

**COMPARISON BETWEEN FAHLMAN'S AND CSM'S
RECURRENT CASCADE-CORRELATION
NEURAL NETWORKS**

by

Tri Cao Nguyen

ProQuest Number: 10783829

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10783829

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.


ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346


T-4240

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematics).

Golden, Colorado

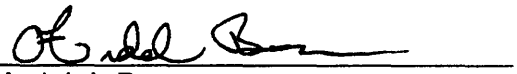
Date Jan 8, 1993

Signed: 
Tri Cao Nguyen

Approved: 
Dr. Aaron J. Gordon
Thesis Advisor

Golden, Colorado

Date 8 January 1993


Dr. Ardel J. Boes
Professor and Department Head
Mathematical and Computer Sciences

ABSTRACT

Artificial neural networks is a developing field in computer science. One neural network architecture is recurrent cascade-correlation developed by Scott E. Fahlman. In contributing to the development of this field, a new recurrent cascade-correlation network, which is a modification of Fahlman's, was recently devised at the Colorado School of Mines (CSM). For this research, representational capability, size, and learning time of the two networks are compared. This comparison is accomplished by training and testing both architectures, implemented in C, on the following problems: sequential XOR, Morse code, learning distance, and learning distance with noise. Number of successful learnings, number of hidden units, error, and learning time of both networks are experimentally obtained and compared. The network that has a broader learning capability, shorter learning time, smaller error and smaller size is more attractive. The comparison results show that CSM's networks are more attractive than Fahlman's for some problems but less attractive for other. In some other problems, there is no conclusion about their difference in performance.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF FIGURES.....	viii
LIST OF TABLES.....	ix
ACKNOWLEDGMENTS	xiii
Chapter 1. SIGNIFICANCE OF THE STUDY.....	1
Chapter 2. NEURAL NETWORK ARCHITECTURES	6
2.1 Human Brain.....	6
2.2 Simple Neural Model	7
2.3 Layered Feed-Forward Networks (Perceptrons).....	10
2.3.1 Single-Layer Neural Networks.....	11
2.3.2 Multi-Layer Neural Networks.....	12
2.3.3 Training Algorithm for Single-Layer Perceptrons.....	13
2.4 Back Propagation.....	15
2.5 Quick Propagation	17
2.6 Window Sliding Technique.....	20
2.7 Jordan's Network.....	21
2.8 Elman's Network	22
2.9 Fully Recurrent Network.....	24
2.10 Cascade-Correlation Networks.....	25

- 2.11 Recurrent Cascade-Correlation Networks.....28
- Chapter 3. FAHLMAN'S AND CSM'S RECURRENT
CASCADE-CORRELATION NETWORKS.....32
 - 3.1 Fahlman's Networks.....32
 - 3.2 CSM's Networks.....33
- Chapter 4. PROGRAM DESCRIPTION AND
GENERAL COMMON METHODOLOGY.....36
 - 4.1 Program Description.....36
 - 4.2 Verification of Correctness37
 - 4.2.1 Fahlman's Version.....38
 - 4.2.2 CSM's Version38
 - 4.3 Training Methodology38
 - 4.3.1 Determination of Learning Parameters39
 - 4.3.2 Actual Training Method.....40
- Chapter 5. INVESTIGATED PROBLEMS.....42
 - 5.1 Sequential XOR.....42
 - 5.1.1 Method.....43
 - 5.1.2 Data.....43
 - 5.1.2.a Fahlman's Version44
 - 5.1.2.a.i Parametric Determination.....44
 - 5.1.2.a.ii Actual Training.....47
 - 5.1.2.b CSM's Version.....47
 - 5.1.2.b.i Parametric Determination.....47
 - 5.1.2.b.ii Actual Training.....50

5.1.3 Comparison	50
5.2 Morse Code.....	52
5.2.1 Method.....	52
5.2.2 Data.....	53
5.2.2.a Falman's Network.....	53
5.2.2.a.i Parametric Determination.....	53
5.2.2.a.ii Actual training	56
5.2.2.b CSM's Network.....	56
5.2.2.b.i Parametric Determination.....	56
5.2.2.b.ii Actual Training.....	59
5.2.3 Comparison	59
5.3 Learning Distance.....	60
5.3.1 Method.....	61
5.3.2 Data.....	61
5.3.2.a Fahlman's Network.....	61
5.3.2.a.i Parametric Determination.....	61
5.3.2.a.ii Actual Training.....	65
5.3.2.b CSM's Network.....	65
5.3.2.b.i Parametric Determination.....	65
5.3.2.b.ii Actual Training.....	69
5.3.3 Comparison	69
5.4 Learning Distance with Noise.....	70
5.4.1 Method.....	70
5.4.2 Data for Learning Distance with Noise.....	71

5.4.3 Comparison	76
Chapter 6. CONCLUSION, INSIGHT, AND RECOMMENDATION.....	78
6.1 Conclusion	78
6.2 Insight.....	78
6.3 Recommendation.....	79
REFERENCES CITED	81
SELECTED BIBLIOGRAPHY.....	83
APPENDIX A.....	84

LIST OF FIGURES

Figure		Page
2.1	Biological neuron	7
2.2	Simple neural model	9
2.3	Step function	9
2.4	Graph of the logistic sigmoidal function	10
2.5	Single-layer perceptron with full connections	11
2.6	Two-layer perceptron with full connections	12
2.7	Jordan's network	22
2.8	Elman's network	23
2.9	Cascade-Correlation	27
2.10	Recurrent Cascade-Correlation	29
3.1	Fahlman's Recurrent Cascade-Correlation	34
3.2	CSM's Recurrent Cascade-Correlation	35

LIST OF TABLES

Table	Page
5.1.1 Determination of input learning coefficient η for sequential XOR with Fahlman's version.....	44
5.1.2 Determination of output learning coefficient η for sequential XOR with Fahlman's version.....	45
5.1.3 Determination of output growing rate μ for sequential XOR with Fahlman's version	45
5.1.4 Determination of input growing rate μ for sequential XOR with Fahlman's version	46
5.1.5 Actual training results of sequential XOR with Fahlman's version	47
5.1.6 Determination of input learning coefficient η for sequential XOR with CSM's version.....	48
5.1.7 Determination of output learning coefficient η for sequential XOR with CSM's version.....	48
5.1.8 Determination of output growing rate μ for sequential XOR with CSM's version.....	49
5.1.9 Determination of input growing rate μ for sequential XOR with CSM's version.....	49

5.1.10	Actual training results of sequential XOR with CSM's version.....	50
5.1.11	Comparing results of sequential XOR.....	51
5.2.1	Determination of input learning coefficient η for Morse code with Fahlman's version	53
5.2.2	Determination of output learning coefficient η for Morse code with Fahlman's version.....	54
5.2.3	Determination of output growing rate μ for Morse code with Fahlman's version	54
5.2.4	Determination of input growing rate μ for Morse code with Fahlman's version	55
5.2.5	Actual training results of Morse code with Fahlman's version	56
5.2.6	Determination of input learning coefficient η for Morse code with CSM's version.....	57
5.2.7	Determination of output learning coefficient η for Morse code with CSM's version.....	57
5.2.8	Determination of output growing rate μ for Morse code with CSM's version.....	58
5.2.9	Determination of input growing rate μ for Morse code with CSM's version.....	58
5.2.10	Actual training results of Morse code with CSM's version.....	59
5.2.11	Comparing results of Morse code.....	60
5.3.1	Arbitrary training for different distances with Fahlman's version	62

5.3.2	Determination of input learning coefficient η for learning distance with Fahlman's version.....	62
5.3.3	Determination of output learning coefficient η for learning distance with Fahlman's version.....	63
5.3.4	Determination of output growing rate μ for learning distance with Fahlman's version	63
5.3.5	Determination of input growing rate μ for learning distance with Fahlman's version	64
5.3.6	Actual training results of learning distance with Fahlman's version	65
5.3.7	Arbitrary training for different distances with CSM's version.....	66
5.3.8	Determination of input learning coefficient η for learning distance with CSM's version.....	66
5.3.9	Determination of output learning coefficient η for learning distance with CSM's version	67
5.3.10	Determination of output growing rate μ for learning distance with CSM's version.....	67
5.3.11	Determination of input growing rate μ for learning distance with CSM's version.....	68
5.3.12	Actual training results of learning distance with CSM's version.....	69
5.3.13	Comparing results for the learning distance problem.....	70
5.4.1	Actual training results for learning distance with 1.0% noise with Fahlman's version for the unchanged training set approach.....	72

5.4.2	Actual training results for learning distance with 1.0% noise with CSM's version for the unchanged training set approach.....	72
5.4.3	Actual training results for learning distance with 1.0% noise with Fahlman's version for the changed training set approach.....	73
5.4.4	Actual training results for learning distance with 1.0% noise with CSM's version for the changed training set approach.....	74
5.4.5	Percentages of correct bits for the unchanged training set approach	75
5.4.6	Percentages of correct bits for the changed training set approach	76
5.4.7	Comparing results for the learning distance with 1.0% noise for the unchanged training set approach.....	77
5.4.8	Comparing results for the learning distance with 1.0% noise for the changed training set approach.....	77

ACKNOWLEDGMENTS

I would like to thank Dr. Aaron Gordon for advising and supporting me for the past few years. His editing skills are greatly appreciated. Thanks to Drs. Ardel Boes and Ruth Maurer for their advice and editing. Finally, I deeply appreciate my parents and my family for their sacrifices for my own future. I owe a lot to my two brothers who died trying to come to America. They have given me silent strength.

Chapter 1

SIGNIFICANCE OF THE STUDY

Professionals, practitioners, and students whose fields relate to computer science, specifically neural networks, will find these studies scientifically significant because representational capability and training time of neural networks are their important features. Representation refers to the ability of the networks to model a specific function, and training time refers to the number of times an entire training set is presented.

Historically, artificial neural networks were inspired by the human brain. Organization and computational abilities of neurons, basic computing units of the human brain, give man the ability to think, act, and move. Such characteristics differentiate man from machine. To make a machine think, act, or move, the human brain needs to be studied.

Inspired by the human brain, a simple neural model was derived by Warren S. McCulloch and Walter Pitts in 1943 [1]. In this model, a neuron of the human brain was simulated by a model unit with incoming connections and an outgoing connection. The input and output connections correspond to synapses and axons in the brain, which are structures for receiving and sending signals, respectively. Each connection has its own weight which is the strength of the structure. Net input signal (or net input) of the unit is summation of products of its input signals and corresponding weights. This net input is modified by an

activation function, which is the threshold function in this model, acting as a potential membrane of the neuron in order to produce the unit's output of binary values, either 0 or 1.

Based on the neural model, artificial neural networks called perceptrons were proposed by F. Rosenblatt in 1958 [2]. The perceptron is formed by stacking one or more groups of units, called layers, on its set of input terminals. Layers between the input terminals and the outer most, or output, layer are called hidden layers, and their units are hidden units. The perceptron has connections from a unit to its next layer's units. There is no path from a unit's output to itself, to units of previous layers, or to units on layers after its next layer. Other names of the perceptrons are feed-forward, or non-recurrent, neural network.

In 1969, Minsky showed that the single-layer perceptron cannot simulate the exclusive-OR function, because it cannot geometrically separate sets of points corresponding to the classes of the input values [3]. There is a large class of functions that cannot be represented by the single-layer perceptron. This problem is referred to as the linearly separable problem, or linear separability, and those functions are linearly inseparable. Since linear separability limits the representational ability of the single-layer, it is important to determine if a given function is linearly separable. Nevertheless, this determination is not simple if the number of variables is large. By the late 1960's this problem was well understood, and later research addressed linearly inseparable problems by using multilayer perceptrons with nonlinear activation functions between layers [4].

In 1986, a learning algorithm called the back-propagation algorithm, or backprop, was devised by Rumelhart, Hinton, and Williams [5,6]. This algorithm

provides a systematic method to train multilayer perceptrons. An input connection weight of a hidden unit is adjusted with a weight step directly proportional to errors it contributes to units to which its output is fed. Input connection weights of the output layer are adjusted using error between the target and actual output. In 1988, Scott E. Fahlman introduced a quick-propagation algorithm, or quickprop [7]. This learning algorithm has improved learning time and stability of artificial neural networks. Adjusting connection weight with possible large weight step reduces the learning time. Stability is increased by choosing new weight steps in the direction toward a local error minimum and by reducing oscillation of connection weights about a local minimum point.

A disadvantage of the perceptron with present learning algorithms is its inability to learn time-dependent, or context-dependent, problems. That is, its current outputs need to be based on the current inputs as well as the history of past inputs. Another disadvantage is that it can not determine the number of hidden units, or size, that it needs.

