importance to neural networks is the representation of data, so the common representations and data types used are discussed. The key aspects that differentiate neural networks—training paradigm, topology, and learning algorithms—are covered in detail. I describe the training process, starting first with the definition of "success" and then

describing the most important learning parameters used to control that process. After neural network training, I explore methods for discovering what the neural network learned. These techniques include visualization, rule generation, sensitivity analysis, and model predictions. Then I discuss how to deploy and maintain neural network applications. Part I ends with a look at intelligent agent technology. In a somewhat symbiotic relationship, intelligent agents can control the mining of data, while data mining can be used to add learning capabilities to intelligent agents.

Chapter

1

Introduction to Data Mining

"Information networks straddle the world. Nothing remains concealed. But the sheer volume of information dissolves the information. We are unable to take it all in."
GUNTHER GRASS (1990)

In this chapter, I discuss the business environment and information technology trends that have made data mining both necessary and achievable. I provide a formal definition for data mining and describe the major steps in the data mining process. Finally, I present a list of the many data mining applications that have been developed using neural network technology.

Data Mining: A Modern Business Requirement

Being a business manager or computing professional today is anything but dull. As wave after wave of new information technology hits the market and slowly gets assimilated into daily operations, the risks (and rewards) grow higher for those who have to place their bets on the technology roulette wheel. Get it right, and you might gain several points of market share at your competitor's expense. Get it wrong or do nothing, and you might have to spend years trying to recover lost ground. As the old Chinese proverb says, "May you live in interesting times." Well, information technology workers have certainly hit the jackpot.

Over the past three decades, the use of computer technology has evolved from the piecemeal automation of certain business operations, such as accounting and billing, into today's integrated computing environments, which

offer end-to-end automation of all major business processes. Not only has the computer technology changed. How that technology is viewed and how it is used in a business has changed. From the new hardware configurations using local and wide area networks for distributed client/server computing to the software emphasis on object-oriented programming, these changes support one overriding business requirement—process more data, faster, in ever more complex ways.

In 1981, the IBM PC was introduced. Costing just $3000, it used a 16-bit Intel 8088 processor, 64 kilobytes (KB) of RAM, and a single 5.25" floppy drive. The first hard drive available was a Seagate 5.25" Winchester hard drive, which stored a whopping 5 megabytes (MB) of data. In late 1995, $3000 will buy a PC with an Intel Pentium processor, 16 MB of RAM, and a 1-gigabyte (GB) hard drive. In just 15 years, the amount of disk storage available in a $3000 PC has increased 200 times.

In 1988, the IBM AS/400 midrange systems were announced with up to 96 MB of main memory and a maximum hard disk capacity of 38 GB using 400-MB drives. In 1995, the AS/400 Advanced System supports 1.5 GB of main memory and up to 260 GB of disk storage using 2-GB drives. In 1996, the IBM DB2 MultiSystem for AS/400 will support databases up to 16 terabytes spread across 32 AS/400 systems. From paper tape to punch cards, to magnetic drum, to the relentless advance in direct access storage devices (DASD using IBM terminology, otherwise known as hard disk drives), the increases in both data storage capabilities and device reliability have been phenomenal. Figure 1.1 shows the recent explosion in the amount of information stored on mainframe computer systems, a supposedly dying breed, from the 1990 through 1995, and projected through 1998.

I have met with IBM customers who are gathering gigabytes of data daily. They are literally unable to store all of their data online and have to put it to tape for backup storage. This is like being a grain farmer with a bumper crop who has to let it rot in the field because he doesn't have storage. Just like a crop in the field, business information decreases in value as it ages, and the cost of planting the crop (gathering the data) has already been paid.

Increasingly, business data is seen as a valuable commodity in its own right, not just as a by-product of processing the day's transactions. Today's operational data represents the current state of your business. When it is combined with historical business data, it can tell you where you are going and where you are. By taking operational data and dumping it to tape, you might be protecting the data, but you are neglecting it as well. With business decisions being made at a breakneck pace, managers and executives need information on which to base those decisions. And that information needs to be online.

But just being online isn't enough. The old query and reporting tools have long since lost their ability to keep up with these information needs. New client/server software that allows free-form queries has helped. But query

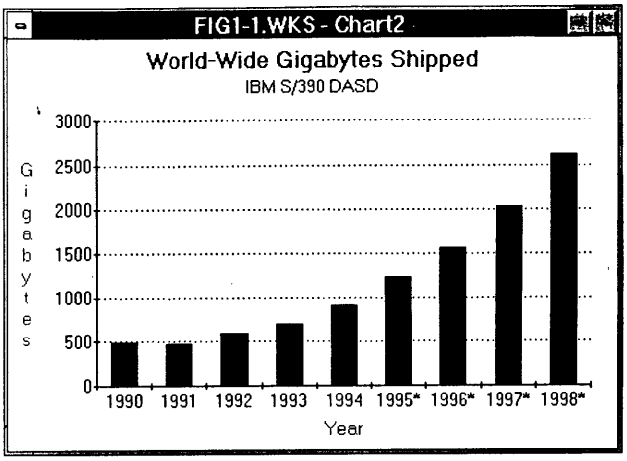


Figure 1.1 Growth in mainframe data storage. *(Source: International Data Corp.)*

tools only help if you know what you are looking for. Multidimensional databases, which provide three-dimensional views of data, and online analytical processing (OLAP) tools are certainly enhancing business data analysis capabilities. However, even they do not suffice in today's competitive environment.

What about all the information buried in your customer transaction files? Maybe there's a trend that says your customers are switching to a different product or configuration, and your inventory is going to be widely out of balance if you don't react. Maybe there is a string of transactions that are totally out of character for that customer (has she lost her credit card?). Maybe the crucial requirements for your next product in development are hidden in the past purchases made by customers in the target market. Wouldn't it be nice to find that information? After all, wasn't that part of the business case for computerizing business operations in the first place? Wouldn't it be nice to really cash in on that investment in computing technology?

If you've asked these questions, then you are not alone. Increasingly, people want to leverage their investments in business data, to use it as an aid in decision making, and to turn it into operational applications. Data mining promises to do just that. More than just complex queries, data mining provides the means to discover information in raw business data. In many industries, it has become a business imperative. If you are not mining your data for all it is worth, you are guilty of underuse of one of your company's greatest assets. Because in that data is information about your customers

and the products they buy. As you will see, the old business maxim of "know your customer" is attainable today using data mining techniques.

The Evolution of Information Technology

The evolution in business computing over the past 30 years has been dramatic. It is often difficult to determine which came first, the change to the flatter business organization or the new distributed computing capabilities. While the raw processing power and storage capabilities of computers has expanded at an astonishing pace, the business community has used that additional computing power to improve efficiency in their operations and to enhance their competitiveness in the markets they serve.

The computing styles have matched the organizations. The first computers were large mainframes that centralized a business organization's data and computer data processing tasks. In many ways this matched the hierarchical command and control management used by large corporations. The management information systems (MIS) staff in their glass house (the raised floors and air-conditioned computer rooms required by the mainframes) controlled access to and processing of all corporate data.

The development of minicomputers or departmental computers was somewhat an extension of the mainframe paradigm and somewhat a precursor of the future. These computers allowed groups of people working on common tasks to control some of their computing environment, though usually with the guidance and blessing of the central MIS organization. Rather than wait for a new report or application to be developed by the MIS organization, the department would hire its own programmers or software engineers to solve its own computing problems.

In the early 1980s, the development of the personal computer completely changed the dynamics of business computing, though it was some time before the central IS organization and the business management realized this. Now an individual could purchase a PC and applications software and work at his or her own desk to solve daily problems. The development of spreadsheets and word processors gave the business justification for these purchases. Over time, the environment evolved from a sprinkling of PCs or workstations in the organization, to the point where nearly every knowledge worker has a PC on his or her desk. While this evolution from centralized to distributed computing has dramatically changed how and where data processing is performed in an organization, perhaps the biggest impact is on how business data is created and managed.

The Data Warehouse

Looking at this computing evolution from a business data perspective raises some interesting issues. In the host-centric computing model, the corporate data was stored on the central computer. This allowed the MIS organization to manage the valuable corporate information, to safeguard it from theft or damage, and to collect new business data as it was created through business transactions. Of course, one downside to this central control to data was that knowledge workers who needed to access the data often had to wait a long time for the IS organization to respond to their needs. Writing new COBOL or RPG programs to generate reports takes time (and programmers).

As department-level systems were introduced, the work group could exercise more control of the data. Often this was off-loaded from the mainframe and was sometimes out of date, but this sometimes stale data was a small price to pay for relatively easy access to vital information. Conversely, if the departmental computer was used to process transactions and create new business data, then the information had to be moved up to the corporate data repository. While this caused some problems for central MIS, they were usually more than happy to get the application backlog down by pushing that work on the departmental users. There was also some discomfort due to the distribution of data, but again, that was a price they had to pay to meet the business needs.

When large numbers of stand-alone PCs were brought into offices, real problems began to surface in the management of key business data. Now the crack financial analyst crunching numbers on his or her PC spreadsheet had key business data that was totally out of the purview of the central MIS organization. Who would back up the data? Who would ensure the security of that information?

As the PCs were connected to the corporate computer network, some of these problems were solved. The knowledge worker could download key information from the departmental or mainframe computers, process the data with PC-based applications, and then return the data to the corporate coffers. Today, the remote administration of PCs by IS has brought this problem full circle, back to the days when all crucial business data was under control—well, not completely.

One problem with the proliferation of computers throughout the business is the large number of databases scattered across systems. As the databases in departmental and PC systems grew, the data was not always passed up to the centralized system. Over time, information about customers, suppliers, product design, and manufacturing operations was stored in separate databases. While moving the data under centralized control is desirable because of operational reality, many of these databases remain where they were originally deployed. They are needed to run the business. However, if strategic new applications are ever going to be developed, this disparate set of data needs to be consolidated under one (figurative) roof. This leads us to one of the most sweeping ideas to hit the database management arena since relational databases—the *data warehouse*.

A data warehouse, as the name implies, is a data store for a large amount of corporate data. The data quality and integrity can be maintained by a

centralized staff. Applications developers do not have to deal with layouts of multiple incompatible and sometimes overlapping databases. In short, when they need to access corporate data, they know where to find it—in the data warehouse. While the idea of centralized data management is not new, how we got to this point is a combination of the history of the evolution of information technology and the tremendous growth in computing storage capability. In a large corporate data warehouse, we are not talking in terms of hundreds of megabytes of data (which can now be stored on a single PC) but in the hundreds of trillions of bytes of data (terabytes). Indeed, the idea is not so much that the data resides physically on a single computer system, but that all of the data is stored and is accessible through a network of distributed systems so that it presents itself as a seamless collection of corporate data.

Figure 1.2 depicts a typical configuration of a corporate data warehouse. Operational data is generated through transactions processed by applications

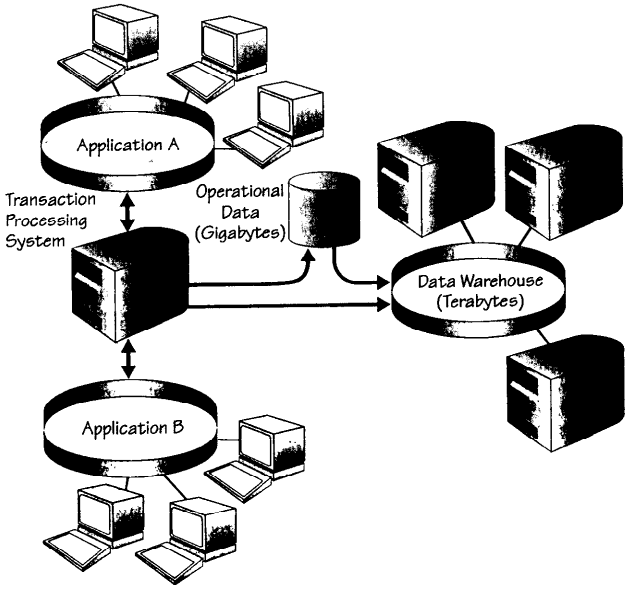


Figure 1.2 Data warehouse architecture diagram.

running on PCs and servers and is then stored in operational databases. These operational databases usually hold several months of data and range from 10 to 50 gigabytes in size. At certain intervals the operational data is moved off the transaction processing systems onto the data warehousing system. Products such as IBM's DataPropagator and Visual Warehouse can automate these data replication tasks. The warehouse might be a network of PC file servers, a midrange computer like an IBM AS/400, an IBM mainframe, or some heterogeneous mix of computer systems. The data warehouse might hold years of data and can swell to terabytes of data.

In addition to the aforementioned benefits for data quality and security, a data warehouse opens new possibilities in terms of executive information systems, decision support systems, and building line-of-business operational applications. With computers, as with people, you can't make good decisions unless you have all of the available data. A good corporate data warehouse makes that data readily available. In addition, it makes possible a whole new class of computing applications, now known as data mining.

Data Mining Overview

Data mining, also referred to as knowledge discovery (Frawley, Piatetsky-Shapiro, and Matheus 1992), has become something of a buzzword in business circles. Everyone wants it, and therefore many computer hardware and software vendors claim that they have it. The only problem, of course, is that not everyone agrees on what it is. To some, it is client/server queries. To others, it is multidimensional databases. To still others, it is OLAP with drill-down capabilities. Seemingly the only points of agreement are that it has to do with database systems and that it is important. In this section, I explore my view of exactly what data mining is, and more importantly, how it is done. First, let's start with a definition: *Data mining is the efficient discovery of valuable, nonobvious information from a large collection of data.*

This innocuous sentence identifies some key attributes that can be used to determine what is and is not "data mining." The operative word in this definition deals with the "discovery" of information from data. Notice that I am not talking about complex queries where the user already has a suspicion about a relationship in the data and wants to pull all of the information together to manually check or validate a hypothesis. Nor are we talking about performing statistical tests of hypotheses using standard statistical techniques. Data mining centers on the automated discovery of new facts and relationships in data. The idea is that the raw material is the business data, and the data mining algorithm is the excavator, sifting through the vast quantities of raw data looking for the valuable nuggets of business information.

A data mining operation is "efficient" if the value of the extracted information exceeds the cost of processing the raw data. When viewed from this

perspective, data mining efficiency is a return on investment statement, not a processing time statement. To some people, a data mining algorithm is efficient only if it completes in under three minutes and supports interactive analysis of the results. However, few people would argue against spending two weeks of processing time (at a cost of $100,000) if a key design or manufacturing process parameter is discovered and will save $1,000,000 in costs over the next two years. Efficiency is a cost-versus-benefit statement.

When we specify nonobviousness as a requirement for data mining, this is also a statement about the efficiency and value of the process. If you spend $10 on data mining only to find out something that was well known in your business, then you have just wasted $10. And while many data mining algorithms are used to process data in order to find relationships and patterns, they often produce voluminous outputs of trivial, obvious information. This information might make you feel better by confirming your own understanding of the business fundamentals in your industry, but it does not add value to your decision-making process. This separating the wheat from the chaff is the back-end analysis of the data mining output. It is every bit as important as the quantity and quality of the raw data, and of the data mining algorithms (the tools) with which you process the data. The information discovered through data mining is "valuable" only if it helps you gain a competitive advantage in your business, or aids in the decision-making process.

A "large collection of data" is certainly a subjective quantity. A small business might consider a gigabyte of data to be a large database worthy of mining. A large corporation might have multiple databases in the tens or hundreds of gigabytes range. To some extent, a database is large enough for data mining if it contains enough data so that the relationships are hidden from view and so that valuable, nonobvious information can be extracted.

The data mining process consists of three major steps, as illustrated in Figure 1.3. Of course, it all starts with a big pile of data. The first processing step is data preparation, often referred to as "scrubbing the data." Data is

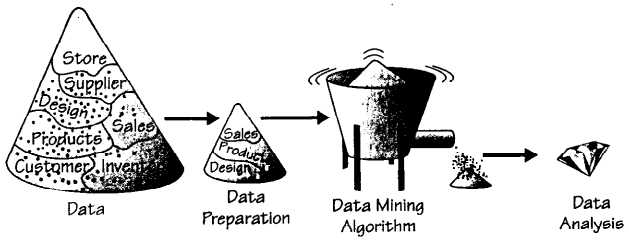


Figure 1.3 The data mining process.

selected, cleansed, and preprocessed under the guidance and knowledge of a domain expert. Second, a data mining algorithm is used to process the prepared data, compressing and transforming it to make it easy to identify any latent valuable nuggets of information. The third phase is the data analysis phase, where the data mining output is evaluated to see if additional domain knowledge was discovered and to determine the relative importance of the facts generated by the mining algorithms. This is where strategic business decisions are made using information gleaned by the data mining process and where operational applications are deployed.

As businesses have computerized their operations, they have gradually developed a collection of separate and sometimes incompatible systems. Either through mergers of distinct information technology departments or simply through the requirements for different applications, customer, sales transactions, inventory, and design information usually exist in more than one place in the corporate information systems. This duplication must be reduced or eliminated in order to perform effective data mining. This consolidation of crucial business data is now being referred to as the corporate "data warehouse." While not an absolute prerequisite to data mining, developing a comprehensive data warehouse is a practical prerequisite to developing a flexible decision support system. To use a mining metaphor, it's a lot easier to mine in an area with good roads and bridges than in the middle of a forest or mountaintop, where the mining tools would have to be airlifted in. Gaining access to the raw material is a part of the cost of the operation and therefore affects the efficiency (return on investment). Having the corporate data consolidated and readily available will make some data mining operations more practical from a cost standpoint than if the data had to be collected from scratch.

Unfortunately, just collecting the data in one place and making it easily available isn't enough. When operational data from transactions is loaded into the data warehouse, it often contains missing or inaccurate data. How good or bad the data is a function of the amount of input checking done in the application that generates the transaction. Unfortunately, many deployed applications are less than stellar when it comes to validating the inputs. To overcome this problem, the operational data must go through a "cleansing" process, which takes care of missing or out-of-range values. If this cleansing step is not done before the data is loaded into the data warehouse, it will have to be performed repeatedly whenever that data is used in a data mining operation.

For most data mining applications, the relatively clean data that resides in the corporate data warehouse must usually be refined and processed before it undergoes the data mining process. This preprocessing might involve joining information from multiple tables, selecting specific rows or records of data, and it most certainly includes selecting which columns or fields of data to look at in the data mining step. Often two or more fields are combined to represent

ratios or derived values. This data selection and manipulation process is usually performed by someone with a good deal of knowledge about the problem domain and the data related to the problem under study. Depending on the data mining algorithm involved, the data might need to be formatted in specific ways (such as scaling of numeric data) before it is processed. While viewed by some as a bothersome preliminary step (sort of like scraping the old paint away before applying a fresh coat of paint), data preparation is crucial to a successful data mining application. Indeed, IBM Consulting and independent consultants confirm estimates that data preparation might consume anywhere from 50% to 80% of the resources spent in a data mining operation.

The second step in data mining, once the data is collected and preprocessed, is when the data mining algorithms perform the actual sifting process. Many techniques have been used to perform the common data mining activities of associations, clustering, classification, modeling, sequential patterns, and time-series forecasting. These techniques range from statistics, to rough sets, to neural networks. See Table 1.1 for a list of the most common data mining functions, the corresponding data mining algorithms, and typical applications. Think of the different data mining algorithms as the drill bits of the mining machine. If the ore is locked in hard rock, then we might need a diamond drill (or algorithm of a certain type). If it is in more porous rock, then we might be able to increase our efficiency by using a less expensive drill bit (or algorithm). The type of data mining function we are trying to perform, along with the quality and quantity of data available combine to specify which data mining algorithm should be used.

TABLE 1.1 Data Mining Functions

Data mining function	Algorithm	Application examples
Associations	Statistics, set theory	Market basket analysis
Classification	Decision trees, neural networks	Target marketing, quality control, risk assessment
Clustering	Neural networks, statistics	Market segmentation, design reuse
Modeling	Linear and nonlinear regression, curve fitting, neural networks	Ranking/scoring customers, pricing models, process control
Times-series forecasting	Statistics ARMA models, Box-Jenkins, neural networks	Sales forecasting, interest rate prediction, inventory control
Sequential patterns	Statistics, set theory	Market basket analysis over time

The third and final step is the analysis of the data mining results or output. In some cases the output is in a form that makes it very easy to discern the valuable nuggets of information from the trivial or uninteresting facts. Figure 1.4 shows the output of a data mining run using the Quest associa-

[Auto Accessories] AND [Tires] ==> [Automotive Services]
(conf = 89.2%, sup = 1.2%

[Home Laundry Appliances] ==> [Maintenance Agreements]
(conf = 61%, sup = 1.0%)

When a customer buys "Auto Accessories" and "Tires" then the customer buys "Automotive Services" in 89.2% of the cases. This pattern is present in 1.2% of the transactions.

When a customer buys "Washer or Dryer" then the customer buys "Maintenance Agreements" in 61% of the cases. This pattern is present in 1.0% of the transactions.

Figure 1.4 Rule output from an association algorithm.

tion algorithm developed by IBM Almaden Research. The relationships between items in a market basket analysis are represented in if-then rule form. The antecedent (left-hand side) lists the items purchased and the association with the consequent (right-hand side) item in terms of confidence (how often the items are purchased at the same time) and support (the percentage of records in which the association appears). With the rules recast into textual form, the valuable information is much easier to identify.

In other cases, however, the results will have to be analyzed either visually or through another level of tools to classify the nuggets according to predicted value. Figure 1.5 illustrates a visualization of another market basket

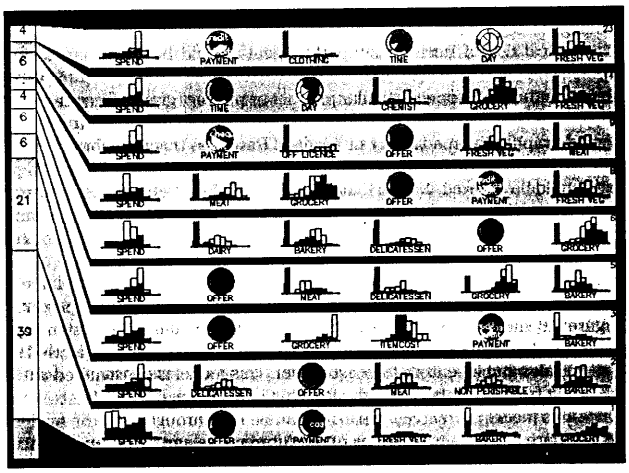


Figure 1.5 Visualization of a clustering algorithm.

analysis using segmentation performed by the IBM UK Scientific Center. The graphic illustrates the statistical profile of the customers in each major segment and how their attributes compare to the whole population of customers. Whatever data mining algorithm is used, the results will have to be presented to the user. A successful data mining application involves the transformation of raw data into a form that is more compact and understandable, and where relationships are explicitly defined.

I've talked about data mining as the process of extracting valuable information from data. Of course, what makes information valuable to a business is when that information leads to actions or market behavior that gives a discernible competitive advantage. There are two major ways for businesses to use the output of a data mining process. The first is to enhance strategic decision support processes. The second is to use the data mining models as part of operational applications. I discuss these two principal uses of data mining in the following sections.

Enhancing Decision Support

As the quantity of business data has grown, a new class of applications and data analysis tools has emerged, called either decision support systems (DSS), or executive information systems (EIS), depending on the software vendor. Whatever it is called, the tool's main thrust is to allow business decision makers to analyze and detect patterns in data, and to aid them in making strategic business decisions.

A typical use of decision support systems would be for a purchasing agent or buyer for a large retailer to create an interactive query for sales and inventory data for a particular product or product group. Once the data is retrieved, the decision support system would allow the data to be displayed graphically in a variety of formats. Based on this transformation of the raw data, the decision maker would decide what quantity of that product should be ordered. Notice that the "discovery" element of this picture is provided by the data analyst and her selection of which data to request and view. In most respects, this is a query application integrated with a graphical display system.

In contrast, a data mining solution to this problem would be to mine a database that contains sales and inventory information on the product. Figure 1.6 shows a typical scenario for using data mining for decision support. Starting with a selection of prepared data, a neural network is used to build a sales and inventory forecast model. This model is constructed automatically from the data, using the learning capabilities of neural networks. Once this model is created, "what-ifs" can be run through it by the analyst to get more accurate predictions of the future sales and inventory requirements, or to determine the sensitivity of sales to changes in any of the input variables. Importantly, when a computational model is generated from the raw data using a data mining algorithm, it also opens the way to complete automation of the process. This is discussed in the next section.

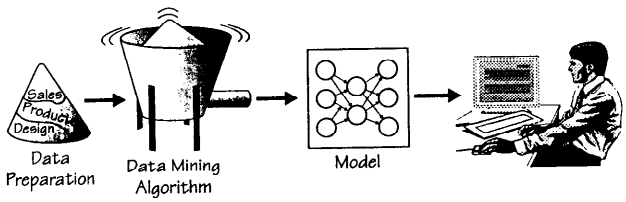


Figure 1.6 Data mining for decision support.

Developing Business Applications

For the past 20 years, the standard approach to developing business applications has been to have a system analyst determine the data that needs to be processed, and identify the major steps of the business process that operates on that data. Once characterized, the problem is broken down into subproblems, and algorithms are designed to operate on the data. This top-down approach works well for a wide variety of problems and has become the standard technique used by business programming shops worldwide.

Another, more modern approach is to perform an analysis of the problem in terms of the business objects that are involved in the process and the operations performed on or by those objects. This so-called object oriented analysis and design, combined with object-oriented programming languages such as Smalltalk and C++ (even OO-COBOL) is fast becoming the preferred application development technique for business. Major advantages are code reuse and improved reliability because new applications are developed using previously developed and tested objects. Although the focus is on objects and their behavior rather than on problem decomposition, object-oriented programming is still just another approach to writing algorithms for digital computers.

As mentioned earlier, a third alternative exists for developing business applications. This approach is based on data mining and the use of models built during the discovery process. Figure 1.7 shows how data mining can be used for automated application development. The prepared data is used to build a neural network model of the function to be performed. Transactions are then run through the neural network, and the outputs are used to make automated business decisions. This use of data mining places different requirements on the data mining algorithms, since the perspicuity is not so important. What is important in building business applications using data

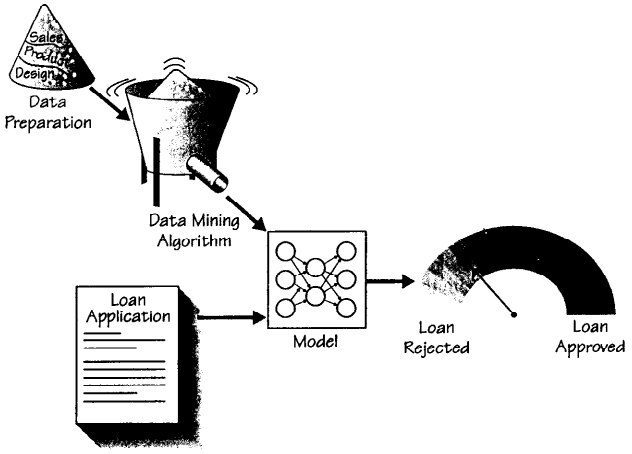


Figure 1.7 Data mining for application development.

mining is that the underlying processing functions—whether they be clustering, classification, modeling, or time-series forecasting—are accurate and reliable. Please note that applications built using a data mining algorithm do not have to necessarily perform the business processing function "better" than applications built through the traditional programmatical approach. Equivalent accuracy would still yield significant benefits because the application is generated automatically, as a by-product of the data mining process. However, there are many cases where traditional programmed applications cannot be developed, since no one in the business understands how the data relates well enough to design or write an algorithm to capture those relationships. It is here where the advantages of using data mining techniques such as neural networks are really compelling.

Example Data Mining Applications

So far I have talked about data mining from a technological perspective. Now let's change our view to a business perspective (Li 1994). How are businesses using data mining and neural networks today? What kind of applications have been successfully deployed? What industries are leading in the adoption of this technology?

Marketing

Every business has to market its products or services, and the cost of marketing efforts must be factored into the ultimate selling price. Any technology that can improve marketing results or lower marketing costs gets a close look by businesses. In conjunction, customer credit, billing, and purchases were some of the first business transactions to be automated with computers, so large amounts of data are readily available for mining. These factors have combined to make marketing one of the hottest application areas for data mining. This general area is referred to as database marketing (Gessaroli 1995).

Customer relationship management is the term most used for the overall process of exploiting customer-related information and using it to enhance the revenue flow from an existing customer. Information on customer demographics and their purchasing patterns are used to segment the customers based on their underlying similarities, whether socioeconomic or by their interests and hobbies as demonstrated by the products they purchase. By determining similar classes of customers, advertising return on investment can be enhanced since the marketing messages are accurately reaching those customers most likely to buy. By segmenting the customer base, different products and services can be developed, which are tuned to appeal to members of the specified group.

Database marketing—using data mining techniques against databases with marketing information—can be used in several different aspects of the customer/business relationship. This information can be used to improve the customer retention rate by identifying customers who are likely to switch to another provider. Since it costs much more to win a new customer than to sell to an existing one, this application can have a significant impact on profits. When knowledge about the customer is combined with product information, specific promotions can be run that increase the average purchases made by that customer segment. By knowing what a particular customer is interested in, marketing costs can be lowered through more effective mailings, and customer satisfaction is improved because they are not receiving what they perceive as "junk mail."

Direct mail response campaigns use data mining to first select a target set of customers. A test mailing is then made to a small subset of this set. Based on the size of the response and the characteristics of those who responded, a determination can be made as to who should be included in the subsequent mass mailing and which offer or offers should be included.

Retail

In the retail sector, perhaps the biggest application is market basket analysis. This involves mining the point of sale transactions to find associations

between products. This information is then used to determine product affinities and suggest promotion strategies that can maximize profits.

A typical scenario is to collect all point-of-sale transactions in a database. The transaction database is then mined to find those products that are most strongly associated. When customers purchase baby diapers, they also tend to purchase baby formula. Thus a retailer would not ordinarily put both diapers and formula on sale at the same time. Rather, using knowledge that there is an association between these two products, the retailer would put one on sale and place the other item next to or in close proximity to the first item. The placement of items in grocery stores is no accident. The output of a market-basket analysis would identify association between products that had never been suspected. One anecdotal tale is of a convenience store chain that noticed that there was a strong association between purchasers of baby diapers and beer. Apparently, when Dad went out to pick up diapers, he often picked up a sixpack along the way.

A related application is the use of sequential patterns to spot temporal trends or buying behavior. This is another application that is based on associations between items or products, only now the focus is on their temporal relationship. An example is when someone purchases a new suit, then with a high likelihood they will return to purchase new dress shirts and ties. A retailer would then use this information to try to encourage the purchase of these related items in a single trip to the store because they might shop elsewhere to pick up the accessories.

Finance

Data mining is in widespread use in the finance industry (Disney 1995). Neural networks are used to detect patterns of potential fraudulent transactions in consumer credit cards (Norton 1995). They are also used to predict interest rate and exchange rate fluctuations in currency markets. Several brokerage houses use neural networks to help in managing stock and bond portfolios (Schwartz 1994). Neural networks have been used for credit risk assessments and for bankruptcy prediction in commercial lending and bond rating.

In the finance industry, different classes of customers are treated differently, based on the perceived risk to the lender (Margarita and Beltratti 1992). Thus, classifying the amount of risk associated with a customer or with a particular transaction is extremely important. A modest improvement in the ability to detect impending bankruptcies, for example, will yield an appreciable revenue increase to the financial institution (Udo 1993).

In the commodities trading arena, a complex set of variables is used to construct trading strategies (Grudnitski and Osburn 1993). While the efficient market theory has been widely accepted for years, many brokerage houses still rely on technical traders to analyze the data and make educated guesses about the markets. The most important ability is to detect trends or changes in movement of the market as a whole or of some particular segment or stock (Komo, Chang, and Ko 1994). This is also true in the currency exchange area. Neural networks' abilities to model time-series and complex nonlinear functions has prompted their use in all of these application areas.

Manufacturing

The complexity in modern manufacturing environments and the requirements for both efficiency and high quality has prompted the use of data mining in several areas. Neural networks are used in computer aided design to match part requirements to existing parts for design reuse, in job scheduling and manufacturing control, in optimization of chemical processes, and to minimize energy consumption. Neural networks are also widely used in quality control and automated inspection applications.

Job shop scheduling is a difficult problem that deals with assigning the sequence of jobs and how work is assigned to specific machines in a manufacturing plant. There are usually many constraints that absolutely must be met, such as the sequence of process steps and whether a particular material can be processed by a specific machine. In addition to these hard constraints, there are soft ones such as optimizing operating efficiencies by avoiding needless setup and reconfiguration of machines by scheduling similar types of products or operations on the same piece of equipment. Neural networks have been used to satisfy these constraints while generating optimized job assignments.

Manufacturing process control deals with the automated adjustment of parameters that control the quality and quantity of products produced by the manufacturing facility. A well-known technique called statistical process control is used to track the quality of the goods produced by a manufacturer by measuring the variability and tolerances in various aspects of the finished goods. Using examples of good and bad parts, neural networks have been used as aids in the control of processes and the detection of subtle flaws in the plant outputs.

In some chemical manufacturing processes, complex mixtures of chemicals must be heated, cooled, mixed, and transported by an automatic control system. Dangerous situations or abnormal operating conditions must be detected and compensated for automatically, or the system could explode. Neural networks have been used to minimize the generation of waste products and to improve the properties of the material produced, such as steel from blast furnaces.

Automated inspection is a requirement in many manufacturing environments where high-speed and high-output quantities can overwhelm the abilities of human inspectors to accurately and reliably spot defects in work in process. Using digital images, neural networks have been used to detect

faults in rolled steel and aluminum, in printed circuit boards, and in consumer product packaging. They have been used to classify the grades of fruit, and to sort products as they come off of an integrated assembly line.

Health and medical

There are two primary uses for data mining in the Health industry: the administration of patient services, billing, insurance, etc., and the diagnosis and treatment of disease.

The health industry is using data mining to detect fraudulent insurance claims from both unscrupulous patients and health care providers. A common approach is to develop a model for a "normal" pattern of activity and then detect and scrutinize "abnormal" behavior. Both clustering and modeling functions are used for this. Another major application is to identify the most cost-effective health care providers. Many aspects of the health industry are under tight government control, and compliance with government regulations must be maintained.

Data mining is also being used to automate the diagnosis of cervical cancer, breast cancer, and heart attacks (Sabbatini 1992). Patient data can be collected on a large population and presented to a neural network. Thus a data mining system can look at more patients in one day than a human doctor could see in a lifetime. Neural networks' abilities to synthesize a large body of data and to detect subtle patterns have proven to be effective (Harrison, Chee, and Kennedy 1994).

Energy and utility

Suppliers of electrical power are subject to large swings in demand for service. A single weather front moving through a region can considerably increase demand for power in a matter of hours. Decisions have to be made as to whether plants should be brought online or taken down for preventive maintenance. Large consumers of electrical power such as manufacturing plants are often charged based on their peak energy usage, so it is in their interest to manage their consumption to minimize excessive demands for service. This major dependency on accurate load forecasts has made the utility industry one of the major users of neural networks (Park et. al. 1990).

Another application in the energy industry is the search for new gas or oil deposits. Neural networks have been successfully used to aid in analysis of soundings taken at test drilling sites for detecting changes in the strata of rock and to identify likely sites for mineral deposits.

Summary

The changes in the business computing environment over the past three decades have been dramatic. Computer processing and storage technology

advances provide businesses with the ability to keep hundreds of gigabytes or even terabytes of data online. However, this is a good news, bad news story. The good news is that now we can have years of historical business data available for decision support applications. The bad news is that traditional data query and analysis methods are not capable of dealing with that much data. The consequence is that businesses are drowning in data.

Data mining or knowledge discovery offers a solution to this problem. With the emphasis on the discovery of valuable information from large databases, data mining provides added value to the investment in the corporate data warehouse. The data mining process consists of three basic steps: data preparation, information discovery by a data mining algorithm, and analysis of the mining algorithm output.

The benefits of data mining are evident in two major business activities, decision support and application development. In decision support systems, data mining transforms the data to reveal hidden information in the form of facts, rules, and graphical representations of the data. The extremely large amounts of data are compressed to reveal the inner relationships among the data elements. When used in the application development cycle, data mining with neural networks provides automated construction of transaction processing systems and forecasting models.

Applications of data mining span all industries. Businesses of all types use data mining to target marketing messages to specific customer sets, both to satisfy their customers' needs and to increase revenues. Retailers use data mining to find associations between products purchased at the same time and to forecast sales and corresponding inventory requirements. The finance industry uses data mining techniques to manage risks and to detect trends in the markets. Manufacturers use neural networks in the design, production scheduling, process control, and quality inspections of their products. Hospitals and insurance companies mine their data to detect fraudulent claims by health care providers and patients, and physicians use advanced pattern recognition capabilities of neural networks to automate laboratory tests. Utilities use neural networks to forecast demand and quickly respond to equipment outages and changes in the weather.

Any business with data about its customers, suppliers, products, and sales can benefit from data mining. When businesses are looking for the slightest edge over their competition, they are willing to travel far and wide and spend millions of dollars to buy information about their markets. Often this information is sitting right in their offices, hidden away in their data warehouses.

References

Anand, T. 1995. Opportunity explorer: navigating large databases using knowledge discovery templates, Journal of Intelligent Information Systems 4, pp. 27–37.

Bahrami, A., M. Lynch, C.H. Dagli. 1995. Intelligent design retrieval and packaging system: application of neural networks in design and manufacturing, International Journal Prod. Res. (UK) Vol. 33, No.2, Feb., pp. 405–26.

Disney, D.R. 1995. Comment: for the real gold in customer data, dig deep, *The American Banker*, May 10, 1995.
Frawley, W.J., G. Piatetsky-Shapiro, and C.J. Matheus. 1992. Knowledge discovery in databases: an overview, *AI Magazine*, Fall, pp. 57–69.
Gessaroli, J. 1995. Data mining: A powerful technology for database marketing, Telemarketing Vol 13, No. 11, May, pp. 64-68.
Grudnitski, G., and L. Osburn. 1993. Forecasting S&P and gold futures prices: An application of neural networks, Journal of Futures Markets, Vol. 13, No. 6, Se., pp. 631–643.
Harrison, R.F., P.L. Chee, and R.L. Kennedy. 1994. Autonomously learning neural networks for clinical decision support, Proceedings of the International Conference on Neural Networks and Expert Systems in medicine and Healthcare, 1994, pp. 15–22.
Komo, D. , C. Chang, and H. Ko. 1994. Stock market index prediction using neural networks, Proceedings of SPIE, Vol. 2243, pp. 516–26.
Li, E.Y. 1994. Artificial neural networks and their business applications, Information and Management, Vol. 27, No. 5, Nov., pp. 303–313.
Margarita, S., and A. Beltratti. 1992. Credit risk and lending in an artificial adaptive banking system, Adaptive Intelligent Systems - Proceedings of the BANKAI Workshop, 1992, pp. 161–76.
Norton M. 1994. Close to the Nerve (credit card fraud), *Banking Technology* (UK) Vol. 11, No. 10, Dec., pp. 28–31.
Park, dc, M.A. El-Sharkawi, R.J. Marks, L.E. Atlas, and M.J. Damborg. 1991. Electric load forecasting using an artificial neural network, in IEEE Transactions on Power Systems, Vol. 6, No. 2, May 1991.
Reggia, J.A. 1993. Neural computation in medicine, Artificial Intelligence in Medicine, Vol 5. No. 2, Apr., pp. 143–157.
Sabbatini, R.M.E. 1992. Applications of connectionist systems in biomedicine, MEDINFO 92, K.C. Lun et. al. (editors) , pp. 418–25.
Schwartz T. 1995. A tale of two traders, *Wall Street and Technology*, Jan. 1995, Miller Freeman, Inc., pp. 42.
Tafti, M.H.A., and E. Nikbakht. 1993. Neural networks and expert systems: new horizons in business finance applications, Information Management & Computer Security, Vol. 1, No. 1, pp. 22–28.
Udo, G. 1993. Neural network performance on the bankruptcy classification problem, Computers and Industrial Engineering, Vol. 25, No.1-4, pp. 377–380.
Wong, B.K., T.A. Bodnovich, and Y. Selvi. 1995. A bibliography of neural network business applications research: 1988-September 1994, Expert Systems (August), Vol. 12, No. 3, pp. 253–262.

Chapter

2

Introduction to Neural Networks

"Man is still the most extraordinary computer of all."
JOHN F. KENNEDY

"We want to replace the computer metaphor with the brain metaphor."
DAVID RUMELHART

Neural Networks: A Data Mining Engine

Neural networks are one of the key technologies used for data mining. In this chapter, I explore the history of neural networks, how they compare to traditional computing approaches, and why they are a natural technology for performing data mining activities.

A Historical Perspective

The history of computing is filled with crucial twists and turns. From the first visions of Charles Babbage and his mechanical computing device, the Difference Engine, to John von Neumann and the development of the modern digital computer, many potential paths and technologies were examined and then rejected as political and market forces made their natural selection. Figure 2.1 shows some of the major milestones in computing, from both a computer hardware and software view. In 1937, Alan Turing developed his theory of the Turing machine, a device that could read instructions from a paper tape and simulate any other computing machine.

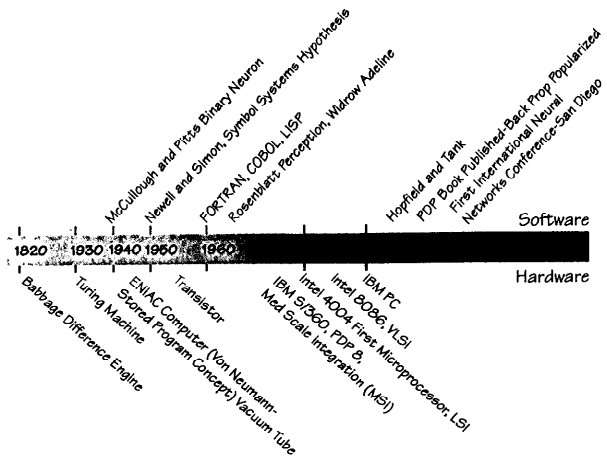


Figure 2.1 Computing evolution time line.

When McCullough and Pitts (1943) wrote their paper on the binary neuron, they were using the human brain as their computational model. John von Neumann picked up these ideas and developed them, along with others, into the computing model we know today, the stored-program, serial, "von Neumann" computer. It is somewhat ironic that what began as a crude model of the brain has, over time, become the accepted metaphor for how the brain actually works.

For a brief period, analog computers competed with digital computers. The analog computers mapped very well to advanced mathematics and calculus and could be used for modeling natural phenomena. However, they could not be used for accurate mathematical computations such as business accounting and inventory management. Here the digital computer proved superior. And as more and more problems were mapped to the digital realm, digital computers became the dominant type of computers. Today, the term "computer" is synonymous with "digital."

A similar story occurred with the development of intelligent computers. In the late 1950s and early 1960s, there were two major schools of thought. One school wanted to model computation on the basic architecture and key attributes of what was known about the human brain. The other school felt that intelligence could be produced in machines through the use of symbol manipulation. The two approaches were tightly coupled to the prevailing

philosophical positions regarding the fundamental nature of intelligence and led to a major debate in the intelligent computing arena.

Intelligent Computing: Symbol Manipulation or Pattern Recognition?

Why has the digital computer become the common metaphor for how the human brain works? Why is the logical, sequential processing of the electronic computer used as the model of the organized mind? Are computers accurate models of the biological brain? The answers to these questions depend on your definition of intelligence.

What separates humans from the lower life forms? For years, great thinkers have claimed that humans manipulate symbols, and that this ability found in the human cerebellum is the unique machinery that gives us intelligence. Not just intelligence enough for survival, but intelligence that allows planning, engineering, and feats of architecture on a grand scale.

When Newell and Simon proposed their physical symbol system hypothesis in 1955, the digital computer was only ten years old (Jubak 1992). They realized that while digital computers were extremely good and fast number crunchers, they could also be extremely fast symbol processors. All that was needed was a simple abstraction mapping symbols to numbers. Their claim was that a "physical symbol system has the necessary and sufficient means for general intelligent action." Not only were they saying that symbol manipulations could lead to intelligent behavior, they stated that it was "necessary." If people exhibit intelligent behavior, then it must be because they are using formal rules to manipulate symbols. This assumption of the logical equivalence between symbol processing computers and the human brain became the basis for most of the artificial intelligence work in the next three decades.

What these scientists seemed to overlook was that our symbol processing forebrain processes information that has already been processed at a subsymbolic level by the body senses. Our hearing, vision, taste, and tactile input provide the human brain with a wealth of information with which to reason about the world. Some people call this subsymbolic processing *feature detection*. And what is feature detection? It is a process of pattern recognition that occurs largely at a subconscious level. People develop many context-sensitive models of what to expect as we interact with the world. Even though we might be thinking about something else, our built-in novelty detectors break through our thoughts and tell us when something out of the ordinary or unexpected is happening. For example, when driving a car, the subconscious often takes over the routine tasks, and our mind is free to wander until we "notice" something unexpected in the traffic.

Figure 2.2 illustrates the major differences in approaches between the symbolic and the subsymbolic (or neural network) school of artificial intelligence. Those espousing the symbolic view would say that knowledge must

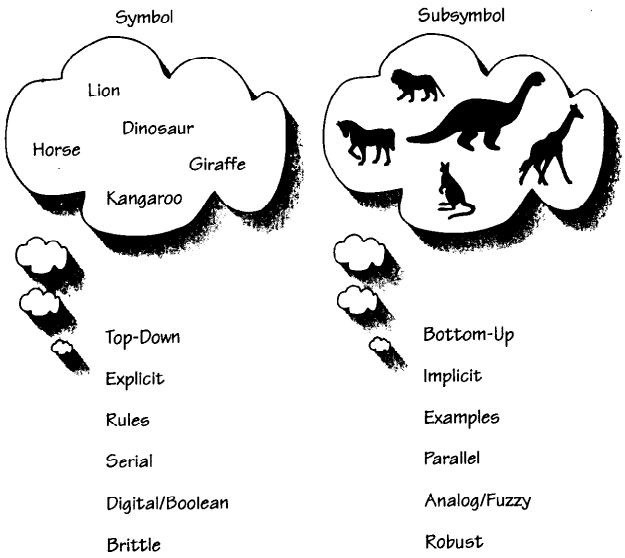


Figure 2.2 Symbol processing and subsymbolic processing.

be explicitly represented by rules, and that the flow of consciousness is best described by a serial process. Those with a subsymbolic or connectionist slant would say that massive parallelism and analog computation is a fundamental aspect of intelligence. This leads to the robust qualities of neural networks in contrast to the well-known "brittleness" associated with Boolean logic rule-based systems when they operate near the edges of their domain knowledge. In some sense, the different technical approaches mirror differences in the philosophy of mind. In one, intelligence is purely a function of the higher-order processes derived in a top-down manner. In the other, intelligence is an emergent phenomena, springing forth from the interactions of many simple processors from the bottom up. Of course, the real answer probably lies somewhere in the middle of these extremes.

Computer Metaphor Versus the Brain Metaphor

In the early 1960s, the symbol processing school recognized that digital computers were good at manipulating numbers, and that numbers could be used to represent symbols. The use of this simple abstraction from symbols to numbers meant that without any changes to the common digital computer architecture, we had a symbol processing computer. The symbol processing researchers demonstrated some significant early successes, such as computer programs that could do college-level calculus and mathematical theorem proving. Solving such seemingly difficult problems suggested that symbol processing was surely the way to go if we wanted to develop intelligent computer systems.

