Neurocontrol of a Cantilever Beam

by

Nicolas Aplincourt

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

Author .....
Department of Civil and Environmental Engineering
December 20, 1999

Certified by .....
Jerome J.J. Connor
Professor, Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by .....
Daniele Veneziano
Chairman, Departmental Committee on Graduate Studies

Neurocontrol of a Cantilever Beam

by

Nicolas Aplincourt

Submitted to the Department of Civil and Environmental Engineering
on December 20, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The civil engineering community is currently moving towards the continuous monitoring of civil structures in order to forecast their unavoidable failure with enough precision. So-called smart technologies seem to be well adapted to this specific task.

For a civil structure, such as a bridge or a dam, a monitoring smart system often includes a set of sensors, whose data is passed onto a controller. The latter analyzes the data and outputs commands to a set of actuators that will modify the structure properties in response to the new sensors' environment. Therefore, the structure can continuously adapt to its surrounding environment.

Artificial neural networks are electronic devices whose structure resembles the structure of the human brain. Such devices can be trained to output desired signals when fed with specific inputs. Consequently, neural networks can theoretically act as controllers in monitoring smart systems.

This thesis first presents artificial neural networks in details, since this topic remains unfamiliar in the civil engineering literature. An entire chapter is also devoted to the training of these artificial neural networks that are likely to be used in civil engineering applications. The thesis then introduces the new concept of neurocontrol, i.e. control using neural networks. Finally, a simulation run under MATLAB applies this concept of neurocontrol to a cantilever beam supporting fluctuating loads.

Thesis Supervisor: Jerome J.J. Connor
Title: Professor, Department of Civil and Environmental Engineering

Acknowledgments

I would like to sincerely thank Professor Jerome J.J. Connor, my thesis advisor, who took the time to structure the writing of this thesis.

I am immensely grateful to the Rotary Foundation of Rotary International, who financially supported my graduate studies at MIT. In addition, members of the Rotary Club of Boston, especially my host, Peter Griffin, transformed my stay into a true American experience.

I wish to express my unconditional love to my brothers, Gilles and Philippe, my sister, Florence, and my grandparents, Yvonne, Jean, and Genevieve, whose phone calls and e-mails brightened my darkest days at the Institute.

Finally, this thesis is dedicated to my parents, Michel and Marie-Agnes, who instilled in me the value of continuous learning. To their education, I owe what I am today, and what I will achieve in the future.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Structural Control

The field of civil engineering is changing. Although civil engineers have the reputation of being utterly cautious about new technologies, the recent years have seen the emergence of new ideas on the building sites. The use of artificial neural networks is one of these ideas.

Artificial neural networks belong to a broader family of technologies used in civil engineering, the so-called smart technologies, which represent the context of this thesis. Artificial neural networks themselves do not form a smart technology. Rather, they are often bundled with a set of actuators and sensors, such as passive strain gauges, and incorporated into a structure, which suddenly acquires "smartness."

The very first question is why one needs structures with a degree of intelligence, instead of the existing bridges, buildings, or dams which just sit there, albeit beautifully. One reason is monitoring. As explained in section 1.2, there is a strong need for technologies that allow engineers to know the condition of every part of a structure at every second of its life.

Another reason is adaptability. It is valuable to know about the degree of change, of decay, of a structure. However, civil engineers want structures that can adapt themselves to their changing condition. The goal of having a structure adapt not only to its inner condition, but also to its surrounding environment, which comprises

elements such as wind, rain, or traffic, is now feasible.

That this kind of structures was tagged "smart" shows how much expectation civil engineers have put in the concept. Section 1.3 presents these new civil engineering design concepts. This section also shows where neural networks fit in the framework of a smart system. Namely, they are used as (neuro)controller.

This thesis focuses on this latter concept, namely, controlling elements of civil structures with neural networks. The first two chapters are dedicated to artificial neural networks, since this field of study is rather new in the civil engineering community. Then, neurocontrol, in parallel with the classical methods of control, is discussed. Finally, neurocontrol of a cantilever beam is simulated using the MATLAB environment.

1.2 The Need for Infrastructure Health Monitoring

Infrastructure is the focus of increasing public and government concern throughout the world. In the United States, hardly a week goes by without a major media report highlighting the failure of a bridge, a building, a pipeline, or some other civil structure. Earthquakes, floods, freezes and hurricanes exacerbate structural degradation due to the passage of time and daily use. As commuters, concert-goers, and apartment dwellers, we take the integrity of our highway bridges, stadiums and high-rises for granted and afford them a degree of permanency increasingly undeserved. Recent U.S. government studies conclude that structural failure and precautionary over-design, resulting from an inability to measure and predict impending failure, cost the U.S. economy over $100 billion every year. Despite this rather startling conclusion, little quantitative information is available to unambiguously document declining safety margins and rising maintenance requirements. Along with the post-war "baby-boom" in this country, came a "bridge-building boom", peaking in the fifties. Many of those structures were built with a 40 to 50 year design life, irrevocably marking our current

decade with the stigma of "suddenly" deficient structures.

The integrity of our infrastructure legacy can no longer be taken for granted. Yet in a society that bothers to track, record and monitor the billions of credit card transactions made every year, the duration and number of every telephone call and the temperature in every city every hour, why the structural health of our bridges and buildings is not monitored remains a mystery.

Federal law mandates a visual inspection of each highway bridge approximately every two years. For buildings, so long as they are built to code, there are, with few exceptions, no annual inspection requirements for their entire lives. These precedents were set in an era when the equipment required to obtain structural data was complex, the accuracy it returned questionable and the per-dollar benefits negligible. Today, that cost-benefit model has inverted. It is getting cheaper and easier by the day to take greater amounts of more and more accurate data. The technology to implement reliable and economical structural health monitoring systems, which can reduce the life cycle cost of maintaining safe infrastructure, is available. That, for both the producers of infrastructure and the consumers, is a new bottom line.

1.3 Smart Technology

Smart technology could be the answer to the problems described in the previous section.

Smart technology has already made its mark as a beneficial and practical discipline in many areas of science and engineering. Recently, this new way of thinking has gained more and more interest from the civil engineer. This short section describes what smart technology is.

The study of smart materials and structures is a field that has been cited by Scientific American as one of the key technologies for the 21st Century. A structure or material can be considered "smart" when it has the ability to sense internal or external conditions and respond in some manner appropriate to alter the effects of those conditions in a favorable way. "Smart" technology has been around for several

decades, debuting in the early 1960's when Corning Glass Works started developing a new kind of glass. This new "photochromic" glass was able to react to the amount of light present in its environment - automatically darkening in the light and automatically lightening in the dark. This is the same glass that now makes-up the large number of sunglasses with variable and self-adjusting transparency. The application of photochromic glass was not limited to sunglasses. It was soon expanded to include glass in buildings and automobiles and was considered for just about every application where glass or mirror was needed.

Soon after the introduction of photochromic glass, scientist, researchers and engineers began to recognize the potential for such "smart" technology. A material that could respond to its environment could be useful far beyond the glass industry. Designs incorporating piezo-electric ceramics and shape memory alloys were a few of the innovations to comprise the next wave of these new smart materials and structures.

More and more, designs are sought for materials and structures that not only serve the purpose for which they are contrived, but do so actively in the most efficient manner possible. "Smart technology" is not only a new kind of technology, it is a new way of thinking.

Smart technology quickly found its way into the designs of the aeronautical, aerospace and mechanical engineers. However, civil engineers are notorious for abiding by methods that are tried-and-tested, and for this reason smart technology is only slowly making its way into the civil engineering community. The most notable examples of smart designs in civil engineering are the active or intelligent buildings, several of which exist in Japan and only one or two of which exist in the United States. These buildings detect movements or vibrations caused by winds or earthquakes, and respond accordingly using damping systems that reduce the effects of such stimuli. Similar designs have been proposed to reduce the wind-induced vibrations on long-span bridges. However, unlike the active building systems, this smart design for bridges has not yet found its way into practical use.

Smart technology, like any newly emerging technology, is not without sizable cost. It is obvious that a smart building with fiber optics, sensors, actuators and other

"smart parts" would be more expensive than its "dumb" counterpart. But when the big picture is considered, the economical feasibility of a smart building becomes evident. Not only are resources such as concrete and steel reduced in a smart bridge, but more importantly, the cost of service and maintenance is reduced while the service life is greatly increased. Furthermore, the technology itself will decrease in cost as it becomes more widely used, thus making smart design more economical than current design.

1.3.1 What makes a Material or Structure Smart?

A material or structure can be considered "smart" when it has the ability to sense internal or external conditions and respond in some manner appropriate to alter the effects of those conditions in a favorable way. The difference between smart materials and smart structures is vague. Many smart structures are smart only so far as the materials they are comprised of are smart. Likewise, the "intelligence" of a lone smart material that is not incorporated in an overall structural system is contestable. Therefore the term "smart technology" has come to encompass not only smart materials and smart structures, but also the way of thinking in which the desired result is a system, structural or otherwise, that can detect and respond to external and/or internal stimuli. This can be as complex as the multi-story building that can detect large vibrations and counter them by actively altering the stiffness of individual structural members.

In general, smart systems include three basic components or ideas:

1. sensors, or the ability to sense or detect important internal or external information;

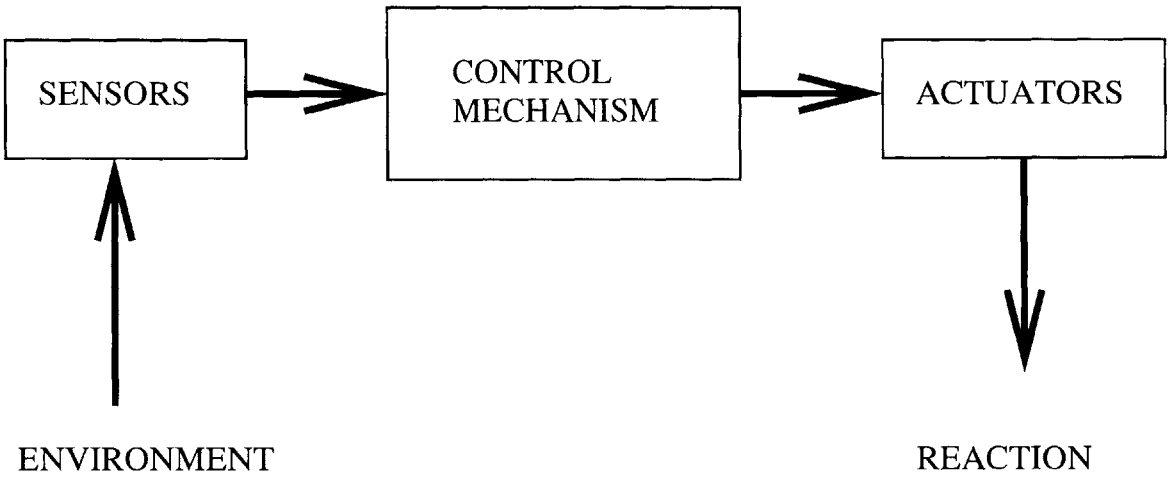
2. a control mechanism to act as the brain of the system. It interprets the information gathered by the sensors and decides on a course of action.

3. actuators, or the ability to respond, react, or in some way alter the state of the system; and

Note that in some systems, the only role of the actuator is to inform some third party of the information the sensors have collected, and not alter the system itself.

A general pattern of logic that all smarts systems follow is illustrated in Figure 1-1. All processes start with the acquisition of information by the sensors. This informa-

Figure 1-1: Logic of a smart system.



tion is then sent to some control mechanism for processing. The control mechanism is programmed to act on this information in a pre-specified way. If called-for, the control mechanism will signal the actuators to alter the system in some way. This entire process of sensing, control processing, and active response is repeated continuously in a smart system.

In order for this model to accurately represent all smart systems, each of the terms used should only be taken in its most generic form.

- By "sensor," it is meant anything that has the ability to collect any type of information, including material properties, mechanical properties, and environmental conditions, to name a few.

- By "actuator," it is meant anything that has the ability to produce an action, or otherwise alter the state, properties, or environment of a system.

- The control mechanism is anything that links the sensors to the actuators in a logically structured way. And this is where artificial neural networks can be of

great help. Indeed, they represent a possible link between signals output from the sensors and signals input to the actuators. This thesis examines the use of neural networks as possible control mechanisms for smart structures.

1.3.2 Existing Smart Materials

There are many kinds of smart materials out there, but some show more potential, and more promise than others. Research focuses on three main classes of smart materials:

1. Piezoelectric ceramics and polymers,
2. shape memory alloys, and
3. electrorheological or magnetorheological fluids.

Piezoelectric ceramics can act as either pressure sensors or mechanical actuators. The electric polarity of their crystal structures allow them to quickly transform any mechanical forces into electric current, or conversely, transform electrical current into mechanical vibrations. They can produce these mechanical vibrations at very high frequencies, and thus are of utmost importance in the development of smart systems that counteract damaging vibrations.

Shape memory alloys are better suited for slower, stronger responses. Below a certain temperature, a shape memory alloy will take on any shape it is bent into. But when heated back above this temperature, it will try to return to its original shape. If something is hindering the restoration of the alloy's shape, it will exert a constant force. This force is the result of the atoms in the alloy attempting to toggle between different geometric arrangements.

An electrorheological fluid is a fluid whose viscous properties may be modified by applying an electric field, and a **magnetorheological fluid** is one whose viscous properties are modified by applying a magnetic field. This change in the viscosity of

these actuator materials may be so extensive that they in effect change from liquid to solid. These fluids consist of fine polarizable particles of ceramic or polymer suspended in a liquid such as silicone oil. When an electric or magnetic field is applied, the particles "organize themselves into filaments and networks", thereby stiffening the material. When the electric field is removed, the process reverses and the organization of the particles disappears - the material becomes fluid again.

1.3.3 Existing Smart Structures

The difference between smart materials and smart structures can be seen as one of scale. The three components that lend smart materials their intelligence operate on a microscopic scale, while those of the smart structures operate on a macroscopic scale. Often in a smart structure, the sensors, actuators and control mechanisms are items that are used regularly in other fields, but not in such a way that they create a smart system.

A typical smart building designed to detect and counteract earthquake movements may include such items as accelerometers, actuators that operate on basic hydraulic principals, and a basic microcomputer as the control mechanism, to interpret the data gathered by the accelerometers and send the appropriate message to the hydraulic actuators. This type of smart structure is commonly known as an active bracing system.

A smart building has been erected on the campus of the University of Vermont incorporating fiber optics, thermistors, and strain gauges to collect different types of information from different parts of the structure. The sensors monitor the building for "cracks and vibrations due to wind, temperature changes, thermal expansion, and occupant loads." This building does not employ any actuators that can bring about a change independent of human interaction, but the control mechanism - the microcomputer that logs the information gathered by the numerous sensors can double as an actuator by alarming when action needs to be taken.

It is obvious that the same type of smart technology described above is useful far beyond the monitoring of buildings. This same system and the same techniques can

be used to monitor just about any structure made of materials, which includes every structure possible. The more researchers, scientists and engineers experiment with the ideas of smart technology, the more possibilities they uncover. They spread their findings to try and include other disciplines so that the overall benefit and value of their work is maximized.

1.4 Objectives and Organization of the Thesis

This thesis evolves in this context of smart technologies. The objectives are twofold, namely:

1. to present neural networks,

2. to demonstrate that neural networks can be used to control an element of a structure. The study is restricted to the control of an horizontal rectangular cantilever beam supporting fluctuating vertical loads.

The material is therefore divided into three section:

- The first part, which comprises chapter 2 "Foundation of Artificial Neural Networks" and chapter 3 "Improving the performance of Supervised Feed-Forward Neural Networks," is dedicated to neural networks. While chapter 2 introduces the basic concepts in this field of study, chapter 3 gives a more detailed description of the neural networks used in civil engineering applications.

- The second part, represented by chapter 4, is about neurocontrol, i.e. control using artificial neural networks. Indeed, as described in previous sections, people in civil engineering entertain the idea that neural networks should be used as control mechanism in smart structures. This chapter 4 presents these new advances in control.

- Ultimately, chapter 5 presents a case study in neurocontrol. Using the MATLAB environment, and more precisely its extensive neural network toolbox, a

neurocontroller is built whose task is to keep a cantilever beam as straight as possible, when it is subjected to loading.

Finally, the conclusion sums up the results of the case study, and proposes directions for further research in the area of neurocontrol.

Chapter 2

Foundation of Artificial Neural Networks

2.1 Definition of Artificial Neural Networks

2.1.1 The Beginning: Interest in the Human Brain

Understanding how the brain works has long been an area of interest to the scientific community. And although physiologists and cognitive scientists started their experiments on animals, their final goal has always been to understand the human brain, which is the most powerful of all brains. Physiological observations inside the brain yield information about the operations of neurons (brain cells) connected in a fixed configuration, but relatively little concerns the detailed temporal evolution of the connectivity structure among neurons.

Because scientists did not (and still do not) know how groups of neurons operate together functionally, the best they could to do at that time was to imitate the brain structure and hope that some of the functionality would be reproduced. Duplicating the structure in its entirety was impractical. However, there has been a well-established belief that much of the information processing in the brain is not only parallel, but also somewhat localized by function. Evidence from neuropathology, neurohistology, and neuropharmacology indicates that there are about 1,000 localized regions

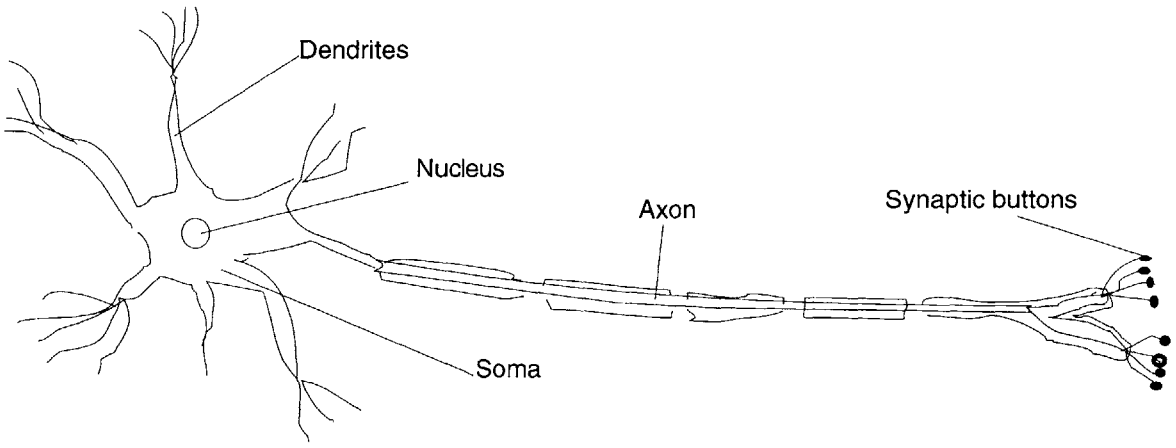
where the bulk of the computation takes place. Because they wanted imitate the behavior of sections of the brain, scientists came up with an artificial tool that locally copied the structure of the brain.

At this point, a brief discussion of brain physiology is presented. The human brain is composed of 10^9 to 10^{12} neurons, linked together to form a very complex network. This is acknowledged as a biological neural network. The anatomy of a biological neuron consists of:

- a branching structure, comprising what are called dendrites, where the neuron is believed to pick-up signals from other neurons;
- a cell-body called soma;
- a long transmission-linelike structure, called the axon; and
- brushlike structures at the tail of the axon, called synaptic buttons.

Figure 2-1 shows the above structure.

Figure 2-1: A biological neuron



The points at which neurons come into close proximity with one another are called synapses. At these points of "contact," neurons influence each other electrochemically. A synapse, in reality, is not a physical connection. When a signal arrives at a synapse, it elicits the release of a neurotransmitter (chemicals present in the

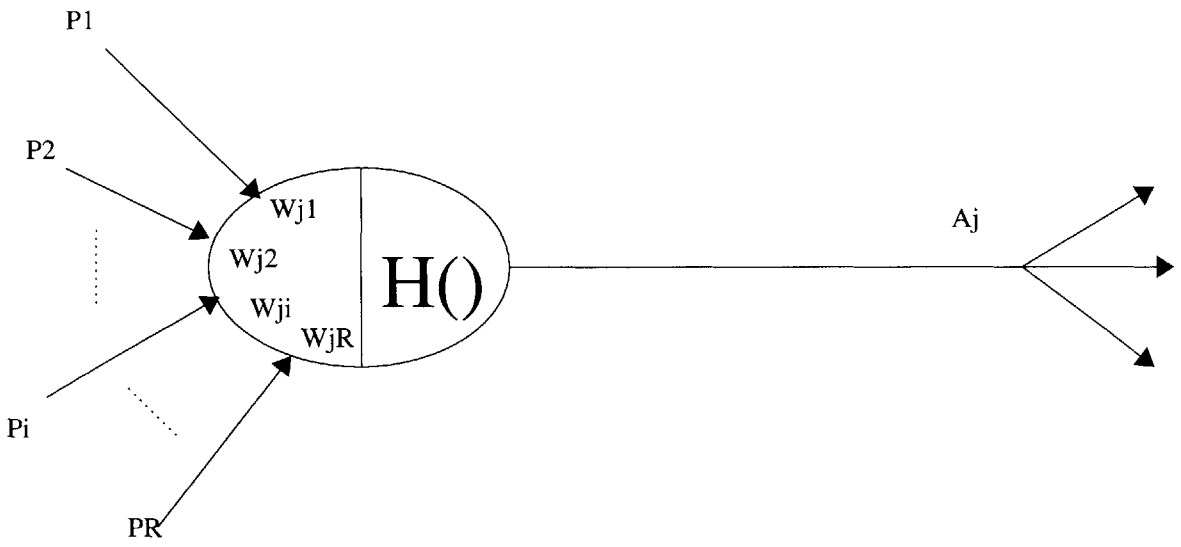
brain), which builds up until its concentration exceed a certain threshold. When this happens, an action potential is elicited in the receiving cell.

A neuron receives signals (typically in the form of a train of pulses) from its neighbors via the synapses, performs a weighted algebraic summation of the inputs, computes a thresholding function of this sum, and when the function value exceed the threshold, produces an output (or "fires"). Because of the one-way transmission at the synapses, both the input and the output pulses normally travel one way from the dendrites, to the soma, on to the axon, and finally to the synaptic button. It is estimated that a typical neuron receives its inputs from as many as 10,000 other neurons and sends its output to perhaps 1,000 neurons.

2.1.2 A First Model

With the information from the previous section, one can already propose a model. Figure 2-2 presents the simplest mathematical abstraction of a single biological (i.e. real) neuron:

Figure 2-2: A first neuron model: $a_j = f(n_j) = H(\sum_{i=1}^{i=R} w_{ji} \cdot p_i)$



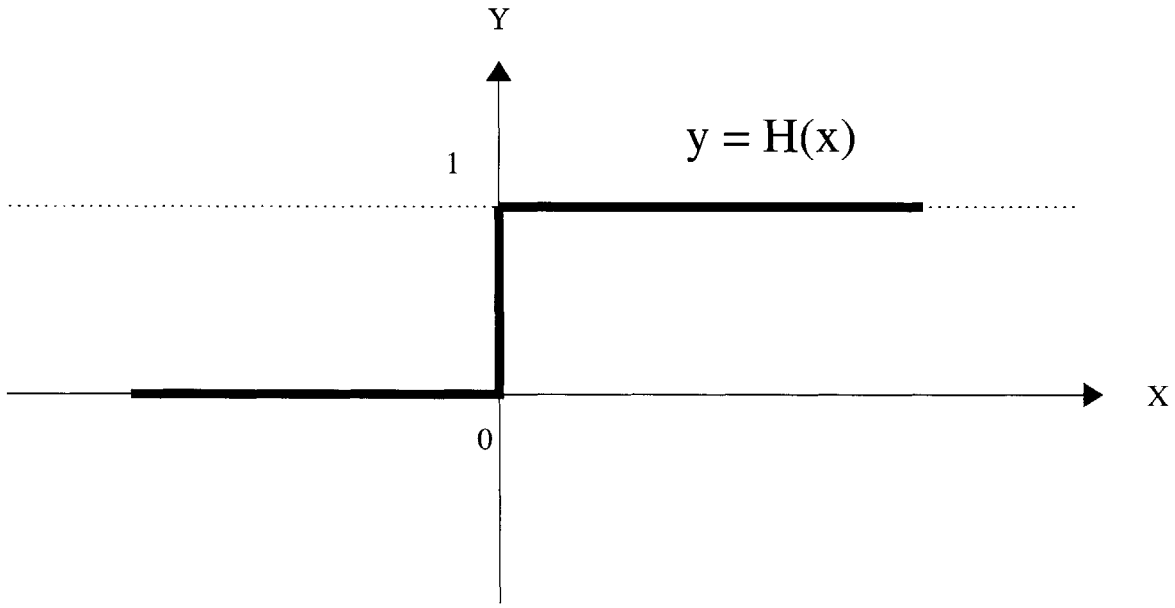
Here, $p_1, \ldots, p_R$ represent the inputs received by neuron j. The w_{ji}'s are the

synaptic strengths (more commonly called weights)[1]. They may be positive or negative so some inputs will be excitatory (positive) and some will be inhibitory (negative) As a result, the net input to the single neuron j is:

$$net\ input = n_j = \sum_{i=1}^{i=R} w_{ji} \cdot p_i$$

The output of neuron j is denoted as y_j. It depends on the thresholding function f used. As a first approximation of the behavior of a real neuron, the Heavyside function $H()$ (also called step function, see Figure 2-3) is used.

Figure 2-3: The Heavyside (or step) function



Consequently, we obtain:

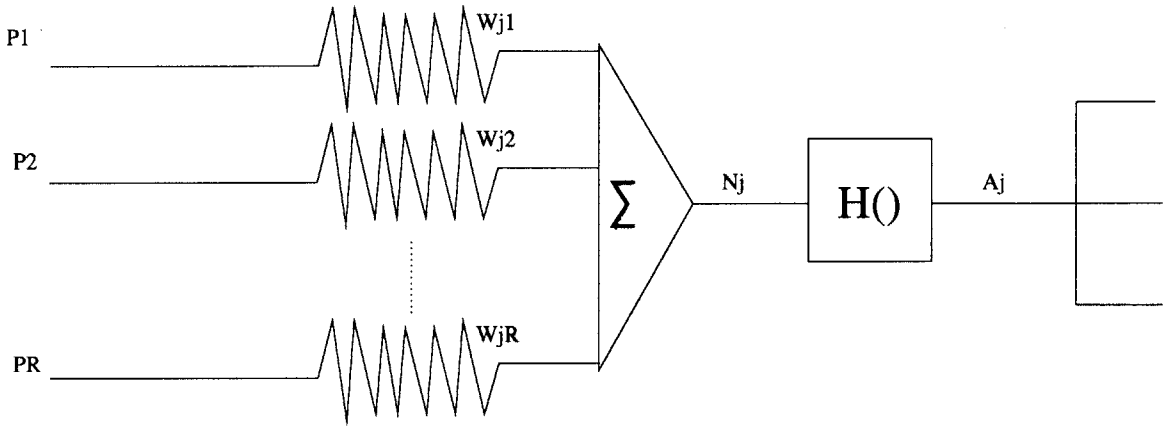
$$y_j = H(\sum_{i=1}^{i=R} w_{ji} \cdot p_i)$$

One of the advantages of such a representation is its facility of implementation. Figure 2-4 shows an electronic-circuit representation of the above mathematical abstrac-

[1]The notation w_{ji} for a synaptic weight may seem a bit strange. The j means that this weight concerns neuron j. The i means that it links input i to neuron j. As will be seen later, this notation is used to deal with more complex representations of neurons and neural networks.

tion.

Figure 2-4: An electronic representation of a neuron



Here, each p_i is a voltage and each w_{ji} is represented by a potentiometer. The triangular box, with a summation symbol inside it, is an operational amplifier configured as an adder. The quantity f is any suitable non-linear squashing function.

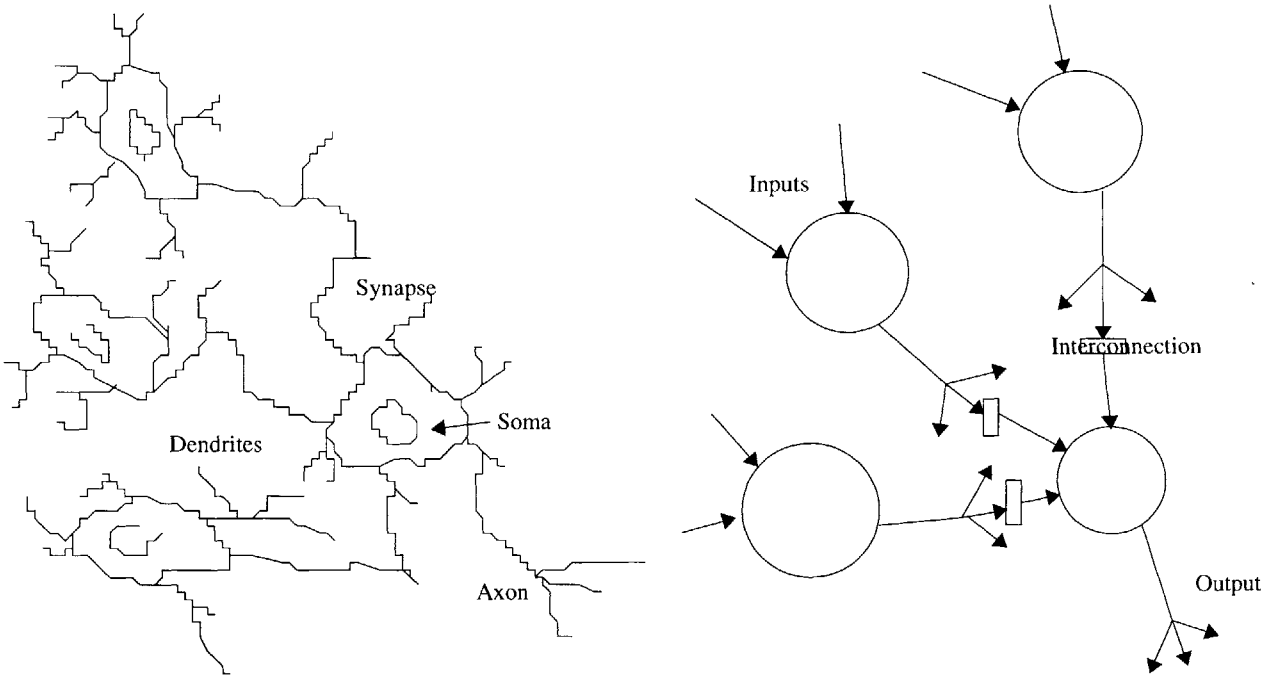
Now, if several of these artificial neurons are combined, one obtains an artificial neural network. Figure 2-5 presents one example of such a structure.

A word on semantics: when one speaks about neural networks, one should more properly specify whether these neural networks are "biological" or "artificial." In this thesis, the terms "network(s)," "neural network(s)," and "artificial neural networks" will always refer to the mathematical concept of neural network or to its electronic implementation.

2.1.3 Definition of a Neural Network

There is no universally accepted definition of an artificial neural network. However, most people in the field would agree that an artificial neural network is a network of many simple processors ("units"), each possibly having a small amount a local memory. The units are connected by communication channels ("connections"), which usually carry numeric (as opposed to symbolic) data. The units operate only on their local data and on the inputs they receive via the connections.

Figure 2-5: Biological and artificial neural networks



(a) Biological neural network

(b) Artificial neural network

Here is a sampling of definitions from the literature. None will please everyone. Perhaps for that reason many neural networks textbooks do not explicitly define neural networks.

According to the DARPA Neural Network Study [1][p60] :

> "... a neural network is a system composed of many simple processing elements operating in parallel whose function is determined by the network structure, the connection strengths, and the processing performed at computing elements or nodes."

According to Haykin [2]:

> "A neural network is a massively parallel distributed processor that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:
>
> 1. Knowledge is acquired by the network through a learning process.

2. Inter-neuron connection strengths, known as synaptic weights, are used to store the knowledge."

According to Nigrin [3][p11] :

"A neural network is a circuit composed of a very large number of simple processing elements that are neurally based. Each element operates only on local information. Furthermore, each element operates asynchronously; thus, there is no overall system clock."

According to Zurada [4][pXV] :

"Artificial neural systems, or neural networks, are physical cellular systems which can acquire, store, and utilize experiential knowledge."

2.1.4 Mathematical Representation of a Neural Network

In this section, the notation is introduced gradually, starting from a single neuron with a single input to a full complex neural network. Graphical representation is used extensively, since it is the most effective way of communicating concepts.

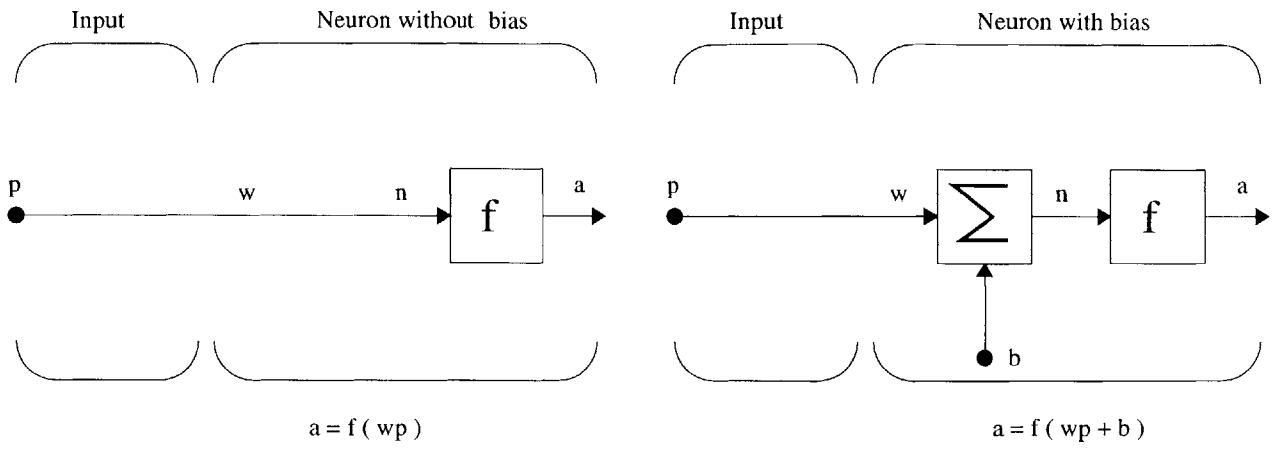
Simple Neuron

A neuron with a single scalar input is shown on the left below.

The scalar input p is transmitted through a connection that multiplies its strength by the scalar weight w, to form the product wp, again a scalar. This product is called the net input $n = wp$. Here, the net input is the only argument of the transfer function f, which produces the scalar output a.

The neuron on the right has an additional property, namely a bias b. A bias value can be treated as a connection weight from an input cell called a "bias unit" with a constant value of one. The single bias unit is connected to every unit that needs a bias value. The bias is simply being added to the product wp as shown by the summing junction.

Figure 2-6: Single neuron with scalar input



Neurons were given this bias property early in the development of neural networks. It improves their performance as described in a later section of this memoir. The bias is an example of digression from the real (biological) neural networks; this is not the only one. Even though artificial neural networks stem from biological networks, the similarity between the two has kept shrinking as scientists, in their search of improvement, have added properties to artificial neural networks.

The transfer function net input is now the sum of the product wp and the bias b: $n = wp + b$. Here f is a transfer function, not the Heavyside function any more, but typically a step function or a sigmoid function[2], that takes the argument n and produces the output a. The variety in transfer functions is another property that was introduced by researchers not from experimental observations but to improve network performances.

Note that w and b are both adjustable scalar parameters of the neuron.The central idea is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior. Thus, we can train the network to do a particular job by adjusting the weight or bias parameters, or perhaps the network itself will adjust these parameters to achieve some desired end.

[2]These two types of transfer functions, as well as others, are described in section XX.

Neuron with Vector Input

The single input neuron is rarely encountered in network architectures. Most neural networks use multiple input neurons. This set of input is represented by a vector.

