Constructive Supervised Learning Algorithms for Artificial Neural Networks

Natalio Simon

Delft University of Technology
Faculty of Electrical Engineering

Type	**Master Thesis**
Number of Pages	97
Date	4 juni 1993

Department	**Computer Architecture and Digital Technique**
Codenumber	**1-68340-28(1993)06**
Author	**Natalio Simon**
Title	**Constructive Supervised Learning Algorithms for Artificial Neural Networks**

Supervisor	Prof.dr.ir. A.J. van de Goor
Mentors	Dr. J. Hoekstra

Abstract

The field of Artificial Neural Networks has received in the last years a great deal of attention from the academic community and the industry. Neural Network applications are now found in a wide range of areas, including Control Systems, Medicine, Communication, Cognitive Sciences, Linguistic, Robotics, Physics and Economics.

This report focuses on one of the key challenges of the field of Artificial Neural Networks, the creation of good general-purpose learning algorithms.

As opposed to conventional programmed computing, where a series of explicit commands (a program) processes information on an associated database, Artificial Neural Networks develop information processing capabilities by learning from examples.

Learning techniques can be roughly divided into two categories: Supervised Learning, and Self-Organization or Unsupervised Learning.

Supervised learning requires a set of examples for which the desired network response is known. The learning process consists then of adapting the network in a way that it will produce the correct response for the set of examples. The resulting network should then be able to generalize (give a good response) when presented with cases not found in the set of examples.

The class of Constructive Supervised Learning Algorithms has been defined in this report. As opposed to non-constructive learning algorithms, where the topology of the network is selected before the algorithm starts and remains fixed, Constructive Supervised Learning Algorithms start from an initial network topology and they add new elements (neurons) in order to learn the set of examples. The final topology and the size of the network are then dynamically determined by the algorithm and are a function of the set of examples chosen and of the learning parameters.

A division has been made between Constructive Supervised Learning Algorithms for continuous connectionist networks, and Constructive Supervised Learning Algorithms for discrete connectionist networks.

The power and applicability of Constructive Supervised Learning Algorithms has been shown by using them in six applications of different nature. The applications are:

1. Solution to the Inverse Kinematic Transformations for Robot Control.
2. Corrective networks for damage Robot Controllers.
3. Hamming Codes Generators and Parity Generators.
4. A Pattern Recognition problem.
5. Mapping of Random Boolean Functions.
6. The Monk's problems.

The above applications have allowed an analysis and comparison of the different constructive learning algorithms presented. The comparison has been done in terms of learning speed, size of the network and generalization accuracy.

Continuous Constructive Supervised Learning Algorithms

For Continuous Constructive Supervised Learning Algorithms, the Cascade-Correlation Learning Algorithm and a modified version of it have been presented.

The Cascade-Correlation Algorithm builds a network by adding a hidden unit at a time between the previously added hidden units and the output units, until the training set is learned. Hidden units are trained to correlate to the sum of the remaining outputs' errors before they are connected.

The Modified Cascade-Correlation Algorithm has been introduced for problems were multiple outputs are required. In this algorithm, a new unit is added per output and is trained to correlate to the remaining error at the respective output before it is connected. Then the hidden unit is connected to all output units.

The Inverse Kinematic Transformations for a two and a three degrees of freedom Robot Controller has allowed for a comparison between the Modified and the original Cascade-Correlation algorithms, For this problem, the Modified Cascade-Correlation was shown to learn quicker, scale better to more training cases and to lower tolerance, and to have a higher success rate than the original Cascade-Correlation.

The Inverse Kinematic Transformation for a two degrees of freedom Robot Controller allowed also a comparison of the Modified Cascade-Correlation algorithm with previous results obtained with the non-constructive Backpropagation learning algorithm. In this case the accuracy of generalization achieved by the Modified Cascade-Correlation algorithm was equal or better and with a much smaller network than achieved by using the Backpropagation algorithm.

The other advantage given by the use of the Modified Cascade-Correlation algorithm over the Backpropagation algorithm is that no trial and error stage is needed to determine the amount and size of the hidden layers in the network, what is the case with the non-constructive Backpropagation algorithms.

The determination of an a-priori network topology when applying non-constructive learning algorithms, so as the Backpropagation algorithm, remains a difficult problem in the area of Robot Control. The results achieved for the Inverse Kinematic Transformation and for a Corrective Networks for a damage Robot Controllers, point to the applicability of the Modified Cascade-Correlation learning algorithm for this area of application.

For the Pattern Recognition problem and for the Hamming Code Generation problem, the Modified Cascade-Correlation Learning Algorithm was also shown to learn faster than the original Cascade-Correlation Learning Algorithm, When tested for generalization accuracy with the Inverse Kinematic Transformations and the Corrective networks for a damage Robot Controllers, the Modified Cascade-Correlation algorithm also scored better.

Discrete Constructive Supervised Learning Algorithms

For Discrete Constructive Supervised Learning Algorithms, the Tilling Learning Algorithm, the Tower Algorithm, the Upstart Algorithm and a new algorithm introduced in this report, the BINCOR Algorithm, have been presented.

The Tilling Learning Algorithm and the Tower Learning Algorithm build a network by adding

a layer at a time, where the internal representation of the input patterns in each layer remains different for input patterns with different outputs. The process goes on until the first unit added at a new layer produces the correct mapping for all input patterns. The convergence of these algorithms for any Boolean mapping is guaranteed. The Tilling Algorithm adds a variable number of units in each layer. The Tower Algorithm adds only one unit in each layer and uses "shortcut" connections, i.e. it not only connects the unit of a new layer to the unit of the previous layer, so as in the Tilling Algorithm, but also to the input units.

The Upstart Algorithm and the BINCOR Algorithm use a different strategy than the previous two algorithms for building a network. In the Tilling and the BINCOR algorithms, units are added between the output unit and the inputs in order to correct the remaining misclassified patterns at the output unit. The convergence of these algorithms for any Boolean mapping is guaranteed. In the Upstart Algorithm a recursive strategy is used, where each unit added receives connections from the input units and tries to correct the misclassified input patterns at the unit previously added, leading to a binary tree network structure. In the BINCOR Algorithm, two units are added at a time between the input units and the output unit, where this units try to correct the remaining misclassified input patterns at the output unit, leading to one of two optional network structures: a single hidden layer or a high-order network of multiple layer, where the units in a hidden layer receive connections from the input units and all previously added units.

The mapping of random Boolean functions has allowed for a comparison between the Tilling, Upstart and BINCOR algorithms in terms of size of the network built. The introduced BINCOR algorithm produced the network with less units of the three algorithms.

The Monk's problems has been used to compare the generalization accuracy of the BINCOR algorithm with a variety of machine learning algorithms. The generalization with the BINCOR algorithm was less accurate than with the Cascade-Correlation Learning Algorithm and the Back-propagation Learning Algorithm, but more accurate than with Decision Tree-Based Learning Algorithms, the mFOIL algorithm and Inductive Learning programs.

Contents

1 Introduction **8**

1.1 What are Artificial Neural Networks 9

1.2 Current state of Artificial Neural Networks 10

1.3 Constructive Learning Algorithms 12

1.4 Organization of the Report 13

2 Non-Constructive Supervised Learning **15**

2.1 The Perceptron Learning Algorithm 16

2.1.1 Basic Definitions 16

2.1.2 Algorithm Description 17

2.1.3 Limitations of Perceptrons 19

2.2 The Backpropagation Learning Algorithm 21

2.2.1 Basic Definitions 21

2.2.2 Algorithm description 23

2.2.3 Discussion of Backpropagation 25

2.2.4 The Quickprop Learning Algorithm 25

2.2.5 The Bold Driver Method 29

3 Continuous Constructive Supervised Learning **30**

3.1 The Algorithms 32

3.1.1 The Cascade-Correlation Learning Algorithm 32

3.1.2 Modified Cascade-Correlation 36

3.2 The Applications 39

3.2.1 Inverse Kinematic Transformations for Robot Control 39

3.2.2 Hamming Code Generation 43

3.2.3 Pattern Recognition 44

3.3 The Simulation Results 46

3.3.1 Inverse Kinematic Transformations for Robot Control 46

3.3.2 Hamming Code Generation 49

3.3.3 Pattern Recognition 50

3.4 Discussion and Future Work 51

3.4.1 Inverse Kinematic Transformations for Robot Control 51

3.4.2 Hamming Code Generation 51

3.4.3 Pattern Recognition 52

3.4.4 Future Work 52

4 Discrete Constructive Supervised Learning **53**

4.1 The Algorithms 54

4.1.1 The Pocket Algorithm 54

4.1.2 The Tilling Algorithm 57

4.1.3 The Tower Algorithm 65

4.1.4 The Upstart Algorithm 68

4.1.5 The BINCOR Algorithm 76

4.2 The Applications 80

4.2.1 Random Binary Functions 80

4.2.2 The Monk's Problems 80

4.3 The Simulation Results 85

4.3.1 Random Binary Functions 85
4.3.2 The Monk's Problems 85
4.4 Discussion and Future Work 90

5 Conclusions and Future Work **91**
5.1 Continuous Constructive Supervised Learning Algorithms 92
5.2 Discrete Constructive Supervised Learning Algorithms 93
5.3 General Conclusions and Future Directions 94

1 Introduction

After having being forgotten for more than two decades, the field of Artificial Neural Networks has received in the last years a great deal of attention from the academic community and the industry.

Characteristic to the field is its multidisciplinary character. Neural Network articles are periodically found in Control Systems, Medicine, Communication, Cognitive Sciences, Linguistic, Robotics, Physics, Economics and Artificial Intelligence journals. Added to this are the journals specific to the field of Neural Networks (see Table 1.1).

Neural Networks have also seen their way to industry in the last two years. Table 1.2 presents a list of the major holders of U.S. Neural Network patents through 1991 [Wenskay, 1992]. This list includes leading companies in Electronics so as Philips, IBM, Intel, Mitsushita and Kodak.

Governments are also encouraging developments in this field. The European Community, as an example, funds several R & D projects in Neural Networks under its ESPRIT II and ESPRIT III programmes [Corsi, 1992], [Angeniol et al, 1992] (see Table 1.3). These are being carried out jointly by industry and research centers.

This chapter begins with an overview of Artificial Neural Networks and a survey of the techniques for learning in Artificial Neural Networks. Then the class of constructive learning algorithms is presented. Finally, a brief overview of the structure of the rest of this report is given.

1.1 What are Artificial Neural Networks

Artificial Neural Networks are the primary information processing structures of a technological discipline called Neurocomputing. Neurocomputing is concerned with parallel, distributed, adaptive information processing systems that develop information processing capabilities in response to exposure to an information environment [Hecht-Nielsen, 1990]. Artificial Neural Networks are applied typically in areas such as sensor processing, patterns recognition, data analysis and control, which may require information processing structures for which the algorithms or rules are not known.

An Artificial Neural Network is in the most general sense an information processing structure whose design is motivated by the design and functioning of human brains and components thereof. The formal definition of a neural network follows.

Definition A neural network is a parallel, distributed information processing structure consisting of processing elements (which can possess a local memory and can carry out localized information processing operations) interconnected via unidirectional signal channels called connections. Each processing element has a single output connection that branches ("fans out") into as many collateral connections as desired; each carries the same signal - the processing element output signal. The processing element output signal can be of any mathematical type desired. The information processing that goes on within each processing element can be defined arbitrarily with the restriction that it must be completely local; that is, it must depend only on the current values of the input signals arriving at the processing element via incoming connections and on values stored in the processing element's local memory.

A neural network can be categorized into:

1. **Continuous Neural Network**, where the input and output space are of a continuous nature, and
2. **Discrete Neural Network**, where the input and output space are of a discrete nature.

As opposed to conventional programmed computing, where a series of explicit commands (a program) processes information on an associated database, Artificial Neural Networks develop information processing capabilities by learning from examples.

Learning techniques can be roughly divided into two categories:

1. Supervised Learning.
2. Self-organization or Unsupervised Learning

Supervised learning requires a set of examples for which the desired network response is known. The learning process consists then in adapting the network in a way that it will produce the correct response for the set of examples. The resulting network should then be able to generalize (give a good response) when presented with cases not found in the set of examples.

In self-organization or unsupervised learning the neural network is autonomous: it processes the data it is presented with, finds out about some of the properties of the data set and learns to reflect

these properties in its output. What exactly these properties are, that the network can learn to recognize, depends on the particular network model and learning method.

Many of the learning techniques are closely connected with a certain (class of) network topology.

1.2 Current state of Artificial Neural Networks

Attention in Artificial Neural Networks is being focused primarily in four areas:

1. Software simulation.
2. Theoretical foundations.
3. Hardware implementation.
4. Applications.

Software simulation

Most studies carried out to compare the performance of the different learning algorithms and network topologies are being done on serial computers that simulate the the operation and flow of data of the different models of neural networks.

For this purpose, various benchmark problems are used. Among the most common [Fahlman, 1988], [Fahlman & Lebiere, 1990]:

- The XOR/Parity Problem.
- The Encoder/Decoder Problem.
- The Two-Spirals Problem.

The restricted scope of these benchmarks has led to the use of real-world problems, where higher number of inputs and outputs are used, to compare the performance of the different algorithms and topologies.

Commonly used real-world problems are:

- Handwritten character set recognition.
- Inverse Kinematic Transformations for Robot Control.
- Time-Series prediction.

The various algorithms are compare in terms of:

- Learning speed.

- Size of the network.
- Generalization accuracy.

By *learning speed*, the number of connection crossing or the number of epochs (one epoch is one cycle over the training set) are used as a measure.

The *size of the network* is given by the amount of neurons and the number of layers (and the number of neurons per layer).

By *generalization accuracy* it is meant how well the network performs for input patterns not found in the training set. For this measurement a *test set* is used and the average, maximum and standard deviation of the network error for the test set is compared.

For real-world problems, the training set represents a small part of the input space so the way this set is chosen to important to how well the network learns. Good learning is reflected in good generalization accuracy.

Another factor that plays a roll in how well the networks learns is the error tolerance in learning the training set. If the training set is too accurately learned, then the *over-training* or *grand-mothering* problem occurs. In this case the network learn the training set very well but generalizes poorly for input patterns outside the training set.

Theoretical foundations

Neural Networks are able to learn from examples, to approximate complex functions, to make predictions of non-linear system outputs. They are used for the diagnosis of faulty equipment, to determine the trajectory of a robot, etc. Still a complete formal foundation of how the various algorithms "learn" and how to achieve optimal learning and optimal network size is lacking. Research is carried out to give a mathematical foundation to the different learning algorithms and network topologies.

Hardware implementation

Artificial Neural Networks are inherently parallel. They consist of simple processing units (neurons) that process data independently of each other. Simulations on conventional sequential computers provide a functionality test. To fully exploit the high-speed potential of neural networks for problems requiring real-time response, work is done in the implementation of neural networks in physical hardware, where each processing element can operate independent and in parallel to other elements. Approaches in hardware implementation include analog, digital as well as hybrid circuits.

Two options exist regarding the implementation of Artificial Neural Networks in hardware: implementing hardware for a specific application and for a specific learning algorithm and topology, or a general purpose architecture suitable for different learning algorithms and topologies and for applications with different sizes of inputs and outputs. The last approach being followed by many researchers and the industry (see [Barhen et al., 1992],[Daud et al., 1992],[Hurdle et al., 1992], [Jutten et al., 1990], [Corsi, 1992]).

Applications

Neural Networks are in a stage of development were researchers continue to discover new applications for this technology. From 1989 there is being an exponential increase in patent activity in Neural Networks, lead by the U.S. with Japan in the second place, followed by Korea, Germany, France and the United Kingdom [Wenskay, 1992].

Neural Networks are applied in a wide range of tasks. These tasks include, among others,

- Quality Control: of video images, VLSI masks, leather, oranges.
- Forecasting: of electricity consumption, economic and financial time-series, chip dimensions.
- Vision: motion detection, recognition of objects, texture.
- Diagnosis: of metal corrosion, machinery.
- Control: robot control, industrial process control.
- Classification: of patterns, spectra.

The impact of Artificial Neural Networks is likely to be in areas of artificial perception - making sense of speech, language and visual images. This is where the promise of going beyond the capabilities of conventional programming arise, simply because conventional programming requires an algorithm which, in turn, requires a full analysis of the problem. But because the mechanisms used by humans when they carry out these tasks are not fully understood, satisfactory algorithms cannot be devised. The expectation is that Artificial Neural Networks will build up such solutions through being trained on appropriate input-output relationships [Aleksander & Morton].

1.3 Constructive Learning Algorithms

This report focuses on a class of *Supervised Learning Algorithms* for Artificial Neural Networks, called Constructive or Dynamic Learning Algorithms.

Constructive or Dynamic Learning Algorithms have the characteristic that they modify the neural network topology in the process of learning from a set of examples. They start from an initial network topology and they add new elements (neurons) in order to learn the set of examples. The final topology and the size of the network are dynamically determined by the algorithm and are a function of the set of examples chosen and of the learning parameters.

The aim of this report is threefold:

1. to present the class of Constructive or Dynamic Learning Algorithms,
2. to show the power and applicability of these algorithms by using them in applications of different nature, and
3. to analyze their advantages and disadvantages in comparison with other non-constructive learning algorithms.

Title:	Neural Networks	Title:	Neural Computation
Publisher:	Pergamon Press	Publisher:	MIT Press
Year:	volume 1 in 1988	Year:	volume 1 in 1989
Title:	Neurocomputing	Title:	IEEE Transaction on Neural Networks
Publisher:	Elsevier	Publisher:	IEEE
Year:	volume 1 in 1989	Year:	volume 1 in 1990
Title:	International Journal of Neural Systems	Title:	Network: Computation in Neural Systems
Publisher:	World Scientific Publishing	Publisher:	IOP Publishing Ltd
Year:	volume 1 in 1990	Year:	volume 1 1990
Title:	Journal of Neural Network Computing, Technology, Design, and Applications	Title:	Connection Science: Journal of Neural Computing Artificial Intelligence and Cognitive Research
Publisher:	Auerback Publishers	Publisher:	Carfax Publishin
Year:	volume 1 1989	Year:	volume 1 in 1989
Title:	International Journal of Neural Networks	Title:	Neural Computing and Applications
Publisher:	Learned Information	Publisher:	Springer International
Year:	volume 1 in 1989	Year:	volume 1 in 1992
Title:	Applied Intelligence: The International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies		
Publisher:	Kluwer Academic Publishers		
Year:	volume 1 in 1991		

Table 1.1: Neural Network Journals

1.4 Organization of the Report

The report is organized as follows:

Chapter 2 presents the class of non-constructive supervised learning algorithms. Here, for discrete networks, the Perceptron Learning Algorithm is discussed. Then, for continuous networks, the Backpropagation Algorithm and two variations of it, the Quickprop Algorithm and the Bold-Drive Method, are discussed.

Chapter 3 presents the class of constructive supervised learning algorithms for continuous networks. Here the Cascade Correlation Algorithm and variations on it are discussed. Then the performances of these and algorithms presented in chapter 2 are compared. The comparison is done for the following applications: a pattern recognition problem, the Hamming Code generation problem, and the inverse kinematic transformations of a two and a three degrees of freedom robot arm.

Chapter 4 presents the class of constructive supervised learning algorithms for discrete networks. Here the Pocket Learning Algorithm, the Tilling Algorithm and the BINCOR Algorithm are discussed. Then the performances of these and algorithms presented in the previous chapters are compared. The comparison is done for the following applications: the Monk's Problem and random boolean functions.

Chapter 5 finalizes this report with a discussion of the performances obtained by the learning algorithms of the previous chapters. Conclusions regarding the suitability of Constructive Learning Algorithms are given. An outline of future work directions is presented.

Miscellaneous Univ.	AT&T
IBM	U.S. Government
Hughes Aircraft	Nestor
GTE	Synaptics
Matsushita	Intel
General Electrics	Philips
Kodak	Motorola
Hitachi	

Table 1.2: Major Holders of U.S. Neural Network Patents Though 1991

UNDER ESPRIT II 1988-1990	UNDER ESPRIT III 1990-
PIGMALION *Neurocomputing*	CONNY *Robot Control Based on Neural Network Systems*
GALATEA *Neurocomputing*	HIMMARNET *Study of Hidden Markov Models and Neural Networks for Robust Isolated Word Recognition*
ANNIE *Applications of Neural Networks for Industry in Europe*	NEUROQUACS *Neural Networks based Vision and Signal System for Industrial Quality Control*
NEUFODI *Neural Networks for Forecasting and Diagnosis Applications*	WERNICKE *Neural Network Based, Speaker Independent, Large Vocabulary, Continuous Speech Recognition System*
SPRINT *Speech Processing and Recognition Using Integrated Neurocomputing Techniques*	MUCOM *Multisensory Control of Movement*
SUBSYM *Self-Organization and Analogical Modeling Using Sub-Symbolic Computing*	INSIGHT *Vision Systems for a Natural Human Environment*
NERVES *Innovative Architectures for Neurocomputing Machines and VLSI Neural Networks*	ELENA-NERVES 2 *Enhanced Learning for Evolutive Neural Architectures*
	SSS *Smart Sensory Systems*

Table 1.3: EC Neural Network Projects

Non-Constructive Supervised Learning

2

Supervised learning consists of training a neural network with a set of examples for which the desired outputs are known. The purpose of the learning algorithm is then to adjust the network so that the network produces the correct outputs for the set of examples.

Here we distinguish between two classes of Learning Algorithms: *Non-Constructive* and *Constructive*. By non-constructive we refer to algorithms for which the topology of the network has to be fixed a-priori, while in constructive ones the algorithm itself automatically determines the topology of the network.

This chapter presents the class of non-constructive supervised learning algorithms. First the Perceptron Learning Algorithm [Rosenblatt, 1962] is described and discussed. Then the Backpropagation Algorithm [Rumelhart, Hinton & Williams, 1986] and two variations of it, the Quickprop Algorithm [Fahlman, 1988] and the Bold-Drive Method [Vogl et al.], are described and discussed.

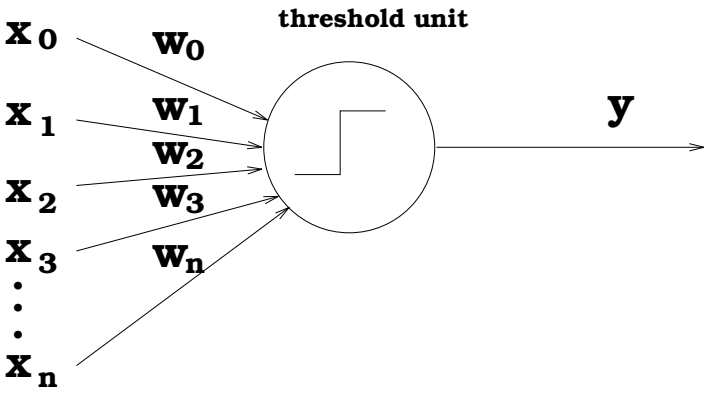


Figure 2.1: Basic Neuron

2.1 The Perceptron Learning Algorithm

In his book "Principles of Neurodynamics", Frank Rosenblatt described the perceptron as a simplified network in which certain properties of real nervous systems are exaggerated while others are ignored. This section starts by presenting the basic model of a Perceptron, then the perceptron learning algorithm is described and finally, the capabilities of the Perceptron are discussed.

2.1.1 Basic Definitions

The purpose of the Perceptron Learning Algorithm is to train a processing unit to produce the correct output when provided with a set of input patterns for which the output is known. The algorithm stops when the processing unit produces the correct output for each of the given input patterns.

The basic processing unit (neuron) of a Perceptron is shown in Figure 2.1.

In Figure 2.1, w_i is the weight of the connection from input x_i. The output of the unit is given by

$$y = f_h(net) \tag{2.1}$$

$$net = \sum_{i=1}^{n} w_i x_i - \theta \tag{2.2}$$

$$= \sum_{i=0}^{n} w_i xi \quad where\ x_0 = 1\ and\ w_0 = \theta \tag{2.3}$$

The activation function, f_h, is the Threshold function shown in Fig 2.2.

$$f_h(x) = \begin{cases} 1 & if \quad x \geq 0 \\ 0 & if \quad x < 0 \end{cases} \tag{2.4}$$

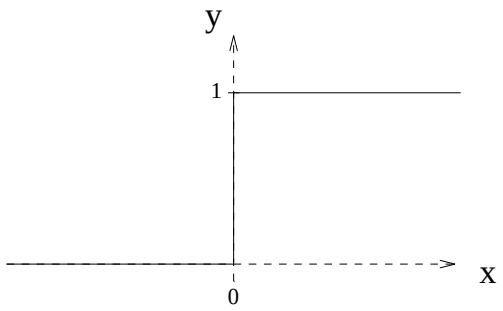


Figure 2.2: The Threshold Function.

2.1.2 Algorithm Description

The Perceptron Learning Algorithm can be summarized as follows [Beale & Jackson, 1990]:

- set the weights randomly
- present an input
- calculate the actual output by taking the thresholded value of the weighted sum of the inputs
- alter the weights to reinforce correct decisions and discourage incorrect decisions- i.e. reduce the error
- present the next input etc.

This leads to the following algorithm:

The Perceptron Learning Algorithm

1. **Initialize weights**

Define $w_i(t)$, $(0 < i < n)$, to be the weight from input i at time t, and θ to the threshold value in the output node. Set w_0 to be $-\theta$, and x_0 to be always 1.
Set $w_i(0)$ to small random values, thus initializing all the weights and the threshold.

