

DATA CLASSIFICATION USING MULTI-LAYERED FEED-FORWARD NEURAL NETWORK

A Dissertation Submitted in partial fulfillment of the requirements
for the award of the degree of

**MASTER OF ENGINEERING
(Computer Technology & Applications)**

By
PIYUSH KUMAR SRIVASTAVA

College Roll No. 17/CTA/04

University Roll No. 8510

Under the guidance of

Dr. S.K.Saxena



Department Of Computer Engineering

Delhi College of Engineering

Bawana Road, Delhi-110042

(University of Delhi)

June-2006

CERTIFICATE

This is to certify that dissertation entitled “**Data Classification Using Multi-layered Feed-forward Neural Network**” which is submitted by **Piyush Kumar Srivastava** in partial fulfillment of the requirement for the award of degree **M.E. in Computer Technology & Applications** to Delhi College of Engineering, Delhi is a record of the candidate own work carried out by him under my supervision.

Dr. S.K.Saxena

Department of Computer Engineering

Delhi College of Engineering

Bawana Road, Delhi

ACKNOWLEDGEMENT

It is a great pleasure to have the opportunity to extend my heartiest felt gratitude to everybody who helped me throughout the course of this project.

I would like to express my heartiest felt regards to **Dr. S.K.Saxena**, Department of Computer Engineering for the constant motivation and support during the duration of this project. It is my privilege and honor to have worked under the supervision. His invaluable guidance and helpful discussions in every stage of this project really helped me in materializing this project. It is indeed difficult to put his contribution in few words.

I would also like to take this opportunity to present my sincere regards to my teachers viz. Professor D. Roy Choudhary, Dr Goldie Gabrani, Mr. Rajeev Kumar and Mrs. Rajni Jindal for their support and encouragement.

I am thankful to my friends and classmates for their unconditional support and motivation during this project.

Piyush Kumar Srivastava

M.E. (Computer Technology & Applications)

College Roll No. 17/CTA/04

Delhi University Roll No. 8510

ABSTRACT

Artificial neural networks can be most adequately characterized as 'computational models' with particular properties such as the ability to adapt or learn, to generalize, or to cluster or organize data, and which operation is based on parallel processing.

Numerous advances have been made in developing intelligent systems, some inspired by biological neural networks. Researchers from many scientific disciplines are designing artificial neural networks (ANNs) to solve a variety of problems in pattern recognition, prediction, optimization, associative memory, and control.

Conventional approaches have been proposed for solving these problems. Although successful applications can be found in certain well-constrained environments, none is flexible enough to perform well outside its domain. ANNs provide exciting alternatives, and many applications could benefit from using them.

Classification is one of the data mining problems receiving great attention recently in the database community. This project will implement an approach to discover symbolic classification rules using neural networks. Neural networks have not been thought suited for data mining because how the classifications were made is not explicitly stated as symbolic rules that are suitable for verification or interpretation by humans. With the proposed approach, concise symbolic rules with high accuracy can be extracted from a neural network.

The network is first trained to achieve the required accuracy rate. Redundant connections of the network are then removed by a network pruning algorithm. The activation values of the hidden units in the network are analyzed, and classification rules are generated using the result of this analysis. The effectiveness of the proposed approach is clearly demonstrated by the experimental results on a set of standard data mining test problems.

TABLE OF CONTENTS

1. INTRODUCTION	1
2. DATA CLASSIFICATION	3
2.1. Data, Information, and Knowledge	3
2.2. What can data mining do?	4
2.3. How does data mining work?	5
2.4. What technological infrastructure is required?	7
2.5. Data Preparation (in Data Mining)	10
3. NEURAL NETWORKS	11
3.1. Introduction	11
3.2. Historical background	12
3.3. Neural networks versus conventional computers	12
3.4. Human and Artificial Neurons - investigating the similarities	13
3.4.1. How the Human Brain Learns?	13
3.4.2. Human Neurons to Artificial Neurons	14
3.4.3. Firing rules	15
3.4.4. Pattern Recognition - an example	17
3.4.5. A more complicated neuron	19
4. DATA PREPARATION	21
4.1. Data Cleansing	21
4.2. Data Selection	21
4.3. Data Preprocessing	22
4.4. Computed attributes	22
4.5. Scaling	23
5. NEURAL NETWORK TOPOLOGIES	24
5.1. Feed-Forward Networks	24
5.2. Limited Recurrent Networks	25
5.3. Fully Recurrent Networks	26
6. NEURAL NETWORK MODELS	29
6.1. Back Propagation Networks	29
6.2. Kohonen Feature Maps	31
6.3. Recurrent Back Propagation	34
6.4. Radial Basis Function	34
6.5. Adaptive Resonance Theory	35
6.6. Probabilistic Neural Networks	36
6.7. Key Issues in Selecting Models and Architecture	37

7. TRAINING AND TESTING NEURAL NETWORK	39
7.1. Back-propagation Algorithm	39
7.2. Defining Success: When Is the Neural Network Trained?	42
7.3. Classification	43
8. ANALYZING NEURAL NETWORKS	45
8.1. Discovering What the Network learned	45
9. IMPLEMENTATION OF THE PROJECT	47
9.1. Data Format	47
9.2. Attribute-Relation File Format	47
9.3. Installation	52
9.4. Configuration	52
9.5. Output	54
9.6. Architecture	58
9.7. Evaluation	59
10. CLASS DESCRIPTION	62
11. FUTURE WORK	98
12. CONCLUSION	99
13. REFERENCES	100

1. Introduction

Data Classification is one of the applications of Data Mining, “**Data Mining** is the efficient discovery of valuable information from large collection of data.” In Data classification, stored data is used to locate data in predetermined groups. There are various conventional methods exist to implement data classification, but no one is as fast as human brain. I am implementing **Data classification** using **Neural Network** which is different from conventional computer approach.

Neural networks take a different approach to problem solving than that of conventional computers. Neural networks process information in a similar way the human brain does. Neural networks learn by example. They cannot be programmed to perform a specific task.

The main aim of my project is to implement a basic neural network which simulates the behavior of neural network that is how the neural networks learn, how they process their nodes and how they classify the data given as input.

Approach: I have used **java 1.5** as programming language to implement this project. The project has been developed on java Text Editor. The **input** format is a subset of the arff(Attribute Relationship file format) format used by Weka2, a popular open source data mining tool. Specifically, the only supported attribute types are numerical (numeric, integer, real), and nominal.

After learning and validation has been performed, the model, i.e. the trained neural network, and some validation metrics are **output** as plain text. The model is output layer by layer, from the input layer over the hidden layers to the output layer, and each layer node by node in order For each node its position in the network and its links to the nodes of the previous layer along with the learned weights are shown.

The system can be divided into the components configuration, command line evaluation, parsing, data representation, data normalization, the neural network,

validation, and output.

The purpose of this project is to implement the basic neural network and to classify the data using implemented neural network.

Why Use Neural Network:

Neural network is advantageous than conventional computer because it can be as fast as human brain. Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include: Adaptive learning, Self-Organization, Real Time Operation, Fault Tolerance via Redundant Information Coding, etc.

2. Data Classification

Data Classification is one of the applications of Data Mining, “**Data Mining** is the efficient discovery of valuable information from large collection of data.”

Generally, data mining is the process of analyzing data from different perspectives and summarizing it into useful information - information that can be used to increase revenue, cuts costs, or both. Data mining software is one of a number of analytical tools for analyzing data. It allows users to analyze data from many different dimensions or angles, categorize it, and summarize the relationships identified. Technically, data mining is the process of finding correlations or patterns among dozens of fields in large relational databases.

Although data mining is a relatively new term, the technology is not. Companies have used powerful computers to sift through volumes of supermarket scanner data and analyze market research reports for years. However, continuous innovations in computer processing power, disk storage, and statistical software are dramatically increasing the accuracy of analysis while driving down the cost.

2.1 Data, Information, and Knowledge

Data

Data are any facts, numbers, or text that can be processed by a computer. Today, organizations are accumulating vast and growing amounts of data in different formats and different databases. This includes:

- operational or transactional data such as, sales, cost, inventory, payroll, and accounting
- nonoperational data, such as industry sales, forecast data, and macro economic data
- meta data - data about the data itself, such as logical database design or data dictionary definitions

Information

The patterns, associations, or relationships among all this *data* can provide *information*. For example, analysis of retail point of sale transaction data can yield information on which products are selling and when.

Knowledge

Information can be converted into *knowledge* about historical patterns and future trends. For example, summary information on retail supermarket sales can be analyzed in light of promotional efforts to provide knowledge of consumer buying behavior. Thus, a manufacturer or retailer could determine which items are most susceptible to promotional efforts.

Data Warehouses

Dramatic advances in data capture, processing power, data transmission, and storage capabilities are enabling organizations to integrate their various databases into *data warehouses*. Data warehousing is defined as a process of centralized data management and retrieval. Data warehousing, like data mining, is a relatively new term although the concept itself has been around for years. Data warehousing represents an ideal vision of maintaining a central repository of all organizational data. Centralization of data is needed to maximize user access and analysis. Dramatic technological advances are making this vision a reality for many companies. And, equally dramatic advances in data analysis software are allowing users to access this data freely. The data analysis software is what supports data mining.

2.2 What can data mining do?

Data mining is primarily used today by companies with a strong consumer focus - retail, financial, communication, and marketing organizations. It enables these

companies to determine relationships among "internal" factors such as price, product positioning, or staff skills, and "external" factors such as economic indicators, competition, and customer demographics. And, it enables them to determine the impact on sales, customer satisfaction, and corporate profits. Finally, it enables them to "drill down" into summary information to view detail transactional data.

2.3 How does data mining work?

While large-scale information technology has been evolving separate transaction and analytical systems, data mining provides the link between the two. Data mining software analyzes relationships and patterns in stored transaction data based on open-ended user queries. Several types of analytical software are available: statistical, machine learning, and neural networks. Generally, any of four types of relationships are sought:

- **Classes:** Stored data is used to locate data in predetermined groups. For example, a restaurant chain could mine customer purchase data to determine when customers visit and what they typically order. This information could be used to increase traffic by having daily specials.
- **Clusters:** Data items are grouped according to logical relationships or consumer preferences. For example, data can be mined to identify market segments or consumer affinities.
- **Associations:** Data can be mined to identify associations. The beer-diaper example is an example of associative mining.
- **Sequential patterns:** Data is mined to anticipate behavior patterns and trends. For example, an outdoor equipment retailer could predict the likelihood of a backpack being purchased based on a consumer's purchase of sleeping bags and hiking shoes.

Data mining consists of five major elements:

- Extract, transform, and load transaction data onto the data warehouse system.
- Store and manage the data in a multidimensional database system.
- Provide data access to business analysts and information technology professionals.
- Analyze the data by application software.
- Present the data in a useful format, such as a graph or table.

Different levels of analysis are available:

- **Artificial neural networks:** Non-linear predictive models that learn through training and resemble biological neural networks in structure.
- **Genetic algorithms:** Optimization techniques that use processes such as genetic combination, mutation, and natural selection in a design based on the concepts of natural evolution.
- **Decision trees:** Tree-shaped structures that represent sets of decisions. These decisions generate rules for the classification of a dataset. Specific decision tree methods include Classification and Regression Trees (CART) and Chi Square Automatic Interaction Detection (CHAID) . CART and CHAID are decision tree techniques used for classification of a dataset. They provide a set of rules that you can apply to a new (unclassified) dataset to predict which records will have a given outcome. CART segments a dataset by creating 2-way splits while CHAID segments using chi square tests to create multi-way splits. CART typically requires less data preparation than CHAID.

- **Nearest neighbor method:** A technique that classifies each record in a dataset based on a combination of the classes of the k record(s) most similar to it in a historical dataset (where $k \geq 1$). Sometimes called the k -nearest neighbor technique.
- **Rule induction:** The extraction of useful if-then rules from data based on statistical significance.
- **Data visualization:** The visual interpretation of complex relationships in multidimensional data. Graphics tools are used to illustrate data relationships.

2.4 What technological infrastructure is required?

Today, data mining applications are available on all size systems for mainframe, client/server, and PC platforms. System prices range from several thousand dollars for the smallest applications up to \$1 million a terabyte for the largest. Enterprise-wide applications generally range in size from 10 gigabytes to over 11 terabytes. NCR has the capacity to deliver applications exceeding 100 terabytes. There are two critical technological drivers:

- **Size of the database:** the more data being processed and maintained, the more powerful the system required.
- **Query complexity:** the more complex the queries and the greater the number of queries being processed, the more powerful the system required.

Relational database storage and management technology is adequate for many data mining applications less than 50 gigabytes. However, this infrastructure needs to be significantly enhanced to support larger applications. Some vendors have added extensive indexing capabilities to improve query performance. Others use new hardware architectures such as Massively Parallel Processors (MPP) to achieve

order-of-magnitude improvements in query time. For example, MPP systems from NCR link hundreds of high-speed Pentium processors to achieve performance levels exceeding those of the largest supercomputers.