Figure 2-7: Neuron with an R-element input vector

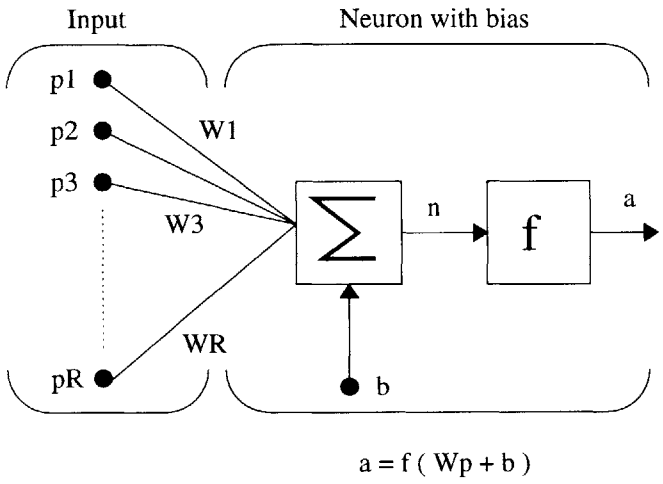


Figure 2-7 shows a neuron with an R-element input vector, i.e. a neuron with R scalar inputs. The individual element inputs $p_1, p_2, \cdots, p_R$ are multiplied by weights $w_1, w_2, \cdots, w_R$ and the weighted values are fed to the summing junction. Their sum is simply $W\vec{p}$, the dot product of the row-matrix W and the vector $\vec{p}$.

The neuron has a bias b, which is summed with the weighted inputs to form the net input n.

$$n = \sum_{i=1}^{i=R} w_i \cdot p_i + b = W\vec{p} + b$$

This sum n is the argument of the transfer function f.

This model of a neuron with vector input needs improvement:

- First of all, in a complex network, a neuron is not alone. However, we need to know which parameter (such as weight, input, or bias) belongs to which neuron. Therefore, another index is used to identify neurons. For example, applying the

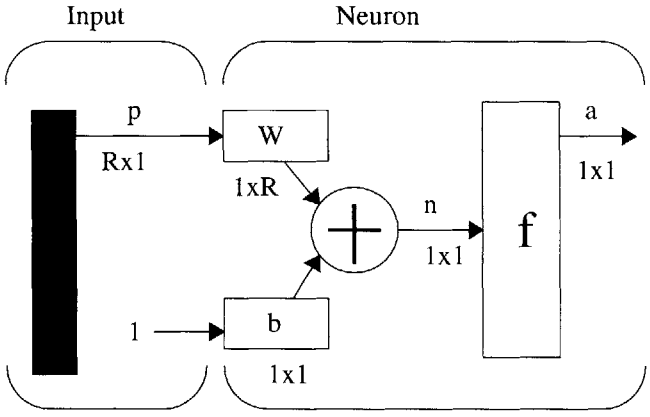
previous expression to a random neuron k would give:

$$n_k = \sum_{i=1}^{i=R} w_{ki} \cdot p_i + b_k = W_k \cdot \vec{p} + b_k$$

Here, the input vector $\vec{p}$ is not given a neuron identifying index since it does not belong to a single neuron. Indeed, several neurons can share the same input.

- Second, the figure of a single neuron shown above contains a lot of detail. When networks with many neurons are considered, there is so much detail that the main thoughts tend to be lost. Thus, an abbreviated notation is used for individual neurons. This notation is used later in circuits of multiple neurons, and is illustrated in the Figure 2-8.

Figure 2-8: Abbreviated notation for individual neurons



Here the input vector $\vec{p}$ is represented by the solid dark vertical bar on the left-hand side. The dimensions of $\vec{p}$ are shown below the symbol $\vec{p}$ in the figure as $R \times 1$. Thus, $\vec{p}$ is a vector of R input elements. These inputs post multiply the single-row R-column matrix W (or W_k if neuron k is concerned). As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias b (or b_k if neuron k is concerned). The net input to the transfer function f is the scalar (dimension 1×1) n (or n_k). This net input is passed to the transfer

function f to get the neuron's output a (or a_k), which, in this case, is a scalar (also dimension 1×1).

Each time this abbreviated network notation is used, the size of the matrices will be shown just below their name. Hopefully, this notation allows the reader to understand the architectures and follow the mathematics associated with them.

Layer of Neurons

Neural networks are complex gatherings of neurons but scientists managed to somehow order them in a visually convenient way. They chose to group them by layers.

A layer of network is defined in Figure 2-9.

A layer includes the combinations of the weights, the multiplication and summing operations (here realized as vector products $W\vec{p}$), the biases b's, and the transfer functions f's. The array of inputs, the vector $\vec{p}$, will not be included in or called a layer.

Note that it is common for the number of input elements to a layer to be different from the number of neurons (i.e. $R \neq S$). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

One-Layer Network

Figure 2-10 shows a one-layer network with R input elements and S neurons.

In this network, each element of the input vector $\vec{p}$ is connected to each neuron i through the single-row R-column matrix W_i. The net input of the ith neuron is the scalar n_i. The various n_i's taken together form an S-element net input vector $\vec{n}$. Finally, the neuron layer outputs form a vector $\vec{a}$. If we assumed that:

- all neurons in the same layer share the same transfer function f (this is very often the case in neural network theory), and
- the neuron biases form a vector called $\vec{b}$, of dimension S,

Figure 2-9: A Network Layer

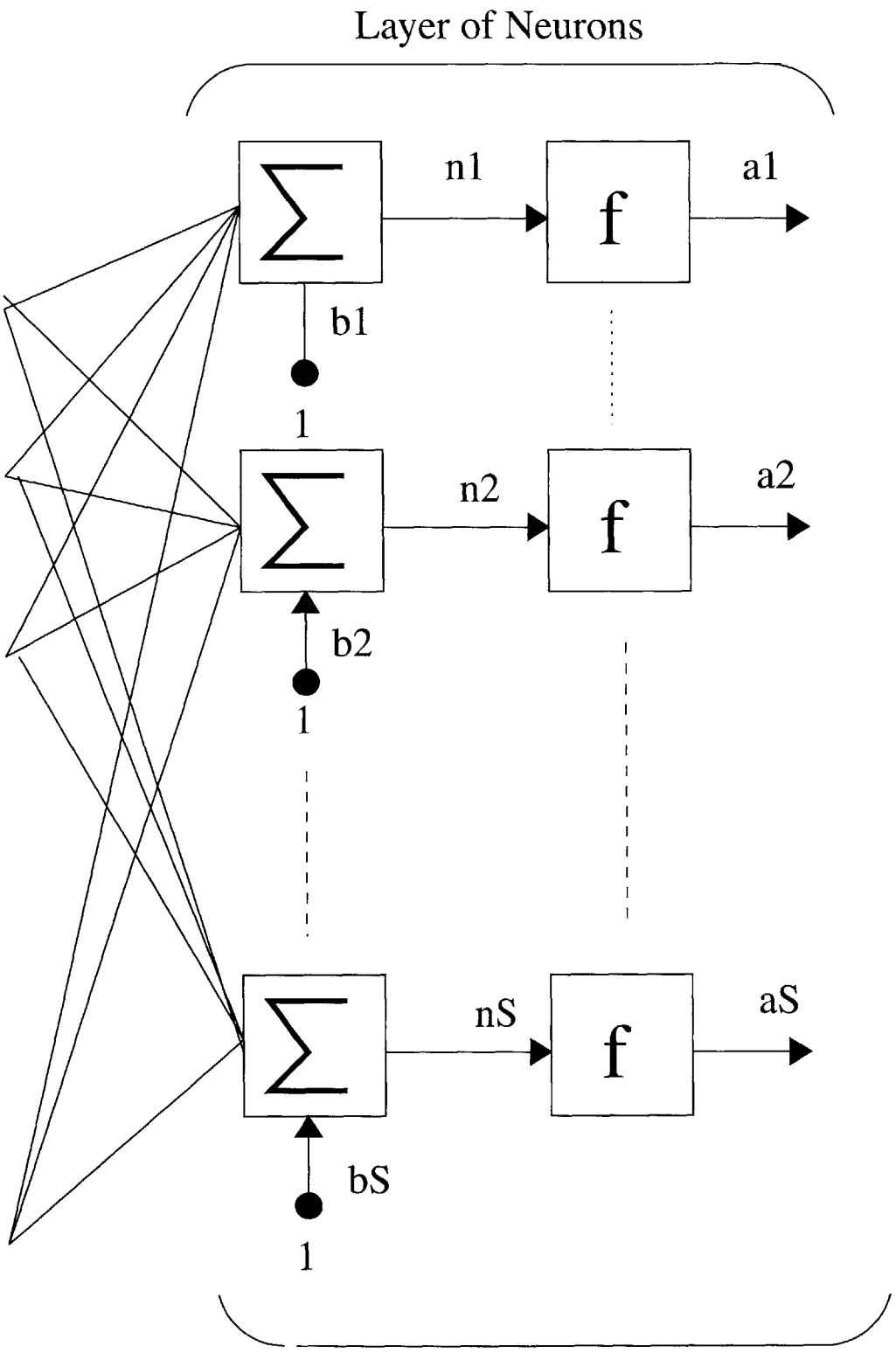
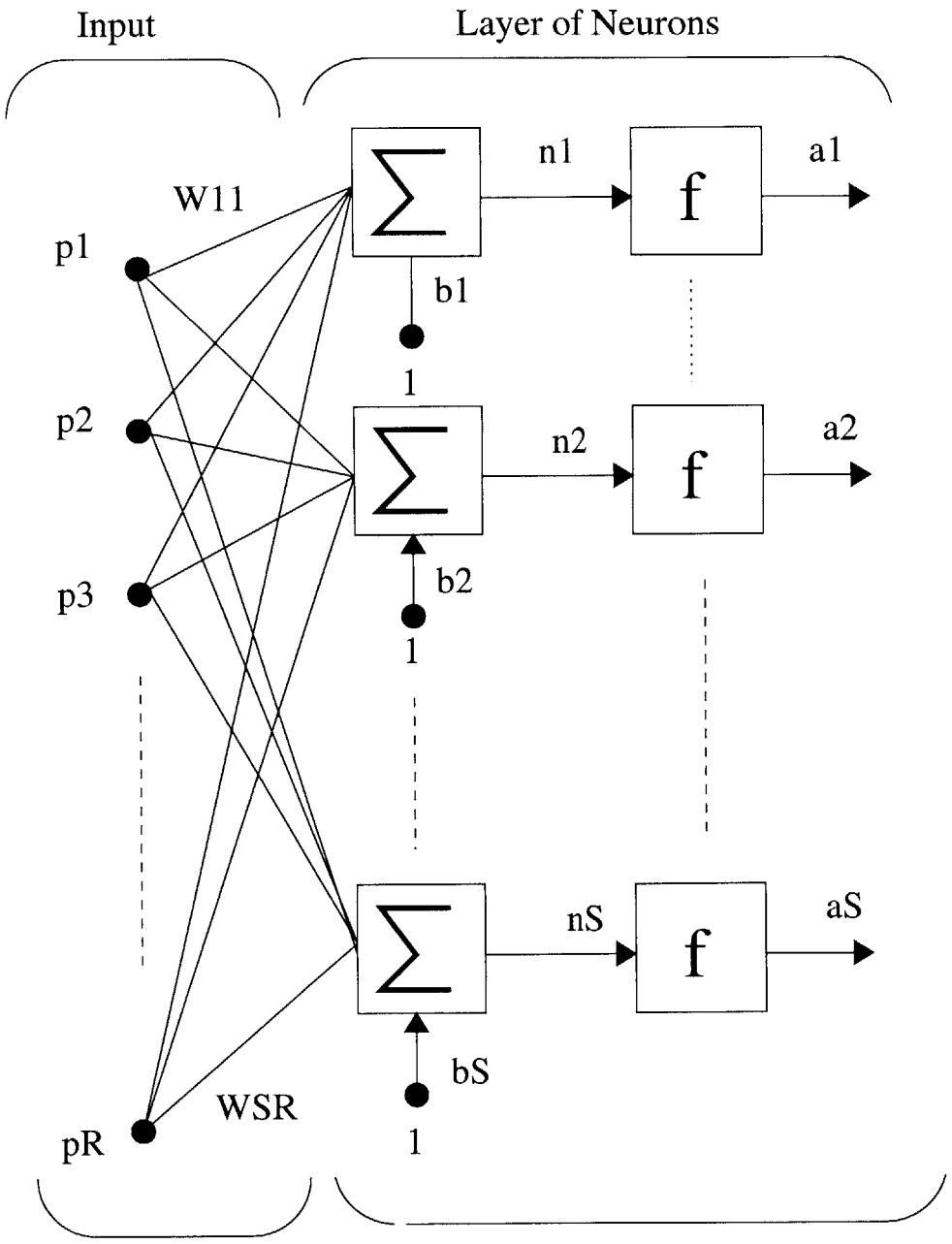


Figure 2-10: A One-Layer Network



then, the expression for $\vec{a}$ becomes:

$$\vec{a} = f(W \cdot \vec{p} + \vec{b})$$

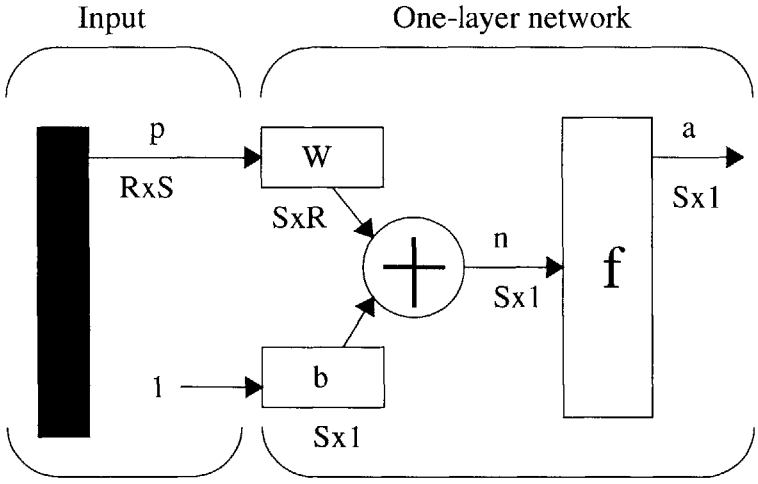
where W is the following $S \times R$ matrix:

$$W = \begin{pmatrix} W_1 \\ W_2 \\ \vdots \\ W_i \\ \vdots \\ W_S \end{pmatrix} = \begin{pmatrix} W_{11} & W_{12} & \cdots & W_{1j} & \cdots & W_{1R} \\ W_{21} & W_{22} & \cdots & W_{2j} & \cdots & W_{2R} \\ \vdots & & & & & \vdots \\ W_{i1} & W_{i2} & \cdots & W_{ij} & \cdots & W_{iR} \\ \vdots & & & & & \vdots \\ W_{S1} & W_{S2} & \cdots & W_{Sj} & \cdots & W_{SR} \end{pmatrix}$$

Note that the row indices on the elements of matrix W (this time a full matrix, not a row-matrix) indicate the destination neuron of the weight and the column indices indicate which source is the input for that weight. Thus, the indices in W_{12} say that the strength of the signal from the second input element to the first neuron is W_{12}.

The S-neuron R-input 1-layer network also can be drawn in abbreviated notation., as shown in Figure 2-11.

Figure 2-11: An S-neuron R-input 1-layer network



Here $\vec{p}$ is an R-length input vector, W is an $S \times R$-matrix, and $\vec{a}$ and $\vec{b}$ are S-

length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector $\vec{b}$, the summer, and the transfer function boxes.

Multi-Layer Network

To discuss networks having multiple layers, the notation needs to be expanded.

Specifically, a distinction must be made between weight matrices that are connected to inputs and weight matrices that connect between layers. Source and destination for each weight matrix need to be identified as well. Finally, the notation should be specific about which output vector $\vec{a}$, which bias vector $\vec{b}$, or which net input vector $\vec{n}$ belongs to which layer. Therefore, from now on:

- Weight matrices connected to inputs are called input weights (IW), and weight matrices coming from layer are called outputs layer weights (LW).

- Besides, a superscript doublet of numbers will identify the source layer (2nd number of the doublet) and the destination layer (1st number of the doublet) of each weight matrix. To come back to the one-layer network example, its weight matrix should more properly be called IW^{11} (input weight matrix from input set 1 to layer number 1).

- As far as the vectors are concerned, a superscript number will now identify which layer they belong to. For instance, the jth layer in a neural network will have bias vector $\vec{b}^j$, net input vector $\vec{n}^j$, and output vector $\vec{a}^j$.

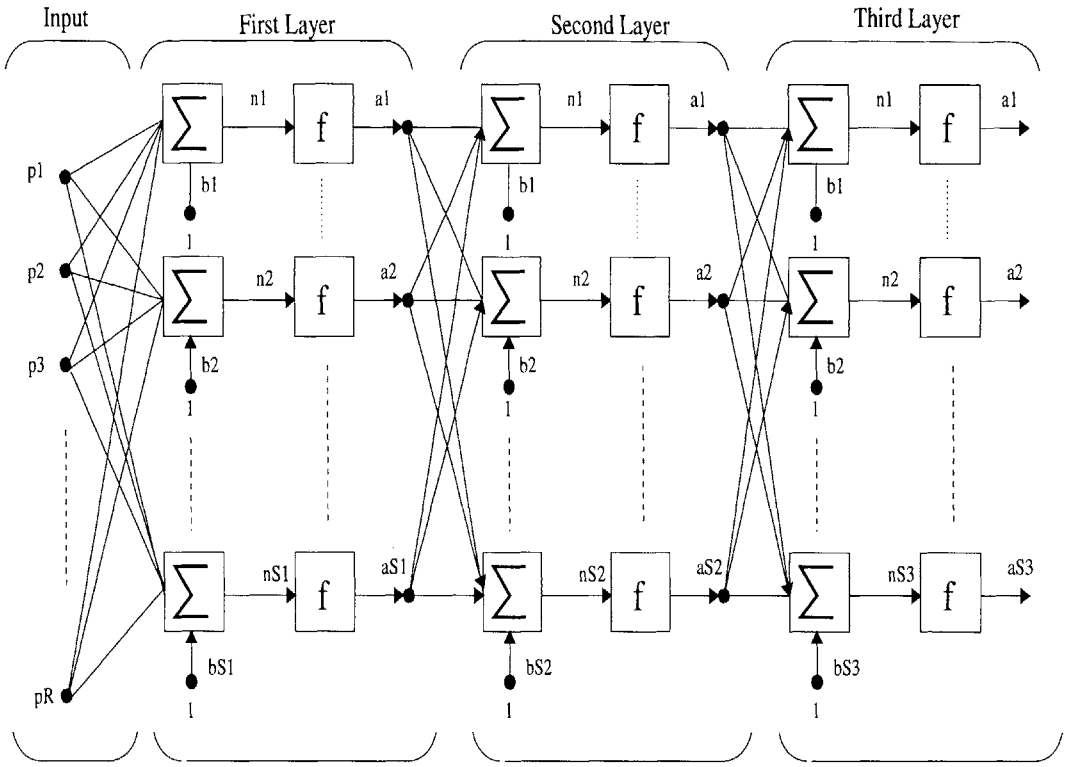
The use of this notation can be seen in the multi-layer network shown in Figure 2-12.

Note that, in a multi-layer network,

- it is common for different layers to have different numbers of neurons.

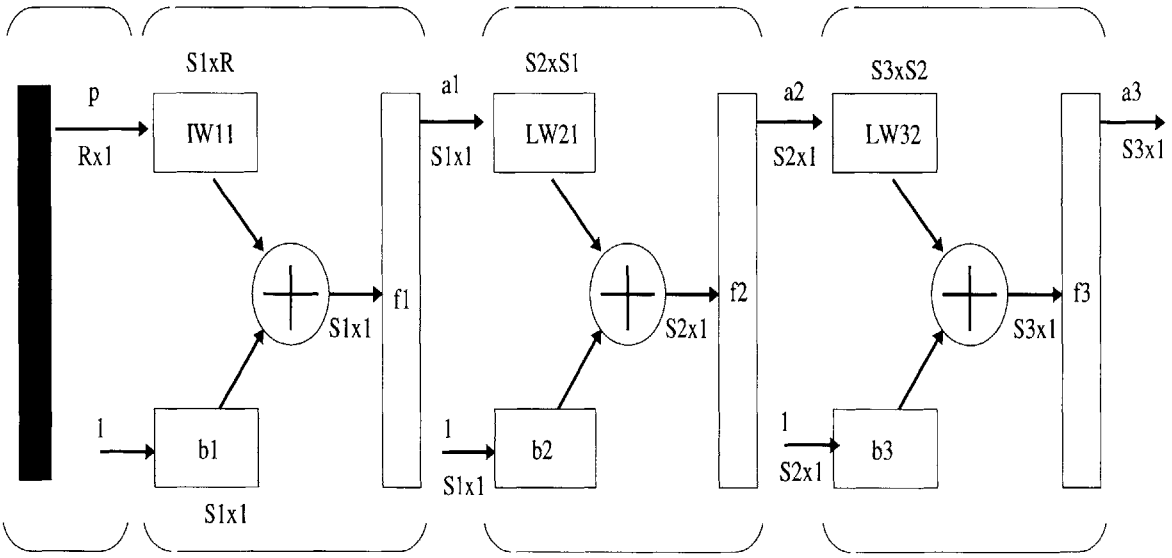
- As stated previously, neurons in the same layer often have the same transfer function. Thus, there is only one index associated with a transfer function, namely, the layer number.

Figure 2-12: An example of multi-layer network



$a1 = f1 (IW11p + b1)$ $a2 = f2 (LW21a1 + b2)$ $a3 = f3 (LW32\, a2 + b3)$

$a3 = f3 (LW32 * f2 (LW21 * f1 (IW11 * p + b1) + b2) + b3)$_



- the outputs of each layer are the inputs to the same or (more often) to another layer.

The equations below the figure highlight the relations between each layer.

How Many Layers in a Network?

The layers of a multi-layer network play different roles. A layer that produces the network output is called an output layer. All other layers are called hidden layers. The network shown in Figure 2-12 has one output layer and two hidden layers.

Now, is this network a 3- or 4-layer network? How to count layers is a matter of considerable dispute.

- Some people count layers of units. But of these people, some count the input layer and some don't.
- Some people count layers of weights. But how do they count skip-layer connections is a mystery.

To avoid ambiguity in the rest of the thesis, a 2-hidden-layer network means a network with an input layer, two hidden layers, and one output layer. Terminology such as "4-layer network" or "3-layer network" is avoided. And if the connections follow any pattern other than fully connecting each layer to its adjacent layer, this deviation will be pointed out.

2.2 Training a Neural Network

2.2.1 Neural Networks Can Learn

The property that is of primary significance for a neural network is the ability to learn from its environment, and to improve its performance through learning. A neural network learns about its environment through an interactive process of adjustments applied to its synaptic weights and bias levels. Ideally, the network becomes more knowledgeable about its environment after each iteration of the learning process.

A definition of learning has been proposed by Mendel and McClaren [5] :

> "Learning is a process by which the free parameters of a neural network are adapted through a process of simulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place."

This definition of the learning process implies the following sequence of events:

1. The neural network is stimulated by the environment.
2. The neural network undergoes changes in its free parameters as a result of this stimulation.
3. The neural network responds in a new way to the environment because of the changes that have occurred in its internal structure.

A set of well-defined rules for the update of free parameters is called a learning algorithm. As one would expect, there is no unique learning algorithm for the design of neural networks. Rather, several learning algorithms are available, each of which offers advantages of its own. Basically, learning algorithms differ from one another in the way in which the adjustment to the synaptic weights and bias levels of a neuron is formulated.

2.2.2 Learning Paradigms

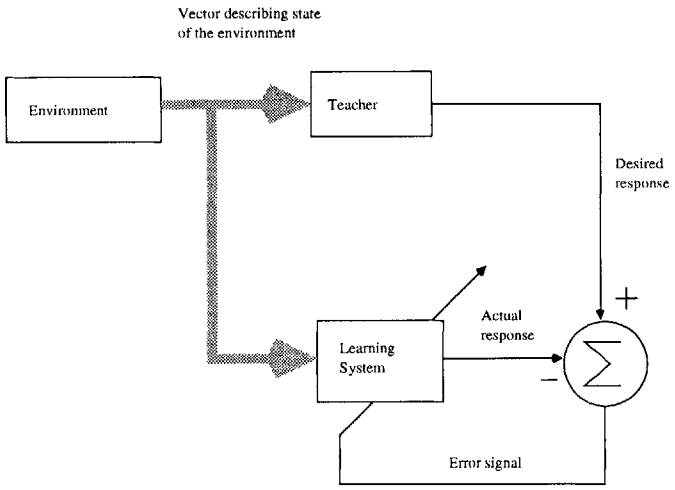
How different neural network trainings may be, they all fall into two neatly separated categories.

Learning with a Teacher

In this form of learning, an external actor is present. The actor knows the environment in which the neural network evolves, and provides the network with a desired response for each training vector. In other words, for each set of inputs presented to the network, there is a set of targets to be reached. The difference between the set of

targets (desired response) and the outputs of the network (actual response) is called error or error signal. Then, the network parameters, such as its weights and bias levels, are adjusted according to this error signal. This adjustment is carried out iteratively in a step-by-step fashion with the aim of eventually making the neural network emulate the teacher. In this way, knowledge of the environment available to the teacher is transferred to the neural network through training. One may then dispense with the teacher and let the neural network deal with the environment completely by itself. Figure 2-13 illustrates the concept of learning with a teacher for a neural network. Learning with a teacher is also called supervised learning.

Figure 2-13: Block diagram of learning with a teacher



Learning without a Teacher

In learning without a teacher, also called unsupervised learning, there is no teacher to oversee the learning process. This is to say, there are no labeled examples of the function to be learned by the network.

At this point, a question can arise:

"What does unsupervised learning learn ?"

Most of the time, unsupervised learning is used to get some information on the data that are fed to the network. Indeed, one of the properties of neural networks is to be

able to assign inputs to one of a prescribed number of classes. This property is also called "pattern recognition." Accordingly, neural networks can also perform what is called "pattern association." In this case, the neural network is required to store a set of patterns (this can be done by repeatedly presenting a set of patterns to the network). The network is subsequently presented a partial description or distorted (noisy) version of an original pattern, and the task is to retrieve that particular pattern. Applications of unsupervised learning can therefore be found in hand-writing or speech recognition.

There exists a flavor of unsupervised learning that is called competitive learning. In this paradigm, the output neurons of a neural network compete among themselves to become active (or fired). In other words, only one single output neurons is active at any one time. This feature makes competitive learning highly suited to discover statistically salient features that may be used to classify a set of input patterns. There are three basic elements to a competitive learning rule (RUMELHART and ZIPSER, 1985):

1. A set of neurons that are all the same except for some randomly distributed synaptic weights, and which therefore respond differently to a given set of input patterns.

2. A limit imposed on the "strength" of each neuron.

3. A mechanism that permits the neurons to compete for the right to respond to a given subset of inputs, such that only one output neuron is active at a time. The neuron that wins the competition is called a winner-takes-all neuron.

Accordingly, the individual neurons of the network learn to specialize on ensembles of similar patterns. In doing so, they become feature detectors for different classes of input patterns.

2.2.3 Generalization

Supervised learning teaches a network to output a desired vector each time it is fed with a corresponding input vector. In this process, the teacher already knows the two sets of input and output vectors and also knows which output vector corresponds to which input vector. Therefore, if a neural network could only do this, from the teacher's point of view, it would not be very useful. However, the teacher's hope in supervised learning is that the neural networks becomes able to generalize, i.e. to output a correct vector even if the input vector does not belong to the initial set of training data.

Generalization is not always possible. There are three conditions that are typically necessary (although not sufficient) for good generalization.

1. The first necessary condition is that the inputs to the network contain sufficient information pertaining to the target, so that there exists a mathematical function relating correct outputs to inputs with the desired degree of accuracy. One can not expect a network to learn a non-existent function. Neural networks are not clairvoyant. For example, if one wants to forecast the price of a stock, a historical record of the stock's prices is rarely sufficient input. Finding good inputs for a net and collecting enough training data often take far more time and effort than training the network.

2. The second necessary condition is that the function to be learned (that relates inputs to correct outputs) be, in some sense, smooth. In other words, a small change in the inputs should, most of the time, produce a small change in the outputs. Very non-smooth functions such as those produced by pseudo-random number generators and encryption algorithms cannot be generalized by neural nets.

3. The third necessary condition for good generalization is that the training cases be a sufficiently large and representative subset of the set of all cases that one wants to generalize to. The importance of this condition is related to

the fact that there are, loosely speaking, two different types of generalization: interpolation and extrapolation. Interpolation applies to cases that are more or less surrounded by nearby training cases; everything else is extrapolation. Interpolation can often be done reliably, whereas extrapolation is notoriously unreliable. Hence, it is important to have sufficient training data to avoid the need for extrapolation. Methods for selecting good training sets are discussed in numerous statistical textbooks on sample surveys and experimental design.

2.3 Taxonomy of Network Architectures

Because of the huge number of parameters that a single neural network can have, and also because of the various possibilities of connections that exist to link neurons, there are now many types of neural networks. Nobody knows exactly how many. New ones (or at least variations of existing ones) are invented every week. Below is a collection of some of the most well known networks, not claiming to be complete.

For one new to the field of neural networks, this variety of network types gives the impression that a new type of neural network is created each time somebody builds a neural network. Besides, in this list of neural network types, it is hard to find the one that will do what is desired. The purpose of the following classification is to clarify the field.

2.3.1 An Example of Classification

First of all, as seen in section 2.2.2, neural networks can be classified according to the learning scheme they require, either supervised or unsupervised. Another distinctive property is the presence (or absence) of feedback loops in a neural network. Feedback is said to exist in a dynamic system whenever the output of an element in the system influences in part the input applied to that particular element, thereby giving rise to one or more closed paths for the transmission of signals around the system.

- If a neural network has at least one single feedback loop, it is called a recurrent

network. For example, a recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons, as illustrated in the architectural graph in Figure 2-19.

- On the contrary, a neural network without any feedback loop is called a feed-forward network. Usually, in such a network, environmental parameters are fed to the source nodes in the input layer, which become activated. Their output signals constitute the input signals to the neurons in the second layer (i.e. the first hidden layer). The output signals of the second layer are then used as inputs to the third layer, and so on for the rest of the network. The architectural graph in Figure 2-14 illustrates the layout of a multi-layer feed-forward network for the case of a single hidden layer.

  Besides, the neural network in this same figure is said to be fully connected in the sense that every node in each layer of the network is connected to every other node in the adjacent forward layer. If, however, some of the communication links are missing from the network, the network is said to be partially connected.

Finally, for each combination of learning paradigm and network architecture, there is a variety of learning algorithms that further classify neural networks into categories. Table 2.1 contains a listing of the major categories of neural networks. Network types written in bold face in this table are described in more detail in the following section of the thesis.

2.4 Some Well-known Networks

2.4.1 Perceptron

The single-layer perceptron is the simplest form of a neural network used for the classification of patterns said to be linearly separable, i.e. patterns that lie on the opposite sides of a hyper-plane. Basically, it consists of a single neuron with adjustable synaptic weights and bias, as presented in Figure 2-15. Note that the transfer function

Figure 2-14: Fully connected feed-forward network

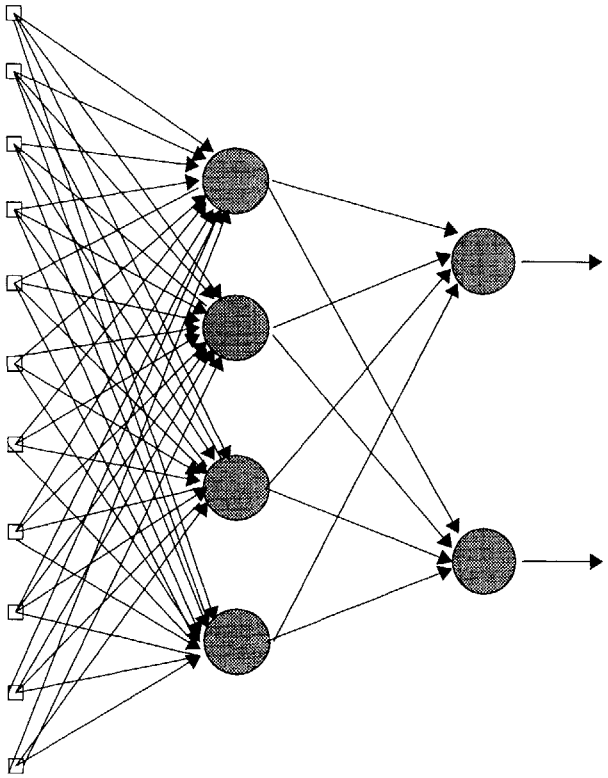
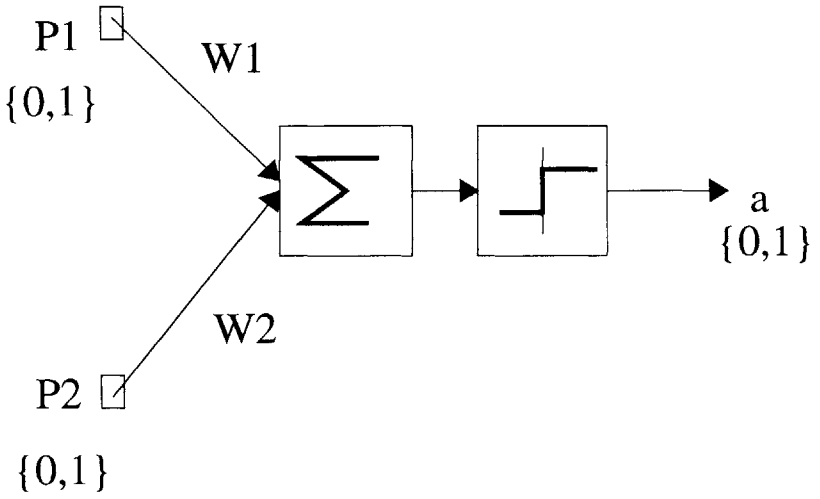


Table 2.1: Classification of neural networks

		Architecture	
		Recurrent	Feed-forward
Learning Scheme	Supervised	Boltzman Machine (clamped conditions)	**Single-layer Perceptron** **Multi-Layer Perceptron** **Radial-Basis Function** Adaline and Madaline Support Vector Machine Committee Machine
	Unsupervised (Competitive)		Principal Component Analysis Sigmoid Belief Network Cognitron and Neocognitron **(Self-Organizing Map)** **(Vector Quantization)** **(Learning Vector Quantization)**

of a perceptron is the Heavyside function $H()$, a graph of which was presented in Figure 2-3.

Figure 2-15: The original perceptron



The algorithm used to adjust the free parameters of this neural network first appeared in a learning procedure developed by Rosenblatt (1958,1962) for his perceptron brain model. Indeed, Rosenblatt proved that if the patterns used to train the perceptron are drawn from two linearly separable classes, then the perceptron algorithm converges and positions the decision surface in the form of a hyper-plane between the two classes. The proof of convergence of the algorithm is known as the perceptron convergence theorem. The perceptron built around a single neuron is limited to performing pattern classification with only two classes.

