THE DISCRETE LEAKY INTEGRATE-AND-FIRE NEURON MODEL APPLIED TO VISUAL TRACKING AND PATTERN RECOGNITION

THESIS

Presented to the Graduate Council
of Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Lon W. Risinger, B.S.

San Marcos, Texas
December 2004

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to Dr. Khosrow Kaikhah. His encouragement and advice were invaluable and without which I could not have accomplished this. I would also like to thank my friends Jason High and Matt Stone for their enthusiasm and belief in my abilities. Wilbon Davis also deserves my gratitude for his wonderful mind opening suggestions and discussion in the area of computer vision.

My thanks and gratitude also go to Dr. Thomas McCabe, and Dr. Carol Hazlewood for their patience, understanding and advice.

This manuscript was submitted on December 8, 2004.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

ABSTRACT

THE DISCRETE LEAKY INTEGRATE-AND-FIRE NEURON MODEL APPLIED TO VISUAL TRACKING AND PATTERN RECOGNITION

by

Lon William Risinger, B.S., M.S.

Texas State University-San Marcos

December 2004

SUPERVISING PROFESSOR: KHOSROW KAIKHAH

The Discrete Leaky Integrate-and-Fire (DLIF) neuron uses a simple discrete LIF neuron model that is capable of diverse spatio-temporal behavior. We explore the behavior of the DLIF when driven by periodic (coherent) and constant (incoherent) input. Results show that the DLIF is capable of oscillatory behavior, amplitude to phase conversion, holographic paging, and spike coincidence detection. Exploiting temporal aspect of the DLIF neuron, a network of DLIF neurons is constructed which is capable of motion

detection, object tracking i.e. pursuit motion, and behavior similar to micro-saccadic behavior exhibited by humans. Additionally, a network of DLIF neurons is constructed which is capable of pattern recognition utilizing an action potential computation technique based on relative firing times of neurons. A novel learning technique is presented which allows selective hebbian learning in time delayed connections in a feedforward network by manipulating the DLIF leak rate during training.

CHAPTER 1 - INTRODUCTION

Artificial Neural Networks have been an exciting prospect since their first inception in 1943 when Warren McCulloch a neurophysiologist, and a young mathematician, Walter Pitts, modeled a simple biological neural network with electrical circuits. They provide a means to directly implement the sort computational abilities found in nature. While the sort of intelligent behavior depicted in science fiction has thus far eluded us, artificial neural networks (ANN) have proven to be extremely useful in solving certain types of problems. ANN's are now being more readily accepted in commercial industry accomplishing tasks such varied tasks as removing noise from phone and satellite transmissions to facial recognition, hand writing recognition, diagnosis of hepatitus and categorizing online shoppers.

For all their success uses ANN's actually provide a crude model of biological networks. Systems in nature are vastly more complex than those found in ANNs. The neuron itself far exceeds the capability and function of artificial neuron models and yet we are able to accomplish a great deal with them. Perhaps if we expand the basic model to accommodate a bit more of the complexity of real neurons and real neural networks, the capability of the ANN would also be expanded.

A particular physical aspect that is largely ignored in ANNs is signal propagation time. In ANNs in general propagation time across a connection is considered instantaneous. This is not so in nature. In real neurons signals propagate along the cellular membranes at a much slower rate than electrical signals travel along wires in circuits. This is because these signals are not truly electric, but rather electro-chemical and depend on the diffusion of ions through the cellular membrane. Signals traveling along two different pathways in a biological network to the same location may have a substantial time delay between their arrivals. By analogy an electric signal traveling along a conductor in a circuit travels at such a high velocity that the difference in arrival time of two signals traveling along different paths really is negligible.

Given the likely hood of substantial time delays between propagation paths, arrival time of signals becomes very important. In addition biological neurons are not perfect accumulators. Some of the signal they receive from their various inputs leaks away with time. This makes the arrival time of signals even more significant. Interestingly this leak rate is not static and can be mediated by the network on an individual neuron basis, or even in a particular region of a single neuron.

There is strong evidence that this apparent weakness of biological neural networks is not a weakness at all, but that this time propagation latency is actually used to encode information. It has long been held that the average firing rate of neurons plays a role in

neural computation and more recently it has been suggested that relative timing of individual signals, also known as spikes, play a role (Hopfield 1998).

Connections are not guaranteed to carry the signal from start to end. There is a relatively low probability (30% in the human neo-cortex) that the connection will transmit the signal. Furthermore this probability is not fixed. It depends on the history of the individual connection.

In order to incorporate this propagation latency we adopt a new model for the neurons in an ANN called the discrete leaky-integrate-and-fire neuron. Biophysicists often compare neuronal membrane properties to the properties of a resistor-capacitor (RC) circuit. They describe these properties in terms of various models described by continuous equations involving current, resistance, capacitance, potential difference and time. We abandon this complexity in favor of a simple discrete model centered around potential difference and time. Connections in this model use a time delays that represents propagation latency and each neuron incorporates a leak rate.

We look at behaviors of this model under input of various structures and apply DLIF networks to solve two problems: pattern recognition, and visual object tracking. Both of these networks encode information in the relative timing of individual spikes.

The problem of visual object tracking is apparently very rarely solved with an ANN. Solutions to this problem usually involve more conventional approaches. The solution presented here provides interestingly similar behavior to biological visual systems. The ANN inherently responds in three different ways which are similar to involuntary responses in humans. One of these is pursuit tracking. This is when the eye smoothly tracks a moving object. Onset is involuntary. Another of these is when an object moves inside the field of view of the eye. The eye is involuntarily drawn to this new point of interest. The last of these behaviors is the tendency of the eye to make small unperceived movements on and around an object being fixated upon.

Pattern recognition uses a method suggested by Hopfield called action potential computation. A novel learning algorithm is described for a general feed forward pattern recognizer.

CHAPTER 2 - RELATED WORK

The DLIF neuron formerly known as the Modified Bifurcating Neuron (Risinger and Kaikhah 2004) was conceptually inspired by the Bifurcating Neuron (BN) (Lee and Farhat 2002), which is a neuron model in which an integrate-and-fire neuron is augmented by coherent modulation from the neural environment. The BN is capable of amplitude to phase conversion and volume-holographic memory. Because of its integrate-and-fire activation model, it exhibits frequency response to incoming pulse timing. When used in a network, BNs have time delays between neuron connections that represent signal propagation latency (Lee and Farhat 2002). A single BN is defined by the following three equations:

$$\theta_t(t) = 1 \qquad \textbf{(Equation 2-1)}$$

$$\frac{dx(t)_t}{dt} = c_t \qquad \textbf{(Equation 2-2)}$$

$$\rho_t(t) = \rho_0 \sin 2\pi ft \qquad \textbf{(Equation 2-3)}$$

where $\theta_t(t)$, and $\rho(t)_t$ are the threshold level, and the relaxation level of BN_t, respectively. The potential $x_t(t)$ rises at a constant rate, c_t, due to the incoherent signal, until it reaches the threshold level θ_t. Then the internal potential immediately drops to

the relaxation level $\rho(t)_i$. The threshold level is constant. The relaxation level, Equation 2-3 is driven by the coherent signal and maintains a sinusoidal oscillation with maximum amplitude ρ_0 and frequency f. See Figure 2-1 (Lee and Farhat 2002).

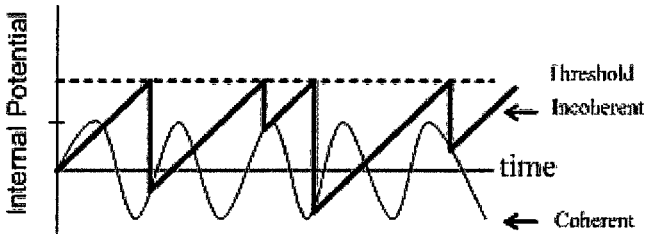

Figure 2-1 Firing Behavior of BN

An analog associative memory was constructed using BNN2. This is a one layer network consisting of an L x L planar region of BNs. A BN was connected to its neighbor if the neighbor was within a hemming distance. Each BN received direct external input, where the input value was supplied by a pixel value obtained from an L x L image raster. In a network of this kind an attractor can be created where the network output maintains a phase shift pattern proportional to the input pattern. Direct external input from the image raster creates a pattern of phase shifts in the BNN2 where each BN firing time is phase shifted with regard to its normal firing time. When external input is removed the BNN2 maintains this phase shift pattern. It maintains this pattern by ensuring that connections between BNs have time delays and weights such that a given BN receives input from other BNs at the time it should fire, when driven by external input thus driving it to a threshold event at precisely the right time. Likewise, its output arrives as the input to

other BNs at precisely the time they should fire thus sustaining the output pattern. This locking of the input pattern is considered an analog associate memory.

Holographic paging was achieved in analog associative memory by exploiting the frequency of the coherent modulation in Equation 2-3. Changing the frequency of the resting potential oscillation, leads to changes in firing frequency and phase shifts. By adding additional connections that correspond to the proper weights and time delays to sustain a different pattern in the analog associative memory under the new oscillation frequency, they were able to link the stored pattern to the oscillation frequency. This linking means that a pattern could only be recalled by the appropriate input if the oscillation frequency was correct. In addition, using this technique a single input could be linked to several different stored patterns depending on the oscillation frequency thus allowing recall based on oscillation frequency.

The BNN2 is defined as a neuron in a noisy neural environment. The rise in potential due to incoherent input i.e. environmental noise, and possibly the oscillation of the resting potential model external environmental influence. If oscillation and incoherent signal are not random noise, but a direct influence of other neural activity such as the previous pattern input into the network, this leads us to consider associations between neural input patterns. The DLIF model will not consider this resting potential oscillation

to be an inherent, always present part of the neuron but rather an external structured input.

In a related work Rizzuto and Kahan explore paired-associate learning in a recurrent network (Rizzuto and Kahan 2001). They propose that this models behavior which more closely fits experimental data collected from human test subjects than other models. Two types of association are addressed:

1. Forward association. $A \rightarrow B$
2. Backward association. $B \rightarrow A$

Normally storing an auto-association allows pattern completion and hetero-association enables recall of a pattern given its paired pattern (B given A). However, they contend that in an attractor network model, that by storing a composite pattern made by summing A and B $(\mathbf{a} + \mathbf{b})$ or by concatenating A and B $(\mathbf{a} \oplus \mathbf{b})$, an auto-associative network can accomplish hetero-association.

The auto-associative model assumes symmetry between forward and backward association and can be represented using one weight matrix while the asymmetric or hetero-associative case needs two Matrices, one for the forward association weights and one for the backward association weights. Support for the symmetric case comes from

experimentation. According to the authors the majority of human paired association experiments, especially with highly imaginable pairs, show that associative learning is symmetric. In other words, it is just as easy to remember A given B as it is B given A. There are some cases where this symmetry is not seen. For example, in free recall exercises where subjects are asked to remember a number of items in no particular order, forward association is almost always the preferred over backward association.

$$W = \sum_{v=1}^{L} (a^v \oplus b^v)(a^v \oplus b^v)^T \quad \textbf{Equation 2-4}$$

The auto-associative network developed in this paper is given by Equation 2-4, where a^v and b^v are binary, N-element vectors representing the items to be associated, $(a^v \oplus b^v)$ denotes the concatenation of a^v and b^v, and W is the 2N x 2N weight matrix (Rizzuto and Kahan 2001). This produces a matrix consisting of four quadrants. Quadrants 1 and 3 contain associative information and quadrants 2 and 4 contain hetero-associative information.

This network was used in an attempt to fit human experimental data from a simple memory association study and was primarily successful. The results show that a high correlation between forward and backward associations closely models human behavior. "These results lend support to models that employ inherently symmetric associative mechanisms, like the holographic models of Murdock and Metcalf."

An interesting use of leaky integrate-and-fire neurons appears in "Robust Sound Onset Detection using Leaky Integrate-and-Fire Neurons with Depressing synapses" (Smith and Frazer 2004). Smith first divides the sound into frequency bands using a band pass filter. He then generates input spikes based on the energy level (measured in decibels) of the sound in that frequency, where each frequency channel is divided into several energy bands of increasing value. When a sound of a certain frequency causes a spike generation in an energy band all energy bands of that frequency below the energy value also produce a spike. These spikes cross an ingenious synapse known as a depressing synapse, in which the first spike to cross receives maximum weighting and subsequent spikes receive diminished weighting until the end of the sound at which time the synapse recovers. This is accomplished by modeling synaptic release and uptake of neuro-transmitters into and from the synaptic cleft. Spikes that cross the synapse contribute to single compartment leaky integrate-and-fire (LIF) neurons described by the first order differential equation (Equation 2-5)

$$\frac{dV}{dt} = -\frac{V}{\tau} + I(t) \qquad \textbf{Equation 2-5}$$

where V is the voltage-like state variable of the neurons, τ , is the membrane time constant, and *I(t)* is the external driving input. LIF neurons produce an output when their potential reach a threshold value theta, θ. LIF neurons leak potential so they require large numbers of simultaneous inputs, meaning that either a sound with high energy in a single frequency band or low energy in multiple frequency bands can elicit a response.

CHAPTER 3 - DISCRETE LEAKY INTEGRATE-AND-FIRE (DLIF) NEURONS

- 3.1 Introduction to Neural Networks
- 3.2 DLIF Basic Definition
- 3.3 Biological Foundation
    - 3.3.1 Synaptic Plasticity
    - 3.3.2 Neural Plasticity
- 3.4 Action Potential Timing
- 3.5 Behavior Under Coherent Modulation
    - 3.5.3.1 Amplitude to Phase Conversion
    - 3.5.3.2 Holographic Paging

3.1 Introduction to Neural Networks

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological neural systems, such as the brain, process information. The

key element of this paradigm is the structure of the information processing system. It is composed of a large number of highly interconnected processing elements working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well.

ANNs are represented as directed graphs where each node is a processing element synonymous with a neuron and each edge of the graph connecting pairs of nodes is synonymous with a synaptic connection. Each connection maintains a weight i.e. connection strength parameter. All elements come in at the same time or remain activated at the same level long enough for computation of the activation function, (Figure 3-1)

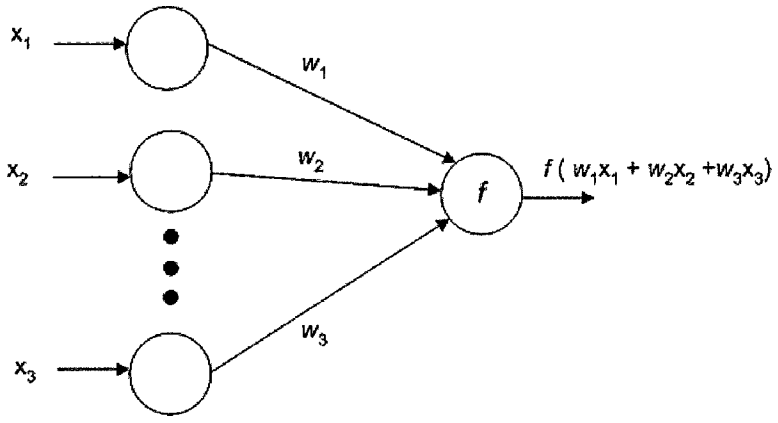

Figure 3-1 General Artificial Neuron Model

The neuron sums up the product of the input and the weight for each incoming connection. The total input is mapped to an activation value by an activation function such as sigmoid.

$$f(y) = \frac{1}{1+e^{-y}} \qquad \textbf{(Equation 3-1)}$$

ANNs are represented in layers. The three types of layers are input, output, and hidden. A network does not have to employ a hidden layer and the input layer can be the same as the output, however in the early days of ANN research, it was shown that neural networks without hidden layers cannot solve complex nonlinear problems. ANNs are named for their network topology. A feed-forward network may have connections from layer i to layer j as long as $i < j$. These also allow lateral connections between neurons in the same layer. A feed-forward network is fully connected if every neuron in layer i is connected to every neuron in layer j. Feedback networks allow connections in the reverse direction from j to i, where $j > i$. Recurrent networks allow connections from a neuron to itself.

ANNs learn by changing the weights of connections. In other words, a network learns if its weight matrix W, which is a set of all weights in the network, obeys the following rule:

$$\frac{dW}{dt} \neq 0 \qquad \textbf{(Equation 3-2)}$$

A vast multitude of different learning techniques have been devised over the years. We are primarily concerned with learning techniques which employ Hebb's rule, one of the oldest and most widely known biological learning mechanisms (1949) often called "Hebbian learning":

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes place in firing it, some growth process or metabolic process takes place in one or both cells such that A's efficiency, as one of the cells firing B is increased.

This rule can be modeled mathematically as (Mehrotra et al 2000):

$$\Delta w_{ij} = c x_i x_j \qquad \textbf{(Equation 3-3)}$$

where c is some small constant, Δw_{ij} denotes the change in weight w, for a connection between the jth neuron, and the ith neuron. x_i and x_j are activation levels of these neurons.

Training can occur in two ways, supervised and unsupervised. Supervised learning is training of the network with data sets whose outputs are known. This is usually done before the network is in use. Unsupervised learning is typically done while the network is in use and when a solution is not known. Networks of this type are often used for categorization when the categories are not known beforehand and referred to as self-organizing or adaptive.

Some applications of neural networks include categorization, clustering, forecasting, vector quantization, system control applications, and pattern association, among many others. We are primarily concerned with pattern association of which pattern recognition is the simplest form. Given an input pattern, the ANN responds if this pattern is one of the patterns or similar to one of the patterns to which the network has been trained to respond. This can be pushed a step further into the concept of associative memory, of which there are two varieties auto-associative and hetero-associative. An auto-associative network maps an input pattern to itself. This is useful when the input sample is a corrupted, noisy, or a partial version of the desired output pattern (Mehrotra et al 2000). The network removes noise from the input. An example of an auto-associative ANN is one that takes a corrupted 32x32 pixel map of an English character and outputs the same character uncorrupted. A hetero-associative network maps the input pattern to a different output pattern. A hetero-associative memory by contrast might translate from a corrupted English character to an uncorrupted Greek character.

3.2 DLIF Basic Definition

A discrete leaky integrate-and-fire (DLIF) neuron is a stateful processing element in an artificial neural network that mimics the behavior of biological neurons to a greater extent than is common in artificial neural networks. A single DLIF neuron is defined by the following three equations,

$$\theta_n = K_t \qquad \textbf{(Equation 3-4)}$$

$$x_n = \lambda_n + \beta_t x_{n-1} \qquad \textbf{(Equation 3-5)}$$

$$\rho_n = C_t \qquad \textbf{(Equation 3-6)}$$

where x_n, θ_n, and ρ_n are the internal potential, the threshold level, and the relaxation level of $DLIF_t$, respectively. Here no distinction is made as to the structure of the input, λ_n. The potential at time n, x_n, is comprised of contributions from an input, λ_n, and the potential, $\beta_t x_{n-1}$, remaining from the previous time step. β_i is a constant between zero and one that denotes what portion of the potential from the previous time step remains. The DLIF neuron operates in three distinct states. In its integrate state the DLIF neuron sums its inputs received during the current time step, λ_n, and its residual internal potential carried over from the previous time step, $\beta_t x_{n-1}$ Equation 3-5. When the neuron reaches threshold defined in Equation 3-4, it transitions to the fire state and an output is produced that is equivalent to an action potential. Following this the neuron transitions to the recover state in which the internal potential falls to the resting potential described in Equation 3-6. After this recovery period the neuron returns to the integrate state in which it receives input as described by Equation 3-5.

If we consider Equation 3-5 to be continuous and consider its rate of change an interesting behavior emerges.

$$\frac{dx(n)_i}{dn} = \frac{d\lambda(n)_i}{dn} + \beta\frac{dx(n)_i}{dn} \qquad \textbf{(Equation 3-7)}$$

Consider input as a growth rate and the leak term as a decay rate, where the decay rate is represented as an exponential of the form $\beta = e^{-\alpha}$. We see that when the growth rate equals the decay rate, the rate of change of the internal potential is zero. This implies that at some point an equilibrium condition will arise in which the internal potential will stabilize at or about some value. In general, it seems that there are two cases to consider: First, the case in which the growth rate is constant in time. In this case, it can be shown that the internal potential will asymptotically approach some static equilibrium point. Secondly, consider the case in which the growth is not constant in time. In this case, no static equilibrium exists. However, if we further assume that the growth rate periodically oscillates around some median value, it can be shown that the internal potential will reach an oscillatory equilibrium about some other median value. We see that this oscillation of the growth rate can be said to drive the internal potential in a steady state. Thus we see a theoretical basis for making distinctions in the structure of the input. This approach to a stable state may also be interpreted as adaptive behavior, i.e. neural plasticity. It somewhat mimics the ability of biological neurons to adapt to sustained structured input.

3.3 Biological Foundation

Artificial neural networks seek to model the behavior of biological neural networks. In conventional ANN this model is very simplistic. In reality the behavior of even a single neuron is amazingly complex. Conventional neural models are capable of rich behavior in spite of this simplicity. DLIF seeks to incorporate a little more of biological neurons complexity in order to extend ANN capability. For this reason it becomes important for us to briefly consider the operation of the biological neuron.

Neurons are composed of a cell body known as the soma, dendrites, which collect incoming signal and an axon which carries the signal produced by the neuron away. The points at which axonal dendrites of one neuron meet the dendrites of another neuron are called synapses (Figure 3-2). There are many different kinds of neurons with different sizes, shapes, and response characteristics. We will discuss a generalized neuron for clarity. However, we will leave out most details of the complex organic chemistry involved with neuronal operation and primarily discuss physical properties.

Neurons are surrounded by various types of glial cells, which serve as caretakers to the neurons. Glial cells provide a structure to hold neurons in place, supply nutrients, and maintain or alter the chemistry of the solution in which neurons reside.

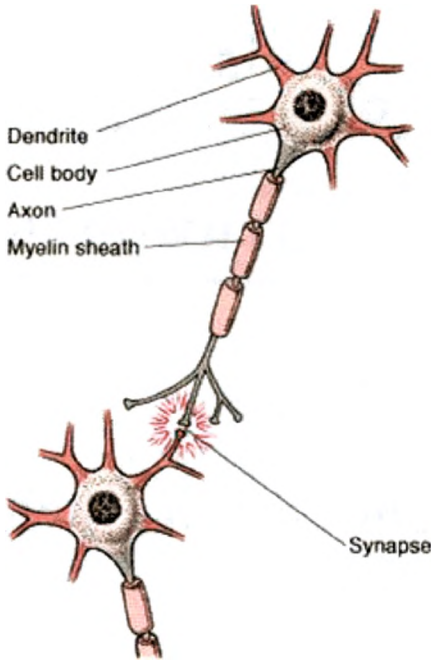

Figure 3-2 Generic Neuron Morphology

The majority of the behavior that we associate with a neuron takes place in the neuronal membrane. In fact, the potential difference across this membrane i.e. the difference between the charge of the interior of the neuron and the charge of surrounding solution mediates the majority of the neuronal mechanism. The solution in which the neuron resides contains four ions which are critical to this mechanism, sodium ions (Na+), chlorine ions (Cl-), K+ (potasium), and calcium ions (Ca2+). These ions cannot freely cross the membrane. (Figure 3-3)

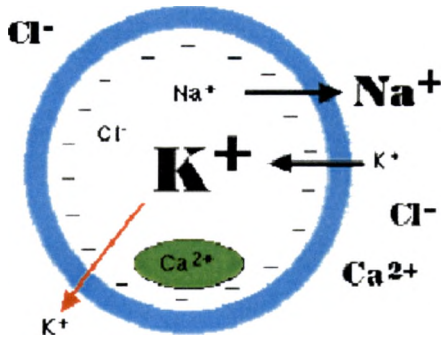

Figure 3-3 Neuronal Ion Distribution

However, the membrane is permeated with many ion gates (channels) that can allow certain types of ions to pass through. The concentration of Cl- inside the membrane is slightly higher than the solution outside, but this is not the case with Na+ and K+. Usually the neuron maintains a concentration of Na+ that is ten times lower inside the membrane than outside. K+ concentration is twenty times higher inside the membrane than outside, giving the membrane a potential difference of -70mV (mili-volts), known as the resting potential. This potential difference is attained using ion pumps which actively move Na+ and K+ ions against their concentration gradient. Some of the ion gates we have mentioned only allow Na+ to pass through them. These Na+ ion gates are opened primarily by type I (ionotropic) neurotransmitters, which are organic molecules that bond with receptors on the surface of the gate like a key in a lock and cause the gate to open. This allows Na+ ions to pass through the membrane, resulting in a localized change in the membrane potential. Na+ ions move into the cell under two forces. First, the process of diffusion where particles in areas of high concentration tend to move to

areas of low concentration until an equilibrium is attained i.e. the particles are more or less equally distributed over the entire region. Second, the electromagnetic (EM) attraction between the positive charge of the Na+ ion and the collection of negatively charged Cl- ions inside the cell. In addition to previously mention membrane properties positive ions have a tendency to 'leak' out of the cell through ion gates or other mechanisms. This leak turns out to be a key factor in neuron behavior. Considering all these factors, the rate at which Na+ ions enter the cell and internal potential increases is controlled by:

a) The number of open gates.
b) How long the gates stay open.
c) The number of Na+ ions already inside the cell. (diffusion)
d) The amount of positive charge inside the cell. (electromagnetic attraction)
e) The rate at which K+ or Na+ leak out of the membrane (leak current)

Neurotransmitter gates are for the most part only located in small areas on the dendrite membrane where the axon of one neuron comes close the dendrite of another. This area is known as synaptic cleft. The entire region is known as a synapse. (Figure 3-1). Once the potential difference across the membrane reaches a certain level usually around -50 mV in mammals a second kind of gate is triggered. At this point, the neuron has reached its threshold level and will produce an action potential. There are two varieties of this

voltage gated channel. One allows Na+ ions to pass through and one allows K+ ions to pass through. Both voltage activated types of gates are densely distributed across the entire membrane surface. Therefore, when the potential difference is increased to the point at which these voltage activated gates are opened, very many of them are opened even if the potential difference change is confined to a relatively small local region. The resulting increase in potential difference causes more voltage activated gates near the point of onset to be activated, thus spreading the change. These two voltage activated gates however have different reaction times. Voltage triggered Na+ gates open first allowing Na+ ions to rush into the cell under the influence of diffusion and EM attraction. These gates generally stay open for a time and then close. During this time the, membrane potential reaches +30mV (Figure 3-4). Based on this positive potential swing we can surmount that diffusion is the dominant factor. This stage of the neuronal mechanism is known as depolarization. Depolarization of a local region in the membrane triggers voltage gates near the local region to be triggered so this depolarization spreads from the source as the initial wave from a stone dropped in the water. By the time the local membrane potential has reached its maximum, the K+ voltage activated gates in that region open and K+ rushes out under the influence of EM repulsion and diffusion. (Figure 3-4) Then K+ voltage activated gates close. This phase is known as re-polarization. This phase overshoots the mark and re-polarizes the cell to a slightly lower potential than it was originally, know as hyper-polarization. This slight overshoot combined with the fact that the cell now contains a much higher concentration of Na+

makes it extremely difficult to raise the internal potential to the threshold level. Now the cell is in a recovery state and unless subjected to extreme stimulus, it will not fire. During this phase ion-pumps exchange Na+ ions for K+ ions through a process known as active transport. This ion-exchange takes place until the K+ and Na+ concentrations reach their initial values.

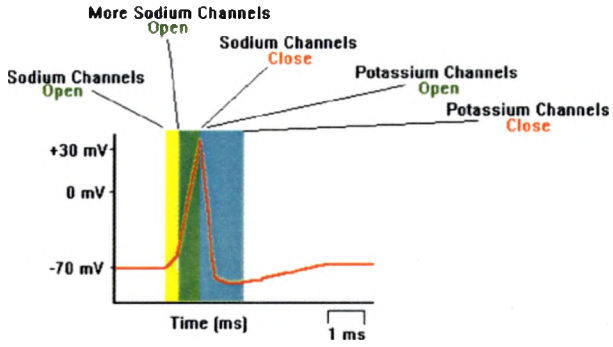

Figure 3-4 Action Potential Generation

The threshold event, in which the internal potential reaches the threshold level, marks the beginning of the 'all or nothing' action potential generation described above and re-polarization marks the end. The action potential travels across the membrane like a wave and enters the axon. Action potential onset is generally thought to originate in the axonal hillock, an area of the neuronal membrane around the root of the axon where contributions from dendrites are integrated together. This implies a passive role for dendrites, meaning that action potential could never originate from a dendrite. There is much evidence that this is not the case and the role of the dendrites in action potential

production is a current research topic. (Koch 1999) Regardless of whether the action potential originates from the dendrite or not, the action potential propagates across the cell membrane to the axon. The axon is almost totally covered with a layer of fatty material known as the myelin sheath, which protrudes from the schwan cell. The myelin sheath acts as an insulator that stops depolarization and re-polarization from occurring along the surface of the axon that it covers. Gaps in the sheath coverage are called nodes of Ranvier and serve to boost signal strength (Figure 3-5). Na+ and K+ ion channels in the nodes cause just enough depolarization to trigger the ion channels at the next node. This makes propagation of the signal along the axon much faster than across normal neural membranes.

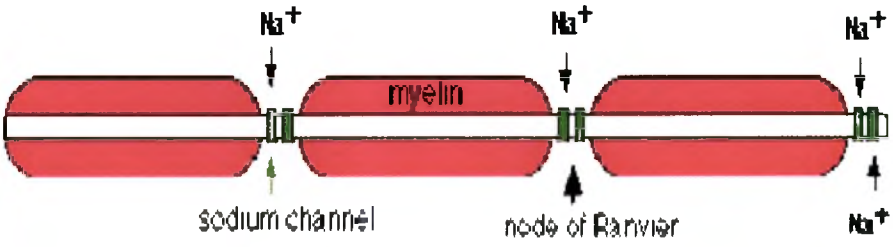

Figure 3-5 Nodes of Ranvier

However in general signal propagation along a series of neurons is much slower than electric signal propagation. This makes sense if we consider the fact that action potential propagation is basically ion exchange through the membrane that depends on the opening and closing of neurotransmitter and voltage triggered ion channels, while current is the

flow of subatomic particles, namely free electrons, through a conductor. On a microscopic scale for all intensive purposes electric current propagation is instantaneous, while action potential propagation is not. In fact two action potentials traveling along a path of the same length may have significant differences depending on properties of the neurons involved. This indicates that when modeling a neural system, propagation latency is likely a significant factor.

There is also another type of neurotransmitter, which does not directly open Na+ gates, metabotropic (M). This metabotropic type triggers the release of a secondary messenger which causes changes in the cell, which may include the opening of Na+ gates. However, this alternate system has a much broader range of possible effects within the post-synaptic cell, changing such cell characteristics as rest potential, leak rate, the potential at which a threshold occurs, Na+ uptake, etc. This type has both a longer lasting effect and a slower onset time, than the former neurotransmitter type (I) and are often used to modify behavior of neurons receiving stimulus from type I neurotransmitters. This correlates to long lasting behaviors such as emotion which tend to change the way input is interpreted on a macroscopic scale.

There are hundreds of types of ionotropic (I) and metabotropic (M) neurotransmitters. However, each synapse generally uses just one type of neurotransmitter to open gates on the post-synaptic neuron. Axons usually synapse onto the dendritic tree of a neuron and

rarely onto the cell body. The synapse is the meeting point between small branch-like structures at the end of the axon known as axonal dendrites and dendrites of the post-synaptic dendrite. At the synapse, one or more small buds protruding from the axonal dendrite project into a small depression in the post-synaptic dendrite known as a synaptic cleft. Neurotransmitters are usually released into the synaptic cleft from small buds protruding from axonal dendrites, which terminate in the synaptic cleft. In the bud, arrival of the action potential triggers a series of events that cause vesicles containing an amount of neurotransmitters to move to the cell membrane of the bud, fuse with quantal release sites and release their neurotransmitters into the synaptic cleft. Each vesicle contains a quantity of neurotransmitter known as a quanta which is approximately equivalent for each vesicle in the synaptic bud. This causes the Na+ ion channels on the post-synaptic dendrite to open. Enzymes in the cleft act to destroy neurotransmitters, so that gates do not remain open. This seems simple enough but there are hundreds of different types of neurotransmitters all with slightly different properties. In an ANN we normally model transmittance across this cleft as weight. However it is actually a more complex relationship. Biophysicist have traditionally used three quantities in relation to synaptic strength or weight: 1) n the number of quantal release sites, 2) p the probability of synaptic release per site, 3) q some measure of the post-synaptic effect per quanta mediated by such factors as how many neurotransmitter gates are available, how long they stay open, and how much Na+ they allow to enter the cell, leak rate, etc (Koch

1999). One possible relationship between these terms is R, the time averaged response size.

$$R = npq \qquad \textbf{(Equation 3-8)}$$

This is a time average and may be vastly different over smaller time segments.

This brings to light an interesting aspect of synapses. They have a probability of release *p*, which means that not every action potential arriving at a synapse will be carried across or at least that the strength of a connection is not constant per action potential. This probability varies depending on the history and location of the synapse. Average vesicle release probability in the cortex is relatively low at less than 30% while the average release probability for synapses in a pathway directly leading to muscle tissue is high at greater than 70%. Furthermore, this probability is not constant in time. It depends on the history of the synaptic activity. An increase in release probability p is known as facilitation. For brevity, we will not fully discuss this idea. Let us suffice to say that when two action potentials arrive at a synapse in rapid succession the probability of release on the second pulse is much higher (Paired Pulse Facilitation) than the first. This carries through to subsequent pulses also, so that each subsequent pulse has a higher probability of transmittance as long as they arrive in rapid succession (augmentation and

post tetanic Potentiation). This probability falls off over time, but at a slower rate than its onset.

3.3.1 Synaptic Plasticity

The discussion above leads us to a more general description of synaptic phenomena known as synaptic plasticity. Synaptic plasticity refers to the ability of synapses to change. Similar forms of plasticity have been observed at synapses of virtually all organisms from crustaceans to mammals. Our common perceptions of these are learning and adaptation. Synaptic plasticity can be mitigated by any of the various factors which influence transmittance across a synapse. They are categorized by the time span on which they act and locus of induction. For example, some effects such as PPF are induced on the pre-synaptic neuron while others are induced on the post synaptic side and still others are induced on both sides. The most famous of these is Long Term Potentiation, which can last for minutes, hours, weeks, or even years. These are our long term memories. LTP obeys Hebb's rule in that its induction requires simultaneous pre and postsynaptic activation (Koch 1999). There is on going debate and research on the precise mechanisms behind LTP and many aspects of synaptic plasticity in general.