2. **Present input pattern and desired output**

Present binary input pattern $X_p = x_1, x_2, x_3, ..., x_n$ and desired binary output $d(t)$.

3. **Calculate actual output**

$y(t) = f_h\left(\sum_{i=0}^{n} w_i(t)x_i(t)\right)$

4. **Adapt weights**

if $(y(t) = 0)$ and $(d(t) = 1)$
  for i **from** 0 **to** n **do**
    $w_i(t+1) = w_i(t) + \eta x_i(t)$
if $(y(t) = 1)$ and $(d(t) = 0)$
  for i **from** 0 **to** n **do**
    $w_i(t+1) = w_i(t) - \eta x_i(t)$
if $(y(t) = d(t))$
  for i **from** 0 **to** n **do**
    $w_i(t+1) = w_i(t)$

where $0 < \eta \leq 1$, is a positive gain term that controls the adaptation rate.

5. **Next input pattern**

Chose next input pattern and go to 2.

The algorithm stops when $y(t) = d(t)$ for each of the patterns of the training set, i.e. when all the training set has been learned.

Step 4 of the algorithm adds the input values to the weights when the output should be on (1), and subtracts the input values to the weights when the output should be off (0). The weights are unchanged if the network makes the correct decision. Also, weights are not adjusted on input lines which do not contribute to the incorrect response, since each weight is adjusted by the value of the input on that line, $x_i(t)$, which would be zero. The gain term η has the effect of slowing down the change in the weights, making the network take smaller steps towards the solutions. Empirical results show that this leads to faster convergence.

Widrow-Hoff Delta Rule

A second learning rule, the Widrow-Hoff delta rule [Widrow, 1962], calculates the difference between the weighted sum and the point at which the threshold function changes value and calls this the *error*.

$$error = \sum_{i=1}^{n} w_i x_i - \theta = \sum_{i=0}^{n} w_i x i \tag{2.5}$$

$$\tag{2.6}$$

Weight adjustment is then carried out in proportion to that error. This leads to big changes in the weights when the weighted sum is long way from the desired value, while altering them only slightly when the weighted sum is close to that required to give the correct solution. This method achieves faster convergence. Applying the Widrow-Hoff Delta Rule means replacing step 4 with the following:

---

4. **Adapt weights - Widrow-Hoff delta rule**

if $(y(t) = d(t))$

$error(t) = 0$
if $(y(t) = 0)$ and $(d(t) = 1)$
$error(t) = \sum_{i=0}^{n} w_i(t)x_i(t)$
if $(y(t) = 1)$ and $(d(t) = 0)$
$error(t) = -\sum_{i=0}^{n} w_i(t)x_i(t)$
for i **from** 0 **to** n **do**
$w_i(t+1) = w_i(t) + \eta error(t)x_i(t)$

where $0 < \eta \leq 1$, is again positive gain function that controls the adaptation rate.

Bipolar Values

The algorithms can be applied to binary input and output spaces, i.e. the inputs and the output can have the value 0 or 1, or they can work with bipolar input and output spaces, where the inputs and the output can have the value -1 or 1. Using binary inputs means that input lines where $x_i = 0$ are not trained, whereas bipolar values allow all input lines to be trained each time. The use of bipolar values help to speed up the convergence process.

2.1.3 Limitations of Perceptrons

The Perceptron Learning Algorithm is capable of classifying two classes of input patterns, or in the case of a layer of n Perceptron units, each unit can be trained in parallel to recognize one class of inputs from the rest, leading to the classification of n different classes of input patterns. An example of this is given in Figure 2.3, where a Percetron is used to classified the ciphers 0 to 9, drawn in a grid of 8x5 pixels (the figure shows the cipher 6 as example).

There is, however, a limitation to the classification capabilities of a Perceptron, it can only correctly classify linearly separable patterns. For example, Classifying binary inputs according to their parity, i.e. the Perceptron outputs 1 when the pattern has odd parity and 0 when the parity is even, is an example of a non linearly separable problem which cannot be solve by the Perceptron. For the case of two dimensional input space, the parity problem then becomes the XOR function. From Figure 2.4 it can be seen that no line can separate one class from the other.

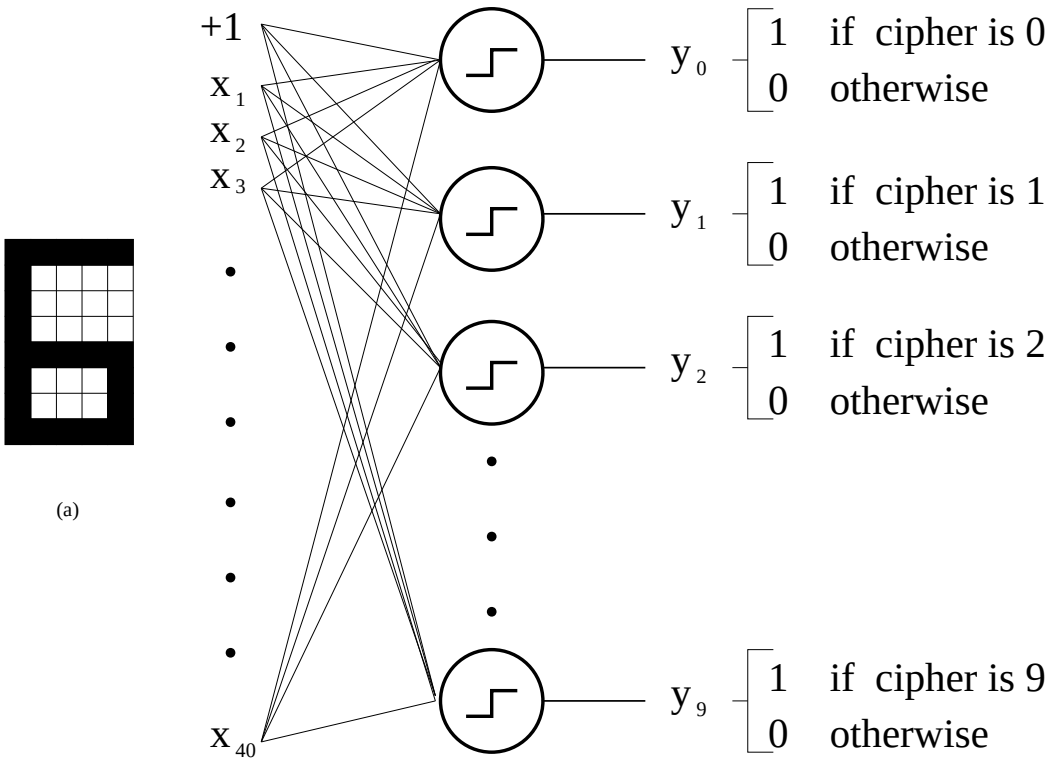


Figure 2.3: Ciphers classification with a Perceptron. (a) Example input: cipher 6. (b) Perceptron Architecture.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

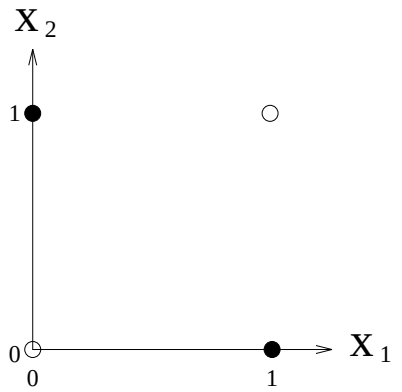


Figure 2.4: The XOR Problem

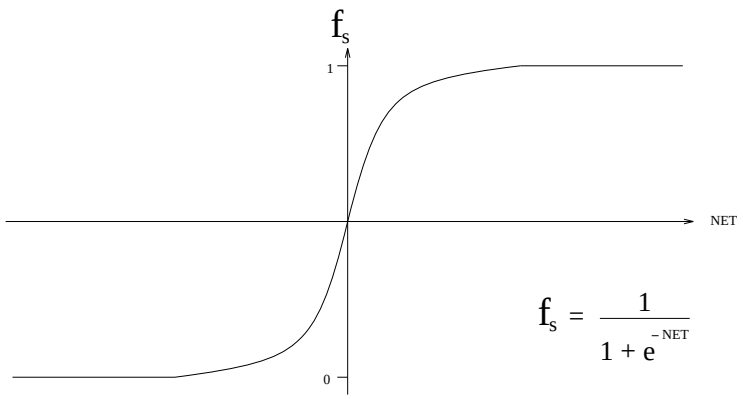


Figure 2.5: The Sigmoid Function

2.2 The Backpropagation Learning Algorithm

The Backpropagation Learning Algorithm was popularized in 1986 by Rumelhart, McClelland and Williams [Rumelhart, Hinton & Williams, 1986]. The Backpropagation algorithm is a supervised learning algorithm that provides a method to adjust the weights in a multi-layer network of connected processing units. The purpose of the weight adjustment is again to produce the correct outputs for a given training set, where the training set consists of patterns of inputs and desired outputs. This section starts by presenting the basic model of a Backpropagation network, then the Backpropagation learning algorithm is described and finally, the advantages and disadvantages of the algorithm are discussed.

2.2.1 Basic Definitions

The basic processing unit (neuron) of a Backpropagation network is similar to the one shown in Figure 2.1.

Here the activation function, f_s, used is the Sigmoid function

$$o = f_s(net) = \frac{1}{1 + e^{-net}} \tag{2.7}$$

where net is defined in equation (2.2). Figure 2.5 shows the shape of a Sigmoid function. The use of the Sigmoid function means that the input and output spaces are continuous.

Taking the derivative of $f_s(net)$ gives

$$\begin{aligned} f_s'(net) &= \frac{e^{-net}}{(1 + e^{-net})^2} \\ &= f_s(net)(1 - f_s(net)) \\ f_s'(net) &= o(1 - o) \end{aligned}$$

The derivative is therefore a simple function of the output.

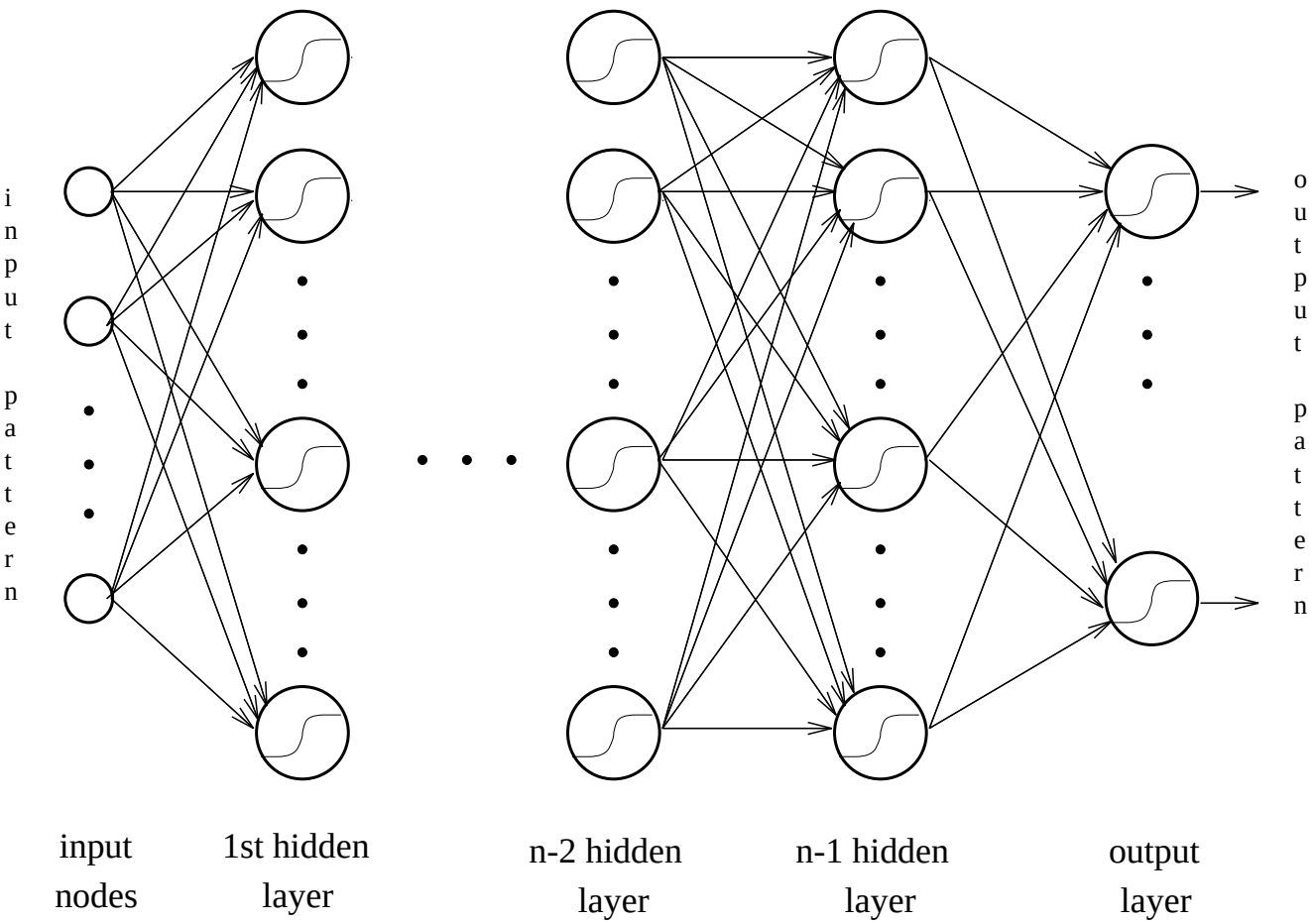


Figure 2.6: Backpropagation Network Topology. n-layers network.

The Backpropagation works on a multi-layer network of connected processing units, where each layer contains one of more units. The determination of the number and size of the network layers is done prior to the learning, usually based on empirical criteria. Figure 2.6 shows a typical network topology.

Backpropagation performs a gradient descend in the weight space in order to minimize the error at the outputs. This is done by calculating the error function for each input pattern and then backpropagating the error from one layer to the previous one. The weights of a node are adjusted in direct proportion to the error in the units to which it is connected.

The error measure used in the algorithm is the sum of the *Mean Square Error* at the outputs of the network, for all patterns of the training set. The error for one pattern p is defined as:

$$E_p = \sum_o (d_{po} - o_{po})^2 \tag{2.8}$$

The total error is defined as:

$$E = \sum_p \frac{1}{2} \sum_o (d_{po} - o_{po})^2 = \sum_p \frac{1}{2} E_p \tag{2.9}$$

where d_{po} is the desired value at output o, for input pattern p and o_{po} is the actual value at output o, for input pattern p. The index p refers to each pattern of the training set of inputs and desired outputs.

To minimize the total error, the change made to each weight is proportional to the derivative of the error measure with respect to that weight, where w_{ij} is the weight between the output of unit i and unit j in the next layer (one layer closer to the output layer).

$$\Delta_p wij = -\eta \frac{\partial E_p}{\partial w_{ij}} \tag{2.10}$$

The derivative of the error is given by

$$\frac{\partial E_p}{\partial w_{ij}} = -\delta_{pj} o_{pi} \tag{2.11}$$

where for an output unit

$$\delta_{pj} = f'_j(net_{pj})(d_{pj} - o_{pj}) \tag{2.12}$$

and for a hidden unit

$$\delta_{pj} = f'_j(net_{pj}) \sum_k \delta_{pk} w_{jk} \tag{2.13}$$

where δ_{pk} are the error signals from one layer closer to the output layer that receive a connection from hidden unit j and k is the number of such connections.

2.2.2 Algorithm description

The Backpropagation Algorithm for a multi-layer network of processing units, where the units have a sigmoid function, is presented below:

The Backpropagation Learning Algorithm

1. **Initialize weights**

Set all weights to small random values.

2. **Present input pattern and desired output**

Present input pattern $X_p = x_{p,0}, x_{p,1}, x_{p,2}, ..., x_{p,n}$ and desired output pattern $D_p = d_{p,0}, d_{p,1}, d_{p,2}, ..., d_{p,m-1}$, where n is the number of input nodes and m the number of output nodes.

3. **Calculate actual output - Forward pass**

Starting from the first hidden layer, calculate hidden units outputs and pass them as inputs to the next layer, until the output layer is reached.

for l **from** 0 **to** N **do**
  for j **from** 0 **to** $n^l - 1$ **do**
    $o_{pj}^l(t) = f_s\left(\sum_{i=0}^{n^{l-1}} w_{ij}^l(t) 0_{pi}^{l-1}(t)\right)$

Where $0 \leq l \leq N$ refers to the layer number, N being the total number of layers. $0 \leq j \leq (n^l - 1)$ refers to the unit number within layer l, n^l being the number of units in layer l, and $0_{pi}^{l-1}(t)$ being the output of unit i of the previous layer $l-1$ (where $0_{pi}^0(t)$ is the input $x_i(t)$). w_{ij}^l is the weight from unit i (of layer $l-1$) to unit j of layer l.

4. **Adapt weights - Backward pass**

Start from the output layer, and working backwards, adapt weights.

for l **from** N **to** 1 **do**
  for j **from** 0 **to** $n^l - 1$ **do**
    $w_{ij}^l(t+1) = w_{ij}^l(t) + \eta \delta_{pj}^l(t) 0_{pj}^l(t)$

where η is a gain term, and δ_{pj}^l is an error term for pattern p on unit j of layer l.

For output units: $\delta_{pj}^N(t) = 0_{pj}^N(t)(1 - 0_{pj}^N(t))(d_{pj} - 0_{pj}^N(t))$

For hidden units: $\delta_{pj}^l(t) = 0_{pj}^N(t)(1 - 0_{pj}^N(t)) \sum_{k=0}^{n^{l+1}-1} \delta_{pk}^{l+1}(t) w_{jk}^{l+1}(t)$

where for hidden units, the sum is over the units of one layer above the layer of unit j.

5. **Next input pattern**

If p is not the last input pattern then p = p + 1 (chose next input pattern) and go to step 2.

6. **Calculate Total Error**

Compute the total error by calculating the outputs (step 3) for all training patterns
and using equation (2.4).
If the total error is within the desired value STOP else select the first pattern and go to step 2.

In practice, the extra forward pass in step 6 for each training input is avoided and the partial errors per pattern are calculated in step 3. This means however that the total error is not the correct one since the weight updates of the latter patterns are not taken into account for the earlier patterns, their influence is neglected.

In order to speed up the learning in the Backpropagation algorithm, a momentum term can be used in the weight update. This momentum term is a part of the previous weight change and is added to the new weight. Step 4 of the Backpropagation algorithm becomes then:

4. **Adapt weights with momentum term - Backward pass**

Start from the output layer, and working backwards, adapt weights.

for l **from** N **to** 1 **do**
  for j **from** 0 **to** $n^l - 1$ **do**
    $w_{ij}^l(t+1) = w_{ij}^l(t) + \eta\delta_{pj}^l(t)0_{pj}^l(t) + \alpha\Delta w_{ji}^l(t-1)$

where δ_{pj}^l remains the same as without a momentum term and $0 < \alpha < 1$.

2.2.3 Discussion of Backpropagation

The Backpropagation algorithm is a non-constructive learning algorithm that requires an a-priori network topology, i.e., the determination of the number and size of the network layers is done prior to the learning.

The Backpropagation Algorithm presents the problem of being very slow at learning for real-world problems where many inputs and outputs are required [Antsaklis, 1992], [Fahlman, 1988], [Daud et al., 1992].

Another problem of the algorithms is the step-size problem. To find the global minimum in the overall error function, the Backpropagation Algorithm computes the first derivative of the overall error function with respect to each weight in the network. If small steps are taken down the gradient vector, a local minimum of the error function may be reached and not the global minimum. If large steps are taken, then the network could oscillate around the global minimum, not reaching it.

The Backpropagation Algorithm is based on the approximation that changes in one weight have no effect on the error gradient at other weights. In reality when one weight is changed, the error gradient at other weights varies as well. The algorithm doesn't take this into consideration, so the descent in the error space may be sometimes wrongly directed, causing a slowdown in the speed of convergence of the algorithm. This problem is referred to as the moving target problem [Fahlman & Lebiere, 1990].

Several variations has been suggested to solve the above mentioned problems and to speed up the convergence of the algorithm. What follows are two variations on the original Backpropagation Algorithm, the Quickprop Learning Algorithm and the Bold Driver Method.

2.2.4 The Quickprop Learning Algorithm

Quickprop [Fahlman, 1988] is a modification on the Backpropagation learning algorithm presented above. The Quickprop Learning Algorithm differs from Backpropagation in the way the weights are adapted. The algorithm assumes that the total error vs. weight curve can be approximated by a parabola and that the change in the slope of the error curve, as seen by each weight, is not affected by all the other weights that are changing at the same time. For each weight, a copy of the previous error slope and the previous weight change is kept. Then, together with the current

error slope, a jump is done to the minimum of the parabola and the weight is updated with this minimum. The weight updating rule then becomes:

$$\Delta w_{ji}^l(t) = \frac{S_{ji}^l(t)}{S_{ji}^l(t-1) - S_{ji}^l(t)} \Delta w_{ji}^l(t-1)$$

where $S_{ji}^l(t)$ and $S_{ji}^l(t-1)$ are the current and previous error slopes with respect to weight $w_{ji}^l(t)$. $S_{ji}^l(t)$ is defined as:

$$S_{ji}^l(t) = \frac{\partial E(t)}{\partial w_{ji}^l(t)} = \sum_{p=0}^{P-1} \frac{\partial E_p(t)}{\partial w_{ji}^l(t)} = \sum_{p=0}^{P-1} -\delta_{pj}^l O_{pi}^l$$

where $E(t)$ and δ_{pj}^l have the same value as in standard back-propagation and P is the number of training patterns.

The error slope is the derivative of the total error measure E, so the error terms, δ_{pj}^l, have to be accumulated over all P input patterns. This means that the weights are only adjusted after the presentation of all patterns of the training set and that extra storage is needed per weight to accumulate the error terms. Also extra storage is needed per weight to keep the previous error slope.

A way is need to start the updating rule or to continue it if the previous error slope, $S_{ji}^l(t-1)$, was 0. This is done in the algorithm by changing the weight updating rule to:

$$\Delta w_{ji}^l(t) = \begin{cases} \frac{S_{ji}^l(t)}{S_{ji}^l(t-1) - S_{ji}^l(t)} \Delta w_{ji}^l(t-1) & if \quad (S_{ji}^l(t).S_{ji}^l(t-1) < 0) \\ \frac{S_{ji}^l(t)}{S_{ji}^l(t-1) - S_{ji}^l(t)} \Delta w_{ji}^l(t-1) + \varepsilon S_{ji}^l(t) & if \quad (S_{ji}^l(t).S_{ji}^l(t-1) \geq 0) \end{cases}$$

Here, the current slope times a gain term, ε, is added to the weight if the previous slope is 0, or if the previous and the current slope are in the same direction, i.e. have the same sign. The last from empirical results showing that this increases the speed of convergence.

The parameters μ is also introduced in the weight updating rule, to avoid taking too big steps. The weight updating rule then becomes:

$$\Delta W_{ji}^l(t) = min(\Delta w_{ji}^l(t), \mu \Delta w_{ji}^l(t-1))$$

where Fahlman suggests a value of $\mu = 1.75$.

The Quickprop Learning Algorithm has been shown to achieves faster convergence for the Encoder/Decoder problem and the XOR problem [Fahlman, 1988].

The Quickprop Learning Algorithm

1. **Initialize weights**

Set all weights to small random values.

2. **Calculate error slopes**

For all the training patterns, accumulate the error slopes.

For all error slopes (error slopes initialization)
    $S^l_{ji}(t-1) = S^l_{ji}(t)$
    $S^l_{ji}(t) = 0$
for p **from** 0 **to** P-1 **do**
    2.1 **Calculate actual output - Forward pass**
    2.2 **Accumulate error slopes - Backward pass**

2.1. **Calculate actual output - Forward pass**

Starting from the first hidden layer, calculate hidden units outputs and pass them as inputs to the next layer, until the output layer is reached.

for l **from** 0 **to** N **do**
    for j **from** 0 **to** $n^l - 1$ **do**
        $y^l_{pj}(t) = f_h\left(\sum_{i=0}^{n^{l-1}} w^l_{ij}(t)\, o^{l-1}_{pi}(t)\right)$

Where $0 \leq l \leq N$ refers to the layer number, N being the total number of layers. $0 \leq j \leq (n^l - 1)$ refers to the unit number within layer l, n^l being the number of units in layer l, and $0^{l-1}_i(t)$ being the output of unit i of the previous layer $l-1$ (where $0^0_i(t)$ is the input $x_i(t)$). w^l_{ij} is the weight from unit i (of layer $l-1$) to unit j of layer l.

