

INVESTIGATION OF SEQUENTIAL
SELF-ORGANIZING MAPS

By

ROGER LEE BOYDSTUN

Bachelor of Science

Oklahoma State University

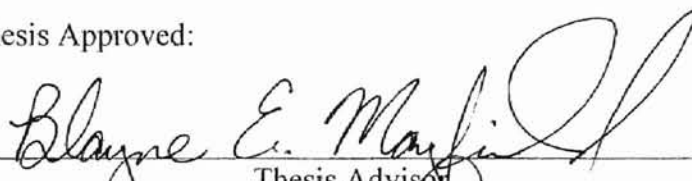
Stillwater, Oklahoma

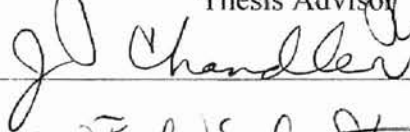
1987


Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1997

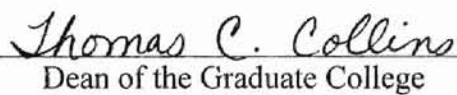
INVESTIGATION OF SEQUENTIAL
SELF-ORGANIZING MAPS

Thesis Approved:


Thesis Advisor






Dean of the Graduate College

Finally, I would also like to recognize the following: Nalini Hosur, Randy Donahoo, Konnie Gindrup, Lynn Wood, Pat Kennedy, Paxton Robey, Jonna Bostian. Bill Gates, and the Mitsubishi Corporation for the fun curves.

iii

TABLE OF CONTENTS

I. INTRODUCTION	1
Artificial Intelligence, a Perspective	1
<i>Useful Artificial Intelligence</i>	2
<i>Origins of Neural Research</i>	3
<i>Neural Modeling</i>	3
<i>Self Organizing Maps (SOMs)</i>	4
<i>Problems with SOMs</i>	6
II. LITERATURE REVIEW.....	8
Biological Neural Networks.....	8
Self-Organizing Maps (SOM).....	12
<i>ANN Architecture</i>	13
<i>Self-Organizing Map Architecture</i>	14
<i>Input and Output Layers</i>	15
<i>Weights</i>	15
<i>Activation Function</i>	16
<i>Training</i>	17
<i>Topology-Preserving Mapping</i>	18
<i>Lateral Feedback</i>	18
<i>Neighborhoods</i>	20
<i>Learning Rate</i>	22
<i>Weight Adjustment</i>	22
Sequential Processing Using Self-Organizing Maps.....	23
<i>Input Sequence Averaging</i>	24
<i>Response Integration Model</i>	25
<i>Pattern Concatenation Model</i>	26
<i>Trace Feature Maps</i>	27
III. METHODS.....	29
Introduction	29
<i>SeqSOM Architecture</i>	30
<i>Example of Input Bundle Formation</i>	32
<i>The Sequential Self-Organizing Map Algorithm</i>	33
IV. IMPLEMENTATION AND TESTING.....	35
Algorithm Implementation.....	35
<i>Development Environment</i>	36

<i>Testing Environment and Platform</i>	36
<i>Test Data Domain</i>	37
<i>Selection of Test FSAs</i>	37
<i>Generation of Test Data</i>	39
<i>Training a SeqSOM Network</i>	42
<i>Building an FSA from a Trained SeqSOM Network</i>	44
<i>Comparison of FSAs</i>	46
<i>Testing For Equivalence</i>	48
<i>Determining Equivalency of SeqSOM and Original FSA</i>	50
<i>Discussion of the Experimental Results</i>	51
<i>Failures to produce an Equivalent FSA</i>	52
<i>SeqSOM as Compared to Simple Recurrent Network</i>	54
V. CONCLUSIONS AND RECOMMENDATIONS	56
<i>Observations</i>	56
<i>Future Work</i>	57
BIBLIOGRAPHY	60
APPENDIXES	63
APPENDIX A <i>PROGRAM PARAMETERS</i>	64
APPENDIX B <i>STRING GENERATOR PROGRAM</i>	66
APPENDIX C <i>FINITE STATE AUTOMATA USED FOR TESTING THE SEQSOM</i> <i>APPLICATION</i>	68

LIST OF TABLES

Table 1. Results for SeqSOM.....	51
----------------------------------	----

LIST OF FIGURES

Figure 1. Biological Neuron.....	8
Figure 2. Six layers of the neocortex.....	11
Figure 3. Illustration of layered organization of an Artificial Neural Networks.....	13
Figure 4. "Mexican Hat" function illustrating positive and negative lateral interaction of neurons as a function of distance from the point of excitation.....	19
Figure 5. Step function illustrating the neighborhood region and neighborhood radius as related to the lateral distance of neurodes.....	21
Figure 6. Architecture for the Response Integration Model.....	25
Figure 7. Architecture of the Pattern Concatenation Neural Network.....	26
Figure 8. Trace Feature Map Architecture.....	27
Figure 9. Illustration of the Self-Organizing Map Architecture.....	31
Figure 10. Algorithm of Kohonen's Self-Organizing Map.....	32
Figure 11. Illustration of the SeqSOM Architecture.....	33
Figure 12. Sequential Self-Organizing Map Algorithm.....	34
Figure 13. The FSA state diagram used for the Ghosh and Karamcheti research.....	38
Figure 14. Illustrates an FSA (from Figure 13) transformed to a set of Prolog lists statements which are used as input for FSA string generator program.	40
Figure 15. Partial list of strings up to ten characters in length.....	41
Figure 16. Bit vector assignment for the FSA.....	42
Figure 17. Sample of the encoded vectors for Figure 15.....	43

Figure 18. Sample output from SeqSOM using the training set as input.	45
Figure 19. Sample output from building of new FSA.	47
Figure 20. Prolog lists describing new FSA generated from a Trained SeqSOM.	48
Figure 21. Definition of indistinguishable states.	49
Figure 22. Aho's and Ullman's Lemma on determining if two states in a finite automata are indistinguishable.	49
Figure 23. Aho's and Ullman's algorithm for testing equivalence between two finite automata.	49
Figure 24. Equivalent FSA generated after training the SeqSOM network for FSA Test Case 1.	52

CHAPTER I

INTRODUCTION

Artificial Intelligence, a Perspective

Philosophers throughout human history have written about the “mind” and theorized about its mechanisms, but only recently has progress been made in understanding intelligence. Artificial Intelligence (AI) is an area of computer science that involves understanding intelligence and its automation. Much of the motivation and energy of this field is devoted to the replication of intelligent traits in machines. Many of these traits have human characteristics, such as the ability to recognize speech or perform the complicated task of launching a space shuttle. Television and film have propagated the notion that the field of AI has achieved great technological advances by portraying imaginary computers and robots that act with human intelligence and self will. This has led to the misconception that AI is only concerned with the re-invention of humans. Currently, the only way known of re-inventing the human is to have and to rear a child. AI should be thought of as a tool applied to machines that produces a degree of intelligence traditionally found in biological organisms.

Useful Artificial Intelligence

Robots that explore the surface of other planets are a classic example of where AI tools should be applied. Machines with AI characteristics, modeled after biological organisms with a central nervous system, could function independently without the need for constant human supervision. Without intelligence, the next pictures received from a roving robotic probe on the edge of a crater might be from the bottom. Although human intelligence is ideally suited for space exploration and other equally challenging jobs, it may not be in the best interest of human resources to do this type of work. A robotic probe must employ aspects of intelligence so that it may fulfill mission objectives. It must learn and make independent decisions based on information gained from its environment. The interaction of a probe with its environment is similar to that of a biological organism and its environment. Both organism and probe must monitor, react to, and control aspects of existence with regard to the various environments. Scientific evidence supports the idea that biological organisms developed central nervous systems as a means of survival, contrasted with bacteria, viruses, and other simplistic organisms that rely on fast replenishing rates and sheer numbers to insure survival. A nervous system provides a biological organism with an ability to react and to affect its environment [KohonenT89]. Neurological systems are especially adept at motor control, processing sensory data, and higher levels of control forming complex behaviors. Many features we wish to automate have already been designed by nature in the central nervous system.

Origins of Neural Research

Neurocomputing, the modeling of biological neural networks using digital computers, is a technique for dealing with problems, such as those discussed above, that are not readily solvable using traditional AI methods [MarkowitzJ94 and LawrenceJ90]. In 1943, shortly before the development of the first commercial computer, neurocomputing was first proposed by McCulloch-Pitts [McCullochW43]. Neurocomputing forms the basis for the field of neural modeling, which is a sub-field of AI that seeks to understand and automate behaviors of the biological nervous system.

Neural Modeling

Currently, there seems to be a two-tiered approach to neural modeling. The first approach is concerned with understanding the neuro-biological machine from the macro (physical structure) and micro (electro-chemical structure) levels. Research in this area is theoretical and is motivated by discovery and its goals are to learn how the neuro-biological systems work. The second approach is neurocomputing, and its focus is the modeling of the phenomena and function of their biological neural networks by using computer programs called Artificial Neural Networks (ANNs). ANNs are simulations designed to emulate aspects of the biological equivalent. There is an ongoing debate between the theoretical and applied research groups as to the need for a precise model and obtaining useful results. The neurocomputing approach is analogous to manned flight in which, "an airplane does not have to flap its wings to fly," meaning that neural network models may have little or no basis in biological reality except for the idea. The

backpropagation training method used to train feedforward and multilayered networks is one of the most common training techniques used and is a classic example of an ANN technique that has no resemblance to any natural structure within the brain, but produces results that are comparable [McClellandJ86].

Even though the foundations of neurocomputing were developed before the first commercial computer, neurocomputing has not always been a hot topic within computer science. Research in this area went through a period of disenchantment and lack of direction due in part to skepticism about the neural network as a viable and useful technique. Neurocomputing research was revitalized during the mid-1980's, and much of the renewed interest was brought about by the multi-layered, feed-forward network, and the backpropagation trained network [MarenA90]. The backpropagation trained networks' continued popularity is a result of its ability to solve a variety of problems that preceding networks could not [McClellandJ86]. Many researchers were diligent through the lean years by continuing to lay the foundations which would later be used for the resurgence of neural networks. Teuvo Kohonen is a researcher who pursued associative and topology-preserving neural networks during this time [MarenA90]. His development of the Self-Organizing Maps, or SOMs, is an important achievement in neurocomputing research.

Self Organizing Maps (SOMs)

The research emphasis of this paper focuses on artificial neural networks, and in particular, a variation of a Self-Organizing Map [KohonenT89]. Chapter II describes the self-organizing map architecture in detail. Generally, the self-organizing map is a

competitive network in which the winner takes all. Self-organizing maps are good at solving problems that consist of data containing relationships that are hidden due to the complexity of the problem or to irrelevant information (noise) obscuring the relationships. The pool of data that contains these relationships is called the dataspace. A hypothetical weather dataspace may include temperature, humidity, wind speed, dew point, pressure, etc. A self-organizing map could classify the relationship of these factors and map how they relate to the formation of different cloud types.

Generally, a SOM can be represented as a two-layer network consisting of “neurodes,” the simplest processing component in the neural network; neurodes of a SOM are modeled after the neurons in the biological nervous system. The SOM uses input signals to compute an output value designating the winner [KohonenT89]. An input signal (also called an input vector) is an ordered sequence of numbers called an n -tuple. Each element of the tuple represents a single trait, such as humidity, from the dataspace. Identifying hidden relationships among vectors in the dataspace is the goal of applying the neural network.

Each neurode contains an n -tuple, called the weight vector. The cardinality and ordering of weight vector values correspond to those in the input vector. In a SOM an input vector is distributed to each neurode simultaneously where it is matched with the corresponding weight vector. An activation function is used as a metric to gauge the similarity between the input vector and the weight vector of the neurode. Each neurode uses the same activation function to calculate the activation values. The neurode with the highest activation value is selected as the winner, and the minimum value may be selected if a different activation function is used. When a neurode is selected as the winner, it

means that the values of its weight vector most closely approximate the values of the input vector. The weight vector of the winning neurode is modified during the training period [CaudillM93 and KohonenT89].

The input vectors that most generally represent the dataspace become the training set for the neural network. However, in practice, the selection of the training sets is often inconclusive. Self-organizing maps should be trained using a dataset that represents well the relationships found in the problem domain. Using the weather example, the training set should contain a wide variety of atmospheric conditions that can be associated with known cloud types, even though not all conditions will be available for training. As discussed earlier, the training process involves modifying the weights of the winning neurode. Once training is complete, the training set is used to create a map of cloud types in relation to the winners. When new data are input to the SOM, they are classified to the best-matching neurode, even though the new input vectors may not duplicate or have existed the in training set.

Problems with SOMs

One problem with SOMs is the fixed size of their input vectors, which limits SOMs to classification of static signal patterns. A standard SOM processes each input vector as a single unit of data. As a result, standard SOMs cannot deal effectively with a collection of input vectors of differing sizes, which is necessary for processing data that has sequential or contextual components. The research herein describes a variation of the traditional SOM that properly handles sequential data and liberates the fixed size constraint on input vectors. This variation is called the Sequential Self-Organizing Map

or SeqSOM. A SeqSOM partitions each input signal and uses feedback to process the portions sequentially to achieve a categorization mapping similar to that produced by standard SOM [BoydstonR95]. A full description of the SeqSOM architecture and process is given in Chapter III.

CHAPTER II

LITERATURE REVIEW

Biological Neural Networks

The brain's neocortex is the site where information processing and intelligence occurs in advanced biological organisms. The human neocortex is a convoluted, layered structure of interconnected neurons, folded repeatedly to accommodate the vast number of neurons, estimated at one hundred thousand million [RitterH91]. Organizational complexity and information processing capabilities of the central nervous system distinguish one species from another.

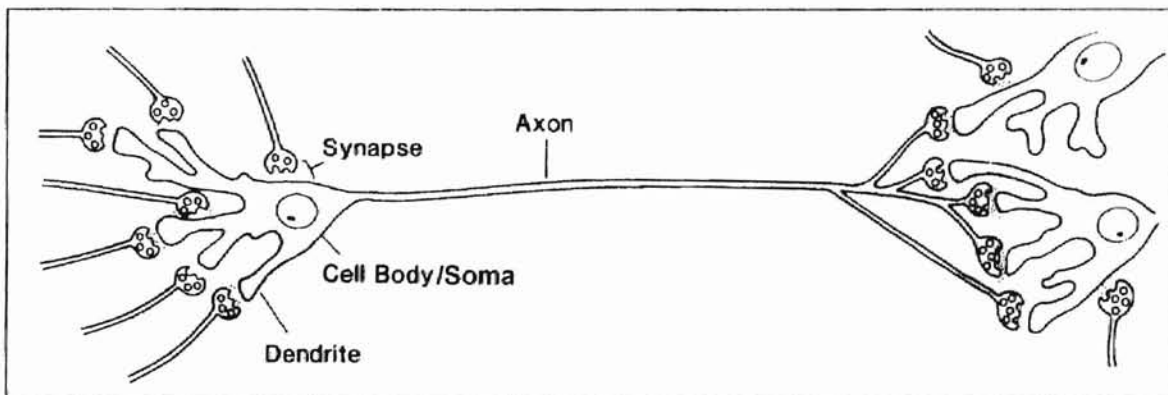


Figure 1. Biological Neuron [McClellandJ86].