Data Mining is an analytic process designed to explore data (usually large amounts of data - typically business or market related) in search of consistent patterns and/or systematic relationships between variables, and then to validate the findings by applying the detected patterns to new subsets of data. The ultimate goal of data mining is prediction. The process of data mining consists of three stages:

Stage 1: Exploration. This stage usually starts with data preparation which may involve cleaning data, data transformations, selecting subsets of records and - in case of data sets with large numbers of variables ("fields") - performing some preliminary [feature selection](#) operations to bring the number of variables to a manageable range (depending on the statistical methods which are being considered). Then, depending on the nature of the analytic problem, this first stage of the process of data mining may involve anywhere between a simple choice of straightforward predictors for a regression model, to elaborate exploratory analyses using a wide variety of graphical and statistical methods (see [Exploratory Data Analysis \(EDA\)](#)) in order to identify the most relevant variables and determine the complexity and/or the general nature of models that can be taken into account in the next stage.

Stage 2: Model building and validation. This stage involves considering various models and choosing the best one based on their predictive performance (i.e., explaining the variability in question and producing stable results across samples). This may sound like a simple operation, but in fact, it sometimes involves a very elaborate process. There are a variety of techniques developed to achieve that goal - many of which are based on so-called "competitive evaluation of models," that is, applying different models to the same data set and then comparing their performance to choose the best. These techniques - which are often considered the

core of [predictive data mining](#) - include: [Bagging](#) (Voting, Averaging), [Boosting](#), [Stacking \(Stacked Generalizations\)](#), and [Meta-Learning](#).

Stage 3: Deployment. That final stage involves using the model selected as best in the previous stage and applying it to new data in order to generate predictions or estimates of the expected outcome.

The concept of *Data Mining* is becoming increasingly popular as a business information management tool where it is expected to reveal knowledge structures that can guide decisions in conditions of limited certainty. Recently, there has been increased interest in developing new analytic techniques specifically designed to address the issues relevant to business *Data Mining* (e.g., [Classification Trees](#)), but *Data Mining* is still based on the conceptual principles of statistics including the traditional [Exploratory Data Analysis \(EDA\)](#) and modeling and it shares with them both some components of its general approaches and specific techniques.

However, an important general difference in the focus and purpose between *Data Mining* and the traditional [Exploratory Data Analysis \(EDA\)](#) is that *Data Mining* is more oriented towards applications than the basic nature of the underlying phenomena. In other words, *Data Mining* is relatively less concerned with identifying the specific relations between the involved variables. For example, uncovering the nature of the underlying functions or the specific types of interactive, multivariate dependencies between variables are not the main goal of *Data Mining*. Instead, the focus is on producing a solution that can generate useful predictions. Therefore, *Data Mining* accepts among others a "black box" approach to data exploration or knowledge discovery and uses not only the traditional [Exploratory Data Analysis \(EDA\)](#) techniques, but also such techniques as [Neural Networks](#) which can generate valid predictions but are not capable of identifying the specific nature of the interrelations between the variables on which the predictions are based.

Data Mining is often considered to be "a blend of statistics, AI [artificial intelligence], and data base research" (Pregibon, 1997, p. 8), which until very recently was not commonly recognized as a field of interest for statisticians, and was even considered by some "a dirty word in Statistics" (Pregibon, 1997, p. 8). Due to its applied importance, however, the field emerges as a rapidly growing and major area (also in statistics) where important theoretical advances are being made (see, for example, the recent annual *International Conferences on Knowledge Discovery and Data Mining*, co-hosted by the *American Statistical Association*).

2.5 Data Preparation (in Data Mining)

Data preparation and cleaning is an often neglected but extremely important step in the data mining process. The old saying "garbage-in-garbage-out" is particularly applicable to the typical data mining projects where large data sets collected via some automatic methods (e.g., via the Web) serve as the input into the analyses. Often, the method by which the data were gathered was not tightly controlled, and so the data may contain out-of-range values (e.g., Income: -100), impossible data combinations (e.g., Gender: Male, Pregnant: Yes), and the like. Analyzing data that has not been carefully screened for such problems can produce highly misleading results, in particular in predictive data mining.

3. Neural Networks

3.1 Introduction

A Neural Network or more appropriately Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. Artificial Neural Network is basically a mathematical model of what goes in our mind (or brain). The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. The brain of all the advanced living creatures consists of neurons, a basic cell, which when interconnected produces what we call Neural Network. The sole purpose of a Neuron is to receive electrical signals, accumulate them and see further if they are strong enough to pass forward.

So simple in its basic functionality but the interconnections of these produces beings (me, u and others) capable of writing about them. Phew! The real thing lies not in neurons but the complex pattern in which they are interconnected. NNs are just like a game of chess, easy to learn but hard to master. As the moves of chess are simple, yet the succession of moves is what makes the game complex and fun to play. Imagine a chess game in which you are allowed only one single move. Would that game be fun to play? In the same way, a single neuron is useless. Well, practically useless. It is the complex connection between them and values attached with them (explained later) which makes brains capable of thinking and having a sense of consciousness (much debated). ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well

3.2 Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras. Many important advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few researchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pitts. But the technology available at that time did not allow them to do too much.

3.3 Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing

elements (neurons) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to be solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks more suited to an algorithmic approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

3.4 Human and Artificial Neurons - investigating the similarities

3.4.1 How the Human Brain Learns?

Much is still unknown about how the brain trains itself to process information, so theories abound. In the human brain, a typical neuron collects signals from others through a host of fine structures called *dendrites*. The neuron sends out spikes of electrical activity through a long, thin strand known as an *axon*, which splits into thousands of branches. At the end of each branch, a structure called a *synapse* converts the activity from the axon into electrical effects that inhibit or excite activity from the axon into electrical effects that inhibit or excite activity in the connected neurons. When a neuron receives excitatory input that is sufficiently

large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.

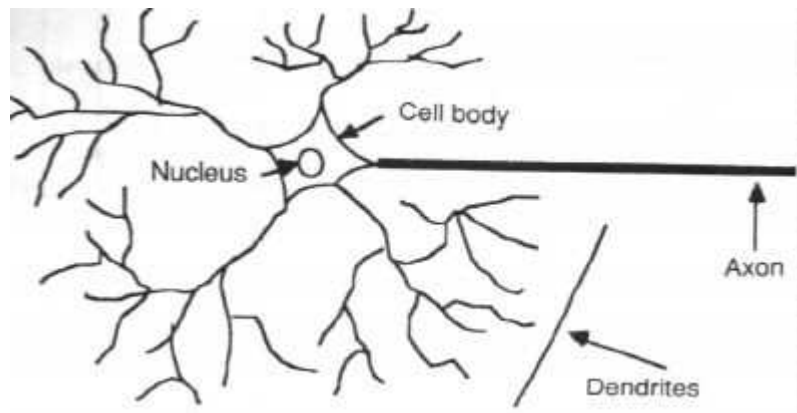


Fig 3.1 Components of a neuron

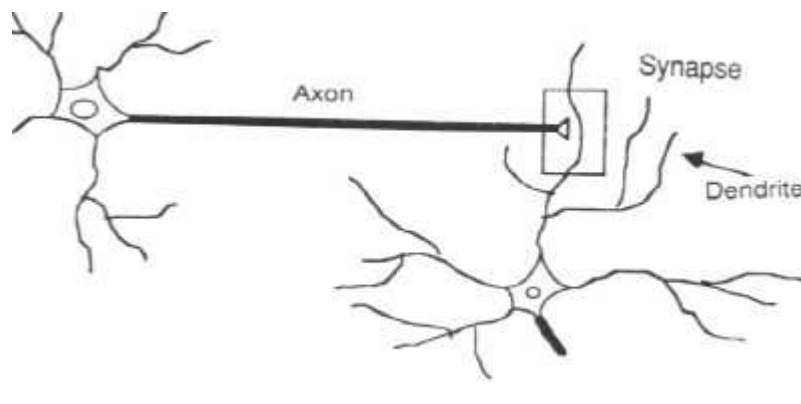


Fig 3.2 the synapse

3.4.2 Human Neurons to Artificial Neurons

We conduct these neural networks by first trying to deduce the essential features of neurons and their interconnections. We then typically program a computer to

simulate these features. However because our knowledge of neurons is incomplete and our computing power is limited, our models are necessarily gross idealizations of real networks of neurons.

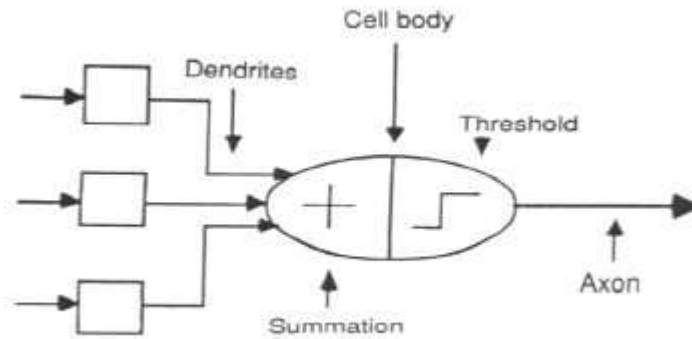


Fig 3.3 the neuron model

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.

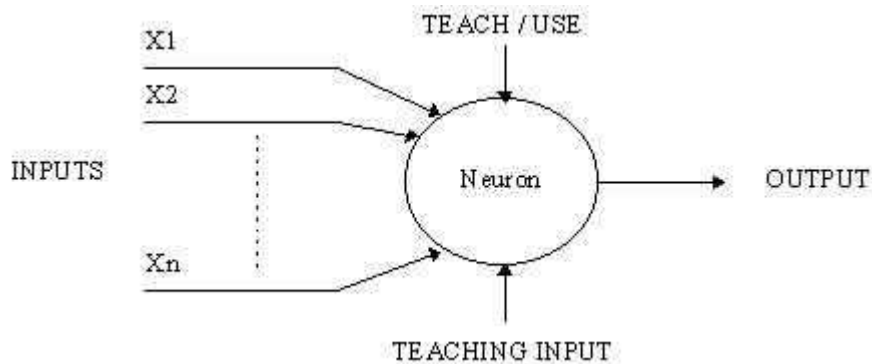


Fig 3.4 A simple neuron

3.4.3 Firing rules

The firing rule is an important concept in neural networks and accounts for their high flexibility. A firing rule determines how one calculates whether a neuron should fire for any input pattern. It relates to all the input patterns, not only the ones on which the node was trained.

A simple firing rule can be implemented by using Hamming distance technique. The rule goes as follows:

Take a collection of training patterns for a node, some of which cause it to fire (the 1-taught set of patterns) and others which prevent it from doing so (the 0-taught set). Then the patterns not in the collection cause the node to fire if, on comparison, they have more input elements in common with the 'nearest' pattern in the 1-taught set than with the 'nearest' pattern in the 0-taught set. If there is a tie, then the pattern remains in the undefined state.

For example, a 3-input neuron is taught to output 1 when the input (X1,X2 and X3) is 111 or 101 and to output 0 when the input is 000 or 001. Then, before applying the firing rule, the truth table is;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0/1	0/1	0/1	1	0/1	1

As an example of the way the firing rule is applied, take the pattern 010. It differs from 000 in 1 element, from 001 in 2 elements, from 101 in 3 elements and from 111 in 2 elements. Therefore, the 'nearest' pattern is 000 which belongs in the 0-taught set. Thus the firing rule requires that the neuron should not fire when the input is 001. On the other hand, 011 is equally distant from two taught patterns that have different outputs and thus the output stays undefined (0/1).

By applying the firing in every column the following truth table is obtained;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0	0/1	0/1	1	1	1

The difference between the two truth tables is called the *generalization of the neuron*. Therefore the firing rule gives the neuron a sense of similarity and enables it to respond 'sensibly' to patterns not seen during training.

3.4.4 Pattern Recognition - an example

An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward (figure 1) neural network that has been trained accordingly. During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the input pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is given as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

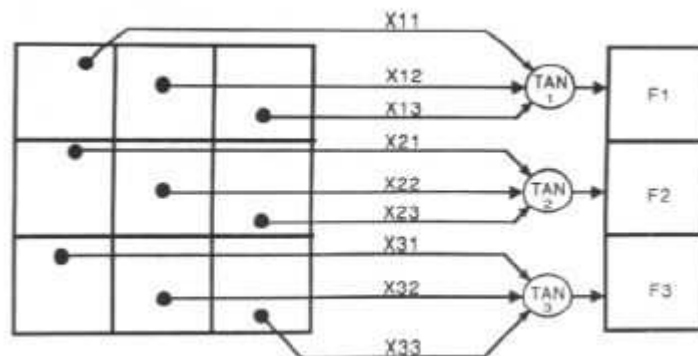


Fig 3.5 a feed-forward neural network

The network of figure 1 is trained to recognize the patterns T and H. The associated patterns are all black and all white respectively as shown below.