2.2 **Accumulate error slopes - Backward pass**

Start from the output layer, and working backwards,

for l **from** N **to** 1 **do**
for j **from** 0 **to** $n^l - 1$ **do**
If l=N (for output units)
$\delta^N_{pj}(t) = 0^N_{pj}(t)(1 - 0^N_{pj}(t))(d_{pj} - 0^N_{pj}(t))$
$S^l_{ji}(t) = S^l_{ji}(t) - \delta^l_{pj}(t)O^l_{pi}(t)$

If $l \neq N$ (For hidden units)
$\delta^l_{pj}(t) = 0^N_{pj}(t)(1 - 0^N_{pj}(t)) \sum_{k=0}^{n^{l+1}-1} \delta^{l+1}_{pk}(t) w^{l+1}_{jk}(t)$
$S^l_{ji}(t) = S^l_{ji}(t) - \delta^l_{pj}(t)O^l_{pi}(t)$

3. **Calculate Total Error**

Compute the total error using equation (2.4).
If the total error is within the desired value STOP,

4. **Adapt weights**

For all weights

$$\Delta w^l_{ji}(t) = \begin{cases} \frac{S^l_{ji}(t)}{S^l_{ji}(t-1) - S^l_{ji}(t)} \Delta w^l_{ji}(t-1) & if \quad (S^l_{ji}(t).S^l_{ji}(t-1) < 0) \\ \frac{S^l_{ji}(t)}{S^l_{ji}(t-1) - S^l_{ji}(t)} \Delta w^l_{ji}(t-1) + \varepsilon S^l_{ji}(t) & if \quad (S^l_{ji}(t).S^l_{ji}(t-1) \geq 0) \end{cases}$$

$$\Delta W^l_{ji}(t) = min(\Delta w^l_{ji}(t), \mu \Delta w^l_{ji}(t-1))$$

5. **Next training set presentation**

Go back to step 2.

2.2.5 The Bold Driver Method

The Bold Driver Method [Vogl et al.] is a modification on the Backpropagation Learning Algorithm. In order to overcome the problem of the Backpropagation Learning Algorithm, the Bold Driver Method makes two changes to the way weights are adjusted:

1. the weights are updated after the presentation of all input patterns (as in Quickprop),
2. the Learning rate η is varied with each weight update, and

The new update rule becomes:

$$\Delta w_{ji}^{l}(t) \quad = \quad \eta(t) \sum_{p} (\delta_{pj}^{l}(t) O_{pi}^{l}(t)) + \alpha \Delta w_{ji}^{l}(t-1)$$

After each update, the total error E(t) is compare with the previous value E(t-1). If the update reduced the error, then the learning rate η is increased by a factor $\rho > 1$. Otherwise all weight changes are rejected and the learning rate is decreased by multiplying it by a factor $\beta < 1$ and α is set to zero. When a successful step is taken, α is reset to its original value.

Continuous Constructive Supervised Learning

3

The previous chapter have presented algorithms that belong to the class of non-constructive supervised learning algorithms. What characterizes the non-constructive algorithms is the need to determine the network topology, i.e. the amount of layers and the number of processing units (neurons) per layer, before the learning algorithm is applied. The learning accuracy and the generalization accuracy are, then, a function of the chosen network topology. Choosing the optimal network topology given a problem remains an unsolved problem.

In this chapter, a different class of learning algorithms, referred to as constructive or dynamic learning algorithms, is presented. What characterizes constructive supervised learning algorithms is that the determination of the network topology is part of the algorithm. Usually constructive algorithms starts with a minimal network consisting of only input and output units interconnected, their number dictated by the input and output space of the problem given. Then the constructive learning algorithm gradually adds units between the input and output units, where these new units can form one or more hidden layers. The criteria when to add new units, how many at a time and where the new units will be placed in the network depends on the algorithm chosen. The growth process is finished when the network has learned the training set to an acceptable degree, which is usually a measure of the remaining output error for the training set, as in the non-constructive learning algorithms presented in the previous chapter.

The advantage of constructive learning algorithms over non-constructive ones comes from not having to determine a priori the topology of the network, which in the non-constructive case is usually done based on empirical criteria and may require a trial and error process to determine the best topology for a given problem and a given training set.

Constructive learning algorithms may also give a faster speed of convergence than non-constructive ones since the training is done on an increasingly growing network, so the weights adjusted at each step are less than in the case of training the full network at each step.

A difficulty that remains with constructive learning algorithms is the fact that the final topology achieved by the algorithm is not necessarily the optimal one for the given problem. The final topology is a function of the construction method of the algorithm, i.e. the way units are added to the network, and a construction method that will achieve the optimal network topology, in terms of generalization accuracy, number of layers and number of units, for any given problem, is yet to be found. One approach, which the results shown later in this chapter suggest, is for a given problem to try different construction methods, and select the one that achieves best results (where the criterioncœ for comparison should usually be the generalization accuracy).

This chapter deals with constructive supervised learning algorithm for continuous networks, where by continuous networks it is understood networks with continuous input and output spaces (as opposed to discrete networks with discrete input and output spaces).

The algorithm first discussed in section 3.1 is the Cascade Correlation Learning Algorithm [Fahlman & Lebiere, 1990]. Variations based on the Cascade Correlation Algorithm are then presented. In the variations [Simon, Corporaal & Kerckhoffs, 1992], [Simon, Kerckhoffs & Corporaal, 1992] and [Sjogaard, 1991], a different construction method is employed.

Neural Networks are a powerful computing paradigm that can be used in a wide range of fields. In section 3.2, three types of applications where Neural Networks can be used are presented. They are the inverse kinematic transformations of two and three degrees of freedom robot arm controllers, a pattern recognition problem and the Hamming Code generation problem.

The inverse kinematic transformations are suitable to compare different learning algorithms, since they constitute a real world application and they require multiple outputs. These characteristics distinguish them from commonly used benchmarks, as the XOR/Parity problem and the two spiral problem, where one output is required, and the Encoder/Decoder problem, where there are multiple outputs, but for all output patterns, only one output is on at a time. The pattern recognition problem and the Hamming Code generation problem, which require multiple outputs, are also used for comparison among the learning algorithms. In this way, the comparison among the algorithms is based upon three problems of different nature, belonging to three different areas.

In section 3.3, the performances of the algorithms in this and the previous chapter, in terms of generalization accuracy, learning speed and size of network, are presented and are used to compare the different algorithms. Section 3.4 discusses the results obtained in the previous section and outlines future work.

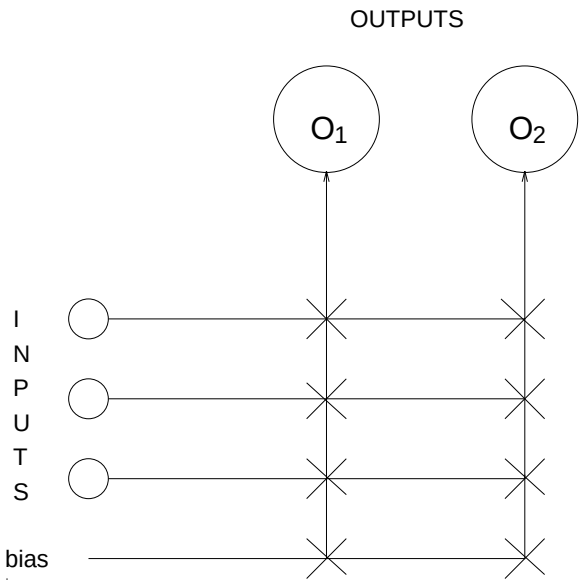


Figure 3.1: Initial Cascade-Correlation Architecture.

3.1 The Algorithms

This section presents and discusses the Cascade Correlation Learning Algorithm and variations based on it.

The Cascade Correlation Learning Algorithm is a constructive supervised learning algorithm for continuous networks. The variations presented here consists of modifications in the construction method of the original algorithm.

3.1.1 The Cascade-Correlation Learning Algorithm

Algorithm Description

The initial architecture of the Cascade Correlation Learning Algorithm is illustrated in Fig. 3.1. It begins as a one layer perceptron, with each input unit connected to each output unit. The "X" shown in the figure represent the connection weights.

The learning algorithm is divided in two parts: training of output units and new unit creation.

In the first part the direct input-output connections are trained as well as possible over the entire training set using the Quickprop learning algorithm [Fahlman, 1988]. Here no back propagation is needed since only one layer is trained.

When no significant error reduction has occurred after a certain number of training cycles, the

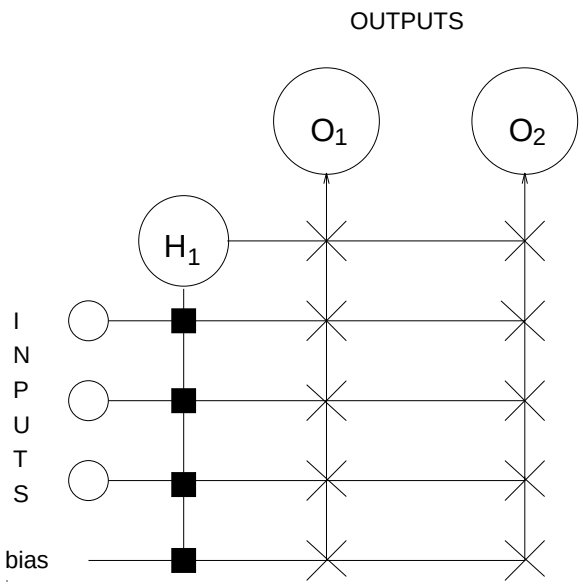


Figure 3.2: Cascade-Correlation Architecture: after 1 hidden unit is added. Boxed connections are frozen, x connections are trained repeatedly.

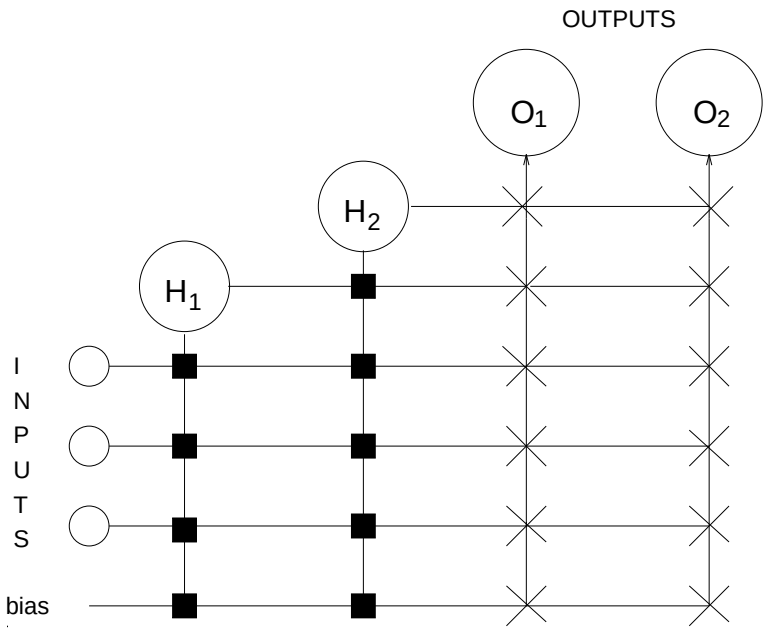


Figure 3.3: Cascade-Correlation Architecture: after 2 hidden units are added.

entire training set is run through the network one last time to measure the error. If the error is less than the specified tolerance the algorithm stops. If it is necessary to reduce this error, then the new unit creation part of the algorithm begins.

In the unit creation part of the algorithm, a new hidden unit, receiving trainable input connections from all of the network's external inputs and from all existing hidden units, is trained as well as possible over the entire training set using the Quickprop learning algorithm. Here the training is done to maximize the correlation of the output of this new hidden unit to the sum of the remaining errors at the outputs.

When no significant error reduction has occurred after a certain number of training cycles, the output of this new unit is connected to all output units of the network. Fig. 3.2 shows the network after one hidden unit has been added. Fig. 3.3 shows the network after two hidden units have been added.

The way a new hidden unit is trained by the Quickprop learning algorithm, to maximize the correlation of its output to the sum of the remaining errors at the outputs, is explained below. Note that the new hidden unit is trained before it is added to the network.

The goal of the training is to maximize S, the sum over all output units o of the magnitude of the correlation between o^h, the new hidden unit's output value, and E_o, the residual output error observed at output unit o. S is defined as

$$S = \sum_o \left| \sum_p (o_p^h - \overline{o^h})(E_{p,o} - \overline{E_o}) \right|$$

where $E_{p,o}$ is the error at output o for input pattern p, o_p^h is the output of the new hidden unit for input patter p, and the quantities $\overline{o^h}$ and $\overline{E_o}$ are the values of o^h and E_o averaged over all patterns.

To maximize S, the partial derivative of S with respect to each of the new unit's incoming weights, w_i^h, is computed. This gives

$$\partial S / \partial w_i^h = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E_o}) f_p^{'} I_{i,p}$$

where σ_o is the sign of the correlation between the new unit's output value, o_p^h, and the error of output o, $E_{p,o}$. $f_p^{'}$ is the derivative for pattern p of the new unit's activation function with respect to the sum of its inputs. $I_{i,p}$ is the input the new unit receives from unit i for pattern p, where unit i is either an input unit or the output of a previously added hidden unit.

Now, a gradient ascent is performed to maximize S using again the Quickprop learning algorithm. Here the only weight that are adjusted by the Quickprop algorithm are the new hidden unit's incoming weights, w_i^h. This means the Quickprop algorithm is used for single layer training. The incoming weights of previously added hidden units remain frozen. In this way no backpropagation is required.

When S stops improving, the new unit's output is connected to each of the network's output units and its input weights remain frozen for the rest of the training. Now the training of the output units starts again, this time with the extra weights coming from the added hidden unit.

To speed up the training of a new unit, a pool of candidate units are trained in parallel, each starting with a different set of random weights. The candidate unit that scores the highest S is chosen as the new unit to be added to the network.

Since the weights of the hidden units don't change, hidden units' output values can be cached.

By correlating the value of the hidden unit to the remaining error at the output units, the output units are able to reduce the error in the next training round since for the input patterns that cause high errors in the outputs, the hidden unit will produce a high value.

Discussion of Cascade-Correlation

The Cascade-Correlation learning algorithm avoids the need to propagate the error backwards though the network, as in the Backpropagation algorithm. This is achieved though training only one layer at a time to decrease the error function, while other weights in the network remain frozen.

By keeping part of the weight space frozen, the moving target problem discussed in chapter 2 for the Backpropagation algorithm, should be reduced.

The step-size problem found discussed in chapter 2 for the Backpropagation algorithm is avoided here by dynamically adding new hidden units to the network. The algorithm won't get stucked at a local minimum because the residual error will cause the algorithm to add a new unit, changing the weight space. This avoids also the the problem of oscillating around the global minimum.

The Cascade-Correlation algorithm still lacks a bound on the number of units it produces. The size of the network is not know in advance and the lack of a bound on the maximum number of units can be a problem for the implementation of the algorithm.

Second Order Cascade-Correlation

A variation made on Cascade-Correlation [Sjogaard, 1991] consists of allowing the network to grow horizontally instead of vertically. In other words, every time a hidden unit is added, it is added to the one and only hidden layer. This is achieved by letting the new hidden units receive connections only from the input and bias units and not from previously added hidden units as in the original version. This is shown in Fig 3,4.

In this case the network build is a two-layer perceptron. Two-layer perceptrons can be used in a way range of classification problems [Sjogaard, 1991].

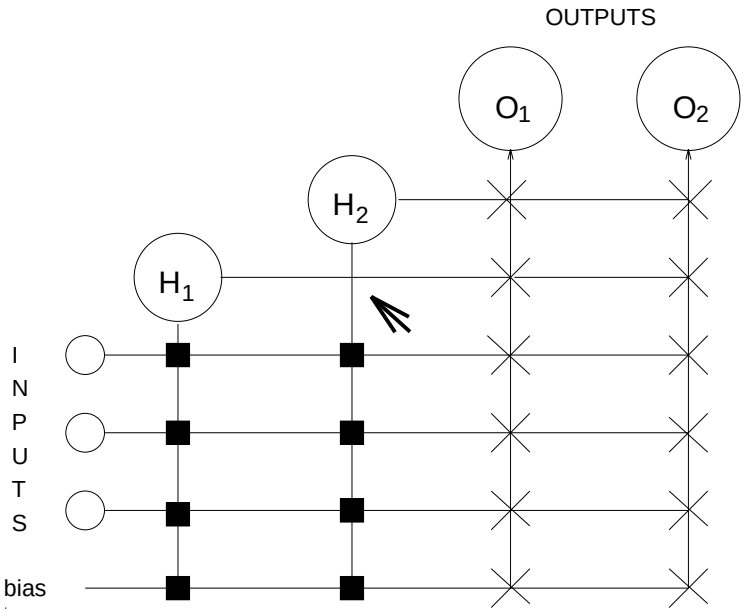


Figure 3.4: Second-Order Cascade-Correlation Architecture

3.1.2 Modified Cascade-Correlation

Algorithm Description

The new version of the Cascade-Correlation algorithm [Simon, Corporaal & Kerckhoffs, 1992] changes the way new units are added to the network. Here a number of new units equal to the number of outputs is added every time the training of output units cannot decrease the error. This should accelerated the convergence of the network, since each hidden unit helps correct the remaining error at one output and not the sum at all outputs, as in the original algorithm.

New units are denoted $H_{l.o}$. The index l denotes the layer to which the hidden unit belongs to. The index o denotes to which output the hidden unit belongs or correlates to ($o = 1, 2, ..., N_o$, where N_o equals the number of output units).

Each new unit is independent of the other new units in the same layer so they can be trained in parallel. Each new unit receives trainable input connections from all of the network's external inputs and from previously added units with the same o index (see Fig. 3,6).

Before the outputs of the new units are connected, a number of passes are run over the training set, adjusting each new unit's input weights after each pass. The goal of this adjustment is to maximize each S_o, the sum of the magnitude of the correlation between $o^{l.o}$, the output value of the new hidden unit $H_{l,o}$, and E_o, the residual output error observed at output unit o. S_o is defined as

$$S_o = \left| \sum_p (o_p^{l.o} - \overline{o^{l.o}})(E_{p,o} - \overline{E_o}) \right|$$

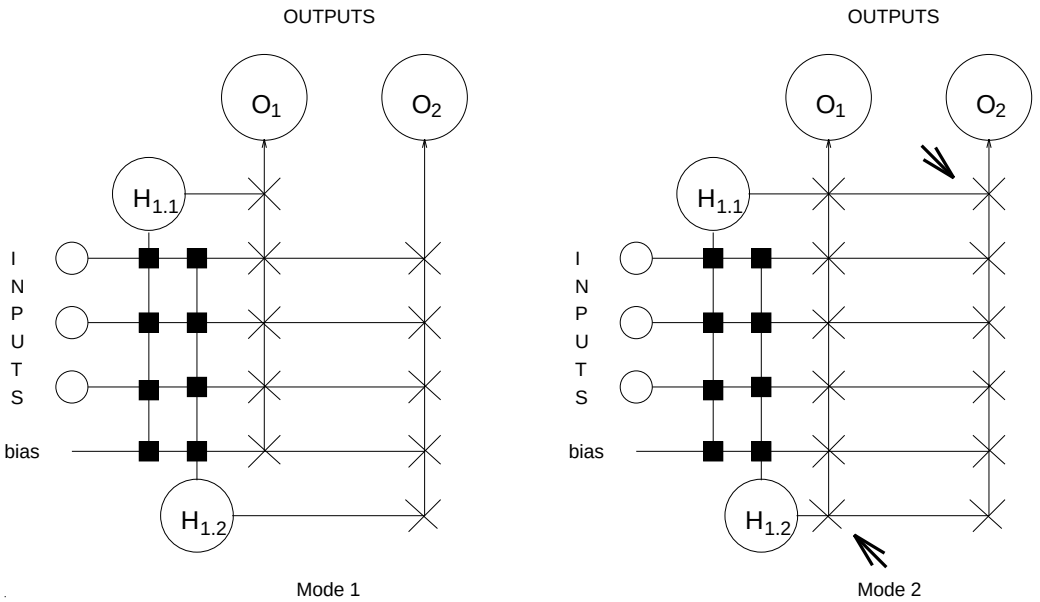


Figure 3.5: New Architecture after 1 groups of 2 hidden units have been added: (a) Mode 1, (b) Mode 2.

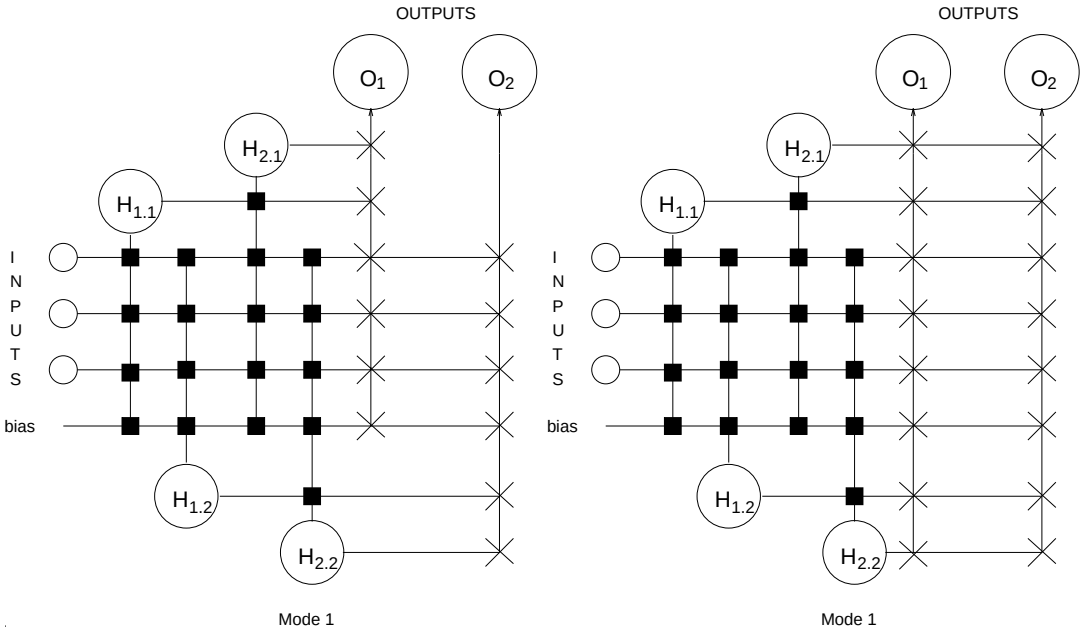


Figure 3.6: New Architecture after 2 groups of 2 hidden units have been added: (a) Mode 1, (b) Mode 2.

where $E_{p,o}$ is the error at output o for input pattern p, $o_p^{l.o}$ is the output of the new hidden unit $H_{l.o}$ for input patter p, and the quantities $\overline{o^{l.o}}$ and $\overline{E_o}$ are the values of $o^{l.o}$ and E_o averaged over all input patterns.

To maximize S_o, the partial derivative of S_o with respect to each incoming weight, $w_i^{l.o}$, to new hidden unit $H_{l.o}$, is computed. This gives

$$\partial S_o / \partial w_i^{l.o} = \sum_p \sigma_o (E_{p,o} - \overline{E_o}) f_p^{'} I_{i,p}$$

where σ_o is the sign of the correlation between the new unit's output value, $o_p^{l.o}$, and the error of output o, $E_{p,o}$. $f_p^{'}$ is the derivative for pattern p of the new unit's activation function with respect to the sum of its inputs. $I_{i,p}$ is the input the new unit receives from unit i for pattern p, where unit i is either an input unit or the output of a previously added hidden unit belonging to the same output, $o_p^{l'.o}$ for $l' < l$.

Now, a gradient ascent is performed to maximize each S_o, using the Quickprop learning algorithm with only the incoming weights to input layer l, $w_i^{l.o}$, being adjusted.

As in the original algorithm, a pool of candidate units, here for each new unit, are trained in parallel, each starting with a different set of random weights.

Two modes exist for the way the hidden units are connected to the output units. In the first mode, when all S_o stop improving, each new unit's output is connected only to the respective output unit (Fig. 3,5(a) and Fig. 3.6(a)). In the second mode, when all S_o stop improving, each new unit's output is connected to all output units (Fig. 3,5(b) and Fig. 3.6(b)). After the hidden units have been connected, either in mode 1 or 2, their input weights remain frozen and the algorithm starts again with the training of the output units.

Discussion of Modified Cascade-Correlation

The Modified Cascade-Correlation changes the way the network is constructed. Now the number of hidden units added at a time is a function of the number of output units. The fact that each added hidden unit is trained to correlate to the error of only one output unit, should lead to a faster correction of the errors at the output units. The fact that the training of the hidden units can occur in parallel should make this algorithm faster for problems with multiple outputs. Notice that in case the network has only one output unit, the original algorithm and the new version are exactly the same.

Second Order Modified Cascade-Correlation

Here, as in the original Cascade-Correlation, the second-order version is achieved by connecting the hidden units only to the output and bias units as shown in Fig. 3.7.