A neuron is the simplest functional component and, in general, is similar across species. The typical biological neuron has four major components: dendrites, a soma, an axon, and synapses. A typical neuron resembles the branches, trunk, and roots of a tree. Figure 1 is an artist's representation of a neuron [McClellandJ86].

Dendrites. The dendrites are physically shaped as branch-like extensions of the cell body, and provide the input functions for the neuron, among other activities. The dendrites receive electro-chemical stimuli from other neurons. These input signals are propagated through the dendrite structure and are brought together at the soma, or cell body.

Soma. The soma is where the "processing" occurs in the neuron. The strength of the input signals at the soma is dependent on the distance and attenuation present during the transmission through the dendrites. The processing that occurs can be characterized as a summation of the dendrite inputs. If the calculated sum exceeds a set threshold, then an output signal is generated. This signal is similar to an electrical spike and is transferred through the axon to other neurons [RitterH91].

Axon. The axon is a long wire-like structure that carries the signal away from the soma to other neurons. Like the dendrites, an axon has an outgrowth of branches at its end. These branches come together with the dendrites of other neurons to form synapses.

Synapse. The synapse is the small space between an axon branch of one neuron and a dendrite of another. A signal pulse is transmitted across a synapse via a chemical process. The axon secretes neurotransmitter chemicals, which create an electrical potential difference between the axon and the dendrite. Synaptic connections may be

excitatory (Type I) or inhibitory (Type II), in that the signal-forwarding may be enhanced or reduced at the target neuron [RitterH91].

The physical shape of a neuron identifies it as having a specific type of synaptic connection. Neurons that are shaped like a pyramid are called pyramidal cells and have Type I synaptic connections. Stellate cells are star-shaped neurons having Type II synapses. Pyramidal neurons have a well-defined axon and a large number of synapses, and like all Type I neurons, they can extend long distances to other regions of the brain and nervous system. Stellate cells are more localized, with the axon branching into synapses limited to the immediate area. Stellate cells act to “corral” the Type I neurons during excitation; that is, it is believed that the stellate cells stabilize the excitation site by inhibiting activity around the stimulated region (lateral inhibition) [RitterH91].

Microcolumns. Neurons in the regions of excitation are thought to form functional groupings called microcolumns. Microcolumns are cylinders of neurons that extend vertically inward from the neocortical surface. Microcolumns generally have Type I cells at the center and Type II cells at the boundary and serve as higher level processing elements. Microcolumns have no real borders but gradually transform to other functional regions.

Some microcolumns are grouped to form still higher organizational structures known as “cortical areas”. According to [RitterH91] cortical areas provide specialized functions for specific tasks such as aspects of speech comprehension, spatial orientation, planning and execution of movements, analysis of edge orientation and of color shades, etc. Over eighty cortical areas have been identified in the human cortex. Cortical arrangement is so regular and so correlated to sensory receptors throughout the body that

the cortical surface almost mirrors the anatomical and physiological relationships of the sensory organs. For example, the cortical areas that stimulate the finger lie “atop” the area that stimulates the palm, reflecting the “shape” of the hand.

The neocortex is a layered structure that typically consists of six distinguishing bands of neurons as in Figure 2.

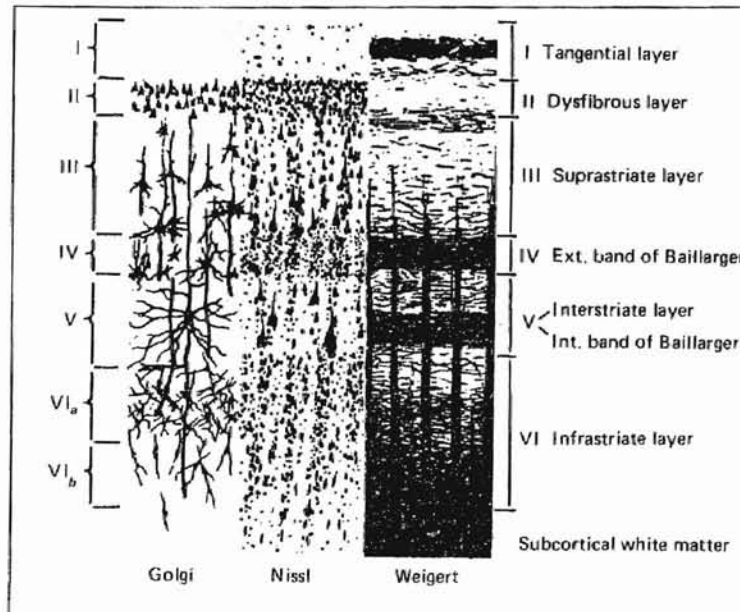


Figure 2. Six layers of the neocortex [CotterillR88].

Evidence suggests that variation in the number and thickness of layers is dependent on the region and its function [McClellandJ86 and CotterillR88].

Layers. Input signals from the outside world enter the neocortex by passing through the thalamus. The thalamus is responsible for forwarding and distributing input. The thalamus neurons project into Layer IV where they provide inputs to other cortical areas. Neurons in the middle layer extend upward to Layers II and III. Layers II and III connect to other cortical areas either on the same side of the brain or on the opposite

hemisphere. Layer I has few neurons. Layers V and VI, or deep layers, are considered the output layers. The neurons in these layers terminate away from the neocortex in other subcortical areas.

This discussion of biological neural networks is meant to show their relationship to an Artificial Neural Network (ANN) called the self-organizing map. Further discussion of the biological aspects is beyond the scope of this thesis. For additional information on biological neural networks, consult the references at the end of this thesis.

Self-Organizing Maps (SOM)

The study of biological neural networks provides a foundation for artificial neural network (ANN) modeling. This relationship is evident in the design of the self-organizing map, because as an artificial neural network model it was formulated from empirical evidence gained in the observation and study of biological neural networks. Self-organizing maps provide the functional processes of topological map formation and dimensional or information reduction that naturally occur in biological neural networks. This section will show the similarity between the biological model discussed in the previous section and the self-organizing map.

As a caveat, artificial neural network models, including SOMs, are incomplete implementations because the fundamental research is either insufficient or undiscovered [KohonenT89]. These gaps in understanding biological neural networks are due in part to moral considerations that limit direct experimentation; therefore, artificial neural

networks are coarse representations of the architectural and behavioral aspects of biological neural networks.

ANN Architecture.

An artificial neural network is an organized collection of artificial neurons. An artificial neuron is analogous to a biological neuron. It is the simplest functional component of artificial neural networks, including the self-organizing map. An artificial neuron is also called a “processing element” and a “neurode”, which is the concatenation of *neur* from “neuron” and *ode* from “node”.

An artificial neural network architecture is comprised of processing elements and their connections to other processing elements. The architecture of many artificial neural networks is similar to that of their biological counterparts in the arrangement of neurons into layers. In the previous section, Figure 2 illustrates the layering of a biological networks. A simpler structure, shown in Figure 3, illustrates a layered architecture of an artificial neural network. Figure 3 illustrates a feed-forward ANN consisting of three layers: the input layer, the hidden layer, and the output layer. The input layer contains no

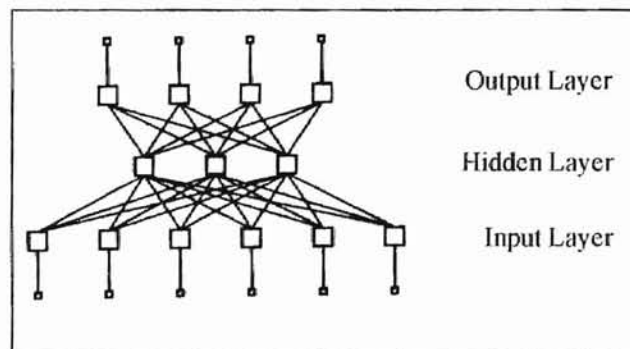


Figure 3. Illustration of layered organization of an Artificial Neural Networks [RitterH91].

neurodes; its purpose is simply to broadcast the input signal to each of the hidden-layer neurodes so their activation values may be calculated. Once the activation values are computed for the hidden layer, they are forwarded to the output layer. Next, the activation values of the output layer neurodes are computed. The activation values of the output form the network's response to a given input. To train the network, an error value is calculated for each output layer neurode by comparing its activation value against its target value. Once all the error values are calculated, they are distributed back to the hidden layer, where these values are used to update the internal weights. The feedforward network is a popular ANN and just one of many different types [FausettL94]. The network architecture for this paper is the self-organizing map.

Self-Organizing Map Architecture

A self-organizing map is a two-layer ANN consisting of an input and output layer [CaudillM93, DayhoffJ90, KohonenT88a, and KohonenT89]. The output layer is typically arranged as a two-dimensional array of neurodes that are not interconnected [DayhoffJ90, FausettL94, KohonenT88b, and KohonenT89]. Self-organizing maps with output layers of higher dimensions are useful for applications such as robot path planning [RitterH91].

One of the desired behaviors of SOMs is that they can produce some of the map structures that are found in biological neural networks [KohonenT89]. The ability to reproduce these naturally occurring mapping behaviors is unique to the SOM and is not found in other artificial neural network techniques even though other ANN architectures can categorize as well [FausettL94].

Input and Output Layers

A pattern that is received by the input layer of an ANN is analogous to the input received at the dendrites of a collection of neurons. The input layer of a self-organizing map consists of a tuple of non-computational elements. The input layer is completely connected to the output layer and delivers the input signal to each neurode in the output layer. An input signal is a vector or k-tuple:

$$\mathbf{i} = (i_1, i_2, \dots, i_k) \quad (1)$$

Each neurode in the input layer is connected to the output layer via a weight vector:

$$\mathbf{w}_j = (w_{j1}, w_{j2}, \dots, w_{jk}) \quad (2)$$

where $1 \leq j \leq$ the number of neurodes in the output layer. The weight vector is the same size as each input vector. That is for each input vector element, k , there is a corresponding weight vector element, k .

Weights

The weights associated with the connections between neurodes represent the strength, or signal capacity, of connections between the output of one neurode and input of another. They are similar to the synaptic gaps of a biological network. Weights are used to calculate the response of a neurode to a given input signal and may be modified during the training, as described below. The modification is based on a training function that provides the learning property.

The neurode weights are “plastic features” that enable a SOM to organize itself through competitive learning and match the topology of the input signal space. Output

neurodes maintain their positional relationships with their neighbors, but as training takes place, their topological relationships are changed.

The property of weight modification gives rise to the desirable learning behaviors of neural networks. In competitive networks, such as the self-organizing map, the neurodes compete for the privilege of representing an input signal. The training function rewards the winner by changing its weights to more closely match the input signal. Competition occurs among the neurodes based on their activation values, which result from the activation function.

Activation Function.

In a biological neuron the activation function is very complex and can be generalized only for small areas of the brain. Likewise, ANNs have the same activation function for each layer, since they are meant only to represent specific areas [KohonenT95]. The activation function can be described as a mechanism that controls the state of excitation of the cell. When a neuron “fires”, it is not due to chance, but is due to a defined behavior within the cell.

In a SOM the value of this function for each neurode is calculated as either the Euclidean distance or the dot product between the input vector and the weight vector. The winning neurode is determined by calculating the activation values of all neurodes in the output layer and then choosing the neurode that has the minimum (for Euclidean distance) or maximum (for dot product) activation value. Neurodes in the vicinity (or neighborhood) of the winning neurode, including the winning neurode itself, are rewarded by having their weight vectors modified to bring them closer to the input

training vector; this reinforces the ability of neurodes in the neighborhood to successfully approximate similar input vectors. The categorization that results from this process may seem “intelligent”, but it is merely the application of an activation function and the adjustments of the appropriate weights [BoydstonR95]. The process of weight modification is called training.

Training

A collection of vectors used to train an ANN is called the training set, and each member of this set is presented to the ANN during the training process. Used with learning rules, the weights may be modified to more closely match the input cases. An ANN may be trained using a supervised method or an unsupervised method. In supervised training, both the input (training sets) and output (desired categorization) are known in advance. During training the learning algorithm will take these known quantities into account by adjusting the weights to map the input cases to the output cases.

Unsupervised training is similar to supervised training, except the output vector is not specified as part of the training data. That is, the network is presented with a set of input vectors, but no output classification is known in advance. Once training has occurred, the training set is used to categorize the sample cases. More training may be required for the unsupervised training method.

A training set should consist of a wide variety of example cases; however, the training set selection and its quality may not be conducive to one's wants or needs. After the training set is chosen, the weight vectors of the SOM are initialized to random values;

however, if preliminary knowledge of neurode weights is available or previous training has taken place, then weights may be loaded from a file. Overtraining is not a problem with SOM networks as it is with other neural network techniques [HiotisA93]. During training, the SOM “learns” to classify the training set of input vectors. After training is complete the SOM is ready to classify new input vectors. These new input vectors are classified according to the best-matching neurode.

Topology-Preserving Mapping

Topology is defined as “the study of properties of geometric form that remain invariant under certain transformations such as bending or stretching” [AmericanH92]. Topology-preserving mappings maintain the ordering associated with multidimensional signal data while reducing the dimension using an onto relationship [KohonenT89]. The signal data can be thought of as meaningful information mixed with noise or extraneous data. The noise and high dimensionality of the data obscure topological relationships. Topology-preserving mappings serve to remove the superfluous data so that the relationships can be seen. Thus, the complexity of the high-dimensional data is “abstracted away”, leaving only the most basic relationships.

Lateral Feedback

The neurons of the outer layer of the brain have both input connections and lateral interconnections. A single neuron can have as many as ten thousand lateral interconnections to surrounding neurons and as many returning to it [KohonenT89]. As discussed earlier in this chapter, biological lateral feedback is associated with neurons of

the brain's outer layers (Layers II and III), called the neocortex. The physical evidence suggests that the degree of lateral interaction is related to the distance at which excitation occurs. Neurons that are physically closest to active cells have positive lateral feedback. The positive feedback diminishes outward from the center of excitation. Further out from the excitation point a region of negative lateral feedback is surrounded by another region of minimal positive feedback [KohonenT89]. The overall physical structure of the outer layer is arranged as a two-dimensional layer, hence the layer arrangement of the SOM model.