3.3.2 Neural Plasticity

In addition to synaptic plasticity there is also non-synaptic plasticity in which the neuron itself adapts to sustained input. The firing rate of a given neuron that is being stimulated

by a sustained high level of input will slow down over time. Initially this firing rate is very high but the neuron in question adapts on a time scale of hundreds of mili-seconds. This can be thought of as a form of learning, in this case learning to adapt out i.e. filter the sustained and therefore predictable component of the input. (Koch 1999)

3.4 Action Potential Timing

Neuron output pulse timing is crucial to the DLIF neuron and encodes much information about the input. The output (threshold) time of a single DLIF neuron can be calculated from Equation 3-5. The first step is to solve the difference equation and derive the internal potential as a function of discrete time, n. To accomplish this we will assume the input, λ, is constant in time and initially internal potential is zero. Thus the time evolution is as follows:

n (time)	X (Internal potential)
0	0
1	λ
2	$\lambda + \lambda\beta$
3	$\beta(\lambda + \beta\lambda)$
4	$\lambda + \beta(\lambda + \beta(\lambda + \beta\lambda))$
	...
N	$\lambda \sum_{0}^{N-1} \beta^k$

$$x(n) = \lambda \sum_{0}^{n-1} \beta^k \qquad \textbf{(Equation 3-9)}$$

The leak term, β, is on the interval (0,1), so β is considered when it is less than one, $\beta < 1$. Equation 3-9 is a finite geometric series, and reformulated as the sum of that series.

$$x(n) = \frac{\lambda(\beta^n - 1)}{(\beta - 1)} \qquad \textbf{(Equation 3-10)}$$

We are looking for the time at which the neuron fires, so we set Equation 3-5 equal to θ and solve for time, n. Skipping the algebra, the following equation is obtained.

$$n = \frac{\ln\left(\frac{\theta(\beta - 1)}{\lambda} + 1\right)}{\ln(\beta)} \qquad \textbf{(Equation 3-11)}$$

Holding β and θ constant we conclude that the firing time of a DLIF, given constant input, λ, is proportional to the log of the inverse input.

$$n \propto \ln\left(\frac{1}{\lambda}\right) \qquad \textbf{(Equation 3-12)}$$

So the greater the input the faster threshold level is reached. This result is as expected and matches observed behavior in biological systems. In the simple case of constant input explored here an explicit solution is easily found. But if input, λ, is not constant in time it cannot be treated as constant in Equation 3-9, and thus Equation 3-10 cannot be reformulated as the sum of a geometric series and an explicit solution such as Equation 3-11 cannot be found. However the solution will clearly still involve some form of log transformation.

3.5 Behavior Under Coherent Modulation

As we noted earlier the temporal structure of DLIF input can have a dramatic affect on neuron behavior. In this section we will examine two general categories of this in more detail. The first category is using oscillatory input, which we will re-label as coherent to match terminology used by Fahat (Lee and Farhat 2002). Secondly, we will examine primarily unstructured input, which is constant or uniform in time. We will again re-label this as incoherent to match Farhat's terminology. This allows us to redefine our DLIF in terms of incoherent and coherent input.

The set of equations to follow describe the behavior of a DLIF, augmented with a coherent and incoherent input. They describe the behavior of the i^{th} DLIF as a function of the discrete time n,

$$\theta_n = K_t \qquad \textbf{(Equation 3-13)}$$

$$x_n = \psi_n + \phi_n + \beta_t x_{n-1} \qquad \textbf{(Equation 3-14)}$$

$$\rho_n = C_t \qquad \textbf{(Equation 3-15)}$$

where x_n, θ_n, and ρ_n are the internal potential, the threshold level, and the relaxation level of $DLIFN_t$, respectively at time step n. Contributions to the internal potential of $DLIFN_t$ noted in Equation 3-14 come from the incoherent signal, ψ_n, and coherent signal, ϕ_n. The potential, x_n, changes in time due to contributions from coherent and incoherent signals, and the remaining potential from the previous time step, $\beta_t x_{n-1}$. This remaining potential is comprised of the potential from the previous time step, x_{n-1}, modified by the 'leak' factor, β (a constant between 0 and 1) representing decay and represents the amount of potential from the previous time step that has not leaked away. This continues until the potential reaches the threshold level, θ_n, which is some constant value K. The internal potential then immediately drops to the relaxation level, ρ_n, which is a constant value C_t. Equation 3-14 is a recursive function dependent on a discrete time step *n*.

3.5.1 Incoherent Signal

An incoherent signal is provided by a regular pulse of variable frequency. In a DLIF network, this can be produced by a DLIF that is connected to every other DLIF and itself.

This DLIF produces a pulse on a constant time interval. This pulse arrives at every other DLIF simultaneously and serves to increase the neuron potential as in Figure 2-1. The rhythmic pulse can be generated utilizing one recurrent connection, whose time delay serves to regulate the firing frequency. This is intended to be loosely synonymous with rhythmic pulses emanating from the hypocambal area in mammalian brains. Various studies have shown a correlation between frequency ranges of this rhythmic signal and general behavioral states in mammals such as heightened alertness, concentration and problem solving, hypnosis, sleep, etc. [3]. Different frequency ranges have been given different names such as Theta (7 – 10 Hz) and Gamma (60 – 100 Hz) (Olufsen and Whittington 2003). Regardless of its specific role or roles in behavioral states, it is sufficient to say that this rhythmic pulse exists, is dynamic, and is propagated to various areas of the mammalian brain.

3.5.2 Coherent Signal

The DLIF coherent signal is not specifically sinusoidal as in the Bifurcating Neuron (BN) (Chapter1 Related Work). It is a spike train, whose pattern repeats over a certain time interval. In other words, it is a series of pulses of various temporal spacing and possibly various amplitudes, which repeat over a time period. It can be spaced in time in such a way that it produces an approximate sinusoidal response in the internal potential of an DLIF, but it is sufficient that this signal causes a change in the internal potential that is not constant overtime within the period, (Equation 3-14). In Figure 3-6, we see a theoretical depiction of a DLIF stimulated by coherent signal alone. We can see that the

presence of the coherent signal affects the system in a similar way as in the BN (Figure 2-1). It causes changes in the timing of threshold events i.e. neuron activation. This couples the temporal spacing of the output spike train to that of the coherent signal.

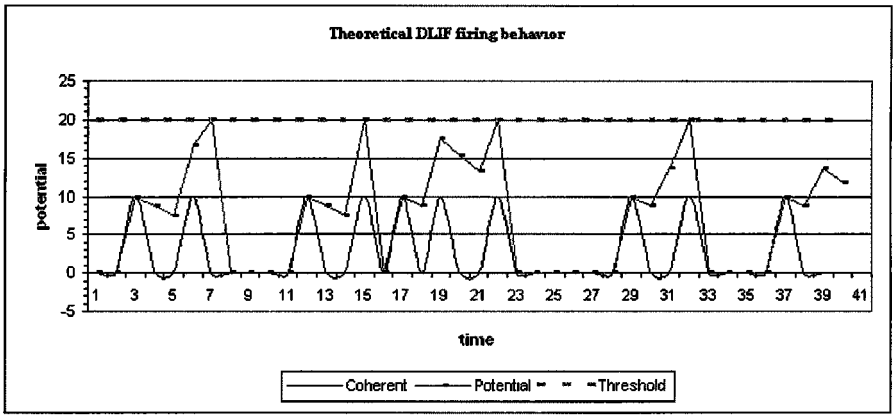

Figure 3-6 DLIF Theoretical DLIF Neuron Firing Behavior

The internal potential is driven to the firing threshold by simulated coherent input. One cycle of periodic coherent input is depicted, so a repeating pattern of threshold events is not observed. In general this figure illustrates how input leads to the temporal spacing of action potential generation. Additionally, it is observed that internal potential falls to zero after a threshold event.

3.5.3 Expected Outcome

We now examine the expected behavior of the DLIF. First consider the DLIF driven by the incoherent signal alone. The incoherent signal is synonymous with a signal that is constant in time. However, due to the discreet nature of the pulse data, this cannot be achieved. Therefore our incoherent signal is only quasi-constant in time. This is represented as a rhythmic pulse of constant amplitude and frequency. According to our discussion in section 4.4, the internal potential will approach a constant value in which

the growth rate equals the decay rate given, a constant input. Therefore, we would expect our quasi-constant input to approach this behavior. However, according to section 4.4, a periodic input will result in a periodic oscillation of the potential.

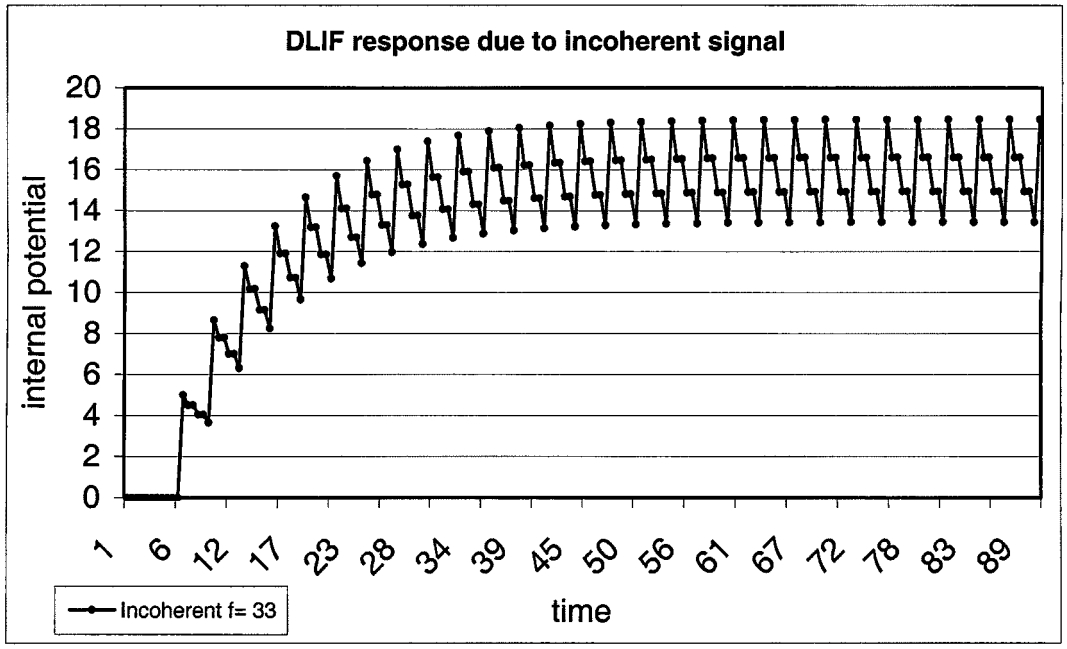

Figure 3-7 DLIF response due to incoherent signal

An DLIF is driven by a quasi-constant input of frequency 0.33 Hz. The internal potential is seen to oscillate periodically around some median value, which asymptotically approaches equilibrium constant.

Figure 3-7 depicts a sample output of a DLIF supplied with this incoherent input. The result is as expected. The internal potential approaches an oscillatory equilibrium in

which it oscillates predictably about some median value that asymptotically approaches a constant value. Likewise for the coherent input, we would expect a similar response. Figure 3-8 depicts the response of an DLIF to coherent input. In this instance, coherent input is titanic, which means that the signal is composed of a series of pulse inputs closely spaced in time (one time step) followed by a period of no input. The whole cycle is then repeated. While this is not sinusoidal, it fits our requirement that the signal is not constant in time and is periodic. The fact that a titanic signal is acceptable here is interesting, considering the prevalence of titanic signals in biological systems.

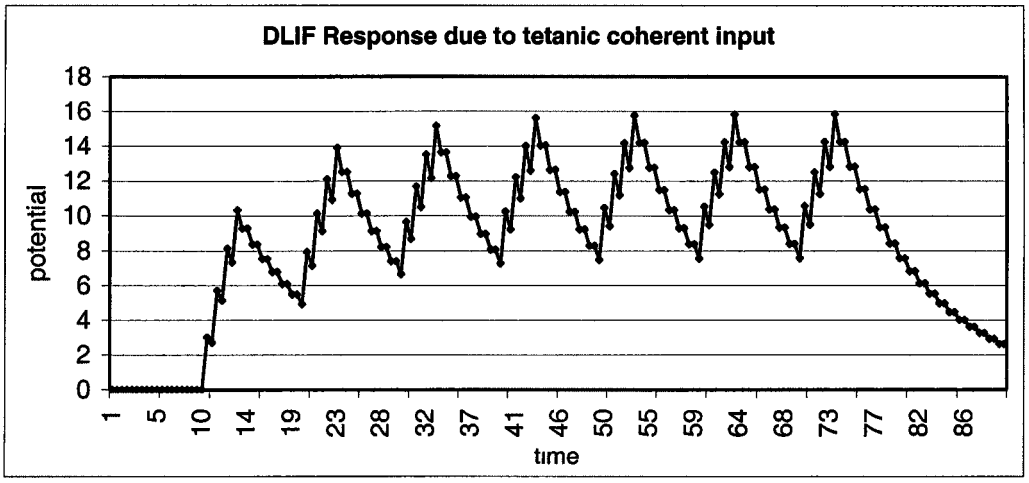

Figure 3-8 Response of DLIF due to titanic coherent input

Tetanic coherent input consists of multiple closely spaced pulses (titanic), which occur periodically. The internal potential is seen to oscillate periodically around some median value. The trailing tail marks the conclusion of the input signal. The potential then falls off exponentially.

3.5.3.1 Amplitude to Phase Conversion

In Amplitude to phase conversion, we start with input patterns that contain both incoherent and coherent signals. These are arranged in such a way that the DLIF produces output spikes on regular intervals that correspond to the period of the coherent input. If an additional input or inputs are introduced during a coherent input period, this causes a phase lead in the output spike for that period. In other words, the DLIF fires before its normal firing time by some amount proportional to the amplitude of the additional input signal. If this signal is repeated in the next period this phase lead will appear again. This effect can be summarized with the addition of a new term to Equation 3-14, representing the contribution to the internal potential from the additional input denoted by $\nu(t)$, referred to as the data input.

$$\frac{dx(t)_i}{dt} = \frac{d\nu(t)}{dt} + \frac{d\psi(t)_i}{dt} + \frac{d\phi(t)_i}{dt} - \beta\frac{dx(t-1)_i}{dt} \qquad \textbf{(Equation 3-16)}$$

Figure 3-9 depicts an example of the DLIF response due to incoherent and coherent input compared to the response of incoherent, coherent, and data input signals. The output pulse timing of response including the data signal is phase shifted.

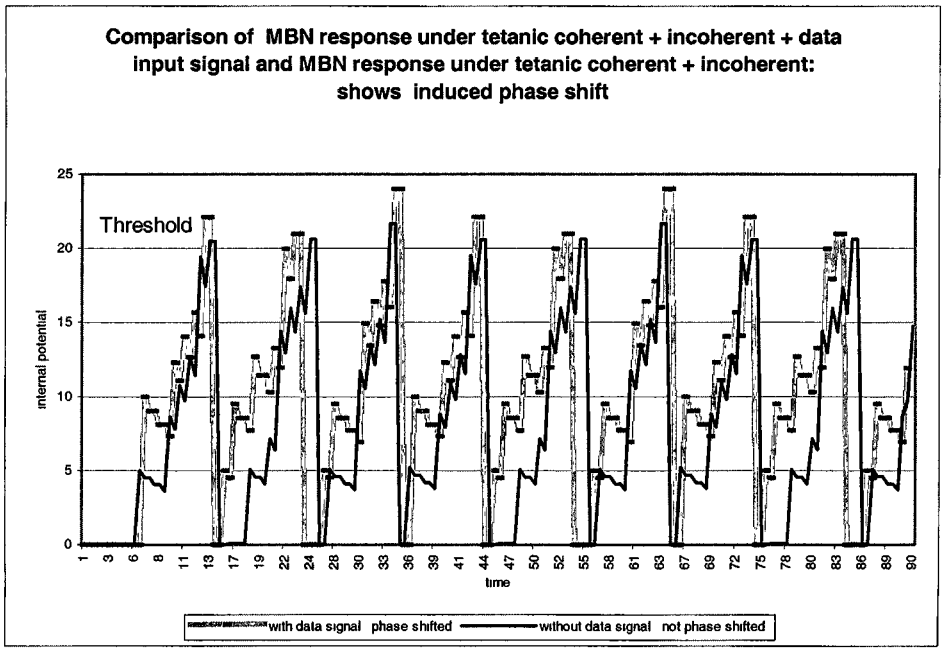

Figure 3-9 Response of DLIF due to titanic coherent, incoherent and data input

Input containing a data signal as well as coherent and incoherent input is phase shifted with respect to the same coherent and incoherent input without the data signal. The firing threshold is twenty. The firing time phase shift is not constant, but this pattern of irregularity repeats over several firing intervals. Notice that the phase shift of the firing time at t = 23 is the same as the spike at t = 54 and t = 85. This is due to the discreet nature of the time sampling.

3.5.3.2 Holographic Paging

Now consider a more general case where the contributions to the internal potential from the data section at time t are given by $v(t)$. If $v(t)$ is maintained, but the coherent signal, $\phi(t)_i$, is altered, it is apparent that the rate of increase of the internal potential and thus the time spacing of output spikes will be altered. In fact, changing the coherent

signal should change intervals on which the DLIF fires even without a data input. Thus, the output spike pattern will only be the same, if both the coherent signal and the data input pattern are unaltered. Significant changes in either will result in an output pattern that does not match the original. Thus a particular output pattern can only be recreated given the appropriate coherent signal. The second major requirement is that storing a new data pattern with a different coherent input does not disturb the original pattern. The original BN accomplished this in a nearest neighbor Pulse Coupled Neural Network PCNN, using higher order synaptic connections. This requires that one or more new connections be added for each stored pattern. The time delays of new synaptic connection are offset by an amount proportional to the induced phase shift.

CHAPTER 4 - DISCRETE LEAKY INTEGRATE-AND-FIRE NETWORKS

- 4.1 Synaptic Connections
- 4.2 Computation with Action potential
- 4.3 Network Input and Output
- 4.4 Object Model Overview
- 4.5 Object Model Implementation Detail

4.1 Synaptic Connections

A Discrete Leaky Integrate-and-Fire (DLIF) network consists of leaky integrate-and-fire neurons and the connections that connect them. Connections are one way between at most two neurons. DLIF connections also connect neurons to the outside world for the purpose of external input and output. Networks may contain single or higher order connections as well as recurrent connections. Unlike conventional neural networks, DLIF

networks use time delays as well as weights in connections between processing elements. This can allow a feedforward network to act as a recurrent network.

Connections carry information between neurons in the form of pulses. These pulses model the concept of biological action potentials traveling across a synaptic pathway from one neuron to another. The amplitude of a pulse generated by one neuron is scaled by the weight assigned to this connection as the pulse passes through the connection. This amplitude then directly contributes to the internal potential of the receiving neuron. The time delay in the connection simulates the propagation latency of the electro-chemical action potential i.e. pulse or spike along the various neural membranes forming a pathway between two primary neurons. In general, propagation latency along an axon is negligible compared to that of the dendrites and the membrane comprising the soma, so this can be left out. In analogy with a biological neural system, this pathway may contain multiple inter-neurons or stellate neurons between primary neurons. Therefore, a connection is viewed as a path from a primary neuron through one or more inter-neurons to another (or the same) primary neuron.

All neurons maintain a universal time value which is used to synchronize activity (n in Equation 3-4, 3-5, and 3-6). Pulses contain an additional time value which determines the time at which their next event occurs. This event is the time at which they are scheduled to arrive at a post-synaptic neuron.

Synapses which carry type (M) neurotransmitters and release secondary messengers that augment neural behavior can easily be modeled by the DLIF synapse. A synapse receiving an input pulse would simply change parameters of the post-synaptic neuron such as leak rate and threshold level, instead of contributing to the internal potential of that DLIF neuron. This change could be permanent, expire after a certain time segment, or dissipate gradually.

4.2 Computation with Action potential

In the limited case, in which all time delays in a given network are the same DLIF networks behave similarly as conventional neural networks. Input to particular neuron is determined solely by the connection weight. But in a network in which connection time delays vary, a different behavior can emerge. The receiving neuron can be thought of as a coincidence detector, where coincidence refers to the proximity of arrival in time. The leak rate of a neuron modulates this factor and effectively controls the accuracy of coincidence detection. The leak rate accomplishes this by specifying how much of contribution from a pulse that arrived earlier will remain when the next pulse arrives i.e. how quickly the potential leaks away. If the internal potential left over from a previous pulse plus the internal potential contributed by a second pulse reaches threshold level, then the receiving neuron has effectively detected coincidence in time between the pulses. If these pulses have arrived via different connections, the connection weight can be interpreted as the relative importance of one pulse over the other. Now if a small

network is considered (Figure 4-1) in which three neurons are connected to a fourth via synaptic connections with different time delays, a particularly interesting way to exploit this behavior emerges. Assume each of these neurons thresholds at a different time and that each connection has a time delay so that all the pulses they produce, even though they are produced at different times, arrive at the fourth neuron simultaneously. Assuming the connection weights are also calibrated so that each connection contributes one third of the threshold value of the receiving neuron, we have a network that detects a temporal pattern. It is evident that the leak rate controls the error tolerance in the arrival time of these contributions. Furthermore, the weights can be adjusted so that some of the connections contribute more to the receiving neuron and some contribute less leading to a concept of importance among the inputs. Hopfield, who pioneered this technique, refers to it as computation with action potentials (Hopfield 1998). In fact, we see that the detection of this temporal pattern is not dependent on an absolute arrival time but the relative arrival time of the contributions. Considering the mathematical property of the DLIF under constant input discussed early, or in fact any leaky integrate and fire model it can be seen that this pattern detection is scale invariant with regard to the original neuron inputs. This is analogous to our common perception of being able to detect visual patterns such as faces under various light levels, or being able to recognize a spoken word at various volumes, or a smell at various intensities.

This relationship can be shown mathematically using the firing time of a DLIF neuron under constant input. (Equation 3-11) Consider a network topology in which three neurons (A, B, C) are connected to a fourth (D) (Figure 4-1). The weights of these connections are θ/3 and the time delays are tuned so that given constant input values of X,Y, and Z to neurons A,B, and C respectively the output pulses produced by A, B, and C all arrive at D simultaneously causing D to reach threshold level, θ. In essence, this arrangement detects the input Pattern XYZ. The time at which A, B, or C produces an output spike is given by Equation 3-11 and depends on the log of the inverse input. For convenience Equation 3-11 will be simplified by combining or dropping constants. This results in the following,

$$n = \ln\left(\frac{S}{\lambda}\right) \qquad \textbf{(Equation 4-1)}$$

where S is a composite constant and λ is the input at each time step. Using three simultaneous equations obtained using input values X,Y, and Z in neurons A, B, and C respectively, gives the firing times of neurons A, B, and C.

$$n_a = \ln\left(\frac{S}{X}\right)$$
$$n_b = \ln\left(\frac{S}{Y}\right)$$
$$n_c = \ln\left(\frac{S}{Z}\right)$$

Now, if the input values are scaled equally by a value α analogously with the discussion above, the following is obtained.

$$n_a' = \ln\left(\alpha\frac{S}{X}\right) = \ln(\alpha) + \ln\left(\frac{S}{X}\right) = n_a + \ln(\alpha)$$
$$n_b' = \ln\left(\alpha\frac{S}{Y}\right) = \ln(\alpha) + \ln\left(\frac{S}{Y}\right) = n_b + \ln(\alpha)$$
$$n_c' = \ln\left(\alpha\frac{S}{Z}\right) = \ln(\alpha) + \ln\left(\frac{S}{Z}\right) = n_c + \ln(\alpha)$$

Therefore, the original spike times (n_a, n_b, n_c) are linearly transformed by the scale factor as opposed to the usual non-linear affect of a scaling. This transformation leaves their relative timing unaltered. Graphically, this appears as a lateral shift of the entire pattern along the time axis (Figure 4-1).

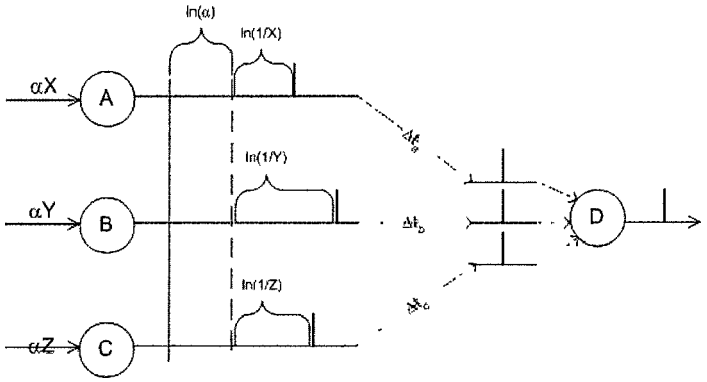

Figure 4-1 Action Potential Timing Scale Invariance

4.3 Network Input and Output

Input from outside the network is supplied as a series of pulses to the network input synapses. These connections are no different than internal connections. However, we can apply a semantic difference to the data that we are injecting. We represent real valued inputs as pulses whose amplitudes correspond to the input. This input can be scaled by the synaptic weight of the connection as needed. It is convenient to think of this input as a discrete sampling of a continuous input that may change over time. In any case, what we are concerned with is transforming or rather encoding the real valued input into a spatio-temporal pattern of spikes that can be processed by the network. In other words, information is encoded in the relative firing times of neurons as well as the neurons that fire.

Network output depends heavily on the purpose of the network. However, we can categorize it in two general classes, binary and real valued. In the case of binary, we are concerned only with whether an output neuron fires or not, i.e. its spatial output pattern. In the real valued case, we are concerned with the output spike temporal pattern. We can extract the real value by looking at the frequency of the output or the sum of the output overtime. This essentially reverses the integrate-and-fire process used to create a temporal pattern at input.

4.4 Object Model Overview

DLIF networks are composed of a collection of objects that carry out the functions implied by their names. The *neuron* object is where most of the processing occurs. Information propagated by each *neuron* is modeled by *pulse* objects. Each *pulse* corresponds to a spike in a spike train i.e. *neuron* output. Each *neuron* contains an axon and a collection of dendrites. These objects serve as containers for incoming and outgoing *pulses*. The *neuron* object delegates direct control of *pulse* objects to these containers. Connections between *neurons* are managed by *synapse* objects. *Synapses* maintain a collection of *pulse*s, which are waiting to arrive at the *neuron* in question, and contain information about the time delay between *neuron*s and the connection strength or weight for this particular connection. A *synapse* can be between at most two *neuron* objects. Time is defined in terms of a universal clock tick that all objects receive. When *pulse*s that have been scheduled to arrive at a *neuron* held in a *synapse* actually arrive, they contribute to the internal potential of a *neuron* object. If this *neuron* reaches its threshold, a new *pulse* is generated and sent to every *synapse* connected to the axon of this *neuron* (Figure 4-2).

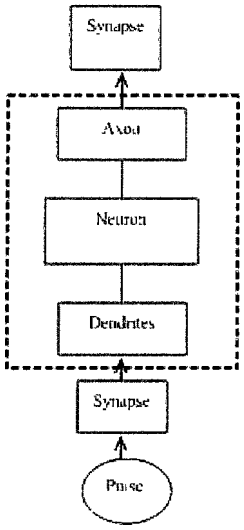

Figure 4-2 DLIF Object Model

4.5 Object Model Implementation Detail

A sparse static class diagram using the Universal Modeling Language (UML) is provided (Figure 4-3). This shows the basic class structure of DLIF.

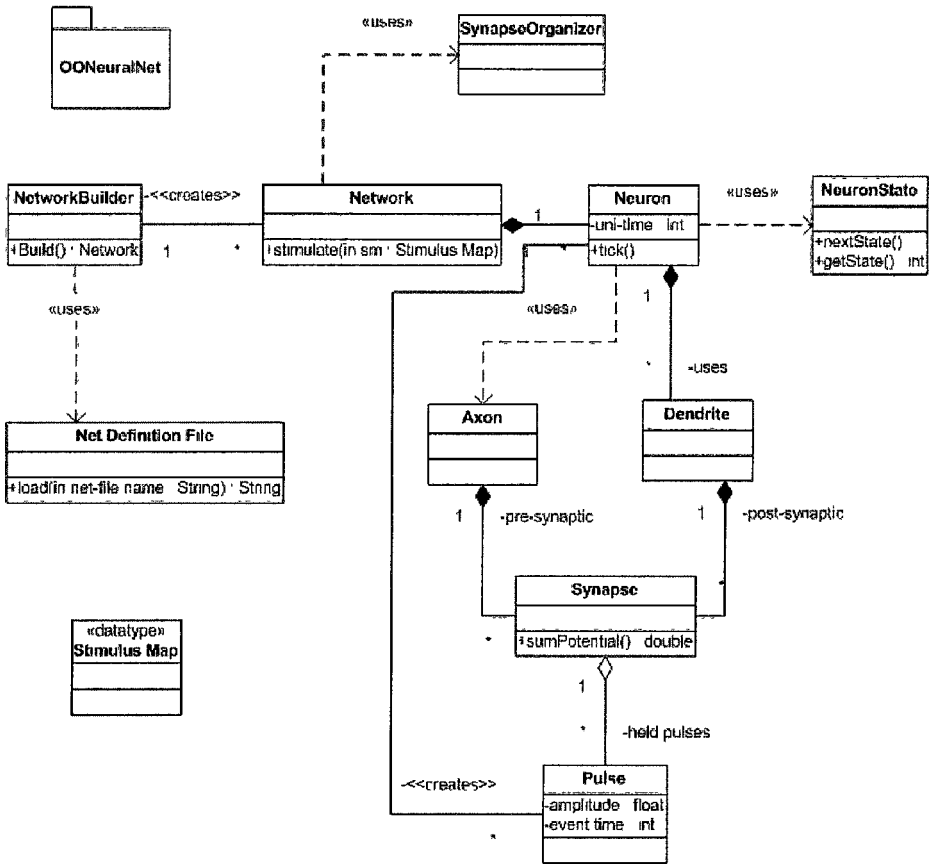

Figure 4-3 Basic DLIF Class Diagram

A network container is built from a `Network Definition File`, which describes the full structure of the network. Using this file `Neurons` and the `Synapses` which connect neurons are constructed. Every `Neuron` object contains an `Axon` object which takes care of delivering `pulses` created by the neuron object to the appropriate synapses. `Pulses` are then held at the `synapse` in question until the event time, which is set when they arrive and denotes their arrival time (uni-time) plus the time delay specified for this `synapse` object. On every tick, every `neuron` sums the potential delivery to it across its connecting `synapses`

by calling a sumPotential function on its dendrites, which in turn call the sumPotential function on every connecting synapse. Each synapse calculates the potential it delivers by multiply the amplitude of pulses whose event has arrived by the synaptic weight of this connection. Once a pulse has contributed its amplitude, it is destroyed. The state of each neuron is maintained by an instance of the NeuronState class. Neurons sum potential in the integrate state. Once the threshold level has been reached the neuron transitions to the fire state and produces a pulse which is synonymous with an action potential. It then transitions to the recovery state for the specified refractory period followed by a transition back to the integrate state.

Network input from the outside world is supplied through the stimulus function of the Network object. The stimulate function uses a Stimulus Map data type which maps input values in the form of pulse amplitudes to the neurons they are intended to stimulate. Input and output synapses are created by the Network upon creation. These are specified by the Net Definition File and may connect to any layer and any subset of neurons in a layer. This complexity is handled by the Synapse Organizer class. Input can be provided to any subset of neurons contained in the network. Likewise output can be collected from any subset but is collected from the entire last layer by default.

CHAPTER 5 - APPLICATIONS

- 5.1 Visual Tracking
    - 5.1.1 Network Architecture
    - 5.1.2 Sensory Input
    - 5.1.3 Extension To Three Dimensions
    - 5.1.4 Static Visual Field and Non-static Visual Fields
    - 5.1.5 Biological Foundation
    - 5.1.6 Motion Detection
    - 5.1.7 Pursuit motion
    - 5.1.8 Micro-Saccades
- 5.2 Pattern Recognition
    - 5.2.1 Network Architecture
    - 5.2.2 Sensory Input
    - 5.2.3 Novel Learning Technique
    - 5.2.4 Results of Pattern Recognition Experiments
- 5.3 Self Organizing Map

5.1 Visual Tracking

As an application of the DLIF neuron we have implemented and tested a visual tracking system. There are three major behaviors associated with the network architecture used in this system that will be discussed in depth.

- Motion detection – response to movement in the visual field.
- Pursuit motion – tracking the motion of an object through the visual field
- Micro-Saccades – small involuntary saccades (eye movements) around objects of interest in a visual field.

The above are inherent aspects of the network. All of these are exhibited by the network architecture with the addition of one or more enhancements for improved pursuit motion. However all architectures are very similar and vary mostly in how we interpret the output and manipulate the input set.

The visual tracking problem will be divided into two phases. In the first phase, the visual field will remain fixed. In this case, network output describes the motion inside this visual field. In the second phase, the visual field itself moves based on network output, which is analogous to our common experience of vision. In the moving visual field phase

the added complexity of the apparent motion of the entire background scene has to be dealt with. This will be discussed in further detail in the sections to follow.

In general, visual tracking is a four stage process: a) we compare the input at the current time step with input at the previous time step. This step looks for changes in the input data on the level of rows and columns; b) we compare changes with one another to look for lateral motion within rows and columns; c) we sum lateral motion in coordinate directions and the neural network outputs the data; d) we interpret this raw data as general object motion and to some extent location. This approach breaks down motion into two dimensional (2D) vector components.

5.1.1 Network Architecture

The visual tracking ANN is a four layer network. Layer one is denoted as the sensory neuron (SN) layer. It is divided into four sections, one for each coordinate direction. (Figure 5-1)

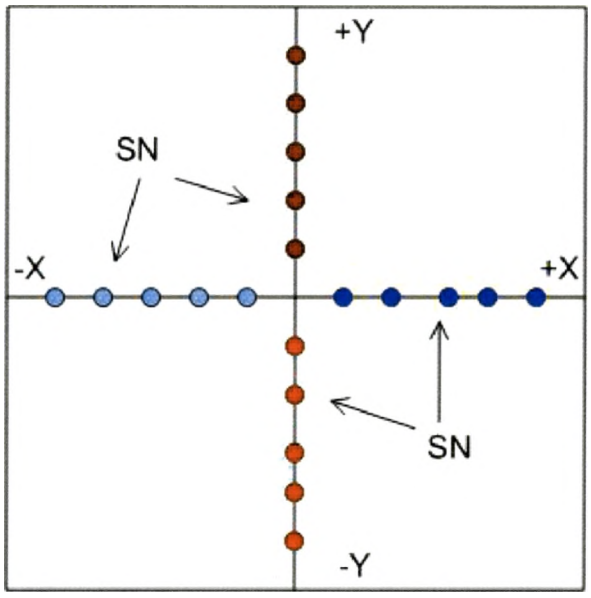

Figure 5-1 SN Spatial Orientation

This figure shows the spatial orientation of sensory neurons (SN) in relation to the visual field.