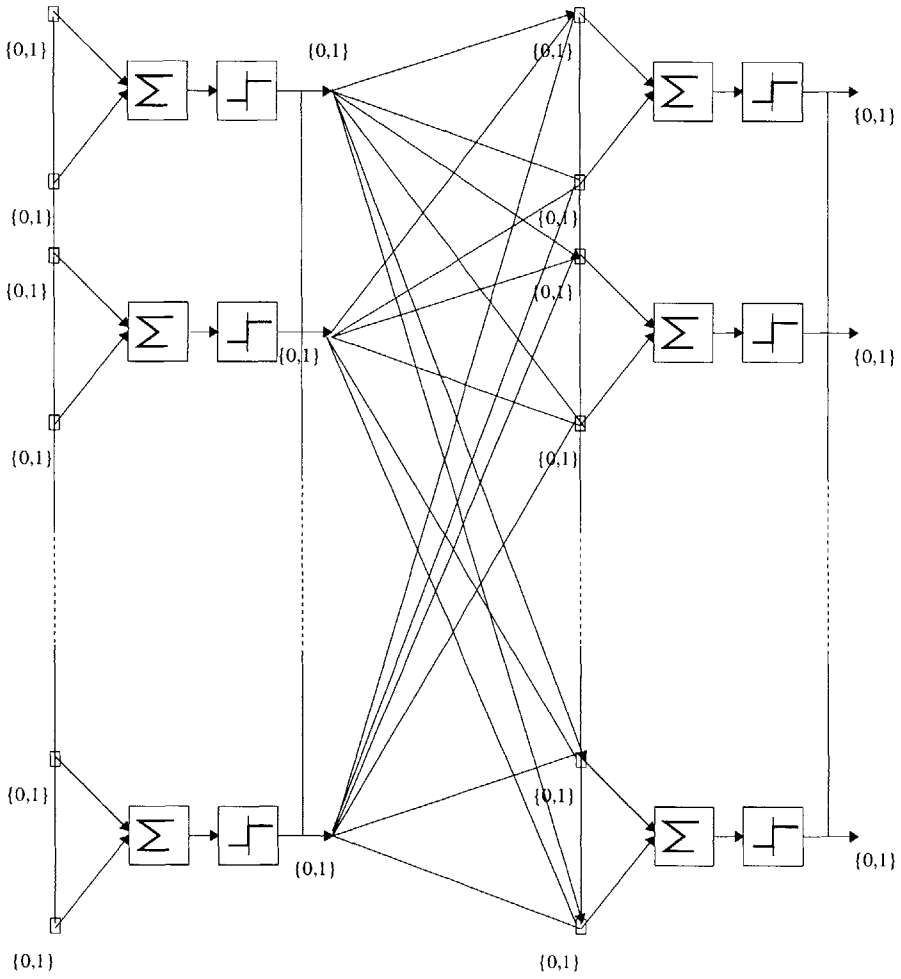
Single-Layer

By extending the output layer of the perceptron to include more than one neuron, one may correspondingly form classification with more than two classes. However, the classes have to be linearly separable for the perceptron to work properly.

Multi-Layer

The Multi-Layer Perceptron or MLP is a further generalization of the perceptron, this time to the field of multi-layer feed-forward networks. One example of MLP is provided in Figure 2-16.

Figure 2-16: A multi-layer perceptron



Although neurons in the first studied multi-layer perceptrons still had the Heaviside function as transfer function, this limitation was soon overcome by the introduction of non-linear transfer functions.

Today, it is agreed in the neurocomputing society that a multi-layer perceptron has three distinctive characteristics:

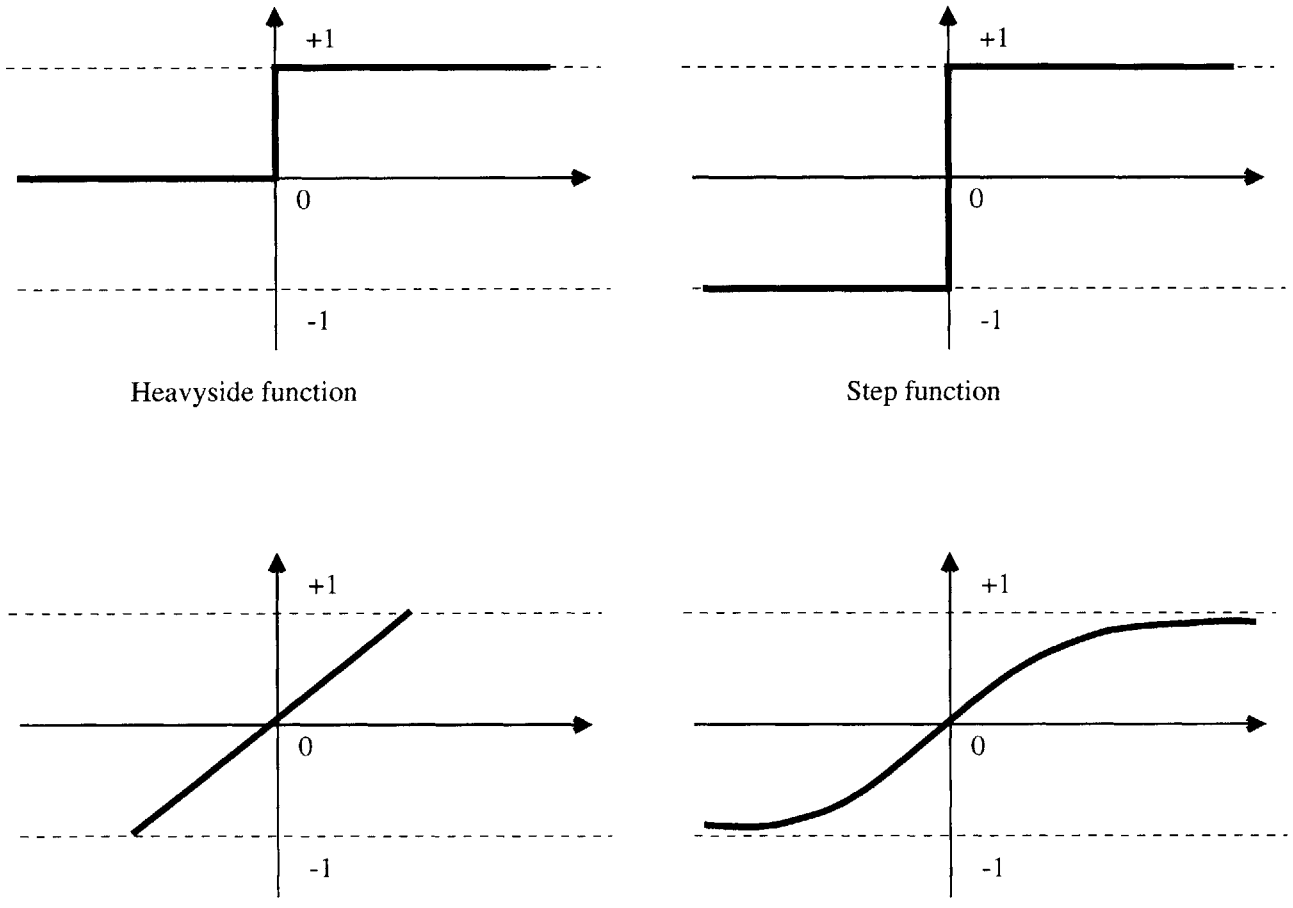
1. The model of each neuron in the network includes a nonlinear activation

function. The important point to emphasize here is that the non-linearity is smooth, i.e. differentiable everywhere, as opposed to the step-function used in the initial perceptron. A commonly used form of non-linearity that satisfies this requirement is the sigmoidal non-linearity, an example of which is given by the logistic function:

$$a_i = \frac{1}{1 + \exp(-n_i)}$$

where n_i is the net input of neuron i, and a_i is its output. Figure 2-17 present other commonly used transfer functions.

Figure 2-17: Examples of transfer functions



2. The network contains one or more layers of hidden neurons. These hidden

neurons enable the network to learn complex task by extracting progressively more meaningful features from the input patterns.

3. The network exhibits a high degree of connectivity, determined by the synapses of the network.

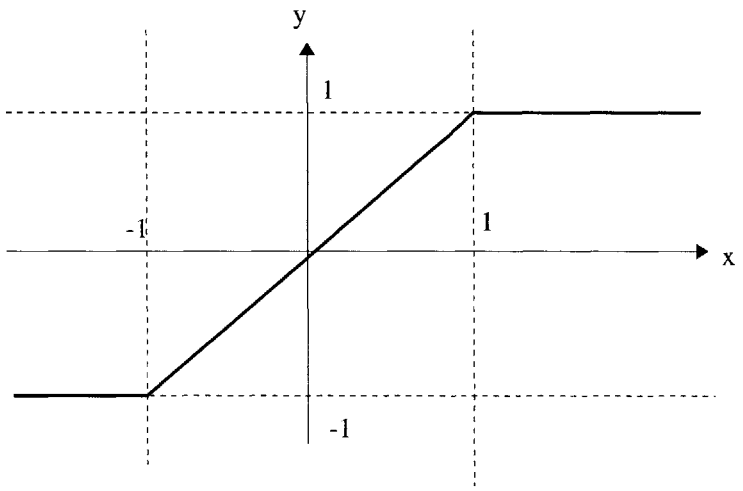
It is through the combination of these characteristics together with the ability to learn through training that the multi-layer perceptron derives its computing power.

These same characteristics, however, are also responsible for the deficiencies of our present state of knowledge on the behavior of the network. First, the presence of a distributed form of non-linearity and the high connectivity of the network make the theoretical analysis of a multi-layer perceptron difficult to undertake. Second, the use of hidden neurons makes the learning process harder to visualize.

2.4.2 Adaptive Linear Filters

The Adaptive Linear Neuron Network (or ADALINE) is similar to a single-layer perceptron, except that the transfer functions of its neurons are linear rather than hard-limiting. An example of such a function is given in Figure 2-18.

Figure 2-18: Linear transfer function used in ADALINE



This allows its outputs to take on any value, whereas the single-layer perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can

only solve linearly separable problems. However, Adaptive Linear Neuron Networks make use of the so-called Least Mean Squares (or LMS) learning rule to update their synaptic weights and bias levels. The LMS learning rule was invented by Widrow and Hoff, and is much more powerful than the perceptron learning rule.

Linear neurons can also be gathered in a multi-layer network. In this case, such a network is called a MADALINE (Multi Adaptive Linear Neuron Network), and also uses the LMS learning rule to set its weights and bias levels.

Note the evolution in neural networks. Scientists started with simple transfer functions, such as the Heavyside function or step functions, and discovered the perceptron rule to update the network parameters. This was the perceptron era. Then, they upgraded to linear transfer functions, and this started the ADALINE and MADALINE era, and introduced the LMS algorithm . As they subsequently chose sigmoid functions for their transfer functions, they needed a better algorithm, which happened to be the back-propagation algorithm. This started the back-propagation era.

2.4.3 Back-Propagation Network

Multi-layer perceptrons have been applied successfully to solve some difficult and diverse problems by training them in a supervised way with a highly popular algorithm known as the error back-propagation algorithm or, in short, back-propagation. Back-Propagation Networks are only Multi-Layer Perceptron trained with this algorithm. See section 3.1.2 for a detailed description of the back-propagation algorithm.

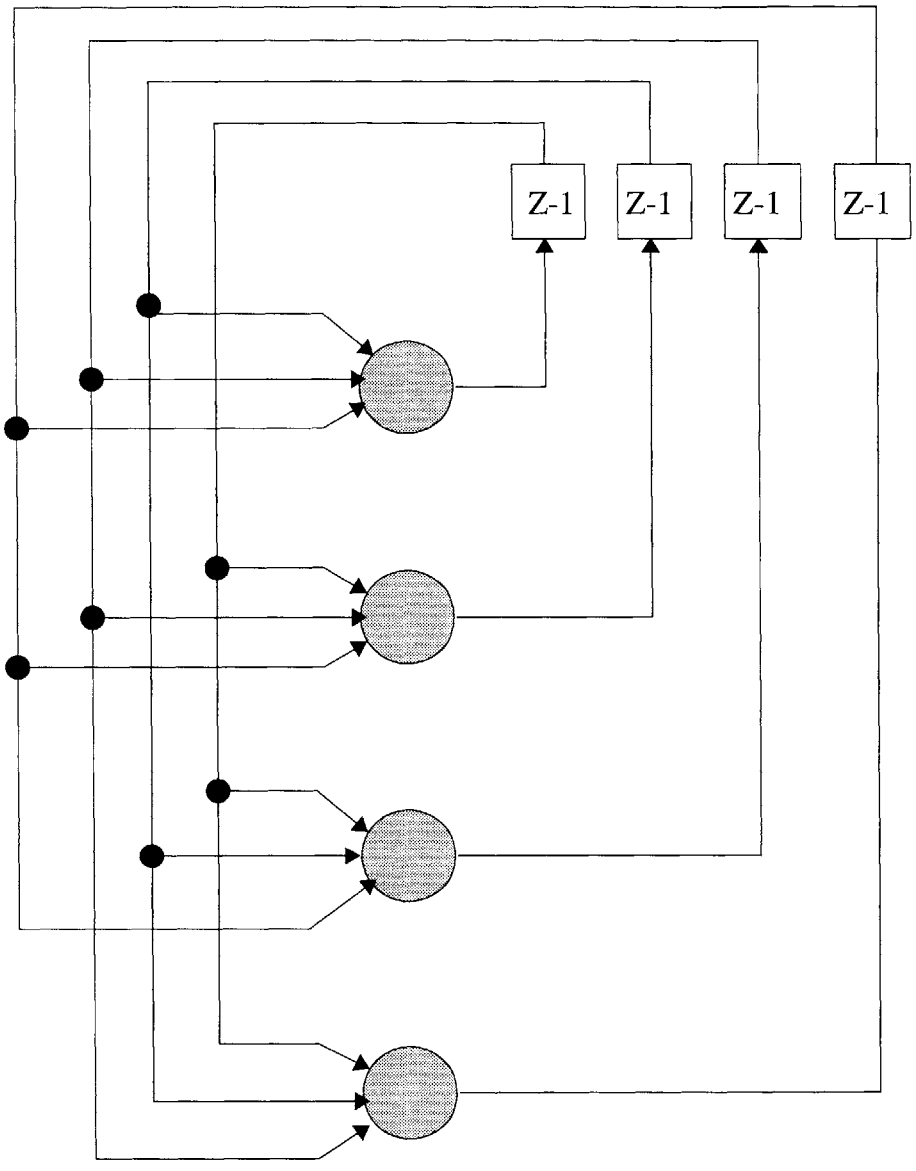
2.4.4 Hopfield Network

All networks considered until now assumed only forward flow from input to output, namely non-recurrent connections. The theory proves that this guarantees network stability. Since biological neural networks incorporate feedback, (i.e. they are recurrent), it is natural that certain artificial networks will also incorporate that feature. The Hopfield neural networks do indeed employ both feed-forward and feedback. Once feedback is used, stability cannot be guaranteed in the general case. Consequently,

the Hopfield network must be one that accounts for stability in its settings.

Figure 2-19 illustrates a single-layer Hopfield network. In this figure, every z^{-1}

Figure 2-19: An example of Hopfield network



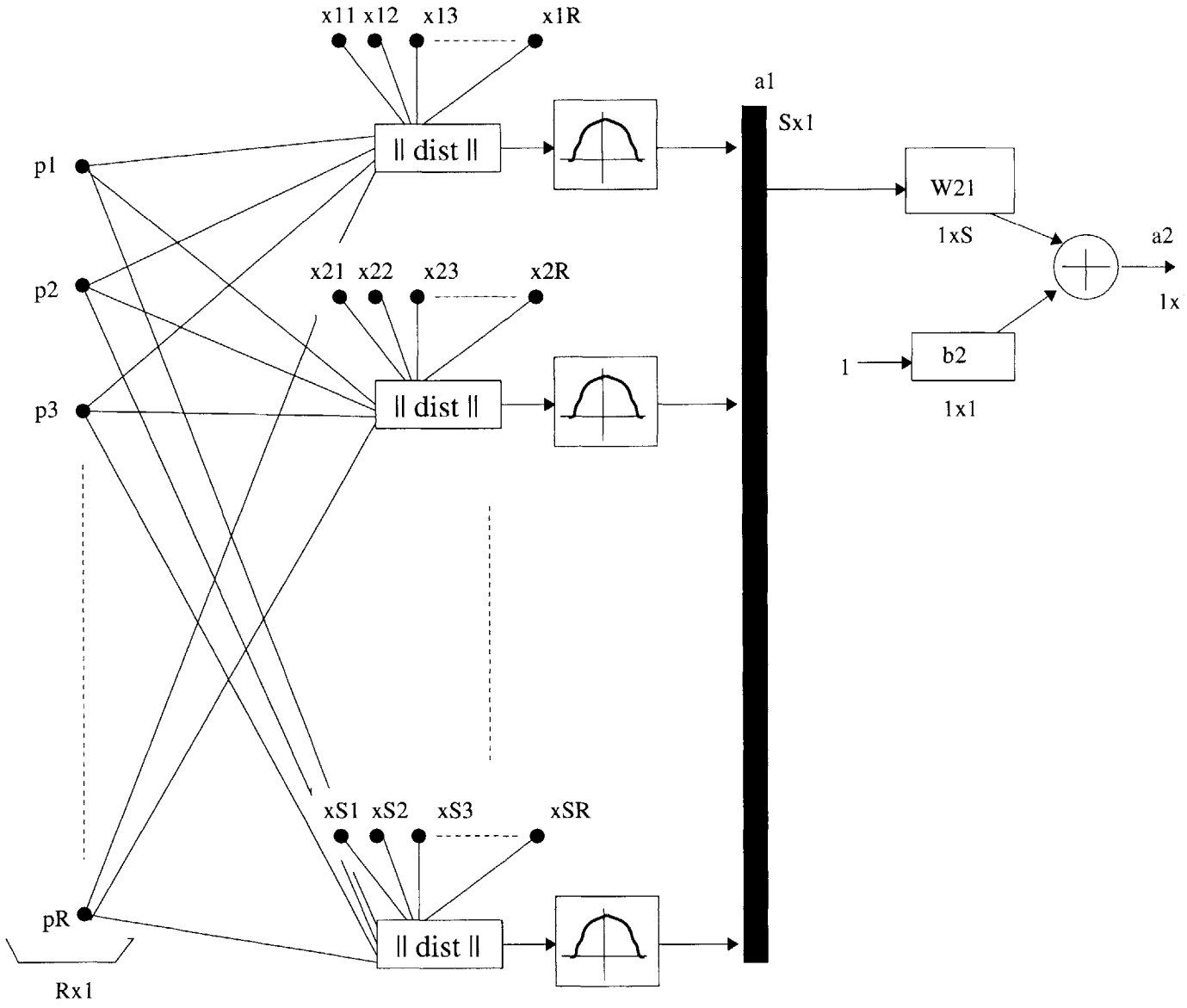
represents a unit-delay operator.

2.4.5 Radial-Basis Function Network

The construction of a radial-basis function (RBF) network, in its most basic form, involves three layers with entirely different roles. The input layer is made up of source

nodes (sensory units) that connect the network to its environment. The second layer, the only hidden layer in the network, contains specific neurons called RBF neurons. The output layer is linear, supplying the response of the network. Figure 2-20 presents an example of RBF network:

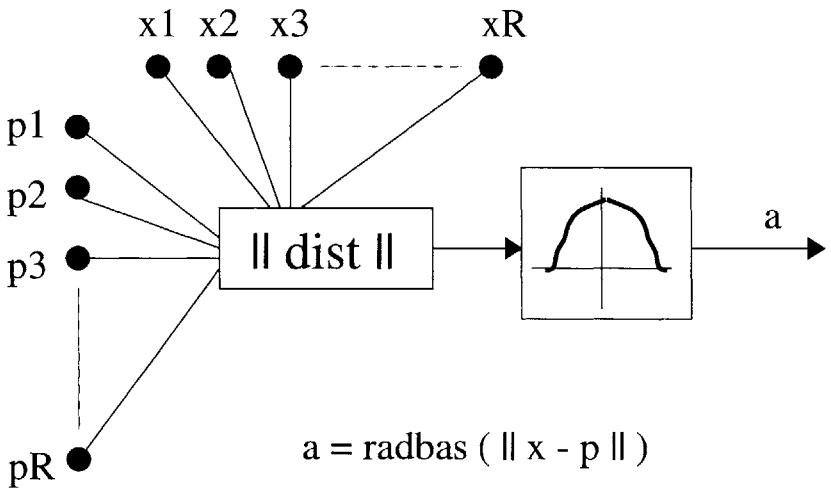
Figure 2-20: An example of RBF network



Note that each neuron is the hidden layer is connected to all source nodes in the input layer. Besides, the output layer consists of a single linear unit (i.e. a neuron with a linear transfer function), being fully connected to the hidden layer.

All the properties of an RBF network come from the specific structure of the neurons in its hidden layer. The model of such a neuron is presented in Figure 2-21.

Figure 2-21: An RBF neuron



Such a neuron adopts a different point of view on its input. For instance, in a perceptron, the input is considered as a set of scalar elements, whose weighted sum is of importance.

On the other hand, for an RBF neuron, the input is considered as a vector, and the scalar elements are the coordinates of this vector in an input space. Each RBF neuron stores a specific vector in its memory. Then it computes the distance between the input vector and the stored vector, using a so-called radial basis function. Usually, a radial basis function is close to a Gaussian function, i.e. close to the following form:

$$a_i = \exp(- \parallel \vec{p} - \vec{x_i} \parallel^2)$$

where a_i is the output of the RBF neuron, $\vec{p}$ the input vector, and $\vec{x_i}$ the stored vector of this neuron i. $\| \cdot \|$ can represent any norm, but usually:

$$\parallel \vec{p} - \vec{x_i} \parallel^2 = \sum_{j=1}^{j=R} (p_j - x_{ij})^2$$

where:

$$\vec{p} = \begin{pmatrix} p_1 \\ \vdots \\ p_j \\ \vdots \\ p_R \end{pmatrix} \quad \vec{x_i} = \begin{pmatrix} x_{i1} \\ \vdots \\ x_{ij} \\ \vdots \\ x_{iR} \end{pmatrix}$$

The set of distances so calculated represents the coordinates of the input vector in the hidden space.

For instance, imagine an RBF network with 10 source nodes in the input layer and only 2 RBF neurons. The dimension of the input space is therefore 10, whereas the dimension of the hidden space is 2. Each input vector presented to the network is associated with two distances that are the coordinates of this input vector in the hidden space. The hidden layer is said to perform a mapping from the input space to the hidden space. In this example, if one draws a graph, with the first distance as the x-coordinate and the second distance as the y-coordinate, one can plot the position of an input vector in the hidden space.

Each RBF neuron then transmits the calculated distance to the single linear neuron, whose role is to output a weighted sum of all the distances. In its turn, the output layer does a mapping from the hidden space to the output space, which happens to be of size 1 (space of real numbers). In our example, every input vector can therefore be associated with a single output number. If one draw a line representing the output space, one can now plot the position of each input vector.

This process can be generalized. An RBF network is designed to perform a non-linear mapping from a m-dimension input space to an hidden space, followed by a linear mapping from the hidden space to the 1-dimension output space. The network can therefore be written as:

$$map : \Re^m \longrightarrow \Re^1$$

This map is represented by a hyper-surface Γ drawn in $\Re^{m+1}$, just as, for instance, the map $s : \Re^1 \rightarrow \Re^1$ where $s(x) = x^2$ is represented by a parabola drawn in $\Re^2$. The

surface Γ is a multi-dimensional plot of the output as a function of the input. In a practical situation, the surface Γ is unknown.

Thus, we are led to the theory of multi-variable interpolation in high-dimensional space. The interpolation problem may be stated:

> *Given a set of N different points $\vec{x_i} \in \Re^m \mid i = 1, 2, \ldots, N$ and a corresponding set of N of real numbers $d_i \in \Re^1 \mid i = 1, 2, \ldots, N$, find a function $F{:}\Re^m \rightarrow \Re^1$ that satisfies the condition:*
>
> $$F(\vec{x_i}) = d_i, i = 1, 2, \ldots, N$$

The radial basis function technique consists of choosing a function F that has the following form:

$$F(\vec{x}) = \sum_{i=1}^{i=N} w_i \varphi_i(\| \vec{x} - \vec{x_i} \|)$$

where the φ_i are arbitrary non-linear functions. The known data points $\vec{x_i} \in \Re^m, i = 1, 2, \ldots, N$ are said to be the centers of the radial basis functions.

Given the previous theoretical analysis, it should be clear that RBF networks are used in interpolation and function approximation. The challenge in RBF networks is to use as few RBF neurons as possible. Indeed, if your training contains a lot of data, i.e. lost of input vectors, it would be prohibitive to build an RBF network with as many hidden neurons as there are input vectors in the training set. An RBF network with fewer hidden neurons needs to be built but it should still be able to approximate well when fed with an unknown vector.

As a conclusion, note that the adjustable parameters of the network are:

1. the synaptic weights of the connections between the neurons in the hidden layer and the output neuron.

2. the parameters of the radial basis function in each neuron of the hidden layer.

2.4.6 Kohonen Networks

Teuvo Kohonen is one of the most prolific researchers in neurocomputing, and he has invented a variety of networks. However, many people refer to "Kohonen networks" without specifying which kind of Kohonen network, and this lack of precision can lead to confusion. The phrase "Kohonen network" most often refers to one of the following three types of networks:

1. Vector Quantization (VQ), competitive networks closely related to cluster analysis (the exercise of finding "clusters" of data in the overall set of input vectors). In a VQ, each competitive unit corresponds to a cluster, the center of which is called a "codebook vector." Kohonen's learning law is an algorithm that finds the codebook vector closest to each training case and moves the "winning" codebook vector closer to the training case.

2. Self-Organizing Map (SOM), competitive networks that provide a "topological" mapping from the input space to the clusters. The SOM was inspired by the way in which various human sensory impressions are neurologically mapped into the brain such that spatial or other relations among stimuli correspond to spatial relations among the neurons. In a SOM, the neurons are organized into a grid, usually two-dimensional. The grid exists in a space that is separate from the input space. A SOM tries to find clusters such that any two clusters that are close to each other in the grid space have codebook vectorclose to each other in the input space.

3. Learning Vector Quantization (LVQ), competitive networks for supervised classification. Each codebook vector is assigned to one of the target classes. Each class may have one or more codebook vectors. A case is classified by finding the nearest codebook vector and assigning the case to the class corresponding to this nearest.

2.5 Neural Networks in Engineering

2.5.1 A Brief History of Neural Network Models

In 1943, W. McCulloch and W, Pitts proposed a theory of information processing based on networks of binary switching elements, which were, somewhat euphemistically, called "neurons," although they were far simpler than their real biological counterparts. Each one of these elements could only take the output value 0 or 1, where 0 represented the resting state and 1 the active state of elementary unit. (they just chose a step function normalized to $[0, 1]$ as their thresholding function). McCulloch and Pitts showed that such networks could, in principle, carry out any imaginable computation, similar to a programmable digital computer. The designers of McCulloch-Pitts-type neural networks now faced the problem of how to choose the weights w_{ji} so that a specific cognitive task was performed by the machine. This question was addressed in 1961 by E. Casaniello, who gave a "learning" algorithm that would allow the determination of the synaptic strengths of a neural network.

Around 1960, F. Rosenblatt and his collaborators extensively studied a specific type of neural network, which they called a "perceptron," because they considered it to be a simplified model of the biological mechanisms of processing sensory information, i.e. perception. In its simplest form, a perceptron consists of two separate layers of neurons, representing the input and output layer, respectively, as illustrate in Figure 2.4.1.

The neurons of the output receive synaptic signals from those of the input layer, but not vice versa, and the neurons within one layer do not communicate with each other. The flow of information is thus strictly directional; hence a perceptron is a feed-forward network. Rosenblatt's group introduced an iterative algorithm for constructing the synoptic weights w_{ji} such that a specific input pattern is transformed into the desired output pattern, and even succeeded in proving its convergence.

However, M.Minsky and S. Papert pointed out a few years later that this proof applies only to those problems which can, in principle, be solved by a perceptron. What made matters worse was that they showed the existence of very simple problems

which cannot be solved by any such two-layered perceptron! The most notorious of these is the "exclusive-OR" (XOR) gate, which requires two input neurons to be connected with a single output neuron in such a way that the output unit is activated if, and only if, one of the input unit is active. The logic XOR gate being a standard problem easily solved in computer design, the result of Minsky and Papert represented a severe blow to the perceptron concept.

Another very fruitful development began when W. Little pointed out the similarity between a neural network of the type proposed by McCulloch and Pitts and systems of elementary magnetic moments (or spins) in solid state physics. Indeed, a lattice of atoms with alternate spins strongly resembles a two-dimension network of switching neurons. The development of this analogy, at first pursued by Little and G. Shaw, but then also by J. Hopfield, led to the introduction of physical concepts in the study of neural networks, such as energetic and thermodynamic properties. Neural networks also acquired stochastic laws of evolution, involving probabilistic functions.

In recent years, the interest in layered, feed-forward networks has been revived. This development was initiated by the discovery of an efficient algorithm for the determination of the synaptic weights in multi-layered networks with hidden layers. The power of this method, initially suggested by Werbos and now known as error back-propagation, was recognized around 1985 by several groups of scientists.

2.5.2 Why Neural Networks Appeal?

Everyday observation shows that the modest brains of lower animals can perform tasks that are far beyond the range of even the largest and fastest modern electric computers. Just imagine that any mosquito can fly around at great speed in unknown territory without bumping into objects blocking its path. No present-day electronic computer has sufficient computational power to match this and similar accomplishments. They typically involve some need for the recognition of complex optical or acoustical patterns, which are not determined by simple logic rules.

Great efforts were made in the past two decades to solve such problems on traditional computers. One product of these efforts was the emergence of the techniques

of artificial intelligence (AI). While the products of AI, which are more appropriately described as expert systems, had a number of impressive successes, e.g. in medical diagnosis, they are much too slow to perform the analysis of optical or speech patterns at the required high rate. Moreover, the concept of AI is based on formal logical reasoning, and can thus only be applied when the logical structure of a certain problem has been analyzed.

Another source of dissatisfaction is that the speed of solid-state switching elements, the basic units of electronic computers, has reached a point where future progress seems to be limited. In order to accelerate computational tasks further, one has therefore turned to the concept of parallel processing, where several operations are performed at the same time. Three problems are encountered here:

1. The basic unit of a traditional computer, the central processing unit (CPU), is already a very complex system containing hundreds of thousands of electronic elements that cannot be made arbitrarily cheap. Hence, there is a cost limit to the number of CPUs that can be integrated into a single computer.

2. Most problems cannot be easily subdivided into a very large number of logically independent subtasks that can be executed in parallel.

3. The combination of a large number of CPUs into a single computer poses tremendous problems of architectural design and programming which are not easily solved if one wants to have a general-purpose computer and not a machine dedicated to one specific task.

These two obstacles (inadequacies of AI and speed limitation) have led to a resurgent interest in neural networks since 1980. One now considers neural networks as prototype realizations of the concept of parallel distributed processing. Indeed, in contrast to the great complexity of the CPU of a traditional electronic computer, the neurons of a neural network are relatively simple electronic devices, which contain only a switching element and adjustable input channels. Even with presentably available technology, it is not unimaginable to arrange tens of thousands of binary decision

elements on a single silicon chip. Many thousands of such identical chips might be integrated into a single neural computer at a reasonable cost. Thus neural computers with a billion or more neuron-like elements appear technically feasible, and a system approaching the complexity of the human brain (about 10^{11} neurons) does not belong to the realm of science fiction.

Finally, a potentially important advantage of neural networks is their high degree of error resistivity. A normal computer may completely fail in its operation if only a single bit stored information or a single program statement is incorrect. In contrast, the operation of a neural network often remains almost unaffected if a single neuron fails, or if a few synaptic connections break down. It usually requires a sizable fraction of failing elements before the deterioration is noticeable. In view of the rapidly increasing use of electronic data processing in vital areas, this is an attractive feature of neural computing, which may become of practical importance.

2.5.3 Neural Network Applications

Non-Civil Engineering Applications

Various neural network applications are listed in the 1988 DARPA Neural Network Study, beginning in about 1984 with the adaptive channel equalizer. This device, which is an outstanding commercial success, is a single neuron network used in long distance telephone systems to stabilize voice signals. The DARPA report goes on to list other commercial applications, including a small word recognizer, a process monitor, a sonar classifier, and a risk analysis system.

Neural Networks have been applied in many other fields since the DARPA report was written. A list of some of applications mentioned in the literature follows:

Aerospace High performance aircraft autopilot, flight path simulation, aircraft control systems, autopilot enhancements, aircraft component simulation, aircraft component fault detection.

Automotive Automobile automatic guidance system, warranty activity analysis.

Banking Check and other document reading, credit application evaluation

Defense Weapon steering, target tracking, object discrimination, facial recognition, new kinds of sensors, sonar, radar and image signal processing including data compression, feature extraction and noise suppression, signal/image identification.

Electronics Code sequence prediction, integrated circuit chip layout, process control, chip failure analysis, machine vision, voice synthesis, nonlinear modeling.

Entertainment Animation, special effects, market forecasting.

Financial Real estate appraisal, loan advisor, mortgage screening, corporate bond rating, credit line use analysis, portfolio trading program, corporate financial analysis, currency price prediction.

Insurance Policy application evaluation, product optimization.

Manufacturing Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, welding quality analysis, paper quality prediction, computer chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, project bidding, planing and management, dynamic modeling of chemical process system.

Medical Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, emergency room test advisement.

Oil and Gas Exploration

Robotics Trajectory control, forklift robot, manipulator controllers, vision systems.

Speech Speech recognition, speech compression, vowel classification, text to speech synthesis.

Securities Market analysis, automatic bond rating, stock trading advisory systems.

Telecommunications Image and data compression, automated information services, real-time translation of spoken language, customer payment processing systems.

Transportation Truck brake diagnosis systems, vehicle scheduling routing systems.

Civil Engineering Applications

They have not been as numerous as in some other fields, but their number has kept growing in the recent years.

The collections of articles published by B.H.V Topping have focused on the applications of advanced technologies in civil engineering. Several of these collections have revealed interesting neural network applications, such as:

- Computing in structural engineering
- Computer automation of structural design
- Modeling of non-linear structures using recurrent neural networks
- Control of large flexible manipulator systems
- Hydro-meteorological modeling
- Evaluation of seismic liquefaction

As stated in the introduction, one of the purposes of this research is to prove the possibility of control of a cantilever beam using neural networks. From this point of view, this thesis describes another application of neural networks in civil engineering.

Chapter 3

Improving the Performance of Supervised Feed-Forward Neural Networks

The previous chapter provided an introductory but thorough description of what neural networks are and what their applications can be. In this chapter, the theory of neural networks is explored in a deeper way since it is explained how an artificial neural network can be modified so that it performs a desired computation. At the same time however, the field of study is restrained. Firstly, only supervised learning, which is also called learning with a teacher (see section 2.2.2), is considered. Secondly, the chapter focuses on feed-forward neural networks only, i.e. networks with no feedback loops.

A thorough understanding of how a neural network learns is indeed necessary if one wants to improve the learning process in one's specific application. In the last part of this thesis, neural networks are used in the study of a cantilever beam that supports fluctuating loads. In other words, neural networks are here considered as possible function approximators and interpolators. Supervised learning is therefore mandatory, and mostly feed-forward networks have been used in this type of application.

As stated in section 2.2.1, neural networks are of interest because they can learn

how to perform a desired computation. Imagine that one has a table of input vectors and output targets corresponding to each other, and wants a device that outputs the right target each time it is fed with an input vector. One can build a neural network and assign random values to all its parameters, which are, for the moment, its synaptic weights and bias levels.

It is highly unlikely at this point that if the network is fed with the first input vector, it will give the desired first target. Somehow, one has to modify the parameters of the network to make it perform the right computation with the elements of the input vectors. This modification is what is called neural network training, and is addressed in the first section of this chapter.

But training is only one side of the process that allows one to improve the performance of a neural networks. Indeed, training occurs only once some basic elements have been chosen, such as what kind of architecture (i.e number of neurons, of layers) will be used, and what kind of data will be fed to the network. Both these elements influence the performance of a network, and, therefore, should not be chosen randomly. The second section of this chapter is devoted to this matter.

3.1 Training Algorithms

3.1.1 Learning Algorithms

Artificial networks learn to perform various tasks by adapting their parameters. Given the number of parameters even in a small neural network, an organized method is needed to update them. Such a method is called a learning algorithm. As previously stated in section 2.2.1, a learning algorithm is "a set of rules used to update the synoptic weights and bias level of a network during its training period."

Finding better, i.e. faster and less memory-consuming, training algorithms is one of the most actively pursued research topics for neural network scientists. Since the first studies of neural networks, many algorithms have been found, and literally hundreds of improvements have been made to them. Section 3.1.3 describes the

most famous algorithms that have resulted from this search for efficiency. Among the original algorithms that have been found, the most famous is certainly the back-propagation algorithm (also called backprop, or BP). The following section (3.1.2) is therefore devoted to its study.