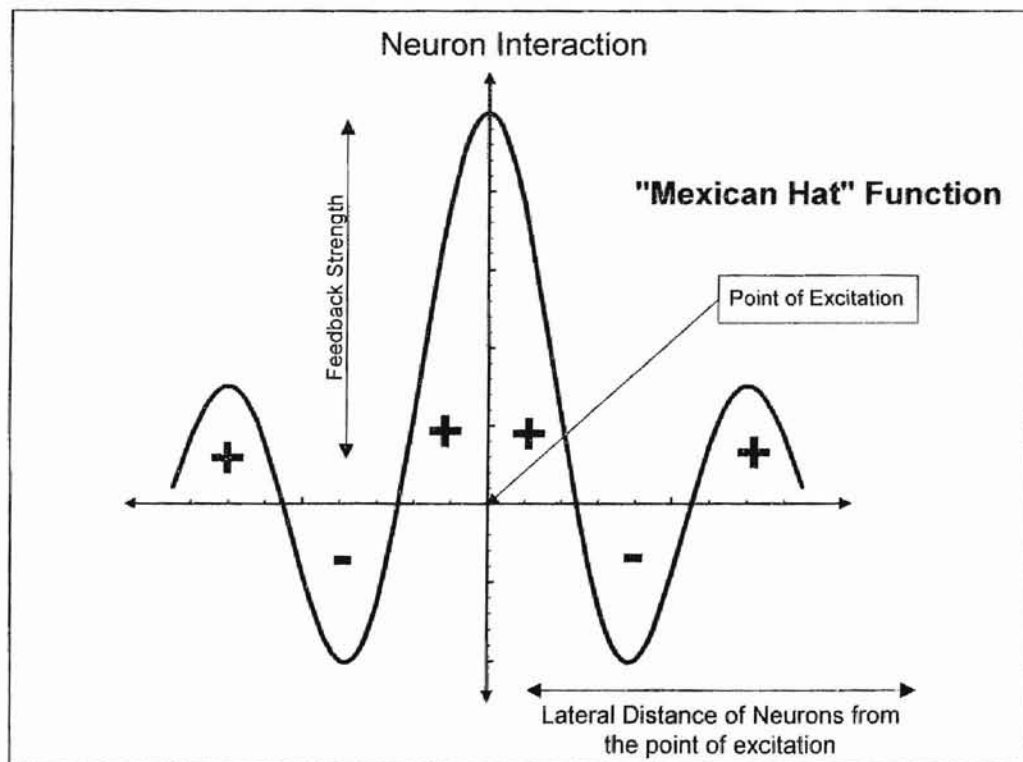


Figure 4. "Mexican Hat" function illustrating positive and negative lateral interaction of neurons as a function of distance from the point of excitation.

The “Mexican Hat” function (Figure 4) illustrates the relationship between the lateral distance of neurons from the excitation point and the strength of lateral feedback. The “Mexican Hat” function corresponds to activity in microcolumns and is analogous to structures of the SOM.

In biological systems, the effect of lateral feedback is that similar signal patterns “cluster” around a winning neuron. Clustering is essential for map formation. The resultant size of the cluster region depends on how the lateral feedback is applied. Increasing positive feedback broadens the cluster, using negative feedback tends to sharpen its edges, and omitting feedback does not allow clusters to form [KohonenT89].

The SOM model differs from biological systems in that it does not directly implement lateral feedback. As mentioned earlier, the neurodes of the SOM’s output layer are not interconnected; there are no lateral connections. However, since lateral feedback is necessary for the formations of clusters, it is indirectly implemented during the training process through the use of neighborhoods [KohonenT89].

Neighborhoods

A neighborhood includes all neurodes within a given radius of a winning neurode. It represents the lateral distance between neurons [KohonenT89]. This corresponds to the center region of the “Mexican Hat” function. Only the weight vectors of neurodes within the neighborhood are modified [KohonenT89], which simplifies the “Mexican Hat” to a step function (Figure 5).

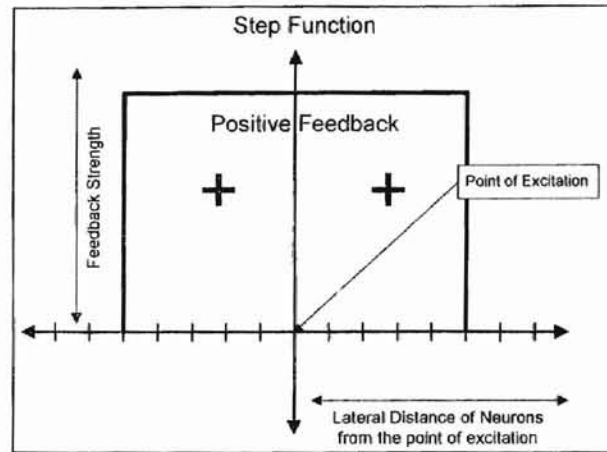


Figure 5. Step function illustrating the neighborhood region and neighborhood radius as related to the lateral distance of neurodes.

The use of this simplified feedback function does not affect the self-organization process. Kohonen states that there are other lateral feedback functions that also produce the clustering effect. The inhibiting feedback is ignored for the basic SOM model, but it may be necessary for other functions such as autoassociative memory [KohonenT89]. The neighborhood radius gradually decreases as training progresses. Kohonen suggests using the following linearly decreasing function:

$$\eta_{new} = \left[\eta_{old} \left(1 - \frac{\tau}{\tau_{max}} \right) \right] \quad (3)$$

where η is neighborhood radius and τ is elapsed training time and τ_{max} is the maximum allowable training time [DayhoffJ90, FausettL94, and KohonenT89]. Decreasing the neighborhood radius over time sharpens the response of winners while creating clusters

of similarity. A neighborhood that includes only the winning neurode (a neighborhood radius of zero) does not develop clusters that characterize similarity patterns.

Learning Rate

The learning rate can be thought of as the rate at which weight vectors are changed. As training progresses, the learning rate is decreased gradually using the linearly decreasing function [DayhoffJ90, FausettL94, and KohonenT89]

$$\alpha_{new} = \alpha_{old} \left(1 - \frac{\tau}{\tau_{max}} \right) \quad (4)$$

where α is learning rate and τ is elapsed training time and τ_{max} is maximum allowable training time. Kohonen states that a geometrically decreasing function produces similar results [FausettL94 and KohonenT89]. The effect of decreasing the learning rate over time causes convergence of similar patterns or formation of clusters [KohonenT89]. A large α value will cause the weight vector of the winning neurodes to vary greatly.

Typically a SOM is trained in two phases before it is used for classifying input vectors. The first phase uses larger α values for major convergence or overall ordering, then the weight vectors can be "fine tuned" with smaller α values [FausettL94].

Weight Adjustment

Modifying only those neurodes inside the neighborhood, the weights within the winner's neighborhood, are modified according to the following adaptive function:

$$w_{ij} = w_{ij} + \alpha (i_n - w_{ij}) \quad (5)$$

where w_n is the a new weight value, α is the current learning rate and i_n is the current input value.

Sequential Processing Using Self-Organizing Maps

Unprocessed human speech is an example of high-dimensional data containing temporal information that is successfully processed by biological neural networks. Every day human brains demonstrate that artificial neural network models require more refinement to emulate them more effectively [KohonenT89]. Kohonen questions whether the temporal components are part of processing or a result of a higher level of organization. Some standard techniques currently being used for handling temporal aspects of human speech include: Hidden Markov models (HMM), Hidden Control Neural Networks (HCNN), and Time Delay Neural Networks (TDNN), [MarkowitzJ94, KohonenT88a]. These techniques tend to use the artificial neural networks as a preprocessing layer [MarkowitzJ94] rather than as a direct implementation of temporal processing within the ANN.

Speech recognition is a difficult task. It is time-dependent; that is, information content is occurring over some period of time. The modifications to self-organizing maps discussed below are directed at speech recognition and processing. Speech processing and recognition are not within the scope of this paper; however, these topics will be discussed as they relate to the methods of temporal processing.

Input Sequence Averaging

Self-organizing maps can be used to classify phonemes (the basic elements of speech) by using discrete samples of the input signal [KohonenT88a]. Jeri Kangas describes several modifications to the self-organizing map that attempt to deal with the time dependencies of input signals. One of these is input sequence averaging [KangasJ90].

Input sequence averaging is a two-stage method consisting of a preprocessing front-end (input vector averaging) and a self-organizing map (ANN). The preprocessing front-end step combines the current input vector and a running weighted average of previous input vectors to compute the input to the SOM. Kangas describes how the weighted average stores the historical contribution of input vector sequences in the resulting contribution from a recursively calculated average [KangasJ90]. This means that the average can be computed as each input vector is fed into the network. Kangas states that the average being used is called a “backward exponentially weighted sum”, as shown in Equation 6 [KangasJ90].

$$x_w(t) = \beta \cdot y(t) + (1.0 - \beta) \cdot x_w(t-1) \quad (6)$$

For each input signal in the time sequence the average is calculated. The contribution that the historical information makes to the average is determined by the weighting factor, β . The parameter t designates the time within the sequence. The parameter $y(t)$ is the current input vector; the weighting factor β will determine the contribution of the current input vector. The remaining portion of the equation,

$(1.0 - \beta) \cdot x_w(t-1)$, is responsible for the historical contribution. Once the average is calculated, the $x_w(t)$ vector is used as input to a traditional self-organizing map [KangasJ90].

The averaging allows the network to tolerate noise by acting as a low-pass filter for the input signal. However, this method is not without problems. If the weight factor is too small, the contribution of the current input is forgotten; and a large factor parameter will cause historical contributions to be lost. This method is modified to deal with shadowing of previous input values.

Response Integration Model

This neural network, as shown in Figure 6, consists of two SOM layers with averaging. The first layer is a traditional Kohonen self-organizing map. A speech signal is fed into the first layer where it is processed. The input is fed as a sequence of vectors. As they are processed, the sequences provide a set of responses from the first layer. These responses are combined using Equation 6.

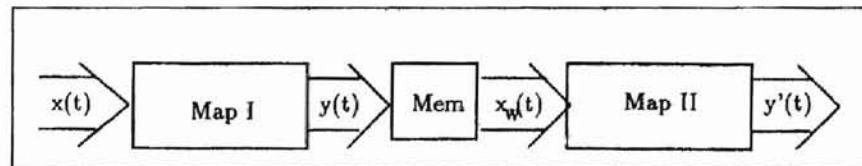


Figure 6. Architecture for the Response Integration Model [KangasJ91a].

The purpose of the first map is to reduce the problem of input values shadowing with previous input values. The second map is then used to average the inputs from the first map [KangasJ91a].

Pattern Concatenation Model

In the pattern concatenation model a series of shift registers saves the current input vector and a small historical set of input vectors. The contents of the shift registers are concatenated and used as input to the SOM where it is classified. Figure 7 illustrates the architecture of this network. The input samples are moved forward through the shift registers in a First-In-First-Out (FIFO) manner so that the next input sample can be incorporated. Each new input sample causes the oldest sample to be discarded. The shift registers provide historical information in the classification of signals with time dependencies.

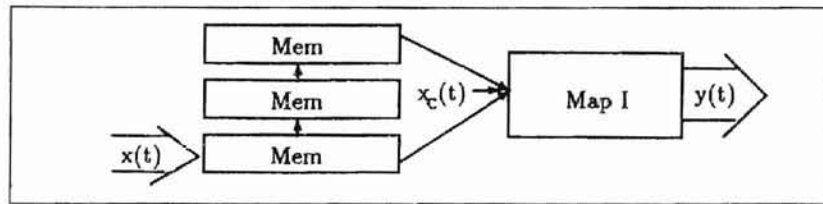


Figure 7. Architecture of the Pattern Concatenation Neural Network [KangasJ91b].

This method does not have the advantage of eliminating the noise as does the Input Sequence Averaging method discussed above; however, there is no diminishing of data values due to a weighting factor. The contents of the shift register $x_c(t)$ at time t can be described by the following equation.

$$x_c(t) = \{x(t), x(t-1), x(t-2), \dots, x(t-(n-1))\} \quad (7)$$

Each of these methods described by Kangas require no changes to the basic self-organizing map architecture. However, training times and parameter values may require adjustment [KangasJ91b].

Trace Feature Maps

Trace Feature Map (TFM) is a neural network algorithm intended for processing elements of speech. The TFM network is proposed as a subcomponent of a larger speech recognition system. TFM uses self-organizing maps to store speech data that is invariant of time. The purpose of TFM is the storage of short acoustic sequences and the reduction of stored data within the map. Zandhuis makes the assumption that within a spoken word there are series of events that can be processed more naturally than breaking them into discrete phonetic components. The events of speech rather than the passage of time drive the TFM network [ZandhuisJ92].

A TFM neural network has a hierarchical architecture consisting of two layers, the C layer (feature classifier layer) and T layer (sequence storing layer), as shown in Figure 8. The first tier classifies features of an input signal onto a two-dimensional self-

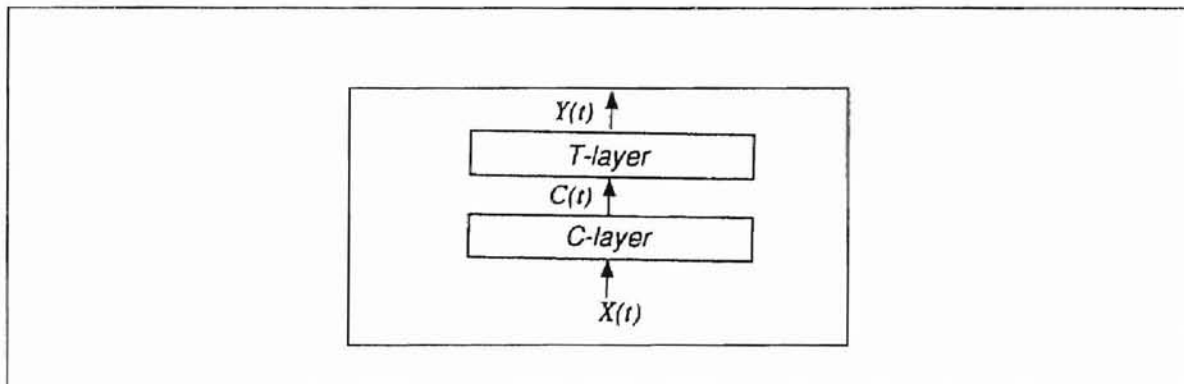


Figure 8. Trace Feature Map Architecture [ZandhuisJ92].

organizing map. The first layer differs from the traditional SOM by its use of a threshold activation function, $F(z, \phi)$. This means a neurode will only be active if its calculated

Trace Feature Maps

Trace Feature Map (TFM) is a neural network algorithm intended for processing elements of speech. The TFM network is proposed as a subcomponent of a larger speech recognition system. TFM uses self-organizing maps to store speech data that is invariant of time. The purpose of TFM is the storage of short acoustic sequences and the reduction of stored data within the map. Zandhuis makes the assumption that within a spoken word there are series of events that can be processed more naturally than breaking them into discrete phonetic components. The events of speech rather than the passage of time drive the TFM network [ZandhuisJ92].