It is similar to a Cartesian coordinate system. Input is spatially mapped to the proper input group. This network uses pixel data as input. The human visual system goes to great lengths to maximize differences between neighboring input regions on the retina. However, difference maximization will be ignored since image data in a computer system is already nicely pixilated. Pixel values are manipulated and summed for an entire column or row and fed into the SN corresponding to that column or row. The mechanics of pixel manipulation will be discussed in the Sensory Input section. Manipulated pixel value sums are represented by pulse amplitudes (DLIF Networks) and fed into neurons in the SN group. SN group neurons have overlapping coverage of pixel data (Figure 5-2).

For instance, SN group Up (+Y) and SN group Left (-X) both get input from pixels in the upper left quadrant of the input area.

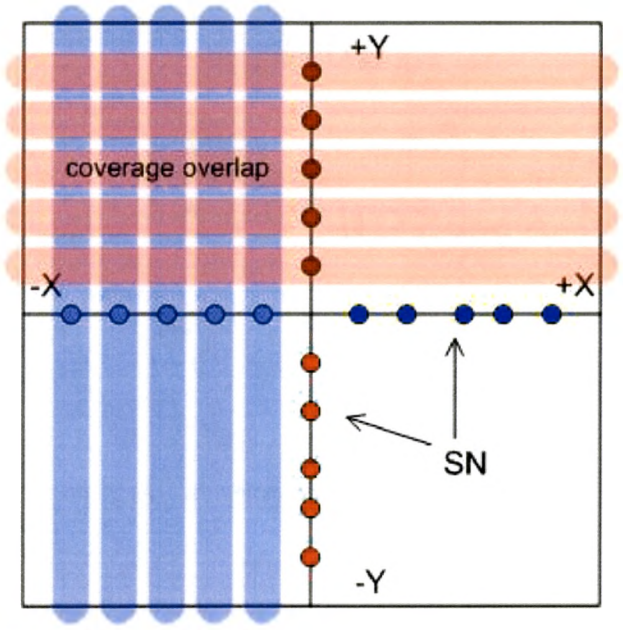

Figure 5-2 SN Coverage Overlap

This figure shows the overlapping coverage of pixels in the visual field by corresponding SN. Pixels are exponentially scaled and summed in rows and columns before being fed into the appropriate SN.

Each SN layer neuron synapses onto a differential neuron (DN) in the DN layer. This layer is designed to respond to positive changes in pixel values. This is accomplished by using two connections from each SN to its corresponding DN. One of these connections is inhibitory while the other is excitatory. The inhibitory connection has a longer time delay than the excitatory connection. Initially, a pulse from an SN will arrive over the

excitatory connection, which is weighted so that this causes the internal potential to increase to or above the threshold level of the DN, also referred to as a threshold event. After this threshold event, the inhibitory synapse delivers its contribution and the DN internal potential is decreased. The DN is essentially inhibited because the next excitatory pulse will not contribute enough to the DN internal potential to cause a threshold event in the DN. The DN remains in this inhibited state while there are no positive changes in its input. However, if the input to SN is increased, SN will fire more frequently (Figure 5-3). Because of the time delay between SN and DN, one or more pulses will arrive over the excitatory connection before they have had a chance to propagate along the inhibitory connection, thus causing a threshold event in the DN. DN group neurons are given a fairly high, i.e. fast, leak rate to keep inhibition from building up to a point where no amount of excitatory input could cause a threshold.

The next layer is the Motion Neuron (MN) layer. The DN layer makes connections onto the MN layer (Figure 5-3), so that a given neuron in the MN layer will connect to two neurons in the DN layer. The time delay of these two connections is such that a threshold in a DN followed by a threshold in an adjacent DN will cause a threshold in the corresponding MN. Since the time delays of these connections are obviously not equal, this implies that each MN group has a preferred direction (Figure 5-3). This is indeed the case and is further exploited to decompose motion information. We can determine not

only in which quadrant a change occurred, but also in which direction the change is moving.

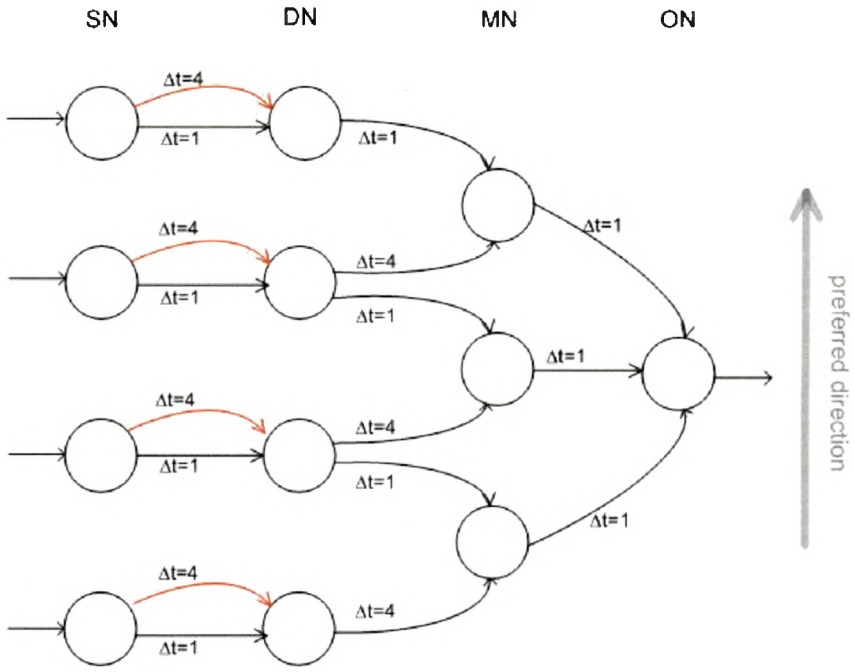

Figure 5-3 Simple Motion Detection Circuit

Depiciton of a simple motion detection circuit showing connection and time delays between network layers.

The next layer sums the MN output into an output neuron (ON) that corresponds to the region and direction of motion. In a full network (Figure 5-4) eight output neurons are produced. Each of the eight represents motion along a Cartesian coordinate axis in a particular direction (Figure 5-4), such as:

1. –X →
2. -X ←
3. +X →
4. +X ←
5. -Y ↑
6. -Y ↓
7. +Y ↑
8. +Y ↓

Since each ON sums the input of several MNs, the firing rate of the ON determines how quickly the MN layer neurons are firing in succession. This also determines how fast the motion is in the visual field. Actually, it provides a vector component velocity since direction information is expressed as well.

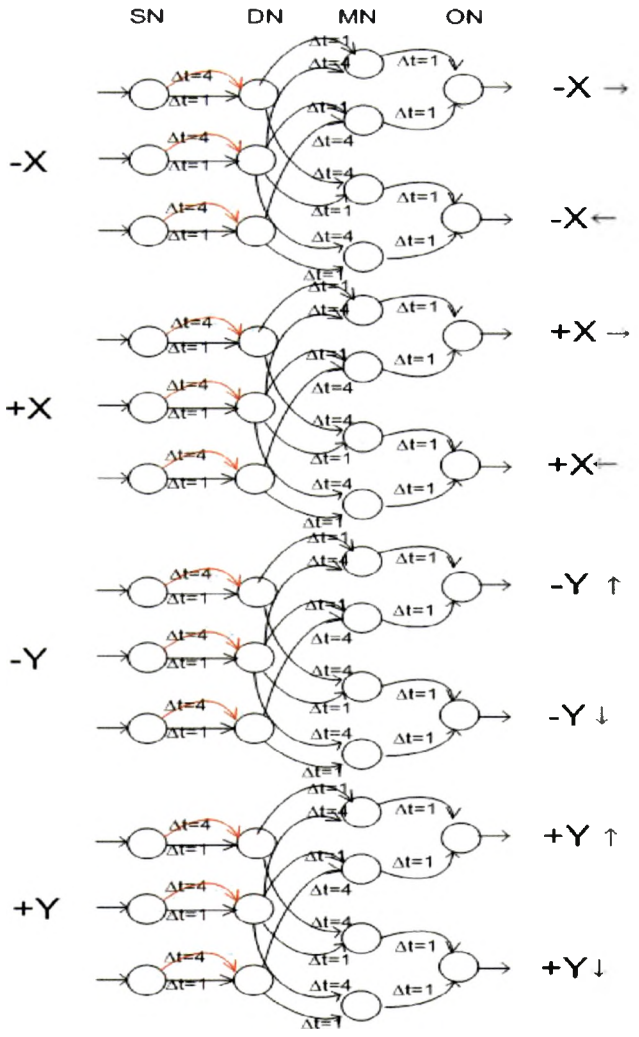

Figure 5-4 Full Motion Detection Network

A full motion detection network for a 6 x 6 pixel visual field where labels on the left show which cartesian coordinates the SN layer neurons map to while labels on the right side denote the coordinate axis and direction of motion.

5.1.2 Sensory Input

As mentioned previously, the image data is already pixilated. It is not necessary to segment analog photonic sensory input into discreet data pulses [11, 9]. Input in

converted to a grey scale image raster of square dimensions. Pixels from this raster are summed by rows and columns (Network Architecture and Figure 5-2 SN Coverage Overlap). Initially this sum was dived by the number of pixels yielding the average pixel value on a scale of 0 to 255. This approach works well in the case of a black background, where the only image data was that of the object being tracked or when the input area was very small (approximately 16 pixels), but yielded poor results when a background image was provided. There was no longer enough difference between a column or row containing the tracked object and one not containing the tracked object from the previous time step to obtain a response in the DN layer. This problem is solved by a process referred to here as exponential scaling, in which the following transformation is applied to the pixel vale p:

$$p' = 2^{\frac{p}{\sigma}} \qquad \textbf{(Equation 5-1)}$$

where σ is a scaling factor and p' is the resultant exponentially scaled pixel value. The scaled pixel values are summed for a column or row and the log is taken. This value v becomes the new input value i.e. input synapse pulse amplitude value (Equation 5-2).

$$v = \log_2\left(\sum_0^N p'\right) \qquad \textbf{(Equation 5-2)}$$

This guarantees that even small positive differences in pixel values dominate the column and make a marked change in the final SN input value v. Exponential scaling resembles gamma correction used in monitor technology, where pixel brightness is scaled to match the human visual perception of brightness.

The visual field is continually sampled in neural network time, known as uni-time. A unit of uni-time is often referred to as a network tick in the neural network. For each network tick relative to the network time the visual field is sampled and fed in as network input. This allows us to talk about a sample rate in network time as well as in real time or rather system time. Also note that the sample rate is synonymous with frame rate since a single input from the visual field is the same as a frame in video. With a 1 frame per network tick sample rate we have achieved a real time frame rate of 11 frames per second for a 128 x128 sample area in a 480 x 480 pixel image.

5.1.3 Extension To Three Dimensions

With regard to SN layer topology (Figure 5-1) input can be represented in a different way. It can be represented as a shadow from the 2D plane of the visual field projected on a 1D line of SN layer neurons, where the movement of a shadow projected by a moving object along our line of sensory neurons is tracked. This is not exactly what is happening during pixel summation and manipulation but it is a convenient model. Therefore, it would be highly plausible to extend our visual tracking network to handle three dimensional tracking. In this case, it would be necessary to define a convenient input

conveying the distance of the visual field from the target object. A wonderful candidate for this input is stereoscopic data gained from having two input devices taking visual input at some offset from each other. This is the primary mechanism by which humans and indeed most animals with at least two eyes judge distance.

5.1.4 Static Visual Field and Non-static Visual Fields

Visual tracking is dealt with in two cases: a static visual field and a non-static visual field. The static visual field is the simplest case, which provides a fixed view of the world. It can be thought of as a stationary camera observing a scene. What the camera can see is synonymous with a visual field. Things move within the scene but the scene itself does not move. In the non-static case a visual field moves within a larger world. So the scene being viewed is constantly changing as the visual field moves within the world. This is the standard case that people experience every day via their eyes. Since this project is not connected to a camera or other external input devices, we use an image as our world and move our visual field within this larger image. For this reason there is no image distortion during camera motion with which to deal. Movements are instantaneous within the world. However, this constant changing of the scene has to be dealt with and will be discussed in depth.

5.1.5 Biological Foundation

Most animals except for humans and other evolved primates cannot respond to an object unless it is moving (Kandel et al 1995). This implies that the ability to detect motion is fundamentally tied to the vision system.

Much of what we are discussing deals with a visual field, which in our case is synonymous with the retina. Therefore, we will discuss some details of the retina. The retina is an extension of the brain. It consists of a layer of photo-receptive neurons and layers of their supporting cells which line the back of the eye. (Figure 5-5 Illustration of the human retina)

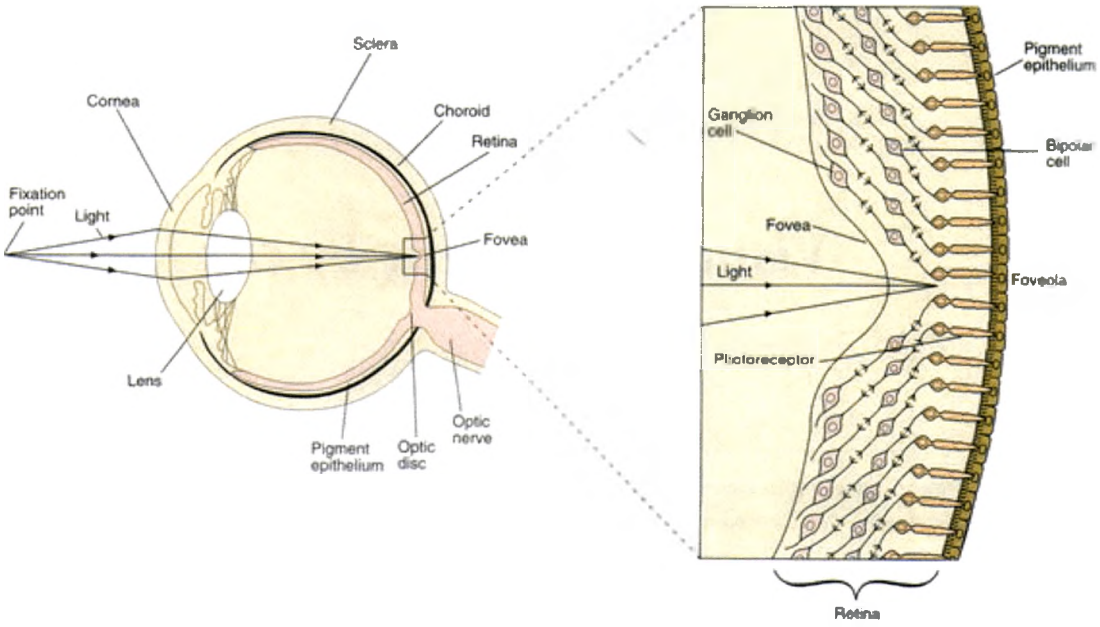

Figure 5-5 Illustration of the human retina

Contrary to what one might think, the photo-reactive neurons are actually at the back of the retina farthest from the light. Other neurons which receive signals from the photo-receptive cells and carry them out of the eye are in front of these cells. Unlike neurons in the rest of the body, these supporting neurons, bipolar and ganglion, are primarily transparent due to their lack of myelin coating, which is a fatty material covering the axons of most neurons. Consequently, light traveling to the back of the eye must go through these neurons, which means that the image is somewhat distorted when it reaches the photo-receptive neurons.

In most animals there is a much higher concentration of photo-reactive neurons in and around the center of the eye. This region is known as the fovea. This is the region of highest visual acuity in most animals. In humans and other evolved primates there is a small pit-like formation in the center of the fovea where the supporting neurons are angled out of the way and light has a more direct path to the photo-receptors. In this region there is much less image distortion. This small pit area, also known as the foveola has the highest acuity for the human eye. Most animals have a foveal region which contains a higher concentration of photo-receptors but only humans and other evolved primates have a foveola. This area is essential to our advanced ability to process shapes, images, and patterns, such as in reading.

In humans and most other primates image data goes from the retina to the primary visual cortex via the lateral genticulate nucleus (LGN). Figure 5-6 scanned from page 429 Essentials of Neuroscience and Behavior.

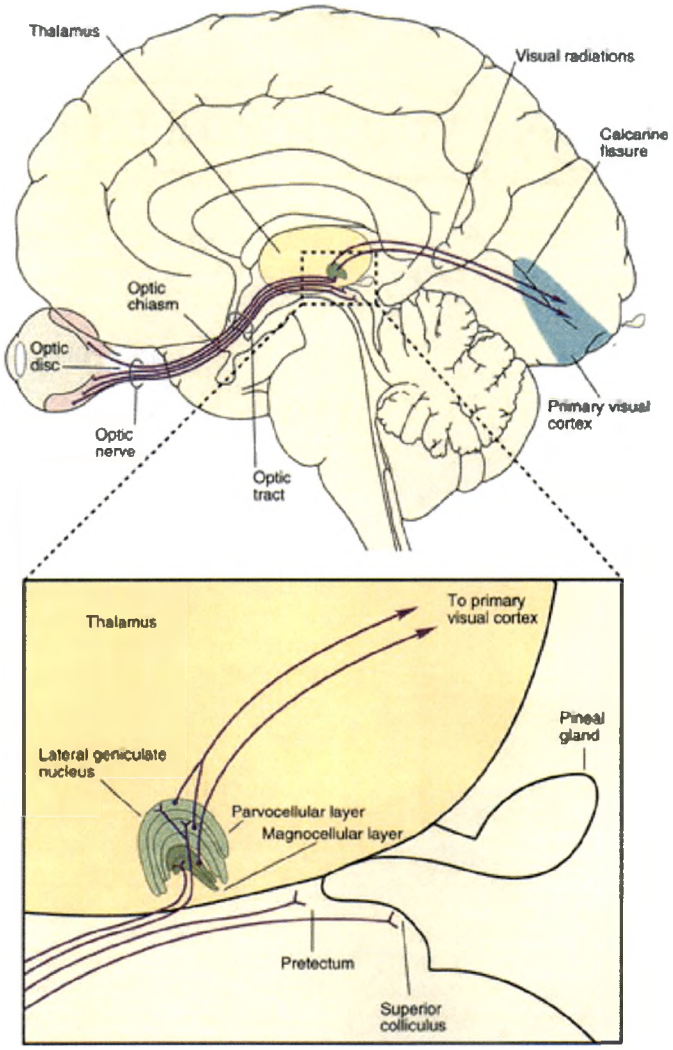

Figure 5-6 Diagram of Projection from Retina to Visual Cortex, and Midbrain.

Information also goes to the midbrain which contains the Pretecum and Superior Colliculus. The LGN and the visual cortex are the only areas that process image information for perception. The pretecum area is important for reflexes of the pupil, while the projection to the superior colliculus mediates eye movements that are guided by visual information. (Kandel et al 1995)

Information is carried to the cortex along several simultaneous pathways. Psychological evidence shows that color, motion, depth and form are carried in various combinations along three parallel visual pathways. (Figure 5-7)

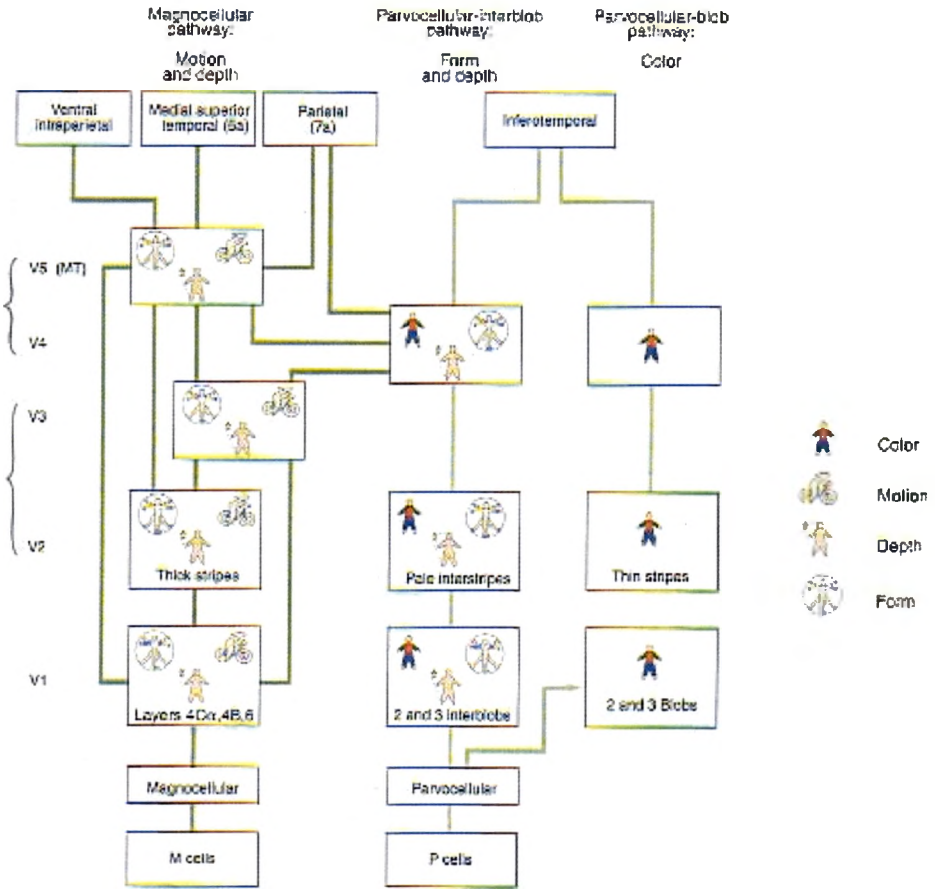

Figure 5-7 Three Major Parallel Visual Pathways

In addition there are parallel simultaneous pathways or channels that deal with image intensity. The pathway of photo-receptors that respond to the presence of light is separate from those that respond to the absence of light. These pathways are separate at least until they reach the midbrain and the LGN. Figure 5-8 (Shepard 2004) shows parallel paths for photoreceptors that respond to positive and negative changes. This can

be imagined as two separate pathways, where one caries the image and one caries the negative image.

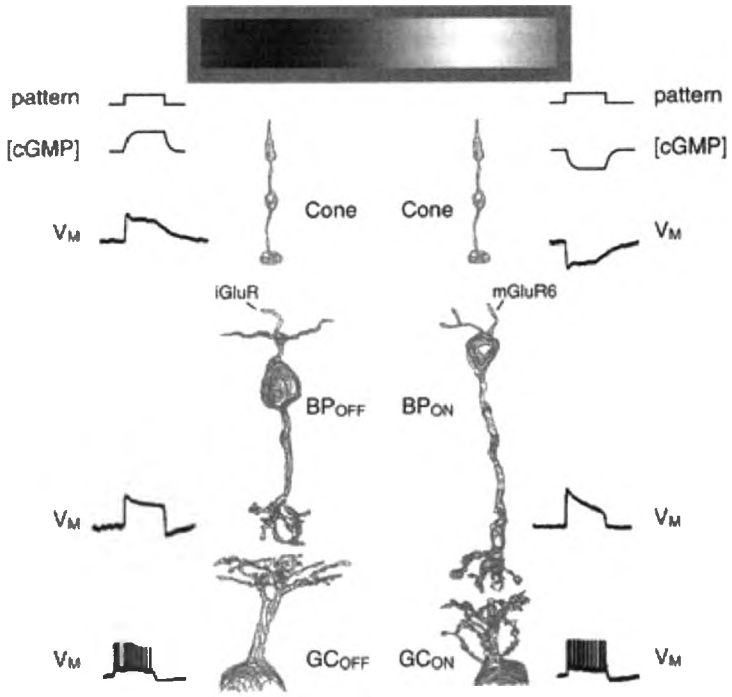

Figure 5-8 Simultaneous positive and negative pathways

The structure of the visual cortex itself is highly complex. Layer four of the visual cortex, which is a layer in the cortex composed of loosely spaced pyramidal neurons imbedded in a densely packed region of inter neurons, is split into four distinct layers, each containing a different morphology of neurons (Figure 5-9). Spiny stellate cells, which dominate the inter-neuron population, are excitatory, while smooth stellate cells are inhibitory. This provides a rich tapestry in which many possible time delay neuronal patterns may be found.

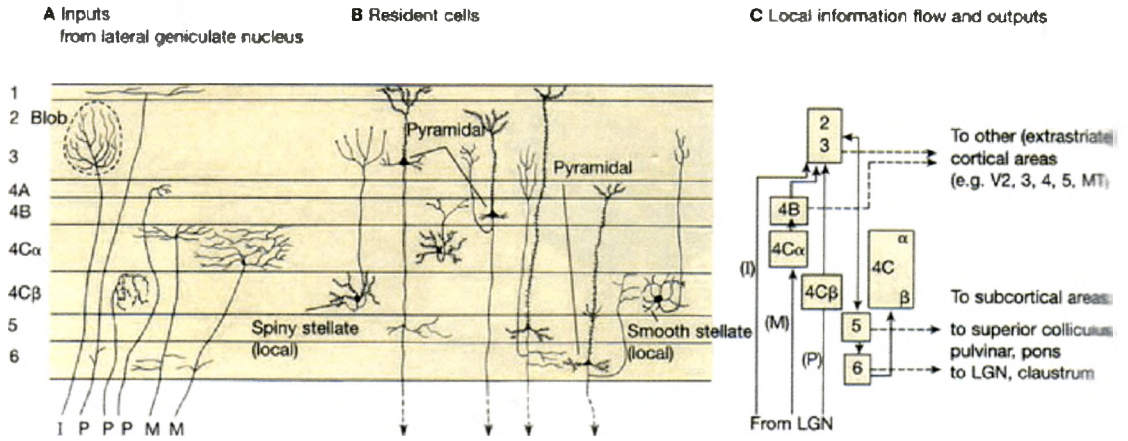

Figure 5-9 Visual Cortex Layers and Neuron Population

Furthermore the primary visual cortex is divided into vertical columns separated by regions known as blobs. Columns react to stimulus in an orientation in a given segment of the visual field, while blobs do not. Blobs are concerned primarily with color. Columns consist of collections of stellate and other inter-neurons centering around one or more pyramidal cells. Lateral connections between these columns exist. (Figure 5-10) Cells in different columns of the visual cortex that respond to stimuli with the same axis of orientation are inter-connected. These orientation columns are grouped together in hyper columns, where a number of distinct orientations are represented. An additional grouping also exists, in which hyper-columns and blobs are grouped together in ocular dominance columns. These ocular dominance columns are involved in aspects of stereoscopic vision. Each column receives input from either the right or left eye and

maps to a specific region of the retina for that eye. Ocular dominance columns are interlaced left and right eye throughout the visual cortex.

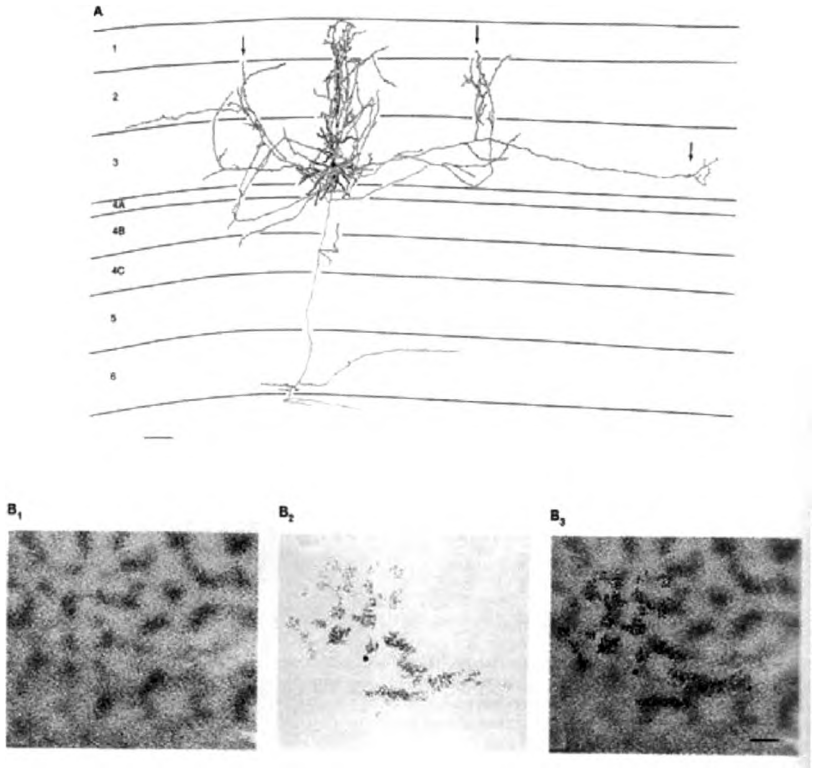

Figure 5-10 Evidence of Functional Columns

A. A camera lucida reconstruction of a pyramidal cell injected with horseradish peroxidase in layers w and 3 in a monkey. Several axon collators branch off the descending axon near the dendritic tree and three other clusters (arrows). The clustered collaterals project vertically into several layers at regular intervals, consistent with the sequence of functional columns of cells. (From McGuire et al. 1990).

B. Cells in different columns of the visual cortex that respond to stimuli with the same axis of orientation are interconnected. 1. A section of cortex labeled with 2-deoxyglucose shows patterns of stripes representing activity of cells in the columns that

respond to stimulus with a particular orientation. 2. Micro-beads injected into the same recording site are taken up by the terminals of neurons and transported to the cell bodies. Reconstruction of the distribution of bead-labeled cells in the same region is visualized in the same section. 3. Superimposition of 1 and 2. The clusters of bead-labeled cells lie directly over the 2 deoxy-glucos labeled areas, showing that groups of cells in different columns with the same axis of orientation are connected. (From Gilbert and Wiesel, 1989) (Kandel et al 1995)

This has been a highly abridged synopsis of the visual pathway highlighting areas of interest.

5.1.6 Motion Detection

Using the network described above we can detect motion in a visual field. This is a general behavior from which pursuit motion and micro-saccades arise. But it is also a behavior on its own. Network output should move the point of interest (static) or move the center of the visual field (non-static) toward the area where the motion is occurring even if the motion is confined to a small area. In mammals this behavior serves to move the point of interest into the foveal region (the center) of the retina for more detailed examination. Most mammals have a higher concentration of photo-reactive cells in the center of their retina.

The problem of motion detection and object tracking has been much studied with a fairly high degree of success and is currently still a highly pursued research topic. However, most solutions do not involve neural networks.

Many approaches center on the concept of a motion field, which is a two dimensional field in which each point is assigned a velocity vector giving the direction, velocity, and distance from the observer. The two main approaches to constructing these motion fields are optical flow where possibly all image points are considered and feature point correspondence where only certain key feature points of the tracked object are considered. The later yields a much smaller and more easily managed motion field.

All approaches depend on analysis of the differences between successive images, whether this is for the entire image as in optical flow analysis or only key points of interest.

Neural Network approaches are few and far between, but usually involve recurrent networks. One such approach is presented in "Bayesian Computation in Recurrent Neural Circuits" (Rao 2004). Rao attempts not only motion detection but also edge orientation discrimination. In general he accomplishes this by using a network architecture commonly used to model the cerebral cortex to implement bayesian inference for an arbitrary hidden Markov model. Rao tackles the problem using a probabilistic approach.

5.1.6.1 Motion Detection in a Static and Non-static Visual Field

Using a static visual field, we move a point of interest, a block or crosshair, inside our field based on network output. The visual field remains unaltered. In the non-static case, the entire visual field moves based on network output to follow the object. This leads to an apparent motion of the background scene in the visual field, which can disrupt motion detection.

5.1.6.2 Interpreting Output

Initially the ON layer output was conceived as output to eye muscles, where output pulses, with some simple logic applied, could directly drive the muscle contractions thus moving the eye based on raw network output. But upon further investigation it was observed that we can also very easily determine which quadrant motion is occurring in and thus produce muscle output based on location as well. In other words, the visual field center moves toward the quadrant in which motion was detected. Experimentally, the best results are obtained using a mixture of these two techniques. For localized motion detection, where the object motion is confined to a small region, location based output is relied upon. This allows the movement of the point of interest or (visual field center) toward the area of motion detection even if pursuit motion is not initiated.

The effectiveness in the case of non-static visual fields could be improved by adding zones to network so that the farther the occurrence of motion from the center of the visual

field, the stronger the network response and thus the faster it moves toward the area of motion detection. However, this can magnify the problem of apparent motion of the background. A solution to this problem is proposed in the pursuit section to follow.

In the visual tracking network described here motion detection is inherent to the network structure. To anthropomorphize the process we can say that this is an involuntary action on the part of the network, just as these responses are involuntary in mammals. They are initiated by the onset of stimulus.

Furthermore, it appears that the network response is strongest when the moving object has high positive contrast with its background. As mentioned in the biological foundation section, the human eye simultaneously considers both positive and negative brightness changes while our network only considers positive changes. Consequently, information is received on the leading edge of a bright object i.e. an object composed of high valued pixels, moving on a dark background. If instead a negative change response was incorporated, the network would respond to a reduction in pixel brightness. This respond to brightness reduction would provide information on the leading edge of a dark object moving on a bright background. Consequently, if a bright object moves on a dark background information on the trailing edge would be obtained. If both of these change responses were used simultaneously, information on the leading edge and trailing edge of

either a dark or light object would be obtained. This is a model for rudimentary edge detection generated by simple motion detection.

5.1.7 Pursuit motion

Pursuit motion is a visual motion type in which the eye moves to match the speed of a moving object in order to keep the object foveated, i.e. in the foveal region of the retina where there is a much higher concentration of photoreceptors. This behavior cannot be initiated voluntarily but rather requires a moving object in the visual field.

In terms of the DLIF network, interpreted network outputs move the visual field so as to keep the object centered for a non-static visual field, and track the motion of the object within the field in the case of a static visual field.

Because there is a non-zero span of time for information to propagate across the network, a time lag exists between detection and response. Since the motion of the field slightly over responds to input, the time lag gives the appearance of being somewhat predictive. This increases the likelihood of being able to match velocity with objects that are accelerating or decelerating and gives us the ability to deal with limited occlusion, where the objects are temporarily hidden behind another object.

5.1.7.1 Motion Detection in a Non-static Visual Field

Using a static visual field, a point of interest, a block or crosshair, is moved inside the field based on network output. The visual field remains unaltered. In the non-static case the entire visual field is moved based on network output. This leads to an apparent motion of the background scene in the visual field, which can disrupt motion detection. This is particularly problematic in the case of pursuit motion, even with no background. Assume an object is being tracked with no background and the system overshoots the location of the object when the visual field is moved. An apparent motion in the opposite direction than that of the motion is created. Once this information has had a chance to propagate through the network, network output will dictate that the visual field be moved in the opposite direction to correct for this. Given that durring pursuit motion the object being tracked continues to move while the network is reacting to this apparent motion, the object may move out of the visual field especially if the object is accelerating. If it does manage to move out of the visual field, the network will no longer be able to track it since its only knowledge of the world is what is currently inside the visual field. If there is a background, this problem is magnified. However, this correction overshoot problem can be minimized with minor enhancements to the network architecture.