During the same time, the brain-based or connectionist researchers were trying to show that connected networks of simple processing units could demonstrate emergent intelligent behavior. Rosenblatt's Perceptron model (1962) and Widrow's Adeline (1960) were two examples of the types of neural networks built during this time. The Perceptron was purported to be a model of the mind, and the capabilities were subjected to hype now reserved for certain 32-bit operating systems. The Adaline was shown to be quite capable of solving hard engineering problems in an area that became known as adaptive filtering. However, while the neural network researchers were able to show some interesting results, they soon hit a brick wall due to seemingly insurmountable theoretical problems. Their learning algorithms could only work for single-layer neural networks, which limited them to solving only simple, linearly-separable problems. Minsky and Pappert, two researchers from the symbol processing school who were very knowledgeable about neural networks, highlighted these theoretical limitations in their critical book, *Perceptrons* (1969). Neural networks research all but ended in the United States by the late 1960s, although some work continued in Europe.

Meanwhile, the symbol processing school of artificial intelligence proceeded full speed ahead. Special-purpose programming languages such as Prolog and Lisp were developed for writing symbol processing programs. Indeed, specialized computers known as AI workstations were developed to provide the high-powered processing needed to simulate intelligence through symbol processing. Researchers moved on from calculus and theorem proving to problems such as image recognition, speech recognition, planning, and scheduling. Rule-based expert systems were developed to simulate the problem-solving techniques used by human experts. Funding and graduate students poured into symbol-based artificial intelligence research year after year. It was not until the mid-1980s that people realized that progress was not being made as fast as was promised.

Symbolic AI was always "just around the corner" from crossing that magical threshold into mainstream applications. All that was needed was more powerful computers, more funding, more time. Not that the research was fruitless. New computer interface techniques such as graphical user interfaces were refined on the AI workstations. New programming paradigms—such as the object-oriented language, Smalltalk—were invented. Expert

systems went commercial and solved some difficult real-world problems. However, the deep results leading to truly intelligent machines did not seem to be coming any closer even after decades of research. This lack of progress on some of the fundamental problems in developing intelligent software systems led researchers to reexamine the work from the 1960s on neural networks and to rediscover the work of a small group of researchers who carried on, even after neural networks "lost" to symbolic AI.

In the mid-1980s, researchers started publishing new results and updating old results on fundamental neural network problems. In the early 1960s when researchers were first working on computers that could learn, both the available computer hardware and the theoretical understanding of the issues were not up to the task. In the intervening years, researchers had developed new neural network training algorithms that overcame the limitations of the early Perceptron and Adaline models. The PDP research group (where PDP means *parallel distributed processing*) had been working for several years and published their two-volume manifesto in 1986 (Rumelhart and McClelland). These books served to entice many young graduate students to pick up the connectionist or neural network cause. The PDP books popularized the backward propagation of errors algorithm, a learning algorithm that allowed multiple-layer neural networks to be constructed. Other articles in the book covered self-organizing and competitive behavior, recurrent neural networks, and applications of neural networks to optimization and cognitive modeling. The First International Conference on Neural Networks, held in 1987, served as the formal kickoff for much of the current research on neural networks. These theoretical advances, along with the availability of relatively cheap computing power, allowed both academic researchers and commercial application developers to explore using neural networks to solve their problems.

A variety of factors play a role in determining whether a technology becomes a commercial success or failure. Perhaps the most crucial point is whether viable alternatives exist to solve the pressing problems of the day. The failure of symbolic artificial intelligence to satisfy industry requirements for robust pattern recognition and adaptive behavior opened the door for neural networks to reenter the technical stage. It also helped that a whole new generation of researchers arrived on the scene who didn't "know" that neural networks wouldn't work. The advance in computing power and integrated circuits, which made putting hundreds of processors on a single chip possible, certainly contributed to the reemergence of neural network technology.

If you have heard about neural networks in the past few years, it might have been in the context of a "hot new technology" that is revolutionizing fields like stock portfolio management (Rugiero 1994) and credit card fraud detection (Norton 1994). Or it might have been in the realm of science fiction. Which is it? Science fiction or science fact? Well, that depends on your point of view. If you are interested in using neural networks to solve practical problems, such as predicting future sales, modeling a manufacturing process, or detecting failures in machines, then neural networks are real, here today, and available (see Table 2.1 for a list of commercial neural network applications). If you want to build Commander Data, the personable android on the "Star Trek: The Next Generation" show, then you are still talking fiction. Neural networks have not allowed the creation of machines with humanlike intelligence or behavior. However, researchers in many fields, from neurophysiology and cognitive science to computer science and electrical engineering, are working toward that goal.

The availability of commercial neural network development tools has increased the number of fielded applications. Tools from vendors such as HNC Inc., IBM Corp., NeuralWare Inc., and Ward Systems Group run on PCs, workstations, minicomputers, and mainframes. These tools provide interactive environments for developing neural networks and the means to deploy applications. See appendix A for a description of the functions provided by the IBM Neural Network Utility.

In summary, we are now at a technological state where neural computing, computing based on a brainlike model, is both possible and practical. It has now been almost 10 years since the reemergence of neural networks. The combination of the march of computing technology and theoretical work has

TABLE 2.1 Commercial Neural Network Applications

Application	Industry	Function
Database marketing	all	Clustering, Classification, Modeling
Customer relationship management	all	Clustering, Classification, Modeling
Fraud detection	Finance, Insurance, Health	Classification, Modeling
Optical character recognition	Finance, Retail	Classification
Handwriting recognition	Computer, Finance	Clustering, Classification
Sales forecasting, inventory control	Manufacturing, Wholesale, Retail, Distribution	Clustering, Time-Series Forecasting
Stock portfolio management	Finance	Classification, Time-Series Forecasting
Bankruptcy prediction	Finance	Modeling
Job shop scheduling	Manufacturing/Process	Constraint Satisfaction
Process control	Manufacturing/Process	Modeling
Bond rating	Finance	Classification
Mortgage underwriting	Finance	Modeling, Time-Series Forecasting
Mineral exploration	Energy	Clustering, Classification
Medical (lab) diagnosis	Health	Classification, Modeling
Power demand prediction	Utility/Manufacturing	Time-Series Forecasting
Computer virus detection	Computer	Classification
Speech recognition	Computer	Clustering, Classification
Market price estimation	Real Estate, Finance	Modeling

created a fundamentally new approach to solving problems with computers. Much more than just an incremental step forward in computing ability, neural networks are a leap forward, providing a completely new paradigm both for formulating problems and for solving them. In the next section, I examine how we must change our fundamental problem-solving approach if we are to exploit this new technology.

Changing the Problem-Solving Paradigm

Solving problems with computers has become commonplace. It is done every day by many people. However, it is by no means natural for most people. Even those with an aptitude for computer programming must be trained in the organized, step-by-step procedures required to write a program to get a computer to solve a problem. Some people are unable to think at the level of detail necessary to specify the sequence of elementary operations needed to perform even the most basic computing functions. Good programmers have linear, sequential thought processes. They view a problem as a connect-the-dots puzzle, where the first and last points are specified, and their job is to link the dots, one by one, until the solution emerges.

The problems we use computers to solve are quite varied. They could be traditional computer applications such as accounting and payroll, or they could be optimization problems such as how to assign shipments to trucks for delivery, or how to manage an inventory replenishment system. In every case, the original problem must be recast into a form that can be solved on a computer, using computer programming languages.

One of the basic tasks that a computer systems analyst must perform is to translate business problems into computer solutions. There are a large number of methodologies for doing this. Perhaps the most common is the top-down structured approach, where a problem is broken down into subproblems. Data is identified and processes are defined for manipulating the data. Well-known techniques are used to design programs and algorithms to solve the data processing problem. After that it is just a simple matter of programming. It's been done a million times.

For the past 20 years, the computer science curriculum in universities has been based on this approach to solving problems with computers. Classes on data structures, algorithms, systems analysis and design, and programming languages such as COBOL, Pascal, and C are all standard offerings. Students are taught all of the important information needed to define, conceptualize, and solve problems on digital computers.

In the past few years, the old top-down structured design and centralized application has given way to an approach based on objects. Instead of focusing on data and then deciding how to process it, object-oriented analysis and design focuses on defining business objects that correspond

to real-world objects. Each object contains some set of data, which defines its current state, and a set of operations, which that object can perform or respond to. Hand in hand with a new problem analysis model is the increasing use of object-oriented programming languages such as Smalltalk and C++ (object-oriented C), which support the constructs used in object-oriented analysis. This major shift in the systems analysis methodology is currently causing a corresponding change in university computer science curriculums. In many ways, the move to object-oriented technology is a major new paradigm for solving problems on digital computers. The rigid waterfall software development process, which flows from requirements, to analysis and design, to code and test, which was the standard development methodology under the structured programming technique, is now giving way to the iterative development and rapid prototyping process more natural for the new object-oriented approach. The software development method has changed, but the fundamental computing architecture has not.

But what if the underlying computing architecture is not a serial digital machine? What if it is massively parallel, with hundreds or thousands of processors? In a similar way, we need to develop a curriculum on parallel computing, on parallelizing processes and synchronizing access to shared variables. When we change the underlying assumptions about the computer architecture, we call into question many of the basic tenets of the computer industry's approach and the accumulated knowledge in solving problems with computers gained over the past 30 years. Classes on parallel architecture and programming are being offered. If we treat the transition to massive parallelism as an evolutionary step from serial computers, we think we can get there. But it is hard to teach many people how to think in parallel. After all, we have just learned how to train people to think in linear, sequential ways so they could program digital computers.

Change is hard, especially when a methodology has proven useful and profitable over the past 30 years. Now suppose we have a new type of computer—not just a logical extension of serial digital computers, but a radically different computing model and architecture. How are we going to teach people to solve problems using these computers? It won't be easy if we approach the problem as one of retraining thousands of programmers. It will be impossible if we try to present a modified version of the familiar waterfall software development process.

What we need is a totally different approach. A fundamentally different computing model requires a rethinking of the software development process from problem definition through testing. Neural networks and neural network development are a different computing model, and solving problems with neural networks is quite similar to the way people naturally solve problems. A neural network learns to solve problems by being given data, examples of the problem, and its solution. People do this all of the time.

Knowledge Workers and Neural Networks

Suppose we just hired a new loan officer to make credit decisions for our bank (let's call her Jennifer). Jennifer will be given some hard and fast rules, but there is a large gray area where the loan decision is up to her. Either she grants the loan and we take our chances, or she doesn't grant the loan and we give up the opportunity to make money on the loan. It's going to take a while for Jennifer to learn how to do this well. Figure 2.3 shows Jennifer at her desk for her first day on the job.

Figure 2.3 Jennifer at work, first day on the job.

Jennifer processes about 10 loans a day. At first, every application is a completely new experience. Some loans she can determine simply based on the lack of a job, history of bankruptcy, or other aspects. Others she needs to just weigh the information and make a "gut call." One person's income history is stable, they have two kids, and the loan is for a good purpose. Jennifer weighs all of these factors and says, "Yes, let's take this business." Another one is not so clear cut. The person has changed jobs recently, there are some late payments on some of the credit history. Jennifer says "No, let's not take this business."

Over time, Jennifer gets feedback on her performance. Each month her manager comes in and lists the customers with late payments. Jennifer reviews those loan applications and now sees telltale signs of problems. She

continues to learn how to judge whether someone is going to pay back the loan on time or not. Of course, Jennifer gets no feedback on the ones she rejects. They might have gone across the street to Second National Bank and been given the loan she rejected, and that bank made the money she gave up. But overall, Jennifer is doing a pretty good job, the bank is lending most of the money it has allocated to loans, and it has a reasonable default rate.

Let's examine what Jennifer has done. She looked at many examples of applicants and learned to classify them as good or bad prospects. More than that, she can say which one is a "better" risk than another. If Jennifer made a wrong determination on someone who later didn't repay the loan, she got feedback that said, "You made a mistake." So Jennifer had to look at the applicant data and adjust her internal weightings for the significance of various factors. In short, Jennifer "learned" how to perform her job. (See Figure 2.4.)

Figure 2.4 Jennifer at work, six months later.

Now let's say the bank's management would like Jennifer to move on to business loans. But they still need to cover the consumer side of the business. Rather than hire and train a new person to learn the distinctions between good and bad credit risks, they would like to "clone" Jennifer (or at least her expertise). They would like to use her accumulated experience in an automated way. Bank management has heard of a new application technology called neural networks, which can learn to do the same job Jennifer does. Jennifer says "I doubt it. How can it learn to act the way I do?" Well, first all of the data from Jennifer's loan decisions must be collected. Each transaction has the application data and her decision and, for the ones she accepted, the profitability and currency of the account. The neural network is presented with the same factors Jennifer used to make her decisions, along with her decision. After a brief "training" period, the neural network

is ready to test. A new application comes in, and it is presented to the neural network. It makes a decision. Jennifer looks over the application and says "Wow, it did just what I would have done. How did it do that?"

Like many decision or knowledge worker tasks, the expert in this example learned to weigh the various input factors and combine them to come up with an overall score or decision. Initially, some decisions were made that were "wrong." In order to correct the mistake, the expert had to adjust her internal weightings so that next time she would not make the same mistake. This is a common framework for many jobs performed by people. They might start out with some rules that can be used in the extreme or clear-cut cases. But the real skill comes in learning how to judge the in-between cases, in recognizing the subtle distinctions between success and failure. The most-often-used metaphors are that the expert "sees the solution," or "the answer jumps out."

Building neural network applications is similar to training a new knowledge worker. We must be able to give examples of the clear-cut extreme cases, and we must be able to give sufficient data in the "gray" cases so that the neural network can learn to accurately make decisions. Will the decision always be correct? No. Does the knowledge worker or expert always make the correct decision? Obviously not. But the expert learns from experience, and so can the neural network.

Making Decisions: The Neural Processing Element

The digital computer architecture (see Figure 2.5) consists of a central processing unit (CPU) with a set of registers and an arithmetic logic unit (ALU), along with a store of addressable memory that holds both instructions and data. The digital computer is called a sequential machine because it starts reading instructions from memory and then walks through memory (with some skips or branches here or there as dictated by the program), reading instructions and data, processing the data using the ALU, and then writing the results back to the memory.

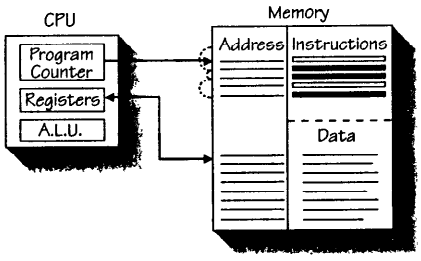


Figure 2.5 Digital computer architecture diagram.

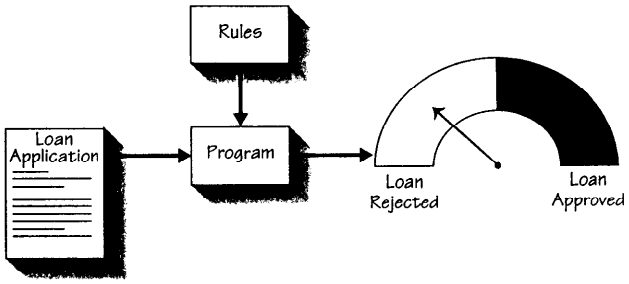


Figure 2.6 Making decisions the digital way.

Using a digital computer to make a decision is a relatively straightforward process. The arithmetic logic unit, as you might expect from the name, performs mathematical operations such as addition and subtraction. It also performs basic Boolean logic functions such as testing whether two numbers are equal, or whether one is larger than another one. Making Boolean or binary yes/no decisions is a fundamental part of a digital computer.

Because of this, mapping from a high-level language computer statement such as "if Income > 100000 then LoanApproved = True else LoanApproved = False" into elementary computer operations is easy. The value of Income and 100000 are loaded into registers from memory. The ALU tests to see if Income is greater than 100000 and if this is True, then the "LoanApproved = True" code is executed, otherwise the "LoanApproved = False" code is executed. While this might seem terribly obvious to you, I point this out to emphasize that the type of decision making that we can do in programming languages is a function of the underlying capability of the digital CPU. The programming languages we use today were built up and derived from the basic binary computing capabilities of digital computers. This relationship colors all of our thinking about how we make decisions with computers today.

Figure 2.6 illustrates how applications are developed using digital computers. A set of business rules or algorithms is translated into a computer program. The input data is fed into the program, the program processes the data and spits out a yes or no decision.

Of course, coding a decision statement is not hard. Knowing what the number should be to test for is the hard part (should it be 100000 or 100500?). The point is that digital computers are great at making binary (yes/no) decisions, as long as you tell them exactly what to compare. Life is not always so simple.

Unlike the digital computer, neural networks and neural computers are based on a model of the brain. A processing element in a neural network

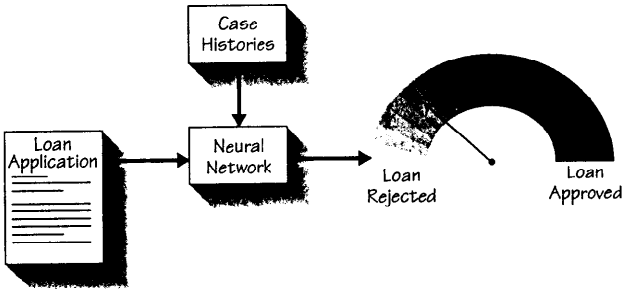


Figure 2.7 Making decisions the neural network way.

makes decisions in a very different way than a digital computer. Rather than reading an instruction from memory and then loading data items into registers and performing the specified logical operation like a digital computer, a neural processing element operates much differently.

A neural processing element receives inputs from other connected processing elements. These input signals or values pass through weighted connections, which either amplify or diminish the signals. Inside the neural processing element, all of these input signals are summed together to give the total input to the unit. This total input value is then passed through a mathematical function to produce an output or decision value ranging from 0 to 1. Notice that this is a real valued (analog) output, not a digital 0/1 output. If the input signal matches the connection weights exactly, then the output is close to 1. If the input signal totally mismatches the connection weights then the output is close to 0. Varying degrees of similarity are represented by the intermediate values. Now, of course, we can force the neural processing element to make a binary (1/0) decision, but by using analog values ranging between 0.0 and 1.0 as the outputs, we are retaining more information to pass on to the next layer of neural processing units. In a very real sense, neural networks are analog computers.

Figure 2.7 shows how a neural network would be used to make a loan approval application. The inputs to the neural network are examples or case histories of the application problem and its solution. The loan application data is fed into the neural network and a value from 0.0 to 1.0 is the result.

Each neural processing element acts as a simple pattern recognition machine. It checks the input signals against its memory traces (connection weights) and produces an output signal that corresponds to the degree of match between those patterns. In typical neural networks, there are hundreds of neural processing elements whose pattern recognition and decision-making abilities are harnessed together to solve problems.

The Learning Process: Adjusting Our Biases

Suppose we present an input pattern to a neural network and it produces an output signal that is wildly incorrect. What mechanism exists to change the output? For example, let's say that the output value is much lower than it should be. One way to increase the output of the neural processing element is to move the memory traces or connection weights closer to the input signal. This would improve the degree of match and increase the output value.

This is exactly the method used to "program" or "train" neural network computers. Examples are presented to a neural network, it makes a prediction, and the connection weights are adjusted so that the output corresponds more closely to the desired output. This adjustment process is done automatically by the learning algorithm being used to train the network. By making connections stronger or weaker, reinforcing or inhibiting, the artificial neural network is mimicking the behavior of the synapses of the brain, which undergo physical changes in response to input patterns and feedback. Figure 2.8 is an example of this process. In step 1, the neural network

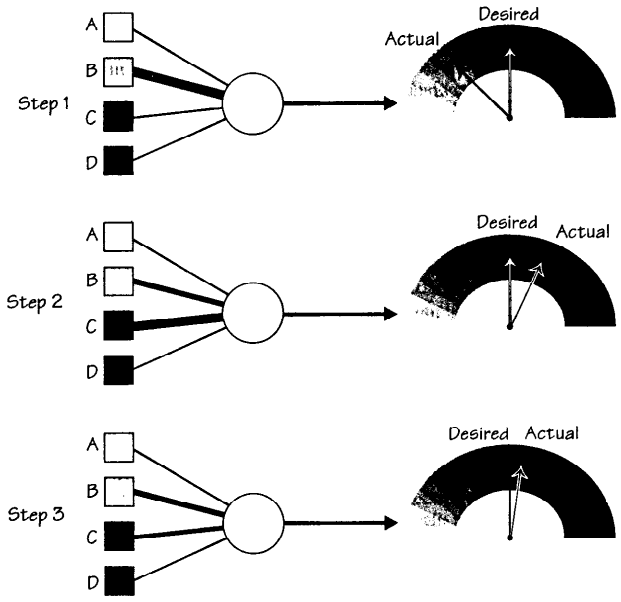


Figure 2.8 Neural networks—learning from experience.

outputs a value less than the desired value. Based on the difference between the desired and actual outputs, the connection weights are modified. In step 2, we see that connection weight B is smaller and weight C is larger, producing an actual output that is now slightly larger than desired. Once again, the weights are adjusted. In this case, weight C is reduced, so that the output for step 3 is very close to the desired output.

Basic Neural Network Functions

It should come as no surprise that neural networks perform many of the kinds of tasks that humans do. These tasks, which are important for our survival as a species, involve simultaneous processing of large amounts of data (vision, hearing, touch), where fast, accurate pattern recognition and responses are required. The architecture of the human brain evolved to solve these types of problems.

Classification

Perhaps the most basic function performed by our brain is that of discrimination, or classification between two things. We are capable of analyzing objects using the subtlest and finest features to assess both the similarities and differences. We can classify animals as friendly or dangerous, plants as good to eat or poisonous, weather as pleasant or threatening. Every day, in hundreds and thousands of cases, we classify things. In the business environment, we have another need to make classifications. Is a loan applicant worthy of a mortgage for a new house? Should we extend a line of credit to a growing business? Should we mail our new catalog to this set of customers or to another one? We make these decisions based on classification.

Clustering

While classification is important, it can certainly be overdone. Making too fine a distinction between things can be as serious a problem as not being able to decide at all. Because we have limited storage capacity in our brain (we still haven't figured out how to add an extender card), it is important for us to be able to cluster similar items or things together. Not only is clustering useful from an efficiency standpoint, but the ability to group like things together (called chunking by artificial intelligence practitioners) is a very important reasoning tool. It is through clustering that we can think in terms of higher abstractions, solving broader problems by getting above all of the nitty-gritty details.

The business applications of clustering are mainly in the marketing arena. By clustering customers into groups based on important similar attributes, such as which products they buy or demographics they share, you can start

to understand your markets in finer detail. This information can be used to target these groups of similar customers with products that many of them have purchased in the past or add-on services, which might appeal to that segment. This use of clustering is also called market segmentation.

Associative memory

The human mind is an amazing storage device. A lifetime worth of memories are stored with an indexing system that would make a database administrator drool. People store information by associating things or ideas with other related memories. There seems to be a complex network of semantically related ideas stored in the brain. One of the fundamental theories of learning in the brain, called Hebbian learning after Donald Hebb (1949), says that when two neurons are activated in the brain at the same time, then the connection between them grows stronger. At the time, Hebb postulated that physical changes in the synapse of the neurons took place. This hypothesis has since been proven by neurophysiologists.

Some of the earliest work in neural networks deals with the creation of associative memories. Unlike classification or modeling where we are trying to learn some fundamental relationship between inputs and output, associative memory requires a mapping of any two items. Neural network models such as Binary Adaptive Memories and Hopfield networks have been shown to be limited capacity, but working, associative memories.

Modeling or regression

People build practical, useful mental models all of the time. Seldom do they resort to writing a complex set of mathematical equations or use other formal methods. Rather, most people build models relating inputs and outputs based on the examples they have seen in their everyday life. These models can be rather trivial, such as knowing that when there are dark clouds in the sky and the wind starts picking up that a storm is probably on the way. Or they can be more complex, like a stock trader who watches plots of leading economic indicators to know when to buy or sell. The ability to make accurate predictions from complex examples involving many variables is a great asset.

By seeing only a few examples, people can learn to model relationships. We use our ability to interpolate between the exact examples to generalize to novel cases or problems we have never seen before. It is the ability to generalize that is a strength of neural network technology.

Time-series forecasting and prediction

Like modeling, which involves making a static one-time prediction based on current information, time-series prediction involves looking at current information and predicting what is going to happen. However, with time-

series predictions, we typically are looking at what has happened for some period back through time and predicting for some point in the future. The temporal or time element makes time-series prediction both more difficult and more rewarding. Someone who can predict the future based on what has occurred in the past can clearly have tremendous advantages over someone who cannot.

People are very good at using context to help modify their predictions of what the outcome of certain situations will be. For example, knowing that it is a day after a holiday would suggest to most people that banks would be busier than normal, and a shopper would take this into account. If a bank manager were planning staffing requirements, then this would be considered. When neural networks are used for time-series forecasting problems, neural networks also must be given this context information so they can factor it into their predictions.

Constraint satisfaction

Most people are very good at solving complex problems that involve multiple simultaneous constraints. For example, we might have a list of errands to run. Knowing that buying groceries should be at the end and that we can perform three of the tasks at a single shopping center would help us in our planning. In business, we often want to maximize conflicting goals, increase customer satisfaction, reduce costs, increase quality, and maximize profits. We can certainly increase customer satisfaction if we sell our products at half price. However, this wouldn't be good for our profitability. We could reduce costs by cutting out inventory down to just a few items, but this would certainly have a negative impact on customer satisfaction.

Having multiple conflicting goals is a natural part of life. People deal with this state of affairs all of the time and think nothing of it. Digital computers and Boolean logic, however, have a hard time dealing with this. In contrast, neural networks with their weighted connections and analog computing have proven themselves extremely adept at solving constraint satisfaction and optimization problems.

Summary

People are familiar with the computer metaphor, that the human brain is nothing more than a computer made of flesh. This view is the result of both the success of the digital computer and of the symbolic school of artificial intelligence where rule processing and symbol manipulation were equated with intelligence. However, neural networks present an alternative model based on the massive parallelism and the pattern recognition abilities of the brain.

Neural networks share much more with the architecture and our current understanding of how people learn and make decisions than with the current digital computer model. Neural networks learn from examples. They take complex, noisy data and make educated guesses based on what they have learned from the past. Given the requirements for data mining against large databases of historical data, neural networks are a natural technology for this type of application.

More than just a new computing architecture, neural networks offer a completely different paradigm for solving problems with computers. In my example, I showed how a knowledge worker learns to do her job by working through examples and getting feedback on her performance. This approach was contrasted to how computers are used to solve these kinds of problems. The neural network training approach is more similar to how people work and learn.

The process of learning in neural networks is to use feedback to adjust internal connections, which in turn affect the output or answer produced. The neural processing element combines all of the inputs to it and produces an output, which is essentially a measure of the match between the input pattern and its connection weights. When hundreds of these neural processors are combined, we have the ability to solve difficult problems such as credit scoring.

Many of the basic functions performed by neural networks are mirrored by human abilities. These include making distinctions between items (classification), dividing similar things into groups (clustering), associating two or more things (associative memory), learning to predict outcomes based on examples (modeling), being able to predict into the future (time-series forecasting), and finally juggling multiple goals and coming up with a good-enough solution (constraint satisfaction).

Neural networks are a computing model grounded on the ability to recognize patterns in data. As a consequence, they have many applications to data mining and analysis. In the remainder of this book, I explore how neural networks can be applied to solve common business problems.

References

Fox, B. 1993. Move over, expert systems: Neural networks the new wave in AI, *Chain Store Age Executive*, Vol. 69, No. 4, Apr., pp. 65–68.

Hebb, D.O. 1949. *Organization of Behavior*, New York: Science Editions.

Jubak, J. 1992. In the Image of the Brain, Boston: Little Brown.

McCullough, W.W., and W. Pitts. 1943. A logical calculus of the ideas immanent in nervous activity, Bulletin of Mathematical Biophysics, Vol. 5, pp. 115–33.

Minsky, M., and S. Pappert 1969. Perceptrons, Cambridge: MIT Press.

Newell, A. 1980. Physical symbol systems, Cognitive Science 4, pp. 185–133.

Norton M. 1994. Close to the nerve (credit card fraud), *Banking Technology* (UK) Vol. 11, No. 10, Dec., pp. 28–31.

Port, O. 1995. Computers that think are almost here, *Business Week*, July 17, 1995, pp. 68–72.
Pracht, W.E. 1991. Neural networks for business applications, Interface: Computers in Education Quarterly, Vol. 13., No. 2, Summer.
Rosenblatt, F. 1962. Principles of Neurodynamics, New York: Spartan Books.
Ruggiero, M.A. 1994. Interpreting feedback to build a better system, *Futures: The Magazine of Commodities & Options*, Vol. 23, No. 8, July, pp. 46–48.
Rumelhart, D.E., McClelland, D (1986) *Parallel Distributed Processing*, Vol. 1. and Vol. 2, Cambridge, MA: MIT Press.
Widrow, B. and M.E. Hoff. 1960. Adaptive switching circuits, 1960 IRE WESCON Convention Record, Vol 4, pp. 96–104. New York: Institute of Radio Engineers.

Chapter

3

Data Preparation

"The real question is not whether machines think, but whether men do."
B.F. SKINNER

In this chapter, I explore the issues related to the first major step in the data mining process—data preparation. Modern database system architectures, data access, data cleansing, and data selection are described. The remainder of the chapter deals with data set management and the preprocessing of data for data mining with neural networks.

Data: The Raw Material

It is certainly true that having data is a necessary prerequisite to doing data mining. However, just having the data is not always sufficient. There is always the question of if we have enough data. Then there is the issue of if we have clean, reliable data. Finally, and most importantly, is the determination of whether we have the right data. Only someone with domain knowledge, someone who understands the data and what it means, can select the right data for a data mining operation. As we will see, this application of knowledge about the data is used in several different ways in data mining, especially in the data preparation phase.

In most cases, the data used for a data mining operation has been just sitting around collecting dust. The data is created as a by-product of performing common business transactions, is stored in an operational database, and is archived to tape for long-term storage. Many companies are implementing corporate data warehouses, which keep the operational data online and avail-

able for extended periods of time. Some companies are creating marketing databases, which keep only information related to customer relationships and purchasing history online. Whatever historical data is available, it is the raw material for the data mining process.

One problem that often arises when a data mining process is proposed is that only part of the business process is computerized. Part of the process is online, and part of it is manual, so only a portion of the data is available on the computer system. For example, in a credit approval process, most of the information is entered into the computer system, but the credit report and appraiser's report are not. They are part of the credit applicant's paper folder. In this case, we have all of the data, but it is not available in a form that can be used immediately for data mining. If the goal is to analyze or automate this process through data mining, we will first have to get the information from the paper documents into the computer system. The cost of this can be substantial. An alternative approach is to automate the entire process and begin collecting the data so that data mining can be performed at some future time.

Although it might sound crazy, we might want to do data mining even though we don't have any data. Obviously this doesn't make sense for decision support applications, but for certain types of application development it works well. The key is that even though we don't have the data, we can generate it. That is, we don't have records of actual decisions, but we can write some rules that can be used to generate training examples, which define 80% of the desired behavior. With this approach, we can off-load 80% of the caseloads from workers (the easy cases), and let them focus on 20% of the cases (the hard cases). In the mean time we can start collecting the decisions they make and later update the neural network so it can cover more cases in the gray area.

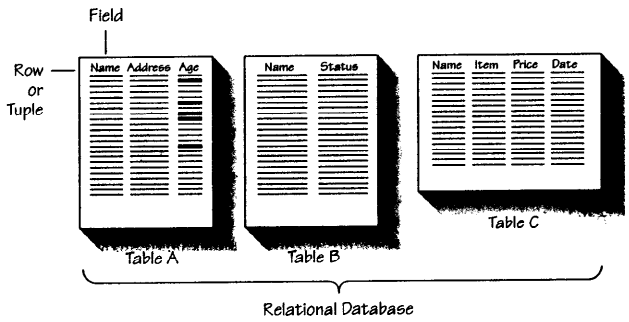
The data used in a data mining project might be stored either in a flat file or in a database. Even when the data is actually stored in a database, it is often dumped to a flat file for processing by the data mining algorithms. This is sometimes done for simplicity, as an easy way of handing off data to a consultant or third party who is actually going to do the data mining. In other cases, this is done to avoid the performance penalties, which are sometime paid when iterating through large relational databases. One issue that arises when large data sets have to be preprocessed is ensuring that there is enough disk space for all of the preprocessed data. In the next section, we look at the basic features of relational database systems and some of the performance issues.

Modern Database Systems

A computer system is not considered complete today without having some sort of database capabilities. Although data can be stored in plain text files easily enough, any significant data processing activity where business transactions are processed, read, and updated today use a relational database. In the past, hierarchical databases such as IBM's IMS or network databases such as CODASYL were used to store business data. The advent of object-oriented programming languages has prompted the development of a new form of database systems based on the object paradigm. Object databases have become very popular, and they are growing quite rapidly. But even though much data still resides in the hierarchical and network databases on mainframe computers, and object databases are growing, the information technology world today relies largely on relational database technology and will for the foreseeable future.

Relational databases such as IBM DB2, Oracle, and Sybase all treat data as a collection of records called *tuples* (see Figure 3.1). Each record or row in the database consists of a collection of columns or fields. Thus a relational database, no matter how large, can always be considered to be a large table of data consisting of rows and columns. The relational algebra first designed by Codd and Date (1990) specifies a set of logical operations that can be used to select rows, extract columns, and join two or more relational tables together in order to get the desired "view" of the data. This manipulation of relational data has been standardized in the industry by the Structured Query Language, SQL. Business application programming languages such as COBOL and RPG support SQL interfaces to relational databases. Even client-based graphical query tools end up ultimately issuing dynamic or static SQL statements to access the host data.

Accessing data from a database consists of selecting columns of data from either all records or from records containing specific values or ranges



Select * from Table A where Age > 18 and Age < 30
Select Item from Table C, Table A where Table A.Name = Table C.Name and Age > 18

Figure 3.1 Relational database systems.

of values in the data. The result of all SQL operations on relational database tables is another table. This table is then accessed either using key fields or sequentially by using a cursor to walk through the table. Thus, when all is said and done, the data coming from a relational database is a single record consisting of a set of columns or fields.

As the amount of data generated in a typical business day has grown, the relational databases that store this data have also needed to grow. Today individual relational databases can be gigabytes in size. In 1994, 32% of the DB2/400 databases ranged in size from 11 to 50 gigabytes, while 18% of Oracle databases were also in that range (Ovum 1994). Almost 30% of the DB2 databases on mainframe computers were larger than 50 gigabytes. Depending on the complexity of the layout of the tables, a query for information from a set of tables might take anywhere from seconds or minutes, to even hours or days to complete. As the business user's focus has turned to the task of examining the business data, demands on database performance has increased to such an extent that much of the differentiation between commercial database vendors is not in the features and function, but in the area of performance.

Parallel Databases

As the quest for better performance has continued, database vendors have moved to parallel hardware configurations in order to provide the needed transaction processing and query response times. These parallel database architectures can be split into two major camps, symmetrical multiprocessing (SMP) or tightly coupled systems, and shared nothing or loosely coupled systems. In the following sections, we explore these two popular parallel database architectures.

SMP database architectures

Symmetrical multiprocessing database systems use multiple processors sharing memory in the same computer system to process queries in parallel. All of the processors in the system can access all of the data on the hard disks, and they also all share the system memory. In most SMP systems, a single query is split into pieces, and these pieces are sent off to the individual processors in the system. Figure 3.2 shows a task or application running on a four-way AS/400 system with the SMP version of DB2 for OS/400. Each processor can work on its part of the problem, accessing the hard disks to retrieve the data and returning its partial results. These partial results then have to be combined to return the overall results of the query to the requesting application program.

One major problem with SMP database systems is that the shared memory becomes a system bottleneck. This is because each processor, while

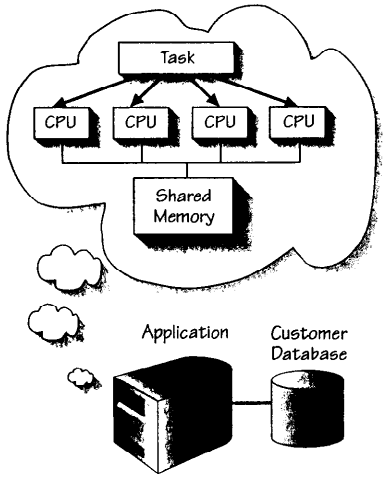


Figure 3.2 SMP database systems.

able to perform independently on a piece of the problem, still has to wait until the main memory is not being accessed by any of the other processors in the system. SMP is usually effective for small number of processors, ranging from 4 up to about 16.

Shared-nothing database architectures

Because of the scaling problems of SMP database systems, vendors have introduced loosely coupled or shared-nothing database systems. In a shared-nothing architecture, each processor has its own disks and its own memory, so that memory is no longer a bottleneck. Figure 3.3 depicts an example of a customer database in a shared-nothing configuration. The individual relational database table is split across three nodes in an three-way shared database system. When a query is run, the query is sent to each processor, which checks the data on its local disks using its own memory and then sends back the partial results. As in an SMP system, the partial results are combined to yield the final query results, which are returned to the application program.

The beauty of shared-nothing parallel databases is that they are inherently scalable. Systems with 128 and more nodes have been built and used

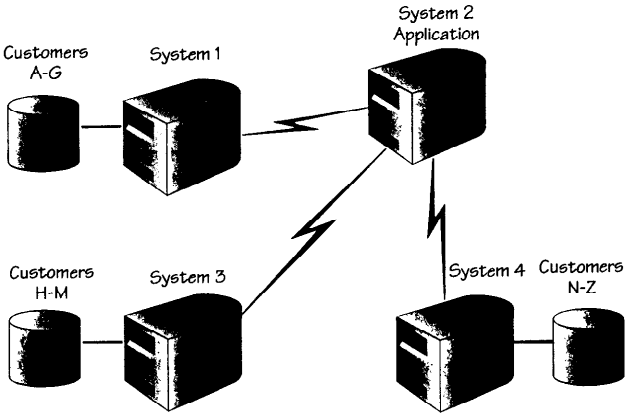


Figure 3.3 Shared-nothing database systems.

in commercial environments. Shared-nothing databases are true parallel databases. All database operations—including inserts, deletes, and indexing operations—can be performed in parallel with the corresponding increases in performance. To many people, data warehousing doesn't really make sense unless you are talking about a shared-nothing database architecture. The IBM SP2 with DB2 Parallel Edition and DB2 MultiSystem for AS/400 are both examples of shared-nothing, highly scalable relational database systems (Finkelstein 1995).

Data Cleansing

When operational data gets loaded into a centralized data warehouse, the data often must go through a process known as "data cleansing." A sad but true fact is that not all operational transactions are correct. They might contain inaccurate values, missing data, or other inconsistencies in the data. The transaction might be checked by an application program, which detects the bad data and notifies the originator of this, but the bad data often remains in the database. This was not such a problem when the database was viewed primarily as an archival mechanism. However, if the data warehouse is to be turned into a fount of raw material for corporate business intelligence gathering, then the data must be as clean and correct as possible.

Several techniques are being used to clean data either before or after it gets into the data warehouse. These include rule-based techniques, which

evaluate each data item against metaknowledge (knowledge about the data) about the range of data expected in that field and constraints or relationships to other fields in the record (Simoudis, Livezey, and Kerber 1995). Visualization can also be used to easily identify outliers, or out of range data, in large data sets. Another approach is to use statistical information to set missing or incorrect field values to neutral, valid values.

Data Selection

Once we have the database to train the neural network, the next step is to decide what data is important for the task we are trying to automate. Maybe our database has 100 fields, but only 10 are used in making a decision. The problem is that, in many cases, we don't know exactly which parameters are important in a decision process. Fortunately, neural networks can be used to help determine which parameters are important and to build a model relating those parameters.

The data selection process really takes place across two dimensions. First is the columns or parameters, which will be part of the data mining process. Second is the selection of rows or records, based on the values of individual fields. The underlying mechanism used to access all relational databases is SQL, as discussed earlier. However, most database front-end tools allow users to specify which data to access using fill-in-the-blank forms.

The data selection step requires some detailed knowledge of the problem domain and the underlying data. Often the data that is stored in the database needs to be massaged or enhanced before data mining can begin. This preprocessing step is described in the next section.

Data Preprocessing

Data preprocessing is the step when the clean data we have selected is enhanced. Sometimes this enhancement involves generating new data items from one or more fields, and sometimes it means replacing several fields with a single field that contains more information. Remember, the number of input fields is not necessarily a measure of the information content being provided to the data mining algorithm. Some of the data could be redundant; that is, some of the attributes are simply different ways of measuring the same effect. Sometimes the data needs to be transformed into a form that is acceptable as input to a specific data mining algorithm, such as a neural network.

Computed attributes

A common requirement in data mining is to take two or more fields in combination to yield a new field or attribute. This is usually in the form of a ratio of two values, but could also be the sum, product, or difference of the

values. Other transformations could be turning a date into a day of the week or day of the year. Computed attributes are often necessary because the transaction processing application was designed to handle the minimum amount of data required to log the transaction. In the past, the focus has been on minimizing storage requirements and processing time, and not on maximizing the amount of information gathered by transactions.

Scaling

Another transformation involves the more general issue of scaling data for presentation to the neural network. Most neural network models accept numeric data only in the range of 0.0 to 1.0 or −1.0 to +1.0, depending on the activation functions used in the neural processing elements. Consequently, data usually must be scaled down to that range.

Scalar values that are more or less uniformly distributed over a range can be scaled directly to the 0 to 1.0 range. If the data values are skewed, a piece-wise linear or a logarithmic function can be used to transform the data, which can then be scaled into the desired range. Discrete variables can be represented by coded types with 0 and 1 values, or they can be assigned values in the desired continuous range.

Normalization

Vectors or arrays of numeric data can sometimes be treated as groups of numbers. In these cases, we might need to normalize or scale the vectors as a group. There are several ways of doing this. Perhaps the most common vector normalization method is to sum the squares of each element, take the square root of the sum, and then divide each element by the norm. This is called the Euclidean norm. A second way to normalize vector data is to simply sum up all of the elements in the vector and then divide each number by the sum. In this way, the normalized elements sum to 1.0, and each takes on a value representing the percentage of contribution they make. A third way to normalize vector data is to divide each vector element by the maximum value in the array. This max norm is also useful since it requires very little overhead processing cost.

Symbolic mappings and taxonomies

In many cases we need to perform transformations of symbols to other symbols before we can turn them into numeric values. A common use would be to aggregate members of some class or group into a single symbol for data representation purposes. For example, a store might sell 100 varieties of juice, all with unique SKUs (store keeping units) and alphanumeric identifiers. If we want to model the purchases of various classes of beverages, we need to treat all of these products as one.

This type of mapping can be used to look at categories at several levels of granularity. For example, we could go through several stages of mappings, depending on our needs. Figure 3.4 shows a common use of taxonomies to view problems at different levels of abstraction. If we managed a grocery store, we might be interested in the relationship between sales of condiments and some other category of food (hot dogs, hamburgers, buns) or the weather. At one level, we think about and group all condiments into one abstract category. Or we could go down one level and think about individual types of condiments such as pickles. Then we could check the relationship between pickle consumption and the number of births recorded at the local hospitals. Or if we need to restock our pickle supply, we need information about particular types of pickles that are selling well. At the next level (not shown) we would have to determine which brand and size to carry in our store.

Symbolic to numeric translations

Symbolic to numeric translations are often required to turn discrete symbols or categories into numerical values for processing by the data mining algorithms. The most basic form this can take is of a simple table lookup, where the symbol is compared against a list of symbols and when it is found, a corresponding numeric value is used. Care must be taken to ensure that illegal or undefined symbols get assigned some common "don't care" or "unknown" value. Another more sophisticated approach is to use a hashing function, which is an algorithm that takes a character string and generates a unique numeric value.

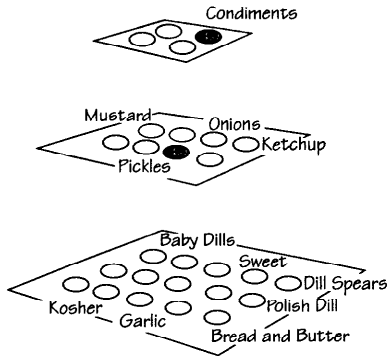


Figure 3.4 Taxonomies in data mining.

Data Representations

Although there are many data types supported in relational database systems, most can be easily mapped into three logical data types. These include continuous numeric values, discrete numeric values, and categorical or symbolic discrete values. Time and date information present certain challenges, but they can also be mapped into numeric values by using the appropriate functions.

Figure 3.5 illustrates the major data types and how they can be represented for neural network data mining operations. If we start with a symbol "Apple," we could either map it to a specific integer value using a symbol table, or we could use a hash function to come up with a unique integer value for the string. This gives us a numeric discrete (integer) value. This value could be presented to a neural network in some cases. However, we usually want to take that value and either scale it or translate it into a coded type. In Figure 3.5, the symbol "Apple" is mapped to a discrete value of "5." This is scaled to 0.5 if we want to use a single real-valued continuous input, or it can be converted into any of the three codes shown on the right. In the following sections, I describe how source data can be translated between these basic logical types for use with neural networks for data mining.

Numeric data representations

Numeric data can be simple binary values (0 or 1) indicating on/off states, or it can be a range of discrete values (1 to 10) or a continuous range from

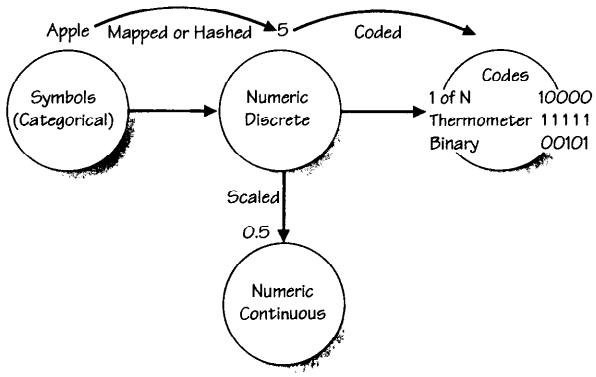


Figure 3.5 Common data representations for neural networks.

−1000 to +1000. In each case we must decide how to scale and represent that data. Most neural networks accept inputs in the range of 0 to 1 or −1 to +1. In this case binary parameters can be represented by the extremes of the input range.

Discrete values

Discrete variables are ones that take on only a fixed set of values. These typically denote a small set of classes, a set of responses to multiple choice questions (for example, A through E), or a fixed interval of integer values. The challenge for neural network representation of discrete values is to present these variable values in such a way that the network is able to discern the differences between values and can tell the relative magnitude of the differences if that information is available. Various coded data types are used to represent these values. In the following sections, I describe the most commonly used codes.

One-of-N codes. When a variable can take on a set of discrete values, it must be transformed into a representation that presents a unique set of inputs to the neural network for each distinct discrete value. Perhaps the most common representation for discrete variables is the one-of-N code. A one-of-N code has a length equal to the number of discrete categories allowed for the variable, where every element in the code vector is a 0, except for the single element, which represents the code value. For example if we have a set with four elements { apples, peaches, pumpkin, pie }, we can represent apples as 1 0 0 0, peaches as 0 1 0 0, pumpkin as 0 0 1 0, and pie as 0 0 0 1. The nice thing about one-of-N codes is that they are simple, easy to use, and the neural network can easily learn to discriminate between the various values. However, for variables with a large number of values, the one-of-N code can be very costly in terms of the size of the neural network. Using this representation, a single variable could expand to 100 or 1000 inputs with a corresponding explosion in the number of input weights.