3.1.2 The Back-Propagation Algorithm

Concept

The Back-Propagation algorithm was proposed in 1986 by Rumelhart, Hinton, and Williams for setting weights and hence for the training of multi-layer perceptrons. Once the Back-Propagation of Rumelhart et al. was published, it was very close to algorithms proposed earlier by Werbos in his Ph.D. dissertation in Harvard in 1974, and then in a report by D.B.Parker at Stanford in 1982, both unpublished and, thus, unavailable to the community at large. It goes without saying that the availability of a rigorous method to set intermediate weights, namely to train hidden layers of artificial neural networks, gave a major boost to the further development of neural networks, opening the way to overcome the single-layer shortcomings that had been pointed out by Minsky, and which nearly dealt a death blow to artificial neural networks.

Explanation

Since the back-propagation algorithm for a general feed-forward network is developed in this section , the convenient notation introduced in section 2.1.4 is used. This notation involves the use of vectors and matrices, which are easy to implement in a computer algorithm.

Consider a feed-forward network with a number O of layers.The output layer is numbered O (letter O) whereas the input layer is numbered 1 (number 1).

- In each layer j, there is a number S^j of neurons.
- As this is usually the case, each neuron in the same layer has the same activation function, i.e. there is only one kind of activation function per layer. Therefore,

the activation function of neurons in layer j will be noted f^{j} [1] .

- The input vector to this network is called $\vec{p}$ and is composed of I elements. The size of this vector is noted $I \times 1$.

- Each neuron in layer j has a bias level. All these bias levels form a vector of size S^j, which is noted $\vec{b^j}$.

- The network is assumed to be fully connected, i.e. each neuron in layer $j-1$ is connected to every neuron in layer j. Therefore, there is always a connection from a neuron in layer $j-1$ to a neuron in layer j. The case where the network is not fully connected can easily be deduced from this study by setting some connection weights to 0.

- The matrix of weights for the connections from layer $j-1$ to layer j is noted $LW^{j,j-1}$, and is of size $S^j \times S^{j-1}$. The matrix of weights for the connection to the first layer (layer 1) is temporarily called $IW^{1,1}$, as specified by the notation in section 2.1.4

- The output elements of layer j form a vector that is called $\vec{a^j}$.

- The net inputs to neurons in layer j form a vector that is called $\vec{n^j}$.

- The target elements of the network form a vector called $\vec{t}$, which is of size $T \times 1$. Since there are as many target elements as output neurons, necessarily $S^O = T$.

[1] Note that for a better understanding, indexes devoted to layers are written in superscript on every variables.

As a result, the network generates the following equations:

$$\begin{array}{lcll}
\vec{a}^1 & = & f^1(\vec{n}^1) & = f^1(IW^{1,1}\vec{p} + \vec{b}^1) \\
\vec{a}^2 & = & f^2(\vec{n}^2) & = f^2(LW^{2,1}\vec{a}^1 + \vec{b}^2) \\
\vec{a}^3 & = & f^3(\vec{n}^3) & = f^3(LW^{3,2}\vec{a}^2 + \vec{b}^3) \\
\vdots & & \vdots & \vdots \\
\vec{a}^{O-1} & = & f^{O-1}(\vec{n}^{O-1}) & = f^{O-1}(LW^{O-1,O-2}\vec{a}^{O-2} + \vec{b}^{O-1}) \\
\vec{a}^O & = & f^O(\vec{n}^O) & = f^O(LW^{O,O-1}\vec{a}^{O-1} + \vec{b}^O)
\end{array}$$

For the sake of simplification, the vector $\vec{p}$ will instead be called $\vec{a}^0$ in the rest of the BP algorithm explanation. Indeed, input vectors can be considered as outputs from a virtual 0th layer. Thus, $IW^{1,1}$ can instead be written $LW^{1,0}$. And the network can finally be summarized by the following set of equations:

$$\vec{a}^j = f^j(\vec{n}^j) = f^j(LW^{j,j-1}\vec{a}^{j-1} + \vec{b}^j) \quad \text{,for j} \in 1,\ldots,\text{O}$$

A training example represents a couple (input vector,target) from the data set. Each training example is presented to the network sequentially, i.e one at a time. The error signal at the output of neuron x in layer j and at iteration n (i.e. presentation of the nth training example) is defined by

$$e_x^j(n) = t_x^j(n) - a_x^j(n), \tag{3.1}$$

where $t_x^j(n)$ represents the desired target corresponding to the input vector, and $a_x^j(n)$ represents the actual output of this neuron consequent to the initial presentation of the input vector, maybe several layers before.

Note that here, a difference must be made between a neuron x belonging to the output layer O and a neuron x belonging to a hidden layer j.

- For the output neuron, $t_x^O(n)$ is an AVAILABLE target element. A vector of such element is known before the network training.

- On the other hand, there is no way to get targets readily for hidden neurons, since, as their name specifies it, there are hidden, and then, unobservable. For

such neurons, virtual targets, and therefore virtual error signal, will have to be computed from other parameters of the network. Here is the whole point of the back-prop algorithm: getting targets and error signals of neurons for which no target value is known, so as to be able to correct their free parameters. Output neurons are the only "visible" neurons for which error signals can be calculated directly.

Hence, an error signal is defined for every single neuron in the network, but finding its value will require some computation.

The instantaneous value of the error energy for neuron x in layer j is defined as $\frac{1}{2}(e_x^j(n))^2$. Correspondingly, the instantaneous value $\varepsilon^j(n)$ of the total error energy of layer j is obtained by summing $\frac{1}{2}(e_x^j(n))^2$ over all neurons in the jth layer. As a result:

$$\varepsilon^j(n) = \frac{1}{2}\sum_{x=1}^{x=S^j}(e_x^j(n))^2 \tag{3.2}$$

Among this error energies, one is more important than the other, namely the one of the output layer

$$\varepsilon^O(n) = \frac{1}{2}\sum_{x=1}^{x=S^O}(e_x^O(n))^2 \tag{3.3}$$

Indeed, this instantaneous error energy $\varepsilon^O(n)$ is a function of all free parameters (i.e. synaptic weights and bias levels) of the network. The objective of the learning process is to adjust the free parameters of the network to minimize this error energy. This minimization can be done using two different ways.

1. Every training example is presented to the neural network. For each training example n, the error $\varepsilon^O(n)$ is computed and the free parameters are updated according to it. In this case, the error energy to minimize is indeed $\varepsilon^O(n)$ and there are as many updates per epoch[2] as there are training examples in the data set. This is called sequential training.

2. Likewise, each training example is presented to the neural network, and again,

[2]one complete presentation of the training set, i.e. the presentation of all the training examples in the data set, is called an epoch

the error energy $\varepsilon^O(n)$ is computed. However, the free parameters are not updated after each training example has been presented. Instead, all training examples of the training set are presented and an average squared error energy is obtained by summing $\varepsilon^O(n)$ over all n and normalizing with respect to the set size N (i.e. the number of training example in the data set), as shown by

$$\varepsilon_{av}^O = \frac{1}{N}\sum_{n=1}^{N}\varepsilon^O(n) \tag{3.4}$$

In other words, an average error energy is computed for each epoch. After one epoch only, the free parameters are updated, this time using ε_{av}^O[3]. Thus, there is only one update per epoch. This is called batch training

The details of both types of training are highlighted in section 3.2.1. The reader will realize that there is little difference in the reasoning caused by this difference in the training process. Indeed, an explanation of the back-propagation algorithm using sequential training, thus invoking $\varepsilon^O(n)$, can be turned into an explanation using batch training, simply by replacing each $\varepsilon^O(n)$ with ε_{av}^O. Therefore, in the rest of this section, ε^O is used to show that both cases are considered at the same time.

Consider now Figure 3-1 which depicts neuron x in layer j being fed by a set of function signals produced by neurons in layer $j-1$ to its left. The net input to the activation function of neuron x at iteration n is therefore

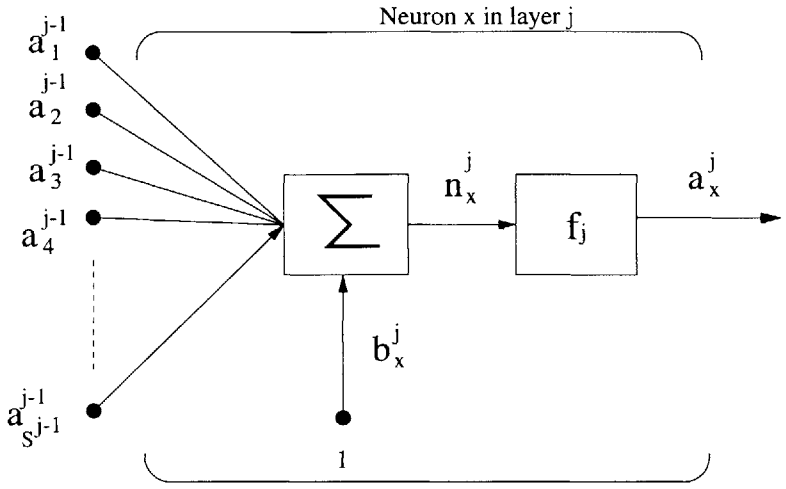
$$n_x^j(n) = \sum_{y=1}^{y=S^{j-1}} w_{xy}^{j,j-1}(n)a_y^{j-1}(n) + b_x^j(n)$$

where S^{j-1} is the total number of inputs applied to neuron x, which is equal to the total number of outputs in the $(j-1)$th layer in a fully connected feed-forward network. Note that:

1. The inputs to this neuron x are either elements of input vectors if this neuron belongs to the input layer, or output of earlier neurons if this neuron belongs

[3] ε_{av}^O, like $\varepsilon^O(n)$, is also a function of all the free parameters in the network

Figure 3-1: A random neuron x in a network



to a hidden or the output layer. This is why, to preserve generalization, they are called $a_y^{j-1}(n)$, a name devoted to neuron outputs.

2. The bias can be considered as a weight of a special connection, namely the one from input a_0, which is always equal to 1. As a result, $w_{x0}^{j,j-1} = 1$ for any x, j, and $j-1$, as well as $a_0^{j-1} = b_x^j$ for any x, j, and $j-1$.

As a result, the previous equation can be written:

$$n_x^j = \sum_{y=0}^{y=S^{j-1}} w_{xy}^{j,j-1} a_y^{j-1} \tag{3.5}$$

The output of neuron x is then

$$a_x^j = f^j(n_x^j) \tag{3.6}$$

where f^j is the activation function of all neurons in layer j.

The back-propagation algorithm then applies a correction $\Delta w_{xy}^{j,j-1}$ to the synaptic weight $w_{xy}^{j,j-1}$, which is proportional to the partial derivative $\frac{\partial \varepsilon^O}{\partial w_{xy}^{j,j-1}}$. The correction is indeed defined by:

$$\boxed{\Delta w_{xy}^{j,j-1} = -\eta \frac{\partial \varepsilon^O}{\partial w_{xy}^{j,j-1}}} \tag{3.7}$$

where η is the learning rate parameter of the back-propagation algorithm. The use of the minus sign in equation 3.7 accounts for steepest gradient descent in weight space, i.e. seeking a direction for weight change that reduces the value ε^O.

According to the chain rule of calculus, this gradient can be expressed as:

$$\Delta w_{xy}^{j,j-1} = -\eta \frac{\partial \varepsilon^O}{\partial a_x^j} \cdot \frac{\partial a_x^j}{\partial n_x^j} \cdot \frac{\partial n_x^j}{\partial w_{xy}^{j,j-1}} \tag{3.8}$$

Differentiating equation 3.5 with respect to $w_{xy}^{j,j-1}$ yields:

$$\frac{\partial n_x^j}{\partial w_{xy}^{j,j-1}} = a_y^{j-1} \tag{3.9}$$

Therefore, the weight correction becomes:

$$\Delta w_{xy}^{j,j-1} = \eta(-\frac{\partial \varepsilon^O}{\partial a_x^j} \cdot \frac{\partial a_x^j}{\partial n_x^j}) \cdot a_y^{j-1} \tag{3.10}$$

The value of the expression in parenthesis is different for every single neuron in the network (every x and j), and is called the local gradient. The following notation is introduced:

$$\boxed{\delta_x^j = -\frac{\partial \varepsilon^j}{\partial a_x^j} \cdot \frac{\partial a_x^j}{\partial n_x^j}} \tag{3.11}$$

With this notation, equation 3.10 takes the form:

$$\boxed{\Delta w_{xy}^{j,j-1} = \eta \cdot \delta_x^j \cdot a_y^{j-1}} \tag{3.12}$$

The key factor involved in the computation of the weight adjustment $\Delta w_{xy}^{j,j-1}$ is the local gradient δ_x^j. In this context, two cases can be identified, depending on where layer j is located in the network.

- In case 1, layer j is an output layer, which exactly means that $j = O$. Again, the chain rule of calculus is used in equation 3.11:

$$\delta_x^O = -\frac{\partial \varepsilon^O}{\partial e_x^O} \cdot \frac{\partial e_x^O}{\partial a_x^O} \cdot \frac{\partial a_x^O}{\partial n_x^O} \tag{3.13}$$

Differentiating both sides of equation 3.3 with respect to e_x^O leads to:

$$\frac{\partial \varepsilon^O}{\partial e_x^O} = e_x^O \tag{3.14}$$

Then differentiating both sides of equation 3.1 with j=O and with respect to a_x^O leads to:

$$\frac{\partial e_x^O}{\partial a_x^O} = -1 \tag{3.15}$$

Finally, differentiating equation 3.6 with $j = O$ and with respect to n_x^O yields:

$$\frac{\partial a_x^O}{\partial n_x^O} = f^{O\prime}(n_x^O) \tag{3.16}$$

where $f^{O\prime}$ stands for the first derivative of f^O, the transfer function of layer O.

As a result, the expression of the local gradient for neuron x in layer O becomes:

$$\delta_x^O = e_x^O \cdot f^{O\prime}(n_x^O) \tag{3.17}$$

- In case 2, layer j is an hidden layer. According to equations 3.11 and 3.16, the local gradient δ_j^x for a hidden neuron x in layer j may be defined as:

$$\delta_x^j = -\frac{\partial \varepsilon^O}{\partial a_x^j} \cdot f^{j\prime}(n_x^j) \tag{3.18}$$

To calculate the partial derivative of $\frac{\partial \varepsilon^O}{\partial a_x^j}$, one may proceed as follows. Firstly, equation 3.2 is differentiated with respect to the function signal a_x^j. This yields:

$$\frac{\partial \varepsilon^O}{\partial a_x^j} = \sum_{z=1}^{z=S^O} e_z^O \cdot \frac{\partial e_z^O}{\partial a_x^j} \tag{3.19}$$

Next, the chain rule is used for the partial derivative on the right-hand side:

$$\frac{\partial \varepsilon^O}{\partial a_x^j} = \sum_{z=1}^{z=S^O} e_z^O \cdot \frac{\partial e_z^O}{\partial n_z^O} \cdot \frac{\partial n_z^O}{\partial a_x^j} \tag{3.20}$$

Equation 3.1 applied to neuron Z in layer j yields:

$$e_z^O = t_z^O - a_z^O = t_z^O - f^O(n_z^O)$$

Therefore,

$$\frac{\partial e_z^O}{\partial n_z^O} = -f^{O\prime}(n_z^O) \tag{3.21}$$

Thus,

$$\frac{\partial \varepsilon^O}{\partial a_x^j} = -\sum_{z=1}^{z=S^O} e_z^O \cdot f^{O\prime}(n_z^O) \cdot \frac{\partial n_z^O}{\partial a_x^j} \tag{3.22}$$

and temporarily,

$$\delta_x^j = f^{j\prime}(n_x^j) \cdot \sum_{z=1}^{z=S^O} e_z^O \cdot f^{O\prime}(n_z^O) \cdot \frac{\partial n_z^O}{\partial a_x^j} \tag{3.23}$$

which, given equation 3.17, is equivalent to:

$$\delta_x^j = f^{j\prime}(n_x^j) \cdot \sum_{z=1}^{z=S^O} \delta_z^O \cdot \frac{\partial n_z^O}{\partial a_x^j} \tag{3.24}$$

This relation is of no use for the moment, so it needs to be developed. This is done gradually.

- Consider the specific case $j = O - 1$:

$$\delta_x^{O-1} = f^{O-1\prime}(n_x^{O-1}) \cdot \sum_{z=1}^{z=S^O} \delta_z^O \cdot \frac{\partial n_z^O}{\partial a_x^{O-1}} \tag{3.25}$$

  Given that

$$n_z^O = \sum_{x=0}^{x=S^O} w_{zx}^{O,O-1} a_x^{O-1} \tag{3.26}$$

  one gets

$$\frac{\partial n_z^O}{\partial a_x^{O-1}} = w_{zx}^{O,O-1} \tag{3.27}$$

  Therefore,

$$\delta_x^{O-1} = f^{O-1\prime}(n_x^{O-1}) \cdot \sum_{z=1}^{z=S^O} \delta_z^O \cdot w_{zx}^{O,O-1} \tag{3.28}$$

– Consider next $j = O - 2$:

$$\delta_x^{O-2} = f^{O-2\prime}(n_x^{O-2}) \sum_{z=1}^{z=S^O} \delta_z^O \cdot \frac{\partial n_z^O}{\partial a_x^{O-2}} \tag{3.29}$$

Now,

$$n_z^O = \sum_{y=0}^{y=S^{O-1}} w_{zy}^{O,O-1} a_y^{O-1} = \sum_{y=0}^{y=S^{O-1}} w_{zy}^{O,O-1} f^{O-1}(n_y^{O-1}) \tag{3.30}$$

which is equivalent to:

$$n_z^O = \sum_{y=0}^{y=S^{O-1}} w_{zy}^{O,O-1} f^{O-1}(\sum_{x=0}^{x=S^{O-2}} w_{yx}^{O-1,O-2} a_x^{O-2}) \tag{3.31}$$

As a result:

$$\frac{\partial n_z^O}{\partial a_x^{O-2}} = \sum_{y=0}^{y=S^{O-1}} w_{zy}^{O,O-1} f^{O-1\prime}(n_y^{O-1}) w_{yx}^{O-1,O-2} \tag{3.32}$$

which yields to:

$$\delta_x^{O-2} = f^{O-2\prime}(n_x^{O-2}) \cdot \sum_{z=1}^{z=S^O} \delta_z^O \sum_{y=0}^{y=SO-1} w_{zy}^{O,O-1} fO - 1\prime(n_y^{O-1}) w_{yx}^{O-1,O-2} \tag{3.33}$$

Then, switching the two $\sum$ signs, one obtains:

$$\delta_x^{O-2} = f^{O-2\prime}(n_x^{O-2}) \cdot \sum_{y=0}^{y=SO-1} [fO - 1\prime(n_y^{O-1}) \sum_{z=1}^{z=S^O} \delta_z^O w_{zy}^{O,O-1}] w_{yx}^{O-1,O-2} \tag{3.34}$$

Using equation 3.28, this yields:

$$\delta_x^{O-2} = fO - 2\prime(n_x^{O-2}) \cdot \sum_{y=0}^{y=S^{O-1}} \delta_y^{O-1} w_{yx}^{O-1,O-2} \tag{3.35}$$

Analyzing equation 3.28 and 3.35, the following pattern is highlighted:

$$\delta_x^j = f^{j\prime}(n_x^j) \cdot \sum_{z=1}^{z=Sj+1} \delta_z^{j+1} w_{zx}^{j+1,j} \tag{3.36}$$

This pattern can be generalized for all j.

The relations that have been derived for both cases of the back-propagation algorithm are now summarized. Firstly, the correction $\Delta w_{xy}^{j,j-1}$ applied to the synaptic weight connecting neuron x in layer $j-1$ to neuron y in layer j is defined by the delta rule:

$$\begin{pmatrix} Weight \\ correction \\ \Delta w_{xy}^{j,j-1} \end{pmatrix} = \begin{pmatrix} learning-rate \\ parameter \\ \eta \end{pmatrix} * \begin{pmatrix} local \\ gradient \\ \delta_x^j \end{pmatrix} * \begin{pmatrix} inputsignal \\ ofneuronj \\ a_y^{j-1} \end{pmatrix} \tag{3.37}$$

Secondly, the local gradient δ_x^j depends on whether neuron x is an output node or a hidden node:

1. If neuron x is an output neuron:

$$\boxed{\delta_x^j = \delta_x^O = f^{O\prime}(n_x^O) e_x^O}$$

2. If neuron j is a hidden node:

$$\boxed{\delta_x^j = f^{j\prime}(n_x^j) \cdot \sum_{z=1}^{z=Sj+1} \delta_z^{j+1} w_{zx}^{j+1,j}}$$

As a result, the application of the back-propagation algorithm consists of two passes through the different layers of the network: a forward pass and a backward pass.

- In the forward pass, an input vector is applied to the input nodes of the network, and its effect propagates through the network layer by layer. Finally, a set of outputs is produced as the actual response of the network.

- During the backward pass, on the other hand, the synaptic weights are all adjusted in accordance with the error-correction rule. Specifically, the actual response of the network is subtracted from a desired (target) response to produce an error signal. This error signal is then propagated backward through the network, against the direction of the synaptic connections – hence the name "back-propagation." The synaptic weight are adjusted to make the actual response of the network move closer to the desired response using the delta rule of error correction.

Matrix representation of the back-propagation algorithm

In each layer of a neural network, a local gradient vector can be defined as:

$$\vec{\delta^j} = \begin{pmatrix} \vec{\delta_1^j} \\ \vec{\delta_2^j} \\ \vdots \\ \vec{\delta_{S^j}^j} \end{pmatrix} c \tag{3.38}$$

Therefore, the expression:

$$\Delta w_{xy}^{j,j-1} = \eta \cdot \delta_x^j \cdot a_y^{j-1} \vee x, \vee y, \tag{3.39}$$

can now be represented as:

$$\Delta w^{j,j-1} = \eta \cdot \vec{\delta^j} \cdot \left(\vec{a^{j-1}}\right)^T \tag{3.40}$$

where T indicates the transpose of vector $\vec{a^{j-1}}$.

The previous formula gives the rule for weight-update between two presentations of data to the neural network. An index n is then given to each presentation. In other words,

- In batch mode, each n corresponds to the presentation of the whole data set.
- In sequential mode, each n corresponds to the presentation of one test case of

the data set.

After presentation n, the weight update for the back-propagation algorithm is therefore:

$$\Delta w^{j,j-1}(n) = \eta \cdot \overrightarrow{\delta^j(n)} \cdot \left(\overrightarrow{a^{j-1}(n)}\right)^T \tag{3.41}$$

Drawbacks of the back-propagation algorithm

The discovery of the back-propagation algorithm dramatically boosted neural network-based research. Backprop allowed scientists and engineers to train artificial neural networks with hidden neurons and layers, which was not possible before. In a very short time, the use of this algorithm spread throughout the neural network community. However, this algorithm is not a panacea, and the following section shows that great resources have been invested to improve it, or to find better algorithms. Among others, the drawbacks of back-propagation are:

(1) Although the elements of the training set may be presented in any order, the training set has to be presented to the network many times, typically hundreds or thousands of times, to bring the error down to an acceptable value.

(2) BP becomes computationally cumbersome as the number of hidden layers is increased. One of the major limitations of artificial neural network technology is the vast amount of computational power required to make the methods converge, even for moderately sized problems. This computational burden is felt during the training phase for the back-propagation method. In any event, the high computational burden is due to the high connectivity[4] of the network.

3.1.3 Better Algorithms

This section is devoted to other algorithms that either differ in some little improvements from the back-propagation algorithm, or implement new ideas but that can

[4]The connectivity of a neural network can be considered as the average number of neighbors to which a specific neuron is connected. It is NOT a measure of the strength of this network connections.

converge from ten to one hundred time faster than the raw back-propagation algorithm. Overall, they still follow the same pattern, which is:

1. Initialize weights and bias levels
2. Present inputs and desired outputs
3. Calculate actual outputs
4. Adapt weights and bias levels
5. If error is too big, repeat by going to step 2. Otherwise, stop.

Naturally, the goal of these new algorithm is to optimize step 4, since this is the most demanding as far as computation is concerned.

Steepest/Gradient Descent

This subsection discusses faster algorithms that use heuristic techniques, which were developed from an analysis of the performance of the standard steepest descent algorithm, back-prop.

Steepest Descent with Momentum This is another incremental learning algorithm for feed-forward networks that often provides faster convergence. Momentum allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a low pass filter, momentum allows the network to ignore small features in the error surface. Without momentum, a network may get stuck in a shallow local minimum. With momentum, a network can slide through such a minimum.

Momentum can be added to back-propagation learning by making weight changes equal to the sum of a fraction of the last weight change and the new change suggested by the back-propagation rule. The magnitude of the effect that the last weight change is allowed to have is mediated by a momentum constant, mc, which can be any number between 0 and 1. When the momentum constant is 0, a weight change is based solely

on the gradient. When the momentum constant is 1, the new weight change is set to equal the last weight change and the gradient is simply ignored.

As a result, the updating rule for the algorithm of steepest descent with momentum is:

$$\Delta w^{j,j-1}(n) = mc \cdot \Delta w^{j,j-1}(n-1) + (1-mc) \cdot \eta \cdot \overrightarrow{\delta^j(n)} \cdot \left(\overrightarrow{a^{j-1}(n)}\right)^T \tag{3.42}$$

Steepest Descent with Variable Learning Rate With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm may oscillate and become unstable. If the learning rate is too small, the algorithm will take too long to converge. It is not practical to determine the optimal setting for the learning rate before training and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

The performance of the steepest descent algorithm can be improved if the learning rate is allowed to change during the training process. An adaptive learning rate will attempt to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some change in the training procedure used by steepest descent. First, the initial network output and error are calculated. At each epoch, new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio, the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by a constant between 0 and 1). Otherwise, the new weights and biases are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by a constant greater than 1).

The variable learning rate procedure can be represented by the following formula:

$$\Delta w^{j,j-1}(n) = \eta(n) \cdot \overrightarrow{\delta^j(n)} \cdot \left(\overrightarrow{a^{j-1}(n)}\right)^T \tag{3.43}$$

Resilient Back-Propagation Multi-layer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called squashing functions, since they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slope must approach zero as the input gets large. This causes a problem when using steepest descent to train a multi-layer network with sigmoid functions, since the gradient can have a very small magnitude, and therefore cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient back-propagation training algorithm is to eliminate these harmful effects of the magnitudes of partial derivatives. Only the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor Δinc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor Δdec whenever the derivative with respect to that weight changes sign from the previous iteration. If the derivative is zero, then the update value remains the same. Whenever the weights are oscillating, the weight change will be reduced. If the weight continues to change in the same direction for several iterations, then the magnitude of the weight change will be increased.

Resilient back-propagation is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. A storage for the update values of each weight and bias is still needed, which is equivalent to storage of the gradient.

Conjugate Gradient Algorithm

The basic back-propagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient). This is the direction in which the performance function is decreasing most rapidly. It turns out that, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithms, a search is performed along conjugate directions.

In the training algorithms that have been discussed so far, a learning rate is used to determine the length of the weight update (step size). In most of the conjugate gradient algorithms, the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size which will minimize the performance function along that line.

Newton's and Quasi-Newton's methods

Newton's method is an alternative to the conjugate gradient methods for fast optimization. Suppose that one has a function $V(\vec{x})$ to minimize with respect to the parameter vector $\vec{x}$, then Newton's method would be:

$$\Delta\vec{x} = -\left[\bigtriangledown^2 V(\vec{x})\right]^{-1} \cdot \bigtriangledown V(\vec{x})$$

where $\bigtriangledown^2 V(\vec{x})$ is the Hessian matrix, and $\bigtriangledown V(\vec{x})$ is the gradient. In addition, if it is assumed that $V(\vec{x})$ is a sum of squares function:

$$V(\vec{x}) = \sum_{i=1}^{N} e_i^2(\vec{x})$$

, it can be shown that

$$\begin{aligned} \bigtriangledown V(\vec{x}) &= J^T(\vec{x}) \cdot \vec{e}(\vec{x}) \\ \bigtriangledown^2 V(\vec{x}) &= J^T(\vec{x}) \cdot J(\vec{x}) + S(\vec{x}) \end{aligned}$$

where $J(\vec{x})$ is the Jacobian matrix

$$J(\vec{x}) = \begin{bmatrix} \frac{\partial e_1(\vec{x})}{\partial x_1} & \frac{\partial e_1(\vec{x})}{\partial x_2} & \cdots & \frac{\partial e_1(\vec{x})}{\partial x_n} \\ \frac{\partial e_2(\vec{x})}{\partial x_1} & \frac{\partial e_2(\vec{x})}{\partial x_2} & \cdots & \frac{\partial e_2(\vec{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_N(\vec{x})}{\partial x_1} & \frac{\partial e_N(\vec{x})}{\partial x_2} & \cdots & \frac{\partial e_N(\vec{x})}{\partial x_n} \end{bmatrix}$$

and

$$S(\vec{x}) = \sum_{i=1}^{N} e_i(\vec{x}) \cdot \nabla^2 e_i(\vec{x})$$

For the Gauss-Newton method, it is assumed that $S(\vec{x}) \approx 0$, and the update becomes

$$\Delta\vec{x} = \left[J^T(\vec{x}) \cdot J(\vec{x})\right]^{-1} \cdot J^T(\vec{x}) \cdot \vec{e}(\vec{x})$$

Applying this method to a neural network problem requires some explanations. First, the function $V(\vec{x})$ to minimize is the error function ε. This is a function of all the parameters of the network. Therefore, these parameters can be gathered in a vector called $\vec{x}$, of dimension n if there are n weights and biases in the network, and then the Gauss-Newton's method can be applied.

This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian matrix must be stored, and its dimension is $n^2 \times n^2$, where, again, n is equal to the number of weights and biases in the network. For very large networks, it may be better to use resilient back-propagation or one of the conjugate gradient algorithms.

The Levenberg-Marquardt Algorithm

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix.

The Marquardt-Levenberg modification to the Gauss-Newton method is:

$$\Delta\vec{x} = \left[J^T(\vec{x}) \cdot J(\vec{x}) + \mu \cdot I\right]^{-1} \cdot J^T(\vec{x}) \cdot \vec{e}(\vec{x})$$

The parameter μ is multiplied by some factor whenever a step would result in an increased $V(\vec{x})$. When a step reduces $V(\vec{x})$, μ is divided by this same factor. Notice that when μ is large, the algorithm becomes steepest descent (with step $1/\mu$), while for small μ the algorithm becomes Gauss-Newton. The Marquardt-Levenberg algorithm can be considered a trust-region modification to Gauss-Newton.

Newton's method is faster and more accurate near an error minimum, so the aim is to shift towards Newton's method as quickly as possible. Thus, μ decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function will always be reduced at each iteration of the algorithm.

3.2 Other Considerations for an Improved Training

3.2.1 Sequential or Batch Mode of Training

From an "on-line" operational point of view, the sequential mode of training is preferred over the batch mode because it requires less local storage for each synaptic connection. Moreover, given that the patterns are presented to the network in a random manner, the use of pattern-by-pattern updating of weights makes the search in weight space stochastic in nature. This, in turn, makes it less likely for the back-propagation algorithm to be trapped in a local minimum.

In the same way, the stochastic nature of the sequential mode makes it difficult to establish theoretical conditions for convergence of the algorithm. In contrast, the use of batch mode of training provides an accurate estimate of the gradient vector; convergence to a local minimum is thereby guaranteed under simple conditions. Also

the composition of the batch mode makes it easier to parallelize than the sequential mode .

When the training data are redundant, it is found that, unlike the batch mode, the sequential mode is able to take advantage of this redundancy because the examples are presented one at a time. This is particularly so when the data set is large and highly redundant.

3.2.2 The Network Design Problem

While learning algorithms are practical methods of properly choosing the synaptic weights and thresholds of neurons, they provide no insight into the problem of how to choose the network architecture and learning rule parameters that are appropriate for the solution of a given problem.

For instance, focusing on the number of neurons and layers in the network, we come to the following compromise: if the number of hidden neurons is too small, no choice of the synapses may yield the accurate mapping between input and output, and the network will fail in the learning stage. If the number is too large, many different solutions exist, most of which will not result in the ability to generalize correctly for new input data, and the network will usually fail in the operational stage. Instead of learning salient features of the underlying input-output relationship, the network simply learns to distinguish somehow between the various patterns of the training set and to associate them with the correct output.

Typically, the understanding of such aspects as architecture and learning parameters is primarily empirical, various rules of thumb are followed that are derived from experience in practical applications. This heuristic approach presents two shortcomings:

1. The space of possible artificial neural network architectures is extremely large.

2. What constitutes a good architecture is intimately dependent on the application, i.e. on the problem that needs to be solved and the constraints on the

neural network solutions (e.g. fast learning and/or low connectivity and/or high accuracy).

As a consequence of these shortcomings, some significant amount of manual trial-and-error experimentation is necessary before adequate performance is achieved. No meaningful attempt is made to determine optimal architectures. Therefore, most applications adopt simple structures, and conservative values of learning parameters.

There are several approaches to finding the optimal network architecture. They are described in the following sections.

Pruning

This first method involves starting from a larger than necessary topology which is trained to learn the desired mapping. Then, individual connections or entire neurons are eliminated if they are not actively used or carry little weight. By this process of clipping or pruning, the network is eventually reduced in size.

The shortcomings of this approach are that one first has to deal with an unnecessarily large network, which is computationally wasteful, and that the pruning process may get stuck in an intermediate-size solution, which cannot be smoothly deformed into the optimal network architecture.

Growing

The second approach, which is also called the dynamic-node-creation method, follows an opposite line, starting with a small network and growing additional neurons until a solution can be found. If performed in a sufficiently careful manner, this method is guaranteed to find the smallest possible network that solves the task, at least for architectures involving only a single layer of hidden neurons.

However, it is necessary to retrain the complete network after the addition of each single new neuron, in order to make sure that a further increase in size occurs only if convergence of the learning procedure cannot yet be achieved. The simplified version of this approach, where only the newly added neuron and its synapses are trained and all old parameters remain frozen, does not, in general, find the optimal solution.

3.2.3 Early Stopping

Another method for improving generalization is called "early stopping." In this technique, the available data is divided into three subsets:

- The first subset is the training set which is used for computing the gradient and updating the network weights and biases.
- The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error will normally decrease during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set will typically begin to rise. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error is returned.
- The test set error is not used during the training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error in the test set reaches a minimum at a significantly different iteration number than the validation set error, this may indicate poor division of the data set.

Early stopping can be used with any of the learning algorithms which were described earlier in this chapter. One simply needs to pass the validation data to the training methods.

3.2.4 Pre-Processing of Data

There is another factor that has to be taken into account when training an artificial neural network, namely its input and output data. And, in this domain, two possibilities are imaginable:

1. Dealing with raw data. i.e. data that has not been transformed before input to the network, and that will not be transformed after output. The main advantage

of this method is that no information is lost. However, there are times when one does want to lose information, for instance when one wants to get rid of data values that lie far outside the possible range, and that must come from an experiment error or imprecision.

2. Pre-processing the input data and post-processing the output data using some standard statistical methods. Here, the salient features of your data is isolated. However, processing of input-output data should be approached with caution because it discards information. If this information is irrelevant, then standardizing cases can be helpful. If this information is important, then standardizing data can be disastrous.

Before starting a discussion on data processing, some definitions are needed.

- "Rescaling" a vector means to add or subtract a constant and then multiply or divide by a constant, as one would do to change the units of measurement of the data, for example, to convert a temperature from Celsius to Fahrenheit.

- "Normalizing" a vector most often means dividing by a norm of the vector, for example, to make the Euclidean length of the vector equal to 1. In the neural network literature, "normalizing" most often refers to rescaling by the minimum and range of the vector, to make all elements lie between 0 and 1.

- Finally, "standardizing" a vector most often means subtracting a measure of location and dividing by a measure of scale. For example, if the vector contains random values with a Gaussian distribution, one might subtract the mean and divide by the standard deviation, thereby obtaining a "standard-normal" random variable with mean 0 and standard deviation 1.

Now the question is, why do neural network designers do any of these things to their data ? First, if your output activation function has a range [0,1], then obviously it is mandatory to ensure that the target values lie within that range. But the main emphasis in the neural network literature has been on input values and the avoidance of saturation, hence the desire to use small random values.