5.1.7.2 Modifications to Architecture

To combat the correction overshoot problem, inhibitive connections are applied from ON layer neurons to MN layer neurons so that when an output neuron in a given direction fires the opposite direction motion neurons are inhibited. If this connection weight is small then motion neurons in question are not significantly inhibited unless pursuit

motion, where we have high ON response in the directions of pursuit, is engaged. This suppresses the tendency to correct for overshoot during pursuit, and fortunately because of a relatively high leak rate this inhibition does not last very long in network time, so tracking of an object, which has changed direction, can continue.

5.1.7.3 Zones

Tracking accuracy might further be increase with the use of zones. Basically the farther from the center of a visual field the motion is detected the greater the response of the network. This can be done by segmenting the visual field into zones originating at the origin. Each zone has its own set of eight output neurons ON corresponding to direction of motion. When the output is interpreted, the contribution of outer zones is simply weighted more than inner zones, meaning that the visual field moves more in response to motion in an outlying zone than it does to motion in an inner zone. This also gives a finer granularity in determining where the object is, since quadrant and zone are known.

5.1.7.4 Results of Motion Detection and Pursuit Motion Experiments

In this section, we will explore some experimental results obtained by using DLIF visual tracking networks. Each experiment examines the ability of the network to detect motion, move to an area of localized motion, and conduct pursuit motion.

Figure 5-11 depicts an experimental result from a test using a non-static visual field. This experiment incorporates motion detection at its core level and displays the ability of a visual tracking network using ON to MN feedback discussed previously to track the motion of a moving object. Results of this experiment reinforce the use of both interpretations of network output discussed in section 5.1.6.2 In the first four images starting from the top left in Figure 5-11, the visual field is moved primarily based on the location i.e. quadrant in which motion was detected. In images five through eleven visual field movement is dominated by movement based on raw network output. In this experiment, motion of the object is in a straight line starting in the upper left hand corner and stopping three quarters of the way toward the lower right hand corner. The block is accelerating along its trajectory until it abruptly stops. Once the block has stopped the non-static visual field continues to make small movements in and around the object which demonstrates micro-saccadic behavior. Evidence of this can be seen in the last image in Figure 5-11 where the path of the visual field center throughout the experiment is exposed. In this case, the exposed path appears to exactly correspond to the movement of the object. However, the visual field actually stutters back and forth along the line of motion as it makes corrections to keep the object in the center of the visual field. Performance is excellent even with a noisy background. Time, given in network ticks, can be seen in the upper left hand corner of each image.

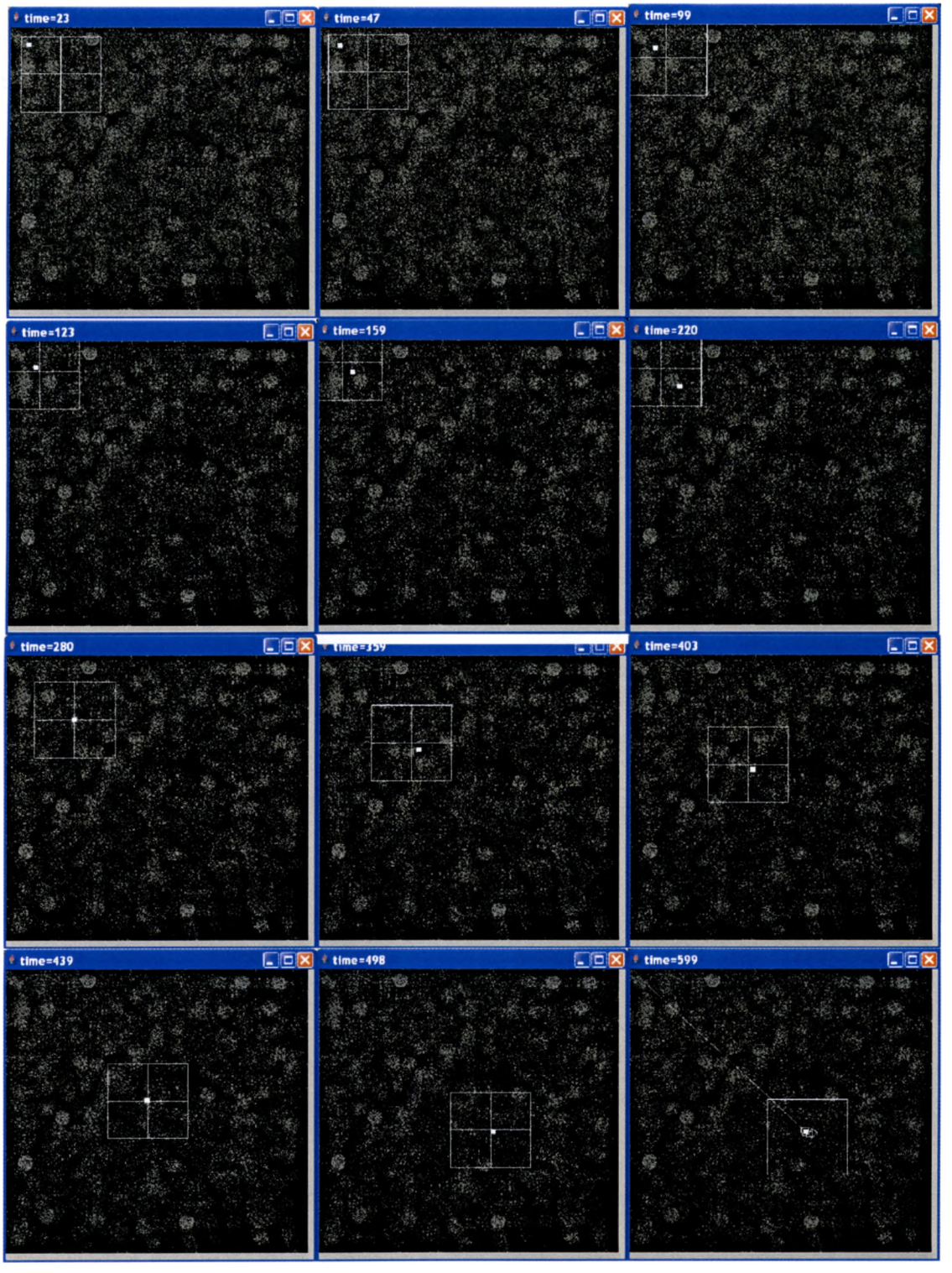

Figure 5-11 Pursuit Tracking on a Noisy Background Using a Non-static visual field

A series of 480 x 480 pixel images showing pursuit tracking of a 6 x 6 pixel block on a noisy background with a 128 x 128 pixel visual field. The block is moving in a straight line from the top left corner toward the bottom right. The block experiences acceleration of 0.008 (pix/tick) pixels per network tick with an initial velocity of 0.02 (pix/tick). In photos 1 through 4 (top left), the visual field is seen to move toward the slow moving block. In images 5 through 7 the visual field overshoots the block which has been accelerating and corrects it path to overtake the block. This marks the onset of pursuit motion behavior. The images that follow show network pursuit motion of the accelerating block. Overshoot response in minimized by inhibitory ON to MN connections. The final image (bottom right) shows the path of the center of the visual field. Note the circular lines around the block in the final image. Block motion ceases at tick 500 but the visual field continues to move around the block, displaying micro-saccadic behavior.

In Figure 5-12 and Figure 5-13 experiments are depicted where an object moves in a curved path, specifically a sinusoidal curve. Initially, since the object is not centered, the visual field moves to center the object using quadrant based network output interpretation. The visual field tracks the object as the object moves in a curved path and finally stops after 500 network ticks. In the bottom middle images of Figure 5-12 and Figure 5-13, the path of the visual field center can be seen. The actual path of the object is a smooth curve which is definitely not the path made by the visual field center. However, the network does manage to keep the object inside the visual field and maintains a fairly accurate tracking of the object. The network handles direction changes

in the arc very well. The final images (bottom right) of Figure 5-12 and Figure 5-13 are overlaid with a graph of the actual path taken by the object.

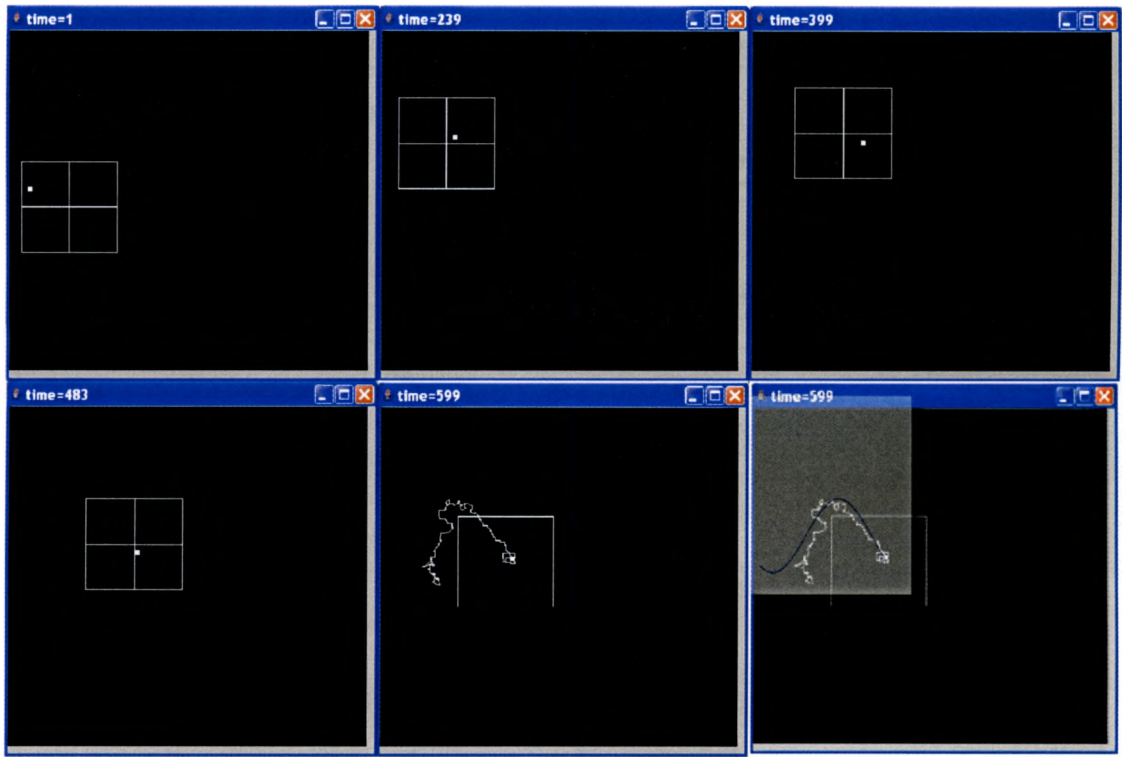

Figure 5-12 Curved path Pursuit Tracking Using a Non-static visual field

A series of 480 x 480 pixel images showing pursuit tracking of a 6 x 6 pixel block with a 128 x 128 pixel visual field. The block is moving in an arc following a sine curve originating from its origin in image 1 (top left). Overshoot response is minimized by inhibitory ON to MN connections. Image five (bottom middle) shows the path of the center of the visual field. Note the circular lines around the block in the final image. Block motion ceases at tick 500 but the visual field continues to move around the block, displaying micro-saccadic behavior. The final image (bottom right) is overlain with the actual path of the object.

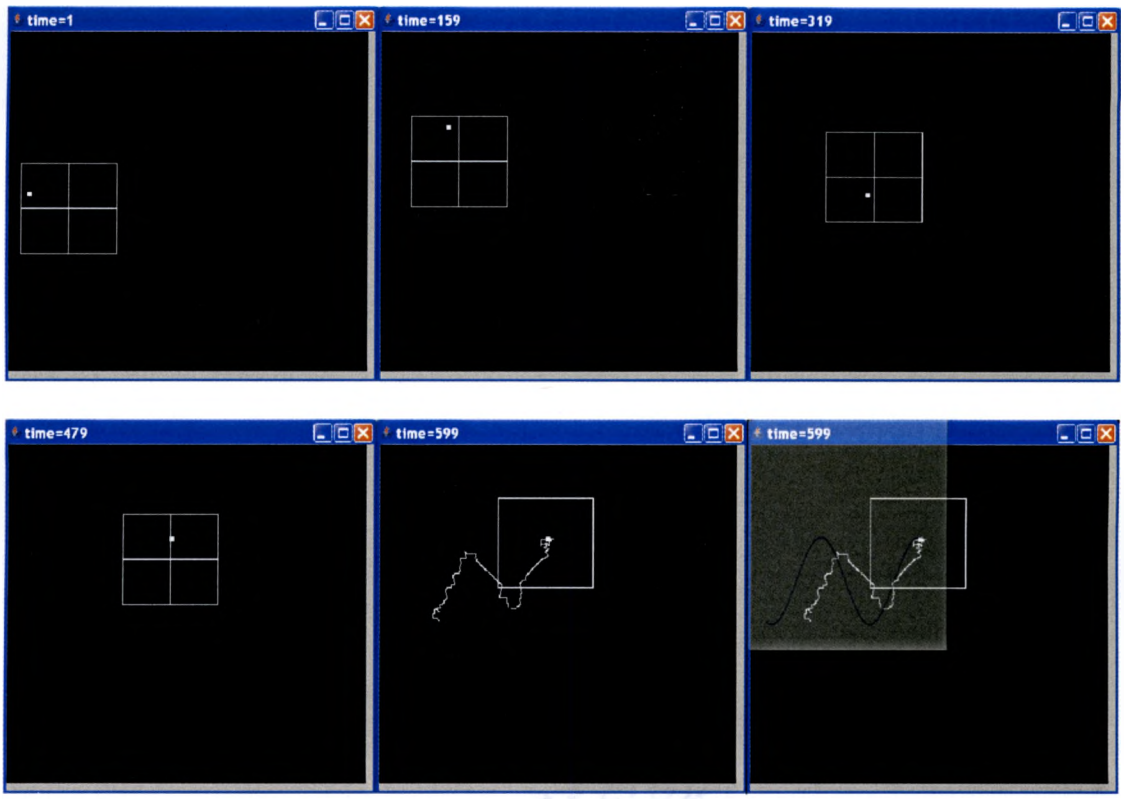

Figure 5-13 Double Curved Path Pursuit Tracking Using a Non-static Visual Field

A series of 480 x 480 pixel images showing pursuit tracking of a 6 x 6 pixel block with a 128 x 128 pixel visual field. The block moves in a sine wave pattern originating from its origin in image 1 (top left). Overshoot response in minimized by inhibitory ON to MN connections. Image five (bottom middle) shows the path of the center of the visual field. Note the circular lines around the block in the final image. Block motion ceases at tick 500 but the visual field continues to move around the block, displaying micro-saccadic behavior. The final image (bottom right) contains is overlaid with the actual path of the object.

5.1.8 Micro-Saccades

Micro-saccades are small involuntary movements of the eye. These are generally unperceivable to humans but can be monitored with machines that track the movement of the pupil, presumably using a more conventional method of motion tracking than employed here. One of the proposed benefits of this sort of small involuntary saccade (movement) is to maintain stimulation when no motion is present in the visual field. Biological neurons adapt to input and their response diminishes. This is generally referred to as neural plasticity. Objects that are being fixated i.e. stared at would eventually cease to elicit a response by photo receptors in the fovea. These micro-saccades may serve to move the image slightly on the fovea and not allow the photo receptors to adapt to the sustained input. Of course, this is only one proposed use of micro-saccades. A visual system based on principles described here would rely on movement in order to detect the presence of objects, which seems to be the model found in nature at least for animals other than humans and higher primates.

The DLIF visual tracking network displays a behavior similar to micro-saccadic motion with non-static visual fields. It is caused by the apparent motion of the background when the visual field moves. The network responds to this motion and the visual field is moved slightly. This second motion generates more apparent motion to which the network responds in like fashion. The center of the visual field seems to dance around in the

scene. These motions, however, tend to gravitate toward areas of high contrast since these areas cause the most network response when the scene is shifted. This is similar to observed behavior in humans (Figure 5-14). However in humans, areas of interest are also affected by higher cognitive functions such as facial recognition.

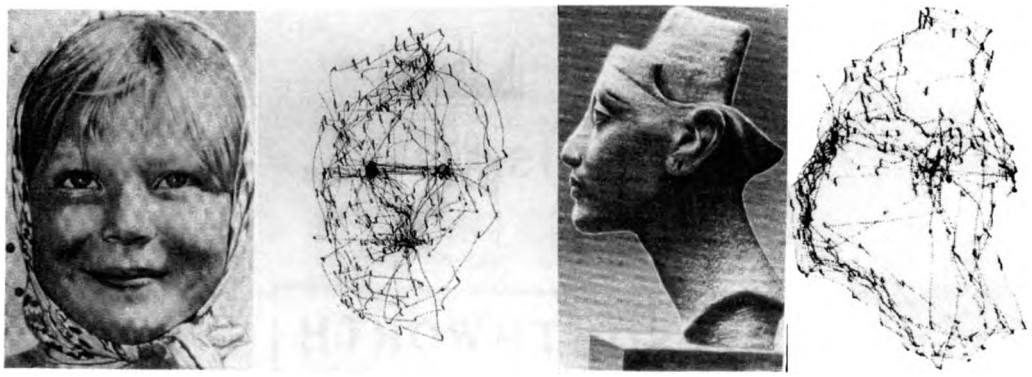

Figure 5-14 Micro-saccades in Humans

In these photos we see of a test subject when instructed to examine the pictures for two minutes. **Left:** "girl from the Volga". **Right:** Head of Egyptian queen Nefertiti, about 1350 BC.

5.1.8.1 Results of Micro-saccadic Behavior Experiments

In the following, we will examine the motion of the center of the visual tracking network in response to various photos and compare them to responses of human test subjects. Since the network only responds to positive changes in brightness the negative images have sometimes been used to simulate the response to negative changes in brightness of

which humans are capable (section 5.1.5 and Figure 5-8). Humans simultaneously respond to positive and negative changes in image brightness. A general similarity is observed between visual tracking network output and human pupil saccades. However, human pupil saccades may be directed by other forces than simple involuntary responses. Higher cognitive functions which would direct points of interest such as facial recognition and emotion analysis are likely influencing pupil saccades. Figure 5-15 shows the response of the visual tracking network when compared to human undirected examination of the illustration. In this case, the intensities have been inverted yielding a negative picture for network input. The path of the visual field center denoted by the rectangular box is shown.

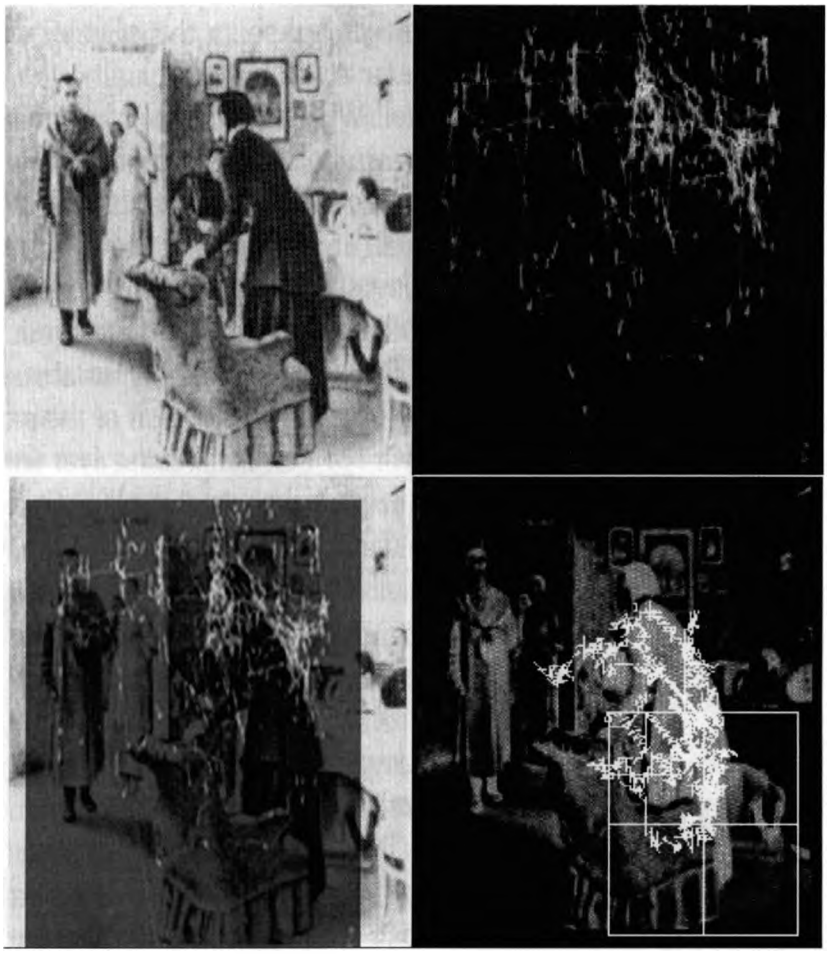

Figure 5-15 Saccades

In this selection of photographs output of the DLIF visual tracking network is compared to collected data from humans. **top left:** original illustration presented to human test subjects. **top right:** pupil movements of a test subject, when directed to examine the illustration. **bottom left:** overlay of pupil movements onto original illustration. **bottom right:** Movements of the visual field center of the DLIF visual tracking network with pursuit motion ON to MN neuron feedback, when exposed to a grayscale image of the illustration for 2099 network ticks. Image intensity values have been inverted.

In another example, a photo entitled "girl from the Volga" is used as DLIF visual tracking network input. In this case, both a positive i.e. normal intensity image and negative i.e. inverted intensity image are used. In the positive image we see a more widespread path while in the inverted intensity image the path is confined to an area around the left eye. The left eye is also the area of highest saccade concentration in the sample of human test data of Figure 5-14. In both positive and negative images the path tends to stay in areas of high brightness relative to the surrounding image.

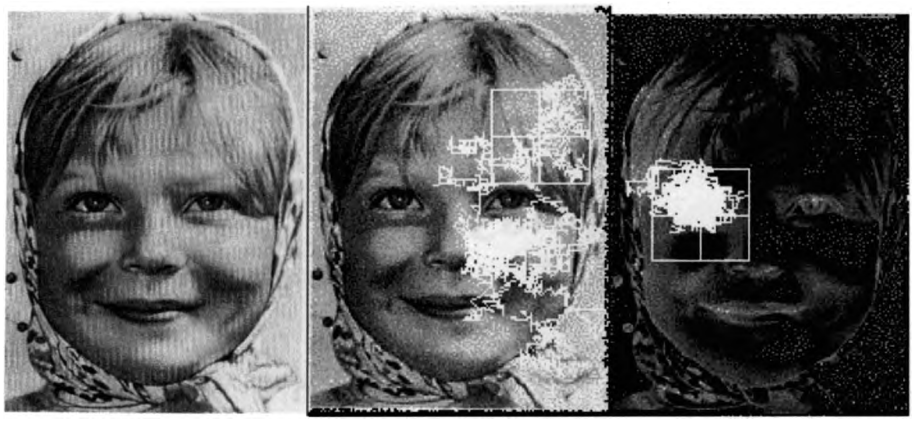

Figure 5-16 Visual Tracking Network Micro-saccades

This sequence of photos illustrates visual tracking network movement of the visual field center on a photo of "girl from the Volga". **left:** original photo. **middle:** Movements of the visual field center of the DLIF visual tracking network with pursuit motion ON to MN neuron feedback when exposed to a grayscale image of the photo for 2099 network ticks. **right:** network movements when exposed to the photo with inverse intensity values.

Micro saccadic behavior can also be seen in pursuit motion experiments (Figure 5-11, Figure 5-12, and Figure 5-13).

5.2 Pattern Recognition

Using action potential computation techniques discussed in section 4.2, patterns can be detected. This technique will be applied to discerning patterns in a visual field that are constants in time. However, it could also be applied to patterns that are not constant in time, yielding the ability to recognize movement patterns or transitions between static patterns. Furthermore, a novel learning technique for memorizing patterns will be discussed.

Originally the concept of DLIF was inspired by the Bifurcating Neuron Network 2 (BNN2) (Lee and Farhat 2002). The BN, however, has no leak ability and depends on coherent modulation. As discussed earlier, DLIF behavior is modified by the presence of coherent modulation but this modulation is not necessary in the system. The BNN2 measured its output in terms of phase shift from the period of coherent modulation. In other words, it is setup so that neurons fire at a constant rate and output is measured in terms of phase shift from regular firing rate on the level of individual spikes. The BNN shares many characteristics with DLIF. For one, real valued input is continuously sampled and mapped to output spike timings in one form or another. Secondly, connections use time delays as well as weights and often require higher order connections.

An analog associative memory was constructed (Lee and Farhat 2002) using BNN2. This is a one layer network consisting of an L x L planar region of BNs. A BN was connected to its neighbor if the neighbor was within a hemming distance. Each BN received direct external input, where the input value was supplied by a pixel value obtained from an L x L image raster. In a network of this kind an attractor can be created where the network output maintains a phase shift pattern proportional to the input pattern. Direct external input from the image raster creates a pattern of phase shifts in the BNN2 where each BN firing time is phase shifted with regard to its normal firing time. When external input is removed the BNN2 maintains this phase shift pattern. It maintains this pattern by ensuring that connections between BNs have time delays and weights such that a given BN receives input from other BNs at the time it should fire, when driven by external input thus driving it to a threshold event at precisely the right time. Likewise, its output arrives as the input to other BNs at precisely the time they should fire thus sustaining the output pattern. This locking of the input pattern is considered an analog associate memory.

Training of this network is referred to as "quasi-online". Initially, there are no connections between the BNs. An input pattern is supplied and the planar region output pattern is studied. Based on this study, connections are manually added. Time delays and weights are set so that the input pattern is locked in as described above. BNN2

suggested an online training method in which multiple weak connections over a spread of time delays are already present. While the training pattern is supplied, the connection strengths for a connection delivering a spike that arrives at the exact time of a BN threshold event will be strengthened. It appears that this method was only suggested by Lee and Farhat and was never implemented (Lee and Farhat 2002). Nevertheless, it is a sound idea. However, it would not succeed in the general case of a multilayer feed-forward network. Logically, only lateral connections can be strengthened in this way since the network must be directly driven by external input to fire. In the DLIF learning method to be discussed in section 5.2.3, this technique is expanded to work in the case of a feed forward network for both lateral and direct connections.

5.2.1 Network Architecture

Network architecture for pattern recognition is very simple (Figure 5-17).

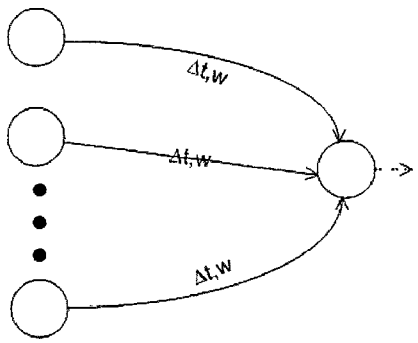

Figure 5-17 Simple Pattern Recognition Architecture

Connections have time delays such that all spikes generated by the input layer SN by the proper input pattern arrive at the pattern recognition (PR) neuron simultaneously (Figure 5-18). Connection weights are set so that the sum of these pulse arrivals cause the PR

neuron to reach threshold level. Inputs may be weighted by their importance in the input pattern, where importance depends on the semantics of the data being used. Of course, if the input pattern to the SN layer is not the correct input pattern or similar to the correct input pattern, the proper relative spike timing needed for the pulses to arrive simultaneously are not produced and the PR neuron does not respond.

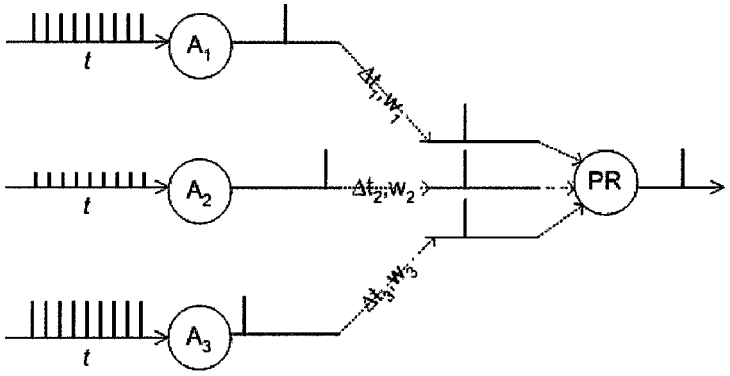

Figure 5-18 Pattern Recognition using Action Potential Computation

External input supplied to each SN (A_1, A_2, A_3), is constant i.e. quasi-continuous pulse inputs of the same amplitude. Inputs drive neurons to their threshold levels as the neuron integrates inputs over time. Neuron A_3 fires first followed by A_1 and then A_2. These pulses experience time delays Δt_1, Δt_2, Δt_3 such that they arrive at PR simultaneously. Weights w_1, w_2 and w_3 are such that these three inputs are able to raise PR to it threshold level, thus producing an output pulse, which signifies recognition.

5.2.2 Sensory Input

In the pattern recognition application described here pixel data is input similarly to the visual tracking application above i.e. raster images of various sizes. However, there is a

one to one correspondence between a pixel and an SN layer neuron. Pixels are not summed by columns or rows. The only manipulation done to the pixel value is a simple scaling to place pixel values inside the SN layer input range.

Sampling is continuous, meaning that input from the visual field is delivered to the SN layer once every network tick. A lower bound on the time it takes the network to detect a pattern is described by the longest time delay of a connection in that pattern recognizer plus the response time of the PR layer neuron, generally one tick. If the pattern is only similar and not an exact match this network response time may extend because the PR has not been stimulated enough on this round of inputs to fire. It may however reach threshold level on subsequent rounds. This behavior leads to an interesting observation. The firing rate of a PR determines how close it is to the pattern in question, the higher the rate the better the match.

One positive attribute of the pattern recognition technique applied in this section in regard to the input data is that the technique depends on relative spike timings and is thus intensity invariant (section 3.4). This applies as long as intensity values, in this case grey-scale pixel values, have been scaled uniformly. The network is noise tolerant in the general sense as this is a general property of artificial neural networks.

5.2.3 Novel Learning Technique

The major difficulty with pattern recognition utilizing action potential computation is that historically time delays had to be inserted by hand (Lee and Farhat BNN2). There was no learning technique capable of finding the correct time delays in a feedforward network. We have developed a learning technique which enables correct time delays to be strengthened through a form of hebbian learning. The technique we have developed also chooses which connections are important to memorizing the pattern, which raises interesting questions related to trait oriented recognition and memory.

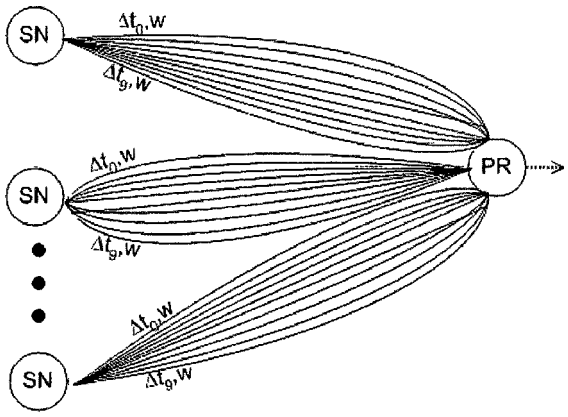

Figure 5-19 Pattern Recognition with Time Delay Learning

The developed technique begins with multiple connections between each SN neuron and the PR neuron. Ten connections are created having time delay 0 through 9 in network ticks (Figure 5-19). Initially all weights are set to a relatively low start value. The initial PR leak rate is chosen to be very low so that even incoming spikes which are loosely

spaced in time contribute to the PR. Due to the low initial leak rate, potential accumulates guaranteeing that PR will fire even if very few spikes arrive simultaneously. In addition, to ensure that PR fires the threshold level is decreased. Every time PR fires, hebbian learning takes place on the incoming synapses. This hebbian learning is divided into two classes: Long Temp Potentiation (LTP) and Long Term Depression (LTD) (Koch et al 1999). In LTP, when a synapse has just contributed to a learning neuron and that neuron fires, the absolute value of that connection weight is increased. In LTD, when a learning neuron has just fired and the connection in question did not contribute, then the absolute value of its weight parameter is decreased. Training continues for a number of sets during which time connections that contribute around firing time have their weights increased and connections that do not contribute have their weights decreased. Then the leak rate of the learning neuron PR is increased. The leak rate increase, in essence makes the ability of PR to detect coincidence more accurate, so that arriving spikes must be more closely spaced in time to cause a threshold event. This process is repeated until the learning rate increases to a predetermined value. Then all the network weights are decreased by the start value. This leaves three groups of connections: connections whose weights are greater than zero, approximately zero, and less than zero. Connections that showed significant LTP are now greater than zero. Inputs arriving across these connections carry traits that significantly contribute to the pattern. Connections that showed significant LTD are less than zero i.e. inhibitory. Inputs from these connections inhibit pattern recognition, meaning that these pixels are

not in the pattern. Synapses undergoing about the same amount of LTP and LTD will be very close to zero. These inputs are insignificant and can be pruned. Of course, the inhibitive input could also be pruned out leaving a recognizer that only looks for pixels that should be there. If the inhibitory connections are left in, patterns that match but have extra pixels may not be recognized. On the other hand, this behavior may be an advantage since the PR firing rate determines how well the pattern matches. PR would likely not fail to recognize the pattern but instead be aware that the pattern match is not perfect.

Mechanisms that change the leak rate, the threshold level, and enable LTP learning are controlled outside the network but could be controlled internally using secondary messenger synapses. These synapses mimic the ability of biological neural networks to change network parameters such as leak rate and threshold level through the use of metabotropic (M) neurotransmitters, which cause internal changes via secondary messenger compounds released inside the cell at the onset of type M neurotransmitter receptor binding. Using this approach, the learning network could mediate its own learning mechanisms.

5.2.4 Results of Pattern Recognition Experiments

The experimental test depicted in Figure 5-20 shows that action potential computation is a viable means for pattern recognition. In this experiment, a network is constructed

which is similar to the network depicted in Figure 5-17 and Figure 5-18, except that it incorporates four groups of SN neurons. Each group connects to one PR neuron, so that there are sixteen inputs (SN) and four outputs (PR). The time delays and weights of these connections are preset so that the pattern is recognized i.e. the PR group responds strongly to the input pattern. Next, the network is exposed to a homogeneous white input pattern, representing maximum input values. The white pattern produces no network response. Next a grey input pattern is supplied which is very similar in value to the gray used in the initial pattern. The homogeneous gray pattern also produces no output. Finally, the initial pattern, to which the network was set to respond, is altered by changing some pixel values, and used as network input. This altered input pattern produces a partial response in the network.

