Computational Intelligence

An Introduction

Computational Intelligence

An Introduction

A.P. Engelbrecht
University of Pretoria
South Africa

To my parents, Jan and Magriet Engelbrecht,
without whose loving support
this would not have happened.

Contents

List of Figures

Preface

Man has learned much from studies of natural systems, using what has been learned to develop new algorithmic models to solve complex problems. This book presents an introduction to some of these technological paradigms, under the umbrella of computational intelligence (CI). In this context, the book includes artificial neural networks, evolutionary computing, swarm intelligence and fuzzy logic, which are respectively models of the following natural systems: biological neural networks, evolution, swarm behavior of social organisms, and human thinking processes.

Why this book on computational intelligence? Need arose from a graduate course, where students do not have a deep background of artificial intelligence and mathematics. Therefore the introductory nature, both in terms of the CI paradigms and mathematical depth. While the material is introductory in nature, it does not shy away from details, and does present the mathematical foundations to the interested reader. The intention of the book is not to provide thorough attention to all computational intelligence paradigms and algorithms, but to give an overview of the most popular and frequently used models. As such, the book is appropriate for beginners in the CI field. The book is therefore also applicable as prescribed material for a third year undergraduate course.

In addition to providing an overview of CI paradigms, the book provides insights into many new developments on the CI research front (including material to be published in 2002) – just to tempt the interested reader. As such, the material is useful to graduate students and researchers who want a broader view of the different CI paradigms, also researchers from other fields who have no knowledge of the power of CI techniques, e.g. bioinformaticians, biochemists, mechanical and chemical engineers, economists, musicians and medical practitioners.

The book is organized in five parts. Part I provides a short introduction to the different CI paradigms and a historical overview. Parts II to V cover the different paradigms, and can be presented in any order.

Part II deals with artificial neural networks (NN), including the following topics: Chapter 2 introduces the artificial neuron as the fundamental part of a neural network, including discussions on different activation functions, neuron geometry and

learning rules. Chapter 3 covers supervised learning, with an introduction to different types of supervised networks. These include feedforward NNs, functional link NNs, product unit NNs and recurrent NNs. Different supervised learning algorithms are discussed, including gradient descent, scaled conjugate gradient, LeapFrog and particle swarm optimization. Chapter 4 covers unsupervised learning. Different unsupervised NN models are discussed, including the learning vector quantizer and self-organizing feature maps. Chapter 5 introduces radial basis function NNs which are hybrid unsupervised and supervised learners. Reinforcement learning is dealt with in chapter 6. Much attention is given to performance issues of supervised networks in chapter 7. Aspects that are included are measures of accuracy, analysis of performance, data preparation, weight initialization, optimal learning parameters, network architecture selection, adaptive activation functions and active learning.

Part III introduces several evolutionary computation models. Topics covered include: an overview of the computational evolution process in chapter 8. Chapter 9 covers genetic algorithms, chapter 10 genetic programming, chapter 11 evolutionary programming, chapter 12 evolutionary strategies, chapter 13 differential evolution, chapter 14 cultural evolution, and chapter 15 covers coevolution, introducing both competitive and symbiotic coevolution.

Part IV presents an introduction to two types of swarm-based models: Chapter 16 discusses particle swarm optimization and covers some of the new developments in particle swarm optimization research. Ant colony optimization is overviewed in chapter 17.

Part V deals with fuzzy systems. Chapter 18 presents an introduction to fuzzy systems with a discussion on membership functions, linguistic variables and hedges. Fuzzy inferencing systems are explained in chapter 19, while fuzzy controllers are discussed in chapter 20. An overview of rough sets is given in chapter 21.

The conclusion brings together the different paradigms and shows that hybrid systems can be developed to attack difficult real-world problems.

Throughout the book, assignments are given to highlight certain aspects of the covered material and to stimulate thought. Some example applications are given where they seemed appropriate to better illustrate the theoretical concepts.

Several Internet sites will be helpful as an additional. These include:

- http://citeseer.nj.nec.com/ which is an excellent search engine for AI-related publications;
- http://www.ics.uci.edu/~mlearn/MLRepository.html, a repository of data bases maintained by UCI;
- http://www.cs.toronto.edu/~delve/, another repository of benchmark problems.

- http://www.lirmm.fr/~reitz/copie/siftware.html, a source of commercial and free software.
- http://www.aic.nrl.navy.mil/~aha/research/machine-learning.html, a repository of machine learning resources
- http://dsp.jpl.nasa.gov/members/payman/swarm/, with resources on swarm intelligence.
- http://www.cse.dmu.ac.uk/~rij/fuzzy.html and
  http://www.austinlinks.com/Fuzzy/ with information on fuzzy logic.
- http://www.informatik.uni-stuttgart.de/ifi/fk/evolalg/, a repository for evolutionary computing.
- http://www.evalife.dk/bbase, another evolutionary computing and artificial life repository.
- http://news.alife.org/, a source for information and software on Artificial Life.

Part I

INTRODUCTION

Chapter 1

Introduction to Computational Intelligence

"Keep it simple:
as simple as possible,
but no simpler."
- A. Einstein

A major thrust in algorithmic development is the design of algorithmic models to solve increasingly complex problems. Enormous successes have been achieved through the modeling of biological and natural intelligence, resulting in so-called "intelligent systems". These intelligent algorithms include artificial neural networks, evolutionary computing, swarm intelligence, and fuzzy systems. Together with logic, deductive reasoning, expert systems, case-based reasoning and symbolic machine learning systems, these intelligent algorithms form part of the field of *Artificial Intelligence* (AI). Just looking at this wide variety of AI techniques, AI can be seen as a combination of several research disciplines, for example, computer science, physiology, philosophy, sociology and biology.

But what is intelligence? Attempts to find definitions of intelligence still provoke heavy debate. Dictionaries define intelligence as the ability to comprehend, to understand and profit from experience, to interpret intelligence, having the capacity for thought and reason (especially to a high degree). Other keywords that describe aspects of intelligence include creativity, skill, consciousness, emotion and intuition.

Can computers be intelligent? This is a question that to this day causes more debate than do definitions of intelligence. In the mid-1900s, Alan Turing gave much thought to this question. He believed that machines could be created that would mimic the processes of the human brain. Turing strongly believed that there was nothing the brain could do that a well-designed computer could not. Fifty years later

his statements are still visionary. While successes have been achieved in modeling biological neural systems, there are still no solutions to the complex problem of modeling intuition, consciousness and emotion – which form integral parts of human intelligence.

In 1950 Turing published his test of computer intelligence, referred to as the *Turing test* [Turing 1950]. The test consisted of a person asking questions via a keyboard to both a person and a computer. If the interrogator could not tell the computer apart from the human, the computer could be perceived as being intelligent. Turing believed that it would be possible for a computer with 10^9 bits of storage space to pass a 5-minute version of the test with 70% probability by the year 2000. Has his belief come true? The answer to this question is left to the reader, in fear of running head first into another debate! However, the contents of this book may help to shed some light on the answer to this question.

A more recent definition of artificial intelligence came from the IEEE Neural Networks Council of 1996: the study of how to make computers do things at which people are doing better. A definition that is flawed, but this is left to the reader to explore in one of the assignments at the end of this chapter.

This book concentrates on a sub-branch of AI, namely Computational Intelligence (CI) – the study of adaptive mechanisms to enable or facilitate intelligent behavior in complex and changing environments. These mechanisms include those AI paradigms that exhibit an ability to learn or adapt to new situations, to generalize, abstract, discover and associate. The following CI paradigms are covered: artificial neural networks, evolutionary computing, swarm intelligence and fuzzy systems. While individual techniques from these CI paradigms have been applied successfully to solve real-world problems, the current trend is to develop hybrids of paradigms, since no one paradigm is superior to the others in all situations. In doing so, we capitalize on the respective strengths of the components of the hybrid CI system, and eliminate weaknesses of individual components.

The rest of this book is organized as follows: Section 1.1 presents a short overview of the different CI paradigms, also discussing the biological motivation for each paradigm. A short history of AI is presented in Section 1.2. Artificial neural networks are covered in Part II, evolutionary computing in Part III, swarm intelligence in Part IV and fuzzy systems in Part V. A short discussion on hybrid CI models is given in the conclusion of this book.

At this point it is necessary to state that there are different definitions of what constitutes CI. This book reflects the opinion of the author, and may well cause some debate. For example, Swarm Intelligence (SI) is classified as a CI paradigm, while most researchers are of the belief that it belongs only under Artificial Life. However, both Particle Swarm Optimization (PSO) and Anto Colony Optimization (ACO), as treated under SI, satisfy the definition of CI given above, and are therefore

included in this book as being CI techniques.

1.1 Computational Intelligence Paradigms

This book considers four main paradigms of Computation Intelligence (CI), namely artificial neural networks (NN), evolutionary computing (EC), swarm intelligence (SI) and fuzzy systems (FS). Figure 1.1 gives a summary of the aim of the book. In addition to CI paradigms, probabilistic methods are frequently used together with CI techniques, which is therefore shown in the figure. Soft computing, a term coined by Lotfi Zadeh, is a different grouping of paradigms, which usually refers to the collective set of CI paradigms and probabilistic methods. The arrows indicate that techniques from different paradigms can be combined to form hybrid systems.

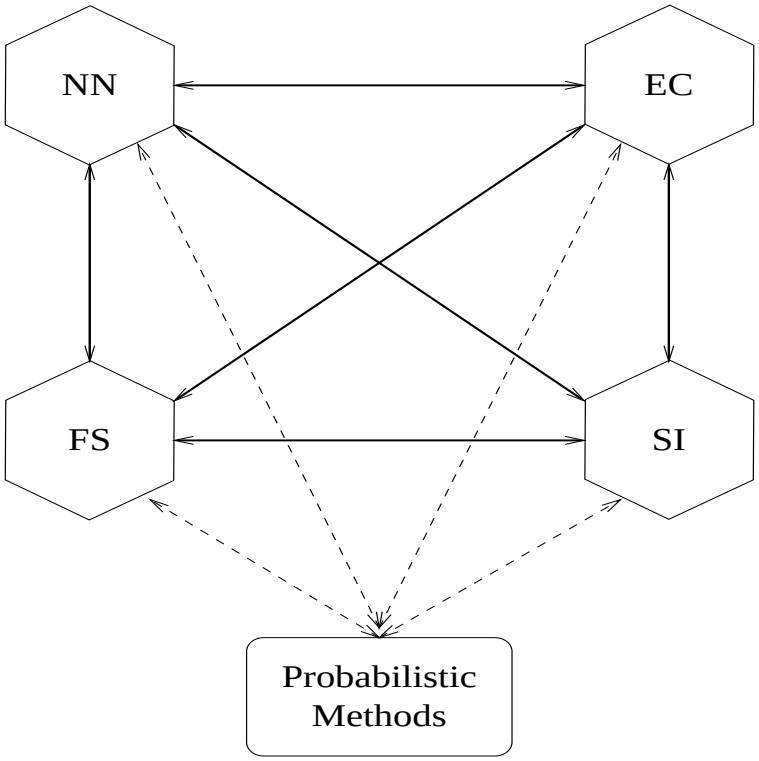


Figure 1.1: Illustration of CI paradigms

Each of the CI paradigms has its origins in biological systems. NNs model biological neural systems, EC models natural evolution (including genetic and behavioral evolution), SI models the social behavior of organisms living in swarms or colonies, and FS originated from studies of how organisms interact with their environment.

1.1.1 Artificial Neural Networks

The brain is a complex, nonlinear and parallel computer. It has the ability to perform tasks such as pattern recognition, perception and motor control much faster than any computer – even though events occur in the nanosecond range for silicon gates, and milliseconds for neural systems. In addition to these characteristics, others such as the ability to learn, memorize and still generalize, prompted research in algorithmic modeling of biological neural systems – referred to as *artificial neural networks* (NN).

It is estimated that there is in the order of 10-500 billion neurons in the human cortex, with 60 trillion synapses. The neurons are arranged in approximately 1000 main modules, each having about 500 neural networks. *Will it then be possible to truly model the human brain?* Not now. Current successes in neural modeling are for small artificial NNs aimed at solving a specific task. We can thus solve problems with a single objective quite easily with moderate-sized NNs as constrained by the capabilities of modern computing power and storage space. The brain has, however, the ability to solve several problems simultaneously using distributed parts of the brain. We still have a long way to go ...

The basic building blocks of biological neural systems are nerve cells, referred to as neurons. As illustrated in Figure 1.2, a neuron consists of a cell body, dendrites and an axon. Neurons are massively interconnected, where an interconnection is between the axon of one neuron and a dendrite of another neuron. This connection is referred to as a *synapse*. Signals propagate from the dendrites, through the cell body to the axon; from where the signals are propagated to all connected dendrites. A signal is transmitted to the axon of a neuron only when the cell "fires". A neuron can either inhibit or excite a signal.

An artificial neuron (AN) is a model of a biological neuron (BN). Each AN receives signals from the environment or other ANs, gathers these signals, and when fired, transmits a signal to all connected ANs. Figure 1.3 is a representation of an artificial neuron. Input signals are inhibited or excited through negative and positive numerical weights associated with each connection to the AN. The firing of an AN and the strength of the exiting signal are controlled via a function, referred to as the activation function. The AN collects all incoming signals, and computes a net input signal as a function of the respective weights. The net signal serves as input to the activation function which calculates the output signal of the AN.

An artificial neural network (NN) is a layered network of ANs. An NN may consist of an input layer, hidden layers and an output layer. ANs in one layer are connected, fully or partially, to the ANs in the next layer. Feedback connections to previous layers are also possible. A typical NN structure is depicted in Figure 1.4.

Several different NN types have been developed, for example (the reader should note

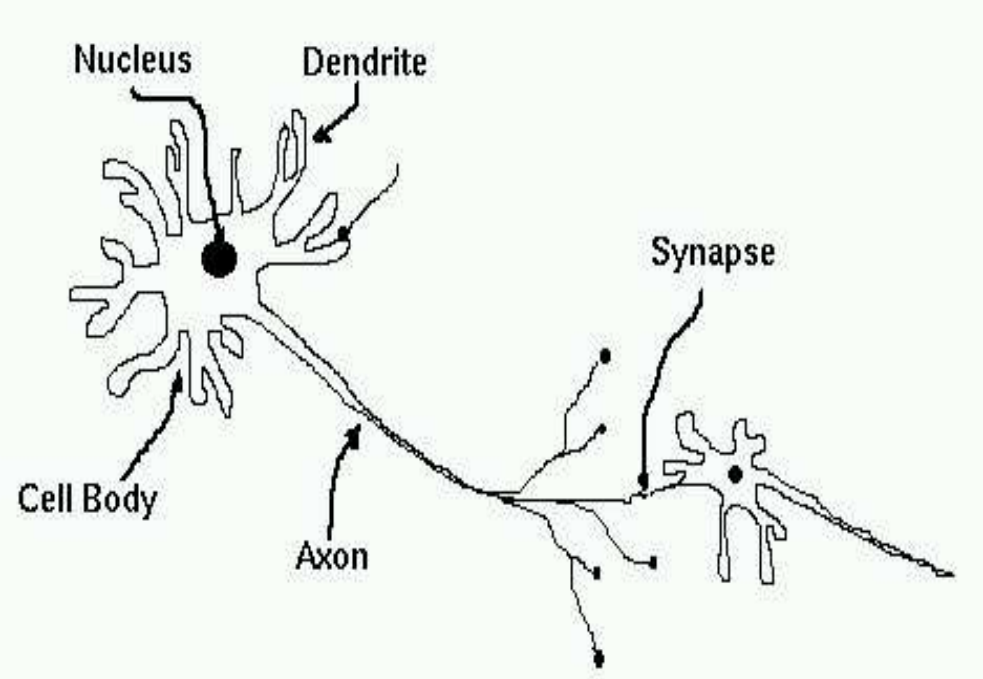


Figure 1.2: Illustration of a biological neuron

that the list below is by no means complete):

- single-layer NNs, such as the Hopfield network;
- multilayer feedforward NNs, including, for example, standard backpropagation, functional link and product unit networks;
- temporal NNs, such as the Elman and Jordan simple recurrent networks as well as time-delay neural networks;
- self-organizing NNs, such as the Kohonen self-organizing feature maps and the learning vector quantizer;
- combined feedforward and self-organizing NNs, such as the radial basis function networks.

These NN types have been used for a wide range of applications, including diagnosis of diseases, speech recognition, data mining, composing music, image processing, forecasting, robot control, credit approval, classification, pattern recognition, planning game strategies, compression and many others.

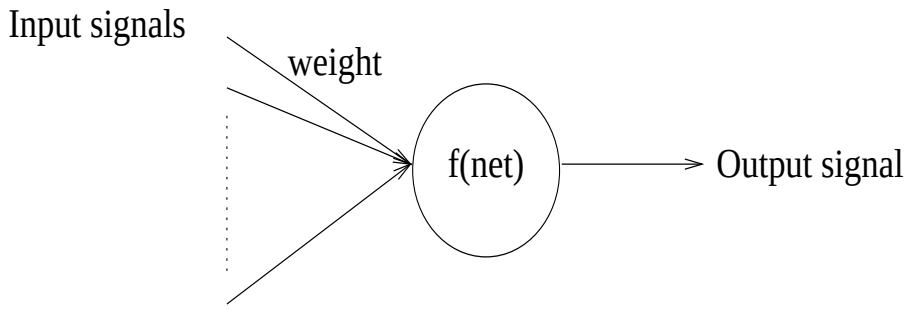


Figure 1.3: Illustration of an artificial neuron

1.1.2 Evolutionary Computing

Evolutionary computing has as its objective the model of natural evolution, where the main concept is survival of the fittest: the weak must die. In natural evolution, survival is achieved through reproduction. Offspring, reproduced from two parents (sometimes more than two), contain genetic material of both (or all) parents – hopefully the best characteristics of each parent. Those individuals that inherit bad characteristics are weak and lose the battle to survive. This is nicely illustrated in some bird species where one hatchling manages to get more food, gets stronger, and at the end kicks out all its siblings from the nest to die.

In evolutionary computing we model a *population* of individuals, where an individual is referred to as a *chromosome*. A chromosome defines the characteristics of individuals in the population. Each characteristic is referred to as a *gene*. The value of a gene is referred to as an *allele*. For each generation, individuals compete to reproduce offspring. Those individuals with the best survival capabilities have the best chance to reproduce. Offspring is generated by combining parts of the parents, a process referred to as *crossover*. Each individual in the population can also undergo mutation which alters some of the allele of the chromosome. The survival strength of an individual is measured using a *fitness* function which reflects the objectives and constraints of the problem to be solved. After each generation, individuals may undergo *culling*, or individuals may survive to the next generation (referred to as *elitism*). Additionally, behavioral characteristics (as encapsulated in phenotypes) can be used to influence the evolutionary process in two ways: phenotypes may influence genetic changes, and/or behavioral characteristics evolve separately.

Different classes of EC algorithms have been developed:

- **Genetic algorithms** which model genetic evolution.
- **Genetic programming** which is based on genetic algorithms, but individuals are programs (represented as trees).

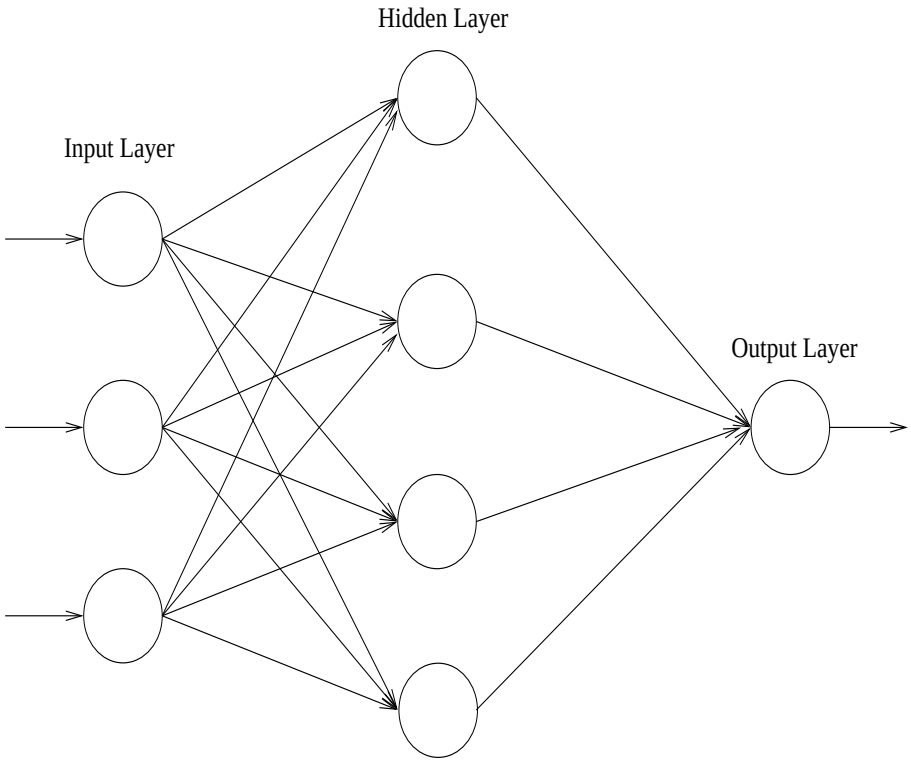


Figure 1.4: Illustration of an artificial neural network

- **Evolutionary programming** which is derived from the simulation of adaptive behavior in evolution (*phenotypic* evolution).
- **Evolution strategies** which are geared toward modeling the strategic parameters that control variation in evolution, i.e. the evolution of evolution.
- **Differential evolution**, which is similar to genetic algorithms, differing in the reproduction mechanism used.
- **Cultural evolution** which models the evolution of culture of a population and how the culture influences the genetic and phenotypic evolution of individuals.
- **Co-evolution** where initially "dumb" individuals evolve through cooperation, or in competition with one another, acquiring the necessary characteristics to survive.

Other aspects of natural evolution have also been modeled. For example, the extinction of dinosaurs, and distributed (island) genetic algorithms, where different populations are maintained with genetic evolution taking place in each population. In addition, aspects such as migration among populations are modeled. The modeling of parasitic behavior has also contributed to improved evolutionary techniques.

In this case parasites infect individuals. Those individuals that are too weak die. On the other hand, immunology has been used to study the evolution of viruses and how antibodies should evolve to kill virus infections.

Evolutionary computing has been used successfully in real-world applications, for example, data mining, combinatorial optimization, fault diagnosis, classification, clustering, scheduling and time series approximation.

1.1.3 Swarm Intelligence

Swarm intelligence originated from the study of colonies, or swarms of social organisms. Studies of the social behavior of organisms (individuals) in swarms prompted the design of very efficient optimization and clustering algorithms. For example, simulation studies of the graceful, but unpredictable, choreography of bird flocks led to the design of the particle swarm optimization algorithm, and studies of the foraging behavior of ants resulted in ant colony optimization algorithms.

Particle swarm optimization (PSO) is a global optimization approach, modeled on the social behavior of bird flocks. PSO is a population-based search procedure where the individuals, referred to as particles, are grouped into a swarm. Each particle in the swarm represents a candidate solution to the optimization problem. In a PSO system, each particle is "flown" through the multidimensional search space, adjusting its position in search space according to its own experience and that of neighboring particles. A particle therefore makes use of the best position encountered by itself and the best position of its neighbors to position itself toward an optimum solution. The effect is that particles "fly" toward the global minimum, while still searching a wide area around the best solution. The performance of each particle (i.e. the "closeness" of a particle to the global minimum) is measured according to a predefined fitness function which is related to the problem being solved. Applications of PSO include function approximation, clustering, optimization of mechanical structures, and solving systems of equations.

Studies of ant colonies have contributed in abundance to the set of intelligent algorithms. The modeling of pheromone depositing by ants in their search for the shortest paths to food sources resulted in the development of shortest path optimization algorithms. Other applications of ant colony optimization include routing optimization in telecommunications networks, graph coloring, scheduling and solving the quadratic assignment problem. Studies of the nest building of ants and bees resulted in the development of clustering and structural optimization algorithms.

As it is a very young field in Computer Science, with much potential, not many applications to real-world problems exist. However, initial applications were shown to be promising, and much more can be expected.

1.1.4 Fuzzy Systems

Traditional set theory requires elements to be either part of a set or not. Similarly, binary-valued logic requires the values of parameters to be either 0 or 1, with similar constraints on the outcome of an inferencing process. Human reasoning is, however, almost always not this exact. Our observations and reasoning usually include a measure of uncertainty. For example, humans are capable of understanding the sentence: "Some Computer Science students can program in most languages". But how can a computer represent and reason with this fact?

Fuzzy sets and fuzzy logic allow what is referred to as *approximate reasoning*. With fuzzy sets, an element belongs to a set to a certain degree of certainty. Fuzzy logic allows reasoning with these uncertain facts to infer new facts, with a degree of certainty associated with each fact. In a sense, fuzzy sets and logic allow the modeling of common sense.

The uncertainty in fuzzy systems is referred to as *nonstatistical uncertainty*, and should not be confused with *statistical uncertainty*. Statistical uncertainty is based on the laws of probability, whereas nonstatistical uncertainty is based on vagueness, imprecision and/or ambiguity. Statistical uncertainty is resolved through observations. For example, when a coin is tossed we are certain what the outcome is, while before tossing the coin, we know that the probability of each outcome is 50%. Nonstatistical uncertainty, or fuzziness, is an inherent property of a system and cannot be altered or resolved by observations.

Fuzzy systems have been applied successfully to control systems, gear transmission and braking systems in vehicles, controlling lifts, home appliances, controlling traffic signals, and many others.

1.2 Short History

Aristotle (384–322 bc) was possibly the first to move toward the concept of artificial intelligence. His aim was to explain and codify styles of deductive reasoning, which he referred to as *syllogisms*. Ramon Llull (1235–1316) developed the *Ars Magna*: an optimistic attempt to build a machine, consisting of a set of wheels, which was supposed to be able to answer all questions. Today this is still just a dream – or rather, an illusion. The mathematician Gottfried Leibniz (1646–1716) reasoned about the existence of a *calculus philosophicus*, a universal algebra that can be used to represent all knowledge (including moral truths) in a deductive system.

The first major contribution was by George Boole in 1854, with his development of the foundations of propositional logic. In 1879, Gottlieb Frege developed the foundations of predicate calculus. Both propositional and predicate calculus formed

part of the first AI tools.

It was only in the 1950s that the first definition of artificial intelligence was established by Alan Turing. Turing studied how machinery could be used to mimic processes of the human brain. His studies resulted in one of the first publications of AI, entitled *Intelligent Machinery*. In addition to his interest in intelligent machines, he had an interest in how and why organisms developed particular shapes. In 1952 he published a paper, entitled *The Chemical Basis of Morphogenesis* – possibly the first studies in what is now known as *artificial life*.

The term *artificial intelligence* was first coined in 1956 at the Dartmouth conference, organized by John MacCarthy – now regarded as the father of AI. From 1956 to 1969 much research was done in modeling biological neurons. Most notable were the work on perceptrons by Rosenblatt, and the *adaline* by Widrow and Hoff. In 1969, Minsky and Papert caused a major setback to artificial neural network research. With their book, called *Perceptrons*, they concluded that, in their "intuitive judgment", the extension of simple perceptrons to multilayer perceptrons "is sterile". This caused research in NNs to go into hibernation until the mid-1980s. During this period of hibernation a few researchers, most notably Grossberg, Carpenter, Amari, Kohonen and Fukushima, continued their research efforts.

The resurrection of NN research came with landmark publications from Hopfield, Hinton, and Rumelhart and McLelland in the early and mid-1980s. From the late 1980s research in NNs started to explode, and is today one of the largest research areas in Computer Science.

The development of evolutionary computing (EC) started with genetic algorithms in the 1950s with the work of Fraser. However, it is John Holland who is generally viewed as the father of EC, most specifically of genetic algorithms. In these works, elements of Darwin's theory of evolution [Darwin 1859] were modeled algorithmically. In the 1960s, Rechenberg developed evolutionary strategies (ES). Research in EC was not a stormy path as was the case for NNs. Other important contributions which shaped the field were by De Jong, Schaffer, Goldberg, Fogel and Koza.

Many people believe that the history of fuzzy logic started with Gautama Buddha (563 bc) and Buddhism, which often described things in shades of gray. However, the Western community considers the work of Aristotle on two-valued logic as the birth of fuzzy logic. In 1920 Lukasiewicz published the first deviation from two-valued logic in his work on three-valued logic – later expanded to an arbitrary number of values. The quantum philosopher Max Black was the first to introduce quasi-fuzzy sets, wherein degrees of membership to sets were assigned to elements. It was Lotfi Zadeh who contributed most to the field of fuzzy logic, being the developer of fuzzy sets [Zadeh 1965]. From then, until the 1980s fuzzy systems was an active field, producing names such as Mamdani, Sugeno, Takagi and Bezdek. Then, fuzzy systems also experienced a dark age in the 1980s, but was revived by Japanese researchers

in the late 1980s. Today it is a very active field with many successful applications, especially in control systems. In 1991, Pawlak introduced rough set theory to Computer Science, where the fundamental concept is the finding of a lower and upper approximation to input space. All elements within the lower approximation have full membership, while the boundary elements (those elements between the upper and lower approximation) belong to the set to a certain degree.

Interestingly enough, it was an unacknowledged South African poet, Eugene N Marais (1871-1936), who produced some of the first and most significant contributions to swarm intelligence in his studies of the social behavior of both apes and ants. Two books on his findings were published more than 30 years after his death, namely *The Soul of the White Ant* [Marais 1970] and *The Soul of the Ape* [Marais 1969]. The algorithmic modeling of swarms only gained momentum in the early 1990s with the work of Marco Dorigo on the modeling of ant colonies. In 1996 Eberhart and Kennedy developed the particle swarm optimization algorithm as a model of bird flocks. Swarm intelligence is in its infancy, and is a promising field resulting in interesting applications.

1.3 Assignments

1. Comment on the eligibility of Turing's test for computer intelligence, and his belief that computers with 10^9 bits of storage would pass a 5-minute version of his test with 70% probability.

2. Comment on the eligibility of the definition of Artificial Intelligence as given by the 1996 IEEE Neural Networks Council.

3. Based on the definition of CI given in this chapter, show that each of the paradigms (NN, EC, SI and FS) does satisfy the definition.

Part II

ARTIFICIAL NEURAL NETWORKS

Artificial neural networks (NN) were inspired from brain modeling studies. Chapter 1 illustrated the relationship between biological and artificial neural networks. But why invest so much effort in modeling biological neural networks? Implementations in a number of application fields have presented ample rewards in terms of efficiency and ability to solve complex problems. Some of the classes of applications to which artificial NNs have been applied include:

- *classification*, where the aim is to predict the class of an input vector;
- *pattern matching*, where the aim is to produce a pattern best associated with a given input vector;
- *pattern completion*, where the aim is to complete the missing parts of a given input vector;
- *optimization*, where the aim is to find the optimal values of parameters in an optimization problem;
- *control*, where, given an input vector, an appropriate action is suggested;
- *function approximation/times series modeling*, where the aim is to learn the functional relationships between input and desired output vectors;
- *data mining*, with the aim of discovering hidden patterns from data – also referred to as knowledge discovery.

A neural network is basically a realization of a nonlinear mapping from $\mathbb{R}^I$ to $\mathbb{R}^K$

$$\mathcal{F}_{NN} : \mathbb{R}^I \rightarrow \mathbb{R}^K$$

where I and K are respectively the dimension of the input and target (desired output) space. The function $\mathcal{F}_{NN}$ is usually a complex function of a set of nonlinear functions, one for each neuron in the network.

Neurons form the basic building blocks of NNs. Chapter 2 discusses the single neuron, also referred to as the *perceptron*, in detail. Chapter 3 discusses NNs under the supervised learning regime, while Chapter 4 covers unsupervised learning NNs. A hybrid supervised and unsupervised learning paradigm is discussed in Chapter 5. Reinforcement learning is covered in Chapter 6. Part II is concluded by Chapter 7 which discusses NN performance issues.

Chapter 2

The Artificial Neuron

An artificial neuron (AN), or neuron, implements a nonlinear mapping from $\mathbb{R}^I$ to $[0, 1]$ or $[-1, 1]$, depending on the activation function used. That is,

$$f_{AN} : \mathbb{R}^I \rightarrow [0, 1]$$

or

$$f_{AN} : \mathbb{R}^I \rightarrow [-1, 1]$$

where I is the number of input signals to the AN. Figure 2.1 presents an illustration of an AN with notational conventions that will be used throughout this text. An AN receives a vector of I input signals,

$$\vec{X} = (x_1, x_2, \cdots, x_I)$$

either from the environment or from other ANs. To each input signal x_i is associated a weight w_i to strengthen or deplete the input signal. The AN computes the net input signal, and uses an activation function f_{AN} to compute the output signal y given the net input. The strength of the output signal is further influenced by a threshold value θ, also referred to as the *bias*.

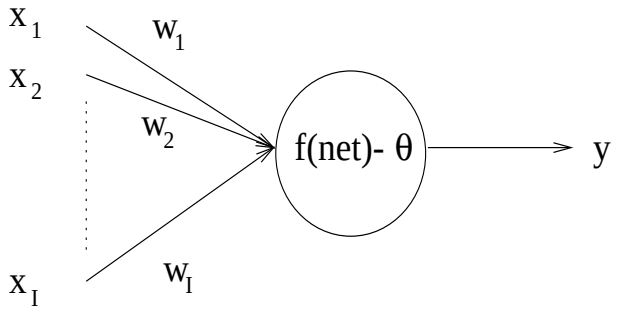


Figure 2.1: An artificial neuron

2.1 Calculating the Net Input Signal

The net input signal to an AN is usually computed as the weighted sum of all input signals,

$$net = \sum_{i=1}^{I} x_i w_i \tag{2.1}$$

Artificial neurons that compute the net input signal as the weighted sum of input signals are referred to as *summation units* (SU). An alternative to compute the net input signal is to use *product units* (PU), where

$$net = \prod_{i=1}^{I} x_i^{w_i} \tag{2.2}$$

Product units allow higher-order combinations of inputs, having the advantage of increased information capacity.

2.2 Activation Functions

The function f_{AN} receives the net input signal and bias, and determines the output (or firing strength) of the neuron. This function is referred to as the *activation function*. Different types of activation functions can be used. In general, activation functions are monotonically increasing mappings, where (excluding the linear function)

$$f_{AN}(-\infty) = 0 \quad \text{or} \quad f_{AN}(-\infty) = -1$$

and

$$f_{AN}(\infty) = 1$$

Frequently used activation functions are enumerated below:

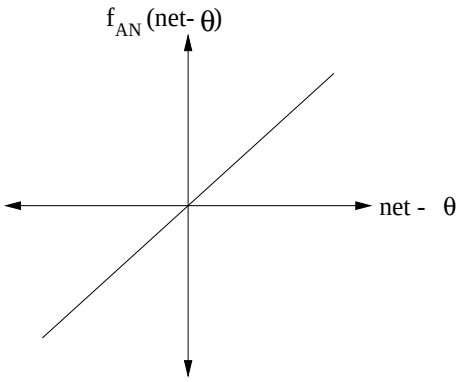
1. **Linear function** (see Figure 2.2(a) for $\theta = 0$):

$$f_{AN}(net - \theta) = \beta(net - \theta) \tag{2.3}$$

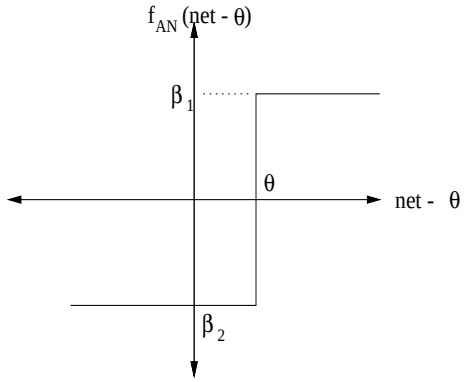
   The linear function produces a linearly modulated output, where β is a constant.

2. **Step function** (see Figure 2.2(b) for $\theta > 0$):

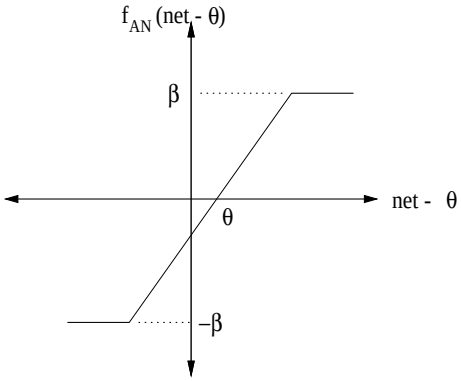
$$f_{AN}(net - \theta) = \begin{cases} \beta_1 & \text{if } net \geq \theta \\ \beta_2 & \text{if } net < \theta \end{cases} \tag{2.4}$$



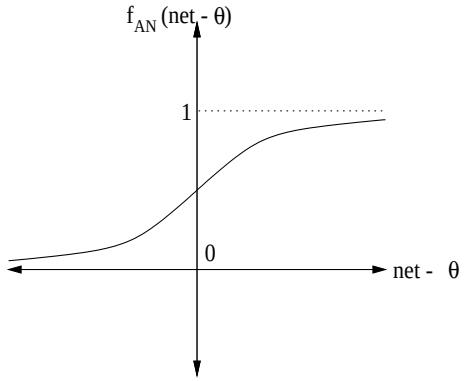
(a) Linear function



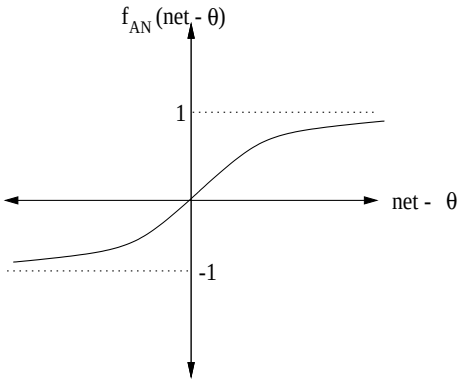
(b) Step function



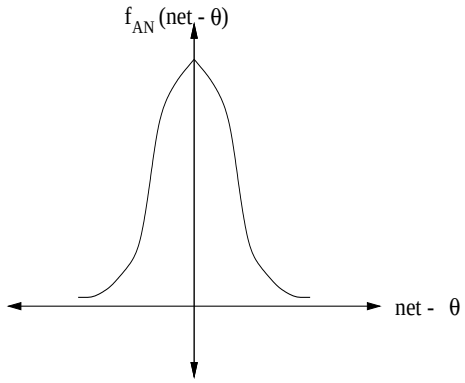
(c) Ramp function



(d) Sigmoid function



(e) Hyperbolic tangent function



(f) Gaussian function

Figure 2.2: Activation functions

The step function produces one of two scalar output values, depending on the value of the threshold θ. Usually, a binary output is produced for which $\beta_1 = 1$ and $\beta_2 = 0$; a bipolar output is also sometimes used where $\beta_1 = 1$ and $\beta_2 = -1$.

3. **Ramp function** (see Figure 2.2(c) for $\theta > 0$):

$$f_{AN}(net - \theta) = \begin{cases} \beta & \text{if } net - \theta \geq \beta \\ net - \theta & \text{if } |net - \theta| < \beta \\ -\beta & \text{if } net - \theta \leq -\beta \end{cases} \tag{2.5}$$

The ramp function is a combination of the linear and step functions.

4. **Sigmoid function** (see Figure 2.2(d) for $\theta = 0$):

$$f_{AN}(net - \theta) = \frac{1}{1 + e^{-\lambda(net-\theta)}} \tag{2.6}$$

The sigmoid function is a continuous version of the ramp function, with $f_{AN}(net) \in (0, 1)$. The parameter λ controls the steepness of the function. Usually, $\lambda = 1$.

5. **Hyperbolic tangent** (see Figure 2.2(e) for $\theta = 0$):

$$f_{AN}(net - \theta) = \frac{e^{\lambda(net-\theta)} - e^{-\lambda(net-\theta)}}{e^{\lambda(net-\theta)} + e^{-\lambda(net-\theta)}} \tag{2.7}$$

or also defined as

$$f_{AN}(net\theta) = \frac{2}{1 + e^{-\lambda(net-\theta)}} - 1 \tag{2.8}$$

The output of the hyperbolic tangent is in the range $(-1, 1)$.

6. **Gaussian function** (see Figure 2.2(f) for $\theta = 0$):

$$f_{AN}(net - \theta) = e^{-(net-\theta)^2/\sigma^2} \tag{2.9}$$

where $net - \theta$ is the mean and σ^2 the variance of the Gaussian distribution.

2.3 Artificial Neuron Geometry

Single neurons can be used to realize linearly separable functions without any error. Linear separability means that the neuron can separate the space of n-dimensional input vectors yielding an above-threshold response from those having a below-threshold response by an n-dimensional hyperplane. The hyperplane forms the boundary between the input vectors associated with the two output values. Figure 2.3 illustrates the decision boundary for a neuron with the step activation function. The hyperplane separates the input vectors for which $\sum_i x_i w_i - \theta > 0$ from the input vectors for which $\sum_i x_i w_i - \theta < 0$.

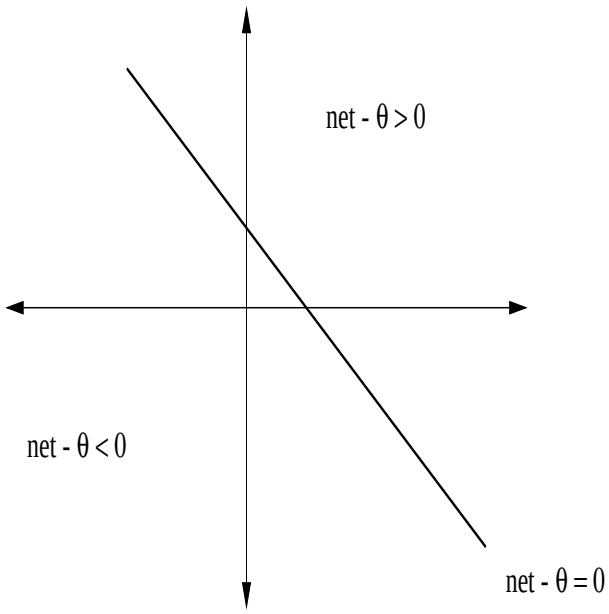


Figure 2.3: Artificial neuron boundary illustration

Thus, given the input signals and θ, the weight values w_i, can easily be calculated.

To be able to learn functions that are not linearly separable, a layered NN of several neurons is required.

2.4 Artificial Neuron Learning

The question that now remains to be answered is, how do we get the values of the weights w_i and the threshold θ? For simple neurons implementing, for example, Boolean functions, it is easy to calculate these values. But suppose that we have no prior knowledge about the function – except for data – how do we get the w_i and θ values? Through learning. The AN learns the best values for the w_i and θ from the given data. Learning consists of adjusting weight and threshold values until a certain criterion (or several criteria) is (are) satisfied.

There are three main types of learning:

- **Supervised learning**, where the neuron (or NN) is provided with a data set consisting of input vectors and a target (desired output) associated with each input vector. This data set is referred to as the training set. The aim of supervised training is then to adjust the weight values such that the error between the real output, $y = f(net - \theta)$, of the neuron and the target output, t, is minimized.

- **Unsupervised learning**, where the aim is to discover patterns or features in the input data with no assistance from an external source. Unsupervised learning basically performs a clustering of the training patterns.
- **Reinforcement learning**, where the aim is to reward the neuron (or parts of a NN) for good performance, and to penalize the neuron for bad performance.

Several learning rules have been developed for the different learning types. Before continuing with these learning rules, we simplify our AN model by introducing augmented vectors.

2.4.1 Augmented Vectors

An artificial neuron is characterized by its weight vector $\vec{W}$, threshold θ and activation function. During learning, both the weights and the threshold are adapted. To simplify learning equations, we augment the input vector to include an additional input unit, x_{I+1}, referred to as the *bias unit*. The value of x_{I+1} is always -1, and the weight w_{I+1} serves as the value of the threshold. The net input signal to the AN (assuming SUs) is then calculated as

$$\begin{aligned} net &= \sum_{i=1}^{I} x_i w_i + x_{I+1} w_{I+1} \\ &= \sum_{i=1}^{I+1} x_i w_i \end{aligned} \tag{2.10}$$

where $\theta = x_{I+1} w_{I+1} = -w_{I+1}$.

In the case of the step function, an input vector yields an output of 1 when $\sum_{i=1}^{I+1} x_i w_i \geq 0$, and 0 when $\sum_{i=1}^{I+1} x_i w_i < 0$.

The rest of this chapter considers training of single neurons.

2.4.2 Gradient Descent Learning Rule

While gradient descent (GD) is not the first training rule for ANs, it is possibly the approach that is used most to train neurons (and NNs for that matter). GD requires the definition of an error (or objective) function to measure the neuron's error in approximating the target. The sum of squared errors

$$\mathcal{E} = \sum_{p=1}^{P} (t_p - f_p)^2 \tag{2.11}$$

is usually used, where t_p and f_p are respectively the target and actual output for pattern p, and P is the total number of input-target vector pairs (*patterns*) in the training set.

The aim of GD is to find the weight values that minimize $\mathcal{E}$. This is achieved by calculating the gradient of $\mathcal{E}$ in weight space, and to move the weight vector along the negative gradient (as illustrated for a single weight in Figure 2.4).

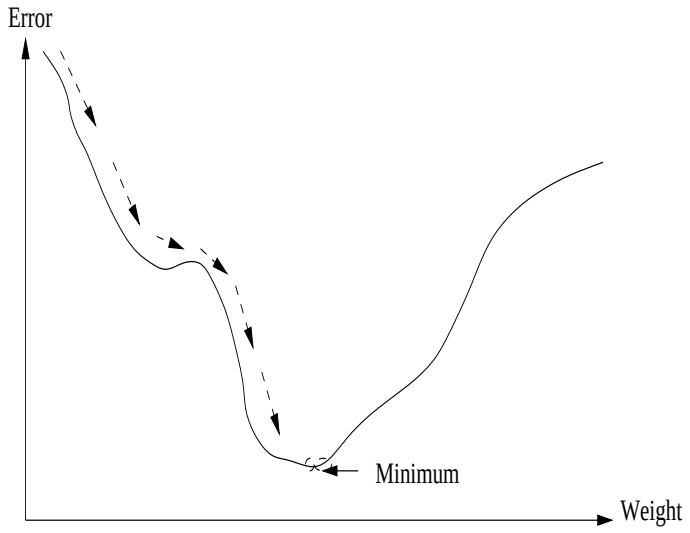


Figure 2.4: GD illustrated

Given a single training pattern, weights are updated using

$$w_i(t) = w_i(t-1) + \Delta w_i(t) \tag{2.12}$$

with

$$\Delta w_i(t) = \eta(-\frac{\partial \mathcal{E}}{\partial w_i}) \tag{2.13}$$

where

$$\frac{\partial \mathcal{E}}{\partial w_i} = -2(t_p - f_p)\frac{\partial f}{\partial net_p}x_{i,p} \tag{2.14}$$

and η is the learning rate (size of the steps taken in the negative direction of the gradient). The calculation of the partial derivative of f with respect to net_p (the net input for pattern p) presents a problem for all discontinuous activation functions, such as the step and ramp functions; $x_{i,p}$ is the i-th input signal corresponding to pattern p. The Widrow-Hoff learning rule presents a solution for the step and ramp functions, while the generalized delta learning rule assumes continuous functions which are at least once differentiable.

2.4.3 Widrow-Hoff Learning Rule

For the Widrow-Hoff learning rule [Widrow 1987], assume that $f = net_p$. Then $\frac{\partial f}{\partial net_p} = 1$, giving

$$\frac{\partial \mathcal{E}}{\partial w_i} = -2(t_p - f_p)x_{i,p} \tag{2.15}$$

Weights are then updated using

$$w_i(t) = w_i(t-1) + 2\eta(t_p - f_p)x_{i,p} \tag{2.16}$$

The Widrow-Hoff learning rule, also referred to as the least-means-square (LMS) algorithm, was one of the first algorithms used to train layered neural networks with multiple adaptive linear neurons. This network was commonly referred to as the Madaline [Widrow 1987, Widrow and Lehr 1990].

2.4.4 Generalized Delta Learning Rule

The generalized delta learning rule is a generalization of the Widrow-Hoff learning rule which assumes differentiable activation functions. Assume that the sigmoid function (from equation (2.6)) is used. Then,

$$\frac{\partial f}{\partial net_p} = f_p(1 - f_p) \tag{2.17}$$

giving

$$\frac{\partial \mathcal{E}}{\partial w_i} = -2(t_p - f_p)f_p(1 - f_p)x_{i,p} \tag{2.18}$$

2.4.5 Error-Correction Learning Rule

For the error-correction learning rule it is assumed that binary-valued activation functions are used, for example, the step function. Weights are only adjusted when the neuron responds in error. That is, only when $(t_p - f_p) = 1$ or $(t_p - f_p) = -1$, are weights adjusted using equation (2.16).

2.5 Conclusion

At this point we can conclude the discussion on single neurons. While this is not a complete treatment of all aspects of single ANs, it introduced those concepts required for the rest of the chapters. In the next chapter we explain learning rules for networks of neurons, expanding on the different types of learning rules available.

2.6 Assignments

1. Explain why the threshold θ is necessary. What is the effect of θ, and what will the consequences be of not having a threshold?

2. Explain what the effects of weight changes are on the separating hyperplane.

3. Which of the following Boolean functions can be realized with a single neuron which implements a SU? Justify your answer.

   (a) $x_1 x_2 \overline{x}_3$

   (b) $x_1 \overline{x}_2 + \overline{x}_1 x_2$

   (c) $x_1 + x_2$

   where $x_1 x_2$ denotes $x_1 \;\; AND \;\; x_2$; $x_1 + x_2$ denotes $x_1 \;\; OR \;\; x_2$; $\overline{x}_1$ denotes $NOT \;\; x_1$.

4. Is it possible to use a single PU to learn problems which are not linearly separable?

5. Why is the error per pattern squared?

6. Can the function $|t_p - o_p|$ be used instead of $(t_p - o_p)^2$?

7. Is the following statement true or false: *A single neuron can be used to approximate the function* $f(z) = z^2$? Justify your answer.

8. What are the advantages of using the hyperbolic tangent activation function instead of the sigmoid activation function?

Chapter 3

Supervised Learning Neural Networks

Single neurons have limitations in the type of functions they can learn. A single neuron (implementing a SU) can be used to realize linearly separable functions only. As soon as functions that are not linearly separable need to be learned, a layered network of neurons is required. Training these layered networks is more complex than training a single neuron, and training can be supervised, unsupervised or through reinforcement. This chapter deals with supervised training.

Supervised learning requires a training set which consists of input vectors and a target vector associated with each input vector. The NN learner uses the target vector to determine how well it has learned, and to guide adjustments to weight values to reduce its overall error. This chapter considers different NN types that learn under supervision. These network types include standard multilayer NNs, functional link NNs, simple recurrent NNs, time-delay NNs and product unit NNs. We first describe these different architectures in Section 3.1. Different learning rules for supervised training are then discussed in Section 3.2. The chapter ends with a discussion on ensemble NNs in Section 3.4.

3.1 Neural Network Types

Various multilayer NN types have been developed. Feedforward NNs such as the standard multilayer NN, functional link NN and product unit NN receive external signals and simply propagate these signals through all the layers to obtain the result (output) of the NN. There are no feedback connections to previous layers. Recurrent NNs, on the other hand, have such feedback connections to model the temporal characteristics of the problem being learned. Time-delay NNs, on the other hand,

memorize a window of previously observed patterns.

3.1.1 Feedforward Neural Networks

Figure 3.1 illustrates a standard feedforward neural network (FFNN), consisting of three layers: an input layer, a hidden layer and an output layer. While this figure illustrates only one hidden layer, a FFNN can have more than one hidden layer. However, it has been proved that FFNNs with monotonically increasing differentiable functions can approximate any continuous function with one hidden layer, provided that the hidden layer has enough hidden neurons [Hornik 1989]. A FFNN can also have direct (linear) connections between the input layer and the output layer.

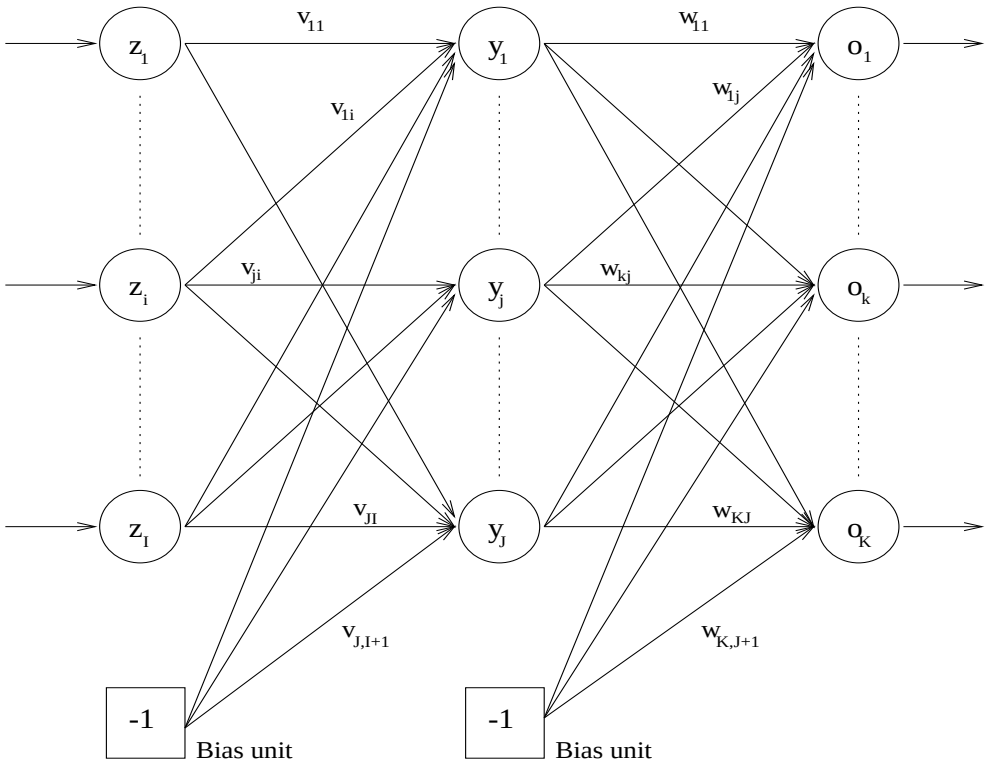


Figure 3.1: Feedforward neural network

The output of a FFNN for any given input pattern p is calculated with a single forward pass through the network. For each output unit o_k, we have (assuming no direct connections between the input and output layers),

$$o_{k,p} = f_{o_k}(net_{o_{k,p}})$$

$$= f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(net_{y_{j,p}}))$$
$$= f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{i=1}^{I+1} v_{ji} z_{i,p}))$$

where f_{o_k} and f_{y_j} are respectively the activation function for output unit o_k and hidden unit y_j; w_{kj} is the weight between output unit o_k and hidden unit y_j; $z_{i,p}$ is the value of input unit z_i for input pattern p; the $(I+1)$-th input unit and the $(J+1)$-th hidden unit are bias units representing the threshold values of neurons in the next layer.

Note that each activation function can be a different function. It is not necessary that all activation functions be the same. Also, each input unit can implement an activation function. It is usually assumed that inputs units have linear activation functions.

3.1.2 Functional Link Neural Networks

In functional link neural networks (FLNN) input units do implement activation functions. A FLNN is simply a FFNN with the input layer expanded into a layer of functional higher-order units [Ghosh and Shin 1992, Hussain *et al.* 1997]. The input layer, with dimension I, is therefore expanded to functional units $h_1, h_2, \cdots, h_L$, where L is the total number of functional units, and each functional unit h_l is a function of the input parameter vector $(z_1, \cdots, z_I)$, i.e. $h_l(z_1, \cdots, z_I)$ (see Figure 3.2). The weight matrix U between the input layer and the layer of functional units is defined as

$$u_{li} = \begin{cases} 1 & \text{if functional unit } h_l \text{ is dependent of } z_i \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

For FLNNs, v_{jl} is the weight between hidden unit y_j and functional link h_l.

Calculation of the activation of each output unit o_k occurs in the same manner as for FFNNs, except that the additional layer of functional units is taken into account:

$$o_{k,p} = f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{l=1}^{L} v_{jl} h_l(\vec{z}_p))) \tag{3.2}$$

The use of higher-order combinations of input units may result in faster training times and improved accuracy (see, for example, [Ghosh and Shin 1992, Hussain *et al.* 1997]).

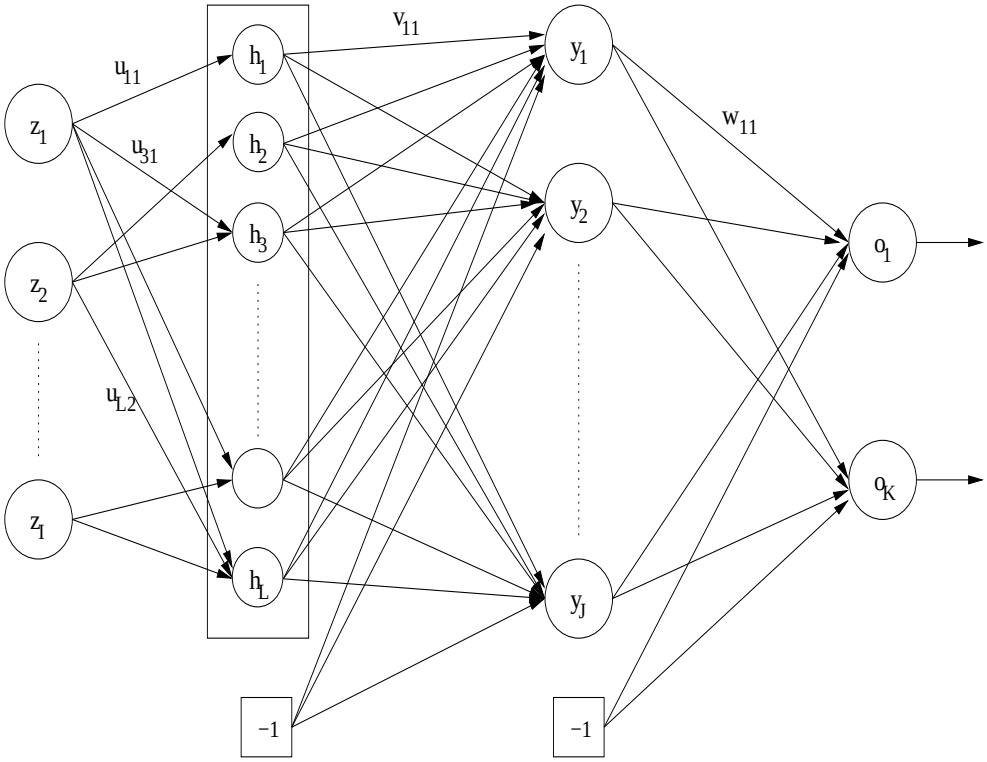


Figure 3.2: Functional link neural network

3.1.3 Product Unit Neural Networks

Product unit neural networks (PUNN) have neurons which compute the weighted product of input signals, instead of a weighted sum [Durbin and Rumelhart 1989, Janson and Frenzel 1993, Leerink *et al.* 1995]. For product units, the net input is computed as given in equation (2.2).

Different PUNNs have been suggested. In one type each input unit is connected to SUs, and to a dedicated group of PUs. Another PUNN type has alternating layers of product and summation units. Due to the mathematical complexity of having PUs in more than one hidden layer, this section only illustrates the case for which just the hidden layer has PUs, and no SUs. The output layer has only SUs, and linear activation functions are assumed for all neurons in the network. Then, for each hidden unit y_j, the net input to that hidden unit is (note that no bias is included)

$$\begin{aligned} net_{y_{j,p}} &= \prod_{i=1}^{I} z_{i,p}^{v_{ji}} \\ &= \prod_{i=1}^{I} e^{v_{ji} \ln(z_{i,p})} \end{aligned}$$

$$= \quad e^{\sum_i v_{ji} \ln(z_{i,p})} \tag{3.3}$$

where $z_{i,p}$ is the activation value of input unit z_i, and v_{ji} is the weight between input z_i and hidden unit y_j.

An alternative to the above formulation of the net input signal for PUs is to include a "distortion" factor within the product, such as

$$net_{y_{j,p}} = \prod_{i=1}^{I+1} z_{i,p}^{v_{ji}} \tag{3.4}$$

where $z_{I+1,p} = -1$ for all patterns; $v_{j,I+1}$ represents the distortion factor. The purpose of the distortion factor is to dynamically shape the activation function during training to more closely fit the shape of the true function represented by the training data.

If $z_{i,p} < 0$, then $z_{i,p}$ can be written as the complex number $z_{i,p} = \imath^2 |z_{i,p}|$ ($\imath = \sqrt{-1}$) which, substituted in (3.3), yields

$$net_{y_{j,p}} = e^{\sum_i v_{ji} \ln |z_{i,p}|} e^{\sum_i v_{ji} \ln \imath^2} \tag{3.5}$$

Let $c = 0 + \imath = a + b\imath$ be a complex number representing $\imath$. Then,

$$\ln c = \ln re^{\imath\theta} = \ln r + \imath\theta + 2\pi k\imath \tag{3.6}$$

where $r = \sqrt{a^2 + b^2} = 1$.

Considering only the main argument, $arg(c)$, $k = 0$ which implies that $2\pi k\imath = 0$. Furthermore, $\theta = \frac{\pi}{2}$ for $\imath = (0, 1)$. Therefore, $\imath\theta = \imath\frac{\pi}{2}$, which simplifies equation (3.9) to $\ln c = \imath\frac{\pi}{2}$, and consequently,

$$\ln \imath^2 = \imath\pi \tag{3.7}$$

Substitution of (3.7) in (3.5) gives

$$\begin{aligned} net_{y_{j,p}} &= e^{\sum_i v_{ji} \ln |z_{i,p}|} e^{\sum_i v_{ji} \pi \imath} \\ &= e^{\sum_i v_{ji} \ln |z_{i,p}|} [\cos(\sum_{i=1}^{I} v_{ji}\pi) + \imath \sin(\sum_{i=1}^{I} v_{ji}\pi)] \end{aligned} \tag{3.8}$$

Leaving out the imaginary part ([Durbin and Rumelhart 1989] show that the added complexity of including the imaginary part does not help with increasing performance),

$$net_{y_{j,p}} = e^{\sum_i v_{ji} \ln |z_{i,p}|} \cos(\pi \sum_{i=1}^{I} v_{ji}) \tag{3.9}$$

Now, let

$$\rho_{j,p} = \sum_{i=1}^{I} v_{ji} \ln |z_{i,p}| \tag{3.10}$$

$$\phi_{j,p} = \sum_{i=1}^{I} v_{ji} \mathcal{I}_i \tag{3.11}$$

with

$$\mathcal{I}_i = \begin{cases} 0 & \text{if } z_{i,p} > 0 \\ 1 & \text{if } z_{i,p} < 0 \end{cases} \tag{3.12}$$

and $z_{i,p} \neq 0$.

Then,

$$net_{y_{j,p}} = e^{\rho_{j,p}} \cos(\pi \phi_{j,p}) \tag{3.13}$$

The output value for each output unit is then calculated as

$$o_{k,p} = f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(e^{\rho_{j,p}} \cos(\pi \phi_{j,p})) \tag{3.14}$$

Note that a bias is now included for each output unit.

3.1.4 Simple Recurrent Neural Networks

Simple recurrent neural networks (SRNN) have feedback connections which add the ability to also learn the temporal characteristics of the data set. Several different types of SRNNs have been developed, of which the Elman and Jordan SRNNs are simple extensions of FFNNs.

The Elman SRNN, as illustrated in Figure 3.3, makes a copy of the hidden layer, which is referred to as the *context layer*. The purpose of the context layer is to store the previous state of the hidden layer, i.e. the state of the hidden layer at the previous pattern presentation. The context layer serves as an extension of the input layer, feeding signals representing previous network states, to the hidden layer. The input vector is therefore

$$\vec{z} = (\underbrace{z_1, \cdots, z_{I+1}}_{actual\ \ inputs}, \underbrace{z_{I+2}, \cdots, z_{I+1+J}}_{context\ \ units}) \tag{3.15}$$

Context units $z_{I+2}, \cdots, Z_{I+1+J}$ are fully interconnected with all hidden units. The connections from each hidden unit y_j (for $j = 1, \cdots, J$) to its corresponding context unit z_{I+1+j} have a weight of 1. Hence, the activation value y_j is simply copied to z_{I+1+j}. It is, however, possible to have weights not equal to 1, in which case the

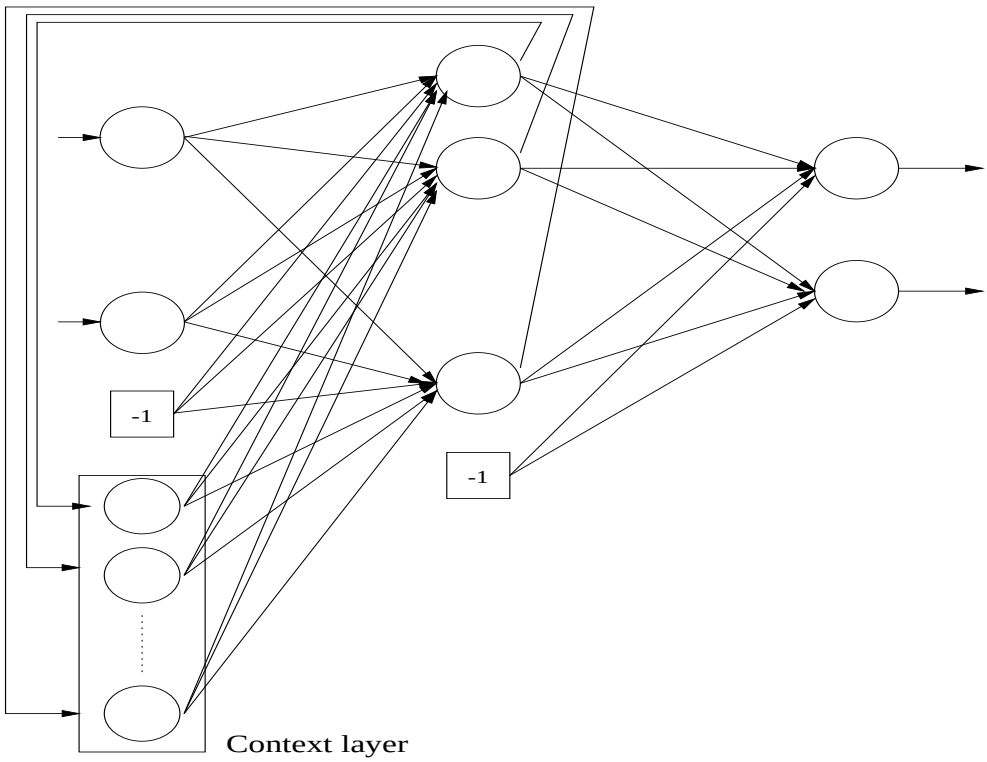


Figure 3.3: Elman simple recurrent neural network

influence of previous states is weighted. Determining such weights adds additional complexity to the training step.

Each output unit's activation is then calculated as

$$o_{k,p} = f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{i=1}^{I+1+J} v_{ji} z_{i,p})) \tag{3.16}$$

where $(z_{I+2,p}, \cdots, z_{I+1+J,p}) = (y_{1,p}(t-1), \cdots, y_{J,p}(t-1))$.