Besides, standardizing input variables can have more important effects on initialization of the weights than simply avoiding saturation. Assume one has a multi-layer perceptron with one hidden layer applied and is therefore interested in the hyper-planes defined by each hidden unit. Each hyper-plane is the locus of points where the net-input to the hidden unit is zero and is, thus, the classification boundary generated by this hidden unit considered in isolation. The connection weights from the inputs to a hidden unit determine the orientation of the hyper-plane. The bias determines the distance of the hyper-plane from the origin. If the bias terms are all small random numbers, then all the hyper-planes will pass close to the origin. Hence, if the data is not centered at the origin, the hyper-plane may fail to pass through the data cloud. If all the inputs have a small coefficient of variation, it is quite possible that all the initial hyper-planes will miss the data entirely. With such a poor initialization, local minima are very likely to occur. It is therefore important to center the inputs to get good random initializations. In particular, scaling the inputs to [-1,1] will work better than [0,1], although any scaling that sets to zero the mean or median or other measure of central tendency is likely to be as good or better.

3.2.5 Genetic Algorithms

What are genetic algorithms?

Genetic algorithms were invented to mimic some of the processes observed in natural evolution. The latter takes place on chromosomes, which can be thought of organic devices for encoding the structure of the living beings. A living being is created partly through a process of decoding chromosomes. The specifics of chromosomal encoding and decoding processes are not fully understood, but here are some general features of the theory that are widely accepted:

- Evolution is a process that operates on chromosomes rather than on the living beings they encode.
- Natural selection is the link between chromosomes and the performance of their decoded structures. Processes of natural selection cause those chromosomes

that encode successful structures to reproduce more often than those that do not.

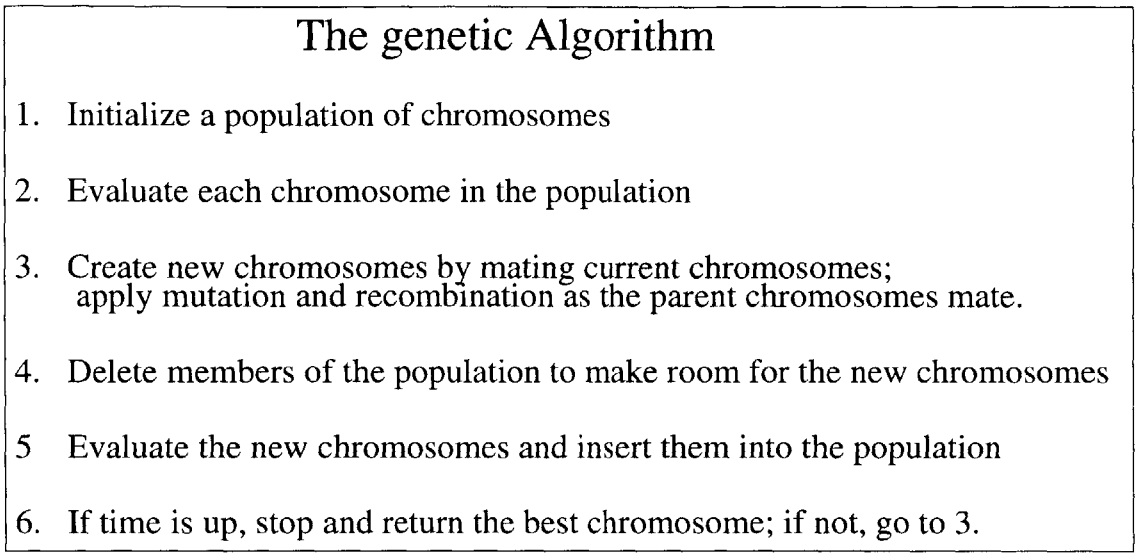
- The process of reproduction is the point at which evolution takes place. Mutations may cause the chromosomes of biological children to be different from those of their biological parents, and recombination processes may create quite different chromosomes in the children by combining material from the chromosomes of two parents.

- Biological evolution has no memory. Whatever it knows about producing individuals that will function well in their environments is contained in the gene pool–the set of chromosomes carried by the current individuals–and in the structure of the chromosome decoder.

These features of natural evolution intrigued John Holland in the early 1970's. Holland believed that, appropriately incorporated in a computer algorithm, they might yield a technique for solving difficult problems in the way that nature does. These algorithms, using simple encodings and reproduction mechanisms, displayed complicated behavior, and they turned out to solve some extremely difficult problems.

There are three important components to the genetic method. First, the technique for encoding solutions, which may vary from problem to problem and from genetic algorithm to genetic algorithm. In Holland's work, and in the work of most of his students, encoding was carried out using bit strings. Second, the evaluation function, which is the link between the genetic algorithm and the problem to be solved. An evaluation function takes a chromosome as input and returns a number or list of numbers that is a measure of the chromosome's performance on the problem to be solved. Given these initial components – a problem, a way of encoding solutions to it, and a function that returns a measure of how good any encoding is – a genetic algorithm can be used to carry out simulated evolution on a population of solutions. Figure 3-2 contains a top-level description of the genetic algorithm. If all goes well throughout this process of simulated evolution, an initial population of unexceptional chromo-

Figure 3-2: Top -level description of a genetic algorithm



somes will improve as parents are replaced by better and better children. The best individual in the final population produced can be highly-evolved solution to the problem.

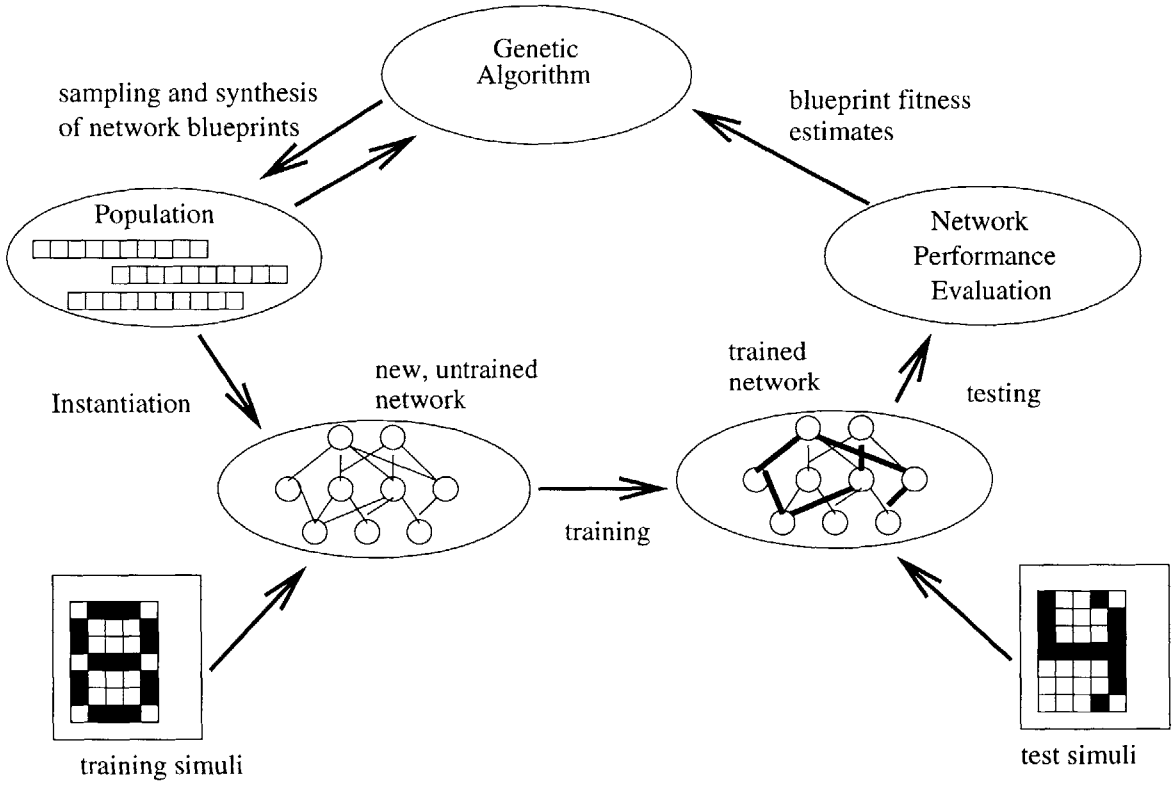
Relevance of the genetic approach to the neural network design problem

The problem of optimizing a neural network structure for a given set of performance criteria is a complicated one. They are many variables, and they interact in a complex manner. The evaluation of a given design is a noisy affair, since the efficacy of training depends on starting conditions that are typically random. In short, the problem is a logical application for genetic algorithms.

In 1990, Steven A. Harp and Tariq Samad built a system called NeuroGenesys, which goal was the genetic (i.e. using genetic algorithms) synthesis of neural network architectures. In their experiments , the system NeuroGenesys begins with a population of randomly generated networks. The structure of each network is described by a chromosome or genetic blueprint - a collection of genes that determine the anatomical properties of the network structure and the parameter values of the learning algorithm. They use back-propagation to train each of these networks to solve the

same problem, and they evaluate the fitness of each network in a population. They previously defined fitness to be a combined measure of worth on the problem, which may take into account learning speed, accuracy, and cost factors, such as the size and complexity of the network. Networks blueprint for a given generation beget offspring according to a reproductive plan that takes into consideration the relative fitness of individuals. A network spawned in this fashion will tend to contain attributes from both of its parents. A new network may also be a mutant, differing in few randomly selected genes from a parent. Novel feature may arise in either case: through synergy between the attributes of parents or through mutation. The basic cycle is illustrated in Figure 3-3. This process of training individual networks, measuring their fitness

Figure 3-3: Genetic synthesis of neural networks



and applying genetic operators to produce a new population of networks is repeated over many generations. If all goes well, each generation will tend to contain more of the features that were found useful in previous generation, and an improvement in overall performance can be realized over the previous generation.

Several interesting research issues are involved in using genetic algorithms for designing neural networks. These include the representation of the blueprint that specifies both the structure and the learning rule, the choice of the underlying space of network architecture to explore, adaptations of the genetic operators used to construct meaningful network structures, and the form of the evaluation function that determines the fitness of a network.

Chapter 4

Neurocontrol

A detailed discussion of neural networks was presented in chapter 2. In this chapter, their application to civil engineering is described. Neural networks in civil structures are part of smart systems, where they act as controllers. This kind of application gave birth to the field of "neurocontrol," literally, control using artificial neural networks. In what follows, a brief overview of control is first presented, and then key aspects of neurocontrol are considered.

4.1 Definition of Control

Dimitris C. Dracopoulos [6] defines the concept of control as follows:

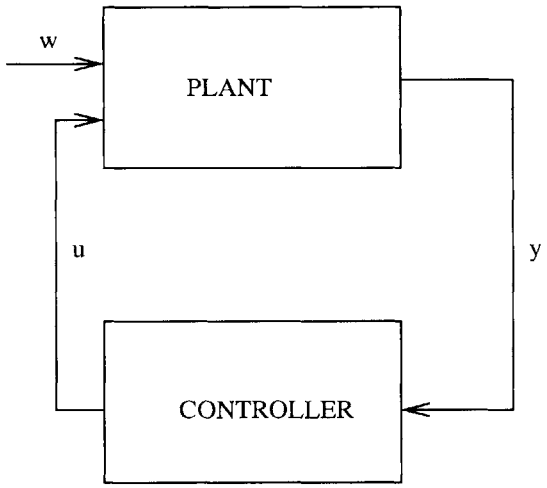
> "Although the study of a particular dynamic situation is sometimes motivated by the simple philosophic desire to understand the world and its phenomena, many analyses have the explicit motivation of devising effective means for changing a system, so that its behavior pattern is, in some way, improved. The means for affecting behavior can be described with the term control.
>
> Control of a process means the ability to direct, alter, or improve its behavior, and a control system is one in which some quantities of interest are maintained accurately around a prescribed value. Control becomes truly automatic when systems are made to be self-regulating."

The latter is done by introducing the concept of feedback, which consists of the triad:

- measurement
- comparison
- correction

By measuring the quantity of interest, comparing it with the desired value, and using the error to correct the process, the familiar chain of cause and effect in a process is converted into a closed loop of interdependent events. Generally in the control literature, the process to be controlled is called the plant and interacts with its environment by means of two types of input and output (see Figure 4-1). The signal

Figure 4-1: Feedback Control



w contains all input that cannot be directly affected by the controller. The signal y, which is the measured output of the plant, contains all data which are available to the controller, such as the values of the state variables. The control signal u is the part of the plant input which can be manipulated. This closed sequence of information transmission, referred to as feedback, underlies the entire technology of automatic control based on self-regulation.

4.2 Concepts of Control

In his book [7], Tomas Hrycej reviews the important results of classical control theory; all of them are useful for neurocontrol. The following sections give an overview of these results.

4.2.1 Linear Control

Linear control theory provides valuable insight into the nature of control. It has developed fundamental frameworks and concepts for the investigation of stability, optimality, robustness, and adaptivity.

Linear control is concerned with systems of the form

$$\dot{z} = A \cdot z + B \cdot u \tag{4.1}$$

with state z, input u, and measurable output

$$y = C \cdot z \tag{4.2}$$

and controllers of the form

$$u = -K \cdot z + M \cdot r \tag{4.3}$$

with r the reference state, that is, the state to which the plant is to be brought with the help of the controller. A, B, C, K, and M are matrices of appropriate dimensions. The goals of linear design are as follows:

- Stability of the closed loop, that is, convergence back to an equilibrium point after being moved away from this point by a disturbance.
- Optimality of the closed-loop behavior in some user-defined sense.

For linear systems, local stability is identical with global stability, which is the convergence to an equilibrium point from all points of the state space. A stability guided design is based on the analysis of closed-loop eigenvectors. The condition to be sat-

isfied is that real parts of all eigenvectors must be negative. Negative eigenvectors real parts are synonymous with the trajectory of the closed loop experiencing either damped oscillations or exponential convergence.

However, with the stability condition satisfied, the controller is still under-determined. There are two additional requirements:

1. A complete model for the closed-loop response (reference model)

2. Optimization

In using a reference model, the properties of a closed loop (consisting of, say, a plant and a feedback controller) can be specified in terms of eigenvalues. Because eigenvalues completely describe the behavior of a linear system, a plant with arbitrary dynamic properties can be transformed, with the help of a feedback controller, into a closed loop with arbitrarily different dynamic properties. This is true under some conditions that, at least in principle and with limited precision, are relatively frequently satisfied in practice. In theory, with an appropriate sequence of control actions , linear discrete systems with n state variables are guaranteed to be able to reach an arbitrary goal state within n steps. But the possibility of changing the dynamic properties of a closed loop can hardly be applied in practice.

Another, and perhaps more straightforward, way to close the under-specification gap is by defining a cost (or utility) function that is to be minimized (maximized).

Most of the methods can be applied under the following two conditions:

Controllability All state variables are affected by control action.

Observability All state variables affect the measurable variables and can thus be reconstructed with the help of consecutive observations.

For both the plant model and the controller, the theory gives hints to the way in which the information about measurable output and input can substitute for the knowledge of a complete state. This defines the structure of such sufficient models and controllers — a valuable help for neurocontrol formulations.

A basic theorem setting the limits of what can be done, under formal conditions, is as follows:

> "A controllable and observable plant (i.e. even one that is unstable in an open loop) can be stabilized by complete feedback.

4.2.2 Non-Linear Control

Non-linear control theory is concerned with general systems of the form

$$\dot{z} = F(z, u) \tag{4.4}$$

with measurable output

$$y = H(z) \tag{4.5}$$

and controllers of the form

$$u = G(z, r) \tag{4.6}$$

An analytical solution is known only for subclass of non-linear systems described by

$$\dot{z} = F_1(z) + F_2(z) \cdot u \tag{4.7}$$

The above analytical approach describes a fairly broad class of real problems. However, the problem of its applicability resides in the computational expense for evaluating the controller (matrix inversion in every sampling period).

The possibilities of formulating control are, in principle, the same as for linear systems: they depend on a reference model or a cost function. The difference is in the feasibility of reference-model-based formulation. The behavior of simple linear reference models, in particular, first- and second-order models can be described in terms of intuitively comprehensive concepts such as damping and time constants. By contrast, higher-order linear models have no such intuitive representation. For non-linear systems, a lot of control engineering competence is necessary to find reference models that can be exactly followed by the plant and, in addition, can express

controller design preferences.

This is why formulating control goals via cost function is frequently the better alternative. Then, some general optimal control approach such as dynamic programming is applied.

The difficulties with genuine non-linear controller designs suggest instead the linearization approach, also known as gain scheduling. It consists of the following steps:

1. The main working points of the plants are elaborated
2. The plant is linearized for each working point
3. Linear design is applied to each working point
4. An interpolation method is used for determining the controller action in states between the distinguished working points.

Stability of non-linear systems

One of the most important achievements of non-linear control theory are the concepts and tools for investigating the stability of non-linear systems. There are several definitions of stability:

Lyapunov Stability , in which the system is sure not to leave a certain region, or *Lyapunov asymptotic stability*, in which the system is sure to converge to an attractor point.

Input/Output Stability , with a closely related *bounded input/bounded output* stability, in which certain types of input behavior will lead to a certain types of output behavior.

Total Stability , in which stability is guaranteed under certain types of disturbances.

The Lyapunov stability concepts seem to be most flexible. A function of the closed-loop state, the Lyapunov function, is constructed with the following properties.:

1. The function is positive definite and has its minimum in the point (or region) for which the stability is to be proved.

2. The time derivative of the function is negative. That is, the trajectory of closed loop follows a path along which the Lyapunov function value diminishes.

Lyapunov stability is somehow intuitive. It assigns to each state a value that can be interpreted as a distance from the goal state. The closed-loop state follows the trajectories along which the distance monotically decreases.

The importance of the Lyapunov function can be made more obvious if we take into account that it defines, to a certain degree, the controller, or, at least, a class of stable controllers for a given problem. Suppose a Lyapunov function (for discrete time) is explicitly known. Then, the controller action is to be such that the next state of the plant has a lower Lyapunov function value than the present state.

4.2.3 Optimal Control

The goal of optimal control theory is to design controllers that are optimized to (or even can be proved to be the best with regard to) a certain performance criterion. For a linear plant and a quadratic performance criterion, the Ricatti controller represents an explicit and global solution of the problem.

Another setting in which a globally optimal control solution can be found is in a discrete space with discrete actions. For each state, one selects and assigns to that state the action producing the best cost-function value resulting from immediate cost of the present state, the cost of the action, and the cumulative recursively computed cost of the next state and all subsequent states passed under the assumption of the optimal action sequence.

Dynamic optimization is a general scheme for state evaluation and selection of the optimal action. Alternatively, if each state at each sampling period is represented by a node in a directed graph and actions are represented by connecting edges of the subsequent states, then the task can also be transformed to the critical path problem of graph theory.

4.2.4 Robust Control

Robust control addresses the problem of controlling a plant whose behavior is slightly different from that of a plant model. The reasons for the difference may be the differing structure or parameters of exact and approximate models, a systematic disturbance, or a random disturbance.

A popular approach to robust control is concerned with preserving stability. The closed-loop eigenvalues are chosen so that they remain in the stability region even if the plant model should change in a defined range.

Another approach is concerned with preserving the closed-loop equilibrium point under plant modifications or systematic disturbances. The means to reach this goal is error integration. Since the integrated error grows with time if the error itself does not converge to zero, a controller using an integral of error as an additional input can be designed for a closed loop to converge to the desired equilibrium point even under model imprecision or systematic disturbance.

Some classical results are available about systematic disturbances and necessary controller inputs:

1. Controller input including an error integral is necessary if stability under constant additive disturbance is to be reached

2. Controller input including the error integral is necessary if stability under linearly changing additive disturbance (i.e., a ramp) is to be reached.

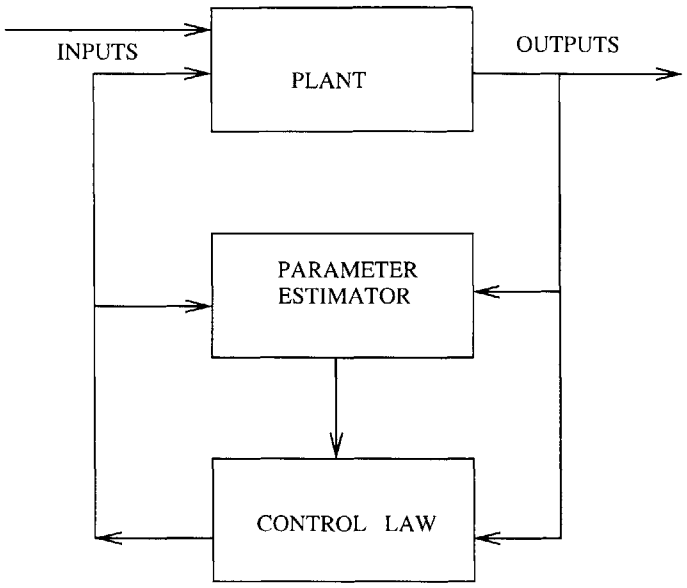
4.2.5 Adaptive Control

According to Dimitris C. Dracopoulos [6], the concept of adaptive control is as follows.

> "There are many design techniques for generating control strategies when the model of the system is known. When the model is unknown, on-line parameter estimation could be combined with on-line control. This leads to adaptive, or self-learning, controllers."

The basic structure of an adaptive controller is shown in Figure 4-2 An adaptive con-

Figure 4-2: Basic structure of an adaptive controller



troller can be defined as a feedback regulator that can modify its behavior in response to changes in the dynamics of the process and the disturbances, so as to operate in an optimum manner according to some specified criterion. Adaptive control techniques have been developed for systems that must perform well over large ranges of uncertainties due to large variations in parameter values, environmental conditions, and signal inputs. These adaptive techniques generally incorporate a second feedback loop, which is outside the first feedback loop. This second loop may have the capability to track system parameters, environmental conditions, and input characteristics. Then feedback control may vary parameters in compensation elements of the inner loop to maintain acceptable performance characteristics.

As Tomas Hrycej explains it in [7],

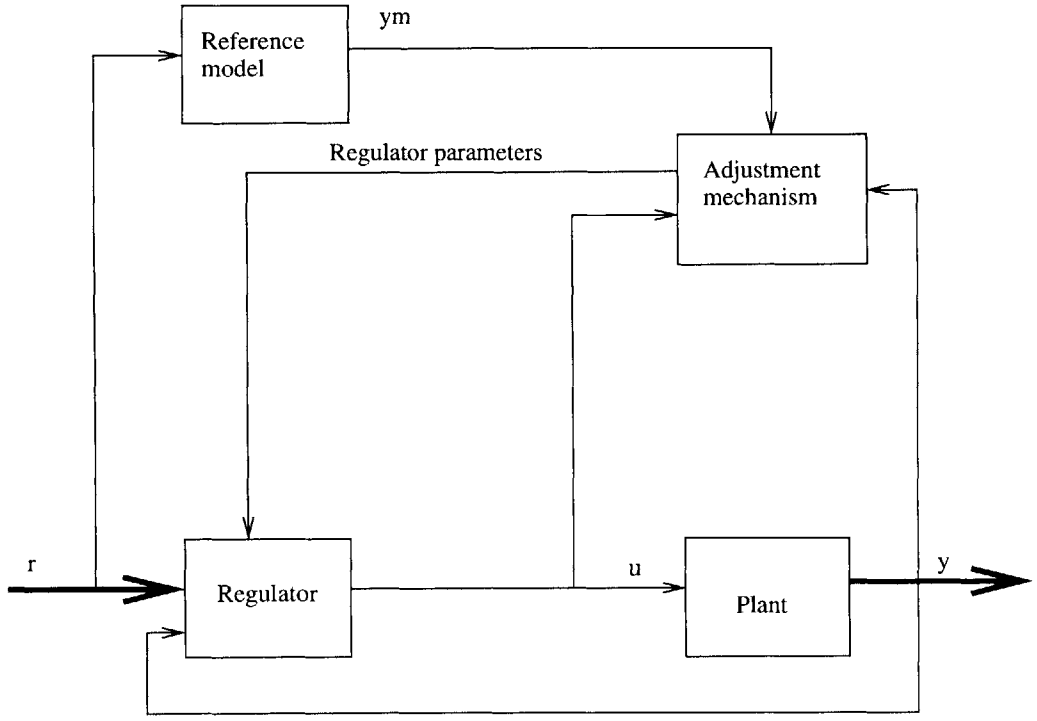
> "Adaptive control is another way to reach a goal similar to that of robust control. Instead of designing robust controllers that work under conditions different from those for which they have been designed, adaptive controllers recognize the difference between the assumption and reality and change to perform better in the new conditions."

The following paragraphs describe two of the most well known traditional adaptive control techniques.

The Model-Reference Adaptive Control

The desired performance is expressed in terms of a reference model, which gives the desired response to a command signal. The system also has an ordinary feedback loop composed of the process and the regulator. The error e is the difference between the output of the system y and the reference model y_m. The regulator has parameters that are changed based on the error. There are thus two loops in Figure 4-3: an inner

Figure 4-3: Block diagram of a model-reference adaptive system (MRAC)

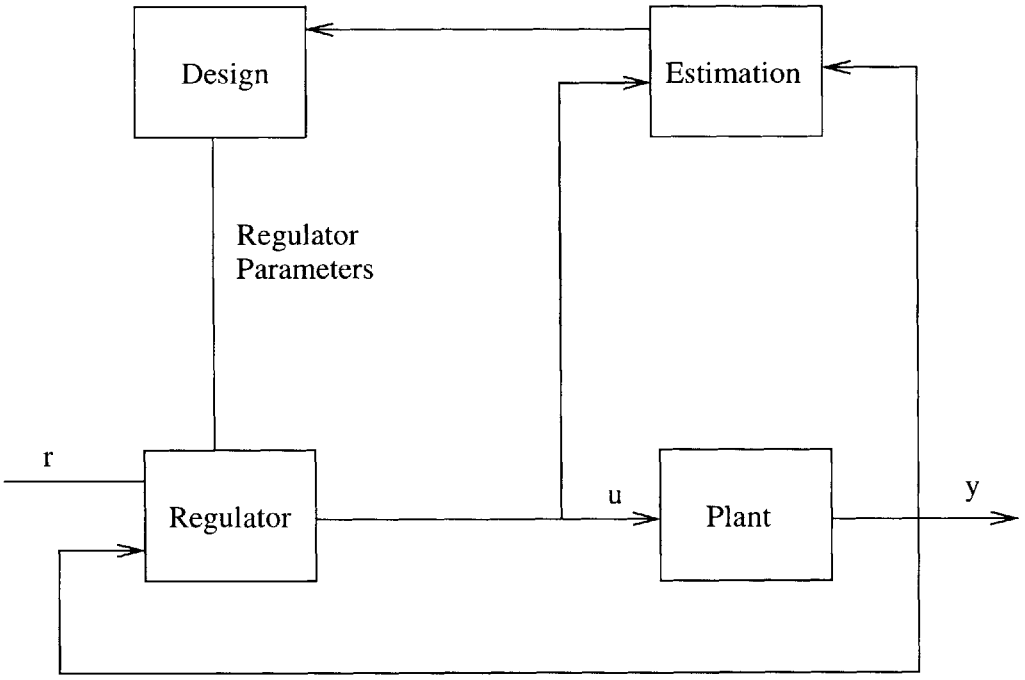


loop which provides the ordinary control feedback, and an outer loop which adjusts the parameters in the inner loop. The inner loop is assumed to be faster than the outer loop.

The Self-Tuning Regulator

In an adaptive system, it is assumed that the regulator parameters are adjusted all the time. This implies that the regulator parameters follow changes in the process. However, it is difficult to analyze the convergence and stability properties of such systems. To simplify the problem, it can be assumed that the process has constant but unknown parameters. When the process is known, the design procedure specifies a set of desired control parameters. The adaptive controller should converge to these parameter values even if the process is unknown. A regulator with this property is called self-tuning, since it automatically tunes the controller to the desired performance.

Figure 4-4: Block-diagram of a self-tuning regulator



The self-tuning regulator (STR) is based on the idea of separating the estimation of unknown parameters from the design of the controller. The basic idea is illustrated in Figure 4-4.

The unknown parameters are estimated on-line, using a recursive estimation method. The estimated parameters are treated as if they were true.

4.2.6 Intelligent control

The objective of the design of an intelligent control system is similar to that of the adaptive control system. However, there is a difference. For an intelligent control system, the range of uncertainty is substantially greater than can be tolerated by algorithms for adaptive systems. The main objective with intelligent control is to design a system with acceptable performance characteristics over a very wide range of uncertainty.

4.3 Fundamental Approaches to Neurocontrol

The definition of neurocontrol is very simple, namely, "control using artificial neural networks." Chapters 2 and 3 show that neural networks have diverse capabilities. Their ability to approximate functions given a set of points is considered in this section.

In neurocontrol, the task performed by neural networks is indeed function approximation, where the function to be approximated is the optimal controller. However, the function values to be approximated (i.e control actions) are usually not explicitly known. What is known are general conditions or optimality criteria for the consequences of these actions.

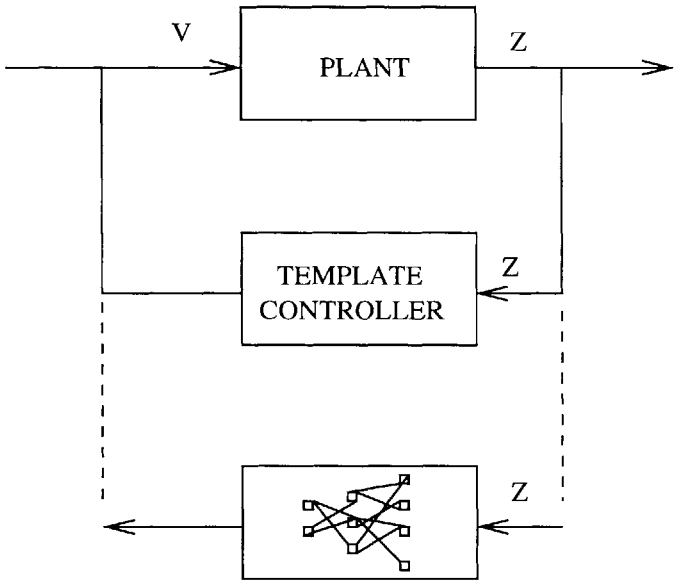
The problem of finding the relationship between the functional values and their evaluated consequences is called the credit assignment problem. This problem is central for neurocontrol methods. So it is appropriate to classify the fundamental approaches to neurocontrol by the way they address the credit assignment problem. Such a classification is attempted in the following sections.

4.3.1 Template learning

One way to solve the credit assignment problem is to take a template controller as a generator of control actions. Then, the functional values are known, and some of the standard learning methods for neural network based functional approximation

can be used. In other words, the fundamental cost function for template learning measures the dissimilarity between template outputs and outputs computed by the neurocontroller for given controller inputs. The scheme of the method is given in Figure 4-5. The drawing suggests the critical question of why one should train a neural

Figure 4-5: Template Learning



controller if the template controller is already available. There are some applications where this approach is reasonable:

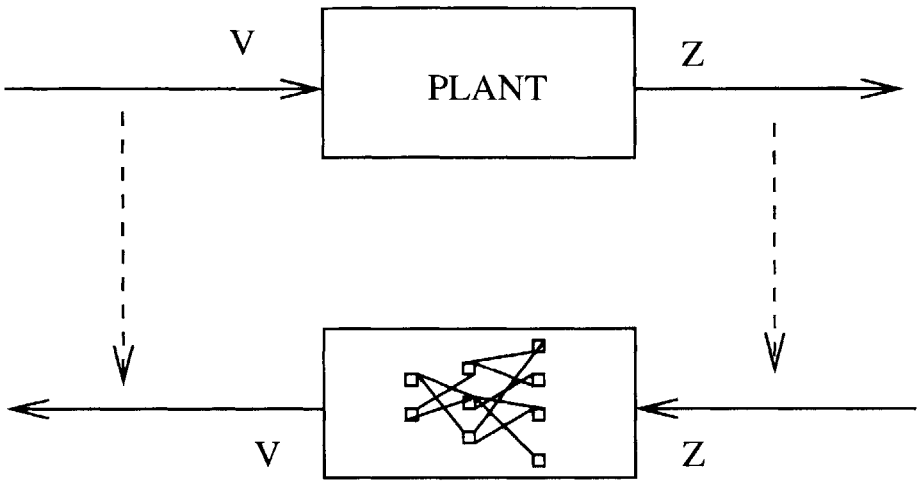
- The template controller is available, but not as an automatic device. For example, one is trying to mimic the behavior of a human controller, which is difficult if the latter is an expert with long experience in process control.
- The template controller is too complex for the target setting. Some controller types such as multidimensional look-up tables gained by dynamic optimization are too large for implementation on micro-controllers.

4.3.2 Learning Plant Inversion

The basic principle in "learning plant inversion is the following." The plant output is viewed as a function of the plant input, that is, the existence of the mapping

input→output is assumed. Basically, the inverse mapping output→input is sought, as can be seen in Figure 4-6. An important asset of the plant inversion approach is

Figure 4-6: Plant Inversion Learning



that it can use well-known, frequently implemented, and conceptually simple methods for training by input/output pairs such as back-propagation-based methods.

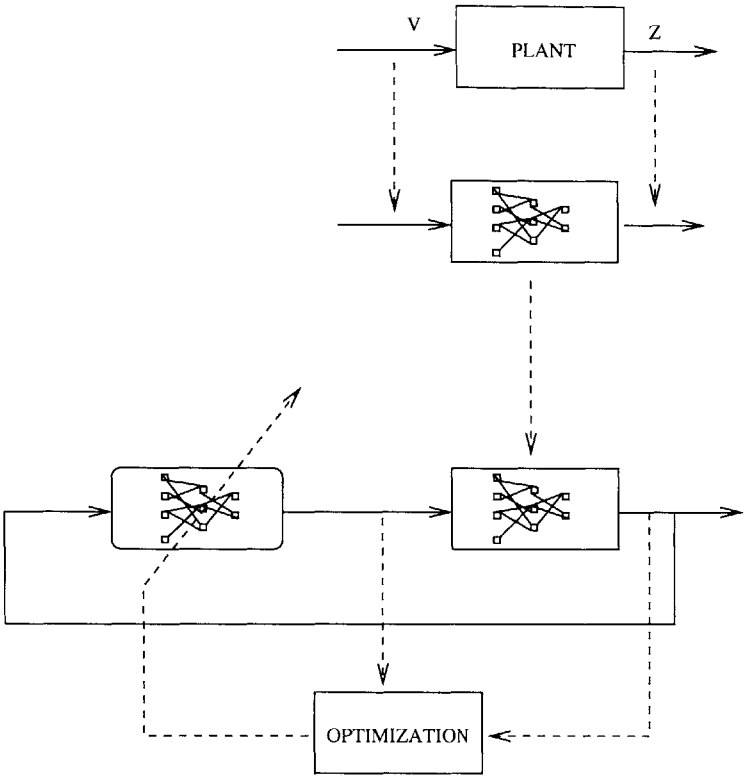
4.3.3 Closed-Loop Optimization

The next group of control approaches is characterized by relating the control performance to an explicit cost function. The only condition this cost function has to satisfy is that it must be a function of plant states and inputs (i.e. of controller actions) in a closed loop. Credit assignment is done by figuring out what effect free controller parameters have on the cost function in a mathematical way.

This approach uses a mathematical description of the plant, referred to as the plant model, which also involves an artificial neural network. A general scheme for control loop optimization is presented in Figure 4-7. The fundamental cost function for plant model identification measures the dissimilarity between measured plant outputs and outputs computed by the neurocontroller for given plant inputs.