A TFM neural network has a hierarchical architecture consisting of two layers, the C layer (feature classifier layer) and T layer (sequence storing layer), as shown in Figure 8. The first tier classifies features of an input signal onto a two-dimensional self-

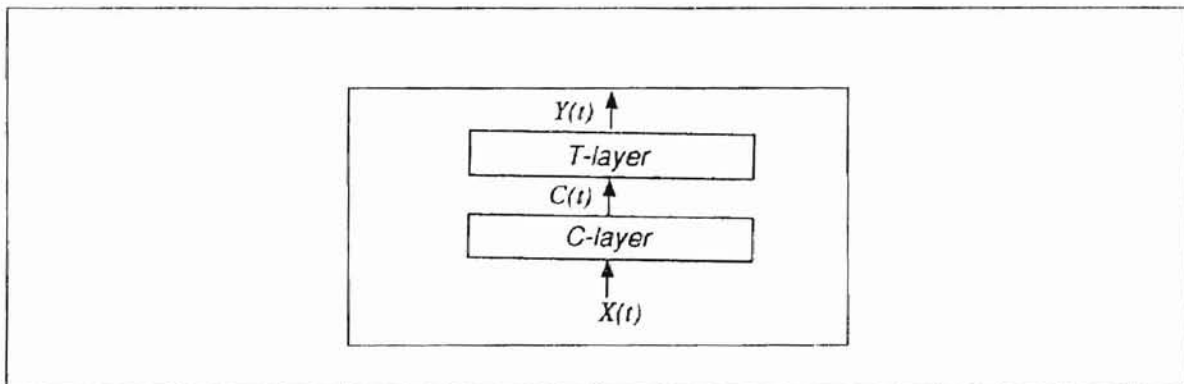


Figure 8. Trace Feature Map Architecture [ZandhuisJ92].

organizing map. The first layer differs from the traditional SOM by its use of a threshold activation function, $F(z, \phi)$. This means a neurode will only be active if its calculated

Euclidean distance is over a particular threshold level, ϕ . This method is known as coarse coding. It has the advantages of accurately classifying inputs while maintaining a smaller map size and maintaining the neighborhood relationships [ZandhuisJ92].

The T-layer is a SOM that uses lateral feedback based on the “Mexican Hat” function to implement excitatory and inhibitory responses. The neurodes of the T-layer seek a state of equilibrium. A neuron will stay at its current state until the next steady state is reached. The effect of these state changes can be seen in the path diagrams produced as the input signal is processed [ZandhuisJ92].

As an acoustical signal is processed by a TFM, a path will form on the second tier that is representative of a signal’s activity within a time window. The size of this time window is variable because ΔT , the utterance time, is dependent on signal changes rather than on time changes. This means that the path of activity for a given signal is independent of duration. This allows utterances to be fast or slow [ZandhuisJ92].

The research presented in Chapter III takes the processing approach of incorporating sequential processing components as part of the self-organizing map neural network. The addition of sequential processing was developed after studying Kohonen’s work on the Phonic Typewriter and his use of input vectors and the resultant paths for continuous speech recognition [KohonenT88a]. The sequential Kohonen neural network algorithm, Figure 11, is a modification to the Kohonen self-organizing map algorithm as shown in Figure 10 [FausettL94].

CHAPTER III

METHODS

Introduction

The Kohonen self-organizing map is well suited to capturing the static relationships of an input signal space; however, standard Self-Organizing Maps (SOMs) are not able to process signals that have contextual or temporal components. If a sequence of input vectors is fed into a standard SOM, the SOM is unable to capture and identify the sequential relationships between the input vectors of the sequence. Variations of self-organizing maps have been developed that have the capacity to deal with contextual and temporal data; these are described in Chapter II. The research presented here describes a new variation of a self-organizing map for processing contextual data. The Sequential Self-Organizing Map (SeqSOM) uses feedback to relate a sequence of input vectors.

Feedback is used in many electrical systems, and one such example is the linear amplifier. In the case of the linear amplifier, a portion of the output is used as feedback with the input signal. Likewise, SeqSOM uses a portion of its output as feedback with the next input vector. The Sequential Self-organizing map is somewhat similar to work done by Ghosh and Karamcheti with the artificial neural network called Simple Recurrent

Network (SRN) [GhoshJ92 and ElmanJ90]. Ghosh and Karamcheti use a modified feedforward network with feedback to provide the contextual processing needed. More information regarding the SRN network architecture can be found by investigating the research of [GhoshJ92 and ElmanJ90] listed in the reference section.

SeqSOM Architecture

The SeqSOM architecture, as illustrated in Figure 11, is similar to that of a traditional self-organizing map; for a comparison, see the SOM architecture shown in Figure 9. Both network architectures have the same basic grid structure of neurodes. In both the SeqSOM and SOM networks, an input vector is distributed to each neurode in the grid for calculation of activation values. The winning neurode is determined from these activation values, and the weight vectors of the winning neurode and its neighbors are modified.

The major difference between the SeqSOM and the SOM architectures is the use of feedback in SeqSOM. In the SeqSOM architecture, a vector is distributed to each neurode in the grid and the activation values are calculated. When the winning neurode is determined, information about the winning neurode is then used as feedback along with the next input vector. The feedback information consists of the spatial coordinates of the winning neurode (i.e., the row, column, and plane coordinates of the winning neurode). The new vector that is formed by the concatenation of feedback coordinates and the next input vector is called an input bundle. The feedback for the first input bundle of each sequence of input vectors is set to zero. All subsequent input bundles for the sequence are formed using the coordinates of the winning neurode from the previous input bundle.

The input bundles are processed sequentially until the complete sequence of input vectors is consumed.

Also, to accommodate the feedback, three additional values must be added to the weight vector of each neurode. The size of the new weight vector matches the size of the input bundle.

The SeqSOM program used for this research can support output layers of up to three dimensions; however, typical applications involving SOMs use a two-dimensional output layer arranged as rows and columns. A two-dimensional network can be viewed as a three-dimensional network consisting of a single plane.

Research results indicate that the feedback values need to be scaled so that they do not skew the activation values in favor of the feedback.

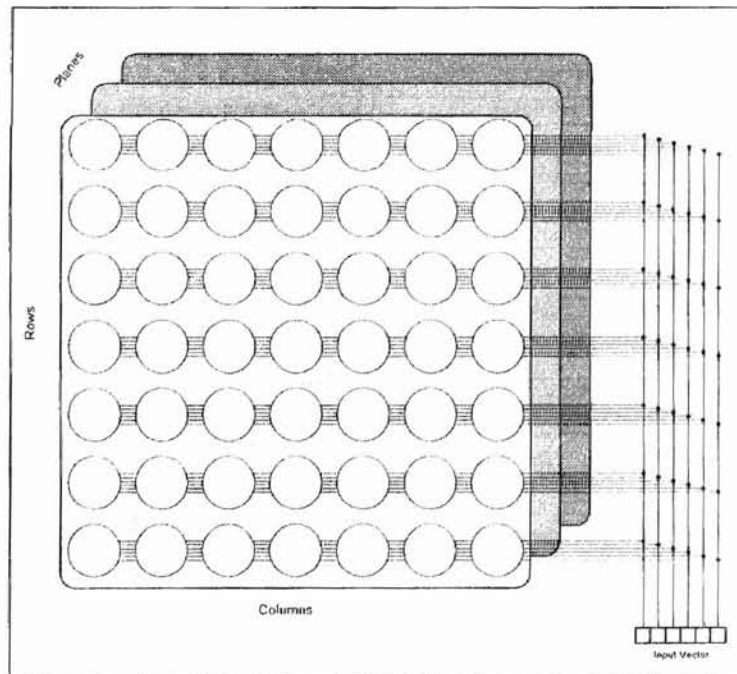


Figure 9. Illustration of the Self-Organizing Map Architecture.

Example of Input Bundle Formation

Consider the SeqSOM example shown in Figure 11. It has an input vector size of six and uses a three-dimensional grid. When the input vector and the feedback are concatenated, they form an input bundle of size nine. The weight vector size for each neurode is also nine. The first input bundle is considered a special case because it has no feedback values available; therefore, the feedback values of the first input bundle are initialized to zero. As in traditional SOMs, a winning neurode is identified for each input bundle processed, but unlike the traditional SOM, the SeqSOM uses the coordinates of the winning neurode to construct the next input bundle.

ALGORITHM 1

```
initialize all neurodes' weight vectors to random values;  
initialize neighborhood radius;  
initialize training time;  
while remaining training time is not zero  
begin  
  for all vectors in the input training set  
  begin  
    calculate the activation value for each neurode;  
    locate the minimum activation value;  
    adjust the weights of winner and its neighbors;  
  end  
  adjust neighborhood radius;  
  adjust learning rate;  
  decrement training time;  
end.
```

Figure 10. Algorithm of Kohonen's Self-Organizing Map.

The Sequential Self-Organizing Map Algorithm

The SOM and SeqSOM training algorithms are shown in Figure 10 and Figure 12, respectively. The SeqSOM algorithm is an extension of the SOM algorithm; these extensions are shown in bold, in Figure 12.

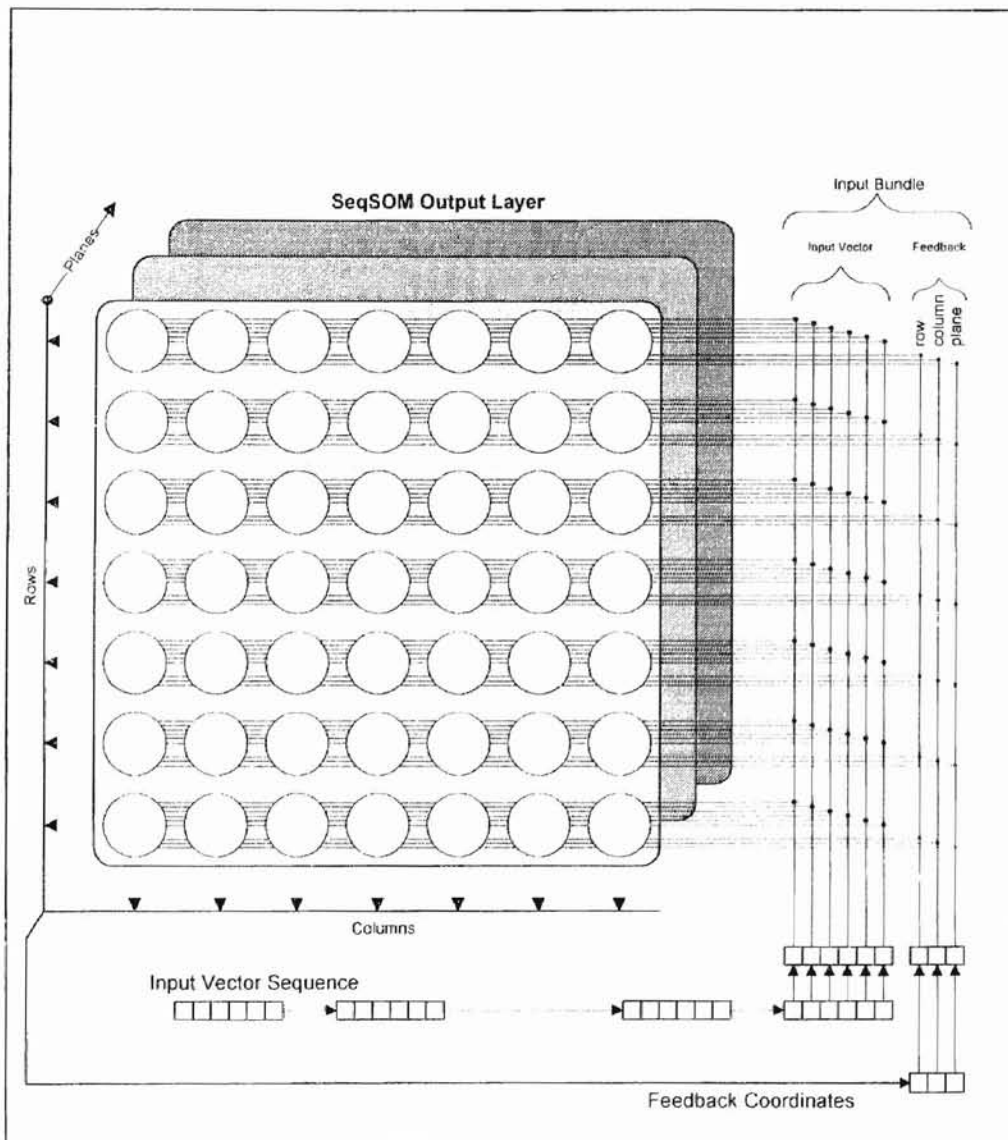


Figure 11. Illustration of the SeqSOM Architecture.

ALGORITHM 2

```
initialize all neurodes' weight vectors to random values;
initialize neighborhood radius;
initialize the training time;
while remaining training time is not zero
begin
  for all input sequences in the training set
  begin
    set feedback to zero;
    for each input vector in the sequence
    begin
      form an input bundle from input vector and feedback;
      calculate the activation value for each neurode;
      locate the minimum activation value;
      adjust the weights of winner and its neighbors;
      set feedback to coordinates of winner;
    end
  end
  adjust neighborhood radius;
  adjust learning rate;
  decrement training time;
end.
```

Figure 12. Sequential Self-Organizing Map Algorithm.

CHAPTER IV

IMPLEMENTATION AND TESTING

Algorithm Implementation

Early in the development phase of the SeqSOM project, software implementations of standard self-organizing maps were examined for potential modification to include feedback. One such package, called SOM_PAK, is available to the public via anonymous FTP: it is Teuvo Kohonen's own implementation of the self-organizing map. SOM_PAK includes all the source code files necessary to compile the program executables for a SOM network with supporting programs. The idea of using an existing application or modifying existing code was impractical due to the extensive changes that would have been required. Rather, SeqSOM was implemented first as a standard self-organizing map based on pseudo code provided by [FausettL94] and then modified to support spatial feedback. Spatial feedback is a new term and refers to the use of the spatial coordinates of a winning neurode as feedback into the next pass through the network. From its beginning, the SeqSOM project had the goal of providing feedback in traditional self-organizing maps.

The SeqSOM program is capable of supporting the use of fixed length input vectors in the same way that they are used in traditional self-organizing maps. In addition SeqSOM supports the use of variable length input vectors. A control file that contains

parameters is used to determine the behavior of a SeqSOM network. This means that by setting parameters within the control file, SeqSOM can be made to behave like a standard SOM.

Development Environment

The SeqSOM program is implemented in the C++ object-oriented programming language. The program was designed as a single executable program for both the training and usage phases. Command line options are specified to control the operation of SeqSOM. These command line options are listed in Appendix A. A control file containing parameters is used to control the behavior of SeqSOM. The execution time of the program varies according to the input vector sizes, length of training time, and number of neurodes in the network. Training time can range from a few minutes to many hours. Using a trained SeqSOM network is much faster than the time involved in training the network. Caching of the activation values is performed by Kohonen's SOM_PAK programs to speed the training process [KohonenT89]. Efficiency considerations of the SeqSOM application are discussed in Chapter V; however, improvement of its efficiency is left as future work.

Testing Environment and Platform

The SeqSOM program was developed for the platforms of Microsoft[®] Windows NT[®] and AT&T UNIX System V operating systems. Using conditional compiles, a single copy of the source code will compile on the different operating environments. The

complete source code for the SeqSOM project is not included in this thesis but it may be obtained for research purposes by contacting the author.