Jordan SRNNs, on the other hand, make a copy of the output layer instead of the hidden layer. The copy of the output layer, referred to as the *state layer*, extends the input layer to

$$\vec{z} = (\underbrace{z_1, \cdots, z_{I+1}}_{actual\ \ inputs}, \underbrace{z_{I+2}, \cdots, z_{I+1+K}}_{state\ \ units}) \tag{3.17}$$

The previous state of the output layer then also serves as input to the network. For each output unit we have

$$o_{k,p} = f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{i=1}^{I+1+K} v_{ji} z_{i,p})) \tag{3.18}$$

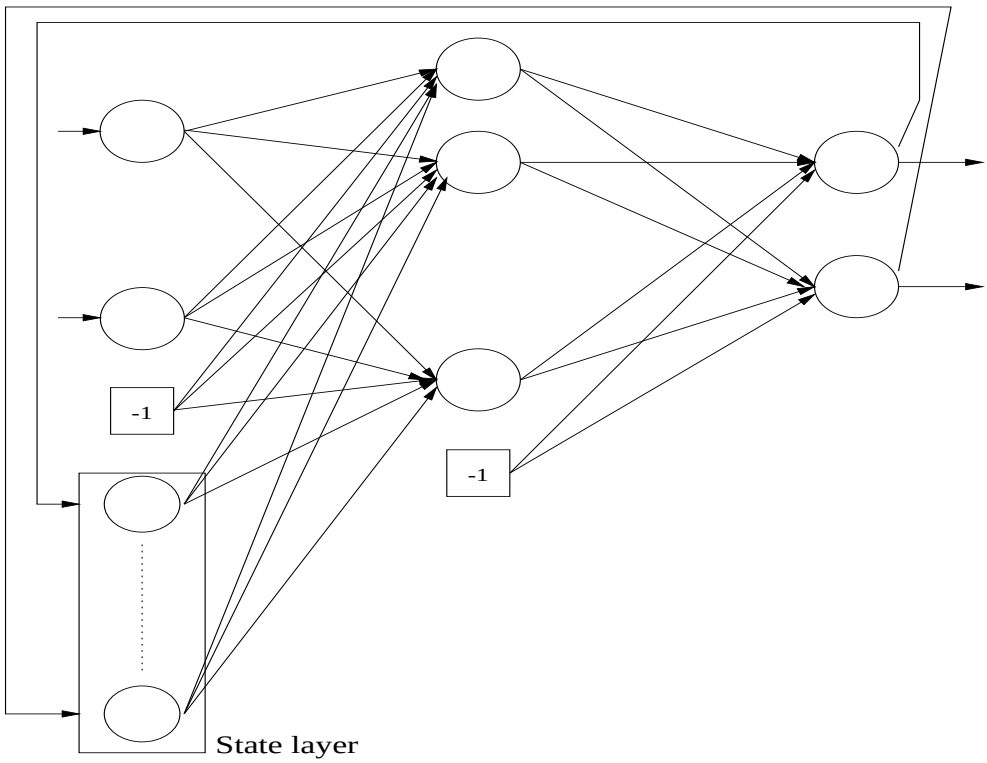


Figure 3.4: Jordan simple recurrent neural network

where $(z_{I+2,p}, \cdots, z_{I+1+K,p}) = (o_{1,p}(t-1), \cdots, o_{K,p}(t-1))$.

3.1.5 Time-Delay Neural Networks

A time-delay neural network (TDNN), also referred to as backpropagation-through-time, is a temporal network with its input patterns successively delayed in time. A single neuron with T time delays for each input unit is illustrated in Figure 3.5. This type of neuron is then used as a building block to construct a complete feedforward TDNN.

Initially, only $z_{i,p}(t)$, with $t = 0$, has a value and $z_{i,p}(t - t^{'})$ is zero for all $i = 1, \cdots, I$ with time steps $t^{'} = 1, \cdots, T$; T is the total number of time steps, or number of delayed patterns. Immediately after the first pattern is presented, and before presentation of the second pattern,

$$z_{i,p}(t-1) = z_{i,p}(t)$$

After presentation of $t^{'}$ patterns and before the presentation of pattern $t^{'} + 1$, for all $t = 1, \cdots, t^{'}$,

$$z_{i,p}(t - t^{'}) = z_{i,p}(t - t^{'} + 1)$$

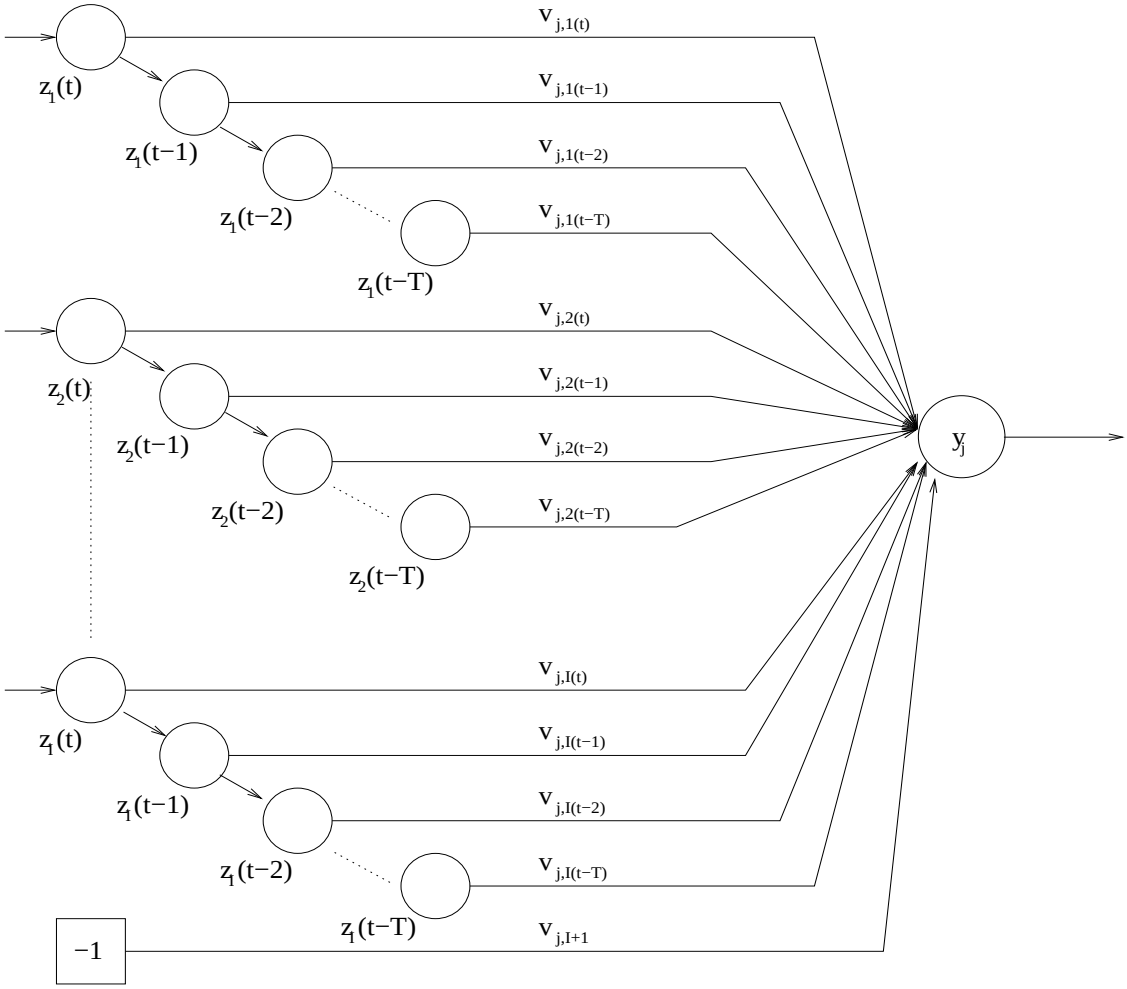


Figure 3.5: A single time-delay neuron

This causes a total of T patterns to influence the updates of weight values, thus allowing the temporal characteristics to drive the shaping of the learned function. The connections between $z_{i,p}(t - t^{'})$ and $z_{i,p}(t - t^{'} + 1)$ has a value of 1.

The output of a TDNN is calculated as

$$o_{k,p} = f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{i=1}^{I}\sum_{t=0}^{T} v_{j,i(t)} z_{i,p}(t) + z_{I+1} v_{j,I+1}))$$

3.2 Supervised Learning Rules

Up to this point we have seen how NNs can be used to calculate an output value given an input pattern. This section explains approaches to train the NN such that the output of the network is an accurate approximation of the target values. First, the learning problem is explained, and then solutions are presented.

3.2.1 The Learning Problem

Consider a finite set of input-target pairs $D = \{d_p = (\vec{z}_p, \vec{t}_p)|p = 1, \cdots, P\}$ sampled from a stationary density $\Omega(D)$, with $z_{i,p}, t_{k,p} \in \mathbb{R}$ for $i = 1, \cdots, I$ and $k = 1, \cdots, K$; $z_{i,p}$ is the value of input unit z_i and $t_{k,p}$ is the target value of output unit o_k for pattern p. According to the signal-plus-noise model,

$$\vec{t}_p = \mu(\vec{z}_p) + \vec{\zeta}_p \tag{3.19}$$

where $\mu(\vec{z})$ is the unknown function. The input values $z_{i,p}$ are sampled with probability density $\omega(\vec{z})$, and the $\vec{\zeta}_{k,p}$ are independent, identically distributed noise sampled with density $\phi(\vec{\zeta})$, having zero mean. The objective of learning is then to approximate the unknown function $\mu(\vec{z})$ using the information contained in the finite data set D. For NN learning this is achieved by dividing the set D randomly into a training set D_T, validation set D_V and a test set D_G. The approximation to $\mu(\vec{z})$ is found from the training set D_T, memorization is determined from D_V (more about this later), and the generalization accuracy is estimated from the test set D_G (more about this later).

Since prior knowledge about $\Omega(D)$ is usually not known, a nonparametric regression approach is used by the NN learner to search through its hypothesis space $\mathcal{H}$ for a function $\mathcal{F}_{NN}(D_T; W)$ which gives a good estimation of the unknown function $\mu(\vec{z})$, where $\mathcal{F}_{NN}(D_T; W) \in \mathcal{H}$. For multilayer NNs, the hypothesis space consists of all functions realizable from the given network architecture as described by the weight vector W.

During learning, the function $\mathcal{F}_{NN} : \mathbb{R}^I \longrightarrow \mathbb{R}^K$ is found which minimizes the empirical error

$$\mathcal{E}_T(D_T; W) = \frac{1}{P_T} \sum_{p=1}^{P_T} (\mathcal{F}_{NN}(\vec{z}_p, W) - -\vec{t}_p)^2 \tag{3.20}$$

where P_T is the total number of training patterns. The hope is that a small empirical (training) error will also give a small true error, or generalization error, defined as

$$\mathcal{E}_G(\Omega; W) = \int (\mathcal{F}_{NN}(\vec{z}, W) - \vec{t})^2 d\Omega(\vec{z}, \vec{t}) \tag{3.21}$$

For the purpose of NN learning, the empirical error in equation (3.20) is referred to as the objective function to be optimized by the optimization method. Several optimization algorithms for training NNs have been developed [Battiti 1992, Becker and Le Cun 1988, Duch and Korczak 1998]. These algorithms are grouped into two classes:

- **Local optimization**, where the algorithm may get stuck in a local optimum without finding a global optimum. Gradient descent and scaled conjugate gradient are examples of local optimizers.
- **Global optimization**, where the algorithm searches for the global optimum by employing mechanisms to search larger parts of the search space. Global optimizers include LeapFrog, simulated annealing, evolutionary computing and swarm optimization.

Local and global optimization techniques can be combined to form hybrid training algorithms.

Learning consists of adjusting weights until an acceptable empirical error has been reached. Two types of supervised learning algorithms exist, based on when weights are updated:

- **Stochastic/online learning**, where weights are adjusted after each pattern presentation. In this case the next input pattern is selected randomly from the training set, to prevent any bias that may occur due to the order in which patterns occur in the training set.
- **Batch/off-line learning**, where weight changes are accumulated and used to adjust weights only after all training patterns have been presented.

3.2.2 Gradient Descent Optimization

Gradient descent (GD) optimization has led to one of the most popular learning algorithms, namely backpropagation, developed by Werbos [Werbos 1974]. Learning iterations (one learning iteration is referred to as an *epoch*) consists of two phases:

1. **Feedforward pass**, which simply calculates the output value(s) of the NN (as discussed in Section 3.1).
2. **Backward propagation**, which propagates an error signal back from the output layer toward the input layer. Weights are adjusted as functions of the backpropagated error signal.

Feedforward Neural Networks

Assume that the sum squared error (SSE) is used as the objective function. Then, for each pattern, p,

$$\mathcal{E}_p = \frac{1}{2}\frac{\sum_{k=1}^{K}(t_{k,p} - o_{k,p})^2}{K} \tag{3.22}$$

where K is the number of output units, and $t_{k,p}$ and $o_{k,p}$ are respectively the target and actual output values of the k-th output unit.

The rest of the derivations refer to an individual pattern. The pattern subscript, p, is therefore omitted for notational convenience. Also assume sigmoid activation functions in the hidden and output layers with augmented vectors. Then,

$$o_k = f_{o_k}(net_{o_k}) = \frac{1}{1 + e^{-net_{o_k}}} \tag{3.23}$$

and

$$y_j = f_{y_j}(net_{y_j}) = \frac{1}{1 + e^{-net_{y_j}}} \tag{3.24}$$

Weights are updated, in the case of stochastic learning, according to the following equations:

$$w_{kj}(t) \quad += \quad \Delta w_{kj}(t) + \alpha \Delta w_{kj}(t-1) \tag{3.25}$$

$$v_{ji}(t) \quad += \quad \Delta v_{ji}(t) + \alpha \Delta v_{ji}(t-1) \tag{3.26}$$

where α is the momentum (discussed later).

In the rest of this section the equations for calculating $\Delta w_{kj}(t)$ and $\Delta v_{ji}(t)$ are derived. The reference to time, t, is omitted for notational convenience.

From (3.23),

$$\frac{\partial o_k}{\partial net_{o_k}} = \frac{\partial f_{o_k}}{\partial net_{o_k}} = (1 - o_k)o_k = f'_{o_k} \tag{3.27}$$

and

$$\frac{\partial net_{o_k}}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}}(\sum_{j=1}^{J+1} w_{kj} y_j) = y_j \tag{3.28}$$

where f'_{o_k} is the derivative of the corresponding activation function. From (3.27), (3.28),

$$\begin{aligned} \frac{\partial o_k}{\partial w_{kj}} &= \frac{\partial o_k}{\partial net_{o_k}} \frac{\partial net_{o_k}}{\partial w_{kj}} \\ &= (1 - o_k)o_k y_j \\ &= f'_{o_k} y_j \end{aligned} \tag{3.29}$$

From (3.22),

$$\frac{\partial E}{\partial o_k} = \frac{\partial}{\partial o_k}(\frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2) = -(t_k - o_k) \tag{3.30}$$

Define the output error that needs to be back-propagated as $\delta_{o_k} = \frac{\partial E}{\partial net_{o_k}}$. Then, from (3.30) and (3.27),

$$\begin{aligned} \delta_{o_k} &= \frac{\partial E}{\partial net_{o_k}} \\ &= \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial net_{o_k}} \\ &= -(t_k - o_k)(1 - o_k)o_k = -(t_k - o_k)f'_{o_k} \end{aligned} \tag{3.31}$$

Then, the changes in the hidden-to-output weights are computed from (3.30), (3.29) and (3.31),

$$\begin{aligned} \Delta w_{kj} &= \eta(-\frac{\partial E}{\partial w_{kj}}) \\ &= -\eta\frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial w_{kj}} \\ &= -\eta\delta_{o_k}y_j \end{aligned} \tag{3.32}$$

Continuing with the input-to-hidden weights,

$$\frac{\partial y_j}{\partial net_{y_j}} = \frac{\partial f_{y_j}}{\partial net_{y_j}} = (1 - y_j)y_j = f'_{y_j} \tag{3.33}$$

and

$$\frac{\partial net_{y_j}}{\partial v_{ji}} = \frac{\partial}{\partial v_{ji}}(\sum_{i=1}^{I+1} v_{ji}z_i) = z_i \tag{3.34}$$

From (3.33) and (3.34),

$$\begin{aligned} \frac{\partial y_j}{\partial v_{ji}} &= \frac{\partial y_j}{\partial net_{y_j}}\frac{\partial net_{y_j}}{\partial v_{ji}} \\ &= (1 - y_j)y_jz_i = f'_{y_j}z_i \end{aligned} \tag{3.35}$$

and

$$\frac{\partial net_{o_k}}{\partial y_j} = \frac{\partial}{\partial y_j}(\sum_{j=1}^{J+1} w_{kj}y_j) = w_{kj} \tag{3.36}$$

From (3.31) and (3.36),

$$\begin{aligned}\frac{\partial E}{\partial y_j} &= \frac{\partial}{\partial y_j}(\frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2) \\ &= \sum_{k=1}^{K}\frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial net_{o_k}}\frac{\partial net_{o_k}}{\partial y_j} \\ &= \sum_{k=1}^{K}\frac{\partial E}{\partial net_{o_k}}\frac{\partial net_{o_k}}{\partial y_j} \\ &= \sum_{k=1}^{K}\delta_{o_k} w_{kj} \end{aligned} \tag{3.37}$$

Define the hidden layer error, which needs to be back-propagated, from (3.37) and (3.33) as,

$$\begin{aligned}\delta_{y_j} &= \frac{\partial E}{\partial net_{y_j}} \\ &= \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_{y_j}} \\ &= \sum_{k=1}^{K}\delta_{o_k} w_{kj} f'_{y_j} \end{aligned} \tag{3.38}$$

Finally, the changes to input-to-hidden weights are calculated from (3.37), (3.35) and (3.38) as

$$\begin{aligned}\Delta v_{ji} &= \eta(-\frac{\partial E}{\partial v_{ji}}) \\ &= -\eta\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_{ji}} \\ &= -\eta\delta_{y_j} z_i \end{aligned} \tag{3.39}$$

If direct weights from the input to the output layer are included, the following additional weight updates are needed:

$$\begin{aligned}\Delta u_{ki} &= \eta(-\frac{\partial E}{\partial u_{ki}}) \\ &= -\eta\frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial u_{ki}} \\ &= -\eta\delta_{o_k} z_i \end{aligned} \tag{3.40}$$

where u_{ki} is a weight from the i-th input unit to the k-th output unit.

In the case of batch learning, weights are updated as given in equations (3.25) and (3.26), but with

$$\Delta w_{kj}(t) = \sum_{p=1}^{P_T} \Delta w_{kj,p}(t) \tag{3.41}$$

$$\Delta v_{ji}(t) = \sum_{p=1}^{P_T} \Delta v_{ji,p}(t) \tag{3.42}$$

where $\Delta w_{kj,p}(t)$ and $\Delta v_{ji,p}(t)$ are weight changes for individual patterns p, and P_T is the total number of patterns in the training set.

Stochastic learning is summarized with the following pseudocode algorithm:

1. Initialize weights, η, α and the number of epochs $\xi = 0$
2. Let $\mathcal{E}_T = 0$
3. For each training pattern p
   (a) do the feedforward phase to calculate $y_{j,p}$ ($\forall \quad j = 1, \cdots, J$) and $o_{k,p}$ ($\forall \ k = 1, \cdots, K$)
   (b) compute output error signals $\delta_{o_{k,p}}$ and hidden layer error signals $\delta_{y_{j,p}}$; then adjust weights w_{kj} and v_{ji} (backpropagation of errors)
   (c) $\mathcal{E}_T += [\mathcal{E}_p = \sum_{k=1}^{K}(t_{k,p} - o_{k,p})^2]$
4. $\xi = \xi + 1$
5. Test stopping criteria: if no stopping criterion is satisfied, go to step 2.

Stopping criteria usually includes:

- Stop when a maximum number of epochs has been exceeded.
- Stop when the mean squared error (MSE) on the training set,

$$\mathcal{E}_T = \frac{\sum_{p=1}^{P}\sum_{k=1}^{K}(t_{k,p} - o_{k,p})^2}{PK} \tag{3.43}$$

  is small enough (other error measures such as the root mean squared error can also be used).
- Stop when overfitting is observed, i.e. when training data is being memorized. An indication of overfitting is when $\mathcal{E}_V > \overline{\mathcal{E}}_V + \sigma_{\mathcal{E}_V}$, where $\overline{\mathcal{E}}_V$ is the average validation error over the previous epochs, and $\sigma_{\mathcal{E}_V}$ is the standard deviation in validation error.

It is straightforward to apply the GD optimization to the training of FLNNs, SRNNs and TDNNs, so derivations of the weight update equations are left to the reader. GD learning for PUNNs is given in the next section.

Product Unit Neural Networks

This section derives learning equations for PUs used in the hidden layer only, assuming GD optimization and linear activation functions. Since only the equations for the input-to-hidden weights change, only the derivations of these weight update equations are given. The change Δv_{ji} in weight is

$$\begin{aligned}\Delta v_{ji} &= \frac{\partial E}{\partial v_{ji}} \\ &= \frac{\partial E}{\partial net_{y_{j,p}}} \frac{net_{y_{j,p}}}{\partial v_{ji}} \\ &= \delta_{y_{j,p}} \frac{\partial net_{y_{j,p}}}{\partial v_{ji}} \end{aligned} \tag{3.44}$$

where $\delta_{y_{j,p}}$ is the error signal, computed in the same way as for SUs, and

$$\begin{aligned}\frac{net_{y_{j,p}}}{\partial v_{ji}} &= \frac{\partial}{\partial v_{ji}} (\prod_{i=1}^{I} z_{i,p}^{v_{ji}}) \\ &= \frac{\partial}{\partial v_{ji}} (e^{\rho_{j,p}} \cos(\pi \phi_{j,p})) \\ &= e^{\rho_{j,p}} [\ln |z_{i,p}| \cos(\pi \phi_{j,p}) - \mathcal{I}_i \pi \sin(\pi \phi_{j,p})] \end{aligned} \tag{3.45}$$

A major advantage of product units is an increased information capacity compared to summation units [Durbin and Rumelhart 1989, Leerink *et al.* 1995]. Durbin and Rumelhart showed that the information capacity of a single PU (as measured by its capacity for learning random Boolean patterns) is approximately $3N$, compared to $2N$ for a single SU (N is the number of inputs to the unit) [Durbin and Rumelhart 1989]. The larger capacity means that functions approximated using PUs will require less processing elements than required if SUs were used. This point can be illustrated further by considering the minimum number of processing units required for learning the simple polynomial functions in Table 3.1. The minimal number of SUs were determined using a sensitivity analysis variance analysis pruning algorithm [Engelbrecht *et al.* 1999, Engelbrecht 2001], while the minimal number of PUs is simply the number of different powers in the expression (provided a polynomial expression).

While PUNNs provide an advantage in having smaller network architectures, a major drawback of PUs is an increased number of local minima, deep ravines and valleys. The search space for PUs is usually extremely convoluted. Gradient descent, which

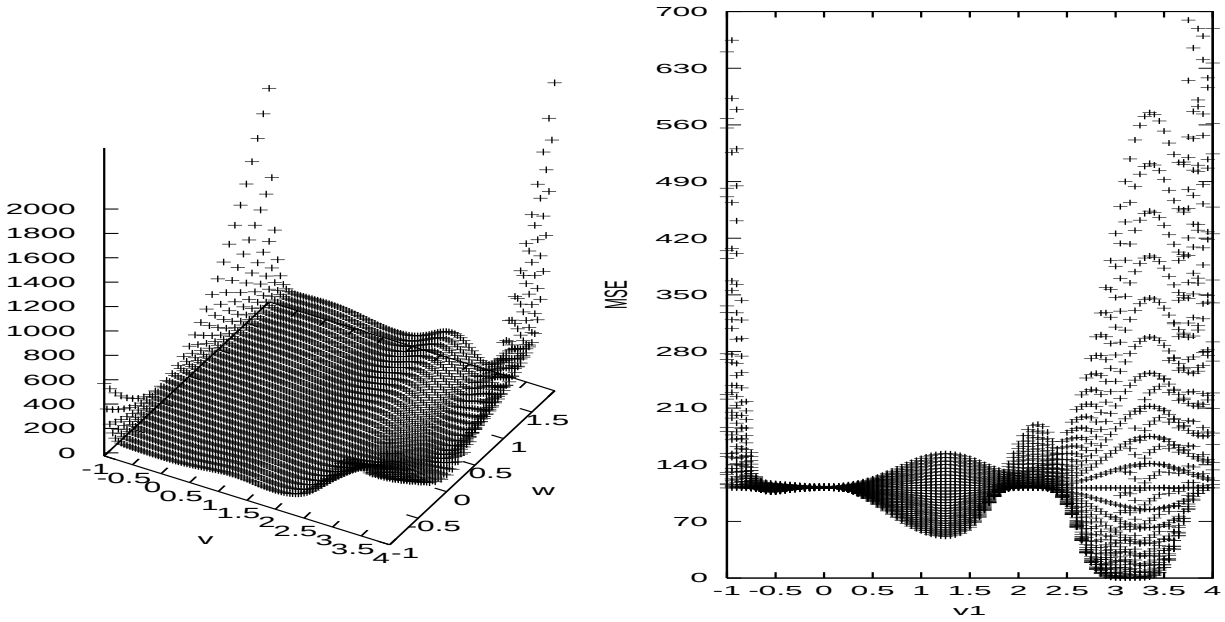
Function	SUs	PUs
$f(z) = z^2$	2	1
$f(z) = z^6$	3	1
$f(z) = z^2 + z^5$	3	2
$f(z_1, z_2) = z_1^3 z_2^7 - 0.5 z_1^6$	8	3

Table 3.1: SUs and PUs needed for simple functions

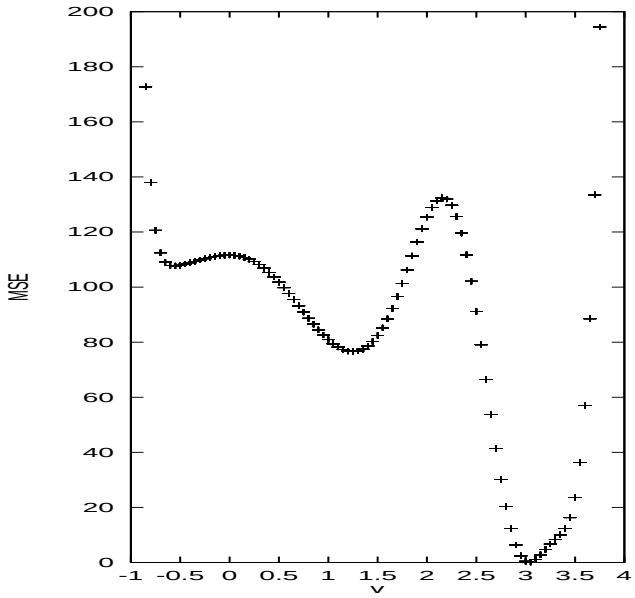
works best when the search space is relatively smooth, therefore frequently gets trapped in local minima or becomes paralyzed (which occurs when the gradient of the error with respect to the current weight is close to zero). Leerink *et al.* illustrated that the 6-bit parity problem could not be trained using GD and PUs [Leerink *et al.* 1995]. Two reasons were identified to explain why GD failed: (1) weight initialization and (2) the presence of local minima. The initial weights of a network are usually computed as small random numbers. Leerink *et al.* argued that this is the worst possible choice of initial weights, and suggested that larger initial weights be used instead. But, large weights lead to large weight updates due to the exponential term in the weight update equation (see equation (3.45)), which consequently cause the network to overshoot the minimum. Experience has shown that GD only manages to train PUNNs when the weights are initialized in close proximity of the optimal weight values – the optimal weight values are, however, usually not available.

As an example to illustrate the complexity of the search space for PUs, consider the approximation of the function $f(z) = z^3$, with $z \in [-1, 1]$. Only one PU is needed, resulting in a 1-1-1 NN architecture (that is, one input, one hidden and one output unit). In this case the optimal weight values are $v = 3$ (the input-to-hidden weight) and $w = 1$ (the hidden-to-output weight). Figures 3.6(a)-(b) present the search space for $v \in [-1, 4]$ and $w \in [-1, 1.5]$. The error is computed as the mean squared error over 500 randomly generated patterns. Figure 3.6(b) clearly illustrates 3 minima, with the global minimum at $v = 3, w = 1$. These minima are better illustrated in Figure 3.6(c) where w is kept constant at its optimum value of 1. Initial small random weights will cause the network to be trapped in one of the local minima (having very large MSE). Large initial weights may also be a bad choice. Assume an initial weight $v \geq 4$. The derivative of the error with respect to v is extremely large due to the steep gradient of the error surface. Consequently, a large weight update will be made which may cause jumping over the global minimum. The neural network either becomes trapped in a local minimum, or oscillates between the extreme points of the error surface.

A global stochastic optimization algorithm is needed to allow searching of larger parts of the search space. The optimization algorithm should also



(a) (b)



(c)

Figure 3.6: Illustration of PU search space for $f(z) = z^3$

not rely heavily on the calculation of gradient information. Simulated annealing [Leerink *et al.* 1995], genetic algorithms [Engelbrecht and Ismail 1999, Janson and Frenzel 1993], the *gbest* version of particle swarm optimization [Engelbrecht and Ismail 1999, Van den Bergh and Engelbrecht 2001] and LeapFrog [Engelbrecht and Ismail 1999] have been used successfully to train PUNNs.

3.2.3 Scaled Conjugate Gradient

Conjugate gradient optimization trades off the simplicity of GD and the fast quadratic convergence of Newton's method. Several conjugate gradient learning algorithms have been developed (look at the survey in [Battiti 1992]), most of which are based on the assumption that the error function of all weights in the region of the solution can be accurately approximated by

$$\mathcal{E}_T(D_T;\vec{W}) = \frac{1}{2}\vec{W}^T H \vec{W} - \theta^T \vec{W}$$

where H is the Hessian matrix. Since the dimension of the Hessian matrix is the total number of weights in the network, the calculation of conjugate directions on the error surface becomes computationally infeasible. Computationally feasible conjugate gradient algorithms compute conjugate gradient directions without explicitly computing the Hessian matrix, and perform weight updates along these directions. Scaled conjugate gradient is one such algorithm [Møller 1993].

SCG also automatically determines the step size. It is a batch learning algorithm, that operates on a single weight vector which reflects all the weights and biases in the network. The standard SCG algorithm is summarized below:

1. Initialize the weight vector $\vec{w}_1$ and the scalars $\sigma > 0, \lambda_1 > 0$ and $\overline{\lambda} = 0$, where the subscript denotes the epoch number. Let $\vec{p}_1 = \vec{r}_1 = -\mathcal{E}'(\vec{w}_1)$, $k = 1$ and $success = true$.

2. If $success = true$ then calculate the second-order information:

$$\sigma_k = \frac{\sigma}{||\vec{p}_k||} \tag{3.46}$$

$$\vec{s}_k = \frac{\mathcal{E}'(\vec{w}_k + \sigma_k \vec{p}_k) - \mathcal{E}'(\vec{w}_k)}{\sigma_k} \tag{3.47}$$

$$\delta_k = \vec{p}_k^T \vec{s}_k \tag{3.48}$$

   where $\vec{p}_k^T$ is the transpose of vector $\vec{p}_k$, and $||\vec{p}_k||$ is the Euclidean norm.

3. Scale $\vec{s}_k$:

$$\vec{s}_k \mathrel{+}= (\lambda_k - \overline{\lambda}_k)\vec{p}_k \tag{3.49}$$

$$\delta_k \mathrel{+}= (\lambda_k - \overline{\lambda}_k)||\vec{p}_k||^2 \tag{3.50}$$

4. If $\delta_k \leq 0$ then make the Hessian matrix positive definite:

$$\vec{s}_k = \vec{s}_k + (\lambda_k - 2\frac{\delta_k}{||\vec{p}_k||^2})\vec{p}_k \tag{3.51}$$

$$\overline{\lambda}_k = 2(\lambda_k - 2\frac{\delta_k}{||\vec{p}_k||^2}) \tag{3.52}$$

$$\delta_k = -\delta_k + \lambda_k||\vec{p}_k||^2 \tag{3.53}$$

$$\lambda_k = \overline{\lambda}_k \tag{3.54}$$

5. Calculate the step size:

$$\mu_k = \vec{p}_k^T\vec{r}_k \tag{3.55}$$

$$\alpha_k = \frac{\mu_k}{\delta_k} \tag{3.56}$$

6. Calculate the comparison parameter:

$$\Delta_k = \frac{2\delta_k[\mathcal{E}(\vec{w}_k) - \mathcal{E}(\vec{w}_k + \alpha_k\vec{p}_k)]}{\mu_k^2} \tag{3.57}$$

7. If $\Delta_k \geq 0$ then a successful reduction in error can be made. Adjust the weights:

$$\vec{w}_{k+1} = \vec{w}_k + \alpha_k\vec{p}_k \tag{3.58}$$

$$\vec{r}_{k+1} = -\mathcal{E}^{'}(\vec{w}_{k+1}) \tag{3.59}$$

$$\overline{\lambda}_k = 0, \quad success = true \tag{3.60}$$

   (a) If $k \; mod \; N = 0$, where N is the total number of weights, then restart the algorithm, with $\vec{p}_{k+1} = \vec{r}_{k+1}$ and go to step 2, else create a new conjugate direction:

$$\beta_k = \frac{||\vec{r}_{k+1}||^2 - \vec{r}_{k+1}^T\vec{r}_k}{\mu_k} \tag{3.61}$$

$$\vec{p}_{k+1} = \vec{r}_{k+1} + \beta_k\vec{p}_k \tag{3.62}$$

   (b) If $\Delta_k \geq 0.75$ then reduce the scale parameter with $\lambda_k = \frac{1}{2}\lambda_k$.

   else a reduction in error is not possible; let $\overline{\lambda}_k = \lambda_k$ and $success = false$

8. If $\Delta_k < 0.25$ then increase the scale parameter to $\lambda_k = 4\lambda_k$.

9. If the steepest descent direction $\vec{r}_k \neq 0$ then set $k = k + 1$ (i.e. go to the next epoch) and go to step 2, else terminate and return $\vec{w}_{k+1}$ as the desired minimum.

The algorithm restarts each N consecutive epochs for which no reduction in error could be achieved, at which point the algorithm finds a new direction to search. The function to calculate the derivative $\mathcal{E}'(\vec{w}) = \frac{\partial \mathcal{E}}{\partial \vec{w}}$ computes the derivative of $\mathcal{E}$ with respect to each weight for each of the patterns. The derivatives over all the patterns are then summed, i.e.

$$\frac{\partial \mathcal{E}}{\partial w_i} = \sum_{p=1}^{P} \frac{\partial \mathcal{E}}{\partial w_{i,p}} \tag{3.63}$$

where w_i is a single weight.

3.2.4 LeapFrog Optimization

LeapFrog is an optimization approach based on the physical problem of the motion of a particle of unit mass in an n-dimensional conservative force field [Snyman 1982, Snyman 1983]. The potential energy of the particle in the force field is represented by the function to be minimized – in the case of NNs, the potential energy is the MSE. The objective is to conserve the total energy of the particle within the force field, where the total energy consists of the particle's potential and kinetic energy. The optimization method simulates the motion of the particle, and by monitoring the kinetic energy, an interfering strategy is adapted to appropriately reduce the potential energy. The reader is referred to [Snyman 1982, Snyman 1983] for more information on this approach. The algorithm is summarized below:

1. Compute an initial weight vector $\vec{w}_0$, with random components. Let $\Delta t = 0.5, \delta = 1, m = 3, \delta_1 = 0.001$ and $\epsilon = 10^{-5}$. Initialize $i = 0, j = 2, s = 0, p = 1$ and $k = -1$.

2. Compute the initial acceleration $\vec{a}_0 = -\nabla\mathcal{E}(\vec{w}_0)$ and velocity $\vec{v}_0 = \frac{1}{2}\vec{a}_0\Delta t$, where $\mathcal{E}(\vec{w}_0)$ is the MSE for weight vector $\vec{w}_0$.

3. Set $k = k + 1$ and compute $||\Delta\vec{w}_k|| = ||\vec{v}_k||\Delta t$.

4. If $||\Delta\vec{w}_k|| < \delta$ go to step 5, otherwise set $\vec{v}_k = \delta\vec{v}_k/(\Delta t||\vec{v}_k||)$ and go to step 6.

5. Set $p = p + \delta_1$ and $\Delta t = p\Delta t$.

6. If $s < m$, go to step 7, otherwise set $\Delta t = \Delta t/2$ and $\vec{w}_k = (\vec{w}_k + \vec{w}_{k-1})/2$, $\vec{v}_k = (\vec{v}_k + \vec{v}_{k-1})/4$, $s = 0$ and go to 7.

7. Set $\vec{w}_{k+1} = \vec{w}_k + \vec{v}_k\Delta t$.

8. Compute $\vec{a}_{k+1} = -\nabla\mathcal{E}(\vec{w}_{k+1})$ and $\vec{v}_{k+1} = \vec{v}_k + \vec{a}_{k+1}\Delta t$.

9. If $\vec{a}_{k+1}^T\vec{a}_k > 0$, then $s = 0$ and go to 10, otherwise $s = s + 1, p = 1$ and go to 10.

10. If $||\vec{a}_{k+1}|| \leq \epsilon$ then stop, otherwise go to 11.

11. If $||\vec{v}_{k+1}|| > ||\vec{v}_k||$ then $i = 0$ and go to 3, otherwise $\vec{w}_{k+2} = (\vec{w}_{k+1} + \vec{w}_k)/2$, $i = i + 1$ and go to 12.

12. Perform a restart: If $i \leq j$, then $\vec{v}_{k+1} = (\vec{v}_{k+1} + \vec{v}_k)/4$ and $k = k + 1$, go to 8, otherwise $\vec{v}_{k+1} = 0, j = 1, k = k + 1$ and go to 8.

3.2.5 Particle Swarm Optimization

Particle swarm optimization (PSO) is a global optimization approach, modeled on the social behavior of flocks of birds and schools of fish [Eberhart *et al.* 1996, Kennedy and Eberhart 1995]. PSO is a population-based search procedure where the individuals, referred to as particles, are grouped into a swarm. Each particle in the swarm represents a candidate solution to the optimization problem. In a PSO system, each particle is "flown" through the multidimensional search space, adjusting its position in search space according to its own experience and that of neighboring particles. A particle therefore makes use of the best position encountered by itself and the best position of its neighbors to position itself toward the global minimum. The effect is that particles "fly" toward an optimum, while still searching a wide area around the best solution. The performance of each particle (i.e. the "closeness" of a particle to the global minimum) is measured according to a predefined fitness function which is related to the problem being solved.

For the purposes of this study, a particle represents the weight vector of a NN, including all biases. The dimension of the search space is therefore the total number of weights and biases. The fitness function is the mean squared error (MSE) over the training set, or the test set (as measure of generalization).

The PSO algorithm is summarized below to illustrate its simplicity. The interested reader is referred to [Corne *et al.* 1999, Eberhart *et al.* 1996, Van den Bergh 2002] for more information on the swarm approach to optimization. PSO will also be revisited in Part III of this book.

1. Initialize a swarm of P D-dimensional particles, where D is the number of weights and biases.

2. Evaluate the fitness f_p, where f_p is the MSE of the associated network over a given data set. given data set.

3. If $f_p < BEST_p$ then $BEST_p = f_p$ and $BESTx_p = x_p$, where $BEST_p$ is the current best fitness achieved by particle p, x_p is the current coordinates of particle p in D-dimensional weight space, and $BESTx_p$ is the coordinates corresponding to particle p's best fitness so far.

4. If $f_p < BEST_{GBEST}$ then $GBEST = p$, where $GBEST$ is the particle having the overall best fitness over all particles in the swarm.

5. Change the velocity V_p of each particle p:

$$\begin{aligned} V_p &= wV_p + c_1 r_1 (BESTx_p - x_p) + \\ &\quad c_2 r_2 (BESTx_{GBEST} - x_p) \end{aligned}$$

   where c_1 and c_2 are acceleration constants, and $r_1, r_2 \sim U(0,1)$; w is the inertia weight.

6. Fly each particle p to $x_p + V_p$.

7. Loop to step 2 until convergence.

In step 5, the coordinates $BESTx_p$ and $BESTx_{GBEST}$ are used to pull the particles toward a minimum, and the acceleration constant controls how far particles fly from one another. Convergence tests are the same as for standard training algorithms, such as GD.

In a similar way, genetic algorithms, evolutionary computing and cultural algorithms can be used to train NNs.

3.3 Functioning of Hidden Units

Section 2.3 illustrated the geometry and functioning of a single perceptron. This section illustrates the tasks of the hidden units in supervised NNs. For this purpose we consider a standard FFNN consisting of one hidden layer employing SUs. To simplify visual illustrations, we consider the case of two-dimensional input for classification and one-dimensional input for function approximation.

For classification problems, the task of hidden units is to form the decision boundaries to separate different classes. Figure 3.7 illustrates the boundaries for a three-class problem. Solid lines represent boundaries. For this artificial problem ten boundaries exist. Since each hidden unit implements one boundary, ten hidden units are required to perform the classification as illustrated in the figure. Less hidden units can be used, but at the cost of an increase in classification error. Also note that in the top left corner there are misclassifications of class $\times$, being part of the space for class $\bullet$. This problem can be solved by using three additional hidden units to form these boundaries. How do we know how many hidden units are necessary without any prior knowledge about the input space? This very important issue is dealt with in Chapter 7, where the relationship between the number of hidden units and performance is investigated.

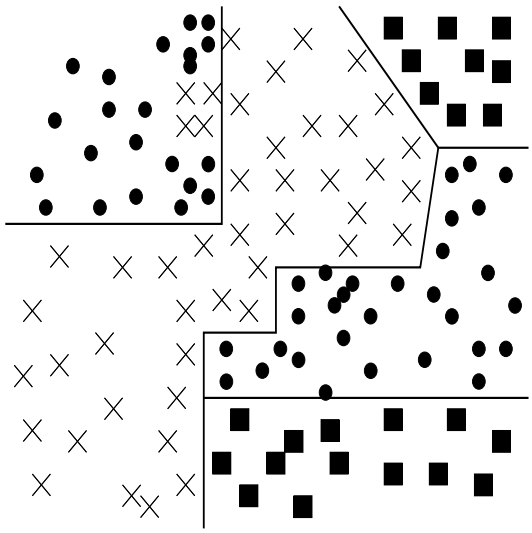


Figure 3.7: Feedforward neural network classification boundary illustration

In the case of function approximation, assuming a one-dimensional function as depicted in Figure 3.8, five hidden units with sigmoid activation functions are required to learn the function. A sigmoid function is then fitted for each inflection point of the target function. The number of hidden units is therefore the number of turning points plus one. In the case of linear activation functions, the hidden units perform the same task. However, more linear activation functions may be required to learn the function to the same accuracy as obtained using sigmoid functions.

3.4 Ensemble Neural Networks

Training of NNs starts on randomly selected initial weights. This means that each time a network is retrained on the same data set, different results can be expected, since learning starts at different points in the search space; different NNs may disagree, and make different errors. This problem in NN training prompted the development of ensemble networks, where the aim is to optimize results through the combination of a number of individual networks, trained on the same task.

In its most basic form, an ensemble network – as illustrated in Figure 3.9 – consists of a number of NNs all trained on the same data set, using the same architecture and learning algorithm. At convergence of the individual NN members, the results of the different NNs need to be combined to form one, final result. The final result of an ensemble can be calculated in several ways, of which the following are simple and efficient approaches:

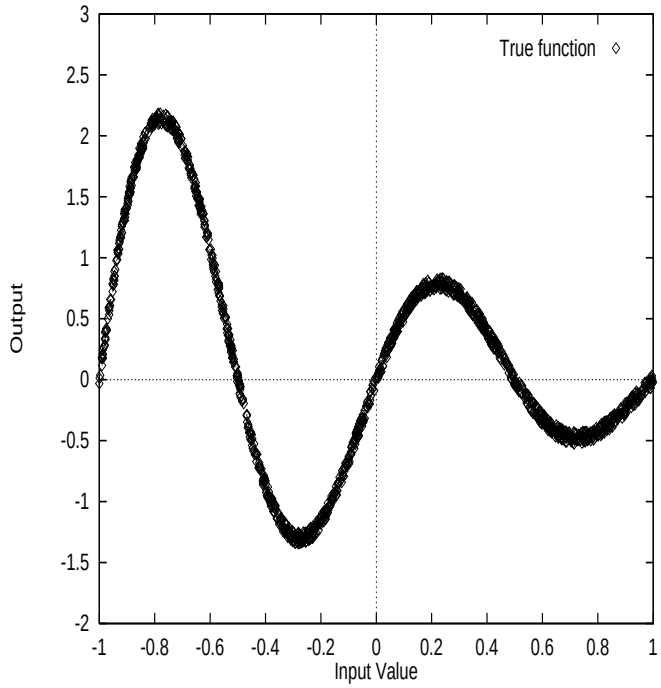


Figure 3.8: Hidden unit functioning for function approximation

- Select the NN within the ensemble that provides the best generalization performance;
- Take the average over the outputs of all the members of the ensemble;
- Form a linear combination of the outputs of each of the NNs within the ensemble. In this case a weight, C_n, is assigned to each network as an indication of the credibility of that network. The final output of the ensemble is therefore a weighted sum of the outputs of the individual networks.

The combination of inputs as discussed above is sensible only when there is disagreement among the ensemble members, or if members make their errors on different parts of the search space.

Several adaptations of the basic ensemble model are of course possible. For example, instead of having each NN train on the same data set, different data sets can be used. One such approach is bagging, which is a bootstrap ensemble method that creates individuals for its ensemble by training each member network on a random redistribution of the original training set [Breiman 1996]. If the original training set contained P_T patterns, then a data set of P_T patterns is randomly sampled from the original training set for each of the ensemble members. This means that patterns may be duplicated in the member training sets. Also, not all of the patterns in the original training set will necessarily occur in the member training sets.

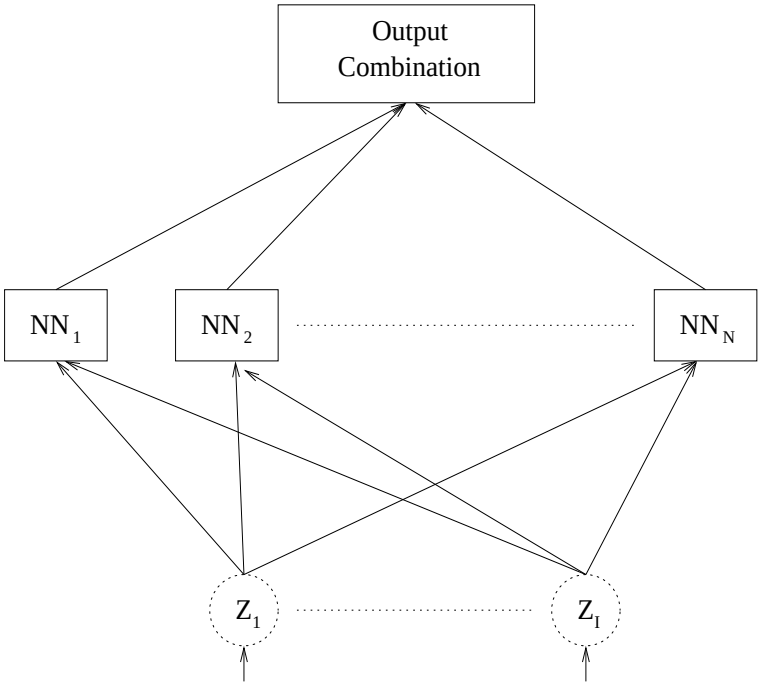


Figure 3.9: Ensemble neural network

Alternatively, the architectures of the different NNs may differ. Even different NN types can be used. It is also not necessary that each of the members be trained using the same optimization algorithm.

The above approaches to ensemble networks train individual NNs in parallel, independent of one another. Much more can be gained under a cooperative ensemble strategy, where individual NNs (referred to as agents) exchange their experience and knowledge during the training process. Research in such cooperative agents is now very active, and the reader is recommended to read more about these.

One kind of cooperative strategy for ensembles is referred to as boosting [Drucker 1999, Freund and Schapire 1999]. With boosting, members of the ensemble are not trained in parallel. They are trained sequentially, where already trained members filter patterns into easy and hard patterns. New, untrained members of the ensemble then focus more on the hard patterns as identified by previously trained networks.

3.5 Conclusion

This overview of multilayer NNs and supervised learning rules is by no means complete. It merely serves to present a flavor of what is available. The interested reader

is directed to the extensive amount of literature available is journals, conference proceedings, books and the Internet. Chapter 7 revisits supervised learning with an extensive discussion of performance aspects.

3.6 Assignments

1. Give an expression for $o_{k,p}$ for a FFNN with direct connections between the input and output layer.
2. Why is the term $(-1)^{v_{j,I+1}}$ possible in equation (3.4)?
3. Explain what is meant by the terms overfitting and underfitting. Why is $\mathcal{E}_V > \overline{\mathcal{E}}_V + \sigma_V$ a valid indication of overfitting?
4. Investigate the following aspects:
   (a) Are direct connections between the input and output layers advantageous? Give experimental results to illustrate.
   (b) Compare a FFNN and an Elman RNN trained using GD. Use the following function as benchmark: $z_t = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$, with $z_1, z_2 \sim U(-1, 1)$, sampled from a uniform distribution in the range $(-1, 1)$.
   (c) Compare stochastic learning and batch learning using GD for the function $o_t = z_t$ where $z_t = 0.3z_{t-6} - 0.6z_{t-4} + 0.5z_{t-1} + 0.3z_{t-6}^2 - 0.2z_{t-4}^2 + \zeta_t$, and $z_t \sim U(-1, 1)$ for $t = 1, \cdots, 10$, and $\zeta_t \sim N(0, 0.05)$ is zero-mean noise sampled from a normal distribution.
   (d) Compare GD and SCG on any classification problem from the UCI repository at http://www.ics.uci.edu/ ˜mlearn/MLRepository.html
   (e) Design and perform an experiment to illustrate if PSO yields better performance than GD.
5. Assume gradient descent is used as optimization algorithm, and derive the learning equations for the Elman SRNN, the Jordan SRNN, TDNN and FLNN.
6. Explain how a SRNN learns the temporal characteristics of data.
7. How can a FLNN be used to improve accuracy and training speed?
8. Explain why bias for only the output units of a PUNN, as discussed in this chapter, is sufficient.
9. Discuss the effect of having an augmented input as part of the product calculation of the net input signal.
10. Explain why the minimal number of hidden units for a PU is simply the number of powers in the function to be approximated, assuming that the function is a polynomial.

11. Design a batch version of the PSO NN training algorithm.

12. What is the main requirement for activation and error functions if gradient descent is used to train supervised neural networks?

13. What is the main advantage of using recurrent neural networks instead of feedforward neural networks?

14. What is the main advantage in using PUs instead of SUs?

Chapter 4

Unsupervised Learning Neural Networks

An important feature of NNs is their ability to learn from their environment. Chapter 3 covered NN types that learned under the guidance of a supervisor, or teacher. The supervisor presents to the NN learner an input pattern and a desired response. Supervised learning NNs then try to learn the functional mapping between the input and desired response vectors. In contrast to supervised learning, the objective of unsupervised learning is to discover patterns or features in the input data with no help from a teacher, basically performing a clustering of input space. This chapter deals with the unsupervised learning paradigm.

Section 4.1 presents a short background on unsupervised learning. Hebbian learning is presented in Section 4.2, while Section 4.3 covers principal component learning, Section 4.4 covers the learning vector quantizer version I, and Section 4.5 discusses self-organizing feature maps.

4.1 Background

Aristotle observed that human memory has the ability to connect items (e.g. objects, feelings and ideas) that are similar, contradictory, that occur in close proximity, or in succession [Kohonen 1987]. The patterns that we associate may be of the same, or different types. For example, a photo of the sea may bring associated thoughts of happiness, or smelling a specific fragrance may be associated with a certain feeling, memory or visual image. Also, the ability to reproduce the pitch corresponding to a note, irrespective of the form of the note, is an example of the pattern association behavior of the human brain.

Artificial neural networks have been developed to model the pattern association ability of the human brain. These networks are referred to as associative memory NNs. Associative memory NNs are usually two-layer NNs, where the objective is to adjust the weights such that the network can store a set of pattern associations – without any external help from a teacher. The development of these associative memory NNs is mainly inspired from studies of the visual and auditory cortex of mammalian organisms, such as the bat. These artificial NNs are based on the fact that parts of the brain are organized such that different sensory inputs are represented by topologically ordered computational maps. The networks form a topographic map of the input patterns, where the coordinates of the neurons correspond to intrinsic features of the input patterns.

An additional feature modeled with associative memory NNs is to preserve old information as new information becomes available. In contrast, supervised learning NNs have to retrain on all the information when new data becomes available; if not, supervised networks tend to focus on the new information, forgetting what the network has learned already.

Unsupervised learning NNs are functions which map an input pattern to an associated target pattern, i.e.

$$\mathcal{F}_{NN} : \mathbb{R}^I \rightarrow \mathbb{R}^K \tag{4.1}$$

as illustrated in Figure 4.1. The single weight matrix determines the mapping from the input vector $\vec{z}$ to the output vector $\vec{o}$.

4.2 Hebbian Learning Rule

The Hebbian learning rule, named after the neuropsychologist Hebb, is the oldest and simplest learning rule. With Hebbian learning, weight values are adjusted based on the correlation of neuron activation values. The motivation of this approach is from Hebb's hypothesis that the ability of a neuron to fire is based on that neuron's ability to cause other neurons connected to it to fire. In such cases the weight between the two correlated neurons is strengthened (or increased). Using the notation from Figure 4.1, the change in weight at time step t is given as

$$\Delta u_{ki}(t) = \eta o_{k,p} z_{i,p} \tag{4.2}$$

Weights are then updated using

$$u_{ki}(t) = u_{ki}(t-1) + \Delta u_{ki}(t) \tag{4.3}$$

where η is the learning rate.

From equation (4.2), the adjustment of weight values is larger for those input-output pairs for which the input value has a greater effect on the output values.

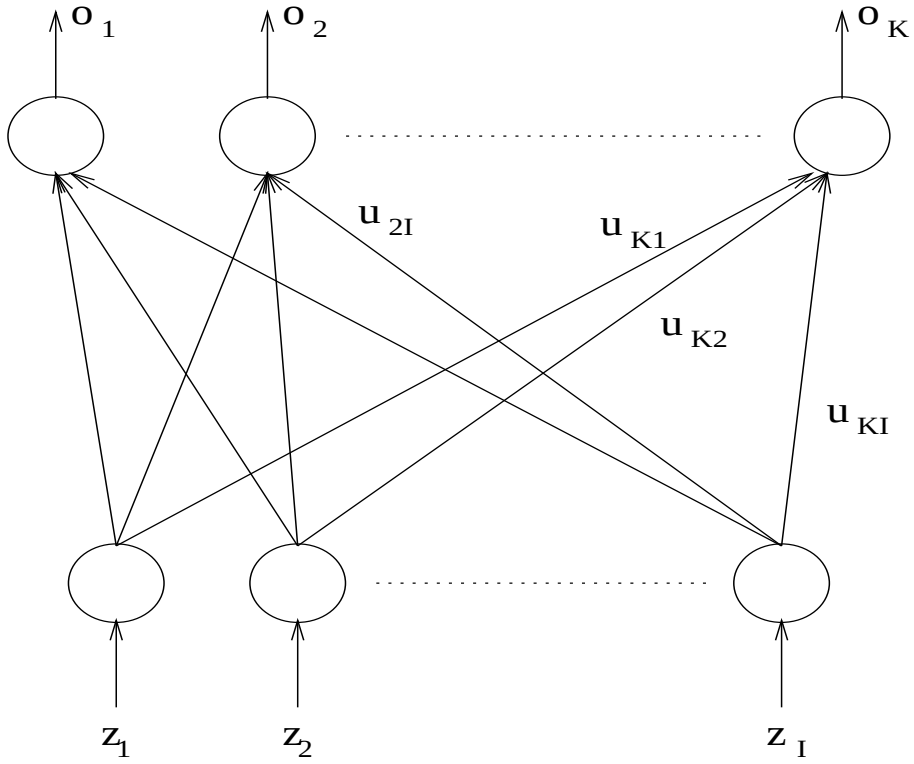


Figure 4.1: Unsupervised neural network

A summary of the Hebbian learning rule is given below:

1. Initialize all weights such that $u_{ki} = 0, \ \forall i = 1, \cdots, I$ and $\forall k = 1, \cdots, K$.
2. For each input pattern $\vec{z}_p$ compute the corresponding output vector $\vec{o}_p$.
3. Adjust the weights using equation (4.3).
4. Stop when the changes in weights are sufficiently small, or the maximum number of epochs has been reached; otherwise go to step 2.

A problem with Hebbian learning is that repeated presentation of input patterns leads to an exponential growth in weight values, driving the weights into saturation. To prevent saturation, a limit is posed on the increase in weight values. One type of limit is to introduce a nonlinear *forgetting factor*:

$$\Delta u_{ki}(t) = \eta o_{k,p} z_{i,p} - \alpha o_{k,p} u_{ki}(t-1) \tag{4.4}$$

where α is a positive constant, or equivalently,

$$\Delta u_{ki}(t) = \alpha o_{k,p}[\beta z_{i,p} - u_{ki}(t-1)] \tag{4.5}$$

with $\beta = \eta/\alpha$. Equation (4.5) implies that inputs for which $z_{i,p} < u_{ki}(t-1)/\beta$ have their corresponding weights u_{ki} decreased by a value proportional to the output value $o_{k,p}$. When $z_{i,p} > u_{ki}(t-1)/\beta$, weight u_{ki} is increased proportional to $o_{k,p}$.

Sejnowski proposed another way to formulate Hebb's postulate, using the covariance correlation of the neuron activation values [Sejnowski 1977]:

$$\Delta u_{ki}(t) = \eta[(z_{i,p} - \overline{z}_i)(o_{k,p} - \overline{o}_k)] \tag{4.6}$$

with

$$\overline{z}_i = \sum_{p=1}^{P} z_{i,p}/P \tag{4.7}$$

$$\overline{o}_k = \sum_{p=1}^{P} o_{k,p}/P \tag{4.8}$$

Another variant of the Hebbian learning rule uses the correlation in the changes in activation values over consecutive time steps. For this learning rule, referred to as *differential Hebbian learning*,

$$\Delta u_{ki}(t) = \eta \Delta z_i(t) \Delta o_k(t-1) \tag{4.9}$$

where

$$\Delta z_i(t) = z_{i,p}(t) - z_{i,p}(t-1) \tag{4.10}$$

and

$$\Delta o_k(t-1) = o_{k,p}(t-1) - o_{k,p}(t-2) \tag{4.11}$$

4.3 Principal Component Learning Rule

Principal component analysis (PCA) is a statistical technique used to transform a data space into a smaller space of the most relevant features. The aim is to project the original I-dimensional space onto an $I^{'}$-dimensional linear subspace, where $I^{'} < I$, such that the variance in the data is maximally explained within the smaller $I^{'}$-dimensional space. Features (or inputs) that have little variance are thereby removed. The principal components of a data set are found by calculating the covariance (or correlation) matrix of the data patterns, and by getting the minimal set of orthogonal vectors (the eigenvectors) that span the space of the covariance matrix. Given the set of orthogonal vectors, any vector in the space can be constructed with a linear combination of the eigenvectors.

Oja developed the first principal components learning rule, with the aim of extracting the principal components from the input data [Oja 1982]. Oja's principal

components learning is an extension of the Hebbian learning rule, referred to as normalized Hebbian learning, to include a feedback term to constrain weights. In doing so, principal components could be extracted from the data. The weight change is given as

$$\begin{aligned}\Delta u_{ki}(t) &= u_k i(t) - u_{ki}(t-1) \\ &= \eta o_{k,p}[z_{i,p} - o_{k,p}u_{ki}(t-1)] \\ &= \underbrace{\eta o_{k,p} z_{i,p}}_{Hebbian} - \underbrace{\eta o_{k,p}^2 u_{ki}(t-1)}_{forgetting\ \ factor}\end{aligned}$$

The first term corresponds to standard Hebbian learning (refer to equation (4.2)), while the second term is a forgetting factor to prevent weight values from becoming unbounded.

The value of the learning rate, η, above is important to ensure convergence to a stable state. If η is too large, the algorithm will not converge due to numerical unstability. If η is too small, convergence is extremely slow. Usually, the learning rate is time dependent, starting with a large value which decays gradually as training progresses. To ensure numerical stability of the algorithm, the learning rate $\eta_k(t)$ for output unit o_k must satisfy the inequality:

$$0 < \eta_k(t) < \frac{1}{1.2\lambda_k}$$

where λ_k is the largest eigenvalue of the covariance matrix, C_z, of the inputs to the unit [Oja and Karhuner 1985]. A good initial value is given as $\eta_k(0) = 1/[2Z^T Z]$, where Z is the input matrix.

Cichocki and Unbehauen provided an adaptive learning rate which utilizes a forgetting factor, γ, as follows [Cichocki and Unbehauen 1993]:

$$\eta_k(t) = \frac{1}{\frac{\gamma}{\eta_k(t-1)} + o_k^2(t)}$$

with

$$\eta_k(0) = \frac{1}{o_k^2(0)}$$

Usually, $0.9 \leq \gamma \leq 1$.

The above can be adapted to allow the same learning rate for all the weights in the following way:

$$\eta_k(t) = \frac{1}{\frac{\gamma}{\eta_k(t-1)} + ||\vec{o}(t)||_2^2}$$

with

$$\eta_k(0) = \frac{1}{||\vec{o}(0)||_2^2}$$

Sanger developed another principal components learning algorithm, similar to that of Oja, referred to as generalized Hebbian learning [Sanger 1989]. The only difference is the inclusion of more feedback information and a decaying learning rate $\eta(t)$:

$$\Delta u_{ki}(t) = \underbrace{\eta(t)[z_{i,p}o_{k,p}}_{Hebbian} - o_{k,p}\sum_{j=0}^{k} u_{ji}(t-1)o_{j,p}] \tag{4.12}$$

For more information on principal component learning, the reader is referred to the summary in [Haykin 1994].

4.4 Learning Vector Quantizer-I

One of the most frequently used unsupervised clustering algorithms is the learning vector quantizer (LVQ) developed by Kohonen [Kohonen 1995]. While several versions of LVQ exist, this section considers the unsupervised version, LVQ-I.

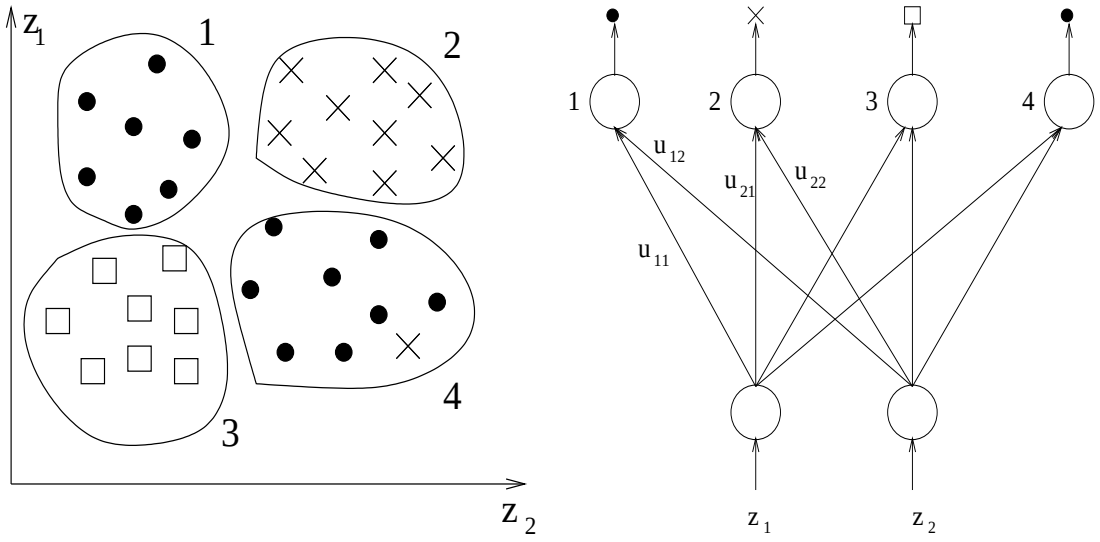
Ripley defined clustering algorithms as those algorithms where the purpose is to divide a set on n observations into m groups such that members of the same group are more alike than members of different groups [Ripley 1996]. The aim of a clustering algorithm is therefore to construct clusters of similar input vectors (patterns), where similarity is usually measured in terms of Euclidean distance. LVQ-I performs such clustering.

The training process of LVQ-I to construct clusters is based on competition. Referring to Figure 4.1, each output unit o_k represents a single cluster. The competition is among the cluster output units. During training, the cluster unit whose weight vector is the "closest" to the current input pattern is declared as the winner. The corresponding weight vector and that of neighbor units are then adjusted to better resemble the input pattern. The "closeness" of an input pattern to a weight vector is usually measured using the Euclidean distance. The weight update is given as

$$\Delta u_{ki}(t) = \begin{cases} \eta(t)[z_{i,p} - u_{ki}(t-1)] & \text{if } k \in \kappa_{k,p}(t) \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

where $\eta(t)$ is a decaying learning rate, and $\kappa_{k,p}(t)$ is the set of neighbors of the winning cluster unit o_k for pattern p. It is, of course, not strictly necessary that LVQ-I makes use of a neighborhood function, thereby updating only the weights of the winning output unit.

An illustration of clustering, as done by LVQ-I, is given in Figure 4.2. The input space, defined by two input units z_1 and z_2, is represented in Figure 4.2(a), while Figure 4.2(b) illustrates the LVQ-I network architecture required to form the clusters. Note that although only three classes exist, four output units are necessary –



(a) Clustering Problem (b) LVQ-I network

Figure 4.2: Learning vector quantizer to illustrate clustering

one for each cluster. Less output units will lead to errors since patterns of different classes will be grouped in the same cluster, while too many clusters may cause overfitting. For the problem illustrated in Figure 4.2(a), an additional cluster unit may cause a separate cluster to learn the single $\times$ in cluster 4.

The Kohonen LVQ-I algorithm is summarized below:

1. **Network initialization**
   Two main approaches to initializing the weights, as follows:

   (a) Either initialize weights to small random values sampled from a uniform distribution, or

   (b) Take the first input patterns as initial weight vectors. For the example in Figure 4.2(b), $u_{11} = z_{1,1}, u_{12} = z_{2,1}, u_{21} = z_{1,2}, u_{22} = z_{2,2}$, etc.

   Also initialize the learning rate and the neighborhood radius.

2. **The unsupervised learning**

   (a) For each pattern p:

      i. Compute the Euclidean distance $d_{k,p}$ between input vector $\vec{z}_p$ and

each weight vector $\vec{u}_k = (u_{k1}, u_{k2}, \cdots, u_{KI})$ as

$$d_{k,p}(\vec{z}_p, \vec{u}_k) = \sqrt{\sum_{i=1}^{I}(z_{i,p} - u_{ki})^2} \tag{4.14}$$

ii. Find the output unit o_k for which the distance $d_{k,p}$ is the smallest.

iii. Update all the weights for the neighborhood $\kappa_{k,p}$ using equation (4.13).

(b) Update the learning rate.

(c) Reduce the neighborhood radius at specified learning iterations.

(d) If stopping conditions are satisfied, stop training; otherwise go to step 2.

Stopping conditions may be

- a maximum number of epochs is reached,
- stop when weight adjustments are sufficiently small,
- a small enough quantization error has been reached, where the quantization error is defined as

$$\mathcal{E}_T = \sum_{p=1}^{P_T} ||\vec{z}_p - \vec{u}_k||_2^2 \tag{4.15}$$

One problem that may occur in LVQ networks is that one cluster unit may dominate as the winning cluster unit. The danger of such a scenario is that most patterns will be in one cluster. To prevent one output unit from dominating, a "conscience" factor is incorporated in a function to determine the winning output unit. The conscience factor penalizes an output for winning too many times. The activation value of output units is calculated using

$$o_{k,p} = \begin{cases} 1 & \text{for } \min_{\forall k}\{d_{k,p}(\vec{z}_p, \vec{u}_k) - b_k(t)\} \\ 0 & \text{otherwise} \end{cases} \tag{4.16}$$

where

$$b_k(t) = \gamma(\frac{1}{I} - g_k(t)) \tag{4.17}$$

and

$$g_k(t) = g_k(t-1) + \beta(o_{k,p} - g_k(t-1)) \tag{4.18}$$

In the above, $d_{k,p}$ is the Euclidean distance as defined in equation (4.14), I is the total number of input units, and $g_k(0) = 0$. Thus, $b_k(0) = \frac{1}{I}$, which initially gives each output unit an equal chance to be the winner; $b_k(t)$ is the conscience factor defined for each output unit. The more an output unit wins, the larger the value of $g_k(t)$ becomes, and $b_k(t)$ becomes larger negative. Consequently, a factor $|b_k(t)|$ is added to the distance $d_{k,p}$. Usually, for normalized inputs, $\beta = 0.0001$ and $\gamma = 10$.

4.5 Self-Organizing Feature Maps

Kohonen developed the self-organizing feature map (SOM), as motivated by the self-organization characteristics of the human cerebral cortex. Studies of the cerebral cortex showed that the motor cortex, somatosensory cortex, visual cortex and auditory cortex are represented by topologically ordered maps. These topological maps form to represent the structures sensed in the sensory input signals.