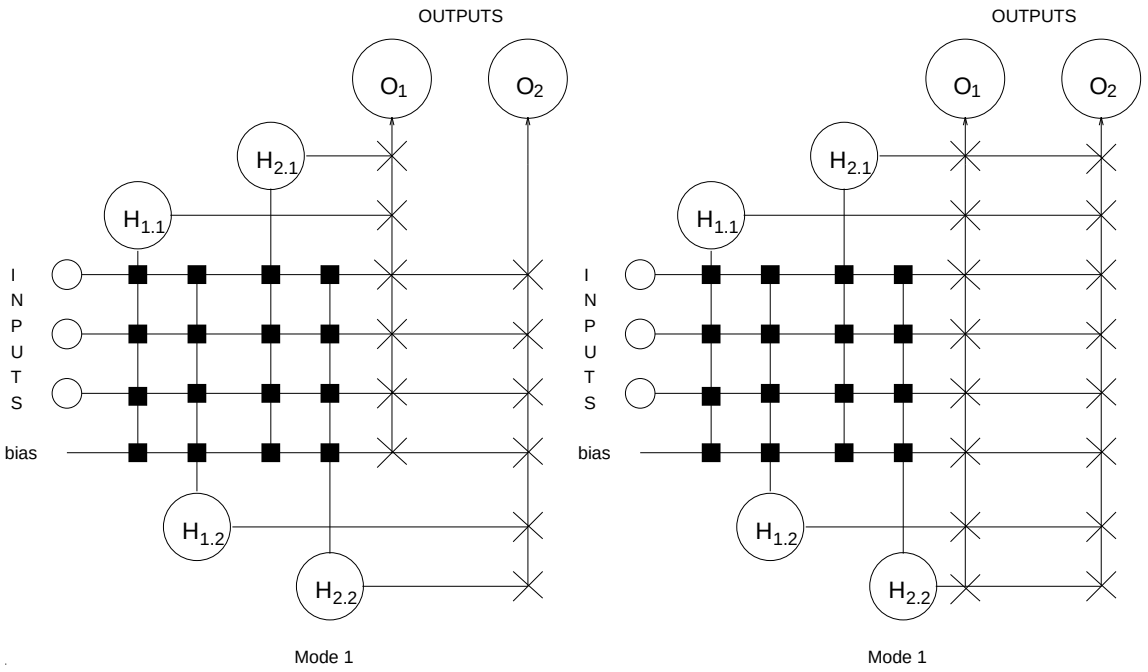


Figure 3.7: Second-Order New Architecture

3.2 The Applications

Neural Networks are a powerful computing paradigm that can be used in a wide range of fields. In this chapter three types of applications where Neural Networks can be used are presented. They are the the inverse kinematic transformations of two and three degrees of freedom robot arm controllers, a pattern recognition problem and the Hamming Code generation problem.

3.2.1 Inverse Kinematic Transformations for Robot Control

Control Engineering is one area for which neural networks are attractive because of their ability to learn, to approximate functions, to classify patterns and because of their potential for massive parallel hardware implementation [Antsaklis, 1992].

Robot Controller

One application in this area is the control of a robot arm. Fig. 3.8 shows shows the configuration of such a robot control.

Consider a jointed two degrees of freedom robot arm that operates in a plane, as shown in Fig. 3.9(a). To make the robot arm move, the coordinates of the desired end point (x,y) are fed to a robot controller so that it generates the required joint angles (ϕ_1, ϕ_2). The solution to the inverse

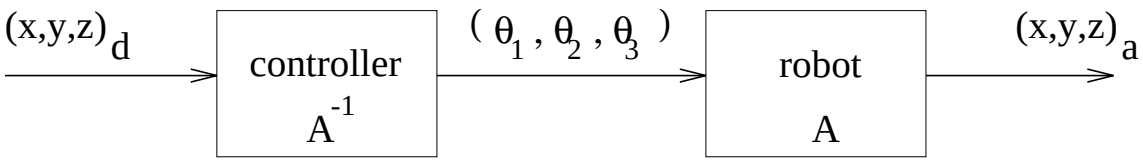


Figure 3.8: Robot Controller Configuration

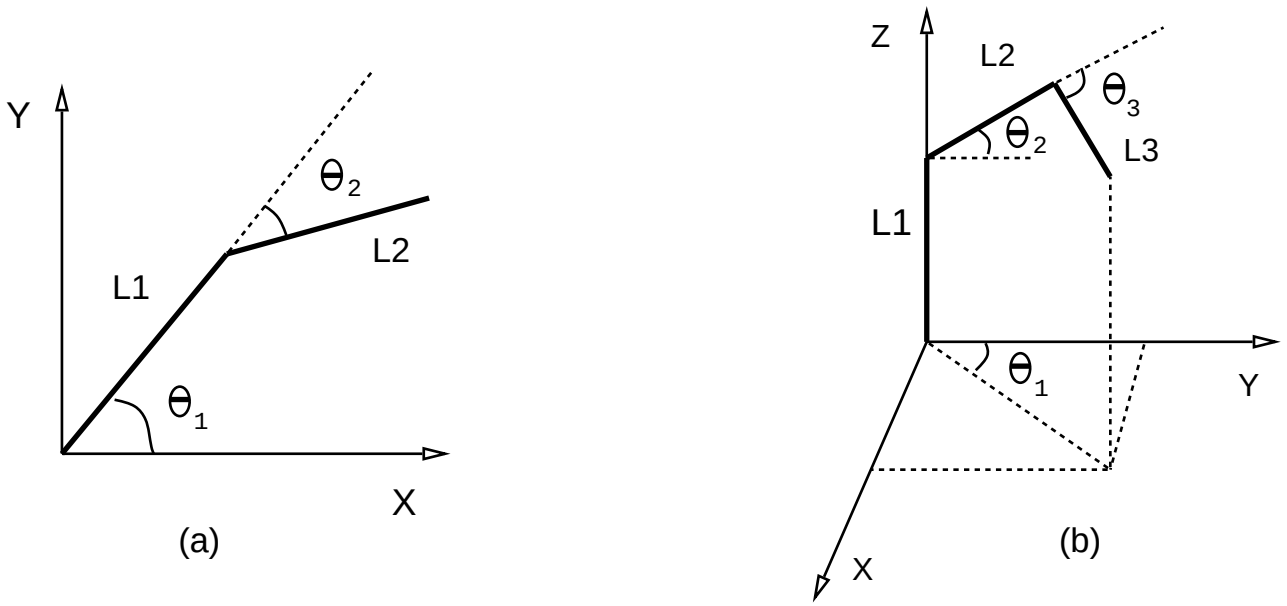


Figure 3.9: Robot Arm: (a) two degrees of freedom, (b) three degrees of freedom.

kinematic transformation that maps the end position to the joint angles is given by

$$\begin{aligned} cos\Theta_2 &= \frac{x^2+y^2-L_1^2-L_2^2}{2L_1L_2} \\ \Theta_1 &= arctan(\frac{y}{x}) - arctan(\frac{L_2 sin\Theta_2}{L_1+L_2 cos\Theta_2}) \end{aligned}$$

Fig. 3.9(b) presents the three-dimensional case where the robot arm consists of three segments. In this case the solution is given by

$$\begin{aligned} \Theta_1 &= -arctan(\frac{x}{y}) \\ cos\Theta_3 &= -\frac{x^2+y^2+(z-L_1)^2-L_2^2-L_3^2}{2L_2L_3} \\ \Theta_2 &= arctan(\frac{z-L_1}{\sqrt{x^2+y^2}}) - arctan(\frac{L_3 sin\Theta_3}{L_2+L_3 cos\Theta_3}) \end{aligned}$$

A conventional robot controller relays on the mathematical model of the coordinate transformations required to direct the robot arm to the desired position. In real world situations the wearing down

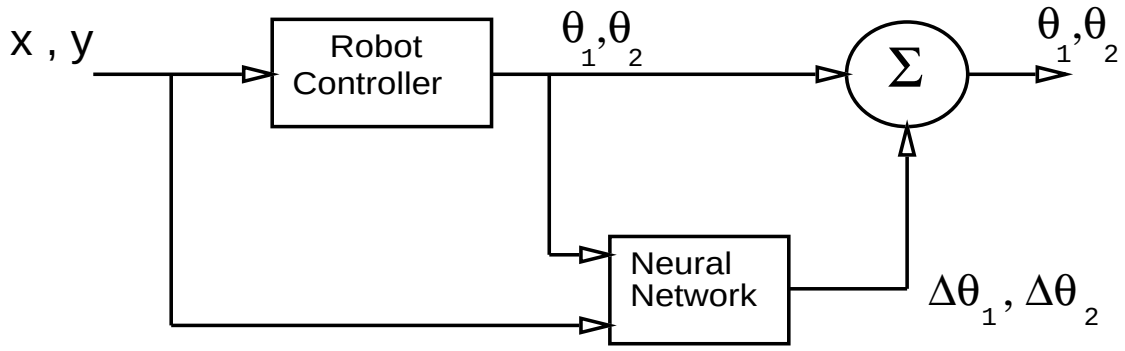


Figure 3.10: Controller with Corrective Neural Network.

of the robot and external disturbances cause the the model to be inaccurate so the controller does not achieve the right coordinate transformations.

A neural network can be used as robot controller by making it learn the inverse coordinate transformations of the robot arm [Rabelo & Avula, 1992]-[Bassi & Bekey, 1989]. The advantage of using neural networks is that no mathematical model of the robot, the wear out and external disturbances is required. The neural network can learn the coordinate transformation of the robot from a set of transformation examples from real robot experiments.

Here, the mathematical model of the coordinate transformations is used to generate the training sets and test sets. Ideal conditions with no disturbances are considered.

Corrective network

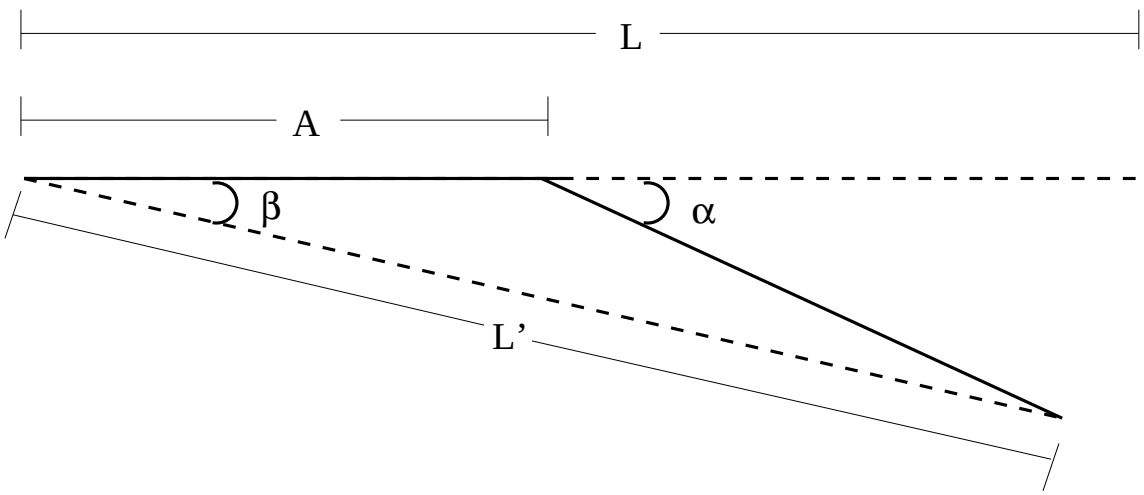
A related problem presented in [Josin, Charney & White, 1988] is the use of a neural network as a corrective network for the controller of a damaged robot arm. The configuration of such a network is shown in Fig. 3.10.

Here the damage is modeled by a bend at a certain point along the robot arm, as shown if Fig. 3.11.

In this case the neural network learns the difference in angles $(\Delta\Theta_1, \Delta\Theta_2)$ needed to correct the error between the theoretically calculated joint angles of the kinematic transformation and the actual angles required to place the robot arm at the desired position according to this model.

In this case the input to the neural network consists of the desired end coordinates and the uncorrected joint angles calculated by the robot controller. The outputs are the joint angle corrections.

In general, the error function of a controller is not precisely known. Therefore it is not possible to directly determine the corresponding correcting output signals of the correcting network (necessary to generate the training set)for a given set of inputs. This problem can be solved in the following way. A training set for the correcting network is composed of a set of input points (spatial locations and controller joint angle response to those locations) and targets (difference between the controller output and the input of the robot that is required to reach the given spatial positions). For each



$$L' = \sqrt{A^2 + (L - A)^2 - 2A(L - A)\,\mathrm{Cos}(\pi - \alpha)}$$

$$\mathrm{Sin}\,\beta = \frac{L-A}{L'}\,\mathrm{Sin}\,\alpha$$

Figure 3.11: Model of a Bent Arm

spatial position, two sets of joint angles are required to calculate the difference that is used as target. One set contains the controller generated angles, the other set contains the actual required angles to reach the spatial position. The problem encountered is that the inverse transformation for a given spatial position is , due to unknown changes in the robot, not know. This is solved by using the controller twice. First the controller output is determined for a given position in space. This will move the robot to a position in space that has a slight position deviation from the input position. By determining the position of the robot, a set of joint angles and the corresponding spatial position is formed. Next the actual spatial position is the input to the controller. This will move the robot to a new spatial position using the newly generated joint angles. The measured spatial position is used as input part of the training set. The target part of the training set is calculated by subtracting the controller generated joint angles for the input point from the actual joint angles corresponding to the input point.

3.2.2 Hamming Code Generation

At present, the benchmark that appears most often in the neural network literature is the exclusive-or problem, also called the XOR problem: given two binary inputs, the output will turn on if one of the inputs is on, but not if both inputs are on. This problem can be generalized to N-input parity problem, where the output is on if an odd number of inputs are on [Tesauro & Janssens, 1988].

Here, another benchmark is presented, the Hamming Code Generation problem. The Hamming Code Generation problem is related to XOR/Parity problem. It represent a more general problem, requiring multiple outputs, for which the XOR/Parity problem is a special case, i.e. a Hamming Code Generation problem with one output becomes the parity problem.

The Hamming Code Generation problem goes one step further than the XOR/Parity problem since it can be defined with any number of outputs. This makes it suitable to test how learning algorithms scale to multiple outputs. How learning algorithms scales to multiple outputs is an essential point of comparison since most real live applications require multiple outputs.

Problem Description

The Hamming Code Generation problem consists of determining the r check bits of the Hamming Code [Hamming, 1950] for a binary input of m bits, where $m + r + 1 \leq 2^r$. Each check bit is the parity bit of a different set of data bits, where the r sets are not disjoint. The uses of redundant information, i.e. the fact that each data bit belongs to more than one set of data bits from which the parity is computed, makes it possible to localize and so to correct faults in the data.

The number of bit positions in which two codewords differ is called the *Hamming Distance*. It significance is that if two codewords are a Hamming Distance d apart, it will require d single bit errors to convert one into the other. Given a list of codewords, the Hamming Distance of the complete code is equal to the Hamming Distance of the two codewords whose Hamming Distance is minimum.

To correct d errors, a code with a Hamming Distance of $2d + 1$ is needed. This since the original codeword will still be closer than any other codeword, so it can be uniquely determined.

To design a code with m message bits and r check bits that will allow all single errors to be

Character	ASCII code	Hamming code
H	1001000	00110010000
a	1100001	10111001001
c	1100011	11111000011
d	1100100	11111001100
m	1101101	11101010101
n	1101110	01101010110

Table 3.1: Hamming code of ASCII characters

corrected, consider that each of the 2^m legal messages has $n = m+r$ illegal codewords at a distance 1 from it (this are formed by systematically inverting each of the n bits in the n-bit codeword formed from the m message bits). Thus each of the 2^m legal messages requires $n + 1$ bit patterns dedicated to it. Since the total number of bit patterns is 2^n, it is required that $(n + 1)2^m \leq 2^n$. Using $n = m + r$ this requirement becomes $(m + r + 1) \leq 2^n$. Given m, this puts a lower limit on the number of check bits needed to correct single errors.

This lower limit can be achieved with the following method [Hamming, 1950]:
the bits of the codeword are numbered consecutively, starting with bit 1 at the left end. The bits that are powers of 2 (1, 2, 4, 8, 16, etc.) are check bits. The rest (3, 5, 6, 7, 9, etc.) are filled up with the m data bits. Each check bit forces the parity of some collection of bits, including itself, to be even. A bit may be included in several parity computations. To see which check bits the data bit in position k contributes to, rewrite k as a sum of powers of 2. For example, $11 = 1+2+8$ and $29 = 1 + 4 + 8 + 16$. A bit is checked by just those check bits occurring in its expansion (e.g. bit 11 is checked by bits 1, 2, and 8).

When a codeword arrives, the receiver initializes a counter to zero. It then examines each check bit k (k=1, 2, 4, 8, ...) to see if it has the correct parity. If not, it adds k to the counter. If the counter is zero after all the check bits have been examined (i.e., if they were all correct, the codeword is accepted as valid. If the counter is nonzero, it contains the number of the incorrect bit. For example, if check bits 1, 2, and 8 are in error, the inverted bit is 11, because it is the only one checked by bits 1, 2, and 8. Table 3.1 shows some 7-bit ASCII characters encoded as 11-bit codewords using a Hamming code. The data are found in bit positions 3, 5, 6, 7, 9, 10, 11.

3.2.3 Pattern Recognition

The following application is a pattern recognition problem, where it is required to identify one of eight different patterns, even if a pattern is rotated 90, 180 or 270 degrees.

The patterns are presented in a 3x3 grid and the are are shown in Fig. 3.12. The binary code shown under each pattern uniquely identify that pattern and should then be the output of the network

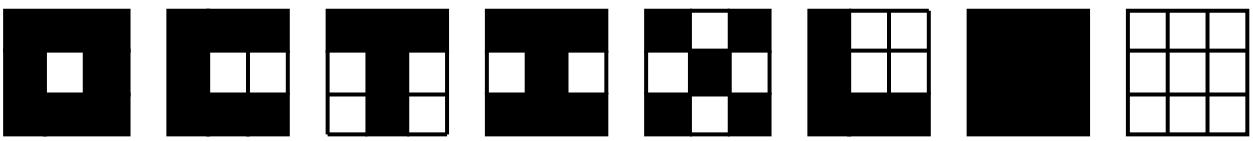


Figure 3.12: Patterns with corresponding binary codes

when that pattern is presented as input, invariant of rotation.

3.3 The Simulation Results

The results presented below give a comparison between the Cascade-Correlation algorithm and the Modified Cascade-Correlation algorithm. For both algorithms the high order version was used.

3.3.1 Inverse Kinematic Transformations for Robot Control

We compare the learning speed of the algorithms as a function of training set size and as a function of the output tolerance, given in radians. The training set consists of uniformly spread points in the first quadrant of the X-Y plane for the robot with two degrees of freedom and for the corrective network, and for the three degrees of freedom robot in the positive X-Y-Z space. All robot links, L_i, are $\frac{1}{2}\sqrt{2}$ length units. The training of the output units uses a linear function and the training of the hidden units the symmetric Sigmoid function. This combination gave the best results. The same training parameters are used in all simulations. The number of candidate units is eight. All candidate units of all new units are trained in parallel. All simulations are run on a Sun4 workstation.

Robot Arm Controller
Table 3.2 compares the average in training epochs, defined as the number of passes through the training set, and number of hidden units for different sizes of the robot controller's training set. The tolerance for the output angles was 0.025 rad. The notation $C - C$ indicates Cascade-Correlation, $Mode$ 1 and $Mode$ 2 indicate the two modes of the Modified Cascade-Correlation and the success rate p/q indicates that p out of q trials were successful, where each trial starts from the initial state with different random weights. The speedup is given for the average epochs.

The results show that the new version learns faster and scales better when the training set increases. The speedup in average epochs increases with the training set size. In the two degrees of freedom case, when the training set size is more than 81, Mode 2 is not only faster but requires the same number or less hidden units than the original algorithm. Notice that for the Cascade-Correlation algorithm, the average number of hidden units is equal to the average number of hidden layers, since each layer has one hidden unit. For the Modified Cascade-Correlation algorithm, the number of average hidden layers is equal to half the number of average hidden units in the two dimensional case, and a third of the number of average hidden units in the three dimensional case. This since for two-output problems, two hidden units are added to each hidden layer and for three-output problems, three hidden units are added to each hidden layer. This means that the Modified Cascade-Correlation algorithm results also in less deep networks.

Mode 2 architecture, where each output receives connections from all hidden units , performs better than Mode 1. This could be explained by the existence of a correlation between the outputs, so hidden units that correlate to one output help other outputs.

We tested the network with 100 previously unseen points forming a circle. Table 3.3 gives the average, maximum and standard deviation of the error for different sizes of the training set. The error is measured as the Euclidean distance between the actual end point of the robot arm and the desired end points in length units. The tolerance during the learning was 0,025 rad. The results are given for the original Cascade-Correlation and for the new algorithm with Mode 2. The original

Training Set	Algorithm	Avg. Units	Avg. Epochs	Success	SpeedUp
4	C-C	1	91	10/10	1
	Mode 1	2	106	10/10	0.86
	Mode 2	2	103	10/10	0.88
9	C-C	5	504	10/10	1
	Mode 1	8	529	10/10	0.95
	Mode 2	6	402	10/10	1.25
25	C-C	12	1753	10/10	1
	Mode 1	23	1419	10/10	1.24
	Mode 2	16	1123	10/10	1.56
81	C-C	30	4368	9/10	1
	Mode 1	33	2042	10/10	2.14
	Mode 2	27	1833	10/10	2.38
121	C-C	31	6200	10/10	1
	Mode 1	40	2534	10/10	2.45
	Mode 2	31	2159	10/10	2.87

(a)

Training Set	Algorithm	Avg. Units	Avg. Epochs	Success	SpeedUp
4	C-C	1	143	10/10	1
	Mode 1	2	163	10/10	0.88
	Mode 2	2	134	10/10	1.07
8	C-C	3	364	10/10	1
	Mode 1	8	318	10/10	1.14
	Mode 2	6	249	10/10	1.46
27	C-C	17	2386	10/10	1
	Mode 1	37	1584	10/10	1.51
	Mode 2	24	1332	10/10	1.79
64	C-C	36	5545	8/10	1
	Mode 1	65	2806	9/10	1.98
	Mode 2	49	2688	10/10	2.06

(b)

Table 3.2: Average training epochs: (a) two degrees of freedom, (b) three degrees of freedom

algorithm could not converge with a training set size of 289 patterns. The results show a decrease in average error and standard deviation as the set size increases. Both algorithms exhibit comparable generalization capabilities.

Comparison with Back-propagation

For the two degrees of freedom case, we can compare the accuracy of generalization of the new algorithm (Mode 2) with that obtained by Josin [Josin, Charney & White, 1988], who used back-propagation. In [Josin, Charney & White, 1988], the architecture used consisted of one hidden layer of 32 neurons, 2 input and 2 output units. Josin reports that the test set, 100 points on a circle of radius 0.2 length units, was within an Euclidean distance of 0.025 length units from the desired end points when the training set, a rectangular grid covering the test circle, was of 8 to 25 points. Less training points could not achieve that error tolerance. Using the same training and test sets, we obtained the same error tolerance of 0.025 length units with a training set of 8 points and the network built by the algorithm had 4 hidden neurons. With a test set of 9 points an error tolerance of 0.015 length units was achieved and the network required 9 hidden units. Finally with a test set of 25 points, an error tolerance of 0.012 length units was achieved with 9 hidden units.

Corrective Network

Table 3.4 presents the average in training epochs and number of hidden units for different tolerances in the corrective network.

The results show that the new version also scales better as low tolerance is required. For a tolerance of $5x10^{-3}$ or less the new algorithm requires less hidden units and converges faster than the original.

Train Set	Algorithm	Avg. Error	Max. Error	Std. Dev.
9	C-C	0.0386	0.0869	0.0425
	Mode 2	0.0503	0.1736	0.0620
25	C-C	0.0298	0.0903	0.0351
	Mode 2	0.0283	0.0928	0.0331
81	C-C	0.0163	0.0501	0.0197
	Mode 2	0.0228	0.1557	0.0300
121	C-C	0.0215	0.0590	0.0253
	Mode 2	0.0189	0.0539	0.0234
289	C-C	--	--	--
	Mode 2	0.0161	0.0801	0.0137

(a)

Train Set	Algorithm	Avg. Error	Max. Error	Std. Dev.
8	C-C	0.2172	0.2848	0.2195
	Mode 2	0.2713	0.3689	0.2794
27	C-C	0.0837	0.7346	0.1479
	Mode 2	0.0667	0.1245	0.0712
64	C-C	0.1228	0.4078	0.1560
	Mode 2	0.0734	0.2250	0.0887
125	C-C	0.0447	0.2261	0.0637
	Mode 2	0.0364	0.1235	0.0262

(b)

Table 3.3: Position Errors: (a) two degrees of freedom, (b) three degrees of freedom

Tolerance	Algorithm	Avg. Units	Avg. Epochs	Success
$5x10^{-2}$	C-C	0	6	10/10
	Mode 1	0	6	10/10
	Mode 2	0	6	10/10
$5x10^{-3}$	C-C	4	1000	10/10
	Mode 1	3	286	10/10
	Mode 2	2	238	10/10
$5x10^{-4}$	C-C	15	3439	7/10
	Mode 1	10	1117	10/10
	Mode 2	8	984	10/10
$5x10^{-5}$	C-C	60*	13270	0/10
	Mode 1	17	1892	10/10
	Mode 2	10	1201	10/10

* Max. no. Units

(a)

Tolerance	Algorithm	Avg. Units	Avg. Epochs	Success
$5x10^{-2}$	C-C	0	8	10/10
	Mode 1	0	8	10/10
	Mode 2	0	8	10/10
$5x10^{-3}$	C-C	13	2951	10/10
	Mode 1	2	327	10/10
	Mode 2	3	359	10/10
$5x10^{-4}$	C-C	31	6982	4/10
	Mode 1	17	2036	10/10
	Mode 2	15	1930	10/10
$5x10^{-5}$	C-C	60*	13466	0/10
	Mode 1	37	4488	10/10
	Mode 2	22	2831	10/10

(b)

Table 3.4: Average training epochs, (a) 9 train cases. (b) 16 train cases.