If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurons after generalization are;

X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Top neuron

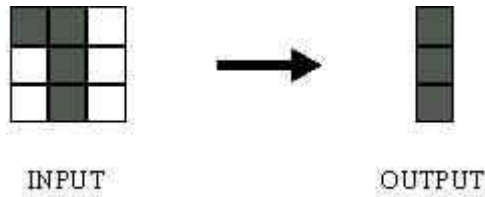
X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

Middle neuron

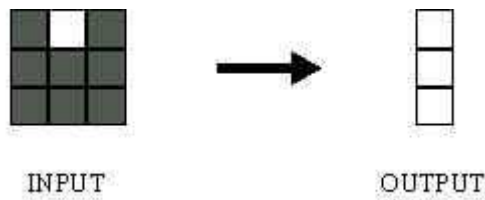
X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0	1	1	0	0	1	0

Bottom neuron

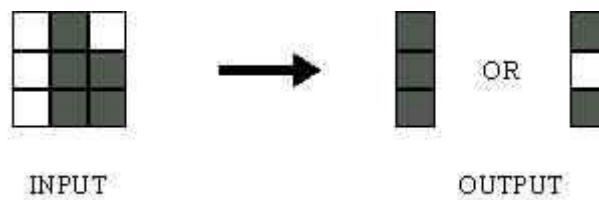
From the tables it can be seen the following associations can be extracted:



In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern.



Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.



Here, the top row is 2 errors away from the T and 3 from an H. So the top output is black. The middle row is 1 error away from both T and H so the output is random. The bottom row is 1 error away from T and 2 away from H. Therefore the output is black. The total output of the network is still in favor of the T shape.

3.4.5 A more complicated neuron

The previous neuron doesn't do anything that conventional computers don't do already. A more sophisticated neuron (figure 2) is the McCulloch and Pitts model (MCP). The difference from the previous model is that the inputs are 'weighted'; the effect that each input has at decision making is dependent on the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs are then added together and if they exceed a pre-set threshold value, the neuron fires. In any other case the neuron does not fire.

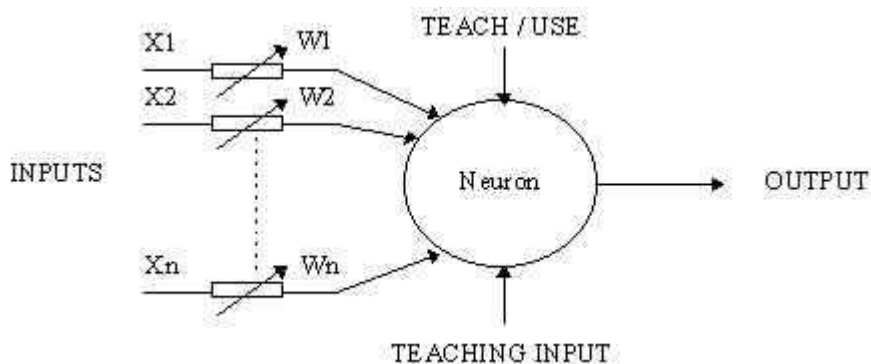


Fig 3.6 An MCP neuron

In mathematical terms, the neuron fires if and only if;

$$X_1W_1 + X_2W_2 + X_3W_3 + \dots > T$$

The addition of input weights and of the threshold makes this neuron a very flexible and powerful one. The MCP neuron has the ability to adapt to a particular situation by changing its weights and/or threshold. Various algorithms exist that cause the neuron to 'adapt'; the most used ones are the Delta rule and the back error propagation. The former is used in feed-forward networks and the latter in feedback networks.

4. Data Preparation

Before giving the data to neural network for training we have to prepare the data, because the data can be in any format we have to make it acceptable.

4.1 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, ill large data sets. Another approach is to use statistical information to set missing or incorrect field values to neutral, valid values

4.2 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 column 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.

4.3 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. 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.

4.4 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.

4.5 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.

5. Neural Network Topologies

The arrangement of neural processing units and their interconnections can have a profound impact on the processing capabilities of the neural networks. 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 feed forward, limited recurrent, and fully recurrent networks are described in detail in the next sections.

5.1 Feed-Forward Networks

Feed-forward 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 5.1, we see a typical feed-forward 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.

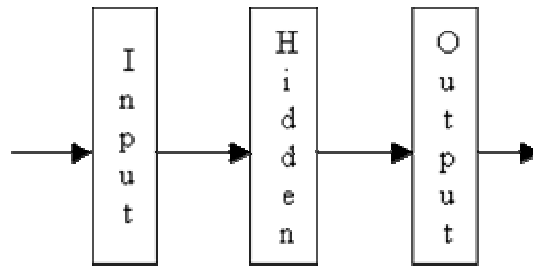


Fig5.1: Feed-forward neural networks.

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 multi-layer 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. 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 5.1, 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 feed-forward 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 **feed-forward** is a definition of connection topology and data flow. It does not imply any specific type of activation function or training paradigm.

5.2 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 5.2).

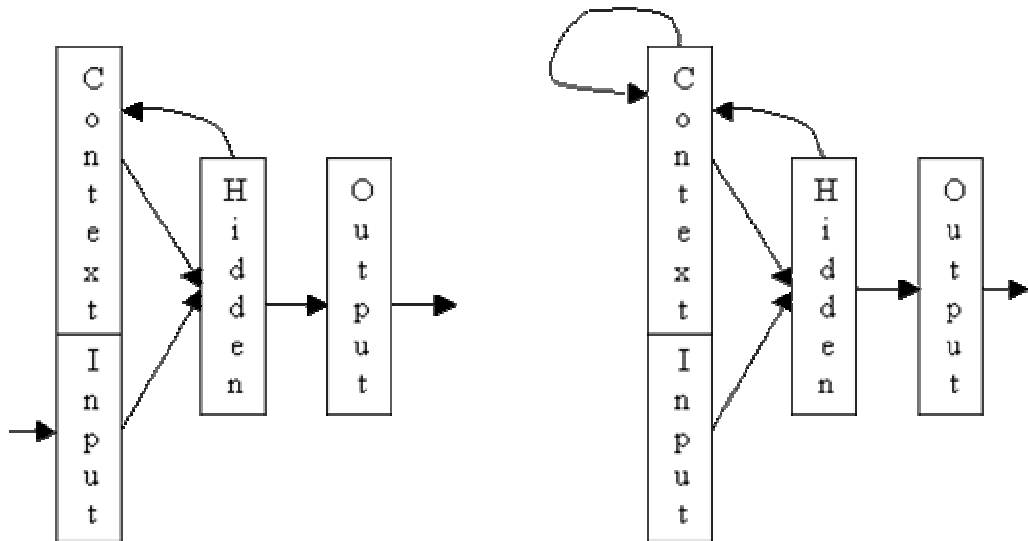


Figure 5.2: Partial recurrent neural networks

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 feed-forward 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.

5.3 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 6.3 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.

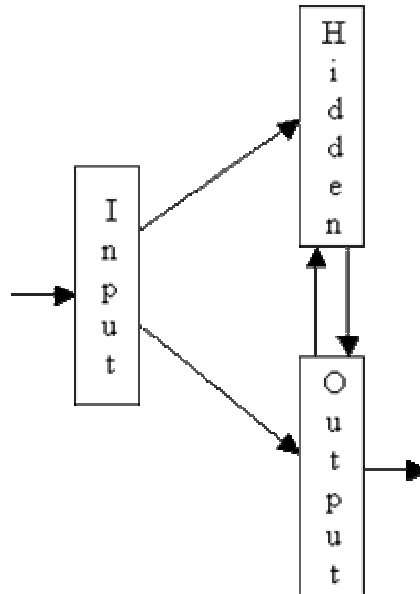


Figure 5.3: Fully recurrent neural networks

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 feed-forward 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 in-determinate 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.

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.

6. Neural Network Models

The combination of topology, learning paradigm (supervised or non-supervised learning), and learning algorithm define a neural network model. There is 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.

6.1 Back Propagation Networks

A back propagation neural network uses a feed-forward 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 mid1980s.

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. 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.

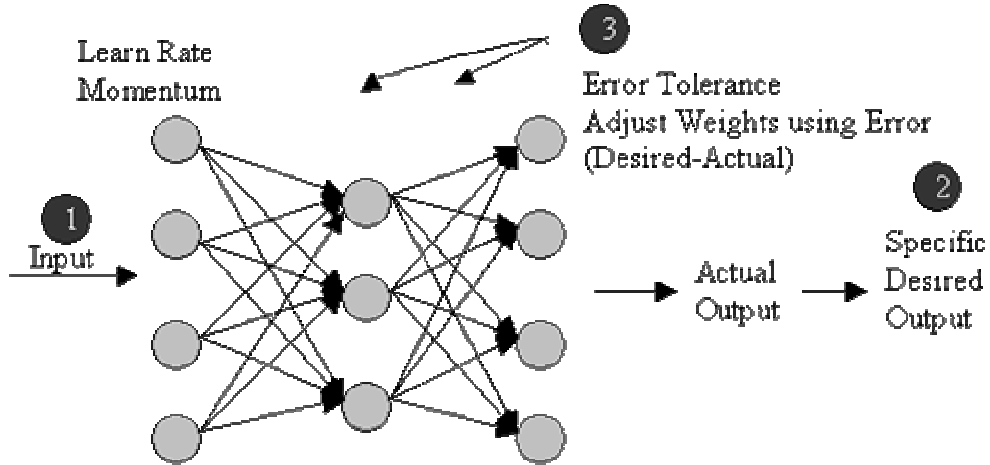


Fig 6.1: Back propagation networks

The basic back propagation algorithm consists of three steps (see Figure 6.1). 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 corresponding 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 discriminate 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.

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.

6.2 Kohonen Feature Maps

Kohonen feature maps are feed-forward 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 Quantisation (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 randomized 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.

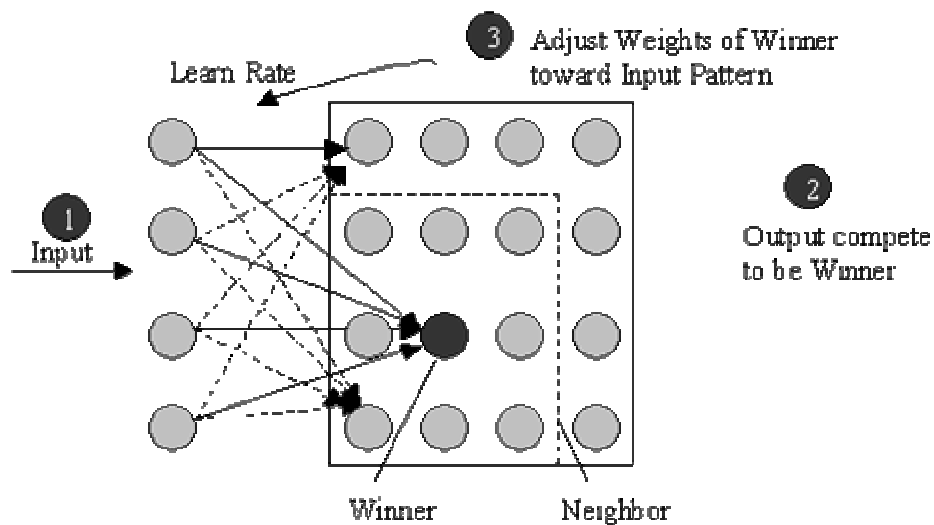


Figure 6.2: Kohonen self-organizing feature maps

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. When combined with appropriate visualization tools and/or analysis of both the population and segment statistics, the makeup of the segments identified by the feature map can be analyzed and turned into valuable business intelligence.

6.3 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.

6.4 Radial Basis Function

Radial basis function (RBF) networks are feed-forward 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*. 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 non-stationary 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 (Wasserrnan 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.

6.5 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. 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 units 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.

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.

6.6 Probabilistic Neural Networks

Probabilistic neural networks (PNN) feature feed-forward 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.

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

Table 6.1: Neural Network Models and Their Functions

6.7 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 6.1 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.

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 or Kohonen maps.