In 1986, Terrence J. Sejnowski and Charles R. Rosenberg introduced the window sliding technique for teaching non-recurrent networks to learn time-dependent problems [8]. However, the problem of determination of the number of hidden units was still unsolved. In 1986, a sequential recurrent network was developed by Michael I. Jordan [9]. Jordan's network could handle time-dependent problems. Output of the network depends on the previous states and outputs of the network. A weakness of Jordan's network is that it can not determine the number of needed hidden units. In 1989, Jeffrey L. Elman

proposed a different sequential recurrent network [10]. In Elman's network, every hidden unit has its own context unit. Output of a hidden unit depends on the previous outputs of all hidden units and all current inputs. Elman's network has the same mentioned strength and weakness as Jordan's network. In 1989, Ronald J. Williams and David Zipser proposed a fully connected network called continually-running fully-recurrent network for handling time-dependent problems [11].¹ Such a network has connections between every pair of units. In addition, the network has connections from each input terminal to each unit.

Disadvantages of the fully connected network are computational expense and the determination of the number of hidden units.

In 1990, Scott E. Fahlman and Christian Lebiere developed the cascade-correlation neural network [12]. New hidden units are added to the network one at a time as they are needed during training. Advantages of this architecture are that it learns very quickly, and it determines its own size. In addition, the structure it has built is retained even if the training set changes. In 1991, Scott E. Fahlman developed a recurrent cascade-correlation architecture which is the cascade-correlation architecture with recurrent connections [13]. This architecture can handle time-dependent, or context-dependent, problems. During training, new hidden units with recurrent connections are added to the network one at a time as they are needed. The advantages of Cascade-Correlation are also retained.

Recently, a modified version of Recurrent Cascade-Correlation was

¹ In this architecture, "recurrent" does not mean self-connection. It means that there is a full path for any set of units.

developed at the Colorado School of Mines (CSM) [14]. However, this network has not been evaluated. Determination of performance of this architecture is necessary.

The purpose of this study is to compare CSM's and Fahlman's Recurrent Cascade-Correlation. Average number of successful learnings, hidden units, learning time (training epochs and CPU running time in seconds), and errors of the two networks for some problems are experimentally collected and compared. The following problems will be investigated: sequential XOR, Morse code, learning distance, and learning distance with noise. The network that has a broader learning capability, shorter learning time, smaller error, and smaller size is more attractive.

Determination of the relative performance of CSM's and Fahlman's Recurrent Cascade-Correlation is a contribution to the development of artificial neural networks.

Chapter 2

NEURAL NETWORK ARCHITECTURES

Before Fahlman's and CSM's Recurrent Cascade-Correlation are presented, an overview of neural networks is presented. This entire chapter is devoted to describing neural network architectures and algorithms.

2.1 Human Brain

Computation of the human brain is performed by neurons, which are its basic computation units. In the human brain, there are about 10^{11} neurons and 10^{15} connections among neurons. Each neuron receives its input signals from other neurons by structures called synapses. These input signals are integrated by the neuron; an output is generated, which is then transmitted to other neurons by structures called axons. Each neuron contains an internal, continuous, potential membrane that will propagate an all-or-none action potential if its integrated input signal exceeds its threshold. Synapses come with 2 different particular types: excitatory and inhibitory. Excitatory synapses make it more likely for the receiving neurons to fire action potentials. On the other hand, inhibitory synapses make the receiving neurons less likely to fire action potentials. Figure 2.1 is a schematic of biological neurons.

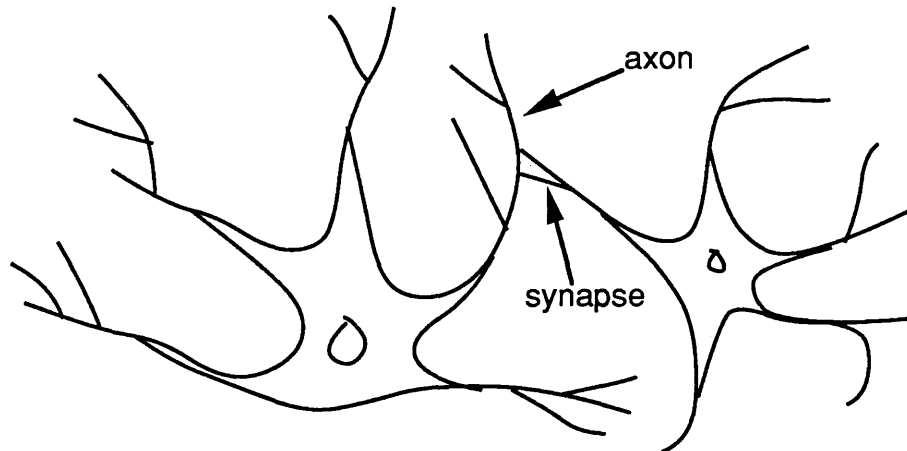


Figure 2.1. Biological neuron

2.2 Simple Neural Model

The simple neural model is an attempt to model the biological neuron. Another name for the simple neural model is the McCulloch and Pitts' neuron because it was proposed by McCulloch and Pitts (1943).

In artificial neural networks, synapses and axons correspond to connections. Synapse strength is referred to as connection strength, connection weight, or just weight. The artificial neural model computes its output by applying the threshold function to its net input. The net input is the difference between its weighted sum and its threshold value, where the weighted sum is the sum of all products of its inputs and corresponding weights of the inputs. The equation for computing output of a neuron i , also known as the McCulloch-Pitts' equation, is:

$$n_i(t+1) = \Theta\left(\sum_{j=1} n_j(t)w_{ij} - \mu_i\right)$$

where n_i and n_j represent states of neuron i and j , respectively, w_{ij} is strength of

the synapse connecting neuron j to i , μ_i is a threshold value of neuron i , and Θ is the step function.

$$\Theta(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Figure 2.2 and 2.3 are schematic diagrams of the model and the step function, respectively.

Since many real neurons perform nonlinear summation of their inputs and do not all have the same fixed delay, the McCulloch-Pitts equation is generalized to

$$n_i = g\left(\sum_{j=1} n_j w_{ij} - \mu_i\right)$$

where g is a nonlinear activation function, gain function, or squashing function. One of the activation functions is the logistic sigmoidal function:

$$g(x) = \frac{\max - \min}{1 + e^{-x}} + \min$$

where \max and \min are upper and lower bound of the function's range, respectively. Figure 2.4 is a graph of the function. If an input value of $+1$, whose connection weight to the neural model is $w_{i0} = -\mu_i$ is added, then the McCulloch-Pitts equation becomes

$$n_i = g\left(\sum_{j=0} n_j w_{ij}\right) = g(h_i)$$

where h_i is a net input of the neuron i . This additional input is called the bias.

For simplicity, network learning equations are written in matrix notation. Let \mathbf{V} be an input row vector of size m where m is number of inputs, and let \mathbf{W} be a connection strength column vector of size m . Then, net input h to a neuron is a

dot product of \mathbf{V} and \mathbf{W} , $h = \mathbf{VW}$, and its output is $O = g(h)$ where g is an activation function.

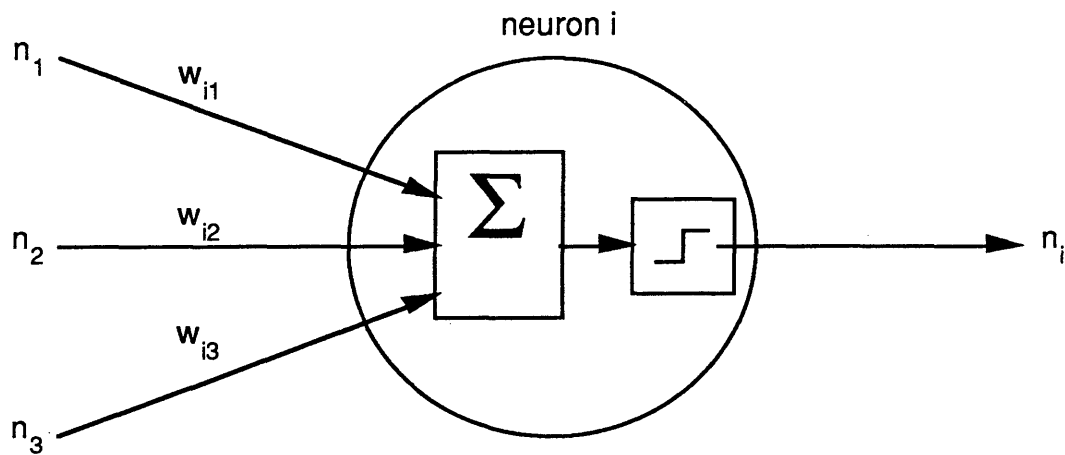


Figure 2.2. Simple neural model

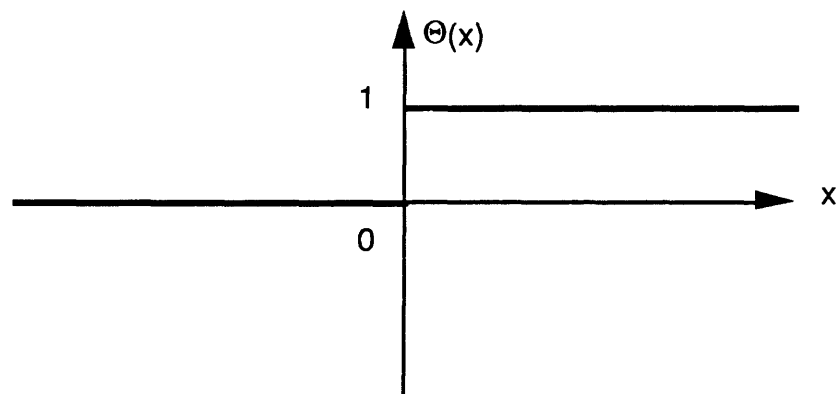


Figure 2.3. Step function

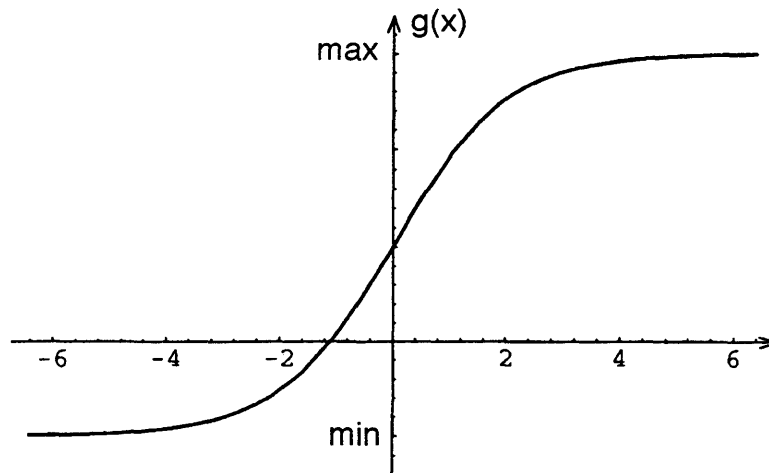


Figure 2.4. Graph of the logistic sigmoidal function.

2.3 Layered Feed-Forward Networks (Perceptrons)

Feed-forward networks, or perceptrons, can have one or more unit layers. There is a set of input terminals whose role is feeding input patterns to the network. After this set of terminals, there are one or more intermediate, or hidden, layers, followed by an output layer. In this paper, the set of input terminals is not counted as a layer. A restriction that applied to this type of network is that there is no connection from a unit to itself, to units on the same layer, on previous layers, and to units on layers after its next layer. For the purpose of generality, a unit receives all outputs from its previous layer, or from all input terminals if it is on the first layer.

2.3.1 Single-Layer Neural Networks

Single-layer networks are networks that have only one layer, the output layer. Thus, they have no hidden layer. These networks are referred to as simple perceptrons. In a simple perceptron, the output of a unit i is computed by the equation

$$O_i = g\left(\sum_{k=0} \xi_k w_{ik}\right) = g(h_i)$$

where g is an activation function, ξ_k is the k^{th} input of a pattern, w_{ik} is a link weight from the k^{th} input to the unit i , and h_i is a net input of the unit i . Figure 2.5 is an example of a single-layer perceptron. There exists a large class of functions that can not be represented by single-layer perceptrons. Their capabilities are limited to linearly separable problems.