Next, the learning algorithm is tested using the same patterns as were provided for the previous experimental test. The network, however, is altered so that all sixteen (4 x 4) SN layer neurons are connected to a single PR neuron. In this learning experiment, all weights are initially set to the same value (0.02) and the network is trained using the pattern depicted in the far left of Figure 5-21. The leak rate is incremented by 2% every 100 ticks for a period of 2160 ticks. After this training phase, the network is exposed to each pattern for 50 ticks. It responds strongly to the pattern it was trained for, not at all to the homogeneous white and gray patterns, and partially to the altered pattern. These results correspond with the previous experiments where the network parameters were

preset. These results show that the learning algorithm has successfully increased the connections' weights with the proper time delays.

Finally, the learning algorithm is tested on a larger scale using a 64 x 64 SN network. Learning LTP and LTD rates must be adjusted but this experiment is successful as well. Figure 5-22 describes the outcome of this larger scale experiment. Run time for the training of this network is significantly higher since it maintains 40960 connections as opposed to the modest 160 connections established in the 4 x 4 network. However, the initial connection weights remain the same (0.02).

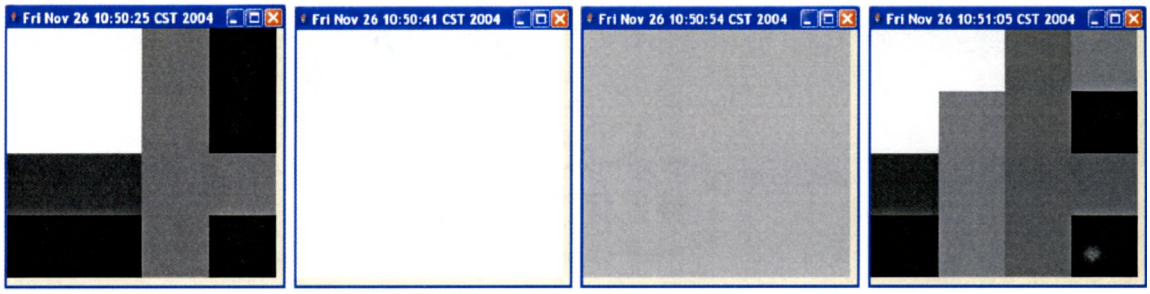

Figure 5-20 Pattern Recognition

This test demonstrates the ability to detect patterns using action potential timing using a 4x4 SN to 4 PR network. Network time delays and weights are set by hand to recognize the far left pattern. After ward the network is exposed to each of the images for fifty network ticks, eliciting the following network responses: **far left:** The network responds strongly to this image which is the training pattern. **middle left:** No response is obtained from the all white (max pixel value) pattern. **middle right:** No response is

obtained from the all gray (50% pixel value) pattern. **far right:** The network displays a partial response to the image, which is an altered version of the training pattern.

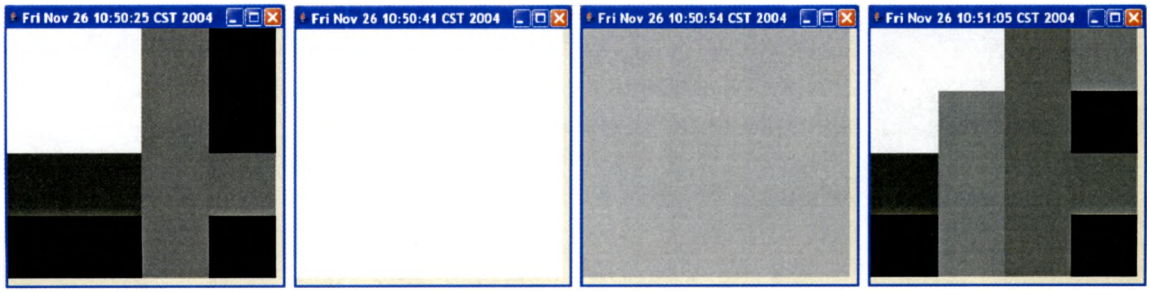

Figure 5-21 Pattern Learning

This test demonstrates learning ability. The pattern on the far left is used for training. A 4x4 SN to 1 PR network is used. Training occurs for 2160 network ticks in which the leak rate is increased by 2% at 100 tick intervals. After ward the network is exposed to each of the images for 50 network ticks, eliciting the following network responses (based on PR output): **far left:** The network responds strongly to this image which is the training pattern. **middle left:** No response is obtained from the all white (max pixel value) pattern. **middle right:** No response is obtained from the all gray (50% pixel value) pattern. **far right:** The network displays a partial response to the image, which is an altered version of the training pattern.

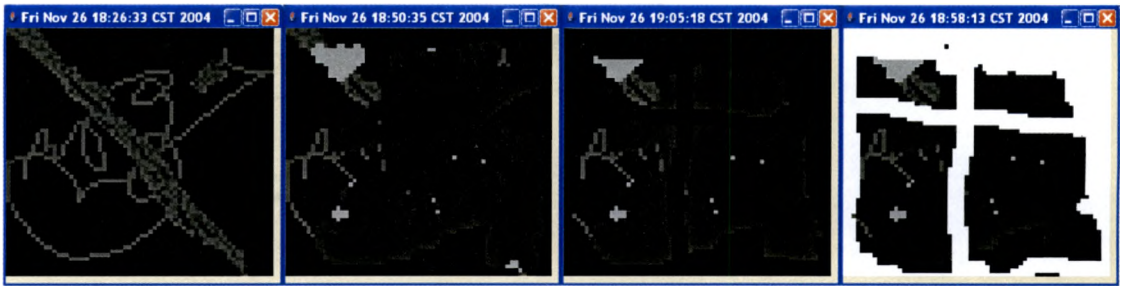

Figure 5-22 Pattern Learning 2

This test demonstrates learning ability. The pattern (64x64 pixel) at the far left supplied as a training pattern to a 64 x 64 SN to 1 PR network. Learning takes place for 2161 ticks during which time the leak rate is increased by 2% at 100 tick intervals. Afterward the network is exposed to each of the images shown for 150 ticks yielding the following results: **far left:** The network responds strongly to this pattern which is the training pattern. **middle left:** The network responds strongly to this pattern, which an altered version of the training pattern. **middle right:** The network responds strongly to this pattern which is a further altered training pattern. **far right:** The network does not respond to this pattern, which is identical to the middle right pattern except that the black area has been inverted to white. **not shown:** The network does not respond to a homogeneous white pattern.

5.3 Self Organizing Map

By extending DLIF concepts to Self-Organizing Maps (SOM), we can construct a SOM which requires little or no training. To do this an input range is mapped to an output map resulting in a fully connected network. Each connection has a time delay associated with

it that is proportional to the distance between the input neuron and the output neuron (Figure 5-23). So that just as in a biological network pulse propagation time between neurons that are close together is less than ones that are far apart. Weights are set to (threshold level)/(number inputs). Thus a neuron fires when it receives a pulse from each of the input neurons simultaneously just as in the pattern recognition networks from the previous section. Each output neuron is inherently configured to react to a particular input pattern of relative spike timings. In a fully connected SOM, uniqueness is not guaranteed, since it is possible for more than one output neuron to react to the same input pattern. However, if the distance (time delay), that an input neuron can extend connections, is limited, this uniqueness problem can be avoided at least for distantly spaced surface map neurons. Network input consists of series of real valued pulse amplitudes as in the pattern recognition networks previously discussed (Figure 5-18). Similar input patterns result in the activation of map surface neurons that are spatially close together. Thus, it is possible that more than one closely spaced map surface neuron may fire from a given input pattern despite the connection limiting proposed above. However, the most highly stimulated neuron will fire first. A winner takes all scenario can be set up by giving each neuron in the map lateral inhibitory synapses onto its nearest neighbors. In contrast, a 'winner takes all' scenario may be a disadvantage since the scenario in which more than one surface map neuron fires at different times yields a spatio-temporal pattern which could be further exploited.

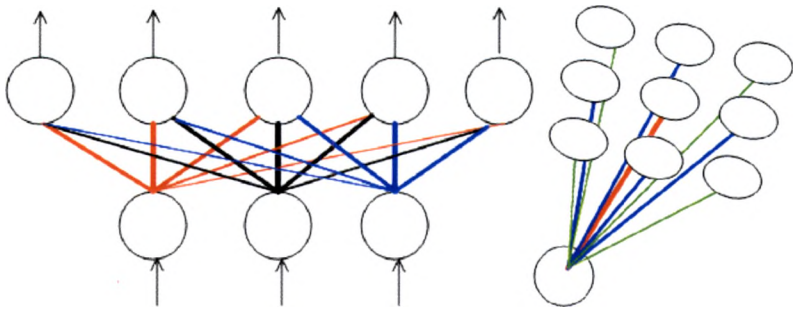

Figure 5-23 DLIF SOM

This figure depicts an illustration of the DLIF Self-Organizing Map showing connections and time delays. Time delays correspond to the distance between neurons in the input layer at bottom and the output layer on top. The weight of each connection corresponds to the threshold value divided by the number of inputs. **left:** A simple 2D SOM slice. The width of each line represents its time delay. Thicker lines correspond to shorter time delays. **right:** This figure depicts the time delays for connections emanating from a single input neuron. Line thickness and color represent the time delays. The thin green lines are the longest time delays, while the thick red line is the shortest.

CHAPTER 6 - CONCLUSION

The basic DLIF neuron model is fundamentally very simple; however, it can exhibit diverse spatio-temporal behavior such as coincidence detection, oscillatory response, amplitude to phase conversion, pulse coupling, and synchronization. Two basic types of input structures have been examined, quasi-constant (incoherent) i.e. unstructured and periodic (coherent), each of which is capable of invoking diverse behavior. We have shown DLIF neurons are capable of amplitude to phase conversion and offer a theoretical basis for holographic memory paging.

DLIF neuron networks also have the added dimension of time. Networks of DLIF neurons use connections that incorporate time delays as well as weight. When these time delays are equal and incoherent input is used, DLIF neuron networks behave similarly to many conventional ANNs. However, temporal aspects of the DLIF neurons give them the ability to exploit temporal coding as well.

An example of one way in which temporal aspects can be exploited is provided by the DLIF visual tracking system. The network that comprises this system uses amplitude to phase conversion and coincidence detection to accomplish the difficult task of motion detection and the tracking of moving objects. It mimics biological systems in its apparent

ability to detect motion, conduct pursuit motion, and exhibit micro-saccadic behavior. More specifically, it attempts to center an object in its visual field by moving the field, and maintaining the centering. When the visual field moves relative to the background, an apparent motion of the background is generated. Small position corrections of the visual field are induced based on this apparent motion. These small position corrections are very similar to the small involuntary eye movements or micro-saccades that humans experience.

Pattern recognition further exploits DLIF network temporal aspects to learn and recognize patterns. Patterns are initially represented as real valued amplitudes. Amplitudes are then encoded in the temporal spacing of output spikes i.e. action potentials. DLIF pattern detection is conducted using action potential computation, a technique pioneered by J.J. Hopfield, which has the property of being scale invariant under a uniform scale transformation. In this technique, time delays are applied which guarantee that all or many action potentials generated by the input pattern arrive at a pattern detection neuron simultaneously if the pattern matches or is similar to the correct pattern. These simultaneous arrivals drive that neuron to output. A learning technique is applied which selectively reinforces connections with appropriate time delays and weakens connections with inappropriate time delays.

Experimental results demonstrate the ability of DLIF networks to perform motion detection, pursuit motion, and display micro-saccadic behavior. The first experiment

tracks the motion of a white pixel block moving and accelerating in a diagonal line on noisy background. This experiment yeilds exelent results. Initially the block is moving very slowly and not centered in the visual field. The field moves to center itself on the accelerating block and maintains this centering to a high degree of accuracy throughout the experiment. After the block has stopped moving the network displays micro-saccadic behavior by making very small movements in and around the block. The next two experiments evaluate the visual tracking system with a moving block in a sinusoidal path in a noise free background. In the first experiment, the block follows a single arced (sinusoidal) path involving one large change in direction. The visual tracking network performance is exellent. Initially the visual field moves to center the block and then attempts to maintain this centering wit a high degree of success. In the second experiment, the block follows the path of a full sinusoidal period involving two large changes in direction over the same latteral distance as the previous experiment. Network performace is very good. The network manages to maintain a high degree of centering, albeit not as high as in the previous experiment. As in the previous experiment the block stops and the visual tracking system exhibits micro-saccadic behavior by moving in and around the stationary block.

In addition, two experiments are conducted that compare visual tracking network micro-saccadic behavior to examples of saccadic behavior data taken from human test subjects. In these two experiments, the visual tracking network is exposed to an image and the

movements of the visual field's center are recorded. The images showing the network saccadic behavior and human saccadic behavior are presented for visual comparison. In general, network micro-saccadic behavior is similar to human saccadic behavior, however, network micro-saccades are more localized. Human saccadic behavior is not completely comprised of involuntary micro-saccades. Higher brain function such as face and object recognition may be influencing saccades. In addition, the human visual system simulatanously consideres both positive and negative changes in image intensity, while the DLIF visual tracking system only considers posative changes. Experimental results are not expected to exactly mimic human behavior considering these major differences but rather seeks to illustrate a general similarity.

Finally, three pattern recognition experiments are conducted. The first experiment verifies the pattern recognition technique to ensure that the action potential computation pattern recognition technique used is a viable means of pattern recognition in DLIF networks. In this experiment, the network is preset to recognize the training pattern. Then network response is checked against the original pattern, a homogeneous white pattern, a homogeneous gray pattern, and an altered version of the original pattern. The experiment is a success. The network responds stongly to the original pattern, not at all to the homogenious patterns, and partially to the altered pattern. The next experiment examines the learning algorithm using the same input patterns as the previous experiment. The network succeeds in learning the pattern and produces the same

responses to each input as the previous experiment. The final experiment examines the learning algorithm on a much larger network. The final pattern recognition experiment is also successful, yeilding similar results as the previous two experiments.

Much of the unique behavior of the DLIF is provided by the leak rate which allows dynamic tuning of coincidence detection accuracy. The leak rate also allows the network to forget input and recover from inhibitory or excitatory input dynamically during operation.

Since the behavior of DLIF neurons more closely matches behavior of biological neurons, DLIF neurons offer a more viable means of directly implementing designs based on the study of biological neural networks. While DLIF neurons still do not come close to the overwhelmingly rich behavior of biological neurons, they implement more temporal aspects than conventional ANNs. Modeling with the DLIF or other neural models that incorporate temporal aspects may lead to a better understanding information representation and processing in biological systems as well as boost the capabilities of artificial neural networks.

APPENDIX A: NETWORK DEFINITION FILE DOCUMENTATION

```
#=================================================


#---------- Documentation---------------------
#================================================
# -- test build script for OONeuralNetwork
# uses Properties class to load text file
# format --
#-------------------
# set the leakRate parameter for all neurons (0.00 to 1.00 ), 1.00 -> no leak
#
#              leakRate = 0.85
#
# set learning parmeters for all synapses
#              LTPRate = 1.005
#              LTDRate = 0.994
#
# set the threshold for all neurons (0.00 to N), default = 20
#              Threshold = 100
#
#
# define input groups for the Network
#               InputGroups = [group (int)], [layer (int)], [weight (float)]
#
#
#               InputGroups tells the network builder where to build input
#               synapses that can be stimulated with an input file.  There may
#               be multiple groups.  Here is an example of defining 2 InputGroups
#               for layer 1. The first has weight .25 and the second weight .99
#
#               InputGroups = 1, 1, .25,  2, 1, .99
#
#                             |---------|  |========|
#
#                               group 1      group 2
#
#               ** group must be unique
#
#-------------------
# define neuron and synapse properties
#
# name :  [type n -- neuron], [layer], [(optional) enable_Learning {t/f} ]
#
#         [(optional) Leak_Rate {0.00 - 1.00} ],  [(optional) Threshold {0.00 - 100.00} ]
#
#        [(optional) Resting_Potential {0.00 to 1.00} ]
#
# name :  [type s -- synapse], [from Neuron], [to Neuron], [weight (0 to 1)],
#
#         [time delay (ticks)], [(optional) enable_Learning {t/f} ]
#
# Note: names must be unique for every line.
#       weight is a double from 0 to 1.
#       time delay is in clock ticks.
#
```

```
#       optional parameters: supplying a dash, -, skips this parameter. Predecessors must
#       be set or skipped e.g. to set the threshold for neuron N11 in layer 3 to 25.34:
#
#                       N11: n, 3, -, -, 25.34
#--------------------
#  Naming Conventions
#
#  file
#       Net-[Test Name][version (int)].txt
#
#       Example: Net-VisualTracking2-1.txt
#
#  Neuron
#
#       N[layer][column]
#
#       Example: N11, N12
#
#  Synapse
#       Learning synapse
#       SL[from N{layer}{column}] [to N{layer}{column}] [index]
#
#       Example: (three synapses from N11 to N22)
#              SL11221
#              SL11222
#              SL11223
#
#-------------------------------------------------------------------------------
```

Example Net Definition File

```
#=======================================
#------ Patter Recognition --------------
#=======================================
# version 1
# First attempt at simple patter recognition
# 4 X 4 field
#---------------------------------------
# input files:
# input_PaternRec1-?.txt
#
#---------------------------------------

leakRate = 0.95

LTPRate = 1.010
LTDRate = 0.999

#Input Group
InputGroups 1, 0, 0.1

#---- Neurons--------

# Sensory Neurons
AA      n       ,0
AB      n       ,0
AC      n       ,0
AD      n       ,0

#Pattern Recog Neurons
PR11    n, 1, t, 0.99, 16

#---- Synapses -------
```

```
# from SN to PR
sAA1    s,      AA      ,PR11   ,0.02   ,       0 , t
sAA2    s,      AA      ,PR11   ,0.02   ,       1 , t
sAA3    s,      AA      ,PR11   ,0.02   ,       2 , t
sAA4    s,      AA      ,PR11   ,0.02   ,       3 , t
sAA5    s,      AA      ,PR11   ,0.02   ,       4 , t

sAB1    s,      AB      ,PR11   ,0.02   ,       0 , t
sAB2    s,      AB      ,PR11   ,0.02   ,       1 , t
sAB3    s,      AB      ,PR11   ,0.02   ,       2 , t
sAB4    s,      AB      ,PR11   ,0.02   ,       3 , t
sAB5    s,      AB      ,PR11   ,0.02   ,       4 , t

sAC1    s,      AC      ,PR11   ,0.02   ,       0 , t
sAC2    s,      AC      ,PR11   ,0.02   ,       1 , t
sAC3    s,      AC      ,PR11   ,0.02   ,       2 , t
sAC4    s,      AC      ,PR11   ,0.02   ,       3 , t
sAC5    s,      AC      ,PR11   ,0.02   ,       4 , t

sAD1    s,      AD      ,PR11   ,0.02   ,       0 , t
sAD2    s,      AD      ,PR11   ,0.02   ,       1 , t
sAD3    s,      AD      ,PR11   ,0.02   ,       2 , t
sAD4    s,      AD      ,PR11   ,0.02   ,       3 , t
sAD5    s,      AD      ,PR11   ,0.02   ,       4 , t
```

APPENDIX B: SOURCE CODE

Neuron.java

```
package ooneuralnet;
import java.util.*;
/**
 * <p>Title: OONeuralNet</p>
 * <p>Description: Object Oriented Nerual Network Model</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Lon Risinger
 * @version 1.0
 */
/**-----------------------------------------------------------------
 * ********************** Change log ****************************
 * -----------------------------------------------------------------
 * +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * 8/2/2003 - Started to do adds and changes for simple hebbian
 * learning in:
 *      fire()
 * +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *
 * -----------------------------------------------------------------
 */


public class Neuron {

  /**
   * Name of this Neuron
   */
  public String name;

  /**
   * Universal clock time.
   */
  public int uniTime;

  /**
   * Time of next event
   */
  public int eventTime;

  /**
   * time of duration of recovery state
   */
```

```
private int recoverTime;

/**
 * State of Neuron
 * Denotes the current state of the Neuron.  ('integrate', 'fire', 'recover')
 * @todo: replace with a Constant object.
 */
public NeuronState state;

/**
 * List of Dendrites attached to this Neuron.
 */
public List dendrites;

/**
 * The axon for this neuron
 */
public Axon axon;

/**
 * The threshold for this Neuron
 */
public double threshold;

/**
 * Current potential of cell (below threshhold)
 */
private double potential;

/**
 * @param restingPotential - potential of this neuron when at
 * rest.  Potential gets set to this initially and during recovery.
 */
public double restingPotential = NeuronConstants.restingPotential;

/**
 * rate at which potential leaks.
 * potential = potential * leakrate
 */
private double leakRate = NeuronConstants.leakRate;

/**
 *  @param learn - enables or disables learning for this neuron.
 *  Learning is hebbian and takes place in the synapse.
 */
private boolean learn = false;

/**
 * show internal potential and states.
 */
private boolean show = NeuronConstants.showInternal;

/**
 * debugLearn
 */
public boolean debugLearn = NeuronConstants.debugLearn;

public Neuron()
{
init();
}
public Neuron(String name)
{
  this.name = name;
```

```
    init();
  }
 private void init()
 {
   this.axon = new Axon();
    this.dendrites = new Vector();
    this.threshold = NeuronConstants.defaultThreshold;
    this.setPotential( restingPotential);
    state = new NeuronState();
    this.recoverTime = NeuronConstants.recoveryDuration;
 }

  public void addPreSynapse(Synapse pre)
  {

    boolean foundOne = false;
    //check to see if dendrites list is empty
    if(dendrites.isEmpty()) //add a dendrite and Synapse
    {
      Dendrite tempd = new Dendrite();
      tempd.synapses.add(pre);
      dendrites.add(tempd);
    }
    else //check to see if we already have a dendrite for this neuron
    {
      Iterator di = dendrites.iterator();
      Dendrite dend;
      //Synapse syn;
      while(di.hasNext())
      {
       dend = (Dendrite)di.next();
       Iterator si = dend.synapses.iterator();
       while (si.hasNext()) //iterate through synapses in dendrite
       {
         Synapse syn = (Synapse) si.next();
         // if the new synapse and an existing synapse are both comming
         // from the same neuron add it to this dendrite.
         if(pre.presynapticRef != null) //check for null
         {
           if (pre.presynapticRef.equals(syn.presynapticRef))
           {
             //add it to this dendrite
             dend.synapses.add(pre);
             foundOne = true;
             break;
           }
         }
       }
      }
      if ('foundOne)   // didn't find any -> add one
      {
        Dendrite newDend = new Dendrite();
        newDend.synapses.add(pre);
        dendrites.add(newDend);
      }
    }
    if (NeuronConstants.debug) System.out.println("Neuron: addPreSynapse adding"+
    " synapse "+ pre.getName() +" = "+pre.toString()+ " to Neuron "+this.name+"="
    +this.toString());
  }
  public void removePreSynapse(Synapse pre)
  {
    if( this.dendrites.isEmpty()) return;
    else
    {
```

```
        Iterator it = this.dendrites.iterator();
        while(it.hasNext())
        {
          Dendrite d = (Dendrite) it.next();
          if(d.synapses.contains(pre))
          {
            // remove synapse
            d.synapses.remove(pre);
            // if dendrite empty remove dendrite
            if(d.synapses.isEmpty())
              it.remove();
          }
        }
    }
  }

  public void addPostSynapse(Synapse post)
  {
    post.presynapticRef = this;
    axon.synapses.add(post);
  }
  public void removePostSynapse(Synapse post)
  {
    post.presynapticRef = this;
    axon.synapses.remove(post);
  }

  /**
   * Function that drives time events.
   * It is called for every clock tick of the universal clock
   */
  public void tick()
  {
    if (NeuronConstants.debug) System.out.println(name + "enter tick uniTime="+uniTime+"
potential="+this.potential);
    this.uniTime++;
    switch(state.getState())
    {
      case(NeuronState.integrate):  //INTEGRATE
      {
        boolean fired = false;
        //if (uniTime >= eventTime) {
          // add potential for pulses that just arrived.
          //this.potential += sumPotential();
          // check threshold
          if (this.potential >= this.threshold)
          // if reached fire and exit
          {
            if (NeuronConstants.debug) System.out.println("About to fire
uniTime="+uniTime+" potential="+this.potential);

            if(show)System.out.println(name + " state=2"+
                        " uniTime="+uniTime+" potential="+this.potential);
            fire();
            fired = true;
          }
          else
          // if not reached then leak Potential
          {
            this.setPotential(potential * this.leakRate);
            //System.out.println(name + " state="+state.getState()+
            //               " uniTime="+uniTime+" potential="+this.potential);

          }
        //}
```

```
        this.setPotential( potential + sumPotential());
        if(show)
        if(!fired)System.out.println(name + " state="+state.getState()+
                            " uniTime="+uniTime+" potential="+this.potential);
      }
      break;
      case (NeuronState.fire):  //FIRE
        if(show)System.out.println(name + " state="+state.getState()+
                            " uniTime="+uniTime+" potential="+this.potential);
      break;
      case (NeuronState.recover): //RECOVER
      {
        // reset potential to ground resting state
        this.setPotential(restingPotential);
        // remove dead pulses
        this.purgeDeadPulses();
        // test for end of recovery
        if (eventTime <= uniTime) state.next();
        if(show)System.out.println(name + " state="+state.getState()+
                            " uniTime="+uniTime+" potential="+this.potential);
      }
      break;

    }//end switch
    //System.out.println(name + " state="+state.getState()+" uniTime="+uniTime+"
potential="+this.potential);
    return;
  }
  /**
   * Neuron fires
   * When the neuron is in the integrate state and threshold
   * has been reached this function creates a Pulse instance
   * and sends it to its axon.  It also marks all apropriate
   * pulse as dead.
   */
  private void fire()
  {
    if (NeuronConstants.debug)System.out.println("fire'  "+this.name+"
potential="+this.potential+" uniTime="+uniTime);
    //transition to firing state
    state.next();
    // produce new pulse and send to axon
    Pulse p = thresholdFunction();
    try{
    axon.sendToSynapses(p);
    }catch (DeadPulseException dpe){dpe.printStackTrace();}

    // mark pulse that have arrived uniTime >= eventime as dead.
    Iterator id = dendrites.iterator();
    Iterator is = null;
    Iterator ip = null;
    //Dendrite d = null;
    while(id.hasNext())  // iterate Dendrite List
    {
      Dendrite d = (Dendrite) id.next();
      is = d.synapses.iterator();
      while (is.hasNext()) // iterate Synapse List
      {
        Synapse s = (Synapse) is.next();
        // send signal to synapse for hebbian learning
        if (this.learn) s.learn();
        ip = s.heldPulses.iterator();
        while (ip.hasNext()) // iterate Pulse List
        {
          Pulse tp = (Pulse) ip.next();
```

```
        if(tp.alive)  // only check live pulses
        {
          if(tp.eventTime <= this.uniTime) tp.kill();
        }
      }
    }
  }
  // transition to recovery state
  state.next();
  // set eventime to get out of recovery state
  this.eventTime = uniTime + recoverTime;

}
public void purgeDeadPulses()
{
  Dendrite d = null;
  Iterator it = dendrites.iterator();
  while(it.hasNext())
  {
    d = (Dendrite) it.next();
    d.purgeDeadPulses();
  }
}
/**
 * marks Pulses held at the synapses of this Neuron as dead
 * if their event time has already passes (eventTime >= uniTime)
 */
/**
 * determines the amplitude of the new pulse
 */
private Pulse thresholdFunction()
{
  // returns pulse of standard amplitude
  return new Pulse(NeuronConstants.defaultPulseAmplitude,this.uniTime+1,this.uniTime);

}
/**
 * Sum of the potential added to this neurons potential at the current uniTime.
 */
public double sumPotential()
{
  double sum=0;
  Iterator di = dendrites.iterator();

  while(di.hasNext())
  {
    Dendrite d = (Dendrite) di.next();
    sum += d.sumPotential(this.uniTime);
  }
  return sum;
}
public void setLeakRate(double leakRate)
{ this.leakRate = leakRate;}

public double getLeakRate()
{ return this.leakRate;}

public void enableLearning(){ this.learn = true;}
public void disableLearning()
{
  System.out.println("disable learning for "+this.name);
  this.learn = false;
}
  public void setPotential( double potential)
  {
```

```
      this.potential = potential;
      if(debugLearn)
        //System.out.println(this.uniTime+" "+this.name+" potential="+potential);
        System.out.println(this.uniTime+"\t    "+potential);
    }
    public double getPotential()
    {
      return this.potential;
    }

   public  boolean isLearnEnabled()
   { return this.learn;}
   }
```

NeuronState.java

```
package ooneuralnet;

public class NeuronState
{
  public static final int integrate = 1;
  public static final int fire = 2;
  public static final int recover = 3;
  private int state;

  /**
   * state is initialized to integrate.
   */
  public NeuronState()
  {state = integrate;}
  public int getState()
  {return state;}
  public int setState(int newState)
  {
    state = newState;
    return state;
  }
  /**
   * increments state
   */
  public int next()
  {
    if (state < recover) state++;
    else state = integrate;
    return state;
  }
}
```

NeuronConstants.java

```
package ooneuralnet;

public class NeuronConstants
{
  /**
   * Number of ticks the recovery cycle takes
   */
  public static int  recoveryDuration = 1;
  /**
   * The rate at which the Potential Leaks from the soma.
   * This is the leak from a Leaky-Capacitor Integrate and fire Neuron
   * Model.
```

```
    * Potential(t+1) = Potential(t)*leakRate
    */
   public static final double leakRate = 0.85;
   /**
    * Resting potential of a non excited Neuron
    */
   public static final double restingPotential = 0.0;

   /**
    * Default Pulse Amplitude
    */
   public static final double defaultPulseAmplitude = 100;

   public static final double defaultThreshold = 20;

   //Learning ---------------------------------
   public static boolean debugLearn = false;
   public static final double defaultLTPRate = 1.005;
   public static final double defaultLTDRate = 0.995;
   public static final double learnSaturation = .2;
   // ------------------------------------------

   /**
    * Show internal states potential etc for neurons and synapses
    */
   public static boolean showInternal = false;

   public static boolean debug = false;

   public NeuronConstants() {}

}
```

Axon.java

```
package ooneuralnet;
import java.util.*;

/**
 * <p>Title: Axon </p>
 * <p>Description: The Axon of a neuron passes pulses to the synapses</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Lon Risinger
 * @version 1.0
 */

public class Axon {

   /**
    * List of synapses that are connected to this axon
    */
   public List synapses;

   public Axon()
   { this.synapses = new Vector();}

   /**
    * Sends generated pulse to Synapses
    * Clones the pulse and sends a copy to every synapse connected to this Axon.
    */
   public void sendToSynapses(Pulse p) throws DeadPulseException
   {
     if ( 'synapses.isEmpty())
     {
```

```
      Iterator si = synapses.iterator();
      Synapse tempS;
      while(si.hasNext())
      {
        tempS = (Synapse) si.next();
        //add a clone to every synapse.
        try{
          tempS.addPulse( (Pulse) p.clone());
        }catch(CloneNotSupportedException cnse){cnse.printStackTrace();}
      }
    }
  }
}
```

Dendrite.java

```
package ooneuralnet;
import java.util.*;

/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 */

public class Dendrite {
  /**
   * List of synapses that are connected to this dendrite
   */
  public List synapses;
  public Dendrite()
  {
    this.synapses = new Vector();
  }
public void purgeDeadPulses()
  {
   Synapse s = null;
   Iterator it = synapses.iterator();
   while (it.hasNext())
   {
     s = (Synapse)it.next();
     s.purgeDeadPulses();
   }
  }
  /**
   * Sums the potential added to the Neuron who owns this dendrite from Pulse
   * that have arrived (eventTime >= uniTime) at the synapses referenced by
   * this dendrite
   *
   * @param unitime Universal Time of calling Neuron
   */
  public double sumPotential(int unitime)
  {
    double sum = 0;
    Synapse s = null;
    Iterator it = synapses.iterator();
    while (it.hasNext())
    {
      s = (Synapse)it.next();
      sum += s.sumPotential(unitime);
    }
    return sum;
```

```
  }
}
```