Test Data Domain

Once the SeqSOM algorithm was implemented as a program, selection of appropriate testing data occurred next. Many types of data were examined for possible use. Since SeqSOM is designed to use feedback, only data sets consisting of strings (i.e., ordered sequences of symbols) were considered. Some examples of data considered are encoded speech [KohonenT88a] and waveform generation [FausettL94]. These were not chosen because of their complexity and were not needed to show the validity of the SeqSOM algorithm. The data that was finally chosen is described by Ghosh and Karamcheti using Elman's feedforward recurrent neural network [GhoshJ92 and ElmanJ90]. Elman describes a neural network application that builds a representation of a Finite State Automaton (FSA) from a subset of the language generated by the FSA [ElmanJ90]. For their tests, Ghosh and Karamcheti used strings generated from a regular grammar [GhoshJ92].

Selection of Test FSAs.

For the Ghosh and Karamcheti tests, strings were generated from the FSA shown in Figure 13. The strings that were chosen initially to test SeqSOM were generated by the same FSA as used by [GhoshJ92]. It was decided to use the same FSA because the results obtained from SeqSOM could be compared readily with the results obtained by [GhoshJ92]. They chose 60,000 to 80,000 strings with a length constraint of 32, but the

data chosen for SeqSOM consisted of all strings of length 10 or less (a total of 103 strings). Other FSAs were chosen later to extend the testing of SeqSOM. These are included in Appendix C.

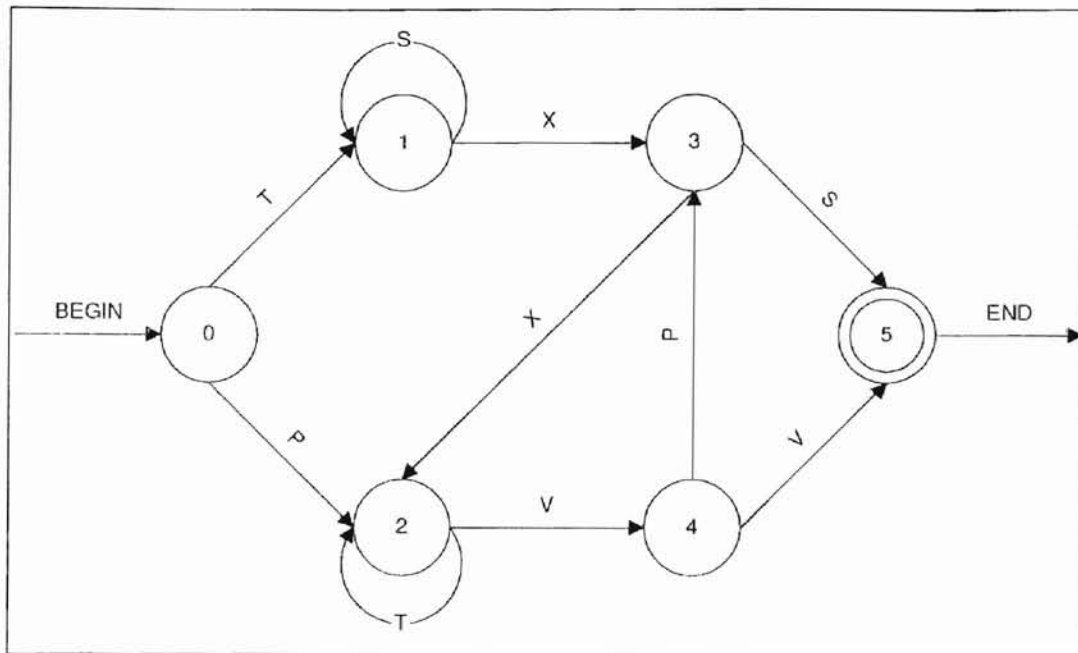


Figure 13. The FSA state diagram used for the Ghosh and Karamcheti research.

The focus of the Ghosh and Karamcheti research was to examine the internal representation of an FSA in an ANN called a Simple Recurrent Network (SRN) [GhoshJ92]. In contrast, the SeqSOM research makes no attempt at explaining how an FSA is represented within an ANN. Instead, the goal was to show that feedback can be used within a Kohonen network.

Generation of Test Data

Four FSAs were used to test SeqSOM, but a single FSA will be used to demonstrate the test process in this section. The other FSAs are included in Appendix C. The FSA used for this discussion was taken from Ghosh and Karamcheti and is illustrated in Figure 13 [GhoshJ92]. Ghosh and Karamcheti were able to show that a feedforward network with recurrence (i.e. feedback) could embody the behavior of an FSA if it were trained using strings from the language generated from that FSA.

Strings from the language were also used in the SeqSOM research, with the goal of showing that a SeqSOM network could be made to embody the behavior of an FSA. Therefore, strings had to be generated for training a SeqSOM network. All strings up to length ten from the language were generated; this is because the training time was more manageable with a set of strings of shorter lengths. As later discovered, the smaller set of shorter strings produced better results than [GhoshJ92] obtained using a significantly larger set of longer strings [FausettL94].

A program was written to generate the strings from the language defined by an FSA. The C++ programming language was not chosen to implement the string generator even though the rest of the project was written in C++. The advisor to this research, Dr. Blayne Mayfield, recommended using the Prolog programming language because it was more suited to generating strings from a language. Since the principle researcher for this project was not familiar with Prolog, Dr. Mayfield provided a Prolog program to generate strings. The Prolog program is presented in Appendix B.

The string generator program is written such that it does not have to be changed for each new FSA. Rather, an FSA is described as a set of Prolog lists that is used as input to the program. The lists are manually constructed and include the start state, final states, and state transition function of the FSA. Another list item specifying the maximum string length constraint is also included. It is used to obtain strings from the language up to a given length and to terminate execution of the generator program.

```
% Prolog lists describing the FSA used by Ghosh and
% Karamcheti.

%The following list contains the Maximum Length, Start State,
%and the set of Final States.

[10, state0, [state5]].

%The following lists describe the state transition function.
%Each list includes the beginning state, ending state and
%output of the transition.

[state0, state1, t].
[state0, state2, p].

[state1, state1, s].
[state1, state3, x].

[state2, state2, t].
[state2, state4, v].

[state3, state2, x].
[state3, state5, s].

[state4, state3, p].
[state4, state5, v].
```

Figure 14. Illustrates an FSA (from Figure 13) transformed to a set of Prolog lists statements which are used as input for FSA string generator program [GhoshJ92].

Figure 14 contains the Prolog lists that describes the FSA pictured in Figure 13.

Four other FSAs were also chosen for experimentation. The Prolog lists that describe these FSAs used for testing SeqSOM are listed in Appendix C.

Figure 15 contains a sample of the strings generated using the Prolog lists shown in Figure 14; the strings have a length constraint of ten characters, not including the beginning (“b”) and ending (“e”) sentinel characters. The sentinel characters are used to designate the beginning and ending points of a string and are used in this research as an attempt to retain, where applicable, the testing techniques of [GhoshJ92]. Even though the two sentinel values are represented as two separate textual characters, they are not encoded as separate values but are mapped onto a single representation. The encoding process is described below. A complete listing of all strings generated for each FSA tested can be found in Appendix C.

b t x s e	b t x x t v p s e
b t x x v v e	b t x x t v p x v v e
b t x x v p s e	b t x x t v p x v p s e
b t x x v p x v v e	b t x x t v p x t v v e
b t x x v p x v p s e	b t x x t t v v e
b t x x v p x t v v e	b t x x t t v p s e
b t x x v p x t v p s e	b t x x t t v p x v v e
b t x x v p x t t v v e	b t x x t t t v v e
b t x x t v v e	

Figure 15. Partial list of strings up to ten characters in length.

The character strings produced by the generator are not used directly as input to SeqSOM; rather they were encoded as bit strings. The length of the bit strings is equal to the alphabet size plus one. (The additional bit is used to represent the sentinel value.) The bit strings chosen to represent the characters are orthogonal to one another. This means that there is a single 1-bit in each bit string, and that the 1-bit is in a different position for

each bit string. This is illustrated in Figure 16 for the FSA shown in Figure 13. The alphabet for that FSA contains six characters, including the sentinel character.

Output Character		Bit Vector Representation
B (Begin) - Sentinel	⇒	100000
T	⇒	000001
P	⇒	000010
X	⇒	000100
V	⇒	001000
S	⇒	010000
E (End) - Sentinel	⇒	100000

Figure 16. Bit vector assignment for the FSA.

A separate “quick-and-dirty” C program was written to encode the strings for each FSA. To illustrate the encoding concept, the bit strings that correspond to the character strings shown in Figure 13 are given in Figure 17. Once the strings are encoded and saved to a file they are ready for use by the SeqSOM program. The training set for Figure 13, consisted of 103 vectors.

Training a SeqSOM Network

Chapter III contains a complete description of the SeqSOM algorithm and its training process. Once a user-specified training session is complete, the weights are saved to a file. The weights embody the knowledge of the network. These weights can later be loaded into an untrained SeqSOM network to recreate the network in its trained state.

```

10000000000010001000100001000000 btxse
100000000000100010000010000100000010001000000 btxxvve
100000000000100010000010000100000000100100001000000 btxxvpse
1000000000001000100000100001000000001000010000010001000000 btxxvpvxvve
100000000000100010000010000100000000100001000001000000001001000001000000 btxxvpvxvpse
1000000000001000100000100001000000001000010000000010010000010001000000 btxxvpxtvve
100000000000100010000010000100000000100001000000001001000000001001000001000000 btxxvpxtvpse
1000000000001000100000100001000000001000010000000010010000000010010000010001000000 btxxvpxtvve
100000000000100010000010000000010010000010001000000 btxxtvve
1000000000001000100000100000000100100000000100100001000000 btxxtvpse
1000000000001000100000100000000100100000000100001000010000010001000000 btxxtvpvxvve
100000000000100010000010000000010010000000010000100001000000001001000001000000 btxxtvpvxvpse
10000000000010001000001000000001001000000001000010000000010010000010001000000 btxxtvpxtvve
100000000000100010000010000000010000000010000010010000010001000000 btxxtvve
1000000000001000100000100000000100000000100000100100000000100100001000000 btxxtvpse
10000000000010001000001000000001000000001000001001000000001000010000010001000000 btxxtvpvxvve
1000000000001000100000100000000100000000100000100100000100010000010001000000 btxxtvve

```

Figure 17. Sample of the encoded vectors for Figure 15.

Once a SOM (including SeqSOM) is trained, the training set is reused to create a map of the behavior of the network. In the case of SeqSOM, the map of behavior is transformed into a new FSA that may be equivalent to the FSA that generated the training set.

Building an FSA from a Trained SeqSOM Network

The training set is used to map the behavior of a trained SeqSOM network to an FSA that recognizes the strings in the training set. As the characters of each string are processed by the network, the sequence of winning neurodes corresponding to those characters is collected. The winning neurode sequences and their associated strings are written to a file for further processing by yet another program. Figure 18 shows the lines written to the file for the strings shown in Figure 15. Each winning neurode in a sequence is treated as a state in the new FSA. Transition from one winning neurode to the next in the sequence results from processing a particular character in the string. Thus, transition from one state to the next in the new FSA results from processing the same character.

Note in Figure 18 that each winning neurode is referenced by a single number even though the network of neurodes is viewed in SeqSOM as a three dimensional matrix, as described in Chapter III. The reason for this is that it is easier to dynamically allocate and manipulate a 1-dimensional array in C++ rather than a 3-dimensional array. The C++ programs in SeqSOM are written to transform the 1-dimensional array addresses to 3-dimensional neurode addresses, and vice versa. The 1-dimensional array address does not affect the results produced by SeqSOM.

The next step is to convert the file illustrated in Figure 18 into a set of Prolog facts that is used by the FSA string generator program. The program that performs this conversion is called BUILDTABLE.EXE. The output of BUILDTABLE.EXE is a file

```
0 224 129 212 4 btxsc
0 224 129 174 150 140 66 btxxvve
0 224 129 174 150 12 210 4 btxxvpse
0 224 129 174 150 12 204 150 140 66 btxxvp xvve
0 224 129 174 150 12 204 150 12 210 4 btxxvp xvvpse
0 224 129 174 150 12 204 70 60 140 66 btxxvp xtvve
0 224 129 174 150 12 204 70 60 11 210 4 btxxvp xtvvpse
0 224 129 174 150 12 204 70 119 105 140 66 btxxvp xttvve
0 224 129 174 70 60 140 66 btxx tvve
0 224 129 174 70 60 11 210 4 btxx tvvpse
0 224 129 174 70 60 11 204 150 140 66 btxx tvp xvve
0 224 129 174 70 60 11 204 150 12 210 4 btxx tvp xvvpse
0 224 129 174 70 60 11 204 70 60 140 66 btxx tvp xtvve
0 224 129 174 70 119 105 140 66 btxx ttvve
0 224 129 174 70 119 105 12 210 4 btxx ttvpse
0 224 129 174 70 119 105 12 204 150 140 66 btxx tvp xvve
0 224 129 174 70 119 74 90 140 66 btxx tt tvve
```

Figure 18. Sample output from SeqSOM using the training set as input.

that consists of a single Prolog fact on each line. The output produced by BUILDTABLE.EXE for the data in Figure 18 is shown in Figure 19.

Duplicate transitions may occur, but this is expected since many strings share common sub-strings. Next, the UNIX sort command is used to remove duplicate Prolog lists, and this file is saved. An additional Prolog list containing the start state, ending states, and maximum string length is then added to the file; this Prolog list is needed for the string generator program. The maximum string length contained in this list matches that of the original FSA, as shown in Figure 13. Finally, all Prolog lists that contain either

the sentinel character “b” or “c” are commented out; this is done because those lists do not contribute to the string generation process. The result of applying the modifications listed above to the file shown in Figure 19 is illustrated in Figure 20. This edited file is called SeqSOM FSA file, and it will be used to compare the FSA produced by SeqSOM to the original FSA.

Comparison of FSAs

The question of equivalence arises once the second FSA is constructed from a trained SeqSOM. To compare the original FSA (M_o) to the SeqSOM FSA (M_s), the sets of strings generated by the FSAs must be compared. This is impractical for many FSAs since they have an infinite number strings in their language. However, an alternative discussed in the next section shows that a limited number of strings can be compared to show FSA equivalency. For convenience, an initial test is run in which a small subset of the two languages (L_o and L_s) are compared; the reason for this is to reduce run time and storage space. The subset chosen for the initial test uses the same string length constraint used to train SeqSOM. If the comparison shows that the two subsets are not equal then it is immediately obvious that the two machines M_o and M_s are not equivalent. But if the subsets are the same, additional comparisons must be made to determine the equivalency of M_o and M_s . The question then becomes, what is the minimum length constraint needed for a comparison to show that M_o and M_s are equivalent? The minimum length constraint and the question of equivalence between two finite state automata can be decided by using an algorithm given by Aho and Ullman as discussed in the next section [AhoA72].