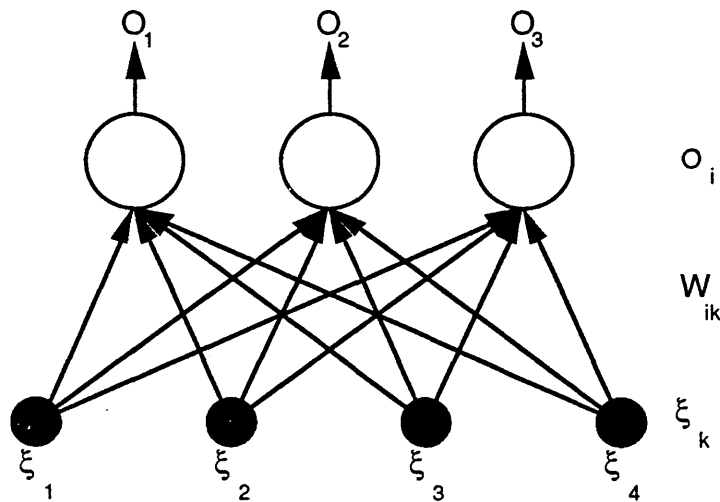


Figure 2.5. Single-layer perceptron with full connections

Using the vector notation, the learning equation is still $\mathbf{OUT} = \mathbf{G}(\mathbf{H}) = \mathbf{G}(\mathbf{VW})$. \mathbf{V} is a row vector of size m , and where m is the number of inputs. \mathbf{W} is a matrix of size m by n where n is the number of outputs. \mathbf{VW} , or \mathbf{H} , is a row vector of size n .

2.3.2 Multi-Layer Neural Networks

Multi-layer networks are formed by simply stacking groups of layers. Figure 2.6 is an example of a two-layer perceptron.

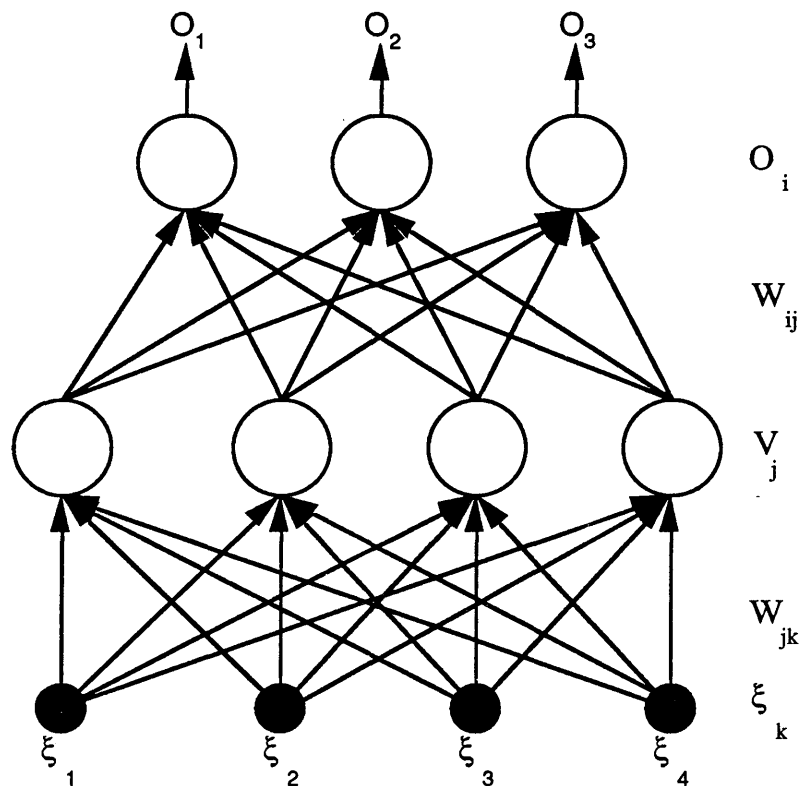


Figure 2.6. Two-layer perceptron with full connections

Consider a two-layer network. Outputs of units at the first, or hidden, layer is matrix \mathbf{OUT}_1 where $\mathbf{OUT}_1 = \mathbf{G}(\mathbf{VW}_1)$, and outputs of units at the second, or output, layer is $\mathbf{OUT}_2 = \mathbf{OUT} = \mathbf{G}(\mathbf{OUT}_1\mathbf{W}_2) = \mathbf{G}(\mathbf{G}(\mathbf{VW}_1)\mathbf{W}_2)$. If \mathbf{G} is a linear activation function, then $\mathbf{OUT} = \mathbf{G}(\mathbf{VW}')$ where $\mathbf{W}' = \mathbf{W}_1\mathbf{W}_2$. Therefore, if the activation functions between layers are linear, then any multilayer network can be replaced by a single-layer one. Nonlinear activation functions between layers are necessary to expand the network's capability beyond that of the single-layer network. Multilayer networks with nonlinear activation functions solved the restriction of linear separability in single-layer networks.

2.3.3 Training Algorithm for Single-Layer Perceptrons

Neural networks learn by adjusting their link weights. The weight change of connection from an input terminal k to an output unit i , Δw_{ik} , is directly proportional to the partial derivative of a cost or error function E with respect to the link weight w_{ik} .

$$\Delta w_{ik} = -\eta \frac{\partial E}{\partial w_{ik}}$$

where η is a learning rate coefficient serving to adjust the size of the weight change.

The cost function E is the sum of the squares of the output errors.

$$E = \frac{1}{2} \sum_{ip} [\zeta_i^p - g(h_i^p)]^2$$

$$\frac{\partial E}{\partial w_{ik}} = - \sum_p (\zeta_i^p - O_i^p) g'(h_i^p) \xi_k^p = - \sum_p \delta_i^p \xi_k^p$$

where ζ_i^p , and O_i^p are the desired output i of a training pattern p , and the corresponding actual output i , g' is the derivative of the unit's activation function with respect to its net input h_i , ξ_k is the k^{th} input of the pattern p , and δ_i^p is the error scaled by g' of the output unit i for the pattern p , $\delta_i^p = (\zeta_i^p - O_i^p)g'(h_i^p)$.

The weight change becomes

$$\Delta w_{ik} = \eta \sum_p \delta_i^p \xi_k^p$$

The above equation for computing the weight change after a presentation of an entire training set is called the Windrow-Hoff, or the delta, rule. The equation for weight change after an input pattern is

$$\Delta w_{ik} = \eta \delta_i^p \xi_k^p$$

The following is the algorithm using the Windrow-Hoff rule for single-layer perceptrons.

Training algorithm for single-layer perceptrons

1. Select an input pattern p , and propagate it forward to compute the network's outputs:

$$O_i = g\left(\sum_k \xi_k w_{ik}\right) = g(h_i)$$

where O_i is an output of a unit i , ξ_k is the k^{th} input of the training pattern, w_{ik} is the link weight from the k^{th} input terminal to the output unit i , g and h_i are an activation function and a net input of that output unit i , respectively.

2. Compute the weight correction associated with the output unit i :

$$\Delta w_{ik} = \eta \delta_i \xi_k$$

where η is a learning rate coefficient serving to adjust the size of the weight change, and δ_i is the error of the output unit i , scaled by the derivative of g with respect to its net input.

3. Update all weights by the equation:

$$w_{ik}^{\text{new}} = w_{ik}^{\text{old}} + \Delta w_{ik}$$

4. Go back to step 1 and repeat for the next pattern until all training patterns have been processed.
5. Test the network by propagating each testing input pattern forward to compute the network's outputs. Compute the total error of outputs.¹
6. If the total error is acceptable or the number of training epochs exceeds the maximum allowable number of training epochs, stop training. Otherwise, go to step 1 and repeat for the next presentation of the training set.

2.4 Back Propagation

Back propagation is an extension of the simple training algorithm in section 2.3.3. It is a method of training multilayer networks. The back propagation algorithm provides a systematic way to adjust weights of the hidden units. During training, given a training pair - an input pattern and its corresponding target output - there are two passes: forward and reverse. During the forward pass, outputs are computed; next, output errors are back propagated to adjust weights during the reverse pass.

¹ Method of output error computation is up to users, e.g. number of wrong outputs, sum of squares of difference between actual and desired output, etc.

This network training algorithm is similar to the single-layer perceptrons' algorithm except for the computation of the error, delta δ , for the hidden units. At the output layer, delta is computed as in the training algorithm of the single-layer perceptron. However, at the hidden layers, computation of delta must be handled differently, because we do not know the desired output. The delta of a hidden unit is directly proportional to the summation of errors it contributes to units to which its output is fed.

Another modification is made to the equation of computing weight correction used in the single-layer perceptron by adding a term directly proportional to the previous weight change of the unit. The purpose of this modification is to give each connection weight some inertia, or momentum, so that it tends to change in the direction of the average downhill force toward a local minimum of the error vs. weight curve, instead of oscillating widely about the local minimum when the learning rate is large. The following is the back propagation algorithm for adjusting connection weights using the Windrow-Hoff rule with more details.

Back Propagation Algorithm

1. Select an input pattern p , ξ^p , and apply it to the input layer ($r = 0$) so that $V_i^0 = \xi_i^p$, where V_i^0 is the output of a unit i on layer 0.
2. Propagate the signals forward through the network to compute outputs for each unit i of layer r until the final outputs V_i of output layer R have all been calculated by the equation:

$$V_i^r = g\left(\sum_j V_j^{r-1} w_{ij}^r\right) = g(h_i^r)$$

for $r = 1, 2, \dots, R$.

3. Compute the error deltas for the output layer by comparing the actual outputs, V_i^R , with desired ones, ζ_i^R , at the output layer R , and scaling these differences by the derivatives of the units' activation functions g with respect to their net inputs, h_i^R , for the pattern p being considered.

$$\delta_i^R = (\zeta_i^R - V_i^R)g'(h_i^R) \quad (2.1)$$

4. Compute the error deltas for the preceding layers by propagating the errors backward.

$$\delta_j^{r-1} = g'(h_j^{r-1}) \sum_i w_{ij}^r \delta_i^r \quad (2.2)$$

for $r = R, R-1, \dots, 2$ until a δ has been calculated for every unit.

5. Update all connection weights by

$$w_{ij}^{\text{new}} = w_{ij}^{\text{old}} + \Delta w_{ij}$$

where

$$\Delta w_{ij} = \eta \delta_i V_j + \alpha \Delta w_{ij}^{\text{old}}$$

and α , a constant, is the momentum factor.

6. Go back to step 1 and repeat for the next pattern until all training patterns have been processed.

2.5 Quick Propagation

Changing the weights during training is an attempt to minimize the total error E in the network. The delta rule in back-propagation can be thought of as

gradient descent in error space. The back propagation algorithm is slow because it takes very small steps down the gradient. In addition, according to equation 2.2, when the derivative of the unit's activation function $g'(h_i)$ approaches zero, only a very small fraction of error is propagated backward. Quickprop, derived by Scott E. Fahlman, is much faster than backprop. Backprop and quickprop are similar except for a modification of the derivative and the method of computing weight steps.

To see the difference, we will first look more closely at what is happening with backprop. Consider the logistic sigmoidal function with a range of $[0,1]$. Then, its derivative with respect to its net input h_i is

$$g'(h_i) = V_i(1 - V_i)$$

where V_i is the output of a unit i . This derivative approaches zero if V_i approaches either 1 or 0. Since delta is directly proportional to the derivative as shown in equation 2.1 and 2.2, it also approaches zero. Thus, only a very small fraction of error is propagated backward although the network's outputs can possibly have maximum error. This case happens when the desired output is 1 and the corresponding actual output is zero, or vice versa. To get around this problem, 0.1 is added to the derivative before using it to scale the error.

Fahlman assumed that the error vs. weight curve is a parabola opening upward [7]. Based on the change in slope of the curve between previous and current time, he proposed the equation for computing next weight step:

$$\Delta w(t) = \frac{S(t)}{S(t-1) - S(t)} \Delta w(t-1) \quad (2.3)$$

where $S(t)$ and $S(t-1)$ are the current and previous value of $\partial E/\partial w$ and are the slopes of the error vs. weight curve.

According to equation 2.3, if the previous weight step is zero, then the new weight step is also zero. So, the network makes no, or little, progress. To start the training process for any weight that has a previous step size of zero, the gradient descent term $\eta\delta_i V_j$ is added to equation 2.3:

$$\begin{aligned}\Delta w_{ij}(t) &= \frac{S(t)}{S(t-1) - S(t)} \Delta w_{ij}(t-1) + \eta\delta_i V_j \\ &= \frac{S(t)}{S(t-1) - S(t)} \Delta w_{ij}(t-1) - \eta S(t)\end{aligned}$$

This equation works well when the weight is moving down the curve. However, if the previous and current slope have opposite direction, or opposite sign, then the gradient descent term will make the new weights overshoot the local minimum of error. To avoid this oscillation about the minimum, the gradient descent term is added only if the previous and current slope have the same direction.

Another problem of using equation 2.3 to update the weights is when the current slope is in the same direction as the previous slope and has the same or larger size in magnitude. In this case, the network would take an infinite weight step, up the current slope toward a local maximum, or actually move backwards. This problem is corrected by not allowing a new weight step to be larger than the maximum growth factor μ times the previous weight step. In summary, the algorithm for computing weight steps used in quickprop is described as follows.

Algorithm for computing weight steps

1. Compute weight step $\Delta w(t)$ by the equation:

$$\Delta w(t) = \frac{S(t)}{S(t-1) - S(t)} \Delta w(t-1)$$

where $S(t)$ and $S(t-1)$ are the current and previous values of $\partial E/\partial w$, or are slopes of the error E vs. weight curve at time t and $t-1$.