Tolerance	Avg. Error	Max. Error	Std. Dev.
$5x10^{-2}$	0.0251	0.0320	0.0256
$5x10^{-3}$	0.0024	0.0049	0.0027
$5x10^{-4}$	0.0058	0.0122	0.0067
$5x10^{-5}$	0.0068	0.0153	0.0083

(a)

Tolerance	Avg. Error	Max. Error	Std. Dev.
$5x10^{-2}$	0.0089	0.0130	0.0095
$5x10^{-3}$	0.0018	0.0043	0.0021
$5x10^{-4}$	0.0047	0.0246	0.0067
$5x10^{-5}$	0.0047	0.0107	0.0054

(b)

Table 3.5: Position Errors: (a) 9 train cases, (b) 16 train cases

m	r	Algorithm	Average Units	Average Epochs	Speedup
3	3	C-C	3	170	1
		Mode 1	3	55	3.09
		Mode 2	3	55	3.09
4	3	C-C	3.5	319	1
		Mode 1	3	86	3.71
		Mode 2	3	90	3.54
5	4	C-C	4,9	455	1
		Mode 1	8	197	2.31
		Mode 2	8	203	2.24
6	4	C-C	6.8	590	1
		Mode 1	8.4	195	3.03
		Mode 2	8,4	208	2.84

Table 3.6: Hamming Code: average training epochs for m data bits and r check bits

For a tolerance of $5x10^{-5}$ the original algorithm is not able to converge after adding 60 hidden units. Again Mode 2 architecture performs better than Mode 1.

Table 3.5 gives the average, maximum and standard deviation of the error for different tolerances for the test circle. The results are again for the new algorithm with Mode 2. The results show the well known over-training problem. When the training set is too accurately learned, the network generalizes less accurately.

3.3.2 Hamming Code Generation

Table 3.6 presents the average in training epochs and number of hidden units for different sizes of m and r. The results were obtained with Sigmoid output units and hidden units whose outputs are a Gaussian function of weighted inputs. This combination gave the fastest learning. The number

Algorithm	Average Units	Average Epochs	Speedup
C-C	2.3	325	1
Mode 1	4.2	233	1.40
Mode 2	3	177	1.84

Table 3.7: Pattern Recognition: average training epochs

of candidate units is eight. The tolerance is 0.4.

The combination $m = 4$ $r = 3$ is optimal: $m + r + 1 = 2^r$.

3.3.3 Pattern Recognition

Table 3.7 presents the average in training epochs and number of hidden units for this problem. The results were again obtained with Sigmoid output units and Gaussian hidden units. The number of candidate units is eight. The tolerance is 0.4.

3.4 Discussion and Future Work

The Cascade-Correlation Learning Algorithm and modified versions presented in this chapter has the advantages that the algorithms determine the size and topology of the network and also automatically increases its size when lower tolerance is required or more training cases are used. This flexibility makes the proposed algorithms attractive for real-world applications and solves the problem of determining the architecture of the network. The algorithm and architecture allow incremental learning.

3.4.1 Inverse Kinematic Transformations for Robot Control

For the robot control problems, Modified Cascade-Correlation was shown to learn quicker, scale better to more training cases and to lower tolerance, and to have a higher success rate than the original Cascade-Correlation. The robot arm control with two degrees of freedom allowed a comparison with previous results using Backpropagation. In this case the accuracy of generalization achieved by the Modified Cascade-Correlation algorithm is equal or better and with a much smaller network than achieved by Josin using back-propagation. Also no trial and error stage is needed to determine the amount and size of the hidden layers.

The results show that only an increase in the training set size improves the accuracy of the controller. Learning the training set with very high accuracy produces the overtraining problem, where for points not in the training set the accuracy decreases.

With the Modified Cascade-Correlation algorithm, connections from hidden units to all outputs make the convergence faster (Mode 2). This could be explained by the existence of a correlation between the outputs, so hidden units that correlate to one output help other outputs.

The results obtained show that a method is required to determine the tolerance in training that will maintain the best generalization capability of the neural network. Further research should focus on this problem.

3.4.2 Hamming Code Generation

For the Hamming Code Generation problem, Modified Cascade-Correlation was shown to learn faster than the original Cascade-Correlation,

For this problem the original Cascade-Correlation algorithm achieved a smaller number of hidden units than the Modified Cascade-Correlation algorithm. On the other hand, since the original Cascade-Correlation algorithm has one unit per hidden layer, the number of layers is less in the Modified Cascade-Correlation algorithm.

Another important point in the results achieved is that for the Hamming Code Generation problem, the Modified Cascade-Correlation algorithm with Mode 1 architecture is faster in learning than Mode 2. This is opposite to the result obtained for the Inverse Kinematic Transformations for Robot Control. This could be explained by the lack of correlation between the check bits.

3.4.3 Pattern Recognition

For the Pattern Recognition problem, Modified Cascade-Correlation was shown to learn faster than the original Cascade-Correlation,

For this problem the original Cascade-Correlation algorithm achieved a smaller number of hidden units than the Modified Cascade-Correlation algorithm. On the other hand, there are less hidden layers in the Modified Cascade-Correlation algorithm.

In this problem, so as with the Inverse Kinematic Transformations for Robot Control, the Modified Cascade-Correlation algorithm with Mode 2 architecture is faster in learning than with Mode 1 architecture. This again could be explained by the existence of a correlation between the outputs.

3.4.4 Future Work

The results presented here show that the answer to the question which algorithm is the best or which is the best architecture is dependent on the type of problem. Future work should test the algorithms presented in this chapter in other types of real-word applications.

Discrete Constructive Supervised Learning

4

In the previous chapter, the class of constructive learning was presented. There constructive supervised learning algorithm for continuous networks were discussed.

This chapter deals with constructive supervised learning algorithm for discrete networks, where by discrete networks it is understood networks with discrete input and output spaces. Five learning algorithms are presented and discussed in sections 4.1. The first four algorithms are the Pocket Algorithm [Gallant, 1990]. the Tilling Algorithm [Mezard & Nadal, 1989], the Tower Algorithm [Gallant, 1990] and the Upstart Algorithm [Frean, 1990]. The fifth, the BINCOR Algorithm, is a new learning algorithm, introduced here. In section 4.2 two types of applications are presented. They are the Monk's problems and random boolean functions. Finally, in section 4.3, the performances of the algorithms in terms of generalization accuracy, learning speed and size of network are presented.

4.1 The Algorithms

4.1.1 The Pocket Algorithm

The Pocket algorithm [Gallant, 1990] is a variation on the Perceptron algorithm. So as the Perceptron algorithm, the Pocket algorithm is used for single unit training.

Although the Pocket algorithm is not a constructive algorithm, it is presented here since it is used by the Tilling algorithm, the Tower algorithm and the BINCOR algorithm, which are constructive algorithms discussed in this chapter.

The background for the Pocket algorithm is the following: since a single unit cannot correctly classify all training patterns of non-linearly separable problems, the best that can be hoped for is a set of weights that correctly classifies as large a fraction of the training patterns as possible. Such a set of weights is called *optimal*.

The Perceptron algorithm is not well behaved for non-linearly separable problems. While it will eventually visit an optimal set of weights, it will not converge to any set of weights. The algorithm can even go from an optimal set of weights to a worst-possible set in one iteration, regardless of how many interactions have been taken previously. The Pocket algorithm makes perceptron learning well behaved by adding positive feedback in order to stabilize the algorithm.

The basic idea of the Pocket algorithm is to run the Perceptron algorithm, while keeping an extra set of weights in a "pocket". Here the Perceptron algorithm is run with a random presentation of the training patterns. Whenever the perceptron weights have a longest run of consecutive correct classifications of the randomly selected training patterns, these perceptron weights replace the pocket weights. The pocket weights are the output of the algorithm.

The Pocket algorithm is presented below.

The Pocket Algorithm

1. **Initialize weights and variables**

Define $w_i(t)$, $(0 < i < n)$, to be the weight from input i at time t, and θ to the threshold value in the output node.

Set w_0 to be $-\theta$, and x_0 to be always 1.

Set $w_i(0)$ to zero, thus initializing all the weights and the threshold.

Set $run_W = run_{W^*} = num_okW = num_okW^* = 0$, where

run_W = number of consecutive correct classifications using perceptron weights W.

run_{W^*} = number of consecutive correct classifications using pocket weights W^*.

num_okW = total number of training examples that W correctly classifies.

num_okW^* = total number of training examples that W^* correctly classifies.

2. **Present input pattern and desired output**

Present randomly chosen input pattern $X_p = x_{p,1}, x_{p,2}, x_{p,3}, ..., x_{p,n}$ and corresponding desired output D_p.

3. **Calculate actual output**

$y(t) = f_h\ (\sum_{i=0}^{n} w_i(t) x_{p,i}(t))$

4. **Adapt weights**

If W correctly classifies X_p, i.e. $y(t)$ is equal to the desired output of pattern X_p,

Then
$run_W = run_W + 1$
If $run_W > run_{W^*}$
Then
Compute num_okW by checking every training pattern
If $num_okW > num_okW^*$
Then
Set $W^* = W$
Set $run_{W^*} = run_W$
Set $num_okW^* = num_okW$
If all training patterns are correctly classified
Then Stop
Else
Adapt weights
$W(t+1) = W(t) + D_p\ X_p$
Set run_W =0

4. **Next iteration**

If the specified number of iterations has not been taken then go to 2.

The output of the algorithm is then the pocket weights W^*.

As an example of the Pocket algorithm, consider the XOR problem. The training set, with bipolar input and output spaces, is given below (including the bias):

$X_1 = (+1\ -1\ -1)$ $\quad D_1 = -1$
$X_2 = (+1\ -1\ +1)$ $\quad D_2 = +1$
$X_3 = (+1\ +1\ -1)$ $\quad D_3 = +1$
$X_4 = (+1\ +1\ +1)$ $\quad D_4 = -1$

Fig. 4.1 gives a sequence of nine iterations of the algorithm. At iteration eight both pocket and perceptron weights are optimal, classifying three out of four patterns correctly. At iteration nine the initial perceptron weights (0 0 0) have been reached. These misclassify every training pattern, whereas the pocket weights, (1 -1 -1), still get three out of four patterns correct.

Iter.	W	run_W	W^*	run_{W^*}	Choice	OK?	Action
1	(0 0 0)	0	(0 0 0)	0	X_4	no	$W = W$ - X_4 $run_W = 0$
2	(-1 -1 -1)	0	(0 0 0)	0	X_4	yes	$run_W = run_W + 1$ $W^* = W$ $run_{W^*} = run_W$
3	(-1 -1 -1)	1	(-1 -1 -1)	1	X_2	no	$W = W + X_2$ $run_W = 0$
4	(0 -2 0)	0	(-1 -1 -1)	1	X_3	no	$W = W + X_3$ $run_W = 0$
5	(1 -1 -1)	0	(-1 -1 -1)	1	X_4	yes	$run_W = run_W + 1$
6	(1 -1 -1)	1	(-1 -1 -1)	1	X_2	yes	$run_W = run_W + 1$ $W^* = W$ $run_{W^*} = run_W$
7	(1 -1 -1)	2	(-1 -1 -1)	2	X_3	yes	$run_W = run_W + 1$ $run_{W^*} = run_W$
8	(1 -1 -1)	3	(1 -1 -1)	3	X_1	no	$W = W$ - X_1 $run_W = 0$
9	(0 0 0)	0	(1 -1 -1)	3	...		

Figure 4.1: Pocket Algorithm iterations.

4.1.2 The Tilling Algorithm

The Tilling algorithm [Mezard & Nadal, 1989] builds a network layer by layer in order to learn the mapping of any Boolean function of N_{inp} inputs. The input and output spaces of the mapping are bipolar, i.e. -1 and 1. The network is constructed from threshold units.

The algorithm is described for mappings requiring one output unit. The training set consists of N_p bipolar patterns, where $N_p \leq 2^{N_{inp}}$.

The algorithm is based on the idea of adding new layers in such a way that input patterns having a different output get a different internal representation in the newly added layer. Suppose that the L_{th} layer has been built, and it is made of N_L units. To each input pattern X_p there correspond a set of values of the neurons in the L_{th} layer. This set of values is referred to as the *internal representation* of input pattern X_p in the layer L (the L=0 layer is the input layer,where the representation is the input patterns themselves).

In the Tilling algorithm the concept of classes is introduced. Two input patterns belong to the same *class* for layer L, if they have the same internal representation, which is called the *prototype* of the class. If there are N_p^L different patterns of $N_L + 1$ elements (1 extra for the bias unit), then there are N_p^L different classes in layer L. Each pattern being the internal representation of at least one input pattern. It holds then

$$\begin{aligned} N_p^L &\leq N_p \\ N_p^0 &= N_p \end{aligned}$$

The algorithm introduces also the concept of *faithful classes.*

Faithful Class: A class in layer L is said to be faithful if and only if all input patterns belonging to that class have the same desired output.

The Tilling algorithm then makes sure all classes in an added layer are faithful. The algorithm then maps the prototype of each class onto its desired output, which is the desired output of all input patterns belonging to that class.

If this mapping is still not linearly separable then only part the classes is mapped and a new layer is required. At each new layer the previously mapped classes are not spoiled and a new internal representation is found for the unmapped classes of the previous layer in such a way that at least a part of them will be correctly mapped. The way this is done is explained below.

In each layer L, the algorithm distinguishes two types of units. The first unit is called the *master unit* and plays a special role. The rest of the units are called *ancillary units.*

There is a certain number of input patterns, N_e^L, for which the output of the master is not the desired output, and the remaining $N_p - N_e^L$, for which the master unit gives the desired output. From layer $L-1$ the algorithm will generate the master unit of layer L trying to minimize the number of errors N_e^L, and, in particular, it will always find a solution such that

$$N_e^L \leq N_e^{L-1} - 1$$

This means that the algorithm will converge after a finite number of layers. In the last layer L^*, $N_e^{L*} = 0$, and the master unit is the desired output unit.

All other units in layer L, the ancillary units, are used to create internal representations in the case the master unit has not become the desired output unit. The addition of ancillary units is done in order to maintain the faithfulness condition.

The following theorem explains how convergence is achieved.

Convergence Theorem: Suppose that all the classes in layer $L-1$ are faithful, and that the number of errors of the master unit, N_e^{L-1}, is non-zero. Then there exists at least one set of weights w connecting the $L-1$ layer to the master unit of layer L such that $N_e^L \leq N_e^{L-1} - 1$. Furthermore, one can construct explicitly one such set of weights.

The proof is this theorem is a way to construct the weight vector.

Proof: Let $X_p^{L-1} = (x_{p,0}^{L-1}, x_{p,1}^{L-1}, x_{p,2}^{L-1}, ..., x_{p,N_{L-1}}^{L-1})$, where $1 \leq p \leq N_p^{L-1}$, be the prototypes in layer $L-1$. Each prototype X_p^{L-1} is the internal representation of a certain number R_p^{L-1} of input patterns

$$\sum_{p=1}^{N_p^{L-1}} R_p^{L-1} = N_p$$

and all of them have the same output o_p^{L-1}. The master unit is connected to the $N_{L-1}+1$ units of the $L-1$ layer by a set of weights $W_1^L = (w_{0,1}^L, w_{1,1}^L, w_{2,1}^L, ..., w_{N_{L-1},1}^L)$. Note that $w_{0,1}^L$ is the weight connecting the bias unit to the master unit of layer L, and $w_{1,1}^L$ is the weight connecting the master unit of layer $L-1$ to the master unit of layer L. Let $X_{p'}^{L-1}$ be one prototype in layer $L-1$ for which $x_{p',1}^{L-1}$ (its master unit value) is not the desired output $o_{p'}$ of the $R_{P'}^{L-1}$ input patterns having $X_{p'}^{L-1}$ as internal representation in layer $L-1$ (the faithfulness condition implies that all these input patterns have the same desired output). Let W_1^L be defined by

$$\begin{aligned} w_{1,1}^L &= 1 \\ w_{j,1}^L &= \lambda^{L-1} o_{p'} x_{p',j}^{L-1}, \quad for\ j \neq 1 \end{aligned}$$

For a prototype X_p^{L-1} the master unit output will then be

$$\begin{aligned} o_{1,p}^{L} &= sgn(\sum_{j=0}^{N_{L-1}} w_{j,1}^{L} x_{p,j}^{L-1}) \\ &= sgn(x_{p,1}^{L-1} + \lambda^{L-1} o_{p'} \sum_{j\neq 1, j=0}^{N_{L-1}} x_{p',j}^{L-1} x_{p,j}^{L-1}) \end{aligned}$$

If

$$\lambda^{L-1} > \frac{1}{N_{L-1}}$$

then the prototype $X_{P'}^{L-1}$ (and all input patterns belonging to that prototype) will have its desired output as output of the master unit of layer L. This is shown below

$$\begin{aligned} o_{1,p'}^{L} &= sgn(x_{p',1}^{L-1} + \lambda^{L-1} o_{p'} \sum_{j\neq 1, j=0}^{N_{L-1}} x_{p',j}^{L-1} x_{p',j}^{L-1}) \\ &= sgn(-o_{p'} + \lambda^{L-1} o_{p'} N_{L-1}) \\ &= o_{p'} \end{aligned}$$

What now remains to shown is that for all other prototypes X_{p}^{L-1}, where $p \neq p'$, the output of the master unit of layer L is equal to the output of the master unit of layer $L-1$ (this means a prototype has been corrected without spoiling other prototypes that could be already correct). Consider then the quantity $|\sum_{j\neq 1, j=0}^{N_{L-1}} x_{p',j}^{L-1} x_{p,j}^{L-1}) |$. It holds then that

$$|\sum_{j\neq 1, j=0}^{N_{L-1}} x_{p',j}^{L-1} x_{p,j}^{L-1}) | \leq (N_{L-1} + 2), \quad where\ p \neq p'$$

since each prototype differs in at least one bipolar element.

If

$$\lambda^{L-1} = \frac{1}{N_{L-1} - 1}$$

then

$$\begin{aligned} \mid \lambda^{L-1} o_{p'} \sum_{j\neq 1, j=0}^{N_{L-1}} x_{p',j}^{L-1} x_{p,j}^{L-1}) \mid &< 1, \quad for\ p \neq p',\ and \\ o_{1,p}^{L} &= x_{p,1}^{L-1}, \quad for\ p \neq p' \end{aligned}$$

The result is then that the master unit of layer L will give

$$N_e^L \leq N_e^{L-1} - R_{P'}^{L-1} \leq N_e^{L-1} - 1$$

The Tilling algorithm is then based upon the theorem of convergence presented above. The algorithm consists then of

1. a strategy for generating the master unit of a new layer, in such a way that at each layer N_e^L is at worst equal to the value given by the weight vector of the previous theorem, and
2. a strategy for adding ancillary units in a layer, in such a way as to end up with faithful representations.

In this way the convergence of the algorithm is guaranteed.

Generating the Master Unit

The generation of the master unit of a new layer employs a variant of the Pocket algorithm. The algorithm tries to learn the mapping $X_p^{L-1} \rightarrow o_p$ for all prototypes $1 \leq p \leq N_p^{L-1}$ of layer $L-1$. This is done with a perceptron algorithm. At each weight update (after presenting the set of prototypes), the algorithm keeps in a "pocket" the weight the produced the smallest number of errors with respect to the input patterns. This is done by weighting each prototype by the number of input patterns R_p^{L-1} that belong to that prototype. If the particular weight vector of the convergence theorem is taken as initial pocket in the algorithm, then it is ensured that $N_e^L \leq N_e^{L-1} - R_{P'}^{L-1} \leq N_e^{L-1} - 1$. The point of the Pocket algorithm is to speed up the convergence (i.e. generate less layers).

If the Pocket algorithm converges, i.e. $N_e^L = 0$ this means that the internal representations of the input patterns in layer $L-1$ are linearly separable. Then the master unit in layer L becomes the output unit and the Tilling algorithm stops.

Generating the Ancillary Units

If the master unit of layer L is not identical to the desired output unit, then the Tilling algorithm adds units to the layer L in order to obtain faithful classes. This is explained below.

Layer $L-1$ produces N_p^{L-1} prototypes of $N_{L-1}+1$ elements (N_{L-1} units plus the bias unit). Each prototype represents a faithful class of layer $L-1$ and is the internal representation of a set of input patterns with the same desired output.

In layer L the zeroth unit is the bias unit and is always one. The first unit is the master unit. At this point the prototypes of layer $L-1$ belong to one of the two classes in layer L whose prototypes, formed by the bias and the master unit, are $x_{1,0}^L = 1, x_{1,1}^L = 1$ and $x_{2,0}^L = 1, x_{2,1}^L = -1$. The fact that the master unit is not equal to the desired output unit means that at least one of the two classes is unfaithful. The next unit is then obtained by picking one of the unfaithful classes, say class 1 ($x_{1,0}^L = 1, x_{1,1}^L = 1$). and trying to learn by a perceptron algorithm the mapping $X_p^{L-1} \rightarrow o_p$ for prototypes of layer $L-1$ belonging to that class, i.e. prototypes for which the master unit of layer L outputs 1. If the mapping is learned, class 1 is then broken into two faithful classes. If not (the mapping is not linearly separable), the perceptron learning algorithm can learn a well chosen part of the class in such a way that it will be broken into two classes, one being faithful and the other remaining unfaithful. Then a new unit is added to separate the the remaining unfaithful class, and so on until the classes are separated and are then faithful. With this procedure, for each new ancillary unit one class is broken into two classes, so that with most N_p^{L-1} units all classes will be faithful.

Fig. 4.2 shows the network architecture achieved by the Tilling algorithm.

An Example: The XOR Problem

The XOR problem will serve as an example to illustrate the working of the Tilling algorithm.

Fig. 4.3(a) shows the four input patterns with their desired outputs for the XOR problem. The tilling algorithm will start by creating the master unit of layer 1 (M_1^1 in the figure). The training of the master unit with the Pocket algorithm cannot map the four patterns to their desired outputs, since the XOR problem is not linearly separable (as shown in Fig. 4.3(a)). After being trained, the master unit will produce the outputs shown in Fig. 4.3(b), leaving one pattern misclassified ($N_e^1 = 1$). Layer 1 have now two classes, class one with prototype (+1,-1) and class two with prototype (+1,+1). Class one, which contains only one input pattern (-1,-1) is faithful. Class two, which contains three input patterns, is an unfaithful class since one input pattern belonging to it has -1 as desired output and the other two patterns belonging to the class have $+1$ as desired output. An ancillary unit is then added (A_2^1 in the figure). This unit is trained by a perceptron algorithm to learn the mapping

A	B	C
-1	+1	+1
+1	-1	+1
+1	+1	-1

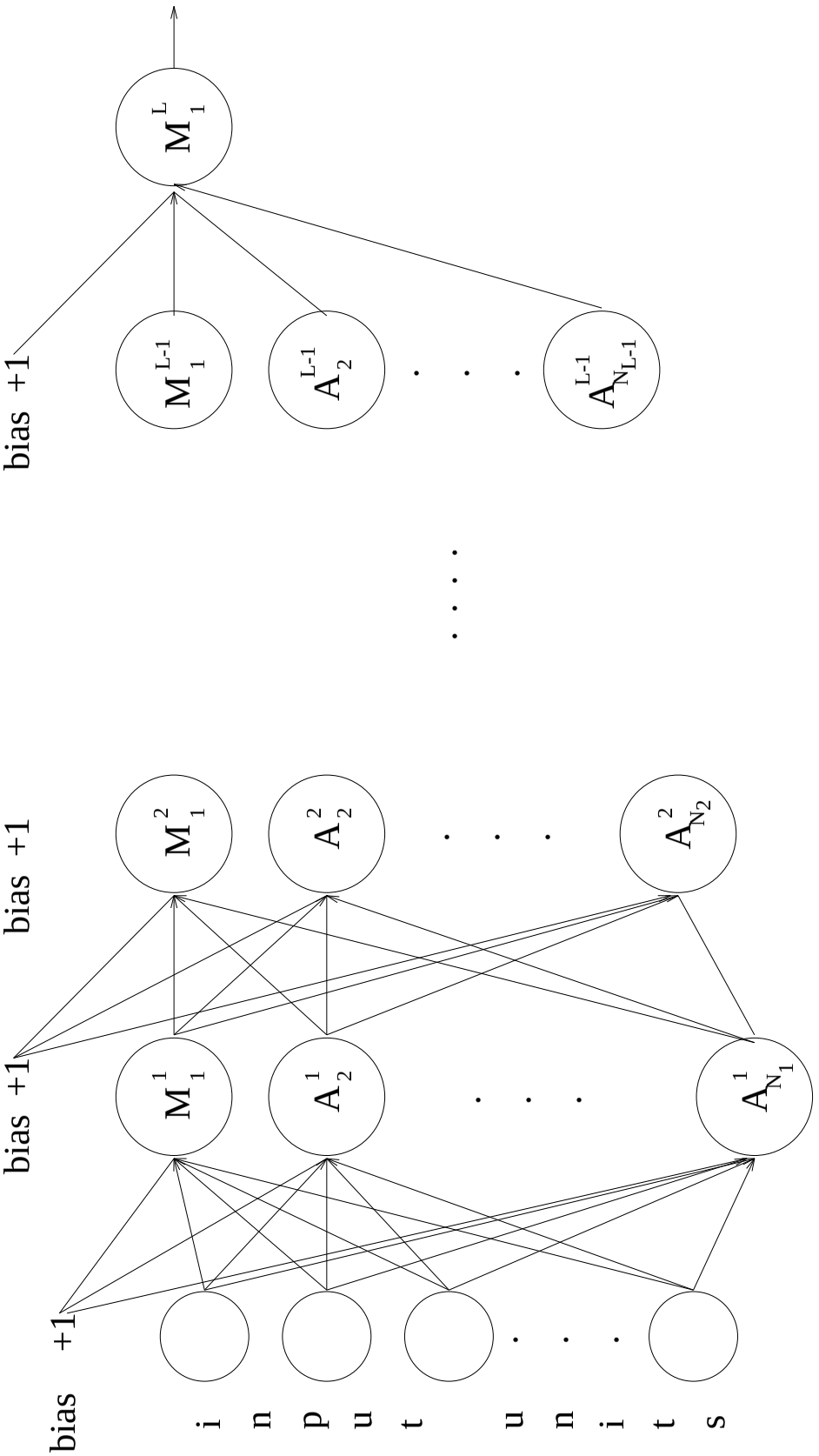


Figure 4.2: The Tilling Architecture.