[state0, state0, b].	[state129, state174, x].	[state0, state0, b].	[state174, state70, t].	[state129, state174, x].
[state0, state224, t].	[state174, state150, v].	[state0, state224, t].	[state70, state60, v].	[state174, state70, t].
[state224, state129, x].	[state150, state12, p].	[state224, state129, x].	[state60, state11, p].	[state70, state119, t].
[state129, state174, x].	[state12, state204, x].	[state129, state174, x].	[state11, state204, x].	[state119, state105, v].
[state174, state150, v].	[state204, state70, t].	[state174, state70, t].	[state204, state150, v].	[state105, state12, p].
[state150, state12, p].	[state70, state60, v].	[state70, state60, v].	[state150, state12, p].	[state12, state210, s].
[state12, state210, s].	[state60, state140, v].	[state60, state140, v].	[state12, state210, s].	[state210, state4, e].
[state210, state4, e].	[state140, state66, e].	[state140, state66, e].	[state210, state4, e].	[state0, state0, b].
[state0, state0, b].	[state0, state0, b].	[state0, state0, b].	[state0, state0, b].	[state0, state224, t].
[state0, state224, t].	[state0, state224, t].	[state0, state224, t].	[state0, state224, t].	[state224, state129, x].
[state224, state129, x].	[state224, state129, x].	[state224, state129, x].	[state224, state129, x].	[state129, state174, x].
[state129, state174, x].	[state129, state174, x].	[state129, state174, x].	[state129, state174, x].	[state174, state70, t].
[state174, state150, v].	[state174, state150, v].	[state174, state70, t].	[state174, state70, t].	[state70, state119, t].
[state150, state12, p].	[state150, state12, p].	[state70, state60, v].	[state70, state60, v].	[state119, state105, v].
[state12, state204, x].	[state12, state204, x].	[state60, state11, p].	[state60, state11, p].	[state105, state12, p].
[state204, state150, v].	[state204, state70, t].	[state11, state210, s].	[state11, state204, x].	[state12, state204, x].
[state150, state140, v].	[state70, state60, v].	[state210, state4, e].	[state204, state70, t].	[state204, state150, v].
[state140, state66, e].	[state60, state11, p].	[state0, state0, b].	[state70, state60, v].	[state150, state140, v].
[state0, state0, b].	[state11, state210, s].	[state0, state224, t].	[state60, state140, v].	[state140, state66, e].
[state0, state224, t].	[state210, state4, e].	[state224, state129, x].	[state140, state66, e].	[state0, state0, b].
[state224, state129, x].	[state0, state0, b].	[state129, state174, x].	[state0, state0, b].	[state0, state224, t].
[state129, state174, x].	[state0, state224, t].	[state174, state70, t].	[state0, state224, t].	[state224, state129, x].
[state174, state150, v].	[state224, state129, x].	[state70, state60, v].	[state224, state129, x].	[state129, state174, x].
[state150, state12, p].	[state129, state174, x].	[state60, state11, p].	[state129, state174, x].	[state174, state70, t].
[state12, state204, x].	[state174, state150, v].	[state11, state204, x].	[state174, state70, t].	[state70, state119, t].
[state204, state150, v].	[state150, state12, p].	[state204, state150, v].	[state70, state119, t].	[state119, state74, t].
[state150, state12, p].	[state12, state204, x].	[state140, state66, e].	[state119, state105, v].	[state74, state90, v].
[state12, state210, s].	[state204, state70, t].	[state0, state0, b].	[state105, state140, v].	[state90, state140, v].
[state210, state4, e].	[state70, state119, t].	[state0, state224, t].	[state140, state66, e].	[state140, state66, e].
[state0, state0, b].	[state119, state105, v].	[state224, state129, x].	[state0, state0, b].	
[state0, state224, t].	[state105, state140, v].	[state129, state174, x].	[state0, state224, t].	
[state224, state129, x].	[state140, state66, e].		[state224, state129, x].	

Figure 19. Sample output from building of new FSA.

[10, state0, [state140, state210, state212]].	[state174, state70, t].
%[state0, state0, b].	[state200, state129, x].
[state0, state224, t].	[state200, state200, s].
[state0, state8, p].	[state204, state150, v].
[state105, state12, p].	[state204, state70, t].
[state105, state140, v].	%[state210, state4, c].
[state11, state204, x].	%[state212, state4, c].
[state11, state210, s].	[state224, state129, x].
[state119, state105, v].	[state224, state200, s].
[state119, state74, t].	[state60, state11, p].
[state12, state204, x].	[state60, state140, v].
[state12, state210, s].	[state70, state119, t].
[state129, state174, x].	[state70, state60, v].
[state129, state212, s].	[state71, state119, t].
[state135, state12, p].	[state71, state60, v].
[state135, state140, v].	[state74, state89, t].
%[state140, state66, c].	[state74, state90, v].
[state150, state12, p].	[state8, state163, t].
[state150, state140, v].	[state8, state60, v].
[state163, state135, v].	[state89, state74, t].
[state163, state71, t].	[state89, state90, v].
[state174, state150, v].	[state90, state11, p].
	[state90, state140, v].

Figure 20. Prolog lists describing new FSA generated from a Trained SeqSOM.

Testing For Equivalence

The algorithm from Aho and Ullman listed in Figure 23 determines if two finite state automata are equivalent. To aid this discussion, a supporting definition and lemma—also from Aho and Ullman—are given in Figure 21 and Figure 22, respectively [AhoA72]. In the following discussion M_o (the original FSA) and M_s (the SeqSOM FSA) play the roles of M_1 and M_2 , respectively.

The application of Algorithm 3, Definition 1, and Lemma 1 to this research can be summarized as follows. The two machines M_o and M_s are combined to construct a new machine M . The alphabets for M_o and M_s are the same; i.e., $\Sigma_o \equiv \Sigma_s$. Thus, the

DEFINITION

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton, and let q_1 and q_2 be distinct states. We say that x in Σ^* *distinguishes* q_1 from q_2 if $(q_1, x) \xrightarrow{*} (q_3, e)$, $(q_2, x) \xrightarrow{*} (q_4, e)$, and exactly one of q_3 and q_4 is in F . We say that q_1 and q_2 are *k-indistinguishable*, written $q_1 \equiv_k q_2$, if and only if there is no x , with $|x| \leq k$, which distinguishes q_1 from q_2 . We say that the two states q_1 from q_2 are indistinguishable, written $q_1 \equiv q_2$, if and only if they are *k-indistinguishable* for all $k \geq 0$.

A state $q \in Q$ is said to be inaccessible if there is no input string x such that $(q_0, x) \xrightarrow{*} (q, e)$.

Figure 21. Definition of indistinguishable states [AhoA72].

LEMMA 1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. States q_1 and q_2 are indistinguishable if and only if they are $(n - 2)$ - indistinguishable.

Figure 22. Aho's and Ullman's Lemma on determining if two states in a finite automata are indistinguishable [AhoA72].

ALGORITHM 3

Input. Two finite automata $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$ such that $Q_1 \cap Q_2 = \emptyset$.

Output. "YES" if $L(M_1) = L(M_2)$, "NO" otherwise.

Method. Construct the finite automaton

$$M = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, q_1, F_1 \cup F_2).$$

Using Lemma 1 determine whether $q_1 \equiv_q q_2$. If so, say "YES"; otherwise, say "NO".

Figure 23. Aho's and Ullman's algorithm for testing equivalence between two finite automata [AhoA72].

alphabet Σ for the combined machine M is equal to Σ_o and Σ_s . The set $\Sigma^{n-2} \subseteq \Sigma^*$ of all strings of length $n-2$ or less is needed by the algorithm, where n is the number of states in the combined machine M ; i.e., $n = |Q_o \cup Q_s|$. Let q_o and q_s be the start states for M_o and M_s , respectively. According to Definition 1, if q_o is $(n-2)$ -indistinguishable from q_s in machine M if they are indistinguishable for each string in Σ^{n-2} . If the two states q_o and q_s are $(n-2)$ -indistinguishable, then by Lemma 1 they are indistinguishable. If q_o and q_s are indistinguishable, then according to Algorithm 3 M_o and M_s are equivalent.

The implication of this is that not all strings in Σ^{n-2} need to be generated; instead, only the subset of Σ^{n-2} that is recognized by M_o and the subset of Σ^{n-2} that is recognized by M_s need to be generated. If the two subsets are equal then M_o and M_s are equivalent.

Determining Equivalency of SeqSOM and Original FSA

To generate the sets of strings necessary for proving the equivalence of M_o and M_s , the string generator program is used to produce the sets of strings from the files (as described earlier) that describe the original FSA and the SeqSOM FSA. (Each FSA file specifies the same maximum string length.) The two files created by the string generator program are each sorted using the UNIX sort command and are saved as new files. The final step is comparing the two sorted files. The UNIX compare command is used to compare the two files. If there are no discrepancies between the two files, then the two FSAs are equivalent in accordance with Algorithm 3.

Discussion of the Experimental Results

There are four original finite state automata that were used for testing SeqSOM. The number of states in these FSAs ranged from 3 to 6, and their alphabet sizes range from 2 to 5 distinct characters. For each of the original FSAs, an new FSA was generated from a trained SeqSOM using a subset of strings as defined by the original FSA. Equivalence was shown in two of the test cases in accordance with Algorithm 3. However, FSA Test Case 2 and FSA Test Case 3 the number of strings necessary to show equivalence was beyond the current storage capacity of the available computing resources. FSA reduction is discussed in Chapter V as a possible means for reducing the number of strings needed to show equivalence. Table 1 lists a summary of the test results for each FSA.

Original FSA				New FSA			
	States	Strings	Length	States	Strings	Length	Equivalent
Test FSA 1	6	103	10	29	76955	27	Yes
Test FSA 2	3	510	9	37	$<2^{35}$	35	Likely
Test FSA 3	5	19	20	43	40	41	Yes
Test FSA 4	5	511	12	90	$<2^{88}$	88	Likely

Table 1. Results for SeqSOM.

Since SeqSOM can generate equivalent FSAs, it can be concluded that a SeqSOM neural network captures in its internal representation the contextual relationship of the input data.

As can be seen from Table 1, the original FSAs all have a small number of states. Once SeqSOM is trained, the number of states needed to define the new FSA is larger in

all cases. The maximum string length needed to prove equivalence using algorithm as shown in Figure 23 is defined as the $\text{Max}(|Q_o|, |Q_s|) - 2$.

Failures to produce an Equivalent FSA

Sometimes SeqSOM fails to generate an equivalent FSA. The generated FSA may produce a set of strings that is a superset of the training set. What observations can be made when SeqSOM fails to produce an equivalent FSA?

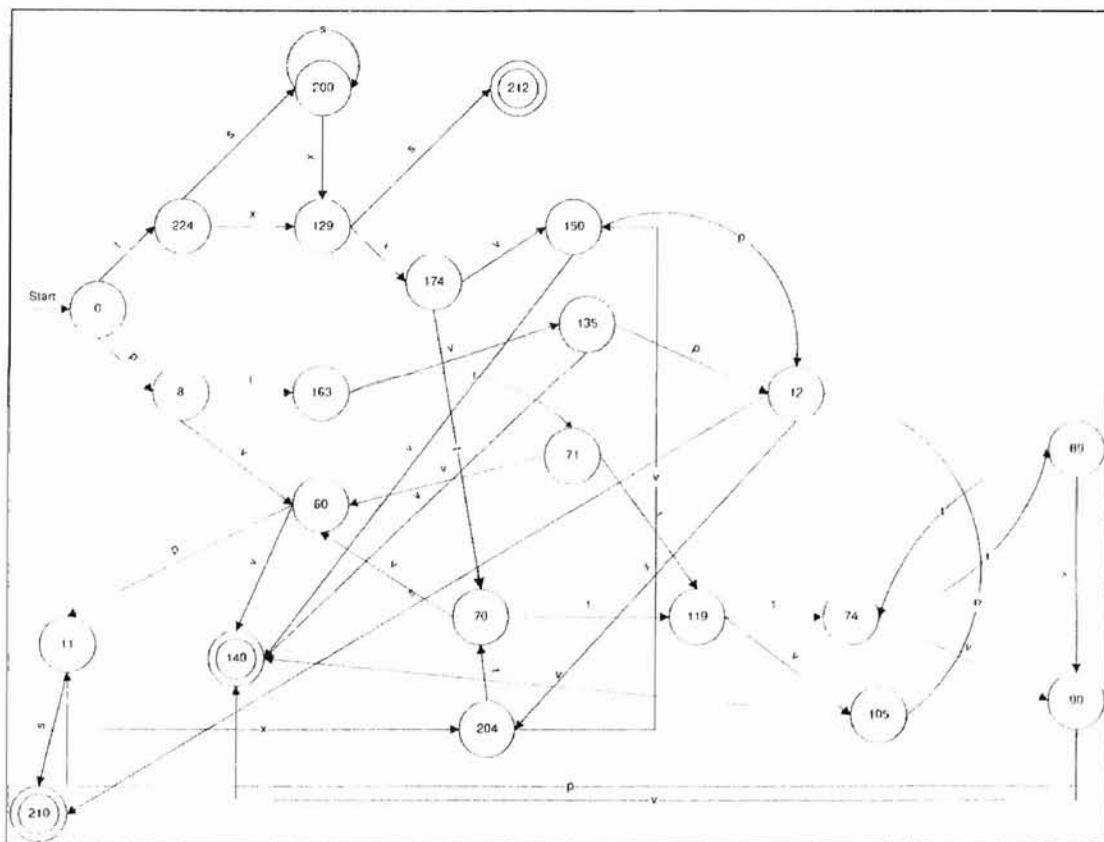


Figure 24. Equivalent FSA generated after training the SeqSOM network for FSA Test Case 1.

During experimentation with SeqSOM several observations were made about the ability of this technique to build an equivalent FSA. One such observation suggests that shrinking the neighborhood size can increase the likelihood that an equivalent FSA will be captured by SeqSOM. In Test Cases 3 and 4, shrinking the neighborhood had the effect of generating an equivalent machine. A possible explanation is that as clusters are formed with a smaller neighborhood they have sharper edges, so the states are more well-defined. In general when a SOM or SeqSOM type network is trained, a smaller neighborhood reduces the area in which clusters are formed and the subsequent transitional regions between clusters. Setting training parameters can be dubious if the original FSA is unknown since a new FSA generated by SeqSOM can be only partially verified with the training strings. The selection of training parameters and their impact on FSA generation using a SeqSOM network is left as future work and is discussed in the Conclusions and Recommendations section.

SeqSOM also will fail to generate an equivalent machine if there is an insufficient number of strings for training. Increasing the training time does not seem to alleviate this problem. This behavior might be accounted for by the fact that there are not enough strings to capture the FSA that generated them. This is evident in FSA Test Case 2. In Test Case 2, the first training set consisted all strings from the language of length seven or less, and after training the SeqSOM failed to produce an equivalent machine. When a larger subset of the language was used, the SeqSOM training session produced an equivalent FSA.

If the training time is insufficient (i.e. if the network is undertrained) then the ANN will not have had enough time to learn. SeqSOM also is prone to under-training

and will fail like other artificial neural networks. Over-training is not a problem for the SOM technique [HiotisA93]. Over-training should not be an issue with SeqSOM since the learning method used is the same as for SOM. Since over-training is not an issue, training for an extended period will help guarantee there is sufficient training to capture the desired behavior in the network.