Several basic variants of this scheme have been proposed. The scheme using a reference model of closed loop tunes the controller to minimize the deviation between

Figure 4-7: Closed-loop Optimization



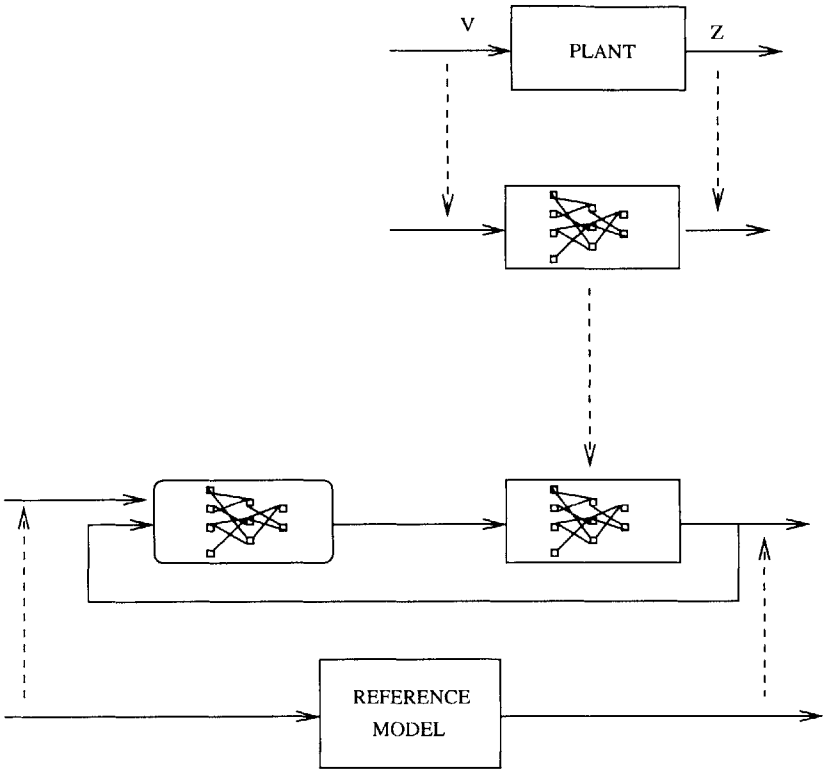
the behavior of the closed loop made by a real plant and the controller on the one hand, and the behavior of the predefined reference model on the other hand. The reference model plays the role of generator of sample pairs for neurocontroller training. Figure 4-8 shows the architecture for this variant.

4.3.4 Critic Systems

A large group of neurocontrol approaches are categorized as critic systems. In such systems, two neural networks are present and they perform different functions.

- The first is the neurocontroller, usually called the action network. This neurocontroller is completely analogous to that of other neurocontrol approaches.
- The second network is the critic. Like the plant model network in closed-loop optimization, this network's role is to train the neurocontroller. In contrast to

Figure 4-8: Closed-loop Optimization with Reference Model



the closed-loop optimization, this network does not model the plant but models individual plant states from the viewpoint of the optimality criterion.

The study and training of a critic system are difficult complex topics beyond the scope of this thesis. Therefore, they are not discussed here. However, a definitive treatment is contained in [7].

Optimization of a closed loop is currently the preferred choice for industrial applications. It is selected for the case study presented in the next chapter.

4.4 A New Direction for Control: Fuzzy Neural Networks

This section describes another advance in control, namely, the use of Fuzzy Logic. Ultimately, fuzzy logic and neural networks can be merge together into a control

system which, on the one hand, seems to have very interesting properties but, on the other hand, still remains to be fully understood.

4.4.1 History of Fuzzy Logic

In 1965, when Lotfi Zadeh at UC Berkeley's Department of Electrical Engineering published his first paper on fuzzy sets, he was implicitly advancing the thesis that one of the reasons humans are better at control than currently existing machines is that they are able to make effective decisions on the basis of imprecise linguistic information. Hence it should be possible to improve the performance of electro-mechanical controllers by modeling the way in which humans reason with this type of information.

The theory developed slowly at first, but by the early 70's it had attracted a small international following. In those days, the interest was spurred primarily by intellectual curiosity, although even then there was a pervasive belief in the theory's ultimate applicability. During this time, investigations focused mainly on the mathematical properties of fuzzy sets and closely related notions, and numerous variants of fuzzy logic were explored.

By the late 70's, interest in fuzzy systems had grown rather explosively, attracting many researchers from around the world and spawning bibliographies with citations numbering in the thousands. Still, most of the work was theoretical. The main topics included fuzzy knowledge representation and reasoning schemes, the philosophical ramifications of fuzzy logic and fuzzy set theory, fuzzifications of various branches of classical mathematics, and several foundational challenges posed by probability theorists and the classical AI community. Partly in response to this, Zadeh put forth "possibility theory", which showed how the fuzzy-sets model of natural language reasoning could be provided with an intuitively acceptable foundation, and at the same time explained how this was distinct from probability theory.

4.4.2 What is Fuzzy Logic?

The problem: real-world vagueness

Natural language abounds with vague and imprecise concepts, such as "Sally is tall," or "It is very hot today." Such statements are difficult to translate into more precise language without losing some of their semantic value: for example, the statement "Sally's height is 152 cm." does not explicitly state that she is tall, and the statement "Sally's height is 1.2 standard deviations about the mean height for women of her age in her culture" is fraught with difficulties: would a woman 1.1999999 standard deviations above the mean be tall? Which culture does Sally belong to, and how is membership in it defined?

While it might be argued that such vagueness is an obstacle to clarity of meaning, only the most staunch traditionalists would hold that there is no loss of richness of meaning when statements such as "Sally is tall" are discarded from a language. Yet this is just what happens when one tries to translate human language into classic logic. Such a loss is not noticed in the development of a payroll program, perhaps, but when one wants to allow for natural language queries, or "knowledge representation" in expert systems, the meanings lost are often those being searched for.

While some of the decisions and calculations could be done using traditional logic, fuzzy systems afford a broader, richer field of data and manipulation of that data than do more traditional methods.

Basic concepts

The notion central to fuzzy systems is that truth values (in fuzzy logic) or membership values (in fuzzy sets) are indicated by a value on the range $[0.0, 1.0]$, with 0.0 representing absolute falseness and 1.0 representing absolute truth. For example, consider the statement:

"Jane is old."

If Jane's age was 75, the statement might be assigned the truth value of 0.80. The statement could be translated into set terminology as follows:

"Jane is a member of the set of old people."

This statement would be rendered symbolically with fuzzy sets as:

$$mOLD(Jane) = 0.80$$

where m is the membership function, operating in this case on the fuzzy set of old people, which returns a value between 0.0 and 1.0.

At this juncture it is important to point out the distinction between fuzzy systems and probability. Both operate over the same numeric range, and at first glance both have similar values: 0.0 representing false (or non-membership), and 1.0 representing true (or membership). However, there is a distinction to be made between the two statements: the probabilistic approach yields the natural-language statement, "There is an 80corresponds to "Jane's degree of membership within the set of old people is 0.80." The semantic difference is significant: the first view supposes that Jane is or is not old. By contrast, fuzzy terminology supposes that Jane is "more or less" old, or some other term corresponding to the value of 0.80. Further distinctions arising out of the operations will be noted below.

The next step in establishing a complete system of fuzzy logic is to define the operations of EMPTY, EQUAL, COMPLEMENT (NOT), CONTAINMENT, UNION (OR), and INTERSECTION (AND). Before this can be done rigorously, some formal definitions are needed:

Definition 1 Let X be some set of objects, with elements noted as x. Thus, $X = x$.

Definition 2 A fuzzy set A in X is characterized by a membership function $mA(x)$ which maps each point in X onto the real interval $[0.0, 1.0]$. As $mA(x)$ approaches 1.0, the "grade of membership" of x in A increases.

Definition 3 A is EMPTY iff for all $x, mA(x) = 0.0$.

Definition 4 $A = B$ iff for all $x : mA(x) = mB(x) (or, mA = mB)$.

Definition 5 $mA' = 1 - mA$.

Definition 6 A is CONTAINED in B iff $mA <= mB$.

Definition 7 $C = A$ UNION B, where: $mC(x) = MAX(mA(x), mB(x))$.

Definition 8 $C = A$ INTERSECTION B where: $mC(x) = MIN(mA(x), mB(x))$.

It is important to note the last two operations, UNION (OR) and INTERSECTION (AND), which represent the clearest point of departure from a probabilistic theory for sets to fuzzy sets. Operationally, the differences are as follows:

- For independent events, the probabilistic operation for AND is multiplication, which (it can be argued) is counterintuitive for fuzzy systems. For example, let us presume that $x = Bob$, S is the fuzzy set of smart people, and T is the fuzzy set of tall people. Then, if $mS(x) = 0.90$ and $uT(x) = 0.90$, the probabilistic result would be:

  $$mS(x) * mT(x) = 0.81$$

  whereas the fuzzy result would be:

  $$MIN(uS(x), uT(x)) = 0.90$$

  The probabilistic calculation yields a result that is lower than either of the two initial values, which when viewed as "the chance of knowing" makes good sense.

  However, in fuzzy terms the two membership functions would read something like "Bob is very smart" and "Bob is very tall." If one presumes for the sake of argument that "very" is a stronger term than "quite," and that "quite" would be correlated with the value 0.81, then the semantic difference becomes obvious. The probabilistic calculation would yield the statement:

  "If Bob is very smart, and Bob is very tall, then Bob is a quite tall, smart person."

  The fuzzy calculation, however, would yield

"If Bob is very smart, and Bob is very tall, then Bob is a very tall, smart person."

- Another problem arises as more factors are incorporated into our equations (such as the fuzzy set of heavy people, etc.). The ultimate result of a series of AND's approaches 0.0, even if all factors are initially high. Fuzzy theorists argue that this is wrong: that five factors of the value 0.90 (let us say, "very") AND'ed together, should yield a value of 0.90 (again, "very"), not 0.59 (perhaps equivalent to "somewhat").

- Similarly, the probabilistic version of $A\ OR\ B$ is $(A + B - A * B)$, which approaches 1.0 as additional factors are considered. Fuzzy theorists argue that a sting of low membership grades should not produce a high membership grade instead, the limit of the resulting membership grade should be the strongest membership value in the collection.

The skeptical observer will note that the assignment of values to linguistic meanings (such as 0.90 to "very") and vice versa, is a most imprecise operation. Fuzzy systems, it should be noted, lay no claim to establishing a formal procedure for assignments at this level; in fact, the only argument for a particular assignment is its intuitive strength. What fuzzy logic does propose is to establish a formal method of operating on these values, once the primitives have been established.

4.4.3 Fuzzy Control

The first steps

The first paper describing a fuzzy logic controller was published by E.H. Mamdani and S. Assilan of Queen Mary College, England, in 1975. For their study, they chose the example of a simple steam engine. The controller for this engine has four input variables—pressure error, speed error, change in pressure error, and change in speed error—and two output variables—heat change and throttle change. The essential idea was strikingly simple. In the conventional PID (Proportion, Integral,

Derivative) controller, the system being controlled is modeled analytically by a set of differential equations whose solution tells what adjustments should be made to the system's control parameters for each type of system behavior. The proposed fuzzy logic controller, on the other hand, was based on a logical model which directly represents the thinking processes that a human operator might go through while controlling the system manually.

Such a logical model is expressed as a set of inference rules of the form "if behavior variable B (input to the controller) is observed to be in the state X, then change control parameter C (output from the controller) by an amount Y" (or perhaps to state Y). The model earns the designation "fuzzy" by virtue of its specifying the amounts X and Y linguistically, using terms like "positive big", "positive medium", "positive small", "no change", "negative small", etc., where each such term is represented as a fuzzy subset of the associated measurement domain.

This experiment, together with a few closely related experiments conducted by others, clearly demonstrated that this was an effective means of automated control. Indeed the logical models have a definite advantage over the traditional analytical models in that

1. they work well even when the relation between the controller's input and output variables is nonlinear, and

2. they are much more robust with respect to changes in the controlled system's parameters, e.g., the desired engine speed.

It is generally held that classical PID controllers cannot be designed for the case of nonlinear control and that, even for linear control, they typically must be designed anew whenever one resets the basic system parameters.

Mamdani's work generated interest in Japan, and approximately 10 years later, Hitachi Corporation announced the Sendai Railway as the first fully automated subway system employing a fuzzy logic controller. Hitachi had for many years been in the business of designing subway control systems, particularly safety mechanisms, and so this next step was a natural evolution of their existing product lines. The new system

controlled all aspects of accelerating to speed and braking for corners or stopping at the next platform, so that the only human operator served essentially as a conductor, watching out for passengers' safety while getting on or off the train.

Also in 1987 occurred another event, which, together with the Sendai Railway, served as the catalyst for an explosion of interest in the subject of fuzzy control. This was Takeshi Yamakawa's demonstration of his inverted pendulum experiment at the Second Congress of the International Fuzzy Systems Association (IFSA-87), held in Tokyo. The inverted pendulum is a classic control problem, amounting to balancing a vertical pole that is attached to a belt by a hinge, so that the pole can fall to the right or the left. The idea is to monitor the angular position and speed of the pole and move the belt to the right or left accordingly, so as to maintain the pole in an upright position.

There were a few negative responses. The latter stemmed from the fact that the controller only maintained vertical, and not horizontal, stability of the inverted pendulum, whereas the classical problem entails both. Moreover, it was shown rather easily that, with that particular system, accomplishing both was impossible. Hence Yamakawa suffered criticism for publishing results that were as yet incomplete.

Less than a year later, however, Yamakawa was able to vindicate himself by producing a system that performed both vertical and horizontal stabilization at the same speed as before. Since that time, Yamakawa has demonstrated the robustness of his system for nonlinear control by attaching a small platform to the top of the inverted pendulum, on which is then placed a wine glass filled with liquid, or even a live white mouse. The controller nicely compensates for the turbulence in the liquid, as well as the totally erratic movements of the mouse. Thus in the latter case, a claim could be made for executing control even beyond nonlinearity, and into truly random, or "chaotic" domains.

Commercializing fuzzy controllers

Before reporting these results, Yamakawa applied for patents on his chips in Japan, the US, and several European nations. He then proceeded to trade his patents to

several Japanese corporation in return for their subsidizing a laboratory in which he could continue his research. Omron, a major producer of second tier electronic devices, was a major proponent and has subsequently decided to invest heavily in fuzzy control. They have been rapidly expanding on Yamakawa's original designs, producing a host of new chips, both analog and digital, and churning out scores of applications. Due to their purchase of Yamakawa's patents, in fact, they have recently become the first Japanese corporation to ever obtain a US patent. As of July 1991, Omron boasted 700 patents for fuzzy logic devices either acquired, pending, or in application. Most of these devices either have appeared, or will appear, in commercial products. Three or four dozen alone are earmarked for use in automobiles, e.g., antilock brakes, automatic transmissions, impact warning and monitoring, windshield washers, and light dimmers. Omron's is also incorporating fuzzy control into products for use in industrial and manufacturing processes.

Numerous commercial products using fuzzy technology are currently available in Japan, and a few are now being marketed in the US and Europe. Canon uses a fuzzy controller in the autofocus mechanism of its new 8mm movie camera. The Matsushita/Panasonic "Palmcorder", currently being promoted on US television, uses fuzzy logic for image stabilization. This happens to be very the first video camera to appear with image stabilization capability. Each of Matsushita, Hitachi, Sanyo, and Sharp now have their own "fuzzy washing machine," which automatically adjusts the washing cycle in response to size of load, type of dirt (soil v.s. grease), amount of dirt, and type of fabric. Other products using fuzzy control include vacuum cleaners, air conditioners, electric fans, and hot plates. One senses that the possibility for such applications is virtually endless.

The future of fuzzy controllers

These few examples illustrate the variety of possible applications for fuzzy logic control. Japanese manufacturers are in fact now opting for fuzzy controllers even where conventional controllers would serve just as well. The reasons are that simple fuzzy logic controllers are much easier to design, require fewer electronic components, and

are therefore cheaper to produce.

The problem of how to design more complex controllers, however, has only recently met with what appears to be a practical solution. Typically the most difficult part of designing any fuzzy logic controller lies in selecting the fuzzy sets to use for the meanings of the linguistic terms appearing in the inference rules. As the number of rules grows large, the trail-and-error method of selecting the optimal collection of membership functions becomes less feasible. Somewhat of a breakthrough on this problem appears to have been achieved by Akira Maeda at Hitachi's System Development Laboratory. Maeda's idea is to use a form of neural net with back propagation to learn the needed membership functions from a set of training examples. As a test case, Maeda and his coworkers applied this technique to the development of a controller which had been designed previously by trail and error. Using this technique, they were able to accomplish in one month what had formerly taken six.

The possibilities for future work, leading to far more sophisticated logic-based controllers, are very clear. This movement will involve shifting from simple one-step rule-based systems to systems employing multi-step reasoning—i.e., rule chaining, together with the necessary truth maintenance systems—and which are integrated with other knowledge representation, reasoning, and learning schemes (e.g., semantic nets, frames, conceptual graphs, neural nets, and case-based reasoning). Taking the theory to this next stage will accordingly require progress in a number of important subareas before realizing the more advanced levels of automatic control.

Chapter 5

Case Study: Neurocontrol of a Cantilever Beam

Chapter 2 explained what artificial neural networks were and presented their different kinds. Chapter 3 gave a more thorough description of feed-forward networks, how they can be trained and how this training can be improved. On the other hand, chapter 4 shifted to control and studied the new advances in this field, one of which being neurocontrol, namely, control using neural networks.

This chapter 5 applies what has been presented so far to a simple element of civil engineering structure. Indeed, a cantilever beam supporting fluctuating loads, i.e. loads whose position on the beam as well as value change over time, is simulated.

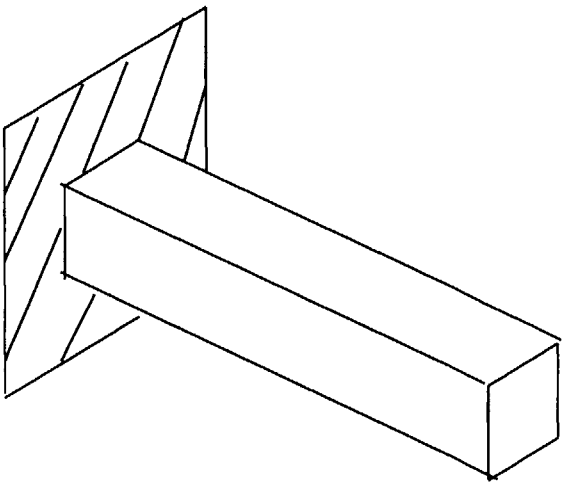
Firstly, section 5.1 presents the characteristics of the cantilever beam. Then, section 5.3 introduces MATLAB, the computer environment in which the simulations have been performed. Section 5.4 displays the scheme that has been adopted to control the beam. Finally, section 5.4.3 develops the simulations and explains their results.

5.1 The Beam Model

5.1.1 Describing the Beam Model

Figure 5-1 presents the chosen model of cantilever beam. As can be imagined, the

Figure 5-1: The simulated beam

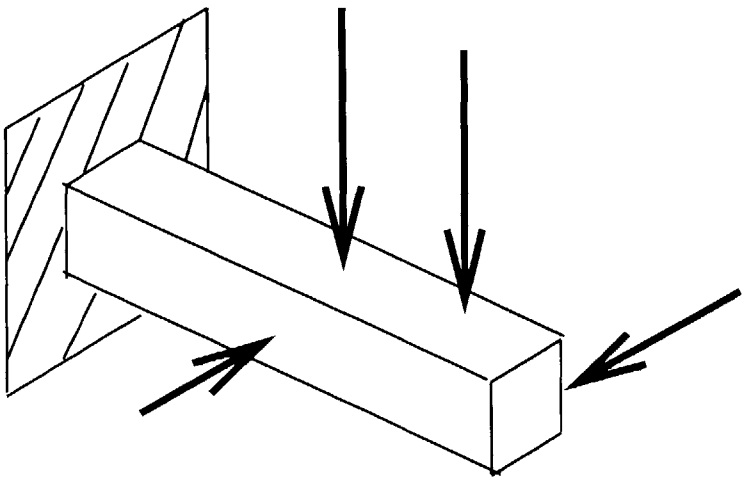


beam is embedded into a fixed support, so that neither displacements nor rotations are allowed at the junction between the beam and its support. The beam is composed of an homogeneous material, such as concrete or steel, so that the beam behavior is the same all along its length. Besides, the beam is in straight line and included in the plane perpendicular to the beam support.

In the simulation, forces are applied on this beam, as represented by Figure 5-2. Under the influence of these forces, the tip of the beam moves. However, at the free end of this beam is a device that can apply a force in any direction. Therefore, it counters the influence of the other forces, and keeps the beam tip stable.

The purpose of this case study is now to build a neural network that will control this free-end device. Assuming that the values and directions of the forces applied to the beam, and also some parameters proper to the beam, are input to this neural network, the latter should output a signal to the device. The device then translates this signal into a force value and a force direction, and exerts the specified force on

Figure 5-2: The simulated beam with forces



the beam. As a result, although the positions and values of the forces exerted to the beam change, the beam tip should remain in a restricted portion of space thanks to the conjugate influences of the neural network and the free-end device.

5.1.2 Limitations

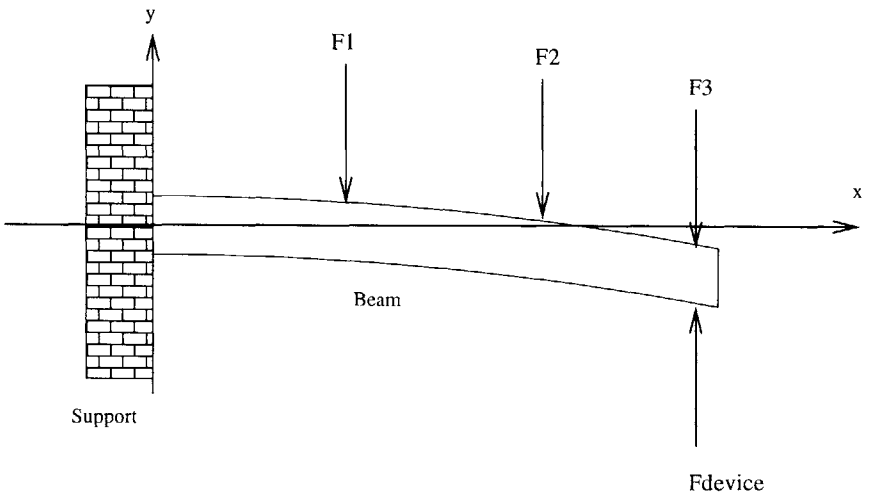
A good model should be close to the reality of the physical effect one wants to simulate, but also simple enough so that its study becomes easy. In the case of a cantilever beam, the following simplifications were chosen:

- The beam behavior is governed by the laws of elasticity, which are presented in subsection 5.2.
- Any effect other than mechanical is not considered in this model. For example, the consequences of dilatation when the beam heats up during its movements are not taken care of.
- The beam has a square cross section. This does not limit the model at all but simplify the calculations later. The area of this square cross section is noted $a \times a$. Besides, the beam section is constant.
- Only forces along the y axis are applied to the beam. This is a strong limit, but

the number of spatial dimensions involved in the problem needs to be reduced. Later, in another study, the three dimensions might be considered.

As a consequence, the only possible movement of the beam is a bending around the z axis. I do not consider other movements, such as torsion around the x axis. Figure 5-3 shows how the model looks like now.

Figure 5-3: The two dimensional model of a cantilever beam



- Except for the beam weight, which is a "distributed" force, all forces exerted on the beam are "concentrated" forces, i.e. forces localized in one point of the beam. For instance, magnetic forces that can influence a beam made of steel are not considered here.

Because of these limitations, the model has been greatly simplified, and, as will be shown in the next section, the beam behavior now depends on a restricted set of parameters only. These parameters are the following:

ρ Mass density of the material composing the beam.

E Young modulus of the material composing the beam.

μ Coulomb modulus of the material composing the beam.

l Beam length.

a Beam height (or Beam width).

F_d Force exerted by the free-end device.

F_i**'s and** X_i**'s** Forces supported by the beam, and their position along the beam.

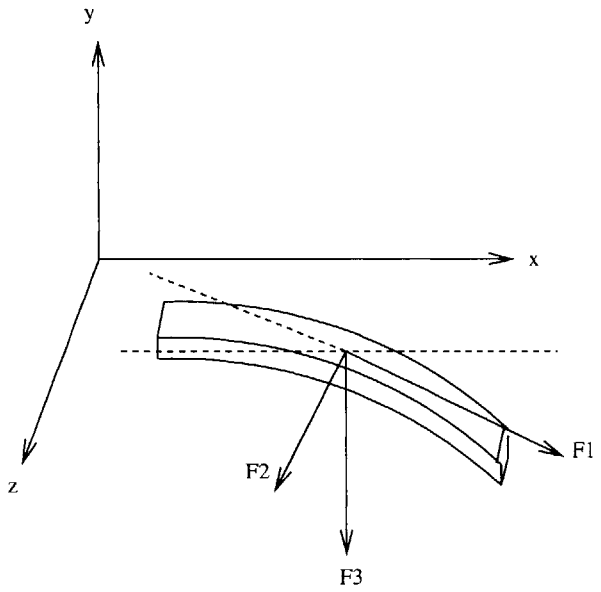
5.2 Theoretical Study

The purpose in this part of the memoir is to get an expression for the displacement of the beam tip when one exerts various forces on the beam. The development starts by giving the expression of this tip displacement as a function of the resultant force and moment at the beam tip. I then develop these resultant values into expressions involving the values of the actual forces supported by the beam, and therefore get a relation between the beam tip displacement and the exerted forces.

5.2.1 Plane beam

Consider a random beam, with the only important characteristic of being contained in a plane, namely, the plane $(\vec{x}, \vec{z})$ in Figure 5-4. In order to study the displacement

Figure 5-4: Studying a beam using the Frenet reference



and rotation of each section of this beam, two systems of reference are envisioned:

- A global one, represented by the $(\vec{x}, \vec{y}, \vec{z})$ vector system in Figure 5-4.
- A local one, represented by the famous Frenet triad, which, for any section of the beam, is composed by the vector $\vec{F_1}$ tangent to the beam curve, the vector $\vec{F_2}$ normal to the beam curve, i.e. pointing in the direction of the curvature center, and the binormal vector $\vec{F_3}$, which is just the vector product of the two previous vectors. The Frenet reference system is different for any beam section. In other words, the Frenet reference is a function of the curvilinear distance s. This variable represents the distance between the beginning of the beam and a specific point, but through the beam. The Frenet reference system of a beam section located at s is therefore noted $(\vec{F_1}(s), \vec{F_2}(s), \vec{F_3}(s))$.

Fortunately, the Frenet vectors can be expressed using the global vectors. Indeed,

$$\vec{F_1}(s) = \frac{dx}{ds}(s) \cdot \vec{x} + \frac{dz}{ds}(s) \cdot \vec{z} \tag{5.1}$$

$$\vec{F_2}(s) = -\frac{dz}{ds}(s) \cdot \vec{x} + \frac{dx}{ds}(s) \cdot \vec{z} \tag{5.2}$$

$$\vec{F_3}(s) = -\vec{y} \tag{5.3}$$

Note that $\vec{F_3}(s)$ is not, in fact, a function of s since the beam is contained in the plane $(\vec{x}, \vec{z})$. Given these two sets of vectors, the equations of Navier and Besse, from the theory of elasticity, give us the expressions of the displacement vector $\vec{u}(s)$ and the rotation vector $\vec{\varphi}(s)$ of each section of the beam. These equations are the following:

$$\frac{d\vec{\varphi}(s)}{ds} = D(s) \cdot \vec{F_1}(s) + E(s) \cdot \vec{F_2}(s) + F(s) \cdot \vec{F_3}(s) \tag{5.4}$$

$$\frac{d\vec{u}(s)}{ds} = \vec{\varphi}(s) \times \vec{F_1}(s) + A(s) \cdot \vec{F_1}(s) + B(s) \cdot \vec{F_2}(s) + C(s) \cdot \vec{F_3}(s) \tag{5.5}$$

As can be seen, these two vectorial equations introduce six variables A(s), B(s),..., and F(s). These six variables contain the specific parameters of the beam. Their

expression is:

$$A(s) = \frac{R_1(s)}{E(s)S(s)} \tag{5.6}$$

$$B(s) = \frac{R_2(s)}{\mu(s)S_2(s)} \tag{5.7}$$

$$C(s) = \frac{R_3(s) - \frac{M_1(s)}{r_c(s)}}{\mu(s)S_3(s)} \tag{5.8}$$

$$D(s) = \frac{M_1(s)}{\mu(s)J(s)} \tag{5.9}$$

$$E(s) = \frac{M_2(s)}{E(s)I_2(s)} \tag{5.10}$$

$$F(s) = \frac{M_3(s)}{E(s)I_3(s)} \tag{5.11}$$

All the variables introduced here represent specific properties of the beam. Note that they all depend on the curvilinear distance s since no hypothesis have been made about the shape of the beam yet. The only requirement so far is that the beam be contained in a plane.

All of these new variables require explanations:

- $(R_1(s), R_2(s), R_3(s))$ are the coordinates of the vector $\vec{R(s)}$ in the Frenet coordinate system. Vector $\vec{R(s)}$ is the resultant force for the beam section located at the curvilinear distance s.

- $(M_1(s), M_2(s), M_3(s))$ are the coordinates of the vector $\vec{M(s)}$ in the Frenet coordinate system. Vector $\vec{M(s)}$ is the resultant moment for the beam section located at s.

- $E(s)$ is the Young modulus of the material composing the beam section located at the curvilign absis s.

- $S(s)$ is the area of the section at s.

- $\mu(s)$ is the Coulomb modulus of the material composing the beam section at s. μ is given by the relation $\mu = \frac{E}{2(1+\nu)}$ where ν is the Poisson coefficient.

- $S_2(s)$ is called the "reduced section" and is used in the calculation of the elastic potential of the beam. Its expression will be given later in the development of this chapter. The 2 means that this section is computed in relation to the $\vec{F_2}$ axis. By the same token, $S_3(s)$ is the reduced section computed in relation to the $\vec{F_3}$ axis.

- $r_c(s)$ represents the curvature radius of the section at s.

- $J(s)$ is the torsion modulus of the section at s.

- $I_2(s)$ and $I_3(s)$ are the principal moment of inertia, in relation to, respectively, axis $\vec{F_2}(s)$ and axis $\vec{F_3}(s)$.

Although this seems like a very long series of variables to deal with, many of them will disappear when a straight beam is considered.

Then, the vectorial equations from Navier and Besse are developed into the global reference $(\vec{x}, \vec{y}, \vec{z})$, which yields six scalar equations:

$$\begin{aligned}
\frac{du_x}{ds} &= \varphi_y(s)\frac{dz}{ds} + A(s)\frac{dx}{ds} - B(s)\frac{dz}{ds} \\
\frac{du_y}{ds} &= \varphi_z(s)\frac{dx}{ds} - \varphi_x(s)\frac{dz}{ds} - C(s) \\
\frac{du_z}{ds} &= -\varphi_y(s)\frac{dx}{ds} + A(s)\frac{dz}{ds} + B(s)\frac{dx}{ds} \\
\frac{d\varphi_x}{ds} &= D(s)\frac{dx}{ds}(s) - E(s)\frac{dz}{ds}(s) \\
\frac{d\varphi_y}{ds} &= -F(s) \\
\frac{d\varphi_z}{ds} &= D(s)\frac{dz}{ds}(s) + E(s)\frac{dx}{ds}(s)
\end{aligned}$$

where $(\varphi_x(s), \varphi_y(s), \varphi_z(s))$ and $(u_x(s), u_y(s), u_z(s))$ are the coordinates of, respectively, the vectors $\vec{\varphi(s)}$ and $\vec{u(s)}$ in the global reference system.

5.2.2 Straight Beam

The beam considered in the simulations is a straight beam, say, along the x axis. Therefore, the curvilinear distance s represents now the same thing as the x coordi-

nate, and:

$$\begin{aligned} ds &= dx \\ dz &= 0 \\ r_c &= \infty \end{aligned}$$

As a result, the six previous equations become:

$$\begin{aligned} \frac{du_x}{dx} &= A(x) \\ \frac{du_y}{dx} &= \varphi_z(x) - C(x) \\ \frac{du_z}{dx} &= -\varphi_y(x) + B(x) \\ \frac{d\varphi_x}{dx} &= D(x) \\ \frac{d\varphi_y}{dx} &= -F(x) \\ \frac{d\varphi_z}{dx} &= E(x) \end{aligned}$$

Besides, the Frenet reference system is now constant along the beam, from a global point of view, and the following equalities are obtained:

$$\begin{aligned} \vec{F_1} &= \vec{x} \\ \vec{F_2} &= \vec{z} \\ \vec{F_3} &= -\vec{y} \end{aligned}$$

All variables expressed in relation to the Frenet reference can be expressed in relation to the global system, using the equalities above to switch from one to the other.

What is the influence of the model on the six variables $A(x)$, $B(x)$,..., $F(x)$? The forces exerted on the simulated beam are all along the y axis. This has two consequences:

1. Only coordinate 3, i.e. the y coordinate, of the resultant force $\vec{R}$ is not null.

2. Only coordinate 2, i.e the z coordinate, of the resultant moment $\vec{M}$ is not null.

This leads to:

$$\begin{aligned} A(x) &= 0 \\ B(x) &= 0 \\ C(x) &= \frac{R_3(x)}{\mu(x)S_3(x)} = -\frac{R_y(x)}{\mu(x)S_y(x)} \\ D(x) &= 0 \\ E(x) &= \frac{M_z(x)}{E(x)I_z(x)} \\ F(x) &= 0 \end{aligned}$$

The model added another set of requirements, expressed by the sentence:

"The behavior of the beam is the same all along its length."

This tells us that the quantities involved in the expression of the variables $C(x)$ and $E(x)$ that represent internal properties of the beam are, in fact, constant all along the beam. They are not functions of x. As a result, the expressions for $C(x)$ and $E(x)$ become:

$$\begin{aligned} C(x) &= -\frac{R_y(x)}{\mu S_y} \\ E(x) &= \frac{M_z(x)}{EI_z} \end{aligned}$$

Inserting these results back into the Navier and Besse equations yields:

$$\begin{aligned} \frac{du_y}{dx} &= \varphi_z(x) + \frac{R_y(x)}{\mu S_y} \\ \frac{d\varphi_z}{dx} &= \frac{M_z(x)}{EI_z} \end{aligned}$$

When looking for an expression of the displacement of the beam tip when the beam supports forces, the variable of interest is $u_y(l)$, where l is the length of the beam.

Given equations 5.12 and 5.12, it appears that:

$$\boxed{u_y(x) = u_M(x) + u_R(x)} \tag{5.12}$$

where

$$\boxed{\frac{d^2u_M}{dx^2} = \frac{M_z(x)}{EI_z}\ , and\ \frac{du_R}{dx} = \frac{R_y(x)}{\mu S_y}} \tag{5.13}$$

However, the beam tip displacement can be found using the resultant force and resultant moment for each section of the beam, but there is still no direct relation between the tip displacement and the exerted forces. This relation is unveiled in the following section.

5.2.3 Relation displacement—applied forces

The positions of the forces exerted along the beam will be given by X_i, for $i = 1$ to N, where N is the number of forces. The value of each one of these forces will be given by F_i. Given the convention adopted in the model, these forces can be represented by the vectors:

$$\vec{F_i} = \begin{pmatrix} 0 \\ F_i \\ 0 \end{pmatrix}\ ,\ for\ i\ =\ 1\ to\ N$$

The force applied by the free-end device will be noted:

$$\vec{F_d} = \begin{pmatrix} 0 \\ F_d \\ 0 \end{pmatrix}$$

The position of this last force is naturally l (beam length). Since these forces can be either upward or downward, F_d and F_i will be either positive (upward force) or negative (downward force).

Although the only forces applied to the beam are "concentrated" forces, meaning localized in one point, there is one "distributed" force that the beam sustains, namely,

its weight. If p denotes the weight of one unit section of the beam (one section of length 1),

$$p = \rho \cdot a^2 \cdot g \tag{5.14}$$

where ρ is the mass density of the material composing the beam, a^2 is the area of the beam section, and g is the gravity constant ($g = 9.81 N kg^{-1}$). As can be seen, p is not a function of x but a constant. This is logical since the beam keeps the same inner properties all along its length.