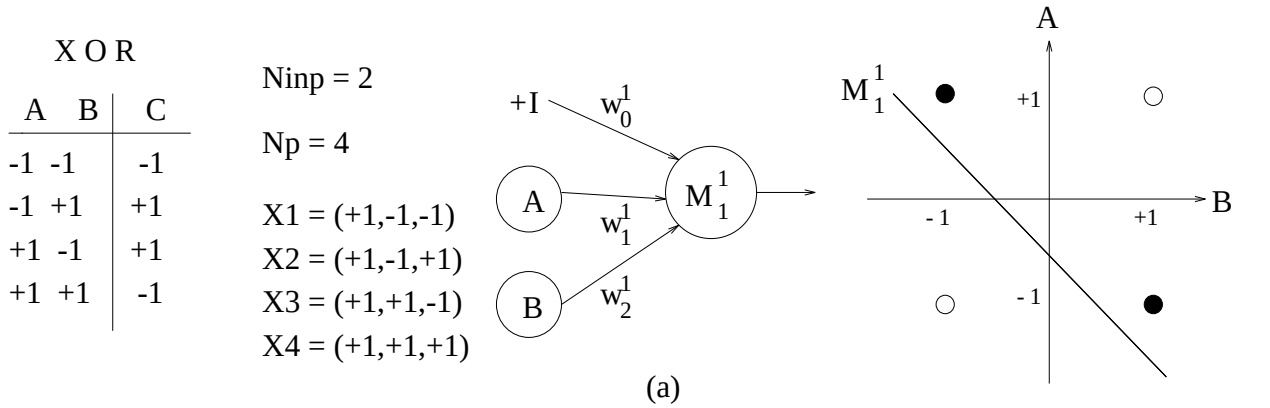
(where only the three patterns belonging to class two are used in the training set). The above mapping is linearly separable (as shown in Fig. 4.3(b)). Now the master unit and the ancillary unit separate the input patterns into three faithful classes whose prototypes are shown in Fig. 4.3(c). Finally a master unit is added in a new layer (layer 2). This master unit has to learn the mapping of the three prototypes of layer 1 to their desired outputs (since the classes of layer 1 are faithful, all input patterns belonging to the same class have the same desired output). This mapping is shown in Fig. 4.3(d) and since, as shown also in the figure, the mapping is linearly separable, the Pocket algorithm will be able to learn the mapping. This means that the master unit of layer two is now the output unit since it produces the desired outputs for all input patterns.

Multiple Outputs

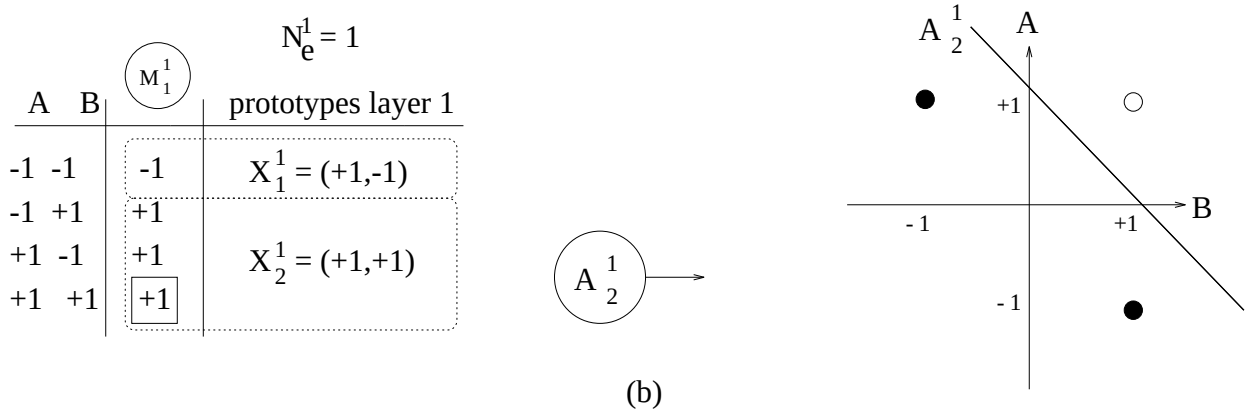
The Tilling algorithm has been described for problems requiring one output. To generalize this algorithm to multiple output units, a master unit per output would be needed in each layer. Then ancillary units would be needed to build up faithful classes in relation to each output. The problem how to generate the ancillary units remains to be researched.

One approach is to consider each output as a separate problem so that a separate group of ancillary units is constructed for each master unit.

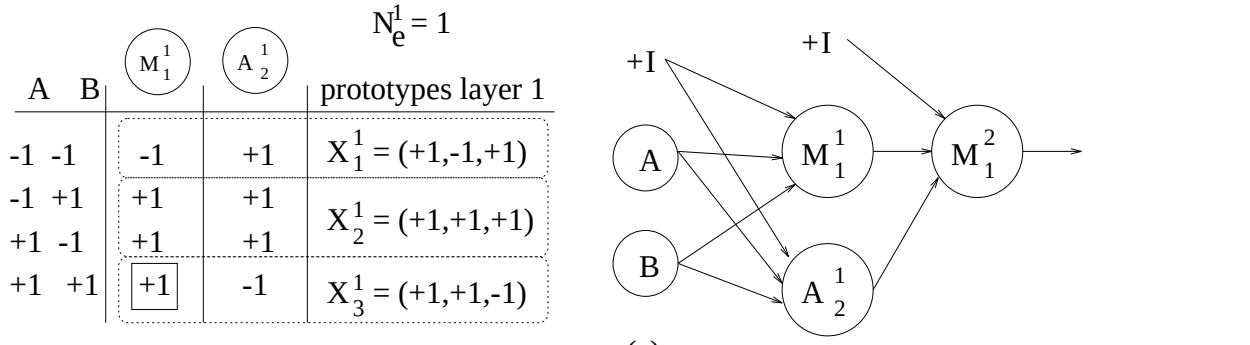
Another approach that should reduce the number of ancillary units is to use common ancillary units to all master units. This approach requires further research. A third approach can be the use of master units belonging to an output as ancillary units for the other master units. This approach requires also further research.



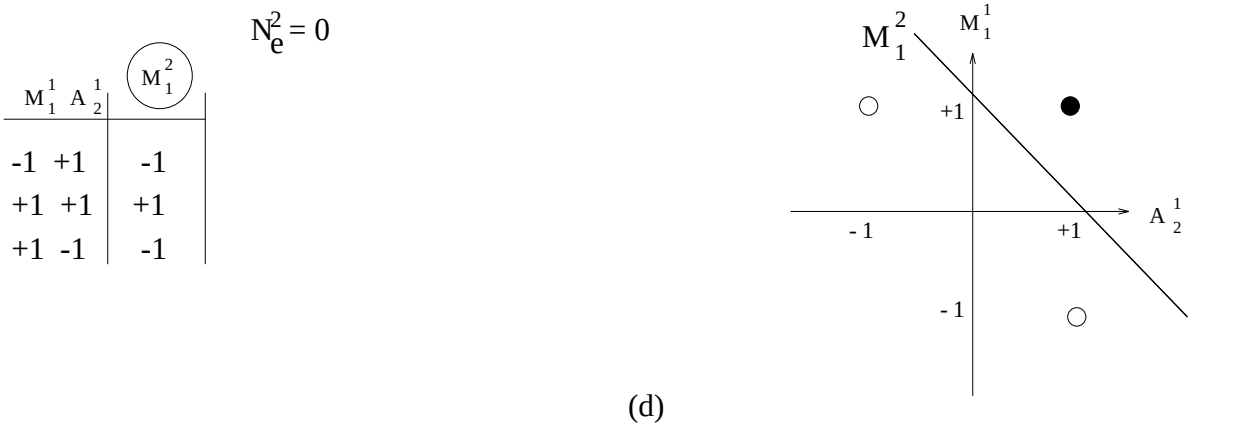
(a)



(b)



(c)



(d)

Figure 4.3: The XOR Problem

4.1.3 The Tower Algorithm

The Tower algorithm [Gallant, 1990] is an algorithm similar to the Tilling algorithm presented above. The difference being that the Tower algorithm employs the input units in order to achieve faithful classes, instead of adding ancillary units at each layer. The architecture of the Tower algorithm is shown in Fig. 4.4.

By connecting the master unit of layer L to the master unit of layer $L-1$ and to all the units of the input layer, then the internal representation of an input pattern in layer $L-1$ is the union of the input pattern itself and of the associated state of the master unit of layer $L-1$. If there are N_p input patterns then there are N_p distinct internal representations, representing N_p faithful classes of one element each.

The master unit is generated with the Pocket algorithm as in the Tilling algorithm.

Multiple Outputs

In [Gallant, 1990] the solution given for problems requiring multiple outputs is the use of a separate network for each output. This is shown in Fig. 4.5(a) for a problem requiring two outputs, where there is a set of master units per output.

Another approach is the connection of all master units of of previous layer to all all master units of the next layer, as shown in Fig. 4.5(b), This could make the classes more easily separable, which may lead to faster convergence.

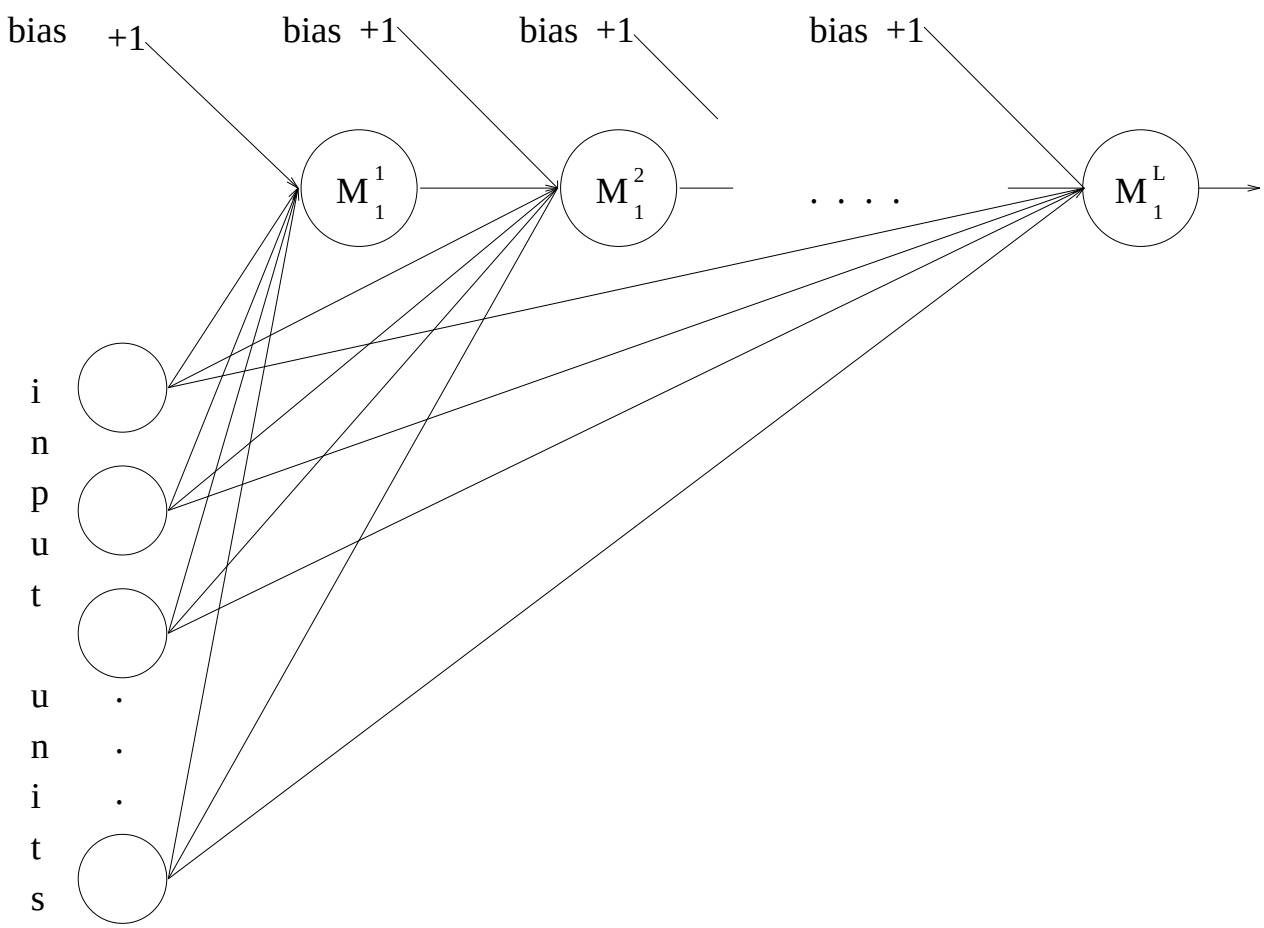


Figure 4.4: The Tower Architecture

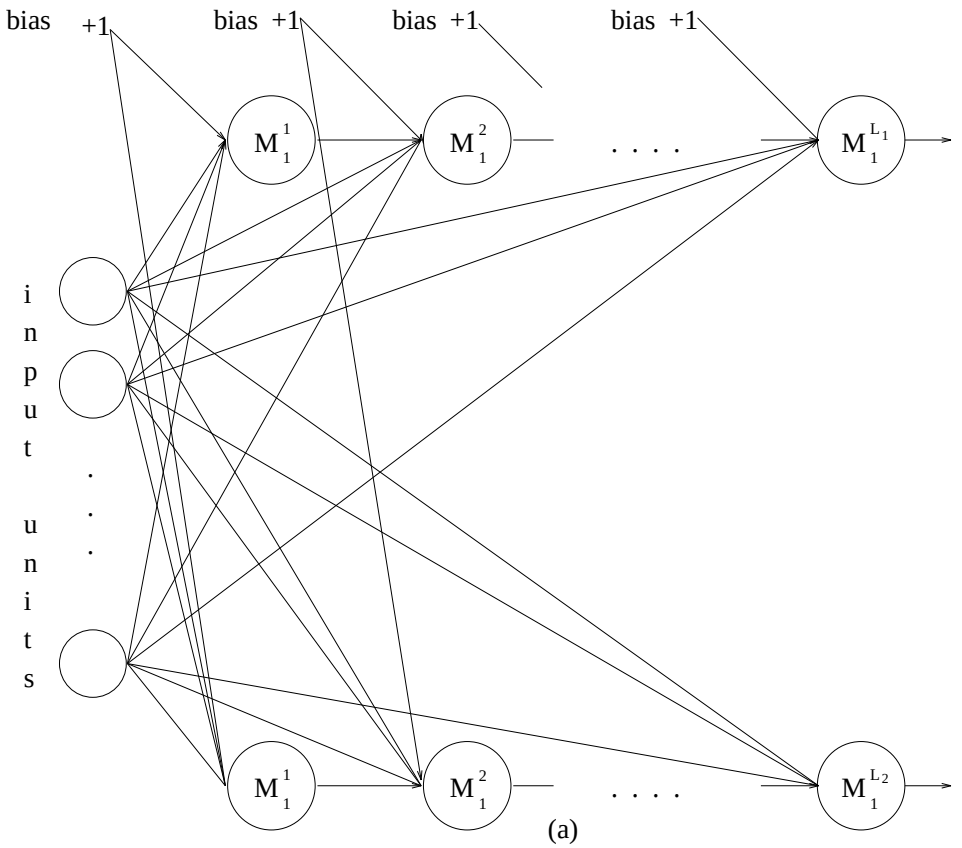


Figure 4.5: The Tower Architecture, (a) separate network, (b) joint network

4.1.4 The Upstart Algorithm

The Upstart algorithm [Frean, 1990] builds a network layer by layer in order to learn the mapping of any Boolean function of N_{inp} inputs. The input and output spaces of the mapping are binary, i.e. 0 and 1. The network is constructed from threshold units.

The Upstart Algorithm uses a different strategy for building a network than the Tilling and the Tower algorithms presented above. Instead of building layers from the input outwards until convergence, new units are added between the input layer and the output. The role of these units is to correct mistakes made by the output unit. This strategy is similar to the one used in the Cascade-Correlation algorithm, but here applied to binary input and output spaces, in a network consisting of threshold units. Also the network produced by the Upstart algorithm as a binary tree of threshold units. This makes the algorithm different, as it will be seen below.

As the Tilling algorithm, the Upstart algorithm uses the Pocket algorithm for one layer training.

The units used in the network are threshold units with output given by

$$o_p = \begin{cases} 1 & \text{if } \phi_p \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$\phi_p = \sum_{i=0}^{N} w_i x_{p,i} \tag{4.2}$$

where w_i are the connection weights and $x_{p,i}$ is the value of the i^{th} component of the input pattern p. Here again $x_{p,0}$ is the bias and is equal to 1, w_0 = - θ, where θ is the threshold.

The algorithm is considered for mappings requiring one output unit. Given a training set of patterns $X_p = x_{p,1}, x_{p,2}, x_{p,3}, ..., x_{p,n}$ and the corresponding desired outputs D_p, there are four possibility for the output of a threshold unit:

1. **Correctly On**: the output of the threshold unit is 1 and the desired output D_p of the input pattern X_p is 1 ($o_p = D_p = 1$).
2. **Correctly Off**: the output of the threshold unit is 0 and the desired output D_p of the input pattern X_p is 0 ($o_p = D_p = 0$).
3. **Wrongly On**: the output of the threshold unit is 1 and the desired output D_p of the input pattern X_p is 0 ($o_p = 1$, $D_p = 0$).
4. **Wrongly Off**: the output of the threshold unit is 0 and the desired output D_p of the input pattern X_p is 1 ($o_p = 0$, $D_p = 1$).

Note that this applies to discrete networks with threshold units. In continuous networks with Sigmoid units the output of the unit and the desired output have continuous values and this doesn't hold.

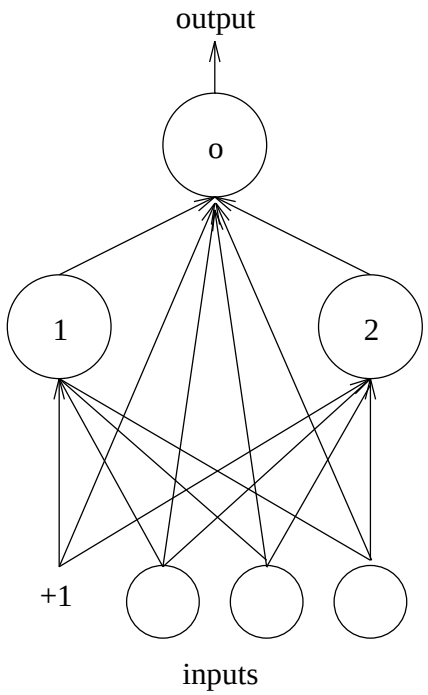


Figure 4.6: Correcting a Parent Unit.

These four possible situations for the output of a threshold unit open the door for the Upstart algorithm and for the BINCOR algorithm to be presented in the next section.

The idea of the algorithms is then to add two daughter units, 1 and 2, between the output unit (the parent) and the input units. Daughter unit 1 will try to correct input patterns that produce "Wrongly On" errors at the parent unit (the output unit), and daughter unit 2 will try to correct input patterns that produce "Wrongly Off" errors at the parent unit (the output unit), This is shown in Fig. 4.6.

Lets now define the size of an error.

Definition: The *size* of an error, for a given input pattern $X_{p'}$, that causes a "Wrongly Off" or "Wrongly On" error at the output, is the absolute value of $\phi_{p'}$.

$$Size(p) = \begin{cases} |\phi_{p'}| & \text{if } o_{p'} \neq d_{p'} \\ 0 & \text{if } o_{p'} = d_{p'} \end{cases} \tag{4.3}$$

where $o_{p'}$ and $\phi_{p'}$ are defined in Eq. (4.1) and (4.2).

Lets now divide the input patterns into four separate sets: $P^o_{correct_on}$, $P^o_{correct_off}$, $P^o_{wrong_on}$ and $P^o_{wrong_off}$, where

1. $P^o_{correct_on}$ is the set of all input patterns that produce a "Correctly On" at the output,

2. $P^o_{correct_off}$ is the set of all input patterns that produce a "Correctly Off at the output,
3. $P^o_{wrongly_on}$ is the set of all input patterns that produce a "Wrongly On at the output,
4. $P^o_{wrongly_off}$ is the set of all input patterns that produce a "Wrongly Off at the output,

Consider the set ($P^o_{wrongly_on}$). The output for these patterns can be corrected if unit 1 is active (equals 1) only for these patterns

$$o^1(p) = \begin{cases} 1 & \text{if } p \in P^o_{wrongly_on} \\ 0 & \text{if } p \in (P^o_{wrongly_off} \cup P^o_{correct_on} \cup P^o_{correct_off}) \end{cases} \tag{4.4}$$

where $o^1(p)$ is the output of unit 1 for input pattern p.

What is needed is to connect unit 1 to the output unit by a negative weight whose absolute value is larger than the largest error size of the input patterns in $P^o_{wrongly_on}$.

$$w^1 = -(MAX(Size(P^o_{wrongly_on})) + \alpha) \tag{4.5}$$

where $\alpha > 0$.

If the output of unit 1 is inactive (equals 0) for patterns belonging to ($P^o_{wrongly_off} \cup P^o_{correct_off} \cup P^o_{correct_on}$), the output at the output unit for these patterns will remain the same.

This is shown below

$$\begin{aligned} \phi^*_p &= \sum_{i=0}^{N} w_i x_{p,i} + w^1 o^1_p \\ &= \phi_p + w^1 o^1_p \end{aligned}$$

where $*$ represents the network after daughter unit 1 has been added, w^1 is the weight connecting unit 1 to the output unit and its value is given in Eq. (4.5), and o^1_p is the output of unit 1 for input pattern p.

For every p' that belongs to $P^o_{wrongly_on}$

$$\begin{aligned} \phi^*_{p'} &= \phi_{p'} + w^1 o^1_{p'} \\ &= \phi_{p'} + w^1 1 \\ &= \phi_{p'} - MAX(P^o_{wrongly_on}) - \alpha \\ &< 0 \end{aligned}$$

so $o_{p'} = 0$, for all p' that belong to $P^o_{wrongly_on}$. The class $P^o_{wrongly_on}$ has then being corrected.

Now for every p" that belongs to $(P^o_{wrongly_off} \cup P^o_{correct_off} \cup P^o_{correct_on})$

$$\begin{aligned} \phi^*_{p"} &= \phi_{p"} + w^1 o^1_{p"} \\ \phi^*_{p"} &= \phi_{p"} + w^1 0 \\ &= \phi_{p"} \end{aligned}$$

so the output is not modified for these classes.

The same idea can be applied to the input patterns for which the output is "Wrongly Off" $(P^o_{wrongly_off})$

$$o^2(p) = \begin{cases} 1 & \text{if } p \in P^o_{wrongly_off} \\ 0 & \text{if } p \in (P^o_{wrongly_on} \cup P^o_{correct_on} \cup P^o_{correct_off}) \end{cases} \tag{4.6}$$

where $o^2(p)$ is the output of unit 2 for input pattern p.

The output for these patterns can be corrected if unit 2 is active (equals 1) only for these patterns. Here a positive connecting weight to the output unit is needed, whose absolute value is larger than the largest error size of the input patterns in $P^o_{wrongly_off}$.

$$w^2 = MAX(Size(P^o_{wrongly_off})) + \alpha \tag{4.7}$$

Note that the strategy for correcting the output unit presented above only works for threshold units whose outputs are binary, or more precisely, for threshold units where one of the two output is zero. If the outputs of the threshold units were bipolar (+1 and -1), this strategy would not work, since while trying to correct "Wrongly On" ("Wrongly Off") errors, new "Wrongly Off" ("Wrongly On") errors would be added.

Unit 1 and unit 2 don't play a roll at the same time since for input patterns belonging to $P^o_{wrongly_on}$ only unit 1 is active, for input patterns belonging to $P^o_{wrongly_off}$ only unit 2 is active, and for input patterns belonging to $P^o_{correct_on}$ or $P^o_{correct_off}$ none of them are active.

Unit 1 and unit 2 can be trained to produce their respective mappings, given in Eq. (4.4) and (4.6). In the Upstart algorithm, the training is done with the Pocket algorithm.