Synapse.java

```
package ooneuralnet;
import java.util.*;

/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 */
/**-------------------------------------------------------------------
 * ********************** Change log ***************************
 * -------------------------------------------------------------
 * +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * 8/2/2003 - Started to do adds and changes for simple hebbian
 * learning.
 *     - HLPulse
 * +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * 6/6/2004 - adding show output to make tracking pulses easier
 * +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * -------------------------------------------------------------------
 */


public class Synapse {
  /**
   * name of this synapse
   */
  protected String name;
  /**
   * The weight of this connection
   */
  private double w;
  /**
   * Time delay.
   * The time it takes for the pulse to reach this synapse from
   * the presynaptic neuron.
   */
  public int deltaT;
  /**
   * List of Pulses held at Synapse
   */
  public List heldPulses;
  /**
   *  Shows internal workings dealing with the summation/delivery of pulses
   */
  public boolean show = NeuronConstants.showInternal;

 //-----------------------------------
 /**
  * Reference to pre-synaptic Neuron.
  * Used for building the Network.
  */
 public Neuron presynapticRef;

 //-----------------------------------
 //-----------------------------------
 // Learning
```

```
 protected boolean learn = false;
 /**
  * Hebbian Learning Pulse marker
  * Flag that denotes whether a presynaptic pulse has just "arrived"
  * and been added to a postSynaptic neuron potential.  If Postsynaptic
  * neuron fires and this flag is set then hebbian learing should occur.
  */
 protected boolean HLPulse;

 private double LTPRate = NeuronConstants.defaultLTPRate;
 private double LTDRate = NeuronConstants.defaultLTDRate;
 private double learnSaturation = NeuronConstants.learnSaturation;
 //---------------------------------
 public Synapse()
 {
 this.w =0;
 this.deltaT =0;
 this.heldPulses = new Vector();
 }
 /**
  * Add a pulse to the held synapse list.
  * Updates pulse event time value.
  */
 public void addPulse(Pulse p) throws DeadPulseException
 {
    if (!p.isAlive()) throw new DeadPulseException("attempted to add dead pulse to
Synapse");
    p.eventTime += this.deltaT;
    heldPulses.add(p);
 }
 public void purgeDeadPulses()
 {
   Pulse p = null;
   Iterator it = heldPulses.iterator();

   while(it.hasNext())
   {
     p = (Pulse) it.next();
     if (!p.isAlive())
      {
        it.remove();
      }
   }
 }
 public void purgeAllPulses()
 {
   heldPulses.clear();
 }
 /**
  * Adds together the amplitude (potential) of the pulses held at this synapse,
  * whose eventTime is greater than or equal to the Universal time (unitime)
  * of the Neuron they belong to.
  *
  * @param unitime Universal Time of the calling Neuron.
  */
 public double sumPotential(int unitime)
 {
   if(NeuronConstants.debug) System.out.println("synapse="+name+" weight="+getW());
   double sum = 0;
   HLPulse = false;
   Iterator it = heldPulses.iterator();
   while(it.hasNext())
   {
     Pulse p = (Pulse) it.next();
     if (p.isAlive())
```

```
      {
        //if (unitime >= p.eventTime) //ie the pulse has arrived at synapse
        if (unitime == p.eventTime)
        {
          sum += this.getW()*p.amplitude;  //add pulse amplitude to Neuron Potential
          p.kill();            //Mark the pulse dead so it cannot contribute twice
          if(show) // for debugging
          {
            String timeMessage = "  t= ---";
            if(this.presynapticRef != null)
              timeMessage = "  t="+this.presynapticRef.uniTime;
            System.out.println("\tSynapse:"+this.name+" w="+this.getW()+"
deltaT="+this.deltaT
                               +timeMessage
                               +"\n\tpulse delivered - In-Amp:"
                               +p.amplitude+" Out-Amp:"+this.getW()*p.amplitude
                               +"  eventTime="+p.eventTime);
          }
        }
      }
    }
    if (sum > 0) HLPulse = true;
    //if(HLPulse && learn) System.out.println("HLPulse");
    return sum;
  }
  public void setW(double w)
  {
    if (w > 1) this.w =1;
    else this.w = w;
    if(NeuronConstants.debugLearn)
      System.out.println("\t"+this.name + "w=" +w );
  }
  public double getW() {return w;}
  public void setName(String name) {this.name = name;}
  public String getName() {return name;}
  /**
   * implements learning rule.
   * In this case it is simple hebbian learning.  If the HLPulse flag is true
   * and learn() is called this means that the postsynaptic neuron fired
   * immediately after this synapse contributede to its potential.  This
   * synapse's weight is increased by a small factor (Longterm Potentiation).
   * If learn() is called and HLPulse is false then the weight is decreased
   * by a small factor (Longterm Depression)
   */
  public void learn()
  {
    //System.out.println(this.name+" learn() ");
    if(this.learn)
    {
      double temp =0.0;
      if (HLPulse)
      {
        //System.out.println("LTP "+this.name+" learn() ");
        temp = LTPRate * w;
        if (temp > learnSaturation) temp = learnSaturation;
      }
      else
      {
        temp = LTDRate * w;
        if (temp < 0) temp = 0; // do not become inhibitive
      }
      this.setW(temp);
    }
  }
  public void enableLearning(){learn = true;}
```

```
  public void disableLearning() {learn = false;}
  public void setLTPRate(double rate)
  { this.LTPRate = rate;}
  public void setLTDRate(double rate)
  { this.LTDRate = rate;}


}
```

Pulse.java

```
package ooneuralnet;

/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 */

public class Pulse implements Cloneable{
  /**
   * Amplitude of Pulse
   */
  public double amplitude;

  /**
   * Time of next Pulse event
   */
  public int eventTime;

  /**
   * Universal clock time.
   */
  public int uniTime;

  /**
   * Flag that keeps track of whether the pulse should exist or not.
   * true = should exist
   * false = should not exist i.e. can be destroyed.
   */
  protected boolean alive;

  /**
   * everthing initialized to 0
   */
  public Pulse()
  {
    amplitude = NeuronConstants.defaultPulseAmplitude;
    eventTime = 0;
    uniTime = 0;
    alive = true;
  }
  /**
   * This constructor should be used to construct the initial
   * set of pulses used to stimulate the network.
   * In this case eventime is synonomous with deltaT and
   * conveys the time spacing of the pulse in the pulse set.
   * Amplitude is default.
   * unitime is 0.
```

```
   */
  public Pulse(int eventTime)
  {
    amplitude = NeuronConstants.defaultPulseAmplitude;
    this.eventTime = eventTime;
    uniTime = 0;
    alive = true;
  }
  /**
   * initialization supplied
   */
  public Pulse(double amplitude, int eventTime, int uniTime)
    {
      this.amplitude = amplitude;
      this.eventTime = eventTime;
      this.uniTime = uniTime;
      alive = true;
    }

    /**
     * Checks to see if this pulse should exist or not.
     * If false the pulse is dead and should be removed.
     */
    public boolean isAlive()
    {return alive;}

    /**
     * Sets is alive flag to false.
     * This indicates that this pulse is dead and should
     * be removed.
     */
    public void kill()
    {alive = false;}

  protected Object clone() throws CloneNotSupportedException
  { return new Pulse(this.amplitude, this.eventTime, this.uniTime);}

}
```

StimulateException.java

```
package ooneuralnet;

public class StimulateException extends Exception {
  public StimulateException() {
  }
  public StimulateException(String message)
  {
    super(message);
  }
}
```

SynapseOrganizer.java

```
package ooneuralnet;

import java.util.*;

/**
 * Synapse Organizer is a container for organizing multiple groups
 * of synapses for use as an input or output synapse groups.
 */
```

```
public class SynapseOrganizer
{
  /**
   * Active Synapses group.

   * The input synapse group or layer we are
   * currently working with.
   * Key is Neuron Name (String)
   * Value is Synapse object.
   */
  private Map activeSynapseGroup;
  /**
   * Synapse groups
   * This is the Primary data structure.  It is a map of maps
   * where the outer maps keys are groups Integer objects
   * that correspond to the group represented in the Network object.
   * Values are input Synapse maps (see activeInputSyn).  Keys for the
   * inner map are Neuron names (String) and values are Synapse objects.
   */
  private Map SynapseGroups;
  public SynapseOrganizer()
  {
    //this.activeInputSyn = new Hashtable();
    this.SynapseGroups = new Hashtable();
  }
  //==============  Methods for filling Synapse organizer ================
  public void addGroup(int group) throws SynapseOrganizerException
  {this.addGroup(new Integer(group));}

  /**
   * Add an Input Synapse Group.
   * This is synonymous with adding a new set or group of inputs.
   * However you must add each neuron/synapse explicitially using
   * setActiveGroup and then addInputSynapse
   */
  public void addGroup(Integer group) throws SynapseOrganizerException
  {
    if (SynapseGroups.containsKey(group))
      throw new SynapseOrganizerException(" Synapse group already exists");
    else
      this.SynapseGroups.put(group, new Hashtable() );
  }
  /**
   * sets the group of synapses we are working with at this time
   */
  public void setActiveGroup(int group)
  { this.setActiveGroup(new Integer(group));}

  public void setActiveGroup(Integer group)
  {
    this.purgeDeadPulses();
    this.activeSynapseGroup = (Map) SynapseGroups.get(group);
  }
  /**
   * adds a neuron/synapse pair to the active synapse layer or group
   */
  public void addSynapse(String neuronName, Synapse syn)
  {this.activeSynapseGroup.put(neuronName, syn);}
  //=========================================================================


  //==============  Methods for retrieving Synapse objects ===============
  public Synapse getSynapse(String neuronName)
  {
    Synapse s = null;
```

```
    s = (Synapse) this.activeSynapseGroup.get(neuronName);
    return s;
  }

  public Iterator getGroupIterator(Integer group)
  {
     this.purgeDeadPulses();
    Map tempM = (Map) this.SynapseGroups.get(group);
    return  tempM.keySet().iterator();

  }

  public Set getGroupSet(Integer group)
  {
     this.purgeDeadPulses();
    Map tempM = (Map) this.SynapseGroups.get(group);
    return  tempM.keySet();
  }
  //=========================================================================
  //==============  Methods purging Synapse objects ===============
  public void purgeDeadPulses()
  {
    //System.out.println("called SynapseOrganizer:purgeDeadPulses()");
    //System.out.print(".");
      Iterator sgit = this.SynapseGroups.values().iterator();
      while(sgit.hasNext()) //iterate through groups
      {
        Map sgMap = (Map) sgit.next();
        Iterator mapsit = sgMap.values().iterator();
        while (mapsit.hasNext())  // iterate through synapses
        {
          Synapse s = (Synapse) mapsit.next();
          s.purgeDeadPulses();
        }
      }
  }
  //=========================================================================
}
```

SynapseOrganizerException.java

```
package ooneuralnet;

public class SynapseOrganizerException extends Exception {
  public SynapseOrganizerException() {
  }
  public SynapseOrganizerException(String message)
 {
   super(message);
 }


}
```

Network.java

```
package ooneuralnet;
import java.util.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileInputStream;
```

```
import java.io.IOException;
import java.text.NumberFormat;

//for toFile()
import java.io.PrintWriter;
import java.io.BufferedWriter;
import java.io.FileWriter;

public class Network
{
  /**
   * The underlying data structure for this container is a
   * Map of Maps.  The outer Map has keys equal to Layer names
   * (Integer objects 0, 1, 2 , ... )and values containing the inner Maps.
   * The inner Maps have keys corresponding to the names of
   * Neurons (String) and values corresponding to Neuron Objects.
   */
  private  Map layers = null;
  /**
   * Synapse Organizer that holds references to input synapses.
   * Key = neuron name
   * Value = Synapse
   */
  private SynapseOrganizer inputSynapses = null;

  private Map outputSynapses = null;
  public Network()
  {
  layers = new Hashtable();
  }
  public void addNeuron(Neuron n, Integer layer)
  {
    Map l = null;
    if (layers.containsKey(layer)) // layer already exists
    {
      l = (Map) layers.get(layer);
      l.put(n.name, n);
    }
    else // make new layer
    {
      l = new Hashtable();
      l.put(n.name, n);
      layers.put(layer, l);
    }
  }
public Neuron findNeuron(String name) throws NetworkBuilderException
  {
   Iterator li = layers.keySet().iterator();
   Map layer = null;
   Neuron n = null;
   Integer key;
   name = name.trim();
   while(li.hasNext())
   {
     key = (Integer) li.next();
     layer = (Map) layers.get(key);
     if (layer.containsKey(name))
     {
       n = (Neuron) layer.get(name);
     }
   }
   if (n == null) throw new
         NetworkBuilderException("Unable to find Neuron " + name);
 return n;
 }
```

```
public void createInputSynapses()
{ createInputSynapses(0 ,0, 1); }

public void createInputSynapses(int group, int layer, float weight)
{
  if(inputSynapses == null)
  {
    inputSynapses = new SynapseOrganizer();
  }
    try{
      inputSynapses.addGroup(group);
    }catch(SynapseOrganizerException soe)
      {soe.printStackTrace(); }
    inputSynapses.setActiveGroup(group);
    Map inputLayer;
    inputLayer = (Map) layers.get(new Integer(layer));
    String key;
    Neuron n = null;
    Iterator il = inputLayer.keySet().iterator();

    while(il.hasNext())
    {
      key = (String) il.next();
      n = (Neuron) inputLayer.get(key);
      Synapse s = new Synapse();
      s.deltaT = 0;
      s.setW(weight);
      s.setName("input"+group+key);
      n.addPreSynapse(s);
      inputSynapses.addSynapse(key, s);
      if (NeuronConstants.debug)
        System.out.println("Network: createInputSynapses adding synapse=("
                           +s.toString()+") Neuron "+ key+" =("+n.toString()+")");
    }
  }

public void createOutputSynapses()
{
  if(outputSynapses == null)
  {
    outputSynapses = new Hashtable();
    Map outputLayer;
    // find largest Layer int - that is the output layer
    Integer outIndex = new Integer(layers.size()-1);
    outputLayer = (Map) layers.get(outIndex);
    String key;
    Neuron n = null;
    Iterator il = outputLayer.keySet().iterator();
    while(il.hasNext())
    {
      key = (String) il.next();
      n = (Neuron) outputLayer.get(key);
      Synapse s = new Synapse();
      s.deltaT = 0;
      s.setW( 1);
      s.setName("output"+key);
      n.addPostSynapse(s);
      outputSynapses.put(key, s);
      if (NeuronConstants.debug)
        System.out.println("Network: createOutputSynapses adding synapse=("
                           +s.toString()+") Neuron "+ key+" =("+n.toString()+")");
    }
  }
```

```
 }

 public void stimulate(Map pulseNeuronPairs, int group) throws StimulateException
 {
   Iterator pi = null;
   pi = pulseNeuronPairs.keySet().iterator();
   inputSynapses.setActiveGroup(group);
   while(pi.hasNext())
   {
     String name = (String) pi.next();
     Pulse p = (Pulse) pulseNeuronPairs.get(name.trim());
     //inputSynapses.setActiveGroup(group);
     Synapse s = (Synapse) inputSynapses.getSynapse(name.trim());
     try{
       if(NeuronConstants.debug) System.out.println("Network:stimulate adding pulse=("
             +p.toString()+") for neuron = ("+name+") to synapse "+s.getName()+"
=("+s.toString()+")"
             + " eventTime ="+p.eventTime);
       s.addPulse(p);
      // if(NeuronConstants.debug) System.out.println("Network:stimulate adding pulse=("
      //       +p.toString()+") to synapse =("+s.toString()+")"+ " eventTime
="+p.eventTime);
     }
     catch(DeadPulseException dpe){dpe.printStackTrace();}
     catch(Exception e){
       String err = ":Problem adding pulse to input synapse for neuron "+ name
           + " in group "+group;
       if (s != null) err = err + "\n\t: synapse name is "+s.getName();
       else err = err + "\n\t : synapse = null -"+name+ " does not have an input synapse.
";
       if(p'= null) err = err + "\n\t : pulse = "+p.toString();
       else err = err + "\n\t : pulse is null - pulse invalidly consrtructed check input
file.";
       if (e.getMessage() != null)
         err = e.getMessage() + "\n\t" + err;
       throw new StimulateException(err);
     }
   }
 }
 /**
  * Stimulate using input from text file
  */
 /*
 public void stimulate2(File f) throws FileNotFoundException, Exception
 {
  if('f.exists()) throw new FileNotFoundException();
  Properties input = new Properties();
  try
  {  input.load(new FileInputStream(f)); }
  catch (IOException ioe)
  {
    ioe.printStackTrace();
    return;
  }
  Collection mapColl = new Vector();
  Iterator nit = input.keySet().iterator();
  while (nit.hasNext()) // iterate through neuron names
  {
    String name = (String) nit.next();
    String pulses = input.getProperty(name);
    StringTokenizer pt = new StringTokenizer(pulses,":");
    while (pt.hasMoreTokens())  // iterate through pulses
    {
      String pvals = pt.nextToken();
      double amp;
```

```
      int etime;
      StringTokenizer pval_tokenizer = new StringTokenizer(pvals, ",");
      if (pval_tokenizer.countTokens() == 2)
      {
        amp = Double.parseDouble(pval_tokenizer.nextToken());
        etime = Integer.parseInt(pval_tokenizer.nextToken());
      }
      else throw new Exception("Network input text file corrupt");
      Iterator mit = mapColl.iterator();
      boolean added = false;
      while (mit.hasNext()) // interate through maps
      {
        Map m = (Map) mit.next();
        if ('m.containsKey(name.trim()))
        {

          m.put(name,new Pulse(amp, etime, 0));
          added = true;
          continue;
        }
      }
      if ('added)
      {
        Map m = new Hashtable();
        m.put(name, new Pulse(amp, etime, 0));
        mapColl.add(m);
      }

    }
  }
  Iterator mit = mapColl.iterator();
  while (mit.hasNext())
  {
    this.stimulate((Map) mit.next());
  }
}
  */
public void stimulate(File f) throws FileNotFoundException, Exception
 {
  if('f.exists()) throw new FileNotFoundException();
  Properties input = new Properties();
  try
  {  input.load(new FileInputStream(f)); }
  catch (IOException ioe)
  {
    ioe.printStackTrace();
    return;
  }
  // get input layer neuron names
  StringTokenizer inTokenizer = new StringTokenizer(input.getProperty("input_group"));
  Vector inputLayerNames = new Vector();
  // first element should be the input group (0 ... n)
  Integer inGroup = new Integer(inTokenizer.nextToken(",").trim());
  while( inTokenizer.hasMoreTokens()) inputLayerNames.add(inTokenizer.nextToken(","));

  Collection mapColl = new Vector();
  Iterator nit = input.keySet().iterator();
  while (nit.hasNext()) // iterate through event_times
  {
    String time = (String) nit.next();
    if (time.trim().equalsIgnoreCase("input_group"))
      {
        continue;
      }
    int etime = 0;
```

```
      //boolean badTime = false;
      //try{
        etime = Integer.parseInt(time.trim());
      //}catch(NumberFormatException nfe)
      //{
        //if not a time break out of loop

      //}
      //if (badTime) break;
      String pulses = input.getProperty(time);
      StringTokenizer pt = new StringTokenizer(pulses,",");
      int index = 0;
      while (pt.hasMoreTokens())  // iterate through pulses
      {
        String name = (String) inputLayerNames.get(index);
        name = name.trim();
        String ampString = pt.nextToken();
        if(ampString.trim().equals("0"))
          {
            index++;
            continue;
          }
        double amp = Double.parseDouble(ampString);

        Iterator mit = mapColl.iterator();
        boolean added = false;
        while (mit.hasNext()) // interate through maps
        {
          Map m = (Map) mit.next();
          if ('m.containsKey(name.trim()))
          {
            //find current unitime for neuron = name
            Neuron ntemp = this.findNeuron(name.trim());
            //create and add pulse
            m.put(name, new Pulse(amp, ntemp.uniTime + etime, ntemp.uniTime));
            added = true;
            //continue;
            break;
          }
        }
        if ('added)
        {
          Map m = new Hashtable();
          //find current unitime for neuron = name
          Neuron ntemp = this.findNeuron(name.trim());
          //create and add pulse
          m.put(name, new Pulse(amp, ntemp.uniTime + etime, ntemp.uniTime));
          mapColl.add(m);
        }
        index++;  // increment name index
      }
    }
    Iterator mit = mapColl.iterator();
    while (mit.hasNext())
    {
      this.stimulate((Map) mit.next() , inGroup.intValue());
    }
  }

  public void tick()
   {
     Map lmap = null;
     Neuron n = null;
     boolean keepGoing = true;
     int i = 0;
```

```
  Integer key = null;
  while(keepGoing)
  {
    lmap = (Map) layers.get(new Integer(i));
    Iterator il = lmap.values().iterator();
    while(il.hasNext())
    {
      n = (Neuron) il.next();
      n.tick();
    }
    i++;
    key = new Integer(i);
    keepGoing = layers.containsKey(key);
  }
}

public Map getOutputSynapses()
{ return outputSynapses; }

public void showOutput(int toTime)
{ showOutput(0, toTime);}
public void showOutput(int fromTime, int toTime)
{
  Map myOut = this.getOutputSynapses();
  //Iterator myOit = myOut.keySet().iterator();
  //sort the keys
  SortedSet ss = new TreeSet(myOut.keySet());
  Iterator myOit = ss.iterator();
  System.out.print("\t");
  //print output layer names
  while (myOit.hasNext())
  {
    System.out.print(" " + (String) myOit.next());
    if (myOit.hasNext())System.out.print("    \t ,  ");
  }
    System.out.println();
for(int i = fromTime; i <= toTime; i++)
{
  String outString = " " + i + "\t";
  boolean outFlag = false;
  //Iterator oit = myOut.keySet().iterator();
  ss = new TreeSet(myOut.keySet());
  Iterator oit = ss.iterator();
  while (oit.hasNext())
      {
        String outKey = (String) oit.next();
        Synapse out1 = (Synapse) myOut.get(outKey);
        Iterator oi = out1.heldPulses.iterator();
        while (oi.hasNext())
        {
          Pulse p = (Pulse) oi.next();
          //System.out.println(p.uniTime + ", " + p.amplitude);
          if (p.uniTime == i)
          {

            outFlag = true;
            outString = outString + " " + p.amplitude;
            // add spaces based on ammplitude digits
            if (p.amplitude < 10) outString = outString + "      ";
            else if(p.amplitude < 100) outString = outString + "    ";
            else outString = outString + "  ";
          }
        }
        //if('outFlag) outString = outString + "\t , /t ";
        if ('outFlag) outString = outString + " 0      ";
```

```
          if(oit.hasNext())  outString = outString + "\t ,  ";
          outFlag = false;
        }

        System.out.println(outString);
      }

  }

  /**
   * Returns an Iterator over the keys i.e. String names of neurons
   * in the specified input group.  This can be used to build a stimulus Map
   * of Pulse-Neuron pairs
   *
   * @param group Integer
   * @return Iterator
   */
  public Iterator getInputGroupKeyIterator(Integer group)
{
  return this.inputSynapses.getGroupIterator(group);
}

public Set getInputGroupKeySet(Integer group)
{
 return this.inputSynapses.getGroupSet(group);
}

  /**
   * reset the unitime value for each neuron to the spcified time value
   * or 0 if no argugment is given.
   */
public void resetUniTime(int time)
  {
    Iterator li = layers.keySet().iterator();
    Map layer = null;
    Neuron n = null;
    Integer key;
    //name = name.trim();
    while(li.hasNext())
    {
      key = (Integer) li.next();
      layer = (Map) layers.get(key);
      Iterator ni = layer.values().iterator();
      while(ni.hasNext())
      {
        n = (Neuron) ni.next();
        n.uniTime = time;
      }
    }
  }
  public void resetUniTime()
    { resetUniTime(0);}
    /**
      * reset the internal potential value for each neuron to the
      * specified potential value
      * or the default resting Potential if no argugment is given.
      */
   public void resetPotential(double potent)
     {
       Iterator li = layers.keySet().iterator();
       Map layer = null;
       Neuron n = null;
       Integer key;
       //name = name.trim();
       while(li.hasNext())
```

```
        {
          key = (Integer) li.next();
          layer = (Map) layers.get(key);
          Iterator ni = layer.values().iterator();
          while(ni.hasNext())
          {
            n = (Neuron) ni.next();
            n.setPotential(potent);
          }
        }
      }
      public void resetPotential()
        { resetPotential(NeuronConstants.restingPotential);}


  /**
   * Removes all pulses from out put synapses.
   * This is necessary after resetting unitime, since pulses will
   * not have a unique event time.  Output will overlap.
   */
public void purgeOutput()
  {
    // reference to each synapse
    Iterator oi = this.outputSynapses.values().iterator();
    Synapse s = null;
    while(oi.hasNext())
    {
      s = (Synapse) oi.next();
      s.purgeAllPulses();
    }
  }
public void reset()
  {
    if(NeuronConstants.debug) System.out.println("resetting network...");
    resetUniTime();
    purgeOutput();
    resetPotential();
  }

  public String toString()
  {
    NumberFormat nf = NumberFormat.getInstance();
    nf.setMaximumFractionDigits(4);

    String s = "";
    String s2 = "";
    // iterate through layers
    for(int i = 0; i < layers.keySet().size(); i++)
    {
      // iterate through neurons in layer
      Map nMap = (Map) layers.get(new Integer(i));
      Iterator keyIter = nMap.keySet().iterator();
      s = s.concat("# layer "+i+" Neurons  \n");
      while(keyIter.hasNext())
      {
        Neuron n = (Neuron) nMap.get(keyIter.next());
        s = s + n.name+" \t n, "+i;
        if (n.isLearnEnabled()) s = s.concat(", t");
        else s = s.concat(", f");
        s=s+", "+ nf.format(n.getLeakRate());
        s=s.concat(", "+ n.threshold);
        s=s.concat(", "+n.restingPotential);
        s=s.concat("\n");
        // iterate through synapses for starting at this neuron
        Iterator denIter = n.dendrites.iterator();
```

```
        while(denIter.hasNext())
        {
          Dendrite d = (Dendrite) denIter.next();
          Iterator synIter = d.synapses.iterator();
          while(synIter.hasNext())
          {
            Synapse syn = (Synapse) synIter.next();
            if(!syn.name.startsWith("input"))
            {
              s2 = s2.concat(syn.getName() + " \t s");

              s2 = s2.concat(", " + syn.presynapticRef.name);
              s2 = s2.concat(", " + n.name);
              s2 = s2.concat(", "+ nf.format(syn.getW()));

              s2 = s2.concat(", "+syn.deltaT);
              if(syn.learn) s2 = s2.concat(", t");
              else s2 = s2.concat(", f");
              s2 = s2.concat("\n");
            }
          }
        }
      }
      s2 = s2.concat("\n");
    }
    s = s.concat("\n# Synapses\n"+s2);
    return s;
  }
public void toFile(String path, String fileName)
{
  PrintWriter out = null;
    File f = new File(path, fileName);
      try{
        out = new PrintWriter(new BufferedWriter(new FileWriter(f)));
      }catch(IOException ioe){ioe.printStackTrace(); out.flush();out.close();}


      NumberFormat nf = NumberFormat.getInstance();
      nf.setMaximumFractionDigits(4);

      String s = "";
      String s2 = "";

      // do Neurons
      // iterate through layers
      for(int i = 0; i < layers.keySet().size(); i++)
      {
        // iterate through neurons in layer
        Map nMap = (Map) layers.get(new Integer(i));
        Iterator keyIter = nMap.keySet().iterator();
        s = s.concat("# layer "+i+" Neurons  \r\n");
        while(keyIter.hasNext())
        {
          Neuron n = (Neuron) nMap.get(keyIter.next());
          s = s + n.name+" \t n, "+i;
          if (n.isLearnEnabled()) s = s.concat(", t");
          else s = s.concat(", f");
          s=s+", "+ nf.format(n.getLeakRate());
          s=s.concat(", "+ n.threshold);
          s=s.concat(", "+n.restingPotential);
          s=s.concat("\r\n");
        }
      }

      out.println(s);
```

```
        s= "";

        // do Synapses
        s2 = s2.concat("\r\n# Synapses\r\n"+s2);

        for(int i = 0; i < layers.keySet().size(); i++)
        {
          // iterate through neurons in layer
          Map nMap = (Map) layers.get(new Integer(i));
          Iterator keyIter = nMap.keySet().iterator();

          while(keyIter.hasNext())
          {
            Neuron n = (Neuron) nMap.get(keyIter.next());
            // iterate through synapses for starting at this neuron
            Iterator denIter = n.dendrites.iterator();
            while(denIter.hasNext())
            {
              Dendrite d = (Dendrite) denIter.next();
              Iterator synIter = d.synapses.iterator();
              while(synIter.hasNext())
              {
                Synapse syn = (Synapse) synIter.next();
                if(!syn.name.startsWith("input"))
                {
                  s2 = s2.concat(syn.getName() + " \t s");

                  s2 = s2.concat(", " + syn.presynapticRef.name);
                  s2 = s2.concat(", " + n.name);
                  s2 = s2.concat(", "+ nf.format(syn.getW()));

                  s2 = s2.concat(", "+syn.deltaT);
                  if(syn.learn) s2 = s2.concat(", t");
                  else s2 = s2.concat(", f");
                  s2 = s2.concat("\r\n");

                  out.print(s2);
                  s2 = "";
                }
              }
              //out.println(s2);
              //s2 = "";
            }
          }
          s2 = s2.concat("\r\n");
        }

        out.flush();
        out.close();
}

  public void offsetSynapses(String neuronName, double offset)
  {
    for(int i = 0; i < layers.keySet().size(); i++)
    {
      Map nMap = (Map) layers.get(new Integer(i));
      Iterator keyIter = nMap.keySet().iterator();
      while(keyIter.hasNext())
      {
        Neuron n = (Neuron) nMap.get(keyIter.next());
        if(n.name.equals(neuronName.trim()) )
        {
          Iterator denIter = n.dendrites.iterator();
          while (denIter.hasNext())
          {
```

```
          Dendrite d = (Dendrite) denIter.next();
          Iterator synIter = d.synapses.iterator();
          while (synIter.hasNext())
          {
            Synapse syn = (Synapse) synIter.next();
            syn.setW(syn.getW() + offset);

          }
        }
      }
    }
  }
}
/**
 * Flatten
 * Makes all synapse synapsing on the given neuron with an
 * absolute value less than the bound equal to zero
 * @param neuronName String

 * @param offset double - posative double
 */
public void flattenSynapses(String neuronName, double bound)
  {
    for(int i = 0; i < layers.keySet().size(); i++)
    {
      Map nMap = (Map) layers.get(new Integer(i));
      Iterator keyIter = nMap.keySet().iterator();
      while(keyIter.hasNext())
      {
        Neuron n = (Neuron) nMap.get(keyIter.next());
        if(n.name.equals(neuronName.trim()) )
        {
          Iterator denIter = n.dendrites.iterator();
          while (denIter.hasNext())
          {
            Dendrite d = (Dendrite) denIter.next();
            Iterator synIter = d.synapses.iterator();
            while (synIter.hasNext())
            {
              Synapse syn = (Synapse) synIter.next();
              if( java.lang.Math.abs(syn.getW()) < bound)
              syn.setW(0);

            }
          }
        }
      }
    }
  }

  public void pruneSynapse(String neuronName, double weight)
    {
      for(int i = 0; i < layers.keySet().size(); i++)
      {
        Map nMap = (Map) layers.get(new Integer(i));
        Iterator keyIter = nMap.keySet().iterator();
        while(keyIter.hasNext())
        {
          Neuron n = (Neuron) nMap.get(keyIter.next());
          if(n.name.equals(neuronName.trim()) )
          {
            Iterator denIter = n.dendrites.iterator();
            while (denIter.hasNext())
            {
              Dendrite d = (Dendrite) denIter.next();
```

```
                Iterator synIter = d.synapses.iterator();
                while (synIter.hasNext())
                {
                  Synapse syn = (Synapse) synIter.next();
                  if (syn.getW() == weight) //prune
                  {
                    Neuron preN = syn.presynapticRef;
                    //remove the synapse from n and preN
                    preN.removePostSynapse(syn);
                    synIter.remove();
                    //n.removePreSynapse(syn);

                    }

                }
              }
            }
          }
        }
      }

}
```

NetworkBuilder.java

```
/**-----------------------------------------------------------------
 * ********************** Change log ****************************
 * -----------------------------------------------------------------
 * +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * 8/2/2003 - added learn (t/f) flag to build();
 * +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *
 * -----------------------------------------------------------------
 */

package ooneuralnet;


import java.util.Properties;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.*;

public class NetworkBuilder
{
  private Properties script = new Properties();
  private File f;
  private Network net = new Network();

  public NetworkBuilder(File scriptFile) throws FileNotFoundException, IOException
  {
   f = scriptFile;
   if(!f.exists()) throw new FileNotFoundException();
   load();
  }
  public NetworkBuilder(String path, String fileName) throws FileNotFoundException,
IOException
  {
   f = new File(path,fileName);
   if('f.exists()) throw new FileNotFoundException();
```

```
 load();
}

private void load() throws FileNotFoundException, IOException
{
  //FileInputStream fstream = new FileInputStream(f);
  script.load(new FileInputStream(f));
}
public Network build() throws NetworkBuilderException
{
  if(NeuronConstants.debug) script.list(System.out);
   // build the Neurons
   Iterator si = script.keySet().iterator();
   StringTokenizer tk = null;
   String type;
   // get one time parameters first
   String gLeakRate = script.getProperty("leakRate");
   String gLTPRate = script.getProperty("LTPRate");
   String gLTDRate = script.getProperty("LTDRate");
   String gThreshold = script.getProperty("Threshold");
   String gInputGroups = script.getProperty("InputGroups", "empty");
   System.out.println("leakRate = "+gLeakRate);
   System.out.println("LTPRate = "+gLTPRate);
   System.out.println("LTDRate = "+gLTDRate);
   while (si.hasNext())
   {
     String key = (String) si.next();
     String values = script.getProperty(key);
     tk = new StringTokenizer(values,",");
     if (tk.countTokens() < 1)
       throw new NetworkBuilderException("Not enough arguments for "+key);
     type = tk.nextToken().trim();
     if (type.equalsIgnoreCase("n")) // build all neurons first
         {
           String name = key;
           Integer layer = new Integer( tk.nextToken().trim() );
           Neuron n = new Neuron(name);
           if(gLeakRate != null) n.setLeakRate(Double.parseDouble(gLeakRate));
           if(gThreshold != null) n.threshold = Double.parseDouble(gThreshold);
           net.addNeuron(n, layer);  //add it to NetworkContainer
           //optional neuron params
           if (tk.hasMoreTokens())
           {
             //enable learning (disabled by default)
             if (tk.nextToken().trim().equalsIgnoreCase("t"))
             {
               n.enableLearning();
             }
               // leak rate for this neuron
               if(tk.hasMoreTokens())
               {
                 String localLeakRate = tk.nextToken().trim();
                 if (localLeakRate != "-")
                   n.setLeakRate(Double.parseDouble(localLeakRate));
                 // set Threshold
                 if(tk.hasMoreTokens())
                 {
                   String threshold = tk.nextToken().trim();
                   if (threshold != "-")
                     n.threshold = Double.parseDouble(threshold);
                   //set resting potential
                   if(tk.hasMoreTokens())
                   {
                     String restPoten = tk.nextToken().trim();
                     if (restPoten != "-")
```

```
                        n.restingPotential = Double.parseDouble(restPoten);
                    }
                  }
                }
              }
            }
        }
        // build Synapses
        si = script.keySet().iterator();
        while (si.hasNext())
        {
          String key = (String) si.next();
          String values = script.getProperty(key);
          tk = new StringTokenizer(values,",");
          if (tk.countTokens() < 1)
            throw new NetworkBuilderException("Not enough arguments for "+key);
          type = tk.nextToken().trim();
          if (type.equalsIgnoreCase("s")) // build only synapses here
              {
                String from = tk.nextToken().trim();
                Neuron n1 = net.findNeuron(from);
                String to = tk.nextToken().trim();
                Neuron n2 = net.findNeuron(to);
                String w = tk.nextToken().trim();
                String tdelay = tk.nextToken().trim();
                Synapse syn = new Synapse();
                syn.setName(key.trim());
                syn.setW( Double.parseDouble(w));
                syn.deltaT = Integer.parseInt(tdelay);
                if(gLTPRate != null) syn.setLTPRate(Double.parseDouble(gLTPRate));
                if(gLTPRate != null) syn.setLTDRate(Double.parseDouble(gLTDRate));
                n1.addPostSynapse(syn);
                n2.addPreSynapse(syn);
                //optional params
                if (tk.hasMoreTokens())
                {
                  //enable learning (disabled by default)
                  if (tk.nextToken().trim().equalsIgnoreCase("t")) syn.enableLearning();
                }


              }
        }
        //-------create input and output synapses--------
        // input
        if (gInputGroups.equalsIgnoreCase("empty"))
          net.createInputSynapses();
        else // parse layers and weight
        {
          // should have 1 or more sets of three args
          // [group (int)], [layer (int)], [weight (float)], ...
          tk = new StringTokenizer(gInputGroups,",");
          if (tk.countTokens() < 3)
            throw new NetworkBuilderException(
               "Not enough arguments for InputLayers");
          while(tk.hasMoreTokens())
          {
            try{
            Integer group = new Integer(tk.nextToken().trim());
            //String ntt = tk.nextToken();
            Integer layer = new Integer(tk.nextToken().trim());
            Float weight = new Float(tk.nextToken().trim());
            net.createInputSynapses(group.intValue(), layer.intValue(),
weight.floatValue());
          }
```

```
          catch(NoSuchElementException nsee) {
            throw new NetworkBuilderException(
               "Invalid number or sequence or parameters specified in InputGroupss");
          }
          catch( NumberFormatException nfe) {
            throw new NetworkBuilderException(
               "A parameter specified in InputGroups cannot be converted to a number");
          }

        }
      }
      // output
      net.createOutputSynapses();
      return net;
  }
  public Network getNetwork(){return net;}
}
```