Binary codes. An alternative representation is the standard binary code. Here each discrete category is assigned a value from 1 to N and represented by a string of binary digits. That is, if we have 64 possible values, we could represent it with a binary code vector of length 6. As long as the discrete values are arbitrary and not ordered in any way, a binary code is a fine way to represent data. However, note that there are large differences in the bit values as the discrete numbers get converted to binary codes. The seventh item has a code of 0 0 0 1 1 1, while the eighth has a code of 0 0 1 0 0 0. The Hamming distance is a measure of the similarity or difference between two binary strings. In this case, going from 7 to 8

results in a Hamming distance of 4. If we want the neural network to treat input patterns with a 7 or 8 as "similar," then we might want to choose the thermometer or temperature code.

Thermometer codes. A thermometer code is used most often when the discrete values are related in some way, usually by increasing or decreasing values. For example, we might have a discrete variable that takes on the following values { poor, good, better, best }. In this case we would like the difference between poor and best to be large (in Hamming distance) and the difference between better and best to be smaller. This is exactly what happens with a thermometer code since poor is represented as 1 0 0 0, while best is 1 1 1 1 (Hamming distance of 4), while better is represented as 1 1 1 0, (better to best is only a Hamming distance of 1 away).

There are other coding schemes that also work for discrete variables, but, in general, the one-of-N, binary and thermometer codes seem to get the job done.

Continuous values

For continuous values, the most common form of data translation operation is scaling of the data. For example, a variable that can take on values from 0 to 100 can be linearly scaled from 0.0 to 1.0. So a 20 would take on a value of 0.2, while an 80 would take on a value of 0.8. For evenly distributed variables like this, simple linear scaling works fine.

But what if the data is skewed in some way? For example, suppose 80% of the data is below 50 and we need to teach the neural network to make fine distinctions between values in the 0 to 50 range. An option is to scale the data using a piece-wise linear approach so that the data in the 0 to 50 range is expanded in representation, while the less important 50 to 100 range is compressed. This can be done by taking the 0, 50, 100 input range and scaling that onto a 0, 0.80, 1.00 range. In this case an input value of 50 gets assigned a value of 0.8, while a value of 75 gets assigned a value of 0.9. A 25-count difference in the input value translates into only a 0.1 difference that the neural network sees. However, an input of 10 would have a value of 0.16, while an input of 25 would be 0.40. Here the neural network sees a bigger difference in the input value and so can more easily discriminate between the differences in the input value. This might or might not be important. But it is important to remember that if a small difference in the input is really significant, say changing from 31 to 33° Fahrenheit, then we want to make sure our representation shows this significance to the neural network.

Another common need is to threshold data so that values out of the range of interest do not needlessly dilute our representation. For example, suppose that we have a range of incomes from 0 to 300,000 dollars. But we are only checking whether the person has an income of 35,000 or more. We might simply threshold the income so that values between 0 and 35,000 are passed through (and then scaled to 0 to 1), while values over 35,000 get thresholded to 35,000 (and so get a value of 1). Since we don't care about the full range of this variable, why make the neural network try to relate differences in income with our decision? There is no need to make the neural network learn something that it doesn't have to. It is wasted effort.

Symbolic data representations

We encounter symbolic data quite often in neural network applications. The most common, and easiest to deal with, are Boolean variables such as yes/no and male/female. However, quite often we must add a third condition (even for Boolean variables), which is the unknown condition. In this case we can use a one-of-N code of length 3 (yes, no, unknown) or a binary code of length 2 (yes, no, unknown, <unused>). Or we can decide to represent no as 0, yes as 1, and unknown as 0.5. All of the representations are valid. It depends on what is required by the application. The trade-off is in network size (the number of inputs) versus the ease of training (reduced training time).

For symbolic data representing unrelated discrete values, we simply map the symbol to an integer from 1 to N. For example, {apples, peaches, pumpkin, pie} maps to 1, 2, 3, 4. Of course, we would probably then scale 1–4 to 0–1 so that apples = 0.0, peaches = 0.33, pumpkin = 0.66, pie = 1.0. In essence we have the same representation options as discussed previously in the numeric discrete case. But each unique symbol must be mapped to a unique numeric value (see Figure 3.5). Depending on the application, we might or might not want to treat symbols with mismatched case (apple versus Apple), for example, as different symbols.

For symbolic data representing related values, such as {good, better, best} we must be careful to map them to consecutive integers and use a data representation that preserves this ranking information, such as a thermometer code or a simple linear scale.

For symbolic data that is of a continuous nature, this is more complex. For example, if we want to be sensitive to a difference of a single character in a string and note that it is similar to another string, the mapping to numeric values becomes more difficult.

Data Representation Impact on Training Time

Data representation is important. If wrong decisions are made regarding representation, it might be impossible for the neural network to learn the relationship we are trying to teach it. However, there is usually a set of possible data representations that are sufficient to train a network. In all cases, it is important to understand how your data representation decisions will affect both the training time for the neural network and the accuracy obtained.

In general, the more explicit the data representation, the easier it will be for the neural network to learn. For example, taking a discrete variable and using a one-of-N vector code will typically train the fastest. However, the cost is that you are adding N input units and a factor of N additional weights to the network. Again, in general, the larger the network in terms of processing units and connection weights, the less well it will generalize and the longer it will take to train. Taking the same discrete variable and assigning it to a single input unit, where each discrete value is represented by a difference of 0.1 in input magnitude, is certainly a more compact representation. A smaller neural network and one that generalizes better is likely to result. However, it will take the neural network a lot longer to adjust its weights from that single input unit in order to learn that a difference of a tenth is a significant difference that indicates a completely unique value for that input variable.

Managing Training Data Sets

A very important aspect of using neural networks for data mining and application development is how to manage your raw material, the historical data. The most common approach is to randomly divide the source data into two or more data sets. One subset of the data is used to train the neural network, and another subset is used to test the accuracy of the neural network. It is important to realize that the neural network never "sees" the test data while it is in training mode. That is, it never learns or adjusts its weights using the test data. Some people suggest that a third subset is required that is withheld even from the developer of the neural network model (not that anyone would cheat!). In this three-subset scenario, the developer uses a train-and-test data set to build the neural network model and a third party independently tests the network using the validation data. Figure 3.6 shows a typical use of data sets in neural network training. Multiple input files or databases are combined in the data preparation step to create the source data set. This data set is then split into a training set, a testing set, and a validation set.

There are some cases when this usual method is not appropriate. One is when the data is of a temporal or time-series nature. This data must be used in continuous temporal sequences in order to maintain the information it contains. Random selection from this data set would be catastrophic. In this case, it is typical to use data from a certain time period for training and the most recent data for the testing and/or validation phases.

Another case is when there is not sufficient data to allow random sampling to reasonably provide a representative sample of the input data population. In this case, statistical techniques might be required to ensure that both the training and test data sets contain representative samples of the data.

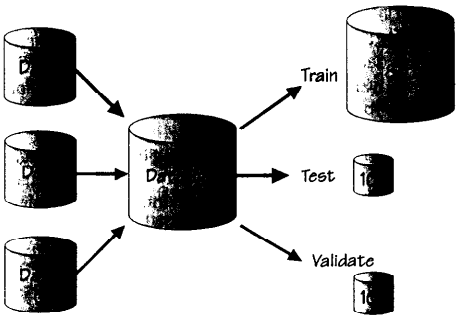


Figure 3.6 Data set management.

Data Quantity

Since data is the most important ingredient in data mining, ensuring that we have enough of our raw material is crucial. In most applications, the amount of data is at a premium, and several techniques must be used to squeeze the most utility out of it.

A rough rule of thumb with neural networks is that you need two data items for each connection in the neural network. So a back propagation network with 10 inputs, 5 hidden units, and 5 outputs would need approximately "2 * (10 * 5) + (5 * 5) = 150" training examples to be able to train accurately. In practice, many successful neural network applications have been developed using less data than this guideline suggests.

When using real data to train a neural network, it is typical to have 98% of the data representing "good" customers or "normal" conditions, and only have a small percentage of examples for the cases we really want to detect (i.e., "bad" customers or "abnormal" operating conditions). One technique to increase the percentage is to simply duplicate the number of training examples that contain the underrepresented class of training pattern. Another technique is to take the small number of test cases and modify them by injecting small amounts of random noise into the input values and then use these noisy inputs as additional training cases. Another option is to create the additional training examples by hand.

Data Quality: Garbage In, Garbage Out

In addition to the management of the data, a major concern in neural network data mining is the quality of the data. Most databases contain incom-

plete and inaccurate data. Depending on the amount of data available, you might be able to simply ignore any obviously bad records. However, in many cases you will have limited data available, and so you will have to try to scrub the data by supplying values for missing fields. The most common techniques is to set the fields to the mean or median value if it is numeric or to the mode of a discrete variable.

As in more traditional statistical analysis, outliers are a concern. A single record with a value one or two orders of magnitude larger or smaller than the rest of the data set can severely impact the performance of a neural network model (Simoudis, Levezey, Kerber 1995). A cursory scan of the range of each variable or a simple scatter plot can usually identify extreme cases when this occurs.

Neural network data mining, as the name implies, is highly dependent on the quality and quantity of data. If ever there was a system where GIGO was the rule (garbage in, garbage out), neural networks is it. They are highly forgiving of noisy and incomplete data, but they are only as good as the data they are trained with. See the paper by Cortes, Jackel, and Chiang (1995) for an excellent discussion of the effects of bad data on learning.

Summary

While the goal of data mining is to extract valuable information from data, it is an undeniable fact that the quality of the results relates directly to the quantity and quality of the data being mined. Data might be generated by transaction processing programs, might be entered into a database from existing manual paper-based processes, or it might even be generated by domain experts. In any case, it is important that missing and out of range values are scrubbed in the data preprocessing phase. The data might be stored in flat files or in databases, or both. Often data has to be selected and combined from several sources before data mining begins.

The majority of data used in data mining resides in relational database systems. I briefly discussed the relational data model of rows or tuples and columns or fields. The Structured Query Language (SQL) is the primary method for manipulating data in relational databases. When we get into the extremely large gigabyte and terabyte data sets, then performance of the relational database system becomes more important. Two primary methods for performance speedup used today are symmetrical multiprocessing (SMP) and the shared-nothing or loosely coupled systems. Both parallel architectures provide improved performance for data access, but the shared-nothing architecture is more scalable and will be required for the larger data warehouse systems.

Data representation and preprocessing are extremely important to neural network data mining. Experienced application consultants estimate that range from 50% to 75% of the development time is spent working with the data before it even sees a neural network. Thus having powerful data access tools, data cleansing, and preprocessing operations are essential to effective data mining.

The basic data types used for mining are categorical data, discrete numeric data, and continuous numeric data. Symbols can be turned into discrete numeric data through hashing functions or through symbol table maps. Numeric data, in turn, can be represented as coded data types such as one-of-N, thermometer, or regular binary codes. Continuous data can be scaled, thresholded, and discretized. Symbolic data can be mapped into different levels of abstraction by using taxonomies. Deciding which representation is best is usually a job of the domain expert who does the data preparation. Understanding the semantics of the data is crucial for selecting the appropriate data representations. The decisions concerning what data representation to use for the various variables can have a significant impact on the performance of the neural network, in terms of training time, processing time required to process transactions, and how well the neural network generalizes to inputs it has never seen before.

References

Cortes, C., L.D. Jackel, and W.P. Chiang. 1995. Limits on learning machine accuracy imposed by data quality. Proceedings of First International Conference on Knowledge Discovery and Data Mining, AAAI Press, pp. 57–62.

Date, C.J. 1990. *An introduction to database systems*, Volume 1 (5th. Edition), Addison Wesley.

Finkelstein R. 1995. Building a Fast and Reliable Data Warehousing Architecture Using the DB2 Family of Products, Performance Computing Inc.

Simoudis, E., B. Livezey, and R. Kerber 1995. Using RECON for data cleaning, Proceedings of First International Conference on Knowledge Discovery and Data Mining, AAAI Press, pp. 282–287.

Wesley, I, and D. Bradshaw. 1994. The future of the database market, Ovum Reports.

Chapter

4

Neural Network Models and Architectures

"A learning machine is any device whose actions are influenced by past experiences."
NILS NILSSON

There are many different types of neural network models or paradigms. At every neural network conference, literally hundreds of variations will be presented. Consequently, after you have decided to use neural networks to do data mining, your next decision is, "Which neural network model do I use?" This chapter explores the most popular neural network models in terms of the learning approaches, their basic connection topology, and their processing functions and capabilities.

The Basic Learning Paradigms

Perhaps the most useful way to categorize the different neural network models is by the basic learning paradigm or approach they use. The three main learning paradigms are supervised, unsupervised, and reinforcement. Supervised is the most common training paradigm used today to develop neural network classification and prediction applications, while unsupervised learning is often used for clustering and segmentation in data mining for decision support. Reinforcement learning, though used less frequently than the other methods today, has applications in optimization over time and adaptive control.

Before we get into the three types in detail, let's quickly relate these training paradigms to situations we are familiar with. Supervised learning is like trying to learn a new task from your mother. After every attempt you make to solve the problem, you have a very attentive teacher who gives you specific, immediate feedback on how well you did. Unsupervised learning is like being given a stack of documents, a file cabinet with unmarked file folders, and having to create a coherent filing scheme from scratch. Reinforcement learning is the most like real life. It's like having a job. You are given a sequence of tasks requiring decisions and at some point down the road, you are given a performance appraisal. You are told whether you are doing well or not, but it's up to you to figure out which decisions were right and which were wrong.

Supervised learning

The supervised learning paradigm is equivalent to "programming by example." In this approach, the neural network is given a problem or case, and it makes a prediction or classification (see Figure 4.1). At this point the supervisor says, "Oh, no, you did it wrong!" and indicates what the answer should be. Now we have something to go on. The learning algorithm takes the difference between the correct or desired output and the actual prediction the neural network made, and the algorithm uses that information to adjust the weights of the neural network so that next time, the prediction will be closer to the correct answer. Unlike people, who usually don't have to be shown the same problem over and over before they get the idea, neural networks are somewhat slow. They must be shown the examples tens, hundreds, or even thousands of times before they can accurately predict the correct answer to some complex problems.

Supervised learning is used when you have a database of examples that contain both problem statements and the answer. Now you might say, "What

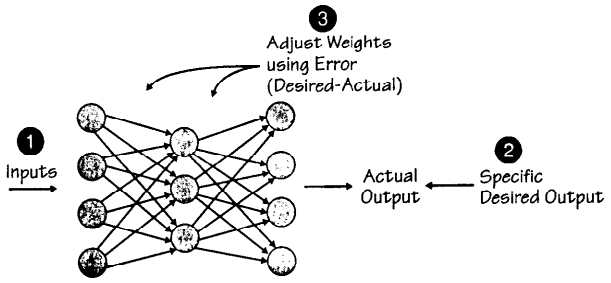


Figure 4.1 Supervised learning paradigm.

good is this, if I already know the answer? What is the neural network doing that I can't do already?" It can learn how the input and output are related. It can learn to look at problems the same way you do and make similar decisions. It can look (and learn) from hundreds or thousands of examples produced by the best performers in your organization. Moreover, it can learn to do this without programming it with instructions such as first do this . . . then do that . . . ad nauseam. In a relatively automated process, a neural network can turn a pile of data into a decision support system! Turning data into line of business applications is data mining at its most powerful best.

Supervised learning is a useful approach for training neural networks to perform classifications, function approximation or models, and time-series forecasting where the network is trained to predict outputs at some point in the future. It is especially useful in problems where data in the form of input/output examples is available, but no one knows the exact transformation for processing the input and producing the output. Despite the amazing things that can be learned from data using statistical and other mathematical analysis techniques, there are still many real-world problems that are highly nonlinear, have complex relationships between multiple variables, and for which the formal mathematical function is not known or cannot be easily derived. Another type of problem for which supervised neural networks are ideal is the case when the problem itself changes over time. If we are trying to control a manufacturing process that is susceptible to changes due to variable weather or machine tool wear, then a neural network can be used to model and adapt to these changing conditions.

Unsupervised learning

Unsupervised learning is used in cases where we have lots of data, but we don't know the answer. We don't know the answer, but we do know the question. If we don't know the question, we might as well quit right now. The question is, "How are these data related? What items are similar or different and in what way?" In effect, we want the neural network to look at the patterns of data and to cluster them so that similar patterns get put into the same cluster (see Figure 4.2). The neural network using unsupervised learning can perform this task with great precision. Of course, you have to represent the data correctly so that the neural network can discriminate between important differences in the data and unimportant ones. Once the partitioning is done, we'll need to do some analysis of the network to get a complete application (we'll talk about that later). This clustering approach is a quite useful function, as we will see.

Neural networks that are trained using unsupervised methods are called self-organizing because they receive no direction on what the desired or correct output should be. When presented with a series of input patterns, the output processing units self-organize by initially competing to recognize

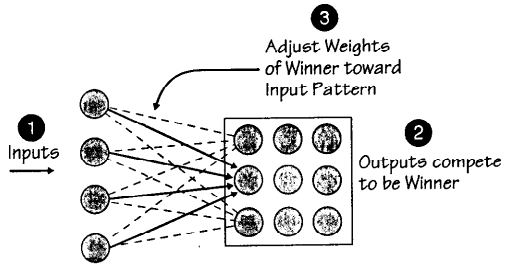


Figure 4.2 Unsupervised learning paradigm.

the pattern, and then cooperating to adjust their connection weights. Over time, an unsupervised network evolves so that each output unit is sensitive to and will recognize inputs from a specific portion of the input space.

Reinforcement learning

The third major neural network training paradigm is called *reinforcement learning*. In reinforcement learning, we have examples of the problem or case, but we do not have the exact answer, or at least not immediately (see Figure 4.3). For example, let's say we are playing a game, we have a board position, we make a move, the opponent makes a move, we make a move, etc. After 10 or 20 moves, we win or we lose. Now this is our reinforcement signal. We make a series of decisions, and only later do we find out whether they were right or wrong (how lifelike!). The neural network reinforcement learning approach allows very difficult temporal (time-dependent) problems to be solved. In some respects, reinforcement learning is the most true-to-life paradigm. For that reason, it is also one of the hardest to use to solve problems.

If exact feedback information is available, then supervised training will almost always be faster and more economical than reinforcement learning. However, when the problem involves some time sequential process or when the exact feedback is not available and only secondary signals are visible, then reinforcement learning is an appropriate technique to use. Researchers have shown that neural network models that use reinforcement learning are performing a mathematical optimization function similar to dynamic programming (Sutton 1988). This approach allows optimal strategies to be derived in economic and control applications.

Neural Network Topologies

The arrangement of neural processing units and their interconnections can have a profound impact on the processing capabilities of the neural net-

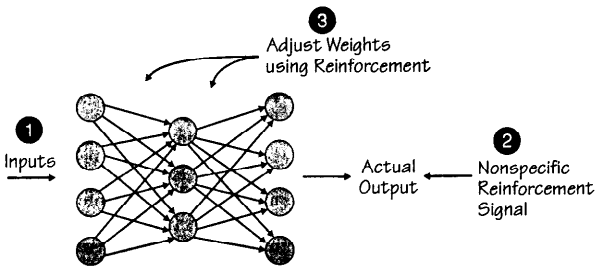


Figure 4.3 Reinforcement learning paradigm.

works. In general, all neural networks have some set of processing units that receive inputs from the outside world, which we refer to appropriately as the "input units." Many neural networks also have one or more layers of "hidden" processing units that receive inputs only from other processing units. A layer or "slab" of processing units receives a vector of data or the outputs of a previous layer of units and processes them in parallel. The set of processing units that represents the final result of the neural network computation is designated as the "output units." There are three major connection topologies that define how data flows between the input, hidden, and output processing units. These main categories—feedforward, limited recurrent, and fully recurrent networks—are described in detail in the next sections.

Feedforward networks

Feedforward networks are used in situations when we can bring all of the information to bear on a problem at once, and we can present it to the neural network. It is like a pop quiz, where the teacher walks in, writes a set of facts on the board, and says, "OK, tell me the answer." You must take the data, process it, and "jump to a conclusion." In this type of neural network, the data flows through the network in one direction, and the answer is based solely on the current set of inputs.

In Figure 4.4, we see a typical feedforward neural network topology. Data enters the neural network through the input units on the left. The input values are assigned to the input units as the unit activation values. The output values of the units are modulated by the connection weights, either being magnified if the connection weight is positive and greater than 1.0, or being diminished if the connection weight is between 0.0 and 1.0. If the connection weight is negative, the signal is magnified or diminished in the opposite direction.

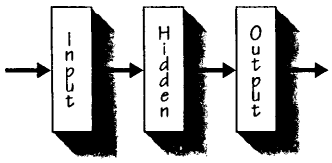


Figure 4.4 Feedforward neural networks.

Each processing unit combines all of the input signals coming into the unit along with a threshold value. This total input signal is then passed through an activation function to determine the actual output of the processing unit, which in turn becomes the input to another layer of units in a multilayer network. The most typical activation function used in neural networks is the S-shaped or sigmoid (also called the logistic) function. This function converts an input value to an output ranging from 0 to 1.0. The effect of the threshold weights is to shift the curve right or left, thereby making the output value higher or lower, depending on the sign of the threshold weight.

As shown in Figure 4.4, the data flows from the input layer through zero, one, or more succeeding hidden layers and then to the output layer. In most networks, the units from one layer are fully connected to the units in the next layer. However, this is not a requirement of feedforward neural networks. In some cases, especially when the neural network connections and weights are constructed from a rule or predicate form, there could be less connection weights than in a fully connected network. There are also techniques for pruning unnecessary weights from a neural network after it is trained. In general, the less weights there are, the faster the network will be able to process data and the better it will generalize to unseen inputs. It is important to remember that "feedforward" is a definition of connection topology and data flow. It does not imply any specific type of activation function or training paradigm.

Limited recurrent networks

Recurrent networks are used in situations when we have current information to give the network, but the sequence of inputs is important, and we need the neural network to somehow store a record of the prior inputs and factor them in with the current data to produce an answer. In recurrent networks, information about past inputs is fed back into and mixed with the inputs through recurrent or feedback connections for hidden or output units. In this way, the neural network contains a memory of the past inputs via the activations (see Figure 4.5).

Two major architectures for limited recurrent networks are widely used. Elman (1990) suggested allowing feedback from the hidden units to a set of additional inputs called context units. Earlier, Jordan (1986) described a network with feedback from the output units back to a set of context units. This

form of recurrence is a compromise between the simplicity of a feedforward network and the complexity of a fully recurrent neural network because it still allows the popular back propagation training algorithm (described in the following) to be used.

Fully recurrent networks

Fully recurrent networks, as their name suggests, provide two-way connections between all processors in the neural network. A subset of the units is designated as the input processors, and they are assigned or clamped to the specified input values. The data then flows to all adjacent connected units and circulates back and forth until the activation of the units stabilizes. Figure 4.6 shows the input units feeding into both the hidden units (if any) and the output units. The activations of the hidden and output units then are recomputed until the neural network stabilizes. At this point, the output values can be read from the output layer of processing units.

Fully recurrent networks are complex, dynamical systems, and they exhibit all of the power and instability associated with limit cycles and chaotic behavior of such systems. Unlike feedforward network variants, which have a deterministic time to produce an output value (based on the time for the data to flow through the network), fully recurrent networks can take an indeterminate amount of time.

In the best case, the neural network will reverberate a few times and quickly settle into a stable, minimal energy state. At this time, the output values can be read from the output units. In less optimal circumstances, the network might cycle quite a few times before it settles into an answer. In worst cases, the network will fall into a limit cycle, visiting the same set of answer states over and over without ever settling down. Another possibility is that the network will enter a chaotic pattern and never visit the same output state.

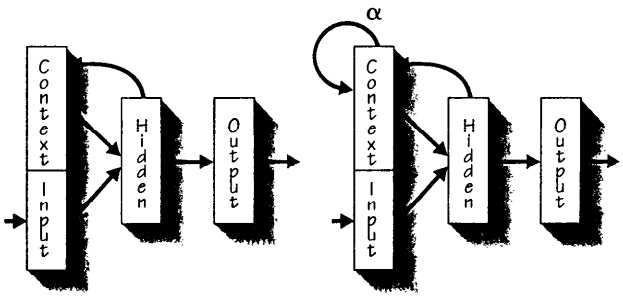


Figure 4.5 Partial recurrent neural networks.

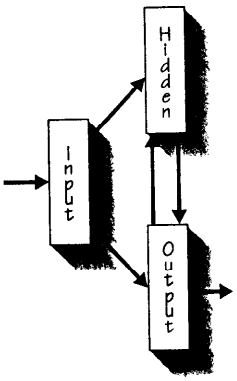


Figure 4.6 Fully recurrent neural networks.

By placing some constraints on the connection weights, we can ensure that the network will enter a stable state. The connections between units must be symmetrical. Fully recurrent networks are used primarily for optimization problems and as associative memories. A nice attribute with optimization problems is that depending on the time available, you can choose to get the recurrent network's current answer or wait a longer time for it to settle into a better one. This behavior is similar to the performance of people in certain tasks.

Neural Network Models

As mentioned earlier, the combination of topology, learning paradigm, and learning algorithm define a neural network model. There are a wide selection of popular neural network models. For data mining, perhaps the back propagation network and the Kohonen feature map are the most popular. However, there are many different types of neural networks in use. Some are optimized for fast training, others for fast recall of stored memories, others for computing the best possible answer regardless of training or recall time. But the best model for a given application or data mining function depends on the data and the function required.

The discussion that follows is intended to provide an intuitive understanding of the differences between the major types of neural networks. No details of the mathematics behind these models are provided. As mentioned in the preface, there are already a large number of textbooks that describe the math derivations in considerable detail. Wasserman's books provide a good introduction to neural network theory (1987, 1993), although his first book is getting a little dated. Hertz, Krogh, and Palmer (1993) give one of

the most comprehensive treatments of the literature and the mathematics associated with the models. If you crave a dose of calculus, these books will not disappoint. But in keeping with the goal of writing a book for an information processing and business audience, the following discussion uses words and graphics, even when a formula might clarify the point for some readers.

Back propagation networks

A back propagation neural network uses a feedforward topology, supervised learning, and the (what else) back propagation learning algorithm. This algorithm was responsible in large part for the reemergence of neural networks in the mid-1980s. Rumelhart, Hinton, and Williams (1986), working as part of the Parallel Distributed Processing group of neural network researchers, popularized the back propagation algorithm (which they called the *generalized delta rule*) with their clear, mathematical derivation and simple examples of the use of the algorithm. Their simple rebuttal of Minsky and Pappert's criticism of neural networks' inability to learn simple problems, such as the exclusive OR logic function, showed that a basic limitation of neural networks had been overcome.

Back propagation is a general-purpose learning algorithm. It is powerful but also expensive in terms of computational requirements for training. A back propagation network with a single hidden layer of processing elements can model any continuous function to any degree of accuracy (given enough processing elements in the hidden layer). There are literally hundreds of variations of back propagation in the neural network literature, and all claim to be superior to "basic" back propagation in one way or the other. Indeed, since back propagation is based on a relatively simple form of optimization known as *gradient descent*, mathematically astute observers soon proposed modifications using more powerful techniques such as conjugate gradient and Newton's methods (see Wasserman, 1993, for a discussion of some of the many variations of back propagation). However, "basic" back propagation is still the most widely used variant. Its two primary virtues are that it is simple and easy to understand, and it works for a wide range of problems.

The basic back propagation algorithm consists of three steps (see Figure 4.7). The input pattern is presented to the input layer of the network. These inputs are propagated through the network until they reach the output units. This forward pass produces the actual or predicted output pattern. Because back propagation is a supervised learning algorithm, the desired outputs are given as part of the training vector. The actual network outputs are subtracted from the desired outputs and an error signal is produced. This error signal is then the basis for the back propagation step, whereby the errors are passed back through the neural network by computing the contribution of each hidden processing unit and deriving the cor-

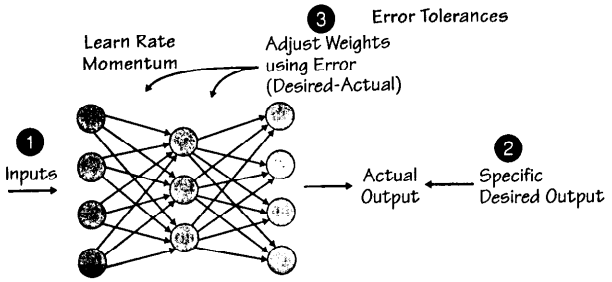


Figure 4.7 Back propagation networks.

responding adjustment needed to produce the correct output. The connection weights are then adjusted and the neural network has just "learned" from an experience.

As mentioned earlier, back propagation is a powerful and flexible tool for data modeling and analysis. Suppose you want to do linear regression. A back propagation network with no hidden units can be easily used to build a regression model relating multiple input parameters to multiple outputs or dependent variables. This type of back propagation network actually uses an algorithm called the *delta rule*, first proposed by Widrow and Hoff (1960).

Adding a single layer of hidden units turns the linear neural network into a nonlinear one, capable of performing multivariate logistic regression, but with some distinct advantages over the traditional statistical technique. Using a back propagation network to do logistic regression allows you to model multiple outputs at the same time. Confounding effects from multiple input parameters can be captured in a single back propagation network model.

Back propagation neural networks can be used for classification, modeling, and time-series forecasting. For classification problems, the input attributes are mapped to the desired classification categories. The training of the neural network amounts to setting up the correct set of discriminant functions to correctly classify the inputs. For building models or function approximation, the input attributes are mapped to the function output. This could be a single output such as a pricing model, or it could be complex models with multiple outputs such as trying to predict two or more functions at once.

Time-series forecasting can be accomplished with back propagation networks through a technique known as the "sliding window." Inputs for a set period of time can be presented to the neural network, and the desired output is the function at the next time period. Various time relations can be learned using this method. For example, the neural network could be trained to predict the next output in the sequence or the output three or four steps in the future. This technique was used by Sejnowski in his famous NetTalk experiment, where he taught a neural network to map text to phonemes for input to a speech synthesizer (1988).

Two major learning parameters are used to control the training process of a back propagation network. The learn rate is used to specify whether the neural network is going to make major adjustments after each learning trial or if it is only going to make minor adjustments. Momentum is used to control possible oscillations in the weights, which could be caused by alternately signed error signals. While most commercial back propagation tools provide anywhere from 1 to 10 or more parameters for you to set, these two will usually produce the most impact on the neural network training time and performance.

Kohonen feature maps

Kohonen feature maps are feedforward networks that use an unsupervised training algorithm, and through a process called self-organization, configure the output units into a topological or spatial map. Kohonen (1988) was one of the few researchers who continued working on neural networks and associative memory even after they lost their cachet as a research topic in the 1960s. His work was reevaluated during the late 1980s, and the utility of the self-organizing feature map was recognized. Kohonen has presented several enhancements to this model, including a supervised learning variant known as Learning Vector Quantization (LVQ).

A feature map neural network consists of two layers of processing units, an input layer fully connected to a competitive output layer. There are no hidden units. When an input pattern is presented to the feature map, the units in the output layer compete with each other for the right to be declared the winner. The winning output unit is typically the unit whose incoming connection weights are the closest to the input pattern (in terms of Euclidean distance). Thus the input is presented and each output unit computes its closeness or match score to the input pattern. The output that is deemed closest to the input pattern is declared the winner and so earns the right to have its connection weights adjusted. The connection weights are moved in the direction of the input pattern by a factor determined by a learning rate parameter. This is the basic nature of competitive neural networks.

The Kohonen feature map creates a topological mapping by adjusting not only the winner's weights, but also adjusting the weights of the adjacent output units in close proximity or in the neighborhood of the winner. So not only does the winner get adjusted, but the whole neighborhood of output units gets moved closer to the input pattern. Starting from ran-

domized weight values, the output units slowly align themselves such that when an input pattern is presented, a neighborhood of units responds to the input pattern. As training progresses, the size of the neighborhood radiating out from the winning unit is decreased. Initially large numbers of output units will be updated, and later on smaller and smaller numbers are updated until at the end of training only the winning unit is adjusted. Similarly, the learning rate will decrease as training progresses, and in some implementations, the learn rate decays with the distance from the winning output unit.

Looking at the feature map from the perspective of the connection weights, the Kohonen map has performed a process called vector quantization or code book generation in the engineering literature. The connection weights represent a typical or prototype input pattern for the subset of inputs that fall into that cluster. The process of taking a set of high dimensional data and reducing it to a set of clusters is called *segmentation*. The high-dimensional input space is reduced to a two-dimensional map. If the index of the winning output unit is used, it essentially partitions the input patterns into a set of categories or clusters.

From a data mining perspective, two sets of useful information are available from a trained feature map. Similar customers, products, or behaviors are automatically clustered together or segmented so that marketing messages can be targeted at homogeneous groups. The information in the connection weights of each cluster defines the typical attributes of an item that falls into that segment. This information lends itself to immediate use for evaluating what the clusters mean (see Figure 4.8). When combined with appropriate visualization tools and/or analysis of both the population and

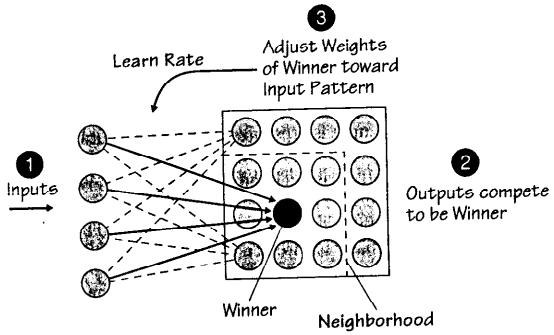


Figure 4.8 Kohonen self-organizing feature maps.

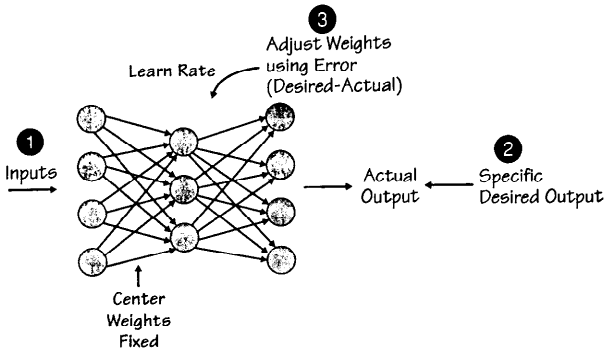


Figure 4.9 Radial basis function networks.

segment statistics, the makeup of the segments identified by the feature map can be analyzed and turned into valuable business intelligence.

Recurrent back propagation

Recurrent back propagation is, as the name suggests, a back propagation network with feedback or recurrent connections. Typically, the feedback is limited to either the hidden layer units or the output units. In either configuration, adding feedback from the activation of outputs from the prior pattern introduces a kind of memory to the process. Thus adding recurrent connections to a back propagation network enhances its ability to learn temporal sequences without fundamentally changing the training process. Recurrent back propagation networks will, in general, perform better than regular back propagation networks on time-series prediction problems.

Radial basis function

Radial basis function (RBF) networks are feedforward networks trained using a supervised training algorithm. They are typically configured with a single hidden layer of units whose activation function is selected from a class of functions called *basis functions* (see Figure 4.9). While similar to back propagation in many respects, radial basis function networks have several advantages. They usually train much faster than back propagation networks. They are less susceptible to problems with nonstationary inputs because of the behavior of the radial basis function hidden units. Radial basis function networks are similar to the probabilistic neural networks in many respects

(Wasserman 1993). Popularized by Moody and Darken (1989), radial basis function networks have proven to be a useful neural network architecture.

The major difference between radial basis function networks and back propagation networks is the behavior of the single hidden layer. Rather than using the sigmoidal or S-shaped activation function as in back propagation, the hidden units in RBF networks use a Gaussian or some other basis kernel function. Each hidden unit acts as a locally tuned processor that computes a score for the match between the input vector and its connection weights or centers. In effect, the basis units are highly specialized pattern detectors. The weights connecting the basis units to the outputs are used to take linear combinations of the hidden units to product the final classification or output.

Remember that in a back propagation network, all weights in all of the layers are adjusted at the same time. In radial basis function networks, however, the weights into the hidden layer basis units are usually set before the second layer of weights is adjusted. As the input moves away from the connection weights, the activation value falls off. This behavior leads to the use of the term "center" for the first-layer weights. These center weights can be computed using Kohonen feature maps, statistical methods such as K-Means clustering, or some other means. In any case, they are then used to set the areas of sensitivity for the RBF hidden units, which then remain fixed. Once the hidden layer weights are set, a second phase of training is used to adjust the output weights. This process typically uses the standard back propagation training rule.

In its simplest form, all hidden units in the RBF network have the same width or degree of sensitivity to inputs. However, in portions of the input space where there are few patterns, it is sometime desirable to have hidden units with a wide area of reception. Likewise, in portions of the input space, which are crowded, it might be desirable to have very highly tuned processors with narrow reception fields. Computing these individual widths increases the performance of the RBF network at the expense of a more complicated training process.

Adaptive resonance theory

Adaptive resonance theory (ART) networks are a family of recurrent networks that can be used for clustering. Based on the work of researcher Stephen Grossberg (1987), the ART models are designed to be biologically plausible. Input patterns are presented to the network, and an output unit is declared a winner in a process similar to the Kohonen feature maps. However, the feedback connections from the winner output encode the expected input pattern template (see Figure 4.10). If the actual input pattern does not match the expected connection weights to a sufficient degree, then the winner output is shut off, and the next closest output unit is declared as the winner. This process continues until one of the output unit's expectation is satisfied to within the required tolerance. If none of the output units wins, then a new output unit is committed with the initial expected pattern set to the current input pattern.

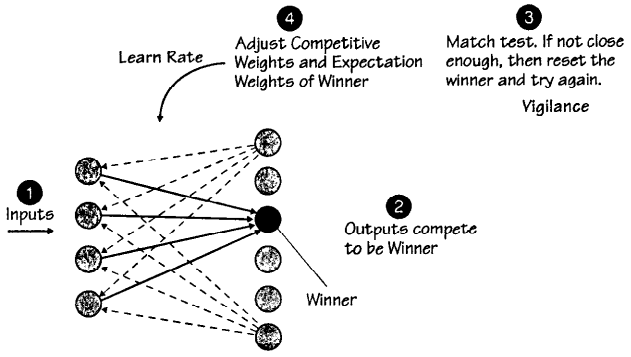


Figure 4.10 Adaptive resonance networks.

The ART family of networks has been expanded through the addition of fuzzy logic, which allows real-valued inputs, and through the ARTMAP architecture, which allows supervised training. The ARTMAP architecture uses back-to-back ART networks, one to classify the input patterns and one to encode the matching output patterns. The MAP part of ARTMAP is a field of units (or indexes, depending on the implementation) that serves as an index between the input ART network and the output ART network. While the details of the training algorithm are quite complex, the basic operation for recall is surprisingly simple. The input pattern is presented to the input ART network, which comes up with a winner output. This winner output is mapped to a corresponding output unit in the output ART network. The expected pattern is read out of the output ART network, which provides the overall output or prediction pattern.

Probabilistic neural networks

Probabilistic neural networks (PNN) feature a feedforward architecture and supervised training algorithm similar to back propagation (Specht 1990). Instead of adjusting the input layer weights using the generalized delta rule, each training input pattern is used as the connection weights to a new hidden unit. In effect, each input pattern is incorporated into the PNN architecture. This technique is extremely fast, since only one pass through the network is required to set the input connection weights. Additional passes might be used to adjust the output weights to fine-tune the network outputs.

Several researchers have recognized that adding a hidden unit for each input pattern might be overkill. Various clustering schemes have been proposed to cut down on the number of hidden units when input patterns are close in input space and can be represented by a single hidden unit. Probabilistic neural networks offer several advantages over back propagation networks (Wasserman 1993). Training is much faster, usually a single pass. Given enough input data, the PNN will converge to a Bayesian (optimum) classifier. Probabilistic neural networks allow true incremental learning where new training data can be added at any time without requiring retraining of the entire network. And because of the statistical basis for the PNN, it can give an indication of the amount of evidence it has for basing its decision.

Other neural network models

While I have presented the major neural network models, there are many more that are used by various people for specific problems. Generalized regression neural network (GRNN) is a relatively new model that subsumes the functionality of RBF and PNN networks (Caudill 1994). Hopfield networks and Boltzmann networks are fully recurrent networks that are used for optimization and constraint satisfaction problems, which are not usually considered as data mining applications of neural networks (Hertz, Krogh, and Palmer 1991).

Key Issues in Selecting Models and Architecture

Selecting which neural network model to use for a particular application is straightforward if you use the following process. First, select the function you want to perform. This can include clustering, classification, modeling, or time-series approximation. Then look at the input data you have to train the network. If the data is all binary, or if it contains real-valued inputs, that might disqualify some of the network architectures. Next you should determine how much data you have and how fast you need to train the network. This might suggest using probabilistic neural networks or radial basis function networks rather than a back propagation network. Table 4.1 can be used to aid in this selection process. Most commercial neural network tools should support at least one variant of these algorithms.

Our definition of architecture is the number of inputs, hidden, and output units. So in my view, you might select a back propagation model, but explore several different architectures having different numbers of hidden layers, and/or hidden units.

Data type and quantity

In some cases, whether the data is all binary or contains some real numbers might help determine which neural network model to use. The standard ART network (called ART 1) works only with binary data and is probably preferable to Kohonen maps for clustering if the data is all binary. If the input data has real values, then fuzzy ART or Kohonen maps should be used.

TABLE 4.1 Neural Network Models and Their Functions

Model	Training paradigm	Topology	Primary functions
Adaptive Resonance Theory	Unsupervised	Recurrent	Clustering
ARTMAP	Supervised	Recurrent	Classification
Back propagation	Supervised	Feedforward	Classification, modeling, time-series
Radial basis function networks	Supervised	Feedforward	Classification, modeling, time-series
Probabilistic neural networks	Supervised	Feedforward	Classification
Kohonen feature maps	Unsupervised	Feedforward	Clustering
Learning vector quantization	Supervised	Feedforward	Classification
Recurrent back propagation	Supervised	Limited recurrent	Modeling, time-series
Temporal difference learning	Reinforcement	Feedforward	Time-series

Training requirements: online or batch learning

In general, whenever we want online learning, then training speed becomes the overriding factor in determining which neural network model to use. Back propagation and recurrent back propagation train quite slowly and so are almost never used in real-time or online learning situations. ART and radial basis function networks, however, train quite fast, usually in a few passes over the data.

Functional requirements

Based on the function required, some models can be disqualified. For example, ART and Kohonen feature maps are clustering algorithms. They cannot be used for modeling or time-series forecasting. If you need to do clustering, then back propagation could be used, but it will be much slower training than using ART of Kohonen maps.

Summary

Neural networks are differentiated along three major axes: the training paradigm, the connection topology, and the learning algorithm. The most used training paradigm is supervised training, where an input pattern and a cor-

responding output pattern are presented to the neural network. The difference between the desired and actual outputs is used to adjust the neural network weights.

Unsupervised learning is used when we want to use the neural network to perform clustering or segmentation of the input data. Reinforcement learning is used in situations where the desired output is not known until some time later in the training sequence. These three training paradigms cover a wide range of application areas.

Neural networks are organized into layers of neural processing units. Most neural networks have a layer of input units, one or more layers of hidden units, and finally a layer of output units. Data can flow between the units in these layers in several ways. In feedforward networks, data comes in the input units, flows through any hidden layers, and then flows to the output units where the answer appears. Limited recurrent networks have some feedback connections, which are used to provide prior-state information, or a memory, to the neural network. This is most useful in problems involving time-dependent patterns. Fully recurrent networks have bidirectional connections between all processing units. The complex dynamics allow fully recurrent networks to model extremely nonlinear functions and to solve optimization and constraint satisfaction problems. However, they can be unstable and might oscillate or fall into limit cycles.

The most popular type of neural network is the back propagation network. It is a feedforward network and uses a supervised training method to adjust its weights. Kohonen feature maps, also known as self-organizing maps, are feedforward neural networks trained using unsupervised learning. Kohonen maps self-organize into topological maps where inputs that are close together in the input space are mapped onto adjacent output units in the neural network output layer. Recurrent back propagation is a hybrid network that uses limited recurrence and the standard supervised back propagation learning algorithm. Radial basis function networks are feedforward networks that are trained with supervised learning and have a single layer of hidden units that use a Gaussian basis function to compute the hidden layer activations. Adaptive resonance theory networks are recurrent networks that are trained using unsupervised learning. Probabilistic neural networks are supervised feedforward networks where a new hidden unit is allocated for each training input. There are many other neural network models that use different combinations of training paradigms, topologies, and learning algorithms.

The processing or data mining function required places definite constraints on which neural network models can be used for applications. Table 4.1 lists the major models and the functions that they can perform well, whether it is classification, clustering, modeling, or forecasting.

References

Caudill, M. 1994. Using neural networks, *AI Expert Magazine*, Miller-Freeman Publishers, Dec., pp. 47–52.

Dayhoff, J. 1990. *Neural network architectures: an introduction*, Van Nostrand Reinhold.

Elman, J.L. 1990. Finding structure in time, Cognitive Science 14, 179–211.

Grossberg, S. 1987. Competitive learning: from interactive activation to adaptive resonance, Cognitive Science 11, pp. 23–63.

Hertz, J., A. Krogh, R.G. Palmer. 1991. *Introduction to the theory of neural computation*, Addison Wesley.

Jordan, M.I. 1986. Attractor dynamics and parallelism in a connectionist sequential machine. In Proceedings of the Eight Annual Conference of the Cognitive Science Society (Amherst 1986), 531–546. Hillsdale: Erlbaum.

Kohonen, T. 1988. *Self-organization and associative memory* (2nd edition). New York: Springer-Verlag.

Moody, J. and C.J. Darken. 1989. Fast learning networks of locally tuned processing units, Neural Computation Vol. 1. No. 2, pp. 281–294.

Rumelhart, D.E., G.E. Hinton, and R.J Williams. 1986. Learning internal representations by error propagation, In *Parallel Distributed Processing*, Vol. 1, pp. 318–62. Cambridge, MA, MIT Press.

Sejnowski, T.J., and C.R. Rosenberg. 1987. Parallel networks that learn to pronounce English text. Complex Systems 1:145–68.

Specht, D.F. 1990. Probabilistic neural networks, Neural Networks 3(1): 109–118.

Sutton, R.S. 1988. Learning to predict by the methods of temporal differences, Machine Learning 3, pp. 9–44.

Wasserman, P.D. 1989. *Neural computing: theory and practice*, Van Nostrand Reinhold.

Wasserman, P.D. 1993. *Advanced methods in neural computing*, Van Nostrand Reinhold.

Widrow B. and M.E. Hoff. 1960. Adaptive switching circuits, 1960 IRE WESCON Convention Record, part 4, pp. 96–104., New York: Institute of Radio Engineers.

Chapter

5

Training and Testing Neural Networks

"Training is everything. The peach was once a bitter almond; cauliflower but cabbage with a college education." MARK TWAIN (1894)

"It's all to do with training: you can do a lot if you're properly trained." QUEEN ELIZABETH II

In this chapter, I explore the issues related to training a neural network to perform a specific processing function, whether it is classification, clustering, modeling, or time-series forecasting. I describe the most important parameters used in the most popular neural network models and how they can be used to control the training process. I talk about the usual training process for both supervised and unsupervised networks, and I also discuss the management of the training data and how it impacts the training process.