7. Training and Testing Neural Network

7.1 Back-propagation Algorithm

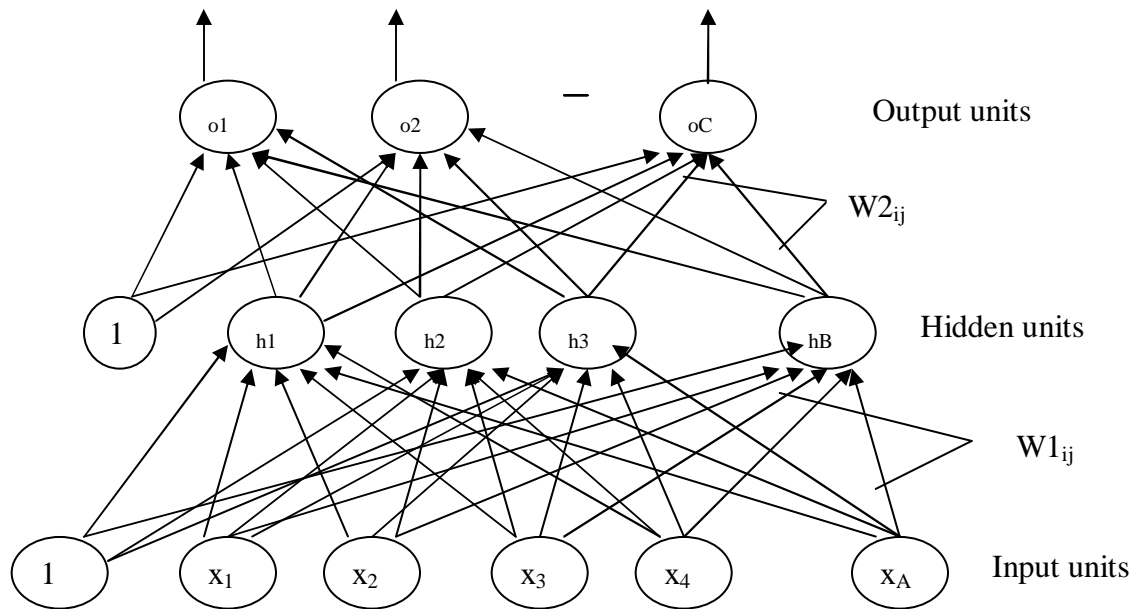


Fig 7.1 A multilayer Network

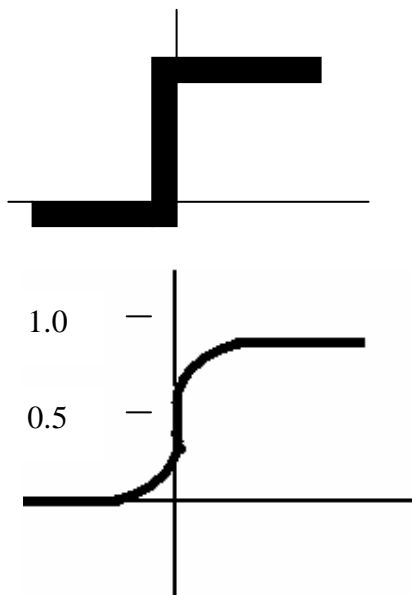


Fig 7.2 The stepwise Activation function of the Perceptron (above), and the Sigmoid Activation Function of the Backpropagation Unit (below)

Given: A set of input-output vector pairs.

Compute: A set of weights for a multi layer network that maps inputs onto corresponding outputs.

1. Let A be the number of units in the input layer, as determined by the length of the training input vectors. Let C be the number of units in the output layer. Now choose B , the number of units in the hidden layer. As figure 7.1, the input and hidden layers each have an extra unit used for shareholding; therefore, the units in these layers will sometimes be indexed by the ranges $(0, \dots, A)$ and $(0, \dots, B)$. We denote the activation levels of the units in the input layer by x_j , in the hidden layer by h_j , and in the output layer by o_j . Weights connecting the input layer to the hidden layer are denoted by w_{1ij} , where the subscript i indexes the input units and j indexes the hidden units. Likewise, weights connecting the input layer to the output layer are denoted by w_{2ij} , with I indexing to hidden units and j indexing output units.

2. Initialize the weights in the network. Each weight should be set randomly to a number between -0.1 and 0.1.

$$W_{1ij} = \text{random}(-0.1, 0.1) \quad \text{for all } i = 0, \dots, A, j = 1, \dots, B$$

$$W_{2ij} = \text{random}(-0.1, 0.1) \quad \text{for all } i = 0, \dots, B, j = 1, \dots, C$$

3 Initialize the activations of the thresholding units. The values of these thresholding units should never change.

$$x_0 = 1.0$$

$$h_0 = 1.0$$

4. Choose an input-output pair. Suppose the input vector is x_i and the target output vector is y_i . Assign activation levels to the input units.

5. Propagate the activation from the units in the input layer to the units in the hidden layer using the activation function of figure 7.2:

$$h_j = \frac{1}{1 + e^{-\sum_{i=0}^A w_{1ij} x_i}} \quad \text{for all } j = 1, \dots, B$$

Note that I ranges from 0 to A . w_{10j} is the thresholding weight for hidden unit j (its propensity to fire irrespective of its inputs). x_0 is always 1.0.

6. Propagate the activations from the units in the hidden layer to the units in the output layer.

$$o_j = \frac{1}{1 + e^{-\sum_{i=0}^B w_{2ij} h_i}} \quad \text{for all } j = 1, \dots, C$$

Again, the thresholding weight w_{20j} for output unit j plays a role in the weighted summation. h_0 is always 1.0.

7. Compute the errors of the units in the hidden layer, denoted $\delta 2_j$. Errors are based on the network's actual output (o_j) and the target output (y_j).

$$\delta 2_j = o_j(1 - o_j)(y_j - o_j) \quad \text{for all } j = 1, \dots, C$$

8. Compute the errors of the units in the hidden layer, denoted $\delta 1_j$.

$$\delta 1_j = h_j(1 - h_j) \sum_{i=0}^C \delta 2_i \cdot w_{2ji} \quad \text{for all } j = 1, \dots, B$$

9. Adjust the weights between the hidden layer and output layer. The learning rate is denoted η ; its function is the same as in perceptron learning. A reasonable value of η is 0.35.

$$\Delta w_{2ij} = \eta \cdot \delta 2_j \cdot h_i \quad \text{for all } i = 0, \dots, B, j = 1, \dots, C$$

10. Adjust the weights between the input layer and the hidden layer.

$$\Delta w_{1ij} = \eta \cdot \delta 1_j \cdot x_i \quad \text{for all } i = 0, \dots, A, j = 1, \dots, B$$

11. Go to step 4 and repeat. When all the input-output pairs have been presented to the network, one epoch has been completed. Repeat steps 4 to 10 for as many epoch is desired.

The algorithm generalizes straightforwardly to networks of more than three layers. For each extra layer, insert a forward propagation step between steps 6 and 7, an error computation step between step 8 and 9, and a weight adjustment step between step 10 and 11. Error computation for hidden units should use the equation in step 8, but with I ranging over the units in the next layer, not necessarily the output layer.

The speed of learning can be increased by modifying the weight modification steps 9 and 10 to include a momentum term α . The weight update formulas become;

$$\Delta w_{2_{ij}}(t+1) = \eta \cdot \delta_{2_j} \cdot h_i + \alpha \Delta w_{2_{ij}}(t)$$

$$\Delta w_{1_{ij}}(t+1) = \eta \cdot \delta_{1_j} \cdot h_i + \alpha \Delta w_{1_{ij}}(t)$$

Where h_i , x_i , δ_{1_j} and δ_{2_j} are measured at time $t+1$. $\Delta w_{1_{ij}}(t)$ is the change the weight experienced during the previous forward-backward pass. If α is set to 0.9 or so, learning speed is improved.

Recall that the activation function has a sigmoid shape. Since infinite weights would be required for the actual outputs of the network to reach 0.0 and 1.0, binary target outputs (the y_j 's of steps 4 and above) are usually given as 0.1 and 0.9 instead. The sigmoid is required by backpropagation because the derivation of the weight update rule requires that the activation function be continuous and differentiable.

7.2 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.

7.3 Classification

The measure of success in a classification problem is the accuracy of the classifier, usually termed 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 .

8. Analyzing Neural Network

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 section, I describe 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.

8.1 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. One approach is to treat the neural network as "black box", probe it with test input and record output. 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 function performed by neural network based on inspection 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 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 neural network has learned is qualitatively different, based on the function it was trained to do. For example if you are clustering customer for a market segmentation application, the output of the neural network is the identifier of the cluster that the customer fell into. At this point , statistical analysis of he 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 following into each output unit (cluster) and analyze them to see what the neural network learned were t6he “prototypical” customer for that segment.

9. Implementation of the Project

I have implemented this project using java 1.5 programming language; I have used Text Editor for writing source code in java. This project consist various classes written in java. In this section I will describe about input and output of the project.

9.1 Data Format

The input format is a subset of the arff format used by Weka2, a popular open source data mining tool.

Specifically, the only supported attribute types are numerical (numeric, integer, real), and nominal. The unsupported types are of no use in the setting of providing a simple neural network. Furthermore, missing and sparse data is not supported as well, as these would require some data preprocessing which is not the focus of this project. Weka comes with some sample data files, as an example here an excerpt from the iris sample:

```
@RELATION iris

@ATTRIBUTE sepallength    REAL REAL REAL REAL
@ATTRIBUTE sepalwidth     {Iris-setosa,Iris-versicolor,Iris-virginica}
@ATTRIBUTE petallength
@ATTRIBUTE petalwidth
@ATTRIBUTE class

@DATA 5.1,3.5,1.4,0.2,Iris-setosa
```

In addition to the definition of the sample name and schema, that is four numeric attributes and the nominal attribute class, the first data record is shown.

9.2 Attribute-relation file format

An ARFF (Attribute-Relation File Format) file is an ASCII text file that describes a list of instances sharing a set of attributes. ARFF files were developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software.

Overview

ARFF files have two distinct sections. The first section is the **Header** information, which is followed the **Data** information. The **Header** of the ARFF file contains the name of the relation, a list of the attributes (the columns in the data), and their types. An example header on the standard IRIS dataset looks like this:

```
% 1. Title: Iris Plants Database
@RELATION iris

@ATTRIBUTE sepallength NUMERIC
@ATTRIBUTE sepalwidth NUMERIC
@ATTRIBUTE petallength NUMERIC
@ATTRIBUTE petalwidth NUMERIC
@ATTRIBUTE class {Iris-setosa,Iris-versicolor,Iris-
virginica}
```

The **Data** of the ARFF file looks like the following:

```
@DATA
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

Lines that begin with a % are comments. The **@RELATION**, **@ATTRIBUTE** and **@DATA** declarations are case insensitive.

Examples

Several well-known machine learning datasets are distributed with Weka in the \$WEKAHOME/data directory as ARFF files.

The ARFF Header Section

The ARFF Header section of the file contains the relation declaration and attribute declarations.

The @relation Declaration

The relation name is defined as the first line in the ARFF file. The format is:

`@relation <relation-name>`
where `<relation-name>` is a string. The string must be quoted if the name includes spaces.

The @attribute Declarations

Attribute declarations take the form of an ordered sequence of **@attribute** statements. Each attribute in the data set has its own **@attribute** statement which uniquely defines the name of that attribute and its data type. The order the attributes are declared indicates the column position in the data section of the file. For example, if an attribute is the third one declared then Weka expects that all that attribute's values will be found in the third comma delimited column.

The format for the **@attribute** statement is:

```
@attribute <attribute-name> <datatype>
```

where the `<attribute-name>` must start with an alphabetic character. If spaces are to be included in the name then the entire name must be quoted.

The `<datatype>` can be any of the four types currently (version 3.2.1) supported by Weka:

- numeric
- `<nominal-specification>`
- string
- date [`<date-format>`]

where `<nominal-specification>` and `<date-format>` are defined below. The keywords **numeric**, **string** and **date** are case insensitive.

Numeric attributes

Numeric attributes can be real or integer numbers.

Nominal attributes

Nominal values are defined by providing an `<nominal-specification>` listing the possible values: {`<nominal-name1>`, `<nominal-name2>`, `<nominal-name3>`, ...}

For example, the class value of the Iris dataset can be defined as follows:

```
@ATTRIBUTE class          {Iris-setosa,Iris-versicolor,Iris-  
virginica}
```

Values that contain spaces must be quoted.

String attributes

String attributes allow us to create attributes containing arbitrary textual values. This is very useful in text-mining applications, as we can create datasets with string attributes, then write Weka Filters to manipulate strings (like `StringToWordVectorFilter`). String attributes are declared as follows:

```
@ATTRIBUTE LCC          string
```

Date attributes

Date attribute declarations take the form:

```
@attribute <name> date [<date-format>]
```

where `<name>` is the name for the attribute and `<date-format>` is an optional string specifying how date values should be parsed and printed (this is the same format used by `SimpleDateFormat`). The default format string accepts the ISO-8601 combined date and time format: "yyyy-MM-dd'T'HH:mm:ss". Dates must be specified in the data section as the corresponding string representations of the date/time (see example below).

ARFF Data Section

The ARFF Data section of the file contains the data declaration line and the actual instance lines.

The @data Declaration

The `@data` declaration is a single line denoting the start of the data segment in the file. The format is:

```
@data
```


The instance data

Each instance is represented on a single line, with carriage returns denoting the end of the instance. Attribute values for each instance are delimited by commas. They must appear in the order that they were declared in the header section (i.e. the data corresponding to the *n*th **@attribute** declaration is always the *n*th field of the attribute).

Missing values are represented by a single question mark, as in:

```
@data
4.4,?,1.5,?,Iris-setosa
```

Values of string and nominal attributes are case sensitive, and any that contain space must be quoted, as follows:

```
@relation LCCvsLCSH

@attribute LCC string
@attribute LCSH string

@data
AG5, 'Encyclopedias and dictionaries.;Twentieth century.'
AS262, 'Science -- Soviet Union -- History.'
AE5, 'Encyclopedias and dictionaries.'
AS281, 'Astronomy, Assyro-Babylonian.;Moon -- Phases.'
AS281, 'Astronomy, Assyro-Babylonian.;Moon -- Tables.'
```

Dates must be specified in the data section using the string representation specified in the attribute declaration. For example:

```
@RELATION Timestamps

@ATTRIBUTE timestamp DATE "yyyy-MM-dd HH:mm:ss"

@DATA
"2001-04-03 12:12:12"
"2001-05-03 12:59:55"
```

Sparse ARFF files

Sparse ARFF files are very similar to ARFF files, but data with value 0 are not be explicitly represented.

Sparse ARFF files have the same header (i.e **@relation** and **@attribute** tags) but the data section is different. Instead of representing each value in order, like this:

```
@data
0, X, 0, Y, "class A"
0, 0, W, 0, "class B"
```

the non-zero attributes are explicitly identified by attribute number and their value stated, like this:

```
@data
{1 X, 3 Y, 4 "class A"}
{2 W, 4 "class B"}
```

Each instance is surrounded by curly braces, and the format for each entry is: <index> <space> <value> where index is the attribute index (starting from 0).

9.3 Installation

The software is implemented as a number of plain Java files. The development version is Java 1.5, thus to compile or run it the same version or a more recent one is recommended, .

The compiled program, that is the NeuralNetwork.jar file, is run by the command

```
java -jar NeuralNetwork.jar [options] file_name
```

The options and file_name parameters are described below.

The source code version NeuralNetwork. needs to be compiled first. The compilation instructions below assume that the source has been unpacked in the current working directory.

9.4 Configuration

The only way to configure the program is by command line flags. The option `--help` lists all available flags, along with their default values and a concise explanation:

```
java -jar NeuralNetwork.jar --help
```

This is the only flag which takes no argument and does not require a file name to be specified. All other flags, as listed below, take exactly one argument. For an explanation of unknown terminology or concepts, refer to section 4.

. --target-attribute

Takes the name of an attribute as specified in the input file. This attribute becomes the target attribute for the classification. If not given, the attribute specified last in the input is used as the target attribute.

. --hidden-layers

Specifies the number of hidden layers and the number of nodes within each hidden layer (see Sec. 4). These are to be given as a comma separated list of non-negative integers, e.g. 4,6,2 for three hidden layers with four, six, and two nodes. If zero nodes are specified for a layer, this layer is omitted. Thus 0 amounts to no hidden layer at all.

. --learning rate

The learning rate of the back-propagation algorithm (see Sec. 4). This must be a real number greater than zero and less than one.

. --momentum

The momentum of the back-propagation algorithm (see Sec. 4). This must be a real number greater or equal than zero and less than one.

. --epochs

The number of times the sample data is fed into the neural network to train it . This must be an integer greater than zero. This is the only termination criterion for the learning process.

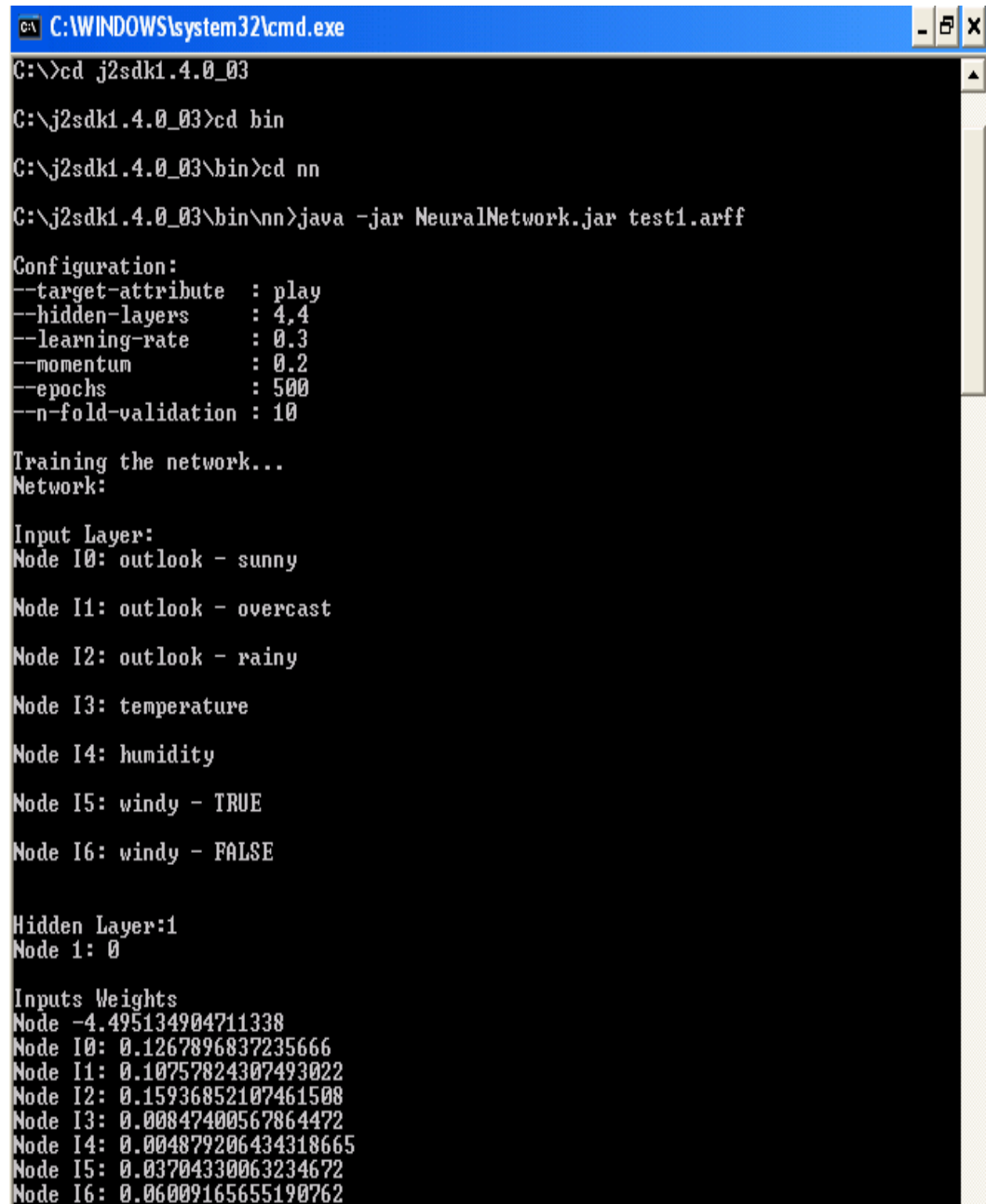
. --n-fold-validation

Cross-validation is performed over the given number of folds of the data sample. This must be an integer greater or equal to zero. For zero no cross validation is performed.

Note that validation is performed over the whole sample in any case.

9.5 Output

After learning and validation has been performed, the model, i.e. the trained neural network, and some validation metrics are output as plain text.



```
C:\WINDOWS\system32\cmd.exe
C:\>cd j2sdk1.4.0_03
C:\j2sdk1.4.0_03>cd bin
C:\j2sdk1.4.0_03\bin>cd nn
C:\j2sdk1.4.0_03\bin\nn>java -jar NeuralNetwork.jar test1.arff

Configuration:
--target-attribute : play
--hidden-layers   : 4,4
--learning-rate   : 0.3
--momentum        : 0.2
--epochs          : 500
--n-fold-validation : 10

Training the network...
Network:

Input Layer:
Node I0: outlook - sunny
Node I1: outlook - overcast
Node I2: outlook - rainy
Node I3: temperature
Node I4: humidity
Node I5: windy - TRUE
Node I6: windy - FALSE

Hidden Layer:1
Node 1: 0

Inputs Weights
Node -4.495134904711338
Node I0: 0.1267896837235666
Node I1: 0.10757824307493022
Node I2: 0.15936852107461508
Node I3: 0.00847400567864472
Node I4: 0.004879206434318665
Node I5: 0.03704330063234672
Node I6: 0.06009165655190762
```

```
C:\WINDOWS\system32\cmd.exe

Node 1: 2

Inputs Weights
Node -4.423375728572748
Node I0: 0.04971276465436773
Node I1: 0.10370221495732698
Node I2: 0.11336664212108145
Node I3: 0.004558508947604547
Node I4: 0.008067775307785793
Node I5: 0.08496193555553919
Node I6: 0.004671301879150341

Node 1: 3

Inputs Weights
Node -4.3868291520729565
Node I0: 0.11543827636678249
Node I1: 0.011143787583566288
Node I2: 0.1208509804551892
Node I3: 0.0074300354139111985
Node I4: 6.223436559499623E-4
Node I5: 0.028392105366078405
Node I6: 7.182614043929929E-4

Hidden Layer:2
Node 2: 0

Inputs Weights
Node -4.25069064669823
Node 1:0: 0.03778272718632037
Node 1:1: 0.053480441085850754
Node 1:2: 0.02893543145924958
Node 1:3: 0.22254434538508164

Node 2: 1

Inputs Weights
Node -4.253869358145067
Node 1:0: 0.21373624201640629
Node 1:1: 0.12923285387686778
Node 1:2: 0.06454946355945963
```

```

Node 2: 2

Inputs Weights
Node -4.254887400772407
Node 1:0: 0.16143322047240583
Node 1:1: 0.14195122053936454
Node 1:2: 0.12141425160104814
Node 1:3: 0.11698578706373745

Node 2: 3

Inputs Weights
Node -4.25173375292424
Node 1:0: 0.05314618039730025
Node 1:1: 0.05526816440824239
Node 1:2: 0.19043527625192672
Node 1:3: 0.17107168205457077

Output Layer:
Node 0: play - yes
Inputs Weights
Node 0.6264963113920418
Node 2:0: 0.23523368943309456
Node 2:1: 0.21811027261594523
Node 2:2: 0.6999660805551225
Node 2:3: 0.4418157902809606

Node 0: play - no
Inputs Weights
Node -0.6782255612494691
Node 2:0: 0.5678015452571529
Node 2:1: 0.6429294377720618
Node 2:2: 0.5834824579194126
Node 2:3: 0.2627649394940725

Performing validation of sample...

Validation of sample:
Mean Absolute Error : 0.4551975113712133
Root Mean Squared Error: 0.47936046010056554

Correctly Classified: 9 / 14

```

```

C:\WINDOWS\system32\cmd.exe
Precision          Recall            F1-measure
yes                1.0              0.0              0.0
no                 0.0              0.0              0.0

Confusion Matrix:
classified as:    yes          no
yes              9            0
no               5            0

Performing n-fold cross validation...