2. If the previous and current slopes have the same sign, or the same direction, then add the gradient descent term $\eta\delta V$, or $-\eta S(t)$, to the computed weight in step 1.

Briefly, if $S(t)S(t-1) > 0$ then $\Delta w(t) = \Delta w(t) - \eta S(t)$

3. If the 2 slopes are the same sign and the magnitude of the current slope is greater or equal to the previous slope, or the new weight step is greater than the previous one, then the new weight step is the maximum growth factor μ times the previous weight step.

In mathematical expression:

if $(S(t)S(t-1) > 0 \text{ AND } |S(t)| \geq |S(t-1)|) \text{ OR } (|\Delta w(t)| > |\Delta w(t-1)|)$

then $\Delta w(t) = \mu \Delta w(t-1)$

2.6 Window Sliding Technique

Window sliding technique, introduced by Terrence J. Sejnowski and Charles R. Rosenberg in 1986, is a method for handling time-dependent problems using non-recurrent networks [8]. The back-propagation algorithm is used in

training. The network's input layer is composed of more than one group of input terminals, called windows. At every next time step, the training set is slid by one window. The sliding direction is always the same during the training. The desired output is associated with the center window. Other windows provide context for this output decision. A problem of this window sliding technique is that it cannot determine the number of hidden units and the number of windows it needs.

2.7 Jordan's Network

Michael I. Jordan proposed a network for handling time dependent problems in 1986 [9]. Back propagation is used for training. Beside having input, hidden, and output layers, the network also has a state layer. Inputs of a state unit are its previous state and previous output of the network. Thus, each state unit has a recurrent connection. The output of a state unit at time $t+1$ is given by

$$s_i(t+1) = s_i(t)w_s + V_j(t)$$

where s_i and w_s are the state and the recurrent weight of a state unit i , respectively, and V_j is the output of unit j of the network. Figure 2.7 is an example of Jordan's network. A disadvantage of Jordan's network is that it cannot determine the number of hidden units that it needs. However, the network has improved over the window sliding technique because there is no need for the input terminal groups. Input patterns are sequentially fed to the net.

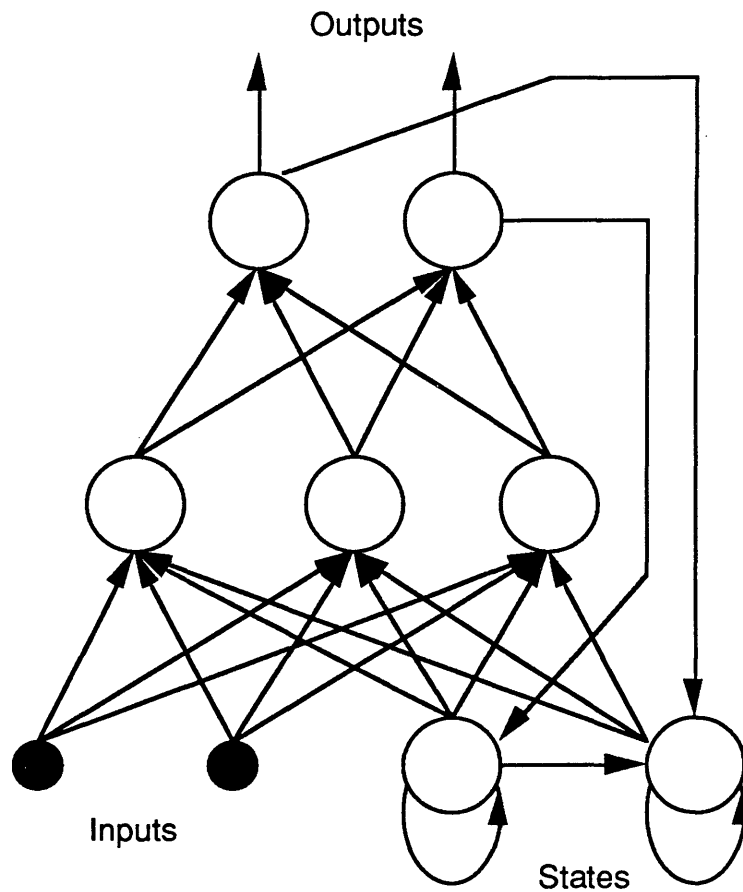


Figure 2.7. Jordan's Network

2.8 Elman's Network

Elman's network is another network for handling time-dependent, or context-dependent, problems [10]. Every hidden unit of the network has its own context unit serving as a memory to store previous output of the hidden unit. The values of context units and input terminals are fed to every hidden unit in the network. Output V_i of a hidden unit i is computed by the equation

$$V_i = g\left(\sum_k w_{ik} \xi_k + \sum_s w_{is} V_s\right)$$

where ξ_k is value of an input terminal, w_{ik} is the connection weight from the input terminal k to unit i , V_s is output of a context unit s , and w_{is} is the weight from the context unit s to the hidden unit i . In turn, outputs of all hidden units activate the network's outputs. The equation for computing the network's outputs V_o is

$$V_o = g\left(\sum_i w_{oi} V_i\right)$$

where V_i is output of a hidden unit i and w_{oi} is the weight connection from unit i to output o . The training algorithm used in Elman's network is back propagation. Elman's network cannot determine the number of hidden units that it needs. Figure 2.8 illustrates an Elman network.

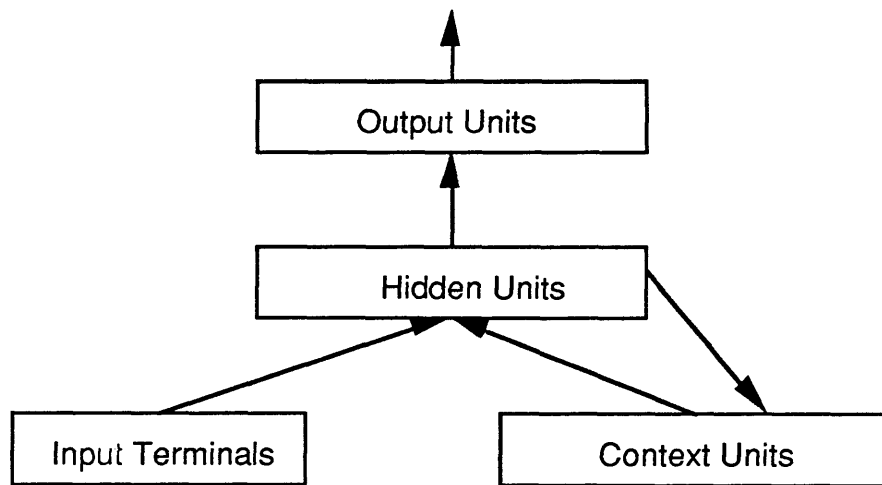


Figure 2.8. Elman's Network

2.9 Fully Recurrent Network

In 1989, a fully recurrent network was devised by Ronald J. Williams and David Zipser to handle time-dependent problems [11].² The network is fully connected. That is, there are connections between every pair of units and between each input terminal and each unit.

In the network, any particular weight change Δw_{ij} can be computed by the gradient descent equation

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

for

$$E = \frac{1}{2} \sum_{k=1}^n e_k^2$$

where n is the number of units in the network and e_k is the output error at each unit k .

$$e_k = \begin{cases} \zeta_k - V_k & \text{if } \zeta_k \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

then

$$\frac{\partial E}{\partial w_{ij}} = -\sum_{k=1}^n e_k \frac{\partial V_k}{\partial w_{ij}}$$

where V_k is the output of a unit k , with

$$\frac{\partial V_k(t+1)}{\partial w_{ij}} = g'(h_k) \left[\sum_{r=1}^n w_{kr} \frac{\partial V_r(t)}{\partial w_{ij}} + \delta_{ik} V_j(t) \right]$$

² In this architecture, "recurrent" does not mean self-connection. It means that there is a full path for any set of units.

where $g'(h_k)$ is the derivative of the unit k 's activation function with respect to its net input h_k , m is the number of input terminals, and δ_{ik} is the Kronecker delta.³

The weight change equation becomes

$$\Delta w_{ij}(t) = \eta \sum_{k=1}^n e_k(t) \frac{\partial V_k(t)}{\partial w_{ij}}$$

with initial conditions

$$\frac{\partial V_k(t_0)}{\partial w_{ij}} = 0$$

A disadvantage of the network algorithm is that it requires non-local communication and is computationally expensive. For a fully recurrent network, a weight matrix with $n^2 + mn$ entries is kept along in order to compute $\partial V_k / \partial w_{ij}$. Since there are n units in the network, a total storage space needed is $n^3 + mn^2$ which is very large if n is big.

2.10 Cascade-Correlation Networks

Cascade-correlation architecture is a network in which single-unit hidden layers are cascaded. Each new hidden unit is added to the network, one at a time, with the magnitude of correlation between its output and the residual error signal maximized.

A cascade-correlation architecture is illustrated in figure 2.9. Initially, it has input terminals, output units, and no hidden units. Every input terminal is connected to every output unit. As a new hidden unit is added to the network, it

³ δ_{ik} is 1 if $i = k$, zero otherwise.

receives connections from the network input terminals and also from all pre-existing hidden units. Output of a new hidden unit is fed to all output units of the network.

The training algorithm has two phases: output and input (or candidate) weight training. During the output weight training phase, hidden unit input weights are frozen; only weights of output connections are trained repeatedly. Output connections are trained using the quickprop algorithm with no need to back propagate error through hidden units. When no significant error reduction has occurred, after a certain number of training cycles (epochs), the network is run one last time over the entire training set to measure the error. If the output errors are acceptable, then training stops; otherwise, the input training phase takes place. At the start of the input training phase, a new hidden unit is added; however, its output is not yet connected to the output units. During this phase, input connections of this new hidden unit are trained. The goal of this training is to maximize the correlation between the candidate output and the network's output errors. After this training phase, this new hidden unit is added to the network. Output weight training is then continued. Instead of having a single candidate unit, the network can have a pool of candidate units, each with a set of random initial weights, and those candidate units are trained in parallel. At the end of the candidate training phase, the candidate with highest correlation is added to the network. The purpose of having a pool of candidate units is that we don't want to accidentally add a useless unit to the network. The algorithm of input-weight training phase is described below.

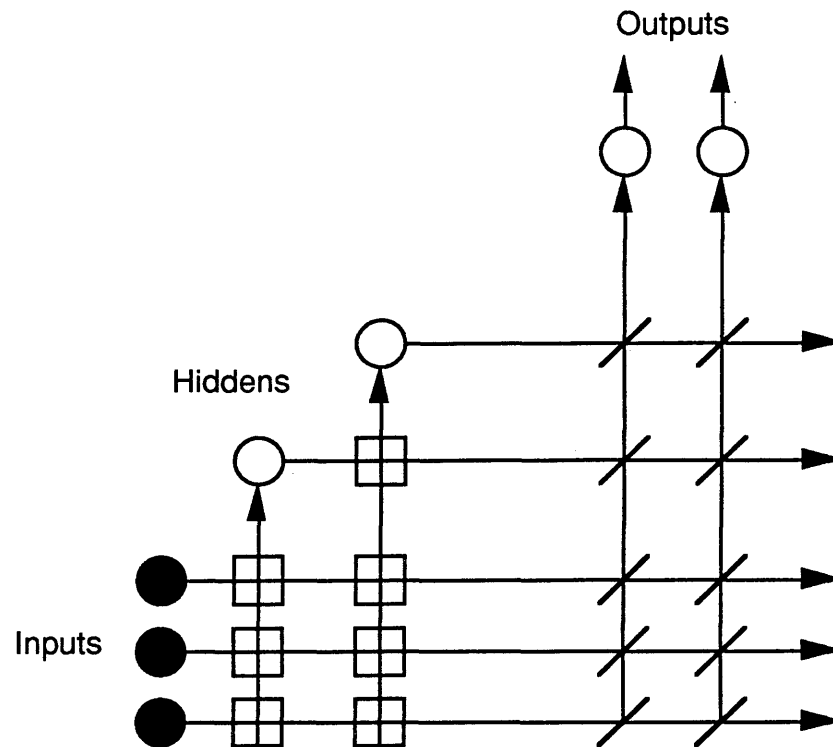


Figure 2.9. Cascade-Correlation

Input or candidate weight training algorithm

1. Compute the correlation S

$$S = \sum_o \left| \sum_p (V^p - \bar{V})(E_o^p - \bar{E}_o) \right|$$

where o is the network's output at which the error E is measured, p is a training pattern, \bar{V} and \bar{E}_o are the average of candidate output V and E_o over all patterns.