Once the data preparation is complete and the neural network model and architecture have been selected, the next step is to train the neural network. Because of the large variety in the types of neural networks, this process can be very dependent on the exact neural model and the function you are trying to train the neural network to perform. Some networks require only one pass through the data, while others might require hundreds or thousands. Some networks have only a few parameters to control the training process, while others might present a bewildering set of parameters to adjust. So when someone asks how long it takes to train a neural network, the answer is, "It depends." It depends on the neural network and its archi-

tecture, if it has ten, or hundreds of processing units, and if there are hundreds or thousands of training patterns. And, of course, it depends on your application and what your definition of "trained" is.

In most cases, we want to train the network with a subset of the examples and then test the network performance with another smaller subset. This train/test split is used to ensure that the neural network has learned the important aspects of the job it is being asked to do.

Most neural networks begin the training process with the connection weights initialized to small random values. The training control parameters are set, and the training data patterns are presented to the neural network, one after the other. As training progresses, the connection weights are adjusted, and we can monitor the performance of the network. In supervised training, we want to alternate between training and test data to ensure that we are getting good generalization by monitoring the average prediction errors. In unsupervised training, we usually want to visualize the arrangement of the output units. Table 5.1 shows the major learning parameters used for the various data mining functions.

At some point, it might become clear that the neural network is not able to learn the function we are trying to teach it. This is when the methodology provided in this chapter will become most useful. Trial and error might seem natural, but it can consume a lot of time and money (O'Sullivan 1993). A disciplined approach to iterative neural network development can be the difference between success and failure in a decision support or application development project.

TABLE 5.1 Learning Parameters for Neural Networks

Parameter	Models	Function
Learn rate	All	Controls the step size for weight adjustments. Decreases over time for some types of neural networks.
Momentum	Back progagation	Smooths the effects of weight adjustments over time.
Error tolerance	Back propagation	Specifies how close the output value must be to the desired value before the error is considered to be zero.
Activation function	All	Selects the activation function which is used by the neural processing unit. Most common is the sigmoid or logistic activation function, but hyperbolic tangent, signum or step function, and Gaussian are also used.
Vigilance	ART	Specifies how similar the input patterns must be to be classified as belonging to the same category.
Neighborhood	Kohenen maps	Defines the size or area of units surrounding the winner which get their weights updated. Neighborhood decreases over time.
Number of epochs	Kohonen maps, others	For networks which train for a fixed number of passes through the training data, determines the number of passes.

Defining Success: When Is the Neural Network Trained?

Once you have selected a neural network model, chosen the data representations, and are all ready to start training, the next decision is, "How do you know when the network is trained?" Depending on the type of neural network and on the function you are performing, the answer to this question will vary. If you are performing classification, then you want to monitor the number of correct and incorrect classifications the network makes when it is in testing mode. When clustering data, the training process is usually determined by the number of passes, or epochs, taken through the training data. If you are trying to build a model or time-series forecaster, then you probably want to minimize the prediction error. Regardless of the function required, once the neural network is trained and meets the specified accuracy, then the connection weights are "locked" so they cannot be adjusted. In the following sections, we explore the acceptance criteria used for training neural network to perform classification, clustering, modeling, and time-series forecasting.

Classification

The measure of success in a classification problem is the accuracy of the classifier, usually defined as the percentage of correct classifications. In some applications, getting an incorrect classification is worse than getting no classification at all. In these cases, a "don't know" or uncertain answer is desired. By selecting your data representation for the network outputs, you can obtain the behavior you require.

For example, let's say we want to classify customers into three types: poor, good, and excellent. We use a one-of-N code to represent our output and then train the network with an error tolerance of 0.1. We created an output filter that selects the highest output unit as the winning category. That is, if the outputs are 0.9, 0.4, and 0.3, we say that the winner is 0.9, and the corresponding category is poor. Note also that if the outputs are 0.9, 0.89, and 0.87, we would still classify the customer as poor, even though the network has high prediction values for good and excellent. Even if the outputs were 0.2, 0.19, and 0.1, the output classification would be that the customer was poor. One way to avoid this problem is to put a threshold limit on the output units before you perform the one-of-N code conversion. Usually we want the output value to be at least 0.6 before we say that the unit is ON.

If we put this threshold value in place, then we could add a fourth category, unknown or undecided, to represent the case where none of the network output units had a value above 0.6.

A confusion matrix is a text or graphic visualization that indicates where the classification errors are occurring. A text version lists the possible output categories and the corresponding percentages of correct and incorrect classifications (see Figure 5.1).

Clustering

The output of a clustering network is usually open to analysis by the user. In most cases, the training regimen is determined simply by the number of times the data is presented to the neural network, and by how fast the learning rate and the neighborhood decay. Kohonen feature maps, for example, might use a linear decay of the learn rate and a linear reduction in a square neighborhood function, or they might use an exponential decay in learn rate and a Gaussian or circular neighborhood function. The user would specify the number of epochs, or complete passes through the training data, and the initial learn rate. The network would train for the specified number of epochs and then stop. Figure 5.2 shows the output activations, represented by a Hinton diagram, of a Kohonen network used to cluster some data. Note that units close to the winner also have low activations, which are denoted by small boxes.

Adaptive resonance network training is controlled primarily by the vigilance training parameter and by the learn rate. The higher the vigilance, the more discriminating the network will be. ART networks are trained until a stable coding is obtained. An adaptive resonance network is considered stable when the training data goes through two complete passes, and each input pattern falls into the same output class as on the previous pass. Depending on the application, you might want to lock the ART network weights so that they will not be adjusted when the neural network is deployed. However, one of the advantages of the adaptive resonance theory model is that it can be used for online learning, where it can recognize novel input patterns and allocate new output categories when necessary. One

	Predicted Output Category		
	Category A	Category B	Category C
Category A	0.60	0.25	0.15
Category B	0.25	0.45	0.30
Category C	0.15	0.30	0.55

Figure 5.1 Confusion matrix for classification problems.

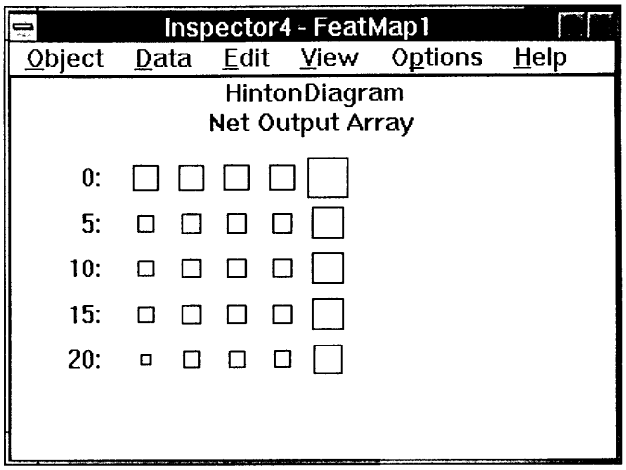


Figure 5.2 Hinton diagram for clustering applications.

point to remember is that ART networks are sensitive to the order of the training data. Thus there is no guarantee that specific input patterns will map to the same output category on consecutive training runs if the training data set is modified in any way.

Modeling

In modeling or regression problems, the usual error measure is the root mean square error. Remember, in modeling problems, we are usually trying to learn some function with multiple inputs and one or more dependent output variables. The average or mean squared error (MSE) or the root mean squared error (RMS) are good measures of the prediction accuracy. When training is just started and the neural network weights have been randomized, the RMS error is usually quite high. The expected behavior is that as the neural network is trained, the RMS error will gradually fall until it reaches a stable minimum. Figure 5.3 shows the RMS error for a single training run of a back propagation network. If the prediction error does not fall, or it begins oscillating up and down, there is a chance that the network has fallen into a local minima. In this case, you will have to reset (or randomize) the neural network weights and start again. If the neural network still does not converge, you might need to change some of the values of your training parameters, or

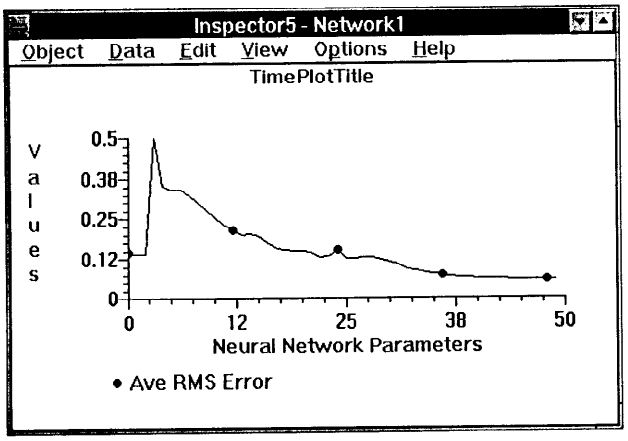


Figure 5.3 Time plot of RMS error during modeling training.

revisit some of your data representation and model architecture decisions (see "Network convergence issues" later in this chapter for more discussion of what to do when a neural network does not converge).

Care must be taken when using the RMS error as the only indicator of neural network performance. In some cases, the neural network learns that the best way to minimize the RMS error is to always output the mean value of the function. This behavior occurs primarily with functions whose output is symmetrical about some value. In this case, it is also useful to monitor the RMS error of the worst pattern. If the average RMS error for the training set is falling, but the RMS error of the worst pattern is growing larger, then it might be the case that the neural network is starting to average rather than fit the function.

Forecasting

Like the modeling applications, forecasting is a prediction problem, and so the root mean square error is used. Another good way to visualize the performance of a forecasting neural network is to use a time plot of the actual and desired network outputs (see Figure 5.4).

Time-series forecasting is a tricky modeling problem. There might be some underlying long-term trend that is also influenced by some cyclical factor such as the time of year (referred to as seasonality). On top of these trends there is usually a random component that causes variability and uncertainty in any prediction. The randomness can be statistically characterized by some probability distribution, or it could be, in fact, caused by a deterministic nonlinear process referred to as a chaotic time series (Rogers and Vemuri 1994).

People have been using statistics to predict linear trends with random components for years (McClave and Benson 1982). However, trying to forecast complex nonlinear or chaotic time series is another matter. Neural networks have shown themselves to be excellent tools for modeling complex time-series problems, especially recurrent neural networks, which are themselves nonlinear dynamic systems.

Controlling the Training Process with Learning Parameters

Once we have determined what network performance is required for our application, we can then start the training process. Once again, depending on the type of learning algorithm and neural network used, there are para-

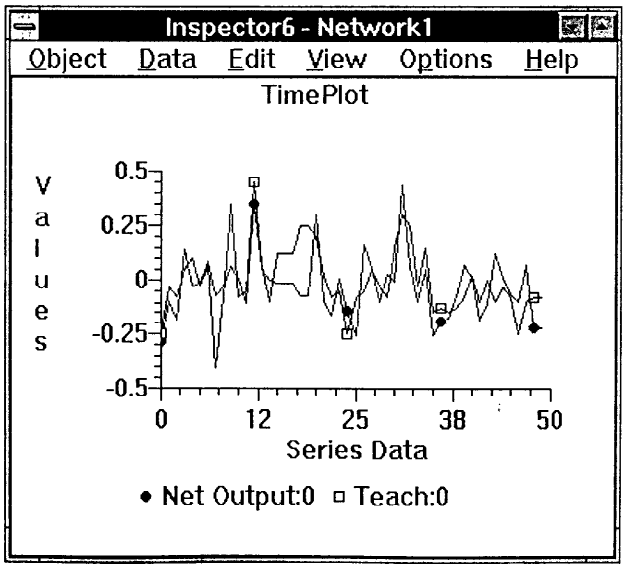


Figure 5.4 Time-series prediction plot.

meters that must be set in order to control the training process. In all of the neural network research papers that have been written, literally thousands of parameters have been defined. In the following section, I discuss the learning parameters you are most likely to encounter, based on the training paradigm, whether supervised or unsupervised.

Supervised training

In supervised training, we present a pattern to the neural network, it makes a prediction, and we compare the predicted output to the desired output. Thus we have explicit information about the performance of the network. The major parameters used in supervised training have to do with how the error is computed and how big a step we take when adjusting the connection weights in the direction of the desired output.

Learning rate. Almost all neural network models have a learning rate parameter associated with them. The learning rate is the knob you can turn to control whether you have a hyperactive student or a slow-and-steady learner. In a typical supervised training case, a pattern is presented to the neural network, it makes an incorrect prediction, and the difference between the desired output and the actual output is used to adjust the weights. The learning rate parameter controls the magnitude of the changes we make when adjusting the connection weights to move them toward the correct value for the current training pattern and desired output. That is, do we take a giant step toward the correct values (large learning rate) or a small step (small learning rate). You might ask, "Why use a small learning rate? Let's get this over with—fast." However, you must remember that you not only want the neural network to learn this pattern, but also the previous one and the next one. With a large learning rate, we are making large changes in the weights after each pattern is presented, maybe causing giant oscillations in their values. Also, remember that we don't necessarily want to get the exact answer on each training pattern. We want the neural network to learn the major features of the problem so that it can generalize to patterns that it has never seen before.

Often a lower learning rate actually gets us to the end of successful training by taking many small steps faster than if we try to take the fast track. Of course, this depends on the problem at hand. There is no harm in trying to train with a relatively large learn rate when you begin training, but even then it is usually beneficial to lower the learning rate over time as the training progresses. The idea is that you make large corrections early on, and then you fine-tune as you go along.

Momentum. Momentum is a training parameter that goes hand in hand with the learning rate. Its effect is to filter out high-frequency changes in the weight values, so that there is less chance that the neural network will start oscillating around a set of values. The momentum parameter causes the errors from previous training patterns to be averaged together over time and added to the current error. So if the error on a single pattern forces a large change in the direction of the neural network weights, this effect can be mitigated by averaging the errors from the previous training patterns.

This is especially true if the previous pattern errors were forcing the network weights in the opposite direction. Instead of using error information from a single training pattern (as would be the case when momentum is set to 0), the errors from the prior patterns are averaged in. The overall result is that the weights are less likely to be driven back and forth in alternate directions.

Error tolerance. Supervised training methods provide the neural network with input/output pairs in the training data. The target or desired outputs are specified in the range of the activation function of the output units. For example, standard back propagation networks, using the logistic activation function, require that the target outputs be in the range of 0.0 to 1.0. Some commercial neural network development tools use the hyperbolic tangent functions, which require outputs in the –1.0 to +1.0 range.

Most commercial tools will allow you to specify an error tolerance. This training parameter is used to control "how close is close enough." In many cases, an error tolerance of 0.1 is used. This means that if the target value is 1.0, a network output value above 0.9 (1.0 – 0.1 = 0.9) is within the tolerance, and the error is treated as 0.0. One of the main reasons for using an error tolerance is to avoid driving the network weights to extreme values. If you keep the output unit activations value in the range of 0.1 to 0.9 (with a tolerance of 0.1), then you are staying in the linear range of the logistic function. As you try to drive the output values up to 1.0 or down to 0.0, the net input (the sum of the input signals to the unit) must be quite large. Since the outputs of the other units will only be in the range of 0 to 1, this will usually require that the weights grow larger. Once the weights grow to large values and the output of the logistic function is above the knee of the S-shape curve, it is quite hard to change the output of the unit. This condition is often called "network paralysis" (Wasserman 1989).

Unsupervised learning

In unsupervised learning, the most important parameter is the selection for the number of outputs. This was described in the network architecture section, but it bears repeating. When a neural network is used for clustering or segmentation, the specification for the number of output units defines the granularity of the segmentations. If this is too large or small, then the results of the segmentation will be disappointing.

However, once the architecture is set, there are several learning parameters that can be used to control the segmentation process. Like supervised neural networks, most unsupervised neural networks also have a learn rate that is used to control the step size in the adjustment of the connection weights. Specific to unsupervised models are the neighborhood parameters for Kohonen maps and the vigilance parameter used in ART networks.

Neighborhood. When a Kohonen self-organizing feature map is used to cluster data, there are two popular methods for controlling which units get their weights changed. One is to use a square neighborhood function with a linear decrease in the learning rate. The other is to use a Gaussian shaped neighborhood with an exponential decay of the learning rate. Although the quality of the solutions is quite similar, the second approach leads to a simpler model, in terms of parameters that must be set by a user.

In this case, the major decision is how quickly (or slowly) you want the neural network to settle down. By selecting a large value for the number of epochs parameter, you are telling the neural network to take its time before it settles on the final clusters. In contrast, when you select a smaller value, you are telling the neural network to make a quick decision. The quick-decision approach is a statement about the training time and processing of the data.

The neighborhood in a Kohonen feature map defines the area around the winning unit, where the nonwinning units weights will also be modified. Typically, this parameter is set to a value roughly half the size of the maximum dimension of the output layer. So if the winning unit is in the center of a 6-by-6 output layer, and the neighborhood is 2, then not only the winning unit, but also the 8 units one step away, and the 14 units 2 steps away will also have their weights adjusted.

The neighborhood value is important in keeping the locality of the topographic maps created by the Kohonen maps. As training progresses, the neighborhood value or scope is decreased, so that at the end, only the winning unit's weights are modified. Remember, if you are using a Gaussian neighborhood function, this is taken care of automatically.

Vigilance. When using adaptive resonance theory (ART) networks, the number of outputs selected in the architecture is a statement about the maximum number of possible outputs, not necessarily how many outputs will actually be used. Adaptive resonance networks have a vigilance parameter that controls how picky the neural network is going to be when clustering the data (Carpenter and Grossberg 1988). Look at it this way: If the vigilance is low and two patterns are similar, then they will be clustered together. That is, for clustering purposes, the two patterns fall into the same output unit or category. However, if we raise the vigilance parameter, then the neural network is more discriminating when evaluating the differences between two patterns. What would have been "close-enough" with a vigilance of 0.5 might not be if the vigilance is 0.8. In this case, the network will say, "Hey, this is a totally new class of input patterns here, so I better commit a new output unit."

The control allowed by the vigilance parameter is one of the nicest features of the adaptive resonance networks. However, if the vigilance parameter is set too high, then the adaptive resonance network will allocate new output units for almost every input, and soon we will use up all of the output units. What do we do in this case? Well, either we can lower the vigilance parameter so the network isn't so picky, or we can change the network architecture by allocating more output units (classes).

Adaptive resonance networks train until they reach a stable state. This is when each input pattern gets classified into the same output unit on two consecutive passes. Be aware that adaptive resonance networks are sensitive to the order in which items are presented. For a given input data set, if the order is randomized, then different clustering could result.

Iterative Development Process

Despite all of your selections, it is quite possible that the first or second time that you try to train it, the neural network will not be able to meet your acceptance criteria. When this happens you are then in a troubleshooting mode. What can be wrong and how can you fix it?

Figure 5.5 shows the iterative nature of the neural network development process. The major steps are data selection and representation, neural network model selection, architecture specification, training parameter selection, and choosing an appropriate acceptance criteria. If any of these decisions are off the mark, the neural network might not be able to learn what you are trying to teach it. In the following sections, I describe the major decision points and the recovery options when things go wrong during training.

Network convergence issues

How do you know when you are in trouble when training a neural network model? The first hint is that it takes a long, long time for the network to train, and you are monitoring the classification accuracy or the prediction accuracy of the neural network. If you are plotting the RMS error, you will see that it falls quickly and then stays flat, or that it oscillates up and down. Either of these two conditions might mean that the network is trapped in a local minima, while the objective is to reach the global minima.

There are two primary ways around this problem. First, you can add some random noise to the neural network weights in order to try to break it free from the local minima. The other option is to reset the network weights

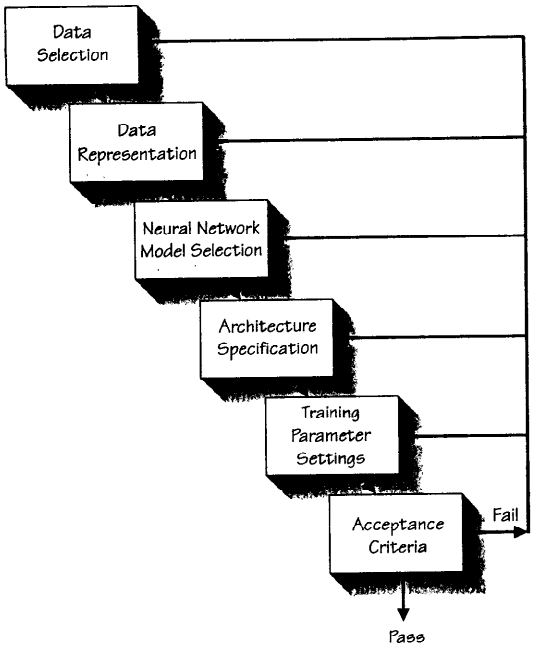


Figure 5.5 Iterative process for training neural networks.

to new random values and start training all over again. However, this might not be enough to get the neural network to converge on a solution. Any of the design decisions you made might be negatively impacting the ability of the neural network to learn the function you are trying to teach.

Model selection. It is sometimes best to revisit your major choices in the same order as your original decisions. Did you select an inappropriate neural network model for the function you are trying to perform? If so, then picking a neural network model that can perform the function is the solution. If not, then it is most likely a simple matter of adding more hidden units or another layer of hidden units. In practice, one layer of hidden units usually will suffice. Two layers are required only if you have added a large

number of hidden units and the network still has not converged. If you do not provide enough hidden units, the neural network will not have the computational power to learn some complex nonlinear functions.

Other factors besides the neural network architecture could be at work. Maybe the data has a strong temporal or time element embedded in it. Often a recurrent back propagation or a radial basis function network will perform better than regular back propagation. If the inputs are nonstationary, that is they change slowly over time, then radial basis function networks are definitely going to work best.

Data representation. If a neural network does not converge to a solution, and you are sure that your model architecture is appropriate for the problem, then the next thing to reevaluate is your data representation decisions. In some cases, a key input parameter is not being scaled or coded in a manner that lets the neural network learn its importance to the function at hand. One example is a continuous variable, which has a large range in the original domain and is scaled down to a 0 to 1 value for presentation to the neural network. Perhaps a thermometer coding with one unit for each magnitude of 10 is in order. This would change the representation of the input parameter from a single input to 5, 6, or 7, depending on the range of the value.

A more serious problem is when a key parameter is missing from the training data. In some ways, this is the most difficult problem to detect. You can easily spend much time playing around with the data representation trying to get the network to converge. Unfortunately, this is one area where experience is required to know what a normal training process feels like and what one that is doomed to failure feels like. This is also why it is important to have a domain expert involved who can provide ideas when things are not working. A domain expert might recognize that an important parameter is missing from the training data.

Model architectures. In some cases, we have done everything right, but the network just won't converge. It could be that the problem is just too complex for the architecture you have specified. By adding additional hidden units, and even another hidden layer, you are enhancing the computational abilities of the neural network. Each new connection weight is another free variable, which can be adjusted. That is why it is good practice to start out with an abundant supply of hidden units when you first start working on a problem. Once you are sure that the neural network can learn the function, you can start reducing the number of hidden units until the generalization performance meets your requirements. But beware. Too much of a good thing can be bad, too!

If some additional hidden units is good, is adding many more better? In most cases, no! Giving the neural network more hidden units (and the asso-

ciated connection weights) can actually make it too easy for the network. In some cases, the neural network will simply learn to memorize the training patterns. The neural network has optimized to the training set's particular patterns and has not extracted the important relationships in the data. You could have saved yourself time and money by just using a lookup table. The whole point is to get the neural network to detect key features in the data in order to generalize when presented with patterns it has not seen before. There is nothing worse than a fat, lazy neural network. By keeping the hidden layers as thin as possible, you usually get the best results.

Avoiding Overtraining

When training a neural network, it is important to understand when to stop. It is natural to think that if 100 epochs is good, then 1000 epochs will be much better. However, this intuitive idea of "more practice is better" doesn't hold with neural networks. If the same training patterns or examples are given to the neural network over and over, and the weights are adjusted to match the desired outputs, we are essentially telling the network to memorize the patterns, rather than to extract the essence of the relationships. What happens is that the neural network performs extremely well on the training data. However, when it is presented with patterns it hasn't seen before, it cannot generalize and does not perform well. What is the problem? It is called overtraining.

Overtraining a neural network is similar to when an athlete practices and practices for an event on his home court. When the actual competition starts and he or she is faced with an unfamiliar arena and circumstances, it might be impossible for him or her to react and perform at the same levels as during training.

It is important to remember that we are not trying to get the neural network to make the best predictions it can on the training data. We are trying to optimize its performance on the testing and validation data. Most commercial neural network tools provide the means to automatically switch between training and testing data. The idea is to check the network performance on the testing data while you are training.

Automating the Process

What has been described in the preceding sections is the manual process of building a neural network model. It requires some degree of skill and experience with neural networks and model building in order to be successful. Having to tweak many parameters and make somewhat arbitrary decisions concerning the neural network architecture does not seem like a great advantage to some application developers. Because of this, researchers have worked in a variety of ways to minimize these problems.

Perhaps the first attempt was to automate the selection of the appropriate number of hidden layers and hidden units in the neural network. This was approached in a number of ways: a priori attempts to compute the required architecture by looking at the data, building arbitrary large networks and then pruning out nodes and connections until the smallest network that could do the job is produced, and starting with a small network and then growing it up until it can perform the task appropriately.

Genetic algorithms are often used to optimize functions using parallel search methods based on the biological theory of natural selection (for a detailed discussion of genetic algorithms, see appendix C). If we view the selection of the number of hidden layers and hidden units as an optimization problem, genetic algorithms can be used to help find the optimum architecture.

The idea of pruning nodes and weights from neural networks in order to improve their generalization capabilities has been explored by several research groups (Sietsma and Dow 1988). A network with an arbitrarily large number of hidden units is created and trained to perform some processing function. Then the weights connected to a node are analyzed to see if they contribute to the accurate prediction of the output pattern. If the weights are extremely small, or if they do not impact the prediction error when they are removed, then that node and its weights are pruned or removed from the network. This process continues until the removal of any additional node causes a decrease in the performance on the test set.

Several researchers have also explored the opposite approach to pruning. That is, a small neural network is created and additional hidden nodes and weights are added incrementally. The network prediction error is monitored, and as long as performance on the test data is improving, additional hidden units are added. The cascade-correlation network (Fahlman 1989) allocates a whole set of potential new network nodes. These new nodes compete with each other and the one that reduces the prediction error the most is added to the network. Perhaps the highest level of automation of the neural network data mining process will come with the use of intelligent agents. In chapter 8, we will explore intelligent agents and data mining in detail.

Summary

Training a neural network is the hardest part of using neural networks for data mining. It is the equivalent step to sitting down and writing the algorithm (and coding and testing it) using a conventional programming language. This chapter presented a methodology for training and testing neural networks that, while not strictly cookbook, is certainly more structured than the "black-art" label usually attributed to the neural network development process. As with any project, the first task is to understand what

function you are trying to perform. From this, a likely candidate can be selected from the many available neural network models.

Once a neural network model has been selected, the next step is to define our measure of success, that is, what level the neural network must achieve in terms of classification or modeling accuracy before we call it "trained." For classification, the appropriate measure is the percentage of correct and incorrect classifications. For modeling and time-series forecasting, it is the mean squared error or the root mean squared error. Successful clustering is more subjective and often is dependent on the completion of cluster analysis after the neural network has self-organized.

There are several training parameters that are used to control the neural network development process. The most common parameter is the learn rate, which controls the size of the adjustments made to the connection weights. Supervised training algorithms also include a momentum term, which averages the weight changes over multiple patterns and serves to minimize oscillations in the weights. The error tolerance is used in supervised training to limit the tendency of neural networks to become paralyzed by extremely large weights, which result from trying to drive the outputs to their extreme values. In Kohonen maps, controlling the decay of the learn rate and the size and rate of decay in the neighborhood parameter are important. Adaptive resonance networks use the vigilance parameter to control the degree of similarity in input patterns that are mapped to the same class.

Neural network training is an iterative process, very similar in principle and technique to rapid application development or object-oriented prototyping. Several iterations are usually required before a successful training run is achieved. The principal steps in the process include data selection and data representation, which are sometimes considered to be part of the data preparation phase. Neural network model selection is next, followed by the definition of the architecture, which is the number of input, hidden, and output units. The training parameters then need to be set and the training data presented to the neural network. The appropriate error or performance measurements must be monitored to determine if the neural network is converging to a solution or if one of the previous steps needs to be revisited.

Overtraining is a degenerate case where a neural network is trained repeatedly on data such that it memorizes or overfits the function relating inputs to outputs. When new data is presented to an overtrained network, it will produce large prediction errors because it has not learned the fundamental relationships in the training data.

Researchers and commercial neural network tool vendors have made progress in automating the neural network development process. From model selection, to selecting the appropriate number of hidden units, to removing unnecessary input variables, to choosing the best data representation, techniques are being developed to simplify things. No matter how automated things become, your thorough understanding of the neural network development process will serve you well.

References

Carpenter, G.A., and S. Grossberg. 1988. The ART of adaptive pattern recognition by a self-organizing neural network, *IEEE Computer* (March), pp. 77–88.

Fahlman, S.E. and C. Lebiere 1990. The cascade-correlation learning architecture, in *Advances in Neural Information Processing Systems II* (Denver 1989), ed. D.S. Touretzky, 524-532. San Mateo: Morgan Kaufmann.

IBM Corp. 1994. Neural Network Utility User's Guide, SC41-0023.

Kohonen, T. 1988. *Self-organization and associative memory* (2nd ed.) New York: Springer Verlag.

McClave, J.T. and P.G. Benson 1982. *Statistics for business and economics*, Second Edition, San Francisco: Dellen.

O'Sullivan, J.W. 1993. Neural nets - a practical primer or. what I wanted I knew four years ago, Proceedings of Artificial Intelligence Applications on Wall Street, pp. 73–80.

Rumelhart, D.E., G.E. Hinton and R.J. Williams. 1986. Learning internal representations by error propagation, In *Parallel Distributed Processing*, Vol. 1, pp. 318–62. Cambridge, MA, MIT Press.

Sietsma, J. and R.J.F. Dow. 1988. Neural net pruning—why and how. In IEEE International Conference on Neural Networks (San Diego 1988), vol. I, pp. 325–333. New York: IEEE.

Vemuri, V.R. and R.D. Rogers. 1994. *Artificial neural networks: forecasting time series*, Los Alamitos: IEEE Computer Society Press.

Wasserman, P.D. 1989. *Neural computing: theory and practice*, Van Nostrand Reinhold.

Wasserman, P.D. 1993. *Advanced methods in neural computing*, Van Nostrand Reinhold.

Chapter

6

Analyzing Neural Networks for Decision Support

"It is a capital mistake to theorize before one has data." SIR ARTHUR CONAN DOYLE

When data mining is used for decision support applications, creating the neural network model is only the first part of the process. The next part, and the most important from a decision maker's perspective, is to find out what the neural network learned. In this chapter, I describe a set of postprocessing activities that are used to open up the neural network "black box" and transform the collection of network weights into a set of visualizations, rules, and parameter relationships that people can easily comprehend.

Discovering What the Network Learned

When using neural networks as models for transaction processing, the most important issue is whether the weights in the neural network accurately capture the classification, model, or forecast needed for the application. If we use credit files to create a neural network loan officer, then what matters is that we maximize our profit and minimize our losses. However, in decision support applications, what is important is not that the neural network was able to learn to discriminate between good and bad credit risks, but that the network was able to identify what factors are key in making this determination. In short, for decision support applications, we want to know what the neural network learned.

Unfortunately, this is one of the most difficult aspects of using neural networks. After all, what is a neural network but a collection of processing elements and connection weights? Fortunately, however, there are techniques for ferreting out this information from a trained neural network. One approach is to treat the neural network as a "black box," probe it with test inputs, and record the outputs. This is the input sensitivity approach. Another approach is to present the input data to the neural network and then generate a set of rules that describe the logical functions performed by the neural network based on inspections of its internal states and connection weights. A third approach is to represent the neural network visually, using a graphical representation so that the wonderful pattern recognition machine known as the human brain can contribute to the process.

The technique used to analyze the neural networks depends on the type of data mining function being performed. This is necessary because the type of information the neural network has learned is qualitatively different, based on the function it was trained to do. For example, if you are clustering customers for a market segmentation application, the output of the neural network is the identifier of the cluster that the customers fell into. At this point, statistical analysis of the attributes of the customers in each segment might be warranted, along with visualization techniques described in the following. Or we might want to view the connection weights flowing into each output unit (cluster) and analyze them to see what the neural network learned were the "prototypical" customers for that segment. We might then want to cluster the customers from a segment into additional clusters. This would allow us to drill down to a finer and finer level of details, as required.

In modeling and forecasting applications, the information discovered by the neural network is encoded in the connection weights. The most obvious use of the trained neural network is to use it to play what-ifs against the model. If a neural network has learned to model a function, even if you don't have a mathematical formula for the function, you can still learn a great deal about it by varying the input parameters and seeing what the effect is on the output. Let's say we built a model of the yield or return on investment for a set of products. If we input the data on a set of proposed development projects, we can use the estimates in our evaluation of their business cases. Or we can do a complete sensitivity analysis of the inputs to determine their relative importance to the return on investment.

Sensitivity Analysis

While there are many different types of information that might be gleaned using data mining with neural networks, perhaps the crucial thing to learn is which parameters are most important for a specific function. If you are modeling customer satisfaction, then it is important to know which aspect of your customer relationship has the most impact on the level of satisfaction.

If you have a fixed number of dollars to spend, should you spend it on a new waiting room for your customers, or should you hire another technician so the average wait is 10 minutes less? Determining the impact or effect of an input variable on the output of a model is called *sensitivity analysis*.

A neural network can be used to do sensitivity analysis in a variety of ways. One approach is to treat the network as a "black box." To determine the impact that a particular input variable has on the output, you need to hold the other inputs to some fixed value, such as their mean or median value, and vary only the input while you monitor the change in outputs. If you vary the input from its minimum to its maximum value and nothing happens to the output, then the input variable is not very important to the function being modeled. However, if the output changes considerably, then the input is certainly important because it affects the output. The trick in performing sensitivity analysis in this way is to repeat this process for each variable in a controlled manner so that you can tell the "relative" importance of each parameter. In this way, you have a ranking of the parameters according to their impact on the output value. For example, let's say we are modeling the price of a stock. We build a model and then perform input sensitivity analysis. When we look at each input variable, we might see that the day of the week is the most important predictor of what is going to happen to the price of the stock. We could then use this information to our advantage.

A more automated approach to performing sensitivity analysis with back propagation neural networks is to keep track of the error terms computed during the back propagation step. By computing the error all the way back to the input layer, we have a measure of the degree to which each input contributes to the output error. Looking at this another way, the input with the largest error has the largest impact on the output. By accumulating these errors over time and then normalizing them, we can compute the relative contribution of each input to the output errors. In effect, we have discovered the sensitivity of the function being modeled to changes in each input.

Rule Generation from Neural Networks

A common output of data mining or knowledge discovery algorithms is the transformation of the raw data into if-then rules. Standard inductive learning techniques such as decision trees can easily be used to generate such rule sets. One of the reasons this is so straightforward to do with decision trees is that each node in the tree is a binary condition or test. If value A is greater than B, then take one branch of the tree, else take the other. However, as has been pointed out before, having to define some arbitrary point as the dividing line between two sets of items will certainly lead to crisp answers, but not necessarily the correct ones.

One of the perennial criticisms of neural networks has been that they are a "black box," inscrutable, unable to explain their operation or how they ar-

rived at a certain decision. One very effective representation for knowledge, especially in classification problems, is to derive a set of rules from the raw data. One can then inspect this set of rules and try to determine which inputs are important. In this case, the neural network data mining process is transforming a set of data examples into a set of rules that tries to explain how the inputs cause the data to be partitioned into different classes.

Early on in the renaissance of neural networks, they were often compared and contrasted with rule-based expert systems. One point became clear. It is quite easy to map from a rule set to an equivalent neural network, but it is not so easy to go the other way. Why? Because the nonlinear decision elements in a neural network have more expressive power than the simple Boolean conditions used in most expert systems. This is not to say that neural networks somehow subsume the functionality that rule-based expert systems provide. In fact, I was one of the first to explore the relationships and synergy between neural networks and expert systems (Bigus 1990). Rather, the point is that neural networks, with their collection of real valued weights and nonlinear decision functions, are quite complex computing devices. Mapping their function onto a set of Boolean rules is challenging but certainly not impossible.

Gallant's (1988) pocket algorithm was one of the first attempts to map neural networks into rules. However, he used a neural network with Boolean decision elements, which simplifies the problem. Several researchers have tried to convert standard back propagation networks into rule sets. Narazaki, Shigaki, and Watanabe (1995) use a technique against trained networks with continuous inputs. They use a function analysis approach to identify regions in the input space that control the network output values. Other research in rule generation from neural networks include work by Kane and Milgram (1994) and Avner (1995).

Visualization

While neural networks are powerful pattern recognition machines in their own right, there is still nothing so powerful as the human ability to see and recognize patterns in two- and three-dimensional data. Consequently, visualization techniques play an important role in the analysis of the outputs of the data mining process. Actually, visualization is often used in the data preparation step to help in the selection of variables for use in the data mining application. In this section we look at a variety of graphical representations of data, of neural networks, and of the outputs of the data mining algorithms.

Standard graphics

Anyone who has used a spreadsheet, such as Lotus 1-2-3, is familiar with the wide range of charts that are used to view data. We take these graphic views to be somewhat standard visualization techniques. They include bar charts or histograms, scatter plots for viewing two-dimensional data, surface plots for viewing three-dimensional data, line plots for seeing a single variable change over time, and pie charts for viewing discrete variables.

The IBM Neural Network Utility (NNU) provides these visualizations through its Inspector windows (IBM 1994). In Figure 6.1, a histogram of the distribution of an input variable is shown. By using this view on each input parameter, we can see whether the data is badly skewed and also whether outliers are present. This information can be used in data cleansing and in deciding what data preprocessing functions are required.

Figure 6.2 illustrates an NNU scatter plot view, where the X axis is the average RMS error and the Y axis is the maximum RMS error. This graphic can be used to easily see whether the neural network is converging to a solution. Although basic, these standard types of graphics can be used to good effect.

Over the past decade, a set of views specific to neural networks has been developed. These provide information about the network architecture, processing unit's state, and connection weights. In the following section I describe these graphical views.

Neural network graphic

A useful technique for viewing the state of a neural network is a network graphic, such as that provided with NNU to show the neural processing

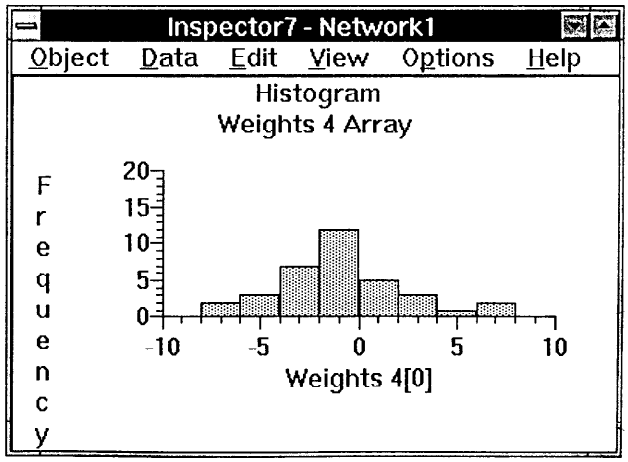


Figure 6.1 Histogram view.

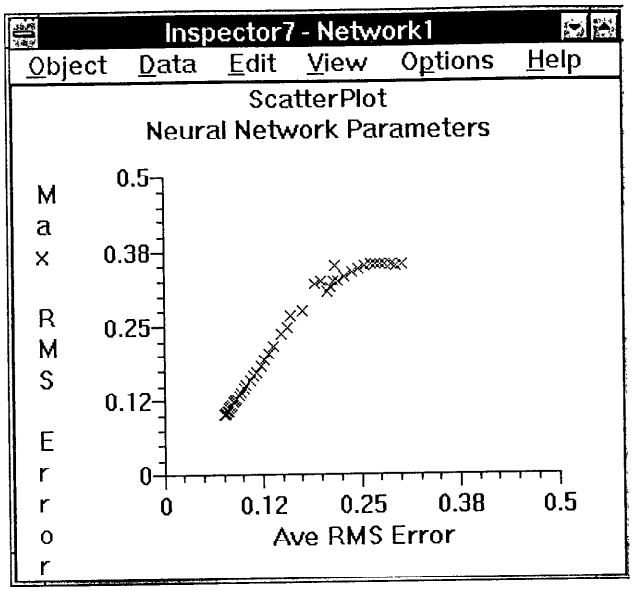


Figure 6.2 Scatter plot view.

units and their interconnections (see Figure 6.3). The topology of the network is apparent from the number of processing elements in each layer and the number of layers drawn in the graphic.

The activations of the processing units are depicted as circles, and their values are indicated through the use of colors. NNU allows thresholds to be set for on/off/undecided states, which are shown using red/blue/white colored circles.

Connection weights are drawn as lines connecting the processing units. The sign of the connection weight is indicated using color (red for positive weights, blue for negative valued weights). In NNU, thresholds can be set so that only weights whose magnitude is larger than the threshold will be shown. This is an easy way to determine which inputs have a large impact on the network output by seeing which input units have large connection weights into the hidden layer. Some neural network developers try to assign labels to the hidden units by watching the unit activations and correlating them with the value of certain inputs.

Hinton diagram

The connection weights contain information about the relative importance and correlation between input variables. In some sense, the absolute magnitude and the sign can be used as an indication of the importance of the input. One of the most popular visualization techniques used with neural networks is the *Hinton diagram*, named after researcher Geoffrey Hinton. A Hinton diagram is a collection of boxes whose size represents the relative magnitude of the connection weight and whose color depends on its sign, positive or negative. Hinton diagrams are an excellent way to visualize the weights in a neural network. Figure 6.4 shows the NNU Hinton diagram view of the weights of a back propagation network.

If we are performing some high-level function with a data mining tool, then we also need to view the results from the same high-level perspective. For example, if we are trying to cluster our customer base, we need tools to help us analyze those clusters and determine what they mean. Even data mining algorithms that output rules, which are supposedly easy to understand, can benefit from visualization techniques.

Clustering and segmentation visualization

IBM in Hursley, UK has developed a set of data mining and visualization tools to support its consulting practice in the insurance, finance, and travel

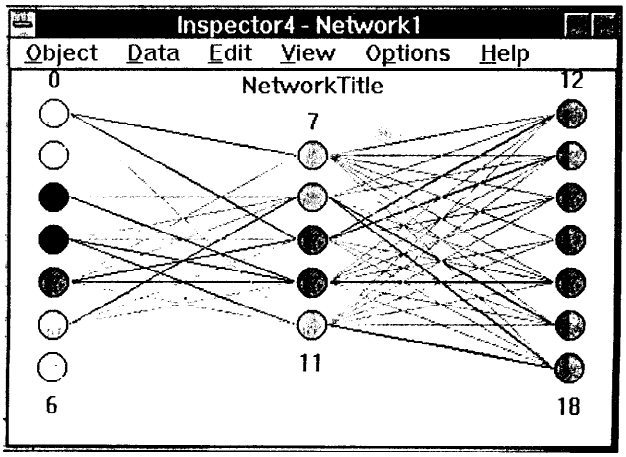


Figure 6.3 Network graphic.

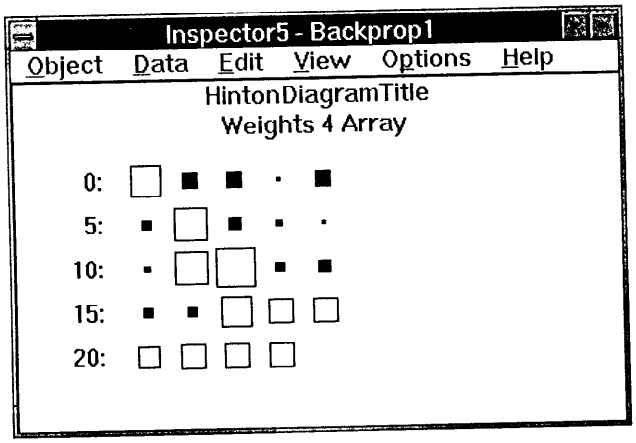


Figure 6.4 Hinton diagram graphic.

industries. Figure 6.5, shows a graphical view of the output of their clustering or segmentation algorithm. The statistical attributes of the members of the selected cluster are displayed as pie charts or bar charts against the population statistics.

Sifting through the Output Using Domain Knowledge

When a data mining algorithm is used to process data, it performs a transform, usually from some high-dimensional data into some more understandable form. In most cases, however, even though the data was transformed, the volume is still too large to be easily digested by an analyst. If we transform 1,000,000 records into 1000 rules or facts, then that is goodness. However, if someone then has to analyze the 1000 facts by hand, then that is not goodness. There might be only ten important facts in the 1000. How can we help the data mining tool provide only those facts that contain important information? One major way is to provide domain knowledge to guide the search. To do this requires objective functions that can be used to measure the value of the generated rules.

Summary

While neural networks are wonderful pattern recognition machines, they do not easily give up the secrets of what they learned. The "knowledge"

they gain through the data mining process is implicit in their structure and in the values of the connection weights. So while we might have transformed a million records into a thousand weights, the task still remains to translate that compressed information into a form that people can easily understand.

For clustering or segmentation, the output of the data mining process is the assignment of each input pattern into a cluster of other similar inputs. Thus the valuable information learned from this process is easily obtainable for use by a data analyst. Likewise, when neural networks are used for classification, modeling, or forecasting, the output of the neural network is, at face value, valuable information that can be used in decision support applications. No one would argue that it is not useful to know that for a given set of inputs, sales would increase 10%, or that by changing the amount of a chemical in a process that the yield would increase 5%. This sort of information is readily available and easily extracted from a trained neural network. But that assumes we are treating the neural network as a "black box" system. Some people are uncomfortable with this type of data mining.

Another approach is to use our "black box" neural network to determine the relative importance or sensitivity of the model to changes in the various inputs. This information is more understandable because it can be represented by rules, such as, "If variable X increases 5%, then output Y decreases 10%." Neural networks that are trained using the back propagation learning algorithm can automatically compute the relative importance or sensitivity of the inputs as a by-product of the training process. This infor-

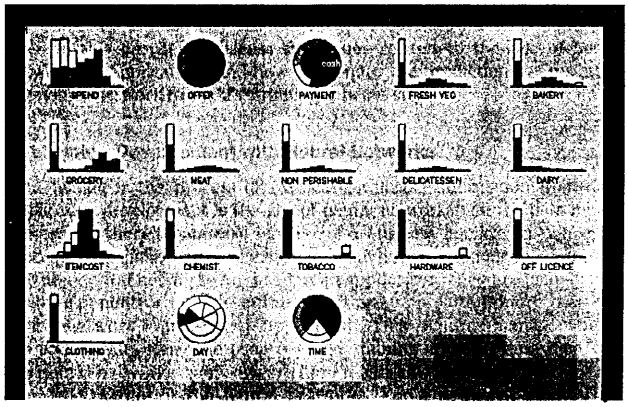


Figure 6.5 IBM Hursley segmentation visualization.

mation includes ranking the inputs by the relative contribution to the prediction error.

Using computer graphics for visualizing the information contained in a neural network is another way to overcome the black-box objection. Neural network graphics can show the sign and magnitude of connections and the activation values on the processing units. Specialized graphics such as Hinton diagrams clearly depict neural network connection weights and clustering results. Other more standard graphics such as line plots, scatter plots, histograms, and surface plots can be used to evaluate input data for data cleansing and preprocessing, and to analyze the accuracy of neural network predictions.

References

Avner, S. 1995. Discovery of comprehensible symbolic rules in a neural network, International symposium on Intelligence in Neural and Biological Systems, pp. 64–67.

Bigus, J. and K. Goolsbey. 1990. Combining neural networks and expert systems in a commercial environment," International Joint Conference on Neural Networks, Washington, D.C.