Mean Absolute Error   : 0.49284201862910143
Root Mean Squared Error: 0.5138878725883095

Correctly Classified: 9 / 14

Precision          Recall            F1-measure
yes                1.0              0.0              0.0
no                 0.0              0.0              0.0

Confusion Matrix:
classified as:    yes          no
yes              9            0
no               5            0

C:\j2sdk1.4.0_03\bin\nn>_

```

The model is output layer by layer, from the input layer over the hidden layers to the output layer, and each layer node by node in order. For each node its position in the network and its links to the nodes of the previous layer along with the learned weights are shown. The validation metrics, mean absolute error and root mean squared error, as well as a confusion matrix with the according f1-measures for a nominal target attribute, are given for the whole sample plus the cross-validation

average.

9.6 Architecture

The system can be divided into the components configuration, command line evaluation, parsing, data representation, data normalization, the neural network, validation, and output. In more detail, this is

Configuration

Classes: Config

Contains the configuration used by all parts of the system, for example the learning rate, the name of the input file, the target attribute, .. .

CommandLine Evaluation

Classes: EvalArgs, Option, OptionInt, OptionNat, OptionDouble, OptionLearningRate, OptionMomentum, OptionString, OptionNats.

The command line flags (see Sec. 2) are checked for correctness and evaluated. A Config object is created and initialized based on these settings.

The different types of flags are represented by different subclasses of Option. These serve to provide the different types of flags, e.g. a string with OptionString or a real numeric with OptionDouble, and to ensure further type restrictions, e.g. the valid range of a real value for OptionLearningRate, by further sub-classing in combination with JML constraints.

Parsing

Classes: ReadArff

Parses the input file given in the arff format, and creates a data schema and sample based on it.

Data Sample

Classes: Sample, Schema, Attribute, AttributeInt, AttributeReal, AttributeNominal

A data sample is represented by its schema and the actual data. The schema specifies the structure of the sample's data records in terms of attributes. That is, each element of a data record must be of a type compatible with the corresponding schema attribute. Attributes of the different numeric and nominal attributes are subclasses of the abstract class Attribute.

Data normalization

Before data can be entered into the neural network it is normalized based on the whole sample (see Sec. 4). NormalizerSample normalizes any data record, using the

appropriate attribute normalizer for each element of the record.

Neural Network

Classes: NeuralNetwork, Node, NodeInput, NodeHidden, NodeOutput, Connection, Weight

The neural network is represented as a network of connected weighted nodes. The subclasses of Node represent nodes of the input, a hidden, or the output layer. Connections are bidirectional, to feed data forward and errors feed backward during the learning process. Weights are attached to each connection and each node, except for the input nodes.

Validation

Classes: Validation, ConfusionMatrix

A network is validated by computing for a sample all outputs, and providing the correct and the computed outputs to a Validation object. It computes the validation metrics described in section 5. If the target attribute is nominal, the validation object does automatically create a confusion matrix.

Output

Classes: Print

Formats and print output of the system, like the help message or the Configuration.

9.7 Evaluation

To get a sense of how good the learned network models the data, some metrics to validate a model and some empirical test results are presented in the following.

Metrics

After a network has been trained on a sample, it is immediately evaluated on this same sample.

In detail, for a sample of size n the metrics are:

Mean Absolute Error

The sum of the absolute differences between the correct and computed output for each record, divided by the number of records: $(\sum_i |l_{output_i} - correct_i|)/n$

Root Mean Squared Error

The square root of the sum of the squared differences between the correct and computed output for each record, divided by the number of records: $\sqrt{(\sum_i (l_{output_i} - correct_i)^2)/n}$

For a nominal target value the following additional metrics are computed:

Correctly Classified

The number of records which have been correctly classified, in contrast to the number of incorrectly classified ones.

Confusion Matrix

A matrix where the rows correspond to the correct target value, the columns to the computed value, and each cell contains how often this case occurred. For example, the confusion matrix

	A	B	C
A	8	1	1
B	3	7	0
C	1	0	9

says that A has been classified 8 times correctly as A (true positive), 2 times incorrectly as B or C (false negative), and 4 times B or C have been incorrectly classified as A (false positive).

Precision

The precision of a value x is the number of times it has been classified correctly, divided by the number of times it has been classified correctly plus the number of times another value has been misclassified as x . Thus, the precision of A is $8/(8 + 4) = 0.66$.

Recall

Similarly, the recall of a value is the number of times it has been classified correctly, divided by the number of times it has been classified correctly plus the number of times it has been misclassified. Thus, the recall of A is $8/(8 + 2) = 0.8$.

F1-measure

Finally, the F1-measure of a value is computed as two times its precision times its recall divided by the sum of its precision and recall. Thus, the precision of A is $(2 * 0.66 * 0.8)/(0.66 + 0.8) = 0.73$.

Cross- Validation

As mentioned, the previous metrics are applied to the original sample, that is the sample is the training and the validation data set at the same time. This is problematic, as it is unclear how the performance of the network will be on unseen data, which comes from the same area as the original data, but was not available for training the net. The model could be perfect for the training data, but at the same time overfit it and not generalize well for any new data, which is clearly not desired.

To get an idea how well the model generalizes, cross-validation is applied. That is, the data sample S is partitioned into n sets S_n , and for the i .th setup $S \setminus S_i$ is the training set, while S_i is the validation set. Now, for this setup a new neural network is trained on the training set, where the network has the same initial structure and initial weights as the model trained over the whole sample S . Then, the new trained network is evaluated over the validation set, which is unseen data for this particular network, and thus a generalization test.

This is done for each of the n partitions. Finally, the n validation results are averaged, thus giving an indication on how well the original model might scale for new data.

10. Class Description

▪ Class Attribute

```
java.lang.Object
└─nn.Attribute
```

Direct Known Subclasses:

AttributeInt, AttributeNominal, AttributeReal

```
public abstract class Attribute
```

```
extends java.lang.Object
```

The specification of an attribute of a data schema, i.e. its name and (by subclassing) its type. An attribute in some data sample may for example be 'width' 'double'.

Field Detail

name

```
private final java.lang.String name
```

Specifications: spec_public

Constructor Detail

Attribute

```
public Attribute(java.lang.String name)
```

Parameters:

name - The attribute's name.

Method Detail

getName

```
public java.lang.String getName()
```

Returns:

The attribute's name.

isNumerical

```
public abstract boolean isNumerical()
```

Instead of doing run time checks to distinguish between different specializations of this class, this is done with this method. I don't like it, but I don't like run time type checks either, and was thus not motivated to learn and use them in Java. And a better design eluded me.

Returns:

If this is a numerical or nominal attribute.

parseValue

```
public abstract java.lang.Object parseValue(java.lang.String value)
```

Used to parse the data input. Converts a value of this attribute type given as a string to the native representation, e.g. Integer, Double, String.

Parameters:

value - The string representation of a valid attribute value.

Throws:

Termination - If value is not a valid value representation.

createNormalizer

```
public abstract NormalizerAttribute createNormalizer()
```

Returns:

A normalizer specific for this attribute's instance.

▪ Class Config

```
java.lang.Object  
└─ nn.Config
```

```
public class Config  
extends java.lang.Object
```

Contains all configuration, i.e. the structure of the neural network, the validation options,

Field Detail

dataFileName

```
private java.lang.String dataFileName  
    Specifications: spec_public
```

options

```
private final java.util.ArrayList options  
    Specifications: spec_public
```

targetAttributeOption

```
private final OptionString targetAttributeOption  
    Specifications: spec_public
```

targetAttribute

private Attribute **targetAttribute**
the target attribute
Specifications: spec_public

targetAttributeIndex

private int **targetAttributeIndex**
the index of the targetAttribute within the attributes in the input data schema, i.e.
the i.th attribute defined in the input (starting counting at 0).
Specifications: spec_public

hiddenLayers

private final OptionNats **hiddenLayers**
Specifications: spec_public

learningRate

private final OptionLearningRate **learningRate**
Specifications: spec_public

momentum

private final OptionMomentum **momentum**
Specifications: spec_public

epochs

private final OptionNat **epochs**
Specifications: spec_public

n_fold_validation

private final OptionNat **n_fold_validation**
Specifications: spec_public

Constructor Detail

Config

public **Config**()
Creates the default configuration.

Method Detail

optionForFlag

public [Option](#) **optionForFlag**(java.lang.String flag)

Parameters:

flag - the command line flag corresponding to this option

Returns:

the option with the given flag, i.e. hiddenLayers for '--hidden-layers'.

getOptions

```
public java.util.Iterator getOptions()
```

Returns:

An iterator over all options.

getDataFileName

```
public java.lang.String getDataFileName()
```

Returns:

Returns the name of the data file.

setDataFileName

```
public void setDataFileName(java.lang.String dataFileName)
```

Parameters:

dataFileName - The name of the data file to set.

getTargetAttribute

```
public Attribute getTargetAttribute()
```

Returns:

Returns the targetAttribute of the data sample.

getTargetAttributeIndex

```
public int getTargetAttributeIndex()
```

Returns:

Returns the index of the targetAttribute within the attributes of the data sample, starting with 0.

getHiddenLayers

```
public java.util.ArrayList getHiddenLayers()
```

Returns:

Returns the number of nodes per hidden layers as an Integer list.

getLearningRate

```
public double getLearningRate()
```

Returns:

Returns the learning rate of the neural network.

getMomentum

```
public double getMomentum()
```

Returns:

Returns the momentum of the neural network.

getEpochs

```
public int getEpochs()
```

Returns:

Returns the epochs used to train the neural network.

getNFCrossValidation

```
public int getNFCrossValidation()
```

Returns:

Returns the number of folds of the n-fold cross validation.

updateTargetAttribute

```
public void updateTargetAttribute(Sample sample)
```

If the target attribute was given as a command line flag its name is verified.

Otherwise, the last attribute given in the input is selected as the target attribute.

Parameters:

`sample` - the data sample to learn

■ **Class ConfusionMatrix**

java.lang.Object

└─ **nn.ConfusionMatrix**

```
public class ConfusionMatrix
```

```
extends java.lang.Object
```

A confusion matrix for a nominal target attribute.

Field Detail

nominal

```
private final java.util.ArrayList nominal
```

Specifications: `spec_public`

sampleSize

```
private int sampleSize
```


Specifications: spec_public

matrix

```
private java.util.HashMap matrix
```

Specifications: spec_public

Constructor Detail

ConfusionMatrix

```
public ConfusionMatrix(AttributeNominal attribute)
```

Creates a confusion matrix for the given nominal Attribute. The confusion matrix is then built incrementally as the records are evaluated against the network and the results are registered ([register \(String, String\)](#)).

Method Detail

register

```
public void register(java.lang.String correct,  
                    java.lang.String computed)
```

Registers the performance of the model on a record, i.e. gives the correct and the computed value of the nominal attribute.

print

```
public void print(java.io.PrintStream out)
```

Prints the evaluation:

- the number of correctly classified records,
- the recall, precision, and f1-measure per value,
- and the confusion matrix.

When target is the value to measure, then

- true positive is the number of times target was correctly classified,
- false negative is the number of times target was the correct output, but another value was computed as output.
- false positive is the number of times target was not the correct output, but was computed as output.

For example, take the confusion matrix

- - A B C
- A 8 1 1
- B 3 7 0
- C 1 0 9

where the rows contain the correct and the columns the computed output. Totally $8 + 7 + 9 = 24$ out of 30 records are classified correctly. For the value A we get

- true positive = 8
- false positive = $3 + 1 = 4$
- false negative = $1 + 1 = 2$

Based on this, for the value A we get

- precision = true positive / (true positive + false positive) = $8 / (8 + 4) = 0.66$
- recall = true positive / (true positive + false negative) = $8 / (8 + 2) = 0.8$
- f1-measure = $2 * \text{precision} * \text{recall} / (\text{precision} + \text{recall}) = 2 * 0.66 * 0.8 / (0.66 + 0.8) = 0.73$

Parameters:

out - The stream to print to.

▪ Class Connection

java.lang.Object
└─ nn.Connection

```
public class Connection
extends java.lang.Object
```

A (weighted) connection between two nodes.

Field Detail

source

```
private final Node source
Specifications: spec_public
```

target

```
private final NodeHidden target
Specifications: spec_public
```

weight

```
private final Weight weight
Specifications: spec_public
```

Constructor Detail

Connection

```
public Connection(Node source,NodeHidden target,  
Config config)
```

Creates the connection between the source and target code. Does not register itself to the source or target node.

Parameters:

source - Connected from this node.

target - Connected to this node.

Method Detail

getSource

```
public Node getSource()
```

Returns:

The source node.

getTarget

```
public NodeHidden getTarget()
```

Returns:

The target node.

getWeight

```
public Weight getWeight()
```

Returns:

The weight.

▪ Class EvalArgs

java.lang.Object

└─ **mn.EvalArgs**

```
public class EvalArgs
```

```
extends java.lang.Object
```

Parses the command line arguments and evaluates them.

Field Detail

helpFlag

```
public static final java.lang.String helpFlag
```

flag for printing a help synopsis.

Constructor Detail

EvalArgs

```
public EvalArgs()
```

Method Detail

evalCommandLine

```
public static void evalCommandLine(Config config,  
                                   java.lang.String[] args)
```

Evaluate command line flags. The flags are mostly given in config, plus special ones like helpFlag defined in this class.

Parameters:

config - The configuration to be modified based on the command line arguments.
args - The command line arguments

Throws:

Termination - If the arguments are malformed.

isFlag

```
protected static boolean isFlag(java.lang.String flag)
```

Is this string a valid flag? Flags start with '--' or '-', e.g. '--help'.

▪ Class Main

```
java.lang.Object  
└─ nn.Main
```

```
public class Main  
extends java.lang.Object
```

Main class - contains the main function.

Constructor Detail

Main

```
public Main()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Main class - evaluates the command line, reads the data, runs the neural network, validates it, and outputs the results.