The self-organizing feature map is a multidimensional scaling method to project an I-dimensional input space to a discrete output space, effectively performing a compression of input space onto a set of codebook vectors. The output space is usually a two-dimensional grid. The SOM uses the grid to approximate the probability density function of the input space, while still maintaining the topological structure of input space. That is, if two vectors are close to one another in input space, so is the case for the map representation. The operation of the SOM can be illustrated as follows: consider input space to be a cloud. The map can then be viewed as an elastic net that spans the cloud.

The SOM closely resembles the learning vector quantizer discussed in the previous section. The difference between the two unsupervised algorithms is that neurons are organized on a rectangular grid for SOM, and neighbors are updated to also perform an ordering of the neurons. In the process, SOMs effectively cluster the input vectors through a competitive learning process, while maintaining the topological structure of the input space.

Section 4.5.1 explains the standard stochastic SOM training rule, while a batch version is discussed in Section 4.5.2. A growing approach to SOM is given in Section 4.5.3. Different approaches to speed up the training of SOMs are overviewed in Section 4.5.4. Section 4.5.5 explains the formation of clusters for visualization purposes. Section 4.5.6 discusses in brief different ways how the SOM can be used after training.

4.5.1 Stochastic Training Rule

SOM training is based on a competitive learning strategy. Assume I-dimensional input vectors $\vec{z}_p$, where the subscript p denotes a single training pattern. The first step of the training process is to define a map structure, usually a two-dimensional grid (refer to Figure 4.3). The map is usually square, but can be of any rectangular shape. The number of elements (neurons) in the map is less than the number of training patterns. Ideally, the number of neurons should be less than or equal to the number of independent training patterns.

With each neuron on the map is associated an I-dimensional weight vector which forms the centroid of one cluster. Larger cluster groupings are formed by grouping

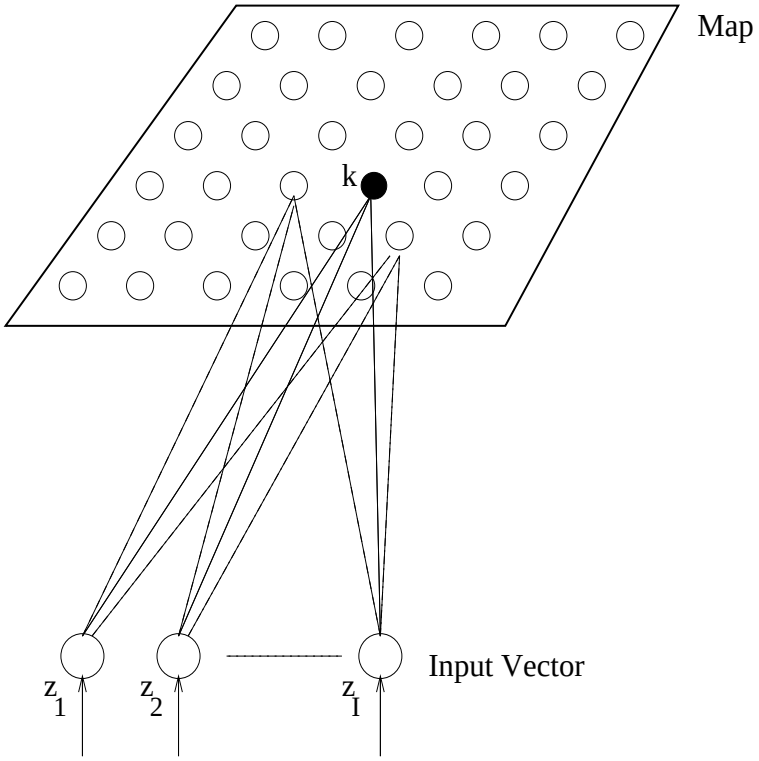


Figure 4.3: Self-organizing map

together "similar" neighboring neurons.

Initialization of the codebook vectors can occur in various ways:

- Assign random values to each weight $\vec{w}_{kj} = (w_{kj1}, w_{kj2}, \cdots, w_{KJI})$, with K the number of rows and J the number of columns of the map. The initial values are bounded by the range of the corresponding input parameter. While random initialization of weight vectors is simple to implement, this form of initialization introduces large variance components into the map which increases training time.

- Assign to the codebook vectors randomly selected input patterns. That is,

  $$\vec{w}_{kj} = \vec{z}_p$$

  with $p \sim U(1, P)$.

  This approach may lead to premature convergence, unless weights are perturbed with small random values.

- Find the principal components of the input space, and initialize the codebook vectors to reflect these principal components.

- A different technique of weight initialization is due to Su *et al.*, where the objective is to define a large enough hyper cube to cover all the training patterns [Su *et al.* 1999]. The algorithm starts by finding the four extreme points of the map by determining the four extreme training patterns. Firstly, two patterns are found with the largest inter-pattern Euclidean distance. A third pattern is located at the furthest point from these two patterns, and the fourth pattern with largest Euclidean distance from these three patterns. These four patterns form the corners of the map. Weight values of the remaining neurons are found through interpolation of the four selected patterns, in the following way:
  - Weights of boundary neurons are initialized as

$$\begin{aligned}
\vec{w}_{1j} &= \frac{\vec{w}_{1J} - \vec{w}_{11}}{J-1}(j-1) + \vec{w}_{11} \\
\vec{w}_{Kj} &= \frac{\vec{w}_{KJ} - \vec{w}_{K1}}{J-1}(j-1) + \vec{w}_{K1} \\
\vec{w}_{k1} &= \frac{\vec{w}_{K1} - \vec{w}_{11}}{K-1}(k-1) + \vec{w}_{11} \\
\vec{w}_{kJ} &= \frac{\vec{w}_{KJ} - \vec{w}_{1J}}{K-1}(k-1) + \vec{w}_{1J}
\end{aligned}$$

    for all $j = 2, \cdots, J-1$ and $k = 2, \cdots, K-1$.
  - The remaining codebook vectors are initialized as

$$\vec{w}_{kj} = \frac{\vec{w}_{kJ} - \vec{w}_{k1}}{J-1}(j-1) + \vec{w}_{k1}$$

    for all $j = 2, \cdots, J-1$ and $k = 2, \cdots, K-1$.

The standard training algorithm for SOMs is stochastic, where codebook vectors are updated after each pattern is presented to the network. For each neuron, the associated codebook vector is updated as

$$\vec{w}_{kj}(t+1) = \vec{w}_{kj}(t) + h_{mn,kj}(t)[\vec{z}_p - \vec{w}_{kj}(t)] \tag{4.19}$$

where mn is the row and column index of the winning neuron. The winning neuron is found by computing the Euclidean distance from each codebook vector to the input vector, and selecting the neuron closest to the input vector. That is,

$$||\vec{w}_{mn} - \vec{z}_p||_2 = \min_{\forall kj}\{||\vec{w}_{kj} - \vec{z}_p||_2^2\}$$

The function $h_{mn,kj}(t)$ in equation (4.19) is referred to as the neighborhood function. Thus, only those neurons within the neighborhood of the winning neuron mn have their codebook vectors updated. For convergence, it is necessary that $h_{mn,kj}(t) \to 0$ when $t \to \infty$.

The neighborhood function is usually a function of the distance between the coordinates of the neurons as represented on the map, i.e.

$$h_{mn,kj}(t) = h(||c_{mn} - c_{kj}||_2^2, t)$$

with the coordinates $c_{mn}, c_{kj} \in \mathbb{R}^2$. With increasing value of $||c_k - c_j||_2^2$ (that is, neuron kj is further away from the winning neuron mn), $h_{mn,kj} \to 0$. The neighborhood can be defined as a square or hexagon. However, the smooth Gaussian kernel is mostly used:

$$h_{mn,kj}(t) = \eta(t) e^{-\frac{||c_{mn} - c_{kj}||_2^2}{2\sigma^2(t)}} \tag{4.20}$$

where $\eta(t)$ is the learning rate factor and $\sigma(t)$ is the width of the kernel. Both $\eta(t)$ and $\sigma(t)$ are monotonically decreasing functions.

The learning process is iterative, continuing until a "good" enough map has been found. The quantization error is usually used as an indication of map accuracy, defined as the sum of Euclidean distances of all patterns to the codebook vector of the winning neuron, i.e.

$$\mathcal{E}_T = \sum_{p=1}^{P} ||\vec{z}_p - \vec{w}_{mn}(t)||_2^2$$

Training stops when $\mathcal{E}$ is sufficiently small.

4.5.2 Batch Map

The stochastic SOM training algorithm is slow due to the updates of weights after each pattern presentation: all the weights are updated. Batch versions of the SOM training rule have been developed which update weight values only after all patterns have been presented. The first batch SOM training algorithm was developed by Kohonen, and is summarized as follows [Kohonen 1997]:

1. Initialize the codebook vectors by assigning to them the first KJ training patterns, where KJ is the total number of neurons in the map.
2. For each neuron, kj, collect a list of copies of all patterns $\vec{z}_p$ whose nearest codebook vector belongs to the topological neighborhood of that neuron.
3. Each codebook vector is then assigned the mean over the corresponding list of patterns.
4. If convergence is reached, then terminate training; otherwise return to step 2.

Based on the batch learning approach above, Kaski *et al.* developed a faster version, as summarized below [Kaski *et al.* 2000]:

1. Initialize the codebook vectors $\vec{w}_{kj}$, using any initialization approach.
2. For each neuron, kj, compute the mean over all patterns for which that neuron is the winner. Denote the average by $\Delta\vec{w}_{kj}$.
3. Adapt the weight values for each codebook vector, using

$$\vec{w}_{kj} = \frac{\sum_{nm} N_{nm} h_{nm,kj} \Delta\vec{w}_{nm}}{\sum_{nm} N_{nm} h_{nm,kj}}$$

   where nm iterates over all neurons, N_{nm} is the number of patterns for which neuron nm is the winner, and $h_{nm,kj}$ is the neighborhood function which indicates if neuron nm is in the neighborhood of neuron kj, and to what degree.
4. Test convergence. If the algorithm did not converge, go to step 2.

4.5.3 Growing SOM

One of the design problems when using a SOM is deciding on the size of the map. Too many neurons may cause overfitting of the training patterns, with small clusters containing a few patterns. Alternatively, the final SOM may have succeeded in forming good clusters of similar patterns, but with many neurons with a zero, or close to zero frequency. The frequency of a neuron refers to the number of patterns for which that neuron is the winner, referred to as the best matching neuron (BMN). Too many neurons also cause a substantial increase in computational complexity. Too few neurons, on the other hand, will result in clusters with a high variance among the cluster members.

An approach to find near optimal SOM architectures is to start training with a small architecture, and to grow the map when more neurons are needed. One such SOM growing algorithm is given below, assuming a square map structure. Note that the map-growing process coexists with the training process.

1. Initialize the codebook vectors for a small, undersized SOM.
2. Grow the map:
   (a) Train the SOM for ξ pattern presentations, using any SOM training method.
   (b) Find the neuron kj with the largest quantization error.
   (c) Find the furthest immediate neighbor mn in the row-dimension of the map, and the furthest neuron op in the column-dimension.
   (d) Insert a column between neurons kj and op and a row between neurons kj and mn (this step preserves the square structure of the map).

(e) For each neuron ab in the new column, initialize the corresponding codebook vectors $\vec{w}_{ab}$ using

$$\vec{w}_{ab} = \alpha(\vec{w}_{a,b-1} + \vec{w}_{a,b+1})$$

and for each neuron in the new row,

$$\vec{w}_{ab} = \alpha(\vec{w}_{a-1,b} + \vec{w}_{a+1,b})$$

where $\alpha \in (0, 1)$

(f) Stop growing when any one of the following criteria is satisfied:

- the maximum map size has been reached;
- the largest neuron quantization error is less than a user specified threshold, λ;
- the map has converged to the specified quantization error.

3. Refine the map:
Refine the weights of the final SOM architecture with additional training steps until convergence has been reached.

A few aspects of the growing algorithm above need some explanation. These are the constants λ, γ and the maximum map size. A good choice for α is 0.5. The idea of the interpolation step is to assign a weight vector to the new neuron ab such that it removes patterns from the largest quantization error neuron kj in order to reduce the quantization error of neuron kj. A value less than 0.5 will position neuron ab closer to kj, with the chance that more patterns will be removed from neuron kj. A value larger than 0.5 will have the opposite effect.

The quantization error threshold, λ, is important to ensure that a sufficient map size is constructed. A small value for λ may result in too large a map architecture, while too large a λ may result in longer training times to reach a large enough architecture.

An upper bound on the size of the map is easy to determine: it is simply the number of training patterns, P_T. This is, however, undesirable. The maximum map size is rather expressed as βP_T, with $\beta \in (0, 1)$. The optimal value of β is problem dependent, and care should be taken to ensure that β is not too small if a growing SOM is not used. If this is the case, the final map may not converge to the required quantization error, since the map size will be too small.

4.5.4 Improving Convergence Speed

Training of SOMs is slow, due to the large number of weight updates involved (all the weights are updated for standard SOM training). Several mechanisms have been developed to reduce the number of training calculations, thereby improving speed of convergence. BatchMap is one such mechanism. Other approaches include the following:

Optimizing the neighborhood

If the Gaussian neighborhood function as given in equation (4.20) is used, all neurons will be in the neighborhood of the BMN, but to different degrees, due to the asymptotic characteristics of the function. Thus, all codebook vectors are updated, even if they are far from the BMN. This is strictly not necessary, since neurons far away from the BMN are dissimilar to the presented pattern, and will have negligible weight changes. Many calculations can therefore be saved by clipping the Gaussian neighborhood at a certain threshold – without degrading the performance of the SOM.

Additionally, the width of the neighborhood function can change dynamically during training. The initial width is large, with a gradual decrease in the variance of the Gaussian, which controls the neighborhood. For example,

$$\sigma(t) = \sigma(0)e^{-t/\tau_1} \tag{4.21}$$

where τ_1 is a positive constant, and $\sigma(0)$ is the initial, large variance.

If the growing SOM (refer to Section 4.5.3) is used, the width of the Gaussian neighborhood function should increase with each increase in map size.

Learning Rate

A time-decaying learning rate may be used, where training starts with a large learning rate which gradually decreases. That is,

$$\eta(t) = \eta(0)e^{-t/\tau_2} \tag{4.22}$$

where τ_2 is a positive constant and $\eta(0)$ is the initial, large learning rate (refer to chapter 7 to read about the consequences of large and small learning rates).

Shortcut Winner Search

The shortcut winner search decreases the computational complexity by using a more efficient search for the BMN. The search is based on the premise that the BMN of a pattern is in the vicinity of the BMN for the previous epoch. The search for a BMN is therefore constrained to the current BMN and its neighborhood. In short, the search for a BMN for each pattern is summarized as

1. Retrieve the previous BMN.
2. Calculate the distance of the pattern to the codebook vector of the previous BMN.

3. Calculate the distance of the pattern to all direct neighbors of the previous BMN.

4. If the previous BMN is still the best, then terminate the search; otherwise, let the new BMN be the neuron (within the neighborhood) closest to that pattern.

Shortcut winner search does not perform a search for the BMN over the entire map, but just within the neighborhood of the previous BMN, thereby substantially reducing computational complexity.

4.5.5 Clustering and Visualization

The effect of the SOM training process is to cluster together similar patterns, while preserving the topology of input space. After training, all that is given is the set of trained weights with no explicit cluster boundaries. An additional step is required to find these cluster boundaries.

One way to determine and visualize these cluster boundaries is to calculate the unified distance matrix (U-matrix), which contains a geometrical approximation of the codebook vector distribution in the map. The U-matrix expresses for each neuron, the distance to the neighboring codebook vectors. Large values within the U-matrix indicate the position of cluster boundaries. Using a gray-scale scheme, Figure 4.4(a) visualizes the U-matrix for the iris classification problem.

For the same problem, Figure 4.4(b) visualizes the clusters on the actual map. Boundaries are usually found by using Ward clustering of the codebook vectors. Ward clustering follows a bottom-up approach where each neuron initially forms its own cluster. At consecutive iterations, two clusters which are closest to one another are merged, until the optimal, or specified number of clusters has been constructed. The end result of Ward clustering is a set of clusters with a small variance over its members, and a large variance between separate clusters.

The Ward distance measure is used to decide which clusters should be merged. The distance measure is defined as

$$d_{rs} = \frac{n_r n_s}{n_r + n_s} ||\vec{w}_r - \vec{w}_s||_2^2$$

where r and s are cluster indices, n_r and n_s are the number of patterns within the clusters, and $\vec{w}_r$ and $\vec{w}_s$ are the centroid vectors of these clusters (i.e. the average of all the codebook vectors within the cluster). The two clusters are merged if their distance, d_{rs}, is the smallest. For the newly formed cluster, q,

$$\vec{w}_q = \frac{1}{n_r + n_s}(n_r \vec{w}_r + n_s \vec{w}_s)$$

and

$$n_q = n_r + n_s$$

Note that, in order to preserve topological structure, two clusters can only be merged if they are adjacent. Furthermore, only clusters that have a nonzero number of patterns associated with them are merged.

4.5.6 Using SOM

The SOM has been applied to a variety of real-world problems, including image analysis, speech recognition, music pattern analysis, signal processing, robotics, telecommunications, electronic-circuit design, knowledge discovery and time series analysis. The main advantage of SOMs comes from the easy visualization and interpretation of clusters formed by the map.

In addition to visualizing the complete map as illustrated in Figure 4.4(b), the relative component values in the codebook vectors can be visualized as illustrated in the same figure. Here a component refers to an input attribute. That is, a component plane can be constructed for each input parameter (component) to visualize the distribution of the corresponding weight (using some color scale representation). The map and component planes can be used for exploratory data analysis. For example, a marked region on the visualized map can be projected onto the component planes to find the values of the input parameters for that region.

A trained SOM can also be used as a classifier. However, since no target information is available during training, the clusters formed by the map should be manually inspected and labeled. A data vector is then presented to the map, and the winning neuron determined. The corresponding cluster label is then used as the class.

Used in recall mode, the SOM can be used to interpolate missing values within a pattern. Given such a pattern, the BMN is determined, ignoring the inputs with missing values. A value is then found by either replacing the missing value with the corresponding weight of the BMN, or through interpolation among a neighborhood of neurons (e.g. take the average of the weight values of all neurons in the neighborhood of the BMN).

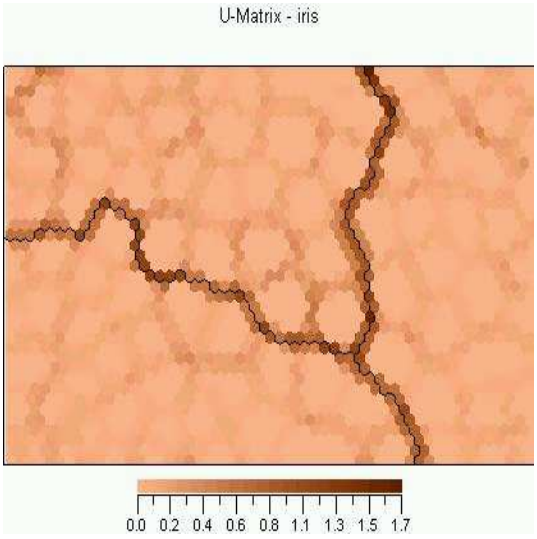
4.6 Conclusion

This chapter gave a short introduction to unsupervised learning algorithms, with emphasis on LVQ-I and SOMs. These algorithms are very useful in performing clustering, with applications in analysis of mammograms and landsat images, customer profiling, stock prediction, and many more. The next chapter presents learning

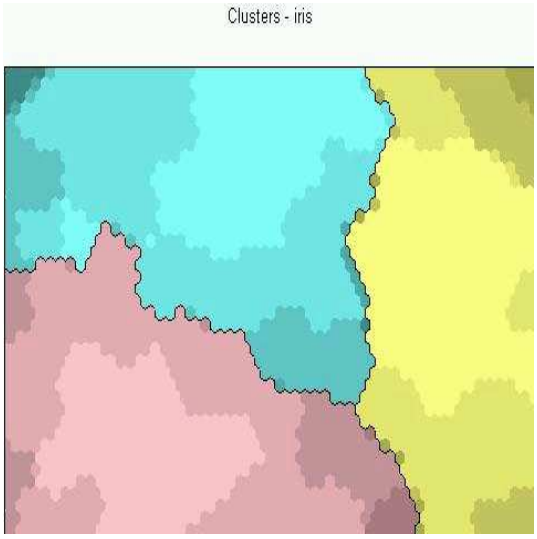
algorithms which combines supervised and unsupervised learning.

4.7 Assignments

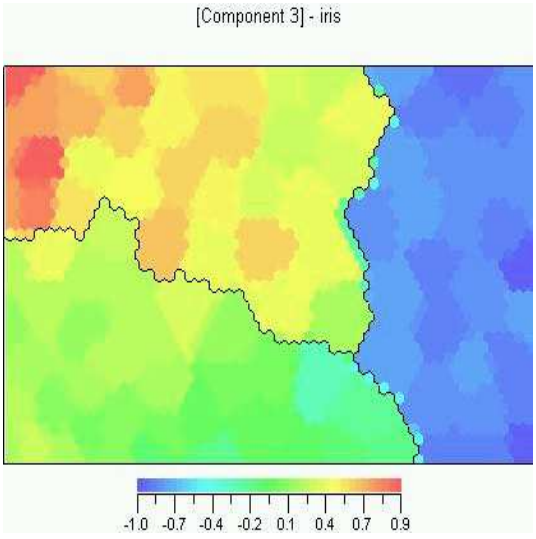
1. Implement and test a LVQ-I network to distinguish between different alphabetical characters of different fonts.
2. Explain why it is necessary to retrain a supervised NN on all the training data, including any new data that becomes available at a later stage. Why is this not an issue with unsupervised NNs?
3. Discuss an approach to optimize the LVQ-I network architecture.
4. How can PSO be used for unsupervised learning?
5. What is the main difference between the LVQ-I and SOM as an approach to cluster multi-dimensional data?
6. For a SOM, if the training set contains P_T patterns, what is the upper bound on the number of neurons necessary to fit the data? Justify your answer.
7. Explain the purpose of the neighborhood function of SOMs.
8. Assuming a Gaussian neighborhood function for SOMs, what can be done to reduce the number of weight updates in a sensible way?
9. Explain how a SOM can be used to distinguish among different hand gestures.
10. Discuss a number of ways in which the SOM can be adapted to reduce its computational complexity.



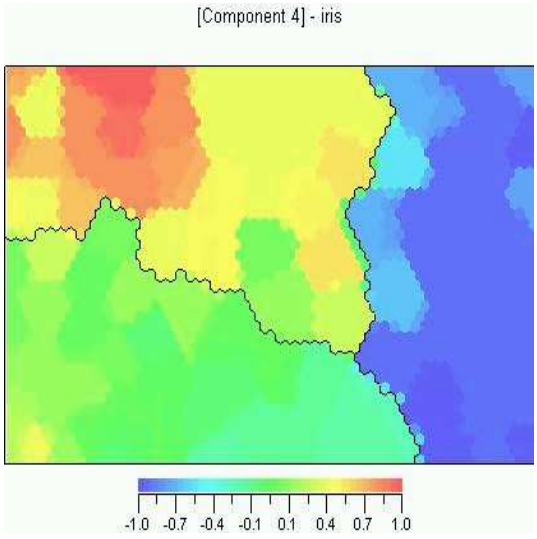
(a) U-Matrix



(b) Map Illustration



(c) Component Map for Third Input



(d) Component Map for Fourth Input

Figure 4.4: Visualization of SOM clusters for iris classification

Chapter 5

Radial Basis Function Networks

Several neural networks have been developed for both the supervised and the unsupervised learning paradigms. While these NNs were seen to perform very well in their respective application fields, improvements have been developed by combining supervised and unsupervised learning. This chapter discusses two such learning algorithms, namely the learning vector quantizer-II in Section 5.1 and radial basis function NNs in Section 5.2.

5.1 Learning Vector Quantizer-II

The learning vector quantizer (LVQ-II), developed by Kohonen, uses information from a supervisor to implement a reward and punish scheme. The LVQ-II assumes that the classifications of all input patterns are known. If the winning cluster unit correctly classifies the pattern, the weights to that unit are rewarded by moving the weights to better match the input pattern. On the other hand, if the winning unit misclassified the input pattern, the weights are penalized by moving them away from the input vector.

For the LVQ-II, the weight updates for the winning output unit o_k are given as

$$\Delta u_{ki} = \begin{cases} \eta(t)[z_{i,p} - u_{ki}(t-1)] & \text{if } o_{k,p} = t_{k,p} \\ -\eta(t)[z_{i,p} - u_{ki}(t-1)] & \text{if } o_{k,p} \neq t_{k,p} \end{cases} \tag{5.1}$$

Similarly to the LVQ-I, a conscience factor can be incorporated to penalize frequent winners.

5.2 Radial Basis Function Neural Networks

A radial basis function (RBF) neural network is a combination of Kohonen's LVQ-I and gradient descent. The architecture consists of three layers: an input layer, a hidden layer of J basis functions, and an output layer of K linear output units. The activation values of the hidden units are calculated as the closeness of the input vector $\vec{z}_p$ to an I-dimensional parameter vector $\vec{\mu}_j$ associated with hidden unit y_j. The activation is given as

$$y_{j,p} = e^{-\frac{||\vec{z}_p - \vec{\mu}_j||^2}{2\sigma_j^2}} \tag{5.2}$$

using a Gaussian basis function, or

$$y_{j,p} = [1 + e^{\frac{||\vec{z}_p - \vec{\mu}_j||}{\sigma_j^2} - \theta_j}]^{-1} \tag{5.3}$$

using a logistic function, where the norm is the Euclidean norm; σ_j and $\vec{\mu}_j$ are respectively the standard deviation and mean of the basis function. The final output is calculated as

$$o_{k,p} = \sum_{j=1}^{J} w_{kj} y_{j,p} \tag{5.4}$$

Figure 5.1 illustrates the architecture of an RBF network. The $\vec{\mu}_j$ vector is the weight vector to the j-th hidden unit (basis function), and σ_j^2 is the bias to the j-th hidden unit.

In the case of classification problems, an RBF network finds the centroids of data clusters, and uses these centroids as the centers of the Gaussian density function. Clusters are formed by fitting these bell-shaped basis functions over the input space. Classifications are then determined from a linear combination of the Gaussian density functions. In the case of function approximation, the target function is approximated by superimposing the basis functions.

Training of an RBF network is achieved in two steps: (1) unsupervised learning of the weights μ_{ji} between the input and hidden layers using LVQ-I, and then, (2) supervised training of the w_{kj} weights between the hidden and output layers using GD. A pseudo-code algorithm for the code is given below (more sophisticated algorithms can be found in the literature):

1. (a) Initialize all μ_{ji} weights to the average value of all inputs in the training set.
   (b) Initialize all variances σ_j^2 to the variance of all input values over the training set.
   (c) Initialize all w_{kj} weights to small random values.

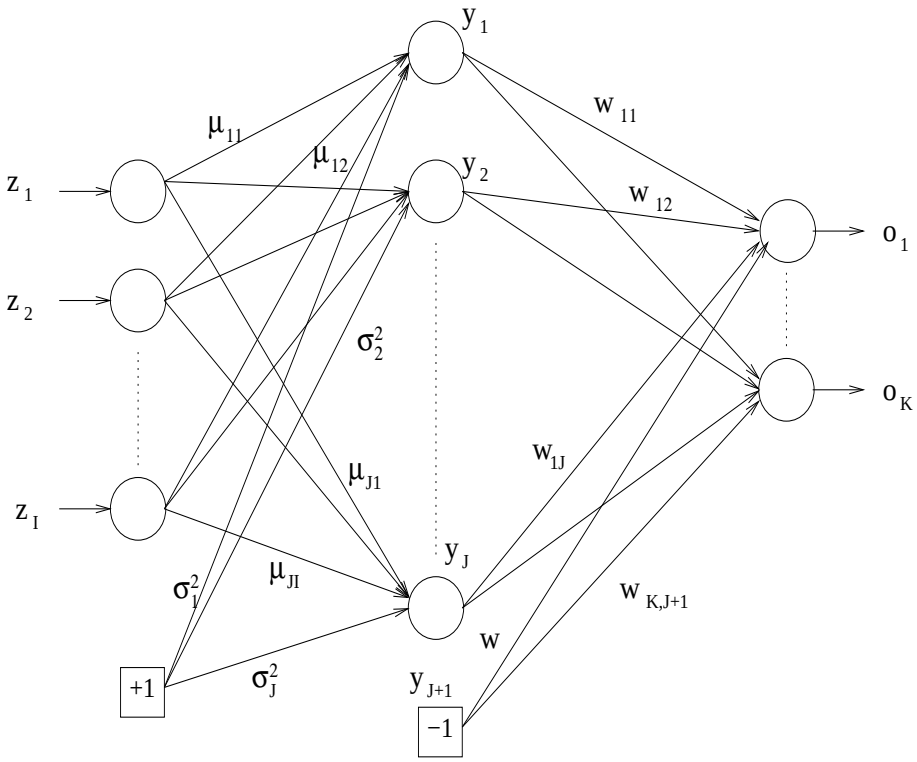


Figure 5.1: Radial basis function neural network

2. Learn the centroids $\vec{\mu}_j$ using LVQ-I:
   After each LVQ-I epoch, set the variance for all winning y_js as the average of the Euclidean distances of the y_js' mean weight vector $\vec{\mu}_j$ to the input patterns for which y_j was selected as the winner.

3. Learn the hidden-to-output weights w_{kj} using the adjustments

$$\Delta w_{kj}(t) = \eta \sum_{k=1}^{K} (t_{k,p} - o_{k,p}) y_{j,p} \tag{5.5}$$

4. Training stops when the GD phase converges.

The activation values of the hidden and output units can be used to compute a degree $\mathcal{P}_{k,p}$ to which a pattern p belongs to each class k:

$$\mathcal{P}_{k,p} = \frac{o_{k,p}}{\sum_{j=1}^{J} y_{j,p}} \tag{5.6}$$

RBF networks present several advantages, including (1) network output units provide a degree of membership to classes, (2) they do not experience local minima

problems as does standard FFNN training using GD, (3) the unsupervised training is insensitive to the order of pattern presentation. RBF networks are, however, slow in training, and may need more hidden units for function approximation problems than FFNNs.

5.3 Conclusion

Many extensions to both LVQ-II and RBF networks have been developed. These extensions all have as their aim improvement of performance. Interested readers are referred to the numerous research articles available on these extensions.

5.4 Assignments

1. Compare the performance of an RBF NN and a FFNN on a classification problem from the UCI machine learning repository.
2. Compare the performance of the Gaussian and logistic basis functions.
3. Suggest an alternative to compute the hidden-to-output weights instead of using GD.
4. Suggest an alternative to compute the input-to-hidden weights instead of using LVQ-I.
5. Investigate alternative methods to initialize an RBF NN.

Chapter 6

Reinforcement Learning

The last learning paradigm to be covered is reinforcement learning, with its origins in the psychology of animal learning. The basic idea is that of awarding the learner for correct actions, and punishing wrong actions. Reinforcement learning occurs in areas wider than just neural networks, but the focus of this chapter is its application to NN learning. The LVQ-II can be perceived as a form of reinforcement learning, since weights of the winning output unit are only positively updated if that output unit provided the correct response for the corresponding input pattern. If not, weights are penalized through adjustment away from that input pattern.

Section 6.2 presents another reinforcement learning rule, but first an overview of learning through awards is given in Section 6.1.

6.1 Learning through Awards

Formally defined, reinforcement learning is the learning of a mapping from situations to actions with the main objective to maximize the scalar reward, or reinforcement signal. Informally, reinforcement learning is defined as learning by trial-and-error from performance feedback from the environment or an external evaluator. The learner has absolutely no prior knowledge of what action to take, and has to discover (or explore) which actions yield the highest reward.

A typical reinforcement learning problem is illustrated in Figure 6.1. The learner receives sensory inputs from its environment, as a description of the current state of the perceived environment. An action is executed, upon which the learner receives the reinforcement signal or reward. This reward can be a positive or negative signal, depending on the correctness of the action. A negative reward has the effect of punishing the learner for a bad action.

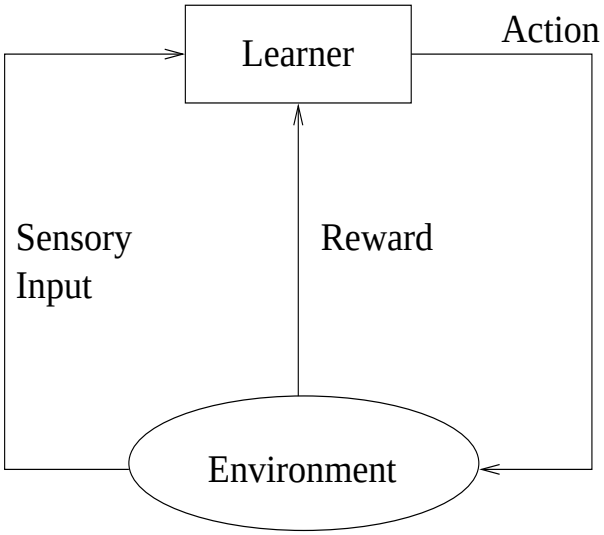


Figure 6.1: Reinforcement learning problem

6.2 Reinforcement Learning Rule

Neural network reinforcement learning usually requires a two-layer architecture, and a training set consisting of input vectors representing, for example, sensory inputs. A target action is also provided for each input pattern. Additionally, an external evaluator is required to decide whether the learner has scored a success or not. The weights are updated using

$$\Delta w_{kj} = \eta(r_p - \theta_k)e_{kj} \tag{6.1}$$

where η is the learning rate, r_p is an indication of success/failure for pattern p as provided by the external evaluator; θ_k is the reinforcement threshold value, and e_{kj} is the eligibility of weight w_{kj}:

$$e_{kj} = \frac{\partial}{\partial w_{kj}}[\ln(g_j)] \tag{6.2}$$

where

$$g_j = Prob(o_{k,p} = t_{k,p}|\vec{w}_k, \vec{z}_p) \tag{6.3}$$

Thus, this NN reinforcement learning rule computes a GD in probability space.

6.3 Conclusion

Reinforcement learning is slow learning process, but works well in situations where all the training data is not available at once. In such environments, the NN learns to adapt to new situations easily as they arise.

6.4 Assignments

1. Discuss how reinforcement learning can be used to guide a robot out of a room filled with obstacles.

Chapter 7

Performance Issues (Supervised Learning)

"For it is pointless to do with more what can be done with less."
-William of Ockham (1285–1349)

Performance is possibly the driving force of all organisms. If no attention is given to improve performance, the quality of life will not improve. Similarly, performance is the most important aspect that has to be considered when an artificial neural network is being designed. The performance of an artificial NN is not just measured as the accuracy achieved by the network, but aspects such as computational complexity and convergence characteristics are just as important. These measures and other measures that quantify performance are discussed in Section 7.1, with specific reference to supervised networks.

The design of NNs for optimal performance requires careful consideration of several factors that influence network performance. In the early stages of NN research and applications, the design of NNs was basically done by following the intuitive feelings of an expert user, or by following rules of thumb. The vast number of theoretical analysis of NNs made it possible to better understand the working of NNs – to unravel the "black box". These insights helped to design NNs with improved performance. Factors that influence the performance of NNs are discussed in Section 7.3.

Although the focus of this chapter is on supervised learning, several ideas can be extrapolated to unsupervised learning NNs.

7.1 Performance Measures

This section presents NN performance measures under three headings: *accuracy, complexity* and *convergence*.

7.1.1 Accuracy

Generalization is a very important aspect of neural network learning. Since it is a measure of how well the network interpolates to points not used during training, the ultimate objective of NN learning is to produce a learner with low generalization error. That is, to minimize the true risk function

$$\mathcal{E}_G(\Omega; W) = \int (\mathcal{F}_{NN}(\vec{z}, W) - \vec{t})^2 d\Omega(\vec{z}, \vec{t}) \tag{7.1}$$

where, from Section 3.2.1, $\Omega(\vec{z}, \vec{t})$ is the stationary density according to which patterns are sampled, W describes the network weights, and $\vec{z}$ and $\vec{t}$ are respectively the input and target vectors. The function $\mathcal{F}_{NN}$ is an approximation of the true underlying function. Since Ω is generally not known, $\mathcal{F}_{NN}$ is found through minimization of the empirical error function

$$\mathcal{E}_T(D_T; W) = \frac{1}{P_T} \sum_{p=1}^{P_T} (\mathcal{F}_{NN}(\vec{z}_p, W) - \vec{t}_p)^2 \tag{7.2}$$

over a finite data set $D_T \sim \Omega$. When $P_T \to \infty$, then $\mathcal{E}_T \to \mathcal{E}_G$. The aim of NN learning is therefore to learn the examples presented in the training set well, while still providing good generalization to examples not included in the training set. It is, however, possible that a NN exhibits a very low training error, but bad generalization due to overfitting (memorization) of the training patterns.

The most common measure of accuracy is the mean squared error (MSE), in which case the training error $\mathcal{E}_T$ is expressed as

$$\mathcal{E}_T = \frac{\sum_{p=1}^{P} \sum_{k=1}^{K} (t_{k,p} - o_{k,p})^2}{P_T K} \tag{7.3}$$

where P_T is the total number of training patterns in the training set D_T, and K is the number of output units. The generalization error $\mathcal{E}_G$ is approximated in the same way, but with the first summation over the P_G patterns in the generalization, or test set, D_G. Instead of the MSE, the sum squared error (SSE),

$$SSE = \sum_{p=1}^{P} \sum_{k=1}^{K} (t_{k,p} - o_{k,p})^2 \tag{7.4}$$

can also be used, where P is the total number of patterns in the data set considered. However, the SSE is not a good measure when the performance on different data set sizes are compared.

An additional error measure is required for classification problems, since the MSE alone is not a good descriptor of accuracy. In the case of classification problems, the percentage correctly classified (or incorrectly classified) patterns is used as a measure of accuracy. The reason why the MSE is not a good measure, is that the network may have a good accuracy in terms of the number of correct classifications, while having a relatively large MSE. If just the MSE is used to indicate when training should stop, it can happen that the network is trained too long in order to reach the low MSE. Hence, wasting time and increasing the chances of overfitting the training data (with reference to the number of correct classifications). But when is a pattern classified as correct? When the output class of the NN is the same as the target class – which is not a problem to determine when the ramp or step function is used as the activation function in the output layer. In the case of continuous activation functions, a pattern p is usually considered as being correctly classified if for each output unit o_k, $((o_{k,p} \geq 0.5 + \theta \ \ and \ \ t_{k,p} = 1)$ or $(o_{k,p} \leq 0.5 - \theta \ \ and \ \ t_{k,p} = 0))$, where $\theta \in [0, 0.5]$ – of course, assuming that the target classes are binary encoded.

An additional measure of accuracy is to calculate the correlation between the output and target values for all patterns. This measure, referred to as the correlation coefficient, is calculated as

$$\begin{aligned} r &= \frac{\sum_{i=1}^{n}(x_i - \overline{x})\sum_{i=1}^{n}(y_i - \overline{y})}{\sigma_x \sigma_y} \\ &= \frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n}\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2}\sqrt{\sum_{i=1}^{n} y_i^2 - \frac{1}{n}(\sum_{i=1}^{n} y_i)^2}} \end{aligned} \tag{7.5}$$

where x_i and y_i are observations, $\overline{x}$ and $\overline{y}$ are respectively the averages over all observations x_i and y_i, and σ_x and σ_y are the standard deviations of the x_i and y_i observations respectively, and can be used to quantify the linear relationship between variables x and y. As measure of learning accuracy, where $x = o_{k,p}$ and $y = t_{k,p}$, the correlation coefficient quantifies the linear relationship between the approximated (learned) function and the true function. A correlation value close to 1 indicates a good approximation to the true function. Therefore, the correlation coefficient

$$r = \frac{\sum_{p=1}^{P} o_{k,p} t_{k,p} - \frac{1}{P}\sum_{p=1}^{P} o_{k,p} \sum_{p=1}^{P} t_{k,p}}{\sqrt{\sum_{p=1}^{P} o_{k,p}^2 - \frac{1}{P}(\sum_{p=1}^{P} o_{k,p})^2}\sqrt{\sum_{p=1}^{P} t_{k,p}^2 - \frac{1}{P}(\sum_{p=1}^{P} t_{k,p})^2}} \tag{7.6}$$

is calculated as measure of how well the NN approximates the true function.

Another very important aspect of NN accuracy is *overfitting*. Overfitting of a training set means that the NN memorizes the training patterns, and consequently loses the ability to generalize. That is, NNs that overfit cannot predict correct output

for data patterns not seen during training. Overfitting occurs when the NN architecture is too large, i.e. the NN has too many weights (in statistical terms: too many *free parameters*) – a direct consequence of having too many hidden units and irrelevant input units. If the NN is trained for too long, the excess free parameters start to memorize all the training patterns, and even noise contained in the training set. Remedies for overfitting include optimizing the network architecture and using enough training patterns (discussed in Section 7.3).

Estimations of generalization error during training can be used to detect the point of overfitting. The simplest approach to find the point of overfitting was developed through studies of training and generalization profiles. Figure 7.1 presents a general illustration of training and generalization errors as a function of training epochs. From the start of training, both the training and generalization errors decrease - usually exponentially. In the case of oversized NNs, there is a point at which the training error continues to decrease, while the generalization error starts to increase. This is the point of overfitting. Training should stop as soon an increase in generalization error is observed.

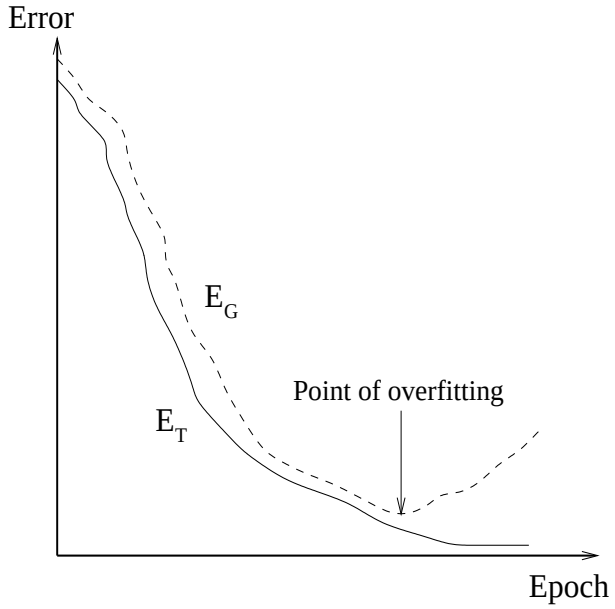


Figure 7.1: Illustration of overfitting

In order to detect the point of overfitting, the original data set is divided into three disjoint sets, i.e. the training set D_T, the generalization set D_G and the validation set D_V. The validation set is then used to estimate the generalization error. Since both the training error and the validation error usually fluctuate, determining the point of overfitting is not straightforward. A moving average of the validation error has to be used. Overfitting is then detected when

$$\mathcal{E}_V > \overline{\mathcal{E}}_V + \sigma_{\mathcal{E}_V} \tag{7.7}$$

where $\mathcal{E}_V$ is the MSE on the validation set, $\overline{\mathcal{E}}_V$ is the average MSE on the validation set since training started, and $\sigma_{\mathcal{E}_V}$ is the standard deviation in validation error.

Röbel suggested the *generalization factor* as an alternative indication of overfitting [Röbel 1994]. Röbel defines the generalization factor $\rho = \frac{\mathcal{E}_V}{\mathcal{E}_T}$, where $\mathcal{E}_V$ and $\mathcal{E}_T$ are the MSE on the validation set D_V and current training subset D_T respectively. The generalization factor indicates the error made in training on D_T only, instead of training on the entire input space. Overfitting is detected when $\rho(\xi) > \varphi_\rho(\xi)$, where $\varphi_\rho(\xi) = \min\{\varphi_\rho(\xi - 1), \overline{\rho} + \sigma_\rho, 1.0\}$; ξ is the current epoch, $\overline{\rho}$ is the average generalization factor over a fixed number of preceding epochs, and σ_ρ is the standard deviation. This test ensures that $\rho \leq 1.0$. Keep in mind that ρ does not give an indication of the accuracy of learning, but only the ratio between the training and validation error. For function approximation problems (as is the case with Röbel's work) where the MSE is used as a measure of accuracy, a generalization factor $\rho < 1$ means that the validation error is smaller than the training error – which is desirable. As ρ becomes large (greater than 1), the difference between the training error and validation error increases, which indicates an increase in validation error with a decrease in training error – an indication of overfitting. For classification problems where the percentage of correctly classified patterns is used as a measure of accuracy, ρ should be larger than 1.

It is important to note that the training error or the generalization error alone is not sufficient to quantify the accuracy of a NN. Both these errors should be considered.

Additional Reading Material on Accuracy

The trade-off between training error and generalization has prompted much research in the generalization performance of NNs. Average generalization performance has been studied theoretically to better understand the behavior of NNs trained on a finite data set. Research shows a dependence of generalization error on the training set, the network architecture and weight values. Schwartz, Samalam, Solla and Denker show the importance of training set size for good generalization in the context of ensemble networks [Schwartz *et al.* 1990]. Other research uses the VC-dimension (Vapnik-Chervonenkis dimension) [Abu-Mostafa 1989, Abu-Mostafa 1993, Cohn and Tesauro 1991, Opper 1994] to derive boundaries on the generalization error as a function of network and training set size. Best known are the limits derived by Baum and Hausler [Baum and Haussler 1989] and Hausler, Kearns, Opper and Schapire [Haussler *et al.* 1992]. While these limits are derived for, and therefore limited to, discrete input values, Hole derives generalization limits for real valued inputs [Hole 1996].

Limits on generalization have also been developed by studying the relationship between training error and generalization error. Based on Akaike's Final Prediction Error (FPE) and Information Criterion (AIC) [Akaike 1974], Moody derived

the Generalized Prediction Error (GPE) which gives a limit on the generalization error as a function of the training error, training set size, the number of effective parameters, and the effective noise variance [Moody 1992, Moody 1994]. Murata, Yoshizawa and Amari derived a similar Network Information Criterion [Murata *et al.* 1991, Murata *et al.* 1994a, Murata *et al.* 1994b]. Using a different approach, i.e. Vapnik's Bernoulli theorem, Depenau and Møller derived a bound as a function of training error, the VC-dimension and training set size [Depenau and Møller 1994].

These research results give, sometimes overly pessimistic, limits that help to clarify the behavior of generalization and its relationship with architecture, training set size and training error. Another important issue in the study of generalization is that of overfitting. Overfitting means that the NN learns too much detail, effectively memorizing training patterns. This normally happens when the network complexity does not match the size of the training set, i.e. the number of adjustable weights (free parameters) is larger than the number of independent patterns. If this is the case, the weights learn individual patterns, and even capture noise. This overfitting phenomenon is the consequence of training on a finite data set, minimizing the empirical error function given in equation (7.2), which differs from the true risk function given in equation (7.1).

Amari *et al.* developed a statistical theory of overtraining in the asymptotic case of large training set sizes [Amari *et al.* 1995, Amari *et al.* 1996]. They analytically determine the ratio in which patterns should be divided into training and test sets to obtain optimal generalization performance and to avoid overfitting. Overfitting effects under large, medium and small training set sizes have been investigated analytically by Amari *et al.* [Amari *et al.* 1995] and Müller *et al.* [Müller *et al.* 1995].

7.1.2 Complexity

The computational complexity of a NN is directly influenced by:

1. **The network architecture**:
   The larger the architecture, the more feedforward calculations are needed to predict outputs after training, and the more learning calculations are needed per pattern presentation.

2. **The training set size**:
   The larger the training set size, the more patterns are presented for training. Therefore, the total number of learning calculations per epoch is increased.

3. **Complexity of the optimization method**:
   As will be discussed in Section 7.3, sophisticated optimization algorithms have been developed to improve the accuracy and convergence characteristics of

NNs. The sophistication comes, however, at the cost of increased computational complexity to determine the weight updates.

Training time is usually quantified in terms of the number of epochs to reach specific training or generalization errors. When different learning algorithms are compared, the number of epochs is usually not an accurate estimate of training time or computational complexity. Instead, the total number of pattern presentations, or weight updates are used. A more accurate estimate of computational complexity is to count the total number of calculations made during training.

7.1.3 Convergence

The convergence characteristics of a NN can be described by the ability of the network to converge to specified error levels (usually considering the generalization error). The ability of a network to converge to a specific error is expressed as the number of times, out of a fixed number of simulations, that the network succeeded in reaching that error. While this is an empirical approach, rigorous theoretical analysis has been done for some network types.

7.2 Analysis of Performance

Any study of the performance of NNs (or any other algorithm for that matter) and any conclusions based on just one simulation are incomplete and inconclusive. Conclusions on the performance of NNs must be based on the results obtained from several simulations. For each simulation the NN starts with new random initial weights and uses a different training, validation and generalization sets, independent of previous sets. Performance results are then expressed as averages over all the simulations, together with variances, or confidence intervals.

Let ϱ denote the performance measure under consideration. Results are then reported as $\overline{\varrho} \pm \sigma_{\varrho}$. The average $\overline{\varrho}$ is an indication of the average performance over all simulations, while σ_{ϱ} gives an indication of the variance in performance. The σ_{ϱ} parameter is very important in decision making. For example, if two algorithms A and B are compared where the MSE for A is 0.001 ± 0.0001, and that of B is 0.0009 ± 0.0006, then algorithm A will be preferred even though B has a smaller MSE. Algorithm A has a smaller variance, having MSE values in the range $[0.0009, 0.0011]$, while B has MSE values in a larger range of $[0.0003, 0.0015]$.

While the above approach to present results is sufficient, results are usually reported with associated confidence intervals. If a confidence level of $\alpha = 0.01$ is used, for example, then 99% of the observations will be within the calculated confidence interval. Before explaining how to compute the confidence intervals, it is important to

note that at least 30 independent simulations are needed. This allows the normality assumption as stated by the Central Limit Theorem: the probability distribution governing the variable $\overline{\varrho}$ approaches a Normal distribution as the number of observations (simulations) tends to infinity. Using this result, the confidence interval associated with confidence level α is estimated as

$$\overline{\varrho} \pm t_{\alpha,n-1}\sigma_{\varrho} \tag{7.8}$$

where $t_{\alpha,n-1}$ is a constant obtained from the t-distribution with $n-1$ degrees of freedom (n is the number of simulations) and

$$\sigma_{\varrho} = \sqrt{\frac{\sum_{i=1}^{n}(\varrho_i - \overline{\varrho})^2}{n(n-1)}} \tag{7.9}$$

7.3 Performance Factors

This section discusses various aspects that have an influence on the performance of supervised NNs. These aspects include data manipulation, learning parameters, architecture selection, and optimization methods.

7.3.1 Data Preparation

One of the most important steps in using a NN to solve real-world problems is to collect and transform data into a form acceptable to the NN. The first step is to decide on what the inputs and the outputs are. Obviously irrelevant inputs should be excluded. Section 7.3.5 discusses ways in which the NN can decide itself which inputs are irrelevant. The second step is to process the data in order to remove outliers, handle missing data, transform non-numeric data to numeric data and to scale the data into the active range of the activation functions used. Each of these aspects are discussed below:

Missing Values

It is common that real-world data sets have missing values for input parameters. NNs need a value for each of the input parameters. Therefor, something has to be done with missing values. The following options exist:

- Remove the entire pattern if it has a missing value. While pattern removal solves the missing value problem, other problems are introduced: (1) the available information for training is reduced which can be a problem if data is already limited, and (2) important information may be lost.

- Replace each missing value with the average value for that input parameter in the case of continuous values, or with the most frequently occurring value in the case of nominal or discrete values. This replacing of missing values introduces no bias.
- For each input parameter that has a missing value, add an additional input unit to indicate patterns for which parameters are missing. It can then be determined after training whether the missing values had a significant influence on the performance of the network.

While missing values present a problem to supervised neural networks, SOMs do not suffer under these problems. Missing values do not need to be replaced. The BMN for a pattern with missing values is, for example, calculated by ignoring the missing value and the corresponding weight value of the codebook vector in the calculation of the Euclidean distance between the pattern and codebook vector.

Coding of Input Values

All input values to a NN must be numeric. Nominal values therefore need to be transformed to numerical values. A nominal input parameter that has n different values is coded as n different binary input parameters, where the input parameter that corresponds to a nominal value has the value 1, and the rest of these parameters have the value 0. An alternative is to use just one input parameter and to map each nominal value into an equivalent numerical value. This is, however, not a good idea, since the NN will interpret the input parameter as having continuous values, thereby losing the discrete characteristic of the original data.

Outliers

Outliers have severe effects on accuracy, especially when gradient descent is used with the SSE as objective function. An outlier is a data pattern that deviates substantially from the data distribution. Because of the large deviation from the norm, outliers result in large errors, and consequently large weight updates. Figure 7.3 shows that larger differences between target and output values cause an exponential increase in the error if the SSE is used as objective function. The fitted function is then pulled toward the outliers in an attempt to reduce the training error. As result, the generalization deteriorates. Figure 7.2 illustrates this effect.

The outlier problem can be addressed in the following ways:

- Remove outliers before training starts, using statistical techniques. While such actions will eliminate the outlier problem, it is believed that important information about the data might also be removed at the same time.

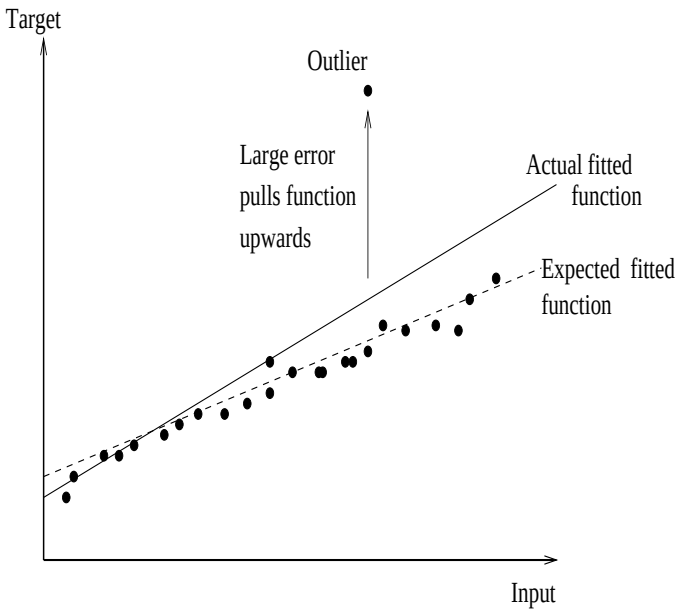


Figure 7.2: Effect of outliers

- Use a robust objective function that is not influenced by outliers. An example objective function is the Huber function as illustrated in Figure 7.4 [Huber 1981]. Patterns for which the error is larger than $|\alpha|$ have a constant value, and have a zero influence when weights are updated (the derivative of a constant is zero).

- Slade and Gedeon [Slade and Gedeon 1993] and Gedeon, Wong and Harris [Gedeon *et al.* 1995] proposed Bimodal Distribution Removal, where the aim is to remove outliers from training sets during training. Frequency distributions of pattern errors are analyzed during training to identify and remove outliers. If the original training set contains no outliers, the method simply reduces to standard learning.

Scaling and Normalization

Data needs to be scaled to the active range and domain of the activation functions used. While it is not necessary to scale input values, performance can be improved if inputs are scaled to the active domain of the activation functions. For example, consider the sigmoid activation function. Simple mathematical calculations show that the active domain of the sigmoid function is $[-\sqrt{3}, \sqrt{3}]$, corresponding to the parts of the function for which changes in input values have relatively large changes in output. Values near the asymptotic ends of the sigmoid function have a very small influence on weight updates. Changes in these values result in very small changes in

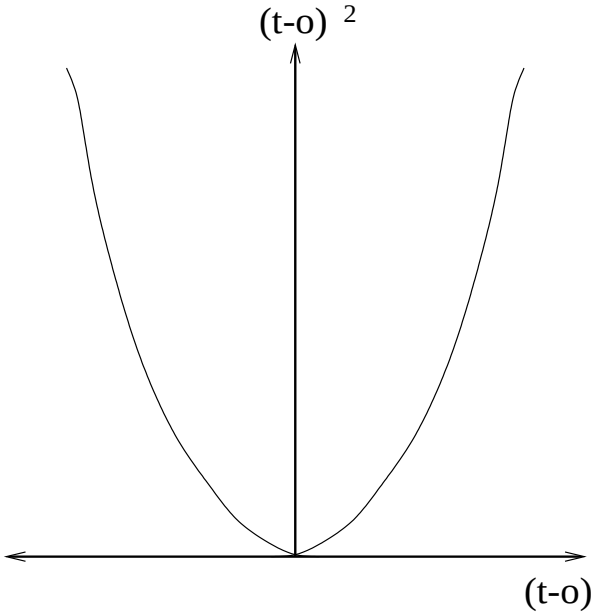


Figure 7.3: SSE objective function

output. Furthermore, the derivatives near the asymptotes are approximately zero, causing weight updates to be approximately zero. Therefor achieving no learning in these areas.

When bounded activation functions are used, the target values have to be scaled to the range of the activation function, for example $(0, 1)$ for the sigmoid function and $(-1, 1)$ for the hyperbolic tangent. If $t_{u,max}$ and $t_{u,min}$ are the maximum and minimum values of the unscaled target t_u, then,

$$t_s = \frac{t_u - t_{u,min}}{t_{u,max} - t_{u,min}}(t_{s,max} - t_{s,min}) + t_{s,min} \tag{7.10}$$

where $t_{s,max}$ and $t_{s,min}$ are the new maximum and minimum values of the scaled values, linearly maps the range $[t_{u,min}, t_{u,max}]$ to the range $[t_{s,min}, t_{s,max}]$.

In the case of classification problems, target values are usually elements of the set $\{0.1, 0.9\}$ for the sigmoid function. The value 0.1 is used instead of 0, and 0.9 instead of 1. Since the output of the sigmoid function can only approach 0 and 1, a NN can never converge to the best set of weights if the target values are 0 or 1. In this case the goal of the NN is always out of reach, and the network continues to push weight values toward extreme values until training is stopped.

Scaling of target values into a smaller range does have the disadvantage of increased training time. Engelbrecht *et al.* showed that if target values are linearly scaled using

$$t_s = c_1 t_u + c_2 \tag{7.11}$$

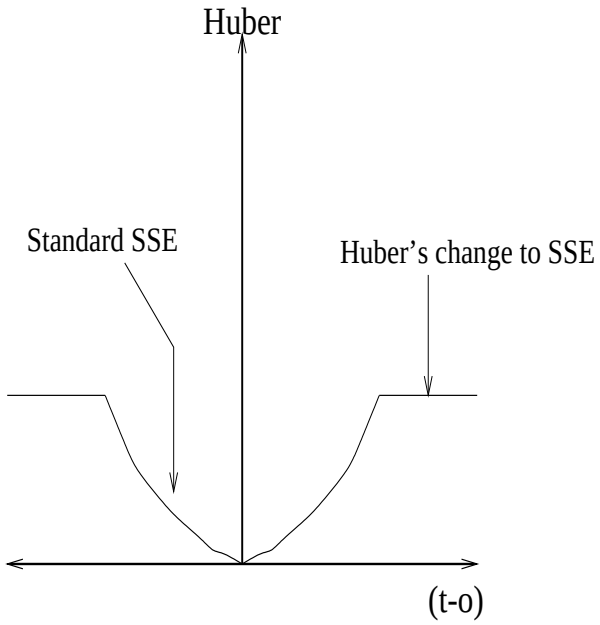


Figure 7.4: Huber objective function

where t_s and t_u are respectively the scaled and original unscaled target values, the NN must be trained longer until

$$MSE_s = (c_1)^2 MSE_r \tag{7.12}$$

to reach a desired accuracy, MSE_r, on the original unscaled data set [Engelbrecht *et al.* 1995a].

The hyperbolic tangent will therefore result in faster training times than the sigmoid function, assuming the same initial conditions and training data.

The scaling process above is usually referred to as amplitude scaling, or min-max scaling. Min-max scaling preserves the relationships among the original data. Two other frequently used scaling methods are mean centering and variance scaling. To explain these two scaling methods, assume that $Z \in \mathbb{R}^{I \times P}$ is a matrix containing all input vectors such that input vectors are arranged as columns in Z, and $T \in \mathcal{R}^{K \times P}$ is the matrix of associated target vectors, arranged in column format. For the mean centering process, compute

$$\begin{aligned} \overline{Z}_i &= \sum_{p=1}^{P} Z_{i,p}/P \\ \overline{T}_k &= \sum_{p=1}^{P} T_{k,p}/P \end{aligned}$$

for all $i = 1, \cdots, I$ and $k = 1, \cdots, K$; $\overline{Z}_i$ is the average value for input z_i over all

the patterns, and $\overline{T}_k$ is the average target value for the k-th output unit over all patterns. Then,

$$\begin{aligned} Z^M_{i,p} &= Z_{i,p} - \overline{Z}_i \\ T^M_{k,p} &= T_{k,p} - \overline{T}_k \end{aligned}$$

for all $i = 1, \cdots, I$, $k = 1, \cdots, K$ and $p = 1, \cdots, P$; $Z^M_{i,p}$ is the scaled value of the input to unit z_i for pattern p, and $T^M_{k,p}$ is the corresponding scaled target value.

Variance scaling, on the other hand, computes for each row in each matrix the standard deviations (I deviations for matrix Z and K deviations for matrix T) over all P elements in the row. Let σ_{z_i} denote the standard deviation of row i of matrix Z, and σ_{t_k} is the standard deviation of row k of matrix T. Then,

$$\begin{aligned} Z^V_{i,p} &= \frac{Z_{i,p}}{\sigma_{z_i}} \\ T^V_{k,p} &= \frac{T_{k,p}}{\sigma_{t_k}} \end{aligned}$$

for all $i = 1, \cdots, I$, $k = 1, \cdots, K$ and $p = 1, \cdots, P$.

Mean centering and variance scaling can both be used on the same data set. Mean centering is, however, more appropriate when the data contains no biases, while variance scaling is appropriate when training data are measured with different units.

Both mean centering and variance scaling can be used in situations where the minimum and maximum values are unknown. Z-score normalization is another data transformation scheme which can be used in situations where the range of values is unknown. It is essentially a combination of mean centering and variance scaling, and is very useful when there are outliers in the data. For z-score normalization,

$$Z^{MV}_{i,p} = \frac{Z_{i,p} - \overline{Z}_i}{\sigma_{z_i}} \tag{7.13}$$

$$T^{MV}_{k,p} = \frac{T_{k,p} - \overline{T}_k}{\sigma_{t_k}} \tag{7.14}$$

$$\tag{7.15}$$

For some NN types, for example the LVQ, input data is preferred to be normalized to vectors of unit length. The values $z_{i,p}$ of each input parameter z_i are then normalized using

$$z'_{i,p} = \frac{z_{i,p}}{\sqrt{\sum_{i=1}^{I} z^2_{i,p}}} \tag{7.16}$$

The normalization above loses information on the absolute magnitude of the input parameters, since it requires the length of all input vectors (patterns) to be the

same. Input patterns with parameter values of different magnitudes are normalized to the same vector, e.g. vectors $(-1, 1, 2, 3)$ and $(-3, 3, 6, 9)$. Z-axis normalization is an alternative approach which preserves the absolute magnitude information of input patterns. Before the normalization step, input values are scaled to the range $[-1, 1]$. Input values are then normalized using

$$z'_{i,p} = \frac{z_{i,p}}{\sqrt{I}} \tag{7.17}$$

and adding an additional input unit z_0 to the NN, referred to as the synthetic parameter, with value

$$z_0 = \sqrt{1 - \frac{L^2}{I}} \tag{7.18}$$

where L is the Euclidean length of the input vector.

Noise Injection

For problems with a limited number of training patterns, controlled injection of noise helps to generate new training patterns. Provided that noise is sampled from a Normal distribution with a small variance and zero mean, it can be assumed that the resulting changes in the network output will have insignificant consequences [Holmström and Koistinen 1992]. Also, the addition of noise results in a convolutional smoothing of the target function, resulting in reduced training time and increased accuracy [Reed *et al.* 1995]. Engelbrecht used noise injection around decision boundaries to generate new training patterns for improved performance [Engelbrecht 2000].

Training Set Manipulation

Several researchers have developed techniques to control the order in which patterns are presented for learning. These techniques resulted in the improvement of training time and accuracy. A short summary of such training set manipulation techniques is given below.

Ohnishi, Okamoto and Sugie suggested a method called Selective Presentation where the original training set is divided into two training sets. One set contains typical patterns, and the other set contains confusing patterns [Ohnishi *et al.* 1990]. With "typical pattern" the authors mean a pattern far from decision boundaries, while "confusing pattern" refers to a pattern close to a boundary. The two training sets are created once before training. Generation of these training sets assumes prior knowledge about the problem, i.e. where in input space decision boundaries are. In many practical applications such prior knowledge is not available, thus limiting the applicability of this approach. The Selective Presentation strategy alternately presents the learner with typical and then confusing patterns.

Kohara developed Selective Presentation Learning specifically for forecasting applications [Kohara 1995]. Before training starts, the algorithm generates two training sets. The one set contains all patterns representing large next-day changes, while patterns representing small next-day changes are contained in the second set. Large-change patterns are then simply presented more often than small-change patterns (similar to Selective Presentation).

Cloete and Ludik have done extensive research on *training strategies*. Firstly, they proposed Increased Complexity Training where a NN first learns easy problems, and then the complexity of the problem to be learned is gradually increased [Cloete and Ludik 1993, Ludik and Cloete 1993]. The original training set is split into subsets of increasing complexity before training commences. A drawback of this method is that the complexity measure of training data is problem dependent, thus making the strategy unsuitable for some tasks. Secondly, Cloete and Ludik developed *incremental training strategies*, i.e. Incremental Subset Training [Cloete and Ludik 1994a] and Incremental Increased Complexity Training [Ludik and Cloete 1994]. In Incremental Subset Training, training starts on a random initial subset. During training, random subsets from the original training set are added to the actual training subset. Incremental Increased Complexity Training is a variation of Increased Complexity Training, where the complexity ranked order is maintained, but training is not done on each complete complexity subset. Instead, each complexity subset is further divided into smaller random subsets. Training starts on an initial subset of a complexity subset, and is incrementally increased during training. Finally, Delta Training Strategies were proposed [Cloete and Ludik 1994b]. With Delta Subset Training examples are ordered according to inter-example distance, e.g. Hamming or Euclidean distance. Different strategies of example presentations were investigated: smallest difference examples first, largest difference examples first, and alternating difference.

When vast quantities of data are available, training on all these data can be prohibitively slow, and may require reduction of the training set. The problem is which of the data should be selected for training. An easy strategy is to simply sample a smaller data set at each epoch using a uniform random number generator. Alternatively, a fast clustering algorithm can be used to group similar pattern together, and to sample a number of patterns from each cluster.

7.3.2 Weight Initialization

Gradient-based optimization methods, for example gradient descent, is very sensitive to the initial weight vectors. If the initial position is close to a local minimum, convergence will be fast. However, if the initial weight vector is on a flat area in the error surface, convergence is slow. Furthermore, large initial weight values have been shown to prematurely saturate units due to extreme output values with associated zero derivatives [Hush *et al.* 1991]. In the case of optimization algorithms such as

PSO and GAs, initialization should be uniformly over the entire search space to ensure that all parts of the search space are covered.

A sensible weight initialization strategy is to choose small random weights centered around 0. This will cause net input signals to be close to zero. Activation functions then output midrange values regardless of the values of input units. Hence, there is no bias toward any solution. Wessels and Barnard showed that random weights in the range $[\frac{-1}{\sqrt{fanin}}, \frac{1}{\sqrt{fanin}}]$ is a good choice, where $fanin$ is the number of connections leading to a unit [Wessels and Barnard 1992].