Routen T., and T. Collins. 1993. Visualization of A.I. techniques, Proceedings of Compugraphics, Third International Conference on Computational Graphics and Visualization Techniques, Alvor Portugal, Dec. 1993.

Narazaki, H., I. Shigaki, T. Watanabe. 1995. A method for extracting approximate rules from neural networks, Proceedings of 1995 IEEE International Conference on Fuzzy Systems, Vol. 4, pp. 1865–1870.

Gallant, S.I. 1988. Connectionist expert systems, *Communications of the ACM*, Vol. 31, No. 2, pp. 152–169.

Kane, R. and N. Milgram. 1994., Financial forecasting and rules extraction from trained networks, Proceedings of IEEE International Conference on Neural Networks, Vol. 5., pp. 3190–3195.

Chapter

7

Deploying Neural Network Applications

Data mining can be used for much more than just decision support applications. When neural networks are used as the data mining algorithm, the output of the process is a trained model. This model can be used to process transactions and perform clustering, classifications, and predictions on data in real time. There is no need to write programs and algorithms to process the inputs and produce the appropriate outputs. In a real sense, this function comes for free, as a by-product of the neural network data mining process. In this chapter, I discuss the issues related to the use of neural networks in applications and how to monitor the predictions or results of the neural network to see if retraining is necessary.

Application Development with Neural Networks

Data mining is the process of extracting valuable information from data. Application development is the use of neural networks and the data mining process for the ultimate goal of fielding a business application. While the processes are similar, application development has some unique requirements.

When neural networks are used in an application, they are usually only one component or module in the entire program. Often, more than one neural network is used in an application. This requires management of the source data files, the preprocessed training and test data files, and the neural networks themselves. There might also be scripts used for automated training of the neural networks. For a deployed application,

there is the question of maintenance and regression testing. While much of the application development process is the same as for a programming project, neural networks differ in several respects. For example, there is no coding phase with neural networks, assuming you are using a commercial neural network development tool. If you are "rolling your own" neural network algorithms, then you have both the problem of coding and testing the algorithm and the training and testing of any neural network models you build.

When developing applications with neural networks, you must keep in mind that it is an iterative process, much like application prototyping in an object-oriented or rapid application development (RAD) environment. In chapter 5, I detailed the iterative steps required for the neural network training and testing process. However, it bears repeating here. Neural networks do not lend themselves to traditional waterfall application development techniques. If you are forced to use a sequential methodology for the overall application development cycle, make sure you allow enough time for several iterations of model building in the neural network component.

Over the years, a recurring question from people has been, "How do you know the neural network is going to work when you deploy it?" The answer is quite simple. You test it! There is no reason why a neural network application could not be as stable and predictable as any other commercially developed application. Actually, many of the attributes of neural networks should make them more robust in a deployed application. Whether you are building C programs, Smalltalk programs, or neural networks, you must have a complete set of test cases. If you have a set of test cases that adequately tests a C module that transforms a set of inputs and returns an output value, then that same test suite could and should be used to validate the performance of a neural network, which is trained to do the same transformation of inputs to outputs. An acceptance test is an acceptance test. If the code module or the neural network performs adequately on a comprehensive test suite, which covers the space of possible inputs, then you ship it. And in most cases, the robust processing attributes of neural networks will react better than a program using Boolean conditions or rules when confronted with unexpected or ill-formed inputs.

Transaction Processing with Neural Networks

When a neural network is used as part of an operational application, the neural network is just another processing module, much like a subroutine or procedure. The neural network was trained to classify or cluster a set of inputs, or to model a function, or to make a prediction over time. Whatever the function, the trained neural network is the application module. All that is required to turn it into a transaction processor is to present input data to it and to retrieve the results from the output units.

In the simplest case, the application might consist of an input dialog or display panel that a data entry clerk uses to enter the data from a transaction. This data probably contains a mixture of categorical or symbolic data, some identifiers such as name or customer ID, and some numeric data. These inputs must be checked for validity, just like you would do for any other application. Once the input data is checked, it has to go through the same preprocessing steps you performed when you trained the neural network. If there are any computed fields or scaled data, then those operations must be done. Once the data is preprocessed, it is passed through the neural network. The output values are then read from the neural network, and any postprocessing function such as scaling, thresholding, or conversion to categorical values is performed. This postprocessed data is then displayed to the user as the result. This is the process that would go into automating the loan approval example in chapter 2.

You might ask, "Wait a minute, what about all this preprocessing and postprocessing stuff? Doesn't that require programming?" The answer is as usual, "It depends." If you are "rolling your own" neural network algorithm from scratch, then yes you would also have to provide the pre- and postprocessing code. If you are using a commercial tool, then you might still have to write the pre- and postprocessing code, since not all tools provide those functions. With the IBM Neural Network Utility, you can perform all of the pre- and postprocessing functions without any additional coding. Building an application consists of constructing the input/output dialog and writing a small program to call the NNU application programming interface (see appendix A for more information on NNU). Actually, NNU can be used to display the dialogs as well.

In more complicated cases, the input to the neural network might come from some other data processing program and is passed through the neural network. Then some additional processing of the neural network output might be performed by the application code. In this case, as in the interactive case, however, NNU can perform the pre- and postprocessing required by the neural network.

The Subroutine Metaphor

In most cases, the function performed by the neural network can be thought of as a simple subroutine or function in C or Pascal, or as a procedure in COBOL. The input data is presented to the neural network, gets processed by it, and the output is returned to the calling program. Of course, we know that often there is preprocessing to be done before the input data is passed to the neural network. Likewise, we often need to scale or transform the raw neural network output data into terms that a user can more easily understand. Figure 7.1 shows a typical use of neural networks when embedded in an application.

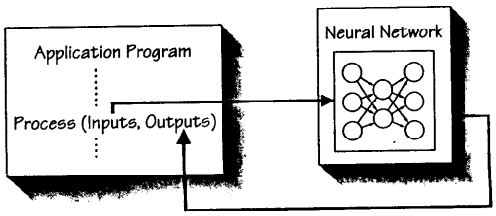


Figure 7.1 Neural networks as subroutines.

Lock It Up, or Online Learning?

Once you have trained the neural network and achieved acceptable performance using whatever error measure is appropriate for the function, you usually enter maintenance mode. In most cases, once a network is trained, you lock the weights, thereby ending any adjustments, and deploy the network in an application. In some rare instances, you might want to deploy the network while it is still in training mode so that it can learn from experience after it is deployed. The idea is that you are deploying a neural network that has been trained with typical data, and it will be further fine-tuned based on how it is actually going to be used. If online learning is a requirement, then you must carefully choose which neural network model you use. Some neural networks cannot easily be used with online learning. Adaptive resonance networks and probabilistic neural networks are two that can be trained online.

Deploying a generically trained neural network and then customizing or adapting it in the field is possible even without online learning. You could simply collect data from the installation over time and monitor the neural network's performance. When the results are out of some specified limit (outside of your original acceptance criteria, for example) then you can force a retraining of the neural network. Some commercial tools provide either application programming interfaces (APIs) or scripts, which can automatically retrain the neural network using the latest data. This accomplishes the same thing as online learning, but without the associated performance penalties, since the retraining can be done offline or at off-peak hours.

Maintaining the Network

Once deployed, the predictive accuracy or performance of the neural network must be monitored, just as you would monitor a new employee. The very same measures that were used to judge the human expert could be applied to the neural network expert. Our expectation is that the neural network would have fewer sick days.

A neural network that is trained and then deployed for a long period is like having an employee who is trained to perform a specific task and then does nothing to update his or her skills. You will periodically need to retrain the neural network using the latest data that captures the latest trends and give feedback to the neural network on its performance. If the neural network made an incorrect prediction or decision, make a new training example by taking the input data and adding the known correct answer as the desired output. In this way, you can modify the behavior of the neural network.

Monitoring Neural Network Performance

If you build an operational neural network application, you should also provide a mechanism for monitoring the neural network's accuracy over time. This can be as simple as gathering the information on the transactions that the neural network performed and the associated predictions or classifications it made. As soon as the outcome of those decisions are known, that information can be used as a new test or validation data set to see if the neural network still meets the original acceptance criteria. If the results fall below some specified level, then either an automatic or manual retraining cycle must be performed.

When Retraining Isn't Enough—Stale Model or Changed World?

If after monitoring a neural network model over time and retraining it, the predictive accuracy still does not meet the acceptance level, there is a good chance that something fundamental has changed in the problem domain. Either a new variable is now significant or the dynamics of the problem have changed in a major way. The first step would be to try adding additional hidden units to see if the network simply can no longer deal with the function it is trying to learn. If this does not improve the performance back to the original levels, then you must go back to step one and review which parameters might now be contributing to the function, and try to collect data on those parameters.

Summary

Application development with neural networks is the use of data mining techniques for the ultimate goal of fielding business applications. The concept of generating application code directly from data is still somewhat radical, but the productivity gains can be impressive. This technique can be used for problems where the conditions are changing, and the application program or rules would have to be constantly recoded in order to keep up. Simply retraining the neural network with the latest data could refresh the

application. In cases where there is no known algorithm for solving the problem, but where data exists, neural networks might be the only method for providing a solution.

A trained neural network can be regarded much like a subroutine in a standard programming language. The main application program passes a buffer or set of input data to the neural network, which processes those inputs and produces one or more outputs. These results can be used for further processing by the application. Sometimes preprocessing and postprocessing steps are required for the data passed into and retrieved from the neural network. Typical operations are scaling the input data and transforming symbolic data to numeric format. Commercial neural network development tools such as the IBM Neural Network Utility will automatically perform these processing steps for you.

Neural network development requires extensive testing, just like traditional program development. It is crucial that the test suite covers all likely input conditions, especially at the extremes. Once deployed, the performance of the neural network should be monitored, either manually or automatically. A degradation of neural network predictive accuracy indicates that retraining is needed or that a fundamental change has occurred in the function being modeled.

References

IBM Corp. 1994. Neural Network Utility: User's Guide, SC41-0223.
IBM Corp. 1994. Neural Network Utility: Programmer's Reference, SC41-0222.

Chapter

8

Intelligent Agents and Automated Data Mining

"The future of computing will be 100% driven by delegating to, rather than manipulating, computers." NICHOLAS NEGROPONTE

This chapter focuses on intelligent agents, a special type of software application that is rapidly gaining acceptance in advanced computing environments such as the Internet and the World Wide Web. Intelligent agents can be used to automate several aspects of the data mining process. In turn, the data mining functions presented in the preceding chapters can be used to provide the ultimate ability to an intelligent agent—the ability to learn or adapt to changes in its environment. In the following sections, I discuss intelligent agent technology in general and then describe how the technology might be used in conjunction with data mining.

What Are Intelligent Agents?

Like data mining, the term "intelligent agent" has become a catchall phrase used more for marketing software than to describe specific functions or capabilities of software components. In general, an intelligent agent consists of a sensing element that can receive events, a recognizer or classifier that determines which event occurred, a set of logic ranging from hard-coded programs to rule-based inferencing, and a mechanism for taking action in the world. Other attributes that are important include mobility and learning. An agent is mobile if it can navigate through a network and perform tasks on re-

mote machines. A learning agent adapts to the wants of its user and can automatically change its behavior in the face of environmental changes.

A primary aspect in the use of intelligent agents is the concept of delegation of authority (Maes 1994). In this case, the user is delegating the responsibility (and drudgery) of performing certain time-consuming computer operations to "smart" software. By virtue of this delegation, the user is free to move on to other tasks and even to disconnect from the computer while the software agent busily sees that the job completes. An important beneficial side-effect of this delegation is that the user does not even have to learn how to do the computer operation in the first place. In some respects, intelligent agents are a layer of software that provides the usability attributes that many novice users have needed from computers for years.

Types of Agents

The nature of intelligent agents is such that they are optimized to perform certain functions or tasks on behalf of a user (or even a computer system). IBM maps intelligent agents onto a graph with two axes, intelligence and agency (Aparicio et. al. 1995). This graph, shown in Figure 8.1, is quite use-

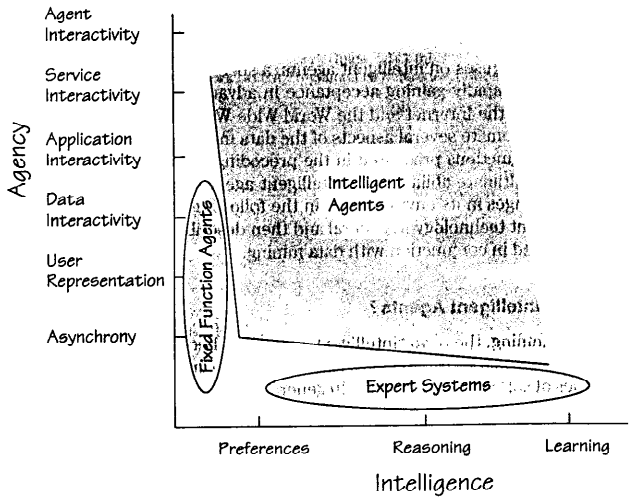


Figure 8.1 IBM intelligent agent graph.

ful for comparing different intelligent agent systems. On the intelligence axis, agents go from simply specifying user preferences, to active reasoning, through rule-based expert systems, all the way up to agents that can learn as they go. *Agency* is defined as the degree of autonomy and authority that the user permits the agent to have. At the very least, agents must run independently or asynchronously on the systems. In this respect, old PC-DOS terminate and stay resident (TSR) programs that intercept keystrokes are simple-minded agents. At the next level of agency, an intelligent agent should represent the user and interact with the operating system. More advanced agents should communicate with applications running on the system, and ultimately, interact with other intelligent agents.

Several categories or types of agents have been defined, based on their abilities and, more often, on the task they are designed to perform. For example, a booking agent might be designed to go out and find jobs for your rock band. You could have a fairly rudimentary agent that goes out and interacts with nightclub owner agents to see if you have a mutual open date, so it can schedule a job. This agent might have fixed logic and only be able to ask about a minimum price and open calendar dates. If you have more money to spend, you might get a more intelligent booking agent that has some knowledge embedded in it to negotiate with the nightclub owner agent to get your band the best possible price. Both of these agents would be called "booking agents." The difference is in their capabilities of doing that job. In the same way, other agents—such as information filtering agents, brokering agents, system agents, and user interface agents—are defined by the type of job they do, not necessarily how "intelligent" they are. It seems that with intelligent agents, as with people, knowledge and adaptability will differentiate the successful ones from the less effective ones. And just calling something an "intelligent agent" doesn't make it one. Even though the classification of intelligent agents is still evolving, the next sections describe the major categories recognized today.

Filtering agents

One of the cries of the information age is that we are drowning in a sea of data. There is simply more information generated each day than any one of us has the time to read through, much less comprehend. Filtering agents, as their name implies, act as a sieve that allows information of particular interest or relevance to us to get through, while stemming the flow of useless or nuisance information.

The filtering agents work in a variety of formats. Perhaps the most widely used format is one where the user provides a template or profile of the topics or subjects that are of interest. When presented with a list of documents in a database, a filtering agent scans the documents and ranks them based on how well their content matches the user's area of interest. Or the filter-

ing agent serves as an e-mail filter, automatically filing and disposing of messages based on their sender (the so-called bozo filter used with newsgroups and forums) or on the information content. Filtering agents could also interact with other agents, if necessary. For example, a filtering agent could send e-mail marked "Urgent!" to a Notifier or Alarm agent, which would inform you that an "Urgent!" message has arrived. IBM's IntelliAgent is an example of an e-mail filtering agent for use with Lotus Notes. It provides a graphical rule editor and a simple inference engine for automating routine mail handling.

A filtering agent with learning ability could automatically adapt the user's interest profile, refining it or broadening it, based on explicit feedback from the user, or by watching which articles or documents get saved and which get deleted.

Information agents

The counterpart to the filtering agent, which cuts down on information received, is the information agent, which goes out and actively finds information for the user. Used primarily on the Internet and World Wide Web, information agents can scan through online databases, document libraries, or through directories in search of documents that might be of interest to the user (Mobus and Aparicio 1994). As a research or intelligence gathering tool, information agents could provide an invaluable service, keeping the user apprised of any new developments in a field or of new web sites that contain information related to their area of interest.

User interface agents

When interacting with a desktop application, a user's skills might range from novice to expert. User interface agents are used to monitor the user interactions with the application and can control various aspects of that interaction, such as the level of prompting or the number of options available. For example, a new user typically needs lots of help and few choices. More experienced users, however, find that verbose help gets in the way, and they want to be able to easily access all features of a product. Coach (Selker 1994) is a user interface agent that monitors the user's interaction with a product and creates personalized help based on that interaction. Open Sesame, from Charles River Analytics, is a user interface agent for Apple Macintosh computers that "watches" the user perform tasks and then interrupts the user when it notices something, and asks if the user would like Open Sesame to automate that task. While somewhat intrusive, this type of user interface agent can be effective.

Maes (1994) defines four ways in which a user interface agent can learn. First, it can observe (through window system events) and imitate the user's behavior. Second, it can adapt based on explicit user feedback, whereby the user "grades" the agent on how well it performed an action. Third, the agent can be trained by the user through explicit examples. Fourth, the agent can learn through communications with other agents. All of these approaches imply supervised learning from a neural network perspective.

Office or work flow agents

An office management agent automates the kind of mundane tasks that take up so much time at the office. These tasks include scheduling meetings, sending faxes, holding design review meetings, and updating process documents. Some of these tasks are now falling under the umbrella of "work group" or "work flow" software because they deal with documents and calendars. One agent has been developed for scheduling meetings based on observed user preferences (Kozierok and Maes 1993). Whatever the name that ultimately gets attached to these agents, their role in automating common business functions will most likely produce some of the biggest efficiency gains of any intelligent agent applications.

System agents

System agents are software agents whose main job is to manage the operations of a computing system or data communications network. These agents monitor for device failures or system overloads and redirect work to other parts of the system in order to maintain a set level of performance and/or reliability. As computer installations become more distributed, the importance of system agents rises.

Network management agents have existed for years. Using Simple Network Management Protocol (SNMP), these agents reside on devices connected to the network and collect and report status information to the managing computer. However, these are considered "dumb" agents by today's standards. Intelligent "system" agents are involved not only in monitoring the status of resources on the computer network, but also they are active managers of those resources. System agents must be proactive, responding not only to specific events in the environment, but opportunistically taking the initiative to recognize situations that call for preemptive actions (Jennings and Wooldridge 1995).

Intelligent agents on a computer system could handle job scheduling to meet performance goals (Bigus 1994a). They also could be used to automatically adapt the allocation of system resources to various classes of jobs (Bigus 1994b). In this case, neural networks are used to model the relationships between the computer work load, available resources, and the resulting performance. Acting as an intelligent resource manager, a neural network controller could respond to changes in the work load by reallocat-

ing the computer system resources to balance the impact on the response times of various job classes. Similar approaches have been used to balance work load across distributed computer systems, and to ensure quality-of-service levels in data communication networks.

Brokering or commercial agents

In the real world, a broker acts as an intermediary to a buyer and seller. An agent that acts as a broker is a software program that takes a request from a buyer and searches for a set of possible sellers using the buyer's criteria for the item of interest. If and when potential sellers are found that can satisfy the buyer's request, then the broker agent can return the results to the user, who chooses a seller and manually executes the transaction. Or else, the agent can automatically execute the transaction on behalf of the user.

This form of electronic commerce is often presented as the ultimate form of intelligent agent application. After all, when we finally get to this point, agents will really have to be good. Both parties will have to have complete trust in their agent's ability to protect their interests and meet their criteria for a successful transaction. This commercial scenario also brings out many of the major issues that must be resolved before agent-based electronic commerce can become a reality. First, each agent must be opaque. That is, its internal knowledge about its ultimate goals and the strategy it will employ must not be visible in any way to the other agent. This implies a level of robustness and integrity not usually associated with commercial software. Second, its identity must be verifiable. When one agent tries to buy something from another agent, it must be able to verify that the other agent actually represents a legitimate seller, not someone who only wants to reach a deal so that they can get access to the buyer's credit card numbers.

There are other issues to be resolved before broker agents reach commercial viability. For example, when a broker agent is sent out onto the network, how does the user know that it will not end up on a server machine that is under the control of hackers who will disassemble that agent in order to discover its inner workings? Because of the financial aspects of their use, security will be the major issue for broker agents. The interaction with other broker agents also relies on resolving many issues of multiagent systems. These are described in more detail in the following section.

Multi-agent systems

While a single intelligent agent is interesting as an intermediary between a single user and a system, the truly exciting applications for intelligent agents usually involve the interaction of multiple agents. Broker agents and system agents, for example, depend on the existence of other intelligent agents in order to do their job. How will the these agents "talk" to each other? How will we ensure that they speak the same language? Will they have the same knowledge representation and belief systems? Unfortunately, the answers to some of these questions are years of research away.

Despite this, efforts are already underway to standardize an agent communication language (ACL). Gensereth and Ketchpel (1994) define ACL as consisting of three parts: its vocabulary, an inner language called Knowledge Interchange Format (KIF), and an outer language called Knowledge Query and Manipulation Language (KQML). In this ACL view of intelligent agents, something is a "real" agent only if it communicates through ACL.

While this effort might be premature, it is certain that some sort of common dialect will be needed for intelligent agents to communicate effectively. Agents might be based on standard object interfaces such as the Object Management Group's Common Object Request Broker (CORBA) and IBM's System Object Model (SOM). It is also possible that compound document architectures such as Lotus Notes, OpenDoc, or Microsoft's OLE will be the integration point for intelligent agents.

Agent Scripting Languages

One of the most basic attributes of agents is that they respond and react to events. This event might be a user input event such as a mouse click, or it might be a system event such as a notice that a piece of electronic mail has arrived. In either case, the event triggers the agent to evaluate the item and make a determination as to what action should be taken.

Interpreted languages such as BASIC, IBM's REXX, and Unix Shell scripts have been used for years as easy ways to quickly automate user tasks. The primary reason for using interpreted scripting languages for agents is their mobility. The source code can be sent around a network and run on any server system that has an interpreter. Another advantage of an interpreted language is that the scripts must run in a virtual machine. This provides some level of security against an agent with destructive intent from causing harm to the system it is running on. The Telescript language from General Magic is one example of a scripting language developed specifically for creating mobile agents (Wayner 1994). Java from Sun Microsystems is also gaining widespread popularity and support.

Adding Domain Knowledge through Rules

An agent that features hard-coded logic is, in most ways, just a piece of interpreted code. Just as no one would call an old BASIC program an intelligent agent, many people would classify this type of agent as being of the "dumb" variety. It is the addition of rules and an inference engine that moves an agent up to the next step in the intelligence hierarchy. These expert systems provide the domain knowledge an agent needs to perform a specific function for a user.

Traditional expert systems

The symbolic school of artificial intelligence made a real contribution to application development with the invention of the rule-based expert system. An expert system combines knowledge, usually represented as a set of if-then rules, an inference engine, which contains the program logic, and a working memory or workspace (Rich and Knight 1991).

The rule base contains the knowledge that applies to a certain problem or domain. The data used in the inferencing can be provided by the user, can be obtained from the system or applications, or can be generated by the rules themselves. The inferencing process works in one of two modes, forward chaining or backward chaining.

In forward chaining, the expert system starts with a set of data. The rule set is evaluated by testing the antecedents, the "if" part, of each rule. Depending on the type of inference engine used, one rule is selected to "fire" using a selection process called conflict resolution. Factors such as how specific the rule is (how many antecedent clauses it has), how recent the working memory values are, whether the rule has just fired, and even rule priorities are used to determine which rule is chosen. The consequent or action part of the rule is then performed. This action will usually change one or more variables in the working memory. Another round of conflict resolution is performed, resulting in another rule being chosen to fire. This process repeats until a state is reached where no rules can be selected to fire. The results of the inferencing process are then read out of the working memory.

An example of a forward chaining expert system application is a configurator. In this case, a set of rules is written that defines all of the constraints that must be met in order to have a valid configuration for a piece of equipment. The initial data is the set of options for the system that the customer wants to order. The inference engine runs through the rules whose consequents generate a list of the parts required to build this piece of equipment.

Backward chaining goes about the inferencing process in the opposite order from forward chaining. In backward chaining, a goal is specified, where the goal is a variable that appears in one or more consequents of rules. One of the rules is chosen and then the variables in the antecedent clauses are given values (bound) to make the rule true. Once these variable have values assigned to them, rules whose consequent clauses are now true are selected and, in turn, their antecedent clauses are made true. This process continues until a solution is found where all variables are bound to legal values. If a conflict arises at any point in the process, then the inference engine backtracks, or undoes some of the variable assignments, and searches in another path through the rules.

A standard backward chaining application is a diagnostic system. The goal is to find the determination of the problem. Medical diagnosis systems were one of the earliest applications of backward chaining expert systems. Here the goal is to find the diagnosis of a disease. As the inference engine back chains through the rule set, the doctor or nurse is asked a set of questions concerning the symptoms or measurements of the person being diagnosed. Only the information that the expert system (through the backward chaining process) deems important is asked of the user. If and when a logically consistent path from a specific diagnosis back to the available set of data on the patient exists, then the expert system stops and returns the diagnosis.

An advantage of both forward and backward chaining expert systems is that their inferencing process can be traced. By logging the sequence of rule firings and the state of the working memory, a user can query most expert systems as to why a piece of information is being requested and what line of inference the expert system is following in pursuit of the answer. This trace facility can be used to train new users in the problem-solving techniques used by experts in the field, or it can be used to give the experts a "warm fuzzy" that the rule-base has actually captured their knowledge and is providing valid solutions to the problems.

One disadvantage of rule-based expert systems is that all of the knowledge must be represented as if-then rules. Experience has shown that very few experts can explicitly define the set of rules they use (probably because they don't use a set of rules) to solve problems presented to them. This knowledge acquisition bottleneck was one of the reasons that data-centric methods such as neural networks have been used in place of traditional expert systems in applications. The rule bases generated by long, tedious interviews of knowledge engineers produced rule sets that defined what the knowledge engineer thought that the expert thought he used to solve the problem.

Fuzzy expert systems

Fuzzy expert systems are a combination of fuzzy logic and forward chaining rule systems (Kandel 1992). Instead of using binary or Boolean logic when evaluating rule clauses, a fuzzy expert system uses fuzzy logic operators. Rather than providing true/false results for each clause, a fuzzy inference engine produces a membership value that ranges from 0 to 1. So fuzzy expert systems are more like analog than digital computers, in the same way that neural networks are more analog than digital. See appendix B for an overview of fuzzy systems.

Another major difference in fuzzy expert systems compared to standard forward chaining expert systems is that there is no rule selection, or conflict resolution, process. In a fuzzy expert system, all rules are fired in parallel. That is, each rule is evaluated based on the current values of the working memory. Rather than having only one rule "fire" and performing its consequent clause, every rule fires, producing a collection of fuzzy sets.

These fuzzy sets are modified by the degree of membership or truth associated with their antecedent clauses, and then they are combined together. The fuzzy set output is then defuzzified into a crisp output value.

The real power in adding Boolean or fuzzy rule-based inferencing to intelligent agents is in the increased flexibility in representing domain knowledge and in the powerful problem-solving techniques that accompany the inference engines. Rather than having a fixed set of logic coded in a program, a rule-based expert system provides a mechanism to incrementally add new behaviors to the agent. After the intelligent agent is deployed, the designer could use feedback on its performance to enhance it through modifications to the agent rule base. Even better would be if the intelligent agent could adapt automatically using online learning techniques such as genetic algorithms or neural networks.

Adding Learning to Agents through Data Mining

While more difficult to implement, a learning agent would also obviously be much more valuable than a fixed-function agent. Learning provides the mechanism for an initially generic filtering agent to adapt and become a truly "personal" filtering agent. Such an agent would become an extension of the user.

Perhaps the ultimate goal of intelligent agents is to have them learn as they perform their tasks for the user. Depending on the technology used to implement the learning functions, learning could be done incrementally on an event by event basis, or the user actions or events could be saved in a log file for batch learning. There are advantages and disadvantages to each method.

Incremental learning is probably the best way to enhance an intelligent agent. After each task is completed by the agent, the experiences (in the form of event/action pairs) are integrated into the intelligent agent's knowledge base or model of the world. Theoretically, the agent will get better and better at doing its job, and the user benefits from this function. Adding learning to intelligent agents is like providing the user with automatic, free upgrades to software. However, the software agent is also subject to being distorted if its experiences with the system or other agents are illogical. The key is going to be to allow the agent to adapt while avoiding severe pathologies in its behavior over time.

One disadvantage to incremental learning is that the learning process might be expensive computationally. If you are using broker agents to trade commodities for your account, then you might want your agents to focus on performing the transactions you specify, and not spend their time trying to learn as they go. This performance penalty could be avoided if the learning is delayed until the agent is inactive.

Batch learning, where the agent only tries to integrate its experiences after it has collected a substantial amount of data, is really just a way of doing automated data mining. At set intervals of time or when sufficient data is collected, the agent would go into a "learning mode," where it examines the trace data, possibly integrating it with information concerning the results of its actions, and performs an automated data mining sequence. The result of this could be rules that could be used in an inference engine. Or more easily, the knowledge base could be a neural network model or a set of neural network models.

A disadvantage of batch learning is that, during the update phase, the agent is essentially unavailable for use. Depending on the type of agent and the environment it operates in, this might not be a problem. For example, taking a system management agent that monitors a nuclear plant offline for an hour to train it would not make sense.

Automating Data Mining with Intelligent Agents

In this chapter, I have talked about how intelligent agents can automate tasks for users, and the role data mining (learning) can play in enhancing agent abilities. In chapters 3 through 5, I presented the process by which we can perform data mining using neural networks. This process has several major steps, including data preparation, neural network model and architecture selection, training and testing the neural network, and finally analysis of the output and conversion into domain knowledge. In this section, I change my focus from agents to data mining, and I explore how intelligent agents can be used to automate the entire data mining process.

Data preparation

Reviewing the steps listed in chapter 3, data preparation involves data selection, data cleansing, data preprocessing, and data representation. With the use of intelligent agents we can automate several of these steps.

Data selection is a form of domain knowledge. The domain expert uses his or her knowledge about the problem and the available data to select data relevant to the data mining function being performed. One possibility for automating this step, especially in unstructured or ill-defined domains, is to perform automatic sensitivity analysis to determine which parameters should be used in learning. This lessens the dependency on having a domain expert available to examine the problem every time something changes in the environment.

Data cleansing could certainly be automated through the use of an intelligent agent with a rule base. When a record is added or updated in a relational database, a trigger could call the intelligent agent to examine the transaction data. The rules in its rule base would specify how to cleanse missing or invalid data.

Data preprocessing also requires domain knowledge. For example, there is no way to know that computed attributes or derived fields need to be gen-

erated. However, more standard preprocessing and data representation steps such as scaling, symbol mapping, and normalization, which are usually specified by the data mining expert, could be automated using rules and basic statistical information about the variables.

Model and architecture selection

In most data mining applications, the neural network model is selected based on the function required. This knowledge could be embedded in an intelligent agent using simple rules. However, the model architecture (the number of layers or hidden units) would have to be chosen based on the data representations used. An intelligent agent could use domain knowledge concerning neural network architectures and the training and testing process to control the search for the optimum architecture, possibly through the use of genetic algorithms (see the discussion in appendix C).

Training and testing

Automating the training and testing process is perhaps the easiest part of the job. The IBM Neural Network Utility, for example, provides a scripting component that allows the user to specify training and testing strategies to control the entire training cycle. This includes setting network parameters and monitoring network performance. However, there is still the issue of what training parameter values produce the neural network with the most accurate predictions or the best generalization. Fuzzy rule systems have been used to control the training of back propagation networks. So an intelligent agent with a fuzzy rule base as its domain knowledge could control the selection of learning parameters and possibly the whole training process.

Output analysis

Once the neural network model is created by the data mining process, the next step is discovering what it learned. While visualization is not a candidate for automation, sensitivity analysis and "black box" testing of the model could be automated. An intelligent agent could scan through the facts or rules generated by some data mining algorithms to identify items that contain valuable information.

Agent-directed data mining

The major advantage of using intelligent agents to automate the data mining task is that it enables data mining of online transactions. In some respects, this is similar to online analytical processing. However, again, the basic distinction between data mining and OLAP is that data mining involves discovery. By mining the data as it comes into the data warehouse or operational database, any crucial information or trend can be detected and handled in an automated way by the controlling business application program. If a filtering agent is the vehicle for doing this, then it can send an event to an alarm or notifier agent to alert a monitoring program or human decision maker. I have proposed this technique in customer briefings for several years, as a way to do automated decision support using neural networks and database triggers. Recently, Agrawal and Psaila (1995) proposed a similar approach, and dubbed it "active data mining."

Summary

Intelligent agents are a new class of software that can automatically perform computer functions for a user. Agents can be classified along two major dimensions: intelligence and agency. Simple agents have hard-coded control logic. Intermediate-level agents use rule-based inferencing to add additional capabilities and task-specific domain knowledge. The most advanced intelligent agents have the ability to learn and adapt as they perform work for the user and interact with the computer system and other agents. On the agency scale, running asynchronously or autonomously is a basic requirement, while interacting and representing users is a more sophisticated behavior. The ultimate goal is for agents to share information, cooperate to solve large problems, and interact with each other.

Intelligent agents are usually classified according to the type of task or function they perform. Thus we have filtering agents that shield users from junk messages, information agents that actively seek out documents of interest to the user, and work flow agents that automate everyday office or work group functions like scheduling meetings. More complex agents include system agents, which manage computer and data networks, and broker agents, which perform financial transactions on behalf of the user. Multiagent systems offer much promise but still need to overcome many technical hurdles to provide the security and communications mechanisms for multiple independent agents to work together to solve problems.

Rule-based expert systems provide a flexible method for adding domain knowledge to an agent, and furnish more powerful problem solving strategies than are usually possible with fixed-logic programs. Both traditional expert systems using Boolean logic and new fuzzy rule systems can be used in agents.

Learning capabilities offered by neural networks add the ultimate function to intelligent agents—the ability to personalize or customize according to the wants of the user. Most often, these learning or discovery processes are done against trace data from user actions or system events. Thus data mining plays a role in allowing agents to adapt to their environments.

Having intelligent agents around to monitor events, classify them, and take action provides a way to automate the data mining process. Agent-

directed data mining could use database triggers to signal when new data should be mined.

References

Agrawal R. and R. Psaila. 1995. Active data mining, Proceedings of the First International Conference on Knowledge Discovery and Data Mining, Menlo Park: AAAI Press, pp. 3–8.

Aparicio, M., B. Atkinson, J. Ciccarino, D. Gilbert, B. Grosov, P. O'Connor, D. Osisek, S. Pritko, R. Spagna, L. Wilson. 1995. IBM applications of intelligent agents, in Proceedings of Intelligent Agents and Their Business Applications - unicom, pp. 197–204.

Bigus. J.P. 1994. Applying neural networks to computer system performance tuning, IEEE International Conference on Neural Networks (Orlando), Vol. 1, pp. 2442–2447.

Bigus, J.P. 1994b. Computer system performance modeling using neural networks, Proceedings of World Congress on Neural Networks - San Diego, Vol. 4, INNS Press, pp. 510–515.

Genesereth, M.R. and S.P. Ketchpel. 1994. Software agents, *Communications of the ACM*, 7, pp. 48–53.

Jennings, N.R. and M. Wooldridge. 1995. Applying agent technology, Applied Artificial Intelligence, Vol. 9, No. 4, July–Aug., pp. 357–369. Kandel, A. 1992. Fuzzy expert systems, Boca Raton: CRC Press.

Kozierok, R. , P. Maes. 1993. A learning interface agent for scheduling meetings, Proceedings of ACM SIGCHI International Workshop on User Interfaces. ACM Press, N.Y. 1993, pp. 81–88.

Maes, P. 1994. Agents that reduce work and information overload, *Communications of the ACM*, 7, pp. 31–40.

McKie, S. 1995. Software agents: application intelligence goes undercover, *DBMS Magazine*, April, pp. 56–60.

Mitchell, T., R. Caruana, D. Freitag, H. McDermott, D. Zabowski. 1994. Experience with a learning personal assistant, *Communications of the ACM*, 7, pp. 81–91.

Mobus, G.E. and M. Aparicio. 1994. Foraging for information resources in cyberspace: intelligent foraging agent in a distributed network, Proceedings of CASCON94.

Rich E. and K. Knight. 1991. Artificial intelligence, 2nd. edition, New York: McGraw-Hill.

Selker, T. 1994. Coach: a teaching agent that learns, *Communications of the ACM*, 7, pp. 93–99.

Wayner, P. 1994. Agents away, *Byte* (May), pp. 113–118.

Part

2

Data Mining Application Case Studies

This section explores several data mining applications in detail. The business problems range from market segmentation to customer ranking, to sales forecasting and inventory control. Each problem can be studied independently of the others. However, each problem solution uses the neural network data mining methodology presented in Part 1. All of the applications are developed using the IBM Neural Network Utility. The discussion focuses on the data, the business problem, and the data mining process, not on the mechanics of using the tool to develop the application. While the Neural Network Utility is a capable commercial development tool, it is not the only one available. Whether your tool of choice is NNU or some other neural network product, the methodology should be applicable. These applications are used for illustration and so are necessarily simplified. While the number of data fields and source data might be larger in "real" applications, I have made every effort to make these applications realistic and useful.

Chapter

9

Market Segmentation

"The secret of business is to know something that nobody else knows."

ARISTOTLE ONASSIS

Problem Definition

One of the major challenges facing any business in any industry is to understand its customers. This understanding needs to occur at many levels. First, what products are customers most interested in and what features or services would they be willing to pay a premium for? Second, who are the customers? What does a "typical" customer look like? Is the customer 20 or 40 years old? Is the average income 20,000 dollars a year or 60,000 a year? Is the customer single or married with children? This information is useful for a variety of reasons. With information on product requirements, a manufacturer or retailer can ensure that it builds or stocks what its customers want. With information about the demographics of its customer set, the business can target sales promotions directly at the current or prospective customers who are most likely to buy from the company. These techniques are called *target marketing*, and they rely principally on the ability to segment or cluster the total market into more specialized niches that can be served with a higher degree of satisfaction (and profit). Taken to the extreme, target marketing gets to the point where there is a "market of one."

This kind of application is a classic example of data mining (Verity and Mitchell 1995). With each business transaction during each business day, raw data is collected concerning the customers and their purchases. Every time a customer is added to the customer database, more information is available

about the types of people who want to do business with your company. Also, with each purchase, the customer is providing information about his or her requirements and what he or she is willing to pay to satisfy those requirements. This is information that businesses have collected as normal bookkeeping logs, and this data can be mined to obtain strategic marketing information. Unfortunately, some businesses do not even realize the value of the data that they routinely dump to tape and put safely away in their computer backup vaults. Unlike fine wines, the value of that data is not going to necessarily increase with age.

What kind of customer information is required to perform data mining? Information concerning a customer's demographics (sex, age, marital status, etc.), economic status (salary, household income, debt ratio, homeowner, etc.) and geographic information (state, city, neighborhood, rural/urban/suburban, etc.) can all be used to define specific customer segments that share similar interests or product requirements.

Information regarding product purchases is easily obtained from point-of-sale systems. This information can then be matched to the customer who made the purchase, along with information details about the product. If follow-up surveys are done to determine the level of customer satisfaction with the product, a clear picture can be drawn of the relationships between a customer, the product, and the degree of satisfaction.

In the next sections, I work through an example of a target marketing application. I use the IBM Neural Network Utility (NNU) to do the data translation and the market segmentation with neural networks (See appendix A for details on NNU). For the analysis of the output results, I use Lotus 1-2-3. Several other commercial neural network tools and other spreadsheets could be used with equivalent results.

Data Selection

In this example, the business is a chain of department stores. The business management has decided to focus on five product categories in the coming year: sporting goods, home exercise equipment, home appliances, entertainment (electronics, music, and videos), and home furnishings. Rather than blanket the existing customer base with advertisements and catalogs featuring merchandise from these categories, the management would like to first understand more about the customers who buy these types of products. The first step will be to analyze the current customer set, and the second will be to buy mailing lists of new or prospective customers to target. An additional question is whether there is any strong correlation between customers who make purchases in one category and any of the others. This information will be used to determine the content and format of direct mail campaigns to the targeted customer set. The goal is to maximize the return on our marketing investment.

The business has information from three databases: a customer database, a product database, and a transaction database. The following data is available for the data mining project:

- Customer: customer name, customer ID, age, sex, marital status, address, income, homeowner
- Product: product name, SKU, price, cost, product category, quantity in stock, quantity on order
- Transactions: Customer name, SKU, date and time, amount

The first order of business is to aggregate the data so that we have the information we need to do the segmentation. Although we have detailed information on each purchase transaction each customer has made, what we really need is to scan the transaction database and determine whether the customer has made purchases from any of the five target categories. Using SQL, we join the product and transaction databases on the SKU, selecting the customer name, product category, and amount of the purchase. This step produces a table with customer name, product category, and amount of the purchase. We then process this table with a simple program such that when a record is read with a product category that matches one of our five targets, we add the amount of the purchase to the corresponding field in the new customer interest table. We also compute the total amount purchased by the customer. Our new table looks like this:

- Customer interests: customer name, sporting, exercise, appliances, entertainment, furniture, total

The next step is to select what data to use as the basis for the segmentation. This selection process involves two dimensions: first, which records or subset of the total customer base do we want to segment, and second, which customer attributes should we use? Now, in some cases, segmenting the entire customer base is desirable. However, in many cases, customers are selected based on some subset of the customer population. For example, we might want to segment the married customers if we are trying to select which children's furniture line to add to our stores. Or we might want to segment the customers who own their own homes. Just as SQL queries are used to drill down and examine finer and finer levels of detail in the data, so can segmentation be used to subset the customers into smaller and smaller groups, based on their shared attributes. In our case, we will start with the entire customer set and look at subsets only if required.

The selection of which fields to use is crucial for the segmentation to be successful. In this application, the customer name is not important for our analysis. We will use the customer age, sex, marital status, income, and whether the customer is a homeowner, combined with the customer inter-

TABLE 9.1 Selected Data for Customer Segmentation

Attribute	Logical Data Type	Values	Representation
Age	Continuous numeric	18–74	Scaled (0.0 to 1.0)
Sex	Categorical	Male, Female, Unknown	1, 0, 0.5
Marital Status	Categorical	Single, Married, Divorced, Unknown	1, 0, 0.5
Homeowner	Categorical	Yes, No, Unknown	1, 0, 0.5
Sporting Goods ($)	Continuous numeric	$0 to $1500	Scaled (0.0 to 1.0)
Exercise Equipment ($)	Continuous numeric	$0 to $2500	Scaled (0.0 to 1.0)
Home Appliances ($)	Continuous numeric	$0 to $5000	Scaled (0.0 to 1.0)
Electronics/Music ($)	Continuous numeric	$0 to $2500	Scaled (0.0 to 1.0)
Furniture ($)	Continuous numeric	$0 to $5000	Scaled (0.0 to 1.0)
Total Amount ($)	Continuous numeric	$0 to $15000	Scaled (0.0 to 1.0)

est information we derived on the five product categories. Table 9.1 shows the selected field names, logical data types, and the range of possible values.

Data Representation

Now that we have selected which fields will be used as inputs for the segmentation, the next step is to decide what data representation and preprocessing is required for each field.

Customer age is given as a continuous numeric field, ranging from 18 to 74. Unlike other data mining algorithms, which require continuous variables to be discretized or broken up into segments, neural networks can take continuous inputs without any problem. When clustering data with neural networks, it is standard practice to normalize the input data to a range of 0.0 to 1.0, so we will scale the input down to 0.0 to 1.0.

Sex is a categorical field containing either an "M" character for male, a "F" character for female, or "U" for unknown. We need to map the "M" and "F" values to numeric values, usually 0 and 1. What if we don't know the sex of the customer? How do we represent a "don't know" value? This is handled by mapping the "U" (or any other character) to a 0.5 value.

Marital status contains information in character form also. The letters are "S" for single, "M" for married, "D" for divorced, "U" for unknown. However, in this case, we don't really care whether the person is single or divorced. So in a preprocessing step, we map any "D" to "S" so that we have only three valid values, "S", "M", and "U". We map these to 0, 1, and 0.5 respectively. Note that this data representation preserves the semantics of this field by giving maximum separation to the single and married states, while the unknown value is placed in the middle of these extremes.

The income field has a range from $0 to $80,000. In this example, we take the simplest approach and scale this down to 0.0 to 1.0. In some applications, there might be a very large range of incomes (from thousands up to millions). If this is so, then taking the log of the income usually works better than simply scaling the data. The homeowner field is another categorical or discrete field. We will map the "Y" to a 1, the "N" to a 0, and "U" (unknown) to 0.5.

The purchase amounts for the five product categories are represented as continuous numeric values. The ranges of these values for the one thousand customers selected for this application are shown in Table 9.1. These values are all scaled linearly down to a range of 0.0 to 1.0.

The Neural Network Utility provides a data translation function that performs all of the required symbol mapping and numeric scaling operations. Figure 9.1 shows the NNU Translate template, which corresponds to our data representation specifications.

Model and Architecture Selection

Once we have decided what data to use and how to represent it to the neural network, the next step is to select which neural network model to use. Table 4.1 showed that Kohonen feature maps and adaptive resonance networks could be used for this kind of clustering. In this example, we will use a Kohonen feature map, which is more commonly used for data mining applications (Kohonen 1988).

Based on our data selection and representation, we know that we have 10 inputs to the neural network (age, sex, marital status, income, homeowner, and the amounts for the product categories). We decide to segment our customers into 4 groups, so we will specify 4 output units for the Kohonen map. Notice that this is an arbitrary decision. We could have just as easily decided to segment the customers into 6, 8, or even 16 groups.

Translate Editor - e:\customer.xlt

File Selected Edit View Windows Help

Name	Usage	Rep	Pre	Source	Table	Dest	Post
Sex	Input	1	None	Symbol	Table	Number	None
MaritalStatus	Input	1	None	Symbol	Table	Number	Scal
Age	Input	1	Scale	Number	None	Number	None
Income	Input	1	Scale	Number	None	Number	None
OwnHome	Input	1	None	Symbol	Table	Number	None
SportingGoods	Input	1	Scale	Number	None	Number	None
ExerciseEquipm	Input	1	Scale	Number	None	Number	None
HomeAppliances	Input	1	Scale	Number	None	Number	None
Entertainment/	Input	1	Scale	Number	None	Number	None
Furniture	Input	1	Scale	Number	None	Number	None
Totals	Ignore	1	Scale	Number	None	Number	None

Fields:11 In:10 Out:0 Ignore:1 | InBuf:11 Num:8 Sym:3 | OutBuf:10 Num:10 Sym:0

Figure 9.1 NNU translate template for segmentation application.

Segmenting Data with Neural Networks

There are three basic steps to doing segmentation with neural networks. We have to specify our source data, preprocess the data, and present this data to the neural network for segmentation. Figure 9.2 shows a Neural Network Utility application module set up for training the Kohonen feature map network. The NNU module editor shows three NNU objects connected to form our segmentation application. The Import object defines the data source, which is a comma-delimited text file. This source data is fed into a Translate Filter, which transforms the symbolic data and scales the numeric data as specified in the Translate template in Figure 9.1. The preprocessed data is then passed to the neural network. Our Network object contains an NNU feature map, which we created with 10 inputs and a 2-by-2 output layer.