2. Compute partial derivative of S with respect to each of the candidate unit's

incoming weights w_j .

$$\frac{\partial S}{\partial w_j} = \sum_{op} \sigma_o (E_o^p - \bar{E}_o) g'(h^p) V_j^p$$

where σ_o is the sign of the correlation between the candidate value and output o , g' is the derivative for pattern p of the candidate unit's activation function with respect to the sum of its inputs h , and V_j^p is the input the candidate unit receives from unit j for pattern p .

3. Using quickprop, compute the weight steps by the gradient ascent rule for correlation S .
4. Update input weight by the equation

$$w_j^{\text{new}} = w_j^{\text{old}} + \Delta w_j$$

5. Go to step 1 until the increase of S is not significant.

Cascade-Correlation has several advantages. It learns very quickly, determines its own size and topology, retains the structures it has built even if the training set changes, and requires no back-propagation signals through the connections of the network.

2.11 Recurrent Cascade-Correlation Networks

Recurrent cascade-correlation architecture is a recurrent version of the Cascade-Correlation learning architecture of Fahlman and Lebiere [12]. In Recurrent Cascade-Correlation, there are recurrent connections for every hidden unit, these connections give the network the capability to learn time-dependent problems. These recurrent connections are implemented by having

context units as in Elman's networks, and are also trained during the candidate training phase. Figure 2.10 is a schematic drawing of Recurrent Cascade-Correlation.

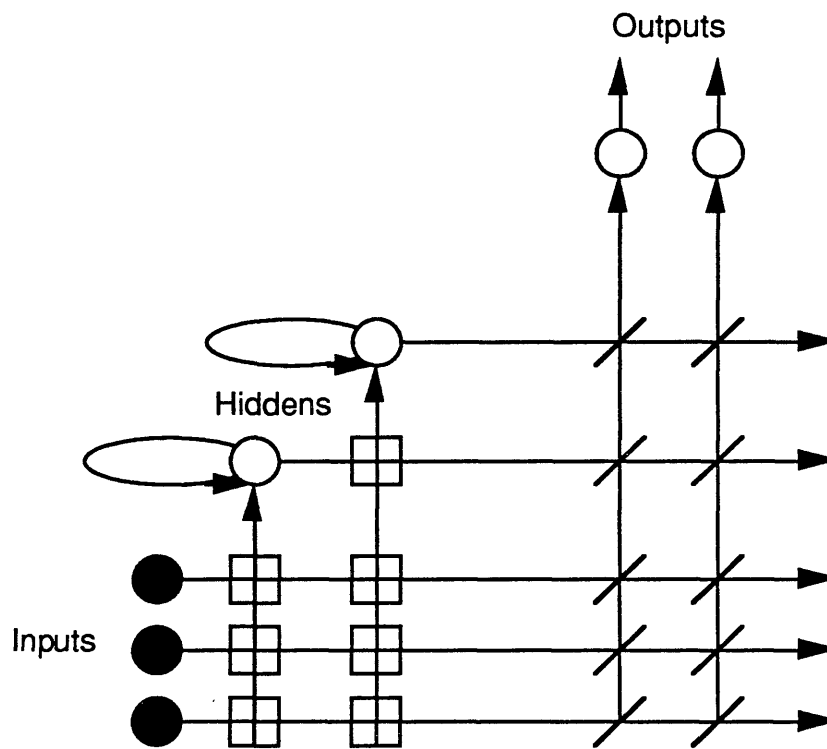


Figure 2.10. Recurrent Cascade-Correlation

During the candidate training phase, the network adjust input weights w_j and recurrent weight w_s for each candidate unit to maximize its correlation score. As in Cascade-Correlation, the correlation S is

$$S = \sum_o \left| \sum_p (V^p - \bar{V})(E_o^p - \bar{E}) \right|$$

where o is the network's output at which the error E is measured, p is a training pattern, \bar{V}^p and \bar{E}_o^p are the average of candidate output V and error E_o over all patterns, respectively.

The partial derivative of S with respect to an input or recurrent weight of the candidate unit is:

$$\frac{\partial S}{\partial w_j} = \sum_{po} \sigma_o (E_o - \bar{E}) \frac{\partial V}{\partial w_j}$$

where σ_o is the sign of the correlation between a candidate value and output o , and w_j is the connection weight to the candidate unit from a hidden unit, an input terminal, or the context unit of the candidate.

For

$$V(t) = g\left(\sum_k w_k V_k(t) + w_s V(t-1)\right)$$

where V_k is input to the candidate unit from previous hidden unit or from an input terminal, and w_s is the recurrent weight of the candidate unit.

Then, the partial derivative of the candidate output with respect to its recurrent weight w_s is:

$$\frac{\partial V(t)}{\partial w_s} = g'(h) \left[V(t-1) + w_s \frac{\partial V(t-1)}{\partial w_s} \right]$$

And, its derivative with respect to other input connection weight is:

$$\frac{\partial V(t)}{\partial w_j} = g'(h) \left[V_j(t) + w_s \frac{\partial V(t-1)}{\partial w_j} \right]$$

Since there are two different equations for computing the partial derivative of a candidate output V , there are also two different equations for the derivative of S . Therefore, care must be taken when computing the partial derivative of S . The training algorithm for Recurrent Cascade-correlation is similar to Cascade-Correlation except for the computation of the partial derivative of S .

Chapter 3

FAHLMAN'S AND CSM'S

RECURRENT CASCADE-CORRELATION NETWORKS

Because the relative performance of Fahlman's and CSM's Recurrent Cascade-Correlation will be compared, it is worth devoting one separate chapter to describing them. For clarity of explanation, the two networks are described by using the Elman style architecture. That is, instead of having a direct self-connection for hidden units, each hidden unit has its own context unit serving as its memory storage to hold its previous output value. Values of context units are fed back to their corresponding hidden units. Since the two networks are similar to Recurrent Cascade-Correlation, described in 2.11, and for the purpose of comparison, their differences in architectures and computations are only described.

3.1 Fahlman's Networks

In Fahlman's networks, actually the recurrent cascade-correlation networks, values of context units are only fed to their corresponding hidden unit. The equations for computing hidden units' and candidates' outputs is

$$V_i(t) = g\left(\sum_j w_{ij} V_j(t) + w_{is} V_i(t-1)\right)$$

where V_i is output of the considered hidden or candidate unit, V_j is outputs of

previous hidden units, or of input terminals.

The equations for computing outputs of the network is

$$V_o(t) = g\left(\sum_i w_{oi} V_i(t)\right)$$

where V_o is an output of the network, V_i is output of hidden units or input terminals, and w_{oi} is the connection weight from units i to o . Figure 3.1 is a schematic of Fahlman's network.

3.2 CSM's Networks

In CSM's network, values of context units are fed to their corresponding hidden units as in Fahlman's network. However, those context values are also propagated to all hidden units of higher layers and to all output units of the network. Output of a hidden unit or a candidate unit is computed by the equation

$$V_i(t) = g\left(\sum_j w_{ij} V_j(t) + w_{is} V_i(t-1) + \sum_j w_{ijs} V_j(t-1)\right)$$

where V_i is output of the computed unit i , V_j is inputs to unit i from previous hidden units, or from input terminals j , w_{ij} is the connection weight from units j to i , w_{is} is the self-connection weight of unit i , and w_{ijs} is connection weight from the context units of hidden units j to unit i .

The equation for computing the network's outputs is

$$V_o(t) = g\left(\sum_i w_{oi} V_i(t) + \sum_i w_{ois} V_i(t-1)\right)$$

where V_o is an output o of the network, V_i is inputs from hidden units, or from input terminals i , w_{oi} is connection weights from units i to output o , and w_{ois} is

connection weight from context units of hidden units i to output o . Figure 3.2 is a schematic of CSM's network.

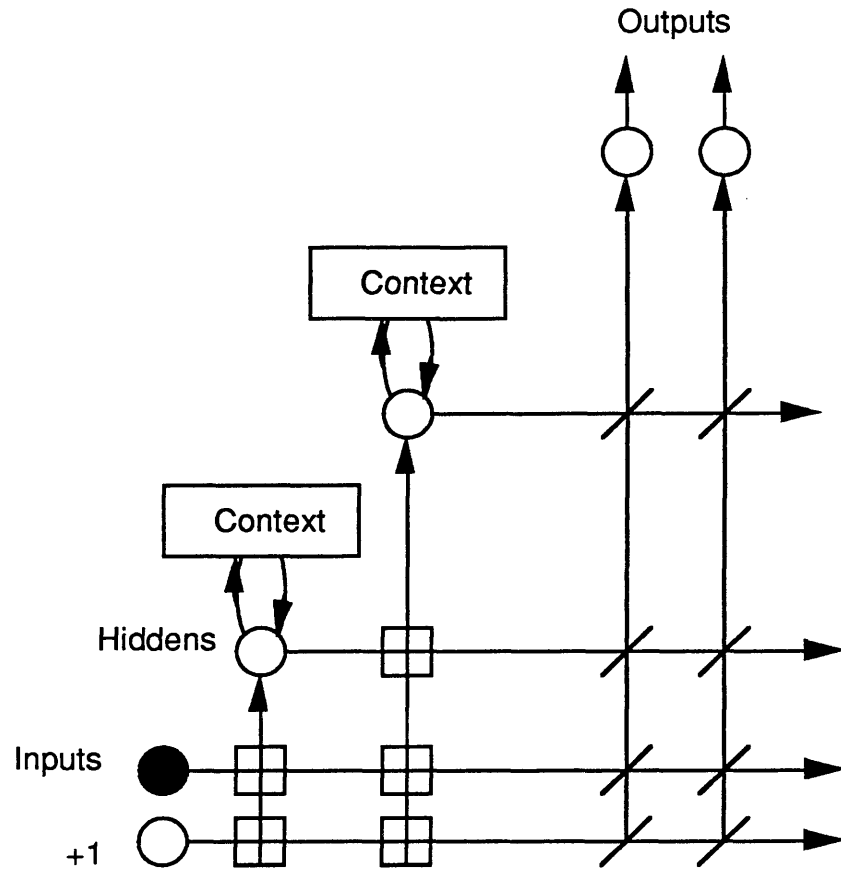


Figure 3.1. Fahlman's Recurrent Cascade-Correlation

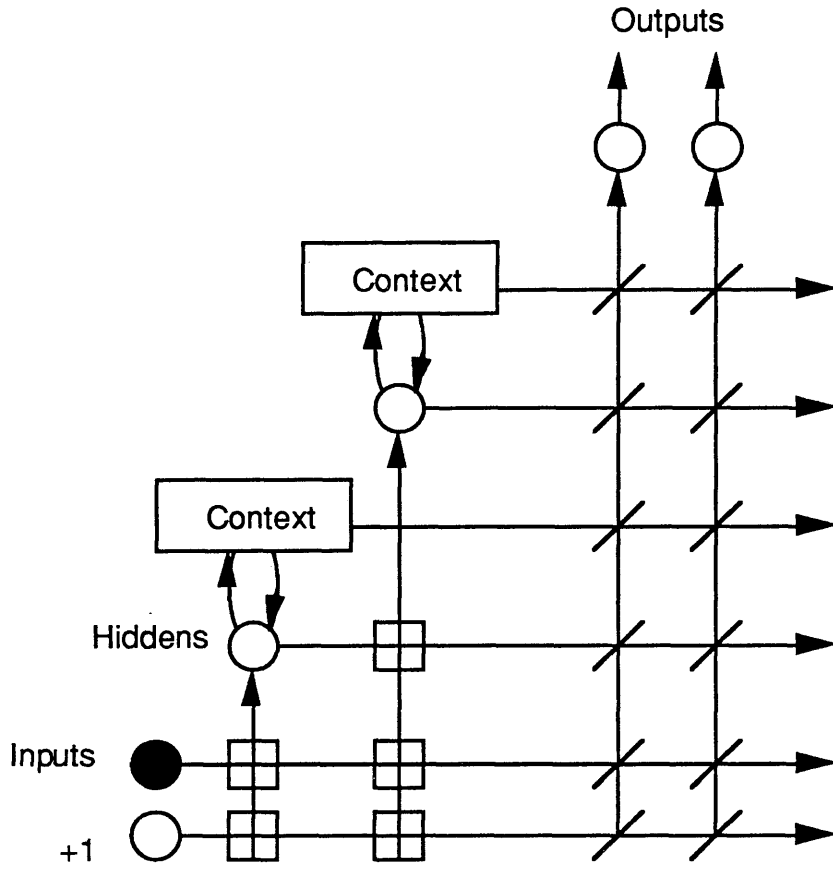


Figure 3.2. CSM's Recurrent Cascade-Correlation

Chapter 4
PROGRAM DESCRIPTION
AND
GENERAL COMMON METHODOLOGY

This chapter will describe the two programs implementing Fahlman's and CSM's network, the bugs that existed in the two programs, and the training method.