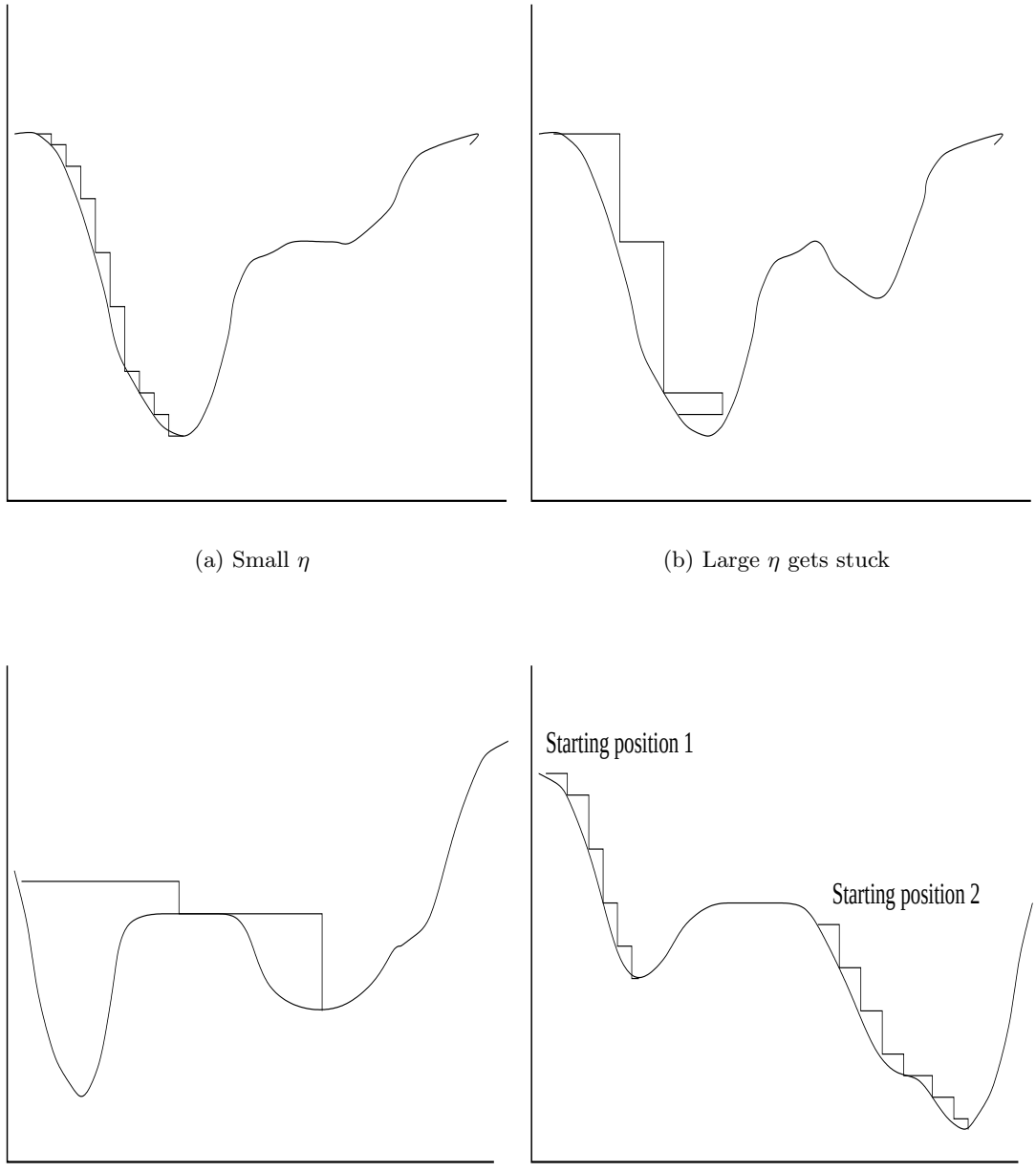
Why don't we just initialize all the weights to zero in the case of gradient-based optimization? This strategy will work only if the NN has just one hidden unit. For more than one hidden unit, all the units produce the same output, and thus make the same contribution to the approximation error. All the weights are therefore adjusted with the same value. Weights will remain the same irrespective of training time – hence, no learning takes place. Initial weight values of zero for PSO will also fail, since no velocity changes are made; therefore no weight changes. GAs, on the other hand, will work with initial zero weights if mutation is implemented.

7.3.3 Learning Rate and Momentum

The convergence speed of NNs is directly proportional to the learning rate η. Considering stochastic GD, the momentum term added to the weight updates also has as its objective improving convergence time.

Learning Rate

The learning rate controls the size of each step toward the minimum of the objective function. If the learning rate is too small, the weight adjustments are correspondingly small. More learning iterations are then required to reach a local minimum. However, the search path will closely approximate the gradient path. Figure 7.5(a) illustrates the effect of small η. On the other hand, large η will have large weight updates. Convergence will initially be fast, but the algorithm will eventually oscillate without reaching the minimum. It is also possible that too large a learning rate will cause "jumping" over a good local minimum proceeding toward a bad local minimum. Figure 7.5(b) illustrates the oscillating behavior, while Figure 7.5(c) illustrates how large learning rates may cause the network to overshoot a good minimum and get trapped in a bad local minimum. Small learning rates also have the disadvantage of being trapped in a bad local minimum as illustrated in Figure 7.5(d). The search path goes down the first local minimum, with no mechanism to move out of it toward the next, better minimum. Of course, all depends on the initial starting position. If the second initial point is used, the NN will converge to the better local minimum.



(a) Small η

(b) Large η gets stuck

(c) Large η overshoots

(d) Small η gets stuck

Figure 7.5: Effect of learning rate

But how do we choose the value of the learning rate? One approach is to find the optimal value of the learning rate through cross-validation, which is a lengthy process. An alternative is to select a small value (e.g. 0.1) and to increase the value if convergence is too slow, or to decrease it if the error does not decrease fast enough. Plaut *et al.* proposed that the learning rate should be inversely proportional to the fanin of a neuron [Plaut *et al.* 1986]. This approach has been theoretically justified through an analysis of the eigenvalue distribution of the Hessian matrix of the objective function [Le Cun *et al.* 1991].

Several heuristics have been developed to dynamically adjust the learning rate during training. One of the simplest approaches is to assume that each weight has a different learning rate η_{kj}. The following rule is then applied to each weight before that weight is updated: if the direction in which the error decreases at this weight change is the same as the direction in which it has been decreasing recently, then η_{kj} is increased; if not, η_{kj} is decreased [Jacobs 1988]. The direction in which the error decreases is determined by the sign of the partial derivative of the objective function with respect to the weight. Usually, the average change over a number of pattern presentations is considered and not just the previous adjustment.

An alternative is to use an annealing schedule to gradually reduce a large learning rate to a smaller value (refer to equation 4.22). This allows for large initial steps, and ensures small steps in the region of the minimum.

Of course more complex adaptive learning rate techniques have been developed, with elaborate theoretical analysis. The interested reader is referred to [Darken and Moody, Magoulas *et al.* 1997, Salomon and Van Hemmen 1996, Vogl *et al.* 1988].

Momentum

Stochastic learning, where weights are adjusted after each pattern presentation, has the disadvantage of fluctuating changes in the sign of the error derivatives. The network spends a lot of time going back and forth, unlearning what the previous steps have learned. Batch learning is a solution to this problem, since weight changes are accumulated and applied only after all patterns in the training set have been presented. Another solution is to keep with stochastic learning, and to add a momentum term. The idea of the momentum term is to average the weight changes, thereby ensuring that the search path is in the average downhill direction. The momentum term is then simply the previous weight change weighted by a scalar value α. If $\alpha = 0$, then the weight changes are not influenced by past weight changes. The larger the value of α the longer the change in steepest descent direction has to persevere to affect the direction in which weights are adjusted. A static value of 0.9 is usually used.

The optimal value of α can also be determined through cross-validation. Strategies have also been developed that use adaptive momentum rates, where each weight has a different momentum rate. Fahlman developed the schedule

$$\alpha_{kj}(t) = \frac{\frac{\partial \mathcal{E}}{\partial w_{kj}(t)}}{\frac{\partial \mathcal{E}}{\partial w_{kj}(t-1)} - \frac{\partial \mathcal{E}}{\partial w_{kj}(t)}} \tag{7.19}$$

This variation to the standard back-propagation algorithm is referred to as *quickprop* [Fahlman 1989]. Becker and Le Cun calculated the momentum rate as a function of the second-order error derivatives [Becker and Le Cun 1988]:

$$\alpha = (\frac{\partial^2 \mathcal{E}}{\partial w_{kj}^2})^{-1} \tag{7.20}$$

For more information on other approaches to adapt the momentum rate refer to [Orr and Leen 1993, Yu and Chen 1997].

7.3.4 Optimization Method

The optimization method used to determine weight adjustments has a large influence on the performance of NNs. While GD is a very popular optimization method, GD is plagued by slow convergence and susceptibility to local minima (as introduced and discussed in Section 3.2.2). Improvements of GD have been made to address these problems, for example, the addition of the momentum term. Also, second-order derivatives of the objective function have been used to compute weight updates. In doing so, more information about the structure of the error surface is used to direct weight changes. The reader is referred to [Battiti 1992, Becker and Le Cun 1988, Møller 1993]. Other approaches to improve NN training are to use global optimization algorithms instead of local optimization algorithms, for example simulated annealing [Rosen and Goodwin 1997], genetic algorithms [Engelbrecht and Ismail 1999, Janson and Frenzel 1993, Kuscu and Thornton 1994], particle swarm optimization algorithms [Corne *et al.* 1999, Eberhart *et al.* 1996, Engelbrecht and Ismail 1999, Van den Bergh 1999, Van den Bergh and Engelbrecht 2000], and LeapFrog optimization [Snyman 1982, Snyman 1983, Engelbrecht and Ismail 1999].

7.3.5 Architecture Selection

Referring to one of Ockham's statements, if several networks fit the training set equally well, then the simplest network (i.e. the network which has the smallest number of weights) will on average give the best generalization performance [Thodberg 1991]. This hypothesis has been investigated and confirmed by Sietsma

and Dow [Sietsma and Dow 1991]. A network with too many free parameters may actually memorize training patterns and may also accurately fit the noise embedded in the training data, leading to bad generalization. Overfitting can thus be prevented by reducing the size of the network through elimination of individual weights or units. The objective is therefore to balance the complexity of the network with goodness of fit of the true function. This process is referred to as *architecture selection*. Several approaches have been developed to select the optimal architecture, i.e. regularization, network construction (growing) and pruning. These approaches will be overviewed in more detail below.

Learning is not just perceived as finding the optimal weight values, but also finding the optimal architecture. However, it is not always obvious what is the best architecture. Finding the ultimate best architecture requires a search of all possible architectures. For large networks an exhaustive search is prohibitive, since the search space consists of 2^w architectures, where w is the total number of weights [Moody and Utans 1995]. Instead, heuristics are used to reduce the search space. A simple method is to train a few networks of different architecture and to choose the one which results in the lowest generalization error as estimated from the generalized prediction error (GPE) [Moody 1992, Moody 1994] or the Network Information Criterion (NIC) [Murata *et al.* 1991, Murata *et al.* 1994a, Murata *et al.* 1994b]. This approach is still expensive and requires many architectures to be investigated to reduce the possibility that the optimal model is not found. The NN architecture can alternatively be optimized by trial and error. An architecture is selected, and its performance is evaluated. If the performance is unacceptable, a different architecture is selected. This process continues until an architecture is found which produces an acceptable generalization error.

Other approaches to architecture selection are divided into three categories:

- **Regularization**: Neural network regularization involves the addition of a penalty term to the objective function to be minimized. In this case the objective function changes to

$$\mathcal{E} = \mathcal{E}_T + \lambda \mathcal{E}_C \tag{7.21}$$

  where $\mathcal{E}_T$ is the usual measure of data misfit, and $\mathcal{E}_C$ is a penalty term, penalizing network complexity (network size). The constant λ controls the influence of the penalty term. With the changed objective function, the NN now tries to find a locally optimal trade-off between data-misfit and network complexity. Neural network regularization has been studied rigorously by Girosi, Jones and Poggio [Girosi *et al.* 1995], and Williams [Williams 1995].

  Several penalty terms have been developed to reduce network size automatically during training. Weight decay, where $\mathcal{E}_C = \frac{1}{2}\sum w_i^2$, is intended to drive small weights to zero [Bös 1996, Hanson and Pratt 1989, Kamimura and Nakanishi 1994, Krogh and Hertz 1992]. It is a simple method

to implement, but suffers from penalizing large weights at the same rate as small weights. To solve this problem, Hanson and Pratt propose the hyperbolic and exponential penalty functions which penalize small weights more than large weights [Hanson and Pratt 1989]. Nowlan and Hinton developed a more complicated soft weight sharing, where the distribution of weight values is modeled as a mixture of multiple Gaussian distributions [Nowlan and Hinton 1992]. A narrow Gaussian is responsible for small weights, while a broad Gaussian is responsible for large weights. Using this scheme, there is less pressure on large weights to be reduced.

Weigend, Rumelhart and Huberman propose weight elimination where the penalty function $\mathcal{E}_C = \sum \frac{w_i^2/w_0^2}{1+w_i^2/w_0^2}$, effectively counts the number of weights [Weigend *et al.* 1991]. Minimization of this objective function will then minimize the number of weights. The constant w_0 is very important to the success of this approach. If w_0 is too small, the network ends up with a few large weights, while a large value results in many small weights. The optimal value for w_0 can be determined through cross-validation, which is not cost-effective.

Chauvin introduces a penalty term which measures the "energy spent" by the hidden units, where the energy is expressed as a function of the squared activation of the hidden units [Chauvin 1989, Chauvin 1990]. The aim is then to minimize the energy spent by hidden units, and in so doing, to eliminate unnecessary units.

Kamimura and Nakanishi show that, in an information theoretical context, weight decay actually minimizes entropy [Kamimura and Nakanishi 1994]. Entropy can also be minimized directly by including an entropy penalty term in the objective function [Kamimura 1993]. Minimization of entropy means that the information about input patterns is minimized, thus improving generalization. For this approach entropy is defined with respect to hidden unit activity. Schittenkopf, Deco and Brauer also propose an entropy penalty term and show how it reduces complexity and avoids overfitting [Schittenkopf *et al.* 1997].

Yasui develops penalty terms to make minimal and joint use of hidden units by multiple outputs [Yasui 1997]. Two penalty terms are added to the objective function to control the evolution of hidden-to-output weights. One penalty causes weights leading into an output unit to prevent another from growing, while the other causes weights leaving a hidden unit to support another to grow.

While regularization models are generally easy to implement, the value of the constant λ in equation (7.21) may present problems. If λ is too small, the penalty term will have no effect. If λ is too large, all weights might be driven to zero. Regularization therefore requires a delicate balance between the normal error term and the penalty term. Another disadvantage of penalty terms is that they tend to create additional local minima [Hanson and Pratt 1989], increasing the possibility of converging to a bad local minimum. Penalty terms

also increase training time due to the added calculations at each weight update. In a bid to reduce this complexity, Finnoff, Hergert and Zimmermann show that the performance of penalty terms is greatly enhanced if they are introduced only after overfitting is observed [Finnoff *et al.* 1993].

- **Network construction (growing)**: Network construction algorithms start training with a small network and incrementally add hidden units during training when the network is trapped in a local minimum [Fritzke 1995, Hirose *et al.* 1991, Hüning 1993, Kwok and Yeung 1995]. A small network forms an approximate model of a subset of the training set. Each new hidden unit is trained to reduce the current network error – yielding a better approximation. Crucial to the success of construction algorithms is effective criteria to trigger when to add a new unit, when to stop the growing process, where and how to connect the new unit to the existing architecture, and how to avoid restarting training. If these issues are treated on an *ad hoc* basis, overfitting may occur and training time may be increased.

- **Network pruning**: Neural network pruning algorithms start with an oversized network and remove unnecessary network parameters, either during training or after convergence to a local minimum. Network parameters that are considered for removal are individual weights, hidden units and input units. The decision to prune a network parameter is based on some measure of parameter relevance or significance. A relevance is computed for each parameter and a pruning heuristic is used to decide when a parameter is considered as being irrelevant or not. A large initial architecture allows the network to converge reasonably quickly, with less sensitivity to local minima and the initial network size. Larger networks have more functional flexibility, and are guaranteed to learn the input-output mapping with the desired degree of accuracy. Due to the larger functional flexibility, pruning weights and units from a larger network may give rise to a better fit of the underlying function, hence better generalization [Moody 1994].

A more elaborate discussion of pruning techniques is given next, with the main objective of presenting a flavor of the techniques available to prune NN architectures. For more detailed discussions, the reader is referred to the given references. The first results in the quest to find a solution to the architecture optimization problem were the derivation of theoretical limits on the number of hidden units to solve a particular problem [Baum 1988, Cosnard *et al.* 1992, Kamruzzaman *et al.* 1992, Sakurai 1992, Sartori and Antsaklis 1991]. However, these results are based on unrealistic assumptions about the network and the problem to be solved. Also, they usually apply to classification problems only. While these limits do improve our understanding of the relationship between architecture and training set characteristics, they do not predict the correct number of hidden units for a general class of problems.

Recent research concentrated on the development of more efficient pruning techniques to solve the architecture selection problem. Several different approaches to pruning have been developed. This chapter groups these approaches in the following general classes: intuitive methods, evolutionary methods, information matrix methods, hypothesis testing methods and sensitivity analysis methods.

- **Intuitive pruning techniques**: Simple intuitive methods based on weight values and unit activation values have been proposed by Hagiwara [Hagiwara 1993]. The *goodness factor* G_i^l of unit i in layer l, $G_i^l = \sum_p \sum_j (w_{ji}^l o_i^l)^2$, where the first sum is over all patterns, and o_i^l is the output of the unit, assumes that an important unit is one which excites frequently and has large weights to other units. The *consuming energy*, $E_i^l = \sum_p \sum_j w_{ji}^l o_j^{l+1} o_j^l$, additionally assumes that unit i excites the units in the next layer. Both methods suffer from the flaw that when an unit's output is more frequently 0 than 1, that unit might be considered as being unimportant, while this is not necessarily the case. Magnitude-based pruning assumes that small weights are irrelevant [Hagiwara 1993, Lim and Ho 1994]. However, small weights may be of importance, especially compared to very large weights which cause saturation in hidden and output units. Also, large weights (in terms of their absolute value) may cancel each other out.

- **Evolutionary pruning techniques**: The use of genetic algorithms (GA) to prune NNs provides a biologically plausible approach to pruning [Kuscu and Thornton 1994, Reed 1993, Whitley and Bogart 1990, White and Ligomenides 1993]. Using GA terminology, the population consists of several pruned versions of the original network, each needed to be trained. Differently pruned networks are created by the application of mutation, reproduction and cross-over operators. These pruned networks "compete" for survival, being awarded for using fewer parameters and for improving generalization. GA NN pruning is thus a time-consuming process.

- **Information matrix pruning techniques**: Several researchers have used approximations to the Fisher information matrix to determine the optimal number of hidden units and weights. Based on the assumption that outputs are linearly activated, and that least squares estimators satisfy asymptotic normality, Cottrell *et al.* compute the relevance of a weight as a function of the information matrix, approximated by [Cottrell *et al.* 1994]

$$\mathcal{I} = \frac{1}{P} \sum_{p=1}^{P} \frac{\partial \mathcal{F}_{NN}}{\partial w} (\frac{\partial \mathcal{F}_{NN}}{\partial w})^T \tag{7.22}$$

  Weights with a low relevance are removed.

  Hayashi [Hayashi 1993], Tamura *et al.* [Tamura *et al.* 1993], Xue *et al.* [Xue *et al.* 1990] and Fletcher *et al.* [Fletcher *et al.* 1998] use Singular Value

Decomposition (SVD) to analyze the hidden unit activation covariance matrix to determine the optimal number of hidden units. Based on the assumption that outputs are linearly activated, the *rank* of the covariance matrix is the optimal number of hidden units (also see [Fujita 1992]). SVD of this information matrix results in an eigenvalue and eigenvector decomposition where low eigenvalues correspond to irrelevant hidden units. The rank is the number of non-zero eigenvalues. Fletcher, Katkovnik, Steffens and Engelbrecht use the SVD of the conditional Fisher information matrix, as given in equation (7.22), together with likelihood-ratio tests to determine irrelevant hidden units [Fletcher *et al.* 1998]. In this case the conditional Fisher information matrix is restricted to weights between the hidden and output layer only, whereas previous techniques are based on all network weights. Each iteration of the pruning algorithm identifies exactly which hidden units to prune.

Principal Component Analysis (PCA) pruning techniques have been developed that use the SVD of the Fisher information matrix to find the principal components (relevant parameters) [Levin *et al.* 1994, Kamimura 1993, Schittenkopf *et al.* 1997, Takahashi 1993]. These principal components are linear transformations of the original parameters, computed from the eigenvectors obtained from a SVD of the information matrix. The result of PCA is the orthogonal vectors on which variance in the data is maximally projected. Non-principal components/parameters (parameters that do not account for data variance) are pruned. Pruning using PCA is thus achieved through projection of the original w-dimensional space onto a $w^{'}$-dimensional linear subspace ($w^{'} < w$) spanned by the eigenvectors of the data's correlation or covariance matrix corresponding to the largest eigenvalues.

- **Hypothesis testing techniques**: Formal statistical hypothesis tests can be used to test the statistical significance of a subset of weights, or a subset of hidden units. Steppe, Bauer and Rogers [Steppe *et al.* 1996] and Fletcher, Katkovnik, Steffens and Engelbrecht [Fletcher *et al.* 1998] use the *likelihood-ratio test statistic* to test the null hypothesis that a subset of weights is zero. Weights associated with a hidden unit are tested to see if they are statistically different from zero. If these weights are not statistically different from zero, the corresponding hidden unit is pruned.

  Belue and Bauer propose a method that injects a noisy input parameter into the NN model, and then use statistical tests to decide if the significances of the original NN parameters are higher than that of the injected noisy parameter [Belue and Bauer 1995]. Parameters with lower significances than the noisy parameter are pruned.

  Similarly, Prechelt [Prechelt 1995] and Finnoff *et al.* [Finnoff *et al.* 1993] test the assumption that a weight becomes zero during the training process. This approach is based on the observation that the distribution of weight values is roughly normal. Weights located in the left tail of this distribution are

removed.

- **Sensitivity analysis pruning techniques**: Two main approaches to sensitivity analysis exist, namely with regard to the objective function and with regard to the NN output function. Both sensitivity analysis with regard to the objective function and sensitivity analysis with regard to the NN output function resulted in the development of a number of pruning techniques. Possibly the most popular of these are OBD [Le Cun 1990] and its variants, OBS [Hassibi and Stork 1993, Hassibi *et al.* 1994] and Optimal Cell Damage (OCD) [Cibas *et al.* 1996]. A parameter saliency measure is computed for each parameter, indicating the influence small perturbations to the parameter have on the approximation error. Parameters with a low saliency are removed. These methods are time-consuming due to the calculation of the Hessian matrix. Buntine and Weigend [Buntine and Weigend 1994] and Bishop [Bishop 1992] derived methods to simplify the calculation of the Hessian matrix in a bid to reduce the complexity of these pruning techniques. In OBD, OBS and OCD, sensitivity analysis is performed with regard to the training error. Pedersen, Hanson and Larsen [Pedersen *et al.* 1996] and Burrascano [Burrascano 1993] develop pruning techniques based on sensitivity analysis with regard to the generalization error. Other objective function sensitivity analysis pruning techniques have been developed by Mozer and Smolensky [Mozer and Smolensky 1989] and Moody and Utans [Moody and Utans 1995].

  NN output sensitivity analysis pruning techniques have been developed that are less complex than objective function sensitivity analysis, and that do not rely on simplifying assumptions. Zurada, Malinowski and Cloete introduced output sensitivity analysis pruning of input units [Zurada *et al.* 1994], further investigated by Engelbrecht, Cloete and Zurada [Engelbrecht *et al.* 1995b]. Engelbrecht and Cloete extended this approach to also prune irrelevant hidden units [Engelbrecht and Cloete 1996, Engelbrecht *et al.* 1999, Engelbrecht 2001].

  A similar approach to NN output sensitivity analysis was followed by Dorizzi *et al.* [Dorizzi *et al.* 1996] and Czernichow [Czernichow 1996] to prune parameters of a Radial Basis Function (RBF) NN.

The aim of all architecture selection algorithms is to find the smallest architecture that accurately fits the underlying function. In addition to improving generalization performance and avoiding overfitting (as discussed earlier), smaller networks have the following advantages. Once an optimized architecture has been found, the cost of forward calculations is significantly reduced, since the cost of computation grows almost linearly with the number of weights. From the generalization limits overviewed in section 7.3.7, the number of training patterns required to achieve a certain generalization performance is a function of the network architecture. Smaller networks therefore require less training patterns. Also, the knowledge embedded in smaller

networks is more easily described by a set of simpler rules. Viktor, Engelbrecht and Cloete show that the number of rules extracted from smaller networks is less for pruned networks than that extracted from larger networks [Viktor *et al.* 1995]. They also show that rules extracted from smaller networks contain only relevant clauses, and that the combinatorics of the rule extraction algorithm is significantly reduced. Furthermore, for smaller networks the function of each hidden unit is more easily visualized. The complexity of decision boundary detection algorithms is also reduced.

With reference to the bias/variance decomposition of the MSE function [Geman *et al.* 1992], smaller network architectures reduce the variance component of the MSE. NNs are generally plagued by high variance due to the limited training set sizes. This variance is reduced by introducing bias through minimization of the network architecture. Smaller networks are biased because the hypothesis space is reduced, thus limiting the available functions that can fit the data. The effects of architecture selection on the bias/variance trade-off have been studied by Gedeon, Wong and Harris [Gedeon *et al.* 1995].

7.3.6 Adaptive Activation Functions

The performance of NNs can be improved by allowing activation functions to change dynamically according to the characteristics of the training data. One of the first techniques to use adaptive activations functions was developed by Zurada [Zurada 1992], where the slope of the sigmoid activation function is learned together with the weights. A slope parameter λ is kept for each hidden and output unit. The lambda-learning algorithm of Zurada was extended by Engelbrecht *et al.* where the sigmoid function is given as [Engelbrecht *et al.* 1995a]

$$f(net, \lambda, \gamma) = \frac{\gamma}{1 + e^{-\lambda net}} \tag{7.23}$$

where λ is the slope of the function and γ the maximum range. Engelbrecht *et al.* developed learning equations to also learn the maximum ranges of the sigmoid functions, thereby performing automatic scaling. By using gamma-learning, it is not necessary to scale target values to the range $(0, 1)$. The effect of changing the slope and range of the sigmoid function is illustrated in Figure 7.6.

A general algorithm is given below to illustrate the differences between standard GD learning (referred to as delta learning) and the lambda and gamma learning variations. (Note that although the momentum terms are omitted below, a momentum term is usually used for the weight, lambda and gamma updates.)

Begin: Given P pairs of training patterns consisting of inputs and targets $\{(\vec{z}_1, \vec{t}_1), (\vec{z}_2, \vec{t}_2), \ldots, (\vec{z}_p, \vec{t}_p)\}$ where $\vec{z}_i$ is $(I \times 1)$, $\vec{t}_k$ is $(K \times 1)$ and $i = 1, \ldots, P$; $\vec{y}$ is $(J \times 1)$ and $\vec{o}$ is $(K \times 1)$.

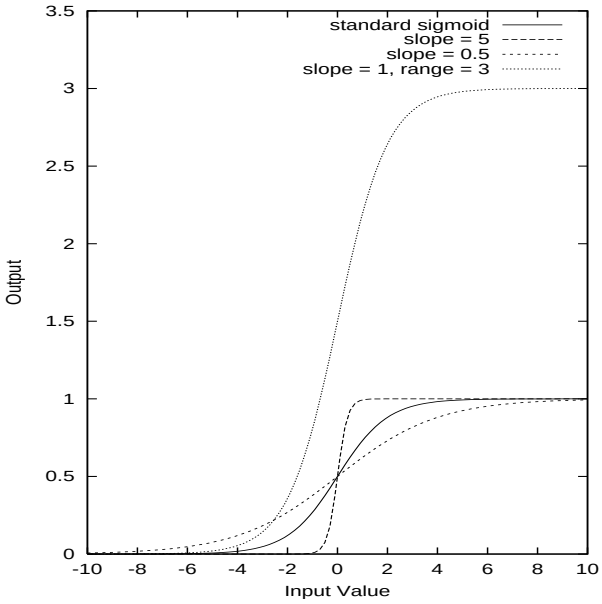


Figure 7.6: Adaptive sigmoid

Step 1: Choose the values of the learning rates η_1, η_2 and η_3 according to the learning rule:

Delta learning rule	$\eta_1 > 0,\ \eta_2 = 0,\ \eta_3 = 0$
Lambda learning rule	$\eta_1 > 0,\ \eta_2 > 0,\ \eta_3 = 0$
Gamma learning rule	$\eta_1 > 0,\ \eta_2 = 0,\ \eta_3 > 0$
Lambda-gamma learning rule	$\eta_1 > 0,\ \eta_2 > 0,\ \eta_3 > 0$

Choose an acceptable training error E_{max}. Weights W $(K \times J)$ and V $(J \times I)$ are initialized to small random values. Initialize the number of cycles q and the training pairs counter p to $q = 1,\ p = 1$. Let $E = 0$ and initialize the steepness and range coefficients

$$\lambda_{y_j} = \gamma_{y_j} = 1 \quad \forall\ j = 1, \ldots, J \quad \text{and} \quad \lambda_{o_k} = \gamma_{o_k} = 1 \quad \forall\ k = 1, \ldots, K$$

Step 2: Start training. Input is presented and the layers' outputs are computed using $f(\gamma, \lambda, net)$ as in equation (7.23):

$$\vec{z} = \vec{z}_p, \quad \vec{t} = \vec{t}_p \quad \text{and} \quad y_j = f(\gamma_{y_j}, \lambda_{y_j}, \vec{v}_j^t \vec{z}) \quad \forall\ j = 1, \ldots, J$$

where $\vec{v}_j$, a column vector, is the j-th row of V and

$$o_k = f(\gamma_{o_k}, \lambda_{o_k}, \vec{w}_k^t \vec{y}) \quad \forall\ k = 1, \ldots, K$$

where $\vec{w}_k$, a column vector, is the k-th row of W.

Step 3: The error value is computed:

$$E = E + \frac{1}{2}(t_k - o_k)^2 \quad \forall\, k = 1, \ldots, K$$

Step 4: The error signal vectors $\vec{\delta}_o$ $(K \times 1)$ and $\vec{\delta}_y$ $(J \times 1)$ of both the output and hidden layers are computed

$$\delta_{o_k} = -\frac{\lambda_{o_k}}{\gamma_{o_k}}(t_k - o_k)o_k(\gamma_{o_k} - o_k) \quad \forall\, k = 1, \ldots, K$$

$$\delta_{y_j} = \frac{\lambda_{y_j}}{\gamma_{y_j}} y_j(\gamma_{y_j} - y_j) \sum_{k=1}^{K} \delta_{o_k} w_{kj} \quad \forall\, j = 1, \ldots . J$$

Step 5: Output layer weights and gains are adjusted:

$$w_{kj} = w_{kj} + \eta_1 \delta_{o_k} y_j \qquad \lambda_{o_k} = \lambda_{o_k} + \eta_2 \delta_{o_k} \frac{net_{o_k}}{\lambda_{o_k}} \qquad \gamma_{o_k} = \gamma_{o_k} + \eta_3 (t_k - o_k) \frac{1}{\gamma_{o_k}} o_k$$

for all $k = 1, \ldots, K$ and $j = 1, \ldots, J$.

Step 6: Hidden layer weights and gains are adjusted:

$$v_{ji} = v_{ji} + \eta_1 \delta_{y_j} z_i \qquad \lambda_{y_j} = \lambda_{y_j} + \eta_2 \frac{1}{\lambda_{y_j}} \delta_{y_j} net_{y_j}$$

$$\gamma_{y_j} = \gamma_{y_j} + \eta_3 \frac{1}{\gamma_{y_j}} f(\gamma_{y_j}, \lambda_{y_j}, net_{y_j}) \sum_{k=1}^{K} \delta_{o_k} w_{kj}$$

for all $j = 1, \ldots, J$ and $i = 1, \ldots, I$.

Step 7: If $p < P$ then let $p = p + 1$ and go to Step 2; otherwise go to Step 8.

Step 8: One training cycle is completed. If $E < E_{max}$ then terminate the training session. Output the cycle counter q and error E; otherwise let $E = 0,\ p = 1,\ q = q + 1$ and initiate a new training cycle by going to Step 2.

7.3.7 Active Learning

Ockham's razor states that unnecessarily complex models should not be preferred to simpler ones – a very intuitive principle [MacKay 1992, Thodberg 1991]. A neural network (NN) model is described by the network weights. Model selection in NNs consists of finding a set of weights that best performs the learning task. In this sense, the data, and not just the architecture should be viewed as part of the NN model, since the data is instrumental in finding the "best" weights. Model selection is then viewed as the process of designing an optimal NN architecture as well as the

implementation of techniques to make optimal use of the available training data. Following from the principle of Ockham's razor is a preference then for both simple NN architectures and optimized training data. Usually, model selection techniques address only the question of which architecture best fits the task.

Standard error back-propagating NNs are passive learners. These networks passively receive information about the problem domain, randomly sampled to form a fixed size training set. Random sampling is believed to reproduce the density of the true distribution. However, more gain can be achieved if the learner is allowed to use current attained knowledge about the problem to guide the acquisition of training examples. As passive learner, a NN has no such control over what examples are presented for learning. The NN has to rely on the teacher (considering supervised learning) to present informative examples.

The generalization abilities and convergence time of NNs are greatly influenced by the training set size and distribution: Literature has shown that to generalize well, the training set must contain enough information to learn the task. Here lies one of the problems in model selection: the selection of concise training sets. Without prior knowledge about the learning task, it is very difficult to obtain a representative training set. Theoretical analysis provide a way to compute worst-case bounds on the number of training examples needed to ensure a specified level of generalization. A widely used theorem concerns the Vapnik-Chervonenkis (VC) dimension [Abu-Mostafa 1989, Abu-Mostafa 1993, Baum and Haussler 1989, Cohn and Tesauro 1991, Hole 1996, Opper 1994]. This theorem states that the generalization error $\mathcal{E}_G$ of a learner with VC-dimension d_{VC} trained on P_T random examples will, with high confidence, be no worse than a limit of order d_{VC}/P_T. For NN learners, the total number of weights in a one hidden layer network is used as an estimate of the VC-dimension. This means that the appropriate number of examples to ensure an $\mathcal{E}_G$ generalization is approximately the number of weights divided by $\mathcal{E}_G$.

The VC-dimension provides overly pessimistic bounds on the number of training examples, often leading to an overestimation of the required training set size [Cohn and Tesauro 1991, Gu and Takahashi 1997, Opper 1994, Röbel 1994, Zhang 1994]. Experimental results have shown that acceptable generalization performances can be obtained with training set sizes much less than that specified by the VC-dimension [Cohn and Tesauro 1991, Röbel 1994]. Cohn and Tesauro show that for experiments conducted, the generalization error decreases exponentially with the number of examples, rather than the $1/P_T$ result of the VC bound [Cohn and Tesauro 1991]. Experimental results by Lange and Männer show that more training examples do not necessarily improve generalization [Lange and Männer 1994]. In their paper, Lange and Männer introduce the notion of a critical training set size. Through experimentation they found that examples beyond this critical size do not improve generalization, illustrating that an excess patterns have no real gain. This critical training set size is problem dependent.

While enough information is crucial to effective learning, too large training set sizes may be of disadvantage to generalization performance and training time [Lange and Zeugmann 1996, Zhang 1994]. Redundant training examples may be from uninteresting parts of input space, and do not serve to refine learned weights – it only introduces unnecessary computations, thus increasing training time. Furthermore, redundant examples might not be equally distributed, thereby biasing the learner.

The ideal, then, is to implement structures to make optimal use of available training data. That is, to select for training only informative examples, or to present examples in a way to maximize the decrease in training and generalization error. To this extent, active learning algorithms have been developed.

Cohn, Atlas and Ladner define *active learning* (also referred to in the literature as example selection, sequential learning, query-based learning) *as any form of learning in which the learning algorithm has some control over what part of the input space it receives information from* [Cohn *et al.* 1994]. An active learning strategy allows the learner to dynamically select training examples, during training, from a candidate training set as received from the teacher (supervisor). The learner capitalizes on current attained knowledge to select examples from the candidate training set that are most likely to solve the problem, or that will lead to a maximum decrease in error. Rather than passively accepting training examples from the teacher, the network is allowed to use its current knowledge about the problem to have some deterministic control over which training examples to accept, and to guide the search for informative patterns. By adding this functionality to a NN, the network changes from a passive learner to an active learner.

Figure 7.7 illustrates the difference between active learning and passive learning.

With careful dynamic selection of training examples, shorter training times and better generalization may be obtained. Provided that the added complexity of the example selection method does not exceed the reduction in training computations (due to a reduction in the number of training patterns), training time will be reduced [Hunt and Deller 1995, Sung and Niyogi 1996, Zhang 1994]. Generalization can potentially be improved, provided that selected examples contain enough information to learn the task. Cohn [Cohn 1994] and Cohn, Atlas and Ladner [Cohn *et al.* 1994] show through average case analysis that the expected generalization performance of active learning is significantly better than passive learning. Seung, Opper and Sompolinsky [Seung *et al.* 1992], Sung and Niyogi [Sung and Niyogi 1996] and Zhang [Zhang 1994] report similar improvements. Results presented by Seung, Opper and Sompolinsky indicate that generalization error decreases more rapidly for active learning than for passive learning [Seung *et al.* 1992].

Two main approaches to active learning can be identified, i.e. *incremental learning* and *selective learning*. Incremental learning starts training on an initial subset of

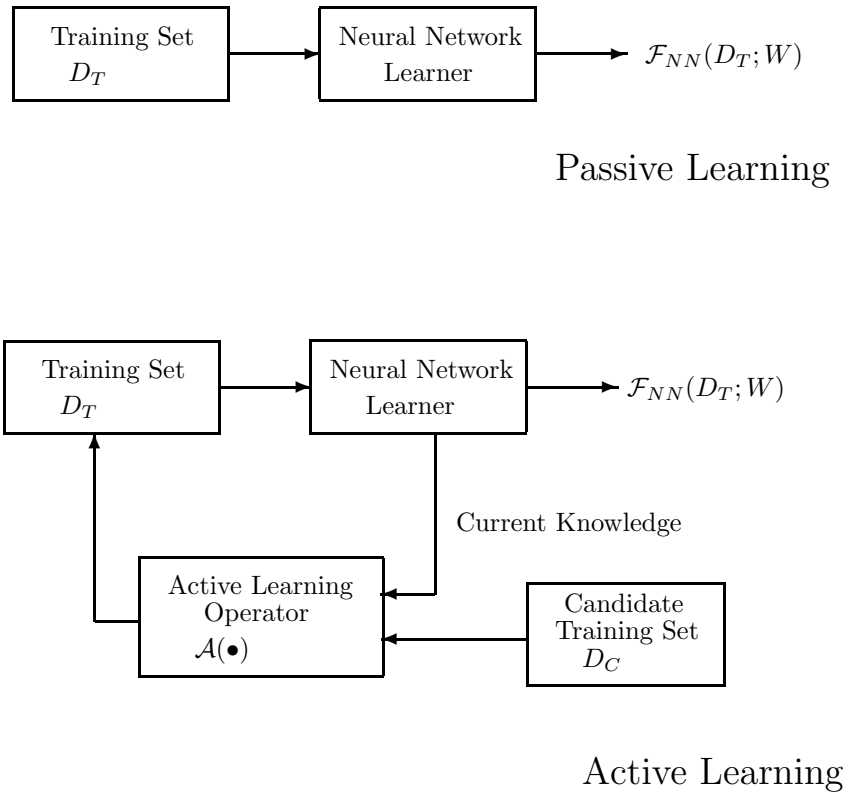


Figure 7.7: Passive vs active learning

a candidate training set. During training, at specified selection intervals (e.g. after a specified number of epochs, or when the error on the current training subset no longer decreases), further subsets are selected from the candidate examples using some criteria or heuristics, and added to the training set. The training set consists of the union of all previously selected subsets, while examples in selected subsets are removed from the candidate set. Thus, as training progresses, the size of the candidate set decreases while the size of the actual training set grows. Note that this chapter uses the term *incremental learning* to denote data selection, and should not be confused with the NN architecture selection growing approach. The term *NN growing* is used in this chapter to denote the process of finding an optimal architecture starting with too few hidden units and adding units during training.

In contrast to incremental learning, selective learning selects at each selection interval a new training subset from the original candidate set. Selected patterns are not removed from the candidate set. At each selection interval, all candidate patterns have a chance to be selected. The subset is selected and used for training until some convergence criteria on the subset is met (e.g. a specified error limit on the subset is reached, the error decrease per iteration is too small, the maximum number of epochs allowed on the subset is exceeded). A new training subset is then selected for

the next training period. This process repeats until the NN is trained to satisfaction.

The main difference between these two approaches to active learning is that no examples are discarded by incremental learning. In the limit, all examples in the candidate set will be used for training. With selective learning, training starts on all candidate examples, and uninformative examples are discarded as training progresses.

Selective Learning

Not much research has been done in selective learning. Hunt and Deller developed Selective Updating, where training starts on an initial candidate training set [Hunt and Deller 1995]. Patterns that exhibit a high influence on weights, i.e. patterns that cause the largest changes in weight values, are selected from the candidate set and added to the training set. Patterns that have a high influence on weights are selected at each epoch by calculating the effect that patterns have on weight estimates. These calculations are based on matrix perturbation theory, where an input pattern is viewed as a perturbation of previous patterns. If the perturbation is expected to cause large changes to weights, the corresponding pattern is included in the training set. The learning algorithm does use current knowledge to select the next training subset, and training subsets may differ from epoch to epoch. Selective Updating has the drawback of assuming uncorrelated input units, which is often not the case for practical applications.

Another approach to selective learning is simply to discard those patterns that have been classified correctly [Barnard 1991]. The effect of such an approach is that the training set will include those patterns that lie close to decision boundaries. If the candidate set contains outlier patterns, these patterns will, however, also be selected. This error selection approach therefore requires a robust estimator (objective function) to be used in the case of outliers.

Engelbrecht *et al.* developed a selective learning approach for classification problems where sensitivity analysis is used to locate patterns close to decision boundaries [Engelbrecht and Cloete 1998a, Engelbrecht and Cloete 1998b, Engelbrecht 2001b]. Only those patterns that are close to a decision boundary are selected for training. The algorithm resulted in substantial reductions in the number of learning calculations due to reductions in the training set size, while either maintaining performance as obtained from learning from all the training data, or improving performance.

Incremental learning

Research on incremental learning is more abundant than for selective learning. Most current incremental learning techniques have their roots in information theory, adapting Fedorov's optimal experiment design for NN learning [Cohn 1994, Fukumizu 1996, MacKay 1992, Plutowski and White 1993, Sung and Niyogi 1996]. The different information theoretic incremental learning algorithms are very similar, and differ only in whether they consider only bias, only variance, or both bias and variance terms in their selection criteria.

Cohn developed neural network Optimal Experiment Design (OED), where the objective is to select at each iteration a new pattern from a candidate set which minimizes the expectation of the mean squared error (MSE) [Cohn 1994]. This is achieved by minimizing output variance as estimated from the Fisher information matrix [Cohn 1994, Cohn *et al.* 1996]. The model assumes an unbiased estimator and considers only the minimization of variance. OED is computationally very expensive because it requires the calculation of the inverse of the information matrix.

MacKay proposed similar Information-Based Objective Functions for active learning, where the aim is to maximize the expected information gain by maximizing the change in Shannon entropy when new patterns are added to the actual training set, or by maximizing cross-entropy gain [MacKay 1992]. Similar to OED, the maximization of information gain is achieved by selecting patterns that minimize the expected MSE. Information-Based Objective Functions also ignore bias, by minimizing only variance. The required inversion of the Hessian matrix makes this approach computationally expensive.

Plutowski and White proposed selecting patterns that minimize the Integrated Squared Bias (ISB) [Plutowski and White 1993]. At each iteration, a new pattern is selected from a candidate set that maximizes the change, ΔISB, in the ISB. In effect, the patterns with error gradient most highly correlated with the error gradient of the entire set of patterns is selected. A noise-free environment is assumed and variance is ignored. Drawbacks of this method are the need to calculate the inverse of a Hessian matrix, and the assumption that the target function is known.

Sung and Niyogi proposed an information theoretic approach to active learning that considers both bias and variance [Sung and Niyogi 1996]. The learning goal is to minimize the expected misfit between the target function and the approximated function. The patterns that minimizes the expected squared difference between the target and approximated function are selected to be included in the actual training set. In effect, the net amount of information gained with each new pattern is then maximized. No assumption is made about the target function. This technique is computationally expensive, since it requires computations over two expectations, i.e. the *a-posteriori* distribution over function space, and the *a-posteriori* distribution over the space of targets one would expect given a candidate sample location.

One drawback of the incremental learning algorithms summarized above is that they rely on the inversion of an information matrix. Fukumizu showed that, in relation to pattern selection to minimize the expected MSE, the Fisher information matrix may be singular [Fukumizu 1996]. If the information matrix is singular, the inverse of that matrix may not exist. Fukumizu continues to show that the information matrix is singular if and only if the corresponding NN contains redundant units. Thus, the information matrix can be made non-singular by removing redundant hidden units. Fukumizu developed an algorithm that incorporates an architecture reduction algorithm with a pattern selection algorithm. This algorithm is complex due to the inversion of the information matrix at each selection interval, but ensures a non-singular information matrix.

Approximations to the information theoretical incremental learning algorithms can be used. Zhang shows that information gain is maximized when a pattern is selected whose addition leads to the greatest decrease in MSE [Zhang 1994]. Zhang developed Selective Incremental Learning where training starts on an initial subset which is increased during training by adding additional subsets, where each subset contains those patterns with largest errors. Selective Incremental Learning has a very low computational overhead, but is negatively influenced by outlier patterns since these patterns have large errors.

Dynamic Pattern Selection, developed by Röbel [Röbel 1994], is very similar to Zhang's Selective Incremental Learning. Röbel defines a generalization factor on the current training subset, expressed as $\mathcal{E}_G/\mathcal{E}_T$ where $\mathcal{E}_G$ and $\mathcal{E}_T$ are the MSE of the test set and the training set respectively. As soon as the generalization factor exceeds a certain threshold, patterns with highest errors are selected from the candidate set and added to the actual training set. Testing against the generalization factor prevents overfitting of the training subset. A low overhead is involved.

Very different from the methods described previously are incremental learning algorithms for classification problems, where decision boundaries are utilized to guide the search for optimal training subsets. Cohn, Atlas and Ladner developed Selective Sampling, where patterns are sampled only within a *region of uncertainty* [Cohn *et al.* 1994]. Cohn *et al.* proposed an SG-network (most specific/most general network) as an approach to compute the region of uncertainty. Two separate networks are trained: one to learn a "most specific" concept s consistent with the given training data, and the other to learn a "most general" concept, g. The region of uncertainty is then all patterns p such that $s(p) \neq g(p)$. In other words, the region of uncertainty encapsulates all those patterns for which s and g present a different classification. A new training pattern is selected from this region of uncertainty and added to the training set. After training on the new training set, the region of uncertainty is recalculated, and another pattern is sampled according to some distribution defined over the uncertainty region – a very expensive approach. To reduce complexity, the algorithm is changed to select patterns in batches, rather than individually. An initial pattern subset is drawn, the network is trained on this

subset, and a new region of uncertainty is calculated. Then, a new distribution is defined over the region of uncertainty that is zero outside this region. A next subset is drawn according to the new distribution and added to the training set. The process repeats until convergence is reached.

Query-Based Learning, developed by Hwang, Choi, Oh and Marks differs from Selective Sampling in that Query-Based Learning generates new training data in the region of uncertainty [Hwang *et al.* 1991]. The aim is to increase the steepness of the boundary between two distinct classes by narrowing the regions of ambiguity. This is accomplished by inverting the NN output function to compute decision boundaries. New data in the vicinity of boundaries are then generated and added to the training set.

Seung, Opper and Sompolinsky proposed Query by Committee [Seung *et al.* 1992]. The optimal training set is built by selecting one pattern at a time from a candidate set based on the principle of maximal disagreement among a committee of learners. Patterns classified correctly by half of the committee, but incorrectly by the other half, are included in the actual training set. Query by Committee is time-consuming due to the simultaneous training of several networks, but will be most effective for ensemble networks.

Engelbrecht *et al.* developed an incremental learning algorithm where sensitivity analysis is used to locate the most informative patterns. The most informative patterns are viewed as those patterns in the midrange of the sigmoid activation function [Engelbrecht and Cloete 1999]. Since these patterns have the largest derivatives of the output with respect to inputs, the algorithm incrementally selects from a candidate set of patterns those patterns that have the largest derivatives. Substantial reductions in computational complexity are achieved using this algorithm, with improved accuracy.

The incremental learning algorithms reviewed in this section all make use of the NN learner's current knowledge about the learning task to select those patterns that are most informative. These algorithms start with an initial training set, which is increased during training by adding a single informative pattern, or a subset of informative patterns.

In general, active learning is summarized by the following algorithm:

1. Initialize the NN architecture. Construct an initial training subset D_{S_0} from the candidate set D_C. Initialize the current training set $D_T \leftarrow D_{S_0}$.

2. Repeat

   (a) Repeat
   Train the NN on training subset D_T

until convergence on the current training subset D_T is reached to produce the function $\mathcal{F}_{NN}(D_T; W)$.

(b) Apply the active learning operator to generate a new subset D_{S_s} at subset selection interval τ_s, using either

$$D_{S_s} \leftarrow \mathcal{A}^-(D_C, \mathcal{F}_{NN}(D_T; W)), \quad D_T \leftarrow D_{S_s}$$

for selective learning, or

$$\begin{aligned} D_{S_s} &\leftarrow \mathcal{A}^+(D_C, D_T, \mathcal{F}_{NN}(D_T; W)) \\ D_T &\leftarrow D_T \cup D_{S_s}, \quad D_C \leftarrow D_C - D_{S_s} \end{aligned}$$

for incremental learning

until convergence is reached.

In the algorithm above $\mathcal{A}$ denotes the active learning operator, which is defined as follows for each of the active learning classes:

1) $\mathcal{A}^-(D_C, \mathcal{F}_{NN}(D_T; W)) = D_S$, where $D_S \subseteq D_C$. The operator $\mathcal{A}^-$ receives as input the candidate set D_C, performs some calculations on each pattern $p \in D_C$, and produces the subset D_S with the characteristics $D_S \subseteq D_C$, that is $|D_S| \leq |D_C|$. The aim of this operator is therefore to produce a subset D_S from D_C which is smaller than, or equal to, D_C. Then, let $D_T \leftarrow D_S$, where D_T is the actual training set.

2) $\mathcal{A}^+(D_C, D_T, \mathcal{F}_{NN}(D_T; W)) = D_S$, where D_C, D_T and D_S are sets such that $D_T \subseteq D_C$, $D_S \subseteq D_C$. The operator $\mathcal{A}^+$ performs calculations on each pattern $p \in D_C$ to determine if that element should be added to the current training set. Selected patterns are added to subset D_S. Thus, $D_S = \{p | p \in D_C, \text{ and } \mathrm{p} \text{ satisfies the selection criteria}\}$. Then, $D_T \leftarrow D_T \cup D_S$ (the new subset is added to the current training subset), and $D_C \leftarrow D_C - D_S$.

Active learning operator $\mathcal{A}^-$ corresponds with selective learning where the training set is "pruned", while $\mathcal{A}^+$ corresponds with incremental learning where the actual training subset "grows". Inclusion of the NN function $\mathcal{F}_{NN}$ as a parameter of each operator indicates the dependence on the NN's current knowledge.

7.4 Conclusion

Using a NN does not involve pasting together a few neurons, giving the NN some data, and out comes the result! NNs should be designed carefully to achieve optimal performance, in terms of accuracy, convergence speed and computational complexity. This chapter discussed several ways to improve NN performance. However, please

note that this is not a complete treatment of ways to improve network performance. The reader is referred to the large source of information available in journals, books and conference proceedings.

7.5 Assignments

1. Discuss measures that quantify the performance of unsupervised neural networks.
2. Discuss factors that influence the performance of unsupervised neural networks. Explain how the performance can be improved.
3. Why is the SSE not a good measure to compare the performance of NNs on different data set sizes?
4. Why is the MSE not a good measure of performance for classification problems?
5. One approach to incremental learning is to select from the candidate training set the most informative pattern as the one with the largest error. Justify and criticize this approach. Assume that a new pattern is selected at each epoch.
6. Explain the role of the steepness coefficient in $\frac{1}{1+e^{-\lambda net}}$ in the performance of supervised NNs.
7. Explain how architecture selection can be used to avoid overfitting.
8. Explain how active learning can be used to avoid overfitting.
9. Consider the sigmoid activation function. Discuss how scaling of the training data affects the performance of NNs.
10. Explain how the Huber function makes a NN more robust to outliers.

Part III

EVOLUTIONARY COMPUTING

The world we live in is constantly changing. In order to survive in a dynamically changing environment, individuals must have the ability to adapt. Evolution is this process of adaption with the aim of improving the survival capabilities through processes such as natural selection, survival of the fittest, reproduction, mutation, competition and symbiosis.

This part covers evolutionary computing (EC) – a field of CI which models the processes of natural evolution. Several evolutionary algorithms (EA) have been developed. This text covers genetic algorithms in Chapter 9, genetic programming in Chapter 10, evolutionary programming in Chapter 11, evolutionary strategies in Chapter 12, differential evolution in Chapter 13, cultural evolution in Chapter 14, and co-evolution in Chapter 15. An introduction to basic EC concepts is given in Chapter 8.

Chapter 8

Introduction to Evolutionary Computing

Evolution is an optimization process, where the aim is to improve the ability of individuals to survive. Evolutionary computing (EC) is the emulation of the process of natural selection in a search procedure. In nature, organisms have certain characteristics that influence their ability to survive and reproduce. These characteristics are represented by long strings of information contained in the chromosomes of the organism. After sexual reproduction the chromosomes of the offspring consist of a combination of the chromosomal information from each parent. Hopefully, the end result will be offspring chromosomes that contain the best characteristics of each parent. The process of natural selection ensures that the more "fit" individuals have the opportunity to mate most of the time, leading to the expectation that the offspring have a similar, or better fitness.

Occasionally, chromosomes are subjected to mutations which cause changes to the characteristics of the corresponding individuals. These changes can have a negative influence on the individual's ability to survive or reproduce. On the other hand, mutation may actually improve the fitness of an individual, thereby improving its chances of survival and of taking part in producing offspring. Without mutation, the population tends to converge to a homogeneous state where individuals vary only slightly from each other.

Evolution via natural selection of a randomly chosen population of individuals can be thought of as a search through the space of possible chromosome values. In that sense, an evolutionary algorithm (EA) is a stochastic search for an optimal solution to a given problem. The evolutionary search process is influenced by the following main components of EA:

- an **encoding** of solutions to the problem as a chromosome;

- a **function** to evaluate the **fitness**, or survival strength of individuals;
- **initialization** of the initial population;
- **selection** operators; and
- **reproduction** operators.

Each of these aspects is introduced and discussed briefly in the sections that follow. More detailed discussions follow in the chapters on the different EC paradigms. A comparison between classical optimization and EC is given in Section 8.7. A general EA is given in Section 8.6.

EAs have been applied to a wide range of problem areas, including

- **planning**, for example, routing optimization and scheduling;
- **design**, for example, the design of filters, neural network architectures and structural optimization;
- **control**, for example, controllers for gas turbine engines, and visual guidance systems for robots;
- **classification** and **clustering**;
- **function approximation** and **time series modeling**;
- **regression**;
- **composing music**; and
- **data mining**.

EAs have shown advantages over existing algorithmic solutions in the above application areas. It is interesting to note that EAs are being increasingly applied to areas in which computers have not been used before.

8.1 Representation of Solutions – The Chromosome

An evolutionary algorithm utilizes a *population* of individuals, where each individual represents a candidate solution to the problem. The characteristics of an individual are represented by a *chromosome*, or *genome*. The characteristics represented by a chromosome can be divided into classes of evolutionary information: genotypes and phenotypes. A *genotype* describes the genetic composition of an individual as inherited from its parents. Genotypes provide a mechanism to store experiential

evidence as gathered by parents. A *phenotype* is the expressed behavioral traits of an individual in a specific environment. A complex relationship can exist between the genotype and phenotype. Two such relationships are [Mayr 1963]:

- pleiotropy, where random modification of genes cause unexpected variations in the phenotypic traits; and
- polygeny, where several genes interact to produce a specific phenotypic trait. To change this behavioral characteristic, all the associated genes need to change.

Each chromosome represents a point in search space. A chromosome consists of a number of genes, where the *gene* is the functional unit of inheritance. Each gene represents one characteristic of the individual, with the value of each gene referred to as an *allele*. In terms of optimization, a gene represents one parameter of the optimization problem.

A very important step in the design of an EA is to find an appropriate chromosome representation. The efficiency and complexity of a search algorithm greatly depend on the representation scheme, where classical optimization techniques usually use vectors of real numbers, different EAs use different representation schemes. For example, genetic algorithms (GA) mostly use a binary string representation, where the binary values may represent Boolean values, integers or even discretized real numbers, genetic programming (GP) makes use of a tree representation to represent programs and evolutionary programming (EP) uses real-valued variables. The different representation schemes are described in more detail in the chapters that follow.

8.2 Fitness Function

The fitness function is possibly the most important component of an EA. The purpose of the fitness function is to map a chromosome representation into a scalar value:

$$\mathcal{F}_{EA} : C^I \rightarrow \mathbb{R} \tag{8.1}$$

where $\mathcal{F}_{EA}$ is the fitness function, and $\vec{C}$ represents the I-dimensional chromosome.

Since each chromosome represents a potential solution, the evaluation of the fitness function quantifies the quality of that chromosome, i.e. how close the solution is to the optimal solution. Selection, cross-over, mutation and elitism operators usually make use of the fitness evaluation of chromosomes. For example, selection operators use the fitness evaluations to decide which are the best parents to reproduce. Also, the probability of an individual to be mutated can be a function of its fitness: highly fit individuals should preferably not be mutated.

It is therefore extremely important that the fitness function accurately models the optimization problem. The fitness function should include all criteria to be optimized. In addition to optimization criteria, the fitness function can also reflect the constraints of the problem through penalization of those individuals that violate constraints. It is not required that the constraints are encapsulated within the fitness function; constraints can also be incorporated in the initialization, reproduction and mutation operators.

8.3 Initial Population

Before the evolutionary process can start, an initial population has to be generated. The standard way of generating the initial population is to choose gene values randomly from the allowed set of values. The goal of random selection is to ensure that the initial population is a uniform representation of the entire search space. If prior knowledge about the search space and problem is available, heuristics can be used to bias the initial population toward potentially good solutions. However, this approach to population initialization leads to opportunistic EAs. Not all of the elements of the search space have a chance to be selected, which may result in premature convergence of the population to a local optimum.

The size of the initial population has consequences for performance in terms of accuracy and the time to converge. A small population represents a small part of the search space. While the time complexity per generation is low, the EA may need more generations to converge than for a large population. On the other hand, a large population covers a larger area of the search space, and may require less generations to converge. However, the time complexity per generation is increased. In the case of a small population, the EA can be forced to explore a larger search space by increasing the rate of mutation.

8.4 Selection Operators

Each generation of an EA produces a new generation of individuals, representing a set of new potential solutions to the optimization problem. The new generation is formed through application of three operators: cross-over, mutation and elitism. The aim of the selection operator is to emphasize better solutions in a population.

In the case of cross-over, "superior" individuals should have more opportunities to reproduce. In doing so, the offspring contains combinations of the genetic material of the best individuals. The next generation is therefore strongly influenced by the genes of the fitter individuals. In the case of mutation, fitness values can be used to select only those individuals with the lowest fitness values to be mutated. The

idea is that the most fit individuals should not be distorted through application of mutation – thereby ensuring that the good characteristics of the fit individuals persevere. Elitism is an operator that copies a set of the best individuals to the next generation, hence ensuring that the maximum fitness value does not decrease from one generation to the next. Selection operators are used to select these elitist individuals.

Several selection techniques exist, divided into two classes:

- **Explicit fitness remapping**, where the fitness values of each individual is mapped into a new range, e.g. normalization to the range $[0, 1]$. The mapped value is then used for selection.
- **Implicit fitness remapping**, where the actual fitness values of individuals are used for selection.

Goldberg and Deb [Goldberg and Deb 1991] presented a comparison of these selection schemes. Based on this comparison, Beasley *et al.* concluded that selection schemes from both classes perform equally well with suitable adjustment of parameters [Beasley *et al.* 1993]. Therefor, no method is absolutely superior to the other.

A summary of the most frequently used selection operators are given in the subsections below.

8.4.1 Random Selection

Individuals are selected randomly with no reference to fitness at all. All the individuals, good or bad, have an equal chance of being selected.

8.4.2 Proportional Selection

The chance of individuals being selected is proportional to the fitness values. A probability distribution proportional to fitness is created, and individuals are selected through sampling of the distribution,

$$Prob(\vec{C}_n) = \frac{\mathcal{F}_{EA}(\vec{C}_n)}{\sum_{n=1}^{N} \mathcal{F}_{EA}(\vec{C}_n)} \tag{8.2}$$

where $Prob(\vec{C}_n)$ is the probability that individual $\vec{C}_n$ will be selected, and $\mathcal{F}_{EA}(\vec{C}_n)$ is the fitness of individual $\vec{C}_n$. That is, the probability of an individual being selected, e.g. to produce offspring, is directly proportional to the fitness value of that individual. This may cause an individual to dominate the production of offspring,

thereby limiting diversity in the new population. This can of course be prevented by limiting the number of offspring that a single individual may produce.

In *roulette wheel* sampling the fitness values are normalized, usually by dividing each fitness value by the maximum fitness value. The probability distribution can then be thought of as a roulette wheel, where each slice has a width corresponding to the selection probability of an individual. Selection can be visualized as the spinning of the wheel and testing which slice ends up at the top.

Roulette wheel selection is illustrated by the following pseudocode algorithm:

1. $n = 1$, where n denotes the chromosome index
2. $sum = Prob(\vec{C}_n)$ using equation (8.2)
3. choose a uniform random number, $\xi \sim U(0, 1)$
4. while $sum < \xi$
   (a) $n++$
   (b) $sum+ = Prob(\vec{C}_n)$
5. return C_n as the selected individual.

It is possible that the population can be dominated by a few individuals with high fitness, having a narrow range of fitness values. Similar fitness values are then assigned to a large set of individuals in the population – leading to a loss in the emphasis toward better individuals. Scaling, or normalization, is then required (explicit fitness remapping) to accentuate small differences in fitness values. In doing so, the objective emphasis toward more fit individuals is maintained. One approach is the normalization to $[0, 1]$ by dividing all fitness values by the maximum fitness.

8.4.3 Tournament Selection

In tournament selection a group of k individuals is randomly selected. These k individuals then take part in a tournament, i.e. the individual with the best fitness is selected. For cross-over, two tournaments are held: one to select each of the two parents. It is therefore possible that (1) a parent can be selected to reproduce more than once, and (2) that one individual can combine with itself to reproduce offspring. The question then arises if this should be allowed or not. The answer is left to the reader.

The advantage of tournament selection is that the worse individuals of the population will not be selected, and will therefore not contribute to the genetic construction of

the next generation, and the best individual will not dominate in the reproduction process.

8.4.4 Rank-Based Selection

Rank-based selection uses the rank ordering of the fitness values to determine the probability of selection and not the fitness values itself. This means that the selection probability is independent of the actual fitness value. Ranking therefore has the advantage that a highly fit individual will not dominate in the selection process as a function of the magnitude of its fitness.

One example of rank-based selection is *non-deterministic linear sampling*, where individuals are sorted in decreasing fitness value. The first individual is the most fit one. The selection operator is defined as

1. let $n = random(random(N))$
2. return $\vec{C}_n$ as the selected individual

Nonlinear ranking techniques have also been developed, for example, where

$$Prob(\vec{C}_n) = \frac{1 - e^{-r(\vec{C}_n)}}{\mu} \tag{8.3}$$

or

$$Prob(\vec{C}_n) = \alpha(1-\alpha)^{N-1-r(\vec{C}_n)} \tag{8.4}$$

where $r(\vec{C}_n)$ is the position (rank) of individual $\vec{C}_n$, μ is a normalization constant, and α is a constant that expresses the probability of selecting the next individual.

These nonlinear selection operators bias toward the best individuals, at the cost of possible premature convergence.

8.4.5 Elitism

Elitism involves the selection of a set of individuals from the current generation to survive to the next generation. The number of individuals to survive to the next generation, without being mutated, is referred to as the *generation gap*. If the generation gap is zero, the new generation will consist entirely of new individuals. For positive generation gaps, say k, k individuals survive to the next generation. These can be

- the k best individuals, which will ensure that the maximum fitness value does not decrease, or
- k individuals, selected using any of the previously discussed selection operators.

8.5 Reproduction Operators

The purpose of reproduction operators is to produce new offspring from selected individuals, either through cross-over or mutation. *Cross-over* is the process of creating a new individual through the combination of the genetic material of two parents. *Mutation* is the process of randomly changing the values of genes in a chromosome. The aim of mutation is to introduce new genetic material into an existing individual, thereby enlarging the search-space. Mutation usually occurs at a low probability. A large mutation probability distorts the genetic structure of a chromosome – the disadvantage being a loss of good genetic material in the case of highly fit individuals.

Reproduction operators are usually applied to produce the individuals for the next generation. Reproduction can, however, be applied with replacement. That is, newly generated individuals replace parents if the fitness of the offspring is better than the parents; if not, the offspring do not survive to the next generation.

Since cross-over and mutation operators are representation dependent, the different implementations of these operators are covered in the chapters that follow.

8.6 General Evolutionary Algorithm

The following pseudocode represents a general evolutionary algorithm. While this algorithm includes all operator types, different EC paradigms use different operators (as discussed in the following chapters).

1. Let $g = 0$ be the generation counter.
2. Initialize a population C_g of N individuals, i.e. $C_g = \{\vec{C}_{g,n} | n = 1, \cdots, N\}$.
3. While no convergence
   (a) Evaluate the fitness $\mathcal{F}_{EA}(\vec{C}_{g,n})$ of each individual in population C_g
   (b) perform cross-over:
      i. select two individuals $\vec{C}_{g,n_1}$ and $\vec{C}_{g,n_2}$
      ii. produce offspring from $\vec{C}_{g,n_1}$ and $\vec{C}_{g,n_2}$
   (c) perform mutation
      i. select one individual $\vec{C}_{g,n}$
      ii. mutate $\vec{C}_{g,n}$
   (d) select the new generation C_{g+1}
   (e) evolve the next generation: let $g = g + 1$

Convergence is reached when, for example,

- the maximum number of generations is exceeded
- an acceptable best fit individual has evolved
- the average and/or maximum fitness value do not change significantly over the past g generations.

8.7 Evolutionary Computing vs Classical Optimization

The no-free-lunch (NFL) theorem [Wolpert and Macready 1996] states that there cannot exist any algorithm for solving all problems that is on average superior to any other algorithm. This theorem motivates research in new optimization algorithms, especially EC. While classical optimization algorithms have been shown to be very successful (and more efficient than EAs) in linear, quadratic, strongly convex, unimodal and other specialized problems, EAs have been shown to be more efficient for discontinuous, nondifferentiable, multimodal and noisy problems.

EC and classical optimization (CO) differ mainly in the search process and information about the search space used to guide the search process:

- **The search process**: CO uses deterministic rules to move from one point in the search space to the next point. EC, on the other hand, uses probabilistic transition rules. Also, EC uses a parallel search through search space, while CO uses a sequential search. The EC search starts from a diverse set of initial points, which allows parallel search of a large area of the search space. CO starts from one point, successively adjusting this point to move toward the optimum.
- **Search surface information**: CO uses derivative information, usually first-order or second-order, of the search space to guide the path to the optimum. EC, on the other hand, uses no derivative information. Only the fitness values of individuals are used to guide the search.

8.8 Conclusion

While this chapter presented a short introduction to the different aspects of EC, the next chapters elaborate on each EC paradigm. Design aspects of each EC paradigm, including representation and operators, are discussed in detail in these chapters. Performance issues are also discussed per EC paradigm.

8.9 Assignments

1. Discuss the importance of the fitness function in EC.
2. Discuss the difference between genetic and phenotypic evolution.
3. In the case of a small population size, how can we ensure that a large part of the search space is covered?
4. How can premature convergence be prevented?
5. In what situations will a high mutation rate be of advantage?
6. Is the following statement valid? *"A genetic algorithm is assumed to have converged to a local or global solution when the ratio $P_r = \overline{f}/f_{max}$ is close to 1, where f_{max} and $\overline{f}$ are the maximum and average fitness of the evolving population respectively."*
7. How can an EA be used to train a NN? In answering this question, focus on
   (a) the representation scheme, and
   (b) fitness function.
8. Show how an EA can be used to solve systems of equations, by illustrating how
   (a) solutions are represented
   (b) the fitness is calculate
   (c) we have a problem with using EAs for solving systems of equations.