Given these notations, the expression of the resultant vectors is:

$$\begin{aligned} R_y(x) &= \left(\sum_{X_i>=x}^{X_i<l} F_i \right) + F_d - \int_x^l p \\ M_z(x) &= \left(\sum_{X_i>=x}^{X_i<l} F_i \cdot (X_i - x) \right) + F_d \cdot (l - x) - \int_x^l p(l-t)dt \end{aligned}$$

In other words,

$$\begin{aligned} R_y(x) &= \left(\sum_{X_i>=x}^{X_i<l} F_i \right) + F_d - (l - x)p \\ M_z(x) &= \left(\sum_{X_i>=x}^{X_i<l} F_i \cdot (X_i - x) \right) + F_d \cdot (l - x) - p\frac{(l-x)^2}{2} \end{aligned}$$

The above results can now be inserted into equations 5.12 and 5.13. This yields,

$$\begin{aligned} \frac{d^2 u_M(x)}{dx^2} &= \frac{1}{EI_z} \cdot \left[\left(\sum_{X_i>=x}^{X_i<l} F_i \cdot (X_i - x) \right) + F_d \cdot (l - x) - p\frac{(l-x)^2}{2} \right] \\ \frac{du_R(x)}{dx} &= \frac{1}{\mu S_y} \left[\left(\sum_{X_i>=x}^{X_i<l} F_i \right) + F_d - (l - x)p \right] \end{aligned}$$

The process of finding the expression of $u_y(l)$ from the above equations is tedious, and I will save it to the reader. In short:

1. one needs to integrate the previous equations to get the expression of $u_M(x)$ and $u_R(x)$. Adding the two together, one gets the expression of $u_y(x)$.

2. however, these integrations bring up some constants. The values of these con-

stants can be found by using boundary conditions. Indeed, since the beam is embedded into its support, the following applies: $u_y(0) = 0$ (no translation at the origin) and $u'_M(0) = 0$ (no rotation at the origin).

3. having $u_y(x)$, it is now very easy to get $u_y(l)$.

The whole computation leads to the following expression for $u_y(l)$:

$$u_y(l) = F_d l(\frac{l^2}{3EI_z} + \frac{1}{\mu S_y}) + \frac{1}{2EI_z} \sum_{X_i>=0}^{Xi<l} F_i X_i^2 (l - \frac{X_i}{3}) - \frac{pl^2}{2}(\frac{1}{\mu S_y} + \frac{l^2}{4EI_z}) \quad (5.15)$$

Three terms are distinguishable in this expression:

- $F_d l(\frac{l^2}{3EI_z} + \frac{1}{\mu S_y})$ represents the influence of the force F_d,
- $\frac{1}{2EI_z} \sum_{X_i>=0}^{Xi<l} F_i X_i^2 (l - \frac{X_i}{3})$ represents the influence of all the other localized forces F_i, and
- $-\frac{pl^2}{2}(\frac{1}{\mu S_y} + \frac{l^2}{4EI_z})$ represents the influence of the beam weight. Naturally, this term is negative since weight is a vertical downward force.

The expression of $u(l)$ can be simplified to make the parameter a (beam width or height) appear. Indeed, the following relations apply:

$$p = \rho a^2 g \quad (5.16)$$

$$S_y = \frac{5a^2}{6} \quad (5.17)$$

$$I_z = \frac{a^4}{12} \quad (5.18)$$

Relation 5.16 has already been presented. Relation 5.17 would need too much explanation about "reduced sections" and "elastic potential." The interested reader is invited to refer to the available documentation on structural engineering. Finally, the expression of I_z directly comes from the its definition, namely, $I_z = \int\int y^2 dxdy$. Inputing these relations in equation 5.15 yields:

$$u(l) = F_d l(\frac{4l^2}{Ea^4} + \frac{6}{5\mu a^2}) + \frac{6}{Ea^4} \sum_{X_i>=0}^{Xi<l} F_i X_i^2 (l - \frac{X_i}{3}) - \frac{\rho g l^2}{2}(\frac{6}{\mu 5} + \frac{3l^2}{Ea^2}) \quad (5.19)$$

As promised, the expression of the beam tip displacement now depends only on a restricted set of parameters. These are:

ρ The mass density of the material composing the beam.

μ The Coulomb modulus of the material composing the beam — $\mu = \frac{E}{2(1+\nu)}$, where ν is the Poisson modulus of the material composing the beam.

E The Young modulus of the material composing the beam.

l The beam length.

a The beam side.

Fd The force applied by the free-end device.

F_i's and X_i's The values of the vertical forces applied and their respective positions.

5.2.4 Simple Cases

It is hard to figure out the implications of formula 5.19 because of its length. Simple cases are helpful in managing the learning curve.

No Exerted Forces

In this case, only the free-end device is active. The beam tip displacement is then:

$$u(l) = F_d l(\frac{4l^2}{Ea^4} + \frac{6}{5\mu a^2}) - \frac{\rho g l^2}{2}(\frac{6}{\mu 5} + \frac{3l^2}{Ea^2}) \tag{5.20}$$

It can be seen that for the beam to remain horizontal, without the presence of any concentrated forces, the free-end device must already exert a force equal to:

$$F_d = \frac{\rho g l}{2} \frac{\frac{6}{5\mu} + \frac{3l^2}{Ea^2}}{\frac{4l^2}{Ea^4} + \frac{6}{5\mu a^2}} \tag{5.21}$$

Since the goal of this example is to get an approximate value for the free-end device force, one usual simplification can be made. Indeed, a closer look at the equation

reveals that some terms are negligible. Consider the numerator of the expression in parenthesis. Comparing the two fractions that form it yields:

$$\frac{\left(\frac{3l^2}{Ea^2}\right)}{\left(\frac{6}{5\mu}\right)} = \frac{5}{3}\left(\frac{\mu}{E}\right)\left(\frac{l}{a}\right)^2 = O\left(\left(\frac{l}{a}\right)^2\right) >> 1 \tag{5.22}$$

given the expression that links E and μ. In addition, the same conclusion can be drawn from comparing the fractions in the denominator. As a result, an approximate expression for F_d is:

$$F_d = \frac{\rho g l}{2} \cdot \frac{3a^2}{4} \tag{5.23}$$

For a regular concrete beam, the values of the parameters in the previous equations are well known.

- g is equal to $9.81 N kg^{-1}$
- ρ is approximately equal to $2,500 kg.m^{-3}$. This depends on the quantity of rebar in the concrete, but for an armed concrete, $2,500$ is the usual number.
- For a regular concrete, E can vary between $30,000\ MPa$ and $43,000\ MPa$. Above $43,000\ MPa$, this is the domain of high-performance concretes. A value of $35,000 MPa$ is chosen in the example.
- It can be inferred from the relation that links E and μ that μ is also going to vary depending on the quality of concrete. However, the Poisson coefficient ν is relatively stable for concrete, and, usually, $\nu = 0.2$. Given the value chosen for E, $15,000 MPa$ is the value obtained for μ.

And if a beam with the following dimensions is considered:

$$l = 5m \tag{5.24}$$

$$a = 0.2m \tag{5.25}$$

then,

$$F_d \approx 1,840\ N \tag{5.26}$$

One Downward Force

In this case, and if the approximation of the previous section is still considered, expression 5.19 becomes:

$$u(l) = \frac{4F_d l^3}{Ea^4} + \frac{6F_1 X_1^2}{Ea^4} \cdot \left(l - \frac{X_1}{3}\right) - \frac{\rho g l^2}{2} \cdot \frac{3l^2}{Ea^2} \tag{5.27}$$

where F_1 is negative. If the free-end device is set up to keep the beam horizontal, this yields:

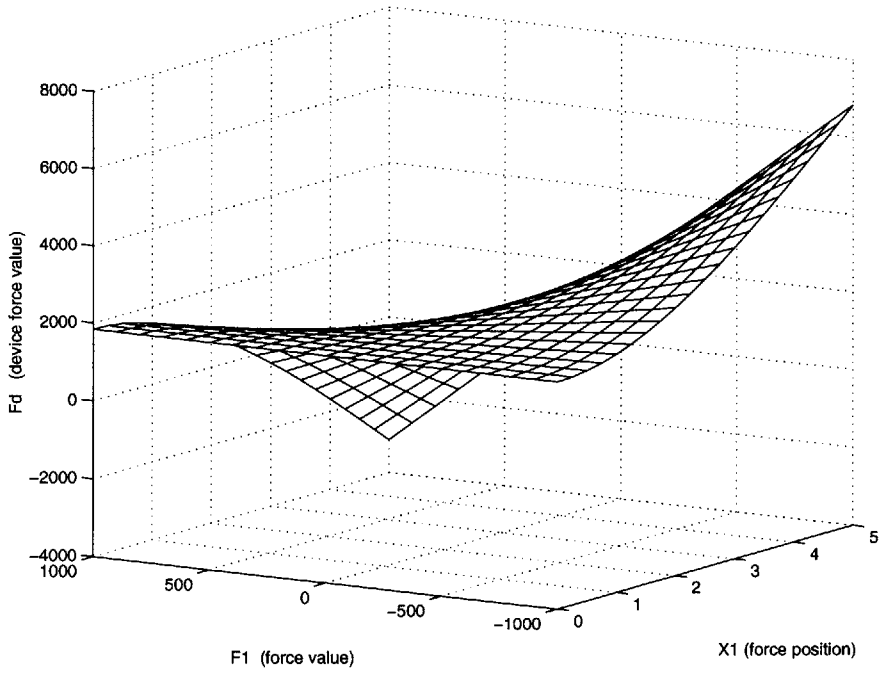
$$F_d = \frac{3\rho g l a^2}{8} - \frac{3F_1 X_1^2}{2l^3}\left(l - \frac{X_1}{3}\right) \tag{5.28}$$

Keeping the same values as the ones given in the previous paragraph for the beam parameters, this expression collapses into:

$$F_d = 1,840 - 0.06F_1 X_1^2\left(5 - \frac{X_1}{3}\right) \tag{5.29}$$

Figure 5-5 presents F_d as a function of X_1 and F_1.

Figure 5-5: $Fd = f(X_1, F_1)$



As can be seen, when F_1 is located at the very beginning of the beam, very close to the beam support, its influence is negligible. However, the greater X_1, i.e. the closer F_1 to the beam tip, the greater its influence, and, therefore, the greater the required action F_d of the free-end device. Naturally, the action of the free-end device counters the one of F_1, which is why, for a specific position X_1, the more negative F_1, the more positive F_d.

5.3 Using the MATLAB Environment

The previous section represented the theoretical foundation for neural network simulations. However, before performing the latter in section 5.4, the MATLAB environment, i.e. the environment in which artificial neural networks are built and trained, is introduced.

If one browses the Internet today, one will find more than a hundred programs that offer neural network simulation possibilities. This goes from the limited (but often free) simulators written by graduate students working on their research, on to the package added to an existing mathematics-oriented program, and finally to the comprehensive (but often costly) independent piece of software.

5.3.1 The MATLAB Neural Network ToolBox

MATLAB, developed by people at The MathWorks, is now a famous mathematics-oriented program. Its Neural Network Toolbox belongs to the second aforementioned category. According to The MathWorks:

> "The Neural Network Toolbox is a powerful collection of MATLAB functions for the design, training, and simulation of neural networks. It supports a wide range of network architectures, with an unlimited number of processing elements and interconnections (up to operating system constraints)."

Although the MATLAB Neural Network Toolbox may not be the most attractive (in terms of user interface) package, it is comprehensive and was, above all, freely available from any station connected to the MIT network.

The following section describes the architecture of the MATLAB Neural Network Toolbox. One needs a clear picture of the MATLAB implementation of neural networks to know what is simulated, and which results mean what. Besides, having this clear picture in mind makes it possible to customize one's own neural network(s) without getting lost in a jungle of parameters, and therefore to fully benefit from the power of the MATLAB functions.

For MATLAB people, a neural network is an object with components and parameters. This object can be passed to MATLAB functions that will alter its components and parameters, and so perform the simulation. By functions, it is meant:

- building functions, which actually reserve space for the network object in the memory of the computer;
- initializing functions, which initialize and set the network components and parameters;
- training functions, which really change the components and parameters of the network object; and
- displaying functions, which graphically display the state of the network at any time in the simulation process.

Readers familiar with object-oriented programming might have recognized here the description of classes and methods. Indeed, The MathWorks selected an object-oriented approach and, in that sense, their Neural Network Toolbox closely resembles a C++ library. The following section describes the classes and variables involved in the composition of a MATLAB neural network.

MATLAB classes

Figure 5-6 provides an overview of the components and parameters that form a neural network object. Since this memoir is not geared at object-oriented programming gurus in the first place, some paragraphs are devoted to the explanation of this figure.

First of all, the following notation is considered:

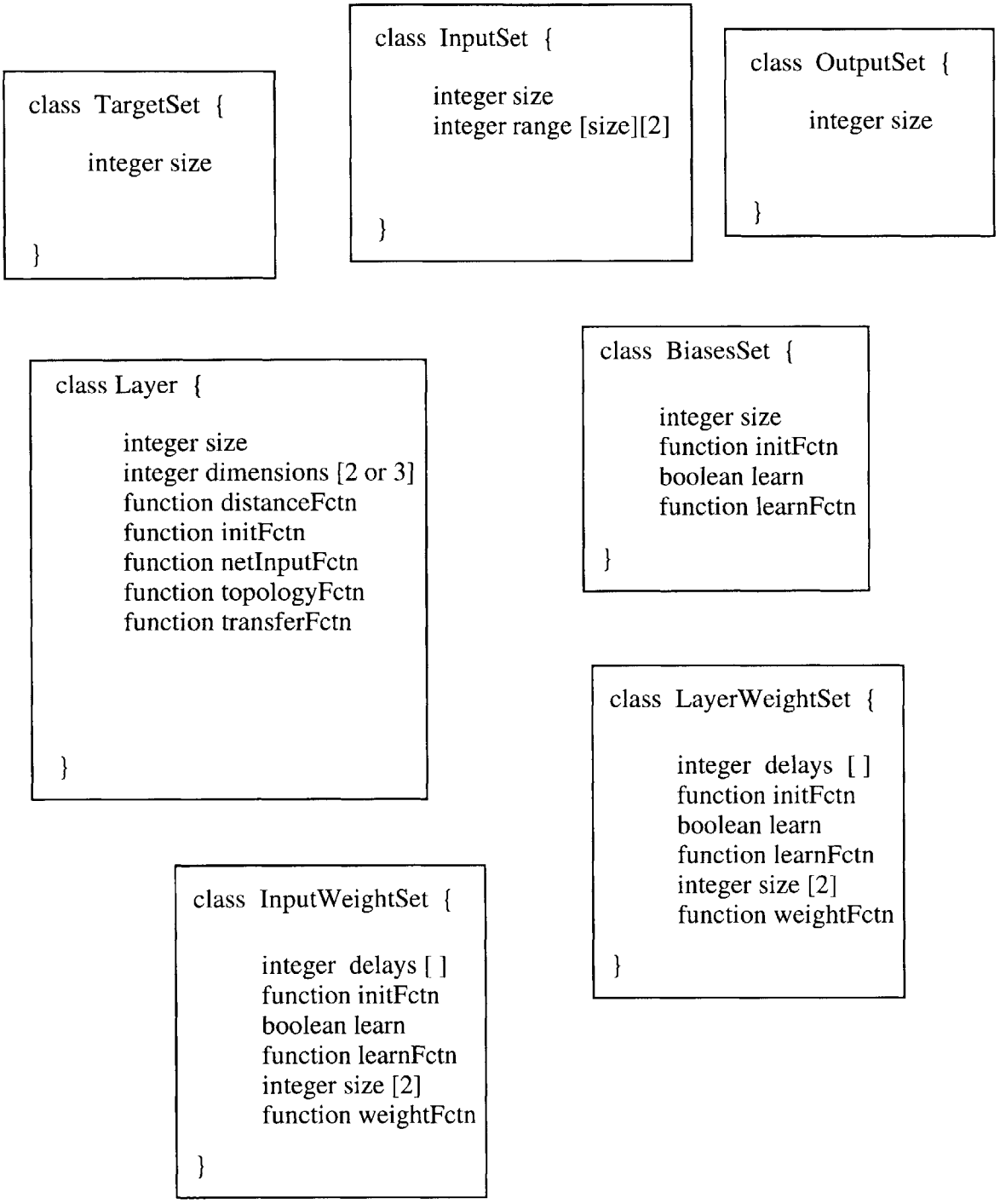
- "class" signals the beginning of an object definition. The next word after "class" is actually the name of this object.
- Typically, variables in a class are declared using a sentence like *datatype variableName*. If brackets ([*dim*]) follow, this variable is actually an array of *datatype*, of dimension *dim*. If there are two series of brackets, this is a 2-dimensional array.
- "unsigned integer" or "integer" both mean an integer greater than zero in the context of this memoir.
- "double" means, roughly, a decimal number.
- "boolean" is the datatype of a variable that can only take true or false as a value. In MATLAB, true is represented by 1, and false is represented by 0.
- "function" is a generic datatype for methods, i.e. black boxes getting input parameters and outputting some results.

Figure 5-7 presents additional MATLAB classes that fall into the composition of a MATLAB neural network.

Figure 5-6: The MATLAB class "Network"

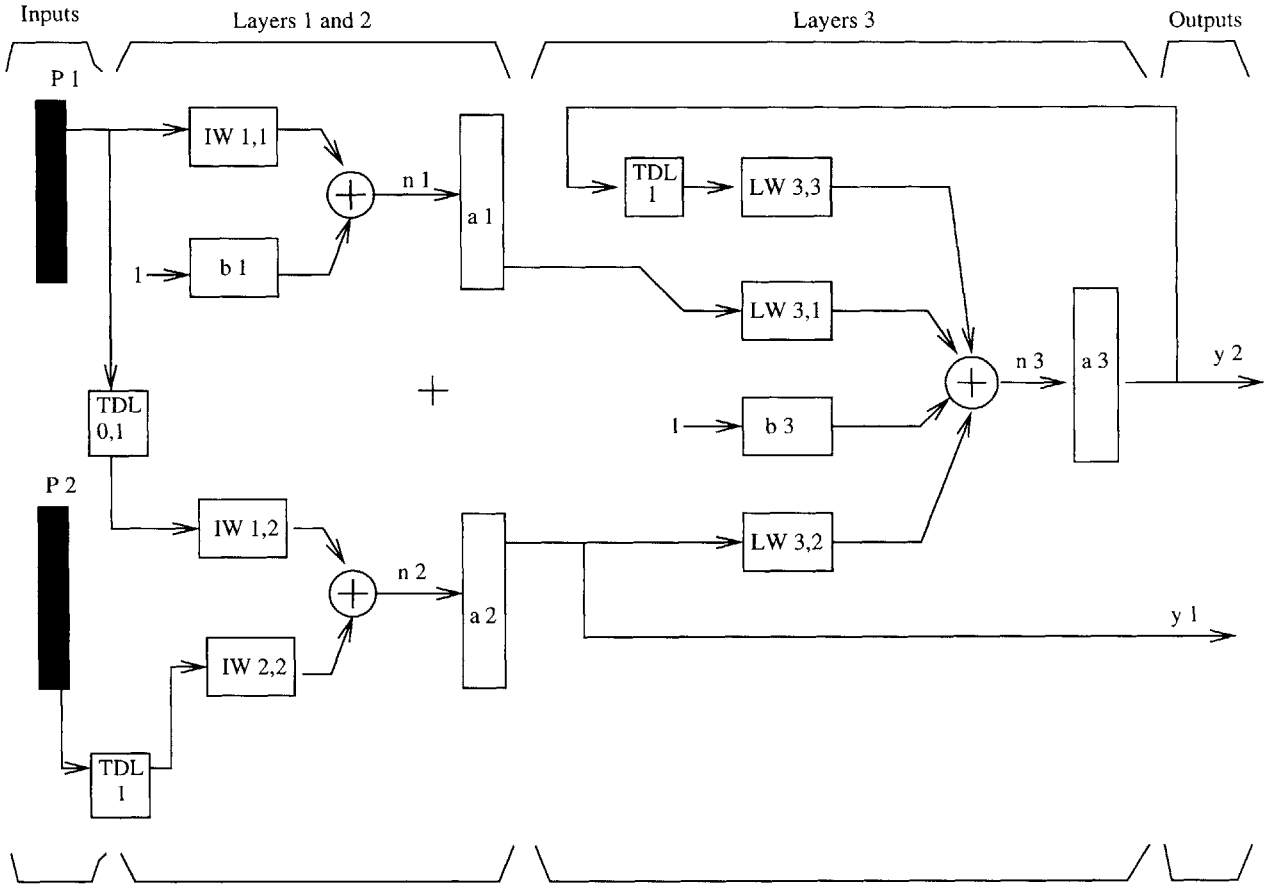
```
class Network {

        unsigned integer numInputs
        unsigned integer numLayers

        boolean biasConnect [numLayers]
        boolean inputConnect [numLayers][numInputs]
        boolean layerConnect [numLayers][numLayers]
        boolean outputConnect [numLayers]
        boolean targetConnect [numLayers]

        unisgned integer numOutputs       *
        unsigned integer numTargets       *
        unsigned integer numInputDelays   *
        unsigned integer numLayerDelays   *

        function adaptFctn   ==> double adaptParam
        function initFctn  ==> double initParam
        function performFctn  ==> double performParam
        function trainFctn  ==> double trainParam

        InputSet  inputs [numInputs]
        Layer  layers [numLayers]
        OutputSet  outputs [numLayers]
        TargetSet  targets [numLayers]
        BiasesSet  biases [numLayers]
        InputWeightSet  inputWeights [numLayers][numInputs]
        LayerWeightSet  layerWeights [numLayers][numLayers]

        doubleMatrix  IW [numLayers][numInputs]
        doubleMatrix  LW [numLayers][numLayers]
        doubleVector  b [numLayers]

}
```

Figure 5-7: Additional MATLAB classes



Before explaining these various classes, some insights about how complex a neural network can be are gained. Figure 5-8 gives an idea of a quite complicated piece of network that can be built using the MATLAB Toolbox.

Figure 5-8: A complicated example of MATLAB network



As can be seen:

- An artificial neural network can have several input sets. These may be linked to different layers in the network.
- Each layer may or may not have a set of biases.
- There can be zero of many feedback loops in the network.
- Some of these feedback loops may involve what is called "Taped Delay Lines (TDL)." Basically, the delay is the amount of time between the moment an

output is collected and the moment it is put back into the network. The greater the delay, the later the network is fed back with the output.

- There can be more than one set of output values. Accordingly, there can be more than one set of target values for a neural network.

Because an artificial neural network can be so complex, its theoretical representation is not an easy task. This is why the MATLAB implementation of artificial neural networks requires the use of all the above classes, whose description is given below:

class InputSet This class gathers the properties (not the values) of a set of inputs to the network. These properties are:

- The *size* of this input set, namely, the number of input values.
- The *ranges* of each input value, given by a *size* $\times$ 2 array of doubles. The two values in each row of the array represent the lower and upper limits of the range of the associated input value.

class Layer This class gathers the properties of a network layer. These properties are:

- The number of neurons in this layer, given by the variable *size*.
- The physical dimensions, i.e. the number of neurons along the x, y, and z axis in a 3-dimension space. This property is represented by an array of 3 integers, each integer representing the number of neurons along one axis.
- The function used to calculate distances between neurons in the layer (*distanceFctn*). These last two properties concern specific networks that won't be used in this study.
- The function (*initFctn*) used to initialize the bias values and weights of these layer. This function only calls, in its way though, the *initFctn* of all its bias sets and weight sets.
- The function (*netInputFctn*) used to compute the net input of each neuron in this layer.

- The function used to calculate a neuron's position given the layer dimensions (*topologyFctn*).
- The transfer function (*transferFctn*) of this layer, which is the function used to compute the output from the net input. MATLAB provides different possibilities for all three functions (init, netInput, transfer).

class TargetSet and OutputSet These classes bundle up the properties (not the values) of a set of, respectively, targets and outputs. These properties are:

- The *size* of this target or output set, namely, the number of target or output values.

class BiasesSet This class gathers the properties (not the values) of a set of biases associated with a network layer. These properties are:

- The *size* of bias set, namely, the number of bias values.
- The function (*initFctn*) used to initialize these bias values. MATLAB provides, among others, two interesting initialization functions,namely, *initzero*, which initializes all bias values to 0, and *rands*, which give biases random values.
- A boolean (*learn*) whose value tells if these bias values are updated during the learning process ($learn = true$) or not ($learn = false$).
- If *learn* equals true, then *learnFctn* describes the function used to update these bias values. MATLAB also provides an extensive set of learning function for biases, which are described later in this chapter.

class InputWeightsSet and LayerWeightSet These classes gather the properties (not the values) of a set of, respectively, input weights and layer weights. Whereas input weights link an input set to a network layer, layer weights represent the junction between two network layers. These properties are:

- An array of integers (*delays*) representing the Tape Delay Line (TDL) associated with these weights.

- The function (*initFctn*) used to initialize the weights.
- A boolean (*learn*) whose value tells if these weights are updated during the learning process ($learn = true$) or not ($learn = false$).
- If *learn* equals true, then *learnFctn* describes the function used to update these weights. There are more than ten possible learning function for input or layer weights.
- The dimensions of the matrix holding the actual weight values. These dimensions are given by *size*, an array of two integers.
- The function *weightFctn*, which is used in the computation of a neuron net input. Indeed, a neuron net input is composed of the bias and another element, which depends on this neuron weights and input values. This other element is the result return by *weighFctn*.

As the reader may have guessed now, these properties are tightly intertwined with one another. For instance, the size of an OutputSet is naturally equals to the size of the corresponding output Layer. When building an artificial neural network with MATLAB, one only specifies some properties. The others are either deduced from what is provided, or assigned default values.

In chapter 3, the neural networks used in civil engineering applications were presented. Given this kind of neural networks, a value can already be assigned to many properties in each one of the above classes. The network parameters will be initialized randomly, using the 'rands' function. Besides, the net input of each layer is the sum ('netsum' in class Layer) of the biases and the product ('dotprod' in class InputWeightSet and LayerWeightSet) of the inputs with the weights. Figure 5-9 sums up the choices made so far.

Figure 5-9: Initializing MATLAB classes for civil engineering networks

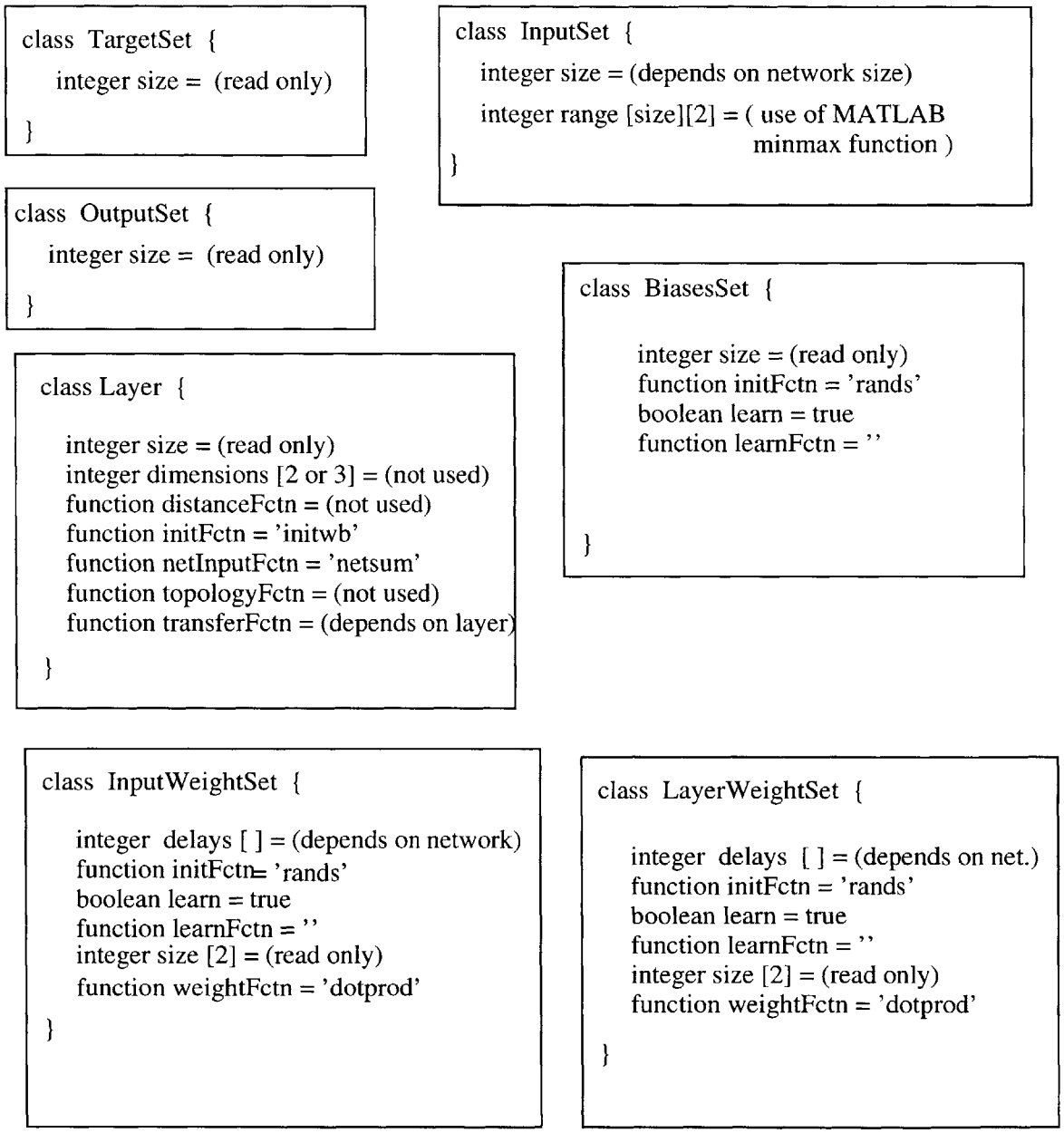


Figure 5-6 is now considered again. Many variables in this class are self-explanatory.

numInputs and *numLayers* give, respectively, the number of inputs and the number of layers in this network.

The *xxxConnect* variables describe the connection between inputs, layers, outputs, biases, and targets.

The following *numXXX* variables are read-only and cannot therefore be customized.

The following four variables are very important. They hold the different function that the network is going to use during the simulation process:

- *adaptFctn* is the function used during sequential training.
- *initFctn* describes the method used to initialize the network,
- *performFctn* is the function used to measure the network performance, and
- *trainFctn* is the function used during batch training, which is the training method used in the simulation.

Then come the different objects composing a network, along with their own properties.

Finally, *IW*, *LW*, and *b* hold the values (and not the properties this time) of the weights and biases of the whole network. MATLAB use matrices of matrices to store these values.

5.3.2 An Example: Comparing Training Algorithms

This first use of the MATLAB environment is to be linked with section 3.1.2 and section 3.1.3 of this thesis, where better alternatives to the traditional back-propagation algorithm were presented.

In this section 5.3.2, the goal is twofold:

1. The MATLAB Neural Network Toolbox is used to create a neural network that models a straight cantilever beam supporting one force and the force of the free-end device. The desired network output is the beam tip displacement.

2. To train this neural network, the different training methods presented in section 3.1.3 are used, and their speed and efficiency are compared.

When a straight cantilever beam supports one force and the free-end device force, the displacement of its tip is given by formula 5.30

$$u(l) = F_d l(\frac{4l^2}{Ea^4} + \frac{6}{5\mu a^2}) + \frac{6}{Ea^4}F_1 X_1^2(l - \frac{X_1}{3}) - \frac{\rho g l^2}{2}(\frac{6}{\mu 5} + \frac{3l^2}{Ea^2}) \qquad (5.30)$$

The program "simulation1" presented in appendix A.1 is a MATLAB snippet of code that repeatedly calls the MATLAB function "beam" with different training functions. This function "beam", which is presented in appendix A.2, creates a network that simulates a beam, and train it with the specified training function. "beam" itself makes use of functions in the MATLAB Neural Network Toolbox.

As can be seen in the code, the network is provided with 3 inputs, namely, X_1, F_1, and F_d. F_1 is supposed to be always negative for sake of simplification. This feed-forward network is composed of 2 layers. The first one contains 12 neurons with a sigmoid function as transfer function. The second (output) layer is composed of only one neuron, with a linear function as transfer function. All parameters of this network are initialized with random values. The performance of the network is measured by the error function described in section 3.1.2, called 'mse' in MATLAB. As can be seen in the code of function "beam," the goal assigned to the network is to get this error down to 10^{-6}. To do so, the network can make use of one of the nine following training functions:

traingd which uses gradient descent back-propagation, i.e. the initial back-propagation algorithm describes in section 3.1.2 .

traingda which uses gradient descent back-propagation with variable learning rate.

traingdm which uses gradient descent back-propagation with momentum.

traingdx which combines gradient descent back-propagation with momentum and variable learning rate. traingda, traingdm, and traingdx have been presented in section 3.1.3, sub-section "Steepest/Gradient Descent."

trainbfg which uses a specific kind of quasi-Newton back-propagation, whose general concept was presented in section 3.1.3, sub-section "Newton's and Quasi-newton's methods."

traincgb,traincgf, and traincgp which use three different kinds of the same algorithmic concept, namely conjugate gradient back-propagation, an overview of which was given in section 3.1.3, sub-section "Conjugate Gradient Algorithms" of this memoir.

trainlm which implements a back-propagation that makes use of the Levenberg-Marquardt algorithm presented in section 3.1.3, sub-section "The Levenberg-Marquard Algorithm" of this memoir.

Each run of the "beam" function creates a network trained using a different algorithm. As a result, some training procedures take more time to reach the assigned goal of 10^{-6}. Table 5.1 compares these different training procedures. In this table, "NUMBER OF EPOCHS" represents the number of epochs it took the network to reach the performance goal. A limit of $1,000$ epochs is imposed, to allow the simulation to run in a reasonable amount of time. When a training procedure could not make the network reach the goal within $1,000$ epochs, the performance was still recorded in the column "ERROR REACHED." The column "GOAL MET" simply assets the network performance from the previous column.

All the values in this table are in fact averages, since the simulation was run 10 times. In the table above, training methods have been gathered by the algorithm they use. As can be seen:

- Methods directly improving from the back-propagation algorithm perform poorly. They take an enormous amount of time or number of epochs to reach a (not

Table 5.1: Training Performances

TRAINING ALGORITHM		NUMBER OF EPOCHS	ERROR REACHED	GOAL MET ?
Back-propagation improvements	traingd	1,000	$6.8e^{-03}$	NO
	traingda	1,000	$5.8e^{-04}$	NO
	traingdm	1,000	$8.1e^{-03}$	NO
	traingdx	1,000	$1.5e^{-04}$	NO
Quasi-newton	trainbfg	359	$9.9e^{-07}$	YES
Conjugate Gradient	traincgb	333	$6.6e^{-06}$	NO
	traincgf	258	$1.2e^{-05}$	NO
	traincgp	398	$1.2e^{-05}$	NO
Lev.- Marquardt	trainlm	22	$9.2e^{-07}$	YES

so low) goal. In the simulations, all these methods could not reach the goal of $10e^{-6}$ they were assigned, and their performance after $1,000$ epochs was about $10e^{-3}$. Note that the introduction of a variable learning rate or momentum does not do much.

- The quasi-newton method used in these simulations performed quite well, since, on average, it reached the goal of $10e^{-6}$ after 359 epochs.

- The behavior of the conjugate gradient methods was different in these simulations. On average, they did not manage to reach the assigned goal, not because the number of epochs was too small, but because the algorithm could not reduce the error any more after about 300 epochs. $1.2e^{-5}$ or $6.6e^{-6}$ were the minimum error that these methods were able to reach. Note that these are not far from the required goal of $10e^{-6}$ though.

- The Levenberg-Marquardt algorithm is, from far, the best algorithm to train this kind of feed-forward network, since it results in the fastest and most precise training procedure.