SeqSOM as Compared to Simple Recurrent Network

The intent of this research is to show that SeqSOM is a technique for processing contextual data by using feedback. Several observations can be made when comparing SeqSOM to the Simple Recurrent Network (SRN) as reported by [FausettL94 and GhoshJ92].

The training sets used by [GhoshJ92] to train an SRN were chosen from the subset of the language with a maximum length of 32 characters; the size of this subset is greater than 355,000 strings. The size of the training sets ranged from 60,000 to 80,000 distinct strings. SeqSOM uses a complete subset for training and for showing the equivalency of the FSA representations (i.e. proof of equivalence).

The research presented by [GhoshJ92] shows that the accuracy of a trained SRN network to be less than perfect. The incomplete training set should raise the question as to whether an ANN trained with these strings will encompass the complete behavior of the original FSA. Ghosh and Karamcheti report accuracy results ranging from .48 to .999, while [FausettL94 and GhoshJ92] reports that other researchers were able to obtain perfect accuracy for the SRN technique. Less than perfect accuracy could be a problem if the ANN were being used in a control system. However with SeqSOM, 100% accuracy is

obtainable. Further, [GhoshJ92] did not show how they would produce an FSA from a trained SRN. In contrast an FSA can be produced from a trained SeqSOM network.

SeqSOM has the advantage of using a small, complete subset for training (e.g. 103 strings for the FSA in Figure 13 as compared to Ghosh and Karamcheti's partial subset of 60,000 to 80,000 strings) [GhoshJ92]. Even though run times for the two methods were not evaluated, it is possible that the reduction in the training set size for SeqSOM could be beneficial in improving performance.

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

The research presented in this thesis discusses the integration of feedback into a Self-Organizing Map to create a new Artificial Neural Network (ANN) architecture called a Sequential Self-Organizing Map (SeqSOM). The experiments performed show that SeqSOM can successfully capture the contextual nature of input data. In this research the language from an FSA was used to test the SeqSOM architecture. This type of test data had been used for another network architecture called the Simple Recurrent Network (SRN) [GhoshJ92]

Like the SRN research, a SeqSOM network was trained with strings from the language given by an FSA. Once the SeqSOM network is trained, a new FSA can be constructed from it and compared to the original. Equivalency of the two FSAs is shown by directly comparing strings generated from the language of both FSAs [AhoA72].

Observations

The results of this research show that the FSA generated from a trained SeqSOM network can successfully and completely capture the behavior of the original FSA. Four different FSAs were tested, and this result was observed in all cases. In general, SeqSOM performed with a higher level of accuracy than SRN.

One of the drawbacks with SeqSOM is that it tends to produce a large numbers of states in the generated FSAs. Because the number of states is used to determine the necessary string length for showing equivalence to the original FSA, there can be difficulty due to the storage space needed for the of strings being generated.

Future Work

This research should be pursued further to show that SeqSOM has more applications than simply building an internal representation of an FSA. SeqSOM, being a new architecture for contextual information, has raised as many questions as were answered. The main question answered in this research was that SeqSOM is a viable ANN architecture. However, much work remains as to its efficiency, behavior, and potential applications.

Efficiency Concerns. One of the limitations of using SeqSOM is the amount of time required for training. The SeqSOM programs implemented for this research recalculate all activation values during each pass through the training set. Many times these activation values do not need to be recalculated because the weight vectors used to calculate them do not change. This inefficiency could be reduced by storing the activation values for each neurode as was implemented in Kohonen's SOM application [KohonenT92]. Along with the activation values, an extra value called the dirty bit could be used to identify the weight vectors modified during the previous pass through the training set. With each pass through the training set, the activation values are recalculated only for those neurodes that have their dirty bit set. From among the new

activation values and the ones saved in prior passes through the training set, the winning neurode is chosen.

Complications of implementing the dirty bit mechanism as described above are that the feedback values in SeqSOM do not remain static even though the training vectors do and the weight vectors may. A possible solution to this problem is to calculate and store a partial activation value and finish the calculation once the feedback values are known. This could have the effect of reducing the number calculations required, thus reducing the overall training time. The dirty bit technique may not be plausible for all training sets, but in the case of the FSAs used in this research, it would have been beneficial.

With the advent of multiprocessor personal computers and operating systems that support them (e.g., Microsoft Windows NT), an obvious means for speeding up the SeqSOM architecture is the use of parallel processor coding techniques (i.e. threads). The calculation of neurode activation values is the first logical candidate for parallelization since this value is calculated independently for each neurode.

FSA Issues. Since FSAs were the main focus of testing SeqSOM in this research, there are many questions with regards to this problem domain that should be investigated. These include, but are not limited to, the following: establishing the optimal training time, learning rate, neighborhood size, network size, etc.

Even though the FSAs used for this research had small training sets, more complicated FSAs exist and may require an excessive number of strings in the training set. This could overwhelm the capacity of available computing power to train and to show equivalence. Therefore, it would be beneficial to establish the lower bounds for the

size of the training set necessary to produce an equivalent FSA. The advantage of knowing this information would be a reduction in time and other resources necessary for training.

A way of dealing with excessive numbers of states is the reduction algorithm for FSAs [AhoA72]. This algorithm could be used to reduce the number of states in the generated FSA, which should help reduce the time and other resources required to show equivalence to the original FSA.

Test with other data domains. Other “Real-World” applications should be investigated using the SeqSOM architecture. Future work should include different kinds of data that have a contextual and time dependencies. As discussed in the literature review, other SOM architectures have been modified for speech analysis, and the SeqSOM architecture could be applicable to this area.

BIBLIOGRAPHY

- [AmericanH92] *American Heritage Electronic Dictionary*. Standard Edition, Version 3.0A. Houghton Mifflin Company, 1992.
- [AhoA72] Aho, Alfred V. and Ullman, Jeffrey D. *The Theory of Parsing, Translation and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, NJ. 1972.
- [BoydstunR95] Boydston, Roger and Mayfield, Blayne E. "Investigation of Sequential Self-Organizing Maps." *Proceeding of the Ninth Mid-America Symposium on Emerging Technologies 1995, MASECT 95*.
- [CaudillM93] Caudill, Maureen. "A Little Knowledge is a Dangerous Thing" *AI Expert*. June 1993. Volume 8, Number 6. pages 16-22.
- [CotterillR88] Cotterill, Rodney. *Computer Simulation in Brain Science*. Cambridge: Cambridge University Press, 1988.
- [DayhoffJ90] Dayhoff, Judith E. *Neural Network Architectures*. New York: Van Nostrand Reinhold, 1990.
- [ElmanJ90] Elman, J. L. "Finding Structures in Time." *Cognitive Science*, 14:179-211, 1990.
- [FausettL94] Fausett, Laurene. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. 1994. Prentice Hall, Englewood Cliffs, NJ 07632.
- [GhoshJ92] Ghosh, Joydeep and Karamcheti. "Sequence Learning and Recurrent Networks: Analysis of Internal Representation." *SPIE Vol 1710 Science of Artificial Neural Networks (1992)*. pp. 449-460.
- [HiotisA93] Hiotis, André. "Inside a Self-Organizing Map." *AI Expert*. April 1993. Volume 8, Number 4. pp. 38-42.
- [KangasJ90] Kangas, Jeri. "Time-Dependent Self-Organizing Maps." *Proc. IJCNN-90-San Diego, International Joint Conference on Neural*

- Networks*. pp. 331-336. IEEE Computer Society Press, Los Alamitos, CA 1990.
- [KangasJ91a] Kangas, Jeri. "Time-Dependent Self-Organizing Maps for Speech Recognition." *Artificial Neural Networks*, Kohonen, T., Makisara, K., Simula, O. and Kangas, J. Editors. pp. 1591-1594. Elsevier Science Publisher: B.V. North-Holland. 1991.
- [KangasJ91b] Kangas, Jeri. "Phoneme Precognition Using Time-Dependent Versions of Self-Organizing Maps." *ICASSP 91: 1991 International Conference on Acoustics, Speech and Signal Processing*. IEEE: Piscataway, NJ. pp. 101-104.
- [KohonenT88a] Kohonen, Teuvo. "The 'Neural Phonetic' Typewriter." *IEEE Computer*. March 1988. Volume 21, Number 3.
- [KohonenT88b] Kohonen, Teuvo. "Self-Organizing Formation of Topologically Correct Feature Maps." *Neurocomputing: Foundations of Research*. Edited by James A. Anderson and Edward Rosenfeld. Cambridge, Massachusetts: MIT Press, 1988.
- [KohonenT89] Kohonen, Teuvo. *Self-Organization and Associative Memory*, (Third Edition). New York: Springer-Verlag, 1989.
- [KohonenT92] Kohonen, Teuvo, Kangas, Jeri, and Laaksonen, Jorma. *SOM_PAK: The Self-Organizing Map Program Package*. Version 1.2. Helsinki University of Technology. November 1992.
- [KohonenT95] Kohonen, Teuvo. *Self-Organizing Maps*. New York: Springer-Verlag, 1995.
- [LawrenceJ90] Lawrence, Jeannette. "Untangling Neural Nets." *Dr. Dobbs's Journal*. April 1990. Volume 15.
- [MarenA90] Maren, Alinna, Harston, Craig, and Par, Robert. *Handbook of Neural Computing Applications*. San Diego: Academic Press, 1990.
- [MarkowitzJ94] Markowitz, Judith. "Networks for Speech." *PC AI*. May/June 1994 Volume 8, Number 3.
- [McClellandJ86] McClelland, James, Rumelhart, David, et. al. "Psychological and Biological Models." *Explorations in Parallel Distributed Processing*. Cambridge, Massachusetts: The MIT Press, 1986.

- [McCullochW43] McCulloch, Warren S. and Pitts, Walter. "A Logical Calculus of the Ideas Immanent in Nervous Activities." *Bulletin of Mathematical Biophysics*. Volume 5. pp. 115-133.
- [RitterH91] Ritter, Hedge, Martinetz, Thomas, and Schulten, Kaus. *Neural Computation and Self-Organizing Maps: An Introduction*. Reading, Massachusetts: Addison-Wesley, 1991.
- [ZandhuisJ92] Zandhuis, J. "Storing Sequential Data in Self-Organizing Feature Maps." *Internal Report MPI-NL-TG-4/92*. Max Planck Institute for Psycholinguistics. Wundtlaan Nijmegen.

APPENDIXES

APPENDIX A

PROGRAM PARAMETERS

Processing Mode		Description		
-t	Training	Places SeqSOM in a training mode.		
-a	Apply	Places SeqSOM in Application mode.		
-l	Label	Label SOM neurodes with a text label		
Additional Flags		Modes	Description	
-v	Version	t a	Print current version of SeqSOM executable.	
-r	Random	t	Use Random	
-fvtr	Vector File	t a l	Filename for retrieval of input vectors.	
-fwts	Weight File	t a l	Filename for storage or retrieval of weight vectors.	
-fprm	Parameter Files	t a l	Filename for parameter file.	
-fxls	Coma Delimited	a	Filename for output of coma delimited file.	
-fout	Output File	a	Filename for other output	
-d	Display Training Grid	t	Currently Not Implemented	
-z	Random	t	Randomize weights using the time of day as seed to random number generator.	
-n	Normalization	t a	Normalize weight and vectors. (See code for current implementation of normalization routines.	
-?	Help	t a l	Print help on command line options.	

APPENDIX B
STRING GENERATOR PROGRAM

```

%== Build a Finite State Generator =====
%
% Last updated 07/19/95 09:30 by Blayne Mayfield
%
% Input file format:
% * Line 1 specifies maximum string length (not including b and e),
%   the start state name, and a list of the final state names. Format:
%   [MaxStringLength, StartStateName, [FinalStateName, ...]].
% * Line 2-n specify transition and output information. Format:
%   [FromThisState, ToThisState, ProducingThisOutput].
%=====

buildFSG(InFile, OutFile) :-
    % Open the input file.
    see(InFile),
    % Read the max string length, start state, and list of final states.
    read([MaxLength, StartState, FinalStates]),
    % assert the necessary facts relating to that input.
    assert(maxLength(MaxLength)),
    % read the first state/state/output line.
    read([State, State2, Output]),
    % Assert the rules that define the machine.
    build2(State, State2, Output),
    % Close the input file.
    seen,
    % Generate a stopping rule for each final state.
    final(FinalStates), !,
    % Generate the rule that stops execution properly.
    Head =.. [StartState, _],
    assert(Head),
    % Open the output file.
    tell(OutFile),
    % Run the program to generate the output.
    CallForm =.. [StartState, [b]],
    call(CallForm),
    % Close the output file.
    told.

build2(State1, State2, Output) :-
    % Assert a rule for this input.
    Head =.. [State1, List],
    RecursiveCall =.. [State2, [Output | List]],
    assert((Head :-
        maxLength(ML), length(List, L), L <= ML,
        RecursiveCall)),
    % read the next state/state/output line.
    read([State1a, State2a, Outputa]),
    % Continue to assert the rules that define the machine.
    build2(State1a, State2a, Outputa).

build2(_,_,_).

final([]).
final([State|Tail]) :-
    Head =.. [State, List],
    assert((Head :- printList([e|List]), nl, fail)),
    final(Tail).

printList([]).
printList([H|T]) :-
    printList(T),
    write(H), write(' ').

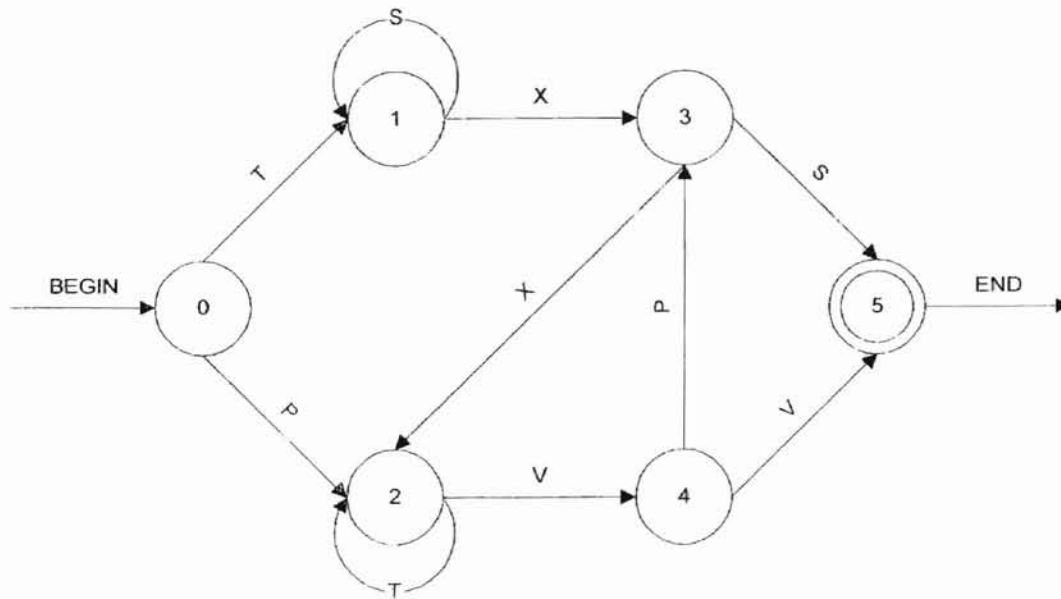
```

APPENDIX C

FINITE STATE AUTOMATA

USED FOR TESTING THE SEQSOM APPLICATION

FSA Test Case 1



%Prolog Lists for FSA Test Case 1
[10, state0, [state5]].