Chapter 9

Genetic Algorithms

Genetic Algorithms (GA) model genetic evolution. The characteristics of individuals are therefore expressed using genotypes. First introduced by John Holland in 1975 [Holland 1975], the GA was the first EC paradigm developed and applied. The original GAs developed by Holland had distinct features: (1) a bit string representation, (2) proportional selection and (3) cross-over as the primary method to produce new individuals, were used. Several changes to the original Holland GA have been developed, which use different representation schemes, selection, cross-over, mutation and elitism operators.

This chapter presents an introduction to genetic algorithms. Section 9.1 starts the chapter with a short overview of pure random search, illustrating similarities with GAs. Pseudocode for a general GA is given in Section 9.2. Chapter 8 introduced the different EA operators, and elaborated on those operators common to all EC paradigms. In this Section the operators specific to GAs are discussed in more detail. Section 9.3 discusses GA chromosome representation schemes, while cross-over operators are discussed in Section 9.4 and mutation in section 9.5. Section 9.6 discusses, in short, an island approach to GAs. An application to routing optimization in telecommunications networks is presented in Section 9.7.

9.1 Random Search

Random search is possibly the simplest search procedure. Starting from an initial search point, or set of initial points, the search process simply consists of random perturbations to the point(s) in search space - until an acceptable solution is reached, or a maximum number of iterations is exceeded. While random search is extremely simple to implement, it can be inefficient. Training time may be very long before an acceptable solution is obtained.

A pseudo-code algorithm for random search is given below:

1. Select a set of N initial search points $C_g = \{\vec{C}_{g,n} | n = 1, \cdots, N\}$, where $\vec{C}_{g,n}$ is a vector of I variables and $g = 0$. Each element $C_{g,ni}$ is sampled from a uniform distribution $U(-max, max)$, where max is a limit placed on variable values.
2. Evaluate the accuracy ("fitness") $\mathcal{F}(\vec{C}_{g,n})$ of each vector $\vec{C}_{g,n}$.
3. Find the best point $\vec{C}_{g,best} = \min_{n=1,\cdots,N}\{\mathcal{F}(\vec{C}_{g,n})\}$.
4. If $\vec{C}_{g,best} < \vec{C}_{best}$, where $\vec{C}_{best}$ is the overall best solution, then $\vec{C}_{best} = \vec{C}_{g,best}$.
5. If $\vec{C}_{best}$ is an acceptable solution, or the maximum number of iterations has been exceeded, then stop and return $\vec{C}_{best}$ as the solution.
6. Perturb each $\vec{C}_{g,n}$ with $\Delta\vec{C}_{g,n}$, where $\Delta\vec{C}_{g,n} \sim N(0, \sigma^2)$, with σ^2 a small variance.
7. Let $g = g + 1$ and go to step 2.

The random search above bear similarities with GAs. If random selection is used for cross-over and mutation, a GA becomes a pure random search.

9.2 General Genetic Algorithm

This section summarizes a general GA in the following pseudocode. Subsequent sections discuss the different operator choices in more detail.

1. Let $g = 0$.
2. Initialize the initial generation C_g.
3. While not converged
   (a) Evaluate the fitness of each individual $\vec{C}_{g,n} \in C_g$.
   (b) $g = g + 1$.
   (c) Select parents from C_{g-1}.
   (d) Recombine selected parents through cross-over to form offspring O_g.
   (e) Mutate offspring in O_g.
   (f) Select the new generation C_g from the previous generation C_{g-1} and the offspring O_g.

9.3 Chromosome Representation

The classical representation scheme for GAs is binary vectors of fixed length. In the case of an I-dimensional search space, each individual consists of I variables with each variable encoded as a bit string. If variables have binary values, the length of each chromosome is I bits. In the case of nominal-valued variables, each nominal value can be encoded as a D-dimensional bit vector, where 2^D is the total number of discrete nominal values for that variable. Each D-bit string represents a different nominal value. In the case of continuous-valued variables, each variable should be mapped to a D-dimensional bit vector, i.e.

$$\phi : \mathbb{R} \rightarrow \{0,1\}^D \tag{9.1}$$

The range of the continuous space needs to be restricted to a finite range $[\alpha, \beta]$. Using standard binary decoding, each continuous variable $C_{n,i}$ of chromosome $\vec{C}_n$ is encoded using a fixed length bit string. For example, if $z \in [z_{min}, z_{max}]$ needs to be converted to a 30-bit representation, the following conversion formula can be used:

$$(2^{30} - 1)\frac{z - z_{min}}{z_{max} - z_{min}}$$

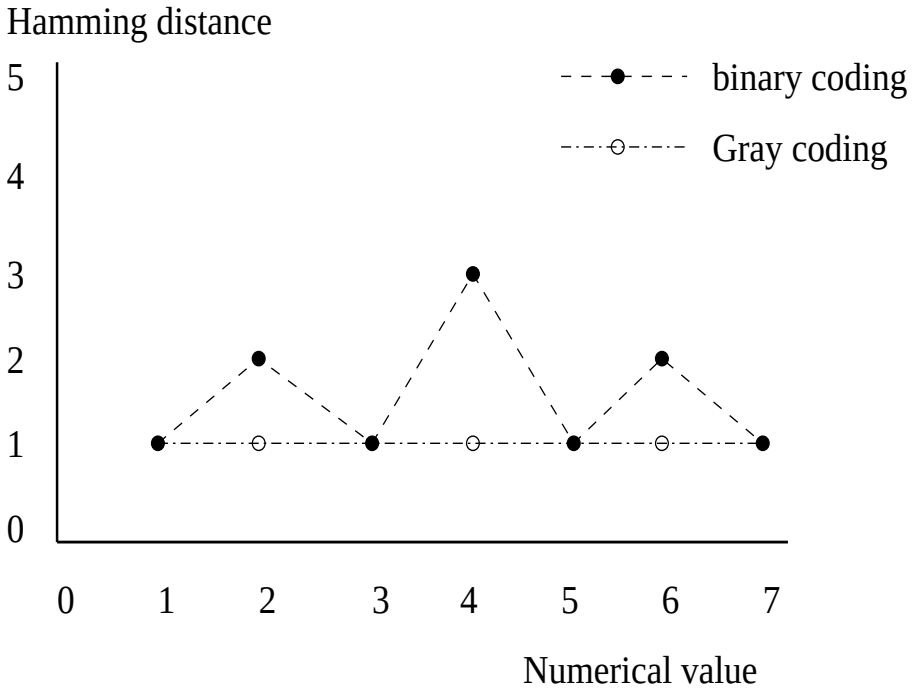


Figure 9.1: Hamming distance for binary and Gray coding

While binary coding is frequently used, it has the disadvantage of introducing Hamming cliffs as illustrated in Figure 9.1. A Hamming cliff is formed when two numerically adjacent values have bit representations that are far apart. For example,

	Binary	*Gray*
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Table 9.1: Binary and Gray coding

consider the decimal numbers 7 and 8. The corresponding binary representations are (using a 4-bit representation) $7 = 0111$ and $8 = 1000$, with a Hamming distance of 4 (the Hamming distance is the number of corresponding bits that differ). This presents a problem when a small change in variables should result in a small change in fitness. If, for example, 7 represents the optimal solution, and the current best solution has a fitness of 8, many bits need to be changed to cause a small change in fitness value.

An alternative bit representation is to use Gray coding, where the Hamming distance between the representation of successive numerical values is one (as illustrated in Figure 9.1). Table 9.1 compares binary and Gray coding for a 3-bit representation.

Binary numbers can easily be converted to Gray coding using the conversion

$$\begin{aligned} g_1 &= b_1 \\ g_k &= b_{k-1}\overline{b}_k + \overline{b}_{k-1}b_k \end{aligned}$$

where b_k is bit k of the binary number $b_1 b_2 \cdots b_K$, with b_1 the most significant bit; $\overline{b}_k$ denotes *not* b_k, $+$ means logical OR, and multiplication implies logical AND.

GAs have also been developed that use integer or real-valued representations [Davis 1991, Janikow and Michalewicz 1991, Sevenster and Engelbrecht 1996] and order-based representations where the order of variables in a chromosome plays an important role [Whitley *et al.* 1989, Syswerda 1991]. Also, it is not necessary that chromosomes are of fixed length (especially in data mining applications) [Goldberg 1989, Holland *et al.* 1986].

9.4 Cross-over

The aim of cross-over is to produce offspring from two parents, selected using a selection operator as discussed in Section 8.4. However, it is not necessary that each group of parents produces offspring. In fact, cross-over takes place at a certain probability, referred to as the *cross-over rate* $p_c \in [0, 1]$. A simple pseudocode algorithm to illustrate cross-over between individuals $\vec{C}_{n_1}$ and $\vec{C}_{n_2}$ is:

1. Compute a random number $\xi \sim U(0, 1)$.
2. If $(\xi > p_c)$ then no cross-over takes place and the parents are simply returned, otherwise go to step 3.
3. $\vec{\alpha} = \vec{C}_{n_1}$ and $\vec{\beta} = \vec{C}_{n_2}$.
4. Compute the mask, $\vec{m}$ (see below).
5. For $i = 1, \cdots, I$, if $(m_i = 1)$ then swap the genetic material:
   (a) $\alpha_i = C_{n_2,i}$
   (b) $\beta_i = C_{n_1,i}$
6. Return the offspring α_i and β_i.

In the procedure above, $\vec{m}$ is a mask which specifies which bits of the parents should be swapped to generate offspring. Several cross-over operators have been developed to compute the mask:

- **Uniform cross-over**: For uniform cross-over the mask of length I is created at random for each pair of individuals selected for reproduction. A bit with value of 1 indicates that the corresponding allele have to be swapped between the two parents. Figure 9.2(a) illustrates uniform cross-over, while the pseudocode for generating the mask is given below:

  1. $m_i = 0$ for all $i = 1, \cdots, I$.
  2. For each $i = 1, \cdots, I$:
     (a) Calculate a random value $\xi \sim U(0, 1)$.
     (b) If $(\xi \leq p_x)$, then $m_i = 1$.
  3. Return the mask vector $\vec{m}$.

  In the above p_x is the cross-over probability at each position in the chromosome. If, for example, $p_x = 0.5$, each bit has an equal chance to take part in cross-over.

- **One-point cross-over**: A single bit position is randomly selected and the bit substrings after that point are swapped between the two chromosomes. One-point cross-over is illustrated in Figure 9.2(b), while the mask vector is calculated as

  1. Calculate a random value $\xi \sim U(1, I-1)$.
  2. $m_i = 0$ for all $i = 1, \cdots, I$.
  3. For each $i = \xi + 1, \cdots, I$ let $m_i = 1$.
  4. Return the mask vector $\vec{m}$.

- **Two-point cross-over**: In this case two bit positions are randomly selected, and the bit substrings between these points are swapped as illustrated in Figure 9.2(c). The mask vector is calculated as

  1. Compute two random variable $\xi_1, \xi_2 \sim U(1, I)$.
  2. $m_i = 0$ for all $i = 1, \cdots, I$.
  3. For each $i = \xi_1, \cdots, \xi_2$ let $m_i = 1$.
  4. Return the mask vector $\vec{m}$.

In the case of continuous-valued genes, and arithmetic cross-over can be used. Consider the two parents $\vec{C}_{n_1}$ and $\vec{C}_{n_2}$. Then, two offspring, $\vec{O}_{n_1}$ and $\vec{O}_{n_2}$ are generated using

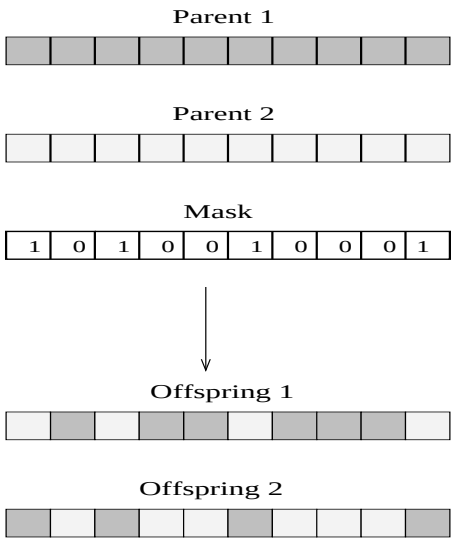
$$O_{n_1,i} = r_1 C_{n_1,i} + (1.0 - r_1) C_{n_2,i}$$
$$O_{n_2,i} = (1.0 - r_2) C_{n_1,i} + r_2 C_{n_2,i}$$

with $r_1, r_2 \in U(0, 1)$.

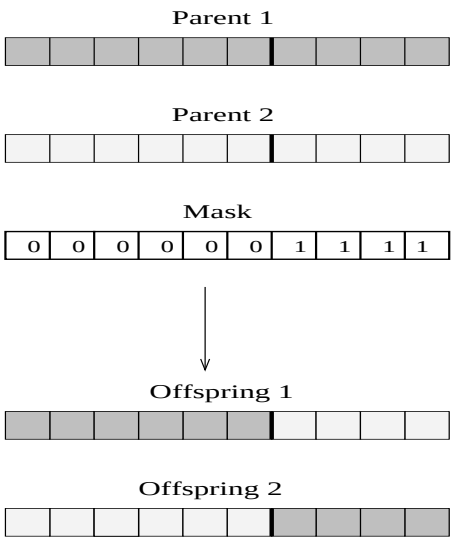
9.5 Mutation

The aim of mutation is to introduce new genetic material into an existing individual; that is, to add diversity to the genetic characteristics of the population. Mutation is used in support of cross-over to make sure that the full range of allele values is accessible in the search. Mutation also occurs at a certain probability p_m, referred to as the *mutation rate*. Usually, a small value for $p_m \in [0, 1]$ is used to ensure that good solutions are not distorted too much. However, research has shown that an initial large mutation rate that decreases exponentially as a function of the number of generations improves convergence speed and accuracy. (An annealing schedule similar to that of equation (4.22) for the learning rate can be used.) The initial large p_m ensures that a large search space is covered, while the p_m becomes rapidly smaller when individuals start to converge to the optimum.

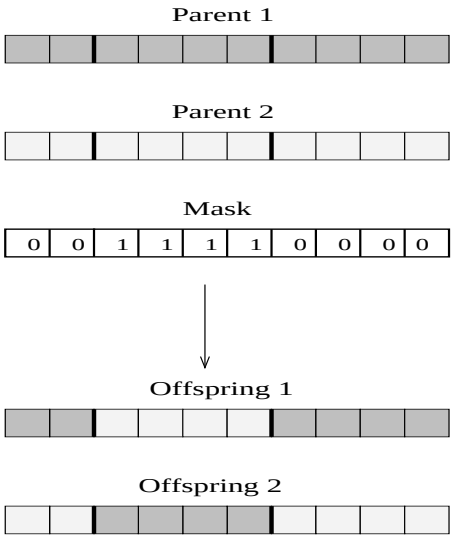
Considering binary representations, the following mutation schemes have been developed:



(a) Uniform Cross-over



(b) One-point Cross-over



(c) Two-point Cross-over

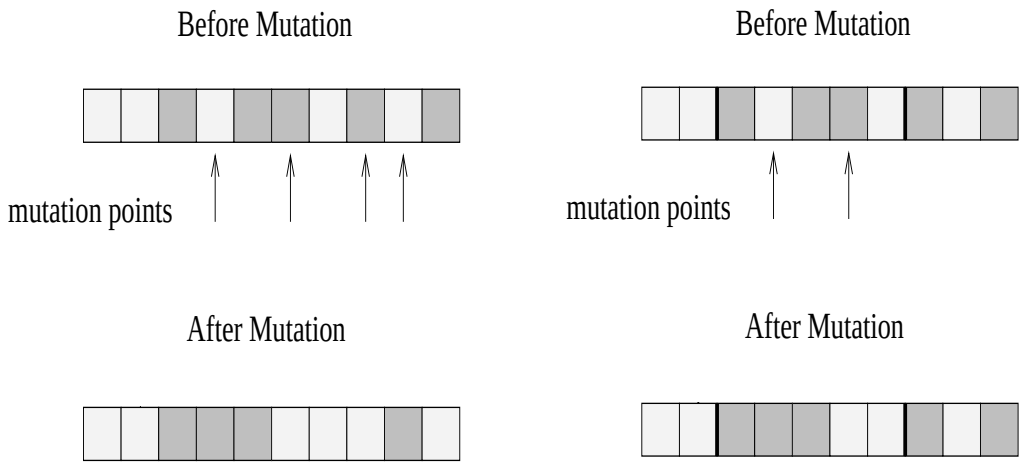
Figure 9.2: Cross-over operators

- **Random mutate**, where bit positions are chosen randomly and the corresponding bit values negated (as illustrated in Figure 9.3(a)). A pseudocode algorithm, for random mutation is the following:

  1. For each $i = 1, \cdots, I$:
     (a) Compute a random value $\xi \sim U(0, 1)$.
     (b) If $\xi \leq p_m$ then $C_{n,i} = \overline{C}_{n,i}$, where $\overline{C}$ is the complement of C.

- **Inorder mutate**, where two bit positions are randomly selected and only bits between these positions are mutated (as illustrated in Figure 9.3(b)). The algorithm for inorder mutation is:

  1. Select two random values $\xi_1, \xi_2 \sim U(1, \cdots, I)$.
  2. For each $i = \xi_1, \cdots, \xi_2$:
     (a) Compute a random value $\xi \sim U(0, 1)$.
     (b) If $\xi \leq p_m$ then $C_{n,i} = \overline{C}_{n,i}$.



(a) Random Mutate (b) Inorder Mutate

Figure 9.3: Mutation operators

In the case of nominal-valued variables, the above mutation operators can be adapted such that the D bits that represent a certain nominal value are replaced by the D bits representing a randomly chosen nominal value. For real-valued representations, mutation occurs by adding a random value to allele, usually sampled from a Gaussian distribution with zero mean and small variance σ^2:

1. For each real-valued gene $C_{n,i}$:

(a) Compute a random value $\xi_i \sim U(0,1)$.

(b) Compute a step size $\eta_i \sim N(0,\sigma^2)$.

(c) If $\xi_i \leq p_m$ then $C_{n,i} += \eta_i$.

Usually, the variance σ^2 is a function of the fitness of the individual. Individuals with a good fitness value will be mutated less, while a bad fitness value will lead to large mutations.

9.6 Island Genetic Algorithms

While GAs with single populations are a form of parallelization, more benefits can be obtained by evolving a number of subpopulations in parallel, which is a cooperative model [Grosso 1985]. In this GA model we have *islands*, where each island represents one population. Selection, cross-over and mutation occur in each subpopulation independently from the other subpopulations. In addition, individuals are allowed to migrate to another island, or subpopulation, as illustrated in Figure 9.4. In this way genetic material is shared among the subpopulations.

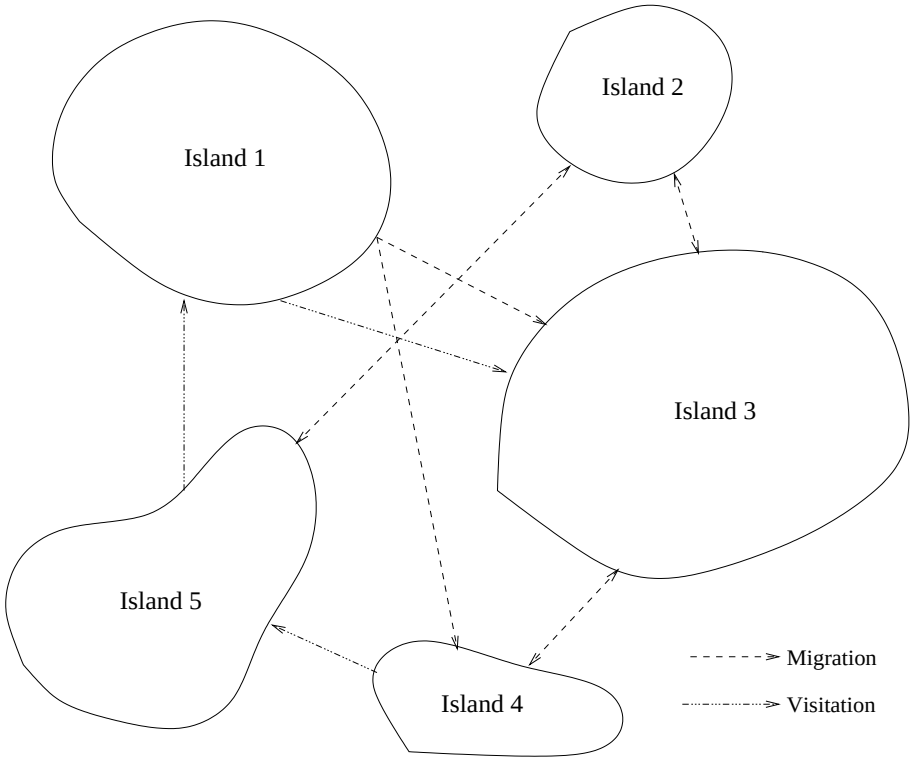


Figure 9.4: An island GA system

Island GA models introduce interesting questions, one of which is how to initialize the subpopulations. Of course a pure random approach can be used, which will cause different populations to share the same parts of the search space. A better approach would be to initialize subpopulations to cover different parts of the search space, thereby covering a larger search space and facilitating a kind of niching by individuals islands. Also, in multicriteria optimization, each subpopulation can be allocated the task to optimize one criterion. A meta-level step is then required to combine the solutions from each island.

Another question is, when can individuals migrate, and from where, to where? The simplest approach is to let migration occur at random. Individuals are selected randomly, and the destination subpopulation is also selected randomly. Alternatively, tournament selection can be used to select migrated individuals, as well as the destination. Intuitively, the best individuals of a poor island may want to migrate to a better island. However, individuals from a poor island may introduce bad genetic material into a good island. Acceptance of an immigrant can then be based on a probability as a function of the immigrant's fitness value compared to that of the intended destination island, or acceptance if the immigrant has the ability to increase the diversity in genetic material, that is, if the individual is maximally different. It is also possible that a highly fit individual from a prosperous island visits islands, taking part in reproduction.

Culling of weak individuals in subpopulations can also be modeled. Immigrants can be allowed, for example, if that immigrant can replace an individual of the destination island with a lower fitness than the immigrant. In doing so, the weaker individual is killed and removed from the population.

It should be noted at this point that the EA employed for islands does not necessarily need to be a GA. It can be any of the other EAs.

A different kind of "island" GA model is the Cooperative Coevolutionary GA of Potter [Potter 1997]. In this case, instead of distributing entire individuals over several subpopulations, each subpopulation is given one gene (one parameter of the optimization problem) to optimize. This is a coevolutionary process, discussed in more detail in chapter 15.

9.7 Routing Optimization Application

This section studies a real-world application of GAs. The problem is the optimization of routes in a telecommunications network [Sevenster and Engelbrecht 1996]. Given a network of M switches, an origin switch and a destination switch, the objective is to find the best route to connect a call between the origin and destination switches. The design of the GA is done in the following steps:

1. **Chromosome representation**: A chromosome consists of a maximum of M switches. Chromosomes can be of variable length, since telecommunication routes can differ in length. Each gene represents one switch. Integer values representing switch numbers are used as gene values - no binary encoding is used. The first gene represents the origin switch and the last gene the destination switch. Example chromosomes are

$$(1 \quad 3 \quad 6 \quad 10)$$

$$(1 \quad 5 \quad 2 \quad 5 \quad 10) = (1 \quad 5 \quad 2 \quad 10)$$

Duplicate switches are ignored. The first chromosome represents a route from switch 1 to switch 3 to switch 6 to switch 10.

2. **Initialization of population**: Individuals are generated randomly, with the restriction that the first gene represents the origin switch and the last gene represents the destination switch. For each gene, the value of that gene is selected as a uniform random value in the range $[1, M]$.

3. **Fitness function:** The multi-criteria objective function

$$F_j = aF_j^{Switch} + bF_j^{Block} + cF_j^{Util} + dF_j^{Cost}$$

is used where

$$F_j^{Switch} = \frac{|r_j|}{M}$$

represents the minimization of route length, where r_j denotes the route and $|r_j|$ is the total number of switches in the route,

$$F_j^{Block} = 1 - \prod_{xy \in r_j}^{|r_j|} (1 - B_{xy} + \alpha_{xy})$$

with

$$\alpha_{xy} = \begin{cases} 1 & \text{if } xy \text{ does not exist} \\ 0 & \text{if } xy \text{ does exist} \end{cases}$$

has as its objective selection of routes with minimum congestion, where B_{xy} denotes the blocking probability on the link between switches x and y,

$$F_j^{Util} = \min_{xy \in r_j} \{1 - U_{xy}\} + \alpha_{xy}$$

maximizes utilization, where U_{xy} quantifies the level of utilization of the link between x and y, and

$$F_j^{Cost} = \sum_{xy \in r_j}^{|r_j|} C_{xy} + \alpha_{xy}$$

ensures that minimum cost routes are selected, where C_{xy} represents the financial cost of carrying a call on the link between x and y. The constants a, b, c and d control the influence of each criterion.

More efficient techniques have been developed to work with multi-objective functions. The reader is referred to [Horn 1993] for a treatment of such techniques, and http://www.tik.ee.ethz.ch/~zitzler/Bibliographies/pareto.html for a list of papers on multiobjective optimization.

4. Use any **selection** operator.

5. Use any **cross-over** operator.

6. **Mutation**: Mutation consists of replacing selected genes with a uniformly random selected switch in the range $[1, M]$.

This example is an illustration of a GA that uses a numeric representation, and variable length chromosomes with constraints placed on the structure of the initial individuals.

9.8 Conclusion

One of the questions that remains to be addressed is how to select the operators to use, as well as the corresponding operator probabilities. Since the optimal combination of operators and probabilities of applying these operators are problem-dependent, extensive simulations have to be conducted to find the optimal operator combinations. That is, of course, a time-consuming process. It is also possible to have a meta-evolution process to evolve the best values for these operators.

For several applications, such as solving systems of equations, it is necessary to locate all minima. Special GAs have been developed to locate multiple minima. These algorithms are referred to as niching GAs. For more information on niching, the reader is referred to [Mahfoud 1995, Horn 1997].

9.9 Assignments

1. Develop a GA to train a FFNN.

2. When is a high mutation rate of an advantage?

3. Is the following strategy sensible? Explain your answer. *Start evolution with a large mutation rate, and decrease the mutation rate with an increase in generation number.*

4. How can GAs be used to optimize the architecture of NNs?
5. Discuss how a GA can be used to cluster data.
6. Investigate a cross-over scheme where more than two parents take part in producing offspring. In this scheme components of offspring consist of that of randomly selected components of a number of selected parents.
7. Design a mutation operator for solving systems of equations.
8. How can a GA be used for constrained optimization?

Chapter 10

Genetic Programming

Genetic programming (GP) is viewed as a specialization of genetic algorithms. Similar to GAs, GP concentrates on the evolution of genotypes. The main difference is in the representation scheme used. Where GAs use string representations, GP represents individuals as executable programs (represented as trees). The aim of GP is therefore to evolve computer programs. For each generation, each evolved program (individual) is executed to measure its performance within the problem domain. The results or performance of the evolved computer program is then used to quantify the fitness of that program.

The operators applied to a GP are similar to that of GAs. The general GA given in Section 9.2 also applies to GP. The next sections concentrate on the differences between the two approaches, which are mainly in the representation scheme (see Section 10.1), fitness evaluation (see Section 10.3), cross-over operators (see Section 10.4) and mutation operators (see Section 10.5).

10.1 Chromosome Representation

GP evolves executable computer programs. Each individual, or chromosome, represents one computer program, represented using a tree structure. For this purpose a grammar needs to be defined that accurately reflects the problem to be solved. Within this grammar a terminal set and function set need to be defined. The terminal set specifies all the variables and constants, while the function set contains all the functions that can be applied to the elements of the terminal set. These functions may include mathematical, arithmetic and/or Boolean functions. Decision structures such as *if-then-else* can also be included with the function set. Using tree terminology, elements of the terminal set form the leaf nodes of the evolved tree, and elements of the function set form the non-leaf nodes.

As an example, consider evolving the following program:

```
y := x * ln(a) + sin(z) / exp(-x) - 3.4;
```

The terminal set is specified as $\{a, x, z, 3.4\}$ with $a, x, z \in \mathbb{R}$. The minimal function set is given as $\{-, +, *, /, sin, exp, ln\}$. The optimum solution is illustrated in Figure 10.1. Terminal elements are placed within circles, while function elements are in the square boxes.

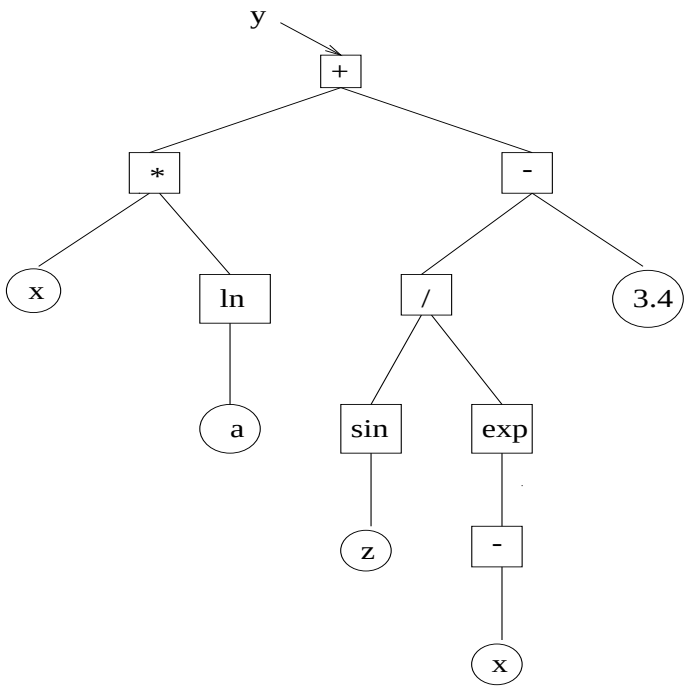


Figure 10.1: Genetic program representation

Each individual in a GP population represents a program, which is an element of the program space formed by all possible programs that can result from the given grammar. The aim of the GP is then to search for a program within the program space that give the best approximation to the objective (the true) program.

Trees within a population can be of a fixed size or variable size. With a fixed size representation all trees have the same depth and all subtrees are expanded to the maximum depth. Variable size tree representations are, however, the most frequently used representation. In this case the only restriction placed on trees is a maximum depth. It is also possible to employ schemes where the maximum depth increases with increasing generation number.

10.2 Initial Population

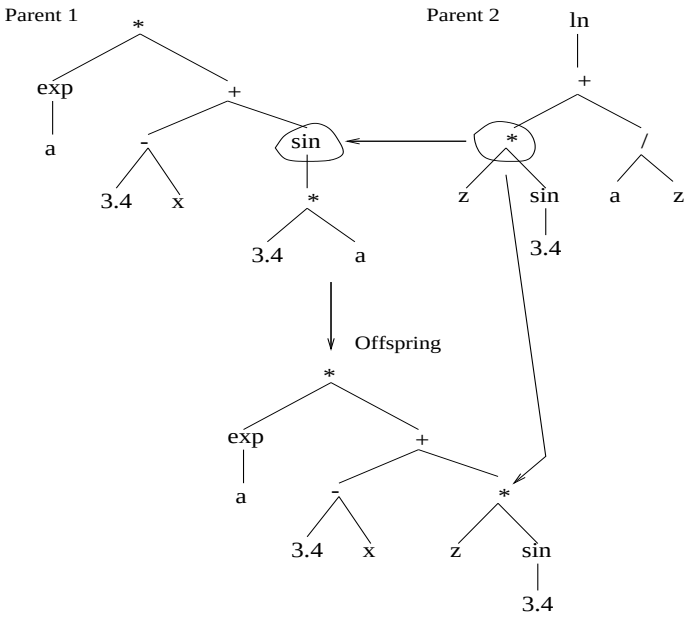
The initial population is generated randomly within the restrictions of a maximum depth and semantics as expressed by the given grammar. For each individual, a root is randomly selected from the set of function elements. The branching factor (the number of children) of the root, and each other non-terminal node, are determined by the arity of the selected function. For each non-root node, the initialization algorithm randomly selects an element either from the terminal set or the function set. As soon as an element from the terminal set is selected, the corresponding node becomes a leaf node and is no longer considered for expansion.

10.3 Fitness Function

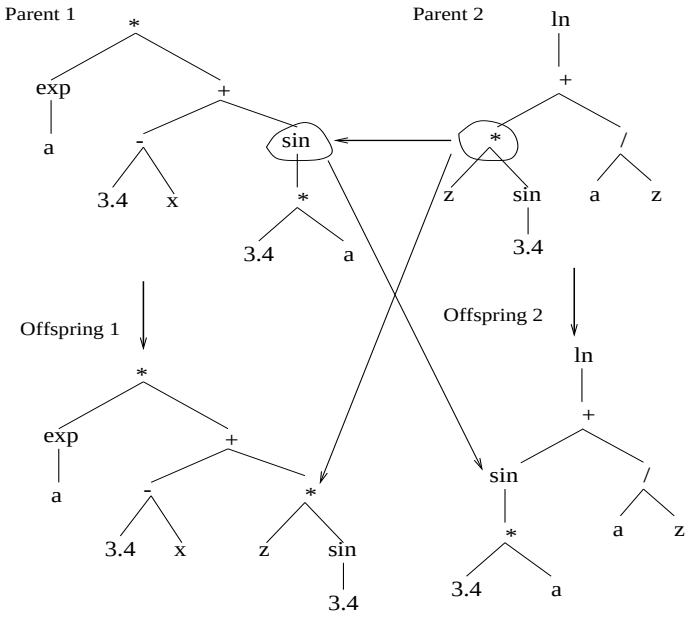
The fitness function used for GP is problem-dependent. Fitness evaluation involves testing each problem on the target domain. This usually requires each individual to be tested on a sample of cases, and the average performance over that sample is used as fitness measure. As an example, return to the program in Figure 10.1. Assume no prior knowledge about the structure of the program, other than the given terminal and function sets. In addition, a data set is available, consisting of a number of data patterns, where each data pattern consists of three input values (one for each of the variables a, x and z) and a target value (i.e. the value of y). The evaluation of each individual entails (1) calculating the output of that individual given the values of a, x and z, and (2) calculating the error made. At the end, the MSE over the given data set is a valid quantification of the fitness of that individual.

In the case where decision trees are evolved, and each individual represents a single decision tree, the fitness of individuals is calculated as the classification accuracy of the corresponding decision tree. If the objective is to evolve a game strategy in terms of a computer program, the fitness of an individual can be the number of times that individual won the game out of a total number of games played.

In addition to being used as a measure of the performance of individuals, the fitness function can also be used to penalize individuals with undesirable structural properties. For example, instead of having a predetermined depth limit, the depth of a tree can be penalized by adding an appropriate penalty term to the fitness function. Similarly, bushy trees (which result when nodes have a large branching factor) can be penalized by adding a penalty term to the fitness function.



(a) Creation of one offspring



(b) Creation of two offspring

Figure 10.2: Genetic programming cross-over

10.4 Cross-over Operators

Any of the previously discussed selection operators (refer to Section 8.4) can be used to select two parents to produce offspring. Two approaches can be used to generate offspring, each one differing in the number of offspring generated:

- **Generating one offspring**: A random node is selected within each of the parents. Cross-over then proceeds by replacing the corresponding subtree in the one parent by that of the other parent. Figure 10.2(a) illustrates this operator for the example of Section 10.1.
- **Generating two offspring**: Again, a random node is selected in each of the two parents. In this case the corresponding subtrees are swapped to create the two offspring as illustrated in Figure 10.2(b).

10.5 Mutation Operators

Several mutation operators have been developed for GP. The most frequently used operators are discussed below with reference to Figure 10.3. Figure 10.3(a) illustrates the original individual before mutation.

- **Function node mutation**: A non-terminal node, or function node, is randomly selected and replaced with a node of the same arity, randomly selected from the function set. Figure 10.3(b) illustrates that function node + is replaced with function node −.
- **Terminal node mutation**: A leaf node, or terminal node, is randomly selected and replaced with a new terminal node, also selected randomly from the terminal set. Figure 10.3(c) illustrates that terminal node a has been replaced with terminal node z.
- **Swap mutation**: A function node is randomly selected and the arguments of that node are swapped as illustrated in Figure 10.3(d).
- **Grow mutation**: With grow mutation a node is randomly selected and replaced by a randomly generated subtree. The new subtree is restricted by a predetermined depth. Figure 10.3(e) illustrates that the node 3.4 is replaced with a subtree.
- **Gaussian mutation**: A terminal node which represents a constant is randomly selected and mutated by adding a Gaussian random value to that constant. Figure 10.3(f) illustrates Gaussian mutation.

- **Trunc mutation**: A function node is randomly selected and replaced by a random terminal node. This mutation operator performs a pruning of the tree. Figure 10.3(g) illustrates that the + function node is replaced by the terminal node a.

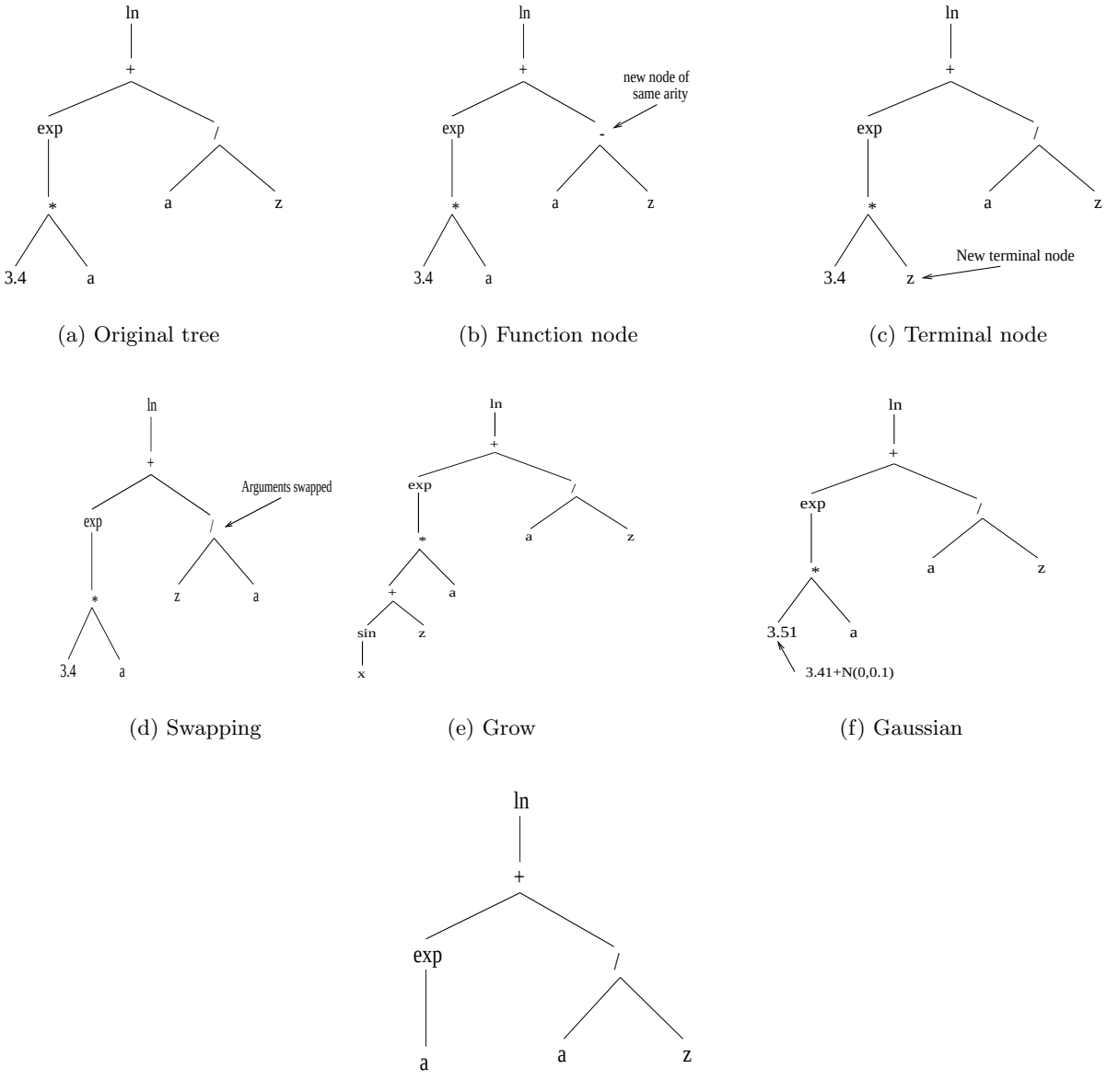
Individuals to be mutated are selected according to a mutation probability p_m. In addition to a mutation probability, nodes within the selected tree are mutated according to a probability p_n. The larger the probability p_n, the more the genetic build-up of that individual is changed. On the other hand, the larger the mutation probability p_m, the more individuals will be mutated.

All of the mutation operators can be implemented, or just a subset thereof. If more than one mutation operator is implemented, then either one operator is selected randomly, or more than one operator is selected and applied in sequence.

10.6 Building-Block Approach to Genetic Programming

The GP process discussed thus far generates an initial population of individuals where each individual represents a tree consisting of several nodes and levels. An alternative approach has been developed in [Engelbrecht *et al.* 2002, Rouwhorst and Engelbrecht 2000] – specifically for evolving decision trees – referred to as a building-block approach to GP (BGP). In this approach, initial individuals consist of only a root and the immediate children of that node. Evolution starts on these "small" initial trees. When the simplicity of the population's individuals can no longer account for the complexity of the problem to be solved, and no improvement in the fitness of any of the individuals within the population is observed, individuals are expanded. Expansion occurs by adding a randomly generated building block (i.e. a new node) to individuals. In other words, grow mutation is applied. This expansion occurs at a specified expansion probability p_e, and therefore not all of the individuals are expanded. Described more formally, the building-block approach starts with models with a few degrees of freedom – most likely too few to solve the problem to the desired degree of accuracy. During the evolution process, more degrees of freedom are added when no further improvements are observed. In between the triggering of expansion, cross-over and mutation occur as for normal GP.

This approach to GP helps to reduce the computational complexity of the evolution process, and helps to produce smaller individuals.



(a) Original tree

(b) Function node

(c) Terminal node

(d) Swapping

(e) Grow

(f) Gaussian

(g) Trunc

Figure 10.3: Genetic programming mutation operators

10.7 Assignments

1. Explain how a GP can be used to evolve a program to control a robot, where the objective of a robot is to move out of a room (through the door) filled with obstacles.

2. First explain what a decision tree is, and then show how GP can be used to evolve decision trees.

3. Is it possible to use GP for adaptive story telling?

4. Given a pre-condition and a post-condition of a function, is it possible to evolve the function using GP?

5. Explain why BGP is computationally less expensive than GP.

Chapter 11

Evolutionary Programming

Evolutionary programming (EP) differs substantially from GAs and GP in that EP emphasizes the development of behavioral models and not genetic models: EP is derived from the simulation of adaptive behavior in evolution. That is, EP considers phenotypic evolution. The evolutionary process consists of finding a set of optimal behaviors from a space of observable behaviors. For this purpose, the fitness function measures the "behavioral error" of an individual with respect to the environment of that individual.

EP further differs from GAs and GP in that no crossover is implemented. Only mutation is used to produce the new population. A general EP algorithm is given in Section 11.1, while Section 11.2 discusses the general mutation process in EP. Section 11.3 illustrates EP on two example applications. Representation issues, the design of fitness functions and specific mutation operators are also discussed in this section.

11.1 General Evolutionary Programming Algorithm

The following pseudocode presents a general EP algorithm:

1. Let the current generation be $g = 0$.
2. Initialize the population $C_g = \{\vec{C}_{g,n} | n = 1, \cdots, N\}$.
3. While no convergence
   (a) Evaluate the fitness of each individual $\vec{C}_{g,n}$ as $\mathcal{F}_{EP}(\vec{C}_{g,n})$.
   (b) Mutate each individual $\vec{C}_{g,n}$ to produce offspring $\vec{O}_{g,n}$.
   (c) Select the new population $C_{g+1,n}$ from C_g and O_g.

(d) Let $g = g + 1$.

Note the absence of crossover in the algorithm above. Reproduction is done through mutation only.

11.2 Mutation and Selection

Mutation is applied to each of the individuals at a certain probability, p_m. The mutation operator used depends on the application, as is illustrated later in this chapter. Offspring generated through mutation compete with the parent individuals to survive to the next generation. After mutation, the new population is selected, in one of the following ways:

- **All the individuals**, i.e. parents and offspring, have the same chance to be selected. Any of the selection operators of Section 8.4, e.g. tournament selection, can be used to create the new population.
- An **elitist** mechanism is used to transfer a group of the best parents to the next generation. The remainder of the population is selected from the remainder of the parents and offspring.
- First **cull** the worst parents and offspring. Then select the new population from the remainder "good" individuals. This will ensure that weak individuals do not survive to the next generation.
- Of course, a combination of an elitist and culling strategy is also possible.

11.3 Evolutionary Programming Examples

This section illustrates EP using two examples, i.e. evolving a finite-state machine and function optimization. Each of the examples discusses mutation operators and representation schemes specific to that application.

11.3.1 Finite-State Machines

EP was originally developed to evolve finite-state machines (FSM). The aim of this application type is to evolve a program to predict the next symbol (of a finite alphabet) based on a sequence of previously observed symbols.

A finite-state machine is essentially a computer program which represents a sequence of actions that must be executed, where each action depends on the current state of

the machine and an input. Formally, a FSM is defined as

$$FSM = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \rho, \phi) \tag{11.1}$$

where $\mathcal{S}$ is a finite set of machine states, $\mathcal{I}$ is a finite set of input symbols, $\mathcal{O}$ is a finite set of output symbols (the alphabet of the FSM), $\rho : \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{S}$ is the next state function, and $\phi : \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{O}$ is the next output function. An example of a 3-state FSM is given in Figure 11.1 (taken from [Fogel *et al.* 1966]). The response of the FSM to a given string of symbols is given in table 11.1, presuming an initial state C.

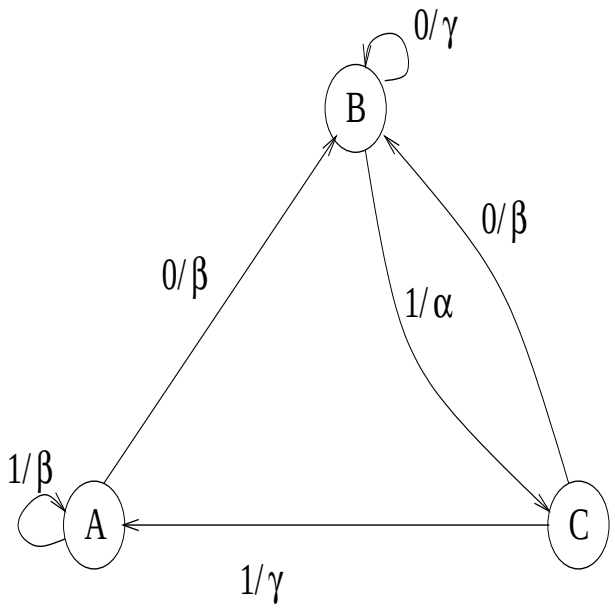


Figure 11.1: Finite-state machine

Present state	C	B	C	A	A	B
Input symbol	0	1	1	1	0	0
Next state	B	C	A	A	B	C
Output symbol	β	α	γ	β	β	γ

Table 11.1: Response of finite-state machine

Representation

Each state can be represented by a 6-bit string. The first bit represents the activation of the corresponding state (0 indicates not active, and 1 indicates active). The second

bit represents the input symbol, the next two bits represent the next state, and the last two bits represent the output symbol. Each individual therefore consists of 18 bits. The initial population is randomly generated, with the restriction that the output symbol and next state bits represent only valid values.

Fitness Evaluation

The fitness of each individual is measured as the individual's ability to correctly predict the next output symbol. A sequence of symbols is used for this purpose. The first symbol from the sequence is presented to each individual, and the predicted symbol compared to the next symbol in the sequence. The second symbol is then presented as input, and the process iterates over the entire sequence. The individual with the most correct predictions is considered the most fit individual.

Mutation

The following mutation operations can be applied:

- The initial state can be changed.
- A state can be deleted.
- A state can be added.
- A state transition can be changed.
- An output symbol for a given state and input symbol can be changed.

These operators are applied probabilistically, in one of the following ways:

- Select a uniform random number between 1 and 5. The corresponding mutation operator is then applied with probability p_m.
- Generate a Poisson number, ξ with mean λ. Select ξ mutation operators uniformly from the set of operators, and apply them in sequence.

11.3.2 Function Optimization

The next example application of EP is in function optimization. Consider, for example, finding the minimum of the function $\sin(2\pi x)e^{-x}$ in the range $[0, 2]$.

Representation

The function has one parameter. Each individual is therefore represented by a vector consisting of one floating-point element (not binary encoded). The initial population is generated randomly, with each individual's parameter C_{nx} selected such that $C_{nx} \sim U(0, 2)$.

Fitness Evaluation

In the case of minimization, the fittest individual is the one with the smallest value for the function being optimized; that is, the individual with the smallest value for the function $\sin(2\pi x)e^{-x}$. For maximization, it is the largest value. Alternatively, if each individual represents a vector of values for which a minimum has to be found, for example, a weight vector of a NN, then the MSE over a data set can be used as fitness measure.

Mutation

Mutation consists of adding a Gaussian random value to each element of an individual. For this example, each individual C_n is mutated using

$$O_{nx} = C_{nx} + \Delta C_{nx} \tag{11.2}$$

where $\Delta C_{nx} \sim N(0, \sigma_{nx}^2)$. The following choices for the variance σ_{nx}^2 exist:

- Use a static, small value for σ_{nx}^2.
- Let σ_{nx}^2 be large, initially, and decrease the variance with increase in the number of generations. This approach allows an exploration of a large part of the search space early in evolution, while ensuring small variations when an optimum is approached. Small variations near the optimum are necessary to prevent individuals from jumping over the optimum point.
- $\sigma_{nx}^2 = \mathcal{F}_{EP}(C_{nx})$, that is, the variance is equal to the error of the parent. The larger the error is, the more the variation in the offspring, with a consequent large move of the offspring from the weak parent. On the other hand, the smaller the parent's error, the less the offspring should be moved away from the parent. This is a simple approach to implementing a self-adaptive mutation step size.
- An alternative self-adaptive mutation step size is having a time-varying step size. To illustrate this, it is best to move to a general case where an individual

$\vec{C}_n$ consists of I genes (parameters). Let ΔC_{ni} be the mutation step size for the i-th gene of individual $\vec{C}_n$. Then, the lognormal self-adaptation of ΔC_{ni} is

$$\begin{aligned} \Delta C_{ni} &= \Delta C_{ni} e^{(\tau' N(0,1) + \tau N_i(0,1))} \\ O_{ni} &= C_{ni} + \Delta C_{ni} N_i(0,1) \end{aligned}$$

where $N_i(0,1)$ indicates that a new random number is generated for each gene. Good values for tau and tau' are [Bäck 1996]

$$\begin{aligned} \tau &= \frac{1}{\sqrt{2\sqrt{I}}} \\ \tau' &= \frac{1}{\sqrt{2I}} \end{aligned}$$

A problem with such self-adapting schemes is that ΔC_{ni} can become too small too quickly. To prevent this from happening, a dynamic lower limit to ΔC_{ni} can be used as developed in [Liang *et al.* 2001].

11.4 Assignments

1. Select a continuous function and compare the performance of a GA and EP in finding the minimum/maximum of that function.
2. Can an EP be used to learn the regular expression of a sequence of characters?
3. Explain how a FSM and EP can be used to evolve the behavior of a robot in walking out of a room full of obstacles.
4. Develop an EP to train feedforward neural networks.
5. The representation scheme used for FSM above can be reduced to use less bits. Suggest a way in which this can be accomplished.
6. Suggest ways in which premature convergence can be prevented for EP.

Chapter 12

Evolutionary Strategies

Rechenberg reasoned that, since biological processes have been optimized by evolution, and evolution is a biological process itself, then it must be the case that evolution optimizes itself [Rechenberg 1994]. Evolution strategies (ES), piloted by Rechenberg in the 1960s [Rechenberg 1973] and further explored by Schwefel [Schwefel 1975], are then based on the concept of *the evolution of evolution*. While ESs consider both genotypic and phenotypic evolution, the emphasis is toward the phenotypic behavior of individuals. Each individual is represented by its genetic building blocks and a set of strategy parameters that models the behavior of that individual in its environment. Evolution then consists of evolving both the genetic characteristics and the strategy parameters, where the evolution of the genetic characteristics is controlled by the strategy parameters. An additional difference between ESs and other EC paradigms is that changes due to mutation are only accepted in the case of success. In other words, mutated individuals are only accepted if the mutation resulted in improving the fitness of the individual. Also interesting in ESs is that offspring can also be produced from more than two parents.

A general ES algorithm is given in Section 12.1. Representation schemes are discussed in Section 12.2, cross-over is discussed in Section 12.3, mutation in Section 12.4 and selection in Section 12.5.

12.1 Evolutionary Strategy Algorithm

The following is an illustration of a general ES. This algorithm deviates from previous notation to conform to the ES conventions.

1. Let the generation be $g = 0$.
2. Initialize the population $C_g = \{\vec{C}_{g,n} | n = 1, \cdots, \mu\}$, where μ is the total number

of parent individuals.

3. Evaluate the fitness of each individual.

4. While no convergence

   (a) For $l = 1, \cdots, \lambda$, where λ is the number of offspring:

      i. choose $\rho \geq 2$ parents at random

      ii. perform cross-over on the genetic building blocks and the strategy parameters

      iii. mutate the genetic material and the strategy parameters of the offspring

      iv. evaluate the fitness of the offspring.

   (b) Select the μ best individuals from the offspring, or from the parents and the offspring, to form the next generation.

   (c) Let $g = g + 1$.

Each of the operators is explained in the sections that follow.

12.2 Chromosome Representation

The chromosome representation of each individual consists of two information types, i.e. genotypic and phenotypic. An individual is represented by the tuple (for ease of notation, the generation subscript is omitted)

$$C_n = (\vec{G}_n, S_n) \tag{12.1}$$

where $\vec{G}_n$ represents the genetic material and S_n represents the strategy parameters (or behavioral characteristics). Strategy parameters include mainly information about mutation, and have as their aim allowing the implementation of self-adaptive mutation step sizes. Strategy parameters that have been used in ESs include standard deviations of mutation step sizes and rotation angles:

- A **standard deviation**, associated with each individual, in which case an individual is represented as

  $$C_n = (\vec{G}_n, \sigma_n) \in \mathbb{R}^I \times \mathbb{R}_+ \tag{12.2}$$

  with $G_n \in \mathbb{R}^I$ (the individual has I genetic variables) and $\sigma_n \in \mathbb{R}_+$ (a positive scalar value).

- A **standard deviation** for each genetic variable of an individual, and **rotation angles** are associated with each individual, represented as

$$C_n = (\vec{G}_n, \vec{\sigma}_n, \omega_n) \in \mathbb{R}^I \times \mathbb{R}_+^I \times \mathbb{R}_+^{I(I-1)/2} \tag{12.3}$$

  where $\vec{\omega}_n$ is a vector of $I(I-1)/2$ rotational angles with $\omega_{nk} \in (0, 2\pi]$. With the assignment of a standard deviation σ_{ni} to each genetic variable G_{ni}, the preferred directions of search can only be established along the axes of the coordinate system. However, it is generally the case that the best search direction is not aligned on these axes. Therefor, optimal convergence is only achieved by chance when suitable mutations are correlated. Correlated mutations are promoted by using the rotational angles, thus preventing the search trajectory to fluctuate along the gradient of the objective function [Schwefel 1981].

  The rotational angles are used to represent the covariances among the I genetic variables in the genetic vector $\vec{G}_n$. Because the covariance matrix is symmetric, a vector can be used to represent the rotational angles instead of a matrix. The rotational angles are used to calculate an orthogonal rotation matrix, $T(\vec{\omega}_n)$, as

$$T(\vec{\omega}_n) = \prod_{i=1}^{I-1} \prod_{j=i+1}^{I} R_{ij}(\vec{\omega}_n) \tag{12.4}$$

  which is the product of $I(I-1)/2$ rotation matrices. Each rotation matrix $R_{ji}(\vec{\omega}_n)$ is a unit matrix with $r_{ii} = \cos(\omega_{nk})$ and $r_{ij} = -r_{ji} = -\sin(\omega_{nk})$, with $k = 1 \Leftrightarrow (i = 1, j = 2), k = 2 \Leftrightarrow (i = 1, j = 3), \cdots$.

The standard deviation and the orthogonal rotation matrix are used to determine the mutation step size (refer to Section 12.4).

The initial population is constructed randomly, with random initialization of the genetic material and the structural parameters.

12.3 Crossover Operators

Crossover is applied to both the genetic variables and the strategic parameters. Crossover is implemented differently from other EC paradigms. Two main approaches to cross-over exist for ES:

- **Local cross-over**, where one offspring is generated from two parents using randomly selected components of the parents.
- **Global cross-over**, where the entire population of individuals takes part in producing one offspring. Components are randomly selected from randomly selected individuals and used to generate the offspring.

In both cases, the recombination of genetic and strategic parameters is done in one of two ways:

- **Discrete recombination**, where the actual allele of parents is used to construct the offspring.
- **Intermediate recombination**, where the allele for the offspring is the midpoint between the allele of the parents (remember that floating-point representations are used).

12.4 Mutation operators

As with EP, Gaussian noise with zero mean is used to determine the magnitude of mutation. Both the genetic material, as encapsulated in $\vec{G}_n$, and the strategic parameters S_n are mutated. The mutation scheme depends on the type of strategic parameters used. Two schemes are discussed below, depending on whether correlation information is used or not:

- For the representation $C_n = (\vec{G}_n, \sigma_n)$ with no inclusion of correlation information, the mutation process is implemented in two stages:
  1. Mutate the standard deviation $\sigma_{g,n}$ for the current generation g and each individual C_n. Different methods exist to mutate deviations, of which the 1/5 success rule of Rechenberg was the first heuristic to be used [Rechenberg 1973]. This rule states that the ratio of successful mutations should be 1/5. Therefor, if the ration of successful mutations is larger than 1/5, the mutation deviation should increase; otherwise, the deviation should decrease. Thus,

$$\sigma_{g+t,n} = \begin{cases} c_d \sigma_{g,n} & \text{if } s_g < \frac{1}{5} \\ c_i \sigma_{g,n} & \text{if } s_g > \frac{1}{5} \\ \sigma_{g,n} & \text{if } s_g = \frac{1}{5} \end{cases}$$

     where s_g is the frequency of successful mutations over t iterations; c_d and c_i are constants. A successful mutation is one that results in an offspring with better fitness than the parent.
     Schwefel proposed the following mutation strategy for the standard deviations [Schwefel 1974]:

$$\sigma_{g+1,n} = \sigma_{g,n} e^{\tau \xi_\tau} \tag{12.5}$$

     where $\tau = \sqrt{I}$, and $\xi_\tau \sim N(0,1)$.
     Another different strategy was proposed by Fogel [Fogel 1992]:

$$\sigma_{g+1,n} = \sigma_{g,n}(1 + \tau \xi_\tau)$$

2. Mutate the genetic material $\vec{G}_{g,n}$ for each individual:

$$\vec{G}_{g+1,n} = \vec{G}_{g,n} + \vec{\sigma}_{g+1,n}\xi \tag{12.6}$$

where $\vec{\xi} \in \mathbb{R}_+^I$ with each $\xi_i \sim N(0,1)$.

- For the representation $C_n = (\vec{G}_n, \vec{\sigma}_n, \omega_n)$, where both the standard deviation and correlation information are used, mutation is a three-stage process:

1. Mutate the rotational angles $\omega_{g,nk}$ for each individual C_n:

$$\omega_{g+1,nk} = (\omega_{g,nk} + \phi\xi_{\omega,k}) \; mod \; 2\pi \tag{12.7}$$

where $\phi > 0$ and $\xi_{w,ki} \sim N(0,1)$ for $k = 1, \cdots, I(I-1)/2$.

2. Let $\vec{\sigma}_{g,n} \in \mathbb{R}_+^I$ represent the standard deviations of all genes, i.e. $\vec{\sigma}_{g,n} = (\sigma_{g,n1}, \cdots, \sigma_{g,nI})$. Let $S(\sigma) = diag(\sigma_{g,n1}, \cdots, \sigma_{g,nI})$ be a diagonal matrix representation of these deviations. Then, each standard deviation is mutated as

$$\sigma_{g+1,ni} = \sigma_{g,ni} e^{(\tau\xi_\tau + \eta\xi_{\sigma,i})} \tag{12.8}$$

where $\tau, \eta \in \mathbb{R}_+$ and $\xi_{\sigma,i} \sim N(0,1)$.

3. Mutate the genes, using

$$G_{g+1,n} = G_{g,n} + T(\omega_{g+1,n})S(\sigma_{g+1,n})\xi \tag{12.9}$$

where $T(\omega_{g,n})$ is the orthogonal rotation matrix calculated from equation (12.4) and ξ a normal random vector with $\xi_i \sim N(0,1)$ for $i = 1, \cdots, I$.

Schwefel found $(\phi, \tau, \eta) = (5\pi/180, (2\pi)^{-1/2}, (4\pi)^{-1/4})$ to be a good heuristic [Schwefel 1995].

Mutated individuals are only accepted if the fitness of the mutated individual is better than the original individual. That is,

$$G_{g+1,n} = \begin{cases} G_{g,n} + \sigma_{g+1,n}\xi & \text{if } \mathcal{F}_{ES}(G_{g,n} + \sigma_{g+1,n}\xi) \geq \mathcal{F}_{ES}(G_{g,n}) \\ G_{g,n} & \text{otherwise} \end{cases} \tag{12.10}$$

in the case of using only standard deviation as strategic parameters; and

$$G_{g+1,n} = \begin{cases} G_{g,n} + T(\omega_{g+1,n})S(\sigma_{g+1,n})\xi & \text{if} \mathcal{F}_{ES}(G_{g,n} + T(\omega_{g+1,n})S(\sigma_{g+1,n})\xi) \\ & \geq \mathcal{F}_{ES}(G_{g,n}) \\ G_{g,n} & \text{otherwise} \end{cases} \tag{12.11}$$

if correlation information is also used.

12.5 Selection Operators

For each generation λ offspring are generated from μ parents and mutated. After cross-over and mutation the individuals for the next generation are selected. Two strategies have been developed:

- $(\mu + \lambda)ES$: In this case the ES generates λ offspring from μ parents, with $1 \leq \mu \leq \lambda < \infty$. The next generation consists of the μ best individuals selected from μ parents and λ offspring. The $(\mu + \lambda)ES$ strategy implements elitism to ensure that the fittest parents survive to the next generation.
- $(\mu, \lambda)ES$: In this case, the next generation consists of the μ best individuals selected from the λ offspring. The $(\mu, \lambda)ES$ requires that $1 \leq \mu < \lambda < \infty$.

Using the notation above, the first ES was the (1+1)-ES developed by Rechenberg [Rechenberg 1973]. This ES consisted of only one parent from which one offspring is generated. For the (1+1)-ES, the 1/5 success rule was used to adapt the mutation deviations, and

$$\vec{G}_{g+1,n} = \vec{G}_{g,n} + \vec{\xi}$$

with $\xi_i \sim N(0, \sigma_g)$. The first Multimembered ES was the (μ+1)-ES developed by Schwefel. For the (μ+1)-ES, one offspring is generated from a population of μ parents by selecting two parents and performing one-point cross-over. The mutation deviation is updated as for the (1+1)-ES. Following on from these strategies are the $(\mu+\lambda)$-ESs. where fit parents have the chance to survive for many generations. The (μ, λ)-ESs were developed to exclude elitism.

12.6 Conclusion

While this chapter presented an overview of pure ES, Part IV presents the particle swarm optimization algorithm from swarm intelligence, which shares some of the aspects of ES.

Chapter 13

Differential Evolution

Differential evolution (DE) is a population-based search strategy very similar to standard evolutionary algorithms. The main difference is in the reproduction step used by DE, which presents the advantage that DE can operate on flat surfaces. DE has been applied successfully to the design of digital filters, mechanical design optimization and evolving game strategies.

This chapter gives a short overview of DE, with specific focus on the reproduction step, which consists of a new arithmetic mutation operator and a selection operator. The reproduction method is discussed in Section 13.1. A pseudocode algorithm is given in Section 13.2 to summarize DE.

13.1 Reproduction

Differential evolution does not make use of a mutation operator that depends on some probability distribution function, but introduces a new arithmetic operator which depends on the differences between randomly selected pairs of individuals. For each parent, $\vec{C}_{g,n}$, of generation g, an offspring, $\vec{O}_{g,n}$, is created in the following way: Randomly select three individuals from the current population, namely $\vec{C}_{g,n_1}, \vec{C}_{g,g_2}$ and $\vec{C}_{g,n_3}$, with $n_1 \neq n_2 \neq n_3 \neq n$ and $n_1, n_2, n_3 \sim U(1, \cdots, N)$. Select a random number $i \sim U(1, \cdots, I)$, where I is the number of genes (parameters) of a single chromosome. Then, for all parameters $j = 1, \cdots, I$, if $U(0, 1) < p_r$, or $j = i$, let

$$O_{g,nj} = C_{g,n_3 j} + \gamma(C_{g,n_1 j} - C_{g,n_2 j})$$

otherwise, let

$$O_{g,nj} = C_{g,nj}$$

In the above, p_r is the probability of reproduction (with $p_r \in [0, 1]$), γ is a scaling factor with $\gamma \in (0, \infty)$; $O_{g,nj}$ and $C_{g,nj}$ indicate respectively the j-th parameter of

the offspring and the parent.

Thus, each offspring consists of a linear combination of three randomly chosen individuals when $U(0,1) < p_r$; otherwise the offspring inherits directly from the parent. Even when $p_r = 0$, at least one of the parameters of the offspring will differ from the parent.

The mutation process above requires that the propulation consists of more than three individuals.

After completion of the mutation process, the next step is to select the new generation. For each parent of the current population, the parent is replaced with its offspring if the fitness of the offspring is better, otherwise the parent is carried over to the next generation.

13.2 General Differential Evolution Algorithm

A pseudocode algorithm is given below to summarize standard differential evolution:

1. Let $g = 0$, and initialize p_r and γ.
2. Initialize a population C_g of N individuals.
3. For each individual, $\vec{C}_{g,n}$, $(n = 1, \cdots, N)$:
   (a) select $n_1, n_2, n_3 \sim U(1, \cdots, N)$, with $n_1 \neq n_2 \neq n_3 \neq n$
   (b) select $i \sim U(1, \cdots, I)$
   (c) for $j = 1, \cdots, I$
   if $(U(0,1) < p_r$ or $j = i)$

$$O_{g,nj} = C_{g,n_3 j} + \gamma(C_{g,n_1 j} - C_{g,n_2 j})$$

   else

$$O_{g,nj} = C_{g,nj}$$

4. Select the new population C_{g+1} of N individuals:

$$\vec{C}_{g+1,n} = \begin{cases} \vec{O}_{g,n} & \text{if } \mathcal{F}_{DE}(\vec{O}_{g,n}) \leq \mathcal{F}_{DE}(\vec{C}_{g,n}) \\ \vec{C}_{g,n} & \text{otherwise} \end{cases}$$

5. Test for convergence. If the algorithm did not converge, go to step 3.

The DE algorithm uses the same convergence tests as for other EAs.

13.3 Conclusion

This chapter presented a short overview of differential evolution. Much more can be done on DE. For example, the effects of different schemes to select parents (e.g. tournament selection) and the new population can be investigated. These are left to the reader's imagination.

13.4 Assignments

1. Investigate the effect if γ in the equations above are randomly selected to be in the range $[0, 1]$.
2. Instead of using random selection for the parents, investigate the effect of using tournament selection.

Chapter 14

Cultural Evolution

Standard evolutionary algorithms have been successful in solving diverse and complex problems in search and optimization. The search process used by standard EAs is unbiased, using little or no domain knowledge to guide the search process. However, the performance of EAs can be improved considerably if domain knowledge is used to bias the search process. Domain knowledge then serves as a mechanism to reduce the search space by pruning undesirable parts of the solution space, and by promoting desirable parts. Cultural evolution (CE), based upon the principles of human social evolution, was developed as an approach to bias the search process with prior knowledge about the domain.