Parameters:

args - command line options

validate

```
private static void validate(Config config,  
                             Sample sample,  
                             NeuralNetwork model,  
                             NeuralNetwork initialNet)
```

performs validation of the model and outputs the computed metrics

Parameters:

config - system configuration

sample - data sample

model - the network trained on the sample

initialNet - a copy of model in its initial state, for performing cross validation

▪ Class NeuralNetwork

java.lang.Object

└ **nn.NeuralNetwork**

```
public class NeuralNetwork
```

```
extends java.lang.Object
```

A fully-connected feed-forward neural network.

Field Detail

config

```
private final Config config
```

Specifications: spec_public

schema

```
private final Schema schema
```

Specifications: spec_public

normalizer

```
private final NormalizerSample normalizer
```

Specifications: spec_public

layers

```
private java.util.ArrayList layers
    Specifications: spec_public
```

Constructor Detail

NeuralNetwork

```
public NeuralNetwork(Config config,
                    Sample sample)
    Creates a neural network based on the given sample.
```

Parameters:

`config` - The configuration.
`sample` - The sample to learn.

NeuralNetwork

```
public NeuralNetwork(NeuralNetwork network)
    Creates an independent copy of network, with the same setup, i.e. identical layers,
    nodes, weights, ...
```

Method Detail

createLayers

```
private void createLayers()
    Creates the network's layers, connects them, and assigns random initial weights.
```

copy

```
public NeuralNetwork copy()
    Returns:
    an independent copy of the network, with the same setup, i.e. identical layers,
    nodes, weights, ...
    Specifications: pure
```

reset

```
protected void reset()
    Clears cached values within the network remaining from the last run.
```

run

```
public void run(Sample sample)
    Trains the network on the given data sample. The sample must use the same
    schema as the schema used when creating the network. The parameters like
    epochs, learning rate, ..., are taken from the config used in the constructor.
    Parameters:
```

sample - The data sample to learn.

validate

```
public void validate(Sample sample,  
                    Validation validation)
```

Run the network on the sample, and tell validation about the correct and computed output for each record. The sample must use the same schema as use to create the network.

Parameters:

sample - The data sample to learn.

validation - The validation object to extend.

printWeights

```
private void printWeights(java.io.PrintStream out,  
                          NodeHidden node,  
                          int layer)
```

Prints the incoming weights of a node with their weight.

Parameters:

out - The stream to print to.

node - The node whose incoming weights are to be printed.

layer - The layer of the node.

print

```
public void print(java.io.PrintStream out)
```

Prints the network, layer by layer and node by node, along with their weight.

Parameters:

out - The stream to print to.

▪ Class Node

java.lang.Object

└ nn.Node

Direct Known Subclasses:

NodeHidden, NodeInput

```
public abstract class Node
```

```
extends java.lang.Object
```

A node of a neural network.

Field Detail

inConnections

```
final java.util.ArrayList inConnections  
    Specifications: spec_public
```

outConnections

```
final java.util.ArrayList outConnections  
    Specifications: spec_public
```

Constructor Detail

Node

```
public Node()  
    Creates a new node.
```

Node

```
public Node(Node node)
```

Method Detail

copy

```
public abstract Node copy()  
    Returns:  
    an independent copy of this node.
```

getInConnections

```
protected java.util.List getInConnections()  
    Returns:  
    The connections from the previous layer.
```

getOutConnections

```
protected java.util.List getOutConnections()  
    Returns:  
    The connections to the next layer.
```

connectFrom

```
public void connectFrom(Connection connection)  
    Adds a connection to the previous layer.  
    Parameters:  
    connection - The connection.
```

connectTo


```
public void connectTo(Connection connection)
```

Adds a connection to the next layer.

Parameters:

`connection` - The connection.

reset

```
public abstract void reset()
```

Needs to be called each time a new record is fed to the network. This invalidates the old output, and the old backpropagation data.

getOutput

```
public abstract double getOutput()
```

Compute the output of this node.

Returns:

The node's output.

propagate

```
public abstract void propagate()
```

Do backpropagation.

▪ **Class NodeHidden**

java.lang.Object

└ **nn.Node**

└ **nn.NodeHidden**

Direct Known Subclasses:

NodeOutput

```
public class NodeHidden
```

```
extends Node
```

A node of a hidden layer of a neural network.

Field Detail

config

```
private final Config config
```

Specifications: `spec_public`

weight

```
private final Weight weight
```

Specifications: `spec_public`

output

private double **output**
 Specifications: spec_public

outputValid

private boolean **outputValid**
 Is output in sync, or does it have to be recomputed?
 Specifications: spec_public

delta

protected double **delta**
 Cache for the current delta of this node (for backpropagation).
 Specifications: spec_public

deltaValid

private boolean **deltaValid**
 Is delta in sync, or does it have to be recomputed?
 Specifications: spec_public

Constructor Detail

NodeHidden

public **NodeHidden**(Config config)
 Creates anew node of a hidden layer and initializes its weight randomly.

NodeHidden

public **NodeHidden**(NodeHidden node)

Method Detail

copy

public Node **copy**()

getWeight

public Weight **getWeight**()
 Returns:
 Returns the weight.

isOutputValid

protected boolean **isOutputValid**()

Returns:

Is the current output valid, or does it have to be recomputed?.

setOutputValid

protected void **setOutputValid**(boolean outputValid)

Parameters:

outputValid - Validate/Invalidate the cached output.

isDeltaValid

protected boolean **isDeltaValid**()

Returns:

Is the current delta valid, or does it have to be recomputed?.

setDeltaValid

protected void **setDeltaValid**(boolean deltaValid)

Parameters:

deltaValid - Validate/Invalidate the cached delta.

reset

public void **reset**()

Description copied from class: Node

Needs to be called each time a new record is fed to the network. This invalidates the old output, and the old backpropagation data.

getOutput

public double **getOutput**()

The output of the node is computed as

- the sum of the weighted input of all incoming connections,
- plus the node's weight,
- normalized by the sigmoid function.

Returns:

The node's output.

setOutput

protected void **setOutput**(double output)

Cache the node's output.

Parameters:

output - The current output.

getDelta

```
public double getDelta()
```

Compute the delta of this node in backpropagation. The delta of a hidden node is computed as

- the node's output,
- times (1 minus the node's output),
- times the sum of the weighted deltas of the outgoing connections.

Returns:

The node's delta.

setDelta

```
protected void setDelta(double delta)
```

Cache the node's delta in backpropagation.

Parameters:

delta - The current delta.

propagate

```
public void propagate()
```

Does backpropagation. Adjusts the node's weight and each incoming connection's weight by calling `weight.propagate(double, double)`.

▪ Class NodeInput

java.lang.Object

└ nn.Node

└ nn.NodeInput

```
public class NodeInput
```

```
extends Node
```

An input node of a neural network.

Field Detail

input

```
private double input
```

The last value fed into this input node.

Specifications: spec_public

Constructor Detail

NodeInput

```
public NodeInput()  
    Creates a node of the input layer.
```

NodeInput

```
public NodeInput(NodeInput node)  
    Specifications: pure  
    public normal_behavior  
    requires node != null;
```

Method Detail

copy

```
public Node copy()
```

reset

```
public void reset()  
    Description: same as class Node
```

setInputValue

```
public void setInputValue(double value)  
    Sets the input of this node (when a record is fed into the network).  
    Parameters:  
    value - Input value.
```

getOutput

```
public double getOutput()  
    Just returns the value fed into this node.  
    Returns:  
    Network Input.
```

propagate

```
public void propagate()  
    Does nothing - input nodes are not adjusted.
```

■ Class NodeOutput

```
java.lang.Object  
├─ nn.Node  
│   └─ nn.NodeHidden  
│       └─ nn.NodeOutput
```

```
public class NodeOutput
  extends NodeHidden
```

A node of the output layer of a neural network.

Field Detail

correctOutput

```
double correctOutput
  The correct output for the record fed into the network.
Specifications: spec_public
```

Constructor Detail

NodeOutput

```
public NodeOutput(Config config)
  Creates a node of the output layer.
```

NodeOutput

```
public NodeOutput(NodeOutput node)
```

Method Detail

copy

```
public Node copy()
Overrides:
  copy in class NodeHidden
```

getDelta

```
public double getDelta()
  The delta of an output node is computed as
```

- the node's output,
- times (1 minus the node's output),
- times (the correct output minus the node's output).

Overrides:

getDelta in class NodeHidden

Returns:

The node's delta.

setCorrectOutputValue

```
public void setCorrectOutputValue(double value)
    Tells the node the correct target value of the input record.
```

▪ Class NormalizerAttribute

```
java.lang.Object
└─ nn.NormalizerAttribute
```

Direct Known Subclasses:

```
NormalizerAttributeInt, NormalizerAttributeNominal, NormalizerAttributeReal
```

```
public abstract class NormalizerAttribute
    extends java.lang.Object
```

A normalizer for an attribute and a sample. A normalizer is created for an attribute (of a specific type), and computes the normalization function based on a concrete sample.

Field Detail

attribute

```
private final Attribute attribute
    Specifications: spec_public
```

Constructor Detail

NormalizerAttribute

```
public NormalizerAttribute(Attribute attribute)
    Initializes the normalizer based on the given attribute.
    Parameters:
    attribute - The attribute to normalize.
```

Method Detail

getAttribute

```
public Attribute getAttribute()
    Returns:
    The normalized attribute.
```

register

```
public abstract void register(java.lang.Object value)
    Registers a data value as part of the sample to normalize.
```

normalize

```
public abstract java.util.List normalize(java.lang.Object value)
    Performs min-max normalization on a numeric attribute, and 1-of-N encoding for
    on a nominal attribute.
```

Parameters:

value - The value to normalize.

Returns:

The normalized value as a list of doubles.

denormalize

```
public abstract java.lang.Object denormalize(java.util.List value)
    Denormalizes a previously normalized value, i.e. is the inverse function of
    normalize(Object)
```

Parameters:

value - The normalized value.

Returns:

Value denormalized.

normalizedSize

```
public abstract int normalizedSize()
```

Returns:

The number of reals returned by normalize(Object).

=====

▪ Class Option

```
java.lang.Object
└─ nn.Option
```

Direct Known Subclasses:

OptionDouble, OptionInt, OptionNats, OptionString

```
public abstract class Option
```

```
extends java.lang.Object
```

A command line flag representing a configuration option Concrete subclasses should add a value field of the wanted type.

Field detail

flag

```
private final java.lang.String flag
```

Specifications: spec_public

description


```
private final java.lang.String description  
Specifications: spec_public
```

Constructor Detail

Option

```
public Option(java.lang.String flag,  
              java.lang.String description)
```

Parameters:

flag - The command line flag, e.g. '--epochs'.

description - A short description, e.g 'number of epochs used to train the neural net'.

Method Detail

getFlag

```
public java.lang.String getFlag()
```

Returns:

The flag corresponding to this option.

getDescription

```
public java.lang.String getDescription()
```

Returns:

The description corresponding to this option.

getType

```
public abstract java.lang.String getType()
```

Returns:

A textual description of the option's type, e.g. 'int'.

valueToString

```
public abstract java.lang.String valueToString()
```

Returns:

The string representation of this configuration value.

setValue

```
public abstract void setValue(java.lang.String argument)
```

Parameters:

argument - The option value is set based on this string. argument must contain a valid value representation for the concrete Option type.

Throws:

[Termination](#) - If argument is not a valid value representation.

▪ Class `OptionString`

```
java.lang.Object
├─ nn.Option
│   └─ nn.OptionString
public class OptionString
extends Option
```

Option specialized for type String.

Field Detail

value

```
private java.lang.String value
Specifications: spec_public
```

Constructor detail

OptionString

```
public OptionString(java.lang.String flag,
                    java.lang.String description,
                    java.lang.String value)
```

Parameters:

`value` - The initial value to be represented by this object.

Method Detail

getType

```
public java.lang.String getType()
```

getValue

```
public java.lang.String getValue()
```

Returns:

The current value represented by this object.

valueToString

```
public java.lang.String valueToString()
```

setValue

```
public void setValue(java.lang.String value)
```

Sets this object's value to the given value.

Parameters:

value - the new value.

▪ **Class Print**

```
java.lang.Object
└─ Print
public class Print
extends java.lang.Object
```

Printing / Output.

Field Detail

flagWidth

```
private static final int flagWidth
    The width of the flag name column.
```

typeWidth

```
private static final int typeWidth
    The width of the type column.
```

Constructor Detail

Print

```
public Print()
```

Method Detail

fill

```
public static void fill(java.lang.StringBuffer aString, int width)
    Extends the string with ' ' at the end until its size is >= width.
```

fill

```
public static java.lang.String fill(java.lang.String aString,
                                     int width)
    Like fill\(StringBuffer, int\) for a String.
```

fill

```
public static java.lang.String fill(int number,
```

```
int width)
```

Like [fill\(StringBuffer, int\)](#) for an int.

fill

```
public static java.lang.String fill(double number,  
int width)
```

Like [fill\(StringBuffer, int\)](#) for an double.

printHelpFlag

```
private static void printHelpFlag(java.io.PrintStream out,  
java.lang.String flag,  
java.lang.String type,  
java.lang.String value,  
java.lang.String description)
```

Prints a command line flag with a short description.