[state0, state1, t].
[state0, state2, p].

[state1, state1, s].
[state1, state3, x].

[state2, state2, t].
[state2, state4, v].

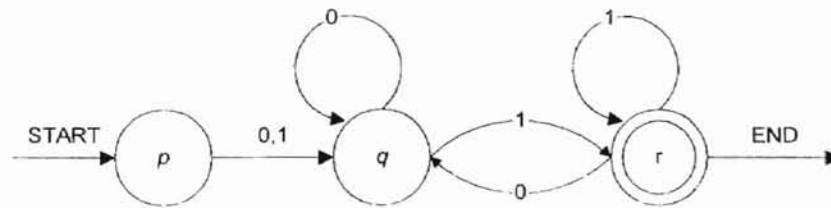
[state3, state2, x].
[state3, state5, s].

[state4, state3, p].
[state4, state5, v].

Complete Training Set for Test Case 1

bptttttttvve	btsxxxxvpse	btxxvve
bpttttttvpse	btsxxxxvve	
bptttttttvve	btsxxxse	
bpttttttvpse	btsxxxxtlvve	
bptttttlvve	btsxxxxtvpse	
bptttttvpse	btsxxxxtlvve	
bpttttvp xvve	btsxxxvpse	
bpttttlvve	btsxxxvve	
bpttttvpse	btsxxse	
bptttvp xtlvve	btsxxxxttlvve	
bptttvp xvvpse	btsxxxxtvpse	
bptttvp xvve	btsxxxxtlvve	
bptttlvve	btsxxxxtvpse	
bptttvpse	btsxxxxtlvve	
bpttvp xtlvve	btsxxxvpse	
bpttvp xtlvpse	btsxxxvp xvve	
bpttvp xtlvve	btsxxxvve	
bpttvp xvvpse	btsxxse	
bpttvp xvve	btsxxxxttlvve	
bptlvve	btsxxxxtvpse	
bptvpse	btsxxxxtlvve	
bptvp xtltlvve	btsxxxxtvpse	
bptvp xtlvpse	btsxxxxtlvve	
bptvp xtlvve	btsxxxvpse	
bptvp xtlvpse	btsxxxvp xtlvve	
bptvp xtlvve	btsxxxvp xvvpse	
bptvp xvvpse	btsxxxvp xvve	
bptvp xvve	btsxxxvve	
bptvve	btxxse	
bpvpse	btxxxxttlvve	
bpvp xtltlvve	btxxxxtvpse	
bpvp xtltlvve	btxxxxtlvve	
bpvp xtlvpse	btxxxxtvpse	
bpvp xtlvve	btxxxxtlvve	
bpvp xtlvpse	btxxxxtvpse	
bpvp xtlvp xvve	btxxxxtvpse	
bpvp xtlvve	btxxxxtvp xvve	
bpvp xvvpse	btxxxxtlvve	
bpvp xvvp xtlvve	btxxxvpse	
bpvp xvvp xvvpse	btxxxvp xtlvve	
bpvp xvvp xvve	btxxxvp xtlvpse	
bpvp xvve	btxxxvp xtlvve	
bpvve	btxxxvp xvvpse	
btsxxxxxxxxse	btxxxvp xvve	
btsxxxxxxxxse		
btsxxxxxxxxse		
btsxxxxxxxxvve		
btsxxxxse		
btsxxxxtlvve		

FSA Test Case 2



%Prolog Lists for FSA Test Case 2
[9, stateP, [stateR]].

[stateP, stateQ, 0].
[stateP, stateQ, 1].

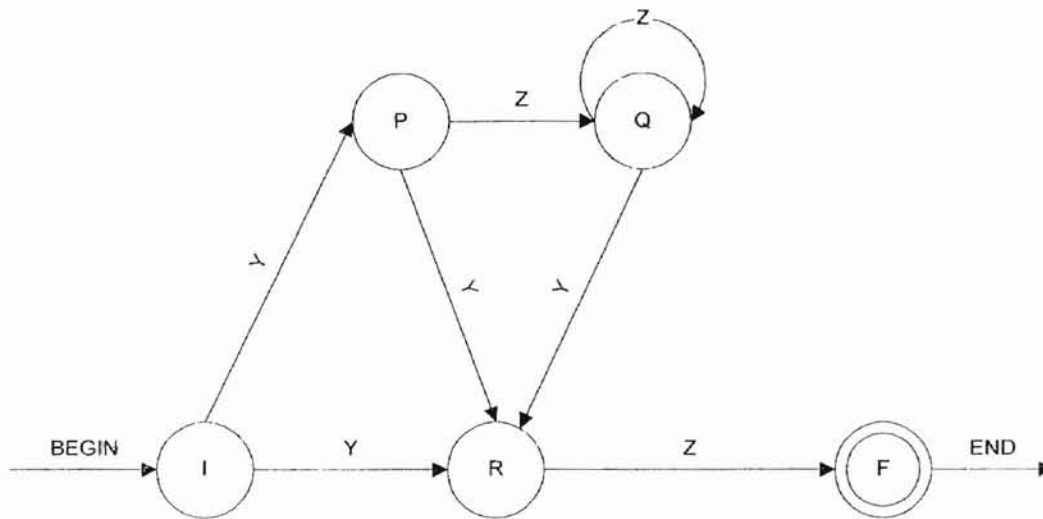
[stateQ, stateQ, 0].
[stateQ, stateR, 1].

[stateR, stateQ, 0].
[stateR, stateR, 1].

Partial Training Set for Test Case 2

b000000001e	b000110111e	b0011001e
b000000011e	b00011011e	b001101001e
b00000001e	b0001101e	b001101011e
b000000101e	b000111001e	b00110101e
b000000111e	b000111011e	b001101101e
b00000011e	b00011101e	b001101111e
b0000001e	b000111101e	b00110111e
b000001001e	b000111111e	b0011011e
b000001011e	b00011111e	b001101e
b00000101e	b0001111e	b001110001e
b000001101e	b000111e	b001110011e
b000001111e	b00011e	b00111001e
b00000111e	b0001e	b001110101e
b0000011e	b001000001e	b001110111e
b000001e	b001000011e	b00111011e
b000010001e	b00100001e	b0011101e
b000010011e	b001000101e	b001111001e
b00001001e	b001000111e	b001111011e
b000010101e	b00100011e	b00111101e
b000010111e	b0010001e	b001111101e
b00001011e	b001001001e	b001111111e
b0000101e	b001001011e	b00111111e
b000011001e	b00100101e	b0011111e
b000011011e	b001001101e	b001111e
b00001101e	b001001111e	b00111e
b000011101e	b00100111e	b0011e
b000011111e	b0010011e	b001e
b00001111e	b001001e	b010000001e
b0000111e	b001010001e	b010000011e
b000011e	b001010011e	b01000001e
b00001e	b00101001e	b010000101e
b000100001e	b001010101e	b010000111e
b000100011e	b001010111e	b01000011e
b00010001e	b00101011e	b0100001e
b000100101e	b0010101e	b010001001e
b000100111e	b001011001e	b010001011e
b00010011e	b001011011e	b01000101e
b0001001e	b00101101e	b010001101e
b000101001e	b001011101e	b010001111e
b000101011e	b001011111e	b01000111e
b00010101e	b00101111e	b0100011e
b000101101e	b0010111e	b010001e
b000101111e	b001011e	b010010001e
b00010111e	b00101e	b010010011e
b0001011e	b0010e	b01001001e
b000101e	b001100001e	b010010101e
b000110001e	b001100011e	b010010111e
b000110011e	b00110001e	
b00011001e	b001100101e	
b000110101e	b00110011e	

FSA Test Case 3



%Prolog Lists for FSA Test Case 3
 [50, stateI, [stateF]].

[stateI, stateP, y].
 [stateI, stateR, Y].

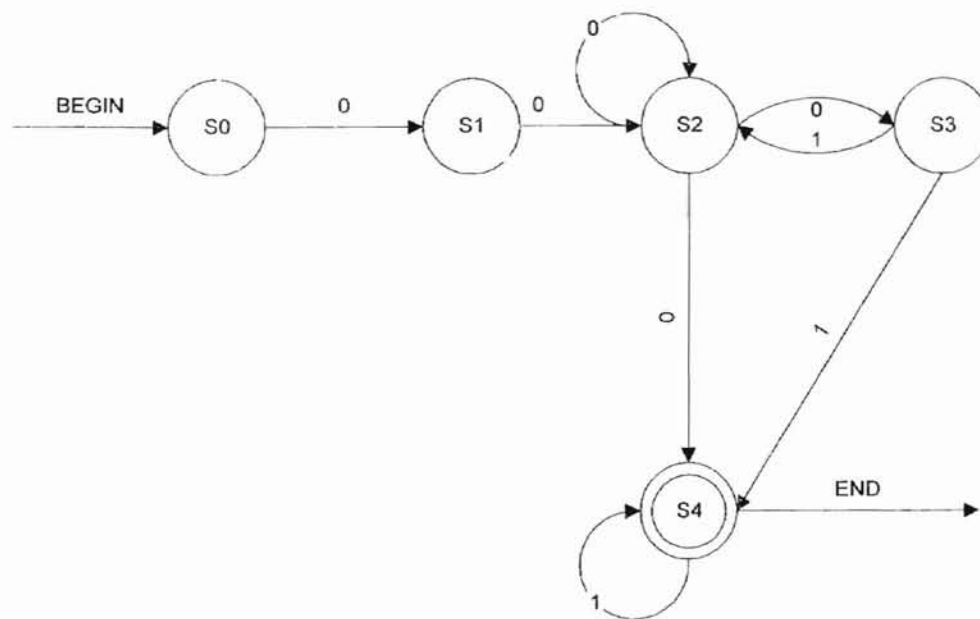
[stateP, stateQ, z].
 [stateP, stateR, y].

[stateQ, stateQ, z].
 [stateQ, stateR, y].
 [stateR, stateF, z]

Complete Training Set for Test Case 3

byzzzzzzzzzzzzzzzzzzzze
byzzzzzzzzzzzzzzzzzzzze
byzzzzzzzzzzzzzzzzzzzze
byzzzzzzzzzzzzzzzzzze
byzzzzzzzzzzzzzzzzzze
byzzzzzzzzzzzzzzzze
byzzzzzzzzzzzzzzzze
byzzzzzzzzzzzzzze
byzzzzzzzzzzzze
byzzzzzzzzzze
byzzzzzzzze
byzzzzzze
byzzzze
byzze
byze

FSA Test Case 4



%Prolog Lists for FSA Test Case 4
[10, state0, [state4]].

[state0, state1, 0].

[state1, state2, 0].

[state2, state2, 0].

[state2, state3, 1].

[state3, state2, 0].

[state3, state4, 1].

[state4, state2, 0].

[state4, state4, 1].

Partial Training Set for Test Case 4

b0000000000011e	b00000110111e	b000011010111e
b0000000000111e	b0000011011e	b00001101011e
b000000000011e	b000001110011e	b000011011011e
b0000000001011e	b000001110111e	b000011011111e
b0000000001111e	b00000111011e	b00001101111e
b000000000111e	b000001111011e	b0000110111e
b00000000011e	b000001111111e	b000011011e
b0000000010011e	b00000111111e	b000011100011e
b0000000010111e	b0000011111e	b000011100111e
b000000001011e	b000001111e	b00001110011e
b0000000011011e	b00000111e	b000011101011e
b0000000011111e	b0000011e	b000011101111e
b000000001111e	b000010000011e	b00001110111e
b00000000111e	b000010000111e	b0000111011e
b0000000011e	b00001000011e	b000011110011e
b0000000100011e	b000010001011e	b000011110111e
b0000000100111e	b000010001111e	b00001111011e
b000000010011e	b00001000111e	b000011111011e
b0000000101011e	b0000100011e	b000011111111e
b0000000101111e	b000010010011e	b00001111111e
b000000010111e	b000010010111e	b0000111111e
b00000001011e	b00001001011e	b000011111e
b0000000110011e	b000010011011e	b00001111e
b0000000110111e	b000010011111e	b0000111e
b000000011011e	b00001001111e	b000011e
b0000000111011e	b0000100111e	b000100000011e
b0000000111111e	b000010011e	b000100000111e
b000000011111e	b000010100011e	b00010000011e
b00000001111e	b000010100111e	b000100001011e
b0000000111e	b00001010011e	b000100001111e
b0000000111e	b000010101011e	b00010000111e
b000000011e	b000010101111e	b0001000011e
b000001000011e	b00001010111e	b000100010011e
b000001000111e	b0000101011e	b000100010111e
b00000100011e	b0000101011e	b00010001011e
b00000100011e	b000010110011e	b000100011111e
b000001001011e	b000010110111e	b00010001111e
b000001001111e	b000010111111e	b0001000111e
b00000100111e	b0000101111e	b000100011e
b0000010011e	b000010111e	b000100100011e
b000001010011e	b000010111e	b000100100111e
b000001010111e	b000010111e	b00010010011e
b00000101011e	b00001011e	b00010010011e
b000001011011e	b000011000011e	b000100101011e
b000001011111e	b000011000111e	b000100101111e
b00000101111e	b00001100011e	b00010010111e
b000001011e	b00001100011e	b0001001011e
b000001100011e	b000011001011e	b0001001011e
b000001100111e	b000011001111e	b000100110011e
b00000110011e	b0000110011e	
b000001101011e	b0000110011e	
b000001101111e	b000011010011e	

VITA

Roger Lee Boydstun

Candidate for Degree of

Master of Science

Thesis: INVESTIGATION OF SEQUENTIAL SELF-ORGANIZING MAPS

Major Field: Computer Science

Biographical:

Personal Data: Born in Chandler, Oklahoma, August 21, 1964, the son of Donnel L. and Ellen M. Boydstun.

Education: Graduated from Perkins-Tryon High School, Perkins, Oklahoma, in May 1982; Receive Associate Degree in Electrical Engineering Technology at Oklahoma State University in May 1986. Received Bachelor Degree in Electrical Engineering Technology at Oklahoma State University in December 1987; completed requirements for the Master of Science degree at Oklahoma State University in May 1997.

Professional Experience: President & CEO, Comprehensive Technology Group, Inc. P.O. Box 196, Perkins, Oklahoma, August 1993 to Present.
Software Engineer, Nomadics, Inc. 1730 Cimarron Plaza, Stillwater, 74075, April 1996 to Present.
Graduate Student Programmer, Office of Student Financial Aid, Oklahoma State University. 326 Hanner, Stillwater, Oklahoma, October 1989 to April 1996.