Evolutionary computing models biological evolution, which is based on the principle of genetic inheritance. In natural systems, genetic evolution is a slow process. Cultural evolution, on the other hand, enables societies to adapt to their changing environments at rates that exceed that of biological evolution. As an example, consider how fast a new hair style, music genre or clothing style catch on in a society or among different societies. The importance of culture in the efficacy of individuals within their society is also nicely illustrated by the difficulties an outsider experiences fitting into a different cultural society.

Culture can be defined as a system of symbolically encoded conceptual phenomena that are socially and historically transmitted within and between social groups [Durham 1994]. In terms of evolutionary computing, culture is modeled as the source of data that influences the behavior of all individuals within that population. This differs from EP and ES where the behavioral characteristics of individuals – for the current generation only – are modeled using phenotypes. Within cultural algorithms (CA), the culture stores the general behavioral traits of the population. Cultural information is then accessible to all the individuals of a population, and over many generations.

Cultural evolution algorithms model two search spaces, namely the population space

and the belief space. It is the latter that distinguishes CAs from standard EAs. The belief space models the cultural information about the population. Both the population and the beliefs evolve, with both spaces influencing one another. Section 14.1 discusses the belief space in more detail, while Section 14.2 discusses a specific type of CA. An example application is illustrated in Section 14.3.

14.1 Belief Space

Cultural evolution algorithms maintain two search spaces: the *population space* and the *belief space*. The population space is searched using any of the standard EAs, for example, a GA or an EP. The population space is therefore searched via genotypic and/or phenotypic evolution. The belief space, on the other hand, models the cultural behaviors of the entire population.

The belief space is also referred to as the *meme pool*, where a *meme* is a unit of information transmitted by behavioral means. The belief space serves as a global knowledge repository of behavioral traits. The meme within the belief space are generalizations of the experience of individuals within the population space. These experiential generalizations are accumulated and shaped over several generations, and not just one generation. These generalizations express the beliefs as to what the optimal behavior of individuals constitutes.

The belief space can effectively be used to perform a pruning of the population space. Each individual represents a point in the population search space: The knowledge within the belief space is used to move individuals away from undesirable areas in the population space toward more promising areas.

Some form of communication protocol is implemented to transfer information between the two search spaces. The communication protocol specifies operations that control the influence individuals have on the structure of the belief space, as well as the influence that the belief space has on the evolution process on the population level. This allows individuals to dictate their culture, causing culture to evolve also. On the other hand, the cultural information is used to direct the evolution on population level toward promising areas in the search space. It has been shown that the use of a belief space reduces computational effort substantially [Spector and Luke 1996, Rychtyckyj and Reynolds 1999, Reynolds 1999].

Various CAs have been developed, which differ in the data structures used to model the belief space, and the EA used on the population level. Typical symbolic representation schemes for cultural knowledge include semantic networks [Rychtyckyj and Reynolds 1999], logic, version spaces using lattices [Reynolds 1991] and memory indices [Teller 1994, Spector and Luke 1996]. Section 14.2 discusses a specific class of CAs, referred to as cultural algorithms.

14.2 General Cultural Algorithms

Cultural algorithms, as developed by Reynolds [Reynolds 1994], are a class of computational models of cultural evolution that supports dual inheritance: evolution occurs in two levels, namely the population level and the cultural level (belief space).

A pseudocode algorithm is given below, and illustrated in Figure 14.1:

1. Let generation be $g = 0$.
2. Initialize
   (a) population C_g
   (b) belief space B_g
3. While not converged
   (a) Evaluate the the fitness of each individual $\vec{C}_{g,n} \in C_g$.
   (b) Adjust(B_g,accept(C_g)).
   (c) Variate(C_g,influence(B_g)).
   (d) g = g + 1.
   (e) Select the new population C_g.

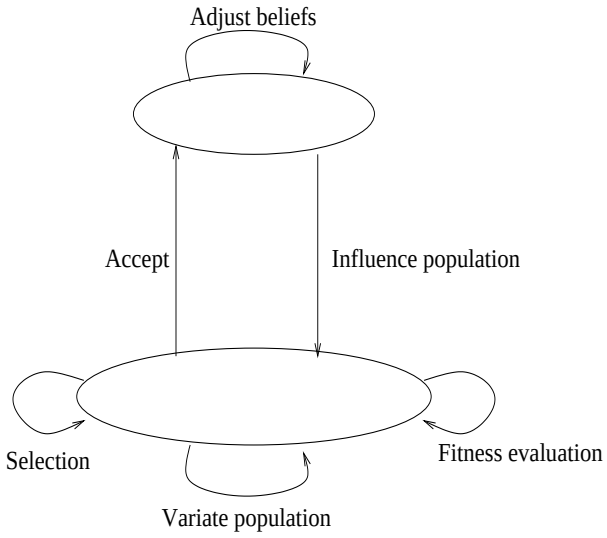


Figure 14.1: Cultural algorithm framework

At each iteration (each generation), individuals are first evaluated using the fitness function specified for the EA on the population level. An *acceptance* function is then used to determine which individuals from the current population have an influence

on the current beliefs. The experience of the accepted individuals is then used to *adjust* the beliefs (to simulate evolution of culture). The adjusted beliefs are then used to *influence* the evolution of the population. The *variation* operators (crossover and mutation) use the beliefs to control the changes in individuals. This is usually achieved through self-adapting control parameters, as functions of the beliefs.

The type of acceptance and influence function depends on the application domain. The next section considers one application of CAs to illustrate these concepts, and gives references to other application types.

14.3 Cultural Algorithm Application

Cultural algorithms have been applied to solve different problems, with one of the first applications modeling the evolution of agriculture in the Valley of Oaxaca (Mexico) [Reynolds 1979]. Other applications of cultural algorithms include concept learning [Sverdlik *et al.* 1992], real-valued function optimization [Michalewicz 1994], optimizing semantic networks [Rychtyckyj and Reynolds 1999], software testing [Ostrowski and Reynolds 1999], and assessing the quality of genetic programs [Cowan and Reynolds 1999].

This section overviews one application, i.e. using a CA for real-valued function optimization. Consider finding the minimum of the (fitness) function $f(x_1, x_2, \cdots, x_I)$, with $x_i \in [-\alpha_1, \alpha_2]$ and $\alpha_1, \alpha_2 \in [0, \infty]$. Assume that an EP is used on the population level: the resulting CA is referred to as a CAEP. The population space consists of N individuals, $\vec{C}_{g,1}, \cdots, \vec{C}_{g,N}$, each of I dimensions, for a specific generation g.

The belief space will contain the best individual since evolution, as well as the interval believed to house the minimum point. These will respectively be referred to as the situational knowledge component, $\mathcal{S}_g$, and the normative knowledge component, $\vec{\mathcal{N}}_g$ (for a specific generation g). While $\mathcal{S}_g$ contains only one component (the best individual), the normative knowledge component consists of I elements, one for each parameter x_i. For each element, $\mathcal{N}_{g,i}$, a minimum value $\mathcal{N}_{g,i}^{min}$ and a maximum value $\mathcal{N}_{g,i}^{max}$ are kept to indicate the range wherein it is believed the best value for that parameter is located. The initial normative knowledge component reflects the initial range constraints placed on the parameters. That is,

$$\begin{aligned} \mathcal{N}_{g,i}^{min} &= -\alpha_1 \\ \mathcal{N}_{g,i}^{max} &= \alpha_2 \end{aligned}$$

for all $i = 1, \cdots, I$.

The influence function is used in conjunction with the mutation operator to generate 10 offspring. In this application, the influence function determines the mutation step size. That is, the variance of the Gaussian distribution function used to determine

the step size, is a function of the range constraints. Individuals within the range constraints as given by the normative knowledge component have a small mutation variance. Individuals further away from the range constraints have a larger mutation variance, hence larger adjustments. Only the top $y\%$ of individuals are accepted to influence the belief space. Let $C_g^{'}$ denote the set of parents and offspring. Then, after the new generation has been selected,

$$\begin{aligned} \mathcal{S}_{g+1} &= \min_{n=1,\cdots,2N}\{\mathcal{S}_g, C_{g,n}^{'}\} \\ \mathcal{N}_{g+1,i}^{min} &= \min_{i=1,\cdots,2N}\{\mathcal{N}_{g,i}^{min}, C_{g,ni}^{'}\} \\ \mathcal{N}_{g+1,i}^{max} &= \max_{i=1,\cdots,2N}\{\mathcal{N}_{g,i}^{max}, C_{g,ni}^{'}\} \end{aligned}$$

under the constraint that

$$\mathcal{N}_{g,i}^{min} \leq \mathcal{N}_{g,i}^{max}$$

The above illustrates the steps of one generation.

14.4 Conclusion

Cultural evolution is still a young EC paradigm, and still needs much research to mine the potential in improving standard EAs.

14.5 Assignments

1. Discuss how CE can be used to train a NN.
2. What are the similarities and differences between CE and ES?
3. Discuss the validity of the following statement: The belief space used for CE can be likened to a blackboard system.

Chapter 15

Coevolution

Coevolution is the complementary evolution of closely associated species. The coevolution between two species is nicely illustrated using Holland's example of the competitive interaction between a plant and insects [Holland 1990]. Consider a certain species of plant living in an environment containing insects that eat the plant. The survival "game" consists of two parts: (1) to survive, the plant needs to evolve mechanisms to defend itself from the insects, and (2) the insects need the plant as food source to survive. Both the plant and the insects evolve in complexity to obtain characteristics that will enable them to survive. For example, the plant may evolve a tough exterior, but then the insect evolves stronger jaws. Next the plant may evolve a poison to kill the insects. Next generations of the insect evolve an enzyme to digest the poison. The effect of this coevolutionary process is that, with each generation, both the plant and the insects become better at their defensive and offensive roles. In the next generation, each species change in response to the actions of the other species during the previous generation.

The biological example described above is an example of *predator-prey* coevolution, where there is an inverse fitness interaction between the two species. A win for the one species means a failure for the other. To survive, the "losing" species adapt to counter the "winning" species in order to become the new winner. During this process the complexity of both the predator and the prey increases.

An alternative coevolutionary process is *symbiosis*, in which case the different species cooperate instead of competing. In this case a success in one species improves the survival strength of the other species. Symbiotic coevolution is thus achieved through a positive fitness feedback among the species that take part in this cooperating process.

In standard EAs, evolution is usually viewed as if the population attempts to adapt in a fixed physical environment. In contrast, coevolutionary (CoE) algorithms (CoEA) realize that in natural evolution the physical environment is influenced by other

independently-acting biological populations. Evolution is therefore not just locally within each population, but also in response to environmental changes as caused by other populations. Another difference between standard EAs and CoEAs is that EAs define the meaning of optimality through an absolute fitness function. This fitness function then drives the evolutionary process. On the other hand, CoEAs do not define optimality using a fitness function, but attempt to evolve an optimal species where optimality is defined as defeating opponents (in the case of predator-prey CoE).

Coevolution has been applied mostly to two-agent games, where the objective is to evolve a game strategy to beat an opponent. Similar principles have been used to evolve attack and defense strategies in military simulations. CoE has also been used successfully in classification, by evolving NNs and decision trees. Other applications include robot control, path planning, structural optimization and investment portfolio management. Being a very young field in evolution, other applications are expected to follow.

The coevolutionary process is discussed in more detail in Section 15.1. Section 15.2 discusses the measuring of relative fitness in the context of competitive coevolution. Symbiotic coevolution is discussed in Section 15.3 with reference to a cooperative coevolutionary genetic algorithm.

15.1 Coevolutionary Algorithm

Coevolution works to produce optimal competing (or cooperating) species through a pure bootstrapping process. Solutions to problems are found without knowledge from human experts, or any *a priori* information of how to solve the problem. The quality of each individual (trial solution) in the evolving population is evaluated by its peers in another population (or more populations) that operates in the same environment. No objective is specified through a fitness function.

The process of competitive coevolution can be described in the following way, considering two coevolutionary genetic algorithms. The initial individuals of both populations are initially extremely unfit. The fitness of each individual within its population is measured using an absolute fitness function. The first population tries to adapt to the initial environment as created by the second population. Simultaneously, the second population attempts to adapt to the environment created by the first population. The performance of each individual in the first population is then tested against the fitness of a sample of individuals from the second population. A relative fitness is thus calculated, which expresses the fitness of the individual with reference to that of individuals in the second population. The relative fitness of each individual in the second population is also computed. Reproduction within each population proceeds using these relative fitness values.

15.2 Competitive Fitness

Standard EAs use a user-defined fitness function that reflects optimality. The fitness of each individual is evaluated independently from any other population using this fitness function. In CoE, the driving force of the evolutionary process is through a relative fitness function that only expresses the performance of individuals in one population in comparison with individuals in another population. The only quantification of optimality is which population's individuals perform better. No fitness function that describes the optimal point is used.

The relative fitness of an individual can be calculated in different ways, as discussed in Section 15.2.1, using different types of sampling techniques from the competing population as discussed in Section 15.2.2.

15.2.1 Relative Fitness Evaluation

The following approaches can be followed to measure the relative fitness of each individual in a population. Assume that the two populations A and B coevolve, and the aim is to calculate the relative fitness of each individual $C_{A,n}$ of population A.

- **Simple fitness**: A sample of individuals is taken from population B, and the number of individuals $C_{B,m}$ for which $C_{A,n}$ is the winner, is counted. The relative fitness of $C_{A,n}$ is simply the sum of successes for $C_{A,n}$.

- **Fitness sharing**: A sharing function is defined to take into consideration similarity among the individuals of population A. The simple fitness of an individual is divided by the sum of its similarities with all the other individuals in that population. Similarity can be defined as the number of individuals that also beats the individuals from the population B sample. The consequence of the fitness sharing function is that unusual individuals are rewarded.

- **Competitive fitness sharing**: In this case the fitness of individual $C_{A,n}$ is defined as

$$\mathcal{F}(C_{A,n}) = \sum_{m=1}^{M} \frac{1}{N_m} \tag{15.1}$$

  where $C_{B,1}, \cdots, C_{B,m}$ form the population B sample, and N_m is the total number of individuals in population A that defeat individual $C_{B,m}$. The competitive fitness sharing method rewards those population A individuals that beat population B individuals which few other population A individuals could beat. It is therefore not necessarily the case that the best population A individual beats the most population B individuals.

15.2.2 Fitness Sampling

The relative fitness of individuals is evaluated against a sample of individuals from the competing population. The following sampling schemes have been developed:

- **All versus all sampling**, wherein each individual is tested against all the individuals of the other population.
- **Random sampling**, where the fitness of each individual is tested against a randomly selected group (consisting of one or more) of individuals from the other population. The random sampling approach is computationally less complex than all versus all sampling.
- **Tournament sampling**, which uses relative fitness measures to select the best opponent individual.
- **All versus best sampling**, where all the individuals are tested against the fittest individual of the other population.
- **Shared sampling**, where the sample is selected as those opponent individuals with maximum competitive shared fitness.

15.2.3 Hall of Fame

Elitism is a mechanism used in standard EAs to ensure that the best parents of a current generation survive to the next generation. To be able to survive for more generations, an individual has to be highly fit in almost every population. For coevolution, Rosin and Belew [Rosin and Belew 1996] introduced the *hall of fame* to extend elitism, to contain the best individual of each generation. By retaining the best individuals since evolution started, the opponent population has a much more difficult task to adapt to these best individuals.

15.3 Cooperative Coevolutionary Genetic Algorithm

The section above discussed a competitive strategy to coevolution where the objective of individuals in the one population is to beat the individuals in the other competing populations. This section discusses a cooperative approach to GAs, where information from different subpopulations are combined to form the solution to the problem. Each subpopulation therefore contributes to the solution.

The cooperative coevolutionary genetic algorithm (CCGA), developed by Potter [Potter 1997], is used to illustrate symbiotic interaction among subpopulations. The

island GA approach discussed in Section 9.6 (another kind of cooperation) distributes complete individuals over a set of subpopulations. Subpopulations evolve in parallel on complete individuals. Each subpopulation provides a complete solution to the problem. Construction of the solution per subpopulation is based on the evolutionary process of the subpopulations and an exchange of information by migrating individuals. This illustrates a different kind of symbiotic cooperation.

CCGA, on the other hand, distributes subcomponents (genes) of individuals over a set of subpopulations. These subpopulations are disjointed, each having the task of evolving a single (or limited set of) gene(s). A subpopulation therefore optimizes one parameter (or a limited number of parameters) of the optimization problem. Thus, no single subpopulation has the necessary information to solve the problem itself. Rather, information of all the subpopulations must be combined to construct a solution.

Within the CCGA, a solution is constructed by adding together the best individual from each subpopulation. The main problem is how to determine the best individual of a subpopulation, since individuals do not represent complete solutions. A simple solution to this problem is to keep all other components (genes) within a complete chromosome fixed and to change just the gene that corresponds to the current subpopulation for which the best individual is sought. For each individual in the subpopulation, the value of the corresponding gene in the complete chromosome is replaced with that of the individual. Values of the other genes of the complete chromosome are usually kept fixed at the previously determined best values.

The constructed complete chromosome is then a candidate solution to the optimization problem.

It has been shown that such a cooperative approach substantially improves the accuracy of solutions, and the convergence speed compared to non-cooperative, non-coevolutionary GAs.

15.4 Conclusion

The particle swarm and ant colony approaches discussed in the next Part are other excellent examples of cooperative behavior, which occurs through social exchange of information.

A complex coevolutionary system can be developed with both cooperating and competing populations.

15.5 Assignments

1. Design a CoEA for playing tic-tac-toe.
2. Explain the importance of the relative fitness function in the success of a CoEA.
3. Discuss the validity of the following statement: CCGA will not be successful if the genes of a chromosome are highly correlated.
4. Compare the different fitness sampling strategies with reference to computational complexity.
5. What will be the effect if fitness sampling is done only with reference to the hall of fame?
6. Why is shared sampling a good approach to calculate relative fitness?

Part IV

SWARM INTELLIGENCE

Suppose you and a group of friends are on a treasure finding mission. Each one in the group has a metal detector and can communicate the signal and current position to the n nearest neighbors. Each person therefore knows whether one of his neighbors is nearer to the treasure than him. If this is the case, you can move closer to that neighbor. In doing so, your chances are improved to find the treasure. Also, the treasure may be found more quickly than if you were on your own.

This is an extremely simple illustration of swarm behavior, where individuals within a swarm interact to solve a global objective in a more efficient manner than one single individual could.

A *swarm* can be defined as a structured collection of interacting organisms (or agents). Within the computational study of swarm intelligence, individual organisms have included ants, bees, wasps, termites, fish (in schools) and birds (in flocks). Within these swarms, individuals are relatively simple in structure, but their collective behavior can become quite complex. For example, in a colony of ants, individuals specialize in one of a set of simple tasks. Collectively, the actions and behaviors of the ants ensure the building of optimal nest structures, protecting the queen and larva, cleaning nests, finding the best food sources, optimizing attack strategies, etc.

The global behavior of a swarm of social organisms therefore emerges in a nonlinear manner from the behavior of the individuals in that swarm: Thus, there exists a tight coupling between individual behavior and the behavior of the entire swarm. The collective behavior of individuals shapes and dictate the behavior of the swarm. On the other hand, the behavior of the swarm determines the conditions under which an individual performs actions. These actions may change the environment, and thus the behaviors of that individual and its peers may also change. The conditions as determined by the swarm's behavior include spatial or temporal patterns.

The behavior of a swarm is not determined just by the behavior of individuals, independently from other individuals. Instead, the interaction among individuals plays a vital role in shaping the swarm's behavior. Interaction among individuals aids in refining experiential knowledge about the environment, and enhances the progress of the swarm toward optimality. The interaction, or cooperation, among individuals is determined *genetically* or through *social* interaction. Anatomical differences may,

for example, dictate the tasks performed by individuals. As an example, in a specific ant species minor ants feed the brood and clean the nest, whereas major ants cut large prey and defend the nest. Minor ants are smaller than, and morphologically different from major ants. Social interaction can be *direct* or *indirect*. Examples of direct interaction is through visual, audio or chemical contact. Indirect social interaction occurs when one individual changes the environment and the other individuals respond to the new environment. This type of interaction is referred to as *stigmergy*.

The social network structure of a swarm therefore forms an integral part of the existence of that swarm. It provides the communication channels through which experiential knowledge is exchanged among individuals. An amazing consequence of the social network structures of swarms is their ability to self-organize to form optimal nest structures, labor distribution, food gathering, etc.

Computational modeling of swarms has resulted in numerous successful applications, for example, function optimization, finding optimal routes, scheduling, structural optimization, and image and data analysis. Different applications originated from the study of different swarms. From these, most notable is the work on ant colonies and bird flocks. This Part gives an overview of these two swarm types. Chapter 16 discusses the *particle swarm optimization* approach, developed from simulations of the social behavior of bird flocks. Chapter 17 overviews *ant colony optimization*, which models mainly the pheromone trail-following behavior of ants.

Chapter 16

Particle Swarm Optimization

The particle swarm optimization (PSO) algorithm is a population-based search algorithm based on the simulation of the social behavior of birds within a flock. The initial intent of the particle swarm concept was to graphically simulate the graceful and unpredictable choreography of a bird flock [Kennedy and Eberhart 1995], the aim of discovering patterns that govern the ability of birds to fly synchronously, and to suddenly change direction with a regrouping in an optimal formation. From this initial objective, the concept evolved into a simple and efficient optimization algorithm.

In PSO, individuals, referred to as particles, are "flown" through hyperdimensional search space. Changes to the position of particles within the search space are based on the social-psychological tendency of individuals to emulate the success of other individuals. The changes to a particle within the swarm are therefore influenced by the experience, or knowledge, of its neighbors. The search behavior of a particle is thus affected by that of other particles within the swarm (PSO is therefore a kind of symbiotic cooperative algorithm). The consequence of modeling this social behavior is that the search process is such that particles stochastically return toward previously successful regions in the search space.

The operation of the PSO is based on the neighborhood principle as social network structure. Section 16.1 discusses the social interaction among neighbors. The PSO algorithm is discussed in detail in Section 16.2, while PSO system parameters are discussed in Section 16.3. Modifications to standard PSO are given in Section 16.4, and a cooperative approach to PSO is discussed in Section 16.5. Section 16.6 relates PSO to EC and Section 16.7 overviews some applications of PSO.

16.1 Social Network Structure: The Neighborhood Principle

The feature that drives PSO is social interaction. Individuals (particles) within the swarm learn from each other, and based on the knowledge obtained, move to become more similar to their "better" neighbors. The social structure for PSO is determined through the formation of neighborhoods. Individuals within a neighborhood communicate with one another.

Different neighborhood types have been defined and studied, where these neighborhoods are determined considering the labels of particles and not topological information such as Euclidean distances [Kennedy 1999]:

- The **star** topology: Each particle can communicate with every other individual, forming a fully connected social network as illustrated in Figure 16.1(a). In this case each particle is attracted toward the best particle (best problem solution) found by any member of the entire swarm. Each particle therefore imitates the overall best particle. The star neighborhood structure was used in the first version of the PSO algorithm, referred to as *gbest*.

- The **ring** topology: In this case each particle communicates with its n immediate neighbors. In the case of $n = 2$, a particle communicates with its immediately adjacent neighbors as illustrated in Figure 16.1(b). Each particle attempts to move closer to the best individual in its neighborhood. This version of the PSO algorithm is referred to as *lbest*. The *lbest* version can be adapted such that particles move toward their neighborhood best as well as the swarm best.

  The circle neighborhood structure has the advantage that a larger area of the search space is traversed, and convergence is slower [Eberhart *et al.* 1996, Kennedy 1999, Kennedy and Eberhart 1999].

- The **wheels** topology: Only one particle is connected to all others, which effectively isolates individuals from one another (illustrated in Figure 16.1(c)). Only this focal particle adjusts its position toward the best particle. If the adjustment to the focal particle results in an improvement in that individual's performance, the improvement is communicated to the other particles.

The neighborhood is determined based on the numerical index assigned to an individual and not on geometric measures such as position, or Euclidean distance.

The specifics of how the social exchange of information is modeled are given in the next section.

16.2 Particle Swarm Optimization Algorithm

A *swarm* consists of a set of particles, where each *particle* represents a potential solution. Particles are then flown through the hyperspace, where the position of each particle is changed according to its own experience and that of its neighbors. Let $\vec{x}_i(t)$ denote the position of particle P_i in hyperspace, at time step t. The position of P_i is then changed by adding a velocity $\vec{v}_i(t)$ to the current position, i.e.

$$\vec{x}_i(t) = \vec{x}_i(t-1) + \vec{v}_i(t) \tag{16.1}$$

The velocity vector drives the optimization process and reflects the socially exchanged information. Three different algorithms are given below, differing in the extend of the social information exchange. These algorithms summarize the initial PSO algorithms. Modifications to these are presented in Section 16.4.

16.2.1 Individual Best

For this version, each individual compares its current position to its own best position, *pbest*, only. No information from other particles is used.

1. Initialize the swarm, $P(t)$, of particles such that the position $\vec{x}_i(t)$ of each particle $P_i \in P(t)$ is random within the hyperspace, with $t = 0$.
2. Evaluate the performance $\mathcal{F}$ of each particle, using its current position $\vec{x}_i(t)$.
3. Compare the performance of each individual to its best performance thus far: if $\mathcal{F}(\vec{x}_i(t)) < pbest_i$ then
   (a) $pbest_i = \mathcal{F}(\vec{x}_i(t))$
   (b) $\vec{x}_{pbest_i} = \vec{x}_i(t)$
4. Change the velocity vector for each particle using:
   $$\vec{v}_i(t) = \vec{v}_i(t-1) + \rho(\vec{x}_{pbest_i} - \vec{x}_i(t))$$
   where ρ is a positive random number.
5. Move each particle to a new position:
   (a) $\vec{x}_i(t) = \vec{x}_i(t-1) + \vec{v}_i(t)$
   (b) $t = t + 1$
6. Go to step 2, and repeat until convergence.

In the algorithm above, the further away the particle is from its previously found best solution, the larger the change in velocity to return the individual toward its best solution. The upper limit of the random value ρ is a system parameter specified by the user. The larger the upper limit of ρ, the more the trajectory of the particles oscillates. Smaller values of ρ ensure smooth trajectories.

16.2.2 Global Best

The global best version, *gbest*, of PSO reflects the star neighborhood structure. The social knowledge used to drive the movement of particles includes the position of the best particle from the entire swarm. In addition, each particle uses its history of experiences in terms of its own best solution thus far. In this case the algorithm changes to:

1. Initialize the swarm, $P(t)$, of particles such that the position $\vec{x}_i(t)$ of each particle $P_i \in P(t)$ is random within the hyperspace, with $t = 0$.
2. Evaluate the performance $\mathcal{F}$ of each particle, using its current position $\vec{x}_i(t)$.
3. Compare the performance of each individual to its best performance thus far: if $\mathcal{F}(\vec{x}_i(t)) < pbest_i$ then
   (a) $pbest_i = \mathcal{F}(\vec{x}_i(t))$
   (b) $\vec{x}_{pbest_i} = \vec{x}_i(t)$
4. Compare the performance of each particle to the global best particle: if $\mathcal{F}(\vec{x}_i(t)) < gbest$ then
   (a) $gbest = \mathcal{F}(\vec{x}_i(t))$
   (b) $\vec{x}_{gbest} = \vec{x}_i(t)$
5. Change the velocity vector for each:
   $$\vec{v}_i(t) = \vec{v}_i(t-1) + \rho_1(\vec{x}_{pbest_i} - \vec{x}_i(t)) + \rho_2(\vec{x}_{gbest} - \vec{x}_i(t))$$
   where ρ_1 and ρ_2 are random variables. The second term above is referred to as the cognitive component, while the last term is the social component.
6. Move each particle to a new position:
   (a) $\vec{x}_i(t) = \vec{x}_i(t-1) + \vec{v}_i(t)$
   (b) $t = t + 1$
7. Go to step 2, and repeat until convergence.

The further away a particle is from the global best position and its own best solution thus far, the larger the change in velocity to move the particle back toward the best solutions. The random variables ρ_1 and ρ_2 are defined as $\rho_1 = r_1 c_1$ and $\rho_2 = r_2 c_2$, with $r_1, r_2 \sim U(0,1)$, and c_1 and c_2 are positive acceleration constants. Kennedy has studied the effect of the random variables ρ_1 and ρ_2 on the particle trajectories, and asserted that $c_1 + c_2 \leq 4$ [Kennedy 1998]. If $c_1 + c_2 > 4$, velocities and positions explode toward infinity.

16.2.3 Local Best

The local best version of PSO, *lbest*, reflects the circle neighborhood structure. Particles are influenced by the best position within their neighborhood, as well as their own past experience. Only steps 4 and 5 change by replacing *gbest* with *lbest*.

While *lbest* is slower in convergence than *gbest*, *lbest* results in much better solutions [Eberhart *et al.* 1996, Shi and Eberhart 1999] and searches a larger part of the search space.

16.2.4 Fitness Calculation

Step 2 of the algorithms above measures the performance of each particle. Here a function is used which measures the closeness of the corresponding solution to the optimum. In EC terminology, this refers to the fitness function. For example, if the objective is to find the minimum of the function $f(x_1, x_2) = \sin x_1 \sin x_2 \sqrt{x_1 x_2}$, the "fitness" function is the function $f(x_1, x_2)$.

16.2.5 Convergence

The algorithms above continue until convergence has been reached. Usually, a PSO algorithm is executed for a fixed number of iterations, or fitness function evaluations. Alternatively, a PSO algorithm can be terminated if the velocity changes are close to zero for all the particles, in which case there will be no further changes in particle positions.

16.3 PSO System Parameters

Standard PSO is influenced by six system parameters, namely the dimension of the problem, number of individuals, upper limit of ρ, and upper limit on the maximum velocity, neighborhood size and inertia weight. PSO has shown to perform better on higher-dimensional problems [Angeline 1998, Van den Bergh 1999]. The influence

of the upper limit has been discussed previously. Other system parameters are discussed below:

- **Maximum velocity,** V_{max}: An upper limit is placed on the velocity in all dimensions. This upper limit prevents particles from moving too rapidly from one region in search space to another. If $v_{ij}(t) > V_{max}$ then $v_{ij}(t) = V_{max}$, or if $v_{ij}(t) < -V_{max}$ then $v_{ij}(t) = -V_{max}$, where $v_{ij}(t)$ is the velocity of particle P_i at time step t in dimension j. Note that V_{max} does not place a limit on the position of a particle, only on the steps made in the hyperdimensional search space.

  V_{max} is usually initialized as a function of the range of the problem. For example, if the range of all x_{ij} is $[-50, 50]$, then V_{max} is proportional to 50. Clerc and Kennedy have shown that V_{max} is not necessary if [Clerc and Kennedy 2002]

$$\vec{v}_i(t) = \kappa(\vec{v}_i(t-1) + \rho_1(\vec{x}_{pbest} - \vec{x}_i(t)) + \rho_2(\vec{x}_{gbest} - \vec{x}_i(t))) \tag{16.2}$$

  where

$$\kappa = 1 - \frac{1}{\rho} + \frac{\sqrt{|\rho^2 - 4\rho|}}{2} \tag{16.3}$$

  with $\rho = \rho_1 + \rho_2 > 4.0$; κ is referred to as the *constriction coefficient*.

- **Neighborhood size**: The *gbest* version is simply *lbest* with the entire swarm as the neighborhood. The *gbest* is more susceptible to local minima, since all individuals are pulled toward that solution. The smaller the neighborhood radius, and the more neighborhoods can be used, the less susceptible PSO is to local minima. A larger part of search space is traversed, and no one solution has an influence on all particles. The more neighborhoods there are, however, the slower the convergence.

- **Inertia weight**: Improved performance can be achieved through application of an inertia weight applied to the previous velocity:

$$\vec{v}_i(t) = \phi\vec{v}_i(t-1) + \rho_1(\vec{x}_{pbest} - \vec{x}_i(t)) + \rho_2(\vec{x}_{gbest} - \vec{x}_i(t)) \tag{16.4}$$

  where ϕ is the inertia weight. The inertia weight controls the influence of previous velocities on the new velocity. Large inertia weights cause larger exploration of the search space, while smaller inertia weights focus the search on a smaller region. Typically, PSO is started with a large inertia weight, which is decreased over time.

A final point on convergence of PSO is now in order: the two most important parameters that drive the behavior of the PSO process are the inertia weight ϕ and the acceleration constants c_1 and c_2. Studies have shown that PSO does not converge for all combinations of values for these parameters [Clerc and Kennedy 2002,

Van den Bergh 2002]. To ensure convergence, the following relation must hold if velocity clamping is not used [Van den Bergh 2002]:

$$\phi > \frac{1}{2}(c_1 + c_2) - 1 \tag{16.5}$$

with $\phi \leq 1$. PSO exhibits cyclic or divergent behavior if equation (16.5) is not satisfied.

16.4 Modifications to PSO

Original developments in PSO resulted in the three PSO versions discussed in Section 16.2, namely pbest, lbest and gbest. Recent research resulted in some modifications to the original PSO algorithms, mainly to improve convergence and to increase diversity. Some of these modifications are discussed in this section, while others have been discussed in the previous section.

16.4.1 Binary PSO

Most PSO implementations are for continuous search spaces. However, Kennedy and Eberhart introduced a binary PSO where the positions of particles in search space can only take on values from the set $\{0, 1\}$ [Kennedy and Eberhart 1997]. For the binary PSO, particle positions are updated using

$$x_{ij}(t+1) = \begin{cases} 0 & \text{if } r_i(t) \geq f(v_{ij}(t)) \\ 1 & \text{if } r_i(t) < f(v_{ij}(t)) \end{cases}$$

where

$$f(v_{ij}(t)) = \frac{1}{1 + e^{-v_{ij}(t)}}$$

and $x_{ij}(t)$ is the value of the j-th parameter of particle P_i at time step t, $v_{ij}(t)$ is the corresponding velocity and $r_i(t) \sim U(0, 1)$.

The velocity update equation is the standard update equation. It is important to note that the velocities should be clamped to the range $[-4, 4]$ to prevent saturation of the sigmoid function [Eberhart and Kennedy 2001].

16.4.2 Using Selection

Angeline showed that PSO can be improved for certain classes of problems by adding a selection process similar to that which occurs in evolutionary computing [Angeline 1999]. The following selection scheme was implemented:

1. For each particle in the swarm, score the performance of that particle with respect to a randomly selected group of k particles.
2. Rank the particles according to these performance scores.
3. Select the top half of the particles and copy their current positions onto that of the bottom half of the swarm, without changing the personal best values of the bottom half of the swarm.

The selection process above is executed before the velocity updates are calculated.

The approach above improves the local search capabilities of PSO, and reduces diversity – which is contradictory to the objective of natural selection.

16.4.3 Breeding PSO

Further modifications to PSO have been made by adding a reproduction step to the standard PSO, referred to as breeding [Løvbjerg *et al.* 2001]. The breeding PSO is summarized below:

1. Calculate the particle velocities and new positions.
2. To each particle, assign a breeding probability p_b.
3. Select two particles as parents and produce two offspring using the arithmetic cross-over operator as explained below. Assume particles P_a and P_b are selected to produce offspring, they are replaced with the offspring as follows:

$$\begin{aligned}
\vec{x}_a(t+1) &= r_1\vec{x}_a(t) + (1-r_1)\vec{x}_b(t) \\
\vec{x}_b(t+1) &= r_1\vec{x}_b(t) + (1-r_1)\vec{x}_a(t) \\
\vec{v}_a(t+1) &= \frac{\vec{v}_a(t)+\vec{v}_b(t)}{||\vec{v}_a(t)+\vec{v}_b(t)||}||\vec{v}_a(t)|| \\
\vec{v}_b(t+1) &= \frac{\vec{v}_a(t)+\vec{v}_b(t)}{||\vec{v}_a(t)+\vec{v}_b(t)||}||\vec{v}_b(t)||
\end{aligned}$$

   where $r_1 \sim U(0,1)$.
4. Set the personal best position of each particle involved in the breeding process to its current position.

It is important to note that the parent selection process does not depend on the fitness of particles, thereby preventing the best particles from dominating the breeding process, and preventing premature convergence.

16.4.4 Neighborhood Topologies

The lbest PSO algorithm is based on the principle of forming neighborhoods of particles, where particles within a neighborhood move toward their own best positions and the best solution of the entire neighborhood. The lbest algorithm as proposed by Eberhart and Kennedy makes use of the indices of particles to determine neighborhoods. Suganthan proposed a different approach where spatial neighborhoods are formed, based on the spatial distance between particles [Suganthan 1999]. A particle P_b is said to be in the neighborhood of particle P_a if

$$\frac{||\vec{x}_a - \vec{x}_b||}{d_{max}} < \xi$$

where d_{max} is the largest distance between any two particles, and

$$\xi = \frac{3t + 0.6t_{max}}{t_{max}}$$

with t the current iteration number and t_{max} the maximum number of iterations.

The spatial neighborhood strategy has the effect of having smaller neighborhoods initially, with the size of neighborhoods increasing with increasing number of iterations, approaching gbest PSO as $t \rightarrow t_{max}$. The advantage of this is obvious: the initial smaller neighborhoods increase diversity, with larger parts of the search space being covered in the initial phases of the optimization process; also, premature convergence is less likely to occur.

Kennedy proposed stereotyping strategies where different neighborhood topologies are formed [Kennedy 1999]. These topologies were discussed briefly in Section 16.1.

16.5 Cooperative PSO

The PSO algorithms discussed thus far solve optimization problems by using J-dimensional particles, where J is the number of parameters to be optimized, i.e. the number of components of the final solution. A swarm therefore has the task of finding optimal values for all J parameters. A cooperative approach has been developed in [Van den Bergh and Engelbrecht 2000, Van den Bergh and Engelbrecht 2001, Van den Bergh 2002] similar to the CCGA approach of Potter (refer to Section 9.6). For the cooperative PSO (CPSO), the J parameters to be optimized can be split into J swarms, where each swarm optimizes only one parameter of the problem. The optimization process within each of the J swarms occur using any of the PSO algorithms discussed previously.

The difficulty with the CPSO algorithm is how to evaluate the fitness of these one-dimensional particles within each swarm. The fitness of each particle of swarm S_j

cannot be computed in isolation from the other swarms, since a particle in a specific swarm represents just one component of the complete J-dimensional solution. A solution to this problem is to construct a context vector from the other J-1 swarms by taking the global best particle from each of these $J-1$ swarms. The fitness of particles in swarm S_j is then calculated by replacing the j-th component in the context vector with that of the particle being evaluated. This approach to the evaluation of fitness promotes cooperation among the different swarms, since each swarm contributes to the context vector.

It is important to note that the CPSO algorithm is mostly applicable to problems where the parameters to be optimized are independent of one another.

16.6 Particle Swarm Optimization versus Evolutionary Computing and Cultural Evolution

PSO has its roots in several disciplines, including artificial life, evolutionary computing and swarm theory. This section illustrates the similarities and differences between PSO and EC. Both paradigms are optimization algorithms which use adaptation of a population of individuals, based on natural properties. In both PSO and EC the search space is traversed using probabilistic transition rules.

There are several important differences between PSO and EC. PSO has memory, while EC has no memory. Particles keep track of their best solutions, as well as that of their neighborhood. This history of best solutions plays an important role in adjusting the positions of particles. Additionally, the previous velocities are used to adjust positions. While both approaches are based on adaptation, changes are driven through learning from peers in the case of PSO, and not through genetic recombination and mutations. PSO uses no fitness function to drive the search process. Instead, the search process is guided by social interaction among peers.

PSO can more closely be related to CE. In this case the population space is searched using an EP, where mutation is a function of previous mutations and distances from the best solutions. The cultural beliefs are defined by the best solutions per individual and neighborhood. In the case of more than one neighborhood, the population can be viewed as consisting of subproblems, each representing one neighborhood.

16.7 Applications

PSO has been used mostly to find the minima and maxima of nonlinear functions [Shi and Eberhart 1999]. PSO has also been used successfully to train NNs [Eberhart *et al.* 1996, Kennedy and Eberhart 1999, Engelbrecht and Ismail 1999,

Van den Bergh 1999, Van den Bergh and Engelbrecht 2001]. In this case, each particle presents a weight vector, representing one NN. The performance of a particle is then simply the MSE over the training and test sets. A PSO has also been used successfully for human tremor analysis in order to predict Parkinson's disease [Shi and Eberhart 1999].

This section illustrates the application of PSO to find the minimum of the function $f(x_1, x_2) = x_1^2 + x_2^2$, for which the minimum is at $x_1 = 0, x_2 = 0$. Particles are flown in two-dimensional space. Consider the initial swarm of particles as represented by the dots in Figure 16.2. The optimum point is indicated by a cross. Figure 16.2(a) illustrates the *gbest* version of PSO. Particle a is the current global best solution. Initially the *pbest* of each individual is its current point. Therefor, only particle a influences the movement of all the particles. The arrows indicate the direction and magnitude of the change in positions. All the particles are adjusted toward particle a. The *lbest* version, as illustrated in Figure 16.2(b), shows how particles are influenced by their immediate neighbors. To keep the graph readable, only some of the movements are illustrated. In neighborhood 1, both particles a and b move toward particle c, which is the best solution within that neighborhood. Considering neighborhood 2, particle d moves toward f, so does e. For the next iteration, e will be the best solution for neighborhood 2. Now d and f move toward e as illustrated in Figure 16.2(c) (only part of the solution space is illustrated). The blocks represent the previous positions. Note that e remains the best solution for neighborhood 2. Also evident is the movement toward the minimum, although slower as for *gbest*.

16.8 Conclusion

PSO has already shown to be efficient and robust, even considering the simplicity of the algorithm. Much research is still needed to tap the benefits of this optimization process.

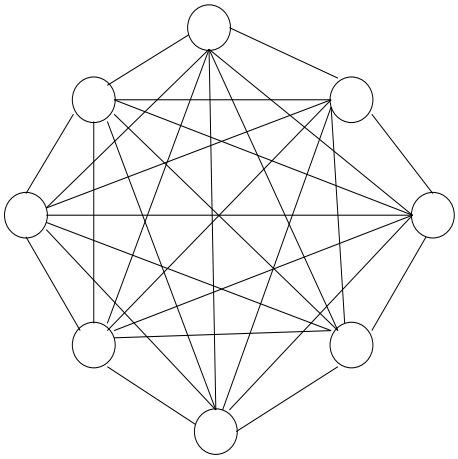
16.9 Assignments

1. Does PSO adhere to all the characteristics of swarms of social organisms?
2. Is PSO a special kind of EP?
3. Is PSO a special kind of CA?
4. What are the main differences between SI and EC?
5. Discuss how PSO can be used to train a NN.
6. Discuss how PSO can be used to cluster data.

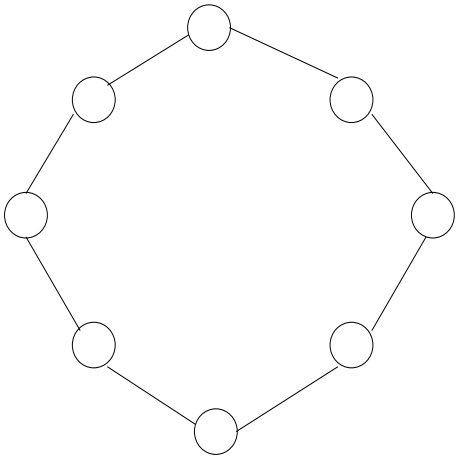
7. Why is it better to base the calculation of neighborhoods on the index assigned to particles and not on geometrical information such as Euclidean distance?

8. Explain how a PSO can be used to approximate functions using a n-th order polynomial.

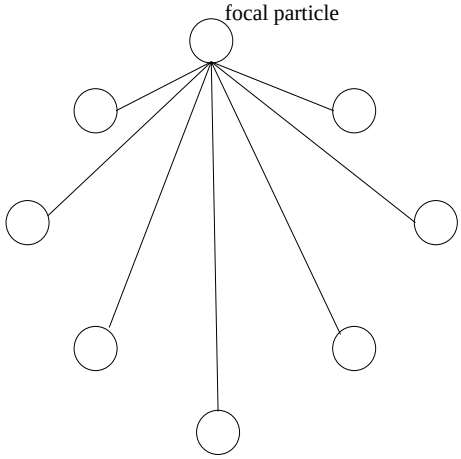
9. Show how a PSO can be used to solve systems of equations.



(a) Star Neighborhood Structure

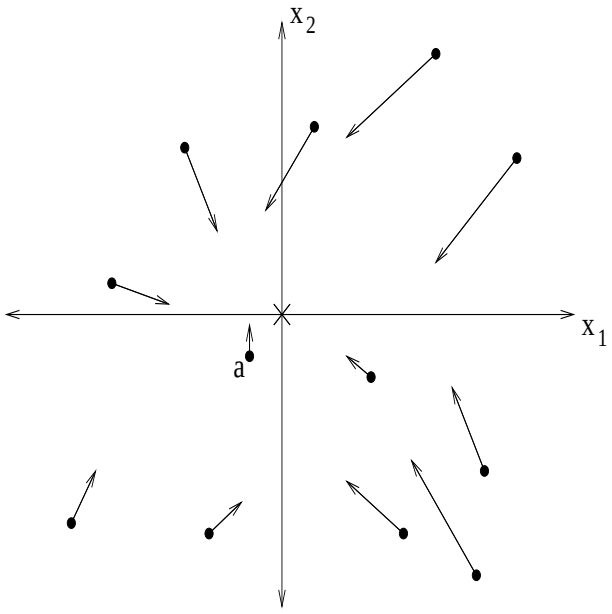


(b) Ring Neighborhood Structure

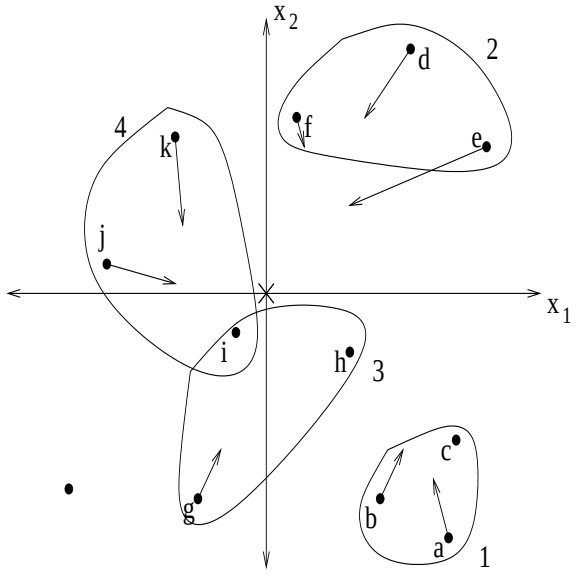


(c) Wheel Neighborhood Structure

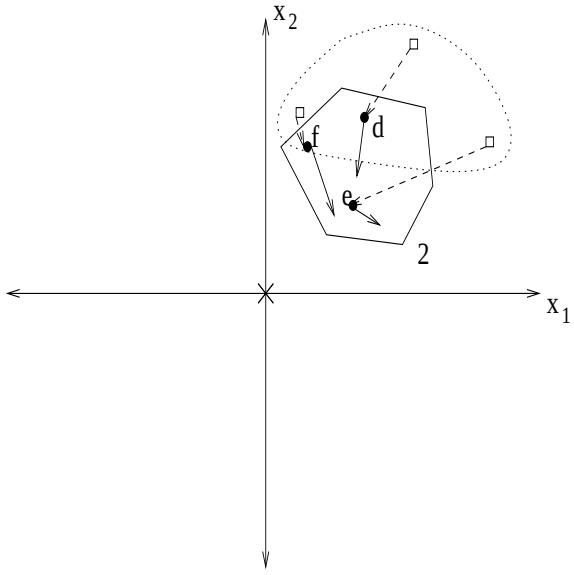
Figure 16.1: Neighborhood structures for particle swarm optimization [Kennedy 1999]



(a) Global Best Illustrated



(b) Local Best Illustrated – Initial Swarm



(c) Local Best – Second Swarm

Figure 16.2: *gbest* and *lbest* illustrated

Chapter 17

Ant Colony Optimization

The previous chapter discussed particle swarm optimization, which is modeled on the choreography of relatively small swarms where all individuals have the same behavior and characteristics. This chapter considers swarms that consist of large numbers of individuals, where individuals typically have different morphological structures and tasks – but all contributing to a common goal. Such swarms model distributed systems where components of the system are capable of distributed operation.

It is roughly estimated that the earth is populated by 10^8 living organisms, of which only 2% are social insects [Dorigo 1999]. That is, only 2% of all insects live in swarms where social interaction is the most important aspect to ensure survival. These insects include all ant and termite species, and some bees and wasps species. Of these social insects, 50% are ants. Ant colonies consist of from 30 to millions of individuals. This chapter concentrates on ants, and how modeling the behavior of ants can be used to solve real-world problems.

Section 17.1 discusses, briefly, the different tasks performed in a typical ant colony. The food collection behavior of ants is discussed in detail in Section 17.2, and illustrates its application to optimization in Section 17.3. Section 17.4 shows how ant colonies can be used for data clustering. The chapter concludes with a summary of applications in Section 17.5, including routing in telecommunications networks, data clustering, and robotics.

17.1 The "Invisible Manager" (Stigmergy)

Operation within an ant colony involves several different tasks, performed by different ant groups. The main tasks within a colony include:

- reproduction – the task of the queen

- defense – done by soldier ants
- food collection – the task of specialized worker ants
- brood care – the task of specialized worker ants
- nest brooming (including cemetery maintenance) – the task of specialized worker ants
- nest building and maintenance – the task of specialized worker ants.

It seems as if the distribution and execution of these tasks occur magically, without a globally centered command center – which is, in fact, the case: distribution and execution of tasks are based on anatomical differences and stigmergy. Anatomical differences, such as size and larger jaw structures, for example, distinguish between army ants and food collectors.

The distributed behavior within a colony is referred to as *stigmergy*. Natural stigmergy is characterized by [Dorigo 1999] as:

1. The lack of central coordination.
2. Communication and coordination among individuals in a colony are based on local modifications of the environment.
3. Positive feedback, which is a reinforcement of actions (e.g. the trail-following behavior to collect food).

Algorithmic modeling of ant colonies is based on the concept of *artificial stigmergy*, defined by Dorigo and Di Caro as the "*indirect communication mediated by numeric modifications of environmental states which are only locally accessible by the communicating agents*" [Dorigo and Di Caro 1999]. The essence of modeling ant colonies, or rather aspects of the operation of ant colonies, is to find a mathematical model which accurately describes the stigmergetic characteristics of the corresponding ant individuals. The next section concentrates on one task, namely food collection, and shows how the pheromone dropping and following of ants can be modeled.

17.2 The Pheromone

One of the first questions investigated with regard to ant behavior was that of food collection. Ants have the ability to always find the shortest path between their nest and the food source. Several experiments have been conducted to study this behavior. These experiments are visualized in Figure 17.1. Dots in this figure indicate ants. The experiments consisted of building paths of different lengths between a nest

and a food source, and monitoring the number of ants following each path. Initially, paths are chosen randomly. It was then observed that, with time, more and more ants follow the shorter path.

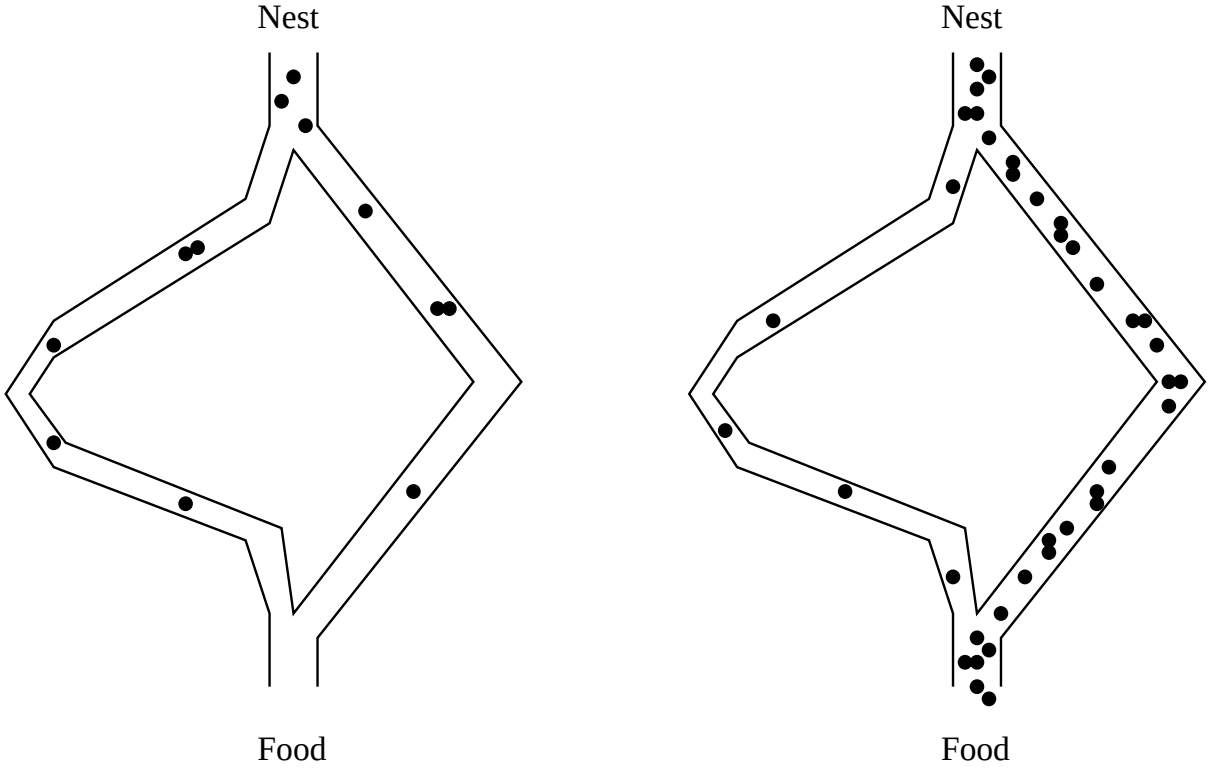


Figure 17.1: Pheromone trail following of ants

This behavior can be explained by the dropping of pheromones by each ant. During their search for food, and on return from the food source to the nest, each ant drops a pheromone deposit on the path. To select a path to follow, ants follow that path with the largest pheromone concentration. The shortest path will have stronger pheromone deposits, since ants return on that path from the food source quicker (dropping more pheromone on their way back to the nest), than the longer path. Also, pheromone deposits evaporate with time. The strength of the pheromone deposits on the longer path will decrease more quickly than for the shorter path.

17.3 Ant Colonies and Optimization

The modeling of pheromone trail following behavior is illustrated next, by solving the traveling salesman problem (TSP). At each city, the task of the ant is to choose the next city on the route, based on some probabilistic rule as a function of the pheromone deposits. Initially, the choice is random, achieved by initializing the

pheromone amounts on each route to a positive, small random value.

A simple transition rule for choosing the next city to visit, is

$$\Phi_{ij,k}(t) = \begin{cases} \frac{\tau_{ij}(t)^\alpha}{\sum_{c \in C_{i,k}} \tau_{ic}(t)^\alpha} & \text{if } j \in C_{i,k} \\ 0 & \text{if } j \notin C_{i,k} \end{cases} \tag{17.1}$$

where $\tau_{ij}(t)$ is the pheromone intensity on edge (i, j) between cities i and j, the k-th ant is denoted by k, α is a constant, and $C_{i,k}$ is the set of cities ant k still have to visit from city i.

The transition rule above can be improved by including local information on the desirability of choosing city j when currently in city i, i.e.

$$\Phi_{ij,k}(t) = \frac{\tau_{ij}(t)^\alpha \eta_{ij}^\beta}{\sum_{c \in C_{i,k}} \tau_{ic}(t)^\alpha \eta_{ic}^\beta} \tag{17.2}$$

where α and β are adjustable parameters that control the weight of pheromone intensity and local information, and

$$\eta_{ij} = \frac{1}{d_{ij}}$$

with d_{ij} the Euclidean distance between cities i and j (using their coordinates on a two-dimensional map),

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Note that the values of $\Phi_{ij,k}$ may be different for different ants at the same city, since ants may travel different routes to the same city. At the end of each route, T_k, constructed by ant k, the pheromone intensity τ_{ij} on the edges of that route is updated, using

$$\tau_{ij}(t+1) = (1 - \rho)\tau_{ij}(t) + \Delta\tau_{ij}(t) \tag{17.3}$$

where

$$\Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij,k}(t)$$

is the sum of pheromone deposits $\Delta\tau_{ij,k}(t)$ of each ant, defined as

$$\Delta\tau_{ij,k}(t) = \begin{cases} Q/L_k(t) & \text{if } (i,j) \in T_k(t) \\ 0 & \text{if } (i,j) \notin T_k(t) \end{cases}$$

The parameter Q has a value of the same order of the length of the optimal route, $L_k(t)$ is the length of the route traveled by ant k, and m is the total number of ants.

The constant $\rho \in [0, 1]$ in equation (17.3) is referred to as the forgetting factor, which models the evaporation over time of pheromone deposits.

The modeling of the behavior of ants usually introduces several parameters for which optimal values must be obtained. The most important of these parameters include the forgetting factor to model pheromone evaporation, and the number of ants used. Too many ants increase computational complexity, and result in fast convergence to suboptimal trails. On the other hand, too few ants limit the synergetic effects of cooperation. In addition to these parameters, equation (17.2) introduced the parameters α and β. A good balance between these parameters should be achieved: if $\beta = 0$, only pheromone information is used, which may lead to suboptimal paths; if $\alpha = 0$, no pheromone information is used, and the approach corresponds to a stochastic greedy search.

To summarize this section on ant colony optimization (ACO) for the TSP, a pseudo-code algorithm is given below (adapted from [Dorigo and Di Caro 1999]).

1. Initialize the pheromone deposits on each edge (i, j) between cities i and j to small positive random values, i.e. $\tau_{ij}(0) \sim U(0, max)$.
2. Place all ants $k \in 1, \cdots, m$ on the originating city.
3. Let T^+ be the shortest trip, and L^+ the length of that trip.
4. For $t = 1$ to t_{max} do the following:
   (a) For each ant, build the trip $T_k(t)$ by choosing the next city $n - 1$ times (n is the number of cities), with probability $\Phi_{ij,k}(t)$.
   (b) Compute the length of the route, $L_k(t)$, of each ant.
   (c) If an improved route is found, update T^+ and L^+.
   (d) Update the pheromone deposits on each edge using equation (17.3).
5. Output the shortest route T^+.

While this section discussed ACO with reference to the TSP, the next section overviews some of the applications of modeling ant behavior.

17.4 Ant Colonies and Clustering

In the previous sections, the foraging behavior of ants was shown to be useful for solving discrete optimization problems. This section explains how the clustering and sorting behavior of ants can be used to design new data clustering algorithms.

Several ant species cluster their corpses into "cemeteries" in an effort to clean up their nests. Experimental work illustrated that ants cluster corpses, which are initially randomly distributed in space, into clusters within a few hours. While this

behavior is still not that well understood, it seems that some feedback mechanism determines the probability that a worker ant will pick up or drop a corpse. Such behavior is easily modeled to produce an algorithmic clustering approach [Bonabeau *et al.* 1999].

The general idea is that a number of artificial ants (or agents) walk around in search space and pick up items, or drop an item based upon some probability measure. For sake of simplicity, assume that the environment has only one type of object, with instances of that object randomly distributed over the search space. The probability p_p for a randomly selected unladen agent to pick up an object is expressed as

$$p_p = (\frac{k_1}{k_1 + f})^2$$

where f is the fraction of objects that the agent perceives in its neighborhood; k_1 is a constant. When there are not many objects in the agent's neighborhood, that is $f << k_1$, then p_p approaches 1; hence, objects have a high probability of being picked up. On the other hand, if the agent observes many objects ($f >> k_1$), p_p approaches 0, and the probability that the agent will pick an object is small.

Each loaded agent has a probability p_d of dropping the carried object, given by

$$p_d = (\frac{f}{k_2 + f})^2$$

where k_2 is a constant. If the agent observes a large number of objects in its neighborhood ($f >> k_2$), then p_d approaches 1, and the probability of dropping the object is high. If $f << k_2$, then p_d approaches 0.

The fraction of objects, f, is calculated by making use of a short-term memory for each agent. Each agent keeps track of the last T time units, and f is simply the number of objects observed during these T time units, divided by the largest number of objects that can be observed during the T time units.

The above approach was initially developed for robotic implementation. The problem now is how to use this principle to develop an algorithmic approach to data clustering. The first part of the solution is to define a dissimilarity $d(\vec{z}_i, \vec{z}_j)$ between data vectors $\vec{z}_i$ and $\vec{z}_j$ using any norm, for example the euclidean norm. The next step is to determine how these dissimilarity measures should be used to group together similar data vectors, such that the clustering has the following properties:

- Intra-cluster distances should be small; that is, the distances between data vectors within a cluster should be small to form a compact, condensed cluster.
- Inter-cluster distances should be large; that is, the different clusters should be well separated.

The ant colony clustering follows an approach similar to that of SOMs, where the larger space of vector attributes is mapped onto a smaller two-dimensional grid space. The clustering performed on the two-dimensional grid should preserve the neighborhood relationships present in the higher-dimensional attribute space. The clustering should not introduce neighbors that do not exist in the attribute space.

Agents move randomly around on the grid, while observing the surrounding area of s^2 sites. The surrounding area is simply a square neighborhood, $\mathcal{N}_{s\times s}(r)$, of the $s \times s$ sites surrounding the current position r of the agent. Assume that, at time step t, an agent on site r of the grid finds data vector $\vec{z}_i$. The "local" density $f(\vec{z}_i)$ of that data vector within the agent's neighborhood is then given as

$$f(\vec{z}_i) = \begin{cases} \frac{1}{s^2} \sum_{\vec{z}_j \in \mathcal{N}_{s\times s}} [1 - \frac{d(\vec{z}_i, \vec{z}_j)}{\alpha}] & \text{if } f > 0 \\ 0 & \text{otherwise} \end{cases}$$

In the above, $f(\vec{z}_i)$ is a measure of average similarity of $\vec{z}_i$ with the other data vectors in the neighborhood. The constant α controls the scale of dissimilarity, by determining when two data vectors should be grouped together. For the above equation, $d(\vec{z}_i, \vec{z}_j) \in [0, 1]$.

The probabilities of picking up and dropping a data vector are expressed as

$$\begin{aligned} p_p(\vec{z}_i) &= (\frac{k_1}{k_1 + f(\vec{z}_i)})^2 \\ p_d(\vec{z}_i) &= \begin{cases} 2f(\vec{z}_i) & \text{if } f(\vec{z}_i) < k_2 \\ 1 & \text{if } f(\vec{z}_i) \geq sk_2 \end{cases} \end{aligned}$$

A compact summary of the ant colony clustering algorithm is given below [Lumer and Faieta 1994]:

1. Initialization:
   (a) place each data vector $\vec{z}_i$ randomly on the grid
   (b) place agents at randomly selected sites
   (c) set values for k_1, k_2, α, s and the maximum number of time steps t_{max}.‘
2. For $t = 1$ to t_{max}, for each agent:
   (a) If the agent is unladen, and the site is occupied by an item $\vec{z}_i$,
      i. Compute $f(\vec{z}_i)$ and $p_p(\vec{z}_i)$.
      ii. If $U(0, 1) \leq p_p(\vec{z}_i)$, pick up data vector $\vec{z}_i$.
   (b) Otherwise, if the agent carries data vector $\vec{z}_i$ and the site is empty:
      i. Compute $f(\vec{z}_i)$ and $p_d(\vec{z}_i)$.
      ii. If $U(0, 1) \leq p_d(\vec{z}_i)$, drop data vector $\vec{z}_i$.

(c) Move to a randomly selected neighboring site not occupied by another agent.

Some remarks about the algorithm above are necessary:

- the grid should have more sites than the number of ants; and
- there should be more sites than data vectors.

The algorithm also has the tendency to create more clusters than are necessary, basically overfitting the data. This problem can be addressed in the following ways:

- Having the ants move at different speeds. Fast-moving agents will form coarser clusters by being less selective in their estimation of the average similarity of a data vector to its neighbors. Slower agents are more accurate in refining the cluster boundaries. Having agents that move at different speeds prevents the clustering process from converging too fast.

  Different moving speeds are easily modeled using

$$f(\vec{z}_i) = \begin{cases} \frac{1}{s^2} \sum_{\vec{z}_j \in \mathcal{N}_{s \times s}} [1 - \frac{d(\vec{z}_i, \vec{z}_j)}{\alpha(1 - \frac{v-1}{v_{max}})}] & \text{if } f > 0 \\ 0 & \text{otherwise} \end{cases}$$

  where $v \sim U(1, v_{max})$, and v_{max} is the maximum moving speed.

- Using short-term memory for each agent, which allows each agent to remember the last m data vectors dropped by the agent, and the locations of these drops. If the agent picks up another data vector similar to the last m data vectors, the agent will move in that direction. This approach ensures that similar data vectors are grouped into the same cluster.

17.5 Applications of Ant Colony Optimization

The study of the behavior of ants has resulted in developing algorithms applied to a variety of problems. Algorithms that model the foraging behavior of ants (e.g. food collection) resulted in new combinatorial optimization approaches (with applications to network routing, job scheduling, etc); the ability of ants to dynamically distribute labor resulted in adaptive task allocation strategies; cemetery organization and brood sorting resulted in graph coloring and sorting algorithms; and the cooperative transport characteristics resulted in robotic implementations. This section reviews some of these applications with reference to the relevant literature.

- ACO has been used to solve the quadratic assignment problem (QAP) [Maniezzo *et al.* 1994]. The QAP concerns the optimal assignment of n activities to n locations. Formally, the QAP is defined as a permutation π of assignments which minimizes

$$c(\pi) = \sum_{i,j=1}^{n} d_{ij} f_{\pi(i)\pi(j)}$$

  where d_{ij} is the distance between locations i and j, and f_{hk} characterizes the flow (e.g. data transfer) between activities h and k.

- ACO has also been used successfully for the job-scheduling problem (JSP) [Colorni *et al.* 1994]. For the JSP, operations of a set of jobs must be scheduled to be executed on M machines in such a way that the maximum completion times of all operations is minimized, and only one job at a time is processed by a machine.

- The graph coloring problem (GCP) is a well-known optimization problem also solved by ACO [Costa and Hertz 1997]. This problem involves coloring the nodes of a graph, using q colors, such that no adjacent nodes have the same color.

- Another NP-hard problem solved by ACO is the shortest common supersequence problem (SCSP) [Michel and Middendorf 1998]. The aim of the SCSP is to find a string of minimum length that is a supersequence of each string in a set of strings.

- ACO has also been used for routing optimization in telephone networks [Schoonderwoerd *et al.* 1996] and data communications networks [Di Caro and Dorigo 1998].

- Robotics is a very popular field for the application of models of ants behavior, especially swarm-based robotics. Some of these applications include (see [Dorigo 1999] for lists of references)

  - adaptive task allocation to groups of agents or robots;
  - robots for distributed clustering of objects and sorting objects;
  - self-assembling (or metamorphic) robots; and
  - cooperative transport by a swarm of robots.

These are just a few applications. For more information on these, and other applications of ACO, the reader is referred to [Dorigo 1999].

17.6 Conclusion

The study of ant colonies is a very young field in CI and Artificial Life, with much more interesting applications still to be explored. While the standard ACO works only for discrete optimization problems, ACO can be applied to continuous problems by discretizing the search space into discrete regions [Bilchev and Parmee 1995], thereby enlarging the application area of ACO.

17.7 Assignments

1. Consider the following situation: ant A_1 follows the shortest of two paths to the food source, while ant A_2 follows the longer path. After A_2 reached the food source, which path back to the nest has a higher probability of being selected by A_2? Justify your answer.

2. Discuss the importance of the forgetting factor in the pheromone trail depositing equation (17.3).

3. Discuss the effects of the α and β parameters in the transition rule of equation (17.2).

4. Show how the ACO approach to solving the TSP satisfies all the constraints of the TSP.

5. Comment on the following strategy: Let the amount of pheromone deposited be a function of the best route. That is, the ant with the best route, deposits more pheromone. How can equation (17.3) be updated to reflect this?

6. Comment on the similarities and differences between the ant colony approach to clustering and SOMs.

7. For the ant clustering algorithm, explain why

   (a) the 2D-grid should have more sites than number of ants;

   (b) there should be more sites than data vectors.