Before we start training the network, we open several NNU Inspectors to view some of the data elements in the NNU application module so we can monitor the progress of the training. Inspector 1 shows the source input data. Inspector 2 shows the output of the Translate Filter object. Notice that the symbolic input fields have been transformed into numeric values, and that the numeric inputs have been scaled down to a range of 0.0 to 1.0. Inspector 3 is configured to display the output layer of our Kohonen network using a Hinton diagram view and a 2-by-2 layout. The Hinton diagram

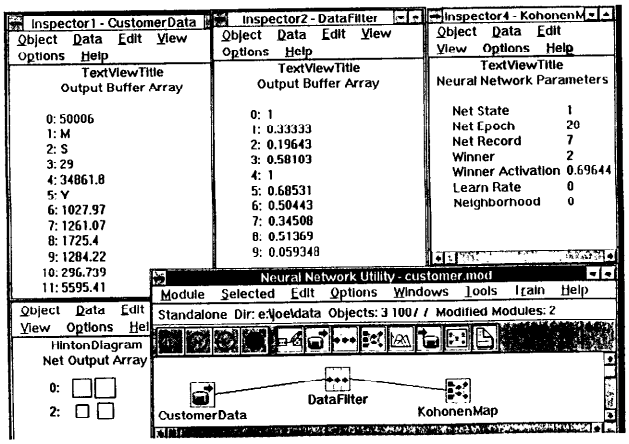


Figure 9.2 NNU module and Inspectors for segmentation application.

view is a graphic showing white (positive) and black (negative) boxes whose sizes vary with the magnitude of the output. Since we are monitoring the activations of the neural network, all of the values will be positive. Also, since the feature map uses a Euclidean measure of the closeness of the input pattern to the connection weights, those outputs with the smallest size are actually the closest to the inputs. Inspector 4 shows the major feature map parameters, which we monitor during the training process. These include the Net State, which tells us whether the neural network is in training or locked mode, the Winner, which identifies the winning output unit, and the Learn Rate.

The default NNU parameters for a feature map network start with a learn rate of 0.1 and decrease by 0.05 per epoch. Thus it takes 20 epochs until the network is trained. To train the network, we use the NNU Run function, which continuously reads patterns from the data set and passes them through the Translate Filter, where they are preprocessed and then presented to the neural network. As the neural network is trained, the Inspectors are continuously updated. As the module runs, we see the Learn Rate parameter in Inspector 4 decrease after each epoch or complete pass through the training data. We can also watch the Hinton diagram to see that there is some variety in which unit wins for each input pattern. When the learn rate goes down to 0.0, the Net State goes from training (0) to locked (1). We can either manually stop the training run when we see that the network is locked, or we could set an NNU breakpoint so that the Run will halt automatically when the Net State changes to 1.

After the Kohonen map has clustered the source data, we want to capture and analyze the results. To do this with NNU, we open an Inspector on the Network, select a Network Analysis view, and log the Net Record, the Winner, and the Winner Activation to a comma-delimited text file. The Net Record tells us which input record we are processing. The Winner identifies which cluster the customer fell into, while the Winner Activation indicates how close the input pattern is to the winning unit's weights. For our analysis, we load these three columns into a Lotus 1-2-3 spreadsheet, right next to the source data. Using the spreadsheet, we sort the data using the cluster number as the primary key and the customer ID as the secondary key. When this is done, we have split the customer base into four segments. We use Lotus macros to compare the makeup of the clusters and compare them to the overall population statistics.

In order to understand the information discovered by the segmentation, we need to first look at the makeup of the overall customer set. The average customer is 42 years old, has a yearly income of $35,000, is more likely to be a female (48% to 42%, 10% unknown), is probably married (50% to 35% single, 15% unknown), and has a 40% chance of being a homeowner. This group as a whole spends, on average, $500 on sporting goods, $1000 on ex-

ercise equipment, $1250 on appliances, $718 on entertainment, and approximately $1100 on furniture. While these averages are interesting, they are a conglomeration of people, and probably do not accurately represent any single customer.

The customer set was split by the Kohonen map into four groups, as shown in Figure 9.3. The largest group makes up 42.8% of the customers and has a very similar makeup to the customer set average. The one big difference is that this group spends almost twice the average on home appliances. The next largest group, 24.9%, is older (52 versus 42), spends almost half the average on sports and exercise equipment, less on appliances and entertainment, but spends over $500 more than the average customer on furniture. The next group, made up of 20.4% of the customer set, mimics the overall customer set with the exception that they spend only ¼ the amount on appliances. The smallest segment, consisting of 11.9%, averages just 26 years of age. They spend twice the average on sports equipment and entertainment, but less than the average on appliances and furniture. Thus our "average" customer has now been broken out into four sets of customers who exhibit very different consumer behaviors. Figure 9.4 illustrates the average spending on the five targeted product categories by the total customer population and the customers in each segment.

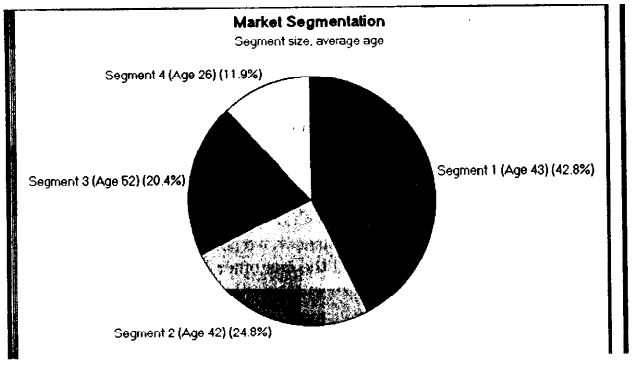


Figure 9.3 Segmentation results: segment size and average age.

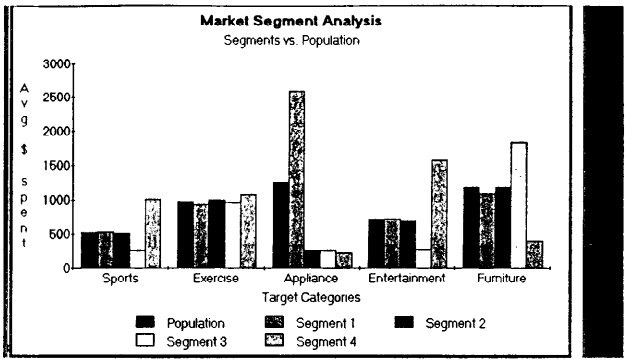


Figure 9.4 Market segmentation analysis.

Related Applications and Discussion

Now that we have our segmentation and analysis of the clusters, the challenge is to interpret this data and turn it into business decisions. We know we have four distinct groups of customers with different spending habits. From a marketing perspective, we can now target promotions at these groups. Customers who fell into the largest group will get promotions on appliances. The second largest group, which is older, will get furniture catalogs, not sporting goods or exercise equipment. The fourth group will be targeted with sporting goods and entertainment advertisements. This information basically "falls out" of the segmentation. It is useful for our tactical sales strategy. But what about strategic issues. Have we learned anything about our customer set?

An analysis of the statistical makeup of the four groups shows that sex is not a determining factor for which cluster they fell into, nor is marital status. The biggest contributing factors are age, income, to some extent, and most importantly whether or not they are homeowners. To find out more about these segments, we might need to cluster at a finer level, say into 9 groups or 16 groups, or we might want to take the customers from just one of the four segments and cluster them again. This would break that segment down into finer detail. A common approach is to first segment the customers into groups that have the behavior we want—for example, customers

who spend a lot on entertainment—and then build a ranking model for that specific group. We could then buy customer lists and run them through the ranking model to determine whom we should send our entertainment catalogs to. Using this approach, we would expect good results because we have selected customers who are most like our current "best" customers.

In doing segmentation or clustering, one of the most basic things to remember is that there is no "right" answer. If you select 10 outputs or clusters, the customers will be partitioned into 10 distinct groups, using a mix of their attributes. If you select 4 outputs, they will be divided into 4 distinct groups. If you run the same network on the same data 4 times, will the results be identical? Probably not. The initial weights of the neural network are randomly selected. So assuming that you have different initial weights, there is a good chance that the results will be slightly different for each training run. Is this a problem? Probably not. Provided there is not some major anomaly in the data, most of the customers will be classified into the same groups for each run. To illustrate this, we segmented the customer set again with a Kohonen map using different parameters for the neighborhood and number of epochs. The results were slightly different, with segments of 44%, 20.5%, 21.9%, and 13%. However, analysis of the makeup of the groups was consistent with the first segmentation results. What is important is that the different segments are identified and that they capture the same basic qualitative information.

The whole process of training the Kohonen maps took approximately one hour using the Neural Network Utility. This included setting up the translate template and training the neural networks.

On the first training run, the first unit was declared the winner on every pattern. That is, the data was not segmented at all, but was placed in the same class. To overcome this problem, we needed to use the "conscience" parameter provided by the NNU implementation of feature maps (DeSieno 1988). A well-known problem with competitive neural networks, like Kohonen maps, is that sometimes only some of the output units are positioned to "win" on input patterns. The result is that other outputs are left out in the cold; they do not ever get to win and therefore do not get their weights adjusted toward the part of the input space where the input data is. The conscience parameter lets units that are not strictly closest in Euclidean space to win and thereby get "in the game." On a test run of a 3-by-3 output Kohonen map, a similar thing happened. The four corner units won all of the inputs, and the other output units weren't used at all. This might or might not be a problem in this case, however, since the 4 outputs that won most likely discovered the same segments as the 2-by-2 network. When a conscience value was added, then all 9 of the outputs were active and the data was clustered into 9 segments.

The main idea used in this decision support application was to segment the customers and to analyze the makeup of those segments. The same basic approach could be used to analyze data on stores. For example, if you have 1000 stores in 40 states that you need to supply, you certainly don't want to have the exact same product mix in all of the stores. However, you also don't necessarily want to build 1000 inventory control models for each product carried by each store. A solution is to segment these stores based on the sales of products in the various categories. If you can cluster them into 16 segments and treat the stores that are grouped in the same cluster as being identical, then this greatly simplifies the logistical problems while still customizing the product mix to meet the stores sales patterns. In essence, we are using the clustering capabilities of neural networks to compress our data from the possibly hundreds of attributes down into the number of segments we desire.

Applying this approach to an inventory application, we could cluster our products based on their attributes. When a new product is introduced that we have no sales history about, it is sometimes difficult to predict what the appropriate inventory level and stock reorder threshold should be. By running the product attributes through a segmentation neural network, we could use the same demand curves as the other products that fall into that segment.

For some reason, many people use back propagation neural networks for these kinds of applications. However, Kohonen feature maps, and to some extent ART networks, have real advantages over back propagation when used for clustering. Their training is usually much faster than training a back propagation network, and they have fewer training parameters to set. The use of unsupervised learning to segment data is a powerful form of data mining, especially when it is combined with specialized analysis and visualization tools. (See Figures 1.6 and 6.5).

Summary

In this example application, we explored a common scenario for using data mining to do market segmentation. Our available data on customers, products, and customer purchases was aggregated and selected to give us a view of the customer attributes and their behavior with respect to five target product categories.

The first step in our application was to select exactly which data should be used for the segmentation and then to choose appropriate data representations. For clustering with neural networks, the data is usually normalized or scaled to a range of 0.0 to 1.0. We used a Kohonen feature map to cluster this data into four groups, and we did some high-level analysis of the customer segments. The IBM Neural Network Utility was used to do the data preprocessing and the clustering. Four distinct groups of customers were identified, in terms of their behavior toward the five target product categories. We used Lotus 1-2-3 to analyze the results.

Clustering or segmentation with unsupervised neural networks is a relatively easy process. In some cases, this is done iteratively. Data is first segmented to target specific groups with desired attributes or behaviors, and then segmented again for detailed analysis of those groups. The major challenge in data mining using segmentation is the analysis of the resulting clusters.

References

DeSieno, D. 1988. Adding a conscience to competitive learning, in Proceedings of IEEE International Conference on Neural Networks, San Diego, IEEE Press, pp. 117–24.

IBM Corp. 1994a. Neural Network Utility: User's Guide, SC41-0223.

IBM Corp. 1994b. Neural Network Utility: Programmer's Reference, SC41– 0222.

Kohonen, T. 1988. *Self-organization and associative memory*, 2nd ed., New York: Springer-Verlag.

O'Brien, T. 1994. Neural nets for direct marketers (software reviews), *Marketing Research: A Magazine of Management & Applications*, Vol. 6, No. 1, Winter, pp. 47–49.

Proctor, R.A. 1992. Marketing decision support systems: a role for neural networking, Marketing Intelligence & Planning, Vol 10., No.1, pp. 21–26.

Port, O. 1995. Computers that think are almost here, *Business Week*, July 17, pp. 68–72.

Venugopal, V., and Baets, W. (1995) Neural networks and statistical techniques in marketing research: a conceptual comparison, Marketing Intelligence & Planning, Vol. 12., No.7, pp. 30–38.

Verity, J.W., and R. Mitchell 1995. A trillion-byte weapon—marketers use massive power to woo customers, *Business Week*, July 31, pp. 80–1.

Chapter

10

Real Estate Pricing Model

"Everything is worth what its purchaser will pay for it." PUBLILIUS SYRUS

One of the most common problems faced by a business in any industry is how much to charge for its products and services. Of course, the costs of providing the product or service must be covered, but what is the appropriate level of profit to make? In markets where cost/plus pricing is used, this is a rather straightforward problem. However, in many cases, the market value of a product or service has only a passing relationship to the intrinsic value of the item. For example, two used cars made by different manufacturers might be on the market. They are the same age, have the same mileage, and might have been purchased for the same price. Does that mean they will have the same market value today? Of course not.

If one is a trendy model that holds its resale value, it could be worth much more. What if a manufacturing defect caused reliability problems with that specific model? This would lower its resale value. What if new car prices have just risen by 20%? This would have a positive impact on the resale value of used cars. While it is easy to look up the current average trade-in value of a car, the factors that affect those values are many. The question arises, "Can you write a formula for computing the market price of an item based on the attributes of the item and on current market conditions?" The answer is, "Probably not." There are usually complex, nonlinear relationships between variables that interact in subtle ways to determine the price someone is willing to pay.

Now just because it is difficult doesn't mean the job is impossible. One job that requires market analysis is a real estate appraiser. In this process, the

estimator looks at the house, noting how old it is, how big it is, how many bedrooms and baths it has, and walks around the property noting the size of the lot and the degree of landscaping. Then the appraiser adds or subtracts for special features such as fireplaces, central air conditioning, extra large garage, deck, permanent siding, etc. In essence, the estimator is collecting information on the attributes of the property, weighing their relative contributions, and coming up with an estimate of the market price. An experienced estimator compares the home to other homes that are similar and that have sold in the recent past. This gives valuable information on the current market value for homes like the one being appraised.

How does a good estimator or real estate agent learn the trade? By starting out with a small collection of homes with which he or she is familiar. As the agent spends more time on the job, additional homes come along with special or unique features. When the homes sell, the agent gets feedback on the market and whether the features enhanced or detracted from the market value of the home. If the real estate agent moves to a different area of the country, the standard features of a home will change, and the relative value of home attributes will vary. For example, an in-ground pool might well detract from the value of a home in Minnesota, while it would be a desired feature of a house in Florida. Likewise, a screened yard is a valuable addition in Florida but would be undesirable in Minnesota. The point is that even though the real estate agent might be quite experienced in appraising homes, much of that knowledge deals with the local tastes and building practices.

If we want to write a real estate appraisal application using COBOL or RPG, we would have to first write code to search through our database for homes that are "similar" to the one being appraised. Then we would have to write a formula that takes all of the home attributes and calculates a market price. Or we could take the price of the closest matching homes and then add or deduct for features that are different between the homes. This would not be a simple application.

However, we can use data mining to build this kind of application more easily. In this example, our training data consists of a set of attributes on a property and the known selling price. The property attributes are those listed on the common multilist forms. The neural network can "see" hundreds or thousands of homes an hour. Over the training time, the neural network can learn the fine distinctions between properties and how the various attributes affect the price. In addition, we could take this very same framework and use data from different cities to create customized real estate pricing models for each locale.

The basic data mining function we need to perform is called modeling or scoring. Statisticians would call this a multivariate nonlinear regression problem. We want to find out how all of those attributes contribute to the market value of the property. Our solution is to build a neural network to model this function. The IBM Neural Network Utility (NNU) is used to perform the data preprocessing and to train the neural network (IBM 1994a). Appendix A presents details on the functions provided by NNU. Other commercial neural network tools could also be used. In the next section, I describe the data used for this application.

Data Selection

A standard multilist form has a large number of attributes about real estate properties for sale. In this example we will build a price estimator for properties in Rochester, MN (voted #1 or #2 in *Money Magazine's* "best places to live" ranking in 1993, 1994, and 1995). The attributes we are interested in include the lot size, the age of the home, the living space, and the number of bedrooms and baths. We track the size of the garage (if any), since in Minnesota it gets extremely cold, and this is an important feature. There are five main home styles in Rochester: small Cape Cods in the center of town, one-story ranches, split entry homes, multilevel homes, and traditional two-story colonials. One important attribute in the Rochester real estate market is the quadrant of the city where the property is located. The southwest area is considered prime because many doctors from the Mayo Clinic reside there. The northeast, with its rolling hills, would be the next area of preference, followed by the northwest area near IBM, quite popular with families, and finally the southeast quadrant. Table 10.1 shows the property attributes we use in this application.

Data Representation

The home style is a categorical field with values of cape, ranch, colonial (two-story), multilevel, and split-entry. These symbolic values will be mapped into integers ranging from 1 to 5. The home style is known to have a strong impact on the home's value, and so a one-of-N encoding is used.

TABLE 10.1 Selected Data for Real Estate Pricing Model

Attribute	Logical data type	Values	Representation
Building style	Categorical	Ranch, split, cape, colonial, multi	One-of-N code
Location (quadrant)	Categorical	NW, NE, SW, SE	One-of-N code
Age	Continuous numeric	0 to 36	Scaled (0.0 to 1.0)
Lot sizes (acres)	Categorical	0.25, 0.33, 0.5, 1.0, 2.0	Scaled (0.0 to 1.0)
Number of bedrooms	Discrete numeric	2, 3, 4, 5	Scaled (0.0 to 1.0)
Number of bathrooms	Discrete numeric	1, 1.5, 2, 2.5, 3	Scaled (0.0 to 1.0)
Garage (number of cars)	Discrete numeric	0, 1, 2, 3	Scaled (0.0 to 1.0)
Living space (sq. ft.)	Continuous numeric	750 to 3000	Scaled (0.0 to 1.0)
Price	Continuous numeric	$80,000 to $180,000	Scaled (0.0 to 1.0)

The location of the home is represented by a categorical field with four values, NW, NE, SW, and SE. These symbols are mapped into integers and then coded using a one-of-N code. Our other option is to use a single input, with 0.0, 0.33, 0.66, and 1.0 representing the locations. However, the one-of-N data representation gives the neural network explicit information about the importance of the location to the selling price of the property.

The age of the home is a continuous numeric field ranging from 1 to 36 years old. It will be scaled down to 0.0 to 1.0. The lot size is a categorical field with values ranging from 0.25 to 2.0. The five common lot sizes are scaled into a range of 0 to 1.

The number of bedrooms is a discrete numeric field ranging from 2 to 5. The number of baths is also a discrete numeric field. Both will be scaled down to a range of 0.0 and 1.0. Note that no information is lost because of the scaling. The data is just compressed.

The size of the garage is a discrete numeric field with values from 0 to 3. This data is scaled to a range of 0 to 1. The living space of the home, which is a continuous numeric field ranging from 750 to 3000 square feet, is scaled down to 0.0 to 1.0.

The selling price of the home is the dependent or output variable. Home prices in Rochester range from $50,000 to $300,000, but in this example we will use $80,000 to $180,000 for simplicity. A linear scaling down to 0.0 and 1.0 will be used for this field. While the scaling used for the input fields is somewhat up to the person doing the modeling, the output variable must be scaled to the same range as the range of the activation function of the output units in the neural network. In our case, we use the standard logistic function, which ranges from 0 to 1. However, if we were using the hyperbolic tangent, we would need to scale from –1 to 1, and if we were using the symmetric logistic function provided by NNU, we would have to scale it from –0.5 to +0.5. This is required because the error in the back propagation learning algorithm is computed as desired minus actual, and they must be in the same range.

An NNU Translate Filter will perform all of the required preprocessing for our data. The categorical data is mapped to integers and then turned into one-of-N codes. The continuous and discrete numeric fields are scaled into our target range of 0.0 to 1.0. NNU requires a Translate template to specify these transformations. Figure 10.1 shows the NNU Translate template for our initial data representations.

Model and Architecture Selection

This kind of modeling problem usually requires a back propagation, recurrent back propagation, or a radial basis function network (see Table 4.1 in chapter 4). We will use the back propagation network because it is by far

Translate Editor - e:\homes.xlt

File Selected Edit View Windows Help

Name	Usage	Rep	Pre	Source	Table	Dest	Post
Style	Input	1	None	Symbol	Table	OneOfN	None
Location	Input	1	None	Symbol	Table	OneOfN	None
Age	Input	1	Scale	Number	None	Number	None
LotSize	Input	1	Scale	Number	None	Number	None
NumBedRooms	Input	1	Scale	Number	None	Number	None
NumBaths	Input	1	Scale	Number	None	Number	None
NumCarGarage	Input	1	Scale	Number	None	Number	None
LivingSpace	Input	1	Scale	Number	None	Number	None
Price	Output	1	Scale	Number	None	Number	None

Fields:9 In:8 Out:1 Ignore:0 | InBuf:9 Num:7 Sym:2 | OutBuf:16 Num:16 Sym:0

Figure 10.1 NNU translate template for modeling application.

the most popular neural network model and it is, in some respects, the easiest to work with.

The architecture of the neural network is mostly subject to our data representation decisions. The number of inputs and the number of outputs are determined by these choices. Our major architectural decision deals with the number of hidden layers and hidden units. Sadly, there is no cut-and-dried technique for making these choices. As described in chapter 4, this is an area ripe for automation. Because a feedforward neural network with one hidden layer can theoretically model any continuous function (and we assume our real estate price is a continuous function), we choose one hidden layer. Our initial choice for the number of hidden units is 10. This is arbitrary and will be modified up or down if we have trouble getting the network to converge.

Training and Testing the Neural Network

In doing modeling, we have the luxury of knowing what the "right answer" is. In this case, we have the set of attributes of the real estate property, and the known selling price. The first decision we need to make is how accurate our model has to be. This depends on exactly what the model is going to be used for. If we are going to use this model as part of a customer service kiosk in a mall, where the customer selects a home style, lot size, location, etc. and then gets an estimate of what a home like that will cost, then probably being within 5% would be fine. However, if the model is being used to validate real estate appraisals as part of a mortgage underwriting process, we might want a 2% or even 1% accuracy. We also must specify if this is the "average" accuracy, or the "worst case" accuracy. We could have an average of 5% but occasionally be off by 20%. This could cause real problems with

our customers. In our case, we would like the worst case to be within 5%. Before we start training the neural network, we don't really know how accurate a model we can get. It might be that it will be impossible to get to the level of accuracy we specify. However, it is important to start out with the goal firmly in mind. Otherwise, how do we know when we are done? Often, it is when the time we have to train the network runs out.

Now is a good time to examine what it really means to have a 5% accuracy in terms of neural network error rates. Our model has multiple inputs and a single output. This output ranges from 0.0 to 1.0, but the source data was scaled down from a range of 80,000 to 180,000 (this working range of $100,000 was chosen to make the following discussion more clear). When the desired output is $170,000, this scales down to a value of 0.9 (80,000 + (0.9 * 100,000) = $170,000). When the actual network output is 0.85 ($165,000), we are within our specified performance range. NNU reports the prediction error in several ways. The one we are most interested in is the average root mean squared error (average RMS). This is computed by taking the difference between the desired and actual output (0.9 – 0.85 = 0.05), squaring it (0.05 * 0.05 = 0.0025), summing up the squares (with only one output this step is not needed) and dividing by the number of outputs (1, in this example) and then taking the square root, which gives us the 0.05 back again. When we have a single output unit, the average RMS error is the same as the average output error. Because we have scaled the output variable by a factor of 100,000, and because there is only a single output, we can interpret the average RMS error as the actual prediction error (scale 0.05 RMS error by 100,000 to get a $5000 prediction error on the price of the house).

Because we not only want the average RMS error to be below 5%, but we also want the worst-case prediction error to be below 5%, we need to monitor the maximum RMS error parameter that NNU provides for back propagation networks. This is the RMS error for the worst pattern we have seen. To bring this discussion back to the problem domain, we want our model to be accurate to within $5000 of the actual selling price for a similar home, and to ensure this level of accuracy, we need to monitor the maximum RMS error parameter and make sure it stays below 0.05. Another important point to remember is that we must reach this level of performance on the test data, not only on the training data. That is the true test of the predictive accuracy of the neural network.

Figure 10.2 shows an NNU application module set up for training a back propagation network. The Import objects define the data sources. One contains 80% of our source data and the other points to a file containing 20% of the data. We can use either one at a time. We start by using the training Import file.

The Import objects are connected to the Translate Filter, which will preprocess the source data before it is fed into the neural network. This Filter uses the Translate template we defined in Figure 10.1. The Translate Filter

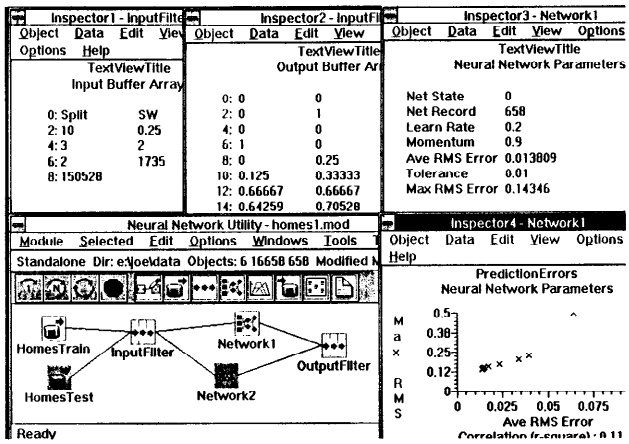


Figure 10.2 NNU module and Inspectors for modeling application.

processes the data and passes it to the Network object. Our neural network is the NNU back propagation model. We specified the number of inputs as 15, the number of hidden units in layer 1 as 10, the number of hidden units in layers 2 and 3 as the default value of 0, and the number of outputs as 1. Remember, the number of input and output units is determined by our data representations. The number of hidden units was chosen arbitrarily, as a reasonable starting point. At this point we are ready to train the neural network.

Before we start training the network, though, we want to open Inspectors or views on some of the data elements in the NNU application module so we can monitor the progress of the training. We open two Inspectors with text views on the Filter object's Input Array and Output Array. This shows us the source data coming from the text file into the Filter, and the translated data coming out of the Filter object.

To watch the state of the neural network as we train, we create two Inspectors on the Network object. On the first one, we select the following network parameters, Net State, Net Record Index, Learn Rate, Momentum, Tolerance, Ave RMS Error, and Max RMS Error. We monitor Net State because that shows us if we are in training (0) mode or if the weights are locked (1). We watch the Net Record Index to see that we are moving through the source data correctly. The Learn Rate, Momentum, and error Tolerance are the three learning parameters we use to control the training process. The average RMS error and maximum RMS error are our most important error mea-

sures for modeling with back propagation networks. On the second Network Inspector, we select the average RMS error and the maximum RMS error parameters and select an XY plot view. The XY plot view is a graphical display showing time on the X axis, the size of the error on the Y axis. The desired behavior is for both error measures to decay over time.

The default NNU parameters for a back propagation network start with a Learn Rate of 0.2 and a Momentum of 0.9. Because we want the accuracy to be within 5%, we set the error Tolerance to 0.05 (instead of the default, which is 0.1). This is important because otherwise the network would only ever try to train to a 10% accuracy level since anything within 0.1 would be considered as having no error.

To train the network, the NNU IDE continuously reads data from the active Import object and passes it through the Translate Filter for preprocessing, and into the Network. After the data is processed by the neural network, the output data is passed to another Translate Filter, which scales the neural network output back up to the 80,000 to 180,000 range.

Our first objective when training a neural network is to get a feel for how difficult the function is going to be to learn. Therefore, we first train the back propagation network for only 25 epochs. As the neural network trains, the prediction error starts to oscillate, so we halt the run. Oscillation of the error indicates that the neural network is having difficulty converging. Sometimes this is because the network is stuck in a local minima and can't get out. Other times, it is because the weights have grown too large and some of the units are forced completely in an on or off state. At this point the average RMS error is 0.020, well within our acceptance limit of 0.05, but the maximum RMS error is 0.119, which is not (remember this corresponds to a prediction error of $11,900 as the worst case). Even though the neural network didn't converge, we lock the network weights and switch to the Test Import object to see how the network will perform on the test data. When these 200 records are run through the network, our average RMS error is 0.049 (3% worse than on the training set), and the worst-case error is 0.076, which is actually better than on the training data.

At this point, we lower the error tolerance to 0.01, reset the network, and start the training cycle again. After 100 epochs, the average RMS error is 0.0085, and the maximum RMS error is 0.089. This is still higher than we'd like, but we are getting closer. We lock the network and switch to the test data again. This time the average RMS error is 0.035 and the maximum RMS error is 0.050. If we only cared about the prediction accuracy on the test data, we would say "ship it." Unfortunately, it looks like there are some examples in the training data that are extremely difficult for the neural network to learn because the worst-case prediction error is higher on the training data than on the test data. We could continue trying to modify the training parameters to see if we can bring the error down on the training set, or we could go back and change our architecture. At this point, we opt to add more hidden units.

We construct a new back propagation network with 15 inputs, 20 units in the hidden layer, and 1 output unit. The error Tolerance is kept at 0.01, the Learn Rate is bumped up to 0.5 (from 0.2) and we repeat the training cycle. We raise the Learn Rate to speed training because the neural network seems reasonably stable. After 200 epochs, the average RMS error is 0.005 and the maximum error is 0.048 (below our 5% threshold). On the test or holdout data, the average RMS error is 0.023 and the worst-case error is 0.027. So we have successfully built a predictive model that can give accurate estimates (within 5%) of the actual selling price for homes in Rochester, MN. The entire training process took approximately two hours.

Deploying and Maintaining the Application

To deliver this neural network model as an application, we need a data entry screen of some kind, as simple or fancy as we want to make it. The input data is the information shown in Table 10.1 (without the price, of course). This text data needs to be preprocessed and passed to the neural network, which produces an output value between 0.0 and 1.0. We then scale this value back up to the price range that the customers will see. Notice that these are the identical steps we performed in our training process.

If we use NNU, we can use the same application module that we used to build the model. We could use NNU's Application Delivery Environment shell to build the stand-alone application. Another option is to write a simple application to call the NNU application programming interface (API), passing in the buffer of input data and retrieving the scaled output for display (IBM 1994b).

After just training the neural network, we know that it is an accurate model of the current real estate market in Rochester, MN. But what about next week or next month? In any modeling application, we need to periodically check the model to see if it is still valid. This could be as simple as keying in 10 recent home sales and checking the model estimate versus the actual selling price. If any of the estimates are more than 5% out of range, then we retrain the network with the latest data. Or we could automatically test the model each night against the day's sales, and if the model is out of the range of our acceptance criteria, we kick off a batch training session to update the model.

Related Applications and Discussion

Once we have an accurate real estate pricing model, we can use it in a number of ways. We could give it to all of our real estate agents to take along in

their notebook computers. We could provide a GUI data entry screen where they enter the attributes of a property someone is selling, and it returns an estimate of the current market value. If the customer is also looking to trade up to a newer house, the real estate agent could enter the customer's wish list and give the customer a feeling for what his or her dream home would cost, all in the privacy of the customer's own home.

We could turn this into a customer service kiosk in our local mall location. The kiosk stops traffic long enough for the "gallery of homes" display to catch people's eye. The potential customers could enter data on the homes they are looking to buy or sell and get a "no-hassles" estimate. Another possibility is to use the model as a training tool for new real estate appraisers, where trainees are given homes to appraise, and check their answers against the pricing model. Or it could be used as a "second opinion" on appraisals submitted for mortgage insurance or for a home mortgage loan. We could let local builders use the tools to select the right type of home to build, which will have the most market appeal. As we have shown, a single pricing model, built using data mining with neural networks, is a versatile information processing tool.

In this example we used real estate data to build a price estimator. However, this simple example represents a broad range of potential data mining applications. The basic processing function performed is called modeling or function approximation. The neural network learned to map a set of inputs to an output value. Although, in this example, we modeled a function with only a single output, neural networks can be trained to mimic complex nonlinear functions with multiple output variables. Instead of real estate data, we could just as easily have used data relating manufacturing process parameters to product quality levels, or financial market information to the value of a futures contract. The real power of data mining with neural networks is that they can turn data into applications. And the nature of the data determines the type of application that is produced.

Summary

In this example application, we took a set of attributes describing real estate properties and built a pricing estimator. Starting with a database of homes and their selling prices, we selected several data fields and chose appropriate data representations based on the type of information contained in those variables. We used a back propagation neural network to build a model relating the input data to the market price of the property. The IBM Neural Network Utility was used to do the data preprocessing and to train and test the neural network. After several iterations through the training process, we developed a pricing model that was accurate to within 5% of the actual value.

Mining data to build predictive models using neural networks is a fast way to develop applications. This approach can be applied to data from any domain or industry and represents a flexible application development methodology.

References

IBM Corp. 1994a. Neural Network Utility: User's Guide, SC41–0223.

IBM Corp. 1994b. Neural Network Utility: Programmer's Reference, SC41–0222.

Kathmann, R.M. 1993. Neural networks for the mass appraisal of real estate, Computers, Environment and Urban Systems, Vol. 17, No. 4, Jul.–Aug., pp. 373–384.

Worzala E., M. Lenk, A. Silva. 1995. An exploration of neural networks and its application to real estate valuation, *The Journal of Real Estate Research*, Vol. 10, No. 2, pp. 185–201.

Chapter

11

Customer Ranking Model

"When you've got them by their wallets, their hearts and minds will follow."

FERN NAITO

Problem Definition

A major requirement for any business to grow is to find new sources of revenue. This expansion of a business can be done in two ways: find new products to sell to your existing customers, or find additional customers. One of the tried-and-true methods for growth is to understand who your current customers are, and then try to find other people who are most like your current customers.

Another similar problem is when a business has a large customer base but wants to target specific promotions at its best customers. In one sense, the business needs to rank its existing customer set based on a set of parameters that define what a "good" customer means. Is it the most profitable customers? Or maybe it is the customers who generate the highest volume of business transactions. Or maybe it is the customers who have the longest relationship with the business, the loyal customers. Or maybe it is some combination of these factors.

Now it is not difficult to do a SQL query against a customer database to list customers based on the dollar amount of their total purchases, or on the length of time since their first transaction with your business. But combining multiple factors, each with different levels of importance (weighting), is not a simple matter at all. This is an example where a complex query will not suffice.

Neural networks have excelled in applications that require a complex weighting of multiple factors. In many ways, we are asking a much more

profound question than, "Which customers generated the most sales last year?" It is, "What are the attributes of my best customers?" If we know that, then we can search for new customers who meet that profile and be fairly certain that our marketing costs will be handsomely rewarded by new business.

In this example, our business is a custom print shop. We have been serving a collection of local businesses for some time, and we now want to target the most profitable segments to try to grow our business. A new salesperson is being hired, and we want to direct him or her toward the most lucrative customers. Our example data mining application is to take information on our existing customers and mine that data to build a neural network classifier that we can then use to rank potential new customers. We will use the IBM Neural Network Utility (NNU) to perform the data preprocessing and data mining (IBM 1994a). (See appendix A for more information on NNU.) Other commercial neural network tools that provide similar functions to NNU could be used in its place. The Lotus 1-2-3 spreadsheet is used to do analysis of the input data.

Data Selection

Our customer database contains information about our customers, such as their names, addresses, how long they have been in business, the type of businesses they are in, their annual revenue, the average number of transactions or jobs they have contracted over the past year, and the average revenue and profitability of those transactions. The first idea that comes to mind is to rank the customers on their average profitability of the transactions. Those with the highest margins will be our first target. As mentioned before, this can be done using an SQL query, and it is a reasonable approach for our current customers. But we want to find out whom we should target for new business.

In this application, we will use all of the available information to build a neural network model that can accurately rank the customers according to their "goodness." Our goodness measure will be a score that is a combination of the average number of transactions and the average profitability of those transactions. That way we will target both high-volume and high-profitability customers. As a secondary data mining application, once we have this model, we will perform a sensitivity analysis to understand which customer attributes are the best predictors of profitability. Table 11.1 shows the data that we have available to build our customer ranking model.

Data Representation

The age of the company, or the number of years in business, is a continuous numeric field ranging from 1 to 15 years. We will scale this down to 0.0 to 1.0. The number of employees is also a continuous numeric field that is somewhat correlated with the age of the company. However, we feel that this is an important indicator and should be considered separately. We will take the range of 1 to 100 and scale it down to 0 to 1.

TABLE 11.1 Selected Data for Customer Ranking Application

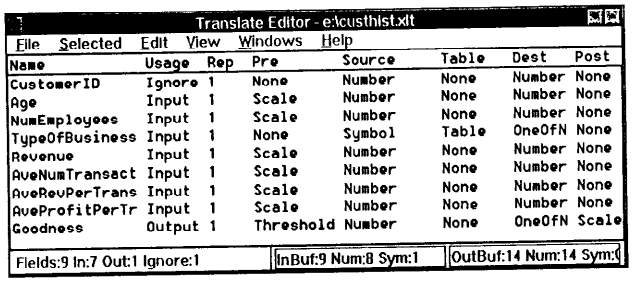
Attribute	Logical data type	Values	Representation
Years in business	Continuous numeric	1–15	Scaled (0.0 to 1.0)
Number of employees	Continuous numeric	1 to 100	Scaled (0.0 to 1.0)
Type of business	Categorical	Manufacturing, Service, Retail, Nonprofit	One-of-N code
Revenue	Continuous numeric	$0 to $4,000,000	Scaled (0.0 to 1.0)
Ave. number of orders	Continuous numeric	1 to 80	Scaled (0.0 to 1.0)
Ave. revenue per order	Continuous numeric	$6 to $250	Scaled (0.0 to 1.0)
Ave. profit per order	Continuous numeric	$1 to $50	Scaled (0.0 to 1.0)
Goodness	Continuous numeric	6 to 2600	One-of-N code

The type of business is one of four categories: manufacturing, service, retail, or a nonprofit organization. In this data mining application, we feel that this is one of the most important factors because the type of work contracted is dependent on this. Manufacturing businesses print sales brochures, product installation guides, and service manuals, which are usually high-volume jobs. Service companies print service contracts and coupons. Retail businesses have varied needs, from one-time signs to color advertising circulars and inserts. Nonprofit organizations have newsletters and materials for direct mail campaigns. Because of its importance, the type of business will be represented by a one-of-N code for best visibility to the neural network.

The revenue of the company is a continuous value ranging from $0 to $4,000,000. A preliminary analysis of this showed that this was normally distributed over this range. So we will simply scale this down to 0.0 to 1.0. If this was skewed, then we would take the log of the revenue to normalize the data.

The average number of orders per year is a continuous numeric field with values ranging from 1 to 80. The average revenue per order ranges from $6 to $250, while the average profit per order is in the range of $1 to $50. These fields are scaled down to 0 to 1.

We will compute a new field, which we will call "goodness," by multiplying the average number of orders by the average profit per order. This factor will be split using a threshold function into three categories, which we will call "A," "B," and "C." For example, companies with a goodness score above 1000 are "A" customers, customers over 500 are "B" customers, and customers below that are "C" customers. These three categories will be turned into a one-of-N code. Note that is a somewhat arbitrary decision. By



Translate Editor - e:\custhist.xlt

File Selected Edit View Windows Help

Name	Usage	Rep	Pre	Source	Table	Dest	Post
CustomerID	Ignore	1	None	Number	None	Number	None
Age	Input	1	Scale	Number	None	Number	None
NumEmployees	Input	1	Scale	Number	None	Number	None
TypeOfBusiness	Input	1	None	Symbol	Table	OneOfN	None
Revenue	Input	1	Scale	Number	None	Number	None
AveNumTransact	Input	1	Scale	Number	None	Number	None
AveRevPerTrans	Input	1	Scale	Number	None	Number	None
AveProfitPerTr	Input	1	Scale	Number	None	Number	None
Goodness	Output	1	Threshold	Number	None	OneOfN	Scale

Fields:9 In:7 Out:1 Ignore:1 | InBuf:9 Num:8 Sym:1 | OutBuf:14 Num:14 Sym:0

Figure 11.1 NNU translate template for classification application.

turning the goodness measure into three discrete categories, we have assigned a grade to our current (and prospective) customers. An alternative is to turn this into a neural network modeling problem with a single output. Either approach is valid. It all depends on how you want to use the ranking model. We could ask the neural network to classify the customers, or we could ask it to simply score them, and then classify them ourselves.

The Neural Network Utility provides a data translation function called a Translate Filter. Using the NNU Translate Editor, we specified our source data and the required symbol mapping and scaling operations. Figure 11.1 shows an NNU Translate Filter set up for this problem.

Model and Architecture Selection

The data mining function we require in this application is classification. Table 4.1 in chapter 4 lists the major neural network models and their primary uses. While several different models can perform classification, we will use the standard back propagation network, which is supported by NNU. Based on our data representation choices, we have 10 inputs (years in business, number of employees, four types of businesses, revenue, average number of orders, average revenue per order, average profit per order) and 3 outputs (goodness represented as A, B, C). Our initial architecture will have one hidden layer of 15 units. The number of hidden units is an arbitrary decision, based on our experience training neural networks. If we have problems getting the neural network to classify the customers, we might have to increase this value.

Training and Testing the Neural Network

To train this neural network classifier, we need to split our source data of 1000 records into a training and a testing data set, with 80% assigned to training and 20% assigned to testing. The Generation function of NNU randomly selects the records and creates the data sets. An alternative is to manually split the data using an editor or to write a program to split the source file into two parts of the correct size. Because NNU already provides this feature, we make use of it. Also, we will use a control script to manage the training of this model. The control script, shown in Figure 11.2, sets the network parameters and then switches back and forth between the training


```
Comment Customer Ranking Application control script
Debug ON
Variable NETWORK
Set NETWORK = Backprop1
Variable TRAINED
Set TRAINED = FALSE
Variable BESTRATIO
Set BESTRATIO = 1.0
ClearAll
Comment Set Training Parameters
Set NETWORK LearnRate = 0.1
Set NETWORK Momentum = 0.9
Set NETWORK Tolerance = 0.3
Reset NETWORK

Comment Loop while training the network
While TRAINED = FALSE
RunMacro TrainBackProp
RunMacro TestBackProp
EndWhile

Macro TrainBackProp
Set NETWORK NetState = 0
Set TrainImport State = ON
Set Module StepsPerCycle = 800

SetBreakPoint NETWORK BadPattern Ratio < 0.10
Run
ClearAll
EndMacro

Macro TestBackProp
Set TestImport State = ON
Set Module StepsPerCycle = 200
Set NETWORK NetState = 1
Cycle
Cycle
If NETWORK BadPatternRatio > BESTRATIO
Set TRAINED = TRUE
Else
GetValue NETWORK BadPatternRatio = BESTRATIO
Endif
EndMacro
```

Figure 11.2 NNU control script for the training neural network classifier.

and testing data sets. When the number of misclassifications on the testing data starts to increase, we stop the training cycle, lock the network weights, and then analyze the network model. Remember, we want to train the neural network to have the best classification accuracy on the test data, not on the training data.

Before we start training the network, we have to decide what our acceptance criteria is in order for the neural network classifier to be considered "trained." Since we have somewhat arbitrarily split the goodness measure into three segments, we need to be somewhat relaxed in our requirements from the classifier. If we did segmentation on the customers and determined the exact number of unique clusters, then that would allow us to be more exacting. The output of this tool is going to be used to help our new salesperson target new customers, so we want it to be fairly accurate. But we also recognize that all good prospects won't necessarily turn into new customers. Our goal is to correctly classify 90% of the customers into classes A, B, and C.

For classification problems, NNU provides a network parameter called Bad Pattern Ratio, which is, as the name suggests, the number of patterns that fall outside of the error tolerance range divided by the total number of patterns. A bad pattern ratio of 0.10, means that 10% of the patterns were misclassified. So our target is to get a 0.10 or lower value in the bad pattern ratio on the test data.

On our first training run, we used a learn rate of 0.1, the default momentum value of 0.9, and an error tolerance of 0.3. The reason for the large error tolerance is that in classification problems, we are driving the neural network output units to 1s and 0s. Our desired output pattern is a one-of-N code with three elements, so there will always be one unit on and two units off. Having an error tolerance of 0.3 means that the output unit only has to be greater than 0.7 to be considered a 1, and less than 0.3 to be considered a zero. This will help avoid a problem known as saturation, which occurs when the connection weights get too large, and the output of the processing unit is forced into either an on (1) or off (0) state.

In our example, after 200 training epochs, the back propagation network was not converging at all. The bad pattern ratio was still above 0.20, so we halted training. The first thing that came to mind was the target range for the classifier. We were using a binary one-of-N code, and maybe this was giving the network problems. So we added a scale postoperator to the NNU Translate Filter so that the one-of-N codes produced are in the range of 0.1 (0) to 0.9 (1). Our hope was that this would improve the convergence of the network. After several more trials, it was clear that this was not the problem either. If it was, using the large error tolerance would have avoided this problem. So back to the drawing board.

Maybe the thresholds we used to split the goodness score into categories were falling in the middle of a natural cluster in the data. So we looked at the distribution again using Lotus 1-2-3 and decided to split the goodness codes into four parts: A, B, C, and D, with breaks at scores of 500, 750, 1000. Now we will classify the customers into four groups, with A and B making up approximately 25% of the customers, C making up 15%, and D the remainder.

On our next training run, the network converged to a bad pattern ratio of 0.08, under our target of 0.10 after 200 training epochs. We locked the network and switched to the test data set containing 200 additional customers. The bad pattern ratio on these customers was 9.8%, which also meets our acceptance criteria. At this point we have a classifier that can take information on prospective customers and make a prediction (based on our current customer base) of what kind of customers they will be.

Sensitivity Analysis

Having a neural network classifier that can accurately predict which companies have the potential to become profitable customers is nice. However, we would also like to understand what factors separate our best customers from our good customers. Knowing this can have a profound effect on how we run our business. Should we only seek out new business from manufacturing companies? Or should we focus on retailers? Are young, fast-growing companies the most profitable, or do we want older, more stable customers? In this section, we will examine how we can use our neural network classifier to answer some of these questions.

Our first approach is to treat the neural network as a "black box." It is a seer that knows all (at least we can pretend)! The key, of course, is knowing which questions to ask, or maybe, how to ask them. In the Neural Network Utility development environment, there is a function called the Import dialog. It allows you to submit "what-if" questions to a neural network. Also, it provides a way to "lock" field values so you can more easily see the effect that changing one or more input values has on the classification. Figure 11.3 shows the NNU Import dialog for our customer ranking application.

First, we lock our neural network so that it is in runtime or prediction mode. We can walk through our training or test data and stop when we come to a case we are interested in exploring. We enter the average values for all of the fields and then lock all but the type of business. In turn, we enter each type of business and see if that has an impact on the classification. Service, manufacturing, and retail businesses are classified as C or average customers. Nonprofit organizations are classified as D customers. Remember, all the other values are held at the average value. This tells us that the average nonprofit organization is less desirable than any other type of customer.