4.1 Program Description

In the two programs, there are three types of sigmoidal activation functions available. These are symmetric, asymmetric, and variable. The range of symmetric sigmoid is [-0.5, 0.5], of asymmetric sigmoid is [0.0, 1.0] and of variable sigmoid is [min, max] where min and max are specified by the users. Choice of the three functions depends on the training sets. The programs implementing the networks have two sets of parameters. One set is for each training phase: output and candidate (or input) training phase. For example, there are two learning rate coefficients called input η and output η . The parameters can be either internally or externally initialized. They are initialized internally by having statements in a procedure called "Initialize Global" and externally by having a network file with a file extension of ".net". To run the

program with internally initialized parameters, just the program name is needed.

With externally initialized parameters, the following command is entered:

```
<ProgName> <NetFileName> < InEpochs> < OutEpochs> <Nhiddens> <Ntrials>
```

where ProgName is the program name, NetFileName is the network file name without file extension, InEpochs and OutEpochs are number of epochs to train input and output weights, Nhiddens is number of new hidden units allowed and Ntrials is number of training times until learning is successful or fails. The initial seed value for the random function, which generates initial connection weights can be either 1 or the clock time. If the number of testing patterns is not specified then the training set is also used for testing. The training and testing set are initially stored in data files and read into arrays when the network is set up. Correlation values, partial derivatives of S, and partial derivatives of inputs or outputs V are also stored in arrays. A pool of candidate units is used at the candidate training phase. Each unit of the networks has a bias input permanently set at +1 with a trainable weight. Appendix A is detailed descriptions of the main modules of the two programs.¹

4.2 Verification of Correctness

Before the two programs simulating the recurrent cascade-correlation of Fahlman and CSM were trained, running tests were performed to verify their correctness. Collected results showed that they had bugs in the code. Those bugs needed to be corrected before actual training.

¹For more information, refer to the two programs.

4.2.1 Fahlman's Version

With Fahlman's version, inconsistency of the results were observed after training and testing the network with the same data set. After learning was successful, training and testing errors were not the same but they should have been. Program code was examined and the mistake was finally located in the testing phase. This inconsistency occurred because, before testing, the values of the context units were not reset. Statements have been added to correct the problem.

4.2.2 CSM's Version

The inconsistency that happened with Fahlman's program also occurred with CSM's program. The same correction was done. Another error during the candidate training phase was also detected. Instead of storing previous candidate values in an array for a pool of candidate units, they were stored into a variable. This overwrote the values of other candidate units. After correction, the program ran properly. However, for Morse code teaching, the network could not learn because this type of problem was made up of many small sequences, not just one continuous sequence. The program has been modified to handle Morse code training. At the start of each letter input pattern, the values of all context units and all hidden units need to be cleared.

4.3. Training Methodology

The learning algorithm used has an impact on the learning time and the size

of the network. Other factors that affect the learning time are the networks' parameters: learning coefficient η and maximum growth factor μ . Values of those parameters need to be determined before actual training. However, there is currently no systematic method to determine those parameters. The following subsections, 4.3.1 and 4.3.2, will describe the method for determining parametric values and the method for actual training.

4.3.1 Determination of Learning Parameters

Initially, the network starts with a parameter set. A range of a parameter is selected to train the network while keeping other parameters unchanged. The best parametric value in that range is chosen based on the average of training results: number of successful trainings, number of hidden units, and number of training epochs. This process is successively applied for each parameter with the values of previous selected parameters set. Marginal offset of 0.4 is used for deciding whether the networks' outputs are correct or not. For an output range of $[b, c]$, any value in the range $[b, b+0.4)$ is considered to be b , and any value in the range $(c-0.4, c]$ is considered to be c . Values in the range $[b+0.4, c-0.4]$ are not considered as correct during training. The network is trained 5 times for each parameter. The maximum number of hidden units allowed is 10 because, after a few trial runs with various parameter values, the number of hidden units for most successful learnings is around 5. For reason of comparison, the same initial seed of 1 is used for the random function to generate initial random weights of the networks. The average number of hidden units, the number of successful learnings (out of 5), and the average

number of training epochs are experimentally collected. The error measurement is based on the number of wrong outputs. Throughout this process of parametric determination, the training and testing set are the same. Two parameters are searched for in this way: learning rate and maximum growth factor. Since there are two training phases, there are four parameters to be determined. Those are input, output learning rate (η_{IN} , η_{OUT}), and input, output maximum growth factor (μ_{IN} , μ_{OUT}). Values of these four parameters are re-determined for each investigated problem. According to Scott E. Fahlman, a value of around 1.0 for η , a value of 1.75 for μ work well for a wide range of problems [6]. The ranges of η and μ used for the process are [0.2, 2.0] and [1.5, 2.5], respectively. Since values of the parameters are interdependent, with the method used, their values are different if their searching order is changed. So, this method only locally gives the best sets of the parametric values.

4.3.2 Actual Training Method

With the above learning parameters, the two networks were trained for four problems: sequential XOR, Morse code, learning distance and learning distance with noise. For more accurate training results and also to keep experimental time low, each training problem is trained for 10 trials with the maximum allowable number of hidden units set at 20. The number of wrong outputs of the network is used to determine whether a training is a success or a failure. A training is successful if there is no wrong output for the learning problem. CPU time in seconds is also collected for each training. The average number of hidden units, the number of successful learnings (out of 10), the

average numbers of training epochs, and the average running CPU time in seconds were also collected. For comparison, percent differences of average results between the two networks are calculated by the following equation:

$$\% \text{difference} = \frac{\text{CSM'sAverage} - \text{Fahlman'sAverage}}{\text{Fahlman'sAverage}} \times 100$$

Chapter 5

INVESTIGATED PROBLEMS

This chapter describes the investigated learning problems, training method, collected results, and comparison of the two networks' performances for each problem. Before each problem is described, particular terminology used needs to be mentioned. The two expressions, "average number of successful learnings" and "victories," are interchangeable for an entry in the result tables. Also, for these tables, "Previous parameters" indicates parameters whose values have been determined; "Hiddens" is the average number of hidden units added during training; "Epochs" is the average number of times that the training set was presented, and bold numbers are the chosen values of the parameters.

5.1 Sequential XOR

Sequential XOR problem is derived from the logical exclusive-OR or XOR function. The XOR function has 2 binary inputs. Its output is a one if either input is one but not both. Otherwise, its output is a zero. The sequential XOR problem inputs are sequences of the two XOR inputs and the resulted output. Its output is the input sequence shifted left one bit. An example of training set is:

```

Inputs  0 1 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 0 ...
Outputs 1 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 0 .....

```

5.1.1 Method

A program for generating random sequential XOR data was written. Each data set has 3,000 patterns which are pairs of one input and one output. After the second random bit of inputs is fed to the network, a correct XOR output should be produced. A training is successful if all of those corresponding bits are generated. Therefore, we only care about the second bit of the output sequence and every third bit after it. Other output bits are ignored. Since each training pair has only one input and output, the network initially has two units: network's output unit and input terminal. The activation of every unit is an asymmetric sigmoidal function whose range is [0,1]. Initially, the network has $\eta_{IN} = 1.0$, $\eta_{OUT} = 1.0$, $\mu_{IN} = 2.00$ and $\mu_{OUT} = 2.00$.¹ Before actual trainings are performed, locally best values of parameters are determined. The method of parametric determination is described in section 4.2. There are two different data sets used for 2 actual trainings. For the first training, if set 1 is used for training, then set 2 is used for testing; reverse use of data sets is for the second training.

5.1.2 Data

The following tables of obtained data are listed sequentially in the process of determination and actual training.

¹ These initial values of the mentioned parameters are used for all problems.

5.1.2.a Fahlman's Version

5.1.2.a.i Parametric Determination

Tables 5.1.1 to 5.1.4 are listed in order of the process of parametric determination with Fahlman's version for sequential XOR problem. From table 5.1.1, Fahlman's network has higher performance with input learning coefficient η of 1.8 among other values of η , because it has more successful learnings (2 out of 5), fewer hidden units, and fewer training epochs. This value of η is used in all subsequent parametric determinations. This method of chosen parametric value is similar for all other parameters.

Table 5.1.1. Determination of input learning coefficient η for sequential XOR with Fahlman's version

Previous Parameters	none									
Input η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	0	2	1	0	1	0	0	0	2	0
Hiddens	10	7.8	9.4	10	9.4	10	10	10	7.6	10
Epochs	639	588	643	634	612	647	643	638	516	641

Table 5.1.2. Determination of output learning coefficient η for sequential XOR with Fahlman's version

Previous Parameters	$\eta_{IN} = 1.8$									
Output η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	0	0	1	2	2	1	0	0	1	0
Hiddens	10	10	9.4	8.6	7.6	8.8	10	10	8.6	10
Epochs	680	671	494	597	516	542	665	627	600	663

Table 5.1.3. Determination of output growing rate μ for sequential XOR with Fahlman's version

Previous Parameters	$\eta_{IN} = 1.8, \eta_{OUT} = 1.0$				
Output μ	1.50	1.75	2.00	2.25	2.50
Victories	0	0	2	0	2
Hiddens	10	10	7.6	10	8
Epochs	680	661	516	701	532

Table 5.1.4. Determination of input growing rate μ for sequential XOR with Fahlman's version

Previous Parameters	$\eta_{IN} = 1.8, \eta_{OUT} = 1.0, \mu_{OUT} = 2.00$				
	1.50	1.75	2.00	2.25	2.50
Input μ	1.50	1.75	2.00	2.25	2.50
Victories	2	3	2	0	0
Hiddens	7.8	6.2	7.6	10	10
Epochs	598	425	516	579	643

5.1.2.a.ii Actual Training

With the defined parametric values, $\eta_{IN} = 1.8$, $\eta_{OUT} = 1.0$, $\mu_{OUT} = 2.0$, and $\mu_{IN} = 1.75$, actual trainings are performed. Table 5.1.5 lists the results.

Table 5.1.5. Actual training results of sequential XOR with Fahlman's version

Parameters	$\eta_{IN} = 1.80, \eta_{OUT} = 1.00, \mu_{IN} = 2.00, \mu_{OUT} = 1.75$				
Training	Starting Seed	Victories	Hiddens	Epochs	CPU Time
1	1	3	15.2	1111	1549.49
	random	3	15.4	1133	1596.55
2	1	2	16.7	1204	1681.42
	random	4	13.5	986	1329.48
Average		3	15.2	1109	1539.24

5.1.2.b CSM's Version

5.1.2.b.i Parametric Determination

Table 5.1.6 to 5.1.9 are data obtained from the process of parametric determination in sequential order with CSM's version.

Table 5.1.6. Determination of input learning coefficient for sequential XOR with CSM's version

Previous Parameters	none									
Input η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	0	1	0	0	3	0	1	3	1	2
Hiddens	10	9.2	10	10	6.8	10	8.6	5.8	8.4	7.2
Epochs	831	760	810	839	556	790	674	479	679	571

Table 5.1.7. Determination of output learning coefficient η for sequential XOR with CSM's version

Previous Parameters	$\eta_{IN} = 1.6$									
Output η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	1	0	0	1	3	3	2	4	3	4
Hiddens	9	10	10	8.4	5.8	6.4	7.8	4.2	6.2	4
Epochs	731	804	819	700	479	547	642	344	473	299

Table 5.1.8. Determination of output growing rate μ for sequential XOR with CSM's version

Previous Parameters	$\eta_{IN} = 1.6, \eta_{OUT} = 2.0$				
Output μ	1.25	1.50	1.75	2.00	2.25
Victories	1	1	1	4	1
Hiddens	8.4	9.2	8.6	4	8.6
Epochs	696	770	735	299	732

Table 5.1.9. Determination of input growing rate for sequential XOR with CSM's version

Previous Parameters	$\eta_{IN} = 1.6, \eta_{OUT} = 2.0, \mu_{OUT} = 2.00$				
Input μ	1.25	1.50	1.75	2.00	2.25
Victories	1	1	1	4	1
Hiddens	8.4	9.2	8.6	4	8.6
Epochs	696	770	735	299	732

5.1.2.b.ii Actual Training

With the determined parametric values, $\eta_{IN} = 1.6$, $\eta_{OUT} = 2.0$, $\mu_{OUT} = 2.0$, $\mu_{IN} = 2.0$, CSM's network was trained. Table 5.1.10 is the result.