8. Devise a dynamic forgetting factor for pheremone evaporation.

Part V

FUZZY SYSTEMS

Two-valued, or Boolean logic is a well-defined and used theory. Boolean logic is especially important for implementation in computing systems, where information, or knowledge about a problem, is binary encoded. Boolean logic also played an important role in the development of the first AI reasoning systems, especially the inference engine of expert systems [Giarratano 1998]. For such knowledge representation and reasoning systems, propositional and first-order predicate calculus are extensively used as representation language [Luger and Stubblefield 1997, Nilsson 1998]. Associated with Boolean logic is the traditional two-valued set theory, where an element either belongs to a class or not. That is, class membership is precise. Coupled with Boolean knowledge, two-valued set theory enabled the development of exact reasoning systems.

While some successes have been achieved using two-valued logic and sets, it is not possible to solve all problems by mapping the domain into two-valued variables. Most real-world problems are characterized by the ability of a representation language (or logic) to process incomplete, imprecise, vague or uncertain information. While two-valued logic and set theory fail in such environments, fuzzy logic and fuzzy sets give the formal tools to reason about such uncertain information. With fuzzy logic, domains are characterized by linguistic terms, rather than by numbers. For example, in the phrases *"it is partly cloudy"*, or *"Stephan is very tall"*, both *partly* and *very* are linguistic terms describing the *magnitude* of the fuzzy (or linguistic) variables *cloudy* and *tall*. The human brain has the ability to understand these terms, and infer from them that it will most probably not rain, and that Stephan might just be a good basket ball player (note, again, the fuzzy terms!). However, how do we use two-valued logic to represent these phrases?

Together with fuzzy logic, fuzzy set theory provides the tools to develop software products that model human reasoning (also referred to as approximate reasoning). In fuzzy sets, an element belongs to a set to a degree, indicating the certainty (or uncertainty) of membership.

The development of logic has a long and rich history, in which major philosophers played a role. The foundations of two-valued logic stemmed from the efforts of Aristotle (and other philosophers of that time), resulting in the so-called *Laws of*

Thought [Karner 1967]. The first version of these laws was proposed around 400 bc, namely the *Law of the Excluded Middle*. This law states that every proposition must have only one of two outcomes: either *true* or *false*. Even in that time, immediate objections were given with examples of propositions that could be true, and simultaneously not true.

It was another great philosopher, Plato, who laid the foundations of what is today referred to as fuzzy logic. It was, however, only in the 1900s that Lukasiewicz proposed the first alternative to the Aristotelian two-valued logic [Lejewski 1967]. Three-valued logic has a third value which is assigned a numeric value between *true* and *false*. Lukasiewicz later extended this to four-valued and five-valued logic. It was only recently, in 1965, that Lotfi Zadeh produced the foundations of infinite-valued logic with his mathematics of fuzzy set theory [Zadeh 1965].

Following the work of Zadeh, much research has been done in the theory of fuzzy systems, with applications in control, information systems, pattern recognition and decision support. Some successful real-world applications include automatic control of dam gates for hydroelectric-powerplants, camera aiming, compensation against vibrations in camcorders, cruise-control for automobiles, controlling air-conditioning systems, archiving systems for documents, optimized planning of bus time-tables, and many more. While fuzzy sets and logic have been used to solve real-world problems, they were also combined with other CI paradigms to form hybrid systems, for example, fuzzy neural networks and fuzzy genetic algorithms [Zhang and Kandel 1998].

A different set theoretic approach which also uses the concept of membership functions, namely rough sets (introduced by Pawlak in 1982 [Pawlak 1982]), is sometimes confused with fuzzy sets. While both fuzzy sets and rough sets make use of membership functions, rough sets differ in the sense that a lower and upper approximation to the rough set is determined. The lower approximation consists of all elements which belong with full certainty to the corresponding set, while the upper approximation consists of elements that may possibly belong to the set. Rough sets are frequently used in machine learning as classifier, where rough sets are used to find the smallest number of features to discern between classes [Mollestad 1997]. Rough sets are also used for extracting knowledge from incomplete data [Mollestad 1997, Polkowski and Skowron 1998]. Hybrid approaches that employ both fuzzy and rough sets have also been developed [Thiele 1998].

The remainder of this Part is organized as follows: Chapter 18 introduces fuzzy logic and fuzzy set theory, while Chapter 19 discusses how fuzzy logic can be used in approximate reasoning systems. Chapter 20 presents a short overview of fuzzy controllers, one of the largest application areas of fuzzy sets and logic. The Part is concluded with an introduction to rough set theory in Chapter 21.

Chapter 18

Fuzzy Systems

Consider the problem of designing a set of all tall people, and assigning all the people you know to this set. Consider classical set theory where an element is either a member of the set or not. Suppose all tall people are described as those with height greater than 1.75m. Then, clearly a person of height 1.78m will be an element of the set *tall*, and someone with height 1.5m will not belong to the set of tall people. But, the same will apply to someone of height 1.73m, which implies that someone who falls only 2cm short is not considered as being tall. Also, using two-valued set theory, there is no distinction among members of the set of tall people. For example, someone of height 1.78m and one of height 2.1m belongs equally to the set! Thus, no semantics are included in the description of membership.

The alternative, fuzzy set theory, has no problem with this situation. In this case all the people you know will be members of the set *tall*, but to different degrees. For example, a person of height 2.1m may be a member of the set to degree 0.95, while someone of length 1.7m may belong to the set with degree 0.4.

Fuzzy logic is an extension of Boolean logic to handle the concept of *partial truth*, which enables the modeling of the uncertainties of natural language. The vagueness in natural language is further emphasized by linguistic terms used to describe objects or situations. For example, the phrase *when it is very cloudy, it will most probably rain*, has the linguistic terms *very* and *most probably* – which are understood by the human brain. Fuzzy logic and fuzzy sets give the tools to also write software which enables computing systems to understand such vague terms, and to reason with these terms.

This chapter formally introduces fuzzy sets and fuzzy logic. Section 18.1 defines fuzzy sets, while membership functions are discussed in Section 18.2. Operators that can be applied to fuzzy sets are covered in Section 18.3. Characteristics of fuzzy sets are summarized in Section 18.4. The concepts of linguistic variables and hedges are discussed in Section 18.5. The chapter is concluded with a discussion of

the differences between fuzziness and probability in Section 18.6.

18.1 Fuzzy Sets

Different to classical sets, elements of a fuzzy set have membership degrees to that set. The degree of membership to a fuzzy set indicates the certainty (or uncertainty) we have that the element belongs to that set. Formally defined, suppose X is the domain, or universe of discourse, and $x \in X$ is a specific element of the domain X. Then, the fuzzy set A is characterized by a membership mapping function

$$\mu_A : X \rightarrow [0, 1] \tag{18.1}$$

Therefore, for all $x \in X$, $\mu_A(x)$ indicates the certainty to which element x belongs to fuzzy set A. For two-valued sets, $\mu_A(x)$ is either 0 or 1.

Fuzzy sets can be defined for discrete (finite) or continuous (infinite) domains. The notation used to denote fuzzy sets differ based on the type of domain over which that set is defined. In the case of a discrete domain X, the fuzzy set can either be expressed in the form of an n-dimensional vector or using the sum notation. If $X = \{x_1, x_2, \cdots, x_n\}$, then, using vector notation,

$$A = \{(\mu_A(x_i)/x_i) | x_i \in X, i = 1, \cdots, n\}$$

Using sum notation,

$$A = \mu_A(x_1)/x_1 + \mu_A(x_2)/x_2 + \cdots + \mu_A(x_n)/x_n = \sum_{i=1}^{n} \mu_A(x_i)/x_i$$

where the sum should not be confused with algebraic summation. The use of sum notation above simply serves as an indication that A is a set of ordered pairs. A continuous fuzzy set, A, is denoted as

$$A = \int_X \mu(x)/x$$

Again, the integral notation should not be algebraically interpreted.

18.2 Membership Functions

The membership function is the essence of fuzzy sets. A membership function, also referred to as the characteristic function of the fuzzy set, defines the fuzzy set. The function is used to associate a degree of membership of each of the elements of the domain to the corresponding fuzzy set. Two-valued sets are also characterized by

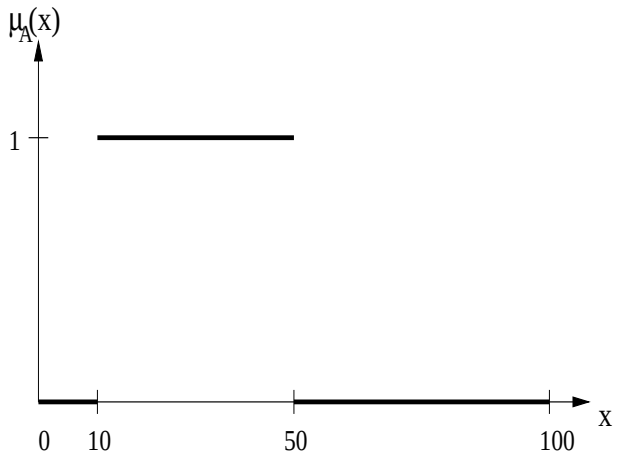


Figure 18.1: Illustration of membership function for two-valued sets

a membership function. For example, consider the domain X of all floating point numbers in the range $[0, 100]$. Define the set $A \subset X$ of all floating point numbers in the range $[10, 50]$. Then, the membership function for the set A is represented in Figure 18.1. All $x \in [10, 50]$ have $\mu_A(x) = 1$, while all other floating point numbers have $\mu_A(x) = 0$.

Membership functions for fuzzy sets can be of any shape or type as determined by experts in the domain over which the sets are defined. While designers of fuzzy sets have much freedom in selecting appropriate membership functions, these functions must satisfy the following constraints:

- A membership function must be bounded from below by 0 and from above by 1.
- The range of a membership function must therefore be the range $[0, 1]$.
- For each $x \in X$, $\mu_A(x)$ must be unique. That is, the same element cannot map to different degrees of membership for the same fuzzy set.

Returning to the *tall* fuzzy set, a possible membership function can be defined as (also illustrated in Figure 18.2)

$$tall(x) = \begin{cases} 0 & \text{if } height(x) < 1.5m \\ (height(x) - 1.5m)/0.3m & \text{if } 1.5m \leq height(x) \leq 2.0m \\ 1 & \text{if } height(x) > 2.0m \end{cases}$$

While the *tall* membership function above used a discrete step function, more complex discrete and continuous functions can be used, for example, triangular functions

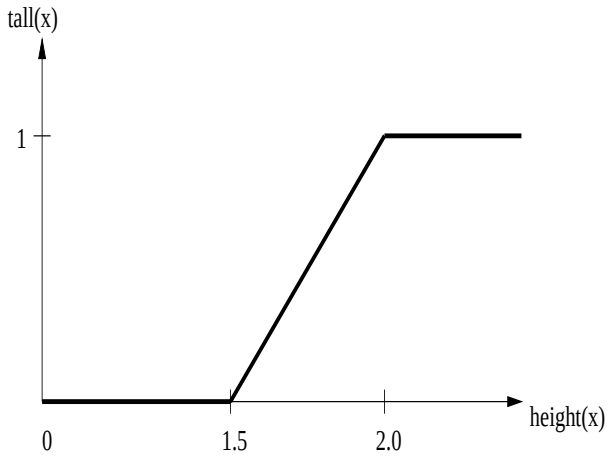


Figure 18.2: Illustration of *tall* membership function

(refer to Figure 18.3(a)), trapezoidal functions (refer to Figure 18.3(b)), logistic functions (refer to Figure 18.3(c)) and Gaussian functions (refer to Figure 18.3(d)). It is the task of the human expert of the domain to define the function which captures the characteristics of the fuzzy set.

18.3 Fuzzy Operators

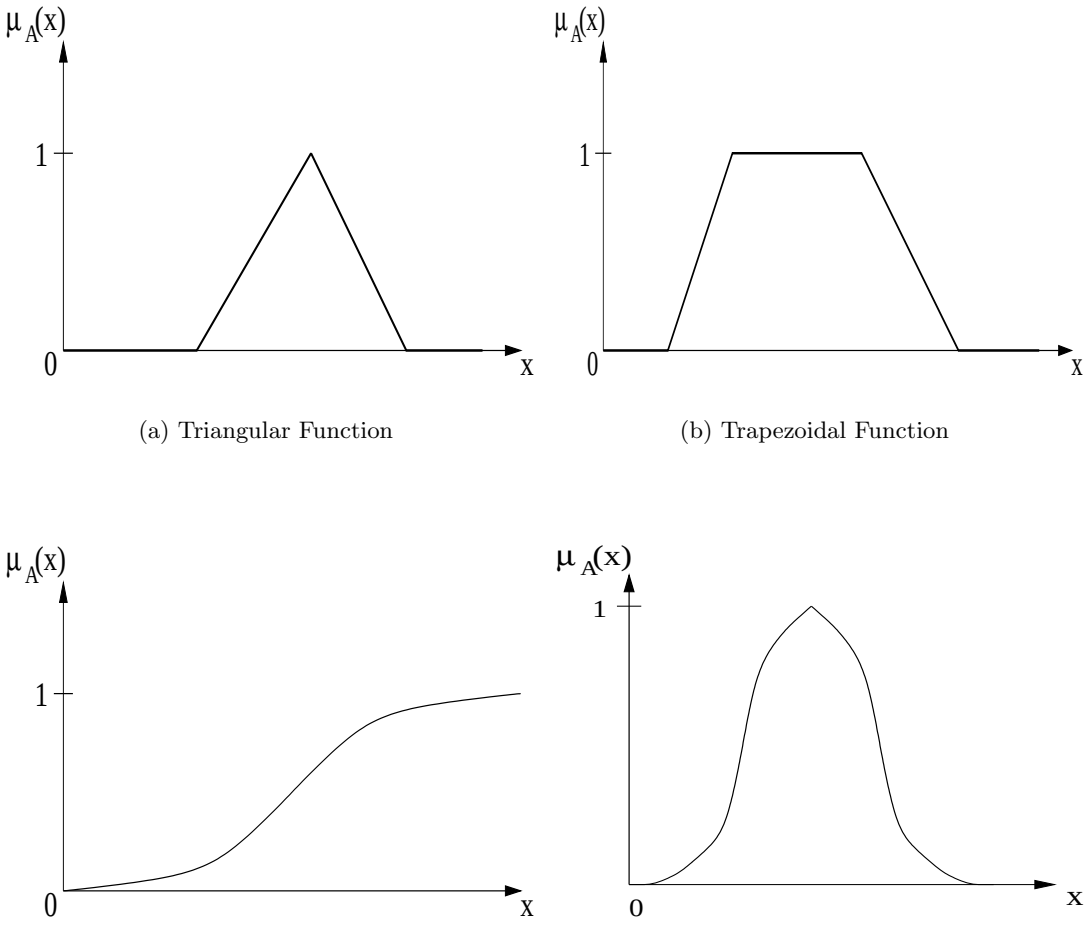
As for Boolean logic, relations and operators are defined for fuzzy logic which enables reasoning about vague information. Each of these relations and operators are defined below. For this purpose let X be the domain, or universe, and A and B are sets defined over the domain X.

Equality of fuzzy sets: For two-valued sets, sets are equal if the two sets have the same elements. For fuzzy sets, however, equality cannot be concluded if the two sets have the same elements. The degree of membership of elements to the sets must also be equal. That is, the membership functions of the two sets must be the same.

Therefor, two fuzzy sets A and B are equal if and only if the sets have the same domain, and $\mu_A(x) = \mu_B(x)$ for all $x \in X$. That is, $A = B$.

Containment of fuzzy sets: For two-valued sets, $A \subset B$ if all the elements of A are also elements of B. For fuzzy logic, this definition is not complete, and the degrees of membership of elements to the sets have to be considered.

Fuzzy set A is a subset of fuzzy set B if and only if $\mu_A(x) \leq \mu_B(x)$ for all $x \in X$. That is, $A \subset B$.



(a) Triangular Function

(b) Trapezoidal Function

(c) Logistic Function

(d) Gaussian Function

Figure 18.3: Example membership functions for fuzzy sets

Complement of a fuzzy set (NOT): The complement of a two-valued set is simply the set containing the entire domain without the elements of that set. For fuzzy sets, the complement of the set A consists of all the elements of set A, but the membership degrees differ. Let $\overline{A}$ denote the complement of set A. Then, for all $x \in X$, $\mu_{\overline{A}}(x) = 1 - \mu_A(x)$.

Intersection of fuzzy sets (AND): The intersection of two-valued sets is the set of elements occurring in both sets. For fuzzy sets, the intersection is the set of all elements in the fuzzy set, but with degrees of membership to the new set determined by one of two operators. Let A and B be two fuzzy sets, then either

- $\mu_{A\cap B}(x) = \min\{\mu_A(x), \mu_B(x)\},\ \forall x \in X$, or
- $\mu_{A\cap B}(x) = \mu_A(x) * \mu_B(x),\ \forall x \in X$

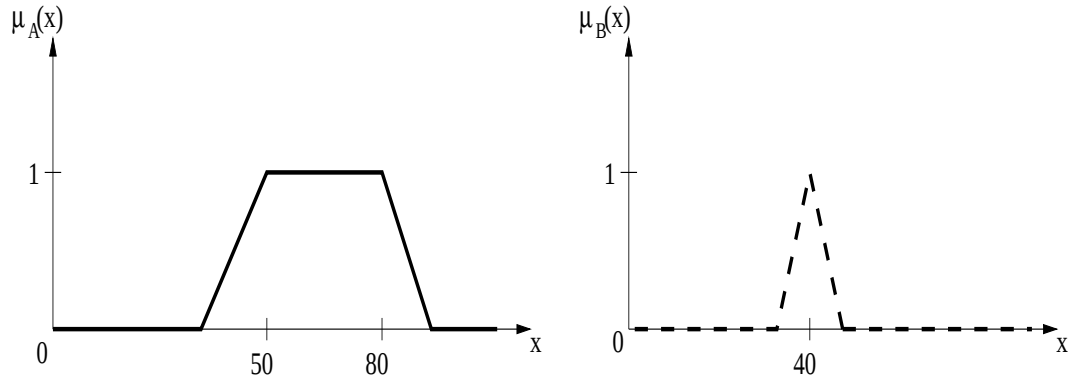
The difference between the two operations should be noted. Taking the product of membership degrees is a much stronger operator than taking the minimum, resulting in lower membership degrees for the intersection. It should also be noted that the ultimate result of a series of intersections approaches 0.0, even if the degrees of memberships to the original sets are high.

Union of fuzzy sets (OR): The union of two-valued sets contains the elements of all of the sets. The same is true for fuzzy set, but with membership degrees determined by one of the following operators:

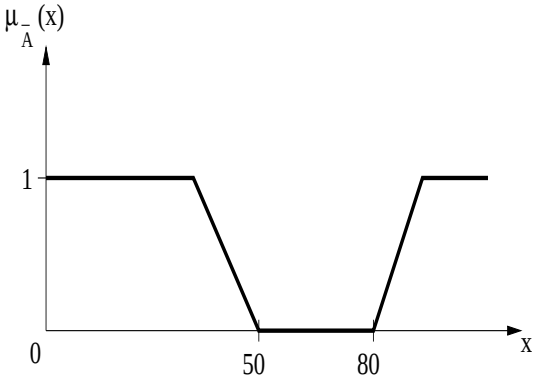
- $\mu_{A\cup B}(x) = \max\{\mu_A(x), \mu_B(x)\},\ \forall x \in X$, or
- $\mu_{A\cup B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x) * \mu_B(x),\ \forall x \in X$

Again, careful consideration must be given to the differences between the two approaches above. In the limit, a series of unions will have a result that approximates 1.0, even though membership degrees are low for the original sets!

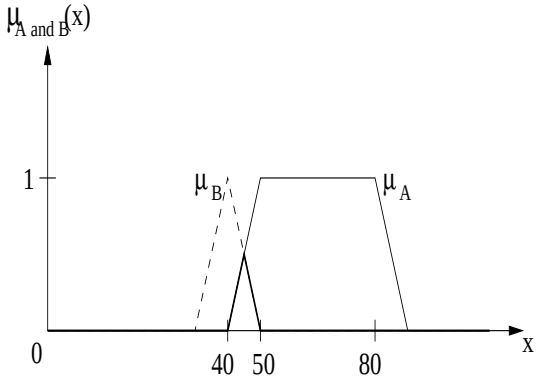
Operations on two-valued sets are easily visualized using Venn-diagrams. For fuzzy sets the effects of operations can be illustrated by graphing the resulting membership function, as illustrated in Figure 18.4. For the illustration in Figure 18.4, assume the fuzzy sets A defined as floating point numbers between $[50, 80]$ and B defined as numbers *about* 40 (refer to Figure 18.4(a) for definitions of the membership functions). The complement of set A is illustrated in Figure 18.4(b), the intersection of the two sets are given in Figure 18.4(c) (assuming the *min* operator), and the union in Figure 18.4(d) (assuming the *max* operator).



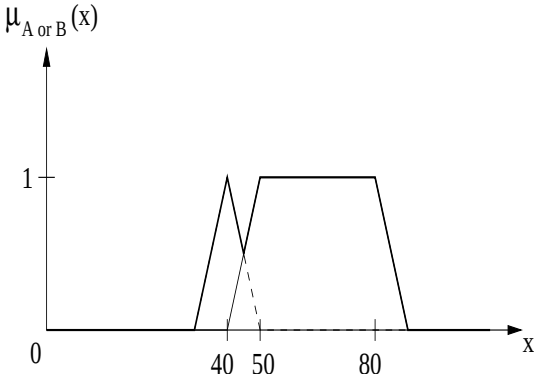
(a) Membership Functions for Sets A and B



(b) Complement of A



(c) Intersection of A and B



(d) Union of A

Figure 18.4: Illustration of fuzzy operators

18.4 Fuzzy Set Characteristics

As discussed previously, fuzzy sets are described by membership functions. In this section, characteristics of membership functions are overviewed. These characteristics include normality, height, support, core, cut, unimodality and cardinality.

Normality: A fuzzy set A is normal if that set has an element that belongs to set A with degree 1. That is,

$$\exists x \in A \bullet \mu_A(x) = 1$$

then A is normal, otherwise, A is subnormal. Normality can alternatively be defined as

$$\sup_x A(x) = 1$$

Height: The height of a fuzzy set is defined as the supremum of the membership function, i.e.

$$height(A) = \sup_x A(x)$$

Support: The support of fuzzy set A is the set of all elements in the universe of discourse, X, that belongs to A with non-zero membership. That is,

$$support(A) = \{x \in X | \mu_A(x) > 0\}$$

Core: The core of fuzzy set A is the set of all elements in the domain that belongs to A with membership degree 1. That is,

$$core(A) = \{x \in X | \mu_A(x) = 1\}$$

α-cut: The set of elements of A with membership degree greater than α is referred to as the α-cut of A:

$$A_\alpha = \{x \in X | \mu_A(x) \geq \alpha\}$$

Unimodality: A fuzzy set is unimodal if its membership function is a unimodal function, i.e. the function has just one maximum.

Cardinality: The cardinality of two-valued sets is simply the number of elements within the sets. This is not the same for fuzzy sets. The cardinality of fuzzy set A, for a finite domain, X, is defined as

$$card(A) = \sum_{x \in X} \mu_A(x)$$

and for an infinite domain,

$$card(A) = \int_{x \in X} \mu_A(x) dx$$

For example, if $X = \{a, b, c, d\}$, and $A = 0.3/a + 0.9/b + 0.1/c + 0.7/d$, then $card(A) = 0.3 + 0.9 + 0.1 + 0.7 = 2.0$.

Normalization: A fuzzy set is normalized by dividing the membership function by the height of the fuzzy set. That is,

$$normalized(A) = \frac{\mu_A(x)}{height(x)}$$

The properties of fuzzy sets are very similar to that of two-valued sets, however, there are some differences. Fuzzy sets follow, similar to two-valued sets, the commutative, associative, distributive, transitive and idempotency properties. One of the major differences is in the properties of the cardinality of fuzzy sets, as listed below:

- $card(A) + card(B) = card(A \cap B) + card(A \cup B)$
- $card(A) + card(\overline{A}) = card(X)$

where A and B are fuzzy sets, and X is the universe of discourse.

18.5 Linguistics Variables and Hedges

Lotfi Zadeh introduced the concept of linguistic variable (or fuzzy variable) in 1973, which allows computation with words in stead of numbers [Zadeh 1975]. Linguistic variables are variables with values that are words or sentences from natural language. For example, referring again to the set of tall people, *tall* is a linguistic variable. Sensory inputs are linguistic variables, or nouns in a natural language, for example, temperature, pressure, displacement, etc. Linguistic variables (and hedges, explained below) allow the translation of natural language into logical, or numerical statements, which provide the tools for approximate reasoning (refer to chapter 19).

Linguistic variables can be divided into different categories:

- Quantification variables, e.g. all, most, many, none, etc.
- Usuality variables, e.g. sometimes, frequently, always, seldom, etc.
- Likelihood variables, e.g. possible, likely, certain, etc.

In natural language, nouns are frequently combined with adjectives for quantifications of these nouns. For example, in the phrase *very tall*, the noun *tall* is quantified by the adjective *very*, indicating a person who is "taller" than tall. In fuzzy systems theory, these adjectives are referred to as hedges. A hedge serves as a modifier of fuzzy values. In other words, the hedge *very* changes the membership of elements of the set *tall* to different membership values in the set *very_tall*. Hedges are implemented through subjective definitions of mathematical functions, to transform membership values in a systematic manner.

To illustrate the implementation of hedges, consider again the set of tall people, and assume the membership function μ_{tall} characterizes the degree of membership of elements to the set *tall*. Our task is to create a new set, *very_tall* of people that are very tall. In this case, the hedge *very* can be implemented as the square function. That is, $\mu_{very\_tall}(x) = \mu_{tall}(x)^2$. Hence, if Peter belongs to the set *tall* with certainty 0.9, then he also belongs to the set *very_tall* with certainty 0.81. This makes sense according to our natural understanding of the phrase *very tall*: Degree of membership to the set *very_tall* should be less than membership to the set *tall*. Alternatively, consider the set *sort_of_tall* to represent all people that are sort of tall, i.e. people that are shorter than tall. In this case, the hedge *sort of* can be implemented as the square root function, $\mu_{sort\_of\_tall}(x) = \sqrt{\mu_{tall}(x)}$. So, if Peter belongs to the set *tall* with degree 0.81, he belongs to the set *sort_of_tall* with degree 0.9.

Different kinds of hedges can be defined, as listed below:

- **Concentration hedges** (e.g. very), where the membership values get relatively smaller. That is, the membership values get more concentrated around points with higher membership degrees. Concentration hedges can be defined, in general terms, as
$$\mu_{A'}(x) = \mu_A(x)^p, \quad for\ p > 1$$
where $A^{'}$ is the concentration of set A.

- **Dilation hedges** (e.g. somewhat, sort of, generally), where membership values increases. Dilation hedges are defined, in general, as
$$\mu_{A'}(x) = \mu_A(x)^{1/p} \quad for\ p > 1$$

- **Contrast intensification hedges** (e.g. extremely), where memberships lower than 1/2 are diminished, but memberships larger than 1/2 are elevated. This hedge is defined as,
$$\mu_{A'}(x) = \begin{cases} 2^{p-1}\mu_A(x)^p & \text{if } \mu_A(x) \leq 0.5 \\ 1 - 2^{p-1}(1-\mu_A(x))^p & \text{if } \mu_A(x) > 0.5 \end{cases}$$
which intensifies contrast.

- **Vague hedges** (e.g. seldom), are opposite to contrast intensification hedges, having membership values altered using
$$\mu_{A'}(x) = \begin{cases} \sqrt{\mu_A(x)/2} & \text{if } \mu_A(x) \leq 0.5 \\ 1 - \sqrt{(1-\mu_A(x))/2} & \text{if } \mu_A(x) > 0.5 \end{cases}$$
Vague hedges introduce more "fuzziness" into the set.

- **Probabilistic hedges**, which express probabilities, e.g. likely, not very likely, probably, etc.

18.6 Fuzziness and Probability

There is often confusion between the concepts of fuzziness and probability. It is important that the similarities and differences between these two terms are understood. Both terms refer to degrees of certainty of events occurring. But that is where the similarities stop. Degrees of certainty as given by statistical probability are only meaningful before the associated event occurs. After that event, the probability no longer applies, since the outcome of the event is known. For example, before flipping a fair coin, there is a 50% probability that heads will be on top, and a 50% probability that it will be tails. After the event of flipping the coin, there is no uncertainty as to whether heads or tails are on top, and for that event the degree of certainty no longer applies. In contrast, membership of fuzzy sets is still relevant after an event occurred. For example, consider the fuzzy set of tall people, with Peter belonging to that set with degree 0.9. Suppose the event to execute is to determine if Peter is good at basketball. Given some membership function, the outcome of the event is a degree of membership to the set of good basketball players. After the event occurred, Peter still belongs to the set of tall people with degree 0.9.

Furthermore, probability assumes independence among events, while fuzziness is not based on this assumption. Also, probability assumes a closed world model where everything is known, and where probability is based on frequency measures of occurring events. That is, probabilities are estimated based on a repetition of a finite number of experiments carried out in a stationary environment. The probability of an event A is thus estimated as

$$p(A) = \lim_{n \to \infty} \frac{n_A}{n}$$

where n_A is the number of experiments for which event A occurred, and n is the total number of experiments. Fuzziness does not assume everything to be known, and is based on descriptive measures of the domain (in terms of membership functions), instead of subjective frequency measures. Fuzzy logic can be used to derive new facts or knowledge from the fuzzy memberships of know facts.

Therefor, fuzziness is not probability, and probability is not fuzziness. Probability and fuzzy sets can, however, be used in a symbiotic way to express the probability of a fuzzy event.

18.7 Conclusion

This chapter gave an overview of fuzzy sets and fuzzy logic. The next chapter deals with fuzzy inferencing, and shows how these mathematical tools can be used in environments with uncertain information, to reason with such information, and to infer actions based on vague descriptions.

18.8 Assignments

1. Perform intersection and union for the fuzzy sets in Figure 18.4 using the product and addition operators.

2. Give the height, support, core and normalization of the fuzzy sets in Figure 18.4.

3. Consider the two fuzzy sets:

$$\begin{aligned} \text{long pencils} &= \{pencil1/0.1, pencil2/0.2, pencil3/0.4, pencil4/0.6, \\ &\quad pencil5/0.8, pencil6/1.0\} \\ \text{medium pencils} &= \{pencil1/1.0, pencil2/0.6, pencil3/0.4, pencil4/0.3, \\ &\quad pencil5/0.1\} \end{aligned}$$

   (a) Determine the union of the two sets.

   (b) Determine the intersection of the two sets.

   (c) Define a hedge for the set *very long pencils*, and give the resulting set.

4. What is the difference between the membership function of an ordinary set and a fuzzy set?

5. Consider the membership functions of two fuzzy sets, A and B, as given in Figure 18.5.

   (a) Draw the membership function for the fuzzy set $C = A \cap \overline{B}$, using the min-operator.

   (b) Compute $\mu_C(5)$.

   (c) Is C normal? Justify your answer.

6. Consider the fuzzy sets A and B such that $core(A) \cap core(B) = \emptyset$. Is fuzzy set $C = A \cap B$ normal? Justify your answer.

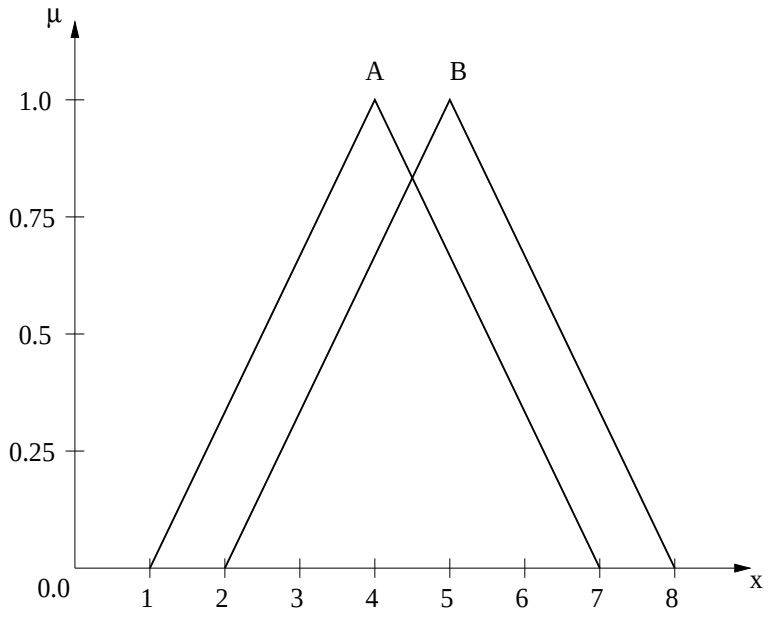


Figure 18.5: Membership functions for assignments

Chapter 19

Fuzzy Inferencing Systems

The previous chapter discussed theoretical aspects of fuzzy sets and fuzzy logic. The fuzzy set operators allow rudimentary reasoning about facts. For example, consider the three fuzzy sets *tall*, *good_athlete* and *good_basketball_player*. Now assume

$\mu_{tall}(Peter) = 0.9$ and $\mu_{good\_athlete}(Peter) = 0.8$

$\mu_{tall}(Carl) = 0.9$ and $\mu_{good\_athlete}(Carl) = 0.5$

If we know that a good basketball player is tall and is a good athlete, then which one of Peter or Carl will be the better basketball player? Through application of the intersection operator, we get

$$\mu_{good\_basketball\_player}(Peter) = \min\{0.9, 0.8\} = 0.8$$

$$\mu_{good\_basketball\_player}(Carl) = \min\{0.9, 0.5\} = 0.5$$

Using the standard set operators, it is possible to determine that Peter will be better at the sport than Carl.

The example above is a very simplistic situation. For most real-world problems, the sought outcome is a function of a number of complex events, or scenarios. For example, actions made by a controller are determined by a set of if-then rules. The if-then rules describe situations that can occur, with a corresponding action that the controller should execute. It is, however, possible that more than one situation, as described by if-then rules, are simultaneously active, with different actions. The problem is to determine the best action to take. A mechanism is therefore needed to infer an action from a set of activated situations. For fuzzy controllers, situations are expressed in terms of membership functions, and fuzzy inferencing upon the given information results in an appropriate action.

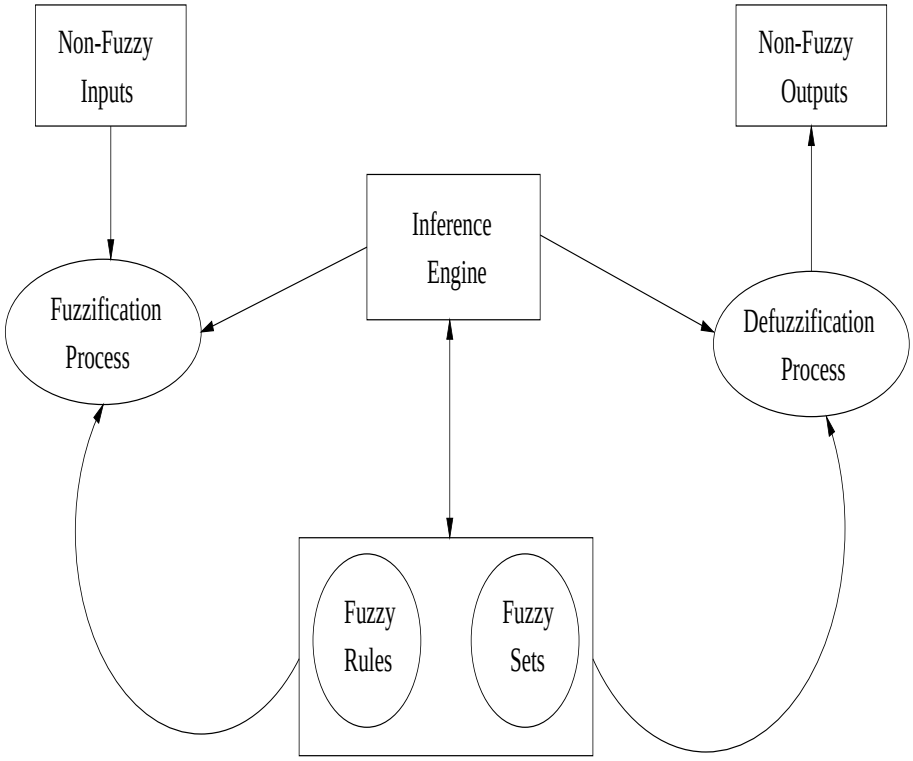


Figure 19.1: Fuzzy rule-based reasoning system

For fuzzy systems in general, the dynamic behavior of that system is characterized by a set of linguistic fuzzy rules. These rules are based on the knowledge and experience of a human expert within that domain. Fuzzy rules are of the general form

if antecedent(s) then consequent(s)

The antecedents of a rule form a combination of fuzzy sets through application of the logic operators (i.e. complement, intersection, union). The consequent part of a rule is usually a single fuzzy set, with a corresponding membership function. Multiple fuzzy sets can also occur withing the consequent, in which case they are combined using the logic operators.

Together, the fuzzy sets and fuzzy rules form the knowledge base of a fuzzy rule-based reasoning system. In addition to the knowledge base, a fuzzy reasoning system consists of three other components, each performing a specific task in the reasoning process, i.e. fuzzification, inferencing and defuzzification. The different components of a fuzzy rule based system are illustrated in Figure 19.1.

The remainder of this chapter is organized as follows: fuzzification is discussed in Section 19.1, fuzzy inferencing in Section 19.2, and defuzzification in Section 19.3.

19.1 Fuzzification

The antecedents of the fuzzy rules form the fuzzy "input space," while the consequents form the fuzzy "output space". The input space is defined by the combination of input fuzzy sets, while the output space is defined by the combination of output sets. The fuzzification process is concerned with finding a fuzzy representation of non-fuzzy input values. This is achieved through application of the membership functions associated with each fuzzy set in the rule input space. That is, input values from the universe of discourse are assigned membership values to fuzzy sets.

For illustration purposes, assume the fuzzy sets A and B, and assume the corresponding membership functions have been defined already. Let X denote the universe of discourse for both fuzzy sets. The fuzzification process receives the elements $a, b \in X$, and produces the membership degrees $\mu_A(a), \mu_A(b), \mu_B(a)$ and $\mu_B(b)$.

19.2 Inferencing

The task of the inferencing process is to map the fuzzified inputs (as received from the fuzzification process) to the rule base, and to produce a fuzzified output for each rule. That is, for the consequents in the rule output space, a degree of membership to the output sets are determined based on the degrees of membership in the input sets and the relationships between the input sets. The relationships between input sets are defined by the logic operators which combines the sets in the antecedent. The output fuzzy sets in the consequent are then combined to form one overall membership function for the output of the rule.

Assume input fuzzy sets A and B with universe of discourse X_1 and the output fuzzy set C with X_2 as universe of discourse. Consider the rule

$$\text{if } A \text{ is } a \text{ and } B \text{ is } b \text{ then } C \text{ is } c$$

From the fuzzification process, the inference engine knows $\mu_A(a)$ and $\mu_B(b)$. The first step of the inferencing process is then to calculate the firing strength of each rule in the rule base. This is achieved through combination of the antecedent sets using the operators discussed in Section 18.3. For the example above, assuming the *min*-operator, the firing strength is

$$\min\{\mu_A(a), \mu_B(b)\}$$

For each rule k, the firing strength α_k is thus computed.

The next step is to accumulate all activated outcomes. During this step, one single fuzzy value is determined for each $c_i \in C$. Usually, the final fuzzy value, β_i, associated with each outcome c_i is computed using the *max*-operator, i.e.

$$\beta_i = \max_{\forall k}\{\alpha_{k_i}\}$$

where α_{k_i} is the firing strength of rule k which has outcome c_i.

The end result of the inferencing process is a series of fuzzified output values. Rules that are not activated have a zero firing strength.

Rules can be weighted *a priori* with a factor (in the range [0,1]), representing the degree of confidence in that rule. These rule confidence degrees are determined by the human expert during the design process.

19.3 Defuzzification

The firing strengths of rules represent the degree of membership to the sets in the consequent of the corresponding rule. Given a set of activated rules and their corresponding firing strengths, the task of the defuzzification process is to convert the output of the fuzzy rules into a scalar, or non-fuzzy value.

For the sake of the argument, suppose the following hedges are defined for linguistic variable C (refer to Figure 19.2(a) for the definition of the membership functions): large decrease (LD), slight increase (SI), no change (NC), slight increase (SI), and large increase (LI). Assume three rules with the following C membership values: $\mu_{LI} = 0.8, \mu_{SI} = 0.6$ and $\mu_{NC} = 0.3$.

Several inference methods exist to find an approximate scalar value to represent the action to be taken:

- The **max-min method**: The rule with the largest firing strength is selected, and it is determined which consequent membership function is activated. The centroid of the area under that function is calculated and the horizontal coordinate of that centroid is taken as the output of the controller. For our example, the largest firing strength is 0.8, which corresponds to the *large_increase* membership function. Figure 19.2(b) illustrates the calculation of the output.

- The **averaging method**: For this approach, the average rule firing strength is calculated, and each membership function is clipped at the average. The centroid of the composite area is calculated and its horizontal coordinate is used as output of the controller. All rules therefore play a role in determining the action of the controller. Refer to Figure 19.2(c) for an illustration of the averaging method.

- The **root-sum-square method**: Each membership function is scaled such that the peak of the function is equal to the maximum firing strength that corresponds to that function. The centroid of the composite area under the scaled functions are computed and its horizontal coordinate is taken as output (refer to Figure 19.2(d)).

- The **clipped center of gravity method**: For this approach, each membership function is clipped at the corresponding rule firing strengths. The centroid of the composite area is calculated and the horizontal coordinate is used as the output of the controller. This approach to centroid calculation is illustrated in Figure 19.2(e).

The calculation of the centroid of the trapezoidal areas depends on whether the domain of the functions is discrete or continuous. For a discrete domain of a finite number of values, n, the output of the defuzzification process is calculated as ($\sum$ has its algebraic meaning)

$$output = \frac{\sum_{i=1}^{n} x_i \mu_C(x_i)}{\sum_{i=1}^{n} \mu_C(x_i)}$$

In the case of a continuous domain ($\prod$ has its algebraic meaning),

$$output = \frac{\int_{x \in X} x\mu(x)dx}{\int_{x \in X} \mu(x)dx}$$

where X is the universe of discourse.

19.4 Conclusion

This chapter presented an overview of fuzzy rule based reasoning systems. In the next chapter a specific example of such systems is discussed, namely fuzzy controllers.

19.5 Assignments

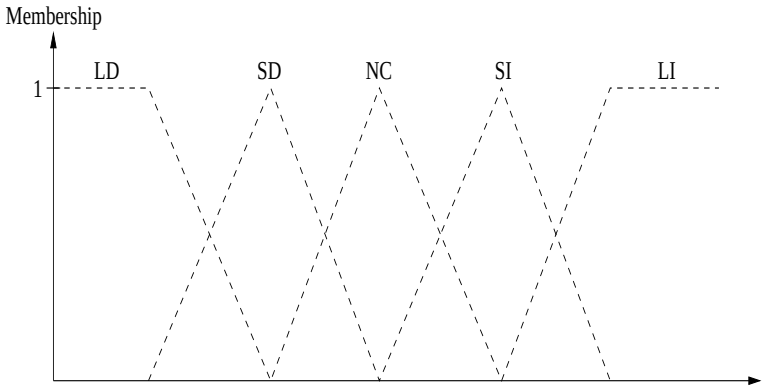
1. Consider the following rule base:

   if x is Small then y is Big
   if x is Medium then y is Small
   if x is Big then y is Medium

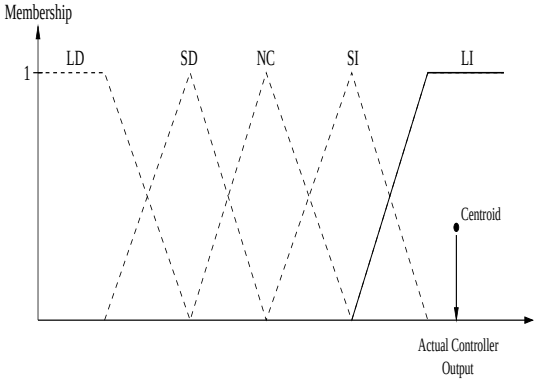
   Given the membership functions illustrated in Figure 19.3, answer the following questions:

   (a) Using the clipped center of gravity method, draw the composite function for which the centroid needs to be calculated, for $x = 2$.
   (b) Compute the defuzzified output on the discrete domain $Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

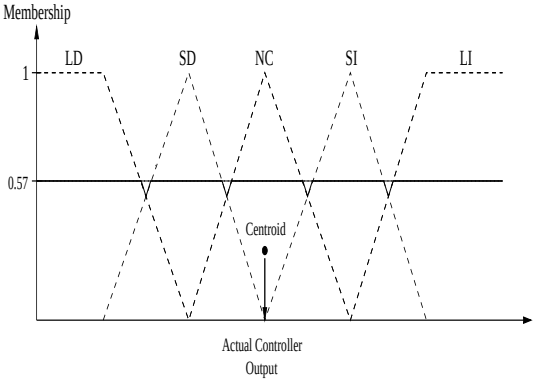
2. Repeat the assignment above for the root-sum-square method.



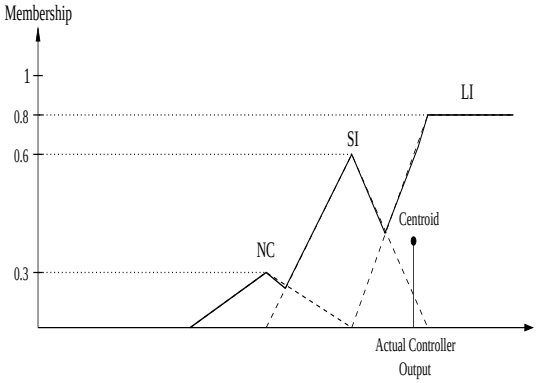
(a) Output Membership Functions



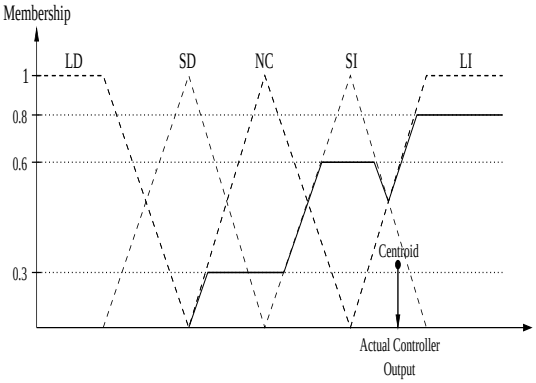
(b) Max-Min Method



(c) Averaging Method

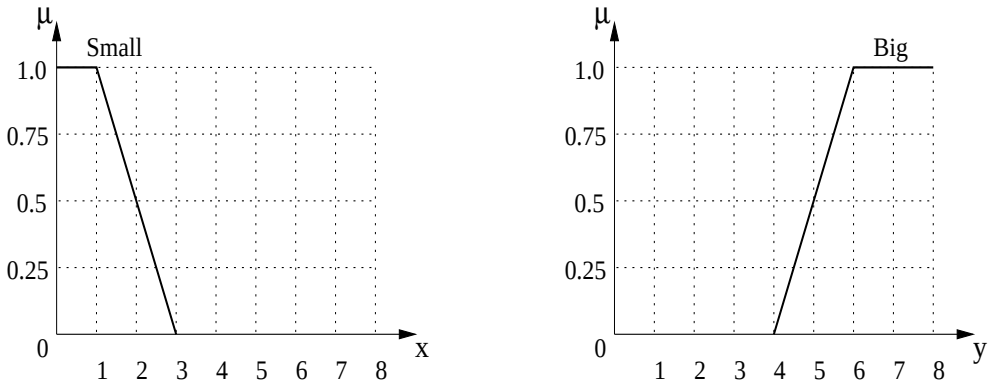


(d) Root-Sum-Square Method

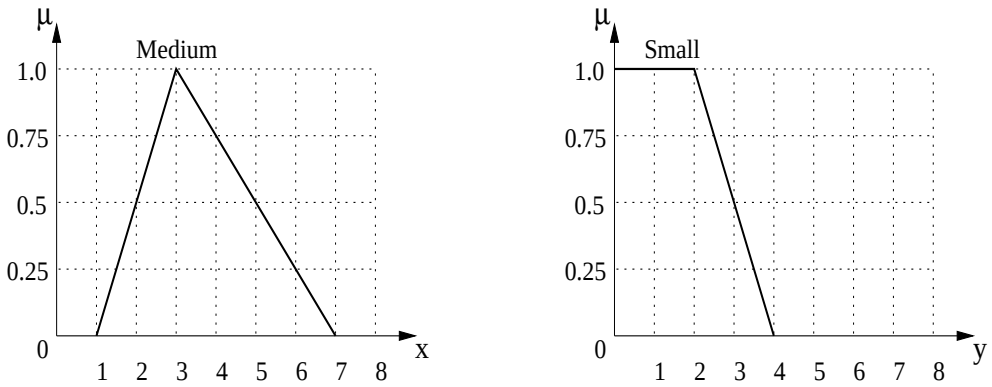


(e) Clipped Center of Gravity Method

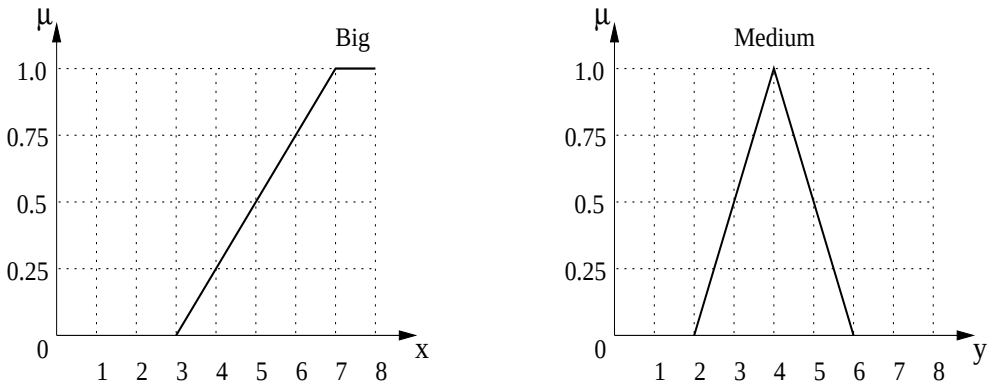
Figure 19.2: Defuzzification methods for centroid calculation



(a) if x is Small then y is Big



(b) if x is Medium then y is Small



(c) if x is Big then y is Medium

Figure 19.3: Membership functions for assignments 1 and 2

Chapter 20

Fuzzy Controllers

The design of fuzzy controllers is one of the largest application areas of fuzzy set theory. Where fuzzy logic is frequently described as *computing with words rather than numbers*, fuzzy control is described as *control with sentences rather than equations.* Thus, instead of describing the control strategy in terms of differential equations, control is expressed as a set of linguistic rules. These linguistic rules are easier understood than systems of mathematical equations.

The first application of fuzzy control comes from the work of Mamdani and Assilian in 1975, with their design of a fuzzy controller for a steam engine [Mamdani *et al.* 1975]. The objective of the controller was to maintain a constant speed by controlling the pressure on pistons, by adjusting the heat supplied to a boiler. Since then, a vast number of fuzzy controllers have been developed for consumer products and industrial processes. For example, fuzzy controllers have been developed for washing machines, video cameras, air conditioners, etc., while industrial applications include robot control, underground trains, hydro-electrical power plants, cement kilns, etc.

This chapter gives a short overview of fuzzy controllers. Section 20.1 discusses the components of such controllers, while Section 20.2 overviews some types of fuzzy controllers.

20.1 Components of Fuzzy Controllers

A fuzzy controller can be regarded as a nonlinear static function that maps controller inputs onto controller outputs. A controller is used to control some system, or plant. The system has a desired response that must be maintained under whatever inputs are received. The inputs to the system can, however, change the state of the system, which causes a change in response. The task of the controller is then to

take corrective action by providing a set of inputs that ensures the desired response. As illustrated in Figure 20.1, a fuzzy controller consists of four main components, which are integral to the operation of the controller:

- **Fuzzy rule base**: The rule base, or knowledge base, contains the fuzzy rules that represent the knowledge and experience of a human expert of the system. These rules express a nonlinear control strategy for the system.

  While rules are usually obtained from human experts, and are static, strategies have been developed that adapt, or refine rules through learning using neural networks or evolutionary computing [Favilla *et al.* 1993, Wang and Mendel 1992].

- **Condition interface (fuzzifier)**: The fuzzifier receives the actual outputs of the system, and transforms these non-fuzzy values into membership degrees to the corresponding fuzzy sets. In addition to the system outputs, the fuzzification of input values to the system also occurs via the condition interface.

- **Action interface (defuzzifier):** The action interface defuzzifies the outcome of the inference engine to produce a non-fuzzy value to represent the actual control function to be applied to the system.

- **Inference engine**: The inference engine performs inferencing upon fuzzified inputs to produce a fuzzy output (refer to Section 19.2).

As stated earlier, a fuzzy controller is basically a nonlinear control function. The nonlinearity in fuzzy controllers is caused by

- the fuzzification process, if nonlinear membership functions are used;

- the rule base, since rules express a nonlinear control strategy;

- the inference engine, if, for example, the *min*-operator is used for intersection and the *max*-operator is used for union; and

- the defuzzification process.

20.2 Fuzzy Controller Types

While there exists a number of different types of fuzzy controllers, they all have the same components and involve the same design steps. The differences between types of fuzzy controllers are mainly in the implementation of the inference engine and the defuzzifier.

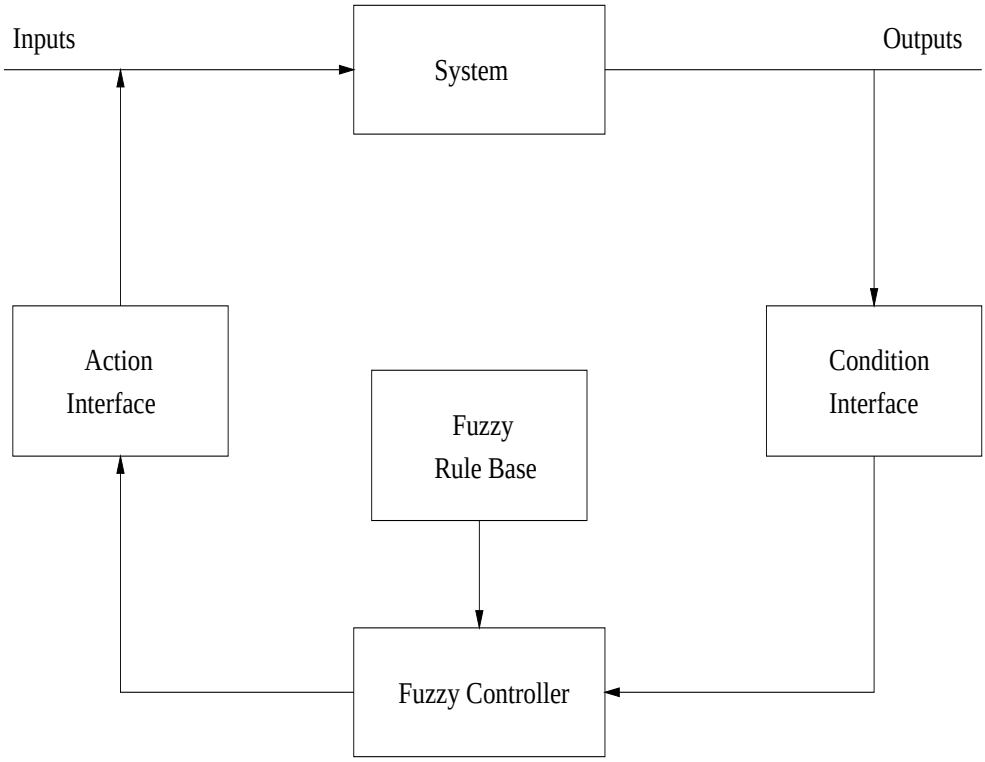


Figure 20.1: A fuzzy controller

The design of a fuzzy controller involves the following aspects: A universe of discourse needs to be defined, and the fuzzy sets and membership functions for both the input and output spaces have to be designed. With the help of a human expert, the linguistic rules that describe the dynamic behavior need to be defined. The designer has to decide on how the fuzzifier, inference engine and defuzzifier have to be implemented, after considering all the different options (refer to chapter 19). Other issues that need to be considered include the preprocessing of the raw measurements as obtained from measuring equipment. Preprocessing involves the removal of noise, discretization of continuous values, scaling and transforming values into a linguistic form.

In the next sections, three controller types are discussed, namely table-based, Mamdani and Takagi-Sugeno.

20.2.1 Table-Based Controller

Table-based controllers are used for discrete universes, where it is feasible to calculate all combinations of inputs. The relation between all input combinations and their corresponding outputs are then arranged in a table. In cases where there are only two inputs and one output, the controller operates on a two-dimensional look-up table. The two dimensions correspond to the inputs, while the entries in the table correspond to the outputs. Finding a corresponding output involves a simple and fast look-up in the table. Table-based controllers become inefficient for situations with a large number of input and output values.

20.2.2 Mamdani Fuzzy Controller

Mamdani and Assilian produced the first fuzzy controller [Mamdani *et al.* 1975]. Mamdani-type controllers follow the following simple steps:

1. Identify and name input linguistic variables and define their numerical ranges.
2. Identify and name output linguistic variables and define their numerical ranges.
3. Define a set of fuzzy membership functions for each of the input variables, as well as the output variables.
4. Construct the rule base that represents the control strategy.
5. Perform fuzzification of input values.
6. Perform inferencing to determine firing strengths of activated rules.
7. Defuzzify, using centroid of gravity, to determine the corresponding action to be executed.

20.2.3 Takagi-Sugeno Controller

For the table-based and Mamdani controllers, the output sets are singletons (i.e. a single set), or combinations of singletons where the combinations are achieved through application of the fuzzy set operators. Output sets can, however, also be linear combinations of the inputs. Takagi and Sugeno suggested an approach to allow for such complex output sets, referred to as Takagi-Sugeno fuzzy controllers [Jantzen 1998, Takagi and Sugeno 1985]. In general, the rule structure for Takagi-Sugeno fuzzy controllers is

$$\text{if } f_1(A_1 \text{ is } a_1, A_2 \text{ is } a_2, \cdots, A_n \text{ is } a_n) \text{ then } C = f_2(a_1, a_2, \cdots, a_n)$$

where f_1 is a logical function, and f_2 is some mathematical function of the inputs; C is the consequent, or output variable being inferred, a_i is an antecedent, or input variable, and A_i is a fuzzy set represented by the membership function μ_{A_i}. The complete rule base is defined by K rules.

The firing strength of each rule is computed using the *min*-operator, i.e.

$$\alpha_k = \min_{\forall i | a_i \in \mathcal{A}_k} \{\mu_{A_i}(a_i)\}$$

where $\mathcal{A}_k$ is the set of antecedents of rule k. Alternatively, the product can be used to calculate rule firing strengths:

$$\alpha_k = \prod_{\forall i | a_i \in \mathcal{A}_k} \mu_{A_i}(a_i)$$

The output of the controller is then determined as

$$C = \frac{\sum_{k=1}^{K} \alpha_k f_2(a_1, \cdots, a_n)}{\sum_{k=1}^{K} \alpha_k}$$

The main advantage of Takagi-Sugeno controllers is that it breaks the closed-loop approach of the Mamdani controllers. For the Mamdani controllers the system is statically described by rules. For the Takagi-Sugeno controllers, the fact that the consequent of rules is a mathematical function, provides for a more dynamic control.

20.3 Conclusion

This chapter gave a short summary of fuzzy controllers and three types of controllers, namely table-based, Mamdani and Takagi-Sugeno. While only a few aspects of fuzzy systems have been covered in this and the previous chapters, the theory of fuzzy systems is much larger than what was covered in this part of the book. The interested reader is encouraged to read more about the subject, especially on the vast number of applications.

20.4 Assignments

1. Design a Mamdani fuzzy controller to control a set of ten lifts for a building of forty storey to maximize utilization and minimize delays.

2. Design a Mamdani fuzzy controller for an automatic gearbox for motor vehicles.

3. Consider the following rule base:

   if x is Small then y is Big

   if x is Medium then y is Small

   if x is Big then y is Medium

   Given the membership functions illustrated in Figure 19.3, answer the following questions: using a Mamdani-type fuzzy controller, what are the firing strengths of each rule?

4. Consider the following Takagi-Sugeno rules:

   if x is A_1 and y is B_1 then $z_1 = x + y + 1$

   if x is A_2 and y is B_1 then $z_2 = 2x + y + 1$

   if x is A_1 and y is B_2 then $z_3 = 2x + 3y$

   if x is A_2 and y is B_2 then $z_4 = 2x + 5$

   Compute the value of z for $x = 1, y = 4$ and the antecedent fuzzy sets

$$A_1 = \{1/0.1, 2/0.6, 3/1.0\}$$

$$A_2 = \{1/0.9, 2/0.4, 3/0.0\}$$

$$B_1 = \{4/1.0, 5/1.0, 6/0.3\}$$

$$B_2 = \{4/0.1, 5/0.9, 6/1.0\}$$

Chapter 21

Rough Sets

Fuzzy set theory is the first to have a theoretical treatment of the problem of vagueness and uncertainty, and have had many successful implementations. Fuzzy set theory is, however, not the only theoretical logic that addresses these concepts. Pawlak developed a new theoretical framework to reason with vague concepts and uncertainty [Pawlak 1982]. While rough set theory is somewhat related to fuzzy set theory, there are major differences.

Rough set theory is based on the assumption that some information, or knowledge, about the elements of the universe of discourse is initially available. This is contrary to fuzzy set theory where no such prior information is assumed. The information available about elements is used to find similar elements and indiscernible elements. Rough set theory is then based on the concepts of upper and lower approximations of sets. The lower approximation contains those elements that belong to the set with full certainty, while the upper approximation encapsulates elements for which membership is uncertain. The boundary region of a set, which is the difference between the upper and lower approximations, thus contains all examples which cannot be classified based on the available information.

Rough sets have been shown to be fruitful in a variety of application areas, including decision support, machine learning, information retrieval and data mining. What makes rough sets so desirable for real-world applications is their robustness to noisy environments, and situations where data is incomplete. It is a supervised approach, which clarifies the set-theoretic characteristics of classes over combinatorial patterns of the attributes. In doing so, rough sets also perform automatic feature selection by finding the smallest set of input parameters necessary to discern between classes.

The idea of discernibility is defined in Section 21.1, based on the formal definition of a decision system. Section 21.2 shows how rough sets treat vagueness by forming a boundary region, while Section 21.3 discusses the treatment of uncertainty in the implementation of the rough membership function.

21.1 Concept of Discernibility

The discussion on rough sets will be with reference to a decision system. Firstly, an information system is formally defined as an ordered pair $\mathcal{A} = (U, A)$, where U is the universe of discourse and A is a non-empty set of attributes. The universe of discourse is a set of objects (or patterns, examples), while the attributes define the characteristics of a single object. Each attribute $a \in A$ is a function $a : U \rightarrow V_a$, where V_a is the range of values for attribute a.

A decision system is an information system for which the attributes are grouped into disjoint sets of condition attributes and decision attributes. The condition attributes represent the input parameters, and the decision attributes represent the class.

The basic idea upon which rough sets rests is the discernibility between objects. If two objects are indiscernible over a set of attributes, it means that the objects have the same values for these attributes. Formally, the indiscernibility relation is defined as:

$$IND(B) = \{(x, y) \in U^2 | a(x) = a(y)\ \forall a \in B\}$$

where $B \subseteq A$. With $U/IND(B)$ is denoted the set of equivalence classes in the relation $IND(B)$. That is, $U/IND(B)$ contains one class for each set of objects that satisfy $IND(B)$ over all attributes in B. Objects are therefore grouped together, where the objects in different groups cannot be discerned between.

A discernibility matrix is a two-dimensional matrix where the equivalence classes form the indices, and each element is the set of attributes which can be used to discern between the corresponding classes. Formally, for a set of attributes $B \subseteq A$ in $\mathcal{A} = (U, A)$, the discernibility matrix $M_D(B)$ is defined as

$$M_D(B) = \{m_D(i, j)\}_{n \times n}$$

for $1 \leq i, j \leq n$, and $n = |U/IND(B)|$, with

$$m_D(i, j) = \{a \in B | a(E_i) \neq a(E_j)\}$$

for $i, j = 1, \cdots, n$; $a(E_i)$ indicates that attribute a belongs to equivalence class E_i.

Using the discernibility matrix, discernibility functions can be defined to compute the minimal number of attributes necessary to discern equivalence classes from one another. The discernibility function $f(B)$, with $B \subseteq A$, is defined as

$$f(B) = \wedge_{i,j \in \{1 \cdots n\}} \vee \overline{m}_D(E_i, E_j)$$

where

$$\overline{m}_D(i, j) = \{\overline{a} | a \in m_D(i, j)\}$$

and $\overline{a}$ is the Boolean variable associated with a, and $n = |U/IND(B)|$; $\vee \overline{m}_D(E_i, E_j)$ is the disjunction over the set of Boolean variables, and $\wedge$ denotes conjunction.

The discernibility function $f(B)$ finds the minimal set of attributes required to discern any equivalence class from all others. Alternatively, the relative discernibility function $f(E, B)$ finds the minimal set of attributes required to discern a given class, E, from the other classes, using the set of attributes, B. That is,

$$f(E, B) = \wedge_{j \in \{1 \cdots n\}} \vee \overline{m}_D(E, E_j)$$

It is now possible to find all dispensible, or redundant, attributes. An attribute $a \in B \subseteq A$ is dispensible if $IND(B) = IND(B - \{a\})$. Using the definition of dispensibility, a reduct of $B \subseteq A$ is the set of attributes $B^{'} \subseteq B$ such that all $a \in B - B^{'}$ are dispensible, and $IND(B) = IND(B^{'})$. The reduct of B is denoted by $RED(B)$, while $RED(E, B)$ denotes the relative reduct of B for equivalence class E. A relative reduct contains sufficient information to discern objects in one class from all other classes.

21.2 Vagueness in Rough Sets

Vagueness in rough set theory, where vagueness is with reference to concepts (e.g. a tall person), is based on the definition of a boundary region. The boundary region is defined in terms of an upper and lower approximation of the set under consideration.

Consider the set $X \subseteq U$, and the subset of attributes $B \subseteq A$. The lower approximation of X with regard to B is defined as

$$\underline{B}X = \cup\{E \in U/IND(B) | E \subseteq X\}$$

and the upper approximation of X,

$$\overline{B}X = \cup\{E \in U/IND(B) | E \cap X \neq \emptyset\}$$

The lower approximation is the set of objects which can be classified with full certainty as members of X, while the upper approximation is the set of objects that may possibly be classified as belonging to X.

The region,

$$BN_B(X) = \overline{B}X - \underline{B}X$$

is defined as the B-boundary of X. If $BN_B(X) = \emptyset$, then X is crisp with reference to B. If $BN_B(X) \neq \emptyset$, then X is rough with reference to B.

Rough sets can thus be seen as a mathematical model of vague concepts. Vagueness can then be defined as

$$\alpha_B(X) = \frac{|\underline{B}X|}{|\overline{B}X|}$$

with $\alpha_B(X) \in [0, 1]$. If $\alpha_B(X) = 1$, the set X is crisp, otherwise X is rough.

21.3 Uncertainty in Rough Sets

A vague concept has a non-empty boundary region, where the elements of that region cannot be classified with certainty as members of the concept. All elements in the boundary region of a rough set therefore have an associated degree of membership, calculated using the rough membership function for a class E,

$$\mu_B^X(E, X) = \frac{|E \cap X|}{|E|}$$

with $\mu_B^X(E, X) \in [0, 1]$, $E \in U/IND(B)$ and $X \subseteq U$.

Using the rough membership function, the following definitions are valid:

$$\begin{aligned} \underline{B}X &= \{x \in U | \mu_B^X(x) = 1\} \\ \overline{B}X &= \{x \in U | \mu_B^X(x) > 0\} \\ BN_B(X) &= \{x \in U | 0 < \mu_B^X(x) < 1\} \end{aligned}$$

The above shows that vagueness can, in fact, be defined in terms of uncertainty.

Some properties of the rough membership function are summarized below:

- $\mu_B^X(x) = 1$ iff $x \in \underline{B}X$
- $\mu_B^X(x) = 0$ iff $x \in U - \overline{B}X$
- $0 < \mu_B^X(x) < 1$ iff $x \in BN_B(X)$
- Complement: $\mu_B^{U-X}(x) = 1 - \mu_B^X(x)$ for any $x \in U$
- Union: $\mu_B^{X \cup Y}(x) \geq \max\{\mu_B^X(x), \mu_B^Y(x)\}$ for any $x \in U$
- Intersection: $\mu_B^{X \cap Y}(x) \leq \min\{\mu_B^X(x), \mu_B^Y(x)\}$ for any $x \in U$

21.4 Conclusion

This chapter gave a short overview of rough set theory, mainly to illustrate a different set theoretic approach to vagueness and uncertainty, and to show the equivalences and differences with fuzzy set theory. There is a vast source of literature on rough sets, and the reader is encouraged to read more on this very young field in Computer Science.