Another point to consider is the fact that if unit 1 is active not only for patterns belonging to $P^o_{wrongly_on}$ but for patterns belonging to $P^o_{correct_off}$, the effect is only to reinforce the output response to $P^o_{correct_off}$, without introducing any error. This means that the patterns of $P^o_{correct_off}$ can be eliminated from the training set of unit 1. Whether unit 1 is active or not for patterns of $P^o_{correct_off}$, is of no importance since no correctly classified patterns will be misclassified. The same

holds for unit 2 and $P^o_{correct_on}$. This elimination of patterns during training makes the problem easier and faster to solve. The training sets for unit 1 and 2 become now

$$o^1(p) = \begin{cases} 1 & \text{if } p \in P^o_{wrongly_on} \\ 0 & \text{if } p \in (P^o_{wrongly_on} \cup P^o_{correct_on}) \end{cases} \quad (4.8)$$

$$o^2(p) = \begin{cases} 1 & \text{if } p \in P^o_{wrongly_off} \\ 0 & \text{if } p \in (P^o_{wrongly_on} \cup P^o_{correct_off}) \end{cases} \quad (4.9)$$

Since the mapping n Eq. (4.8) are (4.9) are not necessarily linearly separable, then units 1 and 2 cannot always fully learn their respective mappings with the Pocket algorithm. This means that extra units have to be added. The Upstart algorithm uses then a recursive strategy in introducing new units, where if unit 1 (unit 2) has not fully learned its mapping then two new units are recursively added to correct the mapping of unit 1 (unit 2), one that corrects the "Wrongly On" errors and the other that corrects the "Wrongly Off" ones. This results in a binary tree network, shown in Fig. 4.7. How this guarantees convergence will now be explained.

Convergence Theorem: Daughter units can always make fewer errors than their parent unit, and connecting a daughter unit to a parent unit with the appropriate weight will always reduce the errors made by the parent.

Proof: The output errors are

$$e(o) = e(o)_{on} + e(o)_{off}$$

where

$$\begin{aligned} e(o)_{on} &= \mid P^o_{wrongly_on} \mid \\ e(o)_{off} &= \mid P^o_{wrongly_off} \mid \end{aligned}$$

Unit 1 can always have less errors than the output unit. For example, unit 1 can be active for at least one input pattern of $P^o_{wrongly_on}$ and inactive for all other patterns (the needed weight vector of unit 1 for this solution is the one given for the the Tilling algorithm, and this weight vector can be the initial pocket of the Pocket algorithm).

Therefore

$$e(1) < e(o)_{on} \leq e(o) \quad (4.10)$$

where $e(1)$ is the number of errors in unit 1 (for the mapping in Eq. (4.8)).

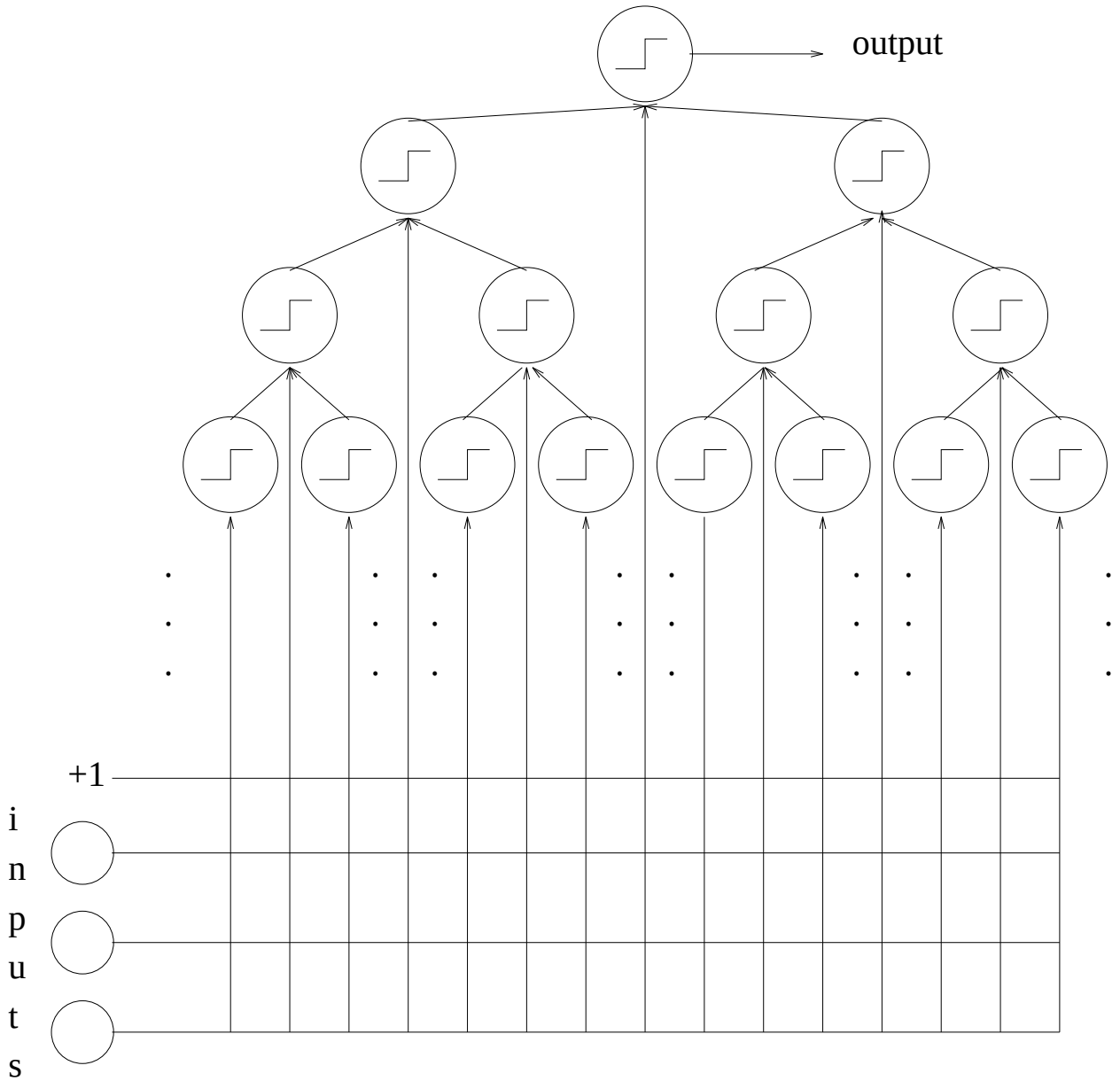


Figure 4.7: The Upstart Architecture.

A similar argument applies to unit 2.

It also follows that the output errors are reduced by connecting unit 1 to the output unit, since

$$\begin{aligned} e(o \; with \; 1) &\leq e(1)_{on} + e(1)_{off} + e(o)_{off} && (4.11) \\ &\leq e(1) + e(o)_{off} && (4.12) \\ &< e(o) && (4.13) \end{aligned}$$

and similarly for unit 2 on its own.

To understand inequation Eq. (4.11), consider that for each pattern belonging to $P^o_{wrongly_on}$, unit 1 should be active. If unit 1 is well active then the pattern is corrected in the output. If not then the pattern remains misclassified in the output, and this pattern is one of the $e(1)_{off}$ pattern of unit 1 (it is "Wrongly Off" in unit 1). For patterns belonging to $(P^o_{wrongly_off} \cup P^o_{correct_on})$, unit 1 should be inactive in order not to misclassify $P^o_{correct_on}$ or make $P^o_{wrongly_off}$ more "off". If unit 1 is well active for some of these patterns, these patterns are "Wrongly On" in unit 1, giving $e(1)_{on}$ errors in unit 1, and they could introduce errors in the output unit, but not necessarily.

When considering the joint action of units 1 and 2, the same result holds:

$$e(o \; with \; 1,2) < e(o) - 1 \tag{4.14}$$

Using this result the convergence of the Upstart algorithm can be proved: the Upstart algorithm adds then recursively two units between unit 1 and the input units, one that corrects $e(1)_{on}$ and the other $e(1)_{off}$. The same is done for unit 2. The inequations in (4.11) to (4.13) show that every time a daughter unit is added, the error at the parent unit is diminished at least by one. So if two units daughter units are added to unit 1, then

$$e(1 \; with \; daughters) < e(1) - 1 \tag{4.15}$$

and when going up the tree, $e(1)$ in inequation in (4.12) is reduced, which leads to a reduction in $e(0)$.

Daughter units are only generated by the algorithm if the parent makes errors, and the number of errors decreases at each branching. It follows that eventually none of the terminal daughters makes any mistake, so neither do their parents, and so on. Therefore every unit in the whole tree produces its target output, including the output unit. Hence the classification is learned. □

The Upstart algorithm is now presented:

The Upstart Algorithm

1. **Initial training of output unit**

Train output unit with the Pocket algorithm.
After the training, the connection weights between the inputs and the output unit remain frozen.
The following two steps are then applied recursively, generating a binary tree of units.

2. **Train daughter units**

If parent unit makes "Wrongly On" mistakes, train daughter unit 1 with the Pocket algorithm, having the input units as inputs, to learn the mapping in Eq.. (4.8) and go to 3.a.

If parent unit makes "Wrongly Off" mistakes, train daughter unit 2 with the Pocket algorithm, having the input units as inputs, to learn the mapping in Eq.. (4.9) and go to 3.b.
(The training of daughter units 1 and 2 can be done in parallel).

If parent unit makes no "Wrongly On" and no "Wrongly Off" mistakes, stop.

3.a **Add daughter unit 1**

Adds daughter unit 1 to the parent unit, the connection weight as defined in equations (4.5).
Go to 2 with unit 1 as parent unit.

3.b **Add daughter unit 2**

Adds daughter unit 2 to the parent unit, the connection weight as defined in equations (4.7).
Go to 2 with unit 2 as parent unit.

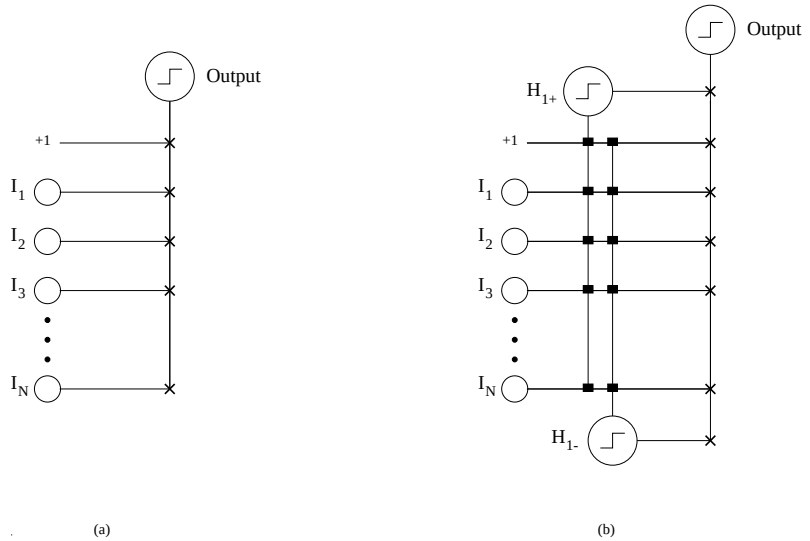


Figure 4.8: BINCOR Architecture: (a) initial, (b) after 2 hidden units are added. Boxed connections are frozen, x connections are trained repeatedly.

4.1.5 The BINCOR Algorithm

The BINCOR algorithm is a new learning algorithm, introduced here. The BINCOR algorithm builds a network in order to learn the mapping of any Boolean function of N_{inp} inputs. The input and output spaces of the mapping are binary, i.e. 0 and 1. The network is constructed from threshold units.

As the Upstart algorithm, the BINCOR algorithm is based on the idea of adds two units at a time between the output unit and the input units. where one unit will try to correct input patterns that produce "Wrongly On" errors at the output unit, and the other unit will try to correct input patterns that produce "Wrongly Off" errors at the output unit, as shown in Fig. 4.6.

The BINCOR algorithm follows a different strategy for adding pairs of units to the network than the Upstart algorithm. The BINCOR algorithm knows two possibilities building the network:

1. **Horizontal growth**: where pairs of units are added to the same hidden layer till the output errors are corrected.
2. **Vertical growth**: where pairs of units are added, each pair forming a new hidden layer containing only the pair, till the output errors are corrected.

Notice that possibility 2 is different from the binary tree architecture of the Upstart algorithm, where the number of units in each layer grows exponentially with factor 2.

The way these two network possibilities are built, and how convergence is achieved is now explained.

Fig. 4.8(a) shows the initial network. The BINCOR algorithm starts as the Upstart algorithm. It trains the output unit with the Pocket algorithm. If misclassified patterns remain, unit 1 and 2 are

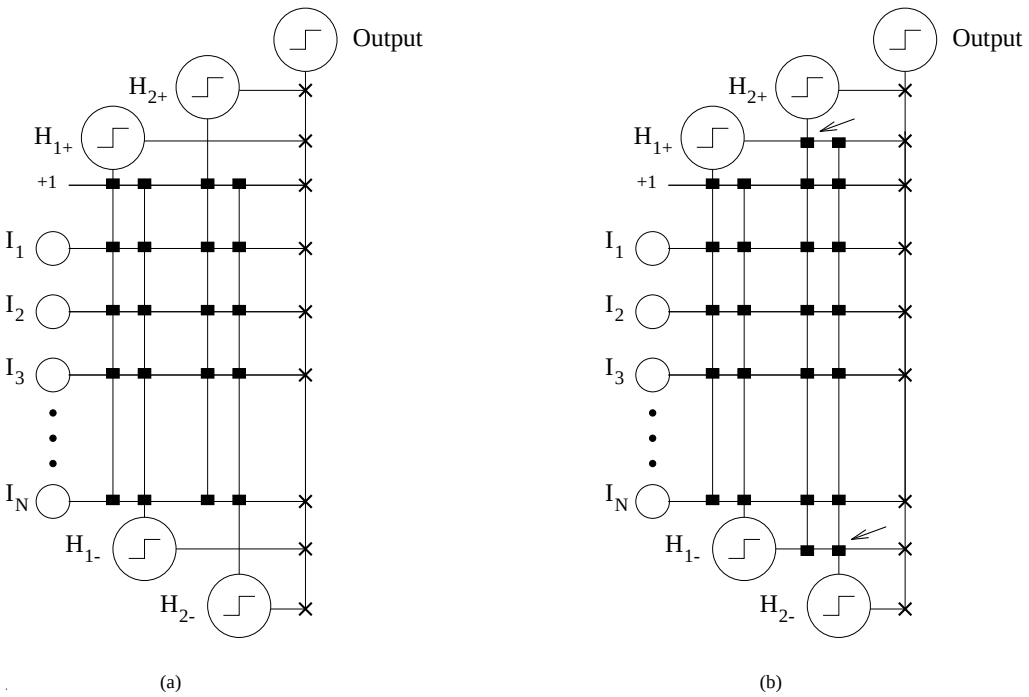


Figure 4.9: Network after adding two pairs of hidden units. (a) Horizontal Growth Network. (b) Vertical Growth Network.

trained with the Pocket algorithm to learn the mappings of Eq. (4.8) and (4.9). These units are then connected to the output unit with the weights w^1 and w^1 as given in Eq. (4.5) and (4.7).

As shown in Eq. (4.11) to (4.13), this leads to a reduction in the number of errors at the output. Fig. 4.8(b) shows the network after unit 1 and 2 are added.

Now the BINCOR algorithm continues in a different way than the Upstart algorithm. The output unit is further trained with Pocket algorithm, this time with the extra two weights from unit 1 and 2. This training will lead to either a further reduction of the number of errors at the output or at worse maintain the previously achieved reduction with the introduction of unit 1 and 2.

If there remain errors at the output unit, two new units, 1' and 2', are added. where unit 1' corrects the input patterns that now produce "Wrongly On" errors at the output unit, and unit 2' the ones that now produce "Wrongly Off" errors at the output unit, This is done again by training these two units with the Pocket algorithm to learn the mappings of Eq. (4.8) and (4.9), this time with the new sets $P^o_{correct_on}$, $P^o_{correct_off}$, $P^o_{wrong_on}$ and $P^o_{wrong_off}$ of input patterns. Units 1' and 2' are then connected to the output unit with the weights w^1 and w^1 as given in Eq. (4.5) and (4.7) (with the new sets of input patterns).

The steps of training the output unit and then adding two new units, correcting the new errors at the output, is repeated till no errors remain.

If the *horizontal growth* variant is chosen, then each pairs of units added have the input units as inputs. If the *vertical growth* variant is chosen, then each pairs of units added have the input units plus all previously added pairs of units as inputs.

Fig. 4.9 shows the network after unit 1' and 2' are added. Fig. 4.9(a) shows the horizontal growth case and Fig. 4.9(b) the vertical growth case. Notice that here new pairs of units try to correct the errors at the output unit, and not the errors at the parent unit, as in the Upstart algorithm.

Convergence is guaranteed by the Convergence Theorem given for the Upstart algorithm, where it was proven that every pair of units added to the output unit reduces the number of errors at the output.

Candidate Units

In order to speed up the learning in Upstart algorithm, the strategy presented below is taken.

Instead of training two units at each iteration, the one correcting the remaining input patterns that produce "Wrong On" errors at the output unit, and the other one correcting the "Wrong Off" ones at the output unit, two sets of "candidate" units are trained. The idea is that the Pocket algorithm starts the training with different random weights for each candidate in a set. From each set of candidates, the candidate that has the best result, i.e. the smallest number of errors, will be connected to the output unit.

Algorithm Description

The BINCOR algorithm is now presented:

The BINCOR Algorithm with Horizontal Growth

1. **Initial training of output unit**

Train output unit with the Pocket algorithm.

2. **Train hidden units**

If the output unit makes "Wrongly On" mistakes, train a new unit with the Pocket algorithm, having the input units as inputs, to learn the mapping in Eq.. (4.8). Then add this unit to the output unit with the connection weight as defined in equations (4.5).

If the output unit makes "Wrongly Off" mistakes, train a second new unit with the Pocket algorithm, having the input units as inputs, to learn the mapping in Eq.. (4.9). Then add this unit to the output unit with the connection weight as defined in equations (4.7).

If output unit makes no "Wrongly On" and no "Wrongly Off" mistakes, stop.

3. **Train output unit**

Train output unit with the Pocket algorithm (this time with the new weights from the added hidden units of step 2).

If output unit makes no mistakes, stop.
Otherwise go to step 2.

The BINCOR Algorithm with Vertical Growth

1. **Initial training of output unit**

Train output unit with the Pocket algorithm.

2. **Train hidden units**

If the output unit makes "Wrongly On" mistakes, train a new unit with the Pocket algorithm, having the input units and the previously added hidden units as inputs, to learn the mapping in Eq.. (4.8). Then add this unit to the output unit with the connection weight as defined in equations (4.5).

If the output unit makes "Wrongly Off" mistakes, train a second new unit with the Pocket algorithm, having the input units and the previously added hidden units as inputs, to learn the mapping in Eq.. (4.9). Then add this unit to the output unit with the connection weight as defined in equations (4.7).

If output unit makes no "Wrongly On" and no "Wrongly Off" mistakes, stop.

3. **Train output unit**

Train output unit with the Pocket algorithm (this time with the new weights from the added hidden units of step 2).

If output unit makes no mistakes, stop.
Otherwise go to step 2.

Multiple Outputs

One way of extending the algorithm for problems with multiple outputs will be to build a separate network for each output. This is shown in Fig. 4.10(a) for a two output problem.

Another approach, that should build smaller networks when some correlation exists between the outputs, is to connect the pairs of units that have being trained to correct the error at one output, to all other outputs. Fig. 4.10(b) shown the network built for a two output problem after two pairs of units have been added, a pair per output unit.

If pairs of units are added, and the vertical growth option is used, then each new pair of hidden units (one per output) will receive connection from all inputs, the bias and their corresponding previously added pairs of hidden units, i.e. the one which correct the same output error.

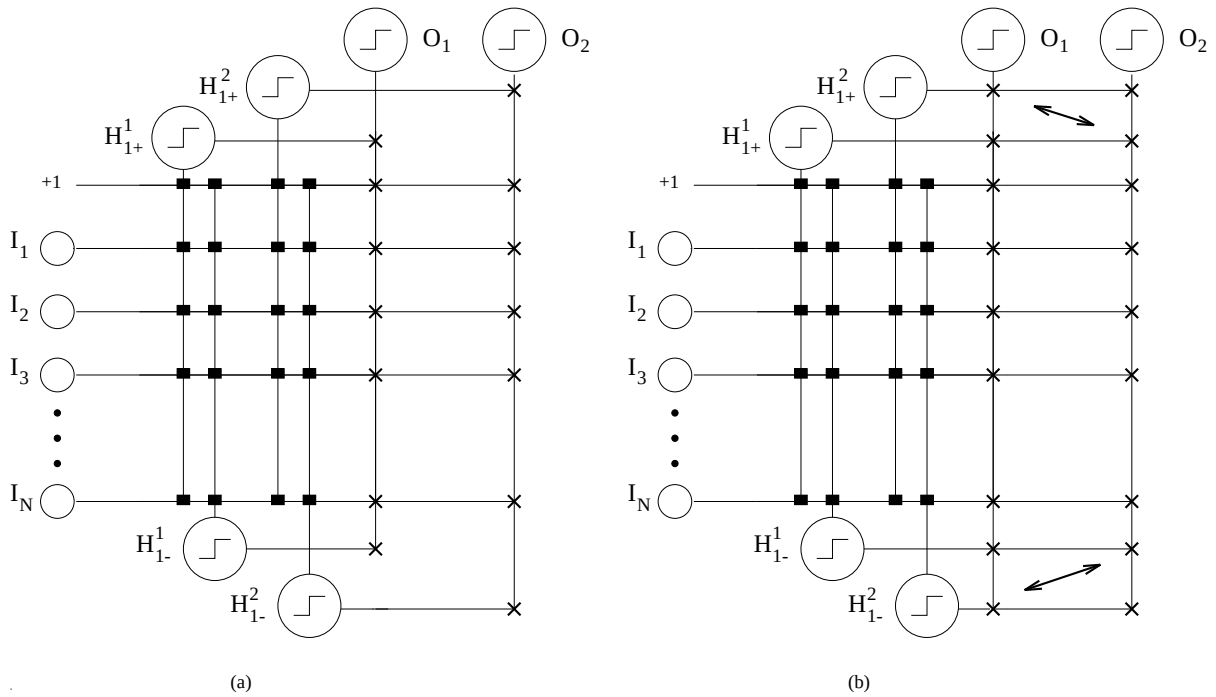


Figure 4.10: Topology for a two output problem. (a) Separate Network. (b) Combined Network.

4.2 The Applications

4.2.1 Random Binary Functions

A random Boolean function is achieved by assigning each of the 2^N input patterns a desired output 0 or 1 with 50% probability.

This is a difficult problem for a learning algorithm since there is no correlations and structure in the input space. This problem is then used to compare the performances of the learning algorithms presented.

4.2.2 The Monk's Problems

The Monk's problems belong to the artificial robot domain. Robots are described by six different attributes [Wnek et al., 1990]:

$$
\begin{aligned}
x_1 : head_shape &\in \text{(round,square,octagon)} \\
x_2 : body_shape &\in \text{(round,square,octagon)} \\
x_3 : is_smiling &\in \text{(yes,no)} \\
x_4 : holding &\in \text{(sword,balloon,flag)} \\
x_5 : jacket_color &\in \text{(red,yellow,green,blue)} \\
x_6 : has_tie &\in \text{(yes,no)}
\end{aligned}
$$

These means that there are 432 possible robots (3x3x2x3x4x2=432).

The learning task is a binary classification task. Each problem is given by a logical description of a class and then the task is to determine whether a robot belongs to that class or not. In order to test how well a learning algorithm generalizes the training set is constructed with a subset of all possible 432 robots, then the network is tested with all 432 robots.

Problem M_1

$(head_shape = body_shape)$ *or* $(jacket_color = red)$

For this problem, 124 robots were randomly selected for the training set. The were no misclassifications.

Problem M_2

exactly two of the six attributes have their first value

For this problem, 169 robots were randomly selected for the training set. The were no misclassifications.

Problem M_3

$((jacket_color = green)$ *and* $(holding = sword))$ *or* $((jacket_color \neq blue)$ *and* $(body_shape \neq octagon))$

For this problem, 122 robots were randomly selected for the training set. The were 5% misclassifications.

Problem M_1 is in standard disjunctive normal form (DNF). Problem M_2 is similar to parity problems. It combines different attributes in a way which makes it complicated to describe in DNF of CNF using the given attributes only. Problem M_3 is again in DNF and serves to evaluate the under the presence of noise.

The resulting diagrams of training and testing sets are found in Fig. 4.11 to Fig. 4.13.

The Monk's problems with the given training and test sets were used at the 2nd European Summer School on Machine Learning, held in Belgium during the summer of 1991 to compare different

Figure 4.11: Problem M_1: Training Set: 124 examples, no noise, and Test Set

Figure 4.12: Problem M_2: Training Set: 169 examples, no noise, and Test Set

Figure 4.13: Problem M_3: Training Set: 122 examples, 6 misclassifications, and Test Set

learning techniques [Thrun et al., 1991].

4.3 The Simulation Results

4.3.1 Random Binary Functions

In [Frean, 1990], a fast and well behaved version of perceptron learning was used to train the weights, in preference to the Pocket algorithm. In this version the weights changes given by the usual Perceptron Learning Rule were multiplied by the factor

$$\frac{T}{T_0} exp(\frac{- \mid \theta_p \mid}{T}) \tag{4.16}$$

Then weights changes then become

$$W(t+1) = W(t) + D_p X_p \frac{T}{T_0} exp(\frac{- \mid \theta_p \mid}{T}) \tag{4.17}$$

The new factor in Eq. (xx) decreases with $\mid \theta \mid$, which measures how "serious" the error is. The rationale behind this approach is that an error where $\mid \theta \mid$ is large is difficult to correct without causing other errors and should therefore be weighted as less significant than those where $\mid \theta \mid$ is small. The temperature T controls how strongly this weighting is biased to small $\mid \theta \mid$. In the results presented in [Frean, 1990] here T was reduced linearly from $T_0 = 1$ to zero, over the training period of 1000 epoch, i.e. 1000 passes over the whole training set. At high T the Perceptron Rule is recovered, but as T decreases the weights are frozen.