This series of simulation shows what a breakthrough the introduction of the Levenberg-Marquardt algorithm was in the field of neural network training. In the simulations, it cut the required number of training epochs from 359 (if one chooses

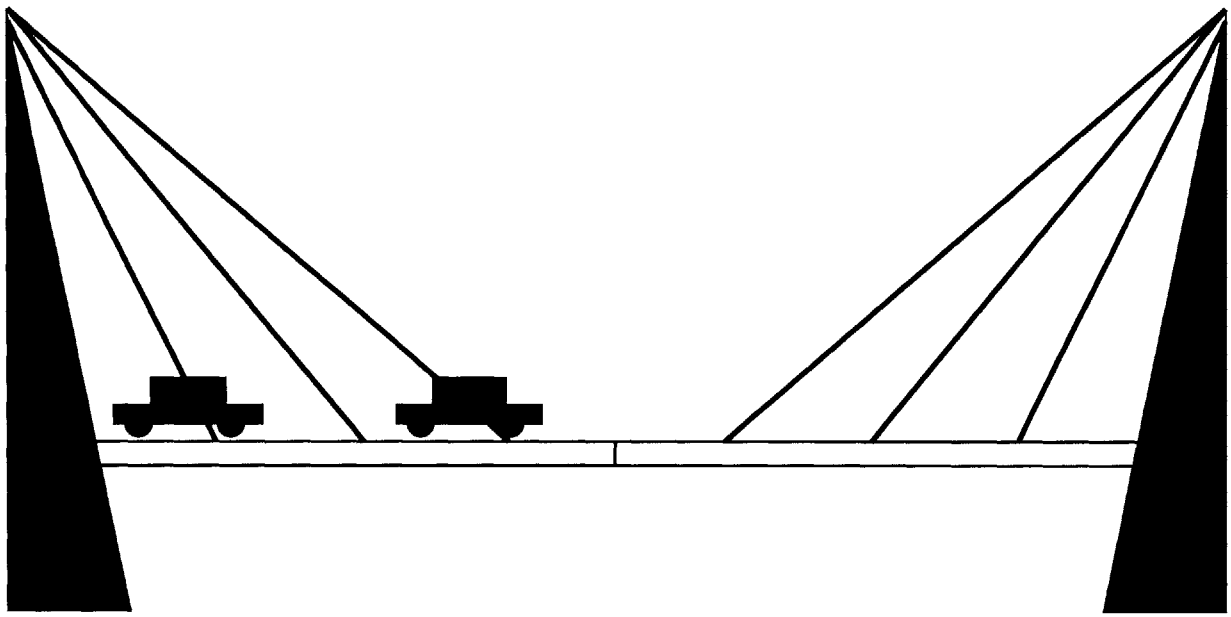
the next successful method) to 22. This is why, in the remaining simulations of this memoir, in section 5.4.3, "trainlm" is used extensively.

5.4 Control scheme and Simulations

5.4.1 Real model

It is now time to implement a real control scheme using an artificial neural network. Imagine one span of a cable-stayed bridge as represented by figure 5-10. Consider

Figure 5-10: A cable-stayed bridge



now one of the horizontal slabs supported by the cables. Two kinds of forces influence the position of this slab:

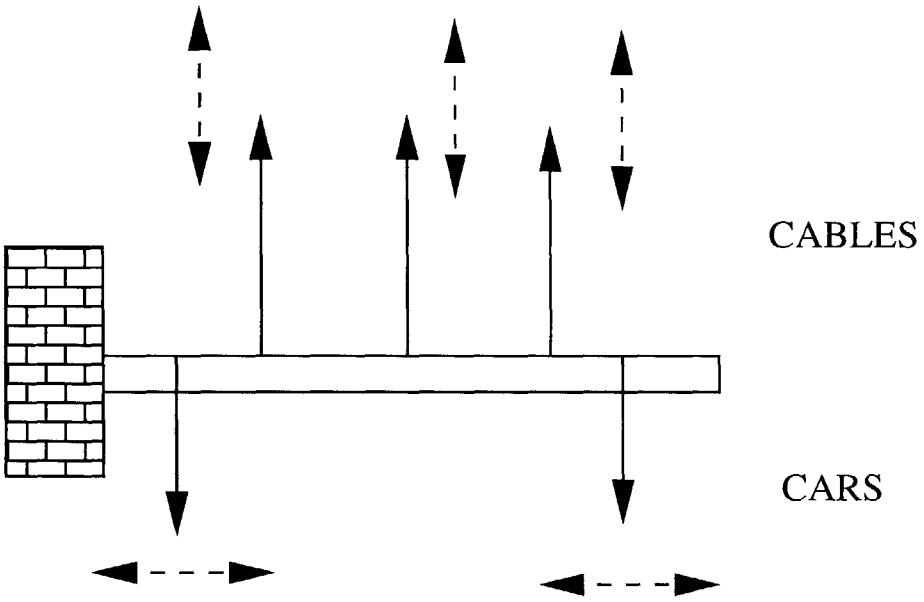
- upward forces, exerted by the cables. The position of these forces is constant, but their amplitude is not, since the wind, the rain, or some other weather conditions can make the cables sway, and therefore, can increase or decrease the tension they exert.

- downward forces, exerted by the cars or trucks driving on the bridge. The forces

have a constant amplitude, as the mass of a car is constant, but their position changes rapidly.

Consider the very simple model presented by Figure 5-11: The numerous slabs com-

Figure 5-11: Model of a cable-stayed bridge



posing the horizontal element of a bridge are so tightened together that the whole set of slabs is usually considered as one single beam. Therefore, the model in Figure 5-11 can be useful in the study of cable-stayed bridges.

Now, if sensors are provided to monitor the different loads on the bridge, and actuators, to move the slab tip up or down, one wants a device that controls these actuators from the information collected by the sensors. The hypothesis is made that this device can be a neuro-controller, and such a controller is built in the next paragraph.

5.4.2 Neurocontrol scheme

The closed-loop control scheme developed in section 4.3.3 is chosen. As a result (see Figure 5-12):

- First, a neural network that models a concrete beam supporting the kind of loads described in the previous section is built. A 2-layer feed-forward network should do the job, as it has been seen that such networks, trained by the Levenberg-Marquardt algorithm, can reach good performances. This model network is created in the first part of the code presented in appendix B.2.

- Then, a 4-layer network is created. The first two layers of this network are the ones to be trained. The last 2 layers are filled in using the model network parameters and will not changed during the training process. The inputs to this total network are the loads and their positions on the beam. Its output is the displacement of the controlled cantilever beam, which is targeted at 0. The back-propagation of the training error represents the feedback part of the closed-loop control scheme.

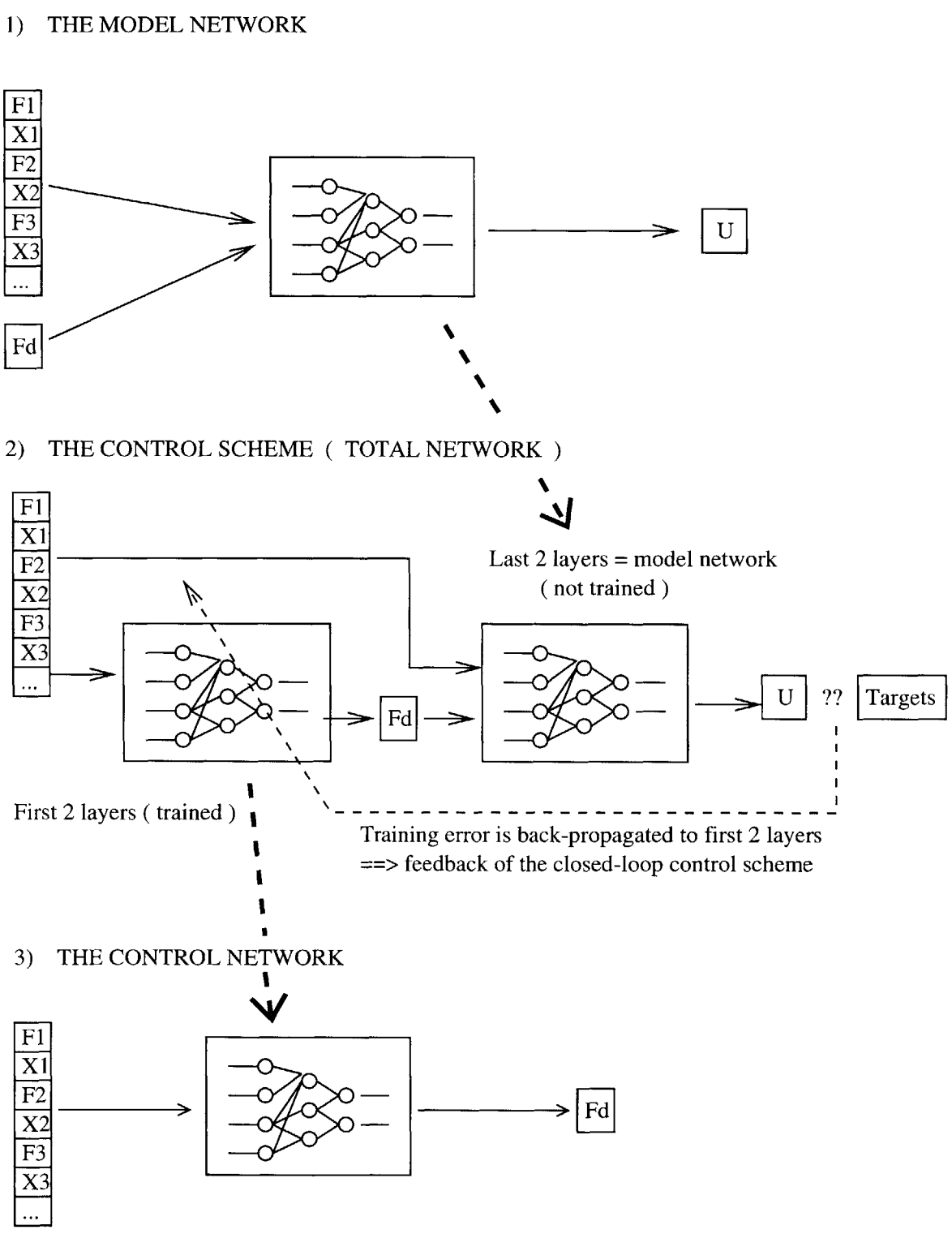
- Once the training is over, the first two layers of the previous 4-layer network are transferred in a 2-layer controller network, which is now a controlling device for the cantilever beam.

- This controlling device can finally be tested by inputing loads to the controlled beam, and see how the beam reacts. This is done in the last part of the program in appendix B.2

5.4.3 Simulation and Results

The program "simulation2" in appendix B.1 performs the aforementioned simulations. It calls the MATLAB function "beam_control()" several times and records its output. This latter function simulates the entire process of building a control scheme, from the model network creation to the control network testing. Note that the different loads and their positions, as declared in the first few lines of the code for "beam_control()," match the influences of cables and cars on the beam. After the control network testing, the "beam_control()" function returns the minimum and maximum values of

Figure 5-12: The control scheme



the tip displacement of the controlled beam. Program "simulation2" records these minimum and maximum values for each run of the "beam_control()" function.

In other words, each run of the beam_control() function creates and customizes a new randomly-initialized neural network. The control efficiency of the latter is then tested by inputting some loads and positions, and by getting the corresponding tip displacements. "simulation2" records the minimum and maximum among these controlled tip displacements.

The output of the program "simulation2" was the following:

Table 5.2: Minimum and maximum values of the beam tip displacement

RUN	MINIMUM VALUE	MAXIMUM VALUE
1	-0.014	0.024
2	-0.016	0.022
3	-0.024	0.013
4	-0.021	0.017
5	-0.027	0.014
6	-0.027	0.018
7	-0.023	0.021
8	-0.007	0.015
9	-0.011	0.008
10	-0.022	0.003
AVERAGE	-0.019	0.016

As can be seen in table 5.2, the control network now manages to keep the beam tip not farther than 2 centimeters from the horizontal line. Considering the length of the simulated beam, namely, 5 meters, it can be deduced than this neurocontroller does a very good job.

5.4.4 Discussion

The previous simulations confirmed that artificial neural networks can be used to control a simple element of structure. However, there is already a plethora of controllers out there that can perform this same task, and their architecture is not based on neural networks. So, what is the advantage of neurocontroller upon those ?

The key advantage of neurocontroller is adaptability.

Indeed, every structure is by essence a mutable edifice. Year after year, materials composing the structure deteriorate and their properties change. Under the influence of creep for instance, even the dimensions of the structure can be altered. Standard controllers can not compensate for these deteriorations and the risk is that they become out-of-date, and do not control the structure for which they were designed any more. What structures need is a "self-adaptive" controller, one that alters its parameters to finely tune the control action to the structure condition.

The idea is not far that neural-networks could be used successfully to build this kind of controllers. To see why, consider the simple example of the cantilever beam. The flow of signals in the monitoring process of the beam is the following: sensors connected to the beam transmit a picture of the structure condition to the controller. If necessary, the controller chooses a course of action to correct this condition. This is characterized by a flow of signals sent by the controller to some actuators having an influence on the structure.

It is not far-fetched to think of the following process. If one adds a sensor that measures the position of the beam tip in relation to the designed position of equilibrium, the neurocontroller now has the possibility to evaluate its own performance. One can now imagine that if this performance is not within a pre-designed range for a specified number of controlling cycles, the neurocontroller switches to learning mode. In other words, the neurocontroller now adapts its parameters during each controlling cycle. Given the performance of the Levenberg-Marquardt algorithm highlighted in previous sections, this update could be very fast. Then, when the error reaches again an acceptable level for a specified number of controlling cycles, the controller could switch back to control-only mode.

As a result, the implementation of a neurocontroller would be done along the following pattern:

1. An artificial neural network is build and trained to simulate the beam.

2. Another network is randomly initialized and connected to the former to act as

a controller.

3. This second network is switched to learning-mode for the first time, so that it learns how to control a perfect beam.

4. Then the control network is connected to the real beam, and some error range and re-learning period numbers are plugged into its memory.

5. Time goes by. The controller works well.

6. Eventually the beam deteriorates so much that the neurocontroller switches back to learning mode to get a clearer picture of the controlled element.

7. This process is repeated several times throughout the life of the structure.

However, this process is not entirely safe. What if the neural network, which has just switched back to learning mode, cannot meet its acceptable performance objective ? It is highly likely that some controllers could end up looking for a picture of the structure so far from the real one that their actions could endanger the latter. From this point of view, artificial neural networks represent a promising concept, but the idea that they can be routed leads to scary conclusions.

Chapter 6

Conclusion

As stated in the introduction, the goals of this thesis were to provide a description of the field of artificial neural networks appropriate for the civil engineering audience, and, to demonstrate that such networks can be used to control a simple structural system.

As presented in chapter 2, the fundamental definition attached to artificial neural networks is that they are signal processing units with the ability to be trained and to learn in order to adapt to their environment. Moreover, chapter 3 highlighted that fully-connected feed-forward networks, the kind of networks most likely to be used within the civil engineering community, can have their efficiency improved by several ways, i.e (1) better training algorithms, (2) some input data pre-processing, and (3) a number of neurons that can evolve during the training phase.

New control devices require new control theories, one of which, the closed-loop neurocontrol method, was introduced in chapter 4 and used in the simulations of chapter 5.

In the latter chapter, the study of a cantilever beam supporting fluctuating loads emphasized the superiority of the Levenberg-Marquardt algorithm over the other existing training methods. In the simulations performed in this thesis, this algorithm cut the number of training cycles on average by a factor of 45, and allows neural network scientists to consider the idea of "instant learning." After presenting simulations which demonstrate that a simple element of structure can theoretically be controlled

by a neurocontroller, chapter 5 proposes the idea of "self-adaptive" controller, which would certainly be an interesting direction for further research.

Appendix A

Simulating a Beam Using Different Training Algorithms

A.1 The "simulation1" MATLAB program

```
%% This program calls function BEAM() several times with different
%% training functions. It stores the result of each simualtion in
%% files called simulation_results_X, where X is a number.

nbOfTrainFunc = 9 ;
nbOfTraining = 10 ;

records{1,1} = 'TRAINING FUNCTION' ;
records{1,2} = 'AVER. EPOCH NB' ;
records{1,3} = 'AVER. PERF' ;

records{ 2,1} = 'traingd ' ;
records{ 3,1} = 'traingda' ;
records{ 4,1} = 'traingdm' ;
records{ 5,1} = 'traingdx' ;
records{ 6,1} = 'trainbfg' ;
records{ 7,1} = 'traincgb' ;
records{ 8,1} = 'traincgf' ;
```

```
records{ 9,1} = 'traincgp' ;
records{10,1} = 'trainlm ' ;

for i=2:(nbOfTrainFunc+1)
  for j=2:3
    records{i,j} = 0 ;
  end
end

i = 0 ;
name = [ 'simulation_results_' , char(i+48)] ;
fid = fopen( name , 'w' );
fprintf(fid, '%s\t%s\t\t%s\n',records{1,1},records{1,2},records{1,3});
for l=2:10
  fprintf(fid, '%s\t\t%f\t\t%e\n',records{l,1},records{l,2},records{l,3});
end
fclose(fid);


for i=1:nbOfTraining
  name = [ 'simulation_results_' , char(i+48)] ;
  for j=2:(nbOfTrainFunc+1)
    temp = beam( records{j,1} ) ;
    records{j,2} = ((i-1)*records{j,2} + temp(1))/i ;
    records{j,3} = ((i-1)*records{j,3} + temp(2))/i ;
    fid = fopen( name , 'w' );
    fprintf(fid, '%s\t%s\t\t%s\n',records{1,1},records{1,2},records{1,3});
    for l=2:10
      fprintf(fid, '%s\t\t%f\t\t%e\n',records{l,1},records{l,2},records{l,3});
    end
    fclose(fid);
  end
end
```

A.2 The "beam" MATLAB function

```
function perf = beam(trainFunc )
% BEAM Simulates a beam using specified training function
% BEAM('trainFunction') builds a feed-forward neural network
% that simulates a beam. The network is trained using trainFunc.

%%%%%%%%%%%%%%%%%%% creating the data

rho = 2500 ;
g = 9.81 ;
nu = 0.2 ;                                                          10
E= 35000000000 ;
mu = E/(2*(1+nu));
l= 5 ;
a= 0.2 ;
F1 = 10000 * rand(1,4000);
X1= l * rand(1,4000);
Fd = 10000 * rand(1,4000);

inputs = [ F1 ; X1 ; Fd ] ;
                                                                    20
%%%%%%%%%%%%%%%%%%% computing targets

u = Fd*l*((4*l^2)/(E*a^4)+(6)/(5*mu*a^2))+(6/E/a^4)*F1.*(X1.^2).*(l-X1/3)...
-rho*g*l^2/2*((6/(mu*5))+(3*l^2)/(E*a^2));

%%%%%%%%%%%%%%%%%%% data pre-processing

% mean and stand. dev.
[inputs1,meaninputs1,stdinputs1,u1,meanu1,stdu1] = prestd(inputs,u) ;
%min and max                                                        30
[inputs2,mininputs1,maxinputs1,u2,minu1,maxu1] = premnmx(inputs1,u1);

%%%%%%%%%%%%%%%%%%% network creation
```

```
                    % 1st = number of inputs
                    % 2nd = number of layers
beam = network(1,2) ;

%%%%%%%%%%%%%%%%%%% network customization

beam.biasConnect = [1;1] ;
beam.inputConnect = [1;0] ;
beam.layerConnect = [0 0;1 0] ;
beam.outputConnect = [0 1] ;
beam.targetConnect = [0 1] ;

beam.inputs{1}.range = minmax(inputs2);
beam.inputs{1}.size = 3 ;

beam.layers{1}.size = 12 ;
beam.layers{1}.initFcn = 'initwb' ;
beam.layers{1}.netInputFcn = 'netsum' ;
beam.layers{1}.transferFcn = 'tansig' ;

beam.layers{2}.size = 1 ;
beam.layers{2}.initFcn = 'initwb' ;
beam.layers{2}.netInputFcn = 'netsum' ;
beam.layers{2}.transferFcn = 'purelin' ;

beam.biases{1}.initFcn = 'rands' ;
beam.biases{1}.learn = 1 ;
beam.biases{1}.learnFcn = '' ;

beam.biases{2}.initFcn = 'rands' ;
beam.biases{2}.learn = 1 ;
beam.biases{2}.learnFcn = '' ;

beam.inputWeights{1}.initFcn = 'rands' ;
beam.inputWeights{1}.learn = 1 ;
beam.inputWeights{1}.learnFcn = '' ;
```

```
beam.inputWeights{1}.weightFcn = 'dotprod' ;

beam.layerWeights{2}.initFcn = 'rands' ;
beam.layerWeights{2}.learn = 1 ;
beam.layerWeights{2}.learnFcn = '' ;
beam.layerWeights{2}.weightFcn = 'dotprod' ;

beam.adaptFcn = '' ;
beam.initFcn = 'initlay' ;
beam.performFcn = 'mse' ;                                          80
beam.trainFcn = trainFunc ;

beam.trainParam.epochs = 1000 ;
beam.trainParam.goal = 1e-6 ;
beam.trainParam.show = 30 ;

%%%%%%%%%%%%%%%%%%% network initialization

beam = init(beam) ;
                                                                   90
%%%%%%%%%%%%%%%%%%% network training

figure(1)
[beam,records] = train(beam, inputs2, u2 ) ;

perf(1) = records.epoch( size(records.epoch,2) );
perf(2) = records.perf ( size(records.perf ,2) );

%%%%%%%%%%%%%%%%%%% network simulation
                                                                   100
results2 = sim(beam, inputs2 ) ;

%%%%%%%%%%%%%%%%%%% data post-processing

results1 = postmnmx(results2,minu1,maxu1);
results = poststd(results1,meanu1,stdu1);
```

```
%%%%%%%%%%%%%%%%%%% post-training analysis

figure(1)                                                    110
[m,b,r] = postreg(results,u);
```

Appendix B

Control Scheme for a Cantilever Beam

B.1 The "simulation2" MATLAB program

```
%% This program calls function BEAM_CONTROL() ten times.
%% Each time, the output of BEAM_CONTROL() is appended to
%% the file simulation2_results.

for i=1:10
  temp = beam_control;
  fid = fopen( 'simulation2_results' , 'a' );
  fprintf(fid, '%f\t%f\n', temp(1), temp(2));
  fclose(fid);
end                                                    10
```

B.2 The "beam-control" MATLAB function

```
function stats = beam_control
% BEAM_CONTROL first creates a model network that simulates
% a beam supporting fluctuating loads. Then, it incorporates
% this model network into a total network implementing the
% closed-loop control scheme. This total network is trained,
% so that its controlling part now constitutes a control
% network for the beam. Finally, the system {controller +beam}
% is tested to see wether the controller really manages to keep
% the beam stable. MINMAX is an array of 2 doubles, representing
% the min and max of the tip displacement of the beam under the
% influence of the control network.

echo off ;
format long g ;
clear ;
clc ;

%%%%%%%%%%%%%%%%%%%%% CREATING THE DATA
%% The beam
g = 9.81 ;
l = 5 ;
a=0.2 ;
rho = 2500 ;
nu = 0.2 ;
E = 35000 ;
mu = 35000/(2*(1+nu));

%% The loads
t = [0:1:600] ; % 10mn of sinusoidal movement

F1 = 10000*sin(t)+1;
X1 = 1 ;
F2 = 10000*sin(t+pi/4)+1;
X2 = 2 ;
```

```
F3 = 10000*sin(t+pi/2)+1;
X3 = 3 ;
F4 = 10000*sin(t+3*pi/4);
X4 = 4 ;

F5 = - 10000 ;                                                    40
X5 = 1/2 *(1+sin(t));
F6 = - 10000 ;
X6 = 1/2 *(1+cos(t));

input = [F1;F2;F3;F4;X5;X6];


        %%%%%%%%%%%%%%%%%%%%%
        %%
        %%   MODEL NETWORK                                        50
        %%
        %%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%% Data creation

Fd = -10000 + 20000 * rand(1,size(t,2));

u = Fd*l*((4*l^2)/(E*a^4)+6/(5*mu*a^2))- rho*g*l^2/2*(6/(5*mu)+(3*l^2)/(E*a^2)) ;
u = u + 6/(E*a^4)*( F1*X1^2*(l-X1/3) + F2*X2^2*(l-X2/3) + F3*X3^2*(l-X3/3)  );      60
u = u + 6/(E*a^4)*( F4*X4^2*(l-X4/3) + F5*X5.^2.*(l-X5/3) + F6*X6.^2.*(l-X6/3) );

inputToMod = [input ; Fd] ;

%%%%%%%%%%%%%%%%%% Data pre-processing

[inputToMod1,meanInputToMod1,stdInputToMod1] = prestd(inputToMod);
[inputToMod2,minInputToMod1,maxInputToMod1] = premnmx(inputToMod1);

[u1,meanu1,stdu1] = prestd(u);                                    70
```

```
[u2,minu1,maxu1] = premnmx(u1);

%%%%%%%%%%%%%%%%%% network creation

mod_net = network(1,2) ;

%%%%%%%%%%%%%%%%%% network customization

mod_net.biasConnect = [1;1] ;
mod_net.inputConnect = [1;0] ;
mod_net.layerConnect = [0 0;1 0] ;
mod_net.outputConnect = [0 1] ;
mod_net.targetConnect = [0 1] ;

mod_net.inputs{1}.range = minmax(inputToMod2);
mod_net.inputs{1}.size = 7 ;

mod_net.layers{1}.size = 12 ;
mod_net.layers{1}.initFcn = 'initwb' ;
mod_net.layers{1}.netInputFcn = 'netsum' ;
mod_net.layers{1}.transferFcn = 'tansig' ;

mod_net.layers{2}.size = 1 ;
mod_net.layers{2}.initFcn = 'initwb' ;
mod_net.layers{2}.netInputFcn = 'netsum' ;
mod_net.layers{2}.transferFcn = 'purelin' ;

mod_net.biases{1}.initFcn = 'rands' ;
mod_net.biases{1}.learn = 1 ;
mod_net.biases{1}.learnFcn = '' ;

mod_net.biases{2}.initFcn = 'rands' ;
mod_net.biases{2}.learn = 1 ;
mod_net.biases{2}.learnFcn = '' ;

mod_net.inputWeights{1}.initFcn = 'rands' ;
```

```
mod_net.inputWeights{1}.learn = 1 ;
mod_net.inputWeights{1}.learnFcn = '' ;
mod_net.inputWeights{1}.weightFcn = 'dotprod' ;

mod_net.layerWeights{2}.initFcn = 'rands' ;
mod_net.layerWeights{2}.learn = 1 ;
mod_net.layerWeights{2}.learnFcn = '' ;
mod_net.layerWeights{2}.weightFcn = 'dotprod' ;

mod_net.adaptFcn = '' ;
mod_net.initFcn = 'initlay' ;
mod_net.performFcn = 'mse' ;
mod_net.trainFcn = 'trainlm' ;

mod_net.trainParam.epochs = 1000 ;
mod_net.trainParam.goal = 1e-6 ;
mod_net.trainParam.show = 5 ;

%%%%%%%%%%%%%%%%%%% network initialization

mod_net = init( mod_net ) ;

%%%%%%%%%%%%%%%%%%% network training

figure(1)
mod_net = train(mod_net, inputToMod2, u2 ) ;



        %%%%%%%%%%%%%%%%%%%%%%%
        %%
        %%   TOTAL NETWORK
        %%
        %%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%% Data creation
```

```
targets = zeros(1,size(t,2));

%%%%%%%%%%%%%%%%%% Data pre-processing

[input1,meanInput1,stdInput1] = prestd(input);
[input2,minInput1,maxInput1] = premnmx(input1);

%%%%%%%%%%%%%%%%%% network creation

tot_net = network(1,4) ;

%%%%%%%%%%%%%%%%%% network customization

tot_net.biasConnect = [1;1;1;1] ;
tot_net.inputConnect = [1;0;1;0] ;
tot_net.layerConnect = [0 0 0 0;1 0 0 0;0 1 0 0;0 0 1 0] ;
tot_net.outputConnect = [0 0 0 1] ;
tot_net.targetConnect = [0 0 0 1] ;

tot_net.inputs{1}.range = minmax(input2);
tot_net.inputs{1}.size = 6 ;

tot_net.layers{1}.size = 12 ;
tot_net.layers{1}.initFcn = 'initwb' ;
tot_net.layers{1}.netInputFcn = 'netsum' ;
tot_net.layers{1}.transferFcn = 'tansig' ;

tot_net.layers{2}.size = 1 ;
tot_net.layers{2}.initFcn = 'initwb' ;
tot_net.layers{2}.netInputFcn = 'netsum' ;
tot_net.layers{2}.transferFcn = 'purelin' ;

tot_net.layers{3}.size = 12 ;
tot_net.layers{3}.initFcn = 'initwb' ;
tot_net.layers{3}.netInputFcn = 'netsum' ;
```

```
tot_net.layers{3}.transferFcn = 'tansig' ;

tot_net.layers{4}.size = 1 ;
tot_net.layers{4}.initFcn = 'initwb' ;
tot_net.layers{4}.netInputFcn = 'netsum' ;
tot_net.layers{4}.transferFcn = 'purelin' ;


tot_net.biases{1}.initFcn = 'rands' ;
tot_net.biases{1}.learn = 1 ;
tot_net.biases{1}.learnFcn = '' ;

tot_net.biases{2}.initFcn = 'rands' ;
tot_net.biases{2}.learn = 1 ;
tot_net.biases{2}.learnFcn = '' ;

tot_net.biases{3}.initFcn = '' ;
tot_net.biases{3}.learn = 0 ;
tot_net.biases{3}.learnFcn = '' ;

tot_net.b{3} = mod_net.b{1} ;

tot_net.biases{4}.initFcn = '' ;
tot_net.biases{4}.learn = 0 ;
tot_net.biases{4}.learnFcn = '' ;

tot_net.b{4} = mod_net.b{2} ;

tot_net.inputWeights{1,1}.initFcn = 'rands' ;
tot_net.inputWeights{1,1}.learn = 1 ;
tot_net.inputWeights{1,1}.learnFcn = '' ;
tot_net.inputWeights{1,1}.weightFcn = 'dotprod' ;

tot_net.inputWeights{1,1}.delays = [0 1 2] ;

tot_net.inputWeights{3,1}.initFcn = '' ;
```

```
tot_net.inputWeights{3,1}.learn = 0 ;
tot_net.inputWeights{3,1}.learnFcn = '' ;
tot_net.inputWeights{3,1}.weightFcn = 'dotprod' ;

tot_net.IW{3,1} = mod_net.IW{1,1}(:,1:6);
                                                                    220
tot_net.layerWeights{2,1}.initFcn = 'rands' ;
tot_net.layerWeights{2,1}.learn = 1 ;
tot_net.layerWeights{2,1}.learnFcn = '' ;
tot_net.layerWeights{2,1}.weightFcn = 'dotprod' ;

tot_net.layerWeights{3,2}.initFcn = '' ;
tot_net.layerWeights{3,2}.learn = 0 ;
tot_net.layerWeights{3,2}.learnFcn = '' ;
tot_net.layerWeights{3,2}.weightFcn = 'dotprod' ;
                                                                    230
tot_net.LW{3,2} = mod_net.IW{1,1}(:,7);

tot_net.layerWeights{4,3}.initFcn = '' ;
tot_net.layerWeights{4,3}.learn = 0 ;
tot_net.layerWeights{4,3}.learnFcn = '' ;
tot_net.layerWeights{4,3}.weightFcn = 'dotprod' ;

tot_net.LW{4,3} = mod_net.LW{2,1} ;

tot_net.adaptFcn = '' ;                                             240
tot_net.initFcn = 'initlay' ;
tot_net.performFcn = 'mse' ;
tot_net.trainFcn = 'trainlm' ;

tot_net.trainParam.epochs = 1000 ;
tot_net.trainParam.goal = 1e-6 ;
tot_net.trainParam.show = 5 ;

%%%%%%%%%%%%%%%%%%% network initialization
                                                                    250
```

```
tot_net = init( tot_net ) ;

%%%%%%%%%%%%%%%%%%% network training

figure(1)
tot_net = train(tot_net, input2, targets ) ;


        %%%%%%%%%%%%%%%%%%%%%%%
        %%                                                    260
        %%   CONTROL NETWORK
        %%
        %%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%% network creation

cont_net = network(1,2) ;

%%%%%%%%%%%%%%%%%%% network customization                     270

cont_net.biasConnect = [1;1] ;
cont_net.inputConnect = [1;0] ;
cont_net.layerConnect = [0 0;1 0] ;
cont_net.outputConnect = [0 1] ;
cont_net.targetConnect = [0 1] ;

cont_net.inputs{1}.range = minmax(input2);
cont_net.inputs{1}.size = 6 ;
                                                              280
cont_net.layers{1}.size = 12 ;
cont_net.layers{1}.initFcn = 'initwb' ;
cont_net.layers{1}.netInputFcn = 'netsum' ;
cont_net.layers{1}.transferFcn = 'tansig' ;

cont_net.layers{2}.size = 1 ;
```

```
cont_net.layers{2}.initFcn = 'initwb' ;
cont_net.layers{2}.netInputFcn = 'netsum' ;
cont_net.layers{2}.transferFcn = 'purelin' ;

cont_net.biases{1}.initFcn = '' ;
cont_net.biases{1}.learn = 0 ;
cont_net.biases{1}.learnFcn = '' ;

cont_net.b{1} = tot_net.b{1} ;

cont_net.biases{2}.initFcn = '' ;
cont_net.biases{2}.learn = 0 ;
cont_net.biases{2}.learnFcn = '' ;

cont_net.b{2} = tot_net.b{2} ;

cont_net.inputWeights{1}.initFcn = '' ;
cont_net.inputWeights{1}.learn = 0 ;
cont_net.inputWeights{1}.learnFcn = '' ;
cont_net.inputWeights{1}.weightFcn = 'dotprod' ;

cont_net.IW{1,1} = tot_net.IW{1,1}(:,1:6) ;

cont_net.layerWeights{2}.initFcn = '' ;
cont_net.layerWeights{2}.learn = 0 ;
cont_net.layerWeights{2}.learnFcn = '' ;
cont_net.layerWeights{2}.weightFcn = 'dotprod' ;

cont_net.LW{2,1} = tot_net.LW{2,1} ;


        %%%%%%%%%%%%%%%%%%%%%%%%%%
        %%
        %%   NETWORK SIMULATION
        %%    AND FINAL CHECK
        %%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%% network simulation

result2 = sim( cont_net , input2) ;

result1 = postmnmx(result2,minInputToMod1(7,1),maxInputToMod1(7,1));
result = poststd(result1,meanInputToMod1(7,1),stdInputToMod1(7,1));


inputToMod = [input;result] ;
[inputToMod1,meanInputToMod1,stdInputToMod1] = prestd(inputToMod);
[inputToMod2,minInputToMod1,maxInputToMod1] = premnmx(inputToMod1);

result = sim( mod_net , inputToMod2) ;

%%%%%%%%%%%%%%%%%% final check

stats(1) = min(result);
stats(2) = max(result);
```

Bibliography

[1] DARPA. Neural network study. Technical report, MIT Lincoln Laboratory, 1988.

[2] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1999.

[3] A. Nigrin. *Neural Networks for Pattern Recognition.* Cambridge, MA: the MIT Press, 1993.

[4] J. Zurada. *Introduction to Artificial Neural Systems.* PWS Publishing Company, 1992.

[5] J.M. Mendel and R.W. McLaren. Reinforcement-learning control and pattern recognition systems. In J.M. Mendel and K.S. Fu, editors, *Adaptive, Learning, and Pattern Recognition Systems: Theory and Applications*, volume 66, pages 287–318. New York: Academic Press, 1970.

[6] D.C. Dracopoulos. *Evolutionary Learning Algorithms for Neural Adaptive Control.* Springer, 1997.

[7] Tomas Hrycej. *Neurocontrol: Towards an Industrial Control Methodology.* Wiley-Interscience, 1997.

[8] Brian D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, 1996.

[9] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford:New York:Clarendon Press; Oxford University Press, 1995.

[10] http://glimpse.cs.arizona.edu/japan/kahaner.reports/fuzzy-ds.91.

[11] http://www.eng.usf.edu/ slaven/thesisbody.html.

[12] http://www.strainmonitor.com/text/exec.html.

[13] James F. Brule. http://life.csu.edu.au/complex/tutorials/fuzzy.html.

[14] D. Nguyen and B.Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pages 21–26, 1990.

[15] W.H. Pitts W.S. McCulloch. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[16] D. O. Hebb. *The organization of behavior.* New York: Wiley, 49.

[17] M. Menhaj M.T. Hagan. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.

[18] J.L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[19] Robert J. Marks Russell D. Reed. *Neural smithing: supervised learning in feedforward artificial neural networks.* Cambridge, Mass. : The MIT Press, 1999.

[20] F. Rosenblatt. *Principles of Neurodynamics.* Washington DC: Spartan Press, 1961.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel Data Processing*, 1:318–362, 1986.

[22] D. E. Rumelhart, J.L. McClelland, and the PDP Research Group. *Parallel Distributed Processing*, volume 1 and 2. Cambridge, MA: The MIT Press, 1986.

1590 - 4