Parameters:

out - Where to print to.

flag - The command line flag.

type - The flag's type.

value - The current value of the flag.

description - A short description of the flag.

printHelpFlag

```
private static void printHelpFlag(java.io.PrintStream out,  
Option option)
```

Prints a command line flag with a short description.

Parameters:

out - Where to print to.

option - The command line flag.

printHelp

```
public static void printHelp(java.io.PrintStream out)
```

Prints a short help including all flags with a short description.

Parameters:

out - Where to print to.

printConfigOption

```
private static void printConfigOption(java.io.PrintStream out,  
java.lang.String option,  
java.lang.String description)
```

Prints a configuration option and its current value.

Parameters:

out - Where to print to.

option - The configuration option.
description - A short description of the option.

printConfigOption

```
private static void printConfigOption(java.io.PrintStream out,  
                                       Option option)
```

Prints a configuration option and its current value.

Parameters:

out - Where to print to.
option - The configuration option.

printConfig

```
public static void printConfig(java.io.PrintStream out,  
                               Config config)
```

Prints the configuration, i.e. all variables and their current values.

Parameters:

out - Where to print to.

=====

▪ Class ReadArff

```
java.lang.Object  
└─ nn.ReadArff  
public class ReadArff  
extends java.lang.Object
```

A very simple parser of the arff format. Doesn't handle missing or sparse data.

Constructor Detail

ReadArff

```
public ReadArff()
```

Method Detail

readArff

```
public static Sample readArff(java.lang.String fileName)
```

Reads data in the arff format from a file into a sample object.

Parameters:

fileName - The file name of the arff file to parse.

Returns:

The data sample created from the data in the input file.

Throws:

Termination - If input can not be parsed.

skipLine

```
private static boolean skipLine(java.lang.String line)
```

Checks if an input line can be skipped because it contains only white space or a comment.

Parameters:
line - The line to check.

Returns:
True iff the line contains a comment or only whitespace.

readName

```
private static java.lang.String  
readName(java.io.BufferedReader reader)  
throws java.io.IOException
```

Reads the name of the data schema.

Parameters:

reader - The input file.

Returns:

The name of the schema.

Throws:

Termination - If '@relation' is not the next valid line.
java.io.IOException

readRelation

```
private static Schema readRelation(java.io.BufferedReader reader)  
throws java.io.IOException,  
Termination
```

Reads the attributes of the data schema.

Parameters:

reader - The input file.

Returns:

The data schema.

Throws:

Termination - If the '@attribute' definitions are not next in the file.
java.io.IOException

readData

```
private static Sample readData(java.io.BufferedReader reader,  
java.lang.String schemaName,  
Schema schema)  
throws java.io.IOException,  
Termination
```

Reads the data of the relation.

Parameters:

reader - The input file.
 schemaName - The name of the schema.
 schema - The schema of the data.

Returns:

The data sample.

Throws:

Termination - If the input is malformed.
 java.io.IOException

parseCSV

```
private static java.util.ArrayList parseCSV(java.lang.String line)
                                     throws Termination
```

Splits a comma separated string into its components. Removes enclosing whitespace.

Parameters:

line - The string to split.

Returns:

The data schema.

Throws:

Termination - If the '@attribute' definitions are not next in the file.

unquote

```
private static java.lang.String unquote(java.lang.String string)
                                     throws Termination
```

Unquotes a string, i.e. removes enclosing "" characters, and removes enclosing whitespace.

Parameters:

string - The string to unquote.

Returns:

The unquoted string.

Throws:

Termination

=====

- **Class Sample**

```
java.lang.Object
└─ nn.Sample
public class Sample
extends java.lang.Object
```

Represents a data sample, i.e. schema and data.

Field Detail

name

```
private final java.lang.String name  
    Specifications: spec_public
```

schema

```
private final Schema schema  
    Specifications: spec_public
```

records

```
private final java.util.ArrayList records  
    Specifications: spec_public
```

Constructor Detail

Sample

```
public Sample(java.lang.String name,  
              Schema schema)
```

Creates an empty sample based on its schema. The actual data is filled in later on.

Parameters:

name - The sample name.

schema - The sample schema.

Method Detail

getName

```
public java.lang.String getName()
```

Returns:

The sample name.

addData

```
public void addData(java.util.ArrayList data)  
    throws Termination
```

Adds a (string) data record to the sample. The record is given in string format, each value is transformed internally to the appropriate attribute value.

Parameters:

data - The record as strings.

Throws:

Termination - If the data does not correspond to the sample schema.

addRecord

```
public void addRecord(java.util.ArrayList record)
```

Adds a data record to the sample. Each record value must be a value corresponding to its attribute type as specified in the data schema.

Throws:

Termination - If the data does not correspond to the sample schema.

addSample

```
public void addSample(Sample sample)
```

Adds all records of sample. Both samples have to use the same schema.

Parameters:

sample - A data sample.

partition

```
public java.util.ArrayList partition(int partitions)
```

Partitions the sample randomly into partition parts.

Parameters:

partitions - The number of partitions to split to.

Returns:

The partitions.

getSchema

```
public Schema getSchema()
```

Returns:

The schema.

getSampleSize

```
public int getSampleSize()
```

Returns:

The current sample size.

getRecord

```
public java.util.List getRecord(int index)
```

Retrieves the index.th record of the data sample.

Parameters:

index - The index of the record to return.

Returns:

The current sample size.

toString

```
public java.lang.String toString()
    Overrides:
    toString in class java.lang.Object
    Returns:
    A string representation of the schema and the sample.
```

▪ Class Schema

```
java.lang.Object
└─ nn.Schema
public class Schema
extends java.lang.Object
```

The schema of a sample, i.e its attributes definitions.

Field Detail

attributes

```
private final java.util.ArrayList attributes
    Specifications: spec_public
```

Constructor Detail

Schema

```
public Schema()
    Specifications: pure
```

Method Detail

addAttribute

```
public void addAttribute(Attribute attribute)
    Adds a new attribute to the schema. Order matters, attributes are indexed in order
    of addition, starting from 0.
    Parameters:
    attribute - A new attribute of the schema.
```

getNumberOfAttributes

```
public int getNumberOfAttributes()
    Returns:
    The number of attributes of the schema.
```

getAttribute

```
public Attribute getAttribute(int index)
```

Parameters:

index - The index of the attribute to return.

Returns:

The requested attribute.

toString

```
public java.lang.String toString()
```

Overrides:

toString in class java.lang.Object

▪ **Class Termination**

```
java.lang.Object  
└─ java.lang.Throwable  
    └─ java.lang.Error  
        └─ nn.Termination
```

All Implemented Interfaces:

java.io.Serializable

```
public class Termination
```

```
extends java.lang.Error
```

This Error class is used to abort the program. As this program is merely a simple demo, no fancy exception handling is done anywhere - it merely terminates with a descriptive error message whenever an unexpected error is encountered.

Constructor Detail

Termination

```
public Termination(java.lang.String message)
```

Parameters:

message - Description of the error

▪ **Class Validation**

```
java.lang.Object  
└─ nn.Validation  
public class Validation  
extends java.lang.Object
```

Validates a model on a sample.

Field Detail

meanAbsoluteError

```
private double meanAbsoluteError  
    Specifications: spec_public
```

rootMeanSquaredError

```
private double rootMeanSquaredError  
    Specifications: spec_public
```

confusionMatrix

```
private final ConfusionMatrix confusionMatrix  
    Generate a confusion matrix for a nominal target attribute.
```

sampleSize

```
private int sampleSize  
    Specifications: spec_public
```

Constructor Detail

Validation

```
public Validation(Config config)  
    Create a validation object for the target attribute specified in config. Build the  
    validation incrementally via register\(List, List, NormalizerSample\).
```

Method Detail

updateMeanAbsoluteError

```
protected void updateMeanAbsoluteError(java.util.List correct,  
                                         java.util.List computed)  
    Updates the mean absolute error with one record's classification result.  
Parameters:  
    correct - The correct classification (denormalized).  
    computed - The classification computed by the model (denormalized).
```

updateRootMeanSquaredError

```
protected void updateRootMeanSquaredError(java.util.List correct,  
                                             java.util.List computed)  
    Updates the root mean squared error with one record's classification result.  
Parameters:  
    correct - The correct classification (denormalized).
```

`computed` - The classification computed by the model (denormalized).

register

```
public void register(java.util.List correct,  
                    java.util.List computed,  
                    NormalizerSample normalizer)
```

Registers the performance of the model on a record, i.e. gives the correct and the computed output, both in normalized form.

Parameters:

`correct` - The correct classification (normalized).

`computed` - The classification computed by the model (normalized).

`normalizer` - Normalizer to denormalize correct and computed.

print

```
public void print(java.io.PrintStream out)
```

Prints the evaluation:

- Mean Absolute Error: The sum of the absolute differences between the correct and computed output for each record, divided by the number of records.
- Root Mean Squared Error: The square root of (the sum of the differences between the correct and computed output squared for each record divided by the number of records).
- Confusion Matrix: `ConfusionMatrix.print(PrintStream)`

Parameters:

`out` - The stream to print to.

=====

▪ **Class Weight**

```
java.lang.Object  
└─nn.Weight  
public class Weight  
extends java.lang.Object
```

A weight of the network.

Field Detail

config

```
private final Config config  
Specifications: spec_public
```

weight

private double **weight**
The actual weight.
Specifications: spec_public

adjustment

private double **adjustment**
The previous weight adjustment.
Specifications: spec_public

Constructor Detail

Weight

public **Weight**(Config config)
Creates a new weight, randomly initialized in [0; 1[

Weight

public **Weight**(Weight weight)
Creates an independent copy of weight.

Method Detail

getWeight

public double **getWeight**()
Returns:
The weight.

setWeight

public void **setWeight**(double weight)
Parameters:
weight - The new weight.

propagate

public void **propagate**(double deltaNode,
double input)
Does backpropagation, i.e. changes the current weight based on the value input to the connected node, and the nodes responsibility for the error. Takes the learning rate `Config.getLearningRate()` and the momentum `Config.getLearningRate()` into account by setting the weight to

- the current weight

- plus the learning rate times the target node's delta, times the target node's input,
- plus the momentum times the previous weight adjustment.

Parameters:

`deltaNode` - The responsibility of the connected node for the error.

`input` - The value previously input to the connected node.

==*****=====*****=====*****=====

11. Future Work

This project can be extended by adding some functionality like:

1. adding graphical user interface, this project only supports command line.
2. it can be implemented for all neural network topologies and model, this project implements few of them.
3. error rate in learning process can be minimize by using efficient algorithms (genetic algorithms, etc.).
4. neural-network methods are thought to have two limitations that make them poorly suited to data-mining tasks: their learned hypotheses are often incomprehensible, and training times are often excessive. We can eliminate these limitations

12. Conclusion

This project aimed at implementing a basic neural network, and providing usable information about the neural networks that is their architecture, functionality and efficiency.

This project will be helpful in understanding neural networks and their behavior, it will show how neural network is useful in data mining problem and can be the best solution for such problem.

We can see that, for some problems, neural networks are more suitable i.e., they do a better job of learning the target concept than other commonly used data-mining methods.

We have not attempted to provide an exhaustive survey of the available neural-network algorithms that are suitable for data mining. Instead, we have described a subset of these methods, selected to illustrate the breadth of relevant approaches as well as the key issues that arise in applying neural networks in a data-mining setting. It is our hope that our discussion of neural-network approaches will serve to inspire some interesting applications of these methods to challenging datamining problems.

13. References

1. Artificial Intelligence, Elaine Rich, Kevin Knight, second edition, TMH publication
2. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
3. Neural Networks at Pacific Northwest National Laboratory
<http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html>
4. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
5. Neural Networks by Eric Davalo and Patrick Naim
6. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
7. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography.
Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
8. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab. Neural Networks, Eric Davalo and Patrick Naim
9. Pattern Recognition of Pathology Images
<http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html>
10. Richard Roiger and Cichael Geatz, *Data Mining: A Tutorial-Based Primer*, Addison-Wesley, 2003.
11. Robert Groth, *Data Mining: Building Compleitive Advantage*, Prentice Hall, 2000.
12. Dorian, P.: Data Preparation for Data Mining, Morgan Kaufmann, 1999.
13. Weiss, S.M. and Kulikowski, C.A.: Computer Systems That Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems, Morgan Kaufmann, 1991.
14. Weiss, S.M. and Indurkhya, N.: Predictive Data Mining: A Practical Guide, Morgan Kaufmann, 1997.