For the BINCOR algorithm, the variant presented above gave also better results than the Pocket algorithm.

In Fig. 4.14 a comparison in terms of number of units generated for a random Boolean function is given between the Tilling algorithm, the Upstart algorithm, the BINCOR algorithm and also the Continuous Constructive Supervised Learning algorithm Cascade-Correlation, presented in chapter 3. Results are given for random Boolean function with 2,4,6,8,9 and 10 inputs.

In Fig 4.15 a comparison is given between the results obtained with the BINCOR algorithm when the Pocket algorithm was used and when the above variant was used. Results are given for $N =$ 2,4,6,8 and 9.

4.3.2 The Monk's Problems

In Table 4.1, the results obtained with the BINCOR algorithm for the three Monk's problems are given. No results were available with the other algorithms presented in this chapter.

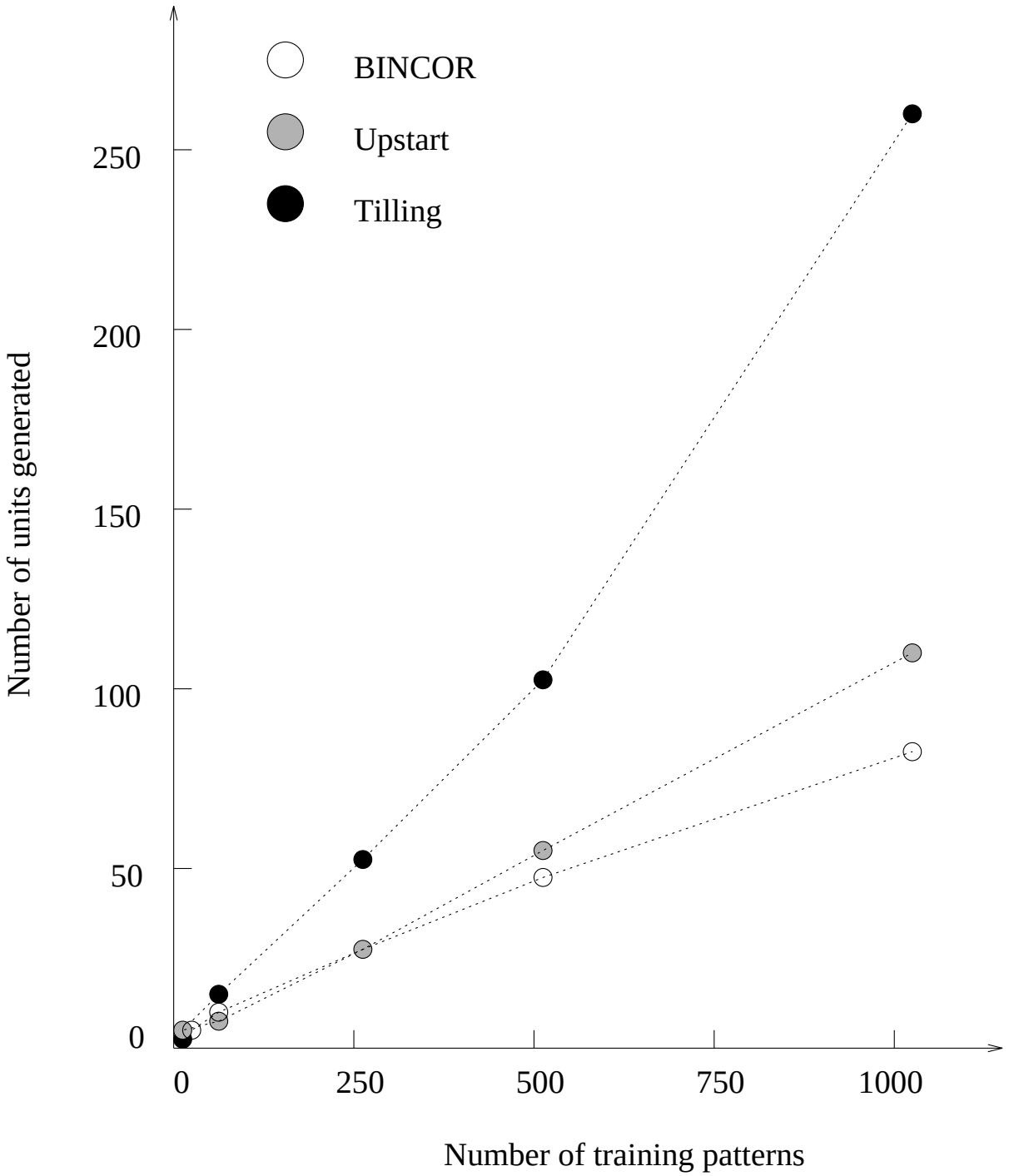


Figure 4.14: Random Binary Functions: Comparison of Results

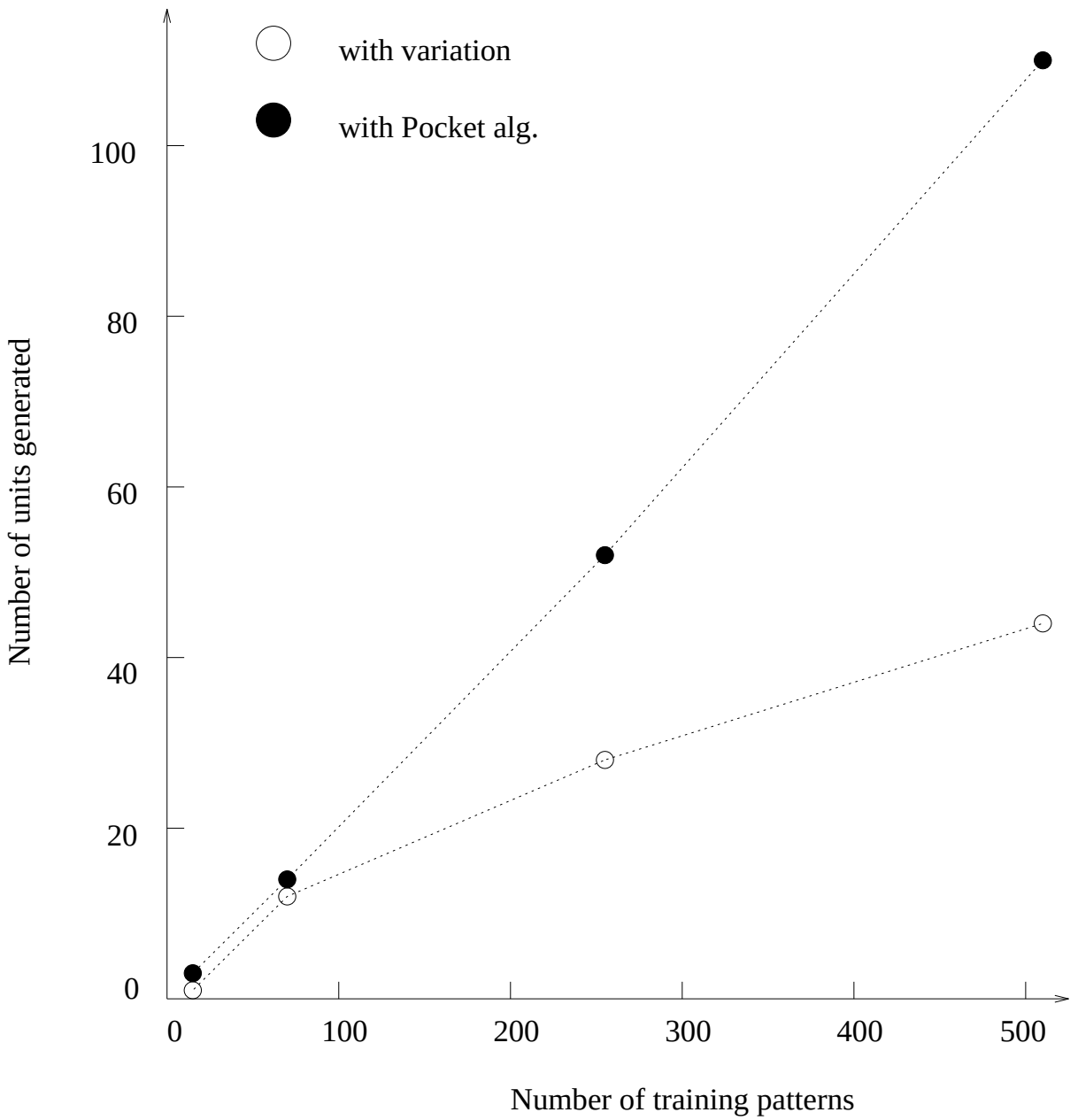


Figure 4.15: Random Binary Functions: BINCOR algorithm

Problem	Network Order	No. Hidden Units	Test Set Error
Monk 1	Second-Order	4	97.7 %
Monk 2	High-Order	34	77.8 %
Monk 3	High-Order	4	96.5 %

Table 4.1: Monk Results of BINCOR

For the first Monk problem, the best results were obtained with a second-order network (horizontal growth). For the others, a high-order network (vertical growth) gave the best results.

In Table 4.2 the results presented in [Thrun et al., 1991] for the Monk's problems are given. Although the Cascade-Correlation and the Backpropagation algorithms achieve better generalization, the BINCOR algorithm scores better than Decision Tree-Based Learning Algorithms, the mFOIL algorithm and Inductive Learning programs.

The results for the Modified Cascade-Correlation algorithm are not reported since there is no difference between the Modified Cascade-Correlation algorithm and the Cascade-Correlation algorithm for single output problems and the Monk's problems require a single output.

Table 4.2: Generalization results for the Monk's problems

4.4 Discussion and Future Work

The results achieved with the random binary functions by the constructive learning algorithms presented in this chapter point to the dependence of these algorithms on the one-layer training algorithm used by them. Further improvements in one layer training algorithms, so as the Perceptron algorithm, should lead to faster learning, smaller networks and better generalization in the constructive algorithms presented.

The comparison the Tilling algorithm, the Upstart algorithm and the BINCOR algorithm for the random binary functions shows that the BINCOR achieves the smallest amount of hidden units. The results obtained with the Tilling algorithm are achieved with the Pocket algorithm, so this results could improve if the Tilling algorithm were to employ the variant used in the Upstart and the BINCOR algorithms.

The Tilling algorithm produces a variable number of units in each hidden layer. The Upstart algorithm produces a binary tree network, so the number grows exponentially with factor two with each subsequent layer. The BINCOR algorithm has two units in each hidden layer.

Future work could be to try the various constructive algorithms with new improved versions of one-layer training algorithms. Also, for a more complete comparison of the algorithms, they should be applied to real-world applications.

Finally, ways of extending these algorithms to multiple output cases in a way that will reduce as much as possible the number of hidden units by sharing hidden units between outputs should be studied.

Conclusions and Future Work

5

As opposed to non-constructive learning algorithms, where the topology of the network is selected before the algorithm starts and remains fixed, Constructive Supervised Learning Algorithms start from an initial network topology and they add new units in order to learn the set of examples. The final topology and the size of the network are dynamically determined by the algorithm and are a function of the set of examples chosen and of the learning parameters.

The power and applicability of Constructive Supervised Learning Algorithms has been shown by using them in six applications of different nature. The applications are:

1. Solution to the Inverse Kinematic Transformations for Robot Control.
2. Corrective networks for damage Robot Controllers.
3. Hamming Codes Generators and Parity Generators.
4. A Pattern Recognition problem.
5. Mapping of Random Boolean Functions.
6. The Monk's problems.

The above applications have allowed an analysis and comparison of the different constructive learning algorithms presented. The comparison have been done in terms of learning speed, size of the network and generalization accuracy.

5.1 Continuous Constructive Supervised Learning Algorithms

For Continuous Constructive Supervised Learning Algorithms, the Cascade-Correlation Learning Algorithm and a modified version of it have been presented.

The Cascade-Correlation Algorithm builds a network by adding a hidden unit at a time between the previously added hidden units and the output units, until the training set is learned. Hidden units are trained to correlate to the sum of the remaining outputs' errors before they are connected.

The Modified Cascade-Correlation Algorithm has been introduced for problems were multiple outputs are required. In this algorithm, a new unit is added per output and is trained to correlate to the remaining error at the respective output before it is connected. Then the hidden unit is connected to all output units.

The Inverse Kinematic Transformations for a two and a three degrees of freedom Robot Controller has allowed for a comparison between the Modified and the original Cascade-Correlation algorithms, For this problem, the Modified Cascade-Correlation was shown to learn quicker, scale better to more training cases and to lower tolerance, and to have a higher success rate than the original Cascade-Correlation.

The Inverse Kinematic Transformation for a two degrees of freedom Robot Controller allowed also a comparison of the Modified Cascade-Correlation algorithm with previous results obtained with the non-constructive Backpropagation learning algorithm. In this case the accuracy of generalization achieved by the Modified Cascade-Correlation algorithm was equal or better and with a much smaller network than achieved by using the Backpropagation algorithm.

The other advantage given by the use of the Modified Cascade-Correlation algorithm over the Backpropagation algorithm is that no trial and error stage is needed to determine the amount and size of the hidden layers in the network, what is the case with the non-constructive Backpropagation algorithms.

The determination of an a-priori network topology when applying non-constructive learning algorithms, so as the Backpropagation algorithm, remains a difficult problem in the area of Robot Control. The results achieved for the Inverse Kinematic Transformation and for a Corrective Networks for a damage Robot Controllers, point to the applicability of the Modified Cascade-Correlation learning algorithm for this area of application.

For the Pattern Recognition problem and for the Hamming Code Generation problem, the Modified Cascade-Correlation Learning Algorithm was also shown to learn faster than the original Cascade-Correlation Learning Algorithm, When tested for generalization accuracy with the Inverse Kinematic Transformations and the Corrective networks for a damage Robot Controllers, the Modified Cascade-Correlation algorithm also scored better.

5.2 Discrete Constructive Supervised Learning Algorithms

For Discrete Constructive Supervised Learning Algorithms, the Tilling Learning Algorithm, the Tower Algorithm, the Upstart Algorithm and a new algorithm introduced in this report, the BINCOR Algorithm, have been presented.

The Tilling Learning Algorithm and the Tower Learning Algorithm build a network by adding a layer at a time, where the internal representation of the input patterns in each layer remain different for input patterns with different outputs. The process goes on until the first unit added at a new layer produces the correct mapping for all input patterns. The convergence of these algorithms for any Boolean mapping is guaranteed. The Tilling Algorithm adds a variable number of units in each layer. The Tower Algorithm adds only one unit in each layer and uses "shortcut" connections, i.e. it not only connects the unit of a new layer to the unit of the previous layer, so as in the Tilling Algorithm, but also to the input units.

The Upstart Algorithm and the BINCOR Algorithm use a different strategy than the previous two algorithms for building a network. In the Tilling and the BINCOR algorithms, units are added between the output unit and the inputs in order to correct the remaining misclassified patterns at the output unit. The convergence of these algorithms for any Boolean mapping is guaranteed. In the Upstart Algorithm a recursive strategy is used, where each unit added receives connections from the input units and tries to correct the misclassified input patterns at the unit previously added, leading to a binary tree network structure. In the BINCOR Algorithm, two units are added at a time between the input units and the output unit, where this units try to correct the remaining misclassified input patterns at the output unit, leading to one of two optional network structures: a single hidden layer or a high-order network of multiple layer, where the units in a hidden layer receive connections from the input units and all previously added units.

The mapping of random Boolean functions has allowed for a comparison between the Tilling, Upstart and BINCOR algorithms in terms of size of the network built. The introduced BINCOR algorithm produced the network with less units of the three algorithms.

The Monk's problems has been used to compare the generalization accuracy of the BINCOR algorithm with a variety of machine learning algorithms. The generalization with the BINCOR algorithm was less accurate than with the Cascade-Correlation Learning Algorithm and the Backpropagation Learning Algorithm, but more accurate than with Decision Tree-Based Learning Algorithms, the mFOIL algorithm and Inductive Learning programs.

5.3 General Conclusions and Future Directions

A next step the research of learning algorithms for Artificial Neural Networks should take is the extension of learning algorithms to multiple output networks. Extension to multiple outputs is of special relevance since many real-world applications require multiple outputs.

This also holds for the Discrete Constructive Supervised Learning Algorithms presented in this report, for which extensions to multiple output networks have to be studied.

There a need to develop good multiple output benchmarks, in order give a proper comparison between the various learning algorithms. Many of the comparison of learning algorithms found in the literature are still done based on single output problems so as the XOR/Parity problem, the two-spirals problem and the mapping of a random Boolean function, or on problems with multiple output but where the outputs are orthogonal, i.e. only one output is "on" at a time, so as the Encoder/Decoder problem.

In this report the Hamming Code Generation problem is suggested as a benchmark requiring multiple outputs. The Hamming Code Generation problem is a more general problem which becomes the Parity problem when there is only one output.

The performances of the algorithms discussed in this report for the various applications presented. show that the best network topology, in terms of speed of learning, size of the network, and generalization accuracy, can vary from for different applications. This is the case for example for the Hamming Code Generation problem and the mapping Inverse Kinematic Transformations, where in the first one, separate networks for each output achieved faster learning and less units, and in the second, a combined network for all outputs achieved the best results.

The above points to the advantage of constructive learning algorithms with optional network topologies, over constructive learning algorithms which lead to a single type of network topology. In this way, various topology options can be tried in order to chose the best one for a given application. The Modified Cascade-Correlation algorithm and the BINCOR algorithm presented are an example of constructive algorithms with optional topologies.

Bibliography

[Corsi, 1992] Corsi P., "The Esprit Program", in Proc. Neuro-Nimes '92, Nimes, France, 1992, pp. 671-682.

[Angeniol et al, 1992] Angeniol, B. et al., "The Galatea Project", in Proc. Neuro-Nimes '92, Nimes, France, 1992, pp. 683-695.

[Wenskay, 1992] Wenskay, D. L., "Intellectual Property Protection for Neural Networks",in Proc. Neuro-Nimes '92, Nimes, France, 1992, pp. 699-710.

[Hecht-Nielsen, 1990] Hecht-Nielsen, R., *Neurocomputing*, Addison-Wesly Publishing Company, London, UK, 1990.

[Fahlman, 1988] Fahlman, S. E., "Faster-Learning Variations on Back-Propagation: An Empirical Study", in Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann.

[Fahlman & Lebiere, 1990] Fahlman, S.E., and Lebiere, C., "The Cascade-Correlation Learning Architecture", *Advances in Neural Information Processing Systems* (Vol. 2), D.S. Touretzky (Ed.), San Mateo, CA, Morgan Kaufmann, 1990.

[Aleksander & Morton] Aleksander, I., and Morton, H., *An Introduction to Neural Computing*, Chapman & Hall, Reading, Massachusetts, 1991.

[Barhen et al., 1992] Barhen, J., Toomarian, N., et al., "New Direc tions in Massively Parallel Neurocomputing", in *Proc. Neuro-Nimes '92*, (Nimes, France), EC2 publ., pp. 543-554, November, 1992.

[Daud et al., 1992] Daud, T. et al., "VLSI Implemented Building Block Neural Network Chips for Classification/Detection Applications",in Proc. Neuro-Nimes '92, Nimes, France, 1992, pp. 565-576.

[Hurdle et al., 1992] Hurdle, J. F., L. Josephson, E. L. Brunvand and, Gopalakrishnan G., "Asynchronous Models for Large Scale Neurocomputing Applications",in Proc. Neuro-Nimes '92, Nimes, France, 1992, pp. 577-588.

[Jutten et al., 1990] Jutten, C., A. Guerin, et al., "Simulation Machine and Integrated Implementation of Neural Networks", in Neural Networks, L.B. Almedia and C.J. Wellekens (eds.), Berlin: Springer-Verlag, pp. 244-266, 1990.

[Rosenblatt, 1962] Rosenblatt, F, *Principles of Neurodynamics* , Spartan Books, New York, 1962.

[Rumelhart, Hinton & Williams, 1986] Rumelhart, D.E., Hinton, G.E., and Williams, R.J., "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Rumelhart, D.E., McClelland, J.L., and the PDP Research Group, Editors, vol. 1. Cambridge,MA. MIT Press, 1986.

[Vogl et al.] Vogl et al.,"Accelerating the Convergence of the Backpropagation Method", Biological Cybernetics, vol 59.

[Beale & Jackson, 1990] Beale, R., and Jackson, T., *Neural Computing: An Introduction*, IOP Publishing Ltd, Bristol, UK, 1990.

[Widrow, 1962] Widrow, B., "Generalization and information storage in networks of ADALINE neurons," in Yovitts, G.T., Editor, *Self-Organizing Systems*, Spartan Books, Washington DC, 1962.

[Antsaklis, 1992] Antsaklis P.J., Guest Ed., Special Issue on Neural Networks in Control Systems, IEEE Control Syst. Mag., Apr. 1992, pp. 8-10.

[Sjogaard, 1991] Sjogaard, S.,"A Conceptual Approach to Generalization in Dynamic Neural Networks",, Masters Thesis, Computer Science Department, Aarhus University, October 1991.

[Simon, Corporaal & Kerckhoffs, 1992] Simon, N., Corporaal, H., and Kerckhoffs, E., "Variations on the Cascade-Correlation Learning Architecture for Fast Convergence in Robot Control", in *Proc. Neuro-Nimes '92*, (Nimes, France), EC2 publ., pp. 455-564, November, 1992.

[Simon, Kerckhoffs & Corporaal, 1992] Simon, N., E. Kerckhoffs and H. Corporaal,"Variations on the Cascade-Correlation Learning Architecture for Fast Convergence", Neural Network World, 1992, vol 2 (5), pp. 497-510, VSP International Science Publishers.

[Rabelo & Avula, 1992] Rabelo L.C. and Avula J.R., "Hierarchical Neurocontroller Architecture for Robotic Manipulation", IEEE Control Syst. Mag., Apr. 1992, pp. 37-41.

[Sobajic, Lu & Pao, 1988] Sobajic D., Lu J. and Pao Y., "Intelligent Control of the INTELLEDEX 6057 Robot Manipulator", in Proc. IJCNN, San Diego, CA, 1988, pp.633-640.

[Josin, Charney & White, 1988] Josin G., Charney D. and White D., "Robot Control Using Neural Networks", in Proc. IJCNN, San Diego, CA, 1988, Vol. II, pp.625-631.

[Guez & Ahmad, 1988] Guez A. and Ahmad Z., "Solution to the Inverse Kinematic Problem in Robotics by Neural Networks", in Proc. IjcNN, San Diego, CA, 1988, PP.617-624.

[Guez & Ahmad, 1989] Guez A. and Ahmad Z., "Accelerated convergence in the Inverse Kinematics via Multi-Layered Feedforward Networks", in Proc. IJCNN, Washington, DC, 1989, pp.341-347.

[Liu, Iberall & Bekey, 1989] Liu H., Iberall T. and Bekey G., "Neural Network Architecture for Robot Arm Control", IEEE Control Syst. Mag., Apr. 1989, pp. 38-42.

[Bassi & Bekey, 1989] Bassi D. and Bekey G., "High Precision Position Control by Cartesian Trajectory Feedback and Connectionist Inverse Dynamics Feedforward", in Proc. IJCNN, Washington, DC, 1989, pp. 325-332.

[Parker, 1985] Parker, D. B., "Learning Logic", Tech. Rep. TR-47, *Center for Computational Research in Economics and Management Science*, MIT, Apr. 1985.

[Tesauro & Janssens, 1988] Tesauro, G., and Janssens, B., "Scaling Relationships in Back-Propagation Learning", *Complex Systems* vol. 2, pp. 39-44, 1988.

[Hamming, 1950] Hamming, R.W., "Error Detecting and Error Correcting Codes", *Bell System Tech. Journal*, vol. 29, pp. 147-160, 1950.

[Gallant, 1990] Gallant, S.I., "Perceptron-Based Learning Algorithm s", in *IEEE Transactions on Neural Networks*, vol. 1, no. 2, pp. 179-191, June 1990.

[Mezard & Nadal, 1989] Mezard, M., and Nadal, J., "Learning in Feedforward Layered Networks: the Tilling Algorithm", *J. Physics A*, vol. 22, no. 12, pp. 2191-2203, 1989.

[Nadal, 1989] Nadal, J., "Study of a Growth Algorithm for Neural Networks", *International J. of Neural Systems*, vol. 1, no. 1, pp. 55-59, 1989.

[Frean, 1990] Frean, M., "The Upstart Algorithm: A Method for Constructing and Training Feed-Forward Neural Network", *Neural Computation*, vol. 2, pp. 198-209, 1990.

[Thrun et al., 1991] Thrun, S.B. et al., The MONK's Problems A Performance Comparison of Different Learning Algorithms. Rep. no. CMU-CS-01-197, Carnegie Mellon University, December 1991.

[Wnek et al., 1990] Wnek, J., Sarma. A., Wahab, A., and Michalski, R., "Comparison Learning Paradigms via Diagrammatic Visualization: A Casa Study in Sigle Concept Learning Using Symbolic, Neural Net and genetic Algorithm Methods", Technical Report, George Mason University, Computer Science Department, 1990.