Table 5.1.10. Actual training results of sequential XOR with CSM's version

Parameters	$\eta_{IN} = 1.60, \eta_{OUT} = 2.00, \mu_{IN} = 2.00, \mu_{OUT} = 2.00$				
Training	Starting Seed	Victories	Hiddens	Epochs	CPU Time
1	1	8	7.0	582	1143.33
	Random	8	7.1	593	1122.20
2	1	7	8.7	696	1515.05
	Random	5	12.1	1033	2526.42
Average		7	8.73	726	1576.50

5.1.3 Comparison

Table 5.1.11 is a summary of actual trainings of the two networks for sequential XOR. Average results and percent differences between Fahlman's and CSM's network are listed. The percent differences show that number of hidden units and number of training epochs with CSM's version reduces by

42.63% and 34.54%, respectively. Number of successful learnings of CSM's version increases by 133.33%. In other words, the size of CSM's network is smaller by 42.63%, the number of times gone over the entire training sets by CSM's version is 34.54% percent less, and CSM's version has 133.33% more successful learning. However, CSM's version takes more CPU time to learn than Fahlman's version, 2.42%. This increase in CPU time of CSM's version might result from the computing time because CSM's network has more connections than Fahlman's. Since percent increase in CPU time for CSM's version is small, other improvements of CSM's version dominates. So, CSM's version has higher performance than Fahlman's version for sequential XOR problem.

Table 5.1.11. Comparing results of sequential XOR

	Victories	Hiddens	Epochs	CPU Time
Fahlman's	3	15.2	1109	1539.24
CSM's	7	8.73	726	1576.50
%difference	+133.33%	-42.63%	-34.54%	+2.42%

To determine the training parameters, a training set of 26 letters in alphabetical order is used. Two actual trainings are performed. The first one uses the 26 letters in alphabetic order as the training set and in arbitrary order as the testing set. The second training is opposite. For this Morse code problem, one more error measurement is collected, the summation of squared errors of the networks' outputs over one training epoch.

5.2.2 Data

5.2.2.a Falman's Network

5.2.2.a.i Parametric Determination

Values of the learning parameters were determined using the method described. The results are table 5.2.1 to 5.2.4.

Table 5.2.1. Determination of input learning coefficient for Morse code with Fahlman's version.

Previous Parameters	none									
Input η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	2	1	1	2	1	1	3	1	1	2
Hiddens	9.6	10.	10.	9.8	9.8	9.8	9.8	9.8	10.	9.6
Epochs	1168	1050	1084	1039	1042	1173	1131	1077	1106	1094

Table 5.2.2. Determination of output learning coefficient η for Morse code with Fahlman's version.

Previous Parameters	$\eta_{IN} = 1.4$									
Output η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	1	1	1	2	3	0	3	5	3	2
Hiddens	10.	10.	9.6	9.8	9.8	10.	9.8	8.8	9.4	9.8
Epochs	1195	1120	1119	1167	1131	1019	1169	1176	1183	1166

Table 5.2.3. Determination of output growing rate μ for Morse code with Fahlman's version

Previous Parameters	$\eta_{IN} = 1.4, \eta_{OUT} = 1.6$				
Output μ	1.50	1.75	2.00	2.25	2.50
Victories	0	3	5	2	1
Hiddens	10.	10.	8.8	10.	10.
Epochs	993	1296	1176	1162	1004

Table 5.2.4. Determination of input growing rate μ for Morse code with Fahman's version

Previous Parameters	$\eta_{IN} = 1.4, \eta_{OUT} = 1.6, \mu_{OUT} = 2.00$				
Input μ	1.50	1.75	2.00	2.25	2.50
Victories	2	3	5	4	2
Hiddens	9.6	9.6	8.8	9.4	9.8
Epochs	1283	1261	1176	1236	1218

5.2.2.a.ii Actual training

Table 5.2.5 is the result of Morse code training with Fahlman's network.

Table 5.2.5. Actual training results of Morse code with Fahlman's version

Parameters	$\eta_{IN} = 1.4, \eta_{OUT} = 1.6, \mu_{IN} = 2.00, \mu_{OUT} = 2.00$					
Training	Init. Seed	Victories	Hiddens	Epochs	Error	CPU Time
1	1	10	10.4	1310	0.7918	960.19
	Random	10	10.8	1345	0.6983	993.31
2	1	10	9.70	1222	0.7482	787.49
	Random	10	10.0	1287	0.7729	837.00
Average		10	10.2	1291	0.7529	894.50

5.2.2.b CSM's Network

5.2.2.b.i Parametric Determination

Table 5.2.6 to 5.2.9 list the results in the process of parametric determination for Morse code problem with CSM's network.

Table 5.2.6. Determination of input learning coefficient η for Morse code with CSM's version

Previous Parameters	none									
Input η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	5	5	5	5	5	5	5	5	5	5
Hiddens	7.4	7.2	7.2	7.6	7.4	6.8	7.4	7.2	7.2	7.0
Epochs	945	941	946	969	960	854	943	929	909	859

Table 5.2.7. Determination of output learning coefficient η for Morse code with CSM's version

Previous Parameters	$\eta_{IN} = 1.2$									
Output η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	5	5	5	5	5	5	5	5	5	5
Hiddens	7.0	6.8	6.6	7.2	6.8	6.6	7.2	7.2	6.4	7.6
Epochs	927	870	826	880	854	819	913	923	796	944

Table 5.2.8. Determination of output growing rate μ for Morse code with CSM's version

Previous Parameters	$\eta_{IN} = 1.2, \eta_{OUT} = 1.8$				
Output μ	1.50	1.75	2.00	2.25	2.50
Victories	5	5	5	5	5
Hiddens	6.8	7.4	6.4	6.6	7.2
Epochs	879	950	796	790	917

Table 5.2.9. Determination of input learning rate μ for Morse code with CSM's version

Previous Parameters	$\eta_{IN} = 1.2, \eta_{OUT} = 1.8, \mu_{OUT} = 2.00$				
Input μ	1.50	1.75	2.00	2.25	2.50
Victories	5	5	5	5	5
Hiddens	6.6	7.0	6.4	7.0	7.2
Epochs	886	930	796	890	883

5.2.2.b.ii Actual Training

Actual training for Morse code with CSM's network was performed. Table 5.2.10 is the result.

Table 5.2.10. Actual training results of Morse code with CSM's version.

Parameters	$\eta_{IN} = 1.6, \eta_{OUT} = 2.0, \mu_{IN} = 2.00, \mu_{OUT} = 2.00$					
Training	Init. Seed	Victories	Hiddens	Epochs	Error	CPU Time
1	1	10	7.0	914	0.6001	645.79
	Random	9	7.9	1035	1.0626	781.69
2	1	10	6.5	904	0.5431	626.91
	Random	10	7.5	1048	0.6203	728.82
Average		9.8	7.2	975	0.7065	695.80

5.2.3 Comparison

Table 5.2.11 is actual training results for Morse code with the two networks. Because their percent differences are small, a conclusion about their performance can not be derived.

Table 5.2.11. Actual training results of Morse code.

	Victories	Hiddens	Epochs	Error	CPU Time
Fahlman's	10	10.2	1291	0.7528	894.50
CSM's	9.8	7.20	975	0.7065	695.80
%difference	-2.00%	-29.41%	-24.48%	-6.15%	-22.21%

5.3 Learning Distance

How far a network can relate backward in a training set to produce its outputs is called its "learning distance." The goal of this training is to find out which network is more attractive for this learning distance problem. Training input of this problem is a sequence of identical sub-sequences with a random numbers of zeros preceding and following the sub-sequences. Each sub-sequence starts with a negative one, followed with a fixed number of zeros (distance), and ends with a positive one. Network's outputs are the input patterns shifting left one bit. The network has 1 input and 1 output. For this problem, we only care about outputs of positive one and outputs of zero of the learning distance. Outputs of random zero and negative one are ignored. An example of training set for this learning distance follows:

Inputs 0 0 0 -1 0 0 0 1 0 0 -1 0 0 0 1 0 0 0 0 -1 0 0 0 1 0 0 ...

Outputs 0 0 -1 0 0 0 1 0 0 -1 0 0 0 1 0 0 0 0 -1 0 0 0 1 0 0

5.3.1 Method

The two networks are trained for four different distances. They are 20, 40, 80, and 100 zeros. A set of learning parametric values for each network, which is locally the best over the four distances, needs to be determined before actual training. To obtain this set, the learning parametric values for a distance among the four, that produces the worst performance (fewest successful trainings, most hidden units, and most training epochs), is searched. This distance is obtained by training the networks with the same initial set of values for the four distances. The process of parametric determination is applied to this distance to obtain the locally best values of the learning parameters. Next, actual training is performed with the determined values. The units' activation function used is a variable sigmoidal function whose range is $[-1.0, 1.0]$ because negative and positive one are lower and upper bound of the networks' outputs.

5.3.2 Data

5.3.2.a Fahlman's Network

5.3.2.a.i Parametric Determination

Table 5.3.1 is the training results for the four different learning distances with Fahlman's version. The distance of 100 zeros has the worst performance. However, there is no successful learning. There is a possibility that the net can not learn at this distance. Instead of using this distance, the distance of 80 zeros, which produces the second worst performance, is used for the process of

parametric determination. Tables 5.3.2 to 5.3.5 are results of the determination with a distance of 80 zeros.

Table 5.3.1. Arbitrary training for different distances with Fahlman's version.

Number of Zeros	20	40	80	100
Victories	5	4	1	0
Hiddens	3.2	8.4	10.	10.
Epochs	222	576	769	702

Table 5.3.2. Determination of input learning coefficient η for learning distance of 80 zeros with Fahlman's version

Previous Parameters	none									
	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Input η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	3	1	1	0	1	0	1	0	1	1
Hiddens	8.6	9.6	9.4	10.	10.	10.	9.8	10.	10.	9.2
Epochs	704	762	754	719	769	739	733	735	724	646

Table 5.3.3. Determination of output learning coefficient for learning distance of 80 zeros with Fahlman's version

Previous Parameters	$\eta_{IN} = 0.2$									
Output η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	1	1	0	2	3	2	0	1	2	1
Hiddens	9.4	9.6	10.	9.0	8.6	10.	10.	9.0	9.0	9.8
Epochs	752	783	772	734	704	766	854	611	697	727

Table 5.3.4. Determination of output growing rate μ for learning distance of 80 zeros with Fahlman's version

Previous Parameters	$\eta_{IN} = 0.2, \eta_{OUT} = 1.0$				
Output μ	1.50	1.75	2.00	2.25	2.50
Victories	0	1	3	0	3
Hiddens	10.	9.6	8.6	10.	8.0
Epochs	728	756	704	744	617

Table 5.3.5. Determination of input growing rate μ for learning distance of 80 zeros with Fahlman's version.

Previous Parameters	$\eta_{IN} = 0.2, \eta_{OUT} = 1.0, \mu_{OUT} = 2.50$				
Input μ	1.50	1.75	2.00	2.25	2.50
Victories	2	1	3	1	2
Hiddens	8.2	9.4	8.0	9.6	9.2
Epochs	761	766	617	679	715

5.3.2.a.ii Actual Training

With values of the parameters defined, Fahlman's network is trained with data sets of the four different distances.

Table 5.3.6. Actual training results of learning distance with Fahlman's version.

Number of Zeros	Victories	Hiddens	Epochs	CPU Time
20	9	7.4	558	760.49
40	7	11.6	878	1254.89
80	4	15.0	1181	1743.39
100	1	19.3	1532	2372.44
Average	5.25	13.3	1037	1532.80

5.3.2.b CSM's Network

5.3.2.b.i Parametric Determination

Table 5.3.7 shows the results of arbitrary trainings for the four different distances with CSM's network. Similarly, as discussed in section 5.3.2.a.i, a distance of 40 zeros is used for parametric determination. Table 5.3.8 to 5.3.11 are results from the process of parametric determination with the learning distance of 40 zeros.

Table 5.3.7. Arbitrary training for different distances with CSM's version.

Number of Zeros	20	40	80	100
Victories	3	1	1	0
Hiddens	6.6	10.	9.0	10.
Epochs	528	862	848	939

Table 5.3.8. Determination of input learning coefficient η for learning distance of 40 zeros with CSM's version.

Previous Parameters	none									
Input η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	1	1	1	1	1	0	1	1	1	1
Hiddens	8.6	9.8	9.0	8.6	10.	10.	10.	9.0	9.2	10.
Epochs	705	909	832	749	862	850	742	775	821	920

Table 5.3.9. Determination of output learning coefficient η for learning distance of 40 zeros with CSM's version.

Previous Parameters	$\eta_{IN} = 0.2$									
Output η	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Victories	0	0	1	0	1	0	0	1	0	1
Hiddens	10.	10.	8.8	10.	8.6	10.	10.	8.6	10.	9.0
Epochs	934	879	859	874	705	878	901	729	927	792

Table 5.3.10. Determination of output growing rate μ for learning distance of 40 zeros with CSM's version.

Previous Parameters	$\eta_{IN} = 0.2, \eta_{OUT} = 1.0$				
Output μ	1.50	1.75	2.00	2.25	2.50
Victories	1	1	1	1	2
Hiddens	9.0	9.0	8.6	8.6	7.6
Epochs	807	840	705	722	723

Table 5.3.11. Determination of input growing rate μ for learning distance of 40 zeros with CSM's version.