Looking at service businesses, if they are average customers they have a C rating. But when we bump the average profit per transaction from 17 (average) to 22, they move into the B category. When their average profit per transaction is above 30, they move into the A category. Of course, finding

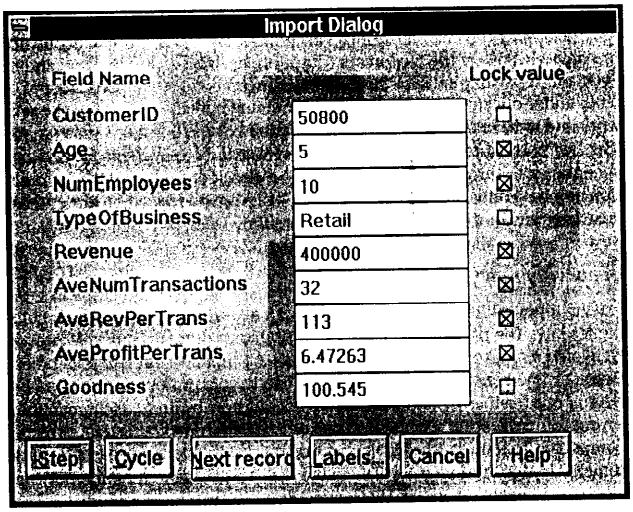


Figure 11.3 Sensitivity analysis of the neural network classifier.

customers who buy only high margin products or services is not easy. Another approach would be to look at increasing the number of orders per customer. If the profit per transaction is average, then the average number of yearly orders needs to be at least 45 (compared to the average of 32) to move into the B category, and at least 60 to move up to the A category. So for our custom printer business, if we can get our service business customers to increase their orders by one a month, they will be B customers, and by two orders a month, they move into the A category, our most profitable customer category.

One way to automate this sensitivity analysis process is to generate a test data file that contains the average values for each attribute and then, one by one, cycles through the range of each individual attribute and records the output. This can be taken to the next level by varying two or more attributes at the same time to identify confounding effects. A spreadsheet or other data analysis tool can be used to graph the decision points. For example, we could plot the attribute values on the horizontal axis and the goodness categories on the vertical axis. We could also use rule-generating data mining algorithms to transform this data into rules. These are just a few of the ways to discover what the neural network learned.

Deploying and Maintaining the Application

Deploying our neural network classifier as an application requires the following steps. First, the source data must be read from a file or input from a data entry screen. This data must then be preprocessed, passed through the neural network classifier, and then postprocessed to convert the network output value into one of the four categories. These are the very same steps we performed while training the neural network.

If we use the Neural Network Utility to deploy this application, we can use the NNU application programming interface (API) to load the same application module we used to train the classifier. We could write a five- or six-line program (as illustrated in appendix A) to automatically process the data in a source file and write the results in an output file.

Maintaining the neural network classifier involves monitoring its performance to see if its classification accuracy begins to fall off. If it does, then retraining with the latest customer information and goodness score will be necessary. This is one of the primary advantages to using data mining techniques for application development. When the application needs to be updated, you simply mine the latest data to refresh the neural network.

Related Applications and Discussion

We can use the classifier as we originally intended, which is to rank new or prospective customers for our business. We can also use our customer ranking model to examine our current customers to identify which ones deserve more attention or special treatment. Customers who fall into our "A" or "B" rank might warrant special credit terms or other incentives because they are very profitable, and we want to keep them as our customers.

The most likely use for our classifier is as a batch application. We can run our current customers and information on new customers through the neural network and store their grades or rankings in another field in the database. Or we can use our classifier as a means to select which customers to target. This approach was used in a telemarketing application developed by Churchill Systems, Inc., for a medical equipment supplier (Kestelyn 1992). In this application, the company had a large number of inactive accounts. A customer ranking application was developed using NNU. The inactive customers were run through the neural network, and their rank or score was added to their customer database record. When telemarketers were free, they would select names from the top of the sorted list of inactive customers who were most like their current best customers. They reported a large increase in reactivation of inactive accounts.

A classifier can be used as an automated decision maker. If data is available on the decisions that an expert makes, then an equivalent neural network classifier can usually be developed. The advantages of these "expert

networks" are that they do not require that the expert describe the decision-making process in terms of if-then rules, and that the performance of two, three, or more experts can be combined into a single "hybrid" or "super" expert.

While in this example we took a value that was continuous and arbitrarily broke it into categories, there are many other applications where the categories occur naturally. For example, in a quality control application, the product under test is either good or bad. A consumer credit application results in a yes or no answer. In some cases, there might be a third category, which we will call "maybe" for items that fell into the gray area. Items that are classified as "maybe" might have to get routed to a human expert who will make the final determination. But automating the easy calls can significantly improve operating efficiency and actually enhance the human expert's job satisfaction, since he or she is seeing more interesting cases.

Summary

In this example application, we mined information on our current customers to build a classifier to rank them according to a goodness score. This score was then broken into three categories, representing A, B, and C customers. We selected several customer attributes and chose appropriate data representations for them. The IBM Neural Network Utility application development tool was used to perform the data preprocessing and to train and test the neural network classifier. After several iterations, we decided to break the goodness into a four-tiered scale, after which we successfully trained a neural network that correctly classified 90% of the customer records into the corresponding categories.

Mining our customer data using neural networks allowed us to perform sensitivity analysis against the neural classifier. We did some preliminary analysis using the NNU Import Dialog to explore the impact that the type of business had on the quality of customers and the effects of increased transactions or profit mix. We discussed how the sensitivity analysis could be automated. Sensitivity analysis provides a way for us to discover what the neural network learned during the data mining process. Mining data with neural network classifiers provides a way to produce applications and a mechanism for extracting strategic business information for decision support.

References

Grupe, F.H., T. von Sadovsky, M.M. Owrang. 1995. An executive's guide to artificial intelligence, Information Strategy: The Executive's Journal Vol 12, No. 1, Fall, pp. 44–48.

IBM Corp. 1994a. Neural Network Utility: User's Guide, SC41–0223.

IBM Corp. 1994b. Neural Network Utility: Programmer's Reference, SC41–0222.

Kestelyn, J. 1992. Application watch: sales support's cutting edge, *AI Expert Magazine*, Jan., pp. 63–64.

Moore, K., R. Burbach, R. Heeler, 1995. Using neural nets to analyze qualitative data. *Marketing Research: A Magazine of Management & Applications*, Vol. 7, No. 1, Winter, pp. 34–39.

Wray, B., A. Palmer, D. Bejou. 1994. Using neural network analysis to evaluate buyer seller relationships, European Journal of Marketing, Vol. 28, No. 10, pp. 32–48.

Chapter

12

Sales Forecasting

"If you can look into the seeds of time and say which grain will grow, and which will not, speak then to me."

SHAKESPEARE, *Macbeth*

The ability to detect patterns over time has proven to be quite useful to humanity. The high priests of ancient civilizations used their understanding of astronomy to predict the passing of the seasons, the course of the weather, and the growth of the crops. Today, one of the most useful business applications of neural networks is using their ability to capture relationships in time-series predictions. Knowing what direction a market is heading or identifying a hot product before your competitors do has obvious implications for your business. If knowledge is money, foreknowledge is money in the bank (hence, the insider trading rules).

In this chapter, I focus on the problem of sales forecasting and inventory management. We have a set of products to sell, and we need to predict sales and order inventory so that we minimize our carrying costs. At the same time, we do not want to lose sales because we are out of a popular item. This is a problem in any manufacturing, wholesale, or retail operation. Convenience stores are especially subject to this problem because someone who is motivated to go out looking for his or her favorite ice cream or beer is certainly willing to move on to the next store if the first one doesn't have what the customer wants (Francella 1995).

Many factors contribute to whether an item is on the shelf when a customer comes in to purchase it. First is the item's supply or availability from the manufacturer and the lead time required to receive new items when

stocks get low. Next is the expected demand for the product. Related to this is whether any advertising or promotion is planned or underway for the product (or related products), which might have a temporary impact on demand.

In this example application, our business is a new and late-model used car dealer. The cost of carrying excess inventory is prohibitive, and managing the number of cars on the lot at any time is a major headache. There are cyclical swings and abrupt short-term demand increases in our sales history. Our current inventory control system amounts to simply replacing the cars we sold. However, the lead times on some models cause lost sales. Management feels that if we can build a relatively accurate sales forecast, both inventory management and staffing operations will be enhanced. We will use the IBM Neural Network Utility (NNU) to build a neural network sales forecasting application (IBM 1994a). Appendix A provides details on NNU data mining capabilities.

Data Selection

Our database contains daily and weekly sales histories on all car models over the past five years. While we know that there are differences in sales based on the day of the week, we are not worried about getting to this fine level of granularity. We have, on average, a two-week lead time from the factory and usually less when we can find a car from the local network of dealers. If we can accurately predict weekly sales two weeks in advance, then we can make sure we have the inventory we need. This information will also be used to schedule sales staff and the part-time car prep technicians.

We also have information on any factory sales incentives at least two weeks in advance. We run local print and radio advertising, which we know have a positive impact on sales. This information is in the form of weekly sales reports. In addition, we have context information that we can use to help the network to learn other environmental factors that could impact the sales. The effect of the month or time of year is called seasonality. By encoding and providing the time information, the neural network can learn to identify seasonal patterns that affect sales (Nelson et. al 1994).

In our sales database we have the following information:

- Sales: Date, Car Model, Model Year, Cost, Selling Price, Carrying Time, Promotions

To construct our sales forecasting system, we take the Date information and process it to get an indicator as to what quarter of the calendar year we are in. Typically the first quarter is slow, the second picks up some, the third is the best, and the fourth quarter lags a bit. In addition, we compute an end-of-month indicator that is turned on for the week, which includes the last few days of each month. This was added because we know there is a end-of-month surge in sales as the sales manager tries to get sales on the books.

TABLE 12.1 Selected Data for Sales Forecasting Application

Attribute	Logical data type	Values	Representation
Promotion	Categorical	None, print, radio, factory	One-of-N
Time of year (quarter)	Discrete numeric	1, 2, 3, 4	Scaled (0.0 to 1.0)
End of month flag	Discrete numeric	0 and 1	Binary
Weekly sales	Continuous numeric	20 to 50	Scaled (0.0 to 1.0)

We have noticed that there is some carryover from one week to the next. Special promotions tend to increase customer traffic for the following week also. Weather also has an impact on our sales. As a consequence, our sales pick up during the warm summer and early fall months. However, since the weather is quite variable, we will only use the calendar quarter to indicate seasonality.

To give the neural network some information on the recent sales figures, we give information from the current week, including the current week's seasonality, promotions, end-of-month marker, and total number of cars sold. This information is combined with the known information for the next week. The goal is for the neural network to accurately predict next week's sales. This sales estimate will then be fed back into the neural network forecaster to predict the sales for following week because we really want to forecast two weeks out. Another approach would be to train a neural network that would predict two weeks into the future. Depending on the application, this is certainly possible. Neural network forecasting models have been used successfully to predict sales six months, and even a year, in advance. However, for this example, we chose to use the simpler one-week model and use it iteratively to forecast two weeks in advance. Table 12.1 shows the data used to build our sales forecasting system.

Data Representation

The time of year is represented by a discrete numeric field with values from 0 to 3 corresponding to each quarter. We scale these values to a range of 0.0 to 1.0. This information is used to give the neural network any information it needs about seasonality in making its sales prediction.

The promotion field is a categorical value that indicates the type of promotion going on, if any. The type of advertising is assumed to have an impact on demand. The values include no promotion, print or radio advertising, or factory incentives. We use a one-of-N coding for the promotion field.

The number of units sold each week over the last five year period ranges from 20 to around 50. In this application, this sales figure is scaled down to

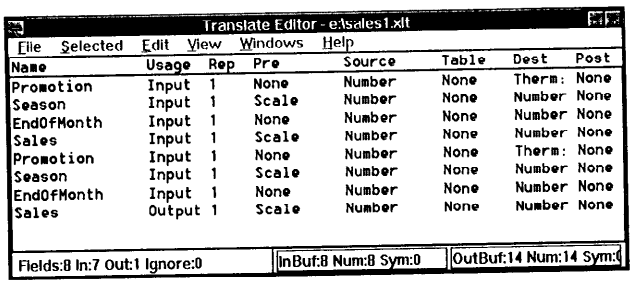


Figure 12.1 NNU translate template for forecasting application.

a range of 0.0 to 1.0. Please note, however, that sometimes it is better to scale the dependent or output variable in a neural network modeling or forecasting application to a range of 0.1 to 0.9. Driving the output units to their extremes (that is, 0.0 and 1.0 for the standard logistic activation function) requires large connection weights. If the neural network model predicts accurately everywhere but at the extremes, then changing the scaling range could help.

The Neural Network Utility provides a data transformation tool called a Translate Filter. The scaling and transformation of data is specified by something called a Translate template. Figure 12.1 shows the NNU Translate template we used in this application.

Model and Architecture Selection

As illustrated in Table 4.1 in chapter 4, there are three major types of neural networks that can be used to build time-series forecasting models: back propagation, which is the jack-of-all-trades of neural networks, radial basis function networks, and recurrent back propagation networks. If there is a lot of variability in the data, then radial basis functions might perform best because their fixed center weights allow the network to learn different mappings for different portions of the input space. As the training data moves around, the rest of the radial basis function weights do not degrade. This stability to nonstationary inputs makes radial basis function networks excellent for modeling problems.

Back propagation networks, on the other hand, are susceptible to something called the "herd effect." As the input data moves around the input space, all of the connection weights are adjusted and tend to move to follow the inputs. This behavior occurs when the weights are adjusted after each training pattern is presented, which is the standard technique used. Another

concern when using back propagation for time-series forecasting is that feedforward neural networks have no "memory." If the function being modeled has complex dynamics that are dependent on three or four prior states, then all of this information must be presented to the back propagation network at the same time. This technique is called the "sliding window" approach. While it works and has been used for many applications, it does have the drawback that it might require large networks with many input units, which results in long training times.

In some cases, a fully recurrent neural network is required to capture the behavior of complex dynamical systems. However, training recurrent networks can be extremely time-consuming, making the standard back propagation training algorithm seem fast in comparison. In this example, we will try to use the standard back propagation network, and then see if the limited recurrent network provided with NNU performs better.

The number of input and output units is determined by our data representation and by the number of prior time steps required by the function. In our example, we will use context information and sales data from the current week plus the context information for the next week to predict the next week's sales. This means that we need 13 input units (4 units for promotion, time of year, end-of-month indicator, sales from the current week, 4 units for next week's promotion, next week's time of year, and next week's end-of-month indicator) and a single output unit representing next week's sales.

In addition, we specify one hidden layer with 25 hidden units. As we discussed previously, this initial choice is somewhat arbitrary and selected based on our experience. This experience also tells us that this number might have to be increased or decreased if we have difficulty training the neural network. Do not labor over whether to use 23 or 27 hidden units. While fewer hidden units usually result in better generalization, spending a lot of time searching for the optimal number is often not worth the effort.

Training and Testing the Neural Network

Before we start training the neural network, we first have to decide what level of prediction accuracy is acceptable. The sales forecasting system is going to be used to predict sales two weeks in the future by making two passes through the forecasting model. This gives us some time to either trade with other dealers for inventory, or go to the auto auction to pick up some late-model used cars. An average prediction accuracy of 10% is considered acceptable. In the car business, we aren't ever really in danger of having no cars to sell. The problem is trying to manage the carrying costs of having an overstock on the lot. For the neural network, this translates into an average root mean square (RMS) error of less than 0.10.

We start with a standard back propagation network. Figure 12.2 shows an NNU application module set up for this problem. The Module editor (on the bottom left) shows two Import objects that have our training and testing data. For the training data, we took four and one-half years of weekly sales data. We will test the forecaster with the most recent six months of data. This decision points out one of the problems with time-series forecasting problems. We are using historical data. What if the world has changed in the past six months? How will our neural network ever learn the data? Actually, this is a problem even if we used the latest data to train the network. Tomorrow something fundamental to our problem could change (for example, interest rates go up 2%), which completely blows our model away. The only way we can deal with this is to try to get all of the relevant variables into our model. For example, if sales are sensitive to interest rates, then, by all means, that should be an input variable.

Back to the NNU module again, the Import objects feed their data into the Translate Filter, which preprocesses the data. This data is passed to the neural network, which uses the data in training. The network output is fed into another Translate Filter, which scales the value back up to the number of cars sold.

To monitor the training of our neural network, we open several NNU Inspectors on the module. We open one on the Input Array of the preprocessing Translate Filter to see the source data coming from either of our

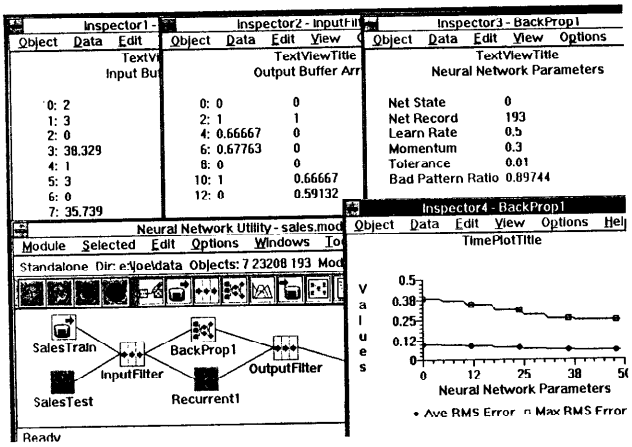


Figure 12.2 NNU module and Inspectors for forecasting application.

Imports, and we open another one on the Output Array of the Translate Filter to see the preprocessed data going into the Network. On the Network object itself, we open two more Inspectors. One is used to view the key neural network parameters, which in this case are the Net State, the Net Record index, the Learn Rate, the Momentum, the error Tolerance, and the Bad Pattern Ratio. The Net State tells us whether the network is locked or is in training mode. The Net Record index indicates which input record we are processing. Learn Rate, Momentum, and error Tolerance are used to control the learning in the neural network. Finally, the Bad Pattern Ratio is used to monitor the percentage of training patterns that are within our specified Error Tolerance. The second Inspector on the Network object is a time-plot of two error parameters, the average RMS error, and the maximum RMS error.

Because this is a time-series problem, we modify the default Learn Rate and Momentum parameters set by NNU. For Learn Rate we use 0.5 instead of 0.2, and for Momentum we use 0.3 instead of 0.9. Our experience with time-series problems tells us that Momentum can have an adverse impact on training. In some cases it works fine, in others not so well. The behavior of the neural network during training is extremely dependent on the network architecture, the data, and the type of function being performed. The initial choices for parameters are not crucial. We typically try a couple of combinations of parameters to see how the network reacts. In general, we want to use the largest learn rate we can to speed up the training. Many developers gradually drop the learn rate as the training progresses to improve predictive accuracy.

We start training the neural network using the NNU Run function. Records are continuously read from the training Import and pass through the connected NNU objects. After approximately 100 epochs or complete passes through the training data, the network has already reached an average RMS error of below 0.05. However, the maximum or worst-case prediction error is above 0.20. The prediction errors were coming down nicely but then started to oscillate. This behavior is made obvious by the time-plot Inspector we opened on the network.

As we step through the training data, we examine the input records that are giving the neural network problems. These are the records with the highest prediction errors. It is clear that whenever there is a promotion code of 4 (factory incentives), the network has trouble. This promotion code seems to be a key factor, whether it appears in the current or the upcoming week's context information. Our initial data representation for the promotion field was a one-of-N code. This should give the network enough indication of the "specialness" of the factory code because a single input unit is devoted to representing factory promotions. However, just to make sure this isn't causing problems, we change the NNU Translate Template for the promotion field so that it uses a thermometer code. When the promo-

tion is factory (code value 4) then four inputs are turned on (to 1). This might help the network recognize the significance of the factory promotion more than just having a single one-of-4 inputs turn on with the one-of-N code representation.

We start another training run, and after 75 epochs the average RMS error is 0.06 and the maximum error is now only 0.165. The same network and same parameters were used. So it seems that changing the data representation of the promotion field helped. But sometimes just the difference in the initial random weights can improve (or worsen) the neural network prediction accuracy. We lock the network and switch over to the test data, the most recent six months of sales data. The network does well. The average RMS error is 0.068 and worst case is only 0.15. Because the network is converging well, we unlock the neural network weights, switch back to the training data, and resume training the network.

After another 75 epochs, the average RMS error is down to 0.047, while the worst case is still around 0.16 on the training data. This time we open an Inspector on the network output and the desired output and use an NNU time-plot view to see how closely the predictions are tracking the actual sales values. We also logged the actual and predicted values to a comma-delimited text file. Figure 12.3 is a graph of the actual versus predicted values. While the network is not catching all of the peaks (which look to be occurring when the end of month flag is on), it has definitely learned the basic sales curve. The prediction problems at the extremes might be caused by our decision to scale the output value to between 0.0 and 1.0, as we discussed in the data representation section.

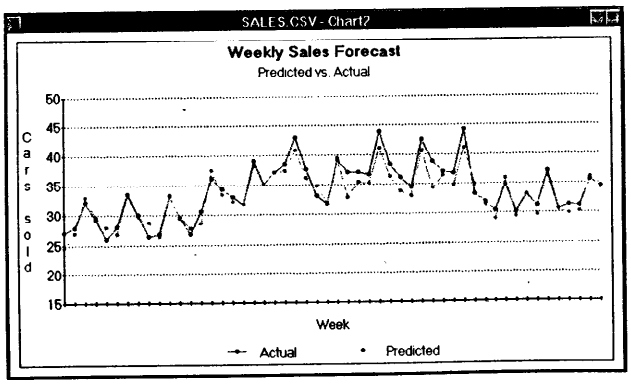


Figure 12.3 Graph of predicted versus actual sales.

Next we try a limited recurrent back propagation network to see how it performs. This type of neural network has a memory and should perform better than regular back propagation on this type of problem. The NNU implementation of the limited recurrent neural networks allows you to specify whether the feedback should come from the first hidden layer of units, or from the output layer. In this example, we use the first hidden layer. The effect is that our 25 hidden units are copied back to 25 extra input or context units. This network is obviously much larger than the back propagation network.

We train this network using the same data and training parameters as above. After 100 epochs, our network has an average RMS error of 0.049 and a maximum RMS error of 0.143. This is better than the back propagation network, but it also took longer to train because of the additional processing units and connection weights. For this time-series forecasting problem at least, the recurrence does not result in any significant improvement in performance.

However, this is one of the nice things about using commercial neural networks tools like NNU. It is very easy to try other neural network models to see if they give better performance than standard back propagation networks.

Deploying and Maintaining the Application

Now that we have an accurate sales forecasting model, the next step is to integrate it into our inventory control system and, to some extent, our staff scheduling system. The inventory control system is an online system that is already automated. However, the staff scheduling system is still done by Charley, the sales manager, who looks at last year's schedule and makes up a new one for this year.

To get a prediction from our neural network, we need to present the same information as during the training process, the prior week's data along with the context information for the next week. In addition, we have to construct the following week's context information and use our first estimate as input to the forecaster a second time. This will give us our forecast for two weeks out.

We have all of this information available, so this is not a problem. The data has to be preprocessed, run through the neural network, and then the output has to be scaled back into stock units or number of cars. If we use the Neural Network Utility to deploy this application, we can use the same application module as we used in the training process. We will have to use the NNU application programming interface (API) to load the application module and process the source data. Fortunately, this works well on our IBM AS/400 system and can be called by our RPG or COBOL programs.

Maintaining this application will be quite easy. Every month, we can add the last four weeks of sales information to the test data and move four more weeks of data into our training set. Training time takes about ten minutes

on a PC, or we can train the neural network on our AS/400 system during our weekend batch processing. As we use the forecasting system, we can see when it is accurate and when it has difficulty. If we notice some new factor that impacts sales, we can start tracking it and add it to our input data at some time in the future. Adding new input variables will require changing the architecture of the neural network and repeating the data mining process we followed in this chapter.

Related Applications and Discussion

Time-series forecasting is a very difficult problem. However, an accurate model can be used very profitably by a business. Mining historical data to discover trends has been one of the most popular uses of neural networks in the past decade (Vemuri and Rogers 1994). Neural networks' ability to model nonlinear functions make them particularly suited to time-series modeling.

Application of time-series prediction include stock price forecasting, electrical power demand forecasting, and sales forecasting. There is a large amount of literature on modeling dynamic systems for control, which is a very similar problem (Narendra 1992). Forecasting the future state of a system is also required for building stable controllers for complex systems, such as computer operating systems (Bigus 1993).

In this example, we used a technique known as the "sliding window" to present past information to the neural network so that it could predict ahead, into the future. The function we were modeling, while nonlinear, was relatively simple and required information about one prior state in order to be modeled with reasonable accuracy. Some cases might require five or even ten past states in order for the neural network to capture the dynamics. Note that we did not attempt to give the neural network a sequence of weekly sales numbers without providing any context variables. The information provided by the end-of-month flag and the type of promotion gave the neural network important clues as to the future direction of the sales. Otherwise, the network would have seen unexpected blips at seemingly random times (but we know that it was the end-of-month sales rush, or a factory incentive). It is vitally important to give the neural network whatever contextual information is available. Predicting the future is hard enough, even with all of the available information.

Summary

In this chapter, we created a neural network forecasting tool to predict weekly sales at an automobile dealer. We provided information on the current week's sales and context information, such as planned promotions, for the future week. We computed time of year and time of month information from the date and tried several data representations in our application. The IBM Neural Network Utility was used to perform the data pre- and postprocessing and to train and test our neural networks. Our forecasting model was able to predict with an average accuracy of greater than 95%. We identified that factory promotions caused extreme fluctuations in the weekly sales figures.

Mining historical data to build time-series forecasters is one way of learning from past experience. Neural networks can discover hidden relationships in temporal data and can be used to develop powerful business applications.

References

Bernard, G.D., V.S. Prakasam. 1994. A neural network application—recognizing items that have reached their reorder threshold in grocery wholesaling, IEEE International Conference on Neural Networks, Vol. 6., pp. 3655–61.

Bigus, J.P. 1993. Adaptive operating system control using neural networks, Ph.D. Dissertation, Lehigh University, UMI Dissertation Services.

Duke, L.S., J.A. Long. 1992. Neural network futures trading, a feasibility study, Adaptive Intelligent Systems - Proceedings of the BANKAI Workshop 1992, pp. 121–32.

Francella, B.G. 1995. Are overstocks costing you an arm and a leg?, Convenience Store News, July 11, 1994, pp. 1.

Narendra, K.S. 1992. Adaptive control of dynamical systems using neural networks, in *Handbook of Intelligent Control*, White and Sofge (eds.), New York: Van Nostrand Reinhold, pp. 141–183.

Nelson, M., T. Hill, B. Remus, M. O'Connor. 1994. Can neural networks applied to time series forecasting learn seasonal patterns: an empirical investigation, Proceedings of the Twenty Seventh Hawaii International Conference on System Sciences, Vol. III, pp. 649–55.

Ruggiero, M.A. 1994. How to build an artificial trader, *Futures: The Magazine of Commodities & Options*, Vol. 23, No. 10, Sep., pp. 56–58.

Vemuri, V.R. and R.D. Rogers (eds.). 1994. *Artificial neural networks: forecasting time series*, Los Alamitos, CA: IEEE Computer Society Press.

Appendix

A

IBM Neural Network Utility

The example applications in Part 2 of this book were developed using the IBM Neural Network Utility running under OS/2 Warp. This appendix describes the major features of the product and how they relate to the data mining methodology described in Part 1.

Product Overview

The IBM Neural Network Utility (NNU) is a family of six products that runs on four different operating systems: AIX on IBM RISC System/6000 systems, OS/400 on IBM AS/400 systems, and OS/2 and Microsoft Windows 3.1 running on personal computers. The Neural Network Utility is a tool for neural network data mining and application development. Version 3.1 of NNU, released in December, 1994, provides support for data preparation, several types of neural network data mining algorithms, and graphical views for data analysis and visualization of the data mining outputs (Bigus 1995).

First brought to market in 1990, the Neural Network Utility was one of the first commercial neural network tools aimed at business users. Designed and developed in IBM's Rochester, Minnesota programming lab, NNU began in 1988 as an advanced technology study to determine whether neural networks would be useful in commercial business processing environments such as the IBM AS/400 systems (Bigus 1991). NNU was originally developed in Pascal, then was redesigned and ported to a combination of C and Smalltalk, and currently is implemented in C and C++. NNU supports multiple platforms with a common code base. Approximately 95% of the source code is shared across platforms.

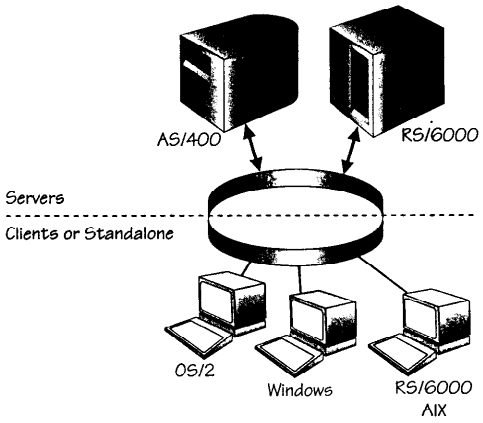


Figure A.1 Neural Network Utility product family.

Figure A.1 shows the platforms and development modes of operation supported by NNU 3.1. OS/2 and Windows can be used as stand-alone development environments, or they can be used as clients in a distributed client/server mode. In client/server mode, the NNU clients display the graphical user interface, but the data resides and the processing is done on the server machine. Both IBM RISC System/6000 (including SP2) and IBM AS/400 machines can be NNU servers, providing data and compute power to the clients.

The Neural Network Utility development environment allows the graphical construction of data mining applications. This includes specifying multiple data sources, multiple data transformation steps, and a mixture of neural network data mining functions and fuzzy rule base processing. The NNU environment is extensible, allowing custom neural network models to be added and other types of data mining and data transformation filters.

Interactive Development Environment

The Neural Network Utility provides the Interactive Development Environment (IDE) for training and testing neural networks and for developing fuzzy rule systems (IBM 1994a). The IDE uses a visual editor to allow the user to graphically connect NNU objects to specify data flow and processing (see Figure A.2). There are several kinds of NNU objects. Imports and

Exports support database and file I/O. Filters are used to transform the data, either by using an NNU Translate template, which specifies data transformations, or a program supplied by the user. Network objects provide neural network processing. Fuzzy rule bases are used to add knowledge-based processing using a fuzzy inference engine. SubModules allow the hierarchical construction of applications using other NNU application modules. Script objects provide a simple procedural language for automating the training and testing of neural networks.

In addition to the Module editor, the NNU IDE provides several special-purpose editors: a Translate editor for specifying data transformations, a Fuzzy Rule editor for creating fuzzy rule bases and fuzzy sets, and a Script editor for creating control scripts. Additional functions in the IDE are the application generation feature, which lets the user select a problem type and then generates the NNU module for that function, and Inspectors for visualization of objects in the NNU application module.

Data Preparation

The Neural Network Utility supports several phases of the data preparation process. In stand-alone mode on a PC or workstation, NNU can access data from flat files, spreadsheets, and relational databases (DB2/2 under OS/2, and any ODBC compliant database under Microsoft Windows 3.1). In client/server mode, NNU can access remote data from IBM AS/400 systems and IBM RS/6000 systems. The NNU Translate Filter provides support for simple data cleansing operations through default symbol substitutions for cat-

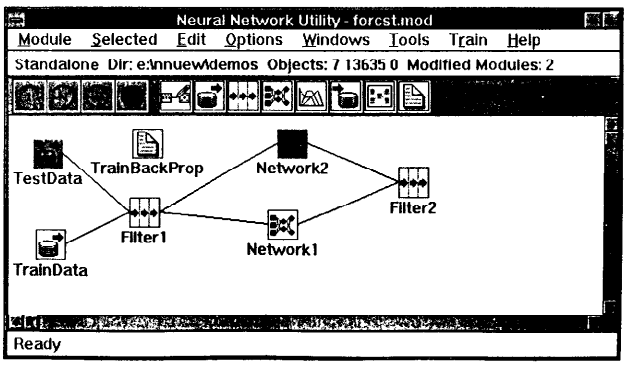


Figure A.2 Neural Network Utility module editor.

egorical data, and thresholding to remove outliers for numeric data when the valid range of values is known.

The NNU Translate Filter provides a rich set of operations for data pre- and postprocessing. Six logical data types are supported, including symbols, numbers, vectors of numbers, one-of-N codes, binary codes, and thermometer codes. The Translate Filter supports data type conversions so that numeric values can be converted into the coded types for presentation to the neural network and can be converted back from codes to numbers on the output side. Symbol-to-number and number-to-symbol conversion is handled through lookup symbol tables. Taxonomies can be created through symbol-to-symbol mappings using the same mechanism.

A single NNU Translate Filter can perform a preprocessing operation on data, then do a logical data type conversion, and then perform a postprocessing operation on the output data. For example, a symbol could be mapped into a discrete numeric value as a preprocessing operation. This number could then be converted into a thermometer code of the required length. This binary string could then be turned into a bipolar value (+0.5 to –0.5) through a scale operation as the postprocessing operation. In NNU, multiple Translate Filters can be strung together to perform an extremely complex series of data transformations.

The pre- and postprocessing operations include transcendental functions, bitwise operations, division and modulus, ceiling, floor, piecewise linear scaling, thresholding with multiple ranges, rounding, truncation, and more. For symbols, strings can be cast into all upper or lowercase and mapped into other symbols. Numeric vectors can be normalized using three different norms.

When the same operation is required on multiple fields in a record, the NNU Translate template has a replication parameter. Using this feature, for example, a 100-question, multiple-choice survey could be represented by a single Translate template entry. Symbols A–F could be mapped into 1–5 and then either scaled to the range 0 to 1 or translated into a coded type.

The NNU Translate Filter functions can be used as a one-time preprocessing operation or as a transformation to be applied to each record before it is presented to the neural network. All of this preprocessing function is available to the data analyst through a template-based Translate Editor (shown in Figure A.3) with no programming required. One function that NNU does not provide is a computed attribute, which is derived from a combination of two other fields.

Through its Application Generation function, NNU can handle data set management, allowing the splitting of a source data set into separate training and testing sets with user-specified percentages. The NNU Generation function scans the source data file, automatically specifies scaling operators for numeric fields, and creates symbol tables for categorical fields.

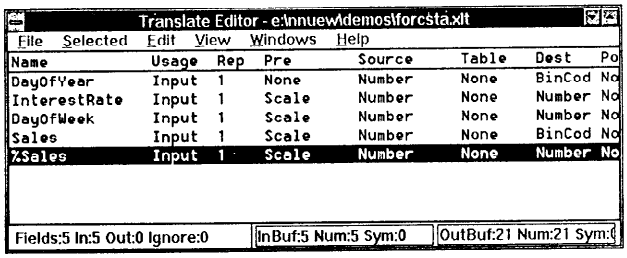


Figure A.3 Neural Network Utility translate editor.

The application generation function first appeared in Version 2 of NNU. It allows a user to specify the basic problem type and the source data file, and then it automatically prepares the data, creates a translate template for it, and selects and generates a neural network model and architecture for the problem. The end result is an NNU application module that is ready for the training and testing stage.

The source data can be either a flat file, a local database file (DB2/2 or an ODBC compliant database), or a remote database. The user specifies how the data should be split into train/test sets by percentages and also selects whether the source data should be automatically translated and saved in an encoded form. This option is for performance, since it requires only a single pass through the data and the data preprocessing is completed. The alternative is to translate or preprocess each record before it is presented to the neural network. This means that the same record keeps getting translated over and over during the training process.

NNU will scan the source file and find the minimum, maximum, and mean of the numeric fields, and set up symbol mapping tables for categorical or discrete symbolic fields. The default is for all data to be scaled to an input range of 0 to 1, although this can be changed by the user to whatever range is required by the neural network model.

Neural Network Models and Architecture

The Neural Network Utility provides seven popular neural network models and includes an open application programming interface for adding custom or user-defined models. These models include back propagation, limited recurrent back propagation, radial basis functions, adaptive resonance networks, Kohonen feature maps, temporal difference learning, and routing networks. The following sections briefly describe the model implementations in NNU.

Back propagation networks

NNU provides a fairly standard implementation of Rumelhart, Hinton, and Williams definition of the back propagation algorithm. It provides learn rate, momentum, and error tolerance to control training. The activation function is the standard logistic with an option for symmetric (bipolar) outputs ranging from –0.5 to +0.5 and a variable temperature parameter that controls the slope of the function. Back propagation networks can be constructed with 0, 1, 2, or 3 hidden layers.

Recurrent back propagation networks

The recurrent back propagation model in NNU is based on the standard NNU back propagation network. An architecture selection parameter is specified when the neural network is created to indicate whether feedback is to be from the first hidden layer or from the output layer. On the forward pass, the activations of the hidden layer or output layer are copied to the input context units. A decay parameter is provided to control the nature of the feedback signals. Recurrent back propagation is identical to back propagation in all other respects.

Kohonen feature maps

The NNU self-organizing feature maps implement the standard Kohonen specification using a Euclidean distance metric, a square neighborhood function, and linear reduction in learn rate over time. An alternative method uses a Gaussian neighborhood function with exponential decay in the learn rate. The user can specify a square or rectangular two-dimensional grid for the output layer.

Adaptive resonance networks

The NNU adaptive resonance theory networks are loosely based on Grossberg's ART algorithm (1987). It has been modified so that it accepts both binary and real input values. The vigilance parameter is the primary means for controlling the training process. Parameters are provided to indicate whether all of the output units have been committed during training.

Radial basis function networks

The NNU radial basis function networks provide several modes of operation. The hidden layer units can use either a Gaussian, thin plate spline, or multiquadratic basis function, and they can either be normalized or not. The basis center weights can either be set explicitly (they can be found using Kohonen feature maps as a preliminary step), or a variation of k-means clustering can be used to automatically determine the centers. The weights to the output layer are adjusted using the delta rule.

Temporal difference learning networks

NNU provides an implementation of temporal difference learning network based on Sutton's definition (1988). This network does not use the standard "desired minus actual" formula to compute the errors. It takes the difference between the previous output and the current output. A lambda parameter is provided to control the amount of effect the past has on the total error term. Temporal difference networks can be used to create adaptive critics using NNU.

Training and Testing Support

The Neural Network Utility provides several features to aid in the training and testing of neural networks used in data mining applications. To automate the training process, NNU provides a scripting language. NNU also provides visualization tools for viewing the neural network parameters and arrays as the training progresses.

Scripting

The Neural Network Utility scripting language has features such as variables, if-then-else statements, macros, while loops, and comments. The language has commands for controlling the data source, locking or unlocking the network weights, setting training parameters, logging data, and turning Inspectors on and off. Variables can be used to build "generic" training scripts, which can be used over and over. The macro construct allows common sequences of operations to be grouped together and called as a subroutine. The if-then-else and while-loop control structures make it easy to implement features such as save-best-net using any criteria the user wants. Switching between training and testing data sources and conditionally logging test results are operations that can be automated using NNU scripts.

The NNU Script Editor, shown in Figure A.4, provides context-sensitive help for constructing valid control script statements. NNU scripts are regular text files, so they can also be written using the text editor of your choice.

Once a script is developed for a particular application, the script can be shipped as part of the NNU application module, allowing automated retraining of the network in the field. Thus scripts are useful not only for use in the initial training process, but also can be used to maintain a neural network once it is deployed.

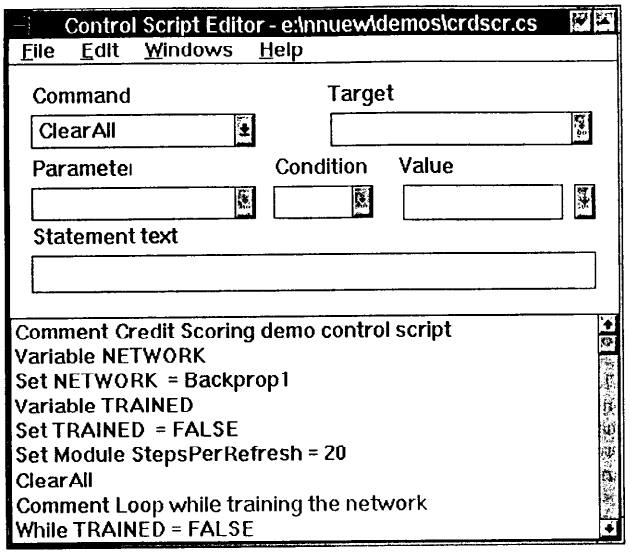


Figure A.4 Neural Network Utility script editor.

Fuzzy Rule Systems

NNU provides a fuzzy rule editor and fuzzy inference engine for doing rule-based processing in conjunction with neural networks. The user must define fuzzy variables and their associated fuzzy sets, and then specify a set of fuzzy if-then rules for processing the data.

Fuzzy variables can be either discrete or continuous. Discrete variables can either be numeric or symbolic. The continuous variables might have multiple fuzzy sets defined over their domain. Fuzzy sets can be either trapezoidal, triangular, rectangular, or made up of arbitrary line segments.

Fuzzy rules are made up of one or more antecedent clauses and a single consequent clause. Each clause is a fuzzy statement of the form "FuzzyVar is FuzzySet" or "FuzzyVar = constant." The rule base can contain rules that depend on other rules in order to be valid. For example, one set of rules could compute the value of variable A. Another set of rules could use fuzzy variable A in their antecedents, in order to compute the value of the result or output variable B. As each consequent variable is computed, it is defuzzified before it is used in the next set of rules.

The Fuzzy Rule editor, shown in Figure A.5, allows the user to graphically define fuzzy variables, fuzzy sets, and fuzzy rules. The fuzzy sets over a fuzzy variable are shown graphically so the user can easily see if the entire domain is covered. For a more detailed discussion of fuzzy logic and fuzzy systems, see appendix B.

Visualization and Analysis

The Interactive Development Environment provides a set of visualization and analysis tools called Inspectors. Inspectors can be opened on an NNU object to view either data in buffers, neural network parameters and arrays, or fuzzy rule system variables and fuzzy set values.

NNU Inspectors allow the user to select a variety of data, depending on the NNU object being inspected. For neural networks, one or more network parameters can be selected. For fuzzy rule bases, fuzzy variables and fuzzy

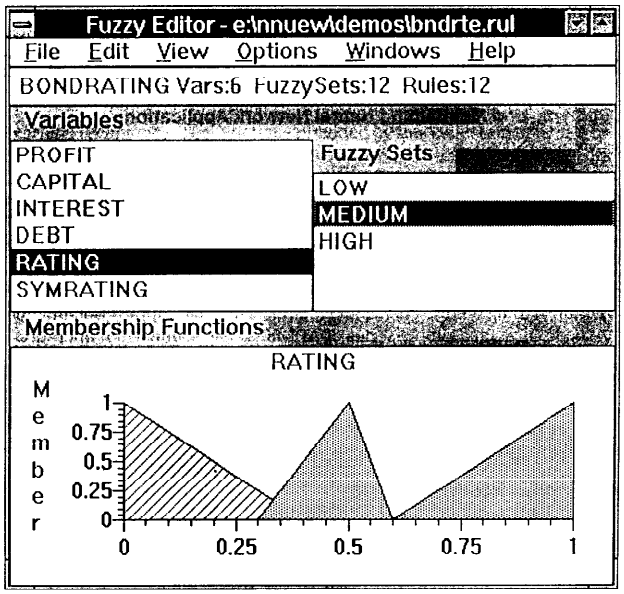


Figure A.5 Neural Network Utility fuzzy rule editor.

rules can be chosen. All objects can display the input and output buffers, and using the series function, up to six independent data items can be specified for inspection.

The NNU Inspectors provide several additional functions besides displaying text and graphical views of information. Any data selected for Inspection can also be logged to a text file. Field labels can be loaded from Translate templates so they are readily identified in text displays.

The Inspector windows can display the selected data using several different graphical representations. Time plots are useful for charting the evolution of errors over time. Hinton diagrams provide an effective way to examine weight arrays or outputs of clustering operations. Line plots and bar charts can be used to visualize classification outputs. Scatter plots help show correlations between two variables. Histograms are used to display the distribution of input data.

Special network graphic views depict the layout of the neural network processing elements and their connections. Fuzzy variables and fuzzy rules can be displayed either in textual form or graphically, by drawing the fuzzy membership functions. Using the network analysis view, combinations of neural network parameters and arrays can be displayed or logged for further analysis.

Deploying and Maintaining Neural Network Applications

The Neural Network Utility provides two mechanisms for deploying application modules developed with NNU. The first is the Application Delivery Environment, which is a simple shell program that allows the user to specify input and output dialogs for processing by an NNU module. The ADE can also process batch files. The second and most commonly used is the NNU application programming interface (API).

The NNU API is a language neutral API with approximately 80 callable functions (IBM 1994b). The user can embed any NNU application module into code through a simple series of API calls. The sequence of operations includes:

1. Initialize the NNU API (INITNNUAPI).
2. Create an NNU module object (CREATEMODULE).
3. Load an NNU application module (OPENMODULE).
4. Processing Loop
   - Set the input data (SET...ARRAY).
   - Step the module to process the data (STEPMODULE).
   - Get the output data (GET...ARRAY).
5. Close the NNU API (CLOSENNUAPI).

NNU saves application modules, network definitions, translate templates, and control scripts as text files. During client/server development, these files are automatically moved from the client up to the server for processing, and then brought back down when the development session is complete. To move a deployed application from one NNU platform to another requires that the text files be moved from machine to machine, and that the NNU API is installed on the target machine.

Summary

The IBM Neural Network Utility provides many of the functions required for successful data mining applications. The NNU Translate Filters can be used for data cleansing and for performing the data transformations required by neural networks. The scripting language allows automated training and testing sequences to be defined and reused. Knowledge-based processing in the form of fuzzy rules can be combined with neural networks to develop applications.

Several different neural network models are provided with NNU. Back propagation, recurrent back propagation, and radial basis function networks can be used for supervised learning. Kohonen feature maps and adaptive resonance networks provide support for unsupervised learning applications. Temporal difference networks are trained using reinforcement learning techniques. The functions provided by this collection of neural network models include classifications, clustering, modeling, and time-series forecasting.

The NNU Interactive Development Environment is a graphical editor for constructing applications using NNU objects. Additional graphical editors are provided for specifying data translation, scripts, and fuzzy expert systems. Inspectors allow graphical analysis of input data and neural network state information during the data mining process. Data logging is supported to allow analysis with other visualization tools. The NNU Application Delivery Environment and the Application Programming Interface allows the deployment of neural network applications.

References

Bigus, J. 1991. Neural networks: teaching the AS/400 new tricks, *NEW 3X/400 Magazine*, Jan., pp. 54–62.

Bigus, J. 1995. Neural Network Utility/400, *AS/400 Magazine*, July/August, pp. 103–104.

Grossberg, S. 1987. Competitive learning: from interactive activation to adaptive resonance, Cognitive Science 11, pp. 23–63.

IBM Corp. 1994a. Neural Network Utility: User's Guide, SC41–0223.

IBM Corp. 1994b. Neural Network Utility: Programmer's Reference, SC41–0222.

IBM International Technical Support Centers 1994., Neural networks in financial forecasting using IBM C++ and the Neural Network Utility for OS/2, ZZ81–0353.

IBM International Technical Support Centers 1993. Artificial intelligence and the AS/400: neural networks and knowledge-based systems, GG24-3793.
Kimmel, D. 1991. Neural networks inch toward mainstream, *Systems 3X/400 Magazine*, Oct., P53-58.
Rodriguez, S.M. 1994. Neural networks demystified, *Systems Magazine*, Feb., pp. 62-68.
Sutton, R.S. 1988. Learning to predict by methods of temporal differences, Machine Learning 3, pp. 9-44.