21.5 Assignments

1. Compare fuzzy sets and rough sets to show their similarities and differences.
2. Discuss the validity of the following two statements:
   (a) two-valued sets form a subset of rough sets
   (b) two-valued sets form a subset of fuzzy sets
   (c) fuzzy sets are special kinds of rough sets.
3. Discuss how rough sets can be used as classifiers.

Chapter 22

CONCLUSION

This book presented a short overview of four paradigms within computational intelligence (CI). These paradigms include artificial neural networks (NN), evolutionary computing (EC), swarm intelligence (SI) and fuzzy systems (FS). The intention was to provide the reader with an overview of the popular CI tools and with some of the current active research directions. The intention was by no means to provide a complete treatment of these paradigms. The interested reader should supplement the material presented with the vast source of information available in journals, conference proceedings, books and on the Internet. The hope is that the book did not just provide new knowledge to the first reader in CI, but also that the material provided some insight to current CI practitioners and researchers – even just by raising new ideas.

While the book left the treatment of CI at studying each paradigm individually, these paradigms can be combined successfully to form efficient hybrid models to solve complex problems. One such example is the use of particle swarm optimization to train NNs. Many other hybrid models exist, with much available literature on this. Agent-based CI tools are also of much interest, where agents of different CI models work together in a cooperative environment to solve problems. For such agent-based approaches the main problem is in developing a strategy of cooperation and exchange of information. Here ideas from swarm intelligence may be very helpful.

Both hybrid CI systems and agent technologies are such large research fields, that this book cannot devote any time on these to be fair. However, excellent books on the topics do exist, and the reader is referred to these.

As a final thought, or rather instruction to the reader, let the ideas flow ...

Bibliography

[Abu-Mostafa 1989] YS Abu-Mostafa, The Vapnik-Chervonenkis Dimension: Information versus Complexity in Learning, *Neural Computation*, Vol 1, 1989, pp 312–317.

[Abu-Mostafa 1993] YS Abu-Mostafa, Hints and the VC Dimension, *Neural Computation*, Vol 5, 1993, pp 278–288.

[Akaike 1974] H Akaike, A New Look at Statistical Model Identification, *IEEE Transactions on Automatic Control*, Vol 19, No 6, 1974, pp 716–723.

[Amari *et al.* 1995] S Amari, N Murata, K-R Müller, M Finke, H Yanh, *Asymptotic Statistical Theory of Overtraining and Cross-Validation*, Technical Report METR 95-06, Department of Mathematical Engineering and Information, University of Tokyo, 1995.

[Amari *et al.* 1996] S Amari, N Murata, K-R Müller, M Finke, H Yang, Statistical Theory of Overtraining – Is Cross-Validation Asymptotically Effective?, in DS Touretzky, MC Mozer, ME Hasselmo (eds), *Advances in Neural Information Processing Systems*, Vol 8, 1996, pp 176–182.

[Angeline 1998] PJ Angeline, Evolutionary Optimization versus Particle Swarm Optimization: Philosophy and Performance Differences, *Proceedings of the 7th International Conference on Evolutionary Programming*, pp 601–610, 1998.

[Angeline 1999] PJ Angeline, Using Selection to Improve Particle Swarm Optimization, *Proceedings of the IEEE International Joint Conference on Neural Networks*, pp 84–89, 1999.

[Bäck 1996] T Bäck, *Evolutionary Algorithms Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, 1996.

[Barnard 1991] E Barnard, Performance and Generalization of the Classification Figure of Merit Criterion Function, *IEEE Transactions on Neural Networks*, Vol 2, No 2, 1991, pp 322–325.

[Battiti 1992] R Battiti, First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method, *Neural Computation*, Vol 4, 1992, pp 141–166.

[Baum 1988] EB Baum, On the Capabilities of Multilayer Perceptrons, *Journal of Complexity*, Vol 4, 1988, pp 193–215.

[Baum and Haussler 1989] EB Baum, D Haussler, What Size Net Gives Valid Generalization?, in DS Touretzky (ed), *Advances in Neural Information Processing Systems*, Vol 1, 1989, pp 81–90.

[Beasley *et al.* 1993] D Beasley, R Bull, RR Martin, *An Overview of Genetic Algorithms: Part I, Fundamentals*, University Computing, 1993.

[Becker and Le Cun 1988] S Becker, Y Le Cun, Improving the Convergence of Back-Propagation Learning with Second Order Methods, in DS Touretzky, GE Hinton, TJ Sejnowski (eds), *Proceedings of the 1988 Connectionist Summer School*, Morgan Kaufmann, 1988.

[Belue and Bauer 1995] LM Belue, KW Bauer, Determining Input Features for Multilayer Perceptrons, *Neurocomputing*, Vol 7, 1995, pp 111–121.

[Bilchev and Parmee 1995] G Bilchev, IC Parmee, The Ant Colony Metaphor for Searching Continuous Design Spaces, *Proceedings of the AISB Workshop on Evolutionary Computing*, in *Lecture Notes in Computer Science*, Vol 993, pp 25–39, 1995.

[Bishop 1992] C Bishop, Exact Calculation of the Hessian Matrix for the Multilayer Perceptron, *Neural Computation*, Vol 4, 1992, pp 494–501.

[Bonabeau *et al.* 1999] E Bonabeau, M Dorigo, G Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, 1999.

[Bös 1996] S Bös, Optimal Weight Decay in a Perceptron, *Proceedings of the International Conference on Artificial Neural Networks*, 1996, pp 551–556.

[Breiman 1996] L Breiman, Bagging Predictors, *Machine Learning*, Vol 24, No 2, pp 123–140, 1996.

[Buntine and Weigend 1994] WL Buntine, AS Weigend, Computing Second Order Derivatives in Feed-Forward Networks: A Review, *IEEE Transactions on Neural Networks*, Vol 5, No 3, 1994, pp 480–488.

[Burrascano 1993] P Burrascano, A Pruning Technique Maximizing Generalization, *Proceedings of the 1993 International Joint Conference on Neural Networks*, Vol 1, 1993, pp 347–350.

[Chauvin 1989] Y Chauvin, A Back-Propagation Algorithm with Optimal use of Hidden Units, in DS Touretzky (ed), *Advances in Neural Information Processing Systems*, Vol 1, 1989, pp 519–526.

[Chauvin 1990] Y Chauvin, Dynamic Behavior of Constrained Back-Propagation Networks, in DS Touretzky (ed), *Advances in Neural Information Processing Systems*, Vol 2, 1990, pp 642–649.

[Cibas *et al.* 1996] T Cibas, F Fogelman Soulié, P Gallinari, S Raudys, Variable Selection with Neural Networks, *Neurocomputing*, Vol 12, 1996, pp 223–248.

[Cichocki and Unbehauen 1993] A Cichocki, R Unbehauen, *Neural Networks for Optimization and Signal Processing*, Wiley, 1993.

[Clerc and Kennedy 2002] M Clerc, J Kennedy, The Particle Swarm: Explosion, Stability, and Convergence in a Multi-Dimensional Complex Space, *IEEE Transactions on Evolutionary Computation*, Vol 6, pp 58–73, 2002.

[Cloete and Ludik 1993] I Cloete, J Ludik, Increased Complexity Training, *International Workshop on Artificial Neural Networks*, in *New Trends in Neural Computation*, J Mira, J Cabestany, A Prieto (eds), in series Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1993, pp 267–271.

[Cloete and Ludik 1994a] I Cloete, J Ludik, Incremental Training Strategies, *Proceedings of the International Conference on Artificial Neural Networks*, Sorrento, Italy, Vol 2, May 1994, pp 743–746.

[Cloete and Ludik 1994b] I Cloete, J Ludik, Delta Training Strategies, *IEEE World Congress on Computational Intelligence, Proceedings of the International Joint Conference on Neural Networks*, Orlando, USA, Vol 1, 1994, pp 295–298.

[Cohn and Tesauro 1991] D Cohn, G Tesauro, Can Neural Networks do Better than the Vapnik-Chervonenkis Bounds?, in R Lippmann, J Moody, DS Touretzky (eds), *Advances in Neural Information Processing Systems*, Vol 3, 1991, pp 911–917.

[Cohn 1994] DA Cohn, *Neural Network Exploration using Optimal Experiment Design*, AI Memo No 1491, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1994.

[Cohn *et al.* 1994] D Cohn, L Atlas, R Ladner, Improving Generalization with Active Learning, *Machine Learning*, Vol 15, 1994, pp 201–221.

[Cohn *et al.* 1996] DA Cohn, Z Ghahramani, MI Jordan, Active Learning with Statistical Models, *Journal of Artificial Intelligence Research*, Vol 4, 1996, pp 129–145.

[Colorni *et al.* 1994] A Colorni, M Dorigo, V Maniezzo, M Trubian, Ant System for Job Scheduling, *Belgian Journal of Operations Research, Statistics and Computer Science*, Vol 34, 1994, pp 39–53.

[Corne *et al.* 1999] D Corne, M Dorigo, F Glover (eds), *New Ideas in Optimization*, McGraw-Hill, 1999.

[Cosnard *et al.* 1992] M Cosnard, P Koiran, H Paugam-Moisy, Complexity Issues in Neural Network Computations, *Proceedings of LATIN92*, São Paulo, Brazil, 1992.

[Costa and Hertz 1997] D Costa, A Hertz, Ants Can Colour Graphs, *Journal of Operations Research Society*, Vol 48, 1997, pp 295–305.

[Cottrell *et al.* 1994] M Cottrell, B Girard, Y Girard, M Mangeas, C Muller, SSM: A Statistical Stepwise Method for Weight Elimination, *Proceedings of the International Conference on Artificial Neural Networks*, Vol 1, 1994, pp 681–684.

[Cowan and Reynolds 1999] GS Cowan, RG Reynolds, Learning to Assess the Quality of Genetic Programs using Cultural Algorithms, *Proceedings of the International Congress on Evolutionary Computing*, Vol 3, 1999, pp 1679–1686.

[Czernichow 1996] T Czernichow, Architecture Selection through Statistical Sensitivity Analysis, *Proceedings of the International Conference on Artificial Neural Networks*, 1996, pp 179–184.

[Darken and Moody] C Darken, J Moody, Note on Learning Rate Schedules for Stochastic Optimization, in R Lippmann, J Moody, DS Touretzky (eds), *Advances in Neural Information Processing Systems*, Vol 3, 1991.

[Darwin 1859] CR Darwin, *On the Origin of Species by Means of Natural Selection or Preservation of Favoured Races in the Struggel for Life*, Murray, London, 1859 (New York: Modern Library, 1967).

[Davis 1991] L Davis, Hybridization and Numerical Representation, in L Davis (ed), *The Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991, pp 61–71.

[Depenau and Møller 1994] J Depenau, M Møller, Aspects of Generalization and Pruning, *IEEE World Congress on Computational Intelligence, Proceedings of the International Joint Conference on Neural Networks*, Vol 3, 1994, pp 504–509.

[Di Caro and Dorigo 1998] G Di Caro, M Dorigo, AntNet: Distributed Stigmergetic Control for Communications Networks, *Journal of Artificial Intelligence Research*, Vol 9, 1998, pp 317–365.

[Dorigo 1999] M Dorigo, *Artificial Life: The Swarm Intelligence Approach*, Tutorial TD1, Congress on Evolutionary Computing, Washington, DC, 1999.

[Dorigo and Di Caro 1999] M Dorigo, G Di Caro, The Ant Colony Optimization Meta-Heuristic, in D Corne, M Dorigo, F Glover (eds), *New Ideas in Optimization*, McGraw-Hill, 1999.

[Dorizzi *et al.* 1996] B Dorizzi, G Pellieux, F Jacquet, T Czernichow, A Muñoz, Variable Selection using Generalized RBF Networks: Application to the Forecast of the French T-Bonds, *Proceedings of Computational Engineering in Systems Applications*, Lille, France, 1996, pp 122–127.

[Drucker 1999] H Drucker, Boosting using Neural Networks, in A Sharkey (ed), *Combining Artificial Neural Nets, Perspectives in Neural Computing*, Springer, 1999, pp 51–78.

[Duch and Korczak 1998] W Duch, J Korczak, Optimization and Global Minimization Methods Suitable for Neural Networks, *Neural Computing Surveys*, Vol 2, 1998.

[Durbin and Rumelhart 1989] R Durbin, DE Rumelhart, Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks, *Neural Computation*, Vol 1, 1989, pp 133–142.

[Durham 1994] W Durham, *Co-Evolution: Genes, Culture, and Human Diversity*, Stanford University Press, 1994.

[Eberhart *et al.* 1996] RC Eberhart, RW Dobbins and P Simpson, *Computational Intelligence PC Tools*, Academic Press, 1996.

[Eberhart and Kennedy 2001] RC Eberhart, J Kennedy, *Swarm Intelligence*, Morgan Kaufmann, 2001.

[Engelbrecht *et al.* 1995a] AP Engelbrecht, I Cloete, J Geldenhuys, JM Zurada, Automatic Scaling using Gamma Learning for Feedforward Neural Networks, International Workshop on Artificial Neural Networks, Torremolinos, Spain, June 1995, in J Mira, F Sandoval (eds), *From Natural to Artificial Neural Computing*, in the series Lecture Notes in Computer Science, Vol 930, pp 374–381.

[Engelbrecht *et al.* 1995b] AP Engelbrecht, I Cloete, JM Zurada, Determining the Significance of Input Parameters using Sensitivity Analysis, International Workshop on Artificial Neural Networks, Torremolinos, Spain, June 1995, in J Mira, F Sandoval (eds), *From Natural to Artificial Neural Computing*, in the series Lecture Notes in Computer Science, Vol 930, pp 382–388.

[Engelbrecht and Cloete 1996] AP Engelbrecht, I Cloete, A Sensitivity Analysis Algorithm for Pruning Feedforward Neural Networks, *Proceedings of the IEEE*

International Conference in Neural Networks, Washington, Vol 2, 1996, pp 1274–1277.

[Engelbrecht and Cloete 1998a] AP Engelbrecht, I Cloete, Selective Learning using Sensitivity Analysis, *IEEE World Congress on Computational Intelligence, International Joint Conference on Neural Networks*, Anchorage, Alaska, 1998, pp 1150–1155.

[Engelbrecht and Cloete 1998b] AP Engelbrecht, I Cloete, Feature Extraction from Feedforward Neural Networks using Sensitivity Analysis, *International Conference on Systems, Signals, Control, Computers*, Durban, South Africa, Vol 2, 1998, pp 221–225.

[Engelbrecht and Ismail 1999] AP Engelbrecht, A Ismail, Training Product Unit Neural Networks, *Stability and Control: Theory and Applications*, Vol 2, No 1/2, 1999.

[Engelbrecht *et al.* 1999] AP Engelbrecht, L Fletcher, I Cloete, Variance Analysis of Sensitivity Information for Pruning Feedforward Neural Networks, *Proceedings of the IEEE International Joint Conference on Neural Networks*, Washington, DC, USA, 1999, paper 379.

[Engelbrecht and Cloete 1999] AP Engelbrecht, I Cloete, Incremental Learning using Sensitivity Analysis, *Proceedings of the IEEE International Joint Conference on Neural Networks*, Washington, DC, USA, 1999, paper 380.

[Engelbrecht 2000] AP Engelbrecht, Data Generation using Sensitivity Analysis, *International Symposium on Computational Intelligence*, Slovakia, 2000.

[Engelbrecht 2001] AP Engelbrecht, A New Pruning Heuristic Based on Variance Analysis of Sensitivity Information, *IEEE Transactions on Neural Networks*, Vol 12, No 6, 2001.

[Engelbrecht 2001b] AP Engelbrecht, Sensitivity Analysis for Selective Learning by Feedforward Neural Networks, *Fundamenta Informaticae*, IOS Press, Vol 45, No 1, 2001, pp 295–328.

[Engelbrecht *et al.* 2002] AP Engelbrecht, S Rouwhorst, L Schoeman, A Building Block Approach to Genetic Programming for Rule Discovery, in HA Abbass, RA Sarker, CS Newton (eds), *Data Mining: A Heuristic Approach*, Idea Group Publishing, 2002, pp 174–189.

[Fahlman 1989] SE Fahlman, it Fast Learning Variations on Back-Propagation: An Empirical Study, in DS Touretzky, GE Hinton, TJ Sejnowski (eds), *Proceedings of the 1988 Connectionist Summer School*, Morgan Kaufmann, 1988, pp 38–51.

[Favilla *et al.* 1993] J Favilla, A Machion, F Gomide, Fuzzy Traffic Control: Adaptive Strategies, *IEEE Symposium on Fuzzy Systems*, San Francisco, 1993.

[Finnoff *et al.* 1993] W Finnoff, F Hergert, HG Zimmermann, Improving Model Selection by Nonconvergent Methods, *Neural Networks*, Vol 6, 1993, pp 771–783.

[Fletcher *et al.* 1998] L Fletcher, V Katkovnik, FE Steffens, AP Engelbrecht, Optimizing the Number of Hidden Nodes of a Feedforward Artificial Neural Network, *IEEE World Congress on Computational Intelligence, Proceedings of the International Joint Conference on Neural Networks*, Anchorage, Alaska, 1998, pp 1608–1612.

[Fogel *et al.* 1966] LJ Fogel, AJ Owens, MJ Walsh, *Artificial Intelligence Through Simulated Evolution*, Wiley, 1966.

[Fogel 1992] D Fogel, Evolving Artificial Intelligence, PhD Thesis, University of California, San Diego, 1992.

[Freund and Schapire 1999] Y Freund, RE Schapire, A Short Introduction to Boosting, *Journal of Japanese Society for Artificial Intelligence*, Vol 14, No 5, 1999, pp 771–780.

[Fritzke 1995] B Fritzke, Incremental Learning of Local Linear Mappings, *Proceedings of the International Conference on Artificial Neural Networks*, Paris, October 1995.

[Fujita 1992] O Fujita, Optimization of the Hidden Unit Function in Feedforward Neural Networks, *Neural Networks*, Vol 5, 1992, pp 755–764.

[Fukumizu 1996] K Fukumizu, Active Learning in Multilayer Perceptrons, in DS Touretzky, MC Mozer, ME Hasselmo (eds), *Advances in Neural Information Processing Systems*, Vol 8, 1996, pp 295–301.

[Gedeon *et al.* 1995] TD Gedeon, PM Wong, D Harris, Balancing Bias and Variance: Network Topology and Pattern Set Reduction Techniques, International Workshop on Artificial Neural Networks, in J Mira, F Sandoval (eds), *From Natural to Artificial Neural Computing*, in the series Lecture Notes in Computer Science, Vol 930, 1995, pp 551–558.

[Geman *et al.* 1992] S Geman, E Bienenstock, R Dousart, Neural Networks and the Bias/Variance Dilemma, *Neural Computation*, Vol 4, 1992, pp 1–58.

[Ghosh and Shin 1992] J Ghosh, Y Shin, Efficient Higher-Order Neural Networks for Classification and Function Approximation, *International Journal of Neural Systems*, Vol 3, No 4, 1992, pp 323–350.

[Giarratano 1998] JC Giarratano, *Expert Systems: Principles and Programming*, Third Edition, PWS Publishing, 1998.

[Girosi *et al.* 1995] F Girosi, M Jones, T Poggio, Regularization Theory and Neural Networks Architectures, *Neural Computation*, Vol 7, 1995, pp 219–269.

[Goldberg 1989] DE Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[Goldberg and Deb 1991] GE Goldberg, K Deb, A Comparative Analysis of Selection Schemes used in Genetic Algorithms, in GJE Rawlins (ed), *Foundations of Genetic Algorithms*, Morgan Kaufmann, 1991, pp 69–93.

[Grosso 1985] P Grosso, Computer Simulations of Genetic Application: Parallel Subcomponent Interaction in a Multilocus Model, PhD Thesis, University of Michigan, 1985.

[Gu and Takahashi 1997] H Gu, H Takahashi, Estimating Learning Curves of Concept Learning, *Neural Networks*, Vol 10, No 6, 1997, pp 1089–1102.

[Hagiwara 1993] M Hagiwara, Removal of Hidden Units and Weights for Back Propagation Networks, *Proceedings of the 1993 International Joint Conference on Neural Networks*, Vol 1, 1993, pp 351–354.

[Hanson and Pratt 1989] SJ Hanson, LY Pratt, Comparing Biases for Minimal Network Construction with Back-Propagation, in DS Touretzky (ed), *Advances in Neural Information Processing Systems*, Vol 1, 1989, pp 177–185.

[Hassibi and Stork 1993] B Hassibi, DG Stork, Second Order Derivatives for Network Pruning: Optimal Brain Surgeon, in C Lee Giles, SJ Hanson, JD Cowan (eds), *Advances in Neural Information Processing Systems*, Vol 5, 1993, pp 164–171.

[Hassibi *et al.* 1994] B Hassibi, DG Stork, G Wolff, Optimal Brain Surgeon: Extensions and Performance Comparisons, in JD Cowan, G Tesauro, J Alspector (eds), *Advances in Neural Information Processing Systems*, Vol 6, 1994, pp 263–270.

[Haussler *et al.* 1992] D Haussler, M Kearns, M Opper, R Schapire, Estimating Average-Case Learning Curves using Bayesian, Statistical Physics and VC Dimension Method, in J Moody, SJ Hanson, R Lippmann (eds), *Advances in Neural Information Processing Systems*, Vol 4, 1992, pp 855–862.

[Hayashi 1993] M Hayashi, A Fast Algorithm for the Hidden Units in a Multilayer Perceptron, *Proceedings of the 1993 International Joint Conference on Neural Networks*, Vol 1, 1993, pp 339–342.

[Haykin 1994] S Haykin, *Neural Networks: A Comprehensive Foundation*, MacMillan, 1994.

[Hirose *et al.* 1991] Y Hirose, K Yamashita, S Hijiya, Back-Propagation Algorithm which Varies the Number of Hidden Units, *Neural Networks*, Vol 4, 1991, pp 61–66.

[Hole 1996] A Hole, Vapnik-Chervonenkis Generalization Bounds for Real Valued Neural Networks, *Neural Computation*, Vol 8, 1996, pp 1277–1299.

[Holland 1975] JH Holland, *Adaptation in Natural and Artificial Systems*, Ann Arbor, MI: University of Michigan Press, 1975.

[Holland *et al.* 1986] JH Holland, KJ Holyoak, RE Nisbett, PR Thagard, *Induction: Processes of Inference, Learning, and Discovery*, MIT Press, 1986.

[Holland 1990] JH Holland, ECHO: Explorations of Evolution in a Miniature World, in JD Farmer, J Doyne (eds), *Proceedings of the Second Conference on Artificial Life*, Addison-Wesley, 1990.

[Holmström and Koistinen 1992] L Holmström, P Koistinen, Using Additive Noise in Back-Propagation Training, *IEEE Transactions on Neural Networks*, Vol 3, 1992, pp 24–38.

[Horn 1993] J Horn, N Nafpliotis, DE Goldberg, A Niched Pareto Genetic Algorithm for Multiobjective optimization, *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, Vol 1, pp 82–87, 1994.

[Horn 1997] J Horn, The Nature of Niching: Genetic Algorithms and the Evolution of Optimal, Cooperative Populations, PhD Thesis, University of Illinois at Urbana-Champaign, 1997.

[Hornik 1989] K Hornik, Multilayer Feedforward Networks are Universal Approximators, *Neural Networks*, Vol 2, 1989, pp 359–366.

[Huber 1981] PJ Huber, *Robust Statistics*, John Wiley & Sons, 1981.

[Hüning 1993] H Hüning, A Node Splitting Algorithm that Reduces the Number of Connections in a Hamming Distance Classifying Network, International Workshop on Artificial Neural Networks, in J Mira, J Cabestanyy, A Prieto (eds), *New Trends in Neural Computation*, in the series Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1993, pp 102–107.

[Hunt and Deller 1995] SD Hunt, JR Deller (Jr), Selective Training of Feedforward Artificial Neural Networks using Matrix Perturbation Theory, *Neural Networks*, Vol 8, No 6, 1995, pp 931–944.

[Hush *et al.* 1991] DR Hush, JM Salas, B Horne, Error Surfaces for Multi-Layer Perceptrons, *International Joint Conference on Neural Networks*, Vol 1, 1991, pp 759–764.

[Hussain *et al.* 1997] A Hussain, JJ Soraghan, TS Durbani, A New Neural Network for Nonlinear Time-Series Modelling, *NeuroVest Journal*, Jan 1997, pp 16–26.

[Hwang *et al.* 1991] J-N Hwang, JJ Choi, S Oh, RJ Marks II, Query-Based Learning Applied to Partially Trained Multilayer Perceptrons, *IEEE Transactions on Neural Networks*, Vol 2, No 1, 1991, pp 131–136.

[Ismail and Engelbrecht 2002] A Ismail, AP Engelbrecht, Improved Product Unit Neural Networks, *IEEE World Congress Computational Intelligence, Proceedings of the International Joint Conference on Neural Networks*, Honolulu, Hawaii, 2002.

[Jacobs 1988] RA Jacobs, Increased Rates of Convergence Through Learning Rate Adaption, *Neural Networks*, Vol 1, No 4, 1988, pp 295–308.

[Janikow and Michalewicz 1991] CZ Janikow, Z Michalewicz, An Experimental Comparison of Binary and Floating Point Representations in Genetic Algorithms, in RK Belew, LB Booker (eds), *Proceedings of the 4th International Conference in Genetic Algorithms*, Morgan Kaufmann, 1991, pp 31–36.

[Janson and Frenzel 1993] DJ Janson, JF Frenzel, Training Product Unit Neural Networks with Genetic Algorithms, *IEEE Expert*, Oct 1993, pp 26–33.

[Jantzen 1998] J Jantzen, *Design of Fuzzy Controllers*, Technical Report 98-E864, Department of Automation, Technical University of Denmark, 1998.

[Kamruzzaman *et al.* 1992] J Kamruzzaman, Y Kumagai, H Hikita, Study on Minimal Net Size, Convergence Behavior and Generalization Ability of Heterogeneous Backpropagation Network, in I Aleksander, J Taylor (eds), *Artificial Neural Networks*, Vol 2, 1992, pp 203–206.

[Kamimura 1993] R Kamimura, Principal Hidden Unit Analysis: Generation of Simple Networks by Minimum Entropy Method, *Proceedings of the 1993 International Joint Conference on Neural Networks*, Vol 1, 1993, pp 317–320.

[Kamimura and Nakanishi 1994] R Kamimura, S Nakanishi, Weight Decay as a Process of Redundancy Reduction, *IEEE World Congress on Computational Intelligence, Proceedings of the International Joint Conference on Neural Networks*, Vol 3, 1994, pp 486–491.

[Karner 1967] S Karner, Laws of Thought, *Encyclopedia of Philosophy*, Vol 4, MacMillan, 1967, pp 414–417.

[Kaski *et al.* 2000] S Kaski, J Venna, T Kohonen, Coloring that Reveals Cluster Structures in Multivariate Data, *Australian Journal of Intelligent Information Processing Systems*, Vol 6, 2000, pp 82–88.

[Kennedy and Eberhart 1995] J Kennedy, RC Eberhart, Particle Swarm Optimization, *Proceedings of the IEEE International Conference on Neural Networks*, Vol 4, pp 1942–1948, 1995.

[Kennedy and Eberhart 1997] J Kennedy, RC Eberhart, A Discrete Binary Version of the Particle Swarm Algorithm, *Proceedings of the Conference on Systems, Man, and Cybernetics*, 1997, pp 4104–4109.

[Kennedy 1998] J Kennedy, The Behavior of Particles, in VW Porto, N Saravanan, D Waagen (eds), *Proceedings of the 7th International Conference on Evolutionary Programming*, 1998, pp 581–589.

[Kennedy 1999] J Kennedy, Small Worlds and Mega-Minds: Effects of Neighborhood Topology on Particle Swarm Performance, *IEEE Congress on Evolutionary Computation*, Vol 3, 1999, pp 1931–1938.

[Kennedy and Eberhart 1999] J Kennedy, RC Eberhart, The Particle Swarm: Social Adaptation in Information-Processing Systems, D Corne, M Dorigo, F Glover (eds), *New Ideas in Optimization*, McGraw-Hill, 1999, pp 379–387.

[Kohara 1995] K Kohara, Selective Presentation Learning for Forecasting by Neural Networks, *International Workshop on Applications of Neural Networks in Telecommunications*, Stockolm, Sweden, 1995, pp 316–323.

[Kohonen 1987] T Kohonen, *Context-Addressable memories*, Second Edition, Springer-Verlag, 1987.

[Kohonen 1995] T Kohonen, *Self-Organizing Maps*, Springer Series in Information Sciences, 1995.

[Kohonen 1997] T Kohonen, *Self-Organizing Maps*, Second Edition, Springer, 1997.

[Koza 1992] JR Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[Krogh and Hertz 1992] A Krogh, JA Hertz, A Simple Weight Decay can Improve Generalization, in J Moody, SJ Hanson, R Lippmann (eds), *Advances in Neural Information Processing Systems*, Vol 4, 1992, pp 950–957.

[Kuscu and Thornton 1994] I Kuscu, C Thornton, *Design of Artificial Neural Networks Using Genetic Algorithms: Review and Prospect*, Technical Report, Cognitive and Computing Sciences, University of Sussex, 1994.

[Kwok and Yeung 1995] T-Y Kwok, D-Y Yeung, *Constructive Feedforward Neural Networks for Regression Problems: A Survey*, Technical Report HKUST-CS95-43, Department of Computer Science, The Hong Kong University of Science & Technology, 1995.

[Lange and Männer 1994] R Lange, R Männer, Quantifying a Critical Training Set Size for Generalization and Overfitting using Teacher Neural Networks, *International Conference on Artificial Neural Networks*, Vol 1, 1994, pp 497–500.

[Lange and Zeugmann 1996] S Lange, T Zeugmann, Incremental Learning from Positive Data, *Journal of Computer and System Sciences*, Vol 53, 1996, pp 88–103.

[Le Cun 1990] Y Le Cun, JS Denker, SA Solla, Optimal Brain Damage, in D Touretzky (ed), *Advances in Neural Information Processing Systems*, Vol 2, 1990, pp 598–605.

[Le Cun *et al.* 1991] Y Le Cun, I Kanter, SA Solla, Second Order Properties of Error Surfaces: Learning Time and Generalization, in RP Lippmann, JE Moody, DS Touretzky (eds), *Advances in Neural Information Processing Systems*, Vol 3, 1990, pp 918–924.

[Leerink *et al.* 1995] LR Leerink, C Lee Giles, BG Horne, MA Jabri, Learning with Product Units, *Advances in Neural Information Processing Systems*, Vol 7, 1995, pp 537–544.

[Lejewski 1967] C Lejewski, Jan Lukasiewicz, *Encyclopedia of Philosophy*, Vol 5, MacMillan, 1967, pp 104- -107.

[Levin *et al.* 1994] AU Levin, TK Leen, JE Moody, Fast Pruning using Principal Components, in JD Cowan, G Tesauro, J Alspector (eds), *Advances in Neural Information Processing Systems*, Vol 6, 1994, pp 35–42.

[Liang *et al.* 2001] K-H Liang, X Yao, CS Newton, Adapting Self-Adaptive Parameters in Evolutionary Algorithms, *Applied Intelligence*, Vol 15, No 3, 2001, pp 171–180.

[Lim and Ho 1994] S-F Lim, S-B Ho, Dynamic Creation of Hidden Units with Selective Pruning in Backpropagation, *IEEE World Congress on Computational Intelligence, Proceedings of the International Joint Neural Networks*, Vol 3, June 1994, pp 492–497.

[Løvbjerg *et al.* 2001] M Løvbjerg, TK Rasmussen, T Krink, Hybrid Particle Swarm Optimiser with Breeding and Subpopulations, *Proceedings of the Genetic and Evolutionary Computation Conference*, San Francisco, 2001.

[Ludik and Cloete 1993] J Ludik, I Cloete, Training Schedules for Improved Convergence, *IEEE International Joint Conference on Neural Networks*, Nagoya, Japan, Vol 1, 1993, pp 561–564.

[Ludik and Cloete 1994] J Ludik, I Cloete, Incremental Increased Complexity Training, *European Symposium on Artificial Neural Networks*, Brussels, Belgium, April 1994, pp 161–165.

[Luger and Stubblefield 1997] GF Luger, WA Stubblefield, *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*, Third Edition, Addison-Wesley, 1997.

[Lumer and Faieta 1994] E Lumer, B Faieta, Diversity and Adaptation in Populations of Clustering Ants, *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, Vol 3, pp 499–508, MIT Press, 1994.

[Lumer and Faieta 1995] E Lumer, B Faieta, Exploratory Database Analysis via Self-Organization, unpublished paper, 1995.

[MacKay 1992] DJC MacKay, Bayesian Methods for Adaptive Models, PhD Thesis, California Institute of Technology, 1992.

[Magoulas *et al.* 1997] GD Magoulas, MN Vrahatis, GS Androulakis, Effective Backpropagation Training with Variable Stepsize, *Neural Networks*, Vol 10, No 1, 1997, pp 69–82.

[Mahfoud 1995] SW Mahfoud, Niching Methods for Genetic Algorithms, PhD Thesis, University of Illinois at Urbana-Champaign, 1995.

[Mamdani *et al.* 1975] EH Mamdani, S Assilian, An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller, *International Journal of Man-Machine Studies*, Vol 7, 1975, pp 1–13.

[Maniezzo *et al.* 1994] V Maniezzo, A Colorni and M Dorigo, *The Ant System Applied to the Quadratic Assignment Problem*, Technical Report IRIDIA/94-28, Université Libre de Bruxelles, Belgium, 1994.

[Marais 1970] EN Marais, *The Soul of the White Ant*, Human and Rousseau Publishers, Cape Town, 1970.

[Marais 1969] EN Marais, *The Soul of the Ape*, London 1969; second edition, Hammersworth, 1973.

[Mayr 1963] E Mayr, *Animal Species and Evolution*, Belknap, Cambridge, MA, 1963.

[Michalewicz 1994] Z Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Second Edition, Springer Verlag, 1994.

[Michel and Middendorf 1998] R Michel, M Middendorf, An Island Model Based Ant System with Lookahead for the Shortest Supersequence Problem, *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature*, 1998, pp 692–701.

[Møller 1993] MF Møller, A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning, *Neural Networks*, Vol 6, 1993, pp 525–533.

[Mollestad 1997] T Mollestad, A Rough Set Approach to Data Mining: Extracting a Logic of Default Rules from Data, PhD Thesis, Department of Computer Science, The Norwegian University of Science and Technology, 1997.

[Moody 1992] JE Moody, The Effective Number of Parameters: An Analysis of Generalization and Regularization in Nonlinear Learning Systems, J Moody, SJ Hanson, R Lippmann (eds), *Advances in Neural Information Processing Systems*, Vol 4, 1992, pp 847–854.

[Moody 1994] JE Moody, Prediction Risk and Architecture Selection for Neural Networks, in V Cherkassky, JH Friedman, H Wechsler (eds), *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*, Springer, 1994, pp 147–165.

[Moody and Utans 1995] J Moody, J Utans, Architecture Selection Strategies for Neural Networks: Application to Corporate Bond Rating Prediction, in AN Refenes (ed), *Neural Networks in the Capital Markets*, John Wiley & Sons, 1995.

[Mozer and Smolensky 1989] MC Mozer, P Smolensky, Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment, in DS Touretzky (ed), *Advances in Neural Information Processing Systems*, Vol 1, 1989, pp 107–115.

[Müller *et al.* 1995] K-R Müller, M Finke, N Murata, K Schulten, S Amari, A Numerical Study on Learning Curves in Stochastic Multi-Layer Feed-Forward Networks, *Neural Computation*, Vol 8, No 5, 1995, pp 1085–1106.

[Murata *et al.* 1991] N Murata, S Yoshizawa, S Amari, A Criterion for Determining the Number of Parameters in an Artificial Neural Network Model, in T Kohonen, K Mäkisara, O Simula, J Kangas (eds), *Artificial Neural Networks*, Elsevier Science Publishers, 1991, pp 9–14.

[Murata *et al.* 1994a] N Murata, S Yoshizawa, S Amari, Learning Curves, Model Selection and Complexity of Neural Networks, in C Lee Giles, SJ Hanson, JD Cowan (eds), *Advances in Neural Information Processing Systems*, Vol 5, 1994, pp 607–614.

[Murata *et al.* 1994b] N Murata, S Yoshizawa, S Amari, Network Information Criterion – Determining the Number of Hidden Units for an Artificial Neural Network Model, *IEEE Transactions on Neural Networks*, Vol 5, No 6, 1994, pp 865–872.

[Nilsson 1998] NJ Nilsson, *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann, 1998.

[Nowlan and Hinton 1992] SJ Nowlan, GE Hinton, Simplifying Neural Networks By Soft Weight-Sharing, *Neural Computation*, Vol 4, 1992, pp 473–493.

[Ohnishi *et al.* 1990] N Ohnishi, A Okamoto, N Sugiem, Selective Presentation of Learning Samples for Efficient Learning in Multi-Layer Perceptron, *Proceedings of the IEEE International Joint Conference on Neural Networks*, Vol 1, 1990, pp 688–691.

[Oja 1982] E Oja, A Simplified Neuron Model as a Principal Component Analyzer, *Journal of Mathematical Biology*, Vol 15, 1982, pp 267–273.

[Oja and Karhuner 1985] E Oja, J Karhuner, On Stochastic Approximation of the Eigenvectors and Eigenvalues of the Expectation of a Random Matrix, *Journal of Mathematical Analysis and Applications*, Vol 104, 1985, pp 69–84.

[Opper 1994] M Opper, Learning and Generalization in a Two-Layer Neural Network: The Role of the Vapnik-Chervonenkis Dimension, *Physical Review Letters*, Vol 72, No 13, 1994, pp 2133–2166.

[Orr and Leen 1993] GB Orr, TK Leen, Momentum and Optimal Stochastic Search, in MC Mozer, P Smolensky, DS Touretzky, JL Elman, AS Weigend (eds), *Proceedings of the 1993 Connectionist Models Summer School*, Erlbaum Associates, 1993.

[Ostrowski and Reynolds 1999] DA Ostrwoski, RG Reynolds, Knowledge-Based Software Testing Agent using Evolutionary Learning with Cultural Algorithms, *IEEE International Congress on Evolutionary Computation*, Vol 3, 1999, pp 1657–1663.

[Pawlak 1982] Z Pawlak, Rough Sets, *International Journal of Computer and Information Sciences*, Vol 11, 1982, pp 341–356.

[Pedersen *et al.* 1996] MW Pedersen, LK Hansen, J Larsen, Pruning with Generalization Based Weight Saliencies: γ OBD, γ OBS, in DS Touretzky, MC Mozer, ME Hasselmo (eds), *Advances in Neural Information Processing Systems*, Vol 8, 1996, pp 521–528.

[Plaut *et al.* 1986] DA Plaut, S Nowlan, G Hinton, *Experiments on Learning by Back Propagation*, Technical Report CMU-CS-86-126, Department of Computer Science, Carnegie-Mellon University, 1986.

[Plutowski and White 1993] M Plutowski, H White, Selecting Concise Training Sets from Clean Data, *IEEE Transactions on Neural Networks*, Vol 4, No 2, March 1993, pp 305–318.

[Polkowski and Skowron 1998] L Polkowski, A Skowron (eds), *Rough Sets in Knowledge Discovery 2: Applications, Case Studies, and Software Systems*, Springer Verlag, 1998.

[Potter 1997] MA Potter, The Design and Analysis of a Computational Model of Cooperative Coevolution, PhD Thesis, George Mason University, Fairfax, Virginia, USA, 1997.

[Prechelt 1995] L Prechelt, *Adaptive Parameter Pruning in Neural Networks*, Technical Report TR-95-009, International Computer Science Institute, Berkeley, California, 1995.

[Rechenberg 1973] I Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der Biologischen Evolution*, Frammann-Holzboog Verlag, Stuttgart, 1973.

[Rechenberg 1994] I Rechenberg, Evolution Strategy, in JM Zurada, R Marks II, C Robinson (eds), *Computational Intelligence: Imitating Life*, IEEE Press, 1994, pp 147–159.

[Reed 1993] R Reed, Pruning Algorithms – A Survey, *IEEE Transactions on Neural Networks*, Vol 4, No 5, 1993, pp 740–747.

[Reed *et al.* 1995] R Reed, RJ Marks II, S Oh, Similarities of Error Regularization, Sigmoid Gain Scaling, Target Smoothing, and Training with Jitter, *IEEE Transactions on Neural Networks*, Vol 6, 1995, pp 529–538.

[Reynolds 1979] RG Reynolds, An Adaptive Computer Model of the Evolution of Agriculture, PhD Thesis, University of Michigan, Michgan, 1979.

[Reynolds 1991] RG Reynolds, Version Space Controlled Genetic Algorithms, *Proceedings of the Second Annual Conference on Artificial Intelligence Simulation and Planning in High Autonomy Systems*, IEEE Press, 1991, pp 6-14.

[Reynolds 1994] RG Reynolds, An Introduction to Cultural Algorithms, *Proceedings of the Third Annual Conference on Evolutionary Computing*, 1994, pp 131–139.

[Reynolds 1999] RG Reynolds, Cultural Algorithms: Theory and Application, in D Corne, M Dorigo, F Glover (eds), *New Ideas in Optimization*, MacGraw-Hill, 1999, pp 367–366.

[Ripley 1996] BD Ripley, *Pattern Recognition and Neural Networks*, Cambridge University Press, 1996.

[Röbel 1994] A Röbel, *The Dynamic Pattern Selection Algorithm: Effective Training and Controlled Generalization of Backpropagation Neural Networks*, Technical Report, Institut für Angewandte Informatik, Technische Universität Berlin, March 1994, pp 497–500.

[Rosen and Goodwin 1997] BE Rosen, JM Goodwin, Optimizing Neural Networks Using Very Fast Simulated Annealing, *Neural, Parallel & Scientific Computations*, Vol 5, No 3, 1997, pp 383–392.

[Rosin and Belew 1996] CD Rosin, RK Belew, *New Methods for Competitive Coevolution*, Cognitive Computer Science Research Group, Department of Computer Science and Engineering, University of California, Technical Report #CS96-491, 1996.

[Rouwhorst and Engelbrecht 2000] SE Rouwhorst, AP Engelbrecht, Searching the Forest: Using Decision Trees as Building Blocks for Evolutionairy Search in Classification Databases, *Proceedings of the IEEE International Conference on Evolutionary Computation*, San Diego, USA, 2000.

[Rychtyckyj and Reynolds 1999] N Rychtyckyj, RG Reynolds, Using Cultural Algorithms to Improve Performance in Semantic Networks, *Proceedings of the IEEE International Congress on Evolutionary Computation*, Washington DC, Vol 3, 1999, pp 1651–1656.

[Sakurai 1992] A Sakurai, n-h-1 Networks Use No Less than nh+1 Examples, but Sometimes More, *Proceedings of the IEEE*, Vol 3, 1992, pp 936–941.

[Salomon and Van Hemmen 1996] R Salomon, JL Van Hemmen, Accelerating Backpropagation through Dynamic Self-Adaptation, *Neural Networks*, Vol 9, No 4, 1996, pp 589–601.

[Sanger 1989] T Sanger, Optimal Unsupervised Learning in a Single-Layer Linear Feedforward Neural Network, *Neural Networks*, Vol 2, 1989, pp 459–473.

[Sartori and Antsaklis 1991] MA Sartori, PJ Antsaklis, A Simple Method to Derive Bounds on the Size and to Train Multilayer Neural Networks, *IEEE Transactions on Neural Networks*, Vol 2, No 4, 1991, pp 467–471.

[Schwefel 1974] H-P Schwefel, *Adaptive Mechanismen in der biologischen Evolution und ihr Einfluβ auf die Evolutionsgeschwindigkeit*, Technical Report, Technical University of Berlin, 1974.

[Schwefel 1975] H-P Schwefel, *Evolutionsstrategie und numerische Optimierung*, PhD Thesis, Technical University Berlin, 1975.

[Schwefel 1981] H-P Schwefel, *Numerical Optimization of Computer Models*, Wiley, 1981.

[Schwefel 1995] H-P Schwefel, *Evolution and Optimum Seeking*, Wiley, 1995.

[Schittenkopf *et al.* 1997] C Schittenkopf, G Deco, W Brauer, Two Strategies to Avoid Overfitting in Feedforward Neural Networks, *Neural Networks*, Vol 10, No 30, 1997, pp 505–516.

[Schoonderwoerd *et al.* 1996] R Schoonderwoerd, O Holland, J Bruten, L Rothkrantz, Ant-Based Load Balancing in Telecommunications Networks, *Adaptive Behaviour*, Vol 5, 1996, pp 169–207.

[Schwartz *et al.* 1990] DB Schwartz, VK Samalam, SA Solla, JS Denker, Exhaustive Learning, *Neural Computation*, Vol 2, 1990, pp 374–385.

[Sejnowski 1977] T Sejnowski, Storing Covariance with Nonlinearly Interacting Neurons, *Journal of Mathematical Biology*, Vol 4, 1997, pp 303–321.

[Seung *et al.* 1992] HS Seung, M Opper, H Sompolinsky, Query by Committee, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, 1992, pp 287–299.

[Sevenster and Engelbrecht 1996] S Sevenster, AP Engelbrecht, GARTNet: A Genetic Algorithm for Routing in Telecommunications Networks, *Proceedings of CESA96 IMACS Multiconference on Computational Engineering in Systems Applications, Symposium on Control, Optimization and Supervision*, Lille, France, Vol 2, 1996, pp 1106–1111.

[Shi and Eberhart 1999] Y Shi, RC Eberhart, Empirical Study of Particle Swarm Optimization, *Proceedings of the IEEE Congress on Evolutionary Computation*, Vol 3, 1999, pp 1945–1950.

[Sietsma and Dow 1991] J Sietsma, RJF Dow, Creating Artificial Neural Networks that Generalize, *Neural Networks*, Vol 4, 1991, pp 67–79.

[Slade and Gedeon 1993] P Slade, TD Gedeon, Bimodal Distribution Removal, International Workshop on Artificial Neural Networks, in J Mira, J Cabestany, A Prieto (eds), *New Trends in Neural Computation*, in the series Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1993, pp 249–254.

[Snyman 1982] JA Snyman, A New and Dynamic Method for Unconstrained Minimization, *Applied Mathematical Modelling*, Vol 6, 1982, pp 449–462.

[Snyman 1983] JA Snyman, An Improved Version of the Original LeapFrog Dynamic Method for Unconstrained Minimization: LFOP1(b), *Applied Mathematical Modelling*, Vol 7, 1983, pp 216–218.

[Spector and Luke 1996] L Spector, S Luke, Cultural Transmission of Information in Genetic Programming, *Proceedings of the First Annual Conference on Genetic Programming*, 1996, pp 209–214.

[Steppe *et al.* 1996] JM Steppe, KW Bauer, SK Rogers, Integrated Feature and Architecture Selection, *IEEE Transactions on Neural Networks*, Vol 7, No 4, 1996, pp 1007–1014.

[Su *et al.* 1999] MC Su, TA Liu, HT Chang, An Efficient Initialization Scheme for the Self-Organizing Feature Map Algorithm, *Proceedings of the IEEE International Joint Conference in Neural Networks*, 1999.

[Suganthan 1999] PN Suganthan, Particle Swarm Optimizer with Neighborhood Operator, *Proceedings of the IEEE Congress on Evolutionary Computation*, 1999, pp 1958–1961.

[Sung and Niyogi 1996] KK Sung, P Niyogi, *A Formulation for Active Learning with Applications to Object Detection*, AI Memo No 1438, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1996.

[Sverdlik *et al.* 1992] W Sverdlik, RG Reynolds, E Zannoni, HYBAL: A Self-Tuning Algorithm for Concept Learning in Highly Autonomous Systems, *Proceedings of the Third Annual Conference on Artificial Intelligence, Simulation, and Planning in High Autonomy Systems*, IEEE Computer Society Press, 1992, pp 15–22.

[Syswerda 1991] G Syswerda, Schedule Optimization using Genetic Algorithms, in L Davis (ed), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991, pp 332–349.

[Takagi and Sugeno 1985] T Takagi, M Sugeno, Fuzzy Identification of Systems and its Application to Modeling and Control, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol 15, No 1, 1985, pp 116–132.

[Takahashi 1993] T Takahashi, Principal Component Analysis is a Group Action of SO(N) which Minimizes an Entropy Function, *Proceedings of the 1993 International Joint Conference on Neural Networks*, Vol 1, 1993, pp 355–358.

[Tamura *et al.* 1993] S Tamura, M Tateishi, M Matumoto, S Akita, Determination of the Number of Redundant Hidden Units in a Three-layer Feed-Forward Neural Network, *Proceedings of the 1993 International Joint Conference on Neural Networks*, Vol 1, 1993, pp 335–338.

[Teller 1994] A Teller, The Evolution of Mental Models, in KE Kinnear (ed), *Advances in Genetic Programming*, MIT Press, 1994, pp 199–219.

[Thiele 1998] H Thiele, *Fuzzy Rough Sets versus Rough Fuzzy Sets – An Interpretation and a Comparative Study using Concepts of Modal Logics*, Technical Report CI-30/98, University of Dortmund, April 1998.

[Thodberg 1991] HH Thodberg, Improving Generalization of Neural Networks through Pruning, *International Journal of Neural Systems*, Vol 1, No 4, 1991, pp 317–326.

[Turing 1950] AM Turing, Computing Machinery and Intelligence, *Mind*, Vol 59, 1950, pp 433–460.

[Van den Bergh 1999] F van den Bergh, Particle Swarm Weight Initialization in Multi-Layer Perceptron Artificial Neural Networks, *Proceedings of the International Conference on Artificial Intelligence*, 1999, pp 41–45.

[Van den Bergh and Engelbrecht 2000] F van den Bergh, AP Engelbrecht, Cooperative Learning in Neural Networks using Particle Swarm Optimizers, *South African Computer Journal*, No 26, 2000, pp 84–90.

[Van den Bergh and Engelbrecht 2001] F van den Bergh, AP Engelbrecht, Using Cooperative Particle Swarm Optimization to Train Product Unit Neural Networks, *Proceedings of the IEEE International Joint Conference on Neural Networks*, Washington, DC, 2001.

[Van den Bergh 2002] F van den Bergh, An Analysis of Particle Swarm Optimizers, PhD Thesis, Department of Computer Science, University of Pretoria, 2002.

[Viktor *et al.* 1995] HL Viktor, AP Engelbrecht, I Cloete, Reduction of Symbolic Rules from Artificial Neural Networks using Sensitivity Analysis, *Proceedings of the IEEE International Conference on Neural Networks*, Perth, Australia, 1995, pp 1788–1793.

[Vogl *et al.* 1988] TP Vogl, JK Mangis, AK Rigler, WT Zink, DL Alken, Accelerating the Convergence of the Back-Propagation Method, *Biological Cybernetics*, Vol 59, 1988, pp 257–263.

[Wang and Mendel 1992] LX Wang, JM Mendel, Generating Fuzzy Rules by Learning from Examples, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol 22, No 6, 1992, pp 1413–1426.

[Weigend *et al.* 1991] AS Weigend, DE Rumelhart, BA Huberman, Generalization by Weight-Elimination with Application to Forecasting, in R Lippmann, J Moody, DS Touretzky (eds), *Advances in Neural Information Processing Systems*, Vol 3, 1991, pp 875–882.

[Werbos 1974] PJ Werbos, Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences, PhD thesis, Harvard University, Boston, USA, 1974.

[Wessels and Barnard 1992] LFA Wessels, E Barnard, Avoiding False Local Minima by Proper Initialization of Connections, *IEEE Transactions on Neural Networks*, Vol 3, No 6, 1992, pp 899–905.

[Whitley *et al.* 1989] D Whitley, T Starkweather, D Fuquay, Scheduling Problems and the Travelings Salesmen: The Genetic Edge Recombination Operator, in JD Schaffer (ed), *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989, pp 116–121.

[Whitley and Bogart 1990] D Whitley, C Bogart, The Evolution of Connectivity: Pruning Neural Networks using Genetic Algorithms, *Proceedings of the IEEE International Joint Conference on Neural Networks*, Vol 1, 1990, pp 134–137.

[White and Ligomenides 1993] D White, P Ligomenides, GANNet: A Genetic Algorithm for Optimizing Topology and Weights in Neural Network Design, International Workshop on Artificial Neural Networks, J Mira, J Cabestany, A Prieto (eds), *New Trends in Neural Computation*, in the series Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1993, pp 332–327.

[Widrow 1987] B Widrow, ADALINE and MADALINE – 1963, Plenary Speech, *Proceedings of the First IEEE International Joint Conference on Neural Networks*, Vol 1, San Diego, 1987, pp 148–158.

[Widrow and Lehr 1990] B Widrow, MA Lehr, 30 Years of Neural Networks: Perceptron, Madaline and Backpropagation, *Proceedings of the IEEE*, Vol 78, 1990, pp 1415–1442.

[Williams 1995] PM Williams, Bayesian Regularization and Pruning Using a Laplace Prior, *Neural Computation*, Vol 7, 1995, pp 117–143.

[Wolpert and Macready 1996] DH Wolpert, WG Macready, *No Free Lunch Theorems for Search*, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1996.

[Xue *et al.* 1990] Q Xue, Y Hu, WJ Tompkins, Analyses of the Hidden Units of Back-Propagation Model, *Proceedings of the IEEE International Joint Conference on Neural Networks*, Vol 1, 1990, pp 739–742.

[Yasui 1997] S Yasui, Convergence Suppression and Divergence Facilitation: Minimum and Joint Use of Hidden Units by Multiple Outputs, *Neural Networks*, Vol 10, No 2, 1997, pp 353–367.

[Yu and Chen 1997] X-H Yu, G-A Chen, Efficient Backpropagation Learning using Optimal Learning Rate and Momentum, *Neural Networks*, Vol 10, No 3, 1997, pp 517–527.

[Zadeh 1965] LA Zadeh, Fuzzy Sets, *Information and Control*, Vol 8, 1965, pp 338–353.

[Zadeh 1975] LA Zadeh, The Concept of a Linguistic Variable and its Application to Approximate Reasoning, Parts 1 and 2, *Information Sciences*, Vol 8, 1975, pp 338–353.

[Zhang 1994] B-T Zhang, Accelerated Learning by Active Example Selection, *International Journal of Neural Systems*, Vol 5, No 1, 1994, pp 67–75.

[Zhang and Kandel 1998] Y Zhang, A Kandel, *Compensatory Genetic Fuzzy Neural Networks and Their Applications*, World Scientific, 1998.

[Zurada 1992] J Zurada, Lambda Learning Rule for Feedforward Neural Networks, *Proceedings of the IEEE International Joint Conference in Neural Networks*, San Francisco, USA, 1992.

[Zurada *et al.* 1994] JM Zurada, A Malinowski, I Cloete, Sensitivity Analysis for Minimization of Input Data Dimension for Feedforward Neural Network, *IEEE International Symposium on Circuits and Systems*, London, May 30–June 3, 1994.

Further Reading

T Bäck, DB Fogel, A Michalewicz (eds), *Handbook of Evolutionary Computation*, IOP Publishers and Oxford University Press, 1997.

CM Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, 1995.

E Bonabeau, M Dorigo, G Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, 1999.

E Cox, *The Fuzzy Systems Handbook: A Practitioner's Guide to Building, Using, & Maintaining Fuzzy Systems*, Morgan Kaufmann Publishers, 1999.

E Fiesler, R Beale (eds), *Handbook of Neural Computation*, IOP Publishers and Oxford University Press, 1996.

DB Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, 1995.

DE Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

MH Hassoun, *Fundamentals of Artificial Neural Networks*, MIT Press, 1995.

JH Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, 1992.

J Kennedy, RC Eberhart, Y Shi, *Swarm Intelligence*, Morgan Kaufmann Publishers, 2001.

T Kohonen, TS Huang, *Self-Organizing Maps*, Springer Verlag, third edition, 2001.

M Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, 1998.

E Ruspini, P Bonissone, W Pedryc (eds), *Handbook of Fuzzy Computation*, IOP Publishers and Oxford University Press, 1998.

LA Zadeh, GJ Klir, B Yuan, *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A Zadeh*, World Scientific Publishing Company, 1996.

JM Zurada, *Introduction to Artificial Neural Systems*, PWS Publishing Company, 1999.

Appendix A

Acronyms

ACO	Ant Colony Optimization
AI	Artificial Intelligence
AN	Artificial Neuron
BGP	Building-Block Genetic Program
BMN	Best Matching Neuron
BN	Biological Neuron
CA	Cultural Algorithm
CAEP	Cultural Algorithm with Evolutionary Program
CCGA	Cooperative Coevolutionary Genetic Algorithm
CE	Cultural Evolution
CI	Computational Intelligence
CO	Classical Optimization
CoE	Coevolution
CoEA	Coevolution Algorithm
CPSO	Cooperative Particle Swarm Optimization
DE	Differential Evolution
EA	Evolutionary Algorithm
EC	Evolutionary Computing
EP	Evolutionary Program
ES	Evolutionary Strategy
FFNN	Feedforward Neural Network
FL	Fuzzy Logic
FLNN	Functional Link Neural Network
FS	Fuzzy Systems
FSM	Finite State Machine
GA	Genetic Algorithm
GCP	Graph Coloring Problem
GCPSO	Guaranteed Convergence Particle Swarm Optimization
GD	Gradient Descent

GP	Genetic Program
ISB	Integrated Squared Bias
JSP	Job-Scheduling Problem
LVQ	Learning Vector Quantizer
MPSO	Multi-start Particle Swarm Optimization
MSE	Mean Squared Error
NFL	No Free Lunch Theorem
NN	Neural Network
OBD	Optimal Brain Damage
OBS	Optimal Brain Surgeon
OCD	Optimal Cell Damage
OED	Optimal Experiment Design
PSO	Particle Swarm Optimization
PU	Product Unit
PUNN	Product Unit Neural Network
QAP	Quadratic Assignment Problem
RBF	Radial Basis Function
RPSO	Restart Particle Swarm Optimization
SCG	Scaled Conjugate Gradient
SCSP	Super Common Subsequence Problem
SI	Swarm Intelligence
SOM	Self-Organizing Map
SRNN	Simple Recurrent Neural Network
SSE	Sum Squared Error
SU	Summation Unit
SUNN	Summation Unit Neural Network
TDNN	Time-Delay Neural Network
TSP	Traveling Salesman Problem

Appendix B

Symbols

This appendix lists symbols used within this book. A separate list of symbols is given for the different chapters of the book. For each chapter, only new symbols are defined, or those symbols which are overloaded with new meaning.

B.1 Part II – Artificial Neural Networks

B.1.1 Chapters 2-3

α	momentum
d_p	p-th pattern, consisting of input and target vector
D	data set
D_G	generalization set
D_T	training set
D_V	validation set
$\delta_{y_j,p}$	error signal for hidden unit y_j for pattern p
$\delta_{o_k,p}$	error signal for output unit o_k for pattern p
$\mathcal{E}$	objective/error function
$\mathcal{E}'$	derivative of the objective/error function
$\mathcal{E}_G$	generalization error
$\mathcal{E}_T$	training error
$\mathcal{E}_V$	validation error
η	learning rate
$\eta(t)$	time-varying learning rate
f_{y_j}	activation function of hidden unit y_j
f'_{y_j}	derivative of activation function of hidden unit y_j
f_{o_k}	activation function of output unit o_k
f'_{o_k}	derivative of activation function of output unit o_k

f_p	neural network output for pattern p
i	input unit index
I	total number of input units (bias unit excluded)
j	hidden unit index
J	total number of hidden units (bias unit excluded)
k	output unit index
K	total number of output units
l	functional link units index
L	total number of functional link units
$N(0, \sigma^2)$	Gaussian distribution with 0 mean, and variance σ^2
$net_{y_j,p}$	net input signal to hidden unit y_j for pattern p
$net_{o_k,p}$	net input signal to output unit o_k for pattern p
o_k	k-th output unit
$o_{k,p}$	output of the k-th output unit for pattern p
p	pattern index
P	total number of patterns
P_G	number of patterns in generalization set
P_T	number of patterns in training set
P_V	number of patterns in validation set
t	time step
T	total number of delayed time steps for TDNN
t_p	target for pattern p
$t_{k,p}$	target for k-th output unit for pattern p
$U(a, b)$	uniform random number between a and b
u_{ki}	direct weight between output unit o_k and input unit z_i
Δu_{ki}	direct weight change between output unit o_k and input unit z_i
u_{li}	weight between functional link unit h_l and input unit z_i
v_{ji}	weight between hidden unit y_j and input unit z_i
Δv_{ji}	weight change between hidden unit y_j and input unit z_i
v_{jl}	weight between hidden unit y_j and functional link unit h_l
w_{kj}	weight between output unit o_k and hidden unit y_j
Δw_{kj}	weight change between output unit o_k and hidden unit y_j
ξ	epoch counter
y_j	j-th hidden unit
$y_{j,p}$	activation of hidden unit y_j
z_i	i-th input unit
$z_{i,p}$	value of input unit for pattern p
$z_{i,p}(t)$	value of input unit for pattern p, delayed for time step t

B.1.2 Chapter 4

c_{kj}	map coordinates of neuron kj

c_{mn}	map coordinates of the BMN
C_z	input covariance matrix
$d_{k,p}(\vec{z}_p, \vec{u}_k)$	Euclidean distance between input vector $\vec{z}_p$ and weight vector $\vec{u}_k$
$\mathcal{E}_T$	quantization error on training set
d_{rs}	Ward distance between clusters r and s
$\eta_k(t)$	time-varying learning rate for output unit o_k
$h_{mn,kj}$	neighborhood function for neuron mn
j	SOM column index
J	total number of columns in SOM
k	SOM row index
K	total number of rows in SOM
kj	index of a SOM neuron
$\kappa_{k,p}(t)$	set of neighbors of winning cluster o_k for pattern p
λ_k	k-th eigenvalue
mn	index of the BMN
n_r, n_s	number of patterns within clusters r and s respectively
N_{mn}	number of patterns for which neuron mn is the BMN
$\vec{o}(t)$	vector of output values at time step t
$\sigma(t)$	time-varying width of Gaussian function
$\vec{u}_k$	vector of weights leading to output unit o_k
$\vec{w}_{kj}$	weight vector of SOM neuron kj
$\Delta\vec{w}_{kj}$	weight vector changes for SOM neuron kj
w_{kji}	weight value from input z_i for neuron kj
$\vec{w}_r, \vec{w}_s$	centroid vector of clusters r and s respectively
$\vec{z}_p$	p-th input vector
Z	input matrix

B.1.3 Chapter 5

$\vec{\mu}_j$	mean vector of Gaussian kernel for hidden unit y_j
$\mathcal{P}_{k,p}$	degree to which pattern p belongs to class k for RBF NN
σ_j^2	variance of Gaussian kernel function for hidden unit y_j

B.1.4 Chapter 6

e_{kj}	eligibility of weight w_{kj}
r_p	external evaluator's indication of NN's success for pattern p
θ_k	reinforcement threshold value

B.1.5 Chapter 7

$\mathcal{A}^-$	selective learning operator
$\mathcal{A}^+$	incremental learning operator
D_C	candidate training set
D_S	training subset
$\overline{\mathcal{E}}_V$	average validation error
$\mathcal{F}_{NN}(D_T; W)$	neural network output function
$\sigma_{\mathcal{E}_V}$	standard deviation in validation error
r	correlation coefficient
ρ	Röbel's generalization factor

B.2 Part III – Evolutionary Computing

B_g	belief space at generation g
C_g	population at generation g
$\vec{C}_{g,n}$	n-th individual in a population at generation g
$\Delta\vec{C}_{g,n}$	change in n-th individual in a population at generation g
$C_{g,ni}$	i-th gene of n-th individual for generation g
$\overline{C}_{g,ni}$	complement of $C_{g,ni}$, if binary variable
$\vec{C}_n$	n-th individual in a population
$\mathcal{F}_{EA}$	fitness function for an evolutionary algorithm
g	generation counter
i	gene index
I	number of genes in an individual
$\vec{m}$	mask vector
n	index for individuals in a population
N	number of individuals in a population
$\vec{\mathcal{N}}_g$	normative knowledge component at generation g
$\mathcal{N}_{g,i}$	normative knowledge component for i-th gene at generation g
$\vec{O}_{g,n}$	n-th offspring at generation g
$\vec{O}_n$	n-th offspring
p_c	probability of cross-over
p_m	probability of mutation
p_n	probability of mutation per node in tree for GP
p_r	reproduction probability
p_x	probability of cross-over per bit for uniform cross-over
$\mathcal{S}_g$	situational knowledge component at generation g

B.3 Part IV – Swarm Intelligence

B.3.1 Chapter 17

c_1, c_2	acceleration constants
d_{max}	largest distance between any two particles (swarm diameter)
$\mathcal{F}$	fitness function
i	particle index
κ	constriction coefficient
P_i	denotes i-th particle
ϕ	inertia weight
$\phi(t)$	time-varying inertia weight
r_1, r_2	uniform random numbers
$r_i(t)$	time varying uniform random number associated with i-th particle
ρ_1, ρ_2	constants in velocity update equations
t	current iteration
t_{max}	maximum number of iterations
$\vec{v}_i(t)$	velocity of i-th particle at time step t
$\vec{v}_{ij}(t)$	velocity of j-th parameter for i-th particle at time step t
V_{max}	maximum particle velocity
$\vec{x}_i(t)$	position of i-th particle at time step t
$\vec{x}_{ij}(t)$	position of j-th parameter for i-th particle at time step t

B.3.2 Chapter 18

α	weight of pheromone intensity
β	weight of local information
$C_{i,k}$	set of nodes to be visited by ant k from node i
d_{ij}	Euclidean distance between nodes i and j
$d(\vec{z}_i, \vec{z}_j)$	dissimilarity between two objects
η_{ij}	local information between nodes i and j
f	fraction of objects in a given neighborhood
$f(\vec{z}_)$	fraction of $\vec{z}_i$ objects in a given neighborhood
i, j	node indices
k	ant index
(i, j)	link between nodes i and j
$L_k(t)$	length of route traveled by ant k
$\mathcal{N}_{s\times r}(r)$	neighborhood area around position r
p_p	probability of picking up an object
$p_p(\vec{z}_i)$	probability of picking up object $\vec{z}_i$
p_d	probability of dropping an object

$p_d(\vec{z}_i)$	probability of dropping object $\vec{z}_i$
$\Phi_{ij,k}(t)$	time-varying probability of ant k to follow link (i,j)
ρ	forgetting factor/rate of pheromone evaporation
s	neighborhood size
$\tau_{ij}(t)$	total pheromone intensity on edge (i,j)
$\Delta\tau_{ij}(t)$	change in pheromone intensity for edge (i,j)
$\Delta\tau_{ij,}(t)$	pheromone deposit on edge (i,j) by ant k
T^+	optimal route
$T_k(t)$	route constructed by ant k at time t
v	travel velocity of an ant
v_{max}	maximum travel velocity of an ant

B.4 Part V – Fuzzy Systems

B.4.1 Chapters 19-21

μ_A	membership function for fuzzy set A
$\mu_A(x)$	membership of x to fuzzy set A
n	number of elements in a discrete fuzzy set
$p(A)$	probability of event A
X	domain of a fuzzy set (universe of discourse
$\sum_{i=1}^{n}$	defining a discrete-domain fuzzy set
$\int_X$	defining a continuous-domain fuzzy set

B.4.2 Chapter 22

$\alpha_B(X)$	vagueness of concept B in X
A	set of attributes
$\mathcal{A}$	information system
$a(E_i)$	indicates that a belongs to equivalence class E_i
$\underline{B}$	rough set
$\overline{B}X$	upper approximation of X
$\underline{B}X$	lower approximation of X
$BN_B(X)$	B-boundary of X
E_i	an equivalence class
$f(B)$	discernibility function
$f(E,B)$	relative discernibility function
$IND(B)$	indiscernibility relation
$\mu_B^X(E,X)$	membership function for class E
$M_D(B)$	discernibility matrix

$RED(B)$	reduct of B
$RED(E, B)$	relative reduct of B
U	universe of discourse
$U/IND(B)$	equivalence classes in relation $IND(B)$
V_a	range of values of attribute a

Index