Previous Parameters	$\eta_{IN} = 0.2, \eta_{OUT} = 1.0, \mu_{OUT} = 2.50$				
Input μ	1.50	1.75	2.00	2.25	2.50
Victories	0	3	2	0	2
Hiddens	10.	8.6	7.6	10.	8.6
Epochs	992	750	623	873	718

5.3.2.b.ii Actual Training

With values of the parameters obtained, actual trainings are performed with different distances. Table 5.3.12 is the results.

Table 5.3.12. Actual training results of learning distance with CSM's version.

Number of Zeros	Victories	Hiddens	Epochs	CPU Time
20	6	10.4	1133	2167.16
40	5	14.2	1505	2975.25
80	0	20.0	2380	4592.86
100	2	17.3	1960	3500.15
Average	3.25	15.5	1744	3308.85

5.3.3 Comparison

Table 5.3.13 shows the average results of distance training for both versions. The percent differences show that Fahlman's version has higher performance than CSM's because CSM's version produces lower number of successful learnings, larger number of hidden units, larger number of training epochs, and greater CPU time.

Table 5.3.13. Comparing results for the learning distance problem.

	Victories	Hiddens	Epochs	CPU Time
Fahlman's	5.25	13.3	1037	1532.80
CSM's	3.25	15.5	1744	3308.85
%difference	-38.09%	+16.54%	+68.18%	+115.87%

5.4 Learning Distance with Noise

The goal for this training is to examine performance of the two networks when there is noise in the training and testing set. Noise can occur at any input of the data sets. The probability of having noise at an input is fixed during the entire training. If noise occurs at an input value, this input value is changed to one of the other two input values. For example, if noise happens at an input value of -1, this input value is changed to either 0 or 1. To determine which value that it changes to, a random function that produces random numbers of range [0, 99] is used. The lower value of the two remained values is chosen if the randomly generated number is less than 50. Otherwise, the larger value is chosen.

5.4.1 Method

The two network programs are modified so that new training sets with noise would be produced. The maximum noise that occurred at an input is set at 1.0%. Because the boundaries of the output range is -1 and +1, the range of the

activation sigmoidal function is $[-1, +1]$. An actual output is considered correct if the absolute difference between the actual output and corresponding desired output is less than the threshold value of 0.4. The same set of learning parametric values as in the learning distance without noise is used. The two architectures are trained with five different distances: 10, 20, 40, 80, and 100 zeros and for two different approaches. One approach is keeping the training set unchanged for the entire training. The other is having new training set at the start of every output and candidate training phase.

5.4.2 Data for Learning Distance with Noise

Table 5.4.1 and 5.4.2 are the training results of learning distance with 1% noise with Fahlman's and CSM's version for the unchanged training set approach.

Table 5.4.1. Actual training results for learning distance with 1.0% noise with Fahlman's version for the unchanged training set approach.

Number of Zeros	Victories	Hiddens	Epochs	CPU Time
10	7	8.8	654	1060.67
20	7	9.2	670	1306.37
40	7	11.0	726	1380.25
80	2	17.8	1413	3072.49
100	2	18.8	1453	3176.59
Average	5	13.1	983	1999.27

Table 5.4.2. Actual training results for learning distance with 1.0% noise with CSM's version for the unchanged training set approach.

Number of Zeros	Victories	Hiddens	Epochs	CPU Time
10	8	9.9	884	2576.76
20	6	10.7	1070	2954.90
40	5	12.8	1389	3878.15
80	0	20.0	2320	6974.11
100	0	20.0	2410	7301.57
Average	3.8	14.7	1615	4737.10

Table 5.4.3 and 5.4.4 show the training results with 1.0% noise with Fahlman's and CSM's version for the changed training set approach.

Table 5.4.3. Actual training results for learning distance with 1.0% noise with Fahlman's version for the changed training set approach.

Number of Zeros	Victories	Hiddens	Epochs	CPU Time
10	8	11.1	2157	5219.69
20	3	15.9	2398	7048.29
40	1	18.6	2638	8216.35
80	0	20.0	3022	9597.71
100	0	20.0	2848	9013.48
Average	2.4	17.1	2613	7819.10

Table 5.4.4. Actual training results for learning distance with 1.0% noise with CSM's version for the changed training set approach.

Number of Zeros	Victories	Hiddens	Epochs	CPU Time
10	8	8.2	1615	3620.07
20	2	16.8	3273	8327.17
40	1	18.4	2734	6293.79
80	0	20.0	2775	6997.82
100	0	20.0	2849	9385.82
Average	2.2	16.7	2649	6924.93

Table 5.4.5 and 5.4.6 show the percentages of correct bits for distance bits of zero and for bits of one that mark the ends of learning distance sub-sequences. The percentages are calculated for an entire test set by the following equations:

$$\% \text{ correct bit one} = \frac{\text{Number of correct bit ones}}{\text{Number of bit ones}} 100\%$$

$$\% \text{ correct bit zero} = \frac{\text{Number of correct bit zeros}}{\text{Number of bit zeros}} 100\%$$

Their averages are also calculated.

Table 5.4.5. Percentages of correct bits for the unchanged training set approach.

Number of Zeros	Fahlman's version		CSM's version	
	% correct of ones	%correct of zeros	% correct of ones	%correct of zeros
10	98.65%	99.89%	99.85%	99.96%
20	96.80%	99.90%	75.21%	55.17%
40	75.43%	99.96%	61.37%	50.12%
80	37.56%	99.92%	52.12%	0%
100	30.00%	90.00%	70.00%	6.00%
Average	67.69%	97.93%	71.71%	42.25%

Table 5.4.6. Percentages of correct bits for the changed training set approach.

Number of Zeros	Fahlman's version		CSM's version	
	% correct of ones	%correct of zeros	% correct of ones	%correct of zeros
10	77.00%	89.34%	84.12%	99.83%
20	50.05%	39.99%	29.97%	69.71%
40	71.13%	19.78%	50.50%	9.55%
80	19.32%	9.12%	58.99%	0%
100	60.20%	9.95%	16.57%	10.00%
Average	55.54%	33.64%	48.04%	37.82%

5.4.3 Comparison

Table 5.4.7 and 5.4.8 are the averages of training results with both networks and their differences for the two approaches. The percent differences in these tables along with the information in tables 5.4.5 and 5.4.6 show that Fahlman's network has better performance than CSM's for the unchanged training set approach. However, for the changed training set approach, a conclusion about their performance can not be derived from their percent differences because they are small.

Table 5.4.7. Comparing results for the learning distance with 1.0% noise for the unchanged training set approach.

	Victories	Hiddens	Epochs	CPU Time
Fahlman's	5.0	13.1	983	1999.27
CSM's	3.8	14.7	1615	4737.10
%difference	-24.00%	+12.21%	+64.29%	+136.94%

Table 5.4.8. Comparing results for the learning distance with 1.0% noise for the changed training set approach.

	Victories	Hiddens	Epochs	CPU Time
Fahlman's	2.4	17.1	2613	7819.10
CSM's	2.2	16.7	2649	6924.93
%difference	-8.33%	-2.34%	+1.38%	-11.44%

Chapter 6

CONCLUSION, INSIGHT, AND RECOMMENDATION

6.1 Conclusion

As mentioned before, determination of a network's performance is based on its representation, size, and learning time. Another way of saying this is that its performance is based on its learning domain, number of hidden units in the net, and number of training epochs (or CPU time). Therefore, a more powerful network should have a broader learning domain, fewer number of hidden units, and fewer training cycles. Experimental resulting data show that CSM's Recurrent Cascade-Correlation has improved performance over Fahlman's for the sequential XOR. However, its performance is lower than Fahlman's version for the problem of learning distance with noise when the training set is unchanged and the problem of learning distance without noise. For the Morse code problem and the learning distance with noise when the training set is changed, there is no conclusion about their performance because their percent differences are small.

6.2 Insight

CSM's version has better performance than Fahlman's for the sequential XOR. This improved performance originates from the difference in structure.

The only structural difference is that the values of context units in CSM's version are propagated to all hidden units added later and to the output units, while those values are only fed to the corresponding hidden units in Fahlman's. This architectural difference might be the source that makes their performance different. In CSM's network, output of a hidden layer's unit depends on the previous and current outputs of all units on previous layers. For Fahlman's network, output of a hidden layer's unit depends on current output of all units on its previous layers and its context value. Also, in CSM's network, outputs of the network depend on all hidden units and all context units. This does not occur in Fahlman's network. However, this structural difference makes the performance of CSM's network lower than Fahlman's for the problem of learning distance with noise when the training set is unchanged and for the problem of learning distance without noise.

6.3 Recommendation

Throughout this research, determination of best values of the learning parameters had to be done. As mentioned before, there is not currently a systematic method to determine the networks' learning parameters. However, if a computer program is written for this purpose and accessible by everyone, experimental time will be greatly reduced, and training results of researchers would be more consistent and comparable (I believe that training a network to determine the parameters takes more time than searching for results which have been obtained by other researchers). The second recommendation is to do a mathematical analysis of CSM's and Fahlman's version. A mathematical result

should confirm the experimental conclusion. Finally, we need to find out why the structural difference makes CSM's performance better in some problems but worse in others.

REFERENCES CITED

1. Warren S. McCulloch and Walter Pitts 1943. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics 5 : 115-133.
2. F. Rosenblatt 1985. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological Review 65: 386-408.
3. Marvin Minsky and Seymour Papert 1969. Perceptron. Cambridge, MA: MIT Press: 1-20 & 73.
4. Phillip D. Wasserman, 1989. Neural computing theory and practice. New York, NY: Van Nostran Reinhold: 19, 33-34.
5. David E. Rumelhart, G. E. Hinton and R. J. Williams 1987. "Learning internal representations by error propagation." Parallel Distributed Processing, Vol. 1. Cambridge, MA: MIT Press: 318-361.
6. David E. Rumelhart, G. E. Hinton and R. J. Williams 1986. "Learning representations by back propagating errors." Nature 323: 533-536.
7. Scott E. Fahlman, May 1988. An empirical study of learning speed in back-propagation networks. CMU-CS-88-162.

8. Terrence J. Sejnowski and Charles R. Rosenberg 1986. "NETtalk: a parallel network that learns to read aloud." The Johns Hopkins University Electrical Engineering and Computer Science Technical Report JHU/EECS-86/01.
9. Michael I. Jordan, May 1986. Serial order: a parallel distributed processing approach. Report of Institute for Cognitive Science. ICS 8604.
10. Jeffrey L. Elman, June 1989. Finding structure in time. Report of Department of Cognitive Science, University of California, San Diego.
11. Ronald J. Williams and David Zipser 1989. A learning algorithm for continually running fully recurrent neural networks. Neural Computation, Vol. 1, No. 2: 270-279.
12. Scott E. Fahlman and Christian Liebre, Feb. 1990. "The cascade-correlation learning algorithm." Report of School of Computer Science, Carnegie Mellon University. CMU-CS-90-100.
13. Scott E. Fahlman, May 1991. "The recurrent cascade-correlation architecture." Report of School of Computer Science, Carnegie Mellon University. CMU-CS-91-100.
14. Steele, John P.H., Aaron Gordon, and Hong Chang, November 1992. Identifying Objects in Signals Using Cascaded Recurrent Cascade Correlation Neural Networks. Artificial Neural Networks in Engineering. St. Louis, MO.

SELECTED BIBLIOGRAPHY

John Hertz, Anders Krogh and Richard G. Parlmer 1991. Introduction to the theory of neural computation. Redwood, CA: Addison-Wesley.

Phillip D. Wasserman 1989. Neural computing theory and practice. New York, NY: Van Nostran Reinhold.

Tarun Khanna 1990. Foundation of neural networks. The United States: Addison-Wesley.

APPENDIX A

This appendix describes the main modules of the programs for both Fahlman's and CSM's network.

Overview of the learning algorithm

Repeat

 Initialize network

 Repeat

Train output weights

 If (learning is not successful AND total number of units in network less than maximum number of units allowed)

 then *train input weights*

 install new hidden unit

 Until (learning is successful OR total number of units in network equals to maximum number of units allowed)

 Test network with the testing set.

Until a certain training trials.

Output weight training

Repeat

Train output weight for one epoch

Until (learning is successful OR number of training epoch equals to maximum input training epochs allowed OR error reduction is not significant after a certain number of training epochs)

Input weight training

Compute and adjust correlation

Repeat

Train input weights for one epoch

Until (learning is successful OR number of training epochs equals to maximum number of training epochs OR error reduction is not significant after a certain number of training epoch)

One-epoch output weight training

Repeat

Propagate signals to compute outputs

Compute errors

Until all training patterns have been processed

If error is not acceptable then update output weights using the quickprop

One-epoch input weight training

Repeat

 Compute slopes

 Until all training patterns have been processed

 Update input weights

 Adjust correlation.