Appendix

B

Fuzzy Logic

"So far as the laws of mathematics refer to reality, they are not certain. And so far as they are certain, they do not refer to reality."
ALBERT EINSTEIN

The world is a fuzzy place. Although people like to think of things as clear cut, black and white, this is more of an artifice of Western ways than a natural phenomena. Bart Kosko opens his book, *Fuzzy Thinking* (1993), with the story of an apple being eaten. It starts out shiny and whole. Undeniably it is an apple. As bite after bite is taken, it becomes less and less recognizable as an apple. Until finally, it is completely gone. At what point did the apple turn into nonapple? To Kosko, the apple being eaten is a fuzzy apple. After the first bite, it is in the gray area between wholeness and nothingness. He states, "Fuzziness is grayness."

If this seems like a familiar point, I made very much this same distinction back in chapter 2 when I discussed how digital computers work with binary logic and how neural networks compute an analog value or degree of match ranging from one extreme to the other. There is a natural synergy between neural networks and fuzzy logic. Zadeh (1994) states that fuzzy logic is concerned with imprecision, while neural networks deal with learning, and probabilistic reasoning (which includes genetic algorithms) focuses on uncertainty. These three disciplines together form the basis for what he calls "soft computing" (also often called natural computing). They compose a set of complementary approaches to intelligent computing.

In this appendix, I present an introduction to fuzzy sets, fuzzy logic, and fuzzy rule systems. I also examine some of the ways that fuzzy logic and neural networks have been combined synergistically.

Introduction

When you hear the term fuzzy logic, what image does it bring to mind? Usually someone says something like "Oh, that's what we use around here!" Unfortunately, the English word "fuzzy" has very negative connotations. It means unclear, imprecise, not well thought out. Combine the term "fuzzy" with "logic," and you get a seeming contradiction in ideas. When people think of logic, it is the ultimate in precision, rock-solid, indisputable. To be logical is perhaps the highest compliment you can give to a scientist or engineer.

When we say "logic," we are usually referring to Aristotle's logic, first developed in 300 A.D. Aristotelian logic is the basis of much of Western thought. It has been studied and explored by thousands of scientists and philosophers since its inception. It is based on a single, simple but all-encompassing idea: A statement is either true or false. It is a binary logic allowing only two values, either something is true or it isn't. There is no middle ground. This is both a strength and a weakness of Aristotelian logic.

Fuzzy Sets

The concept of fuzzy sets was first proposed by Dr. Lotfi Zadeh, then the head of the electrical engineering department at the University of California at Berkeley, in 1965. There was a strong reaction from other scientists against fuzzy logic. Some of it was a reaction to the label, and in some sense it is unfortunate the Zadeh chose the term "fuzzy." Another valid description is multivalued logic, which does not carry the negative connotations of "fuzzy" in the English language.

Zadeh's fuzzy sets are based on a simple extension of standard binary sets. In order to understand fuzzy sets, on which fuzzy logic and fuzzy rules are built, let's first review standard set theory.

A set consists of a group of items or objects. An object is either in the set or not in the set. For example, we can have a set of tall people and a set of short people. Larry, Curly, and Moe are tall people. Groucho, Zeppo, and Hippo are short.

```
Set A = { Larry, Curly, Moe }
Set B = { Groucho, Zeppo, Hippo }
```


We can define membership functions for each person mentioned previously:

```
member(A,Larry) = 1, or Larry is in set A.
member(B, Larry) = 0, or Larry is not in set B.
```

Similarly, we can define truth values for the other people. In order to be logical, we need to have a rule to determine who is tall and who is short: "Anyone over 6 feet in height is tall." Figure B.1 shows our rule in graphic form.

Now we have a new person, Jay, who is exactly 6 feet tall. Does he belong in the tall set, A, or in the short set, B? Our rule says tall people are "over 6 feet," so Jay is in the short set, B. It seems kind of arbitrary, doesn't it? But when using binary logic, we have to make a decision one way or the other, but not both.

"Aha," you say, "All we have to do is divide the population into more than two sets." We can have very short people, sort-of-short people, kind-of-short people, and short people. Also, a single person could be a member in more than one set. This is true. However, the basic point remains the same. A binary decision must be made: "Is Jay a member of the set or not?"

This example illustrates a major weakness of binary logic. It doesn't necessarily hold against the real world. In our world there are many cases where black-or-white, binary logic simply doesn't apply. Yet we make it apply, since much of our science is based on making this distinction.

In fuzzy sets, Zadeh introduced the idea that an item can have partial membership in a set. At the extremes, fuzzy logic is equivalent to binary logic. For example, someone 4 feet tall has a membership value of 0 in the fuzzy tall set. Another person who is 7 feet tall has a membership value of 1 in the fuzzy tall set. Compare the fuzzy set membership function in Figure B.2 to the binary set function shown in Figure B.1.

How about our 6-foot person, Jay? Well, he could have a membership value of 0.75 in the fuzzy tall set as shown in Figure B.2. His membership

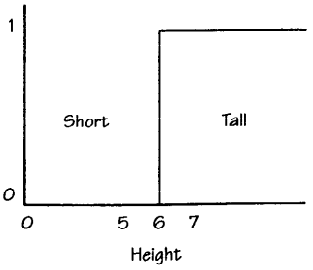


Figure B.1 Boolean tall set.

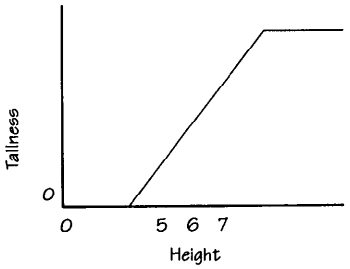


Figure B.2 Fuzzy tall set.

value in the fuzzy short set (not shown) could have a value of 0.2. In fuzzy logic, he doesn't have to be A or B, tall or short, one or the other. Jay can be a member of the fuzzy set A and fuzzy set B to some degree, which ranges from 0 to 1. This is the basic difference between binary sets and fuzzy sets. Instead of A AND not-A = ϕ, fuzzy set theory says that A AND not-A < > ϕ. Or a glass can be both half empty and half full.

In standard set theory, we can combine the elements of two sets with the union or AND operator. In fuzzy set theory, this is usually performed by taking the MIN operator on paired elements of both sets. In standard set theory, we can take the intersection or OR of two sets. In fuzzy set theory this is usually computed as the MAX of the paired set elements. There is also a fuzzy complement operation, which is usually computed as 1-x. So if x is 0.5, then "not x" is 0.5. Using fuzzy logic, the glass is half empty and half full!

Western society traditions are steeped in the idea of the yes/no binary logic. Far Eastern countries, notably Japan and China, do not have this bias. As a result, fuzzy logic, which was invented by a professor in an American university 30 years ago, is now estimated to be a two billion dollar industry in Japan, while it is just starting to gain attention in the United States and Europe.

Fuzzy Logic

One of the chief advantages of fuzzy logic is that it maps well to our intuitive understanding of the world. Let's go back to the example of tall people. Instead of a binary rule "IF person > 6 feet THEN person is tall ELSE person is short," we can use a fuzzy rule as shown in Figure B.2. Also, we can add modifiers (called linguistic hedges) such as sort-of tall, very tall, and extremely tall. These modifiers change the membership values of the items in predictable ways.

A 4-foot person is not tall, and so the member, (tall, 4 feet) = 0. However, a 7-foot person is clearly tall. A 5-foot, 6-inch person is sort of tall with a membership value of 0.4. A person 6 feet in height is tall with a membership

value of 0.8. By changing the shape of the membership functions, we can map from our linguistic world view to a mathematical functional view using fuzzy membership functions. There is nothing imprecise or fuzzy about fuzzy logic. It is mathematics, plain and simple.

Fuzzy or Linguistic Variables

In the previous section, I described the attributes of a fuzzy set called Tall. In using fuzzy sets for an application, we would typically need to define something called a fuzzy variable, or a "linguistic" variable. In this case, we would probably call our variable "Height." The linguistic variable "Height" does not just define a number, x inches, or x meters, but instead defines a concept. "Height" is the degree of tallness (or shortness). We can define a number of fuzzy sets over the continuum of "Height." We can have a fuzzy set called "Short," a fuzzy set called "Average," and another fuzzy set called "Tall." These fuzzy sets will overlap, since in the real world the words we use to describe things overlap. There is not some arbitrary cutoff between short and average, and average and tall. In addition, we can define and use linguistic hedges to these three fuzzy sets and have quite a lot of descriptive power, which is very natural for people to use and to understand. Zadah (1994) points out that this compression of data, from 20 to 100 inches into 3 functions is fundamental to the power of fuzzy systems. Zadeh calls this granulation. He differentiates this from quantization, or the breaking up of the domain of a variable into intervals. Granulation with linguistic variables is more general than quantization, and it matches the way people think about linguistic values. Furthermore, crossing the boundary between related linguistic variables is gradual, not abrupt. This leads to greater continuity and robustness.

Fuzzy Rules

One of the prime success areas of work in artificial intelligence has been in the form of rule-based expert systems. Expert systems consist of three parts: a set of if-then rules called the knowledge base; a program called an inference engine, which processes the rules and external input data; and a working memory area that is used to store information about the current state.

Many expert systems have been built for a variety of application areas. One of the problems associated with rule based systems is that the number of rules can quickly grow very large. For example, a rule-based configurator for the IBM 9370 system required over 1500 rules. Rule-based expert systems are based on binary logic. A rule is either true or false. If it is true, then we perform the then part. However, what if the rule is only partially true? Researchers in artificial intelligence tackled this problem by using confidence or certainty

factors. Thus variables in some expert systems have not only a current value, but also a corresponding certainty of the correctness of the value.

Fuzzy rule processing is a more natural approach to handling this uncertainty in rule-based systems. A rule is more or less true based on the values of the linguistic variables and their membership functions. For example, if we had a fuzzy rule set to control an air conditioner, we might have the following fuzzy rule set (Kosko 1993):

Rule 1. If the temperature is cold then motor speed is stopped.

Rule 2. If the temperature is cool then motor speed is slow.

Rule 3. If the temperature is just right then motor speed is medium.

Rule 4. If the temperature is warm then motor speed is fast.

Rule 5. If the temperature is hot then motor speed is cookin'.

This example has two linguistic or fuzzy variables, temperature and motor speed. For each fuzzy variable, we define one or more fuzzy sets. For example, five fuzzy sets on temperature are cold, cool, just right, warm, and hot. In order to map from these linguistic fuzzy rules into mathematics, we need to define the membership functions for each of the fuzzy sets we use.

Figure B.3 shows a diagram of the fuzzy membership functions defined for the temperature variable fuzzy sets. They are shaped either as triangles or as half triangles. Any functional form with an output range of 0 to 1 can be used. Triangles or trapezoids are most common.

In practical implementations, a fuzzy rule set is usually processed using the following three steps:

1. Fuzzification of the inputs. The raw input values (temperature) are mapped onto a range of 0.0 to 1.0 using the fuzzy membership functions.
2. All of the rules are fired in parallel. This results in a set of "truth" values for each rule in the rule set. The OR operation is usually taken on

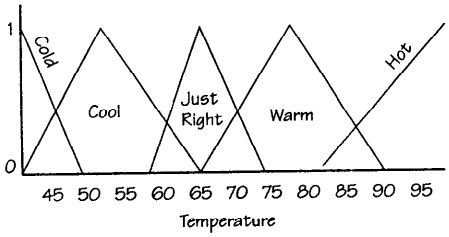


Figure B.3 Fuzzy sets for fan motor control.

these "truth" values and a single fuzzy output value (ranging from 0.0 to 1.0) is computed.

3. The output value is defuzzified, converted into a crisp numeric value (the process is called defuzzification). In our air conditioner example above, a fuzzy output value of 0.5 say, must be translated back into a voltage for the fan motor.

Linguistic variables and fuzzy sets provide an intuitive framework for domain experts to represent their knowledge. This expressiveness has allowed fuzzy rule systems to quickly develop commercially and compete with traditional Boolean logic-based expert systems. While expressing knowledge in fuzzy if-then format is easy, the mapping of the fuzzy sets onto membership functions is usually the hardest part of developing working fuzzy rule systems. Researchers have turned to neural networks to learn the appropriate membership functions.

Fuzzy Logic Meets Neural Networks

Fuzzy sets are functions that transform input values into a 0.0 to 1.0 output value. Neural processing elements transform a collection of inputs into a 0.0 to 1.0 output value. This correspondence has led many researchers to explore mappings from fuzzy logic to neural networks, and visa versa. Fuzzy neural networks are multilayer feedforward networks that use fuzzy logic in the processing units or the connection weight representations (Buckley and Hayashi 1994). The marriage of neural networks with fuzzy logic serves to produce a synergy that overcomes weaknesses in each of the respective technologies. By using intuitive fuzzy rules to represent knowledge and converting them into feedforward neural networks, we have a way of imparting explicit domain knowledge to neural networks, without the need for training (Okada et. al. 1992). This technique allows starting training of a neural network, which already has some degree of competence at the problem it is asked to learn. This results not only in better initial performance by the neural network, but also in faster learning.

The advantage from a fuzzy systems point of view is that by casting fuzzy rules into feedforward neural networks, we can use their learning ability to adjust and fine-tune the fuzzy membership functions (Ishibuchi, Fujioka, and Tanaka 1993). This solves the difficult and time-consuming process of defining the fuzzy sets by hand. Once the optimal fuzzy set definitions are learned by the neural network, they can be cast back into explicit fuzzy rule form. This turns out to be another advantage for neural networks because it solves the "black box" problem.

Summary

Fuzzy logic is a mathematical approach to dealing with the imprecise nature of everyday language and of the world around us. Fuzzy set theory extends the concepts of set membership from the binary all-or-nothing view of traditional logic to a more natural one, where items can have degrees of membership ranging from 0.0 to 1.0. The basic set operations of union, intersection, and complement have been redefined to work on fuzzy sets.

Fuzzy rule systems are a hybrid technology that combines the well-known if-then knowledge representation of traditional expert systems with the concepts of linguistic variables and fuzzy logic inferencing. Fuzzy rules can be used to represent initial knowledge, which can then be converted into neural network form. This technique can take advantage of neural network learning to fine-tune the fuzzy membership functions used in the rules, or as a method of initializing the neural networks with some minimal level of competency for a task.

The mixture of fuzzy logic, neural networks, and genetic algorithms (see appendix C) constitutes a powerful framework for solving difficult real-world problems.

References

Buckley, J.J. and Y. Hayashi. 1994. Fuzzy neural networks: A survey, Fuzzy Sets and Systems (Netherlands), Vol. 66, No.1, Aug. p1–13.

Ishibuchi, H., R. Fujioka and H. Tanaka. 1993. Neural networks that learn from fuzzy if-then rules, IEEE Transactions of Fuzzy Systems, Vol. 1, No. 2, May, pp. 85–97.

Kandel, A. (ed.) 1993. *Fuzzy expert systems*, Boca Raton: CRC Press.

Kosko, B. 1993. *Fuzzy thinking, the new science of fuzzy logic*, New York: Hyperion.

Kosko, B. 1992. *Neural networks and fuzzy systems: A dynamical systems approach to machine intelligence*, New York: Prentice Hall.

Mamdani, E.H. 1993. Twenty years of fuzzy control. Experiences gained., Second IEEE Conference on Fuzzy Systems, IEEE, pp. 339–344.

Okada, H., N. Watanabe, A. Kawamura, K. Asakawa, T. Taira, K. Isida, T. Kaji and M. Narita. 1992. Initializing multilayer neural networks with fuzzy logic, Proceedings of the IEEE International Conference on Neural Networks, Vol. I, pp. 239–250.

Zadeh, L.A. 1994. Fuzzy logic, neural networks, and soft computing, *Communications of the ACM*, 3, pp. 77–84.

Appendix

C

Genetic Algorithms

"I have called the principle, by which each slight variation, if useful, is preserved by the term of Natural Selection." CHARLES DARWIN

While neural networks are proposed as a model for the massively parallel, adaptive capabilities of the human brain, genetic algorithms are used as a metaphor for the powerful biological optimization process driven through genetics. Genetic algorithms can be used to simulate the process of natural selection in species. There are some interesting parallels with neural networks. Genetic algorithms also exhibit a natural emergence phenomena, where through wholly local actions, a global goal is approached or optimized.

Basics

Genetic algorithms are used to find solutions to problems that are encoded as a string of values or chromosomes (Holland 1975). A population of strings is created where each string represents a possible solution to the problem. A fitness function is defined to evaluate each string to determine its "fitness" or "goodness" as a solution to the problem. Based on their relative fitness scores, the population strings are used to create a new generation through the application of a set of genetic operations on the strings. These operators include crossover, which takes two parent strings to create a new child string, and mutation, which causes a random change in the genetic material of a single string. This new generation is then evaluated and the least fit individual strings "die." In this way, through the survival of the fittest, each generation of strings represents a better solution to the problem.

A set of control parameters is used to limit the size of the population, the probabilities of the various operators being used, and the chance that any individual string is selected for modification by an operator. These settings can have a profound effect on the quality of the solution produced by a genetic algorithm. In essence, genetic algorithms are performing a parallel search of the solution space of the problem. Two somewhat opposing goals must be pursued through the genetic process. The population needs to be diverse, so as to thoroughly search the solution space and find the true optimum solution to the problem. While at the same time, the population needs to be somewhat uniform so it can perform local optimization or hill climbing around a certain point in the solution space. These two goals must be balanced through appropriate control parameter settings to ensure that a good solution is found. The following sections explore the issues related to the key steps in the use of genetic algorithms.

Encoding

The most fundamental part of using genetic algorithms to solve an optimization or search problem is to encode the problem as a string. In most cases, the string is a binary string of 1s and 0s. Representing the problem as a binary string makes the application of the genetic operators very straightforward using bit manipulation operations. However, a large number of problems have real-valued continuous variables, which must be represented by a fixed-length binary string. While this coding of real values keeps the string in binary form, it introduces a lot of overhead in the evaluation function, since everything has to be converted back from binary codes to real numbers. When genetic algorithms are applied to neural networks, the string usually represents the connection weights, which are all real values. In this case, a more direct representation, first proposed by Montana and Davis (1988), is to encode the connection weights as a string of real values as illustrated in Figure C.1.

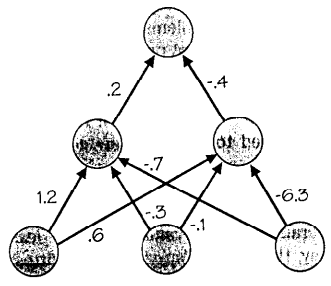


Figure C.1 Genetic algorithm encoding of a neural network.

Fitness or evaluation function

Once a problem is encoded as a string so that it can be modified with the defined genetic operators, a fitness or evaluation function must also be derived. This function serves as the basis for the selection process and so must accurately measure the goodness of the individual string. In most cases, this function is exactly the same as the objective function for the problem we are trying to optimize. The string is converted into a set of parameters for the function, and the function is evaluated. The answer is used as the fitness and as the basis for comparison with the other individuals in the population.

For neural networks, the fitness function would typically be the same measure used during the training process. For example, evaluating a string representing a back propagation network would involve doing a complete epoch of forward passes through the network, computing the root mean squared error, and using that as the fitness value. The lower the error, the "better" the neural network represented by the string. If the search space is the architecture of the network, then each string represents a different neural network. Each string is used to construct a neural network, which is then trained for some fixed number of epochs. Depending on the function being performed, their RMS error or the number of correct classifications would be used as their fitness.

Selection

The output of the fitness function is used as the basis for selecting which individuals get to procreate and contribute their genetic material to the next generation. There are several selection schemes that are widely used in the genetic algorithm literature (Srinivas and Patnaik 1994). In the proportionate selection scheme, each string's fitness value is normalized by dividing by the average fitness of the entire population. This number represents the number of expected offspring of the string. Another selection scheme is the roulette wheel approach. Each string is given a slot on a roulette wheel, with the size of the slot determined by the normalized fitness of the string. A random number is used to pick a slot on the wheel. The winner string is reproduced. This process continues until the entire next generation is created.

Genetic operators

Genetic operators are used to combine or modify individuals in the current population to create members of the next generation. The major operators are crossover and mutation. Depending on the encoding of the problem, problem-specific operators might also need to be defined.

Crossover. Crossover is a sexual operation where a pair of strings or parents are selected from the population to share their genetic material. A random

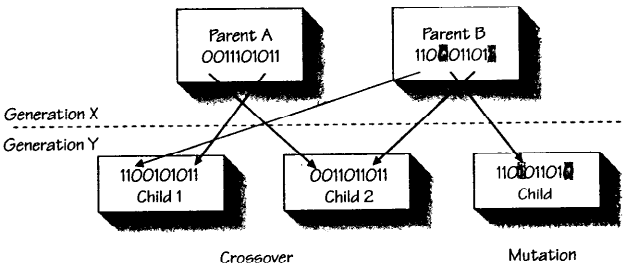


Figure C.2 Genetic operators: crossover and mutation.

point is selected as the crossover point where material from the first parent is replaced with material from the second parent (see Figure C.2). Because all strings are the same length, this is a simple operation. A control parameter, called the crossover rate, is used to determine whether the two strings selected actually produce a child or not. Since crossover controls the creation of entirely new individuals, the crossover rate parameter essentially controls the breadth of the search.

Mutation. Mutation is an asexual modification of a string where a single bit or string value is modified. A mutation rate is used to control the probability that an individual is changed. Since mutation will only slightly modify an individual, the mutation rate is in some sense a measure of the amount of local search that takes place. Figure C.2 shows an example of a mutation of a parent string.

The generation cycle

There are several alternate strategies used to create succeeding populations of individuals (Srinivas and Patnaik 1994). In one approach, all members of a population die and only live on through genetic contributions to the next generation. In another approach, a fit individual can live on and on through successive generations. Holland et. al. (1986) used this technique when applying genetic algorithms to the evolution of if-then rules in their classifier systems.

Applications to Neural Networks

There are several uses of genetic algorithms with regard to neural networks. They are used to search the connection or weight space, using evolution rather than training to adjust the neural network weights. Genetic

algorithms are used to explore the architecture space of neural networks to find the optimal number of hidden layers and hidden units. Genetic algorithms also can be used to find the best neural network model to solve a particular problem.

Finding network weights

While back propagation is the most popular learning algorithm used to train neural networks, it has the disadvantage of being slow to converge and somewhat sensitive to the settings of the learn rate and momentum parameters. One of the first applications of genetic algorithms to neural networks was to use them to search the weight space (Davis 1991). Thus the initial population contains a set of neural networks with identical architectures, but with different random initial weights. The genetic operators combine the genetic material (the connection weights) from various members of the population. Rather than moving single weights, special crossover operators are used to move coherent sets of weights, such as those weights leading into a single processing unit, between the neural networks. As the genetic algorithm evolves, the weights from the neural networks with the lowest prediction error would be propagated and shared with the new generation. This approach to training a neural network amounts to doing parallel, but directed, stochastic search. One group in IBM used genetic algorithms to train neural networks for use with the Neural Network Utility and reported excellent results, both in decreased training times compared to back propagation, and in the generalization abilities of the resulting neural network.

Finding the neural network architecture

Once a specific neural network model is selected for a problem, there are still the issues of how many hidden layers there should be and how many hidden units should be in each hidden layer. Since these are the most difficult design decisions to make when developing neural networks, genetic algorithms are often used to solve these problems. The initial population is chosen to include neural networks with a wide range of hidden layers and hidden units. The neural network-specific genetic operators must be able to deal with encoded strings of different lengths (i.e., networks with more hidden units will have more connection weights and so have longer strings). Another option is to have all members of the population coded as the largest possible network and to use zero weight values to indicate networks with fewer processing units. As the genetic algorithm evolves toward a solution, those neural networks that have the lowest average prediction errors will reproduce, and the population will converge to the optimum network architecture for the problem. Care must be taken, however, because if we do not use the testing data in the

evaluation function, the genetic algorithm will optimize for the training data only. This could result in overtraining of the neural network and poor generalization ability.

Selecting the neural network model

In chapter 4, I described the most popular neural network models and their functions. Often, it is not apparent from the data whether one type of supervised neural network should be used or another (for example, back propagation versus recurrent back propagation). Genetic algorithms can be used to search through the available neural network models by using several types of networks in the initial population. This heterogeneous population might contain several back propagation networks, radial basis function networks, recurrent back propagation networks, and others, all with the same number of hidden units. As the successive generations are produced, the neural network model that best applies to the problem will have more members in the population. Those that are not suited will die off or have only a few instances. At the end of the genetic algorithm run, the neural network with the best fitness function is the best neural network model for the job.

Summary

Genetic algorithms provide a way to find solutions to difficult optimization problems. The first step in using genetic algorithms is to encode the problem into a binary or real-valued string. Each string represents a single solution to the problem. Using the biological notion of the process of natural selection, an initial population of candidate solutions is generated (as encoded strings). Genetic operators, crossover, and mutation are probabilistically applied to members of the population, mixing the genetic material (partial solutions to the problem) and generating new members from the most "fit" members in the current population. As the population evolves, the encoded strings compete with each other, so that at the end of the run, the best solutions are left.

Genetic algorithms have been applied to neural networks in several ways. They are used to train neural networks by adjusting the connection weights. They are used to explore the architecture space to find the best number of hidden units for a particular problem. And they have even been used to select which type of neural network model should be used. The powerful optimization capabilities of genetic algorithms can be applied naturally to these common neural network development issues to provide a better fit between the application problem and the neural network solution.

References

Davis, L. 1991. *Handbook of genetic algorithms*, New York: Van Nostrand Reinhold.

Holland, J.H. 1975. *Adaptation in natural and artificial systems*, University of Michigan Press, Ann Arbor, Mich.

Holland, J.H., K.J. Holyoak, R.E. Nisbett, and P.R. Thagard. 1986. *Induction: processes of inference, learning, and discovery*, Cambridge: MIT Press.

Montana, D.J., and L. Davis. 1988. Training feedforward neural networks using genetic algorithms, BBN Inc., article preprint.

Srinivas, M., and L.M. Patnaik. 1994. Genetic algorithms: a survey, *IEEE Computer Magazine*, Vol. 27, No. 6, pp. 17–26.

Glossary

activation The current output value of a neural processing unit. Usually ranges from 0.0 to 1.0.

activation function A function used by the neural processing unit to compute the output or activation of the unit. Common activation functions include sigmoid, hyperbolic tangent, signum, and Gaussian.

adaptive resonance theory (ART) A recurrent neural network that has input and output layers joined by two sets of connection weights. ART uses an unsupervised training algorithm to adjust the weights to cluster input patterns based on their degree of similarity.

back propagation The most popular form of neural network. Back propagation networks are feedforward multilayer networks that use a supervised training algorithm (backward propagation of errors) to adjust the connection weights.

binary code A binary data representation for discrete data where each distinct category is assigned an integer value and coded as a standard binary string.

classification The process of learning to distinguish and discriminate between different input patterns using a supervised training algorithm.

clustering The process of grouping similar input patterns together using an unsupervised training algorithm.

connection weights Real-valued parameters that modify the signals flowing between neural processing units in neural networks.

crossover A genetic operator used to combine two parents to produce a new child.

data cleansing A processing step where missing or inaccurate data is replaced with valid values.

data mining The efficient discovery of nonobvious, valuable information from a large collection of data. The data mining process consists of data preparation, use of a data mining algorithm, and analysis of the mining output or results.

data preparation The first step of the data mining process, which includes data selection, data cleansing, data preprocessing, and data representation.

data preprocessing A processing step where data is combined to create new fields, or is otherwise transformed before it is input to the data mining algorithm.

data representation The transformation of the source data into a format acceptable to the data mining algorithm.

data selection The process where certain records or columns of data are chosen for processing in a data mining application.

data warehouse A large database where corporate data is kept for long-term online storage.

epoch A single pass through a complete data set while training or testing a neural network.

expert system A data processing system composed of a knowledge base (rules), an inference engine, and a working memory.

feedforward A neural network topology consisting of two or more layers of neural processing units, connected so that inputs flow in one end, flow through the connections and processing units in a single direction, and come out the other end.

fuzzy logic A system of logic based on fuzzy set theory.

fuzzy set A set of items whose degree of membership in the set may range from 0 to 1.

fuzzy systems A set of rules using fuzzy linguistic variables described by fuzzy sets and processed using fuzzy logic operations.

genetic algorithms A method for solving optimization problems using parallel search, based on the biological paradigm of natural selection and "survival of the fittest."

genetic operators The operations on population member strings in a genetic algorithm that are used to produce new strings.

intelligent agent A software application that assists a system or user by automating a task. Intelligent agents must recognize events and use domain knowledge to take appropriate actions based on those events.

Kohonen feature map A feedforward neural network that uses unsupervised learning to cluster or segment input data.

modeling The process of training a neural network to model a function or to relate a set of inputs to one or more outputs.

mutation A genetic operator that modifies a single population member by changing (mutating) portions of its encoded string.

neural network A computing model based on the architecture of the brain consisting of multiple simple processing units connected by adaptive weights.

neural processing unit A simple processor used in a neural network that takes the sum of the input signals coming into it and computes an output value or activation.

one-of-N code A binary data representation for discrete data where each distinct value is represented by a single 1 digit and the rest are 0 digits.

probabilistic neural network (PNN) A feedforward neural network trained using supervised learning that allocates a hidden unit for each input pattern.

radial basis function (RBF) A feedforward neural network trained using supervised learning and having a single layer of hidden units that use a Gaussian activation function.

recurrent A neural network topology where the units are connected so that inputs signals flow back and forth between the neural processing units until the neural network settles down. The outputs are then read from the output units.

recurrent back propagation A feedforward neural network trained using supervised learning and the back propagation learning algorithm. Limited recurrence is provided by a set of context units that take on the values of either the hidden layer unit or the output layer units from the prior input pattern.

reinforcement learning A training paradigm where the neural network is presented with a sequence of input data, followed by a reinforcement signal.

relational database A collection of data arranged in rows and columns and manipulated using relational algebraic operations.

self-organizing map See Kohonen feature map.

shared-nothing architectures A technique for parallelizing work in a computer system where multiple processors are connected, each having its own private memory and disk storage.

supervised learning A training paradigm where the neural network is presented with an input pattern and a desired output pattern. The desired output is compared with the neural network output, and the error information is used to adjust the connection weights.

symmetrical multiprocessing (SMP) A technique for parallelizing work in a computer system where a single task runs on multiple processors with shared access to a common main memory.

testing The process of determining whether the predictive accuracy of the neural network meets the requirements when the network weights are locked and test data is presented to the neural network.

thermometer code A binary data representation for symbolic or categorical data that has an implicit order such as good, better, best.

time-series forecasting The process of using neural networks to learn to predict temporal sequences of patterns so that when given a set of patterns, it can predict a future value.

training The process of adjusting the connection weights in a neural network under the control of a learning algorithm.

unsupervised learning A training paradigm where the neural network is presented with input data, and it self-organizes to cluster or segment the data by learning to recognize statistical similarities between the input patterns.

validation An independent test process whereby the performance of the neural network is tested against the acceptance requirements.

visualization The graphical display of data that helps the user in understanding the structure and meaning of the information contained in the data.

weight See connection weights.

Bibliography

Churchland, P.S. 1986. *Neurophilosophy: toward a unified science of the mind-brain*, Cambridge, MA: MIT Press, 546 pages. This is a book that tries to describe and resolve the differences between the top-down approaches of philosophy and symbolic artificial intelligence and the bottom-up or emergent approaches represented by neuroscience and neural network computing models. If you are interested in the deep philosophical questions concerning the human mind and intelligence, then this book will provide excellent food for thought.

Dayhoff, J. 1990. *Neural network architectures: an introduction*, New York: Van Nostrand Reinhold, 259 pages. This introductory text describes a biological approach to neural networks. Hopfield networks, back propagation, and Kohonen feature maps are described in detail, along with information about the biology of the human brain, neurons, and synapses. The book has plenty of illustrations, many taken from the original research papers. There is a discussion of neural network applications that is now somewhat dated. In general, a good introduction to neural network theory.

Hertz, J., A. Krogh, R.G. Palmer, 1991. *Introduction to the theory of neural computation*, Redwood City, CA: Addison-Wesley, 337 pages. This text provides a very thorough treatment of the mathematics and theory behind neural networks. The authors' approach is based on statistical mechanics, and they recommend that readers be familiar with multivariate calculus, ordinary differential equations, probability and statistics, and linear algebra. They cover Hopfield networks and their application to optimization problems, Perceptrons, multilayer networks, including back propagation, recurrent networks including Boltzmann machines and recurrent back propagation, and unsupervised learning. The book provides an extensive bibliography and is an excellent resource if you are interested in developing your own neural network code.

Jubak, J. 1992. *In the image of the brain: breaking the barrier between the human mind and intelligent machines*, Boston: Little, Brown, 348 pages. This book is a popular science account of the forces and personalities behind the latest trends in artificial intelligence: neural networks and artificial life. It provides a very gentle introduction to the fundamental concepts driving this research and gives the personal perspectives of some major researchers, including Carver Mead, David Rumelhart, and Chris Langton.

Kosko, B. 1992. *Neural networks and fuzzy systems: a dynamical systems approach to machine intelligence*, Englewood Cliffs, N.J.: Prentice-Hall, 449 pages. This text provides a thorough mathematical treatment of neural networks and fuzzy systems. It includes end-of-chapter exercises and demonstration software to go along with the exercises. As the subtitle indicates, the material is explored from the perspective of dynamical systems and therefore requires a strong background in calculus. Kosko covers supervised and unsupervised neural networks, provides a thorough mathematical derivation of back propagation, and discusses the stability issues of recurrent networks.In the section on fuzzy logic, he compares fuzziness to probability, covers fuzzy associative memories, and presents applications of fuzzy systems to image compression and adaptive control.

Kosko, B. 1993. *Fuzzy thinking: the new science of fuzzy logic*, New York: Hyperion, 318 pages. Kosko explains fuzzy logic concepts through a variety of anecdotes and personal epiphanies that caused him to question many of the basic tenets of Western science. More than just an introduction to the theory of fuzzy logic, this book presents the philosophy behind it and explores the ramifications of "fuzzy thinking" to the existing framework of science and mathematics. Requires no advanced mathematics training, but definitely requires an open mind.

McClelland, J.L. and Rumelhart, D.E. 1986. *Parallel distributed processing: explorations in the microstructure of cognition*, Volume 2: psychological and biological models, Cambridge, MA: MIT Press, 611 pages. If you are interested in how neural networks perform as models of human cognition, then this book will satisfy your curiosity. There are six chapters of cognitive psychology experiments, as well as several chapters on the biology of the brain. While many of these issues are still open to debate, this collection of papers represents the first unified statement from the connectionist crowd. Even if your interests lie toward more practical applications, this is interesting reading.

Mead, C. 1989. *Analog VLSI and neural systems*, Reading, MA.: Addison-Wesley, 371 pages. This text is aimed specifically at electrical engineers and those interested in understanding how neural networks can be implemented in silicon. As the title suggests, this book deals with analog computing and, in some sense, it is a tutorial for all of those EEs who have been working in the digital world so long that they forgot, or never learned, about the intricacies of designing and building analog circuitry. Mead covers a lot of material, from device physics to neurons, and describes his implementation of a silicon retina and electronic cochlea. Mixed in with the technical discussion are his philosophy about the nature of intelligence and the requirement this places on hardware that tries to mimic this intelligence.

Miller, W.T., R.S. Sutton, and P.J. Werbos (eds.). 1991. *Neural networks for control*, Cambridge, Mass: MIT Press, 524 pages. This book contains a collection of papers from a 1988 workshop on neural network applications to robotics and control. With contributions from Andy Barto, Paul Werbos, Ron Williams, Kumpati Narendra, and Rich Sutton, there is good coverage from both neural network experts and control system theorists. With a focus on modeling real dynamic systems and deriving neural network controllers, many of the ideas presented here can easily be applied to intelligent agents.

Rich, E. and K. Knight. 1991. *Artificial intelligence*, second edition, New York: McGraw-Hill, 619 pages. Probably the most readable and usable textbooks on artificial intelligence, this book covers both traditional symbolic AI approaches as well as learning and connectionist (neural network) techniques. Organized into three sections dealing with problems and search, knowledge representations, and advanced topics including game playing, planning, understanding, and expert systems, this book nearly covers it all. Most of the "standard" AI problems are explored, and the end of chapter exercises and the bibliography make this an excellent reference.

Rumelhart, D.E. and J.L. McClelland. 1986. *Parallel distributed processing: explorations in the microstructure of cognition*, Volume 1: foundations, Cambridge, MA: MIT Press, 547 pages. The "bible" of neural networks and connectionism, PDP volume 1 presents the philosophical argument for neural networks (parallel distributed processing) as well as provides the theoretical basis for much of the work that has occurred since it was published. Competitive learning, dynamics of neural networks, and the "standard" definition of back propagation are all in this volume. This book includes an excellent introduction to linear algebra and how it relates to neural computation. Still a "must have" book if you are serious about neural network research.

Wasserman, P.D. 1989. *Neural computing: theory and practice*, New York: Van Nostrand Reinhold, 230 pages. One of the first textbooks on neural networks, Wasserman takes the mishmash of neural network terminology and notation and describes the major neural network architectures using a standardized approach. Perceptrons, Hopfield networks, bidirectional associative memories, back propagation, adaptive resonance theory and counter propagation are all covered. The mathematics is not too difficult, and there is an appendix on linear algebra for those who need it. For people coming from an engineering background, this is an excellent introductory text.

Wasserman, P.D. 1993. *Advanced methods in neural computing*, New York: Van Nostrand Reinhold, 255 pages. Described as the "next logical step" from his first book, Wasserman uses the same basic approach. He provides comprehensive treatments of field theory methods, probabilistic neural networks, and radial basis function networks. Some of the advances over the basic back propagation algorithm are presented, along with chapters on genetic algorithms and fuzzy logic. Like his first book, this is an excellent textbook on some of the latest trends in neural networks.

White, D.A. and D.A. Sofge (eds.). 1992. *Handbook of intelligent control: neural, fuzzy, and adaptive approaches*, New York: Van Nostrand Reinhold, 568 pages. This book presents a broad introduction to the subject of intelligent control, and features an excellent overview chapter on the subject written by Astrom and McAvoy. The topics include fuzzy control systems, control of dynamical systems, and optimal control. Applications covered include control of manufacturing and chemical processes and hypersonic aircraft. Definitely worthwhile if you have an engineering background and are interested in taking control of key business processes.

Related Articles

Amoroso, A.J., M.A. Garwood. 1995. Database mining offers competitive advantages, *National Underwriter Life & Health*, June 5, pp. 32.

Classe A. 1995. Caught in the neural net, Accountancy (UK), Vol. 115, No. 1218, Feb., pp. 58–9.

Costanzo, C. 1995. Best practices in commercial lending, *Bank Technology News*, July, pp. 1.

Curry, B., L. Moutinho. 1993. Using advanced computing techniques in banking, International Journal of Bank Marketing, Vol. 11, No. 6, pp. 39–46.

Dasgupta, C.G., G.S. Dispensa, S. Ghose. 1994. Comparing the predictive performance of a neural network model with some traditional market response models, International Journal of Forecasting, Vol. 10, pp. 235–244.

Disney, D.R. 1995. Comment: for the real gold in customer data, dig deep, *The American Banker*, May 10.

Duggal, S.M., P.R. Popovich. 1993. Practical applications of neural networks in business, Journal of Computer Information Systems, Winter, 1992–1993, pp. 8–13.

Dutta, S., S. Shekhar, W.Y. Wong. 1994. Decision support in non-conservative domains: generalization with neural networks, Decision Support Systems, Vol. 11, No. 5, pp. 527–544.

Fenn, J., P. Hogdon. 1995. How emerging artificial intelligence technologies will affect direct marketing, *Direct Marketing* (USA), Vol. 58, No. 4, Aug., pp. 28–30.

Fletcher, D., E. Goss. 1993. Forecasting with neural networks: an application using bankruptcy data, Information and Management (Netherlands), Vol. 24, No. 3, March, pp. 159–67.

Fogler, H.R. 1995. Investment analysis and new quantitative tools, Journal of Portfolio Management, Vol. 21, No. 4, Summer, pp. 39–48.

Fox, B. 1993. Move over, expert systems: neural networks the new wave in AI, *Chain Store Age Executive*, Vol. 69, No. 4, Apr, pp. 65–68.

Francella, B.G. 1994. Are overstocks costing you an arm and a leg?, Convenience Store News, July 11, pp. 1.

Frawley, W.J., G. Piatetsky-Shapiro, C.J. Matheus. 1992. Knowledge discovery in databases: an overview, *AI Magazine*, Fall, pp. 57–69.

Garceau, L., C. Foltin. 1995. Neural networks: new technology for accountants & the impact on IS audit and control professionals, IS Audit and Control Journal, Vol.2, pp. 52–7.

Gessaroli, J. 1995. Data mining: A powerful technology for database marketing, *Telemarketing* Vol 13, No. 11, May, pp. 64–68.

Grudnitski, G., L. Osburn. 1993. Forecasting S&P and gold futures prices: an application of neural networks, Journal of Futures Markets, Vol. 13, No. 6, Sep., pp. 631–643.

Grupe, F.H., M.M. Orwang. 1995. Database mining: discovering new knowledge and competitive advantage, Information Systems Management, Vol.12, No.4, Fall, pp. 26–31.

Grupe, F.H., T. von Sadovsky, M.M. Owrang . 1995. An executive's guide to artificial intelligence, Information Strategy: The Executive's Journal, Vol. 12, No. 1, Fall, pp. 44–48.

Hammer, R.K. 1995. Comment: data models can make money out of chaos, *The American Banker*, March 21.

Haverson, P. 1993. We have the technology (artifical intelligence in banking), Banking Technology (UK), Vol. 10, No. 9, Nov., pp. 20–24.

Hruschka, H. 1993. Determining market response functions by neural network modeling: a comparison to econometric techniques, European Journal of Operational Research, Vol. 66, No. 1, Apr., pp. 27–35.
Hsieh, C.T. 1993. Some potential applications of artificial neural systems in financial management, Journal of Systems Management, Vol. 44, No. 4, April, pp. 12–15.
Kluytmans, J., B. Wierenga, M. Spigt. 1993. Developing a neural network for selection of sales promotion instruments in different marketing situations, Proceedings of the Second Annual International Conference on Artificial Intelligence Applications on Wall Street, pp. 160–4.
Lenard, M.J, A. Pervaiz, G. Madey. 1995. The application of neural networks and a qualitative response model to the auditor's going concern uncertainty decision, Decision Sciences, Vol. 26, No.2, Mar/Apr, pp. 209–227.
Li, E.Y. 1994. Artifical neural networks and their business applications, Information and Management, Vol. 27, No. 5, Nov., pp. 303–313.
Malhotra, D.K., R. Malhotra, R.W. McLeod 1994. Artificial neural systems in commercial lending, *Bankers Magazine*, Vol. 177, No. 6, Nov/.Dec, pp. 40–44.
McKie, S. 1995. Software agents: application intelligence goes undercover, *DBMS Magazine*, April, pp. 56–60.
Moore, K., R. Burbach, R. Heeler. 1995. Using neural nets to analyze qualitative data. *Marketing Research: A Magazine of Management & Applications*, Vol.7, No.1, Winter, pp. 34–39.
Norton M. 1994. Close to the nerve (credit card fraud), *Banking Technology* (UK), Vol. 11, No. 10, Dec., pp. 28–31.
O'Brien, T. 1994. Neural nets for direct marketers (software reviews), *Marketing Research: A Magazine of Management & Applications*, Vol. 6, No. 1, Winter, pp. 47–49.
Poh, H.L. 1994. A neural network approach for decision support, International Journal of Applications of Expert Systems, Vol. 2, No. 3, 1994, pp. 196–216.
Poh, H.L., T. Jasic. 1995. Forecasting and analysis of marketing data using neural networks: a case of advertising and promotion impact, Proceedings of the 11th Conference on Artificial Intelligence for Applications, pp. 224–30.
Port, O. 1995. Computers that think are almost here, *Business Week*, July 17, pp. 68–72.
Pracht, W.E. 1991. Neural networks for business applications, Interface: Computers in Education Quarterly, Vol 13., No. 2, Summer.
Proctor, R.A. 1992. Marketing decision support systems: a role for neural networking, Marketing Intelligence & Planning, Vol. 10, No.1, pp. 21–26.
Purcell, L. 1994. Roping in risk (neural networks in banks), *Bank Systems and Technology* (USA) Vol. 31, No. 5, May, pp. 64–68.
Robbins, G. 1993. Credit scoring: Can retailers benefit from neural networks?, *Stores*, Vol. 75, No. 4, April, pp. 34–35.
Ruggiero, M.A. 1994a. Interpreting feedack to build a better system, *Futures: The Magazine of Commodities & Options*, Vol 23, No. 8, July, pp. 46–48.
Ruggiero, M.A. 1994b. How to build an artificial trader, Futures: The Magazine of Commodities & Options, Vol 23, No. 10, Sep., pp. 56–58.
Sandler, S. 1993. Injecting more intelligence into mortgage lending, Bank Technology News, Nov., pp. 26.
Schocken, S., G. Ariav. 1994. Neural networks for decision support: problems and opportunities, Decision Support Systems, Vol. 11, No. 5, June , pp. 393–414.
Schwartz T. 1994. Rounding out intelligent systems, *Wall Street and Technology*, Oct., pp. 58–62.
Schwartz T. 1995. A tale of two traders, Wall Street and Technology, Jan., pp. 42.
Seitz, J., E. Stickel. 1992. Consumer loan analysis using neural networks, Adaptive Intelligent Systems—Proceedings of the BankAI Workshop, pp./177–192.
Sirin, I, H.A. Guvenir. 1992. Prediction of stock market index changes, Adaptive Intelligent Systems—Proceedings of the BANKAI Workshop , pp. 145–59.
Siriopoulous, C., S. Perantonis, G. Karakoulos. 1994. Artificial intelligence models for financial decision making, Information Strategy: The Executive's Journal, Vol. 11, No. 1, Fall, pp. 49–54.
Star, M.G. 1994. New AI contender looms: LBS goes against Fidelity in neural net mutual funds, *Pensions & Investments*, Oct. 17, pp. 48.

Tafti, M.H.A., E. Nikbakht. 1993. Neural networks and expert systems: new horizons in business finance applications, Information Management & Computer Security, Vol. 1, No. 1, pp. 22–28.
Udo, G. 1993. Neural network performance on the bankruptcy classification problem, Computers and Industrial Engineering, Vol. 25, No.1–4, pp. 377–380.
Venugopal, V., W. Baets. 1994a. Neural networks and statistical techniques in marketing research: a conceptual comparison, Marketing Intelligence & Planning, Vol 12., No.7, pp. 30–38.
Venugopal, V., W. Baets. 1994b. Neural networks & their applications in marketing management, Journal Systems Management, Vol. 45, No.9, Sep., pp. 16–21.
Verity, J.W., R. Mitchell. 1995. A trillion-byte weapon—marketers use massive power to woo customers, *Business Week*, July 31, pp. 80–1.
Wilson, R.L., Sharda, R. 1994. Bankruptcy prediction using neural networks, Decision Support Systems, Vol. 11, No. 5, June, pp. 545–557.
Zahavi, J., N. Levin. 1995. Issues and problems in applying neural computing to target marketing, Journal of Direct Marketing, Vol 9. No. 3, Summer, pp. 33–45.

Index

ABOUT THE AUTHOR

Dr. Joseph P. Bigus is the architect and development team leader of the IBM Neural Network Utility product family, based in the IBM AS/400 programming laboratory in Rochester, MN. He has more than eight years of experience working with neural networks and their applications. Dr. Bigus has authored several articles and technical papers and currently holds seven patents on neural network technology. He holds a B.S. in Computer Science from Villanova University, and a M.S. and Ph.D. in Computer Science from Lehigh University.