NetworkBuilderException.java

```
package ooneuralnet;

public class NetworkBuilderException extends Exception
{
  public NetworkBuilderException() {super();}
  public NetworkBuilderException(String gripe) {super(gripe);}


}
```

PixelSynapseModel.java

```
package ooneuralnet.visualtracking;


import ooneuralnet.Pulse;

import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;
import java.util.Iterator;
import java.util.Map;
import java.util.Hashtable;
import java.awt.image.Raster;


/**
 *
 * <p>Title: OONeuralNet Visual Tracking</p>
 * <p>Description: Objcet Oriented Nerual Network Model</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Lon Risinger
 * @version 1.0
*
* Pixel Synapse Model is used to tell VisualTool.fillStimulusMap() how
* to fill the stimulus map with pixel information.  See model type fields
* for a brief explanaition.
*
 */
```

```
public class PixelSynapseModel
{
  /**
   * Pixel Synapse Model Type used for one to one mapping.
   * Maps one pixel per input synapse.
   */
  static public final int OneForOne = 1;
  /**
   * Pixel Synapse Model Type used to map an entire columns input to one synapse.
   * Sums one column per input synapse.
   */
  static public final int ColumnPerSynapse = 2;
  /**
   * Pixel Synapse Model Type used to map an entire row input to one synapse.
   * Sums one row per input synapse.
   */
  static public final int RowPerSynapse = 3;


  int modelType =0;

  int minx =0;
  int miny =0;
  int maxx =0;
  int maxy =0;
  int p[] ;
  double p2[];
  Set groupKeys = null;
  SortedSet sortedKeys = null;
  Iterator gKeyIter = null;
  Map stimMap = null;
  Raster r = null;

  /**
   * This constructor builds an object that loads pixel information contained in
   * Raster r into a stimulus map.
   *
   * Use genStimulusMap() to generate a new stimulus Map based on the
   * provided information.
   *
   * @param modelType int - the particular way pixel should be assigned to
   * input group neurons (see class fields above)
   */
  public PixelSynapseModel(int modelType)
  {

    this.modelType = modelType;

    /*
    // establish pixel ranges
    minx = r.getMinX();
    miny = r.getMinY();
    maxx = r.getMinX()+r.getWidth()-1;
    maxy = r.getMinY()+r.getHeight()-1;
    // set pixel type -- gray scale  ... Color is p[3] for R G B values
    p = new int[1];
    //make it big -- avoid collisions and resizing
    stimMap = new Hashtable((maxx-minx)*(maxy-miny));
    this.groupKeys = groupKeys;
    this.r = r;
    */
  }
  /**
   *
   * @param r Raster - image Raster
```

```
 * @param groupKeys Iterator - iterator over keys in input group.
 * This iterator must be sorted
 */
public void init(Raster r, Set groupKeys)
{
  // establish pixel ranges
  minx = r.getMinX();
  miny = r.getMinY();
 maxx = r.getMinX()+r.getWidth()-1;
 maxy = r.getMinY()+r.getHeight()-1;
 // set pixel type -- gray scale  ... Color is p[3] for R G B values
  p = new int[1];
  //make it big -- avoid collisions and resizing
  stimMap = new Hashtable((maxx-minx)*(maxy-miny));
  this.groupKeys = groupKeys;
  sortedKeys = new TreeSet(groupKeys);
  this.r = r;
}



/**
 * Returns a stimulus map based on Raster, input group, and model
 * type information provided in the constructor.
 *
 * @param eventtime int - event time for pulses in generated stimulus map
 * @param unitime int - universal time for pulses in gerated stimulus map
 * @return Map - Stimulus Map
 */
public Map genStimulusMap(int eventtime, int unitime)
{
  gKeyIter = sortedKeys.iterator();

  switch(modelType)
  {
      case OneForOne:
           return genSMOneForOne(eventtime, unitime);
      case ColumnPerSynapse:
           return genSMColumnPerSynapse(eventtime, unitime);
      case RowPerSynapse:
           return genSMRowPerSynapse(eventtime, unitime);
  }
  return null;
}

protected Map genSMOneForOne(int eventtime, int unitime)
{
  int x = minx;
  int y = miny;
  while(gKeyIter.hasNext())
  {
    // image data for this pixel
    r.getPixel(x,y,p);

    // adjust pixel data range from (0-255) to (0 to 100)
    p[0] = p[0]*100/255;

    // build the pulse
    Pulse pulse = new Pulse(p[0]/2 , eventtime, unitime);

    // add to stimulus Map
    String neuronName = (String) gKeyIter.next();
    stimMap.put(neuronName, pulse);

    //update x y coordinates
```

```
    //System.out.println("y="+y);
    // if this is the last x in row wrap
    if(x == maxx)
    {
      x = minx;
      y++;
    }
    else
      x++;
    //System.out.println("x="+x);
    // do not overflow
    if (y == maxy+1) break;
  }
  return stimMap;

}
protected Map genSMColumnPerSynapse(int eventtime, int unitime)
{
  int x = minx;
  int y = miny;
  //int cols[] = new int[maxx - minx+1];
  double cols[] = new double[maxx - minx+1];
  while(true)
  {
    // image data for this pixel
    r.getPixel(x,y,p);

    // adjust pixel data range from (0-255) to (0 to 100)
    //p[0] = p[0]*100/255;

    // sum columns
    cols[x] += expScale(p[0]);

    //update x y coordinates
    //System.out.println("y="+y);
    // if this is the last x in row wrap
    if(y == maxy)
    {
      y = miny;
      x++;
    }
    else
      y++;
    //System.out.println("x="+x);
    // do not overflow
    if (x == maxx+1) break;
  }

  x = 0;
  while(gKeyIter.hasNext())
  {

    // build the pulse
    //Pulse pulse = new Pulse(cols[x]/(maxy-miny), eventtime, unitime);
    Pulse pulse = new Pulse(unExpScale(cols[x]/(maxy-miny)), eventtime, unitime);
    //Pulse pulse = new Pulse(unExpScale(cols[x]), eventtime, unitime);


    // add to stimulus Map
    String neuronName = (String) gKeyIter.next();
    stimMap.put(neuronName, pulse);
    x++;
  }
  return stimMap;
```

```
}
protected Map genSMRowPerSynapse(int eventtime, int unitime)
{
  int x = minx;
  int y = miny;
  //int rows[] = new int[maxy - miny+1];
  double rows[] = new double[maxy - miny+1];
  while(true)
  {
    // image data for this pixel
    r.getPixel(x,y,p);

    // adjust pixel data range from (0-255) to (0 to 100)
    //p[0] = p[0]*100/255;

    // sum columns
    //rows[y] += p[0];
    rows[y] += expScale(p[0]);

    //update x y coordinates
    //System.out.println("y="+y);
    // if this is the last x in row wrap
    if(x == maxx)
    {
      x = minx;
      y++;
    }
    else
      x++;
    //System.out.println("x="+x);
    // do not overflow
    if (y == maxy+1) break;
  }

  y = 0;
  while(gKeyIter.hasNext())
  {

    // build the pulse
    //Pulse pulse = new Pulse(rows[y]/(maxx-minx), eventtime, unitime);
    Pulse pulse = new Pulse(unExpScale(rows[y]/(maxx-minx)), eventtime, unitime);
    //Pulse pulse = new Pulse(unExpScale(rows[y]), eventtime, unitime);

    // add to stimulus Map
    String neuronName = (String) gKeyIter.next();
    stimMap.put(neuronName.trim(), pulse);
    y++;
  }
  return stimMap;

}
/**
 * scale value exponential 2^(val/10)
 * @param val int
 * @return int
 */
public double expScale(int val)
{
  double base = 2;  // orig = 2
  double r = Math.pow( base ,(double)val/2);
  return r;
}
public int unExpScale(double val)
  {
    double base = 2.0; // orig = 2
```

```
      //System.out.println("---------------------------------------Pre-unExpScale="+val);
      int r = (int) (Math.log((val -48 + 1))/Math.log(base));
      //int r = (int) (Math.log((val -48 + 1)));
      r -= 28;  //orig = 30
      r = r/18;
      if(r <0) r = 0;
      //System.out.println("-----------------------------------------unExpScale="+r);
      return r;
    }


}
```

VisualTool.java

```
package ooneuralnet.visualtracking;
import ooneuralnet.Pulse;

import java.awt.Image;
import java.awt.Canvas;
import java.awt.Toolkit;

import java.awt.*;
//import java.awt.event.*;
import javax.swing.*;
import javax.imageio.*;


import java.awt.Image;
import java.awt.image.BufferedImage;
import java.awt.Toolkit;
import java.awt.image.Raster;
import java.awt.image.WritableRaster;
import java.awt.image.ImageProducer;
import java.awt.MediaTracker;


import java.util.*;
import java.io.File;
import java.text.NumberFormat;

//for scalling
import java.awt.geom.AffineTransform;
import java.awt.image.BufferedImageOp;
import java.awt.image.*;


/**
 * <p>Title: OONeuralNet</p>
 * <p>Description: Objcet Oriented Nerual Network Model</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Lon Risinger
 * @version 1.0
 */

public class VisualTool extends Canvas
{
  public Image image;
  public BufferedImage bi;


  private boolean scale = false;
```

```
    /**

     * empty constructor
     */
    public VisualTool()
   {}

   /**
    * constructs tool with given image
    *
    * @param imageName String
    */
   public VisualTool(String path, String imageName)
   { setImage(path, imageName);}


   public void setImage(File f)
   {
    //System.out.println(f.getParent()  + f.getName());
    setImage(f.getParent() +f.separatorChar, f.getName());
   }



   /**
    * set the image for this tool
    *
    * @param imageName String
    */
   public void setImage(String path, String imageName)
   {

        image = Toolkit.getDefaultToolkit().createImage( path + imageName);
        try
        {
          MediaTracker tracker = new MediaTracker(new JPanel());
          tracker.addImage(image, 0);
          tracker.waitForID(0);
        }
        catch ( Exception e ) {}


        int iw = image.getWidth(this);
        int ih = image.getHeight(this);

        this.setSize(iw,ih);
        // create a buffered image of the right type
        //bi = new BufferedImage(iw, ih, BufferedImage.TYPE_INT_RGB);
        bi = new BufferedImage(iw, ih, BufferedImage.TYPE_BYTE_GRAY);

        //draw the image into it
        Graphics2D big = bi.createGraphics();
        big.drawImage(image,0,0,null);

        // show internal data
        WritableRaster wr = bi.getRaster();
        System.out.println("-------------------- set image = "+ imageName +" ----------");
        System.out.println(wr.toString());
/*
        int p[] = null;
        for(int y =0; y < ih; y++)
        {
          for (int x =0; x<iw; x++)
          {
```

```
            p = wr.getPixel(x, y, p);
            for (int i = 0; i < p.length; i++)
            {
              //System.out.print(p[i] + " ");
              //change the data
              //if((x > iw/2) && (x < iw/2+30)) p[0] = p[0] / 2;
              //wr.setPixel(x,y,p);


            }
          }
          //System.out.println();
        }
*/
  }
  public void createMotionImageSeries(MotionEq me,int blockSize, int numFrames, Dimension
imageDim, String path, String fileNameBase)
  {
    createMotionImageSeries(me ,blockSize, numFrames, imageDim, path, fileNameBase,
null);
  }

  public void createMotionImageSeries(MotionEq me,int blockSize, int numFrames, Dimension
imageDim, String path, String fileNameBase, String backGroundImage)
  {

    NumberFormat nf = NumberFormat.getInstance();
    nf.setMinimumIntegerDigits(3);
    int blockSide = blockSize;
    int[] pix= new int[blockSide*blockSide];
     for (int i = 0; i < numFrames; i++)
     {
       // create blank image
       BufferedImage buffImg = new BufferedImage(imageDim.width, imageDim.height,
                                                 BufferedImage.TYPE_BYTE_GRAY);

       // use background
       if(backGroundImage != null)
       {
         this.setImage(path, backGroundImage);
         Graphics2D big = buffImg.createGraphics();
         big.drawImage(image,0,0,null);
       }

       // put block in image according to motion
       WritableRaster wr = buffImg.getRaster();
       //printRawPixelData(wr);
        // check boundary
        if ((me.getX()+blockSide < buffImg.getMinX()+buffImg.getWidth()) &&
            ((me.getY()+blockSide < buffImg.getMinY()+buffImg.getHeight()))&&
            (me.getY() >= buffImg.getMinY()) &&  (me.getX() >= buffImg.getMinX()))
        {
          wr.getPixels(me.getX(), me.getY(), blockSide, blockSide, pix);
          for (int x = 0; x < pix.length; x++) {
            pix[x] = 255;
          }
          wr.setPixels(me.getX(), me.getY(), blockSide, blockSide, pix);
          //me.step((int)(imageDim.getWidth()/numFrames));
          me.step(1);
          buffImg.setData(wr);

          // save created image to gif file
          try {
            /*
            ImageIO.write(buffImg, "jpg",
```

```
                        new File(path, fileNameBase + "-" + nf.format(i) + ".gif"));
              }
              */              //System.out.println("Frame = "+i);
              System.out.print(i);
            }
            catch (Exception e) {
              e.printStackTrace();
            }
          }
      }



  }
  public void saveImage(String path, String name)
  {
    try{
    // save captured image to PNG file
    ImageIO.write(bi, "jpg", new File( path ,name ));
    }catch (Exception e) { e.printStackTrace();}
  }
public void printRawPixelData(Raster r)
{
  printRawPixelData(r, false);
}

public void printRawPixelData(Raster r, boolean scaled)
{

    int p[] = null;
    int ih = r.getHeight();
    int iw = r.getWidth();
    for(int y =0; y < ih; y++)
    {
      for (int x =0; x<iw; x++)
      {
        p = r.getPixel(x, y, p);
        if(scaled)
          for(int k = 0; k < p.length; k++)
          {
            p[k] = p[k]*100/255;
          }
        System.out.print("[ ");
        for (int i = 0; i < p.length; i++)
        {
          System.out.print(p[i] + " ");
          if(p[i] < 10) System.out.print("  ");
          else if (p[i] < 100) System.out.print(" ");
          //change the data
          //if((x > iw/2) && (x < iw/2+30)) p[0] = p[0] / 2;
          //wr.setPixel(x,y,p);
        }
        System.out.print("] ");
      }
      System.out.println();
    }
    System.out.println();
}


public void setImage(BufferedImage bimg)
{
         bi = bimg;
         //draw the image into it
```

```
        //Graphics2D big = bi.createGraphics();
        //big.drawImage(image,0,0,null);
        //big.drawImage(image,0,0,null);


        // show internal data
        WritableRaster wr = bi.getRaster();
        System.out.println("--------------------show image data----------");
        System.out.println(wr.toString());
        /*
        int p[] = null;
        for(int y =0; y < ih; y++)
        {
          for (int x =0; x<iw; x++)
          {
            p = wr.getPixel(x, y, p);
            for (int i = 0; i < p.length; i++)
            {
              //System.out.print(p[i] + " ");
              //change the data
              //if((x > iw/2) && (x < iw/2+30)) p[0] = p[0] / 2;
              //wr.setPixel(x,y,p);


            }
          }
          //System.out.println();
        } */

}

  public WritableRaster getImageRaster()
  {
       // show internal data
       WritableRaster wr = bi.getRaster();

       return wr;
  }

  public BufferedImage createSizedImage(int h, int w)
  {
    BufferedImage buffImg = new BufferedImage(w, h, BufferedImage.TYPE_BYTE_GRAY);
    this.paint(this.getGraphics());

    return buffImg;
  }
  public void changeImageSize(int h, int w)
  {
    bi = new BufferedImage(w, h, BufferedImage.TYPE_BYTE_GRAY);
    setImage(bi);

  }
  public void paint(Graphics g)
  {
    // draw image from the image object
   //g.drawImage(image, 0,0, null);

   // draw from buffered image
   if(scale)
     scaleImage(g);
   else
     g.drawImage(bi,0,0,null);

  }
  /**
```

```
    * This method fills a stimulus Map containing Neuron-Pulse pairs
    * with input from the image Raster and returns a reference to stimulus Map
    *
    *
    *
    * @param imageRaster Raster  - image data to be placed in stimulusMap
    * @param stimulusMap Map - input data for a particular Network inputGroup
    * @return Map returns a reference to stimulusMap
    *
    * @todo figure out if these iterators are in the same order.
    */
   public Map fillStimulusMap( Raster imageRaster, Map stimulusMap)
   {
     //stimulus

     return null;
   }

   /**
      * This method fills a stimulus Map containing Neuron-Pulse pairs
      * with input from the image Raster. A new Map is
      * returned.  Otherwise input is placed in supplied stimulus Map
      *
      *
      *
      * @param imageRaster Raster  - image data to be placed in stimulusMap

      * @param inputGroupKeys Set - an iterator of keys that will
      * be used to build the stimulus Map
      * @return Map  - input data for a particular Network inputGroup
      */
   public Map fillStimulusMap( Raster imageRaster, Set inputGroupKeys, int unitime, int
eventtime)
   {

     SortedSet sortedKeys = new TreeSet(inputGroupKeys);
     Iterator isk = sortedKeys.iterator();

     Map sMap = new Hashtable(sortedKeys.size());


     int x = imageRaster.getMinX();
     int y = imageRaster.getMinY();
     int maxx = imageRaster.getMinX()+imageRaster.getWidth()-1;
     int maxy = imageRaster.getMinY()+imageRaster.getHeight()-1;
     int p[] = {0};

     while(isk.hasNext())
     {
       // image data for this pixel
       imageRaster.getPixel(x,y,p);

       // adjust pixel data range from (0-255) to (0 to 100)
       p[0] = p[0]*100/255;

       // build the pulse
       Pulse pulse = new Pulse(p[0]/2 , eventtime, unitime);

       // add to stimulus Map
       String neuronName = (String) isk.next();
       sMap.put(neuronName, pulse);

       //update x y coordinates
       //System.out.println("y="+y);
       // if this is the last x in row wrap
```

```
      if(x == maxx)
      {
        x = imageRaster.getMinX();
        y++;
      }
      else
        x++;
      //System.out.println("x="+x);
      // do not overflow
      if (y == maxy+1) break;
    }
    return sMap;
  }
  /**
   * Fill a stimulus Map with image data for a particular region specified
   * by Rectangle region.  The location of this rectangle is the coordinate
   * of the upper left corner of the rectangular region inside the image.
   *
   * @param psm PixelSynapseModel
   * @param imageRaster Raster  - original image
   * @param region Rectangle    - region of image data to fill Map with
   * @param inputGroupKeys Set
   * @param unitime int
   * @param eventtime int
   * @return Map  - stimulus map
   */
  public Map fillStimulusMap(PixelSynapseModel psm, Raster imageRaster, Rectangle region,
Set inputGroupKeys, int unitime, int eventtime)
    {
      Point origin = region.getLocation();
      Raster child =null;
      try{
        child = imageRaster.createChild(origin.x, origin.y, region.width,
                                         region.height, 0, 0, null);
      }
      catch (java.awt.image.RasterFormatException rfe)
      {
        // this means the origin is out of bounds -- reset
        if((origin.x >= 0)&& (origin.y >= 0))
          throw rfe;
        if (origin.x < 0)
          origin.x =0;
        if (origin.y < 0)
          origin.y =0;
        child = imageRaster.createChild(origin.x, origin.y, region.width,
                                         region.height, 0, 0, null);
      }
      psm.init(child, inputGroupKeys);

      return psm.genStimulusMap(eventtime, unitime);
    }

  public Map fillStimulusMap(PixelSynapseModel psm, Raster imageRaster, Set
inputGroupKeys, int unitime, int eventtime)
  {
    psm.init(imageRaster, inputGroupKeys);

    return psm.genStimulusMap(eventtime, unitime);
  }
  /**
   *  Once fillStimulusMap(PixelSynapseModel psm, Raster imageRaster, Set inputGroupKeys,
int unitime, int eventtime)
   * has been called.  If the the Raster and inputGroupKeys are the same this method can
be called to generate stimulus Maps.
   * However the original psm instance must be used.
```

```
   *
   * @param psm PixelSynapseModel -- instance previously used by fillStimulusMap(.......)
   * @param unitime int
   * @param eventtime int
   * @return Map
   */
  public Map fillStimulusMap(PixelSynapseModel psm, int unitime, int eventtime)
  {
    return psm.genStimulusMap(eventtime, unitime);
  }

public void scaleImage()
{
  scale = true;

}
public void  scaleImage(Graphics g)
{

    Graphics2D g2 = (Graphics2D) g;
          //g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
          //                    RenderingHints.VALUE_ANTIALIAS_ON);
          //g2.setRenderingHint(RenderingHints.KEY_RENDERING,
          //                    RenderingHints.VALUE_RENDER_QUALITY);
          int w = getSize().width;
          int h = getSize().height;



              int iw = bi.getWidth(this);
              int ih = bi.getHeight(this);
              int x = 0, y = 0;

              AffineTransform at = new AffineTransform();
              //at.scale((w-14)/2.0/iw, (h-34)/2.0/ih);
              at.scale((w-14)/1.0/iw, (h-34)/1.0/ih);

              BufferedImageOp biop = null;
              BufferedImage bimg = new BufferedImage(iw,ih,BufferedImage.TYPE_BYTE_GRAY);

              //case 3 : x = w/2+3; y = h/2+15;
              RescaleOp rop = new RescaleOp(1000.0f,10.0f, null);
              rop.filter(bi,bimg);

              biop = new AffineTransformOp(at, AffineTransformOp.TYPE_BILINEAR);
              //biop = new AffineTransformOp(at,
AffineTransformOp.TYPE_NEAREST_NEIGHBOR);

              g2.drawImage(bimg,biop,x,y);
              //g2.drawImage(bimg,x,y,null,null);
              //TextLayout tl = new TextLayout(theDesc[i],
g2.getFont(),g2.getFontRenderContext());
              //tl.draw(g2, (float) x, (float) y-4);

}


  public Dimension getPreferedSize()
  {
    return new Dimension(image.getWidth(null), image.getHeight(null));
  }
```

```
}
```

NetFileCreator.java

```
package ooneuralnet.tool;
import java.util.Properties;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.lang.Character;

import java.text.NumberFormat;


import java.io.*;

/**
 * <p>Title: OONeuralNet</p>
 * <p>Description: Objcet Oriented Nerual Network Model</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Lon Risinger
 * @version 1.0
 */

public class NetFileCreator
{
  Properties p = new Properties();
  double leakRate = 0;
  double LTPRate =  0;
  double LTDRate =  0;
  PrintWriter out = null;

  public NetFileCreator()
  { p = new Properties(); }
  public NetFileCreator(File netFile)
  {
    p = new Properties();
    try
    {
      p.load(new FileInputStream(netFile));
    }
    catch(FileNotFoundException fne)
    {fne.printStackTrace();}
    catch(IOException ioe)
    {ioe.printStackTrace();}
  }


  public void setLeakRate(double r)
  {leakRate = r;}
  public void setLTPRate(double r)
  {
    LTPRate = r;
  }
  public void setLTDRate(double r)
  {
    LTDRate = r;
  }

  public void createMotionDet(int numSNperDir, String path, String fileName)
```

```
{
  NumberFormat nf = NumberFormat.getInstance();
  nf.setMinimumIntegerDigits(2);
  File f = new File(path, fileName);
  try{
    out = new PrintWriter(new BufferedWriter(new FileWriter(f)));
  }catch(IOException ioe){ioe.printStackTrace();}
  finally
  {
      out.flush();
      //out.close();
      //System.exit(-1);
  }

  this.setLeakRate(0.84);
  this.setLTPRate(1.005);
  this.setLTDRate(0.994);

  // neuron name base
  String nbase;
  // direction
  String[] dir = {"L","R","U","D"};
  String[] dirLong = {"Left","Right","Up","Down"};
  String[] pDir = {"1","2"};
  // layer number
  int layer = 0;



  String s = "";
  s=s.concat("# ------------------------------------------------\r\n");
  s=s.concat("#  "+ f.getName()+"\r\n");
   s=s.concat("# ------------------------------------------------\r\n");
  s=s.concat("leakRate = "+leakRate+"\r\n");
  s=s.concat("LTPRate = "+LTPRate+"\r\n");
  s=s.concat("LTDRate = "+LTDRate+"\r\n");
  s = s+"#Input group \r\n"
      +"InputGroups = 1, 0, 1, 2, 1, 1\r\n"
      +"#--- Neurons -------\r\n"
      +"#sensory neurons - \r\n";
  nbase = "SN";
  layer = 0;
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";

    for (int i = 1; i <= numSNperDir; i++)
    {
      s = s+nbase+dir[j]+nf.format(i)+" \t n, "+layer+"\r\n";
    }
    if(j >=1) layer = 1;

  }

  s = s+ "# Derivative Neurons\r\n";
  nbase = "DN";
  layer = 2;
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    for (int i = 1; i <= numSNperDir; i++)
    {
      s = s+nbase+dir[j]+nf.format(i)+" \t n, "+layer;
      s = s + ", f , 0.45"; // set leak rate for DN
      s= s +"\r\n";
```

```
  }
}

s = s+ "# Motion Neurons\r\n";
nbase = "MN";
layer = 3;
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    for (int i = 1; i <= numSNperDir-1; i++)
    {
      s = s+nbase+pDir[k]+dir[j]+nf.format(i)+"\t  n, "+layer+"\r\n";
    }
  }
}
s = s+ "# Output to Muscle Neurons\r\n";
nbase = "ON";
layer = 4;
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    //s = s+ "#  "+dirLong[j]+"\r\n";
    s = s+nbase+pDir[k]+dir[j]+"\t  n, "+layer+"\r\n";
  }
}

// Synapses ------------------------------------------------
String sbase;  //synapse base
String fromNBase; // from neuron base
String toNBase; // to Neuron base
String w;       // weight
String deltaT;  // time delay

s = s+ "# Synapses ----------------------\r\n";
sbase = "sa";
fromNBase = "SN";
toNBase = "DN";
w = "-0.16";
deltaT = "4";

s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {
    s = s+sbase+dir[j]+nf.format(i)+" \t s, "
        +fromNBase+dir[j]+nf.format(i)+", "
        +toNBase+dir[j]+nf.format(i)+", "
        +w+", "
        +deltaT+"\r\n";
  }
}
// SN to DN
sbase = "sb";
fromNBase = "SN";
toNBase = "DN";
w = "0.21";
deltaT = "1";
```

```
  s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    for (int i = 1; i <= numSNperDir; i++)
    {
      s = s+sbase+dir[j]+nf.format(i)+" \t s, "
          +fromNBase+dir[j]+nf.format(i)+", "
          +toNBase+dir[j]+nf.format(i)+", "
          +w+", "
          +deltaT+"\r\n";
    }
  }
// DN to MN
  sbase = "sm";
  fromNBase = "DN";
  toNBase = "MN";
  w = "0.12";
  deltaT = "4";
  String deltaT2 = "1";
  int count = 1;
  s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
  for(int k = 0; k < pDir.length ; k++)
  {
    s = s+ "#  direction "+pDir[k]+"\r\n";
    for(int j =0; j< dir.length; j++)
    {
      s = s+ "#  "+dirLong[j]+"\r\n";
      count = 1;
      for (int i = 1; i <= numSNperDir-1; i++)
      {
        s = s+sbase+pDir[k]+dir[j]+nf.format(count)+" \t s, "
            +fromNBase+dir[j]+nf.format(i)+", "
            +toNBase+pDir[k]+dir[j]+nf.format(i)+", "
            +w+", "
            +deltaT+"\r\n";
        count++;
        s = s+sbase+pDir[k]+dir[j]+nf.format(count)+"\t s, "
            +fromNBase+dir[j]+nf.format((i+1))+", "
            +toNBase+pDir[k]+dir[j]+nf.format(i)+", "
            +w+", "
            +deltaT2+"\r\n";
        count++;
      }
    }
    //swap(deltaT, deltaT2);
    String temp = new String(deltaT);
    deltaT = new String(deltaT2);
    deltaT2 = temp;

  }

  //out.print(s);
  //System.out.print(s);
  //s = "";
 // MN to ON
 sbase = "so";
 fromNBase = "MN";
 toNBase = "ON";
 w = "0.2";
 deltaT = "1";
 s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
 for(int k = 0; k < pDir.length ; k++)
 {
   s = s+ "#  direction "+pDir[k]+"\r\n";
```

```
      for(int j =0; j< dir.length; j++)
      {
        s = s+ "#  "+dirLong[j]+"\r\n";
        count = 1;
        for (int i = 1; i <= numSNperDir-1; i++)
        {
          s = s+sbase+pDir[k]+dir[j]+nf.format(count)+" \t s, "
              +fromNBase+pDir[k]+dir[j]+nf.format(i)+", "
              +toNBase+pDir[k]+dir[j]+", "
              +w+", "
              +deltaT+"\r\n";
           count++;
        }
      }
    }


    System.out.print(s);
     //out.print(s);
     out.println(s);
     out.flush();
     out.close();

  }
//--------------------  Motion Det 2 ---------------------------
  /**
   * Uses feedback from ON to DN to fight overshoot problem -- unsuccessful
   * @param numSNperDir int
   * @param path String
   * @param fileName String
   */
  public void createMotionDet2(int numSNperDir, String path, String fileName)
  {
    NumberFormat nf = NumberFormat.getInstance();
    nf.setMinimumIntegerDigits(2);
    File f = new File(path, fileName);
    try{
      out = new PrintWriter(new BufferedWriter(new FileWriter(f)));
    }catch(IOException ioe){ioe.printStackTrace();}
    finally
    {
        out.flush();
        //out.close();
        //System.exit(-1);
    }

    this.setLeakRate(0.84);
    this.setLTPRate(1.005);
    this.setLTDRate(0.994);

    // neuron name base
    String nbase;
    // direction
    String[] dir = {"L","R","U","D"};
    String[] dirLong = {"Left","Right","Up","Down"};
    String[] pDir = {"1","2"};
    // layer number
    int layer = 0;



    String s = "";
    s=s.concat("# ------------------------------------------------\r\n");
    s=s.concat("#  "+ f.getName()+"\r\n");
     s=s.concat("# ------------------------------------------------\r\n");
```

```
s=s.concat("leakRate = "+leakRate+"\r\n");
s=s.concat("LTPRate = "+LTPRate+"\r\n");
s=s.concat("LTDRate = "+LTDRate+"\r\n");
s = s+"#Input group \r\n"
    +"InputGroups = 1, 0, 1, 2, 1, 1\r\n"
    +"#--- Neurons -------\r\n"
    +"#sensory neurons - \r\n";
nbase = "SN";
layer = 0;
NeuronString ns = new NeuronString();
ns.setLayer(layer);
for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {

    ns.setName(nbase+dir[j]+nf.format(i));
    ns.setLayer(layer);
    s += ns.toString();
    //s = s+nbase+dir[j]+nf.format(i)+" \t n, "+layer+"\r\n";

  }
  if(j >=1) layer = 1;

}


s = s+ "# Derivative Neurons\r\n";
nbase = "DN";
layer = 2;
ns.reset();
ns.setLayer(layer);
for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {

    ns.setName(nbase+dir[j]+nf.format(i));
    ns.setLeakRate(0.45);
    s += ns.toString();

    //s = s+nbase+dir[j]+nf.format(i)+" \t n, "+layer;
    //s = s + ", f , 0.45"; // set leak rate for DN
    //s= s +"\r\n";
  }
}

s = s+ "# Motion Neurons\r\n";
nbase = "MN";
layer = 3;
ns.reset();
ns.setLayer(layer);
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    for (int i = 1; i <= numSNperDir-1; i++)
    {
      ns.setName(nbase+pDir[k]+dir[j]+nf.format(i));
      s += ns.toString();
      //s = s+nbase+pDir[k]+dir[j]+nf.format(i)+"\t  n, "+layer+"\r\n";
```

```
    }
  }
}
s = s+ "# Output to Muscle Neurons\r\n";
nbase = "ON";
layer = 4;
ns.reset();
ns.setLayer(layer);
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    //s = s+ "#  "+dirLong[j]+"\r\n";

    ns.setName(nbase+pDir[k]+dir[j]);
    s += ns.toString();
    //s = s+nbase+pDir[k]+dir[j]+"\t  n, "+layer+"\r\n";
  }
}

// Synapses ------------------------------------------
String sbase;  //synapse base
String fromNBase; // from neuron base
String toNBase; // to Neuron base
String w;       // weight
String deltaT;  // time delay

s = s+ "# Synapses ----------------------\r\n";
sbase = "sa";
fromNBase = "SN";
toNBase = "DN";
w = "-0.16";
deltaT = "4";

SynapseString ss = new SynapseString();
ss.setWeight(-0.16);
ss.setDetlaT(4);

s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {
    ss.setName(sbase+dir[j]+nf.format(i));
    ss.setFrom(fromNBase+dir[j]+nf.format(i));
    ss.setTo(toNBase+dir[j]+nf.format(i));

    s += ss.toString();
    /*
    s = s+sbase+dir[j]+nf.format(i)+" \t s, "
        +fromNBase+dir[j]+nf.format(i)+", "
        +toNBase+dir[j]+nf.format(i)+", "
        +w+", "
        +deltaT+"\r\n";
     */
  }
}
// SN to DN
sbase = "sb";
fromNBase = "SN";
toNBase = "DN";
w = "0.21";
deltaT = "1";
```

```
  ss.reset();
  ss.setWeight(0.21);
  ss.setDetlaT(1);
  s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    for (int i = 1; i <= numSNperDir; i++)
    {
      ss.setName(sbase+dir[j]+nf.format(i));
      ss.setFrom(fromNBase+dir[j]+nf.format(i));
      ss.setTo(toNBase+dir[j]+nf.format(i));
      s+= ss.toString();
      /*
      s = s+sbase+dir[j]+nf.format(i)+" \t s, "
          +fromNBase+dir[j]+nf.format(i)+", "
          +toNBase+dir[j]+nf.format(i)+", "
          +w+", "
          +deltaT+"\r\n"; */
    }
  }
// DN to MN
  sbase = "sm";
  fromNBase = "DN";
  toNBase = "MN";
  w = "0.12";
  deltaT = "4";
  String deltaT2 = "1";
  ss.reset();
  ss.setWeight(0.12);
  //ss.setDetlaT(1);
  int count = 1;
  s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
  for(int k = 0; k < pDir.length ; k++)
  {
    s = s+ "#  direction "+pDir[k]+"\r\n";
    for(int j =0; j< dir.length; j++)

    {
      s = s+ "#  "+dirLong[j]+"\r\n";
      count = 1;
      for (int i = 1; i <= numSNperDir-1; i++)
      {
        ss.setName(sbase+pDir[k]+dir[j]+nf.format(count));
        ss.setDetlaT(Integer.parseInt(deltaT.trim()));
        ss.setFrom(fromNBase+dir[j]+nf.format(i));
        ss.setTo(toNBase+pDir[k]+dir[j]+nf.format(i));
        s+=ss.toString();

        /*
        s = s+sbase+pDir[k]+dir[j]+nf.format(count)+" \t s, "
            +fromNBase+dir[j]+nf.format(i)+", "
            +toNBase+pDir[k]+dir[j]+nf.format(i)+", "
            +w+", "
            +deltaT+"\r\n";*/
        count++;
        ss.setName(sbase+pDir[k]+dir[j]+nf.format(count));
        ss.setFrom(fromNBase+dir[j]+nf.format((i+1)));
        ss.setTo(toNBase+pDir[k]+dir[j]+nf.format(i));
        ss.setDetlaT(Integer.parseInt(deltaT2.trim()));
        s+=ss.toString();

        /*
        s = s+sbase+pDir[k]+dir[j]+nf.format(count)+"\t s, "
            +fromNBase+dir[j]+nf.format((i+1))+", "
```

```
            +toNBase+pDir[k]+dir[j]+nf.format(i)+", "
            +w+", "
            +deltaT2+"\r\n";*/
        count++;
      }
    }
    //swap(deltaT, deltaT2);
    String temp = new String(deltaT);
    deltaT = new String(deltaT2);
    deltaT2 = temp;

 }

 //out.print(s);
 //System.out.print(s);
 //s = "";
// MN to ON
sbase = "so";
fromNBase = "MN";
toNBase = "ON";
w = "0.2";
deltaT = "1";
ss.reset();
ss.setWeight(0.2);
ss.setDetlaT(1);

s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    count = 1;
    for (int i = 1; i <= numSNperDir-1; i++)
    {
      ss.setName(sbase+pDir[k]+dir[j]+nf.format(count));
      ss.setFrom(fromNBase+pDir[k]+dir[j]+nf.format(i));
      ss.setTo(toNBase+pDir[k]+dir[j]);
      s+= ss.toString();
      /*
      s = s+sbase+pDir[k]+dir[j]+nf.format(count)+" \t s, "
          +fromNBase+pDir[k]+dir[j]+nf.format(i)+", "
          +toNBase+pDir[k]+dir[j]+", "
          +w+", "
          +deltaT+"\r\n"; */
       count++;
    }
  }
}
// -- added for motion det problem
// ON to DN
sbase = "sd";
fromNBase = "ON";
toNBase = "DN";

ss.reset();
ss.setWeight(-0.005);
ss.setDetlaT(4);

s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
```

```
      {
        s = s+ "#  "+dirLong[j]+"\r\n";
        count = 1;
        for (int i = 1; i <= numSNperDir-1; i++)
        {
          ss.setName(sbase+pDir[k]+dir[j]+nf.format(count));
          //ss.setTo(toNBase+pDir[k]+dir[j]+nf.format(i));
          ss.setTo(toNBase+dir[j]+nf.format(i));
          ss.setFrom(fromNBase+pDir[k]+dir[j]);
          s+= ss.toString();
          count++;
        }
      }
    }


    System.out.print(s);
     //out.print(s);
     out.println(s);
     out.flush();
     out.close();

  }     .


//--------------------  Motion Det 3 ---------------------------
  /**
   * Uses ON to MN feedback inhibition to fight overshoot problem.
   * @param numSNperDir int
   * @param path String
   * @param fileName String
   */
  public void createMotionDet3(int numSNperDir, String path, String fileName)
  {
    NumberFormat nf = NumberFormat.getInstance();
    nf.setMinimumIntegerDigits(2);
    File f = new File(path, fileName);
    try{
      out = new PrintWriter(new BufferedWriter(new FileWriter(f)));
    }catch(IOException ioe){ioe.printStackTrace();}
    finally
    {
        out.flush();
        //out.close();
        //System.exit(-1);
    }

    this.setLeakRate(0.84);
    this.setLTPRate(1.005);
    this.setLTDRate(0.994);

    // neuron name base
    String nbase;
    // direction
    String[] dir = {"L","R","U","D"};
    String[] dirLong = {"Left","Right","Up","Down"};
    String[] pDir = {"1","2"};
    // layer number
    int layer = 0;



    String s = "";
    s=s.concat("# ------------------------------------------------------\r\n");
    s=s.concat("#  "+ f.getName()+"\r\n");
```

```
 s=s.concat("# ----------------------------------------------------\r\n");
s=s.concat("leakRate = "+leakRate+"\r\n");
s=s.concat("LTPRate = "+LTPRate+"\r\n");
s=s.concat("LTDRate = "+LTDRate+"\r\n");
s = s+"#Input group \r\n"
    +"InputGroups = 1, 0, 1, 2, 1, 1\r\n"
    +"#--- Neurons -------\r\n"
    +"#sensory neurons - \r\n";
nbase = "SN";
layer = 0;
NeuronString ns = new NeuronString();
ns.setLayer(layer);
for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {

    ns.setName(nbase+dir[j]+nf.format(i));
    ns.setLayer(layer);
    s += ns.toString();
    //s = s+nbase+dir[j]+nf.format(i)+" \t n, "+layer+"\r\n";

  }
  if(j >=1) layer = 1;

}


s = s+ "# Derivative Neurons\r\n";
nbase = "DN";
layer = 2;
ns.reset();
ns.setLayer(layer);

for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {

    ns.setName(nbase+dir[j]+nf.format(i));
    ns.setLeakRate(0.45);
    s += ns.toString();

    //s = s+nbase+dir[j]+nf.format(i)+" \t n, "+layer;
    //s = s + ", f , 0.45"; // set leak rate for DN
    //s= s +"\r\n";
  }
}

s = s+ "# Motion Neurons\r\n";
nbase = "MN";
layer = 3;
ns.reset();
ns.setLayer(layer);
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    for (int i = 1; i <= numSNperDir-1; i++)
    {
      ns.setName(nbase+pDir[k]+dir[j]+nf.format(i));
```

```
      s += ns.toString();
      //s = s+nbase+pDir[k]+dir[j]+nf.format(i)+"\t  n, "+layer+"\r\n";
    }
  }
}
s = s+ "# Output to Muscle Neurons\r\n";
nbase = "ON";
layer = 4;
ns.reset();
ns.setLayer(layer);
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    //s = s+ "#  "+dirLong[j]+"\r\n";


    ns.setName(nbase+pDir[k]+dir[j]);
    s += ns.toString();
    //s = s+nbase+pDir[k]+dir[j]+"\t  n, "+layer+"\r\n";
  }
}

// Synapses ------------------------------------------------
String sbase;  //synapse base
String fromNBase; // from neuron base
String toNBase; // to Neuron base
String w;       // weight
String deltaT;  // time delay

s = s+ "# Synapses ----------------------\r\n";
sbase = "sa";
fromNBase = "SN";
toNBase = "DN";
w = "-0.16";
deltaT = "4";

SynapseString ss = new SynapseString();
ss.setWeight(-0.16);
ss.setDetlaT(4);

s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {
    ss.setName(sbase+dir[j]+nf.format(i));
    ss.setFrom(fromNBase+dir[j]+nf.format(i));
    ss.setTo(toNBase+dir[j]+nf.format(i));

    s += ss.toString();
    /*
    s = s+sbase+dir[j]+nf.format(i)+" \t s, "
        +fromNBase+dir[j]+nf.format(i)+", "
        +toNBase+dir[j]+nf.format(i)+", "
        +w+", "
        +deltaT+"\r\n";
     */
  }
}
// SN to DN

sbase = "sb";
```

```
  fromNBase = "SN";
  toNBase = "DN";
  w = "0.21";
  deltaT = "1";
  ss.reset();
  ss.setWeight(0.21);
  ss.setDetlaT(1);
  s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
  for(int j =0; j< dir.length; j++)

  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    for (int i = 1; i <= numSNperDir; i++)
    {
      ss.setName(sbase+dir[j]+nf.format(i));
      ss.setFrom(fromNBase+dir[j]+nf.format(i));
      ss.setTo(toNBase+dir[j]+nf.format(i));
      s+= ss.toString();
      /*
      s = s+sbase+dir[j]+nf.format(i)+" \t s, "
          +fromNBase+dir[j]+nf.format(i)+", "
          +toNBase+dir[j]+nf.format(i)+", "
          +w+", "
          +deltaT+"\r\n"; */
    }
  }
// DN to MN
  sbase = "sm";
  fromNBase = "DN";
  toNBase = "MN";
  w = "0.12";
  deltaT = "4";
  String deltaT2 = "1";
  ss.reset();
  ss.setWeight(0.12);
  //ss.setDetlaT(1);
  int count = 1;
  s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
  for(int k = 0; k < pDir.length ; k++)
  {
    s = s+ "#  direction "+pDir[k]+"\r\n";
    for(int j =0; j< dir.length; j++)
    {
      s = s+ "#  "+dirLong[j]+"\r\n";
      count = 1;
      for (int i = 1; i <= numSNperDir-1; i++)
      {
        ss.setName(sbase+pDir[k]+dir[j]+nf.format(count));
        ss.setDetlaT(Integer.parseInt(deltaT.trim()));
        ss.setFrom(fromNBase+dir[j]+nf.format(i));
        ss.setTo(toNBase+pDir[k]+dir[j]+nf.format(i));
        s+=ss.toString();

        /*
        s = s+sbase+pDir[k]+dir[j]+nf.format(count)+" \t s, "
            +fromNBase+dir[j]+nf.format(i)+", "
            +toNBase+pDir[k]+dir[j]+nf.format(i)+", "
            +w+", "
            +deltaT+"\r\n";*/
        count++;
        ss.setName(sbase+pDir[k]+dir[j]+nf.format(count));
        ss.setFrom(fromNBase+dir[j]+nf.format((i+1)));
        ss.setTo(toNBase+pDir[k]+dir[j]+nf.format(i));
        ss.setDetlaT(Integer.parseInt(deltaT2.trim()));
        s+=ss.toString();
```

```
        /*
        s = s+sbase+pDir[k]+dir[j]+nf.format(count)+"\t s, "
            +fromNBase+dir[j]+nf.format((i+1))+", "
            +toNBase+pDir[k]+dir[j]+nf.format(i)+", "
            +w+", "
            +deltaT2+"\r\n";*/
        count++;
      }
    }
    //swap(deltaT, deltaT2);
    String temp = new String(deltaT);
    deltaT = new String(deltaT2);
    deltaT2 = temp;

 }

 //out.print(s);
 //System.out.print(s);
 //s = "";
// MN to ON
sbase = "so";
fromNBase = "MN";
toNBase = "ON";
w = "0.2";
deltaT = "1";
ss.reset();
ss.setWeight(0.2);
ss.setDetlaT(1);

s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    count = 1;
    for (int i = 1; i <= numSNperDir-1; i++)

    {
      ss.setName(sbase+pDir[k]+dir[j]+nf.format(count));
      ss.setFrom(fromNBase+pDir[k]+dir[j]+nf.format(i));
      ss.setTo(toNBase+pDir[k]+dir[j]);
      s+= ss.toString();
      /*
      s = s+sbase+pDir[k]+dir[j]+nf.format(count)+" \t s, "
          +fromNBase+pDir[k]+dir[j]+nf.format(i)+", "
          +toNBase+pDir[k]+dir[j]+", "
          +w+", "
          +deltaT+"\r\n"; */
       count++;
    }
  }
}
// -- added for motion det problem after motion det2 failed
// ON to MN
sbase = "sd";
fromNBase = "ON";
toNBase = "MN";

ss.reset();
ss.setWeight(-0.005); //-0.035 //-0.15
ss.setDetlaT(8);
```

```
    s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
    for(int k = 0; k < pDir.length ; k++)
    {
      s = s+ "#  direction "+pDir[k]+"\r\n";
      for(int j =0; j< dir.length; j++)
      {
        s = s+ "#  "+dirLong[j]+"\r\n";
        count = 1;
        for (int i = 1; i <= numSNperDir-1; i++)
        {
          ss.setName(sbase+pDir[k]+dir[j]+nf.format(count));
          //ss.setTo(toNBase+pDir[k]+dir[j]+nf.format(i));
          if(k>0)
            ss.setTo(toNBase+pDir[k-1]+dir[j]+nf.format(i));
          else
            ss.setTo(toNBase+pDir[k+1]+dir[j]+nf.format(i));
          ss.setFrom(fromNBase+pDir[k]+dir[j]);
          s+= ss.toString();
          count++;
        }
      }
    }


    System.out.print(s);
     //out.print(s);
     out.println(s);
     out.flush();
     out.close();

  }

// --------------------------------Zones-----------------------------------------

  public void createMotionDetZones(int numSNperDir,int numZones, String path, String
fileName)
    {


      NumberFormat nf = NumberFormat.getInstance();
      nf.setMinimumIntegerDigits(2);
      File f = new File(path, fileName);
      try{
        out = new PrintWriter(new BufferedWriter(new FileWriter(f)));
      }catch(IOException ioe){ioe.printStackTrace();}
      finally
      {
          out.flush();
          //out.close();
          //System.exit(-1);
      }

      this.setLeakRate(0.84);
      this.setLTPRate(1.005);
      this.setLTDRate(0.994);

      // neuron name base
      String nbase;
      // direction
      String[] dir = {"L","R","U","D"};
      String[] dirLong = {"Left","Right","Up","Down"};
      String[] pDir = {"1","2"};
      // layer number
      int layer = 0;
```

```
String s = "";
s=s.concat("# ------------------------------------------------\r\n");
s=s.concat("#  "+ f.getName()+"\r\n");
 s=s.concat("# ------------------------------------------------\r\n");
s=s.concat("leakRate = "+leakRate+"\r\n");
s=s.concat("LTPRate = "+LTPRate+"\r\n");
s=s.concat("LTDRate = "+LTDRate+"\r\n");
s = s+"#Input group \r\n"
    +"InputGroups = 1, 0, 1, 2, 1, 1\r\n"
    +"#--- Neurons -------\r\n"
    +"#sensory neurons - \r\n";
nbase = "SN";
layer = 0;
for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {
    s = s+nbase+dir[j]+nf.format(i)+" \t n, "+layer+"\r\n";
  }
  if(j >=1) layer = 1;

}

s = s+ "# Derivative Neurons\r\n";
nbase = "DN";
layer = 2;
for(int j =0; j< dir.length; j++)
{
  s = s+ "#  "+dirLong[j]+"\r\n";
  for (int i = 1; i <= numSNperDir; i++)
  {
    s = s+nbase+dir[j]+nf.format(i)+" \t n, "+layer;
    s = s + ", f , 0.45"; // set leak rate for DN
    s= s +"\r\n";
  }
}

s = s+ "# Motion Neurons\r\n";
nbase = "MN";
layer = 3;
for(int k = 0; k < pDir.length ; k++)
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    for (int i = 1; i <= numSNperDir-1; i++)
    {
      s = s+nbase+pDir[k]+dir[j]+nf.format(i)+"\t  n, "+layer+"\r\n";
    }
  }
}
s = s+ "# Output to Muscle Neurons\r\n";
nbase = "ON";
layer = 4;
for(int zone =1; zone <= numZones; zone++)
{
  s = s + "#  Zone " + zone +"\r\n";
  for (int k = 0; k < pDir.length; k++)
  {
    s = s + "#  direction " + pDir[k] +"\r\n";
    for (int j = 0; j < dir.length; j++)
```

```
        {
          //s = s+ "#  "+dirLong[j]+"\r\n";
          s = s + nbase + pDir[k] + dir[j] + zone +"\t  n, " + layer
              + ", f, 0.65"+ "\r\n";
        }
      }
    }

    // Synapses ------------------------------------------
    String sbase;  //synapse base
    String fromNBase; // from neuron base
    String toNBase; // to Neuron base
    String w;       // weight
    String deltaT;  // time delay

    s = s+ "# Synapses ----------------------\r\n";
    sbase = "sa";
    fromNBase = "SN";
    toNBase = "DN";
    w = "-0.16";
    deltaT = "4";

    s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
    for(int j =0; j< dir.length; j++)
    {
      s = s+ "#  "+dirLong[j]+"\r\n";
      for (int i = 1; i <= numSNperDir; i++)
      {
        s = s+sbase+dir[j]+nf.format(i)+" \t s, "
            +fromNBase+dir[j]+nf.format(i)+", "
            +toNBase+dir[j]+nf.format(i)+", "
            +w+", "
            +deltaT+"\r\n";
      }
    }
    // SN to DN
    sbase = "sb";
    fromNBase = "SN";
    toNBase = "DN";
    w = "0.21";
    deltaT = "1";

    s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
    for(int j =0; j< dir.length; j++)
    {
      s = s+ "#  "+dirLong[j]+"\r\n";
      for (int i = 1; i <= numSNperDir; i++)
      {
        s = s+sbase+dir[j]+nf.format(i)+" \t s, "
            +fromNBase+dir[j]+nf.format(i)+", "
            +toNBase+dir[j]+nf.format(i)+", "
            +w+", "
            +deltaT+"\r\n";
      }
    }
  // DN to MN
    sbase = "sm";
    fromNBase = "DN";
    toNBase = "MN";
    w = "0.12";
    deltaT = "4";
    String deltaT2 = "1";
    int count = 1;
    s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
    for(int k = 0; k < pDir.length ; k++)
```

```
{
  s = s+ "#  direction "+pDir[k]+"\r\n";
  for(int j =0; j< dir.length; j++)
  {
    s = s+ "#  "+dirLong[j]+"\r\n";
    count = 1;
    for (int i = 1; i <= numSNperDir-1; i++)
    {
      s = s+sbase+pDir[k]+dir[j]+nf.format(count)+" \t s, "
          +fromNBase+dir[j]+nf.format(i)+", "
          +toNBase+pDir[k]+dir[j]+nf.format(i)+", "
          +w+", "
          +deltaT+"\r\n";
      count++;
      s = s+sbase+pDir[k]+dir[j]+nf.format(count)+"\t s, "
          +fromNBase+dir[j]+nf.format((i+1))+", "
          +toNBase+pDir[k]+dir[j]+nf.format(i)+", "
          +w+", "
          +deltaT2+"\r\n";
      count++;
    }
  }
  //swap(deltaT, deltaT2);
  String temp = new String(deltaT);
  deltaT = new String(deltaT2);
  deltaT2 = temp;

}

//out.print(s);
//System.out.print(s);
//s = "";
// MN to ON
int numSNperDirperZone = Math.round((numSNperDir-1)/numZones);
int rmd = 0;
int rmd2 = 0;
int rmd3 = 0;
sbase = "so";
fromNBase = "MN";
toNBase = "ON";
w = "0.13";
deltaT = "1";
s = s+"# from "+fromNBase+" to "+toNBase+"\r\n";
for (int zone = 1; zone <= numZones; zone++)
{
  if (zone == numZones) rmd =  (numSNperDir-1)%numZones;
  s = s + "#  zone " + zone + "\r\n";
  for (int k = 0; k < pDir.length; k++)
  {
    s = s + "#  direction " + pDir[k] + "\r\n";
    for (int j = 0; j < dir.length; j++)
    {
      s = s + "#  " + dirLong[j] + "\r\n";
      count = 1;
      if(j%2 != 0)
      {  //odd  R and D  start from 1 -> num MNs
        for (int i = (numSNperDirperZone * zone - numSNperDirperZone) + 1;
             i <= numSNperDirperZone * zone + rmd; i++)
        {
          s = s + sbase + pDir[k] + dir[j] + zone + "-" + nf.format(count) +
              " \t s, "
              + fromNBase + pDir[k] + dir[j] + nf.format(i) + ", "
              + toNBase + pDir[k] + dir[j] + zone + ", "
              + w + ", "
              + deltaT + "\r\n";
```

```
                count++;
              }
            }
            else
            {// even L and U   start from num MNs --> 1
                //for (int i = (numSNperDirperZone * (numZones -zone + 1) -
numSNperDirperZone) + 1;
                //   i <= numSNperDirperZone * (numZones - zone + 1)+rmd; i++)
                if (zone == 1) // first
                 {
                  rmd3 = rmd2 = (numSNperDir - 1) % numZones;
                 }
                  else if (zone == numZones) //last
                  {
                    rmd2 = 0;
                    rmd3 = (numSNperDir - 1) % numZones;
                  }
                   else
                     rmd2 = rmd3 = 0;  // otherwise
                for (int i = (numSNperDir-1) - numSNperDirperZone * zone  + rmd2; i <
(numSNperDir-1) - numSNperDirperZone * (zone - 1)  +rmd3; i++)
                {
                  s = s + sbase + pDir[k] + dir[j] + zone + "-" + nf.format(count) +
                      " \t s, "
                      + fromNBase + pDir[k] + dir[j] + nf.format(i) + ", "
                      + toNBase + pDir[k] + dir[j] + zone + ", "
                      + w + ", "
                      + deltaT + "\r\n";
                  count++;
            }

            }
          }
        }
      }


     System.out.print(s);
      //out.print(s);
      out.println(s);
      out.flush();
      out.close();

    }

  /**
   *
   * @param numSNeurons int  must have an integer square root (16 = 4 X 4)
   * @param path String
   * @param fileName String
   */
  public void createPatternRec(int numSNeurons, String path, String fileName)
  {
    File f = new File(path, fileName);
      try{
        out = new PrintWriter(new BufferedWriter(new FileWriter(f)));
      }catch(IOException ioe){ioe.printStackTrace();}
      finally
      {
          out.flush();
          //out.close();
          //System.exit(-1);
      }
```

```
int rowLength = (int) Math.sqrt((double)numSNeurons);
String[] nNames= new String[numSNeurons];
// create sn neuron names
int r = 1;

int cnt =0;
for (int g = 0; g < nNames.length; g++)
{

  for (int i = 0; i < rowLength; i++)
  {
    nNames[cnt] =  new String("R"+r+"C"+(i+1));
    //System.out.println(nNames[cnt] + " cnt="+cnt);
    g++;
    cnt++;
  }
  r++;

}


this.setLeakRate(0.95);
this.setLTPRate(1.010);
this.setLTDRate(0.999);


// layer number
int layer = 0;

// setup and Header ------------------------------------------

String s = "";

s=s.concat("# ------------------------------------------------\r\n");
s=s.concat("#  "+ f.getName()+"\r\n");
 s=s.concat("# ------------------------------------------------\r\n");
s=s.concat("leakRate = "+leakRate+"\r\n");
s=s.concat("LTPRate = "+LTPRate+"\r\n");
s=s.concat("LTDRate = "+LTDRate+"\r\n");
s = s+"#Input group \r\n"
    +"InputGroups 1, 0, 0.1\r\n"
    +"#--- Neurons -------\r\n"
    +"#sensory neurons - \r\n";


// sensory Neurons ------------------------------------------
layer = 0;
for (int i = 0; i < numSNeurons; i++)
{
  s = s+nNames[i]+" n, "+layer+"\r\n";
}
// patter recog neurons ---------------------------------------
layer = 1;
s = s+"# Pattern Recognition Neurons\r\n";
s = s+"PR11   n, "+layer+", t, 0.99, 16\r\n";

// synapses---------------------------------------------------
s = s+"#---- Synapses -------\r\n";

String sbase = "sa";
int synNum = 10;
String w = "0.02";
for (int i = 0; i < numSNeurons; i++)
{
  for(int j = 0 ; j < synNum; j++)
```

```
        {
          s = s + sbase + nNames[i] +"-"
              + (j+1) +"  s, "
              + nNames[i] +", PR11, "
              +w+", "+j+",t \r\n";
        }
        System.out.print(nNames[i]+" ");
        s=s+"\r\n";
        if(i%100 == 0)
        {
          out.print(s);
          s ="";
        }
      }
   out.print(s);
   out.flush();
   out.close();

   //System.out.println(s);
  }
  /**
   *
   * @param numSNeurons int -- must have an integer square root
   * @param SNperPRN int    -- must have an integer square root and evenly
   * divide into numSNeurons
   * @param path String
   * @param fileName String
   */
  public void createPatternRec(int numSNeurons,int SNperPRN, String path, String
fileName)
  {
    File f = new File(path, fileName);
      try{
        out = new PrintWriter(new BufferedWriter(new FileWriter(f)));
      }catch(IOException ioe){ioe.printStackTrace();}
      finally
      {
          out.flush();
          //out.close();
          //System.exit(-1);
      }

      int rowLength = (int) Math.sqrt((double)numSNeurons);
      String[] nNames= new String[numSNeurons];
      // create sn SN neuron names
      char r = 'A';
      int cnt =0;
      for (int g = 0; g < nNames.length; g++)
      {

        for (int i = 0; i < rowLength; i++)
        {
          nNames[cnt] =  new String(Character.toString(r)+(i+1));
          //System.out.println(nNames[cnt] + " cnt="+cnt);
          g++;
          cnt++;
        }
        r++;
      }
      //create PR neuron names
      int aLength= numSNeurons/SNperPRN;
      String[] prnNames= new String[aLength];
      for(int i = 0; i < prnNames.length; i++)
      {
        prnNames[i] = "PR"+i;
```

```
    }



    this.setLeakRate(0.95);
    this.setLTPRate(1.010);
    this.setLTDRate(0.999);


    // layer number
    int layer = 0;

    // setup and Header ------------------------------------------

    String s = "";

    s=s.concat("# ----------------------------------------------------\r\n");
    s=s.concat("#  "+ f.getName()+"\r\n");
     s=s.concat("# ----------------------------------------------------\r\n");
    s=s.concat("leakRate = "+leakRate+"\r\n");
    s=s.concat("LTPRate = "+LTPRate+"\r\n");
    s=s.concat("LTDRate = "+LTDRate+"\r\n");
    s = s+"#Input group \r\n"
        +"InputGroups 1, 0, 0.1\r\n"
        +"#--- Neurons -------\r\n"
        +"#sensory neurons - \r\n";


    // sensory Neurons ------------------------------------------
    layer = 0;
    for (int i = 0; i < numSNeurons; i++)
    {
      s = s+nNames[i]+" n, "+layer+"\r\n";
    }
    // patter recog neurons ---------------------------------------
    layer = 1;
    s = s+"# Pattern Recognition Neurons\r\n";
    for (int i = 0; i<prnNames.length; i++)
    {
      s = s + prnNames[i]+"  n, " + layer + ", t, 0.99, 16\r\n";
    }

    // synapses---------------------------------------------------
    s = s+"#---- Synapses -------\r\n";

    String sbase = "sa";
    int synNum = 10;
    String w = "0.02";
    int side = (int) Math.sqrt((double) SNperPRN);
    int prCnt = 1;
    for (int i = 0; i < numSNeurons; i++)
    {
      for(int j = 0 ; j < synNum; j++)
      {
        //System.out.println("index ="+translateIndex(i,rowLength,side));
        s = s + sbase + nNames[i] +"-"
            + (j+1) +" s, "
            + nNames[i] +", "+prnNames[translateIndex(i,rowLength,side)]+", "
            +w+", "+j+",t \r\n";
      }
      s=s+"\r\n";
    }
out.print(s);
out.flush();
```

```
 out.close();

 //System.out.println(s);
}
protected int translateIndex(int i,int rowLength, int subRowLength )
{

  int numBlocks = rowLength/subRowLength;
  int row = (int) i/rowLength;  // row should be 0 - (rowLength -1)
  int col = (int) i%rowLength;  // col should be 0 - (rowLength -1)
  if((col > 2) ||(row > 2))
  {
    //System.out.println("col="+col );
    //System.out.println("row="+row );
  }

  int blockRow = (int) row/subRowLength;
  int blockCol = (int) col/subRowLength;
  if((blockCol > 2) ||(blockRow > 2))
  {
  //System.out.println("blockCol="+blockCol );
  //System.out.println("blockRow="+blockRow );
  }

  int blockNum = blockCol + blockRow*(numBlocks);

  return blockNum;
}


protected class NeuronString
{
  private int paramCnt = 0;
  private int optCnt = 0;
  String name = null;
  String layer ="-";
  String learn = "f";
  String leakRate = "-";
  String threshold = "-";
  String restPotential = "-";
  public NeuronString() {}
  public void setName(String name)
  {
    this.name = name;
    if (paramCnt < 2) paramCnt++;
  }
  public void setLayer(int l)
  {
    layer = ""+ l;
    if (paramCnt < 2) paramCnt++;
  }

  public void setLearn(boolean learn)
  {
    if (learn) this.learn = "t";
    if (optCnt <1) optCnt = 1;
  }
  public void setLeakRate(double rate)
  {
    leakRate = ""+rate;
    if (optCnt <2) optCnt = 2;
  }
  public void setThreshold(double t)
  {
    threshold = ""+t;
```

```
    if (optCnt <3) optCnt = 3;
  }
  public void setRestPotential(double r)
  {
    restPotential = ""+r;
    if (optCnt <4) optCnt = 4;
  }

  public void reset()
  {
   paramCnt = 0;
   optCnt = 0;
   leakRate = "-";
   threshold = "-";
   restPotential = "-";
  }
  public String toString()
  {
    String s = null;
    if(paramCnt ==2)
    {
      s = this.name+ " n, "+layer;
      if (optCnt >0)
      {
        switch(optCnt)
        {

        case 1:
          s += ", "+learn;
          break;
        case 2:
          s += ", "+learn+","+leakRate;
          break;
        case 3:
          s += ", "+learn+","+leakRate+", "+threshold;
          break;
        case 4:
          s += ", "+learn+","+leakRate+", "+threshold+", "+restPotential;
          break;
        }
      }
      s += "\r\n";
    }
    return s;
  }
}
public class SynapseString
{
  String name;
  String fromN;
  String toN;
  String w;
  String deltaT;
  String learn = "f";
  public void setName(String name){this.name = name;}
  public void setFrom(String neuronName){fromN = neuronName;}
  public void setTo(String neuronName){toN = neuronName;}
  public void setWeight(double w){this.w = Double.toString(w);}
  public void setDetlaT(int timeDelay) {this.deltaT = Integer.toString(timeDelay);}
  public void setLearn(boolean learn)
  {
    if (learn)this.learn = "t";
    else this.learn = "f";
  }
  public void reset()
```

```
    {
      name ="";
      fromN ="";
      toN="";
      w="";
      deltaT="";
      learn = "f";
    }
    public String toString()
    {
      return name + "\t s, "+fromN+", "+toN+", "+w+", "+deltaT+", "+ learn + "\r\n ";
    }
  }
}
```

REFERENCES

M.F. Bear, B.W. Connors, M.A. Paradise, *Neuroscience: Exploring the Brain*, Sec Ed, Lippincott Williams & Wilkins, Maryland 2001.

E. Domany, J. L. van Hemmen, K.Schulten, Models of Neural Networks II: Temporal Aspects of Coding and Information Processing in Biological Systems, Springer-Verlag, New York, 1994.

M. Eldefrawy and N. Farhat, The Bifurcating Neuron: Characterization and Dynamics In *Photonics for Computers, Neural Networks and Memories*, SPIE proceedings, San Diego, CA, (July 2, 1992), vol 1773 p. 23-34.

J.J. Hopfield, Pattern Recognition Computation Using Action Potential Timing for Stimulus Representation, *Nature* 376 (1995), p. 33-36.

J.J. Hopfield, C.D. Brody, S. Roweis Computing with action potentials, *Neural Information Processing Systems* 10 (MIT Press 1998), p. 166-172.

E.R. Kandel, J. H. Schwartz, T.M. Jessel, *Essentials of Neuroscience and Behavior*, Appleton & Lange, Connecticut 1995.

C. Koch, *Biophysics of Computation: Information processing in Single Neurons*, Oxford University Press, 1999.

G. Lee and N. Farhat, The Bifurcating Neuron Network 2: an analog associative memory. *Neural Networks* 15 (2002), p. 60-84.

K. Mehrotra, C. Mohan, S. Ranka, *Elements of Artificial Networks*, MIT Press, Cambridge, Massachusetts, 1997.

M. Olufsen, M. Whittington, et al, New Roles for Gamma Rhythm:Population Tunning and Preprocessing for the Beta Rhythm, *Journal of Computational Neuroscience* 14 (2003), p. 33-54.

R.P.N. Rao, Bayesian Computation in Recurrent Neural Circuits In *Neural Computation* 16(1) (Jan 2004) p. 1-38.

L. Risinger, K. Kaikhah, Modified Bifurcating Neuron with Leaky-Integrate-and-Fire Model, *Seventeenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, (IEA/AIE)*, (May 2004)

D. Rizzutto and M. Kahana, An Autoassociative Neural Network Model of Paired-Associative Learning, *Neural Computation* 13 (2001), p.2075-2092.

G. M. Shepard, *The Synaptic Organization of the Brain fifth ed*, Oxford University Press, Oxford, 2004.

L.S. Smith, D.S. Fraser, Robust Sound Onset Detection Using Leaky Integrate-and-Fire Neurons With Depressing Synapses *IEEE Transactions On Neural Networks* 15 (5) p. 1125-1134 (2004)

M. Sonka, V. Hlavac, R. Doyle, *Image Processing, Analysis and Machine Vision, sec ed*, PWS Publishing, New York, 1999.

VITA

Lon William Risinger was born March 24, 1971, in Houston, Texas, the son of William Voss Risinger and Ruby Faye Risinger. After completing his work at Sulphur Springs High School in May of 1989, he entered into military service as a member of the Texas State National Guard. During this time he briefly attended East Texas State University but decided to pursue a full time military career. In September of 1990 he entered into fulltime active duty service with the U.S. Army as an Infantryman. During his four year military career he was stationed in Germany, Colorado, Korea, and Georgia and fought in the first war with Iraq, Operation Desert Storm, as a member of the 1st Armored Division, 7th Core based in Germany. At the end of his military enlistment, he left the army with an honorable discharge. Later that year, he began attending the University of Houston in Houston, Texas, with funding provided by the military educational benefits. He then transferred to the University of Texas at Austin in the spring of 1996, where he remained until his graduation with a Bachelor of Science in Physics in May of 1999. Shortly after, he took a position in the software industry in Austin, Texas. At the close of the 'Dot Com' era he decided it was time to pursue further academic interests. In the fall of 2002 he began the pursuit of a Master of Science in Computer Science at Texas State University in San Marcos.

Permanent Address: 5641 Valerie Street
Houston, TX 77081

This thesis was typed by Lon Risinger.